Universitat Autònoma de Barcelona

**ADVERTIMENT**. L'accés als continguts d'aquesta tesi doctoral i la seva utilització ha de respectar els drets de la persona autora. Pot ser utilitzada per a consulta o estudi personal, així com en activitats o materials d'investigació i docència en els termes establerts a l'art. 32 del Text Refós de la Llei de Propietat Intel·lectual (RDL 1/1996). Per altres utilitzacions es requereix l'autorització prèvia i expressa de la persona autora. En qualsevol cas, en la utilització dels seus continguts caldrà indicar de forma clara el nom i cognoms de la persona autora i el títol de la tesi doctoral. No s'autoritza la seva reproducció o altres formes d'explotació efectuades amb finalitats de lucre ni la seva comunicació pública des d'un lloc aliè al servei TDX. Tampoc s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant als continguts de la tesi com als seus resums i índexs.

**ADVERTENCIA**. El acceso a los contenidos de esta tesis doctoral y su utilización debe respetar los derechos de la persona autora. Puede ser utilizada para consulta o estudio personal, así como en actividades o materiales de investigación y docencia en los términos establecidos en el art. 32 del Texto Refundido de la Ley de Propiedad Intelectual (RDL 1/1996). Para otros usos se requiere la autorización previa y expresa de la persona autora. En cualquier caso, en la utilización de sus contenidos se deberá indicar de forma clara el nombre y apellidos de la persona autora y el título de la tesis doctoral. No se autoriza su reproducción u otras formas de explotación efectuadas con fines lucrativos ni su comunicación pública desde un sitio ajeno al servicio TDR. Tampoco se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al contenido de la tesis como a sus resúmenes e índices.

**WARNING**. The access to the contents of this doctoral thesis and its use must respect the rights of the author. It can be used for reference or private study, as well as research and learning activities or materials in the terms established by the 32nd article of the Spanish Consolidated Copyright Act (RDL 1/1996). Express and previous authorization of the author is required for any other uses. In any case, when using its content, full name of the author and title of the thesis must be clearly indicated. Reproduction or other forms of for profit use or public communication from outside TDX service is not allowed. Presentation of its content in a window or frame external to TDX (framing) is not authorized either. These rights affect both the content of the thesis and its abstracts and indexes.

Universitat Autònoma de Barcelona

**Escola d'Enginyeria**

**Departament d'Arquitectura de Computadors i Sistemes Operatius**


# Read mapping on heterogeneous systems: scalability strategies for bioinformatic primitives.


Thesis submitted by **Alejandro Chacon** for the degree of philosophae Doctor by the Universitat Autonoma de Barcelona, under the supervision of Dr. Juan Carlos Moure and Dr. Antonio Espinosa, developed at the Computer Architectures and Operating Systems department, PhD in High Performance Computing.



Barcelona, January 2021

# Read mapping on heterogeneous systems: scalability strategies for bioinformatic primitives.

Thesis submitted by **Alejandro Chacón** for the degree of Philosophiae Doctor by the Universitat Autonoma de Barcelona, under the supervision of Dr. Juan Carlos Moure and Dr. Antonio Espinosa, at the Computer Architecture and Operating Systems Department, Ph.D in High performance Computing.

Supervisors

Dr. Juan Carlos Moure    Dr. Antonio Espinosa

Barcelona, January 2021

# Dedications

To my family
To my friends
To the ones who joined this *unexpected journey*

Words are pale shadows of forgotten names. As names have power, words have power. Words can light fires in the minds of men. Words can wring tears from the hardest hearts. There are seven words that will make a person love you. There are ten words that will break a strong man's will. But a word is nothing but a painting of a fire. A name is the fire itself.

*The Name of the Wind* – **Patrick Rothfuss**

# Abstract

Genomic sequencing is the key component of new advances in medicine, and its democratization is an important step in improving accessibility for the patient. The benefits involved in discovering new genomic variations are vast and include everything from early cancer detection to personalized medicine, drug design and genome editing. All of these potential uses have greatly increased the interest of the scientific community in the field of bioinformatics in recent years. Moreover, the emergence of next-generation sequencing methods has contributed to the rapid reduction of sequencing costs, enabling new applications of genomics in precision medicine.

The main goal of this thesis is to improve the state of the art in performance and accuracy for genome sequencing through the use of heterogeneous computing platforms and hybrid hardware systems. More specifically, the work is focused on accelerating the problem of short-read mapping, as it is described as one of the most computationally expensive parts of the pipeline process. Overall, we aim to reduce the processing time and cost of genome sequencing, and then increasing the availability of this type analysis.

The main contribution of this thesis is the full GPU integration of the GEM3 mapper (GEM3-GPU), reporting significant improvements in performance and competitive accuracy results. The mapper reports the same output files for CPU and GPU and is one of the first GPU mappers to allow very long and variable read alignment. The proposals have been validated using real data, since the mapper has been running in production at a genomic sequencing center (Centro Nacional de Análisis Genómico (CNAG)).

Together with the GEM3-GPU mapper, a complete bioinformatics CUDA library (GEM-cutter) has been created. The library provides the basic building blocks for genomic applications, which are highly optimised to run on GPUs. Gem-cutter offers an API based on send and receive primitives (message passing) and incorporates a scheduler to balance the work. Furthermore, the library supports all GPU architectures and Multi-GPU execution.



**Keywords:** Heterogeneous systems, GPU, DNA sequencing, Short read mapping, indexing, string matching

# Resumen

La secuenciación genómica es un componente clave en nuevos avances en medicina, y su democratización es un paso importante hacia la accesibilidad para el paciente. Los beneficios implícitos en el descubrimiento de nuevas variantes genéticas son muy amplios, incluyendo desde la detección precoz de cáncer como la medicina personalizada, pasando por el diseño de fármaco y la edición genómica. Estos usos potenciales han incrementado exponencialmente el interés de la comunidad científica en el campo de la bioinformática durante los últimos años. Además, el surgimiento de los métodos de Secuenciación de Nueva Generación ha contribuido a la reducción rápida de los costes de secuenciación, permitiendo el desarrollo de nuevas aplicaciones genómicas. El principal objetivo de esta tesis es el de mejorar el rendimiento y precisión del estado del arte de la secuenciación genética a través del uso de plataformas de computo heterogéneo y sistemas de hardware híbridos. Más específicamente, el trabajo se ha centrado en la aceleración del problema del short-read mapping, dado que se describe como uno de los estadíos del pipeline con un mayor coste computacional. De forma global, se aspiraba a reducir el tiempo de procesado y el coste de la secuenciación genética, incrementando su disponibilidad.

La principal contribución de esta tesis es la integración GPU del mapper GEM3 (GEM3-GPU). Este mapper reporta los mismos datos de salida para CPU y GPU, y es uno de los primeros mappers GPU que permite el alineamiento de reads largos y variables. Las propuestas han sido validadas utilizando datos reales, dado que el mapper ha estado corriendo en producción en un centro de secuenciación (Centro Nacional de Análisis Genómico (CNAG)).

En conjunción con el mapper GEM3-GPU, durante esta tesis se ha creado una librería bioinformática en CUDA (GEM-cutter). La librería provee bloques de primitivas GPU básicas que han sido altamente optimizadas. Gem-cutter ofrece una API basada en primitivas de send and receive (message passing), e incorpora un scheduler para balancear el trabajo. Además, la librería soporta todas las arquitecturas GPU y Multi-GPU.

**Palabras clave:** Sistemas heterogéneos, GPU, secuenciación ADN, Short read mapping, indexación, string matching

# Resum

La seqüenciació genòmica és un component clau en nous avenços en medicina, i la seva democratització és un pas important per millorar l'accessibilitat per al pacient. Els beneficis implícits en el descobriment de noves variants genètiques són molt amplis, incloent des de la detecció precoç de càncer com la medicina personalitzada, passant pel disseny de fàrmacs i l'edició genòmica. Tots aquests usos potencials han incrementat exponencialment l'interès de la comunitat científica en el camp de la bioinformàtica durant els últims anys. A més, el sorgiment dels mètodes de Seqüenciació de Nova Generació ha contribuït a la reducció ràpida dels costos de seqüenciació, permetent el desenvolupament de noves aplicacions genòmiques.

El principal objectiu d'aquesta tesi és el de millorar el rendiment i precisió de l'estat de l'art de la seqüenciació genètica a través de l'ús de plataformes de còmput heterogeni i sistemes de computació híbrida. Més específicament, el treball s'ha centrat en l'acceleració de el problema de mapeig de reads curts, ja que es descriu com un dels estadis del pipeline amb un major cost computacional. De forma global, s'aspirava a reduir el temps de processament i el cost de la seqüenciació genètica, incrementant la disponibilitat d'aquest tipus d'anàlisi.

La principal contribució d'aquesta tesi és la integració GPU del mapper GEM3 (GEM3-GPU). Aquest mapper reporta les mateixes dades de sortida per CPU i GPU, i és un dels primers mappers GPU que permet l'alineament de reads llargs i variables. Les propostes han estat validades utilitzant dades reals, ja que el mapper ha estat corrent en producció en un centre de seqüenciació genòmica (Centre Nacional d'Anàlisi Genòmica (CNAG)).

En conjunció amb el mapper GEM3-GPU, durant aquesta tesi s'ha creat una llibreria bioinformàtica en CUDA (GEM-cutter). La llibreria aporta blocs de primitives GPU bàsiques que han estat altament optimitzades. Gem-cutter ofereix una API basada en primitives send and receive (message passing), i incorpora un scheduler per balancejar el treball. A més, la llibreria suporta totes les arquitectures GPU i Multi-GPU.

**Paraules clau:** Sistemes heterogenis, GPU, sequenciació ADN, Short read mapping, indexació, string matching

# Contents


**1 Introduction** **1**

1.1 Context . . . 2

1.1.1 Sequencing and its applications . . . 2

1.1.2 Downstream applications: Genomic sequencing pipelines . . . 4

1.1.3 Heterogeneous computing systems . . . 6

1.2 Motivation . . . 8

1.3 Objectives . . . 10

1.4 Methodology . . . 11

1.5 Contributions . . . 17

1.6 Collaborations . . . 19

1.7 Thesis outline . . . 20

**2 Bioinformatics Analysis** **25**

2.1 An introduction to DNA and genomics . . . 26

2.1.1 DNA structure . . . 26

2.1.2 Evolution of DNA sequencing . . . 27

2.1.3 Costs of sequencing . . . 29

2.2 Sequence alignment . . . 30

2.2.1 Sequencing pipelines (WGS, WES and WBS) . . . 30

2.2.2 Sequence pipeline: primary, secondary and tertiary analysis . . . 31

2.2.3 Seed and extend . . . 33

2.2.4 Short Read Mappers: definitions and characteristics . . . 36

2.3 GEM3 Mapper . . . 36

2.3.1 GEM3 general features . . . 36

2.3.2 GEM3 stages . . . 36

2.4 GEM3 contributions to GPU . . . 38

2.5 Conclusions . . . 39

**3 Heterogeneous Computing** **41**
3.1 Introduction . . . 42
3.2 Latency (CPU) and Throughput (GPU) processors . . . 43
3.3 General overview to GPU architecture and its CUDA programming model . 44
3.3.1 Nvidia GPU architectures and its hierarchies . . . 46
3.3.2 CUDA GPU execution model . . . 47
3.3.3 CUDA Parallelism model at thread level . . . 49
3.3.4 GPU Memory hierarchy . . . 50
3.3.5 General best practices for performance . . . 52
3.3.6 Different CUDA platforms as HPC and Embedded . . . 55
3.4 GPU Challenges on genomic algorithms . . . 55
3.4.1 Why the task-parallel approach fails on genomic applications . . . 56
3.4.2 Random memory accesses on genomic applications . . . 57
3.4.3 Irregular work on bioinformatic applications . . . 58
3.4.4 Host to device transferences . . . 59
3.5 Optimising Genomic algorithms on GPU . . . 60
3.5.1 Exploiting inter- and intra-task parallelism . . . 60
3.5.2 Rethinking bioinformatic algorithms: intra-task parallelism scheme . 61
3.5.3 Advanced parallel schemes: Combining the inter- and intra-task . . 62
3.5.4 Bypassing intra-task limitations . . . 63
3.6 Conclusions . . . 64

**4 FM-index: text indexing building blocks** **67**
4.1 Text indexing and exact string matching . . . 68
4.2 Suffix-trie: Forward- and Backward-Search . . . 68
4.3 Suffix-Array and Suffix-Array Intervals . . . 70
4.4 Exact string matching powered by FM-index . . . 71
4.4.1 Burrows-Wheeler Transform . . . 71
4.4.2 LF-Mapping . . . 71
4.4.3 FM-index: the backward search . . . 73
4.5 Sampled indexes: reduce space and search complexity . . . 75
4.5.1 Sampled FM-index design . . . 76
4.5.2 Advanced LF-mapping designs . . . 77
4.6 Performance analysis of LF-mapping . . . 78
4.7 Conclusions . . . 80

**5 FM-index: algorithmic and design proposals** **81**
5.1 Motivations and performance factors . . . . . . . . . . . . . . . . . . . . . . 82
5.2 *k*-step FM-index: a faster bloated index . . . . . . . . . . . . . . . . . . . . . 83
5.2.1 *k*-step BWT: a two-dimensional BWT . . . . . . . . . . . . . . . . . . 84
5.2.2 Sampled *k*-step FM-Index design . . . . . . . . . . . . . . . . . . 85
5.2.3 *k*-step FM-Index backward search . . . . . . . . . . . . . . . . . . 85
5.3 Alternate Counters: reducing memory requirements . . . . . . . . . . . . . . . 87
5.4 Performance analysis on CPU . . . . . . . . . . . . . . . . . . . . . . . . . 88
5.4.1 Experimental Setup and Methodology . . . . . . . . . . . . . . . . 88
5.4.2 Performance of the *k*-step FM-Index search . . . . . . . . . . . . . 90
5.4.3 Efficiency of sequential and parallel execution . . . . . . . . . . . . . 91
5.4.4 Efficiency of Memory operations . . . . . . . . . . . . . . . . . . . 92
5.4.5 Trading Memory requirements for Performance . . . . . . . . . . . 93
5.5 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

**6 FM-index: GPU Parallel designs for LF-mapping primitive** **95**
6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 96
6.2 Exploiting inter- and intra-task parallelism on LF-mapping . . . . . . . . . 97
6.3 LF-Mapping: Task-parallel designs . . . . . . . . . . . . . . . . . . . . . . 98
6.4 LF-Mapping: Thread-cooperative designs . . . . . . . . . . . . . . . . . . . 100
6.4.1 Memory oriented cooperative design . . . . . . . . . . . . . . . . . 100
6.4.2 Memory and compute oriented cooperative design . . . . . . . . . . 100
6.5 *k*-step FM-index and Alternate Counters on GPUs . . . . . . . . . . . . . . . 101
6.6 Experimentation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
6.6.1 Experimental Setup and Methodology . . . . . . . . . . . . . . . . 103
6.6.2 Overall performance results . . . . . . . . . . . . . . . . . . . . . 104
6.6.3 FM-index compression features on GPUs . . . . . . . . . . . . . . 105
6.6.4 Detailed performance analysis . . . . . . . . . . . . . . . . . . . . 106
6.6.5 Classical *1*-step sampled FM-index . . . . . . . . . . . . . . . . . . 108
6.6.6 *2*-step FM-index and alternate counters . . . . . . . . . . . . . . . 110
6.6.7 Comparison of GPU architectures . . . . . . . . . . . . . . . . . . . 111
6.7 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

**7 GEM3: approximate pattern search in a mapper GPU** **115**
7.1 Search by filtering and seed selection algorithms . . . . . . . . . . . . . . 116

7.2 Search by Filtration . . . 117
7.2.1 Seed generation . . . 117
7.2.2 Complete searches and pidgeonhole principle . . . 117
7.2.3 Static vs Adaptative seed selection algorithms . . . 118
7.2.4 Accelerating seed selection with multi-level LUT . . . 120
7.3 Experimentation . . . 120
7.3.1 Experimental Setup and Methodology . . . 120
7.3.2 Overall performance results . . . 121
7.4 Conclusions . . . 122

**8 Text filtering building blocks 127**
8.1 Introduction . . . 128
8.2 Computing Levenshtein distance . . . 130
8.2.1 Myers' bit-parallel algorithm . . . 130
8.3 Task-parallel designs . . . 132
8.4 Thread Cooperative Approach . . . 133
8.4.1 Intra-task SIMD vectorisation: 1 warp per task . . . 133
8.4.2 Intra- and Inter-task SIMD: 1 warp per r tasks . . . 135
8.5 Optimisation details . . . 136
8.6 Experimentation . . . 137
8.6.1 Experimental setup and methodology . . . 137
8.6.2 Overall Performance Results . . . 138
8.6.3 Task Parallel: Performance limiters . . . 139
8.6.4 Thread Cooperative: Performance limiters . . . 141
8.7 Conclusions . . . 144

**9 GEM-Cutter: high-performance bioinformatic library 147**
9.1 Introduction . . . 148
9.2 GEM3-GPU internal workflow . . . 149
9.2.1 Parallelism at thread, pipeline and task levels . . . 149
9.2.2 GEM3-GPU high-level workflow . . . 151
9.3 GEM-Cutter library . . . 152
9.3.1 GEM3-GPU mapper: specialisation techniques . . . 153
9.3.2 GPU kernel-level uschedulers . . . 153
9.3.3 CPU and GPU fine grain collaboration . . . 154

9.3.4 Transparent GPU transfers for the user . . . . . . . . . . . . . . . . . 154
9.3.5 Residency policies of the data structures . . . . . . . . . . . . . . . . 154
9.4 GEM3-GPU internal workflow in detail . . . . . . . . . . . . . . . . . . . 155
9.4.1 GEM3-GPU: Emulated mode . . . . . . . . . . . . . . . . . . . . . . . . 157
9.5 GEM3-GPU special features . . . . . . . . . . . . . . . . . . . . . . . . 158
9.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158

**10 GEM3-GPU Mapper benchmarking and experimentation 161**
10.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
10.2 Experimentation environment and methodology . . . . . . . . . . . . . . 162
10.2.1 Compute systems . . . . . . . . . . . . . . . . . . . . . . . . . . . 162
10.2.2 Short read mapping applications . . . . . . . . . . . . . . . . . . . 164
10.2.3 Description of the datasets . . . . . . . . . . . . . . . . . . . . . 166
10.2.4 Description of the sequencing technologies . . . . . . . . . . . . . . 167
10.2.5 Description of metrics . . . . . . . . . . . . . . . . . . . . . . . . 167
10.2.6 Experimentation reproducibility . . . . . . . . . . . . . . . . . . . 170
10.2.7 Final notes on used methodology . . . . . . . . . . . . . . . . . . . 171
10.3 General overview on performance results . . . . . . . . . . . . . . . . 172
10.3.1 Query size and thread scalability . . . . . . . . . . . . . . . . . . 173
10.4 Detailed comparison with the state-of-art . . . . . . . . . . . . . . . . 175
10.4.1 ROC curves analysis . . . . . . . . . . . . . . . . . . . . . . . . . 175
10.5 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 178

**11 Conclusions 183**
11.1 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 184
11.2 Future lines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 186
11.3 List of publications . . . . . . . . . . . . . . . . . . . . . . . . . . . 188
11.4 Acknowledgements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 190

**Bibliography 193**

# List of Figures


1.1 Progressive reduction in the costs of sequencing in relationship with the Moore's Law. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.2 Percentage of super-computers using accelerators in top500 . . . . . . . . . 6
1.3 Computing architecture eras by physical design limitations events . . . . . . 7
1.4 Stages of the research methodology . . . . . . . . . . . . . . . . . . . . . . 12

2.1 DNA structure showing the relationship in between nucleotides . . . . . . . . 27
2.2 Evolution of costs of sequencing per Human Genome over the last 20 years, in relationship with Moore's Law [1] . . . . . . . . . . . . . . . . . . . . . 29
2.3 GATK pipeline execution times by stage [2] . . . . . . . . . . . . . . . . . 32
2.4 Scheme of the main steps involved in the seed and extend strategy . . . . . 34
2.5 Internal workflow and algorithmic stages for GEM3 . . . . . . . . . . . . . . 37

3.1 Differences in resource dedication in the chip of CPU and GPU architectures. 44
3.2 General scheme of the elements of a typical GPU. Computation and memory elements are hierarchically organised. . . . . . . . . . . . . . . . . . . . . 46
3.3 Computation in a heterogeneous GPGPU system alternates the execution of serial an massively parallel phases. . . . . . . . . . . . . . . . . . . . . . . 48
3.4 Typical steps for invoking the execution of a computation kernel in a GPU. . 49
3.5 CUDA's thread hierarchy. All the threads that form a kernel are grouped in a set called grid. Threads within a grid are grouped in subsets called blocks. All blocks contain the same number of threads. . . . . . . . . . . . . . . . 50
3.6 Range, visibility, location and execution scope of each memory . . . . . . . . 51
3.7 Diagram relating the thread hierarchy with the data visibility in memory. A register can only be accessed by a thread; the shared memory can only be accessed by the threads of the same block; and the global, texture and constant memories can be accessed by any thread of the grid. . . . . . . . . . . . . . 52
3.8 Coalesced and non-coalesced accesses. . . . . . . . . . . . . . . . . . . . . 53

3.9 SM internal architecture showing the grouping of threads into warps. . . . . 54
3.10 SIMT architecture: threads in the same warp may divergence due to different conditions. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
3.11 Memory bandwidth for random accesses on the Titan GPU (6GB GDRAM) . 57
3.12 PCI-e interconnection bandwidth for different transfer sizes . . . . . . . . . 59

4.1 Forward and backward-search of $Q = acaa$ in $R = acaaacatat$ using the *suffix-trie* of $R$ and rev($R$) . . . . . . . . . . . . . . . . . . . . . . . . . 69
4.2 a) Illustration of the process of constructing the *SA* index for the input string $R = acaaacatat$; b) final *SA* index representation, where pointers/indexes to the original $R$ string represent the $R$ suffixes. . . . . . . . . . . . . . . . 70
4.3 Forward-search process for the query $Q = acaa$ in the reference $R = acaaacata$ using a Suffix-Array index. . . . . . . . . . . . . . . . . . . . . . . . . 72
4.4 Generating the BWT representation of string $R$, denoted $B$, using the input string and its Suffix Array, *SA*. . . . . . . . . . . . . . . . . . . . . . . 73
4.5 Backward-search process for finding all the occurrences of the query $Q = acaa$ in the reference $R = acaaacata\$$ using a FM-index and applying successive LF-mapping operations, decomposed as $LF(B,s,pos) := C(B,s) + Occ(B,s,pos)$. . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.6 Sampled FM-index $F$ with samples at distance $d$. Each entry in $F$ contains sampled counters, $rLF$, and a bitmap representation of the symbols in $B$ corresponding to the sampled interval, $BMP$. . . . . . . . . . . . . . . . . 76
4.7 Impact on performance –in Giga base query operations per second– of (a) varying reference size $n$, and (b) query size $m$; and (c) impact on index size of varying sampling distance $d$ . . . . . . . . . . . . . . . . . . . . . . . . 79

5.1 Backward-search of $Q = acaa$ in $R = acaaacatat$ using the *suffix-trie* of $R$ with a) single-step backward-search, and b) 2-step backward-search. . . . . 84
5.2 $k_B$ is the $k$-step BWT generated from $R$ and $SA$. . . . . . . . . . . . . . 85
5.3 $k_F$ is the $k$-step FM-index of $k_B$. . . . . . . . . . . . . . . . . . . . . 86
5.4 Backward-search process for finding all the occurrences of the query $Q = acaa$ in the reference $R = acaaacata\$$ using a k-step FM-index and applying a reduced number of LF-mapping operations, decomposed as $k_LF(B,s,pos) := C(B,s) + k_Occ(k_B,s,pos)$. . . . . . . . . . . . . . . . . . . . . . . . 87
5.5 Different layouts of the FM-index proposals . . . . . . . . . . . . . . . . . 88

5.6 Memory footprint of our FM-index implementations for the human genome using several sampling distances $d$= 32, 64, 128, 256; steps $k$= 1, 2, 3 and 4; and indexing structures: a) with a general k-step configuration, and b) including the technique of alternate counters. . . . . . . . . . . . . . . . . . 90
5.7 (a) Performance (ns/query) when varying $k$ and $|R|$; (b) Speedup when increasing $k$ for selected values of $|R|$. . . . . . . . . . . . . . . . . . . . . . . . 91
5.8 Efficiency of Execution: (a) Instructions per Cycle per Core; (b) Speedup due to multi-threading. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.9 Analysis of data read requirements: (a) Bytes requested per query; (b) DRAM bandwidth consumption. . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.10 Trading Performance versus Memory requirements by increasing the distance $d$ between counters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 94

6.1 GPU parallelization alternatives: a) **task-parallel**: each thread performs independent LF operations; b) **memory-cooperative**: threads cooperate on reading data from index; and c) **full-cooperative**: threads cooperate both on reading data and on counting symbol occurrences. Each search step comprises 16 queries, and in this example we consider the case $d$=448. We depict all the 32 threads in a warp participating in the execution of 32 LF operations. Memory read operations are shown in blue, and computation on the data (basically, counting symbols) in red. . . . . . . . . . . . . . . . . . . . . . . . . . . . 99
6.2 (a) Best FM-index search performance results compared to NVBIO library for CPU and GPU platforms; (b) CPU and GPU performance, measured in Giga bases processed per second. . . . . . . . . . . . . . . . . . . . . . . 105
6.3 (a) Thread scalability for our proposals (sampled FM-index with $d$=64) and the Nvidia NVBIO implementation; (b) Performance of task-parallel (both with maximal and optimal thread occupancy) and thread-cooperative designs for increasing sampling distance $d$. . . . . . . . . . . . . . . . . . . . . . . 107
6.4 Performance effect of varying reference size $n$ and sampling distance $d$ on different indexing schemes . . . . . . . . . . . . . . . . . . . . . . . . . . 108
6.5 (a) Memory Bandwidth evolution for different $d$ values; (b) Instructions per Cycle for full-cooperative scheme. . . . . . . . . . . . . . . . . . . . . . . 109
6.6 (a) Comparison among different GPUs (Full Cooperative); (b) Performance of random memory accesses for different GPUs (index entry size is 128 Bytes). 112

6.7 Performance comparison (a) and energetic efficiency (b) of our thread-cooperative strategy on different CPU/GPU architectures . . . . . . . . . . . . . . . . . 113

7.1 Example of query division in 3 seeds (factors), which shows that completeness is granted to 2 errors. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117
7.2 FM-index: Static seed (factor) selection . . . . . . . . . . . . . . . . . . . . . . . 118
7.3 GPU performance for Static and Adaptative seeding selection . . . . . . . . . 121

8.1 (a) Core operation of Myers' basic algorithm; (b) Myers' blocked-based algorithm. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 131
8.2 Thread Cooperation: r queries (*m*=400) and varying #words processed per thread . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
8.3 Performance overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137
8.4 GPU Task Parallel: local memory . . . . . . . . . . . . . . . . . . . . . . . . . 138
8.5 GPU Task Parallel: shared memory . . . . . . . . . . . . . . . . . . . . . . . . 139
8.6 GPU Thread Cooperative: 1 *word/thread* . . . . . . . . . . . . . . . . . . . . . 141
8.7 Performance for varying *words/thread* . . . . . . . . . . . . . . . . . . . . . . 142
8.8 Impact of varying *words/thread* on *instructions/cell* and GPU occupancy . . 143
8.9 Speedup of several GPUs vs CPU . . . . . . . . . . . . . . . . . . . . . . . . . 144

9.1 Short read mapping, interactions between primary analysis and secondary analysis. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149
9.2 GEM3-GPU: Overview at system-level of the internal workflow . . . . . . 150
9.3 GEM3-GPU: (a) exposing parallelism by pipelining between stages and (b) showing the batch dependences . . . . . . . . . . . . . . . . . . . . . . . . . 151
9.4 GEM3-GPU: Relationship between CPU threads, CPU/GPU internal buffers, data streams, simultaneous kernels and multiple devices . . . . . . . . . . . 152
9.5 GEM3: internal mapper workflow, stages and relationships . . . . . . . . . 153

10.1 Performance speedup of GEM3, using only CPU and using GPU-acceleration, compared to BWA-MEM (CPU-only) on a typical HPC system (P9+Volta) . 172
10.2 Performance speedup of GEM3, using only CPU and using GPU-acceleration, compared to BWA-MEM (CPU-only) on a typical embedded system (Tegra TX2) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 173
10.3 (a) Performance and (b) speedup comparison of GEM3-CPU and GEM3-GPU compared to BWA-MEM, on the P9+V100 CPU+GPU platform. . . . . . . 174

10.4 (c) Scalability and (d) efficiency of GEM3-CPU and GEM3-GPU compared to BWA-MEM, on the P9+V100 CPU+GPU platform. . . . . . . . . . . . . . 174
10.5 ROC Curves: Performance and Accuracy comparison Illumina Sim. single- and pair-end (100nt) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 178
10.6 Performance and Accuracy comparison with the most relevant mappers on the state-of-the-art using simulated data . . . . . . . . . . . . . . . . . . . . 180
10.7 Performance and Accuracy comparison with the most relevant mappers on the state-of-the-art using real data . . . . . . . . . . . . . . . . . . . . . . . . . 181

# List of Tables


6.1 Hardware specifications of the experimentation platforms . . . . . . . . . . 104
6.2 Warp instructions executed per query base . . . . . . . . . . . . . . . . . . . . 111

8.1 Dynamic Programming tables for sequences $P$=TAGAC and $T$=ATCGAG . . 131
8.2 Ratio of effective GDRAM accesses versus estimated GDRAM accesses . . 138
8.3 Detailed performance metrics for best performing cases . . . . . . . . . . . 140

10.1 Synthetic and Real (Pair-end and Single-end) datasets for the experimentation 168
10.2 Overview of the top 5 mappers with best performance for all the datasets (Synthetic and Real; Pair-end and Single-end) . . . . . . . . . . . . . . . . 176
10.3 Overview of the top 5 mappers with best accuracy for all the datasets (Synthetic and Real; Pair-end and Single-end) . . . . . . . . . . . . . . . . . . . . . 177

# 1
# Introduction

*"Success is a journey, not a destination. The doing is often more important than the outcome."*

**Arthur Ashe**

"This chapter presents a general overview of the thesis, by introducing the following subsections: the scientific context, the main motivations of this research, its general and specific objectives, and its contributions. A final subsection divides and summarizes the content of the thesis presented ahead chapter by chapter. In addition, we explain the involved collaborations along all the years and their outcomes."

In this chapter, we present a general overview of the thesis, by introducing the following subsections: the scientific context, the main motivations of this research, its general and specific objectives, and its contributions. Chapters 2 and 3 complement this introduction with more detailed information to cover the concepts necessary to understand the contributions of the thesis. Subsequent subsections break down the content of the thesis presented below, and the final chapter describes the collaborations involved throughout all the years.

## 1.1 Context

This section provides a brief introduction of the context, theoretical framework and problems in the field of HPC applied to bioinformatics. The following sub-chapters will go into more detail in each of the points. Our intention is that they serve as an introductory guide to the reader for the following chapters 2 and 3. The objective is to introduce the state of the art and the trends of HPC and Bioinformatics and to understand the problems that are interrelated and the main motivation of the thesis.

### 1.1.1 Sequencing and its applications

Genetic information is made up of nucleotides, usually coded using the A, C, G and T letters [3]. Determining the order of nucleic acid residues in biological samples is an integral component of a wide variety of research applications. Over the last fifty years, large numbers of researchers have applied themselves to the production of techniques and technologies to facilitate this feat of sequencing DNA and RNA molecules. This time-scale has witnessed tremendous changes, moving from sequencing short nucleotides to sequencing millions of bases, from struggling towards the deduction of the coding sequence of a single gene to rapid and widely available whole genome sequencing. The ability to measure or infer such sequences is called sequencing, and is imperative to current biological research.

The sequencing technology for the acquisition of genomic data is evolving rapidly, showing an exponential increase in the throughput achieved and a great reduction in costs. The cost reduction of genome sequencing is allowing most healthcare centres, clinics, and research institutions to install their own sequencing facilities. The prompt adoption of this technology is going to provide significant benefits for society. Soon we will see applications of personalized medicine to improve prevention and detect a wide range of health conditions, accelerate diagnosis and enhance treatments, especially for complex diseases such as cancer. The main consequence of the adoption of this technology will be the generation of an unprecedented

large amount of data, most coming from public genomic population studies. This situation will boost research in all aspects of health care [4], with the objective of improving data correlation.

In the near future, the cost of sequencing and computation is going to decrease faster than the cost of storage resources. Consequently, for specific scenarios, re-sequencing is going to become a more cost-effective option than using complex high-throughput storage solutions for maintaining all the raw sequencing data. Therefore, the availability of affordable, commodity computational systems for genomic sequencing analysis will be key for the success of personalized medicine in hospitals and health care institutions [5].

Next-generation sequencing (NGS) technologies, or also called second-generation sequencing, provide a list of techniques that allow fast and affordable DNA and RNA sequencing. This is achieved through massive parallel sequencing, allowing for millions of short sequences of nucleotides of hundreds of bases long to be sequenced in a shorter period of time. The complete human genome can currently be sequenced in less than a day with a cost slightly below a thousand dollars (see Figure 1.1).

However, increased speed and reduced costs are not the only advantages of the new sequencing machines. They also allow the identification of novel variants of the genome (e.g. mutations, rare cancer variants), given that the previous phase of discovering each of the genetic mutations involved in a disease is not needed. NGS techniques also require less DNA/RNA sample material as input to be analysed, and have higher experimental reproducibility than traditional methods. For these reasons, it is expected that medical research related to genetic diseases will speed up over the following years.

Several companies, such as Illumina, Ion Torrent and BGI, currently provide NGS solutions. Illumina techniques attach a fluorescent signal to each base, which allows the identification of all bases at the same time. In contrast, Ion Torrent measures the release of protons that occurs when individual bases are incorporated by the enzyme DNA polymerase. BGI technology is based on nanoball sequencing, in which rolling circle replication is used to amplify small fragments of DNA.

Third generation sequencing systems (TGS) are going to provide a significant advance in genomics [6]. Additional features promised by TGS are substantial reduction of sequencing costs, longer sequence input data, and random distribution of acquisition error. These features will allow new methods of genomic analysis and sequencing quality improvements of up to two orders of magnitude [7]. From here, new kinds of analysis will be affordable like Haplotype analysis that determines a genomic variation origin from the mother or father, or

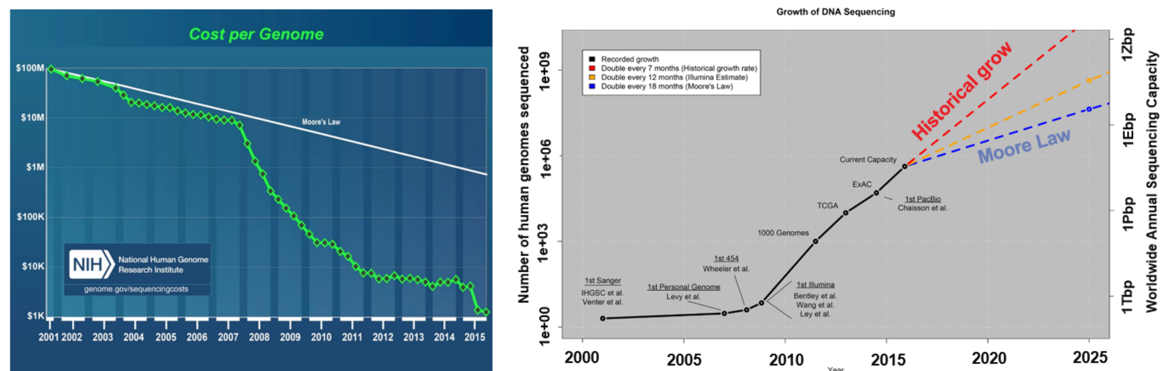


**Figure 1.1:** Progressive reduction in the costs of sequencing in relationship with the Moore's Law.

detection of structural genomic variations as reshuffled DNA commonly presented in cancer. TGS companies, like Pacific Biosciences and Oxford Nanopore are continuously announcing new advances on their devices closing current technical gaps.

NGS technology exacerbates the computational requirements for the sequence analysis. Usually remarkably high sequencing error rate demands more sequence coverage (redundancy), and the (up to thousand times) larger read lengths puts much more pressure on those algorithmic steps having quadratic complexity like sequence alignment. These and other similar factors demand more efficient data processing methods and a better use of modern computational resources.

### 1.1.2 Downstream applications: Genomic sequencing pipelines

The downstream sequence analysis is composed by three different stages (1) primary, (2) secondary, and (3) tertiary analysis. Currently, primary and secondary analysis are the stages identified as the most computationally expensive, where large dataset processing can take several days to complete.

Primary analysis, also referred as base calling, is the data acquisition step from the sample to the DNA representation. It involves the collection of the chemical data (such as the light intensity) from the sequencer machine into scores representing the DNA/RNA strands. This sequencing step has been greatly improved over the previous years, and most sequencing applications perform the base call automatically and in real-time (RTA). The main computational methods used for the primary analysis are based on Computer Vision and Machine/Deep Learning techniques. Current Illumina machines could take a couple of days to process a hundred whole human genomes in parallel.

Secondary analysis involves the mapping and alignment of the collection of short nucleotide sequences from a sequencing system into a full sequence (Short Read Mappers), and then finding any genetic variants from the reference genome (Variant Calling). Secondary analysis involves managing a large amount of data to be processed with complex methods and therefore has a large computational and storage demand. The most common methods used are the Burrows-Wheeler Alignment (BWA) [8] and the Genome Analysis Tool Kit (GATK) [2]. During the first stage, BWA performs the alignment and mapping, whereas GATK is afterwards in charge of the identification of the relevant genomic variants. For the case of Whole Genome Sequencing, this analysis could take 36 hours on a modern HPC node [2]

Finally, tertiary analysis involves the interpretation of the data to assess the origin of the variants and the functionality of each sequence. In this stage, the lab data, biological data and clinical data are combined to determine the relevance of the findings into disease aetiology and disease prevention.

**Downstreaming tools: Short Read Mapping**

As already described, secondary analysis is made up of different stages where the short read mapping stage is the most computationally expensive. Short read mappers are software tools used in most applications involving high-throughput sequencing, so they need to be continually improved to meet processing time requirements of constantly growing genomic datasets. Modern mappers rely on seeding heuristics, which makes them fast but inexact. They encompass complex algorithms to solve approximate string-matching problems, where the short read sequences and the reference do not match perfectly due to biological divergences or measurement errors of the sequencing machinery. The traditional aim of mapping algorithms is to pursue a multi objective target of optimizing read matching accuracy and data processing speed while maintaining a low memory footprint.

This thesis will explore the applicability of heterogeneous HPC systems to process downstream sequence analysis. More specifically, we are interested in accelerating the short read mapping problem as it is described as one of the most computationally expensive parts of the pipeline process. Our aim is to show the benefits of computational accelerators like GPU systems in bioinformatics to reduce the processing time and cost of analysis, increasing the democratisation of this kind of analysis.

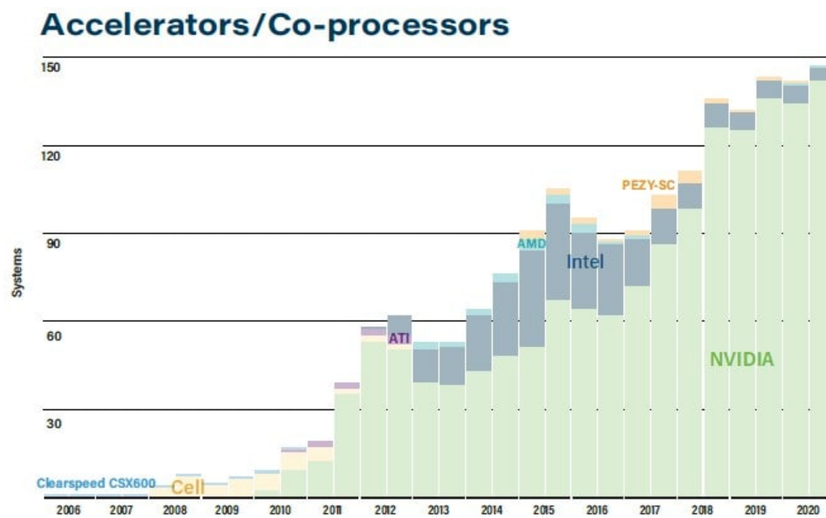


**Figure 1.2:** Percentage of super-computers using accelerators in top500

### 1.1.3 Heterogeneous computing systems

Heterogeneous computing makes reference to computational systems composed by different types of processors and where different software and hardware components interact to solve a computational problem. Current applications must exploit the characteristics of modern heterogeneous systems so that specific tasks of the applications can improve their execution performance or energy efficiency. This will be reflected in a reduction of processing cost and time for the users.

Current compute nodes usually combine a general processor (Host) with multiple accelerators (Devices), whose architecture usually targets a specific application domain (DSA), as opposed to general-purpose architectures (e.g., CPUs).

In the last years, heterogeneous computing has seen an increasing irruption in the HPC field. Data centres and cloud providers (for hyperscalers) started to adopt accelerators (e.g., GPUs, Intel Phi, FPGAs, TPUs. . . ) as a solution to cover the increasing demand for computing and higher scalability. Figure 1.2 shows the growing adoption of accelerators of the top 500 super computers: around 40% of the top-500 performance in 2019 are provisioned by accelerators.

CPU designers faced two physical design limitations that heavily affect the design principles of current general-purpose architectures: the end of Dennard's scaling and the slowdown of Moore's Law [9]. Dennard's scaling shows a relationship between transistor size and power density: since both voltage and current scaled with transistor length, more transistors can be

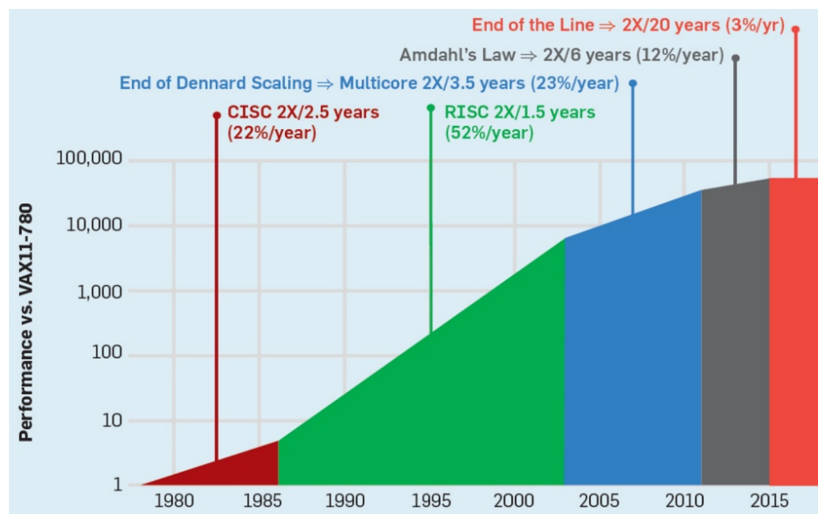


**Figure 1.3:** Computing architecture eras by physical design limitations events

used without an overall increase in power consumption. When that relationship no longer held, it was no longer possible to improve CPU performance simply by increasing the clock frequency, due to higher power consumption and limitations of thermal dissipation of the device. Technically, aspects of Moore's Law continue to move forward, but this is not along the lines of the true notion of Gordon Moore's observation because you no longer get more transistors for lower cost. Also, as adding more general-purpose cores is not improving the execution time of the serial part of the applications, the next trend in computer architecture is to design specific domain application architectures that do a reduced set of specific tasks very well.

Following figure 1.3shows a decreasing trend in performance gains for general processors over the years. The improved performance of general processors was driven mainly by technological silicon advances [9]. The reduction of Dennard's Scaling and Moore's law in the last years have prevented the easy performance increase of general purpose processors and promoted the adoption of accelerators.

The level of heterogeneity in modern computing systems is gradually increasing as further scaling of fabrication technologies allows for formerly discrete components to become integrated parts of a system-on-chip, or SoC. For example, many new processors now include built-in logic for interfacing with other devices as input/output (SATA, PCI, Ethernet, USB, memory controllers . . . ), as well as, programmable functional units and hardware accelerators (GPUs, cryptography co-processors, programmable network processors, encoders and decoders for multimedia,...).

Heterogeneous computing systems present new challenges not found in typical homogeneous systems. The presence of multiple processing elements raises all the issues where the level of heterogeneity of the system can introduce non-uniformity in system development, programming practices, and overall system capability. Heterogeneous computing GPU challenges studied during the last years present many issues that the application programmers must solve: data and compute irregularities, data pre and post processing, data allocation, management of the internal memory limited size, data movements between system memories, identification of potential parts of the processing that can be offloaded to other resources like CPU, load unbalance between processors and management of different memory space addressing.

On top of all these changes, technologies as multi-chip integration, programmable function units and reconfigurable architectures (like CGRAs) will further promote the emergence of DSAs for specific fields.

Personally, I feel that this decline of computers as a general-purpose technology is opening a new era of computer architecture innovations. In a close future, we will witness many architecture changes inspired by algorithmic advances on the software side. In the past, some representative areas had similarities with the design of current accelerator architectures and software stack. These innovations are currently present in areas such as compression, cryptography, graphics, audio and video coding-decoding, network processing and machine learning.

One of the motivations of this thesis is to pave the way and explore the algorithmic, software state and bioinformatics framework challenges that future accelerators will need to face. Many learnings from the use of GPUs as a demonstrator platform in this thesis are being transferred to processor dedicated architectures.

## 1.2 Motivation

Personalized medicine is changing the patient diagnosis and treatment, replacing the "one size fits all in" traditional approach. Our health is determined by our inherent differences combined with our lifestyle and environment. The combination of genomic information and other clinical and diagnostic information is one of the key components in identifying individual risks of developing a specific disease. Access to this information allows the disease to be diagnosed in very early stages before the patient's symptoms even develop.

Never before it has been possible to predict how each of our bodies will respond to specific

interventions or to identify whether we are at risk of developing a specific disease. New possibilities are now emerging as we bring together new approaches, such as genomic sequencing data and informatics, wearable technology and patient monitoring. The interconnection of these innovations is making possible to move to a new era designated by the personalised medicine. Also, other areas as drug design, feed design, agriculture, take benefit of all these advances. Sequencing is a basic operation for synthetic biology projects.

Genomic sequencing is the key component of the new advances on medicine, and its democratization is an important step on the accessibility for the patient. The big throughput of the sequencers and the rapid reduction of sequencing costs is allowing new applications of genomics on the precision medicine. Some clear examples are large-scale population analysis, in-vitro prenatal testing, or early cancer detection. Downstream analysis is one of the fundamental computational pipelines on genomics that is present in most of all the analysis. The cost of sequencing is a real limiting factor for the field. Providing cheaper data acquisition will enable new type of applications on precision medicine moving the cost and requirements to the computational part.

A clear example of early disease detection, more specifically early cancer detection, is ctDNA (circulating tumour DNA). ctDNA is released into the blood stream from tumours, and therefore their levels can be measured with a simple blood sample (which is referred as “liquid biopsy”). This simplifies cancer diagnosis, which can now be performed with non-invasive (and less expensive) methods, as well as at earlier stages of the disease (stage I vs stage IV cancer disease). However, the amount of ctDNA in circulation is usually low. As a result, high sequencing depth is used to increase the accuracy of detection of these low-abundance variants. This increases the compute requirements by three orders of magnitude when compared to regular DNA sequencing. Usually, Whole Genome Sequencing requires for the genome to be re-sequenced around 30 times (called “coverage”), whereas ctDNA detection requires a coverage of 60.000 times.

The main challenges to achieve real-time genome sequencing when using commodity computation resources are algorithmic and computational. We need both advanced algorithms to ensure reliable, high-quality and robust analysis, and also energy-efficient high-performance compute devices to cope with the large requirements.

In addition to the previous motivations of the problem, we would like to highlight that the claims and solutions reached in this thesis are supported and supervised under a collaboration with National Center of Genomic Analysis - CNAG, which gave us the opportunity to validate and benchmark the results in a real environment of genomic sequencing. This enabled end-to-

end analysis and software verification, that is, from wet-lab to the final delivery of variant results.

## 1.3 Objectives

This subsection introduces the main objectives of this thesis. The main goal of this project is to improve the state-of-art of the performance and accuracy of the genomic sequencing downstream process. We will focus our proposals on the mapping and alignment phases for high-throughput sequencing analysis.

In order to reach this goal, we will evaluate and discuss the benefits of the use of heterogeneous computational platforms for genomics data processing. For that, we want to improve the existing methods for hybrid hardware systems. We optimize the genomic analysis pipelines using NVIDIA CUDA architectures and show the benefits of applying accelerators to this field, presenting GPUs processing capabilities that enable a significant shift in the speed of genomic data analysis.

Our final goal is to deliver a solid software pipeline accelerated by GPUs. The system will be able to process the streamed sequencing data obtaining a high quality output that will guarantee the necessary reliability for subsequent biomedical analysis. The work presented in this thesis shows the suitability of device-specific accelerators (e.g., GPUs) to bring HPC-bioinformatics capabilities to cost-effective commodity enclosures and make real-time sequencing affordable.

With the aim of achieving these objectives, we have devised the following specific sub-objectives:

- Identify and characterize the most relevant algorithms used by the bioinformatics community.
- Determine the most relevant performance bottlenecks of the studied algorithms and provide an algorithmic and architectural reference platform to provide analysis and relevant results regarding the performance, energy and cost requirements in a representative production infrastructure.
- Propose novel parallelization strategies at thread and task level in order to efficiently leverage the high computation capabilities provided by the accelerator.
- Evaluate the performance impact of the solutions proposed in comparison with state of the art solutions well adopted by the community.

- Provide the appropriate interfaces and promote standards to simplify the comparison of the work presented with previous published research.
- Validate the work with representative data and an appropriate, well-defined methodology widely-adopted by the community. This will be addressed by adopting a de facto standard on protocols and applications.
- Make contributions accessible to researchers, institutions and the bioinformatics community in general. This will be addressed through the use of public repositories for the code, and disseminating the work at relevant international conferences and technical talks.

## 1.4 Methodology

This section describes the research methodology applied on this work, which can be summarised in the following stages:

1. Study of the state-of-art.
2. Algorithmic analysis of the basic building blocks of the relevant applications.
3. Proposal of the potential contributions.
4. Experimentation (a) and validation of the proposals (b).
5. Dissemination.
6. Technological transfer to production environments.

Figure 1.4 represents these stages of research methodology. By applying this methodology, the critical objective is to identify the following potential bioinformatics application problems:
(A) possible accuracy limitations
(B) problem size restrictions
(C) computational cost problems on bioinformatics applications.

More specifically, the objective is to evaluate if these problems appear in heterogeneous systems. Specific tasks developed at each stage of the methodology are described in the following points.

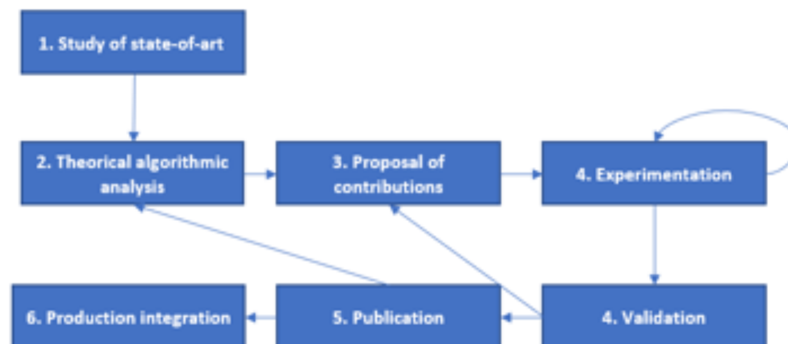


**Figure 1.4:** Stages of the research methodology

1. Study of the current state-of-art on sequencing and heterogenous architectures

   This step will require a good understanding of the different sequencing technologies, identification of the best tools available to the community, both in terms of performance and accuracy, and algorithmic analysis of the fundamental building blocks included in these tools.

   (a) Characterisation of the state-of-art in whole genome sequencing pipelines, identifying the current mapping and aligning algorithms along different sequencing pipelines:

      i. Determine representative data workloads of genomic analysis and computational system requirements, potential hardware resource limitations for each identified workflow (or full application).

      ii. Identify the algorithmic complexity and memory footprint for a cost comparison analysis.

      iii. Classify each pipeline component by its turnaround time and cost effectiveness in today's production environments.

   (b) Analyse the state-of-art and identify the limitations of the most important algorithms for an end-to-end pipeline from the previous point. Search for possible algorithmic alternatives available in the literature, books and reference papers on the current field.

   (c) Search for available emerging parallel heterogeneous architectures (e.g., GPUs) and their characteristics, and explore their suitability to implement the above

bioinformatic algorithms.

i. Recognize the parallel computation patterns suitable for highly parallel devices. Chapter 3 will explain these patterns on the context of bioinformatics.
ii. Study the GPU microarchitecture, host-device interaction, software stack and system level differences with current traditional general-purpose platforms.
iii. Study the performance bottlenecks that are intrinsic to target computer architectures, such as memory management limitations: pre-defined sizes, data transfers, data layout changes to exploit data locality, job decomposition-scheduling on irregular parallel work, data dependent tasks, and explicit parallelism definitions.

2. Theoretical analysis of the algorithmic building blocks

   After becoming familiar with the principal components of a genomic pipeline and its requirements, it is necessary to analyse and characterise the main building block algorithms (from step 1 of the methodology) and search for better implementations that increase hardware efficiency:

   (a) Explore potential algorithmic improvements to reduce computational and memory complexity.
   (b) Explore potential hardware-aware optimisations to improve the performance of the building blocks, considering alternative data layouts to increase data locality, work regularisation, code specialisation, increasing and exposing parallelism and increasing memory level parallelism.
   (c) Analyse how to parallelise the algorithms and identify potential suitable parallel patterns, studied on 1c, and analyse their applicability to GPU architectures using computational performance models (e.g., roofline models)
   (d) Evaluate alternative algorithms with improved performance and identify their limitations in terms of potential risks on the accuracy of the results.

3. Proposal of contributions

   We will be using hybrid systems with CPU and GPU resources as the target platform for the proposed ideas. After identifying a list of potential bioinformatics building

blocks (from step 2 of the methodology), we will follow these steps to evaluate the proposals:

(a) Define and create the input datasets, both from real use cases and from simulators (synthetic) [10]. Some algorithms will require an image of the internal, temporal data generated by the applications from real inputs. The appropriate infrastructure must be built for that purpose.

(b) Build a "sandbox" environment to integrate all baseline implementations and contributions with enhancements (CPU or GPU), unify all possible input and output formats and provide flexibility for future testing and comparison

(c) Design a baseline program implementation of the state-of-art algorithms on CPU for experimentation and validation purposes.

(d) Develop early implementations of the algorithms and improvement proposals, to run on both CPU and GPU.

(e) Compare the performance of different implementations using the "sandbox" environment to isolate the analysed modules without interferences from other parts of the pipeline.

Early results from this stage will provide the necessary feedback to assess whether the proposed ideas are feasible and provide promising performance results. In that case, we will move to point 4, experimental steps; otherwise, an additional iteration to step 2 and 3 will be required.

4. Experimentation

   Here we describe a general experimental methodology and guidelines for the entire thesis. The experimentation for each algorithm and proposal consists of (1) defining and configuring the computational environment, (2) validating results, and (3) evaluating the improvements. More details on the experimentation are given on every chapter, depending on its specific requirements.

   (a) Setting the experimental environment and datasets

      - Performance evaluation requires access to heterogeneous computational resources with characteristics consistent with the environment present in the production centres. Each experimentation will run on a system with

GPU and CPU processors, carefully selected as representative for HPC and/or low power embedded devices.

- Different CPU architectures as x86, aarch64 or ppcle, will be evaluated, in conjunction with different generations of NVIDIA GPU architectures.
- Since full control of the node resources is necessary, performance analysis will always run in dedicated or exclusive mode to avoid possible noise introduced by other processes
- It is necessary to set representatives datasets for the experimentation. A combination of real and synthetic data will be used, depending on the specific requirements for each experiment. Synthetic data generated by simulators (e.g. mason [10]) provide golden observable results for accurate validation.
- Representative parameters for each algorithm will be selected from the study of real mappers: the experience from point 3 will help in this task.
- Several custom tools must be developed to support the development, debugging and profiling tasks.
- Last tool chains, tracers, and profilers providing samples of hardware performance counters are required for empirical analysis (some examples are likwid, vtune, perf, nvprof and nvvp).

(b) Research methodology for results validation

The aim of this phase is to validate the correctness of the results and prove the benefits of the contributions from phase 3. The main idea is that each specific proposal will be completely isolated in the sandbox and will contain the necessary infrastructure for comparison with the proposals of relevant research in the state of the art.

- Simulated data and data dumps from other applications will be used.
- The CPU baseline reference implementations developed in point 3 and external reference implementations from other researchers will be used to validate the correctness of the results.
- Validation of mature proposals from expert scientists as external collaborators.

(c) Research methodology for evaluation of the proposals

Definition of the environment experimentation: tools, computational hardware, datasets, and evaluation metrics. The evaluations will be based on empirical analysis by using tracers, profilers, and hardware counters sampling.

- Define the libraries, SDK and software stack used in the experimentation.
- Define the general performance metrics to consider, such as tasks per second, raw and effective memory bandwidth, transfers per second ...
  - Define the hardware dependent evaluation metrics, such as number of instructions executed, instructions per cycle (IPC), cache misses ...
  - Definition and adoption of metrics that are standard and well known by the community for specifying the work done by each algorithm (e.g., Giga Cells Updated per Second (GCPUs))
- Performance assessment on the same sandbox comparing our proposals with implementations from the state of the art.

NVIDIA software stack support for GPUs has been increasing in recent years. CUDA 2.3 was released at the time that this project was started, with the consequent limitations on tools and development API. These limitations impacted our work, forcing us to develop our ad-hoc tools for the problems we wanted to solve.

If the validation process does not successfully pass the required performance target, we must go back to point 3 and look for new proposals.

5. Dissemination of the work and reproducibility The work described and relevant results have been published in high-impact international conferences and journals with peer review mechanisms. Additionally, the work has been presented and discussed with other researchers, PhD colleagues and mentors from the internships.

   In order to provide full reproducibility of work and results to external researchers, all source code, datasets and scripts for performance experiments will be published as open-source in public repositories. In addition, the work has been documented and usage scripts are published as open-source and free access in repositories.

   We will present our contributions and the developed tools to the bioinformatics researchers of different genomics research institutions, such as CNAG and CRG from Barcelona. Additionally, we will discuss the requirements to run our tool in production.

A final dissemination of the work is done by this document.

6. Production deployment (framework level)

   Our goal is to verify that all the proposed ideas and contributions, which have been already published, are also valid in a real production environment. For that purpose, we will cooperate with the CNAG centre on the development of the mapper tool GEM3. We will integrate our GPU-accelerated algorithms on the GEM3 mapper to be used on a real, end-to-end production pipeline, from wet-lab to the final reported variants. This will require working with a multidisciplinary team, in collaboration with the CNAG centre.

   We will develop a library API called GEM-cutter providing an interface to make transparent all the GPU complexities for the developers, including CPU and GPU batch processing, task schedulers, custom engines for data partitioning and transfers, and others. The library will be released as an open-source project that can be integrated on external projects and tools. The GPU version of GEM3-mapper uses the GEM-cutter library, proving its value.

   The experimental sandbox will be reused with the GEM3 mapper to simplify the validation process. Also, a tracing system will be integrated on the mapper to identify the input data (e.g., low-level profiling API NVIDIA). For the experimentation and validation, we will get support in later stages from our collaborators, to validate that the data is representative and generated results are valid.

   Genomic data from available public repositories (GenBank) and provided by CNAG will be collected for the experiments. State-of-the-art mapper applications found in the literature, whether accelerated by GPU or using standalone CPUs, will be installed on the same system for comparison. Finally, new performance and accuracy metrics are defined, such as sensitivity and specificity, for a comparison between algorithms and implementations.

## 1.5 Contributions

The ultimate contribution of this work is to develop a full-fledge short read mapper, improving the state-of-art on performance and accuracy terms, and providing a fully functional accelerated solution using GPU architectures. The following contributions result from the accomplishment of the previous goal:

- Identification, characterization and performance analysis of the most relevant and computationally expensive algorithms for short read mapping and sequence alignment tasks in common genomic pipelines.
- Algorithmic contributions that improve the efficiency and performance of several bioinformatics algorithms. Most relevant improvements presented consider reducing computational complexity, improving data locality, leveraging fine grained and massive parallelisation, and balancing the workload on GPU accelerators. The specific algorithms are:
  - *String matching and retrieval (details on chapters 4, 5, 6, 7):* FM-index occurrence count and FM-index position retrieval.
  - *String comparison and alignment (details on chapter 8):* Bit-Parallel Myers and text filtering.
- A message-passing distributed paradigm to program heterogeneous systems suitable for bioinformatics applications. The proposed abstractions allow the user to program the system without expertise on GPUs. A library has been developed as an example, which includes the highly optimised versions developed on the thesis for the core bioinformatic algorithms.
- The following dissemination activities:
  - Publication of results in international conferences, recognised journals; in addition to posters and technical talks on relevant research centres and events, all described in detail in chapter 11.
  - All the source code, documentation, datasets and pipeline scripts are released with an open-source license on public repositories for a fully reproducibility of the results.
  - Technical support to genomic production centres and other external researchers or developers of the bioinformatics field regarding questions on the published tools, data and documentation from this thesis.
- A real genomic aligner (gem-3) that integrates our proposals, validated on real bioinformatics pipeline cases (exome, genome and bisulfite). The work has been compared to similar state-of-the-art tools. The following codes and applications were developed and released for that purpose:

**git-cutter** Gem-Cutter, a GPU core-library that includes all the basic block GPU algorithms presented in the thesis as modules and the required schedulers that can be used in different bionformatics applications through their general API. The modules are described in the following chapters.
https://github.com/achacond/gem-cutter

**git-gem3** Gem3-GPU, a batched-oriented version of a mapper that integrates all the GPU modules of GEM-cutter and all the improvements.
https://github.com/achacond/gem3-mapper

**git-bench** Gem-Bench, an automated framework using the methodology applied in this chapter that includes all the workflows, scripts and data used for a mapping evaluation. https://github.com/achacond/gem-gpu-benchmark

- Internships and collaborations with international centres: Improved NVBIO open-source official bioinformatics library from NVIDIA [11] Deployment in production and full validation of the genomic mapper with GPU support developed in this thesis and assist to the CNAG infrastructure definition.

## 1.6 Collaborations

Several collaborations were carried out related to this thesis. The most relevant collaboration is the development of a full fledge genomic mapper and the provision of a full library accelerated by GPU.

A collaboration between the Autonomous University of Barcelona (UAB) and the National Center of Genomic Analysis (CNAG) was a backbone component of the thesis. The work was validated with real data and integrated on the production pipeline of CNAG. These activities had a positive impact on the centre that contributed to increase the sequencing quality, reducing the turn-around time and cost of the internal and external international projects of the centre.

The internship carried out at NVIDIA headquarters offices at Santa Clara, California, was providing the opportunity to be working on their open-source Genomic Library (NVBIO) accelerated by GPU under the libraries team. The acquired experience on this thesis was a positive impact on their library, contributing with new algorithms and improving the performance of other parts. Most of the work was publicly published in the NVIDIA repository.

Additionally, several works were carried out internally at NVIDIA Research on parallel

dynamic data structures with impact on their programming languages and software systems present on the current NVIDIA GPUs. Some described developments from this thesis and further NVIDIA collaborations contributed to award the compute department of CAOS at UAB the certifications of NVIDIA Research and Teaching Excellence centre.

This internship at USA gave to the author the opportunity to stay at John Owens Lab in UC Davis University as a research visitor. It was very worth to learn how other research teams work and exchange knowledge on the same research field.

In addition, it is worthy to mention the impact of the last years of research and development carried out as employee at Arm (UK), Xilinx (Ireland) and NVIDIA (UK) in parallel to this thesis, that highly influenced the final results of the project and my vision of the field.

There is a dedicated chapter on the thesis for collaborations and acknowledges because of their relevance on the outcome of the project, describing with details the previous points. Carrying out the previously described collaborations was positive personal experience contributed to improve the overall quality of the thesis and, more importantly, giving to me the opportunity to grow and consolidate my current skills as a researcher, engineer and human being.

## 1.7 Thesis outline

The work presented in this thesis is divided in the following chapters. Chapters 2and 3 show the basic concepts and the context of genomics on HPC necessary to understand the motivations of the work. Chapters 4, 5 and 6 describe basic concepts and algorithmic proposals related to GPU indexed search acceleration. Chapter 8 shows the proposals related to GPU text filters. Chapter 7 shows strategies for integrating the core proposals into a real genomics application. Finally, chapter 10 provides the details of the final experimentation on production environments and chapter 11 shows the main conclusions of the work.

- **Chapter 2: Bioinformatics Analysis**; This chapter contains a brief introduction to most widely adopted genomic sequencing technologies, the computational challenges that arise and their impact in the current analysis. State-of-the-art bioinformatics pipelines are introduced and the algorithmic problems faced on the latest computational platforms are described. We present the blueprint of our own GEM3 aligner software, describing each of their stages and the aforementioned methodology utilized to corroborate the algorithmic proposals effectively. At the end of the chapter, we present a review of the state-of-the-art on the different alignment

tools with the aim to provide to the reader all the necessary concepts required in the following chapters.

- **Chapter 3: Heterogeneous Computing**; We introduce heterogeneous computing systems and discuss the main challenges of the usage of massive parallel systems and the interaction with general-purpose processors for HPC applications. We also describe the computational problems presented on the thesis due to the utilization of heterogeneous systems on bioinformatics pipelines. Finally, we explain why current state-of-art software proposals must be improved tackling scalability and efficiency issues.

- **Chapter 4: FM-index - text indexing building blocks**; The basic primitives and index structures related to the FM-index are introduced. Computational analysis and benchmarking with real genomic data is carried out to characterise the performance behaviour of the primitives on heterogeneous systems. Results offer the motivation for the proposals presented on the following chapters for the seeding steps on the mapping.

- **Chapter 5: FM-index - algorithmic and design proposals**; The algorithmic and design proposals for the text indexing core primitives are described and discussed. The proposed approach extends the properties of the state-of-the-art. The introduced indexes achieve lower memory footprints and fewer random memory accesses, mitigating memory hierarchy congestion and improving the overall performance of the system. These proposals are architectural oblivious and further discussion and experimentation on CPU and GPU systems are provided to quantify the performance improvements.

- **Chapter 6: FM-index: GPU Parallel designs for LF-mapping primitive**; This chapter presents a fine grain parallelization strategy of backward search for massive parallel and vector-based systems. The proposal is compared to other GPU-based proposals found in the state-of-the-art. Our approach exposes higher performance and avoids the very restrictive on-chip memory limitations compared to the previous proposals. Further discussion and experimentation are presented with real genomic data on different GPU architectures to validate and quantify the contributions.

- **Chapter 7: GEM3: approximate pattern search in a mapper GPU**; This chapter introduces concepts and strategies used by state-of-the-art read mappers to deal with

the error sequencing rate and human mutations on the mapping process. We propose and describe stages for adaptative seed search and decodification on the GEM3 mapper, which accelerate the execution on vector processors like GPUs. Additional algorithmic improvements are presented as memorization to boost the performance by reducing the computation. Finally, all the combined proposals are evaluated using real data on GPUs and CPU, showing their clear advantage on performance and power consumption terms.

- **Chapter 8: Text filtering building blocks**; This section introduces basic concepts used in filtration stages of a mapper. Cost-effective algorithms for filtration are necessary to reduce the huge amount of work generated by the previous stages, meanwhile assuring high levels of accuracy. Fine-grain parallelization strategies for the K-MER and Bit-Parallel Myers algorithms are proposed. The experimentation shows big performance gains compared to the current state-of-the-art. Computational analysis and benchmarking with real genomic data is carried out to characterize the performance behaviour on heterogeneous systems, motivating the proposals presented in the following chapters.

- **Chapter 9: GEM-Cutter: high-performance bioinformatic library**; This chapter presents all the internal strategies applied to integrate the thesis proposals into the GEM3 mapper. Fine grain collaboration and highly optimized transferences between the processors are essential to reach high efficiency. We propose a scheduling method to increase the task parallelism and reach higher utilization of the computing resources. We discuss how strategies such as data structure specialisation and task distribution are essential and must be combined with problem decomposition and data regularisation to reach high performance. Finally, we introduce an interface based on the message passing model to abstract the heterogeneous hardware.

- **Chapter 10: GEM3-GPU Mapper benchmarking and experimentation**; This section evaluates all the integrated proposals in the GEM3 mapper using genomic data from different sequencing technologies. A detailed comparative with other state-of-the-art GPU-accelerated proposals is carried out in terms of performance and accuracy. The different implementations of each GEM3 mapping stage, using CPU or GPU, and the impact of different parameters is analysed both in performance and work terms. The experimentation is extended to different devices, from low-power embedded devices to large-scale systems for data centres. The analysis shows

how the proposals surpass the performance of current state-of-the-art mappers in more than one order of magnitude at the same accuracy level, but also enables the possibility to reach better accuracy results at the expense of a reasonable performance reduction.

- **Chapter 11: Conclusions**; The experiences gained and the conclusions derived from this thesis are described. We outline the viable open lines to provide more efficient tools on heterogeneous systems.

# 2

# Bioinformatics Analysis

*"The human genome is a life written in a book where every word has been written before. A story endlessly rehearsed."*

**Johnny Rich**

This chapter provides key concepts by introducing the following subsections: an introduction to DNA and genomics and sequence alignment, which includes an explanation of the full alignment pipeline. The two final subsections are focused on the GEM3 mapper and its GPU version created for this thesis.

The current chapter introduces the reader in the key biological concepts necessary to understand the structure of the genetic material, the history of genomics and the advances in the cost of genome sequencing. Afterwards, a full review on the process of genomic sequence alignment is presented. The two last sections of the chapter are specifically focused on the GEM3 mapper and its GPU version.

## 2.1 An introduction to DNA and genomics

Over the following section, some general aspects of the DNA biology and structure will be provided, which are necessary to understand future aspects of the sequence alignment. A brief review is then given of the major sequencing advances that allowed modern sequencing alignment, leading to the current cost-effective process.

### 2.1.1 DNA structure

The DNA (deoxyribonucleic acid) is a complex molecule that carries the genetic code. It belongs to the family called nucleic acids, and it is located inside the cell nucleus. A DNA molecule is formed by two twisted helical chains that are connected through four types of nucleotides: adenine, cytosine, guanine and thymine (A,C,G,T). Each nucleotide connects with another nucleotide located in the second helix. The nucleotides have a one-to-one correspondence, and therefore the A nucleotides only pair with the T nucleotides, whereas the C always pair with G. Therefore, a DNA helix can be considered as a chain over an alphabet (ACGT) [3]. Knowing the correspondence of nucleotides, the complimentary chain can be deducted (see Figure 2.1).

The other major type of nucleic acid in the organism is the RNA (ribonucleic acid), which is responsible for protein synthesis. The RNA structure is a simpler version of the DNA helix, and it is composed of a single strand of nucleotides. The nucleotides forming the RNA are adenine, guanine, cytosine and uracil (replacing the DNA's thymine). Different types of RNA exist: messenger RNA (mRNA), transcript RNA (tRNA) and ribosomal RNA (rRNA). The mRNA interacts with the DNA and carries the instructions for the protein synthesis from the cell's nucleus to the ribosomes, which are the sites of protein transcription. Afterwards, tRNA will translate the information provided by the mRNA so that the rRNA can proceed with the protein synthesis.

The genome is the complete sequence of an organism's DNA. The complete genome is distributed in different blocks called chromosomes. Chromosomes contain part of an

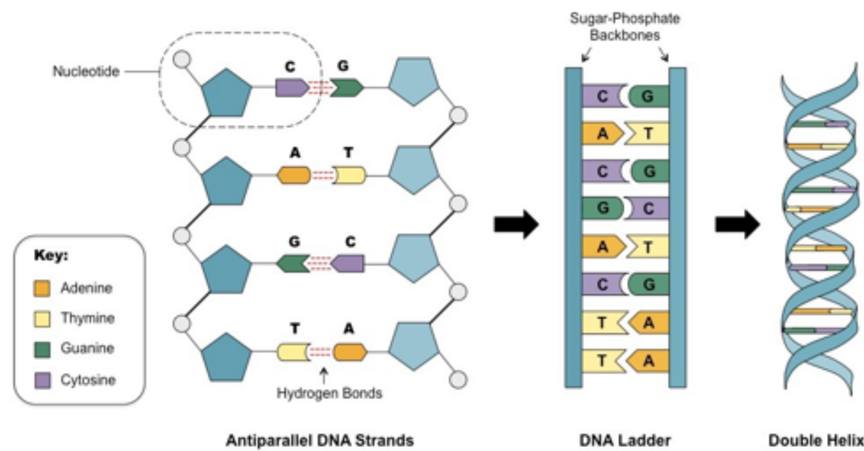


**Figure 2.1:** DNA structure showing the relationship in between nucleotides

organism's genetic sequence, and are divided in regions according to their functionality. Some of these regions are called genes, which contain the code for the protein synthesis. The adjacent regions between genes are called intergenomic regions. A human genome contains 3 x 109 pbs (pairs of bases), which corresponds to  715 MB of information (using 2 bits per base), distributed over 23 chromosomes. The entire genome contains around 30,000 genes, each one measuring between 2000 and 3000 pbs.

### 2.1.2 Evolution of DNA sequencing

Genome sequencing tools allow biologists to identify and understand fundamental regions of the DNA that have a direct impact in genetic diseases. Currently, the scientific community requires efficient tools for genetic analysis.

The genome is the complete DNA of an individual, including genes and non-codifying areas. Genomic mutations can make us more susceptible to specific diseases. Also, the stored genetic information within genes will be inherited from one generation to another, and it is the reason for which many health conditions have family patterns. If the location and function of all genes were known, then techniques for activation and inactivation of vital genes involved in disease could be explored.

Moreover, the genome study can help to understand other important regions such as regulatory regions, which have important roles in the development of certain hereditary diseases. Finally, genomic analysis is a cornerstone for phylogenetic studies to identify common ancestries between species.

Genome sequencing has been at the centre of interest in the biomedical field over the past several decades and is now leading toward an era of personalised medicine. Sequencing tools allow biologists to identify and understand fundamental regions of the DNA that have a direct impact in genetic diseases. If the genomic variations responsible for genetic diseases are known, early detection of diseases is possible, potentially improving the outcome and prognosis of multiple conditions. In the future, should genome sequencing reach even more affordable costs, personalised medicine could even lead to the design of specific drugs for each individual, or even potentially performing techniques for activation and inactivation of genes involved in disease.

Since the mid-60s, DNA sequencing methods have evolved from the laboured gel electrophoresis, through automated multicapillary electrophoresis to the next generation technologies of cyclic array, hybridisation based, nanopore and single molecule sequencing. Next generation sequencing played an essential role in the sequencing of the Human Genome, which was the cornerstone for the foundation of the field of modern genomics. Since then, sequencing evolved rapidly with an increased accuracy and marked cost reduction.

In 1965 the first whole nucleic acid sequence was produced and a technique based on the detection of radiolabelled fragments was created. This led to the first complete protein-coding gene sequence in 1972 and its complete genome 1976. However, at that time base determination involved a considerable amount of analytical chemistry. In 1977 Sanger developed the "chain-termination" technique, which was the first one to be widely adopted, and it is considered the origin of "first generation" DNA sequencing.

Over the following years the first family of commercial DNA sequencing machines appeared, which produced short reads (slightly less than one kilobase in length). Eventually, newer sequencers such as the ABI PRISM (by Applied Biosystems) allowed simultaneous sequencing of hundreds of samples and were used in the Human Genome Project.

Next-generation sequencing (NGS) has intensified the need for robust pipelines, because millions of short DNA sequences are used as input sources. Also, NGS tends to involve steps that are both time-intensive and parameter-heavy. Over the past years, massively parallel DNA sequencing platforms have become widely available, reducing the cost of DNA sequencing, and democratising the field. These new technologies are rapidly evolving, and challenges include the development of protocols for the generation of sequencing libraries and building effective approaches to data-analysis.

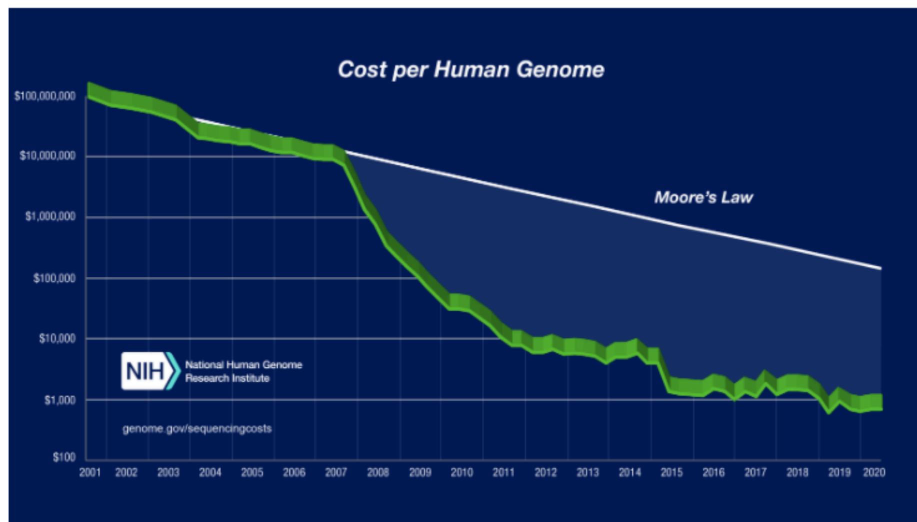


**Figure 2.2:** Evolution of costs of sequencing per Human Genome over the last 20 years, in relationship with Moore's Law [1]

### 2.1.3 Costs of sequencing

The total costs of sequencing have greatly reduced since the Human Genome Project achieved the complete first mapping and sequencing of the human genome. The cost reduction has allowed a wider availability of genome sequencing, and therefore a speed-up of the research on genomics, allowing the field to emerge over the last years. Actually, one of the goals of the most recent advances on genomic sequencing is to further reduce the economic costs of each genome sequencing, with the aim of making the techniques widely available for day-to day diagnosis.

The costs of the Human Genome Project, which was completed in 2003, are difficult to calculate, given that it was performed across multiple institutions and several years, however it was estimated to cost around 500 million dollars. However, by 2006 the cost to generate a whole human genome sequence had reduced to 14 million dollars. Over the following decade, an exponential drop in the cost of sequencing was observed, reaching 4000 dollars by mid 2015 and just below 1500 dollars by the end of that same year [1]. Generally, the current costs of a whole human genome sequencing are considered to be below 1000 dollars (see Figure 2.2).

## 2.2 Sequence alignment

The term Next-generation sequencing (NGS) refers to the techniques that allow DNA and RNA sequencing at a more rapid and economic manner. This is achieved by massive parallel sequencing, allowing for millions of nucleotides to be sequenced in a shorter period of time (the complete human genome can now be sequenced in less than a day).

However, increased speed and reduced costs are not the only advantages of these tools. They also allow the identification of novel variants of the genome (such as mutations and rare cancer variants), given that a priori knowledge is not needed. Because the previous phase of discovering each of the genetic mutations involved in a disease is not needed, it is expected that medical research related to genetic diseases will speed up over the following years. NGS techniques also require less DNA/RNA material as input to be run, and have higher reproducibility.

Several companies, such as Illumina, Ion Torrent and BGI currently provide NGS solutions. Illumina techniques attach a fluorescent signal to each base, which allows for all the bases to be identified at the same time. In contrast, Ion Torrent measures the release of protons that occurs when individual bases are incorporated by the enzyme DNA polymerase. BGI technology is based on nanoball sequencing, in which rolling circle replication is used to amplify small fragments of DNA.

### 2.2.1 Sequencing pipelines (WGS, WES and WBS)

Whole Genome Sequencing (WGS) is the process to digitalise and analyse all the individual genomic data. Sequencing machines determine the order of the nucleotides (A,C,G,T) along an individual's genome, so that DNA can be analysed to identify genomic events and mutations related to genomic diseases. As mentioned in the following sections of this chapter, the sequencing process is performed in three steps: primary, secondary and tertiary analysis.

Whole Exome Sequencing (WES) involves the sequencing of the protein-coding regions of the genome. The exomes are the regions of the DNA that contain information for the protein synthesis. Because most genetic diseases are related to the incorrect synthesis of proteins due to mutations in the exomes, WES is essential for the study of genomic conditions. It is also a very cost efficient alternative to WGS, given that the whole exome represents a very small part of the entire genome (less than 2%) but it is involved in the majority of the disease-related mutations (around 85%)

Whole Bisulfite Sequencing (WBS) requires a pre-treatment of the DNA with bisul-

fite. The pre-treatment is performed prior to the sequencing, and allows to find a pattern of DNA methylation. When bisulfite is applied, all the cytosines are converted to uracil, except for the ones that were already methylated, which will be unaffected by the process. Therefore, the amount of information is simplified to determining the percentage of a single nucleotide (methylated cytokines). Identifying methylated cytosines is useful because they are responsible for temporal and spatial gene expression, as well as chromatic remodelling. DNA methylation is also involved in embryonic development, gene expression and cell differentiation, and errors/mutations in this system can lead to diseases such as cancer.

**Somatic and germline sequencing**

When DNA or RNA samples are sequenced, a number of variants (mutations) will be identified. However, the impact of a mutation in the individual can be very different depending on where is located. The variants that occur in the germ cells (the egg and sperm) will be hereditary and will be passed to the next generations. These variants are called germline variants. On the other hand, mutations occurring in any of the other cells of an organism will not be transferred to the future generations, and are called somatic variants. Somatic mutations are involved in the development of diseases over the course of the individual's life, such as cancer. The study of somatic variations allows us to develop prognostic biomarkers and early-detection cancer techniques.

The sequencing process can be performed with a trigger towards identifying somatic or germline variations. As explained in the following section, GATK [2] is one of the most common software used during the phase of variant detection. This software has specific tool kits for the detection of somatic or germline mutations (HaplotypeCaller and Mutatek2, respectively).

### 2.2.2 Sequence pipeline: primary, secondary and tertiary analysis

The sequencing process is performed in three steps: primary, secondary and tertiary analysis. The primary analysis is also referred to as base calling. It involves the collection of the chemical data (such as the light intensity) into scores representing the DNA/RNA strands. This step of sequencing has been greatly improved over the previous years, and most sequencing softwares perform the base call automatically (real-time analysis, RTA).

Secondary analysis involves the mapping and alignment of the short nucleotide sequences into a full sequence, to afterwards find any genetic variants from the reference genome (variant

calling). The secondary analysis involves a large amount of data to process, and therefore it has a large computational and storage demand. Two of the most common methods used in the secondary analysis are the Burrows-Wheeler Alignment (BWA) and the Genome Analysis Toolkit (GATK) [2]. During the first stage, BWA-MEM performs the alignment and mapping, whereas GATK is (a posteriori) in charge of the identification of the relevant genomic variants.

BWA is a software tool that can align short sequences (queries) against the reference genome [8]. Three BWA algorithms are available: BWA- backtrack (for short reads), BWA-Smith Waterman (BWA-SW) and BWA-MEM. BWA-MEM is the most recent and faster algorithm. It is recommended for short reads and it is more accurate in the detection of variants (such as insertions and deletions).

GATK is a software package created by the Broad Institute which is used to analyse high-throughput sequencing data. GATK is mainly focussed on the discovery of variants and it is divided in three stages. Over the first stage, the duplicate marking, all inputs and outputs are analysed to identify any duplicated sequence that forms artifactually. The second stage filters all found duplicates to identify the ones that suppose a clinical relevance. The last stage, variant calling, lists the filtered duplicates and determines the likelihood of genomic variants. As shown in 2.3, the mapping phase (BWA-Mem), is the stage with major computational costs, which is the reason why the current thesis is based on mapping. Further details and more pipeline analysis for GATK and Freebayes are detailed in [12].

| Tools (secondary analysis | Single-thread CPU (hours) | 36 thread CPU (hours) |
|---|---|---|
| BWA Mem | 92:03 | 3:50 |
| Picard SortSam | 7:55 | 6:36 |
| Picard MarkDuplicates | 6:34 | 5:45 |
| GATK RealignerTarget | 5:58 | 0:18 |
| GATK IndelRealigner | 7:24 | 3:52 |
| GATK BaseRecalibrator | 19:49 | 1:56 |
| GATK PrintReads | 23:54 | 7:28 |
| GATK HaplotypeCaller | 63:39 | 6:18 |
| TOTAL EXECUTION TIME | 227:19 | 36:07 |

**Figure 2.3:** GATK pipeline execution times by stage [2]

Finally, tertiary analysis involves the interpretation of the data to assess the origin of the variants and the functionality of each sequence. In this stage, the lab data, biological data and clinical data are combined to determine the relevance of the findings into disease aethiology and disease prevention.

### 2.2.3 Seed and extend

One of the problematics that strongly enhanced genomic analysis was the strong need to identify new genes in genomes. This problem raised the interest of different interdisciplinary scientific areas which are closely related: informatics, applied maths, statistics, computational science, artificial intelligence, chemistry and biochemistry; and that required the use and development of different techniques.

To date, the methods of identification and analysis of sequences (through biological and chemical processes, called assembly methods) provide better results, but they become impractical when applied to large genomic sizes given the long processing times and costs. This was one of the main reasons for which alternative analytic techniques were sought, reference mapping and alignment methods.

From the reference methods, one of the most widely used is the seed and extend strategy. The approach is based on the assumption that two highly matching sequences should contain shorter substrings (called seeds) that are exactly or almost exactly matching too. The technique is split in two steps (seed and extend, see Figure 2.4). The seed phase consists in finding the exact location of the matching substrings, while the extension phase aligns the read to the region of the found substrings. Aligners such as Novoalign [13], BWA-MEM [8], Bowtie2 [14] and Cushaw2 [15] use seed and extend strategies.

BWA is one of the most popular software applied to the WGA problem [8]. BWA is an algorithm that can align short sequences against the reference genome using BWT and FM index for mapping. The algorithm was created by MIT in conjunction with the Broad Institute. Three BWA algorithms are available: BWA- backtrack (for short reads), BWA-Smith Waterman (BWA-SW) and BWA-MEM. BWA-MEM is the most recent and faster algorithm. It is recommended for short reads like the ones from Illumina, and it is highly accurate in the detection of variants (such as insertions and deletions). One of its key features is that BWA-MEM forms a BWT index from the combination of the forward and reverse DNA strands.

The current section is designed to be an introductory guide for the following chapters, and therefore provides basic data regarding the seed and extend technique.

#### Seeding phase: Indexing techniques

During the seeding phase, an off-line index of the reference genome is created, and therefore the seeding phase is also referred as the indexation phase. Indexing is a method to accelerate the search of patterns within a sequence or string. The search can be performed in an exact

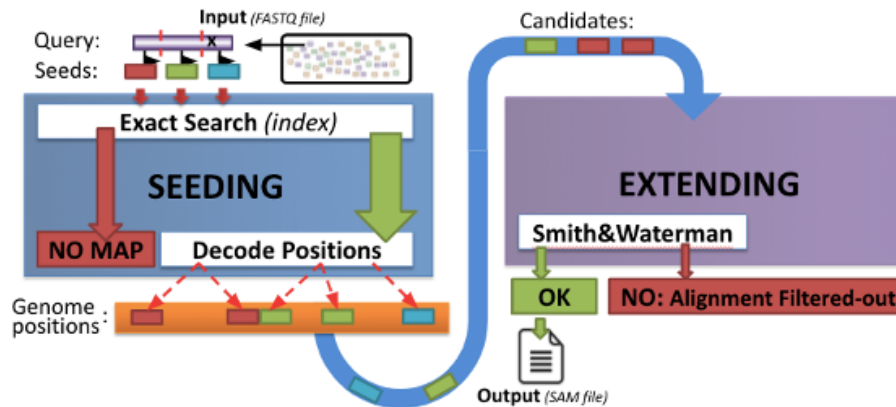


**Figure 2.4:** Scheme of the main steps involved in the seed and extend strategy

or approximate manner. Although a considerable amount of time is invested for the index creation, this will be compensated by the speed-up of the searching stage, given that a large number of searches is needed.

### Exact matching and approximate matching

When we think of a string matching, this can be defined on the following way: let $R[0..n-1]$ be a reference text or string with n symbols over an alphabet $S$, where $R[i]$ is the $i^{th}$ symbol of the string, $R[i \ldots j]$ is a substring of $R$, $R_i$ is a suffix of $R$ starting at position $i$, and $|R| = n$ represents the length of $R$. Let $Q[0..m-1]$ denote a query pattern or string with $m \times n$.

Considering this, the exact string matching refers to the finding of all the occurrences of Q into R. Exact pattern search over a large reference string is accelerated by using different types of indexing data structures, such as *hash tables, suffix-trees, suffix-tries, suffix-arrays or FM-indexes*.

- Suffix trees are data structures that contain all the suffixes of a string within a tree. This allows space-efficient data storage.
- A Suffix Array is an array of all the suffixes of a string T which had been lexicographically organised [16]. Therefore, it contains all the values of the leaves of a suffix tree in order, however it does not have the tree structure. Because no tree structure is needed, suffix arrays have the advantage of using less space than suffix trees.

- FM-index is one of the cornerstones of the current thesis and is one of the most used methods in current sequencing. FM-index was originally proposed by Ferrangina and Manzini [17], and it is based on the Burrows-Wheeler transform (BWT [18]) as well as in the suffix array structure. One of the advantages of the FM-index is that the indexed data is compressed and that this space reduction does not slow the performance of the indexation phase.

**Extend phase: pairwise alignment algorithms**

During the extend phase, pairwise alignment is used to identify similar regions between two sequences. In pairwise alignment, two sequences (A and B) are compared to find the best scored alignment. Different pairwise algorithms are available with different memory and compute complexity trade-offs.

Pairwise algorithms can be classified as local or global, and optimal or heuristic. The goal of local sequence algorithms is to find highly similar regions between two sequences. In contrast, the goal of global sequence alignment is to find the best overall alignment of both sequences.

Smith-Waterman's algorithm is one of the most broadly used local sequence alignment algorithms [19]. The algorithm searches for all possible alignments and finds the optimal local alignment by reading in a scoring matrix, the matrix is formed by the values of every possible nucleotide match.

Bit Parallel Myers' approach improves the Levensthein distance computation by creating an algorithm to compute the DP matrix using bitwise operations. In this way, several matrix cells are handled simultaneously, so the total computational work and the memory storage requirements are reduced.

KMER Filtering is a method based on counting k-mers (which are subsequences of length k that are contained within a biological sequence). The goal is to identify overlapping k-mers and to assemble them to obtain a similarity score between sequences corresponding to associated regions.

Other well-known pairwise algorithms are O(nd) and BitPal, although more in-depth details regarding pairwise algorithms will be exposed over further chapters of this thesis.

### 2.2.4 Short Read Mappers: definitions and characteristics

Short read mapping is the problematic of aligning short read within a targeted reference genome. Short Read Mappers are complex algorithms to solve string-matching problems.

The majority of current mappers will be using approximate string-matching, in which the goal is to optimize the matching speed and accuracy, although still allowing some degree of inaccuracy. Thus, an efficient mapper requires an indexing phase with very few false positives, and a seeding phase with an efficient processing of the reported regions.

A large amount of short read mappers has been developed and published. The majority of them will be based on the BWT [18] (such as BWA [8], HPG-align [20], Bowtie2 [14], nvBowtie [21], Soap3-dp-GPU [22] or CUSHAW [23]) or in hash tables (such as SNAP [24] or Novoalign [13]).

## 2.3 GEM3 Mapper

The following section of this chapter will be focused on detailing the general characteristics of GEM3, the mapper in which the present thesis is based. A description of the GEM3 stages is also given, to provide the reader with an introduction for better understanding of the GPU work that has been performed and that will be detailed in the next section.

### 2.3.1 GEM3 general features

GEM mapper is a highly sensitive mapper that can perform complete search results [25]. GEM uses a mapping model based on filtering, and warranties complete search results for the mappings (see Figure 2.5). One of the main characteristics of GEM is the scalability on processing reads. The mapper also performs faster than BWA-MEM, achieving a total time of 162 minutes ( 2.7 hours) for WES and 3,458 minutes ( 57 hours) for WGS [2], [12].

A GPU version of GEM3 has been created as a part of this thesis, which will be introduced in the next section of this chapter.

### 2.3.2 GEM3 stages

#### Adaptative Exact Search (CPU/GPU)

This step searches for candidates' regions from a reference genome that could map correctly to the read. This process is based on greedy heuristic algorithms to extract non-overlapped

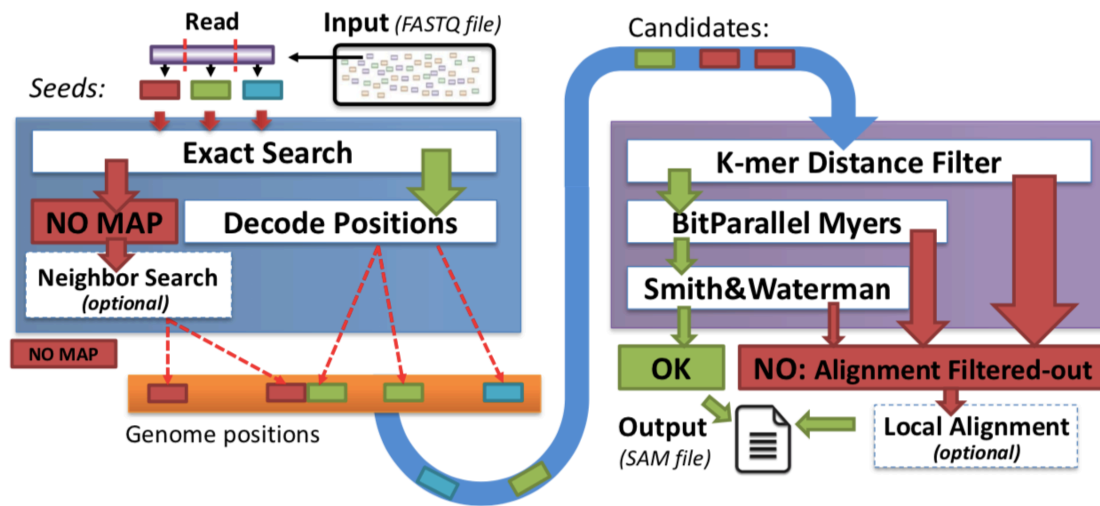


**Figure 2.5:** Internal workflow and algorithmic stages for GEM3

substrings from the read that match exactly on the genome. The core of this phase allows to extract the maximum number of substrings per read without overpassing a certain work threshold to not compromise the computational cost. All the core primitives could be revisited on chapters 4 and 5. Insightful explanations could be found in 9 for the GPU implementation.

#### Decoding Candidate Occurrences (CPU/GPU)

This step turns the candidate occurrences (reported by the first step) from index domain representation into reference domain positions. All the core primitives could be revisited on chapters 4 and 5.

#### Pre-filtering of candidates (CPU/GPU)

This stage is a pre-filtering that prone the highly diverging candidates reported by the adaptative search process from the candidate list. Thus, only the sensitive reads with a certain error are reported to the next stage.

#### Filtering of candidates (CPU/GPU)

A bitparallel approximate string comparison method is applied in this step. Edit distance events are reported and a final score classifies the candidate position due to its homology with the region. The core algorithm of this step is the Bit-Parallel Myers algorithm which can

exploit higher throughput computer vector resources from GPUs. This mapping is described as pseudo-alignments and can be used as a final output result. Depending on the requirements for the user pipeline in the data analysis process.

#### Global alignment (CPU/GPU)

In order to report a cigar string for each mapping found, this stage processes a dynamic programming algorithm that performs a global alignment between the read and the genome region reported. A GPU version has been created, which will be discussed with further detail in a later stage of the chapter.

#### Neighborhood search (CPU)

This step is focused in the preconditioned FM-index Bidirectional search, which is applied here to search for deeper stratas with cost-effective methods. This method is just applied to the more conflictive reads, which one contain higher sequencing error rates or mutations, being an optional phase of the process.

#### Local realignment (CPU)

This last stage is executed for the more divergent candidate alignments. Local realignment consists in a partial trimming of the ends of the strings for each candidate in order to search for local solutions in the alignments. The main goal of this stage is to report high quality read regions. This stage is using a Smith&Waterman banded version of the algorithm and aligning by extension.

## 2.4 GEM3 contributions to GPU

The present thesis is based on full GPU integration of GEM3 (GEM3-GPU). The mapper reports the same output files for CPU and GPU (diff command equals) and is also one of the first GPU mappers allowing the alignment of very long and variable reads. An additional feature of GEM3-GPU is that it supports GPU architectures since CUDA 5.0 and has been ported to ARM, Power and x86 architectures. GEM3-GPU has been running on production on a genomic sequencing centre (Centro Nacional de Analisis Genomico, CNAG).

Furthermore, the GEM-cutter library was created as a part of this thesis work. This is a bioinformatic GPU library that provides basic block genomic primitives which are highly

optimized for GPUs. GEM-cutter offers an API based on send and receive primitives (message passing), and incorporates a scheduler to balance the work. Furthermore, the library supports all GPU architectures and Multi-GPUs, and manages heterogeneous coupled GPUs.

## 2.5 Conclusions

Over the following years, the aims of genome sequencing field are to reduce the costs of equipment, as well as to increase the accuracy and improve the running times to make the technology broadly-available. Thus, it will be possible to routinely run DNA analysis in the population with a simple blood sample in order to detect early markers of disease. In order to achieve this, the technology must be available for day to day analysis in the majority of clinics and hospitals, and therefore it is necessary for the techniques and protocols to become simpler and easier to reproduce. With the aims of becoming more affordable and available in a routine setting, there is already a trend of the sequencer market towards promoting smaller affordable sequencers (e.g., bench-top), more efficient computational methods and scalable computational systems that can easily fit within a general practice setting.

# 3
# Heterogeneous Computing

*"It's the questions we can't answer that teach us the most. They teach us how to think. If you give a man an answer, all he gains is a little fact. But give him a question and he'll look for his own answers."*

The name of the wind - **Patrick Rothfuss**

The current chapter introduces general GPU concepts. Section 3.1 will introduce the motivations of computing on GPUs, section 3.2 describes GPU hardware and software features. Following subsections 3.3. and 3.4 describe optimizations and techniques for bioinformatic algorithms.

This chapter introduces the necessary GPU concepts for better understanding of the thesis contributions by explanation of the foundational heterogeneous systems (CPU-GPU) programmability. Section 3.1 will introduce the principal motivations of compute capabilities on GPUs, section 3.2 describes the main hardware and software GPU features. Following subsections 3.3. and 3.4 describe fundamental optimisations and techniques explored on this thesis for bioinformatic algorithms and their suitability.

## 3.1 Introduction

As introduced in chapter 1, innovations on emerging architecture designs have developed over the last years due to the unprecedented diminished power, performance and area (PPA) benefit returns using the last advances on silicon integration processes. We are reaching the point in which (1) the computer clock frequencies cannot continue to increase, (2) production yields are heavily impacted and (3) cost integration is increasing. All of these problems did not allow to achieve an efficient use of the growing number of transistors included in a single chip that Moore's Law kept offering. For that reason, processors started to evolve differently, adding a larger number of parallel computational resources that need to be exploited by the software, by modifying algorithms in order to create explicit executions using parallel tasks.

In the last years, the growing demand of multimedia applications resulted in the emergence of specialized hardware solutions with the requirement of covering complex graphics primitives. Over the years, this increasing scenario boosted the emergency of the first programable graphic processors. At that point, although with many limitations, graphic processors started to allow the execution of simple general-purpose algorithms.

Graphic processor designs are strongly influenced by the massive parallelism present in the graphic primitives (rendering, rasterization, physics ...), and therefore general-purpose algorithms can explicitly extract this level of parallelism can strongly benefit from these designs.

The advances in graphic processor programmability introduced an interesting new paradigm: heterogeneous computation. The lack of stand-alone execution features on a GPU has promoted the emergence of a new approach, heterogeneous computation, in which CPU and GPU are combined within the same execution system. This approach leverages CPUs as latency-oriented processors (prioritizing a reduction on the answer time per operation) and GPUs as throughput-oriented processors (prioritizing the increased operations performed per unit of time).

These new changes in the evolution of processors were a rule changer for the software engineers, where they previously needed to wait until the next processor release to see a significant increase in the performance. Currently, programmers understand the process and make an extra effort, designing their tools to take the maximum benefits of these additional resources that new architectures offer. Therefore, the programmer is conditioned to obtain better knowledge of the architecture that is developing to be able to profit from these additional resources. Additional effort must be made to identify the potential parallel sections of the code, with the aim of re-distributing the work in between the computational resources that the processor offers and to obtain a better performance of the application.

Over the past years, the scientific community had a special interest on taking advantage of the high compute density of GPUs. The objective is to achieve an efficient execution of traditional CPU algorithms on GPU devices; this set of techniques is called General Purpose Computing on Graphic Processing Units (GPGPU).

Massive parallel heterogeneous platforms, as Nvidia GPUs, have encouraged the development of SDK toolkits and software stacks for the execution of general purpose code. Compute Unified Device Architecture (CUDA) allows the implementation and execution of GPU accelerated applications, providing features and abstractions for the resource managing and interoperation between CPUs and GPU devices on a heterogeneous environment.

## 3.2 Latency (CPU) and Throughput (GPU) processors

This subsection introduces the differences on the main architectural features of CPUs and GPUs, and explains the characteristics of the applications that each platform is targeting. CPU and GPU architectures have had a very different design evolution due to the different needs that they were required to cover.

The objectives of CPU multicore architectures are to reduce as much as possible the latency in the operations that they are running. In order to achieve low latencies in the execution, they implement user-transparent hardware techniques, such as dynamic out-of-order execution of instructions; speculative execution mechanisms; complex data pattern pre-fetchers; coherency on the full memory hierarchy; branch predictors and register renaming. CPU architectures offer higher clock frequencies and reduced latencies for the execution of instructions and memory accesses, benefitting sequential algorithms.

On the other hand, GPU architectures are oriented to increase the throughput (operations per second) as much as possible. To achieve this, they implement other techniques such as

light context switch between threads (multithreading) with almost free cost, which tolerate higher operation latencies better than CPUs. GPU architectures are characterized by a large memory bandwidth, great computational power and high grade of parallelism at the thread level. This facilitates massively parallel tools with a high workload.

Therefore, each kind of processor has taken different approaches in its architecture design and has different benefits for the applications. Considering the area of chip's occupation, CPU multicore architectures have tens of computational units (cores), a complex and advanced control logic and large internal cache memories. On the other hand, GPU architectures dedicate a large part of their chip's area to introduce thousands of computational resources (cores) in a lock-step execution fashion, which simplifies the control's logic and reduces cache complexity, Figure 3.1.

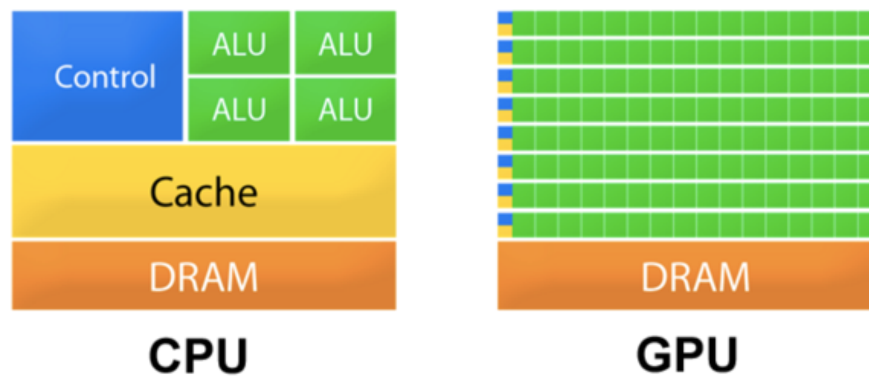


**Figure 3.1:** Differences in resource dedication in the chip of CPU and GPU architectures.

## 3.3 General overview to GPU architecture and its CUDA programming model

Since its release in 2006, CUDA has become the most popular architecture for general-purpose GPU computing. The CUDA programming model defines a computation hierarchy formed by *kernels*, *thread blocks*, *warps*, and *threads*.

GPUs are composed of tens of processing components, called Streaming Multiprocessors (SMs) by Nvidia [26]. SMs share a L2 cache of hundreds of KBytes, and an external global memory of several GBytes. Each SM contains hundreds of SIMD cores that perform in-order execution of instructions. Each SM contains tens of KBytes of local storage that is partitioned

into explicitly-managed registers and shared memory banks, and several implicitly-managed cache memories.

The unit of work sent from the CPU to the GPU is called a kernel. The CPU can launch several kernels for parallel execution. Tens of thousands of threads must be launched simultaneously to achieve high performance. The CUDA programming model is based on a hierarchy of threads executing the same program on different sets of data. A *thread-block* is a group of threads that may cooperate using the registers and shared memory available in a given SM. Thread-blocks in a *grid* (or *kernel*) are scheduled non-deterministically for independent MIMD execution into SMs. A thread-block is divided into batches of 32 threads, called *warps*, which are the smallest scheduled unit. Between 32 and 64 warps from one or multiple thread-blocks are dynamically scheduled for execution in the same SM. This mechanism, often known as H/W multithreading, is the main latency-hiding strategy on GPUs.

A warp is executed in a SIMD/vector fashion; threads in a warp are executed in a lock-step manner operating on 32 values in parallel. If threads in the same warp need to follow different control flows, all paths must be executed one after another, with some threads active and the remaining threads stalled.

The thread block contains multiple warps that are executed independently. The warp instructions from multiple blocks are scheduled for execution on a vector processing unit called streaming multiprocessor (SM). The excess of parallelism expressed in terms of more warp instructions than the available computation resources helps alleviating the operation latencies.

The GPU memory is basically organised as three logical spaces: global, shared and local. The global memory is shared by all threads in a kernel and has a capacity of several GBs. It is located in the GDRAM of the GPU and the reuse of accessed data is exploited via on-chip cache memory. The shared memory is accessible by all warps belonging to the same block, while the local memory is private to each thread and mapped to a set of registers. Registers have the highest bandwidth and lowest latency. The shared memory is slower than the registers, whereas the GDRAM has very high access latency and limited bandwidth.

An instruction executed by a subset of the threads in a warp is said to be divergent. Divergence is an inherent performance limitation of SIMD architectures, and must be addressed when designing the algorithm. Control flow divergence among the threads in a warp causes the sequential execution of the divergent paths, and hence it must be avoided.

Another critical performance issue is the memory access pattern of the program. When

executing a SIMD/vector load or store instruction, the memory addresses provided by all the threads in the same warp are combined, or *coalesced*, to generate one or multiple memory block access requests (memory blocks of 32 to 128 Bytes). High memory performance is achieved only when all the data requested from global memory is used by the program. In practice, that means requested data is coalesced into one or a few memory blocks.

### 3.3.1 Nvidia GPU architectures and its hierarchies

Nvidia GPU processors present a vectorial architecture, in which different resources are doing the same operation simultaneously over different datasets. GPU architecture consists of several SMs (Stream Multiprocessors) that are interconnected by a shared bus and share a data cache and a global GDDR5 or 3D stack HBM2 memory (Figure 3.2).

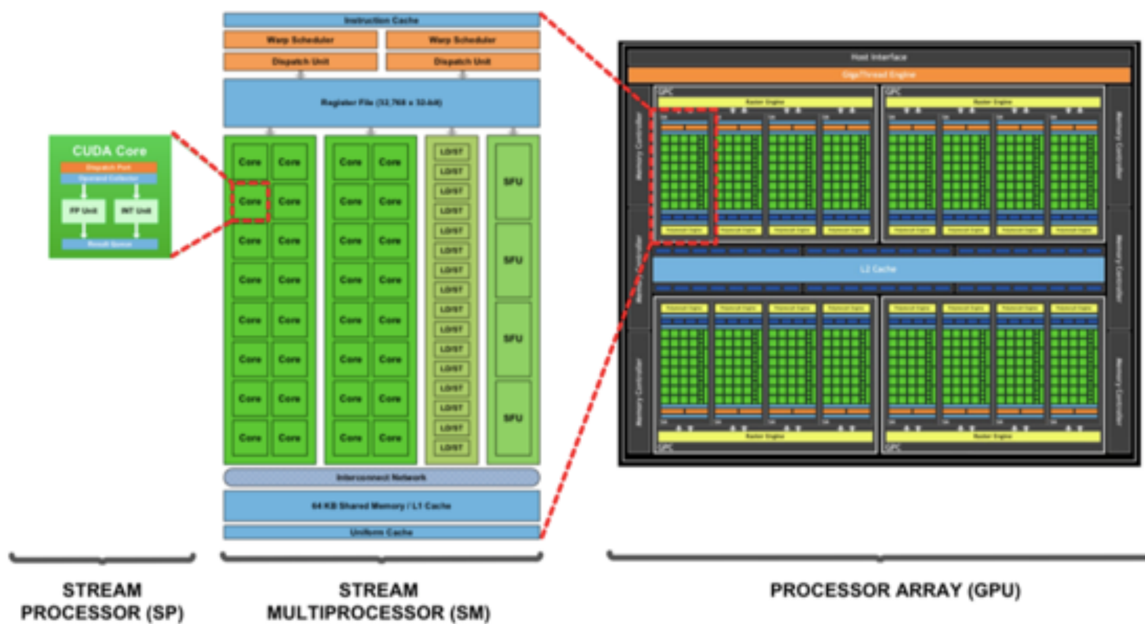


**Figure 3.2:** General scheme of the elements of a typical GPU. Computation and memory elements are hierarchically organised.

The SMs are grouped into TPCs (Thread Processing Clusters), sharing different elements to make the architecture scalable (e.g., warp and block schedulers, texture units). Within the SM of a TPC, the L2 cache is shared for constant memory, and a L1 cache is shared for textures.

More in detail, each SM has several computational and memory units. In later architecture

generations the SM is divided into four separate processing blocks (referred to as SMPs), where each SMP contains its own instruction buffer, scheduler, and computational resources:

- 64 or 128 SPs (Stream Processors) divided in groups of 32 cores, called lanes. Each group of 32 SPs must execute simultaneously the same instruction, and therefore working together over different data. SPs can perform integer operations, floating-point operations or bitwise logical.
- 8 SFUS (Special Function Units). These units are reserved to perform special arithmetic calculations, called transcendental, such as square roots, sinus, etc. This resource also has a vectorial structure, and all eight Units must execute the same operation simultaneously.
- 16 LD/ST (Loads/ Stores). They manage the memory instructions. Similar to the previous SM resources, they must execute the same instruction simultaneously.
- A 192 KB shared memory per SM, which can be set as on different configurations e.g., 16 KB / 48 KB or 48 KB / 16 KB to act as a L1 data cache or as a scratch-pad memory.
- Every generation is including more new specialised hardware blocks on the SM targeting new application domains, as the case for Tensor cores (AI), Raytracing cores (Graphics) and FP64 support (HPC).

The amount of SMs in the GPU depends on the specific device. For example, GeForce GTX 3080 uses an Ampere architecture that has 8704 CUDA cores (68 SMs x 128 SPs) and a 384-bit memory bus with a bandwidth of 760 GB/s. The size of the global memory of this model is 11 GBytes, but there are HPC professional devices with more than 80 GBytes of global memory.

### 3.3.2 CUDA GPU execution model

Each execution phase of an algorithm has different behaviours (serial or parallel). The ideal implementation of an algorithm in a GPGPU system consists of identifying these phases and executing each one either in the CPU or GPU, with the aim of improving the performance. Figure 3.3 shows an example of this model: a program presents two serial and two parallel phases, and each of them is executed in a different CPU or GPU environment.

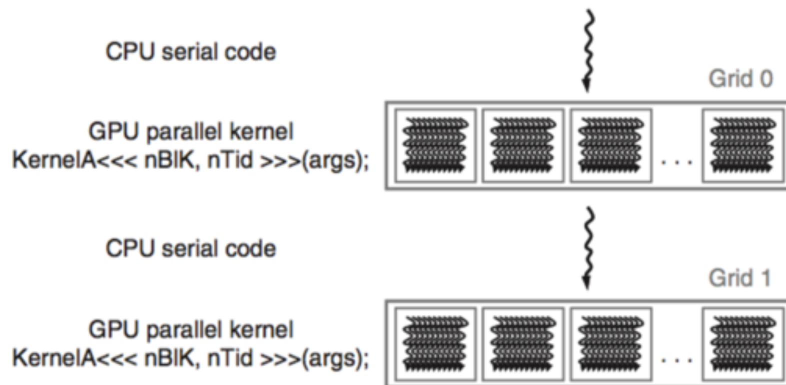


**Figure 3.3:** Computation in a heterogeneous GPGPU system alternates the execution of serial an massively parallel phases.

Currently, an Nvidia GPU needs support from a typical CPU, also called host, to perform some of the tasks required for the execution of a program. This is because the GPU does not have direct support to access the main system memory or the system's input and output devices (e.g., hard drives or network). GPU-CPU communication is generally performed via a PCI Express port.

The part of the program executed in the GPU is called a kernel. In order to launch a kernel in the GPU it is necessary to perform the steps depicted in figure 3.4, and summarised as follows:

1. The code starts executing in the CPU, declaring, allocating and initializing all the data structures required on the host side.

2. The CPU reserves space in the GPU memory for the input and output data that will be used on the device side. Then the input data is transferred from the CPU main memory to the GPU memory.

3. The CPU reserves the GPU memory space for the entry data and the results.

4. The CPU launches the execution of the kernel in the GPU, configuring the necessary execution parameters.

5. Once the GPU execution is finished, the resulting data are copied from the GPU memory to the main memory.

6. The CPU frees the reserved memory on the GPU memory space.

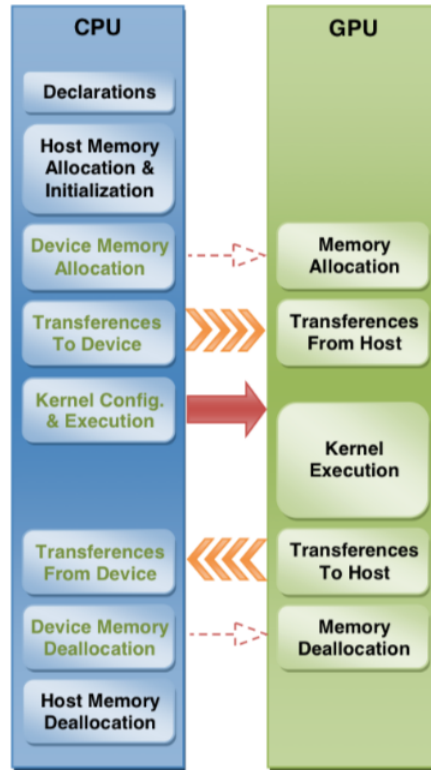


**Figure 3.4:** Typical steps for invoking the execution of a computation kernel in a GPU.

The typical amount of main memory available on a GPU is several times less than the amount normally available on a CPU. In case the amount of data required to be processed in a GPU is greater that the size of the GPU memory then the parallel algorithm must be adapted to perform the data processing in different iterations.

### 3.3.3 CUDA Parallelism model at thread level

NVIDIA GPUs implement a hierarchical parallel model that allows scaling an algorithm to millions of threads without causing an increased cost in the execution. In addition, this model allows a better abstraction of the program parallelism over the hardware parallelism. This characteristic provides performance portability of the code between different GPU devices.

The parallel model used by CUDA is based on a hierarchical structure of threads, Figure 3.5. Each thread group defines the available thread cooperation mechanisms and the access to

a common memory space.

Threads into different blocks are completely independent in between them, because the GPU does not ensure a specific order for the block execution, and does not allow barrier synchronization in between blocks. All blocks must have the same size (same number of threads).

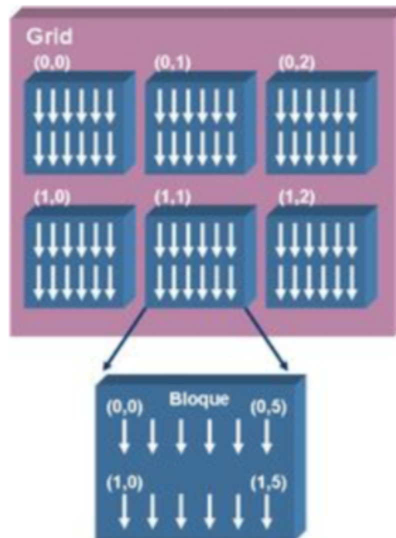


**Figure 3.5:** CUDA's thread hierarchy. All the threads that form a kernel are grouped in a set called grid. Threads within a grid are grouped in subsets called blocks. All blocks contain the same number of threads.

Each block in the grid is identified by a unique number (block id), and within a block, each thread is also identified by a unique number (thread id). Therefore, an unequivocal identification is available for each thread. This pair of numbers allows the programmer to divide the compute between each thread, typically associating a different small data set to each thread.

### 3.3.4 GPU Memory hierarchy

Using memory appropriately is critical to obtain high performance in GPUs, where dozens of thousands of threads are being executed at the same time. Although GPUs are designed to hide memory latencies, global memory bandwidth remains a potential bottleneck since it is very likely to have thousands of threads making simultaneous requests to the global memory. GPUs implement a wide set of different memories that help reducing the pressure on the GPU

main memory. Each memory has specific characteristics that benefit different access patterns: some knowledge of how these memories behave is required to execute programs efficiently.

Different types of memory resources are present in CUDA: local memory, constant memory, shared memory, global memory and texture memory. The following table (Figure 3.6) shows the range, visibility, location and live time that each stored variable has in each memory:

| Memory | Location | GPU Access | Visibility | Execution scope |
|---|---|---|---|---|
| **Registers** | On chip (SM) | R/W | Thread | Kernel (thread) |
| **Local** | Off chip (GDDR) | R/W | Thread | Kernel (thread) |
| **Shared** | On chip (SM) | R/W | Block | Kernel (thread) |
| **Global** | Off chip (GDDR) | R/W | CPU & GPU | Program |
| **Constant** | On chip (TPC) | R | CPU & GPU | Program |
| **Textures** | On chip (TPC) | R | CPU & GPU | Program |

**Figure 3.6:** Range, visibility, location and execution scope of each memory

Registers are the faster memory of the GPU. They are intended to store intermediate data of the calculations and data that is used very frequently.

The local memory allows to store private data of each thread, sharing the same visibility and lifetime of registers. It is useful when there are no registers available in the SM, so that the private data of each thread is stored in the global memory. This scenario is called register spilling.

The global memory is a memory located outside the chip, which conceptually is similar to the main memory of the CPU. It has a large size, but the access and bandwidth are smaller than in the other memories. The newer GPUs includes two levels of cache hierarchy that benefits from temporal and spatial locality on the accesses to global memory.

The shared memory, which is located within the chip, is a very valuable GPU resource. This memory allows to distribute the data among threads of the same block, allowing to perform efficient thread cooperation. It is a scratchpad memory that is explicitly managed by the programmer, which indicates the transactions between the variables allocated in the global memory and the variables allocated in the shared memory.

The texture memory has been historically used to read image textures in graphics. This type of memory tries to substitute the functionality of the constant cache memory. It is designed to optimise the access to local data in 2 dimensions. Given the high complexity (Figure 3.7) in programming, developers try to avoid the use of the texture memory.

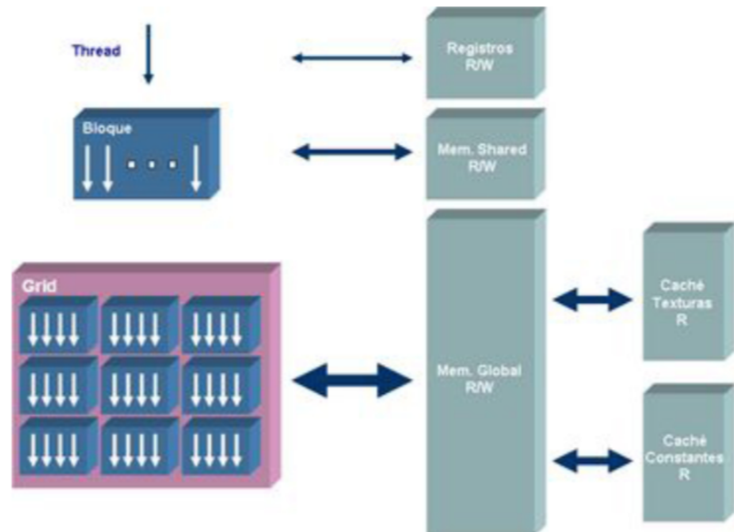


**Figure 3.7:** Diagram relating the thread hierarchy with the data visibility in memory. A register can only be accessed by a thread; the shared memory can only be accessed by the threads of the same block; and the global, texture and constant memories can be accessed by any thread of the grid.

The constant memory can be seen as a small memory inside the chip that works as a cache and has a quick access. In this memory (as in the texture memory) data can only be read, it is written by the CPU before the initiation of the kernel, and it is visible during all the program's execution.

### 3.3.5 General best practices for performance

Next, we present several guidelines to help developers obtain the best performance from CUDA GPUs. We describe parallelization and optimization techniques and explain coding metaphors and idioms that can greatly simplify and improve programming for GPU architectures.

#### Coalesced access to the memory

The correct usage of the GPU memories is essential. The large number of active threads implies that the number of simultaneous memory requests can be very high. Therefore, although GPUs allow hiding memory latencies and have a large bandwidth when compared to MultiCore CPUs, memory access can easily become a performance bottleneck.

GPUs try to combine the separate memory requests made by each thread in a group in order to minimize the high number of requests. A memory access that combines several independent accesses is called coalesced. To allow the combination, the requested data needs to be adjacent, as shown in figure 3.8. A coalesced access unifies up to 32 requests into a single one, if the resolved addresses are in the same memory block.

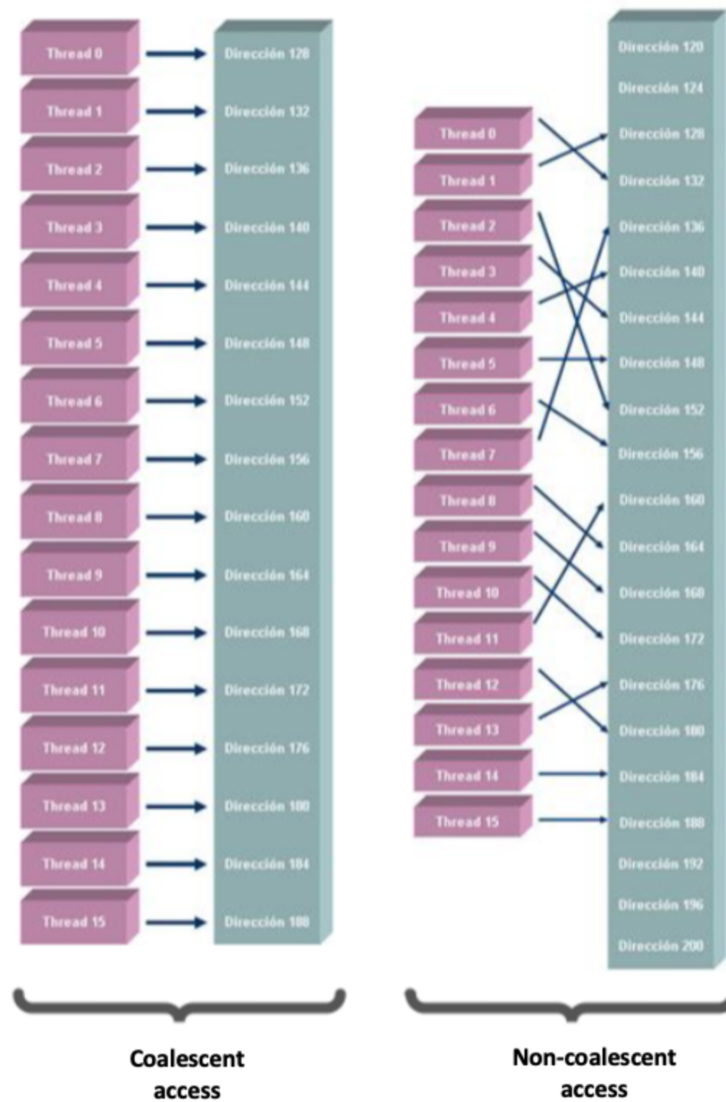


**Figure 3.8:** Coalesced and non-coalesced accesses.

### Local and shared memory

Shared memories have a great importance when talking about GPU efficiency. They allow to use the temporal and special location of the data and to reduce the global memory requests.

This is a scratchpad memory, meaning that the programmer must explicitly transfer data. The main advantage of these memories is the reduced latency and high bandwidth. They also allow performing completely random access without an additional cost.

The process runs as follows: the programmer fetches data into the shared memory, performs the computation on the data and finally transfers the results into the global memory. All accesses on the shared memory during the computing have been avoided in the global memory, and therefore reserving the bandwidth to other uses.

### Warps and thread cooperation

Within a block, 32 adjacent threads are grouped into a set called warp. A warp of threads is an abstraction that reflects the internal operation of the hardware (Figure 3.9), which actually executes SIMD instructions with 32 lanes. The GPU programming model conceals SIMD operations by exposing each physical thread as a number of 32 logical threads, the SIMD width.

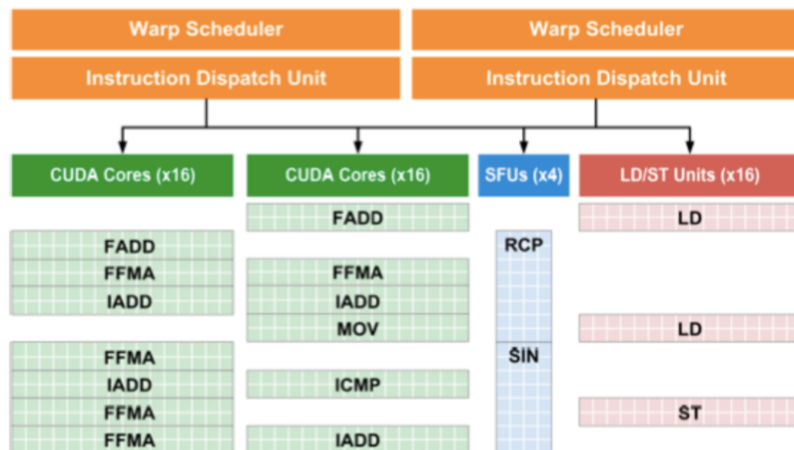


**Figure 3.9:** SM internal architecture showing the grouping of threads into warps.

Knowing how warps operate can help planning the program's parallelism differently. It can help to put into practice techniques of thread cooperation such as cooperative memory access.

### SIMT execution and thread divergence

Given that a warp must execute the same instruction for all its threads, when executing a conditional structure, it is possible that each thread takes different execution paths. This creates a reduction in the performance efficiency because the execution of the paths becomes serial.

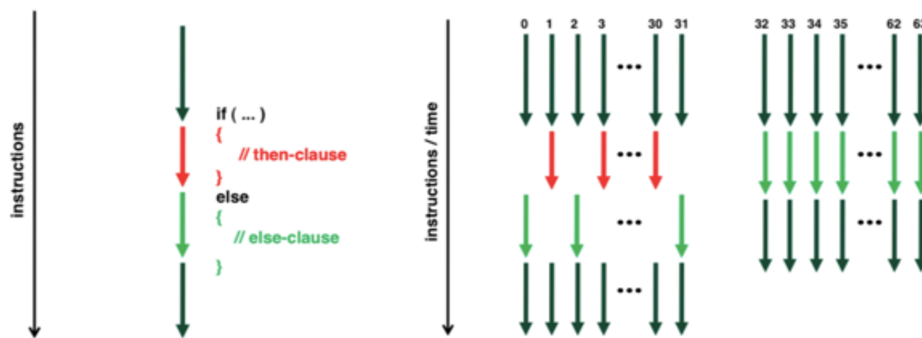


**Figure 3.10:** SIMT architecture: threads in the same warp may divergence due to different conditions.

Figure 3.10 presents an example of a situation where the conditional structure of the if within the warp of the threads 0 -31 takes two different execution paths. In this case, all threads must traverse both execution paths, but only some of them are active during some portions of the execution. In the case of the warp with threads 32 – 63, there is no divergence because all the threads take the same path.

### 3.3.6 Different CUDA platforms as HPC and Embedded

The performance portability of CUDA allows to run our GPU code on different platforms from HPC accelerator GPUs to low power embedded GPU devices, and then check if performance actually scales. Over chapter 10, a more detailed explanation is provided, and detailed results are also presented.

## 3.4 GPU Challenges on genomic algorithms

The current section is a compendium of the main performance challenges when programming bioinformatic applications for GPU execution, and the potential optimization strategies. They will be discussed in more detail on future chapters.

A fundamental characteristic of bioinformatic applications is the existence of great amount of parallelism. This chapter introduces the most important lessons from the thesis to leverage the large computational and memory bandwidth resources from GPUs.

### 3.4.1 Why the task-parallel approach fails on genomic applications

Searching billions of short DNA strings in a large genomic reference is a problem that can be solved by resorting to the simplest parallel programming pattern, the `map` pattern [27]: an elemental function is applied in parallel to all the elements of the input set, usually producing an output set with the same shape as the input. A straightforward `map` GPU implementation would make each thread read its input data, perform the elemental function, and generate the output data. While this *task-parallel* approach is very effective on multicore CPUs, it can be problematic on GPUs due to some of their exclusive architecture features:

1. Accesses to global memory must be *coalesced* to achieve high efficiency. Coalesced accesses occur when all the threads in a warp address memory positions belonging to the same memory blocks.
2. The ratio of available on-chip memory per executing thread is very small; for example, Nvidia Kepler and Maxwell architectures provide a ratio of just 24-32 and 128 Bytes per thread for the shared memory and register storage, respectively. This SM on-chip memory ratio is a common architectural characteristic which has endured until nowadays, even for last Ampere architectures.

A *working set* of a task is the aggregate active data set that must be kept in memory during the task execution. Due to feature (1), a simple task-parallel approach is inefficient on the GPU when each single task has to access a relatively large amount of input or output data. On the other hand, when the working set of a task becomes large, due to feature (2) one has to face two possible performance problems. If the working set is placed on local registers or the shared memory, the excessive capacity requirements will ultimately reduce the maximum number of threads being executed in parallel (defined as the *GPU occupancy*), thus exposing the latencies of the compute operations. If the working set is placed in global memory, then the on-chip L2 cache will probably be overflown, and a higher GDRAM traffic will be generated. While the latter effect is similar to what happens on the CPU, its relevance on the GPU is much bigger due to the larger number of threads involved.

### 3.4.2 Random memory accesses on genomic applications

Many GPU applications present random memory accesses, such as their index data structures used to efficiently perform exact or approximate string text searches on large reference texts. Traditional algorithms are widely find on the applications briefly described in the previous Chapter 2, e.g., FM-index, Suffix arrays, Hash tables, and Suffix trees.

GPUs provide very high memory access bandwidth, in the order of hundreds of GB/s, for sequential coalesced accesses along relatively large contiguous portions of memory. However, performance suffers very much when a program accesses relatively small data blocks located at random memory addresses (Figure 3.11). Unfortunately, typical string search algorithms (as the well-known FM-index in mappers) happen to show a pseudo-random hash table-like memory access pattern [28]. In fact, the FM-index search can be described as a loop that successively (1) loads a memory block from a given memory address, and then (2) calculates the address of the next needed memory block using the data just read [17]. The generated set of memory addresses is fairly unpredictable, and uniformly distributed along the whole memory footprint.

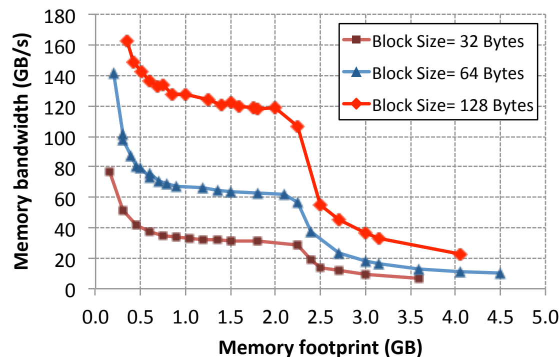


**Figure 3.11:** Memory bandwidth for random accesses on the Titan GPU (6GB GDRAM)

Figure 3.11 depicts the peak memory bandwidth achieved by our best GPU FM-index implementations (with coalesced accesses) on our test machine, for different block sizes and index sizes (memory footprint). Two main results can be read from the plot:

- *Large blocks are free*: accessing small blocks (32 Bytes) at random positions achieves suboptimal bandwidth; one can read larger blocks at the same cost without saturating the memory system.
- *Memory footprint size matters*: performance drops heavily as accesses are scattered along a larger memory region; two clear inflection points exist for memory foot-

prints of 0.5 GB and 2.5 GB. In particular, the 2.5 GB threshold cannot be easily explained by any architectural feature documented by the manufacturer, albeit it seems to appear on several GPUs (see section 6.6.7). A plausible explanation for this behaviour might be the undisclosed existence of TLBs on the GPU, which has been put forth for instance in [29]. In fact, recently published reverse engineering work from Citadel [30], describes the presence of a TLB system on the last Volta architecture and a characterization of number of entries and page sizes of the TLB.

### 3.4.3 Irregular work on bioinformatic applications

The common nature of bioinformatics applications is that (1) the amount of work and (2) the internal pipeline workflow depends on the contents of the genomic data analysed; this data dependency raises different irregularities that heavily affect the overall performance of the GPU application.

- **Batching and parallelism extraction:** Traditionally, bioinformatics applications are developed with a CPU execution workflow in mind. A typical design of the internal workflow processes one query at a time and leverages several work cut strategies, which indirectly introduces many data dependencies on the processing pipeline. This design limits the potential parallelism of the application and introduces challenges at GPU efficiency level. The full application should be redesigned to work on query batches, several queries at a time. In this new scenario, advanced schedulers and work balance engines are critical to extract the level of parallelism that is necessary for GPUs. More details of these techniques can be found on chapter 9.
- **Work regularization and specialization:** A batch-based processing pipeline requires regularization strategies to maximize GPU utilization. Fine grain parallelisation schemes described on chapters 6 and 8 greatly facilitates the reduction of execution inefficiencies, due to the same task is processed by a larger number of threads that cooperate between them on the same problem. Chapter 9 explains different techniques, at the batch level, to decompose tasks into smaller and more regular jobs.
- **Memory constraints and penalizations:** Due to the historical limitations of the size of the GPU main memory and internal memory compared with the memory of

a CPU, different algorithmic approaches have to be implemented as well as data structures. For example, a different data layout may be needed for different GPU architectures in order to exploit full memory coalescing and improved data locality. Other data structures are customised for the most common cases, as for example FM-index from chapter 6, reducing their features support on the queries to the most common cases, high compression ratios or more compacted memory footprint to fit better on the GPU characteristics. These optimizations and other similar examples are explained in more detail in the chapter 9.

### 3.4.4 Host to device transferences

Efficient memory transfers are critical for CPU-GPU communication. Figure 3.12 shows how the size of the data transfers affects the efficiency of the communications between host and device. When designing an application and deciding their data partitioning and job scheduling, device transferences must be considered: the maximum efficiency is achieved when the size of the transferred block is around 1MByte.

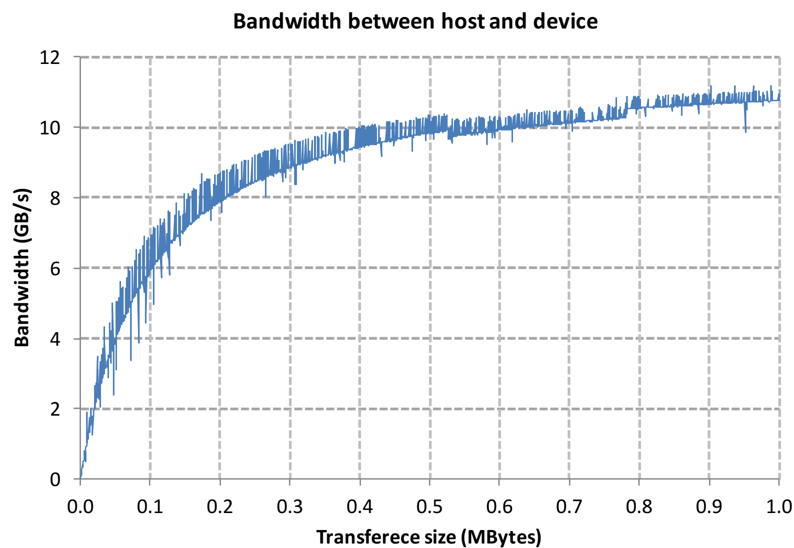


**Figure 3.12:** PCI-e interconnection bandwidth for different transfer sizes

## 3.5 Optimising Genomic algorithms on GPU

This section will describe generic algorithmic strategies to improve the performance of bioinformatic applications on GPU systems. The subsections introduce the following concepts: (1) different intra- and inter task parallel schemes and their analysis, (2) identified GPU limitations on genomic algorithms, (3) improving the performance by the use of alternative parallel schemes and (4) more advanced optimizations.

### 3.5.1 Exploiting inter- and intra-task parallelism

In the previous section we anticipated some GPU performance issues present in inter-task parallelisations, and the relationship with the architecture design.

Efficient GPU programming requires the explicit extraction of massive parallelism. Most of the parallelisations of bioinformatics primitive operations found in the literature represent an inter-task parallel approach (multiple tasks in parallel). This straightforward parallelism based on multi-core CPU approaches usually under-utilizes GPU processor resources, and the next points enumerate some of the issues presented when using that approach:

- **Compute-thread divergence:** Algorithms with an irregular amount of work or with divergent execution flow (branchy codes) prevent the full utilization of the vector computational units of the GPUs. Higher performance on GPU is obtained when most of the time all the threads of the same warp execute the same operation at a time
- **Memory-thread divergence:** High efficiency utilization of the hierarchical memory of GPUs requires coordinated memory accesses. Full utilization demands that all threads in a warp access simultaneously the data placed on the same cache line, since the access of different cache lines incurs instruction re-executions. Consecutive data accesses along the threads (coalesced accesses) provide the best memory utilization on GPUs.
- **Large memory footprints:** The small amount of on-chip memory per thread on GPUs exposes two performance problems: (1) cache memory pressure and (2) low multi-threading utilization (thread occupancy). Larger memory footprints exacerbate the next issues:

- **Cache memory pressures:** The opportunity of data reuse on the cache memory system is low due to the aggressive multi-threading, only 128 cache lines can be kept alive on the L1 cache for the 2048 threads running simultaneously. Threads compete for the cache memory space and usually produce a high number of data evicts, preventing benefits of data reuse.
- **Low multithreading utilization:** Avoiding data eviction from the cache memories can be done by explicitly using the internal scratchpad memories. Another alternative is to reduce the number of active threads per GPU multiprocessor to fit the memory footprint of all the threads on these memories. The reduction of thread-level parallelism will penalize performance for algorithms exhibiting dependencies involving large latency operations, since many stalls will appear on the processor pipeline.

### 3.5.2 Rethinking bioinformatic algorithms: intra-task parallelism scheme

The vector-oriented architecture of GPUs enable efficient fine-grain parallelisations using an extremely reduced amount of work per thread. Thread managing operations are significantly more efficient on GPUs than on traditional CPU systems. The lightweight overhead of fine-grain thread creation and the existence of scalable synchronizations opens the door to new techniques exploiting intra-task parallelism. Next we analyse the involved performance trade-offs of this approach.

To reduce the task-parallel inefficiencies previously introduced, threads collaborate in a shared task to generate their output. We call this parallelization approach as 'thread-cooperative'. This strategy allows assigning constant work to each thread and dynamically increasing or decreasing the number of cooperating threads in order to fit with the task size requirements.

- **Increasing the parallelism:** Dividing the task into several parallel subtasks increases the parallelism of the whole application, which is critical to cope with the massive GPU compute resources.
- **Efficiently coalesced memory access:** The internal parallelism obtained for the cooperative parallel part allows to better exploit the spatial locality of the application. Consecutive threads can easily perform coalesced memory requests, accessing consecutive data at the same time.

- **Reduction of computational divergence:** Forcing threads to share the same task reduces thread divergence, ensuring that all the threads will be processing the same amount of work.

- **Reduction of local memory usage:** Assigning a fixed amount of work to each thread can ensure that the memory footprint fits on the on-chip memories, reducing memory eviction or thread under-occupation.

Intra-task parameters (as for example the ideal size granularity) are constrained by architectural design definitions. This architectural dependence limits the maximum internal parallelism to be exploitable, promoting the combination of inter- and intra-parallelism to obtain efficient and thread-scalable approaches. The next section introduces some of them.

### 3.5.3 Advanced parallel schemes: Combining the inter- and intra-task

Intra-task parallel strategies take into consideration the vector-based design of GPUs to enhance performance. Threads associated to the same task are mapped within a subset of the same warp, taking advantage of the step-lock warp features: these threads move forward in the execution synchronously by executing each of the instructions simultaneously, meanwhile, they are sharing the same computational resources.

#### Warp-oriented optimisations

Optimisations dependent on the architecture limit the internal parallelism due to limits on the warp shared resources (threads, registers and local memory per warp). Below we describe efficient warp-oriented optimisations that increase the efficiency of intra-task parallelization. We describe why these techniques prevent to exploit all the internal parallelism and hence to scale the size of the problem. In the end, we describe some proposed techniques to overcome scalability limitations.

- **Avoiding expensive thread synchronizations:** Communications between the threads of the same warp are much cheaper compared to the communications between threads on different warps. Expensive thread synchronization barriers can be avoided by enclosing all the work at the warp level; forcing these thread scheduling strategies limits the number of active cooperative threads available to a single warp (currently 32 threads), reducing the overall parallelism.

- **Using registers as local memory:** GPUs contain a large number of registers (currently 32K per multiprocessor). Unlike CPUs, the largest on-chip storage space on GPUs corresponds to registers. The memory space of L1 and L2 is lower than the register file, in a design that is usually called a reverse memory hierarchy. In addition, the registers are the fastest on-chip memory with the highest bandwidth. Fine-grain parallel designs can exploit the register space to fit the memory footprint, to reduce the number of LD / ST instructions, and in general to improve memory access performance. The disadvantage of using registers is to handle the finite number of registers per multiprocessor, which reduces the potential number of active threads per multi-processor.

- **Register-to-register thread communications:** Latter GPU architectures are improving hardware support for thread collaborative schemes, and more complicated and faster operations between threads are now possible. Exchange register data or vote operations between threads are supported efficiently inside a warp. Reducing the task parallelism to 32 threads allows leveraging these instructions and increase the overall performance of the application, reducing expensive memory-to-memory thread.

### 3.5.4 Bypassing intra-task limitations

Warp-oriented optimization techniques improve the overall performance in spite of reducing the overall thread parallelism, limiting the scalability of the algorithm.

Advanced techniques presented in this thesis will prove that inter- and intra-task parallelism can be used together to efficiently solve competitive large-scale problems. In next Chapter 9, details on the meta-scheduler level techniques and CPU to GPU cooperation will be exposed, showing how to divide and exploit extra parallelism to overcome the cooperative scalability issues on bioinformatics algorithms.

- **Sets of threads inside a warp:** When the intra-task parallelism is lower than the parallelism inside a warp then an underutilization of the warp resources is conducted. To fix these inefficiencies more than one task can be assigned to a single warp, generating sub-cooperative groups per warp.

- **Thread work granularity:** The amount of work per thread plays a crucial role on the scalability issues and also can reduce the work-instructions. The limited

TLP (32 threads) provided by a warp can be compensated with instruction level parallelism (ILP), reaching larger problems. Similar to the traditional loop-unroll optimization, where loop managing instructions are reduced due to the increased work done per iteration, increasing the amount of work per thread can avoid inter-warp communications and instruction overheads both from the thread managing operations. There is an effective trade-off between the increased memory footprint and the reduced number of instructions executed.

- **Thread group binning and Warp specialization:** The irregular amount of work per task results in a divergent number of threads per task. The above explained technique to fully use all the treads in a warp consists on compose threads from different tasks in the same warp. That irregular number of threads per task, usually, brings constrains to perform the thread group composition inside a warp. Groups with the same number of threads are desirable in the same warp due to the opportunity to apply thread managing simplifications and code optimisations. Those techniques require (1) fast and low-complexity binning operations for processing and (2) warp specialisation, where each warp executes code specialised for each thread group configuration instead of launching one kernel per code.

Parameters related to the amount of work per thread can be manually tuned, and the number of threads cooperating per task can be defined dynamically. An interesting future work is trying to dynamically identify at run time the amount of work per task, depending on the configuration of the system GPU architecture. Next chapters will go in deep on details related to these solutions.

## 3.6 Conclusions

At the start of this thesis, mapper designs accelerated by GPUs were very innovative, and only a scant number of applications were available [31]. From an architecture design exploration on genomic accelerator architectures, the present thesis' work is relevant because it provides a deep analysis of the most relevant bioinformatic algorithms and their bottlenecks when deployed to GPU architectures. For that reason, the knowledge generated on this work is useful to define the hardware designs that are more suitable for bioinformatic applications, and for future specific bioinformatics architectures.

This thesis deeply analyses the most commonly used bioinformatic algorithms, and the

generated knowledge can be applied to other applications in the field, benefiting from it. Some examples of GPU mappers that could use this knowledge and contributions are analysed in depth over the chapter 10 (CUSHAW2, nvBowtie, Soap3-dp-GPU).

Moreover, the GEM-cutter library has been generated for this thesis, which can be integrated into other applications and take profit from the contributions of this thesis. This library and the used design to make the GPU usage transparent is explained in detain in the future Chapter 9.

# 4

# FM-index: text indexing building blocks

*"Home is behind, the world ahead, and there are many paths to tread through shadows to the edge of night, until the stars are all alight."*

**J.R.R Tolkien**

The current chapter presents and explains the basic concepts required to define and understand the *FM-index* data structure and its associated operations. We briefly review the state-of-art on the design of indexes for read mapping applications, and then focus on the FM-index, with its excellent computational characteristics. We end with a performance analysis that will motivate and guide the algorithmic optimisations proposed in the following chapters.

Section 4.1 motivates text indexing, defines the problem of exact pattern matching and describes basic syntax notation. Sections from 4.2 to 4.4 introduce the basic structures and primitive operations involved in the task of exact pattern search using the FM-index. Section 4.5 presents practical implementation issues that must be faced on real scenarios. Finally, 4.6 assesses the performance of pattern searching and identifies the performance bottlenecks that will motivate the proposals in the next chapter.

## 4.1 Text indexing and exact string matching

Indexing a reference sequence or string is a method to accelerate pattern search. The time spent on creating the index is conveniently amortised when a large enough number of searches are presented. Total memory capacity requirements to store this index must also be considered. As mentioned in previous chapters, *Ferragina Manzini index* (FM-index) is the preferred indexing method used in most sequence alignment software tools due to its low computation complexity on search operations and its reduced memory footprint [17]. Next, we introduce the fundamental concepts behind the FM-index data structure and operation.

The string mathing basic building block can be defined as, let $R[0\dots n-1]$ be a *reference* text or string with n symbols over an alphabet $\Sigma$, where $R[i]$ is the $i^{th}$ symbol of the string, $R[i\dots j]$ is a substring of $R$, $R_i$ is a suffix of $R$ starting at position $i$, and $|R|$=$n$ represents the length of $R$. Let $Q[0\dots m-1]$ denote a *query* pattern or string, with $m \ll n$. Solving the *exact matching* problem is tantamount to finding all the occurrences of $Q$ into $R$ (i.e. the positions of all substrings of $R$ that are equal to $Q$). Exact pattern search over a large reference string is accelerated by using different types of indexing data structures, like for example, *hash tables*, *suffix-trees*, *suffix-tries*, *suffix-arrays* or *FM-indexes*. The Ferragina Manzini indexing properties introduced in this chapter are rooted in the intrinsic relationship between suffix-trie, suffix-array and FM-index structures. Due to this, we will describe them to help defining and understanding the FM-index design.

## 4.2 Suffix-trie: Forward- and Backward-Search

The *suffix-trie* of a string $R$ is a tree-like data structure storing the sorted suffixes of $R$. The path from the root of the suffix-trie to a leaf node defines a single suffix. Each leaf node represents a single suffix and each internal node determines an *interval* of lexicographically consecutive suffixes.

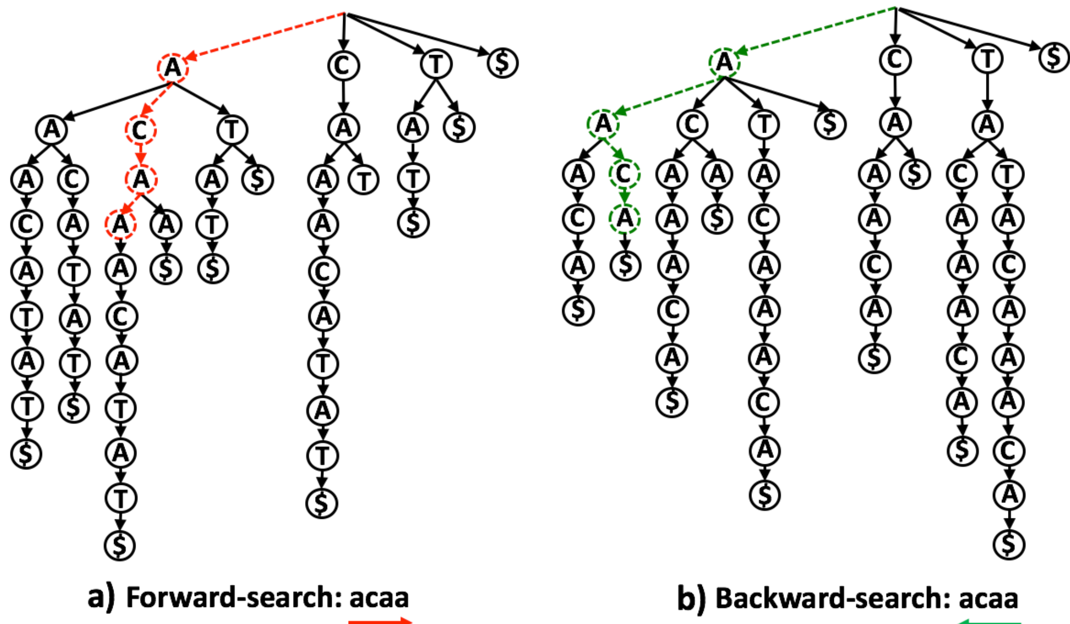


**Figure 4.1:** Forward and backward-search of $Q = acaa$ in $R = acaaacatat$ using the *suffix-trie* of $R$ and rev($R$)

The query search process consists of traversing the suffix-trie, starting from the root, and matching successive symbols of a query $Q$ with the visited nodes. Each symbol $Q[i]$ involves an index look-up and represents a single search step. Each step progressively bounds the interval of matching suffixes. The result is a sub-tree with a single leaf node, which represents a single match.

The query can be *forward*-searched or *backward*-searched [16]. Forward and backward indicate the order in which the symbols of $Q$ are used, from $Q[0]$ to $Q[|Q|-1]$ or vice versa. Figure 4.1 shows an example of suffix-trie that illustrates both a forward- and backward-search involving $|Q|$=4 steps. The result of each search process is a sub-tree with a single leaf node, which represents the only single match that has been found. In a general case, though, the result could be an internal node of the tree representing a set of matches: all the suffixes that are represented by the leaves that are reached from that internal node.

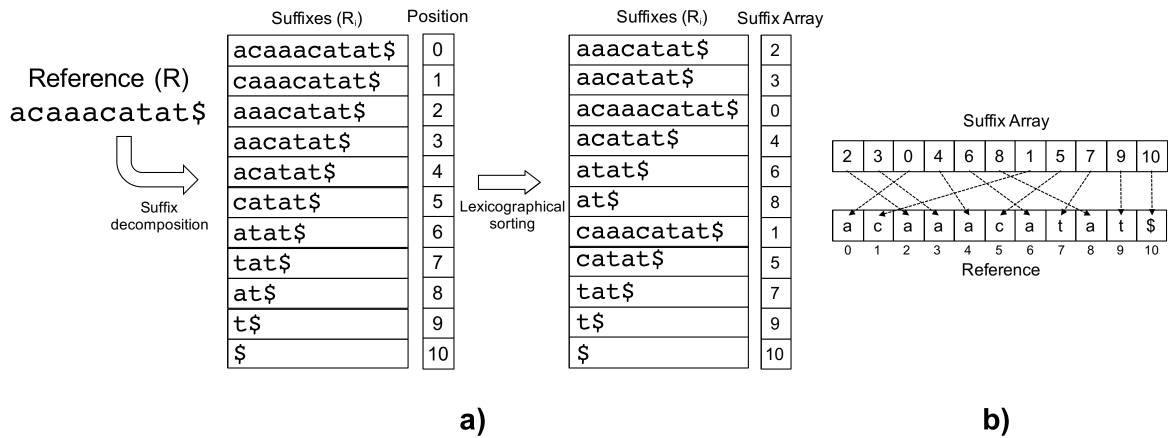


**Figure 4.2:** a) Illustration of the process of constructing the *SA* index for the input string $R = acaaacatat$; b) final *SA* index representation, where pointers/indexes to the original *R* string represent the *R* suffixes.

## 4.3 Suffix-Array and Suffix-Array Intervals

The *suffix-array* structure [32] was proposed as a way to search $Q$ on $R$ using a much smaller memory footprint than suffix-trie, and most importantly, the search process in that index can be efficiently represented using the proposed *SA intervals* and being a basic fundamental concept for the advanced self-index structures as the FM-index described later.

The suffix-array (*SA*) of *R* is the permutation of the sorted suffixes of *R*, where each suffix $R_i$ is represented by its starting position, $i$. To simplify the search operation, the symbol \$, lexicographically higher than all symbols in $\Sigma$, is generally appended at the end of *R* before generating *SA*. Note that the suffix-trie on Figure 4.1 already contained that special symbol as a mark for identifying a leaf node. Figure 4.2.a and 4.2.b show the process of constructing the *SA* index and the resulting representation for our example $R$=*acaaacatat*\$. As shown in the right part of the figure, the sorted suffixes are $R_2$ $R_3$ $R_0$ $R_4$ $R_6$ $R_8$ $R_1$ $R_5$ $R_7$ $R_9$ $R_{10}$. The *SA* vector contains the indexes of the sorted suffixes, $[2,3,0,4,8,6,1,5,7,9,10]$. Note that the SA index of a string can be obtained by enumerating each suffix-trie leaf lexicographically, i.e. from left to right (see the example of trie shown in Figure 4.1).

We define the *SA interval* of a pattern $Q$ as $(l,\ h)$, being $l$ and $h$-1 the ranks of the lexicographically-lowest and highest suffixes of $R[0\ldots n-1]$ that contain $Q$ as a prefix, respectively (the case $l$=$h$ indicates that $Q$ does not occur in $R$). A binary search algorithm can compute the *SA* interval of $Q[0\ldots m-1]$ using $\log n$ steps of complexity $\Theta(m)$, and the $h$-$l$+1 occurrences of $R$ can subsequently be obtained from *SA*. In other words, the solutions found

in the search process can be represented in the *SA* domain, using *SA* coordinates, instead of in the *R* domain, resulting in a very compact representation using just two index values instead of a list with *R* positions. Figure 4.3 illustrates the step-by-step forward-search operation using the *SA* index corresponding to the input string and query considered along this chapter. The result of the exact search operation is the *SA* interval $(2,3)$, *i.e.*, a single solution identified in position 2 of the *SA* and pointing to position 0 in the reference string *R*.

## 4.4 Exact string matching powered by FM-index

The *FM-index* [17] was proposed as a way to traverse an input string *R* represented using the *Burrows-Wheeler Transform* (or BWT) [18] scheme in a tree-like way. It combines the compression advantage of the BWT representation with the low computational complexity of tree-like searching algorithms. This is a remarkable improvement over other traditional indexes [33] [32] [34], reducing memory footprints till 2 orders of magnitude while maintaining the excellent search properties of tree-like algorithms. This section will show the basics and the relationship with the indexes introduced previously.

### 4.4.1 Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) [18] of a string *R*, denoted as *B*, is a permutation of the symbols of *R*. Each value $B[i]$ stores the symbol immediately preceding the $i^{th}$ smallest suffix, and then can be generated from *R* and its suffix array by using the following expression: $B[i]$:=$R[(SA[i]$-$1)mod|R|]$. Hence in the example considered so far, if $R = acaaacatat\$$, then $B$=$ca\$atcaaaat$.

Figure 4.4 shows how to generate *B* from *R* and its suffix array, *SA*. The remarkable properties of the new representation of *R* are: (1) *B* can be compressed using general algorithms and achieve large compression ratios; (2) it is considered a self-index, allows to retrieve the original *R* from *B* without requiring additional structures; and (3) it is possible to perform the same search operations in *B* as in the *suffix-trie* described before. In this work we are mainly interested in the last two properties.

### 4.4.2 LF-Mapping

The LF-Mapping operation, standing for ”*Last-to-First column mapping*”, is the basic primitive used both to retrieve the original text and to perform text search operations. Given a

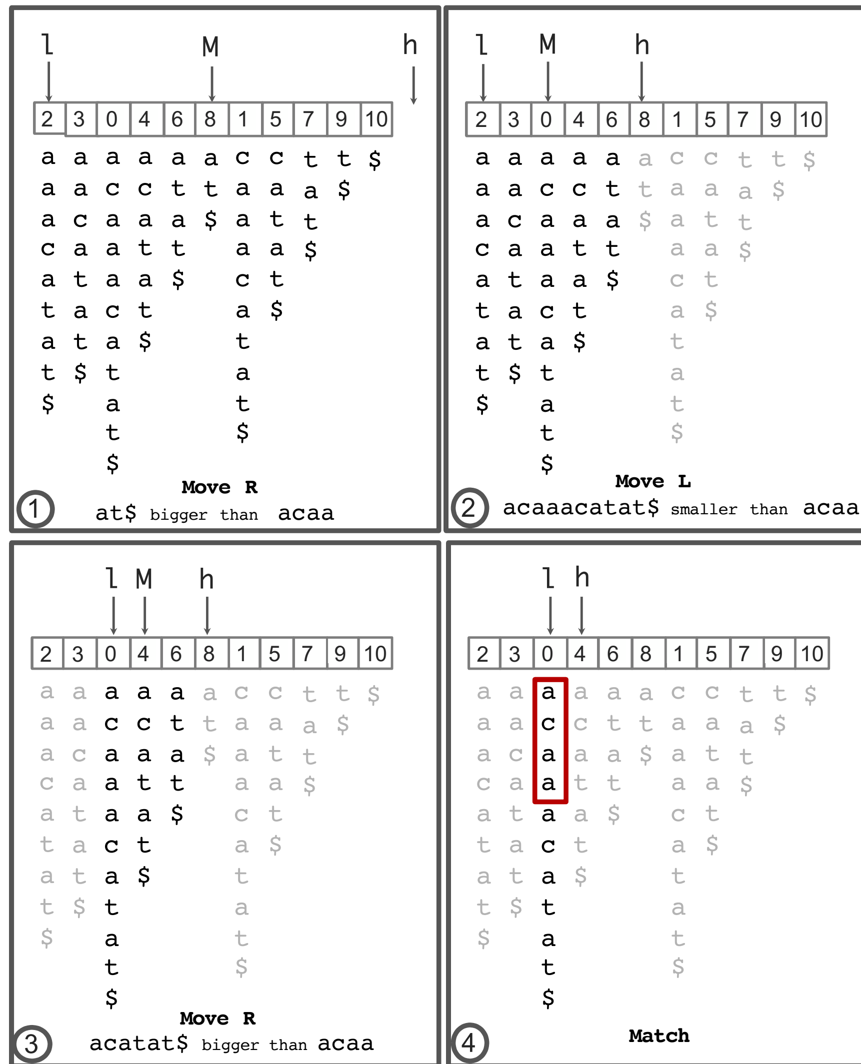


**Figure 4.3:** Forward-search process for the query $Q = acaa$ in the reference $R = acaaacata$ using a Suffix-Array index.

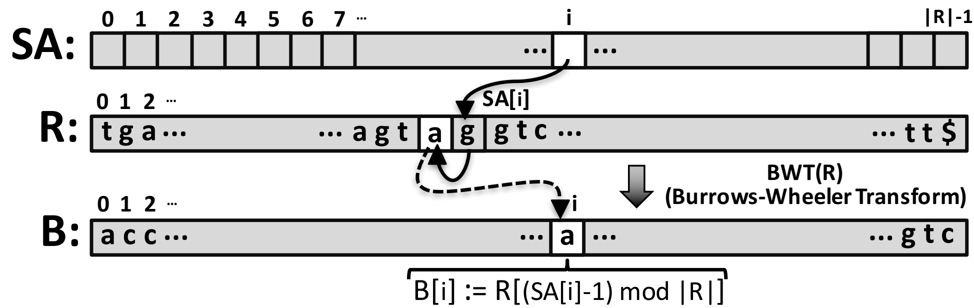


**Figure 4.4:** Generating the BWT representation of string $R$, denoted $B$, using the input string and its Suffix Array, $SA$.

$SA$ position of the suffix $S_i$, the LF-mapping function returns the SA position corresponding to suffix $S_{i-1}$. LF-Mapping is classically defined as the addition of two counting functions, $LF(B,s,pos) := C(B,s) + Occ(B,s,pos)$. Function $C(B,s)$ counts the number of occurrences in $B$ of symbols that are lexicographically lower than a given symbol $s$. Function $Occ(B,s,pos)$ counts the number of times a symbol $s$ appears in $B[0 \dots pos-1]$ (*i.e.* before position $pos$).

### 4.4.3 FM-index: the backward search

The FM-index *backward search* method (see Algorithm 1) computes the $SA$ interval of $Q[0 \dots m-1]$ using $m$ steps without requiring $R$ or $SA$. The initial search interval is set to the whole set of suffixes of $R$. The main loop performs $|Q|$ search steps. Each step applies the LF-mapping operation to both points of the interval using the FM-index data structure, $F$, and a new symbol of the query, $Q[i]$, and reduces the search space. The last step, which uses the first symbol of the query, provides the final interval (maybe empty). Figure 4.5 illustrates the details of the backward search process applied to the example considered so far. It consists of four steps, one per symbol in the input query, and each step involves two invocations of the LF-mapping operation.

#### Constant search complexity by memoizing LF-mapping

The computational cost of a straightforward naive implementation of the counting functions $C()$ and $Occ()$ is linear on the size of the BWT representation of the indexed text (and therefore linear on $|R|$=$n$), and would be prohibitive as a building block for a general searching algorithm. Instead, Ferragina and Manzini proposed an improvement to accelerate the computation of

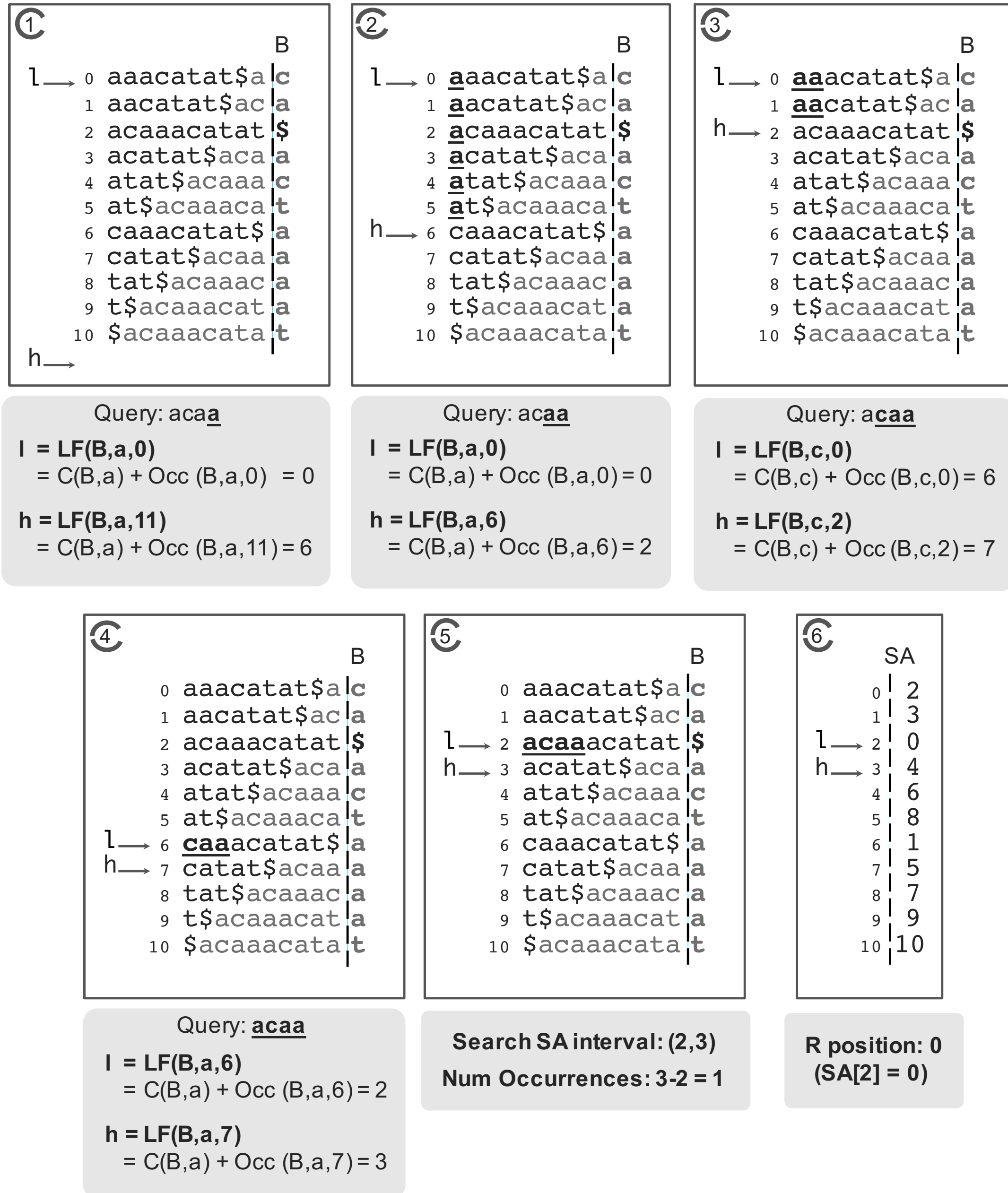


**Figure 4.5:** Backward-search process for finding all the occurrences of the query $Q = acaa$ in the reference $R = acaaacata\$$ using a FM-index and applying successive LF-mapping operations, decomposed as $LF(B,s,pos) := C(B,s) + Occ(B,s,pos)$.

**Algorithm 1:** Exact pattern search using the FM-index

```
   input  : B: BWT of reference R,  Q: query
   output : (l,h): SA interval of occurrences of Q in R
1  Function backward_search (B, Q)
2  |   (l,h) ← (0,|R|)
3  |   for i = |Q|−1 to 0 do
4  |   |   l ← LF (B,Q[i],l)
5  |   |   h ← LF (B,Q[i],h)
6  |   return (l,h)
```

both functions and, as a consequence, the LF-mapping operation [17]. The idea is to pre-compute or *memoize* the results of function $C()$ and $Occ()$ into an array data structure denoted as $LF$. The result of functions $C()$ and $Occ()$ for each input symbol and for each position of $B$ can be stored in the $LF$ array, so that the complexity of the LF-mapping operations is always $\Theta(1)$, independently of the position of the searching interval. Again, a straightforward naive representation of the $LF$ array requires at least $|\Sigma| \times n$ counters of $log_2 n$ bits, which is a prohibitive amount of data. Some authors, like [35], propose to store only a small fraction of the precomputed counters, and then calculate the counting functions by combining the use of those counters and the traditional representation of $B$. Next section explores index design strategies taking into account this fundamental idea, obtaining indexes with parametrised trade-off between index size and search complexity.

## 4.5 Sampled indexes: reduce space and search complexity

There are several details that must be considered when facing practical implementation issues addressed to real scenarios. Here we describe an implementation of the FM-index that introduces a parametrised trade-off between memory footprint and search execution time. We refer to this data structure as sampled FM-index, and it is still denoted as $F$. Additionally we present the relevant modifications of the LF-mapping primitive for a sampled index. More advanced optimizations and designs present in the literature or included in bioinformatics software in an undocumented way are also described. All these strategies are implemented in the proposal of this work and are presented as baseline in the experimental process of the thesis.

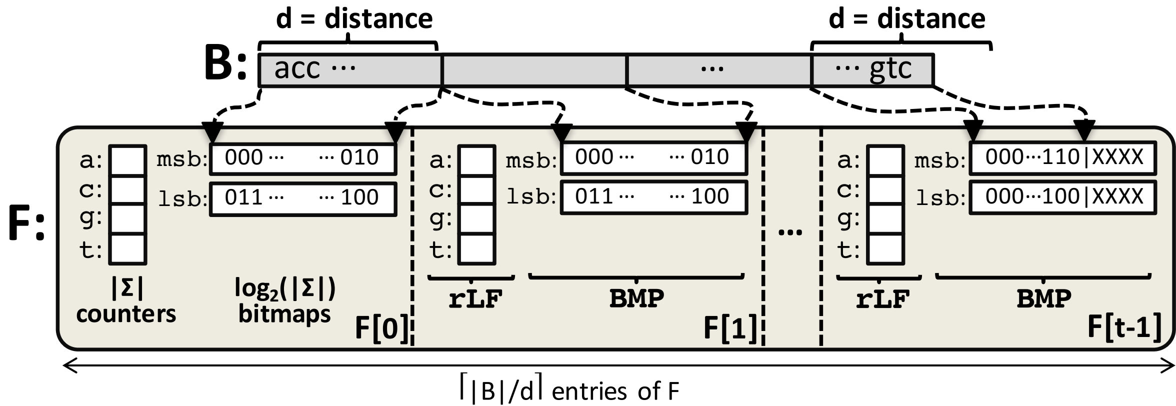


**Figure 4.6:** Sampled FM-index $F$ with samples at distance $d$. Each entry in $F$ contains sampled counters, *rLF*, and a bitmap representation of the symbols in $B$ corresponding to the sampled interval, *BMP*.

### 4.5.1 Sampled FM-index design

Figure 4.6 shows the data layout of the sampled FM-index corresponding to the BWT representation, $B$, of a given text. It is divided into blocks of $d$ consecutive symbols of the string $B$, together with their associated precomputed counters. The proposed data layout improves the memory access performance by grouping the data of each block into a single entry, $F[i]$, stored in a contiguous chunk of memory. Each substring of $d$ symbols is represented using bitmaps of the different bits of the binary representation of each symbol, denoted as *BMP*. Additionally, each entry in the sampled FM-index contains an array of counters, denoted *rLF*, with one counter per symbol.

Next, we describe the main characteristics of our proposed data structure, emphasizing the similarities and differences from what has been presented so far.

1. *Sampling the LF array*: proposals in the literature usually store only a small fraction of the precomputed *LF* array [35]. We also use a reduced *LF* table, denoted *rLF*, that holds the values of *LF* for the positions in the BWT representation, $p$, that are multiple of a certain fixed *sampling distance* $d$: *rLF*[s,p]=*LF*$[s, p\times d]$, for every symbol $s$. The remaining counters can be reconstructed from the sampled counters and $B$ in a maximum of $d$ steps. Therefore, parameter $d$ introduces a trade-off between memory footprint and computational complexity: while *rLF* will be $d$ times smaller than *LF*, each of the $m$ steps of the search algorithm will now have complexity $\Theta(d)$.

2. *Interleaving rLF and B*: memory locality is improved by splitting the contents of the $F$ array into $\lceil n/d \rceil$ blocks of $d$ consecutive symbols of the string $B$ (see Figure 4.6). Each block $F[p]$ holds both the bitmap representation of the symbols, encoded in $d \times \log_2 |\Sigma|$ bits (named *BMP*), and $|\Sigma|$ pre-computed counters (*rLF* in the figure).

3. *Memory-aligned data layout*: counters and bitmaps can be arranged into memory-aligned table entries (see for instance [36]). We select sampling distances $d$ such that the size of each table entry is an exact multiple of 32 Bytes, i.e. the size of a typical cache line. Then, for instance, a 32-Byte table entry allows encoding 4 integer counters of 4-Bytes per counter, and 2 bitmaps, each of 64 bits, that encode $d$=64 2-bit symbols.

### 4.5.2 Advanced LF-mapping designs

Algorithm 2 defines the implementation of the LF-mapping operation performed on a sampled FM-index design. The input position, $p$ is converted into an index and an offset, which are used to access the appropriate entry in the $F$ table, and to bound the scope of the symbols that must be counted. The pseudo-code of the *count* function illustrates how the bitmaps are handled using bit-wise *not*, *and* and *population_count* operations.

The main modifications and enhancements of the LF-mapping primitive for FM-index sampled indexes are described below:

1. *Speeding up symbol counting*: changing the data layout of the $B$ string to a bitmap representation allows counting symbols in terms of the logical and bit counting instructions available on current processors. If the word size of the processor is $w$ bits, then the complexity of counting symbols on a string of $d$ characters is $\Theta(d/w)$.

2. *Data prefetch and query interleave*: there is internal parallelism in the LF-mapping function, the intervals can be computed independently (as opposed to the search steps that are dependent). This allows the large memory latency to be overlapped between accesses to F. An efficient way to do this is by prefetching software. The $F$ entries are requested non-blocking in memory and are stored in the highest level cache until they are processed. In addition, this technique can be extended to concurrent searches, allowing overlapping requests of $F$ entries of several interleaved queries.

3. *Overlapping intervals*: it is common for $h$ and $l$ intervals to process the same input $F$, this situation occurs in the latest search steps where $(h-l) \geq d$. This fact is

easily identifiable at runtime, an efficiently strategy is request to memory a single F entry and reuse the data to process both intervals. This situation is accentuated for high sampling $d$, because it depends on the entry size.

**Algorithm 2:** LF-mapping operation on a sampled FM-index

**input :** $F$: sampled FM-index, $s$: symbol, $p$: position in $F$, $d$: sampling distance
**output :** $p'$: new position in $F$

```
1  Function LF (F, s, p)
2      index ← p/d
3      offset ← p mod d
4      entry ← F[index]
5      occ ← entry.rLF[s]
6      bitmap ← entry.BMP
7      cnt ← count (s, bitmap, offset)
8      return (occ + cnt)

9  Function count (s, bmp, len)
10     bmask ← (~0) << len
11     for i = 0 to |Σ|−1 do
12         smask ← 1 << i
13         if (s & smask) == 0 then
14             bmp[i] ←~ bmp[i]
15         bocc ← bocc & bmp[i]
16     return (popCount (bocc & bmask))
```

For clarity, we assume in our explanations the use of a DNA string with 4 bases (`A`, `C`, `G` and `T`) of up to 4 Gigabases. With $|\Sigma|$=4 only two bitmaps are needed. With a limit of $|B|$= 4 gigas, 32-bits are enough for storing the symbol counters. As in most implementations, the string terminator $ is not encoded; instead, its position in BWT is stored apart and checked whenever an LF-mapping is performed. Production setups might require alphabets with more than 4 symbols and counters larger than 32 bits. FM-index proposals described in follow chapters are extensive for other alphabet and reference parameters, moreover chapter 7 will analyse in deep how to deal with real data, applying the ideas presented in the thesis with larger alphabets and longer genomes, and how to combine sophisticated strategies to reduce the amount of computation required in a real genomic application.

## 4.6 Performance analysis of LF-mapping

Exact pattern searching using an FM-index (see Algorithm 1) performs recurrent LF-mapping operations to progressively close the query search space. At each step, two $F$ entries are read from the index and then some computation is done with the contents of each entry in order to generate the output $SA$ interval. Due to the characteristics of the BWT and search process, memory accesses are randomly spread along the whole $F$ data structure; for large references or query strings most of the memory accesses miss on the on-chip cache memories, resulting in an important performance drop.

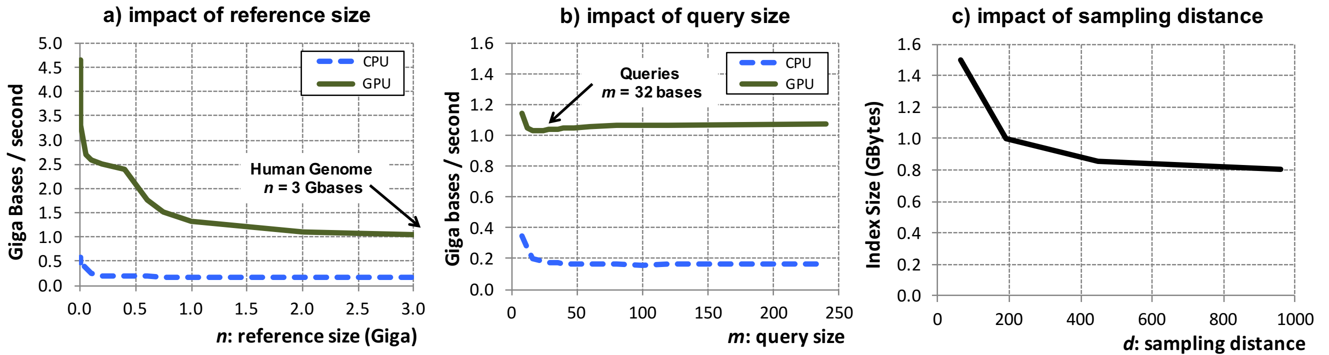


**Figure 4.7:** Impact on performance –in Giga base query operations per second– of (a) varying reference size $n$, and (b) query size $m$; and (c) impact on index size of varying sampling distance $d$

This section shows the performance of two implementations of the modified FM-index. The first one is a multi-threaded version running on an Intel Nehalem CPU architecture with four execution cores, and each core using its hyper-threading capacity. The second one is a highly optimized and massively parallel version (using a thread-cooperative scheme described later) running on a Nvidia Kepler GPU. Both of them exploit all the thread-level parallelism that is possible to use in each processor, and the CPU version uses explicit data prefetching and techniques described in 4.5 to increase the overlapping of memory latencies. The objective is to exploit as much of the memory bandwidth available in the system as possible.

Along this work, the performance of the search algorithm will be expressed in terms of query bases processed per unit of time. This metric is theoretically independent on the sizes of both the reference and the query, but in practice the processor memory hierarchy works more efficiently for small input problem sizes, which exhibit higher data access locality. Figure 4.7.a shows that, as expected, performance is higher for small reference sizes, both on CPU and GPU, in an scenario where most of the data read from the index is reused inside the on-chip caches. On the contrary, performance drops as the reference size increases, and the cache memory is not able to hold most of the reused data. We can conclude from the figure that the GPU reuses data more effectively than the CPU for index sizes lower than 500 million symbols. However, this work will address the analysis of bigger references, like the human genome, which are more important in real-life applications.

Figure 4.7.b shows that searching very small patterns ($m \leq 10$) provides a moderate performance advantage, especially on CPUs. This behaviour is also found in other search-tree traversal algorithms, where the index nodes on the top of the tree are accessed more frequently. With small queries, only a small portion of the FM-index is effectively accessed, which increases the temporal locality of the memory accesses. As queries get longer, the SA

intervals generated during the search process become narrower for most of the steps of the search process and both ends of the interval tend to point to the same index entry. This last behaviour provides additional memory locality, which explains why performance slightly improves for growing query sizes. Since performance is very similar for a large range of query sizes, we set all our experiments to use a query size of $m$=32 symbols.

The sampling distance, $d$, varies the compression ratio of the FM-index and creates a trade-off between memory capacity and computation requirements: the larger $d$, the smaller the size of $F$ but the higher the number of memory request and counting operations. Figure 4.7.c illustrates the reduction of the index size (memory capacity requirements) as $d$ is increased.

## 4.7 Conclusions

We have presented the basic concepts to understand the *FM-index* data structure and its associated operations. Afterwards, we have briefly reviewed the state-of-art on the design of indexes for read mapping applications, with the aim to highlight LF-mapping operation as the fundamental building block used to implement a high-performance exact matching algorithm. The next chapter will present several proposals to accelerate the LF-mapping operation, which are focused on (1) reduce the total number of random memory accesses and (2) reduce the memory footprint. We will show the performance impact of those techniques and a study of the index design parameters that can be tuned for building high performance search operations on large input strings.

# 5

# FM-index: algorithmic and design proposals

*"Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest."*

**Isaac Asimov**

Section 5.1 goes through the motivations of the $k$-step FM-index and *Alternate Counters* proposals. Section 5.2 describes the algorithmic details to build the $k$-step FM-index and to implement search operations. Section 5.3 explains the basics behind the *Alternate Counters* technique applied to FM-index designs. In section 5.4, we present a characterisation and performance analysis for both proposals in a CPU-based computer system, and analyze different index designs and setups in order to evaluate the trade-off between memory space and performance.

The current chapter presents two algorithmic proposals to improve the FM-index; the combination of both proposals opens the door for index designs that are more suitable to achieve higher performance in current computers. First, we introduce the *k-step FM-index*, which is a bloated index that exhibits higher spatial locality on data accesses and reduces the total number of random data accesses. Next, we describe the *Alternate Counters* technique, which reduces the memory footprint of the index through removing some redundancy on the preprocessed counters. We end the chapter by evaluating the proposals and the trade-off between memory space and performance.

## 5.1 Motivations and performance factors

As previously explained in chapter 3, computers are designed to perform efficient memory accesses to large data blocks (with data items placed in consecutive memory addresses) localised in a certain limited region of memory. System performance suffers very much when a program accesses small data blocks from a large memory area and the blocks are randomly scattered along that area. Chapter 4 showed that FM-index backward search matches that last access pattern and, for index sizes that are large enough, its performance becomes bounded by the main memory bandwidth of the system. The combination of its inherent random pattern of data accesses and the large indexed references used on bioinformatic applications, pushes the memory system over the edge, exposing large inefficiencies. The following proposals are oriented to alleviate these penalties (1) increasing the memory request size along with reducing the number of random accesses and (2) localising the accesses in a smaller data structure.

The inner loop of the search operations using an FM-index consists of a recurrence of dependent memory loads and counting operations (read stage followed by compute stage, followed by read stage . . . and so on). Basically, each iteration: (1) reads two entries of the FM-index at addresses calculated using the extreme index positions of a certain SA interval, and then (2) uses the contents of both entries to count symbols and generate the next SA interval. Furthermore, due to the pseudo-random nature of the input queries and the characteristics of the transformed text (using the Burrows-Wheeler transform), memory accesses will be spread along the whole FM-index structure with almost non temporal access locality. For large references, the consequence is that most of the accesses will miss the Last-Level Cache (LLC) and Translation Lookaside Buffer (TLB) issuing very inefficient data reads from external main memory. In addition, the lack of memory-level parallelism on

the inner loop exposes those large memory latencies in the execution time.

There is plenty of potential parallelism when considering independent query searches that can be exploited in the form of Thread-Level Parallelism (TLP) to improve memory bandwidth utilization. Multiple threads generating independent memory requests may fill the memory pipeline and increase the utilization of the available bandwidth. However, FM-index accesses exhibit very low spatial locality, since only a 4-Byte counter and an 8-Byte bitmap are read on each memory read step. The proposed implementation described in Chapter 4 groups counters and bitmaps together in a single contiguous entry to maximise spatial locality, but it is not enough to fully exploit memory bandwidth.

The $k$-step algorithm that we introduce in this chapter collapses $k$ search iterations into a single one, replacing $k$ pseudo-random, dependent memory requests to scattered small data items by one request to approximately the same amount of data placed into a larger data block. The overall effect is an increase of spatial locality and a better exploitation of the available memory bandwidth. Slightly less data is requested by the program because a single counter replaces $k$ counters, but the total amount of bitmap information read by the program is basically the same regardless of $k$.

On the other hand, as we pointed out in the previous chapter, the performance of random accesses drops for excessively large memory footprints. A proposal that we call *Alternate Counters* reduces the memory footprint by dispensing with half of the preprocessed counters at the cost of a small increase in computation. The key insight is that there is some redundancy in the information of the counters that can be exploited to provide a more compact representation of the index.

## 5.2 *k*-step FM-index: a faster bloated index

We propose an extension of the original FM-index design, denoted as $k$-step FM-index, which is a bloated index that exhibits higher spatial locality and reduces the total amount of random data accesses. The proposal accelerates the backward-search operation at the expense of increasing the index size. The number of LF-mapping operations is reduced by $k$ because the traversal of the search tree gives steps of $k$ symbols at a time.

The proposed idea is illustrated in Figure 5.1, which shows an example of a search operation on two suffix-tries: (a) a single-step suffix-trie requires four search steps to find the occurrences of a query of 4 symbols in the represented index, while (b) a 2-step suffix-trie needs two search steps to achieve the same result. As we will show, the total amount of

computational work and the total amount of data read on the task of searching a query in the index remains almost the same as for the original algorithm. The advantage of the proposal relies on the increased spatial locality of data accesses, which allows applications that are already bounded by random memory accesses to read larger blocks from memory almost for free.

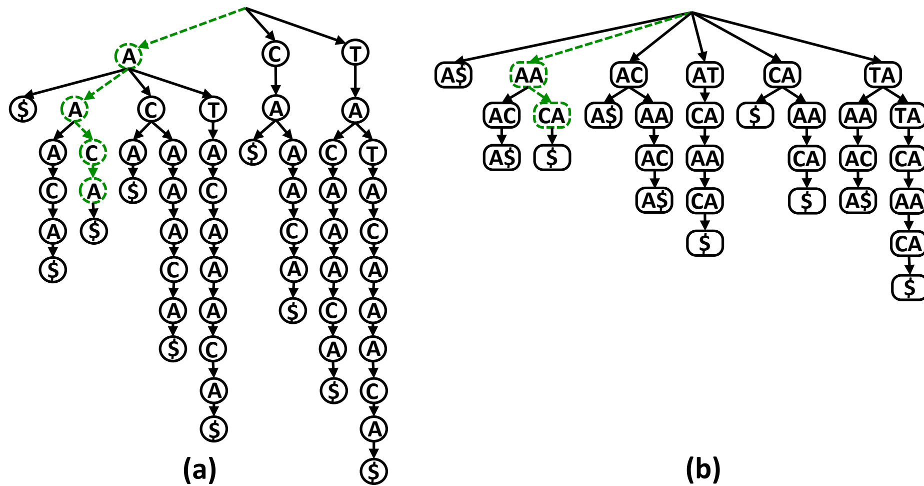


**Figure 5.1:** Backward-search of $Q = acaa$ in $R = acaaacatat$ using the *suffix-trie* of $R$ with a) single-step backward-search, and b) 2-step backward-search.

### 5.2.1 *k*-step BWT: a two-dimensional BWT

The proposal uses a two-dimensional Burrows-Wheeler Transform (BWT) of a string $R$, denoted $k_B$, that keeps the same properties of the original BWT. $k_B$ is composed by a set of $k$ strings, $k_B = \{B[0], B[1] \dots B[k\text{-}1]\}$, containing a total of $k \cdot |R|$ symbols. Each string is a different permutation of the symbols of $R$ and represents the BWT of $R$ for a different depth $j$, $0 <= j <= k$-1. For example, $B[0]$ represents a different notation for $B$, the traditional BWT. The $i^{th}$ position of string $B[j]$ is computed as $B[j][i] := R[(SA[i] - j - 1) \bmod |R|]$. Figure 5.2 illustrates the process of generating $k_B$. For example, the $2_B$ transform of $R = acaaacatat\$$ is $\{B[0], B[1]\}$ = {ca\$actaaaat, actaaa\$acta}.

The $k$-step BWT of a string $R$ can be constructed from the one-step BWT of $R$ without requiring the original $R$ nor the Suffix Array $SA$. This can be an appropriate strategy on systems with low memory capacity. The idea is that each BWT string corresponding to a certain depth $j$ can be constructed from the BWT string corresponding to depth $j$-1 using the LF-mapping operation described in the previous chapter: $B[j][LF(B[j-1][i], i)] := B[j-1][i]$.

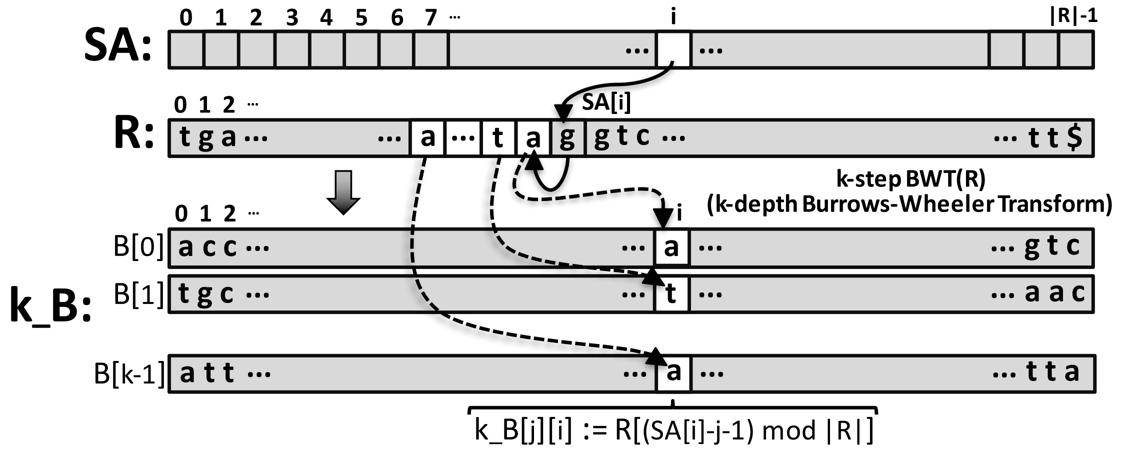


**Figure 5.2:** $k_B$ is the $k$-step BWT generated from $R$ and $SA$.

Using this strategy, the index can be constructed using just $2 \times |R|$ characters, each character represented with $log_2|\Sigma|$ bits.

### 5.2.2 Sampled *k*-step FM-Index design

The strategies described in section 4.5 for the sampled FM-index are also used for the $k$-step version. Figure 5.3 shows the generation of $k_F$ ($k$-step FM-index) from $k_B$. Again, one entry is generated for every block of size $d$ in $k_B$, and each entry contains $|\Sigma|^k$ $rLF$ counters and $k \cdot log_2|\Sigma|$ bitmaps of size $d$. While the size dedicated to bitmaps still grows linearly with $k$, the size dedicated to counters now grows exponentially with $k$. For example, with $d = 32$, $|R|$= 1.5G and $k = \{1,2,3,4\}$ the sizes of $k_F$ are $\{1.0GB, 3.6GB, 12.8GB, 48.3GB\}$. As we will analyse later, increasing $d$ allows reducing the memory requirements for counters. Figure 5.5.b shows the memory layout of a 2-step FM-index, which can be compared to the memory layout of a single-step FM-index in Figure 5.5.a.

### 5.2.3 *k*-step FM-Index backward search

Algorithm 3 shows the pseudo-code of our proposal. A search step groups $k$ consecutive symbols $s_1 \cdot s_2 \ldots s_k$ of $Q$ from alphabet $\Sigma$ and generates a new symbol $s$ from the alphabet $\Sigma^k$. Then, $s$ indexes the appropriate $rLF$ counter, and function $k_LF$ counts the occurrences of $s$ in the actual $k_F$ block. The function uses the same bit-level optimisations described in section 4.5, but requires $k$ times more operations. Also, since \$ appears in $k$ positions of $k_B$,

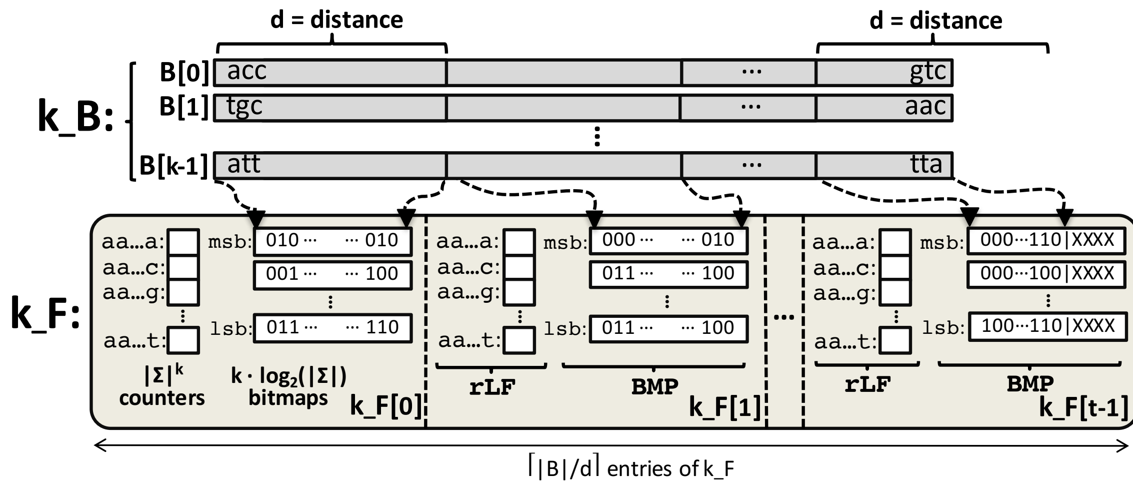


**Figure 5.3:** *k_F* is the *k*-step FM-index of *k_B*.

*k* conditions must be checked per search step. However, since the number of search steps is reduced by *k*, the total number of comparisons per query will remain the same. The LF operation on *k_F* is exactly the same as that depicted in Algorithm 2, but this time using a symbol from a larger alphabet and larger data structures.

**Algorithm 3:** *k*-step Backward Search ($*$ special case required when $|Q|$ is not multiple of $k$)

**input :** *k_F*: *k*-step FM-index of reference *R*, *Q*: *query*, *s*: symbol, *p*: position in *k_F*, *d*: sampling distance
**output :** $(l,h)$: SA interval of occurrences of *Q* in *R*

1 **Function** *backward_search (k_F, Q)*
2   $(l,h) \leftarrow (0,|R|)$
3   $*$ **for** $i = |Q|-k$ **to** 0 ***step*** $-k$ **do**
4     $s \leftarrow Q[i \dots i+k-1]$
5     $l \leftarrow k_LF\ (k_F, s, l)$
6     $h \leftarrow k_LF\ (k_F, s, h)$
7   **return** $(l,h)$

8 **Function** *k_LF* $(k_F, s, p)$
9   $entry \leftarrow k_F[p/d]$
10   $offset \leftarrow p \bmod d$
11   $occ \leftarrow entry.rLF[s]$
12   $bitmap \leftarrow entry.BMP$
13   $cnt \leftarrow k_count\ (s, bitmap, offset)$
14   **return** $(occ + cnt)$

A final corner case happens when $|Q|$ is not multiple of *k*, and the last search step involves less than *k* symbols, say *r*. The solution is to aggregate all the *rLF* counters matching with the *r* initial symbols of *s*, and counting occurrences on *k_B* ignoring the last *k*-*r* symbols.

Figure 5.4 illustrates the process of searching a query in a k-step FM-index, with $k = 2$. Notice that only 2 search steps are needed to find a query of 4 characters.

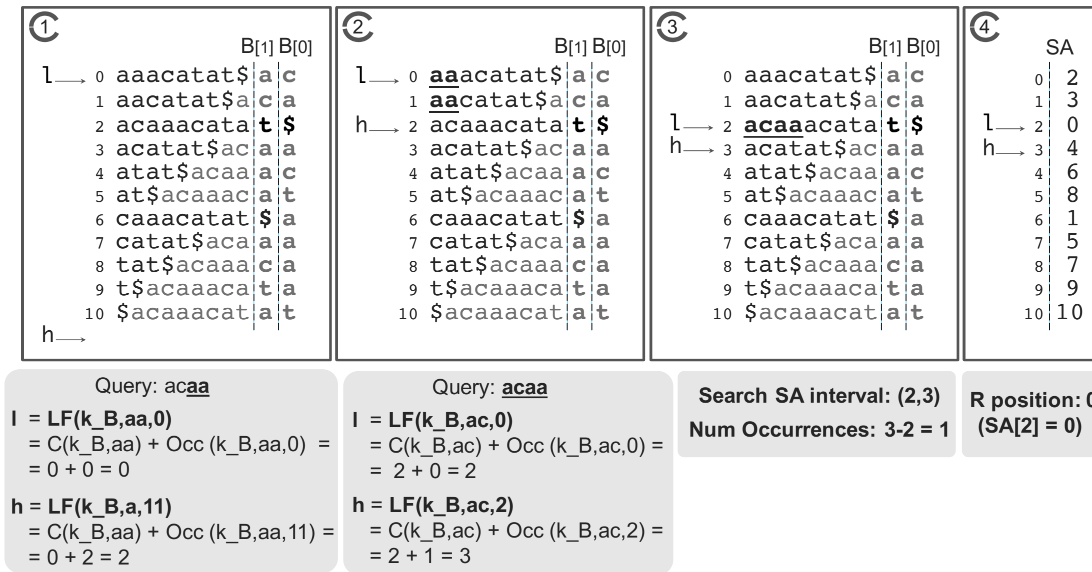


**Figure 5.4:** Backward-search process for finding all the occurrences of the query $Q = acaa$ in the reference $R = acaaacata\$$ using a k-step FM-index and applying a reduced number of LF-mapping operations, decomposed as $k_LF(B,s,pos) := C(B,s) + k_Occ(k_B,s,pos)$.

## 5.3 Alternate Counters: reducing memory requirements

One way to reduce the size of the FM-index is to use large sampling distances, but this strategy also increases the amount of computational work. We propose a different way of reducing the memory footprint at the cost of a small increase in computation, i.e. by dispensing with half of the counters. More in detail, we use alternate counters as depicted in Figure 5.5.c: odd FM-index entries contain *rLF* counters for the first half of the symbols, while even FM-index entries contain counters for the second half of the symbols.

Algorithm 4 illustrates an LF operation on a $k$-step sampled FM-index with alternate counters. $s$ is an input symbol that concatenates $k$ original symbols. Depending on whether the identifier for the index entry is odd or even, and on whether $s$ belongs to the first or second half of the symbols, the operation is performed as usual. Otherwise, the counters of the next entry of $k_F$ must be used, and the symbols in the BWT bitmaps must be counted backward. Counting forward or backward has the same computational cost, and the extra access to a contiguous $k_F$ entry is often free, given the performance behaviour of random accesses.

Figure 5.6 compares the memory footprints of the different indexing schemes proposed so far for several values of the sampling distance, $d$, and for several values of the number of steps, $k$. Using alternate counters halves the amount of memory devoted for counters, which

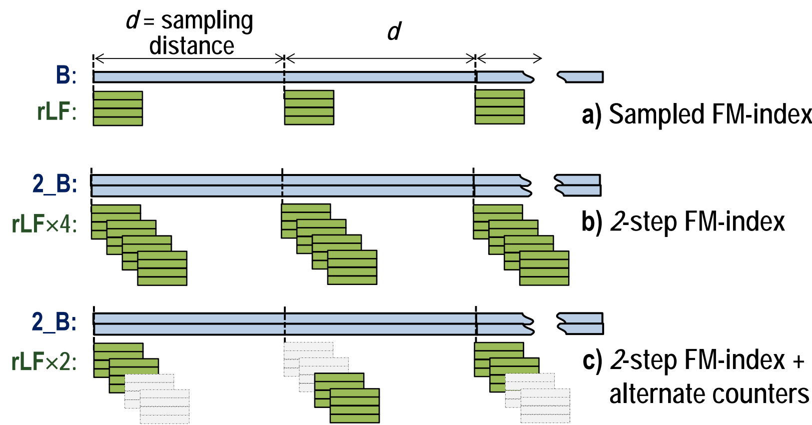


**Figure 5.5:** Different layouts of the FM-index proposals

has more impact on indexes with a higher step size, $k$, or lower sampling distance, $d$. Notice the larger scale in the vertical axis for the bars at the right side, for step sizes of $k$=3 and 4.

## 5.4 Performance analysis on CPU

We have executed the proposed $k$-step FM-index algorithm in a multicore CPU system in order to evaluate its performance and identify the architectural features that contribute to that performance.

### 5.4.1 Experimental Setup and Methodology

The computer system that we use in our experiments is a dual-socket Intel Xeon E5645, each socket containing 6 Westmere cores at 2.4 GHz, and each core being able to execute 2 hardware threads using hyperthreading technology. Therefore, it can simultaneusly execute up to 24 threads. 6 DIMMs of 8 GiB 1333-MHz DDR3 RAM memory per 2 sockets provide 96 GiB of storage capacity. The Last Level Cache (LLC) provides 12 MiB of shared storage for all the cores in the same socket. Each socket provides a peak external memory bandwidth of 32 GB/s, and the Quickpath interconnection (QPI) between the two sockets provides a peak bandwidth of 11.72 GB/s per link direction (a total of 23.44 GB/s).

The genome references used in our experiments are built from the Human genome (GRCh37). The sequencing reads used as queries are generated from pseudo-random positions on the genome reference using the application script *genreads* from the *Mummer* project [37].

**Algorithm 4:** LF operation using alternate counters

**input :** $k_F$: $k$-step FM-index of reference $R$, $s$: symbol, $\sigma$: $|\Sigma|^k$, $p$: position in $k_F$, $d$: sampling distance
**output :** $p'$: new position in k_F

```
1  Function k_LF (k_F, p, s)
2      idx ← p/d
3      offset ← p mod d
4      entry ← k_F[idx]
5      if ((s < σ/2) == even(idx)) then
6          cnt ← forward_count_k (s, entry.BMP[0...offset-1])
7          return entry.rLF[s mod (σ/2)] + cnt
8      else
9          nextEntry ← k_F[idx + 1]
10         cnt ← backward_count_k (s, entry.BMP[offset...d-1])
11         return nextEntry.rLF[s mod (σ/2)] - cnt
```

The factors analyzed in our experiments are: (1) the number of steps used by the FM-index, $k = \{1, 2, 3, 4\}$; (2) the reference size, $|R| = \{500, 2K, 5K, 20K, 60K, 200K, 600K, 2M, 8M, 50M, 100M, 400M, 750M, 1500M\}$; (3) the number of threads that cooperate in the execution of the whole task, $thr = \{1, 12, 24\}$; and (4) the distance between counters in the FM-index representation, $d = \{32, 64, 128, 256\}$. When not mentioned, we assume $thr$=24, $d$=32 and $|R|$=1.5G, which represents the best performing configuration for this reference size.

The parameters fixed in our experiments are the total number of queries, set to 10 million, and the query length, $|Q| = 120$. We have checked that, as expected, execution time grows linearly with the total number of queries. Therefore, using a larger number of queries does not provide any additional information. Also, experiments with shorter reads, from 60 to 120 bp, produce similar performance results and are skipped from our analysis.

Each execution experiment is repeated 10 times, and the provided metric is computed as the average of the 3 experiments with lower execution time. The input data, the reference text, $R$, and all the queries, are stored in DRAM just before starting execution. The *Likwid tool* [38] is used for reading H/W performance counters: execution cycles, instructions executed, and number of Bytes read and written from DRAM. A Likwid command cleans up the NUMA domain at execution start to assure the same behaviour in each execution.

Overall performance is computed as the average CPU time for each query (*time/query*), and the amount of Bytes read from memory is also averaged per query (Bytes/Query). Other metrics are the instructions executed per query (Icount/query), IPC (instructions per cycle

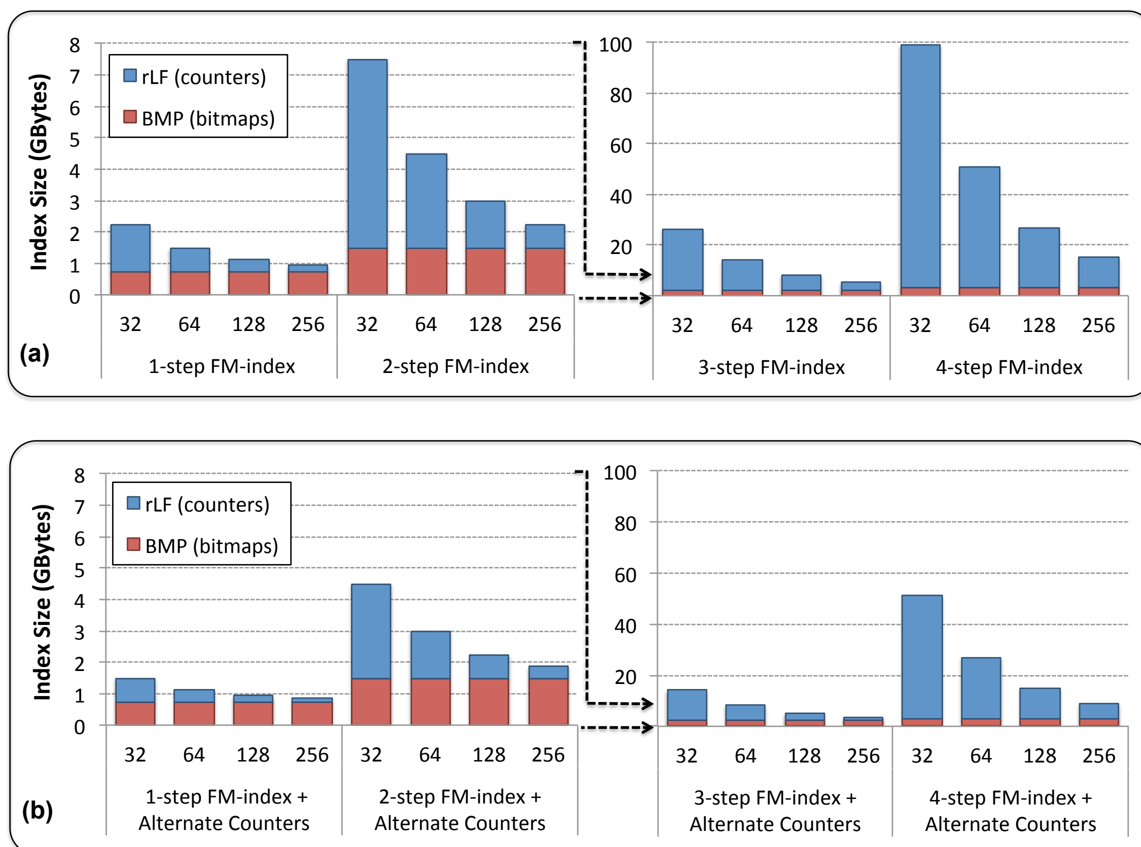


**Figure 5.6:** Memory footprint of our FM-index implementations for the human genome using several sampling distances $d$= 32, 64, 128, 256; steps $k$= 1, 2, 3 and 4; and indexing structures: a) with a general k-step configuration, and b) including the technique of alternate counters.

rate) per execution core, and total DRAM bandwidth consumption.

We use OpenMP to statically distribute among threads the task of searching the 10 million input queries in the reference text. Data prefetching instructions are used on each search step to shift the memory request in time respect to the point when the data is actually needed, increasing the overlap between memory access and computation, and reducing memory waiting time [39].

## 5.4.2 Performance of the $k$-step FM-Index search

Figure 5.7.a compares performance for varying reference sizes, $|R|$, and steps, $k$. Increasing $k$ almost always provides better performance in the range analyzed. There is a sharp performance degradation when $|R|$ reaches values on the order of 10M as the corresponding index does not fit into the LLC. It is after this point when our proposal provides the higher benefits, achieving

speedups between 1.4× and 2.4× (see Figure 5.7.b). As stated in the introduction, interesting sequence alignment problems handle very large references.

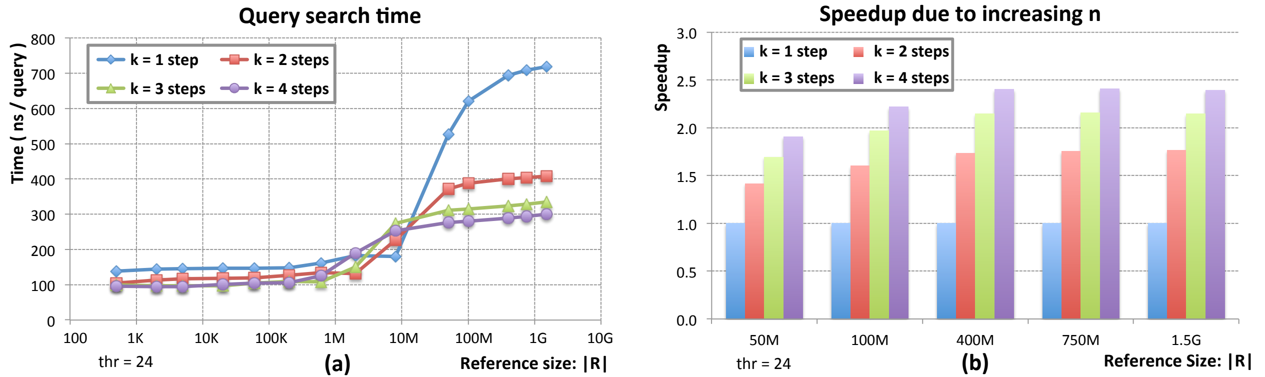


**Figure 5.7:** (a) Performance (ns/query) when varying $k$ and $|R|$; (b) Speedup when increasing $k$ for selected values of $|R|$.

### 5.4.3 Efficiency of sequential and parallel execution

The $k$-step algorithm collapses $k$ search iterations into a single one, reducing the total instruction count and, in a lower extent, the amount of data read by the program. The effect on the instruction count is similar to that produced by loop-unrolling: fewer loop control instructions and folding up some operations. Additionally, slightly less data is requested by the program because a single counter replaces $k$ counters, but the total amount of bitmap information read by the program is basically the same regardless of $k$.

As expected, the instruction count per query (*Icount*/Query) is almost constant for growing $|R|$. Results for $k = \{1,2,3,4\}$ are *Icount*/Query = $\{10K, 7.7K, 7.4K, 7.2K\}$, with a 23% reduction for $k = 2$, and around 27% for larger $k$'s. Figure 5.8.a shows the IPC rate per core for varying $k$ and $|R|$. Small values of $|R|$ allow achieving IPC rates between 2 and 2.5, fairly near to the theoretical peak of 3-4. Large values of $|R|$ provoke LLC misses that reduce IPC rate between 3 and 5 times. The effect of large DRAM latencies is clearly seen here and performance becomes memory-bounded. Also, increasing $k$ provides higher IPC rates, meaning a better tolerance to memory performance problems.

Figure 5.8.b shows the good scalability of our implementation of FM-index when exploiting TLP. Using more execution cores provides almost linear speedup. Exploiting the H/W multithreading capability of each core still improves performance by $\{1.67\times, 1.68\times, 1.56\times, 1.49\times\}$ for growing $k$, which proves that single-thread performance is limited by memory dependencies and latencies, as expected.

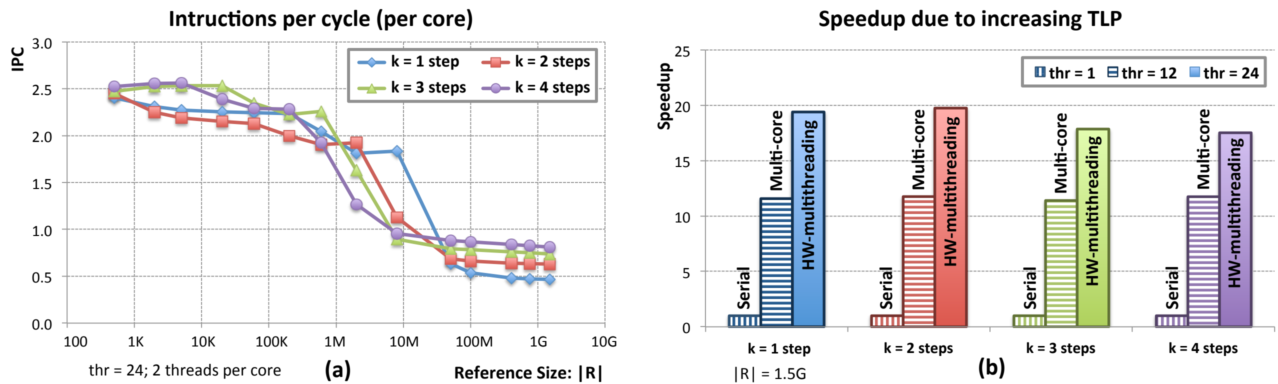


**Figure 5.8:** Efficiency of Execution: (a) Instructions per Cycle per Core; (b) Speedup due to multi-threading.

Multithreading, memory prefetching, and our $k$-step proposal have been effective to hide large DRAM latencies, but the best performing configuration still works with IPC rates that are 3 times lower than the case where indexes fit into the LLC. Therefore, performance for large indexes is still bounded by memory bandwidth issues.

### 5.4.4 Efficiency of Memory operations

Figure 5.9.a compares the amount of data requested by the application (left) with the amount of data actually read from DRAM (right). The ratio requested/read for $k = \{1,2,3,4\}$ is $\{0.16, 0.24, 0.28, 0.28\}$, indicating a very low efficiency of the memory hierarchy. Our proposal, which increases the spatial locality of data accesses, improves DRAM access efficiency by $1.5\times$-$1.8\times$. Results also show a $1.2\times$-$1.4\times$ reduction of data read requirements when increasing $k$. These two factors provide the major benefits of our proposal.

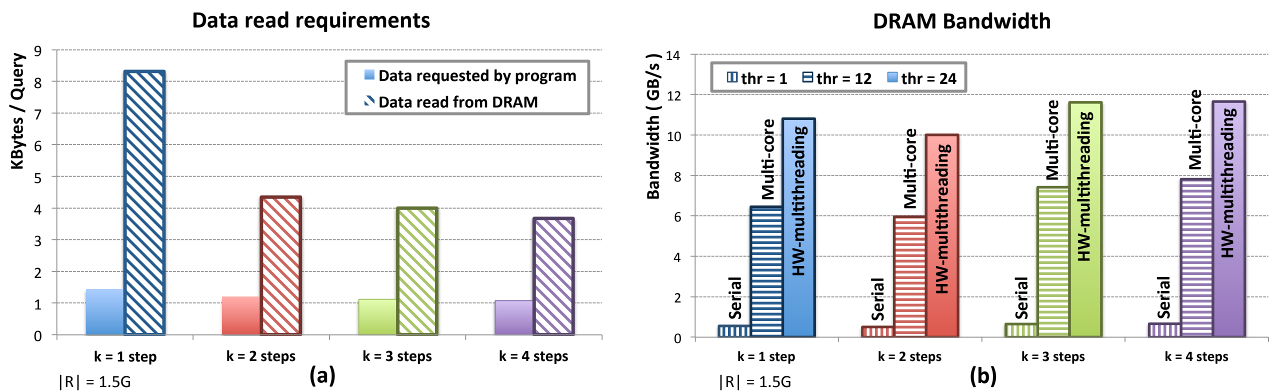


**Figure 5.9:** Analysis of data read requirements: (a) Bytes requested per query; (b) DRAM bandwidth consumption.

Figure 5.9.b illustrates the effective DRAM bandwidth consumed by the program for different step sizes, $k$, and total number of executing threads, $thr$. DRAM bandwidth seems to saturate for all values of $k$ at around $11GB/sec$, which is around 6 times lower than the peak DRAM bandwidth and around 2 times lower than the peak QPI bandwidth. There must be a number of restrictions, other than link bandwidth, that are limiting performance. Examples are the total number of DRAM pages that can be simultaneously open, the size of intermediate queues, TLB misses ... The pseudo-random nature of the data access pattern poses higher difficulties to the H/W memory pipeline, which is optimized for sequential data streams. Our proposal improves performance by reducing the total number of such costly memory accesses.

### 5.4.5 Trading Memory requirements for Performance

The cost of the $k$-step FM-index is an increase in memory storage requirements. We can reduce index size using a larger distance $d$, which increments computational work and the total amount of data read from memory. We have evaluated an initial version of the algorithm for large values of $d$. This version is based on the highly-optimized implementation for $d$ = 32, and, for example, does not fully exploit the extra data-level parallelism provided by larger $d$'s (by means of SIMD instructions).

Figure 5.10 shows how performance degrades as we increase $d$. Compression is more effective for large values of $k$, since it reduces the number of counters, which grow exponentially with $k$. However, at certain point, increasing $d$ provides lower compression ratios at the cost of higher performance penalties. The case example depicted in Figure 5.10 shows that when DRAM size is very restrictive (less than 1.2 GB) the best solution is always the single-step method with an appropriate value of $d$. However, if we have additional DRAM space, we can use it to improve performance. With 2 GB, we may improve performance by 1.2× by using a 2-step FM-index with $d = 128$. Doubling DRAM size to 4 GB provides an additional 1.5× improvement when reducing $d$ to 32. Additional DRAM capacity does improve performance, but with diminishing returns.

## 5.5 Conclusions

We have described and evaluated two algorithmic proposals that improve the FM-index described in Chapter 4: the $k$-step FM-index and the Alternate Counters technique. They represent a trade-off between required memory space and memory access locality, which also translates into a trade-off between the amount of computation work and the efficiency

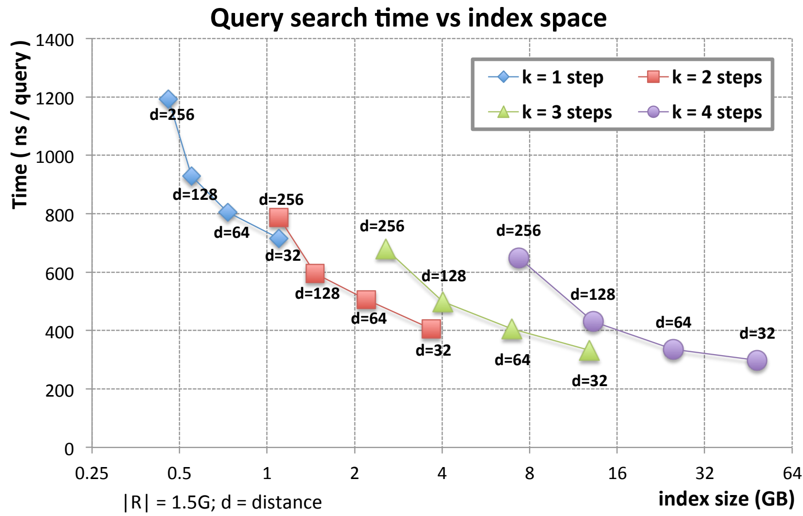


**Figure 5.10:** Trading Performance versus Memory requirements by increasing the distance $d$ between counters.

of performing memory accesses. The strategies used to generate more compact indexes come at the expense of additional computation work. Using alternate counters increases computation very slightly, due to the conditions that must be checked in order to select which counters to use. More computation work is required when the sampling distance, $d$, is increased, since more bitmap data must be read and processed on each step. However, this additional computation work is usually not a major concern on the GPUs, as there the FM-index search algorithms are typically memory-, and not computation-, bound. We will explore the acceleration of the algorithm using GPUs in the next chapter.

# 6

# FM-index: GPU Parallel designs for LF-mapping primitive

*"Science, my lad, is made up of mistakes, but they are mistakes which it is useful to make, because they lead little by little to the truth."*

**Jules Verne**

The current chapter describes a fine-grain massive parallelization proposal for GPU-acceleration that improves the backward FM-index search process. The proposal improves the overall search performance, with excellent performance scalability, and achieves a high index compression ratio without sacrificing performance. The benefit of a large compression ratio, in combination with the techniques presented in Chapter 5, makes the FM-index a suitable structure for GPUs, which outperforms current proposals for CPU. The chapter presents and evaluates the performance of three state-of-the-art parallelization schemes, identifying the potential performance hot-spots suffered by each of them. We end the chapter by evaluating the trade-offs between memory space and performance, and the power consumption improvements of our proposals when executed on different CPU and GPU devices.

## 6.1 Introduction

We defined in Chapter 4 the LF-mapping operation as the fundamental primitive for searching a query inside a text indexed by using FM-index structures. Our computational analysis revealed that the random memory access pattern that is inherent to the LF-mapping operation degrades the performance of the system significantly. In this chapter we present a thread-cooperative parallelization of the LF-mapping primitive that exploits the inter-task parallelism in order to alleviate those memory performance inefficiencies. The proposal improves (1) the overall search performance, (2) the performance scalability and (3) the index compression ratio.

In Chapter 5 we presented two algorithmic proposals, the *k*-step FM-index organization and the Alternated Counters scheme, that increase the data locality exhibited by the accesses to the index, and alleviate the inefficiencies of the memory subsystem on current CPU processors. The current chapter explains how to adapt those proposals to GPU systems, and evaluates their effectiveness. The large compression ratio achieved by the proposed thread-cooperative scheme in combination with the later algorithmic ideas makes the FM-index a suitable index structure for GPU-acceleration, which largely outperforms state-of-the-art proposals running on CPUs.

Section 7.1 describes in detail the fine-grain parallelization, .. Section 6.3 describes the traditional parallelizations for GPU applied to FM-index, which is relevant to understand the contribution of the proposal. Experimentation methodology analyzes their compute-memory overlapping behaviour. In sections from 7.3.2 to 6.6.6 a experimentation is conducted to evaluate the impact of the proposal on GPU systems.

Section 6.6.7 describes the performance impact in terms of architectural memory system behaviour and analyzes the influence on the performance of each contribution in different GPU systems, in addition, a comparison with the Nvidia NVBIO library [11] framework is performed.

Finally an extra experimentation is carried out evaluating the suitability of different index parameters and their impact in the system. A performance and power efficient evaluation is performed for different desktop and low-consumption GPU systems. Section 6.7 provides a discussion of the key results from the previous thoughtful analysis.

## 6.2 Exploiting inter- and intra-task parallelism on LF-mapping

Current section provides a performance analysis and characterisation about the LF-mapping primitive on GPU architectures. The next points review the inter-task issues on GPU, their performance impact on the LF-mapping primitive and discuss the benefits of the intra-task parallel proposed on this chapter.

State-of-art LF-mapping proposals accelerated by GPU are based on the inherent straight-forward parallelism traditionally exploited on multi-core CPU approaches. These inter-task parallel approaches under-used GPU processor resources and present some performance issues described in detail on chapter 3. The major LF-mapping GPU performance issues come from:

- **Compute-thread divergence:** The FM-index exact search is conducted by a set of chained LF-mapping operations, the amount of work performed in terms of LF-mapping operations depends on (1) the pattern length and (2) the contents of the pattern and reference. A critical performance situation arise when few threads from a CUDA block are processing large patterns and all of them match to the genome reference. These irregular number of LF-mappings operations per thread cause an under-utilization of vector computational units at warp and stream processor levels. Proposals in next sections show how to divide the internal work along different threads to increase the work regularisation and therefore the search performance.

- **Memory-thread divergence:** An LF-mapping operation needs to request from main memory a full FM-index entry. Each of these entry requests present (1) an inherent random memory access pattern (as we analysed on chapter 4) and (2) a different number of loads to read the full entry. Traditional task-parallel approaches are compelled to bring the complete entry into local memories and those require different iterative loads subjected to the entry size. This situation, combined with the random memory pattern, transforms all the vectorized loads into gather memory operations (memory-thread divergence) which are executed with extremely poor performance on GPUs. Next section will show how to obtain high efficient memory utilization with the LF-mapping primitive on GPUs using a coordinated thread memory accesses. Full use of thread warp memory access demands that all threads access simultaneously to consecutive data and that is only possible using a thread-cooperative oriented solutions.

As we introduce in chapter 3, GPU architectures allow efficient fine-grain parallelizations even when the amount of work per thread is certainly limited, as is the case for the LF-mapping primitive. That comes from the fact that GPU thread managing operations are significantly more efficient than traditional CPU systems. Next sections will describe how to reduce the task-parallel inefficiencies previously exposed, with the usage of complex thread communications and synchronizations allowing to the threads collaborate in a shared LF-mapping task to report the result (here we denominate this parallelization 'thread-cooperative').

In addition, we will show to how implement this cooperative approach for the LF-mapping primitive, allowing to assign constant work per thread and dynamically increase or decrease the thread number with the aim to fit for the best task size requirements.

## 6.3 LF-Mapping: Task-parallel designs

Published FM-index implementations on GPU are based on straightforward $task-parallel$ approaches, where each task corresponds to searching a different query in a shared FM-index. On those GPU designs, each thread carries out a single query search independently of the other threads, narrowing the *SA* interval by computing both the $l$ and $h$ positions along the search process. The performance of the task-parallel scheme is suboptimal due to the 32 threads of a warp requesting data words from different scattered memory locations; this access pattern forces the GPU to re-issue the load instruction for each non-coalesced memory block, making the caches (L2 and TLBs) the main performance bottleneck.

As shown in the benchmark section, a simple way of enhancing this design can be achieved by using two separate threads to operate on each *SA* interval; each thread applies LF operations to either the previous $l$ or the previous $h$ position of the interval. Most of the time the L and R pointers of the *SA* interval from the same query are mapped to the same index entry and hence half of the threads in a warp are requesting the same data than the other half. This improves the coalescing process performed by the GPU when handling the memory requests of a warp. Figure 6.1.a provides a representation of the improved task-parallel execution flow. In the figure, each thread executes multiple consecutive memory load instructions to read a full index entry from global memory (the arrow with a surrounded blue square) and then executes the corresponding counting instructions (the arrow with a surrounding red square).

In spite of those improvements, the task-parallel design is bound to become worse when each thread needs to read a larger memory block, as is the case for large sampling distances $d$ or when using the $k$-step approach.

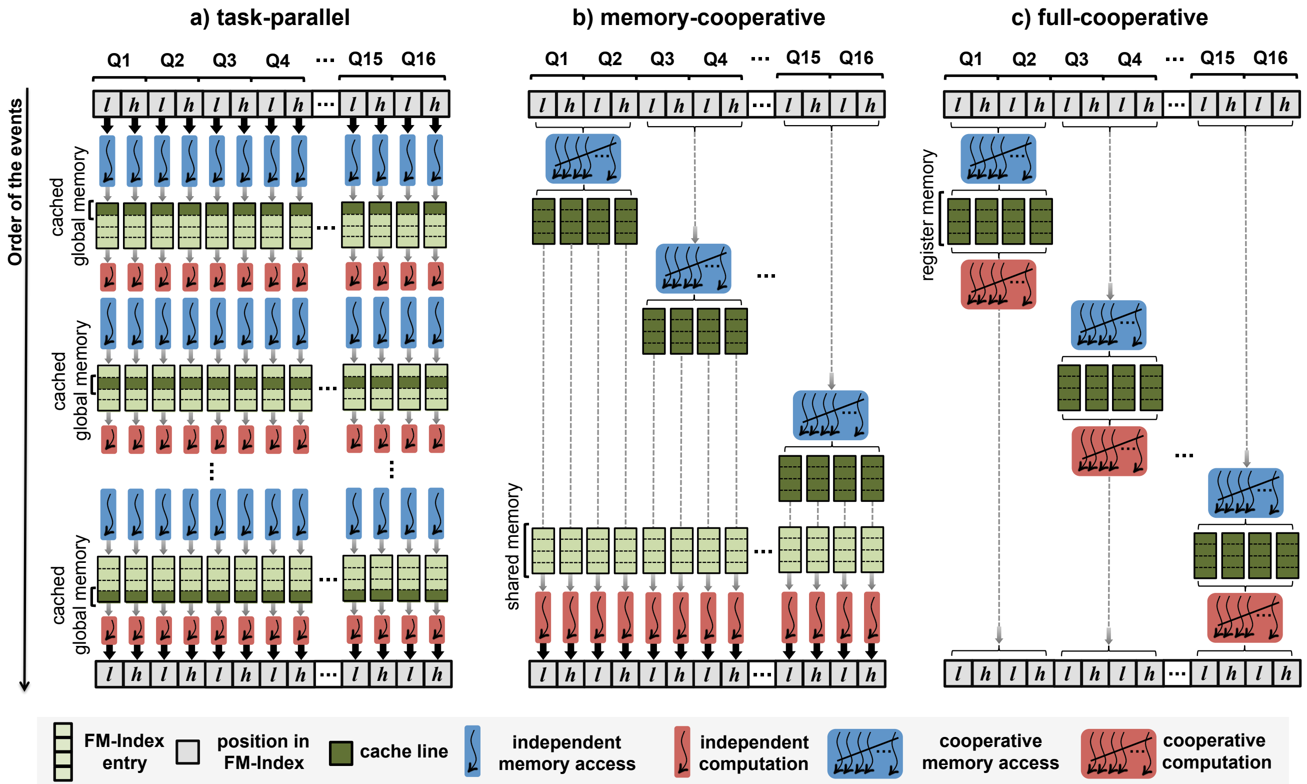


**Figure 6.1:** GPU parallelization alternatives: a) **task-parallel**: each thread performs independent LF operations; b) **memory-cooperative**: threads cooperate on reading data from index; and c) **full-cooperative**: threads cooperate both on reading data and on counting symbol occurrences. Each search step comprises 16 queries, and in this example we consider the case $d$=448. We depict all the 32 threads in a warp participating in the execution of 32 LF operations. Memory read operations are shown in blue, and computation on the data (basically, counting symbols) in red.

## 6.4 LF-Mapping: Thread-cooperative designs

A better parallel design is achieved if we promote thread cooperation. Higher memory performance can be attained when threads cooperate to perform their memory accesses. Also, cooperation on the same fine-grain task allows reducing the memory storage requirements per thread, and as a result making a better usage of the scarce on-chip storage resources, like processor registers and shared memory. Next we will present two cooperative strategies in an incremental way.

### 6.4.1 Memory oriented cooperative design

A large performance improvement can be obtained by using thread cooperation in order to coalesce multiple data requests of different distant blocks of memory. Figure 6.1.b shows the execution flow of a *memory-cooperative* design: the threads in the warp jointly request multiple complete index entries. The example depicted in the figure uses a warp of 4×8 threads to retrieve 4 complete entries (of 128 Bytes each) from memory with a single 16-Byte load instruction (the best performing option), for a total of 32×16= 512 Bytes. Every 8 consecutive threads in the warp cooperate to retrieve a memory block of 128 consecutive Bytes. The process iterates (8 times in the example) to copy the 32 entries from main memory into shared memory. Finally, each thread can efficiently access the shared memory to read the data corresponding to its entry and perform the LF-mapping operation, avoiding the costly non-coalesced accesses to the device external GDRAM and L2 cache.

The main drawback of the memory-cooperative scheme is that all the index entries read by a warp must fit simultaneously into shared memory. A relatively large sampling distance $d$ coupled with a $k$-step strategy puts pressure on the capacity of the shared memory, and may ultimately lead to a significant reduction of thread occupancy. Experiments shown in the next section reveal a severe performance degradation for index entries of 128 Bytes or larger.

### 6.4.2 Memory and compute oriented cooperative design

A much better approach to use the GPU resources efficiently is to reduce the working set of each thread (and hence of the whole application) by making threads also cooperate on the computational part of the algorithm (counting symbol occurrences and generating the output *SA* intervals), and not only on reading data. Figure 6.1.c presents the full-cooperative design: in the example shown there, the threads belonging to a warp cooperate to read 4 index

entries, and then process the entries to generate 4 outputs. This approach allows adjusting the working set of each thread to a given target size with the objective of maximising the actual GPU occupation. Comparing figures 6.1.b and 6.1.c we notice that the full-cooperative scheme must simultaneously keep only four index entries (512 Bytes) in fast memory instead of 32 (4096 Bytes). In other words, the granularity of the work assigned to each warp can be maintained constant even when the entry size is increased.

Since all cooperative operations proposed in our design are performed at the warp level, there is no need of costly explicit synchronisation: through *shuffle* instructions, Kepler and later CUDA architectures provide support for cooperating at the register level, which is faster and more efficient than cooperating using the shared memory.

In detail, based on *shuffle* instructions we implemented the following communication patterns:

1. Multicast among threads $l$ and $h$ values
2. Generate a single cooperative memory load
3. Multicast among threads the symbol that must be applied to an LF-mapping operation
4. Parallel symbol counting by all threads
5. Parallel reduction of partial counters by groups of threads
6. Parallel gathering of results.

A drawback of the full cooperative design with respect to a task parallel one is that it increases the amount of executed instructions. As explained before, however, the choice of diverting part of the vast amount of computational power provided by the GPUs into solutions designed to improve memory performance happens to pay off in terms of the overall computational efficiency of the implementation.

## 6.5 *k*-step FM-index and Alternate Counters on GPUs

As we have explained, the memory system of the GPU presents inefficiencies on scenarios with (1) large memory footprints and (2) random memory accesses to the data structures. The current chapter proposes a fine-grain parallelization that uses thread cooperation strategies to alleviate these inefficiencies. In that section we will show how combined with the form shown with the previous chapter proposal, k-step FM-index to increase the locality of the index thus reducing the number of random accesses. And with Alternate counting proposal to eliminate redundancies in the counters and thus reduce the size of the index. Both optimizations can be

applied to the actual proposal of parallel cooperation to obtain a higher performance on LF-Mapping primitive. The limited space in GPU main memory and the strong requirements of the TLBs presents the GPUs as a strong candidate to combine and exploit these optimisations.

The LF-mapping has been evaluated also using the k-step and Alternate Counters configuration. Its fine grain parallelization raises more complex challenges compared to the single-step FMI:

- The reduced number of operations required to count characters from the bitmaps with k-step is far more complex. Thus, the bigger the k value, the higher number of communications and synchronisations required to reduce operations. In addition, there is performance penalty due to fewer amount of work that can be done locally per thread. The number of instructions to execute is higher, incrementing the latency between dependent entry requests, which potentially prevents better bandwidth usage.
- Alternate counters introduce higher conditional code to process the k-FMI entry required to choose between left or right counter entries.
- Coalesced accesses are impacted when two coalesced accesses are available for each LF-mapping, preventing higher bandwidth use.

## 6.6 Experimentation

In this section we benchmark the execution on CPU and GPU platforms of the exact searches performed with our implementations of the FM-index. After presenting the experimental methodology, we describe the overall performance results and the index compression ratios achieved by using GPUs. We compare the performance of our proposals with the performance of the equivalent implementation provided by the Nvidia NVBIO library [11]. Then we present a detailed performance analysis of all considered solutions (task-parallel design, thread-cooperative design, 2-step approach), in order to identify the main architectural bottlenecks. In addition, we classify the application between memory or compute bounded and emphasize the key parameters involved in the different GPU systems analyzed. Finally we examine how performance and energetic efficiency vary with the GPU model.

### 6.6.1 Experimental Setup and Methodology

The experimentation platform is a heterogeneous CPU-GPU node. The CPU is a dual-socket Intel Xeon E5-2650, with eight 2-way hyperthreaded cores per socket, providing a memory bandwidth of 102 GB/s. Unless explicitly noted, the GPU results shown in the following figures were obtained on our best card, a Nvidia GTX Titan with 2688 Kepler CUDA cores and 6 GB of main memory providing up to 288 GB/s. In order to perform comparative GPU analysis (section 6.6.7) we also used a Kepler K20 card and a Maxwell GTX750Ti card. Table 6.1 gathers all platforms hardware specs as declared by the manufacturer. All the experimental codes for CPU were generated with GCC version 4.8, and codes for GPU with Nvidia compiler v6.0.

The input of our tests was a set of 10 million input DNA queries produced with widely used simulation tools ([10] and [40]) following standard procedures; input queries were searched in the human genome reference GRCh37. Before starting measurements we always made sure that the FM-index and the reads were already residing in the CPU or GPU memory. Performance results are expressed in terms of the number of query bases searched in the index per time unit.

Our multicore CPU implementation uses 16×2 threads (via OpenMP) to exploit hyper-threading, and memory access is optimised by using strategies introduced in Chapter 4. The sampling rate $d$ is set by default to 64 for the CPU. Our GPU implementations set the number of threads per block and the total number of thread blocks to values providing the highest performance.

The Nvidia NVBIO library [11] contains a suite of components to build new bioinformatic applications for massively parallel architectures. It offers methods for performing exact searches (via the `match` primitive) on a sampled FM-index both for GPU and CPU. On the GPU a different search will be executed by each of the threads in a kernel using a task-parallel approach. For the NVBIO code we have selected the best-performing FM-index configuration, with a sampling distance $d$=64 and a decoupled *SA* (partial FM-index). In some tests we also adapted the GPU implementation of NVBIO to control the thread occupancy, as explained below. The NVBIO library version 0.9.7 used in our experiments was compiled with release-mode settings.

The designed experimental methodology and result analysis are conducted by two types of experiments in order to identify specific memory inefficiencies:

- **Scalability measures when increasing the number of active threads:** We include a conditional statement that controls at run time the total number of threads per-

| | Architecture | Cores | Hardware threads | Frequency (Ghz) | Bandwitdh (GB/s) | Main Memory (GBytes) | TDP (watts) |
|---|---|---|---|---|---|---|---|
| Nvidia Kepler K20 | 2nd Kepler | 2496 | 26624 | 0.71 | 208 | 5 | 225 |
| Nvidia GTX Titan | 2nd Kepler | 2688 | 28672 | 0.84 | 288 | 6 | 250 |
| Nvidia GTX 750Ti | 1st Maxwell | 640 | 10240 | 1.02 | 88 | 2 | 60 |
| 2×Intel Xeon E5-2650 | Sandy Bridge | 16 | 32 | 2.00 | 102 | 256 | 190 |

**Table 6.1:** Hardware specifications of the experimentation platforms

forming index searches. This experiment allows classifying execution performance as latency-bound or computation-bound, and identifying memory cache pressures and low data re-utilization. Scalability issues can be determined in order to prevent under-utilization in large scale GPUs.

- **Performance measures without memory access penalties (*computation-only*):** We use the same query for all the search operations, forcing all threads to actually access the same piece of data in the local cache. The goal of the latter experiment is to estimate the performance of the computation part of the code isolated from the effect of memory performance. This experimentation allows to classify kernels as computational or memory bound.

### 6.6.2 Overall performance results

Figure 6.2 summarises the main results of our experiments, where we benchmark exact searches in the whole human genome (size is 3 Gbases). The results are presented as billions (Giga) of query bases processed per second, and correspond to the best-performing configuration for each implementation, both in the case of NVBIO and of our proposals (1-step, and 2-step with alternate counters). For comparison purposes we also include the timings achieved by the NVBIO code after the improvements we obtained by tweaking it: configuring NVBIO with a surprisingly low thread occupancy (9%) improves performance about 1.6× with respect to a configuration with maximal occupancy. We'll show next that this is due to its underlying task-parallel design.

The figure shows a clear speedup of our best proposals compared to the NVBIO library, both on the CPU (2.0×) and on the GPU (3.1× when considering the tweaked NVBIO code, and 4.9× versus the stock version of NVBIO distributed by NVIDIA).

The 2-step design outperforms the simple FM-index by 1.8× on the GPU and by 1.4×

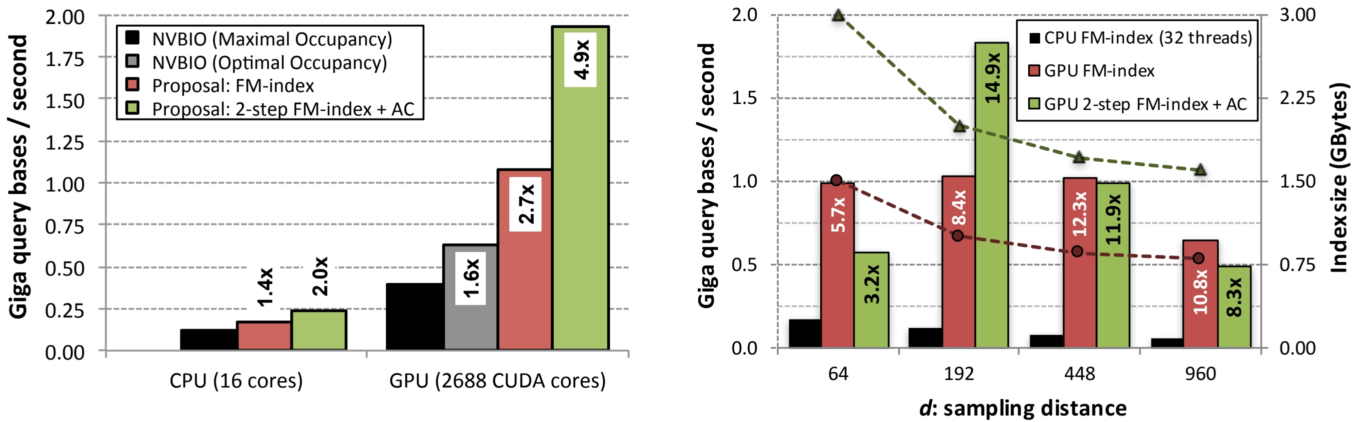


**Figure 6.2:** (a) Best FM-index search performance results compared to NVBIO library for CPU and GPU platforms; (b) CPU and GPU performance, measured in Giga bases processed per second.

on the CPU. As previously discussed, the moderate speedup on the CPU is due to the higher computational cost of the new design. However, such increased cost has a very limited impact on the GPU, where excess computational power is available to be used.

Finally, our best-performing implementation on the GPU (which in absolute terms delivers almost 2 Gigabases of query searched per second) is $8.1\times$ faster than our best implementation on the CPU. Interestingly, this speed-up is higher than the ratio of the raw bandwidths for sequential memory access delivered by the two platforms (which is around 3 times faster on the GPU than on the CPU). This fact confirms that our implementation strategies aimed at obtaining better performance for random memory accesses are particularly effective on the GPU.

### 6.6.3 FM-index compression features on GPUs

Figure 6.2.b shows the single-step FM-index performance of the task-parallel approach on the CPU and that of the full-cooperative approach on the GPU. In both cases, we show the effect of increasing the FM-index sampling distance $d$ from 64 to 960. Presented results correspond to the best-performing configuration for each sampling distance and for each system implementation. The figure shows a clear speed-up of the GPU version as compared to the multicore CPU version, in a range between $5.7\times$ and $12.3\times$ (corresponding in absolute terms to 0.7—1.2 Giga bases processed per second).

The plotted dot line correlates performance and FM-index size as the sampling distance is increased. While the CPU suffers from a steady performance deterioration due to the increased computation work associated to a larger $d$, the GPU tolerates the index compression without

any noticeable performance penalty up to *d*=960. Apart from enabling backward search on larger genomes, this parameter setup can be of special interest (1) when using low-end GPUs that provide smaller amounts of memory and (2) devices with big memory performance penalties accessing large data structures. Next we analyze in deep the performance issues of each proposed backward search version on GPUs.

### 6.6.4 Detailed performance analysis

In this section we analyse the inefficiencies of the task-parallel (section 6.3) and memory-cooperative strategies as opposed to the full-cooperative solution (section 7.1). Since the 2-step FM-index implementation exhibits a similar behaviour, we restrict our first analysis to the classical sampled FM-index. Figure 6.3.b compares the performances of the proposed GPU parallelization schemes, displaying the best results for each case. As explained before, the full-cooperative design outperforms the other two. The performance of the task-parallel scheme is only competitive for $d = 64$; similarly, the memory-cooperative scheme does not scale to $d > 192$.

#### Performance versus number of active threads

In figure 6.3.a we benchmark the full cooperative version, three different implementations of the task-parallel scheme, and the NVBIO implementation, showing their performance as a function of the number of threads used. The naive task-parallel versions assign two LF-mapping operations to each thread, both using 4-Byte ("naive") and 16-Byte memory accesses ("naive+16 Bytes"). The task-parallel approach labelled as "improved" uses 16-Byte memory loads and assigns a single LF-mapping operation per thread, which is a limited form of cooperation (see section 6.3). In all cases we use a sampling distance *d*=64, which is the most favourable for the task-parallel strategy.

The task-parallel schemes exhibit the problems anticipated in the previous section: performance first increases with more active threads, and then suddenly drops and flattens. The performance peak is located at some specific, relatively small number of active threads. The NVBIO implementation, which also uses a task-parallel approach, suffers from the same problem. On the other hand the performance behaviour of our cooperative version is very robust, scaling gracefully up to 4 thousand active threads. Eventually, as more threads are executed, a larger number of requests are issued that end up saturating the memory system.

The origin of the observed behaviour cannot lie in the GDRAM system, since searches in

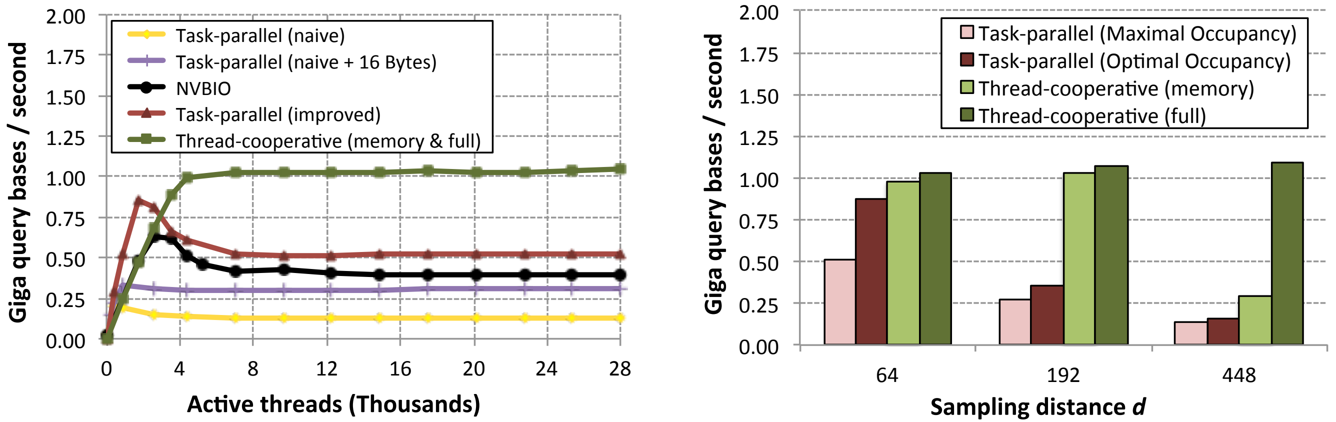


**Figure 6.3:** (a) Thread scalability for our proposals (sampled FM-index with $d$=64) and the Nvidia NVBIO implementation; (b) Performance of task-parallel (both with maximal and optimal thread occupancy) and thread-cooperative designs for increasing sampling distance $d$.

small indexes that fit into the L2 cache and require no data transfer from GDRAM show the same performance anomaly (data not shown). Instead, the reason is due to the fact that the data transfer mechanism between the L2 cache and the executing units is strongly optimised to favour spatial locality and coalesced accesses, and its scarce temporary storage becomes easily saturated when many threads compete to request data from the L2 cache. This is why the implementation issuing 16-Byte loads performs better than the one issuing 4-Byte loads. Consistently, the "improved" version achieves better results because it issues less load instructions.

### Performance versus sampling distance $d$

Figure 6.3.b shows the performance of the proposed parallelization schemes as a function of the sampling distance $d$, displaying the best results for each case. The task-parallel scheme is only competitive for $d = 64$; larger values of $d$ worsen the problem of non-coalesced accesses. Similarly, the memory-cooperative scheme does not scale to $d > 192$, as the shared memory capacity becomes exhausted by the requirements of too many threads. For $d = 448$, the GPU occupancy is 12% of the maximum number of active threads, and there is not enough parallelism to hide memory latencies. As expected, the full-cooperative design outperforms the other two in all cases.

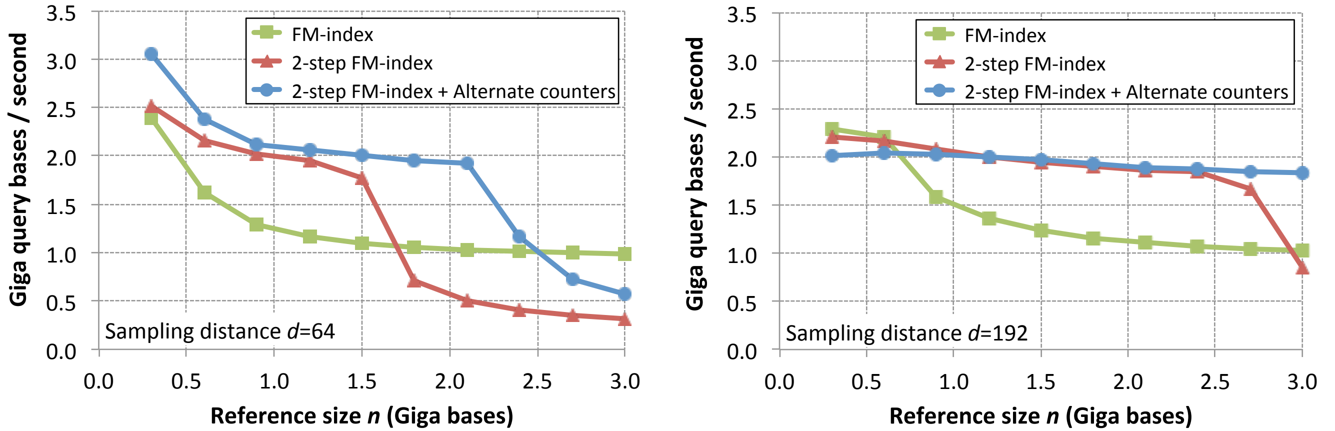


**Figure 6.4:** Performance effect of varying reference size $n$ and sampling distance $d$ on different indexing schemes

**Performance versus reference size**

The next two sections will analyse the empirical dependence of the FM-index on the size of the reference (which in turn stems from the empirical performance of random memory accesses on the GPU seen in section 3.4.2). We examine how the combination of our thread-cooperative design (section 7.1) and our $k$-step indexing strategy (section 5.2) can lead to the best performance results shown in section 7.3.2.

## 6.6.5 Classical *1*-step sampled FM-index

As mentioned in section 4.6, in theory the complexity of the FM-index search is independent of index size $n$. Quite to the contrary, figure 6.4 shows that in practice index size is a relevant parameter for the classical *1*-step sampled FM-index: when the index size grows, performance decreases for all the values of sampling distance $d$. This effect is due to the underlying performance of random memory accesses on GPU, and is a direct consequence of figure 3.11.

For large indexes ($n$ bigger than 1.5 Gbases) performance reaches saturation due to random memory accesses, leading to similar results for all the three compression ratios analysed. In other words, one can compress the index (as per figure 5.6) without any performance penalty. In fact, for some genome sizes a more compressed index also provides better performance: for instance, for $n$=0.7 Gbases, the choice $d$=192 is better than $d$=64. In contrast, for smaller indexes where $n$ is below 0.5 Gbases the performance is always better for smaller values of $d$. This happens because in this case the search is always computation-bound. The next section discusses this behaviour with a thorough analysis.

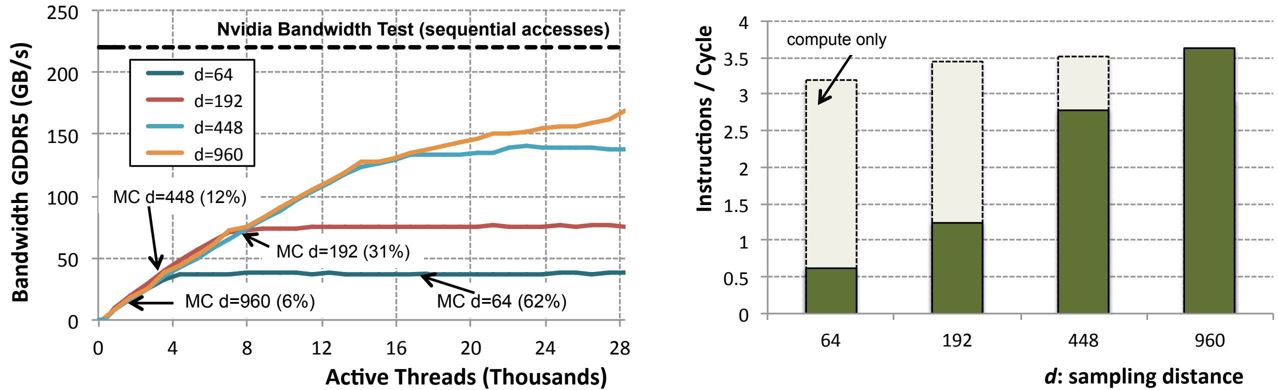


**Figure 6.5:** (a) Memory Bandwidth evolution for different *d* values; (b) Instructions per Cycle for full-cooperative scheme.

### Overlapping memory bandwidth and compute utilization

After verifying that the full-cooperative scheme scales gracefully in most scenarios, the next question is how well is exploiting the GPU available memory bandwidth and computation resources. For this purpose we depict in Figure 6.5 the effective memory bandwidth and the instructions per cycle (IPC) rate achieved by the proposal. For reference, Figure 6.5.a includes the peak empirical bandwidth of our target GPU for sequential accesses (220 GB/s) as measured by the Nvidia bandwidth test tool. As anticipated in section 3.4.2, pseudo-random memory access patterns, as expressed in our algorithm, are well below the peak bandwidth. Increasing the sampling distance creates more spatial locality (larger FM-index entries) and this is reflected in a higher effective bandwidth (two times more bandwidth as entry size is duplicated).

One can check if the performance patterns we observe are strictly due either to computation or to memory access limitations by executing a ”computation-only” benchmark. As explained at the beginning of this section, we construct this synthetic benchmark by forcing all the search operations (an all the running threads) to use the same query, and then access the same small amount of data in the FMindex. If the execution time of this ”computation-only” benchmark improves, it means that the application was bounded by memory. Figure 6.5.b confirms that performance is not completely bounded by computation until $d$=960. The shaded bars represent the IPC obtained when a computation-only version is executed, while the solid bar indicates the actual IPC. An IPC measured figure of 3.5 is very close to the IPC=4 value achieved by several computation-bound applications published by NVidia. For very large entries ($d$=960), the performance limit is not memory bandwidth anymore but the amount of computation. The application has to execute more instructions per FM-index entry,

including the overhead due to thread cooperation, while reading a large entry has almost the same performance cost as reading a smaller entry (because of the characteristic access pattern of the algorithm).

### 6.6.6 *2*-step FM-index and alternate counters

The *k*-step strategy trades reading less blocks for reading bigger blocks, and also benefits from the first performance principle seen in section 3.4.2: large blocks are free for random-access memory patterns. Its drawback, though, is a larger memory footprint that can be detrimental if the index size goes beyond the empirical limit of 2.3 GB (see Figure 3.11). For instance, this is what happens in the case of the human genome when a 2-step approach with distances *d*=64 or 192 is used: the indexes thus generated will require 4.5 and 2.5 GB, respectively. However, the use of alternate counters reduces the index sizes to 3 GB and 2 GB, respectively, thus restoring the efficiency of the choice *k*=2, *d*=192.

Figure 6.4 allows us to compare performance for different reference sizes, sampling distances and FM-index configurations. According to our previous observation, the performance drop of the 2-step configurations occurs when the index size exceeds the 2.3 GB limit. In addition, the configuration *k*=2, *d*=192 and alternate counters turns out to be the best option for references larger than 1 Gbases and smaller than 3.5 Gbases. For bigger references, the 2-step design generates an index that is too large. For references smaller than 1 Gbases the GPU provides higher memory bandwidths and the execution may become computation-bound, similar to what happens for the case *k*=1; the most effective solution for such reference sizes is to reduce the compression rate in order to reduce the computational burden.

#### Computational cost

Table 6.2 measures the computational cost of our indexing schemes. The number of executed instructions collected from our benchmarks confirms that indeed the computational cost of the cooperative design on the GPU is significantly (2.5×) higher than that of the task-parallel design. Also, the cost of compressing the FM-index (higher *d*) grows as expected from the definition of section 4.6: doubling the entry size doubles the number of instructions (and the amount of Bytes read from memory). Finally, the last two rows of table 6.2 show that with respect to the 1-step strategy the 2-step strategy incurs only a moderate computational overhead (10% to 18%), which is negligible for large indexes but can be detrimental for short indexes. Overall, the complete lack of correlation between the entries of this table and the

| Sampling distance | $d$=64 | $d$=192 | $d$=448 |
|---|---|---|---|
| Task-parallel FM-index | 3.08 | 6.15 | 12.14 |
| Full-cooperative FM-index | 7.68 | 15.63 | 31.43 |
| 2-step FM-index | 8.49 | 17.70 | 35.10 |
| 2-step FM-index + alt. counters | 8.89 | 17.10 | 37.10 |

**Table 6.2:** Warp instructions executed per query base

corresponding performance values confirms the predominant role played by memory effect when exact searches are performed on the FM-index.

### 6.6.7 Comparison of GPU architectures

In this section we want to describe how the performance of the proposed algorithms varies on three different GPUs: two Kepler cards (GTX Titan and K20c) and a recent Maxwell card (GTX 750Ti).

#### Analysis of overlapping bandwidth and compute

First of all, in figure 6.6.b we extend figure 3.11 and compare the random memory access bandwidth of the three cards. Quite surprisingly the largest bandwidth is provided by the commodity GTX Titan; the professional Tesla K20c shows a similar performance profile, but with about 30% less performance. In particular, at 2.3 GB the two cards share the same sweet spot that maximises the product of bandwidth and memory footprint. The low profile Maxwell card gets its maximum throughput with 1GB memory footprints. As its cost and power consumption are only a fraction of those of the other professional GPUs, this card can still be appropriate for small genomes, or to process bigger genomes on multiple cards.

This final subsection compares the performance achieved by the FM-index search algorithm on different Kepler GPU cards and the recent Maxwell GTX 750Ti. We expect that the memory performance of each GPU architecture will be the major factor to determine the overall performance. We also expect differences in the point where the cooperative scheme becomes computation-bound, which will be correlated with the ratio of computation and memory bandwidth offered by each GPU. Performance results are shown in Figure 6.6.a.

The right side of the chart shows the case where the performance of all the GPUs is computation-bound. In this case, the performance achieved correlates very well with the

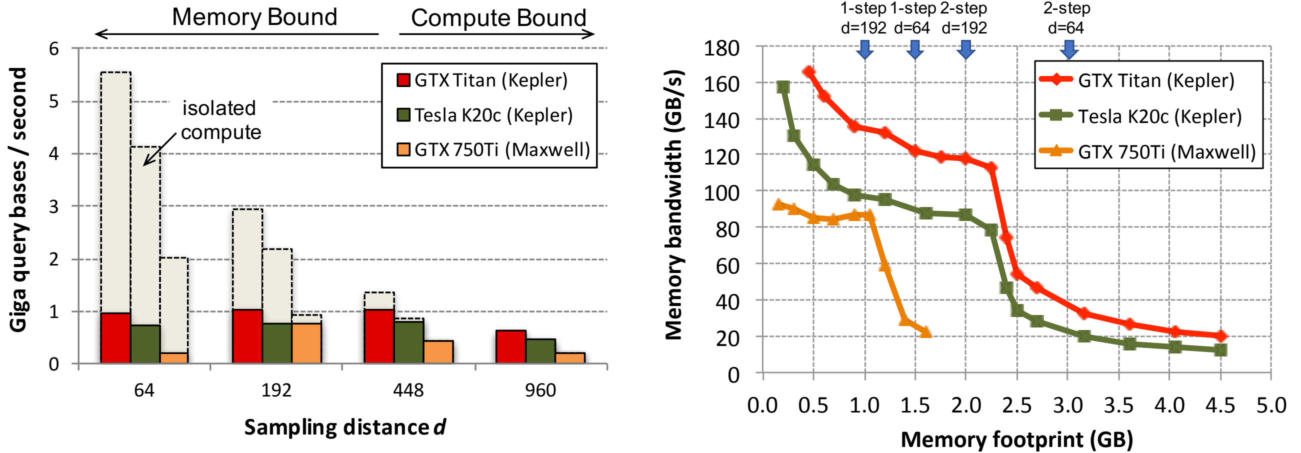


**Figure 6.6:** (a) Comparison among different GPUs (Full Cooperative); (b) Performance of random memory accesses for different GPUs (index entry size is 128 Bytes).

potential performance offered by each GPU. Notice that the low-end Maxwell GPU becomes computation-bound before all Kepler GPUs, for $d$=448.

The left-side of the chart shows the case where the performance of all the GPUs is memory-bound. In this case, performance does not clearly follow the potential memory bandwidth offered by GPUs, which is measured for sequential memory accesses. For random memory access patterns, memory performance is not as different on the range of GPUs analysed as could be inferred from the published bandwidth figures. An interesting case happens for $d$=192, where a Maxwell GTX 750ti performs like a Tesla K20c (with 0.42$\times$ the potential bandwidth).

For pseudo random memory access algorithms where the performance is far from computational capabilities offered by hardware, the relatively low cost and low energy consumption offered by Maxwell card provides a good target platform. That is, we can get a relatively good performance with a small fraction of the cost and energy consumption of higher-end GPU cards as the Tesla K20c or the GTX Titan.

### Analysis performance and energy efficient

In figure 6.7.a we compare the performance of the proposed algorithms for the case of the human genome. The Titan and K20c GPUs show similar performance profiles for the different algorithms; since the search algorithm is memory-bound, the observed throughput reflects well the memory bandwidth profile of each GPU depicted in figure 6.6.b. On the other hand, on the GTX 750Ti the human genome can be indexed only with $k$=1 due to the smaller

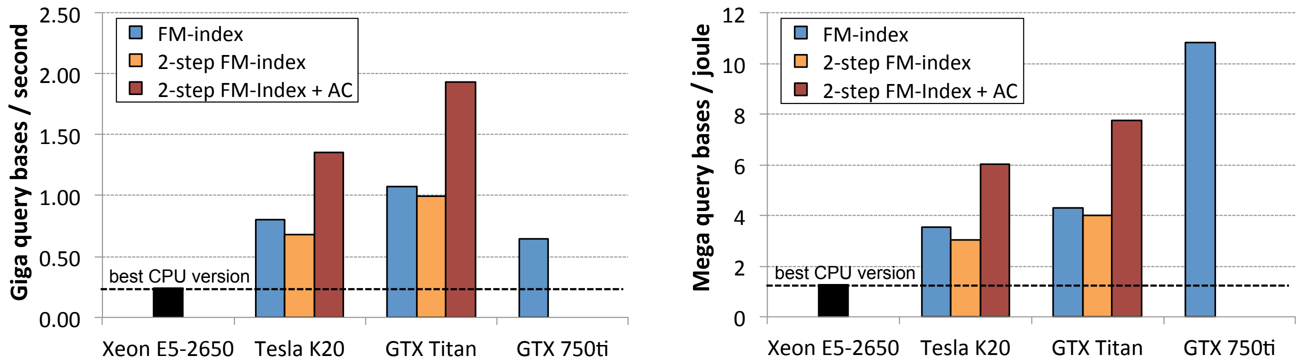


**Figure 6.7:** Performance comparison (a) and energetic efficiency (b) of our thread-cooperative strategy on different CPU/GPU architectures

memory size available. Despite this, the performance is still quite good (only 40% worse than that of the $k$=1 version on the Titan, and about 2.5$\times$ worse than that of the best 2-step version on the Titan). However, when comparing the nominal energetic efficiencies (figure 6.7 right panel, obtained from the performances and table 6.1) one notes that the GTX 750Ti stands out among all other platforms in terms of the number of queried bases/joule. Compared with the CPU, the GTX 750Ti has 8.5$\times$ better energetic efficiency, while still providing a 2.6$\times$ better performance. All the GPUs considered in this study are far more energetically efficient than the CPU (from 4.8$\times$ to 8.5$\times$ if the best implementations are considered).

## 6.7 Conclusions

Technological improvements in memory performance are mostly achieved by incrementing the size of the data transfer bursts between main memory and the CPU/GPU. While this feature can greatly improve the performance of algorithms accessing large blocks of sequential data, it is neutral for algorithms requesting relatively small data blocks spread across distant random locations. In fact we are expecting to see that, in terms of efficiency, pseudo-random memory access patterns like those shown by straightforward FM-index implementations will steadily lag behind sequential access patterns even in upcoming next-generation memory systems. In such a scenario, the performance cost is determined by the total number of blocks accessed and not by the amount of data accessed. Therefore, we must favour algorithmic variations that access similar amounts of data but concentrated on less and bigger data blocks, even at the expense of more computation. This is precisely what our $k$-step FM-indexing strategy does: it trades reading less blocks for reading bigger blocks.

Current GPUs (and CPUs) increase their memory bandwidth capabilities with wider data access pipelines. While this feature can be used to greatly improve the performance of algorithms accessing memory in a sequential fashion, it is not very useful for algorithms exhibiting pseudo-random access patterns, like the one analysed in this thesis. We propose a methodology that allowed an algorithm with a pseudo-random data access pattern to find opportunities for extra computation. While the index size cannot be reduced too much on the CPU due to the excessive computational costs entailed, the same operation has a very limited impact on the GPU, where excess computational power is available to be used. This way, a new design can be devised to improve the performance and, in some conditions, overcome memory bandwidth bounds.

The working set granularity also plays a crucial role in GPU performance. In fact, a simple task-parallel approach to FM-indexing is inefficient because the addition of more threads will turn into a larger and larger working set. However, when threads cooperate on a single task the working set is distributed among the cooperating threads. This allows us to efficiently process the bigger index entries produced by the $k$-step strategy. The increase in computational cost due to cooperation has a limited impact on the GPU, where excess computational power is available to be used, and overall our solution turns out to be successfully trading more work for less memory accesses.

The combination of a compact, size-tunable FM-index, and a novel thread-cooperative approach, can be used to tip the algorithmic bottleneck away from memory access. We present an implementation that is able to process about 2 Gbases of queries per second on our test platform, being about 8$\times$ faster than a comparable multi-core CPU version, and about 3$\times$ to 5$\times$ faster than the FM-index implementation on the GPU provided by the recently announced Nvidia NVBIO bioinformatics library.

While the profile that correlates index footprint size and memory bandwidth for random accesses varies on different GPUs, it will anyway be one of the strongest determinants of the performance of our best FM-index search implementation. Hence it will be necessary to adapt the indexing scheme to the target GPU and the reference genome of interest. Luckily, in our framework performance can be easily optimized by selecting suitable values for parameters $k$ and $d$. We anticipate that this feature of our algorithm is going to be more and more relevant for the forthcoming GPU systems. Thanks to our results, one might use a few cheap and energy-effective low-end GPUs to replace a high-end GPU.

# 7

# GEM3: approximate pattern search in a mapper GPU

*"We've always defined ourselves by the ability to overcome the impossible"*

Interstellar - **Christopher Nolan**

This chapter reviews the strategies for the approximate text search problem and its two stages: seed generation and extend (position decoding). Then, we propose a design to accelerate a re-engineered version of the GEM mapper. Finally, GPU performance experimentation and comparison with different proposals is carried out.

The problem of read mapping in the context of a production application using real genomic data is complicated by the existence of DNA mutations and sequencing errors, and requires the development of efficient, approximate text searching algorithms. All these algorithms are built upon the fundamental block that we have described in Chapters 5 and 6, namely, the LF-mapping primitive. This chapter reviews the most frequently used strategies for the approximate text search problem and how they are introduced in a mapper as two typical stages: (1) generation of seeds and (2) position decoding. Then, we propose a design that leverages our previous GPU-parallel algorithms to accelerate a re-engineered version of the GEM mapper. Finally, performance experimentation and comparison with different proposals is carried out on GPU environments.

## 7.1 Search by filtering and seed selection algorithms

A read mapper must cope with variations against a reference genome in order to cover sequencing errors and the genomic mutations present in the input samples. The *approximate string matching* problem is defined as the problem of finding the position of a sequenced read into a genome text with the minimum number of error events. This search procedure can be organised by stratas, where each strata of the search is defined as the collection of all occurrences of the read on the genome text that match with an specific read error distance. For example, the first strata corresponds to all the exact occurrences of the input text (with 0 error events), the second strata corresponds to all occurrences with 1 event error and so on. A search result is considered complete with $\varepsilon + 1$ error events when it reports all the occurrences included on the first $\varepsilon + 1$ stratas.

The approximate string matching problem can be solved by using the basic exact matching primitives revisited in Chapter 4, and all the proposals from Chapters 5 and 6 can straightforward applied. Seed selection is the core algorithm to describe efficient exact search compositions. In the current chapter we will briefly review the most relevant seed selection algorithms and the necessary concepts to understand the chapter contributions. A computational comparison between the most basic strategies as a motivation for the Adaptative solutions optimized for accelerators. Finally, benchmarking is carried out for CPU and GPU systems.

## 7.2 Search by Filtration

Due to the unfeasible amount of computation required for large texts, the approximate string matching problem is typically solved using offline string search techniques. Most competitive read mapping applications combine text indexation and exact search methods to emulate approximate string matching. Some well-known methods are search-tree search with branch pruning [41] and search by filtration [25]. The low computational cost and the reasonable accuracy offered by filtration methods spread a wave of new modern mappers applying just this method or hybrid heuristics that combining both.

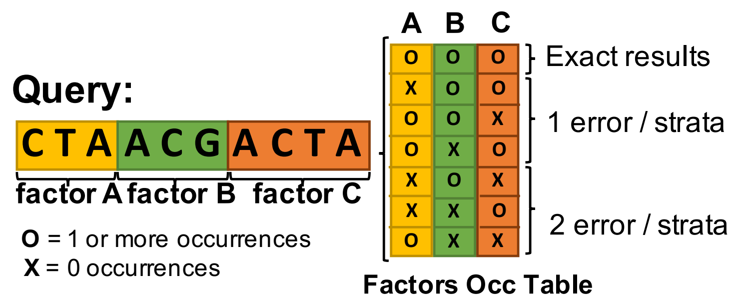


**Figure 7.1:** Example of query division in 3 seeds (factors), which shows that completeness is granted to 2 errors.

### 7.2.1 Seed generation

Search by filtration methods are based on the premise that an exact substring read occurrence (typically called seed) could be part of the desired read approximate string matching solution. Filtration exploits this idea decomposing the mapping problem in two main steps: seed generation and candidates verification. Seed generation is in charge of read mapping sensitivity: this step proceeds by locating all the possible genome regions where a read could map correctly with certain percentage of error. And on the other hand, candidates verification classifies and filters-out the genome regions reported by the seed generation stage, based on the homology between region and read. Literature shows different seed generation proposals to improve the trade-off between accuracy and computation.

### 7.2.2 Complete searches and pidgeonhole principle

The completeness in the search results is really important to provide robust and confident data accuracy on mappers. In order to ensure the completeness on the results, the Pidgeonhole

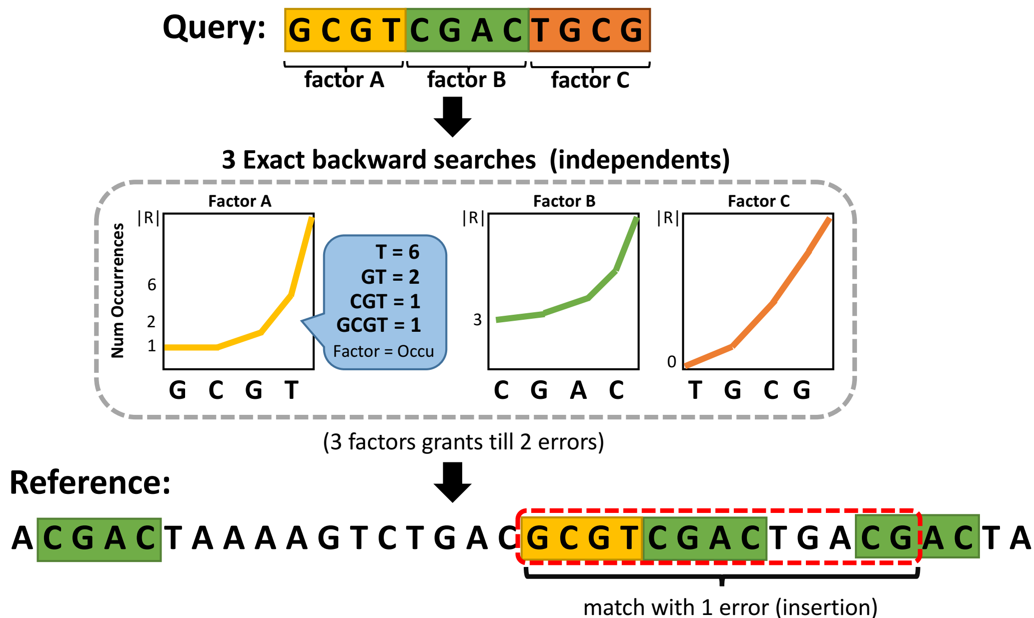


**Figure 7.2:** FM-index: Static seed (factor) selection

principle (also known as Dirichlet's drawer principle) can be combined with an specific selection of seeds (also known as non-overlapping seeds) from the read. The Pidgeonhole principle determines that in order to guarantee that we find all the possible locations of a read that matches with $\varepsilon$ error events, the read has to be divided into $\varepsilon + 1$ error-free substrings called seeds (or non-overlapping seeds). Error events can be insertion, gaps or mismatches. By definition, generating a higher number of seeds implies that more stratas are explored and then a higher sensitivity for the approximate search. The problem of generating more seeds is that the amount of required computation grows. Several proposals explore how to improve the trade-off between accuracy and computational cost with a more clever selection of seeds 7.1.

When there are errors in a read, the read can still be correctly mapped as long as one seed of the read exists that is error free. The error-free seed can be obtained by breaking the read into many non-overlapping seeds; in general, to tolerate e errors, a read is divided into e + 1 seeds, and based on the Pigeonhole Principle, at least one seed will be error free [42].

### 7.2.3 Static vs Adaptative seed selection algorithms

This section will introduce two algorithms for exact sequence search: (1) *Static seed search* and 2) *Adaptative seed search*. As described at the beginning of the current chapter, the use of a specific setting of exact search primitives allows to create an algorithm that warrants

the approximate text search up until a specific error threshold. On top of the threshold, it is possible that the algorithm returns some additional solutions that have a higher error. Both algorithms described in this section are used for the selection of seeds. It is interesting to point that this type of selection can report a different number of candidates. A HPC sequence alignment program has the goal of reducing the number of candidates while maintaining the warrantied degree of the *error* search.

Both the Static seed and the Adaptative seed algorithms are described in the pseudocodes 5 and 6. They are presented in pseudocode and fully serial to simplify the explanations. The CUDA implementations included in *GEM-cutter* use all the contributions shown in Chapters 4, 5 and 6 as extension to this work. All the CUDA implementations can be reviewed at https://github.com/achacond/gem-cutter.

The pseudocode 5 refers to the *static seed selection* algorithm. As it can be seen in the pseudocode 5, this algorithm divides the read initially in e+1 candidates (not necessarily equidistant), being e the maximum error that guarantees the search. Once segmented, the search can be performed independently between them and fully parallel. This is one of the advantages of a static seed. However, there is no mechanism to controlling the reported number of candidates, or to adjust the work to the error of the original read. Possible improvement are shown in the pseudocode 5 (1) at line 22, where all the seeds with a value higher than a certain threshold $occThreshold$ are discarded. In this way, the search cannot be guaranted and (2) at line 13 on which there are implemented an early exit condition, if there are not more candidates on that seed we can stop the search.

The pseudocode 6 refers to the seed Adaptative algorithm. The *adaptative search selection* considers (1) the content of the read and (2) its mappability to the genome, to decide which seeds extract from the read. The partition for the seeds could contain variable seed sizes. The algorithmic core of this Adaptative seed selection stage is based on a greedy approach, where a FM-index backward-search is performed; for each LF-mapping (L,R intervals) the intervals are evaluated (line 19) which provides the current number of seeds detected. The threshold will be leading the seed extraction position, deciding at which position of the read we have to extract the seed (line 13). If the threshold ($occThreshold$) is below, it means that seed is promising and we will try to optimise it (line 23). The optimisation step, tries to extend the seed meanwhile the extension is promising (line 9, pseudocode 7), there is a limit of extension $maxStep$ and $occShrink$ defines the suitability of the extension by updating $occThreshold$ with $occShrink$ reduction.

### 7.2.4 Accelerating seed selection with multi-level LUT

In this section we will briefly introduce the use of a LUT (memorisation table) for the acceleration of the FM-index backward search. This table can be applied to any possible search based on the backward search, including the Adaptative and static algorithm for seed selection. This can be done because the index is generated offline and doesn't change in run time. The table stores all the possible SA intervals for a subset of a suffix from the read. Then, we can generate a table for all the possible intervals for any seeds with an specific size or smaller. That said, a table it grows exponentially, so it is an strategy that doesn't scale with the read size. Specifically for the Adaptative search, we can store the last valid interval (compared to $occThreashold$) on the MSB of each number, that points to an specific level on the LUT. Using this optimisation, we can reduce the number of accesses to the LUT just to 1, instead of a number of accesses being dependent on the size of the seed.

## 7.3 Experimentation

In this section we benchmark the execution of GPU platforms running the exact searches performed with our implementations of the FM-index. After presenting the experimental methodology, we describe the overall performance results and the index compression ratios achieved by using GPUs. We compare the performance of our proposals with the performance of the equivalent implementation. Then we present a detailed performance analysis of all considered solutions, in order to identify the main architectural bottlenecks. In addition, we classify the application between memory or compute bounded and emphasise the key parameters involved in the different GPU systems analysed.

### 7.3.1 Experimental Setup and Methodology

The experimentation platform is a heterogeneous CPU-GPU node. The CPU is a dual-socket Intel Xeon E5-2650, with eight 2-way hyperthreaded cores per socket, providing a memory bandwidth of 102 GB/s. The GPU results shown in the following figures were obtained with an Nvidia GTX Titan with 2688 Kepler CUDA cores and 6 GB of main memory providing up to 288 GB/s. In order to perform comparative GPU analysis. All the experimental codes for CPU were generated with GCC version 4.8, and codes for GPU with Nvidia compiler v6.0.

The input of our tests was a set of 10 million input DNA queries produced with widely used simulation tools ([10] and [40]) following standard procedures; input queries were

searched in the human genome reference GRCh37. Before starting measurements we always made sure that the FM-index and the reads were already residing in the CPU or GPU memory. Performance results are expressed in terms of the number of query bases searched in the index per time unit.

### 7.3.2 Overall performance results

For the experimentation, the performance of three different algorithms was benchmarked on GPU systems:

- Static seeding (as described in section 7.2.3 of this chapter)
- Adaptative seeding (also described in section 7.2.3 of this chapter).
- Adaptative seeding with an additional LUT in order to accelerate performance.

Three different experiments were then designed comparing the before mentioned algorithms. The first experimentation evaluated the performance in Giga LF-mappings/second; while the second measured the performance in Gigabases/second. Finally, the work performed was evaluated in the third experimentation by measuring the LF-mapping operations/base. The results of the three settings can be found summarised in Figure 7.3.

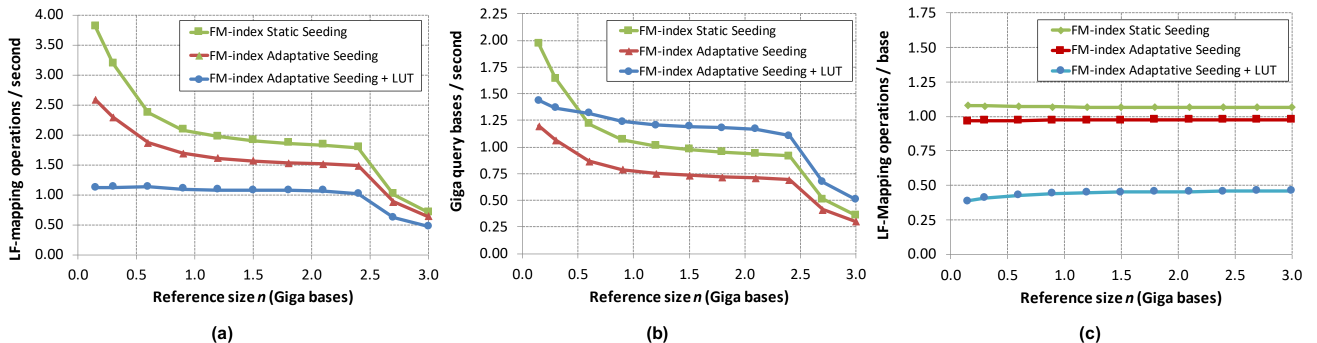


**Figure 7.3:** GPU performance for Static and Adaptative seeding selection

More specifically, Figure 7.3.a shows that the algorithm with a greater performance in LF mappings/second is the *Static Seeding*, which can reach 3.8 Giga LF-mappings/second in short references. The performance is followed by the *Adaptative Seeding* which ranges from 0.6 to 2.6 Giga LF-mappings/second. The *Adaptative Seeding* with LUT algorithm, however, only reaches performances between 0.5 and 1.1 Giga LF-mappings/second. In all algorithms,

however, we can observe that the performance will drop in all cases when the size of the index is reaching around 2.5 Gigabases from Human Genome.

Figure 7.3.b shows the second experimentation, in which performance is measured by Gigabases/second. The proposal of this new metric is experimentation in order to normalise the data and allow better comparisons between algorithms. This is due to the fact that the chosen algorithms do not generate the same amount of LF-mapping operations when processing the same read. The Figure shows that the Adaptative seeding with LUT has a grater performance that the rest of the algorithms, except for very small data of the human genome (approximately 0.5 Gigabases). Therefore, by applying this normalisation, it can be concluded that the Adaptative seeding algorithm with LUT can reach a 66% performance improvement compared to the other algorithms. For smaller sizes of the genome, the static seeding algorithm obtains a better performance, reaching the 2 Gigabases/second; followed by the Adaptative with LUT, which obtains a performance in small sizes of 0.5 to 1.40 Gigabases / second. The Adaptative seeding reaches a performance between 0.3 and 1.50 Gigabases/second.

Figure 7.3.c shows the third experimentation, in which the amount of work performed by each algorithm is calculated. Both the static seed and the Adaptative seed algorithms perform a similar number of operations per base, which is stable independently of the size of the reference. On the other hand, the Adaptative seed with LUT algorithm performs a higher amount of work when the sequence is larger. We can see that the Adaptative seed with LUT optimisation is effective and allows a reduction of the work performed by the algorithm of between 2x and 3x. On the other hand, the static seed and the Adaptative seed algorithms perform a very similar amount of work, which is not unexpected as both algorithms have a linear complexity.

## 7.4 Conclusions

The current chapter explores different algorithmic strategies to optimise the text search problem, including static seeding, Adaptative seeding, and the use of a LUT added to the Adaptative seeding in order to improve the performance. A seed selection algorithm try to reduce the number of reported candidates, and to maximise the covered error in the search.

We noticed that the Adaptative seed selection algorithm has a better trade-off in between the number of reported candidates and the errors that can cover in the search. The Adaptative

strategy analyses the content of the read to perform a more efficient selection, however this selection is dependent of each of the previous generated LF-mappings intervals. This transforms the algorithm to be pure serial, so it can not be divided in multiple searches or be parallelised as the static seed selection. We notice that the GPU version of the Adaptative seed selection, the performance its being heavily penalized by the thread divergence from our experimentation.

The results show that the Adaptative seeding algorithm with LUT can reach up to 66% improved performance compared to the other algorithms. In addition, we can identify that the reference size has an impact to the LUT table performance, as larger is the genome, less effective is the LUT table.

**Algorithm 5:** Static seeds search algorithm.

**input** : $F$: FM-index, $Q$: query, *sizeSeed*: max seed size, *maxSeeds*: max number of seeds, *occThreshold*: max occurrences per seed

**output** : *numSeeds*: number of generated seeds, *regionsList*: list containing the generated seeds

```
1  Function static_search(F, Q, sizeSeed, maxSeeds, occThreshold)
2      // Declarations relative to the search
3      idBase ← 0
4      idSeed ← 0
5      // Locate and generate the seeds
6      while (idBase < Q.size) && (idSeed < maxSeeds) do
7          // Search initializations
8          seed.l ← 0
9          seed.h ← F.size
10         seed.end ← Q.size − idBase − 1
11         endBase ← min(idBase + sizeSeed, Q.size)
12         // Searching for the next seed
13         while idBase < endBase do
14             // Advance step FMI reducing the interval search
15             if (seed.h ≠ seed.l) then
16                 base ← Q.str[Q.size − idBase − 1]
17                 seed.l ← LF(F, base, seed.l)
18                 seed.h ← LF(F, base, seed.h)
19                 seed.start ← Q.size − idBase − 1
20             idBase++
21         // Evaluate current seed suitability
22         if (seed.h − seed.l) <= occThreshold then
23             // Save generated seed (SA intervals and Q positions)
24             seedsList[idSeed] ← seed
25             idSeed++
26     numSeeds ← idSeed
27     return (numSeeds, seedsList)
```

**Algorithm 6:** Adaptative search algorithm.

**input** : $F$: FM-index, $Q$: query, $maxSeeds$: max number of seeds, $maxSteps$: max optimization LF steps, $occThreshold$: max ocurrences per seed, $occShrink$: expected occurrence reduction between LF steps

**output** : $numSeeds$: number of generated seeds, $seedsList$: list containing the generated seeds

```
1  Function adaptative_search(F, Q, maxSeed, maxSteps, occThreshold, occShrink)
2      // Declarations relative to the search
3      idBase ← 0
4      idSeed ← 0
5      // Extracts and locates each seed
6      while (idSeed < maxSeeds) && (idBase < Q.size) do
7          // Search initializations
8          seed.l ← 0
9          seed.h ← F.size
10         occ ← seed.h − seed.l
11         seed.end ← Q.size − idBase − 1
12         // Searching for the next seed
13         while (occ > occThreshold) && (idBase < Q.size) do
14             // Advance step FMI reducing the interval search
15             base ← Q.str[Q.size − idBase − 1]
16             seed.l ← LF(F, base, seed.l)
17             seed.h ← LF(F, base, seed.h)
18             seed.start ← Q.size − idBase − 1
19             occ ← seed.h − seed.l
20             idBase++
21         // Evaluate current generated seed (discard or optimize)
22         if occ <= occThreshold then
23             {seed, idBase} ← as_optimize_steps(F, Q, seed, idBase, maxSteps, occShrink)
24             // Save extracted region (SA intervals and Q positions)
25             seedsList[idSeed] ← seed
26             idSeed++
27     numSeeds ← idSeed
28     return (numseeds, seedsList)
```

**Algorithm 7:** Optimization process algorithm for the Adaptative search.

**input** : $F$: FM-index, $Q$: query, *seed*: seed to optimize, *idBase*: query position to start optimization, *maxSteps*: max optimization LF steps, *occShrink*: expected occurrence reduction between LF steps

**output** : *seed*: optimized seed, *idBase*: updated query position to continue searching

```
1  Function as_optimize_steps(F, Q, seed, idBase, maxSteps, occShrink)
2      // Extension initialization
3      l ← seed.l
4      h ← seed.h
5      occ ← h − l
6      occThreshold ← occ/occShrink
7      endBase ← min(idBase + maxSteps, Q.size)
8      // Last steps extension (exploration for consecutive maxSteps bases)
9      while (idBase < endBase) && (occ ≠ 0) do
10         // Advance step FMI reducing the interval search
11         base ← Q.str[Q.size − idBase − 1]
12         l ← LF(F, base, l)
13         h ← LF(F, base, h)
14         occ ← h − l
15         // Update seed information
16         if occ < occThreshold then
17             seed.l ← l
18             seed.h ← h
19             seed.start ← Q.size − idBase − 1
20             occThreshold ← occ
21         occThreshold ← occThreshold/occShrink
22         idBase++
23     idBase ← Q.size − seed.start
24     return (seed, idBase)
```

# 8

# Text filtering building blocks

*"Nothing in life is to be feared, it is only to be understood. Now it is time to know more, so that we can fear less"*

**Marie Curie**

On this chapter, the computing Levenshtein distance is first described to afterwards introduce the Myers' bit-parallel algorithm. Sections 8.3 and 8.4 are focused on Task parallel designs and thread cooperative approach, respectively. The optimisation details are found in section 8.5. Finally, section 8.6 is focused in the experimentation, with a discussion of the results found in section 8.7

Approximate string matching is very important in computational biology; where the fast computation of string distance is essential. Myers' bit-parallel algorithm improves the dynamic programming approach to Levenshtein distance computation, offering a competitive performance on CPUs. The main challenge when designing efficient GPU implementations is to expose enough SIMD parallelism while keeping a small working set for each thread.

In this work we implement and optimise a CUDA version of Myers' algorithm suitable to be used as a building block for DNA sequence alignment. We achieve high efficiency by a cooperative parallelisation strategy for (1) very-long integer addition and shift operations, and (2) several simultaneous pattern matching tasks. In addition, we explore the impact obtained when using features specific to the Kepler architecture. Our results show an overall performance of the order of tera cells updates per second using a single high-end Nvidia GPU, and factor speedups in excess of $20\times$ with respect to a sixteen-core, non-vectorised CPU implementation.

## 8.1 Introduction

Recent sequence alignment software tools, like BWA [41] or GEM [25], use a two-step alignment strategy. The first step (based on a seeded search in the case of BWA, or on filtration in the case of GEM) extracts substrings from the query (or *read*); such substrings are searched in the reference genome (which has been previously turned into an indexed form allowing fast pattern matching, for instance an FM-index [35]) generating candidate match positions. The second step uses online approximate string matching [43] to verify the similarity between the query and the region adjacent to every candidate position; it returns as valid matches the regions that differ from the query, in terms of some string distance, by less than a value specified by the user.

In the context of biological sequence alignment one often employs *Levenshtein distance*, i.e. the minimum number of *edit operations* needed to transform the query into the match. Each operation can be either a substitution, or an insertion, or a deletion of a single character. Levenshtein distance is typically evaluated in terms of *dynamic programming* (DP) [44], which casts the problem into the computation of (a subset of) a suitable integer-valued matrix. Improving upon a vast previous literature, Myers [45] devises an algorithm to compute the DP matrix using bit-wise operations; each multi-bit operation can handle several matrix cells simultaneously, thus reducing both the total computational work and memory storage requirements.

A typical read-mapping job turns billions of query sequences into tens of billions of candidate regions. This provides plenty of task-level parallelism in the form of multiple DP matrix calculations. While *inter-task parallelism* is a simple way of benefiting from the MIMD and H/W multithreading capabilities of GPUs, however, it is not adequate to efficiently exploit their SIMD/vector potential. In addition, running a pattern matching task per thread would not scale with the query size, due to the impossibility of fitting the working sets of the threads into available on-chip GPU memory even for relatively short queries.

Within the low-memory DP framework of Myers', we propose and analyse a scheme to make several threads cooperate on one or multiple pattern matching tasks (through *intra-task parallelism*). This approach allows us to tune the amount of data per thread, which enables the efficient usage of GPU registers and shared memory. We test different cooperative mechanisms, among them the new Kepler *shuffle* instruction.

Finally, we present a performance analysis methodology to identify the most relevant bottlenecks of our GPU algorithm. From it, we derive a new solution that uses register memory effectively by means of thread cooperation, and we are able to (1) overcome the memory-bandwidth bottleneck and (2) achieve a more efficient use of computational resources.

Our main contributions can be summarised as follows:

- We develop an algorithmic approach to solve the problem of computing Levenshtein distance in a thread-cooperative way, suited to a SIMD-based computational model. It relies upon a fast method to communicate carries by means of collective very-long integer add and shift operations

- We provide a CUDA-specific implementation of our algorithm, describing our optimisation strategies on the GPU

- We present an in-depth performance analysis showing that our CUDA code is computation-bound and scalable, and more efficient than simpler task-parallel CPU and GPU implementations. Performance is on the order of TCUPS (Tera Cells Updated Per Second).

In section 2 we review some terminology and prerequisites about Levenshtein distance, Myers' algorithm and GPU architectures. Section 3 contains our parallelisation proposal (first, by using a task-parallel approach, and next by introducing a thread-cooperative approach). In section 4, we present the experimental results we obtain when benchmarking our proposal on

several GPU systems. Section 5 discusses related work and, finally, section 6 summarises our results, describing future work.

## 8.2 Computing Levenshtein distance

Let $\Sigma$ be an alphabet of size $\sigma$, and the *pattern* $P_{[1..m]}$ and the *text* $T_{[1..n]}$ two strings over $\Sigma$. DNA strings generated by sequencing machines can usually be represented with the alphabet $\{$A,C,G,T,N$\}$, where A,C,G and Ts encode bases adenine, cytosine, guanine and thymine, respectively, and N indicates a base which is unknown due to some technical problem occurred during sequencing.

Levenshtein distance can be computed with DP techniques by using the following recurrence [44] to fill a score matrix $C$, with $0 \leq i \leq m$ and $0 \leq j \leq n$:

$$\begin{cases} C_{i,0} = i,\ C_{0,j} = 0 \\ C_{i,j} = min\{C_{i-1,j-1} + \delta(i,j); C_{i-1,j} + 1; C_{i,j-1} + 1\} \end{cases} \tag{8.1}$$

where $\delta$(i,j) is 0 if $P_{[i]}$ = $T_{[j]}$, and 1 otherwise. A score value $C_{m,j} = k$ identifies an occurrence of $P$ with Levenshtein distance $k$, ending at text character $T_{[j]}$. An example of score matrix is given in Table 8.1.a. The time complexity of the classical DP algorithm is $O(nm)$, i.e. proportional to the number of cells in matrix $C$.

We define the *maximum allowed error rate* as $\varepsilon = k/m$.

### 8.2.1 Myers' bit-parallel algorithm

Ukkonen [46] noticed that adjacent values in matrix $C$ can differ at most by $\pm 1$. A matrix of differences equivalent to $C$ can be represented using two bits per cell. Table 8.1.b shows a matrix of vertical differences, $\Delta v$, where $\Delta v_{i,j} = C_{i+1,j} - C_{i,j}$. Myers [45] used these adjacency properties to exploit bit parallelism and compute difference cells using bit-wise logical, shift, and addition operations. Time complexity becomes $O(n)$ if an $m$-cell column of $\Delta v$ fits into a computer word of size $w$ (typically $w$=32 or 64). Otherwise, a block strategy achieves complexity $O(n\lceil m/w \rceil)$. Hyyrö et al. [47] improved Myers algorithm by reducing the number of bit-wise operations.

Function $\delta()$ can be implemented using a *query profile* (see Table 8.1.c). Each of the $\sigma$ different columns is a bit-vector codifying the occurrences of each letter into the query. Also,

if matrix $C$ is constructed column-wise only one column needs to be kept in memory at a time, resulting in total memory space requirements of $O(\sigma \times m)$ (measured in bits).

Algorithm 8 shows pseudo-code for Myers' proposal. The main program and variables are at the top, while the time-consuming code, invoked once for each of the $n$ columns, is at the bottom. PV and NV are $w$-bit vectors encoding positive and negative differences in a given column. Text $T$ is scanned symbol by symbol, and each symbol $T_{[i]}$ determines the appropriate query profile in PEq[]. Function *advance_block*() executes 17 logical/arithmetic operations to transform the input (i.e. the previous column encoded as PV and NV) into the next column, i.e. to compute $m$ new vertical cells. It also provides a carry (the last cell in the column), which is the penalty to be added to the alignment score.

The basic algorithm assumes $m \leq w$ and is depicted in Figure 8.1a. Patterns larger than $w$ can be partitioned into $w$-bit blocks [45]. The block-based strategy needs to generate and send special carries between consecutive blocks, as shown in Figure 8.1b. This is achieved by means of a slightly modified version of function *advance_block*().

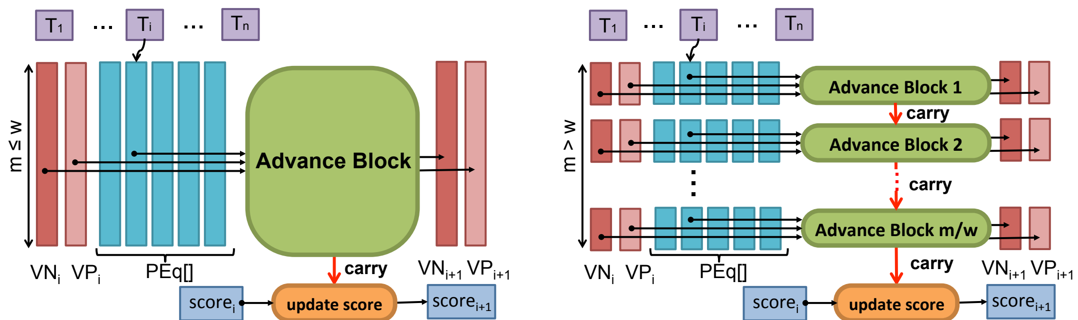


**Figure 8.1:** (a) Core operation of Myers' basic algorithm; (b) Myers' blocked-based algorithm.

| | | A | T | C | G | A | G |
|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| T | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| A | 2 | 1 | 1 | 1 | 2 | 1 | 2 |
| G | 3 | 2 | 2 | 2 | 1 | 2 | 1 |
| A | 4 | 3 | 3 | 3 | 2 | 1 | 2 |
| C | 5 | 4 | 4 | 3 | 3 | 2 | 2 |

(a) $C$: Score Matrix

| | | | | | | |
|---|---|---|---|---|---|---|
| +1 | +1 | 0 | +1 | +1 | +1 | +1 |
| +1 | 0 | +1 | 0 | +1 | 0 | +1 |
| +1 | +1 | +1 | +1 | -1 | +1 | -1 |
| +1 | +1 | +1 | +1 | +1 | -1 | +1 |
| +1 | +1 | +1 | 0 | +1 | +1 | 0 |

(b) $\Delta v$: vertical-differences

| | A | C | G | T |
|---|---|---|---|---|
| T | 1 | 1 | 1 | 0 |
| A | 0 | 1 | 1 | 1 |
| G | 1 | 1 | 0 | 1 |
| A | 0 | 1 | 1 | 1 |
| C | 1 | 0 | 1 | 1 |

(c) Query profile $\equiv \delta()$

**Table 8.1:** Dynamic Programming tables for sequences $P$=TAGAC and $T$=ATCGAG

**Algorithm 8:** Myers' algorithm for $m \leq w$

```
input  : P=pattern, T=text, m=|P|, n=|T|, σ=|Σ|
output : (minScore, position) with lower # differences
1  begin
2      bitvector<w> PV, NV, HMASK, EQ, PEq[ σ ]
3      ( PV, MV ) ⟵ ( ∼0, 0 )
4      HMASK ⟵ 1 ≪ (m − 1)
5      PEq[ σ ] ⟵ preprocess( P, σ )
6      for i=1 to n do
7          EQ ⟵ PEq[ T[i] ]
8          (c, PV, NV) ⟵ advance_block(EQ, PV, NV)
9          score ⟵ score + c
10         if ( score < minScore ) then
11             ( minScore, position ) ⟵ ( score, i )
12     return ( minScore, position )
13 Function advance_block (bitvector<w> EQ, PV, NV )
14 begin
15     bitvector<w> XV, XH, PH, NH
16     XV ⟵ EQ | NV
17     XH ⟵ ( ( (EQ & PV) + PV ) ∧ PV ) | EQ
18     PH ⟵ NV |∼ ( XV | PV )
19     NH ⟵ PV & XH
20     carry ⟵ ( PH & HMASK ) − ( NH & HMASK )
21     PH ⟵ PH ≪ 1
22     NH ⟵ NH ≪ 1
23     PV ⟵ NH | ( XV | PH )
24     NV ⟵ PH & XV
25     return ( carry, PV, NV )
```

This section describes and discusses two CUDA implementation strategies for Myers' bit-parallel algorithm: (1) task-parallel and (2) thread-cooperative. The work presented addresses the computation of Levenshtein distance for DNA strings, but can easily be extended to different alphabets.

## 8.3 Task-parallel designs

We assume there is a large number of input sequence reads, and each query must be compared to multiple regions in a large genome text. Having lots of independent query-text comparisons provides a straightforward source of task parallelism. This approach has been used on GPUs in

[48] [49]. We have developed our own implementation, putting our best effort on optimising the code. Apart from some implementation details described at the end of this section, the most performance-critical issue is handling the local storage for each task.

Bit-vectors PV, NV and PEq[] are accessed $n$ times during the algorithm execution. For the sake of performance, it is important to reuse this intermediate data, keeping them in on-chip memory and avoiding costly main memory transfers. The problem is that the aggregated size of this intermediate data grows both with the query size and with the number of running threads. For moderate and large query sizes either (1) memory performance suffers because intermediate data exceed the available on-chip GPU memory, or (2) GPU occupancy is sacrificed to make intermediate data fit into on-chip memory. Section 4 evaluates performance when storing intermediate data either in local memory or shared memory.

## 8.4 Thread Cooperative Approach

One way to deal with the previous problem is by making threads cooperate on the same task (intra-task parallelism) so that the amount of intermediate data per thread is reduced. Another advantage of thread cooperation is to enable the allocation of GPU registers for all intermediate variables. Registers provide more storage capacity and throughput than any other kind of on-chip memory.

### 8.4.1 Intra-task SIMD vectorisation: 1 warp per task

Finding enough intra-task parallelism to be efficiently exploited by even a single warp (SIMD operation) is challenging. Dynamic programming approaches present a well-known dependence pattern: any cell of the score matrix can be computed only after the values of the left and above cells are known.

There is potential parallelism when computing cells on the same anti-diagonal, but it is difficult to exploit, since it grows and diminishes as the anti-diagonal enlarges and shrinks while traversing the score matrix. Having said that, Myers' method for computing Levenshtein distance is interesting, as it allows processing all cells in a column simultaneously.

We revisit Myers' idea to exploit bit parallelism not only at the word level, but also at the SIMD level. Each thread (or SIMD lane) holds a word-size slice of the column information stored in bit-vectors PEq[], PV and NV. This scheme reduces and fixes the total local memory required per thread, which is now independent of $m$, the query size. Then, the CUDA compiler can easily allocate registers for the local data of each thread.

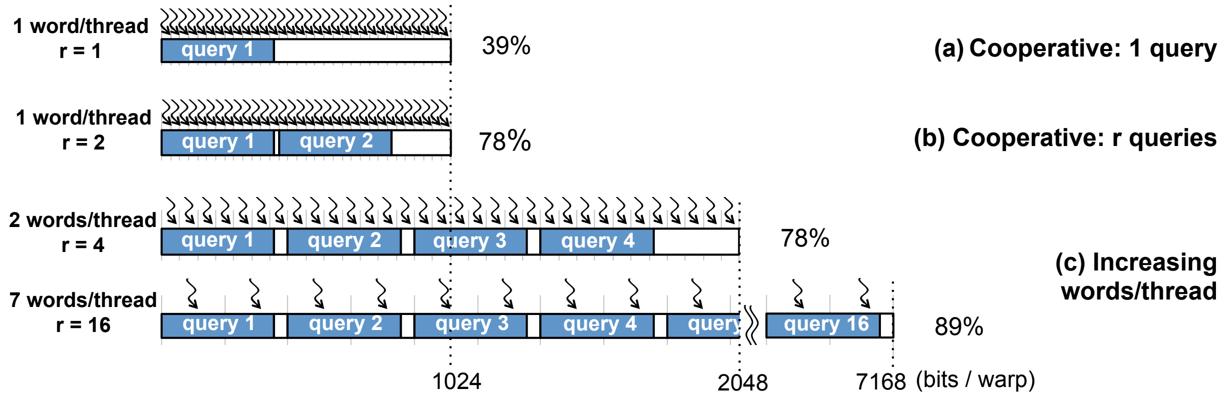


**Figure 8.2:** Thread Cooperation: r queries ($m$=400) and varying #words processed per thread

Most of the bit-wise operations on Algorithm 8 are inherently parallel (and, or, xor, not ...) and are trivially converted to SIMD/warp instructions. The exceptions are the add and shift operations inside *advance_block*() function. Algorithm 9 depicts the pseudo-code of our proposed thread-cooperative $m$-bit addition and shift operations. Each thread executes the code, receives a portion of each bit-vector input and generates a portion of the output. The cooperative shift requires one extra carry propagation step between neighbour threads. The cooperative addition uses a simple ripple-carry scheme. First, all threads perform a bit-wise addition of their corresponding portion of the input. Then, a cooperative loop of communication and carry addition steps iterates until no carries need to be propagated. Most times, it takes just one or two loop iterations to complete.

It is not surprising to find that most of the complexity falls in the addition operation. Indeed the "magic" of Myers's method resides in converting cell dependencies into the carry dependencies within the addition operation. This strategy ultimately benefits from the very efficient hardware implementation of the addition operation, which solves the carry chain dependence very quickly.

The 1-*warp-per-task* strategy works reasonably well for certain query sizes, but fails with others. Figure 8.2.a shows how a query of size $m$=400 is partitioned into 13 words, and exactly 13 threads cooperate on the matching task while the remaining 19 threads are idle. This case involves a disappointing thread utilisation of 39%. The next step to achieve high GPU performance requires the threads in a warp to cooperate on processing several queries.

**Algorithm 9:** Thread-Cooperative *m*-bit Addition and Shift functions executed by each thread

```
1  Function thread_cooperative_add (bitvector<w> a, b)
2  begin
3      bitvector<w> result
4      (result, c_add) ← a + b
5      while (check_any_thread (c_add != 0)) do
6          next_c ← send_to (threadID+1, c_add)
7          (result, c_add) ← result + next_c
8      return (result)
9  Function thread_cooperative_shift (bitvector<w> a)
10 begin
11     bitvector<w> result
12     c_shft ← a ≫ (w - 1)
13     next_c ← send_to (threadID+1, c_shft)
14     result ← (a ≪ 1) | next_c
15     return (result)
```

### 8.4.2 Intra- and Inter-task SIMD: 1 warp per r tasks

Combining intra- and inter-task parallelism enables two types of performance improvements. First, several small queries may be used to "fill" a 1024-bit SIMD vector and provide useful work for as many threads in a warp as possible. Figure 8.2.b shows how $r$=2 queries of size $m$=400 occupy $2\times13$=26 words (and threads), with an utilisation that raises to 78% (800 bits used from 1024).

Second, we can use a larger number of queries per warp in order to increase the total work per thread. In this case, more words are handled by each thread, as measured by the quantity *words/thread*. Increasing work per thread helps reducing query fragmentation and increase SIMD efficiency. Figure 8.2.c shows examples for *words/thread*=2 and 7 ($r$=4 and 16 queries), with thread utilisation rising to 89%.

But the most important advantage of increasing the amount of work per thread is the reduction of the total number of overhead instructions: those not included in the 17 bit-wise original operations in Myers' algorithm. The extra instructions needed for inter-thread carry propagation represent an important portion of this overhead. The drawback of increasing *words/thread* is that the amount of local memory required per thread also increases; this may compromise the efficient usage of GPU registers and GPU occupancy. As we will show in the next section, the best *words/thread* configuration depends on the query size but also on the

GPU architecture.

An extreme thread-cooperative configuration with $r$=32 is in fact purely task parallel, as there is no actual need of thread cooperation.This option, however, only makes sense for small queries. An advantage with respect to previous proposals is that the static declaration of variables allows using GPU registers instead of local memory.

The mechanism to let several threads cooperate on several queries requires identifying those threads responsible of the last slice of each query. They must be inhibited on carry propagation phases, but are responsible for generating the final result for each query.

## 8.5 Optimisation details

We simplify the inner code loop as much as possible to reduce the amount of divergence and instruction overhead. We help the compiler to generate non-divergent code by replacing conditional control flow structures by computation.

Since the input text can be very large, it is stored in binary form, with several symbols packed into a single $w$-bit data word. Divergence appears when threads access multiple text regions simultaneously and extract symbols from different positions of a data word. We apply a loop peeling optimisation [50] to move the extra control instructions and the associated divergence out of the main loop.

Additionally, divergence and instruction overhead outside the main loop is further reduced by extending text regions to start and finish in aligned locations.

Query pre-processing is moved out of the main code, so that each query is preprocessed just once, and not once for each candidate text region. All query profiles are created and stored into global memory before running the comparison code. For small alphabet sizes, like DNA, query profiles are just slightly larger than the original query strings.

Special GPU assembly instructions (*addc* and *add.cc*) implement carry propagation for local extended additions. Also, the Kepler-specific *funnelshift* instruction is used to propagate the carry in extended shift operations.

Thread-cooperative operations are implemented using thread communication at the warp level. We take advantage of the warp's lock-step execution to avoid synchronisation primitives. Several intra-warp communication techniques for carry propagation (shared memory, ballot and shuffle instructions) are implemented and evaluated. The Kepler-specific *shuffle* instruction is the most efficient alternative, with an improvement close to 20%.

## 8.6 Experimentation

We ran several implementations of Myers algorithm on different multi-core and GPU platforms. We first assess overall performance and then present a detailed analysis in order to identify the main architectural bottlenecks.

### 8.6.1 Experimental setup and methodology

The experimentation platform is a heterogeneous CPU-GPU node. The CPU is a dual-socket Intel Xeon E5-2650, with eight 2-way hyperthreaded cores per socket running at 2.0Ghz. Most of the GPU measurements were done on an Nvidia GTX Titan with 14 Kepler SMs (993Mhz). We also used a Tesla 2090 with 16 Fermi SMs (1.3 Ghz) and a Tesla K20c with 13 Kepler SMs (705Mhz).

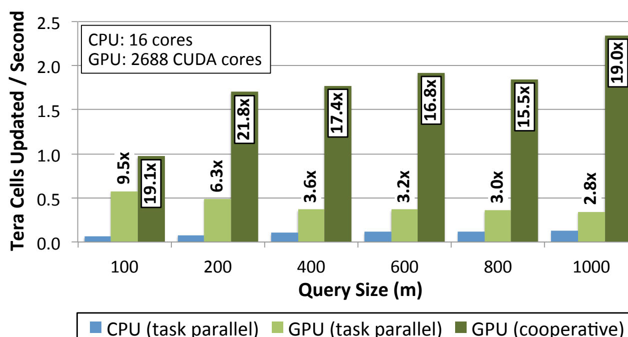


**Figure 8.3:** Performance overview

Commonly-used simulation tools [10] [40] are a standard way of providing the query input sets. Each input set contains a million reads. We have used a modified version of GEM [25] to generate all the candidate matching positions in the human genome (GRCh37) for such inputs. The accepted error rate is $\varepsilon$=0.2. At most 20 million query-candidate pairs (i.e., at most 20 candidates per query) are processed. The genome text and query profiles reside in CPU and GPU memory before starting execution measurements. Results are obtained by averaging over the 3 best executions, and expressed in terms of cell update operations per time unit. The variability of the measures is very low (on the level of the 1%).

The multi-core CPU implementation is task-parallel, with 16$\times$2 threads (OpenMP) to exploit hyperthreading, and is not vectorized. GPU implementations set the thread-block size to 128 for Kepler and 256 for Fermi, since they provide the highest performance.

| Query size ($m$) | 100 | 200 | 400 | 600 | 800 | 1000 |
|---|---|---|---|---|---|---|
| Task parallel (Local Mem.) | 54071× | 145270× | 368912× | 546418× | 724375× | 931655× |
| Task parallel (Shared Mem.) | 7515× | 3042× | 1149× | 1082× | 1059× | 1058× |
| Cooperative (1 word/thread) | 1.60× | 1.28× | 1.14× | 1.11× | 1.07× | 1.05× |

**Table 8.2:** Ratio of effective GDRAM accesses versus estimated GDRAM accesses

### 8.6.2 Overall Performance Results

Figure 8.3 shows performance on CPU (task-parallel approach) and GPU (both using task parallelism and thread cooperation) for increasing query sizes ($m$ from 100 to 1000). The presented results correspond to the best-performing configuration for each query size and implementation version.

The thread-cooperative GPU algorithm provides the best performance, surpassing the Tera-CUP barrier (from 1.0 up to 2.3). These results are between 15× and 22× better than those obtained by the multi-core CPU. Additionally, on the GPU the cooperative approach outperforms the task-parallel scheme by 2×-7×.

In general, longer queries provide better relative performance. This is expected since the relative weight of the initialisation phase and parallelisation overheads are reduced. However, the performance of the GPU task-parallel version reduces by a factor of up to 0.6× as the query length increases. This unexpected result is studied in detail in the next subsection. The analysis done helps understanding the reasons behind the thread-cooperative solution results.

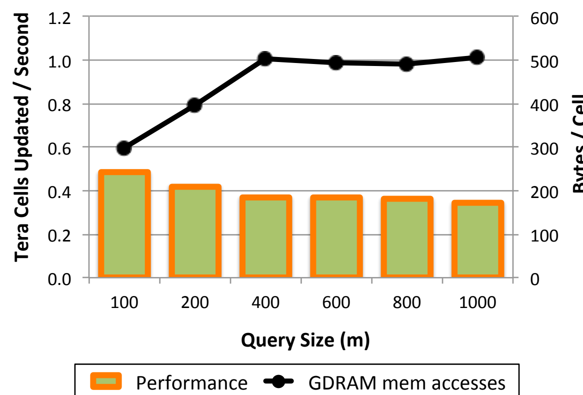


**Figure 8.4:** GPU Task Parallel: local memory

### 8.6.3 Task Parallel: Performance limiters

The task parallel scheme uses one thread for each query-candidate pair. This is a coarse-grained approach that performs well on a CPU but not on GPUs. In the next sections, we analyse the performance bottlenecks of the GPU implementation, either using local or shared memory to describe the reasons for these results.

#### Using Local Memory: high miss rate

Square bars on Figure 8.4 quantify how performance degrades up to 1.41× when increasing query size and using local memory. The solid line indicates an increase of 1.7× in the number of GDRAM memory accesses, from 297 to 506 Bytes/cell. There is a clear correlation between increasing the amount of GDRAM accesses and performance reduction.

The amount of local memory needed by the application grows linearly with the number of simultaneous queries and the query size. The number of queries is determined by the total number of threads launched for execution. Increasing query size decreases temporal locality and the L1 and L2 GPU caches become less effective to filter GDRAM accesses. For example, with a query size of $m$=1000, 94% of L1 and 79.5% of L2 accesses are misses.

Once GDRAM memory is identified as the main performance bottleneck, we need to see if the problem is latency- or bandwidth-bound. We measured empirical GDDR5 bandwidth to be between 185 GB/s and 210 GB/s, which range between 85% and 95% of the maximum bound provided by the Nvidia bandwidth test. Therefore, we conclude that the task-parallel GPU implementation using local memory is bound by GDRAM bandwidth. In contrast, owing to larger on-chip caches the CPU implementation is not memory- but computation-bounded.

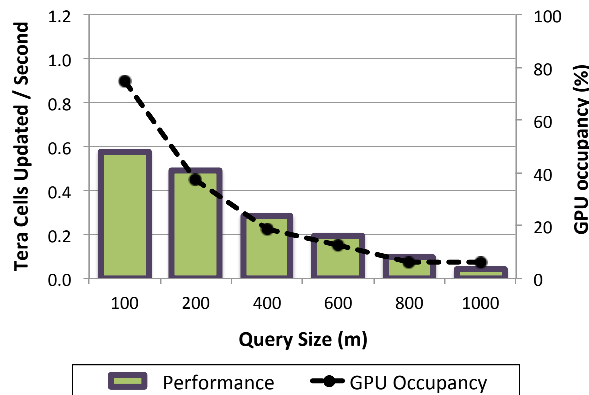


**Figure 8.5:** GPU Task Parallel: shared memory

| (m, words/thread) | (100, 4) | (200, 8) | (400, 4) | (600, 4) | (800, 8) | (1000, 8) |
|---|---|---|---|---|---|---|
| Bitmap operations/Column | 29.23 | 26.95 | 33.35 | 29.49 | 30.17 | 24.09 |
| Effective/Estimated GDRAM accesses | 5.39 | 1.57 | 1.12 | 1.07 | 1.03 | 1.02 |
| Bandwidth (GB/s) | 29.19 | 7.25 | 2.69 | 1.85 | 1.29 | 1.29 |
| IPC | 2.59 | 3.66 | 4.73 | 4.52 | 4.00 | 4.06 |

**Table 8.3:** Detailed performance metrics for best performing cases

### Using Shared Memory: low GPU occupancy

The classical solution to overcome GDRAM bandwidth memory problems is to foster data reuse by explicitly using shared memory. The best performance is achieved when we store columns PV and NV in shared memory, but maintain query profiles, PEq[], in local memory. Measured GDRAM bandwidth values for query sizes $m$=100, 200, 400, 600, 800, 1000 are now 41.2, 8.30, 1.56, 0.98, 0.72, 0.57 GB/s. Therefore, using shared memory actually prevents GDRAM bandwidth from becoming a bottleneck.

Table 8.2 compares effective GDRAM memory accesses with an estimation of best local data reuse. The estimation assumes that all data requests imply no additional GDRAM accesses if elements are already placed in on-chip memory.

A task parallel approach with local memory exhibits very limited data reuse. The use of shared memory increases the latter, but there is a significant amount of requests that are still fetched from GDRAM and not from on-chip memory.

Figure 8.5 shows the performance of the shared memory implementation. Bars indicate a performance degradation from 1.18$\times$ to 13.66$\times$ as query size increases. Again, this is due to the higher amount of local data, but now the effect is revealed by a reduction of GPU occupancy (i.e. the percentage of active versus potential running threads, depicted by the dashed line in Figure 8.5). Shared memory is a scarce resource that must be assigned equally to each thread. The GPU cannot allocate the same amount of active threads if each thread requires more memory; as a result, GPU occupancy is reduced to levels that strongly reduce overall performance.

Comparing Figure 8.4 and Figure 8.5 we conclude that using shared memory only benefits small query size cases, $m \leq 200$, when GPU occupancy is high enough to hide memory latencies.

### 8.6.4 Thread Cooperative: Performance limiters

We analyse performance and limiting factors of the cooperative approach. We first address the case of assigning a slice of the column to each thread, using one word per thread. Subsequently, we explore the performance advantage of using several words per thread. Finally, we analyse the execution in detail to find out performance bottlenecks.

#### Cooperation: one word/thread

Figure 8.6 presents results for the best combination of $m$ (query size) and $r$ (tasks or queries assigned to each warp). Performance varies between 0.6 and 1.0 TCUPS, always higher than the results obtained with the task parallel approach.

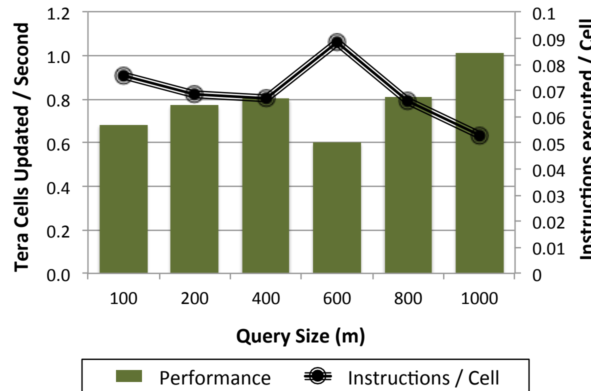


**Figure 8.6:** GPU Thread Cooperative: 1 *word/thread*

Table 8.2 shows that the cooperative approach drastically reduces the amount of GDRAM memory accesses, almost reaching the theoretical minimum. In fact, effective measured GDRAM bandwidth is lower than 7 GB/s for all query sizes. Also, all the executions achieve 100% GPU occupancy. Therefore, neither memory nor GPU occupancy are performance bottlenecks here.

We measured the total instruction count (in warp instructions) and computed the cell-normalised rate, that we denote by $instructions/cell$. This metric is depicted by the solid line in Figure 8.6 and exhibits a strong correlation with performance, which is inversely proportional to $instructions/cell$. This result suggests that GPU execution is now computation-bound.

In fact, the reason for the performance variations discovered in Figure 8.6 has to be found elsewhere. Warp instructions can simultaneously operate with 32 bits $\times$ 32 threads = 1024 cells. For each query size $m$, we must adjust the number of simultaneous queries $r$ to use a total number of bits as close to 1024 as possible. Figure 8.2 was showing the problem of low

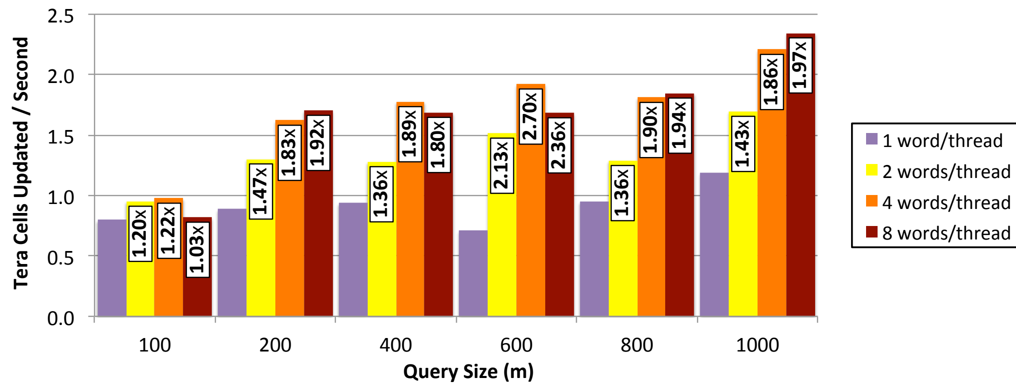


**Figure 8.7:** Performance for varying *words/thread*

thread utilisation. For the cases of Figure 8.6, thread utilisation is 78%, 78%, 78%, 59%, 78% and 97%, respectively. Considering that overhead instructions are relatively less frequent for larger query sizes, thread utilisation correlates almost perfectly with *instructions/cell*.

### Cooperation: several words/thread

Figure 8.7 depicts the performance impact of increasing the amount of work per thread (measured in *words/thread*) by processing more queries per warp. For fixed values of $m$ and *words/thread* the optimal value of $r$ is derived empirically. Results show performance speedups from 1.22× to 2.70× when increasing the amount of work per thread.

Also for this scenario we carried out an in-depth performance analysis, which can help generating new optimisation ideas. Figure 8.8 shows the performance trade-off involved when increasing the amount of work assigned to each thread.

On one hand, *instructions/cell* is reduced between 1.39× and 2.33× when increasing *words/thread*. This is due to the reduction of the instructions devoted to communication and synchronisation among the cooperating threads, and explains why the overall performance increases.

On the other hand, GPU occupancy falls sharply. As local data increases, more registers per thread are required and, hence, GPU occupancy decreases. In the examples shown in the Figure, the numbers of allocated registers are 28, 38, 56, and 92, respectively. The sharp plunge of GPU occupancy explains why overall performance flattens and even worsens.

In summary, for each query size $m$ one can find a configuration of $r$ (number of queries) and *words/thread* that maximises performance.

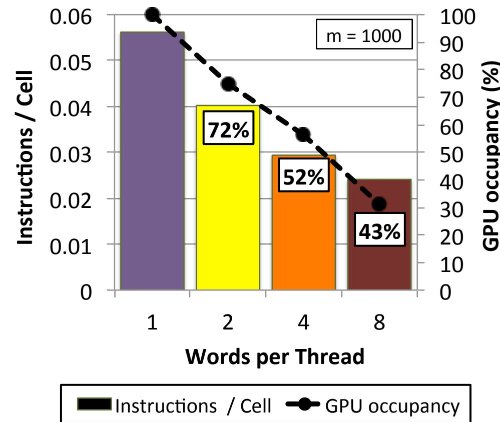


**Figure 8.8:** Impact of varying *words/thread* on *instructions/cell* and GPU occupancy

### Detailed Performance Analysis

We also measured the performance impact of using Kepler-specific instructions such as *shuffle* and *funnelshift*. Execution time is improved up to 28% and an average of 18%, meaning that Kepler GPUs have an important performance advantage with respect to previous-generation Fermi GPUs.

Table 8.3 provides data from relevant experiments with selected maximum performance values of $m$ and $words/thread$ to help understand the final performance limits of our GPU implementation. The first row of the table shows the empirical number of bitmap operations needed to compute a column, which varies between $\sim$24 and $\sim$33. The theoretical minimum is 17 bitmap operations [45] but this value does not consider the operations for score calculation, management of conditional structures, synchronisation and memory access. We conclude from those results that the parallelization overhead is limited and acceptable.

The second row of Table 8.3 shows the ratio between effective and estimated GDRAM accesses. This is a measure of data reuse, which is between 1.02 and 5.39. Effective GDRAM bandwidth is listed in the third row of the Table, and complements previous information. Measured bandwidth is found to be between 1.3 GB/s and 29 GB/s, very far from GPU memory system limits. From those results we conclude that memory reuse is very effective.

Finally, Table 8.3 shows an IPC (Instructions Per Cycle) value between 2.59 and 4.73. We consider these figures as quite close to the limit: the theoretical architecture maximum is 7, and many sources from Nvidia state that values above 4.5 are rarely obtained in real applications.

As a conclusion, the cooperative solution is computation-bound and exploits all GPU

resources very efficiently.

### Performance on different GPUs

We have repeated our performance analysis on different GPU architectures, namely Fermi and Kepler. Speedups with respect to the 16-core CPU are also included as a reference in Figure 8.9. The normalised performance obtained for all the GPUs is between 0.5 and 0.86 GCUPS per core and GHz. For a fixed query size, normalised performance (obtained by factoring out the architectural advantage of the Kepler instructions) is very similar in all three GPUs. This means that performance scales fairly well with the number of cores and clock frequency, even when using GPUs with different CUDA capabilities (Fermi and Kepler). Such results back the expectation that our proposal will show a good performance scaling even on future, more powerful GPUs.

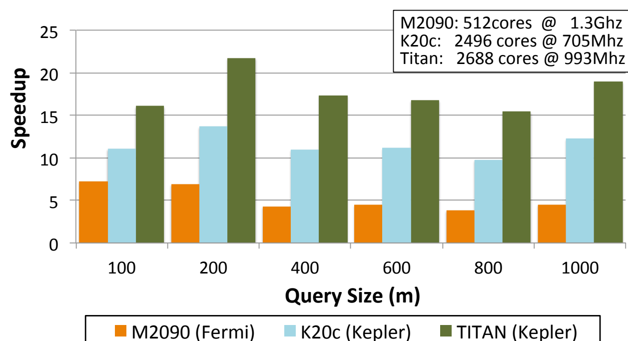


**Figure 8.9:** Speedup of several GPUs vs CPU

## 8.7 Conclusions

Upcoming sequencing technologies will produce longer reads at reduced cost. This will put additional stress on current sequence alignment algorithms, that will quickly become the bottleneck of the pervasive analysis pipelines used to process resequencing data.

In this work we improve on the GPU Myers' algorithm, which computes the Levenshtein distance between two strings and constitutes a basic block of several popular aligners. Experimental results show that our best implementation obtains on a single GPU performance speedups of 20× with respect to a sixteen-core, non-vectorised CPU version, providing a peak performance of 2.3 TCUPS.

The solution presented here is ready to be efficiently executed on any current GPU. To tune it to the target architecture it is sufficient to adjust the work-per-thread ratio; if more local memory is available on the GPU, an appropriate reconfiguration will improve performance.

From a methodological standpoint, this thesis provides an example of how task-parallel CPU approaches can be redesigned into cooperative multi-thread algorithms adapted to many-core architectures like the GPU; the main principle guiding our implementation has been to get the most from local memory system and reduce the number of instructions. We have also demonstrated how specific Kepler architecture instructions can be used to further improve algorithmic performance.

From the standpoint of the analysis of sequencing data, we have shown that GPUs are computational platforms suitable to efficiently implement string-comparison algorithms. Our results indicate that GPUs can become an additional source of computational power in order to perform high-quality alignment of longer sequence reads in acceptable times.

As future work, we will implement on the Intel MIC architecture a version of the cooperative-parallel algorithm that uses explicit SIMD instructions; its performance will provide us with a comparison of the benefits offered by the two architectures. Also, we plan to integrate our GPU algorithm into the GEM mapper [25], thus demonstrating the practical relevance of our results.

# 9

# GEM-Cutter: high-performance bioinformatic library

*"Any sufficiently advanced technology is indistinguishable from magic."*

**- Arthur C. Clarke**

This chapter is focused on the GEM3-GPU and GEM-cutter. After a brief (1) introduction, the GEM3-CPU pipeline will be described in section (2), so that the GEM3-GPU internal workflow can be detailed in section (3). Section (4) will be dedicated to the GEM-cutter library that was created as a part of the current thesis. The last two sections provide details of additional considerations and special features of the GEM3-GPU.

The objective of the current chapter is to provide the reader with a global vision of the GEM3-GPU mapper, providing with essential concepts and knowledge needed for the understanding of the following Chapter 10.

The GPU modules of the GEM-cutter are fully described. The chapter provides with a description of the scheduling and data management processes, together with the GEM-cutter library strategies, through which the use of GPUs becomes transparent for all users.

Also, the strategies of task partitioning, regularisation and fine grain parallelism (previously introduced in Chapter 3) are detailed, as well as how all of these methods are integrated on a real-production GPU mapper though a developed library (GEM-cutter).

Finally, we describe the advantages of GEM3-GPU in comparison with other GPU mappers, as well as the specific features of this mapper.

## 9.1 Introduction

As previously described in Chapter 2, a traditional end-to-end genomic downstreaming pipeline for sequence analysis is composed by primary, secondary and tertiary analysis. At this point, it is well known that secondary analysis is the most computationally expensive stage of the pipeline, being short read mapper applications one of its key components [2]. Read mappers are used extensively in sequencing projects to solve the problematic of aligning billions of sequences (reads) generated by the sequencer against a large reference sequence (e.g., whole human genome). The final goal is to reconstruct the targeted genome by finding the original position of each read and calculate its alignment, while filtering artifacts that have been erroneously introduced by the sequencer and reporting the actual identified variants from the analysed sequences. Figure 9.1 shows the pipeline steps between primary and secondary analysis, and the requirements of high sequencing coverage (by redundancy) to recover the sequencing artifacts.

On this chapter we will briefly review a short read mapper GEM3 [25], to which we have been contributing in a research collaboration, both on its development and the full integration of the GPU acceleration features. GEM3 is based on a reference alignment method and, as briefly explained in Chapter 2, it performs a search by filtration, as a seed and extend method, being divided by two well-defined stages, (1) search of exact and approximate seeds and (2) a filtering and alignment of the most promising candidates generated from the previous stage.

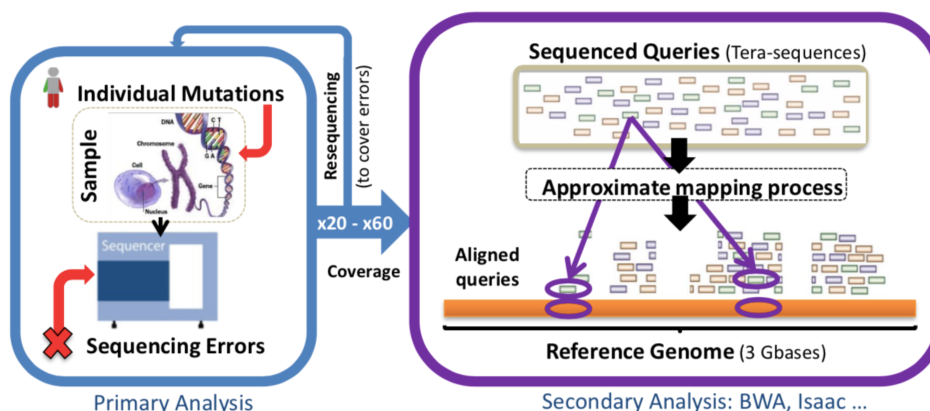


**Figure 9.1:** Short read mapping, interactions between primary analysis and secondary analysis.

## 9.2 GEM3-GPU internal workflow

Through the GEM3 CPU execution, the mapper performs the input/output processing in batch mode (due to storage bandwidth efficiency reasons), then a set of queries is loaded from the fastq file in a blocking strategy; and finally results from the sam file are saved in a similar blocking approach. GEM3 has a pool of input blocks available during the execution, each thread of the system is assigned to an input block to start a full independent local analysis on that set of reads, and finally CPU threads must synchronise between themselves to store the results in the right order in the sam file. Note that each thread processes its input block of reads serially; and therefore a single read is processed end-to-end before starting the processing of the following read (batch 1).

### 9.2.1 Parallelism at thread, pipeline and task levels

The previous approach has different benefits, as it minimises memory consumption, reduces latency of the results per read, and allows to apply straightforward cut-off strategies to reduce the amount of work. Besides their benefits, the major drawbacks of the previous proposal workflow is the limited amount of parallelism that it exposes. GPUs require hundreds of thousands of parallel tasks to take profit of their massive amount of computational resources. This motivated us to fully redesign the previous CPU workflow mapper to be end-to-end batch-oriented and to create pipeline tasks between CPU, GPU and data transfers along all the stages.

The next section will describe the applied strategies to explicitly extract parallelism at

different levels; batch, pipeline, multi GPU and thread kernel-level. These changes expose some design challenges that are explained over the following sections, figures 9.2, 9.3, and 9.4

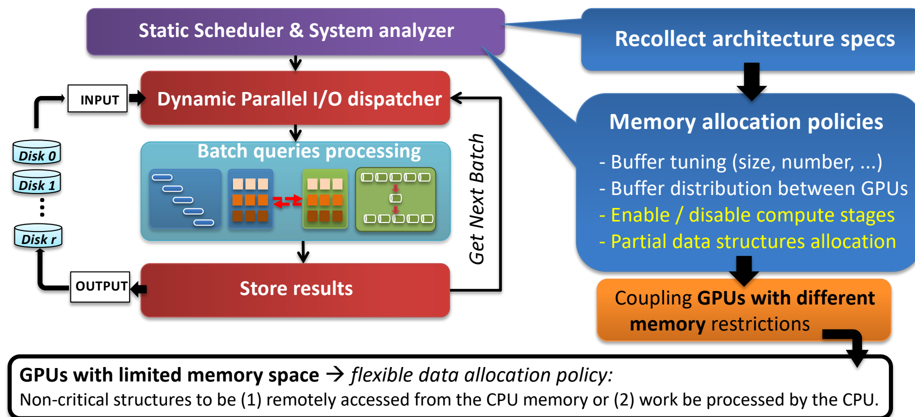


**Figure 9.2:** GEM3-GPU: Overview at system-level of the internal workflow

- *Batching parallelism*: Figure 9.2 describes the high-level workflow, the main idea is to generate block pool from the input file (fastq) in order to increase the number of reads that can be processed in parallel, the block size will indicate the degree of parallelism of the system. More details regarding the necessary: (a) dynamic schedulers and (b) the characterisation of the system to support these batching features could be found in section 9.2.2.

- *Pipelining parallelism*: Figure 9.3 shows how to increase the parallelism by increasing the number of buffers assigned per stage. This breaks the typical dependencies of a pipeline and allows to execute simultaneous different stages of CPU, GPU and transfer processing. Gem-cutter is in charge of providing round robin re-utilisation of all the buffers to ensure the best utilization by a) exposing parallelism by pipelining between stages and (b) showing the batch dependencies.

- *MultiGPU parallelism*: Figure 9.4 shows how Gem-cutter could identify all the GPUs from the system and characterise them, in order to automatically tune the number of buffers and its size for every GPU in the system. The picture shows how different levels of parallelism are managed by the library, at thread level, batch level, pipeline level and multiGPU (where different buffers can be processed in the same GPU in parallel) or a task spited between different GPUs to distribute the work on

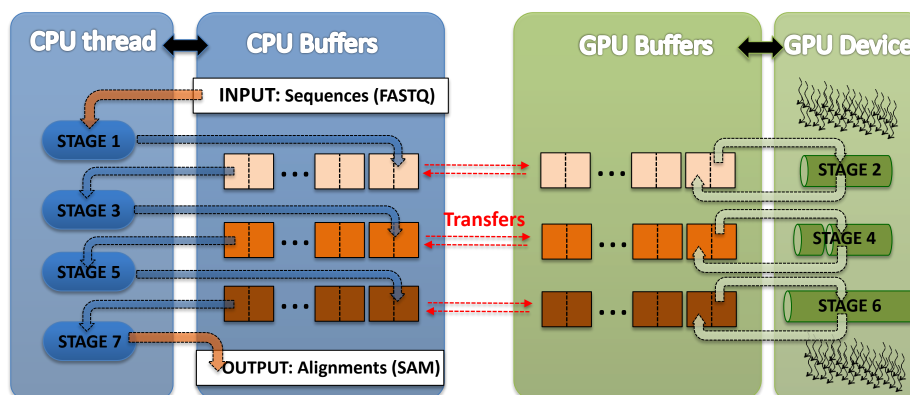


**Figure 9.3:** GEM3-GPU: (a) exposing parallelism by pipelining between stages and (b) showing the batch dependences

different buffers. Relationship between CPU threads, CPU/GPU internal buffers, data streams,simultaneous kernels and multiple devices is also described.

### 9.2.2 GEM3-GPU high-level workflow

Figure 9.2 describes the high-level workflow at system level, which is responsible for managing the heterogeneous resources utilised by the GEM3. Firstly, the mapper performs an initialisation and setup for all the system using the following modules:

- Characterisation of the machine: takes into account the number of GPUs, CUDA Cores, CPU Cores, uarchitectures, frequencies, GDDR and DDR bandwidth and memory available, cache hierarchy, NUMA and PCI-e/nvLink hierarchy. These characterisations are used to define (a) the size and number of buffers per GPU in each stage, (b) buffer distribution per GPU devices, (c) initial pining between CPU threads and GPUs and GPU stream creations, (d) Enable or disable different GPU accelerated primitives (mapper stages) (e) data structure policy allocation.

- Dynamic Parallel dispatcher: There are two levels dispatchers to dynamically schedule work; first there is a i/o block dispatcher that assigns input data from the read file in blocks to each different thread without additional work to do (a controlled work stealing pool); and the second there is a dynamic scheduler to dispatch blocks of input.

- Flexible dynamic allocation policies: due to different GPU memory sizes, predefined policies in combination with the characterisation of the node information will decide

which data structures are allocated on the main memory of the GPU or CPU. More details can be found in section 9.3.1.

The main idea is having a pool of buffers already allocated on the GPU in order to serve them to a CPU thread as available buffer for the task of each stage. Gem-cutter manages the transfers and synchronizations, and it is completely transparent to the user thought its API.

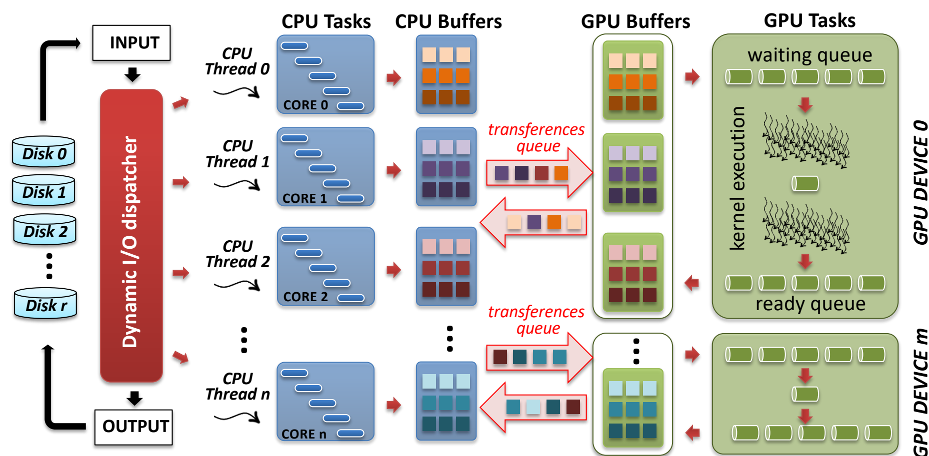


**Figure 9.4:** GEM3-GPU: Relationship between CPU threads, CPU/GPU internal buffers, data streams, simultaneous kernels and multiple devices

## 9.3 GEM-Cutter library

This section will explain the details of the integration of GEM-Cutter on GEM3 and its internal workflow. Gem-cutter is a high-performance bioinformatic library for GPUs. The techniques included on the library allow scaling for large problems by using fine-grain parallelism. The library comprises the most used building blocks used for DNA sequence mapping and alignment software. All the algorithms are highly optimized using custom structures and low-level optimisations specifically for each Nvidia GPU architecture (from Fermi to Volta). This is explained in full detail on Chapters 5, 6 and 7. The library uses a message passing programming model to make all the GPU specific programming details transparent to the user. The core library is fully integrated and tested with GEM3-mapper. It is self-contained including all the necessary mechanisms for this correct integration on other tools; such as dynamic load balance schedulers, I/O modules, memory and compute resource allocators, batch and buffering interfaces, multi-GPU support and wrappers for automatic asynchronous transfers, as detailed in the previous sections.

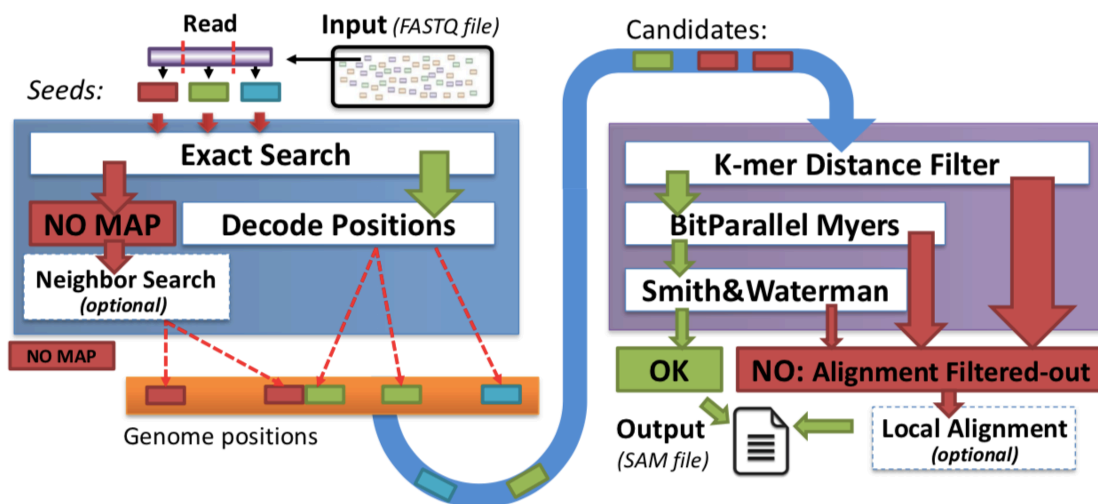


**Figure 9.5:** GEM3: internal mapper workflow, stages and relationships

### 9.3.1 GEM3-GPU mapper: specialisation techniques

We consider GEM3-GPU as an heterogeneous mapper, in the sense that CPU and GPU integration is very tight and the processing parts of the mapper are highly specialised for each processor. A clear example are the internal index data structures; FM-index, Suffix-Array and Human Reference are allocated in both addresses spaces but using a different representation. Given that GPU data structures are highly specialised, we decided to (1) redesign the data structures for the most common cases (with less features and flexibility than CPU version), (2) consider in the structure design the architecture of GPUs to reach the maximum possible performance and (3) use alternative data structure designs and compression due to the GDDRX memory size limitations on GPU. For that reason, the CPU will be processing the uncommon, corner cases or extremely irregular tasks.

### 9.3.2 GPU kernel-level uschedulers

All modules from GEM-cutter are based on fine-grain parallel implementations where several threads collaborate to perform the same tasks, as explained on Chapter 3. Intra-task parallelization has multiple benefits: lower memory footprint, less thread divergence, less gather/scatter memory patters, task regularisation, etc.

On the other hand, this arises additional challenges, (1) the group of threads that collaborates on the same task has to be allocated to the same warp and (2) additionally, it is highly recommended to aggregate sets of tasks with similar task size in the same warp.

Each primitive module from GEM-cutter includes a micro-scheduler that guarantees the previous requirements. Actually, the same execution kernel performs a kernel-fused to perform (1) the binning process by task size and (b) introduces disabled threads to the used warp, in order to avoid fragmenting a task with different threads in two different warps.

This process has to be transparent to the user, so data results from the kernel have to be resorted again with the original order in the buffer. Notice that additional intermediate memory for the re-arrangement is necessary inside the buffer.

### 9.3.3 CPU and GPU fine grain collaboration

As explained in the previous section 9.2.1, batching allows us to increase the required parallelism to exploit GPU resources, but at same time it prevents us to apply several work reduction strategies (e.g., using historics of previous executions to take better decisions in the workflow or some work-cut strategies).

This is the main reason to search for an intermediate solution, allowing us to reduce the batch size without losing performance, this is being critical to potentially reduce the amount of executed work on the GPU workflow. Notice that increasing parallelism is another complementary solution that greatly help to search for better trade-offs.

Once all these challenges are surpassed, a highly efficient and very fine grain communication and task collaboration between the CPU and GPU will be necessary to apply these work-cut strategies.

### 9.3.4 Transparent GPU transfers for the user

Gem-cutter is hiding all the complexities related with the GPUs with a specialised library. Under the hood, we are taking care of selecting the right buffer, increase automatically the buffer if needed, send / receive the transfers, synchronisations, stream creation and control, workload balancing for different heterogeneous GPU architectures. The responsibility of the user is just to fill a buffer and send it as an asynchronous transference.

### 9.3.5 Residency policies of the data structures

We have implemented residency policies for each accelerated module from GEM-cutter. The mapper automatically decides which data structures prioritise on the main memory of the GPU. The data structures could have different status, (a) not allocated (there is no GPU acceleration),

(2) allocated on the global memory of the GPU, located on the host memory (as zero-copy) and still using the GPU (but using less memory bandwidth, due to all memory accesses are thought pci-e) or (3) emulated by CPU. The GEM-cutter library takes this decisions as initialisation of the process automatically, although the user could force the policy by a parameter in the library API.

## 9.4 GEM3-GPU internal workflow in detail

This section will describe all the stages performed by the GEM3-mapper and the additional features implemented to be executed on GPU.

*Stage 1: Adaptative Exact Search*
It is worthy to highlight, that every step from the *seeding phase* is generating more work and parallelism on the GPU workflow, meanwhile the *extending phase* is reducing the amount of work and parallelism.

Figure 9.5, depicts the high-level internal workflow; (1) in the *Exact Search* step, read sequences are loaded from the input file (fastq); and each read is divided in fragments (called seeds). The number of seeds generated depends on the percentage of errors from the read that we want to cover by the search; e.g., 12% of errors of 100nt reads requires 13 seeds, (see pigeonhole principle Chapter 7).

By using default execution parameters on GEM3, this stage executes the adaptative version of the FMI search [25], meaning that the number of read partitions are selected at runtime depending on (a) the content of the read and (b) its mappability to the genome. The algorithmic core of this adaptative seed selection stage is based on a greedy approach, the Chapter 7 describes in detail the low-level GPU implementation of this core algorithm and describes the necessary changes in respect Chapters 5 and 6 to be integrated in GEM3 for production. It is worth mentioning that this stage could be executed as static seeding, where the size of the seed is decided statically and independently of its content, or as adaptative version providing a better trade-off between compute and sensitivity on the search.

GEM-cutter includes a high performance GPU version of the adaptative search, this version is full cooperative (8 threads cooperate to perform a single backward search). In addition, the implementation could use an additional LUT structure on GPU to accelerate even further the search process. It allows to search the first steps of the seed just using a single memory access to the LUT entry. The thread cooperation and the LUT table increase the regularity

of the search; reducing thread divergences with other groups in the same warp. We use a micro-scheduler as a pre-process before the search to classify all the queries by size and regularise the amount of work inside the warp. At the end, the original binning should be reversed.

*Stage 2: Decoding Candidate Occurrences and Seed chaining*
This step turns the candidate occurrences explained in Chapter 7 (reported by the first step) from index domain representation into reference domain positions. All the core primitives could be revisited on Chapters 4 and 5. The GPU implementation is using cooperative threads (using same ideas than in the previous search step). A micro-scheduler (binning process) is necessary because 10 threads per backward search are cooperating, and 2 threads per warp should be idle. Using the decodification with default parameters it creates a Suffix Array sampled 1:8, where 8 will be the average number of LF-mapping consultations to the FM-index.

After decoding the positions for every seed, a seed chaining algorithm is used to identify in the genome the region of the candidate. For that, GEM3 will search for overlapping seeds. It is interesting to note that different seeds could reach exactly the same position region in the reference genome. This is a purely CPU process, in the future it could be interesting to explore how to port this phase to GPU.

*Stage 3: K-mer distance pre-filtering (CPU/GPU)*
This stage is a pre-filtering that prone the highly diverging candidates reported by the adaptative search process from the candidate list. Thus, only the sensitive reads with a certain error are reported to the next stage. There is an GPU implementation K-mer. GEM-cutter incorporates a K-mer GPU implementation, it is using a tiling method to increase the filtering ratio and improve the regularisation of the work for the thread cooperative implementations. A micro-scheduler before the kernel execution is implemented to improve the performance and regularisation.

*Stage 4: BitParallel Myers Filtering (CPU/GPU)*
A bitparallel approximate string comparison method is applied in this step. Edit distance events are reported and a final score classifies the candidate position due to its homology with the region. The core algorithm of this step is the Bit-Parallel Myers algorithm which can exploit higher throughput computer vector resources from GPUs. This mapping is described as pseudo-alignments and can be used as a final output result. Depending on the requirements

for the user pipeline in the data analysis process. There is a very advanced micro-scheduler that regularise the work by (1) decomposing the alignment matrices in smaller ones (for large candidates) and (2) classifying them by work size. After all the process candidate results are reorganised to make this scheduling process transparent to the user.

*Stage 5: Global alignment (CPU and GPU)*
In order to report a cigar string for each mapping found, this stage processes a dynamic programming algorithm that performs a global alignment between the read and the genome region reported. A GPU version has been created. The BMP align algorithm is accelerated by GPU, producing the score matrix and generating a cigar string for each global alignment. There is a very advanced micro-scheduler that regularise the work by (1) decomposing the alignment matrices in smaller ones (targeting large candidates) and (2) classifying them by work size. Also, the micro-scheduler tracks the error rate reported by each partial alignment and performs cut-off strategies to reduce work on the GPU. In addition, an S&W GPU implementation was developed for this stage at NVIDIA, but it is not currently integrated to GEM-cutter. [11]

Notice that the other stages *neighbourhood search* and *local realignment* are both purely executed on CPU, we can consider them optional and more oriented to be activated on difficult reads/datasets to map.

### 9.4.1 GEM3-GPU: Emulated mode

The mapper can be executed in emulated mode; exactly the same processes and algorithms from GPU are executed on CPU. This is very useful for (1) validation of results (2) analysing which is the overhead introduced by the GPU workflow. Each stage could be activated or deactivated independently to be running CPU or GPU. All these infrastructure allows us to quantify the overhead of the batched workflow for GPU compared to the CPU. This mode also helps to evaluate the performance impact of specific data structures used on the GPU.

The emulated mode could provide feedback from where inefficiencies are coming from: (a) cases in which we are re-processing the same task more than once because GPU doesn't have support for that feature and (b) more expensive computations due the memory size limitations, usually smaller (pruned) data structures and compression are necessary. (c) The additional overhead for scheduling and managing for a data transformation (regularisation) techniques.

## 9.5 GEM3-GPU special features

We would like to mention certain characteristics that are not common (or even present) in other GPU mappers; and where the GEM3-GPU specially shines.

After a very detailed analysis of all the GEM3 components and the literature, we selected and evaluated which sections of the mapper were more CPU or GPU suitable, and therefore GEM3 mapper effectively uses CPU and GPU by specialisation of each part, on code, functionality and internal data structures. It is also important to point that the CPU and GPU versions of the mapper report exactly the same SAM output result file (they are diff command equal). The mapper supports multi-GPU systems (without specific limitation on the number of GPUs), and can run on system with heterogeneous GPUs (different uarchitectures and main memory size) on the same system.

GEM3-GPU can run in a commodity in GPUs with $\tilde{1}0$ GBytes of GDDR memory, and less using the memory policies from Gem-cutter. Moreover, there is no limitation of the GEM3-GPU to a maximum read input size, supporting a variable read length processing at runtime (with no prior assumptions). Thus, it can scale with the size of the read. On the contrary, GPU mappers usually have restrictions on these points, making them impracticable or difficult to use on production environments.

As previously mentioned, the application has been tested and deployed in production environments, providing on this process a very good robustness by mature-processing with real data and being stressed on HPC systems. We can say that it has been benchmarked with a wide number of datasets from different sequencing technologies [12]. GEM3 is also used in international re-sequencing projects and has been tested on other institutions and private companies.

Finally, GEM3-GPU has full compatibility for CUDA SDK, but is also backward compatible from CUDA 5.0 to current versions. The GPU codes are optimized and fine-tuned for all the GPUs released by NVIDIA from Fermi to current uarchitectures (Ampere). In addition,the codes are portable as well, so that can be run on a broad number of host and device architectures (x86, aarch64, ppc)

## 9.6 Conclusions

There are several reasons that make the current chapter essential for this thesis.First, the GEM3-GPU and GEM-cutter implementations detailed in this chapter are the result of several

years of work and experimentation. The chapter is a compendium of all the optimisations seen in previous chapters, as well as the ones required for the mapper to be running in production. Thus, the different optimisation methods explained in Chapter 3 are applied in the current chapter, being also essential to understand the results of the next experimentation Chapter 10. Finally, the chapter describes the developed library that facilitates the GEM3 integration process and facilitates the task by using the right abstractions in the interfaces.

# 10

# GEM3-GPU Mapper benchmarking and experimentation

*”It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts”*

**Sherlock Holmes**

”The objective of this chapter is to evaluate all the previous proposals by integrating them into an end-to-end read mapper for GPU-accelerated HPC and embedded heterogeneous systems, and running in real production environments. We will present (1) a methodology for performance and accuracy evaluation; (2) results in production environments; (3) a comparative evaluation that covers the current state-of-the-art; and (4) final conclusions”

## 10.1 Introduction

This chapter shows the final results of this project and demonstrates the robustness of the contributions throughout the thesis. The objective is to evaluate an end-to-end read mapper for GPU-accelerated HPC and embedded heterogeneous systems in real production environments. We will present the GEM-cutter library, which integrates the basic GPU-accelerated building blocks used for implementing the read mapper, which we call GEM3-GPU.

The chapter starts by describing a methodology for performance and accuracy evaluation. Then, we will present results obtained in production environments using representative data. Next, we will show a detailed comparison with the most relevant proposals that cover the current state-of-the-art. At the end of the chapter, we will present our thoughts and conclusions.

## 10.2 Experimentation environment and methodology

This section describes all the issues related to the experimentation setup, including the computer systems and platforms, the datasets, the metrics, and the methodology. We have evaluated performance and accuracy of GEM3-GPU against other short read aligners with CPU and CPU+GPU support. For completeness of the results, we have used a wide variety of datasets which are described in the next subsections.

### 10.2.1 Compute systems

The entire experimentation from this chapter is made using three different environments, which are described below. The objective of that specific selection of three systems has been to mimic a very similar environment to the production centres.

The common configuration of a DNA sequencing platform is a shared memory server node with dozens of CPU cores, large amounts of main memory (hundreds of GBytes) and fast input/output storage. Interestingly, these applications are commonly using single node execution, and bioinformatic applications using distributed parallel systems (e.g., using MPI) over network are not very common.

It is worth to mention that low power devices are opening a new line of research for sequencing due to the recent interest of the community to process sequencing analysis in a more restricted low-power and form-factor environments. An example of this is Illumina, that

recently released Novaseq sequencers with FPGA built-in for on premise sequencing fully integrated with their base calling (primary) analysis.

These are the key points to decide to include in the experimentation environment, different host architectures (ppc, x86, aarch64), storage solutions (ramdisk, flash drives, parallel systems) and different GPU architectures, which covers the discussed from HPC to built-in sequencer processing environments.

- **System ppcle-volta:**
  *Host:* 2 x IBM Power9 8335-GTG @ 3.00GHz (20 cores x 4 threads/core)
  *Main Memory:* 512GB @ 2666MHz of main memory
  *Devices:* GPU NVIDIA V100 (Volta) with 16GB HBM2 using nvLink 2.0.
  *Storage:* 2 x 3.2TB NVME

- **System x86-kepler:**
  *Host:* 2 sockets Intel Xeon E5-2630 v3 (Haswell) 8 cores at 2.4 Ghz.
  *Devices:* GPU NVIDIA K80 (Kepler) with 12GB using PCI-express 3.
  *Main memory:* 128 GB DDR4 @ 2133 MHz
  *Storage:* Lustre distributed storage system

- **System arm-maxwell:**
  *Host:* 2 cores NVIDIA Denver2 + 4 cores ARM Cortex-A57
  *Device:* 256-core Pascal GPU
  *Main memory:* 8GB LPDDR4 128-bit interface
  *Storage:* 32GB eMMC

**Compute systems clarifications**

- arm-maxwell system is an embedded platform provided in a developer kit by the Nvidia Tegra TX2 system. The machine is configured with all the core sets with full performance configuration and using the maximum frequency of memory to avoid instabilities on the executions due to power throttling.

- It is important to notice that the benchmarked system ppc-volta is an in-order execution platform, requiring a high aggressive SMT per core to increase their

resources utilization effectively.

- Using lustre (parallel i/o storage) is the most common scenario for production analysis. Usually, final output and intermediate data is stored for every analysis, to lead reproducibility or to reduce the re-sequencing data of other analysis. However, some centres are using NVMEs to store intermediate data in order to accelerate the full pipeline; or using specialised nodes for transfer and compress/decompress tasks.

### 10.2.2 Short read mapping applications

In this chapter we will evaluate 10 different read mapping applications. A thorough selection has been made to be representative of the most widely accepted by the scientific community.

BWA-MEM [8] is widely accepted by the community as one of the most important applications for short read mapping. BWA-MEM was developed (in 2013) by *Heng Li* from the *MIT* and *Broad Institute* (now at Dana-Farber Cancer Institute and Harvard Medical School). The mapper is highly tested on production systems because: (1) it was developed under Broad Institute, one of the most important productions centres; (2) it is currently part of *GATK Best practices* [51], the most adopted bioinformatic pipeline used in production; and (3) it is under open-source license and the community is very active reporting issues and feedback, which constantly improves its robustness. We present BWA-MEM first because,given its importance to the community, we will use it as a benchmark for performance and accuracy to compare with other mappers.

Regarding the GEM3 benchmarking, to carry out the evaluation we are analysing stable versions of (a) the batched GEM3-GPU mapper (Chapter 9) and (b) the presented bioinformatic GPU GEM-Cutter core library (Chapter 9), which integrates all the described strategies of previous chapters. The coupling of each software allows us to perform a functional and performance validation for a full-fledge CPU-GPU mapper.

The following mappers were selected for the state-of-art comparison with GEM3-CPU and to validate all the contributions. The following detailed list is classifying them by index feature:

*BWT based mappers*

**Bowtie2** [14] is an aligner based on a modified FM index to match the reads. It allows several mismatches on the seed search, while the extend phase is implemented using an Smith&Waterman local alignment. Bowtie2 was created at John Hopkins University.

**nvBowtie** [21] was created by NVIDIA and is a GPU-accelerated re-engineering of Bowtie2. Essentially, it was re-written to reproduce the majority of Bowtie2 characteristics in GPU. However, it has the advantage of providing a higher throughput and accuracy than Bowtie2 by the use of GPU massive parallelism. The aligner allows a wide range of read sizes and mismatches.
**Soap3-dp-GPU** [22] is an aligner created by the University of Hong Kong and BGI. It allows a massive parallel search of multiple mismatch alignments and gapped alignments by using a tailor-made GPU-BWT. It exploits GPUs to provide performance improvements and sensitivity while providing compressed indexing and memory optimisation.
**Cushaw** [23] is a parallel short read aligner created at the Johannes Gutenberg University, which is based on CUDA. The searching strategy is based on BWT and FM-Index and can be used for either exact or approximate matching. A quality-aware searching approach is used to reduce the space and improve the search quality.

*Suffix-Array based mappers*:
**HPG-align** [20] was created by the University of Cambridge and the Centro de Investigacion Principe Felipe (CIPF). The strategy combines the BWT (for mapping) and SW (for local alignment), latest versions could use Suffix-Array for seeding. In this chapter we are using the Suffix-array indexation.

*Hash based mappers*:
**SNAP** [24] uses a hash index instead of the Burrows- Wheeler Transform and Bitparallel Myers algorithm for the alignment phase. SNAP was developed by a team from Microsoft Research in collaboration with UC Berkeley and UC San Francisco (UCSF).
**Novoalign** [13] uses hash tables, which are created by dividing the reference genome into overlapping sequences. Novoalign then uses the Needleman-Wunsch algorithm to find the global optimum alignment during the mapping phase. The mapper was created by the company Novocraft, and it is the only one of the mentioned that it is commercial and not opensource.

### GEM3-GPU highlighted features

We would like to mention certain characteristics that are not common (or even present) in other GPU mappers; and where the GEM3-GPU specially shines. Most of them are important for a mapper targeting production environments with GPUs. The Chapter 9 will describe in

detail the features and their benefits.

### 10.2.3 Description of the datasets

This subsection describes all the datasets used for the experimentation, including (1) synthetic data (simulated using Mason [10]) and (2) real data. Both types of datasets include single-end and pair-end technologies based on Illumina. Other technologies as Iontorrent or Moleculo are included by completeness.

We consider Illumina HiSeq and MiSeq data as the most relevant and representative. Our analysis will focus on these data sets, since these sequencers are the most widely used in all research and production centers.

The main details for each of the workloads are included in the following table 10.1. It includes the accuracy reported by BWA-MEM using default parameters, which is our main reference for accuracy. This is valuable information that provides us with an early estimate of the complexity required to map each dataset.

The synthetic data is generated by Mason [10] with default parameters. Mason is a read simulator software that provides position specific error rates and base quality values. All the details are available in the aforementioned gem-bench repository. Therefore, the same data can be generated by execution of those specific scripts. The advantage of using synthetic data is that it allows us to unequivocally know which are the correct results that the application must generate. Having these golden data results, we can generate the ROC curves in detail, given that we have the original position of each synthetic read from the original genome reference. Using this method, we can compare the reference results with the benchmarked data and determine the true positives and true negatives for each read and MAPQ value reported.

The real data was provided by the National Genomics Center of Spain (CNAG) from real re-sequencing projects, and are described in more detail in the table. All the datasets were aligned against the reference CRCh37, which is a human reference genome. Notice that the amount of data generated per dataset is equivalent, in the sense that it contains the same number of nucleotides (5x coverage for Human Genome). This is a way to normalise the datasets between them, providing an insightful way to analyse the impact of the characteristics of each sequencing technology into the GEM mapper.

### 10.2.4 Description of the sequencing technologies

The process of sequencing can be performed in two different manners: pair-end and single-end. In pair-end sequencing, both ends of a DNA chain are sequenced simultaneously, which increases both accuracy and confidence. Paired end reads are helpful to detect rearrangements, repetitive sequences and insertion-deletions. However, this sequencing method produces twice the number of reads than single-end reading, and therefore the costs are higher and the running time is longer. On the other hand, single-end sequencing consists of reading the fragments from only one end of the reading In some studies that do not require the accuracy of paired-end sequencing, single-end reading can be considered as a convenient, economic and faster alternative.

**HiSeq**: is a high throughput sequencing system that provides high-quality data, either with single-end or paired-end sequencing. HiSeq can operate with large reads of up to 150bp and has run times of 7 - 10 days. HiSeq can run up to 6 human genomes simultaneously.

**MiSeq**: is an economic benchtop sequencer that is able to perform amplification, sequencing, base calling, alignment, variant calling and reporting. It can operate with reads of up to 300bp and has run times of between 4 and 55 hours. MiSeq is able to perform single end and paired-end sequencing and to run up to 24 bacterial genomes at the same times.

**Moleculo**: developed a technology for long read generation by combining a library prep method and genomic analysis tools. First, the DNA is fragmented and sequenced by Illumina platforms and then these are assembled into longer synthetic reads that contain haplotype information and can be used for genome finishing or de novo sequencing. The advantages are the high accuracy characteristic from Illumina sequencing methods combined with long reads of thousands bps.

**IonTorrent**: this sequencing method is based on detecting the hydrogen ions that are released every time that a nucleotide is incorporated into the DNA strand. The hydrogen ions produce a change in pH that is detected by the Ion torrent chip. The technology generates reads of 200-600bp and has the advantage of a competitive cost and short run-times. However, they wave a higher error rate than other sequencing methods such as Illumina technologies.

### 10.2.5 Description of metrics

This chapter explains the methodologies that are widely adopted by the community for performance and accuracy benchmarking. It is based on a work from Heng Li [8] who was heavily influenced by classification methodologies on statistics and AI fields.

| Dataset | Num Reads | Min size | Max Size | Average Size | Total MBases | BWA Sens. | BWA Spec. |
|---|---|---|---|---|---|---|---|
| **Synthetic Datasets** | | | | | | | |
| HiSeq SE Sim | 150M | 100 | 100 | 100 | 15000 | 100.00% | 98.06% |
| MiSeq SE Sim | 50M | 300 | 300 | 300 | 15000 | 100.00% | 99.19% |
| Illumina SimSE 500 | 30M | 500 | 500 | 500 | 15000 | 100.00% | 99.32% |
| Illumina SimSE 1000 | 15M | 1000 | 1000 | 1000 | 15000 | 100.00% | 99.50% |
| HiSeq PE Sim | 150M | 100 | 100 | 100 | 15000 | 99.99% | 98.41% |
| MiSeq PE Sim | 50M | 300 | 300 | 300 | 15000 | 99.80% | 99.08% |
| **Real Datasets** | | | | | | | |
| HiSeq SE Real | 150M | 100 | 100 | 100 | 15000 | 99.40% | - |
| MiSeq SE Real | 50M | 300 | 300 | 300 | 15000 | 98.86% | - |
| IonTorrent SE Real | 79M | 4 | 188 | 2716 | 15000 | 57.42% | - |
| Moleculo SE Real | 0.43M | 30 | 3817 | 19497 | 1660 | 99.62% | - |
| HiSeq PE Real | 150M | 98 | 100 | 100 | 14800 | 98.32% | - |
| MiSeq PE Real | 50M | 294 | 300 | 300 | 14703 | 98.41% | - |

**Table 10.1:** Synthetic and Real (Pair-end and Single-end) datasets for the experimentation

A more complete evaluation can be performed through all the pipelines. This evaluation is exposed in [12], where different mapper and variant calling combinations are shown for real WGS and WES pipelines.

#### Time and performance

The benchmarks evaluate the mapping throughput, the task latency and the peak memory used by the different mappers in each computing system. The following terminology is defined for the experimentation discussion.

- **Mapping Throughput**: Millions of pair bases (bps) mapped per second, computed as the total number of mapped queries multiplied by its size (number of pair bases) and divided by the total elapsed time in seconds.
- **Peak of memory**: Maximum resident set size (max RSS) of the running process. It represents the maximum amount of physical memory used at any instant of the execution.

- **GPU overhead**: the GPU and the CPU workflows used by GEM3 are different by design. Both versions report exactly the same results, but the GPU workflow executes more work than the CPU workflow. GPU overhead is defined as the time devoted to the extra work executed by the GPU workflow.
- **Init-End overhead**: Execution time (seconds) for all the necessary tasks done before and after to read mapping and alignment of all the reads. It includes all the tasks with constant time cost, which is independent to the number of reads. Examples of those tasks are reading the input sequences and index, initialisation of data structures, memory pooling creation, freeing resources at the end, etc
- **I/O overhead**: This metric tries to isolate and identify the overhead time (seconds) dedicated to all the tasks that are unrelated to the effective compute in order to obtain the final results. E.g., CPU thread scheduling, task allocation from input file, parsing of the input file, writing the final results on disk... This is equivalent to execute the mapper and process the full input file without performing any useful computation.

### Measuring accuracy

We measure the accuracy along different datasets and mappers

- **Sensitivity**: Percentage of reads reported by the mapper, out of the number of reads in the entire input data set, that are mapped at any location in the genome. Defined as (TP + FP) / total reads, where TP = True Positives and FP = False Positives.
- **Specificity**: Percentage of reads reported by the mapper, out of the number of reads in the entire input data set, that are mapped at the correct location in the genome. Defined as TP / total reads.
- **MAPQ**: It stands for MAPping Quality. A mapping quality is basically the probability that a read is aligned in the wrong place. The probability is calculated as a values between 0 and 255, rounded to the nearest integer:

  $$!p = 10^{-q/10}$$

  These values are critical for the next steps of the pipeline after the alignment (variant calling). This data provides additional information for better confidentiality of the

results, allowing prune alignments by their quality reported on the SAM file. It allows bioinformaticians to customise the analysis according to the quality of the input data that has been obtained by the sequencer. The most important ranges are 20 and 30, because they are the default values that are used by most variant callers by default; less likely assignments are discarded. We will focus our final analysis on this ranges.

- **ROC curves**: (Receiver Operating Characteristic) is a graphic plot that shows the diagnostic ability of binary classifiers. The curve represents two parameters: a trade-off between the TP and FP for different MAPQ values. Values located under the ROC curve are interpreted as better results. ROC curves not only help us to identify if the mapper has better specificity, but also provide additional information regarding the false positive results in the final SAM file, giving feedback of how much noise the mapper introduces in the final results.

### 10.2.6 Experimentation reproducibility

In order to guarantee the reproducibility of the experimentation and to provide access to the work to other researchers in the field; all the tools, scripts, programs and data generated have been released and are available publicly.

The latest version of all the software can be found in the following repositories:

- git-gem3: Gem3-GPU, batched mapper that integrates all the GPU modules.
  https://github.com/achacond/gem3-mapper
- git-cutter: GEM-Cutter, GPU core-library that includes all the basic block GPU algorithms as modules and schedulers that can be used in different bio applications thought their general API. The modules are described in the previous chapter.
  https://github.com/achacond/gem-cutter
- git-gem-bench: Gem-Bench, automated framework using the methodology applied in this chapter that includes all the workflows, scripts and data used for a mapping evaluation.
  https://github.com/achacond/gem-gpu-benchmark

For reproducibility we recommend to use the mentioned Gem-Bench project, that was specifically developed on this thesis to carry out all the (1) compilation and installation of the tools;

(2) downloading and preparation of the datasets, (3) execution of the experimentation and (4) profiling and representation of the final results.

### 10.2.7 Final notes on used methodology

It is crucial to understand that all the mappers included on the experimentation are considered state-of-the-art, but not all of them would be considered production-grade. Next, we will describe some limitations that we identified on different mappers that promoted us to adapt our benchmarking methodology.

- Some mappers, specially those that are GPU-accelerated, exhibit restrictions on the size of the input file (FASTQ) or the query size. We realised that some of them just consider a maximum read size as worst case, performing a controlled exit of the application, while others just simply crash. It is difficult to develop a GPU-accelerated mapper that scales with the read size. For proper execution, mappers have been recompiled for new CPU-GPU architectures, the hard-coded constants have been changed to allow larger read sizes and we have identified in advance the maximum read size from the input file.

- Some mappers crash or never finish the execution when real datatasets are processed. In these cases we have isolated and then extracted from the original dataset the few specific reads that generate errors for that mapper. The impact on execution time is negligible, since a very low percentage of the reads are affected. They are included on the output file as non-mapped reads to keep consistency with the rest of the methodology.

- MAPQ values are very important for the following stages of the genomic pipeline. There is a large discrepancy in the community on how they have to be assigned. This can be observed on the ROC curve experimentation on the following experimental sections.

- Many mappers (CPU and GPU) do not report in the output file (SAM) the reads that could not be mapped to the reference. To be consistent with our methodology, the non-reported reads are included offline into the SAM file, and assigned a MAPQ value equal 0.

- Some mappers do not keep the same order of reads from the input file (FASTQ) in the output file (SAM). This order is important not only for benchmarking and

comparison purposes, also for the next stages on the pipeline, where there are tools that assume the same order on FASTQ and SAM files.

The previous limitations make a mapper implementation unfeasible for the restrictions of production environments.

## 10.3 General overview on performance results

This section overviews the main performance results of all the presented proposals, integrated into a real end-to-end mapper application with GPU support. It is evaluated on a production environment and compared with the mapper considered the reference in the field, BWA-MEM.

All the experiments use the simulated Illumina-like datasets described on Table 10.1. The evaluation covers two different computer systems to characterise the behaviour of the mapper: (a) a high-performance node and (b) a low-power embedded system. Due to their different form factor sizes, computational requirements and power consumption, both systems are considered suitable realistic computational environments for the diverse high-throughput sequencing scenarios.

Figures 10.1.a and 10.1.b show the throughput and speedup compared to BWA-MEM on a high-performance system (p9-volta). Our proposal provides an overall throughput between 22.8x and 39.4x higher than BWA-MEM; using GPU acceleration provides performance that is 3.2x to 29.1x higher than GEM without GPU acceleration.

Figures 10.2.a and 10.2.b show the throughput and speedup compared to BWA-MEM on a low-power embedded system (arm-maxwell). The performance evaluation reports between

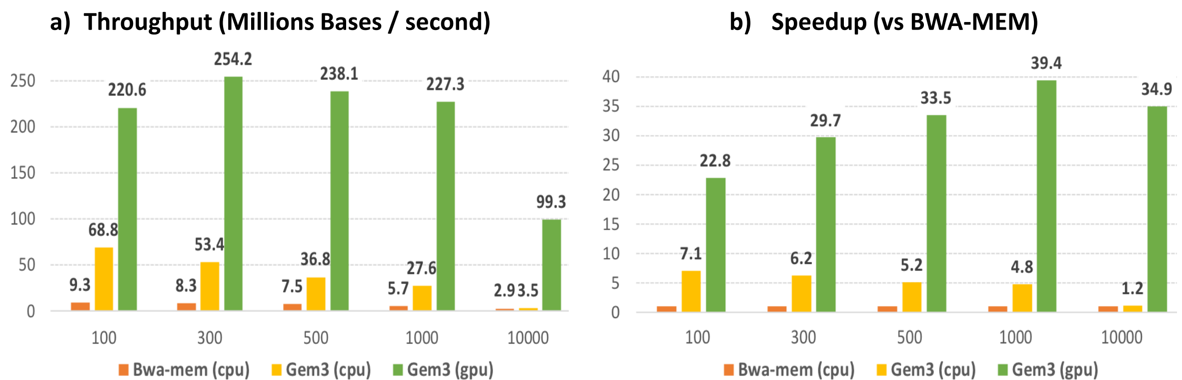


**Figure 10.1:** Performance speedup of GEM3, using only CPU and using GPU-acceleration, compared to BWA-MEM (CPU-only) on a typical HPC system (P9+Volta)

7.2 and 10.2 times the performance of BWA-MEM. In this scenario, using GPU acceleration provides performance that is 2.1x to 2.7x higher than GEM without GPU acceleration. This benchmark is using half of the human genome for practicability; GPU and CPU memory share the same 8 GBytes of LPDDR4 modules; the full data structures doesn't fit in memory; and lower index sampling is used in order to increase the compression of the index, at the expense of increasing computation work.

### 10.3.1 Query size and thread scalability

The following experimentation aims to analyse in more detail the performance and behaviour of the GEM3 mapper, performing a characterization with different sets of synthetic data.

The input data is specifically generated Using the Mason simulator to better understand the behaviour of the mapper in different scenarios. We check the (1) impact on performance of using different read sizes, or query size scalability (100nt, 1Knt, 10Knt), and the (2) effect on performance of increasing the number of CPU threads running the mapper application, or multithreading scalability. We will evaluate performance (Mbases/s), speedup compared to BWA-MEM, thread scalability and efficiency. All the performance metrics are normalised across the different datasets.

**Overall performance**

Figure 10.3.a shows that GEM-GPU has persistently a better performance along all the cases compared to the CPU mapper versions providing speedups from 18x to 36x compared to the best configuration runs for BWA and GEM3-CPU. GEM3-GPU can achieve 312Mbases/s with a single GPU on short reads (100nt) and 248Mbases/s (10Knt). Last but not least, GEM3-CPU achieved significant speedups from 8,3x (100nt) to 1,5x (10Knt) compared to BWA.

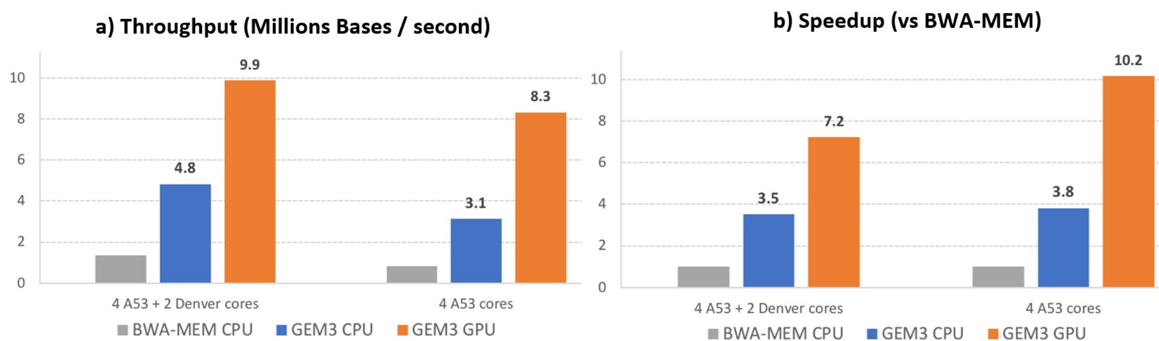


**Figure 10.2:** Performance speedup of GEM3, using only CPU and using GPU-acceleration, compared to BWA-MEM (CPU-only) on a typical embedded system (Tegra TX2)

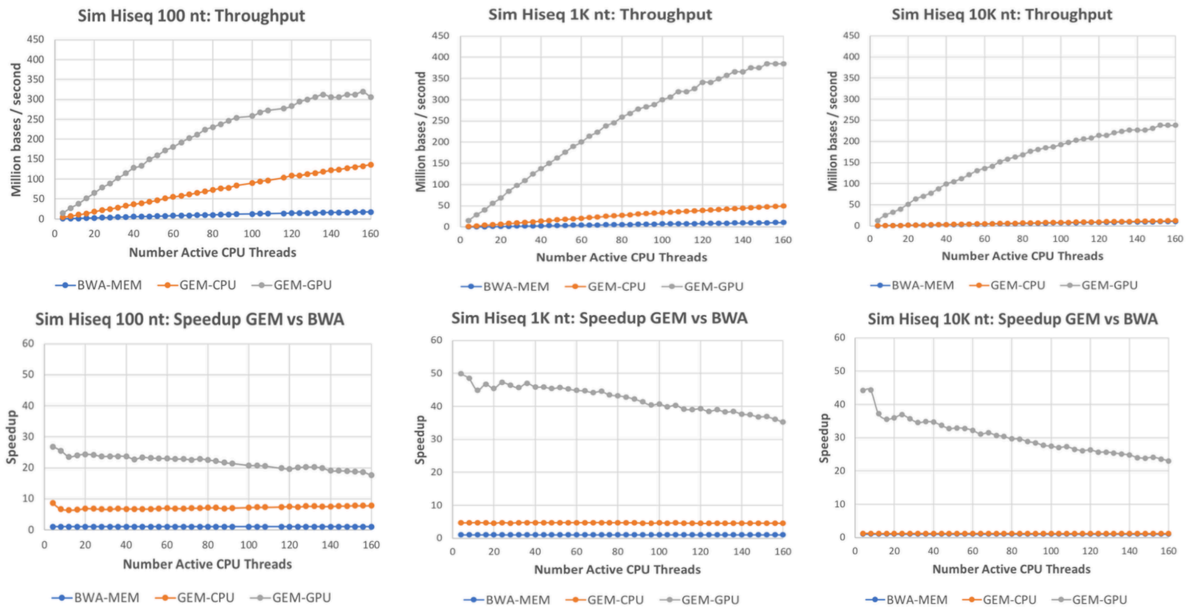


**Figure 10.3:** (a) Performance and (b) speedup comparison of GEM3-CPU and GEM3-GPU compared to BWA-MEM, on the P9+V100 CPU+GPU platform.

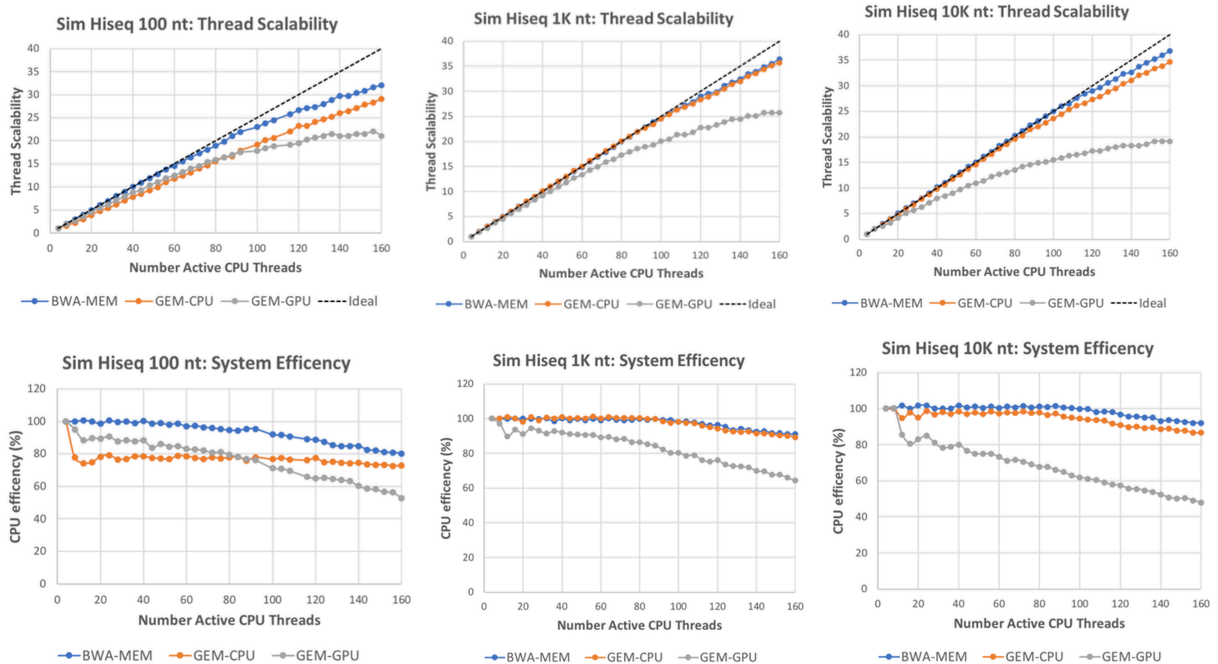


**Figure 10.4:** (c) Scalability and (d) efficiency of GEM3-CPU and GEM3-GPU compared to BWA-MEM, on the P9+V100 CPU+GPU platform.

**Query scalability**
We can observe that the increase in the query size from 100nt to 10Knt results in scalability performance degradations of 8.4x (GEM3-GPU), 8.4x (GEM3-CPU), and 1.4x (BWA). GEM3-GPU and BWA-mem scale quite well, being the GEM3-GPU the mapper with better scaling with the query length. GEM-CPU presents an order of magnitude performance degradation at the expense of increase of the query size.

**Thread scalability**
It is important to note that the benchmarked system ppc-volta is an in-order execution platform, requiring a high aggressive SMT per core to increase the utilization of their resources effectively. In general, CPU mappers expose an almost linear scalability increasing the number of threads. Figure 10.3.c shows that speedups between GEM3-CPU and BWA are maintained constant when increasing the number of threads. Additionally, increasing the query size improves the thread scalability of the CPU mappers.

**Execution efficiency**
Figure 10.4.d depicts that the overall efficiency for CPU mappers on the ppc-volta platform is quite high, from 80% to 94% on an scenario of maximum thread occupancy. However, BWA shows better efficiency than GEM3 across all cases. We can observe that, by increasing the read size, the efficiency increases consistently over all CPU platforms. On the other hand, the GPU version clearly has less efficiency, given that the GPU results show an efficiency of around 50%. Future work will explore the reasons for this, which could include the insufficient task parallelism for the GPU to cope the performance.

This is relevant for pipelines that utilise cloud services (on demand executions), due to the advantages of reduced cost by saturating GPU performance with less CPU threads, allowing to reduce economical costs.

## 10.4 Detailed comparison with the state-of-art

### 10.4.1 ROC curves analysis

The next section is a broad accuracy evaluation of all the mappers described on section 10.2.2 and analysed on Figure 10.5. The experimentation shows the accuracy for all the MAPQ mapping values using the datasets Sim.Hiseq (read length 100nt) for single- and pair-end on the platform x86-kepler and running with the same default parameters than section 10.4. This experimentation will provide insightful information of how confident the output results of

| | Real | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Single-end | | | | | | | | Paired-end | | | |
| | HiSeq (100) | | MiSeq (300) | | IonTorrent | | Moleculo | | HiSeq (100) | | MiSeq (300) | |
| 1st | GEM3(GPU) | 285s | GEM3(GPU) | 638s | HPG-Aligner | 999s | GEM3(GPU) | 313s | GEM3(GPU) | 297s | GEM3(GPU) | 1884s |
| 2nd | Soap3 | 503s | nvBowtie | 1157s | GEM3(GPU) | 1178s | HPG-Aligner | 325s | GEM3(CPU) | 834s | nvBowtie | 2287s |
| 3rd | GEM3(CPU) | 648s | HPG-Aligner | 1230s | GEM3(CPU) | 1659s | SNAP | 975s | SOAP3 | 857s | HPG-Aligner | 2287s |
| 4th | SNAP | 812s | GEM3(CPU) | 1237s | BWA | 3462s | BWA | 2305s | SNAP | 1617s | GEM3(CPU) | 2404s |
| 5th | nvBowtie | 1361s | Soap3 | 1402s | Bowtie2 | 3667s | GEM3(CPU) | 2501s | BWA | 3390s | Soap3 | 2409s |

| | Simulated | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Single-end | | | | | | | | Paired-end | | | |
| | HiSeq (100) | | MiSeq (300) | | Illumina (500) | | Illumina (1000) | | HiSeq (100) | | MiSeq (300) | |
| 1st | GEM3(GPU) | 224s | GEM3(GPU) | 204s | GEM3(GPU) | 240s | GEM3(GPU) | 264s | GEM3(GPU) | 361s | GEM3(GPU) | 334s |
| 2nd | GEM3(CPU) | 571s | SNAP | 736s | HPG-Aligner | 736s | HPG-Aligner | 474s | SNAP | 589s | SNAP | 384s |
| 3rd | SOAP3 | 854s | GEM3(CPU) | 747s | GEM3(CPU) | 747s | GEM3(CPU) | 1337s | GEM3(CPU) | 720s | GEM3(CPU) | 813s |
| 4th | nvBowtie | 1313s | HPG-Aligner | 1066s | SNAP | 1016s | SNAP | 2238s | Soap3 | 1190s | HPG-Aligner | 1102s |
| 5th | SNAP | 2065s | Soap3 | 1172s | nvBowtie | 1151s | BWA | 3885s | HPG-Aligner | 2140s | Soap3 | 2665s |

**Table 10.2:** Overview of the top 5 mappers with best performance for all the datasets (Synthetic and Real; Pair-end and Single-end)

GEM3 are compared with the state-of-art.

We are using the ROC curves methodology described in sections 10.2.7. Figures 10.5.a and 10.5.b present single-end and pair-end executions respectively. In overall, both figures show 2 sets of mappers that provide very different results. In particular, novoalign, bwa-mem and GEM3 are outperforming the rest of the mappers both in accuracy and reduction of false positives. From a performance point of view, in overall, GEM3 using GPU acceleration is higher than an order of magnitude faster than the rest of CPU or GPU mappers. Compared with novoalign, the best-performing accuracy mapper, we can observe that GEM3 is 27.8x and 33.7x faster, on single- and pair-end datasets respectively.

### Breaking down the accuracy results

Analyzing the single-end dataset we can see that GEM3 (fast mode) is reporting slightly better accuracy than Bwa-mem and gem (default) can provide 92% of true positive mappings with less than 10-5 percentage of false positives in the dataset. In addition, novoalign and gem (sensitive) can provide higher accuracy >93.5% with a similar number of false positives. In case of the other mappers analysed, in order to reach similar accuracy results they perform between 1 to 2 orders of magnitude more false positives, meanwhile the best performance mappers at that point can reach 94.0% – 94.5%.

The pair-end datasets show that, in general, mappers provide higher accuracy with a similar number of false positives than single-end mappers. As expected, the more contextual

| | Real (sensitivity) | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Single-end | | | | | | | | Paired-end | | | |
| | HiSeq (100) | | MiSeq (300) | | IonTorrent | | Moleculo | | HiSeq (100) | | MiSeq (300) | |
| 1st | BWA | 99.40 | BWA | 98.86 | BWA | 57.42 | BWA | 99.62 | GEM3 | 98.40 | BWA | 98.41 |
| 2nd | HPGAligner | 99.13 | HPGAligner | 98.24 | GEM3 | 53.31 | GEM3 | 99.21 | Soap3 | 98.34 | GEM3 | 97.63 |
| 3rd | GEM3 | 98.68 | GEM3 | 94.54 | HPGAligner | 39.54 | Bowtie2 | 99.12 | BWA | 98.32 | HPG-Aligner | 92.86 |
| 4th | nvBowtie | 97.22 | Novoalign | 93.58 | SNAP | 29.49 | HPGAligner | 99.06 | nvBowtie | 97.22 | Novoalign | 96.85 |
| 5th | SNAP | 97.20 | Soap3 | 85.69 | Novoalign | 27.27 | SNAP | 91.55 | Bowtie2 | 96.84 | nvBowtie | 54.15 |

| | Simulated (specificity) | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Single-end | | | | | | | | Paired-end | | | |
| | HiSeq (100) | | MiSeq (300) | | Illumina (500) | | Illumina (1000) | | HiSeq (100) | | MiSeq (300) | |
| 1st | HPG-Aligner | 98.99 | GEM3 | 99.76 | GEM3 | 99.86 | GEM3 | 99.94 | HPGAligner | 99.60 | GEM3 | 99.88 |
| 2nd | GEM3 | 98.77 | BWA | 99.32 | BWA | 99.19 | BWA | 99.50 | GEM3 | 99.58 | BWA | 99.08 |
| 3rd | BWA | 98.07 | Soap3 | 99.07 | Soap3 | 99.07 | Soap3 | 99.39 | Soap3 | 99.39 | CUSHAW | 98.62 |
| 4th | Soap3 | 98.44 | HPGAligner | 98.96 | Cushaw | 98.58 | Bowtie2 | 98.90 | BWA | 98.41 | HPGAligner | 98.61 |
| 5th | SNAP | 96.41 | SNAP | 98.20 | nvBowtie | 98.53 | Novoalign | 98.23 | nvBowtie | 97.74 | nvBowtie | 98.27 |

**Table 10.3:** Overview of the top 5 mappers with best accuracy for all the datasets (Synthetic and Real; Pair-end and Single-end)

information provided by the pair-end linked reads allows to solve ambiguous cases. Highlighting the differences with the single-end dataset, we can observe that cushaw2 and nvbowtie perform much better on pair-end datasets, meanwhile SNAP loses accuracy and introduces a lot of false positives.

Notice that mappers such as HPC and SOAP3dp-GPU are reporting a very limited number of MAPQ values providing low sensitive ROC curves, in some cases resulting in only 2 different data points, which makes it difficult to perform fair comparisons.

### A practical view from MAPQ values

ROC curves help to normalise results and to visually identify the noise introduced in the final results (due to the reported false positives). The following graphics allow the comparison of specific relevant MAPQ points between mappers. For example, the next graphics show the MAPQ 20 (99.0%) and 30 (99.9%) of mapping confidence, represented as O and X respectively. These values are of vital importance, given that the majority of bioinformatic applications use them as default parameters. Therefore, they only consider these results and discard the MAPQ with different or lower MAPQ, as they are not considered reliable enough. In both graphics we can see three different groups for the same MAPQ values.

- In the case of single-end, we can see that novoalign, BWA and GEM3 are in the range of 10-5 and 10-4, having a number of false positive errors one order of magnitude

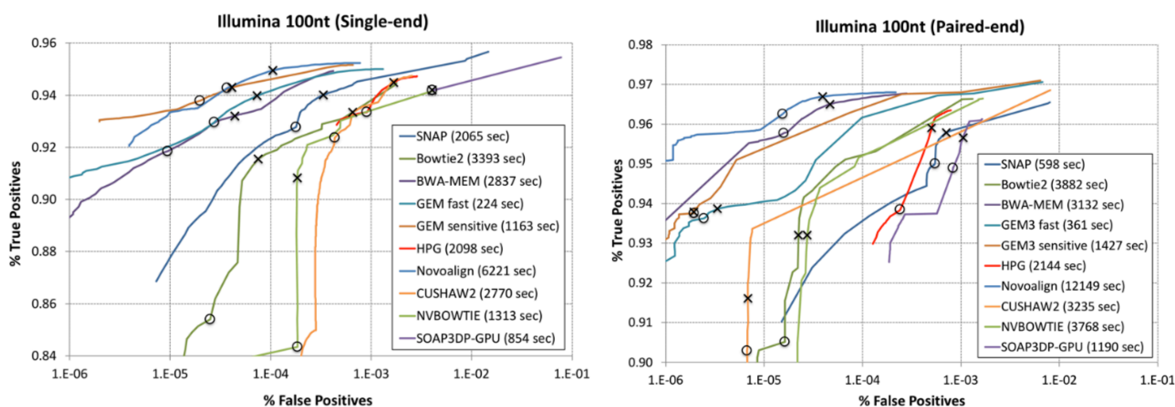


**Figure 10.5:** ROC Curves: Performance and Accuracy comparison Illumina Sim. single- and pair-end (100nt)

less than SNAP, Bowtie, nvBowtie and cushaw2; and 2 orders of magnitude less than SOAP3dp-gpu and HPG. All the groups show a correlation of 0.01 and 0.02 percentage points in the specificity per group.

- In the case of pair-end, we can see that GEM3 reports a very low number of false positives, in the range of 10-6 and 10-5 outperforming the rest of the mappers and being very conservative in the level of specificity reported. Mappers as bwa-mem and novoalign outperform the rest of the mappers in accuracy, scarifying an order of magnitude of false positives in the signal. The rest of the mappers, such as HPG, SNAP and soap3dp-gpu, show less specificity than bwa-mem and novoalign at same time that report 3 orders of magnitude more false positives. Cushaw2 and bowtie are in the bottom accuracy list.

Specifically, if we compare the pair- and single-end datasets reported by GEM3, we can observe that both of them report a very similar specificity for both MAPQ points, with a difference of one order of magnitude on false positives, showing that pair-end provides a signal with much less noise or uncertainty in the mapping.

## 10.5 Conclusions

The presented data shows that GEM3 provides a very competitive accuracy for single- and pair-end datasets, and also very low false positives for the most widely-used MAPQ values in the community. It is worthy to mention that a collaboration with CNAG was published on

Journal of Human Mutations (HUMU, [12]), with a validation of all the pipeline from wet lab to clinical mutations, showing benchmarking on production infrastructures and validating the final results with experts on bioinformatics. Comparison with BWA was carried out, and testing the intersection with different pipelines as freebayes or GATK.

We would like to show our gratitude to Novocraft to provide us a free license for all the evaluations of their mapper in this work. We also want to thank CNAG for providing support and feedback for the generation of the datasets and the real subset for all the validations.

| Mapper | Configuration | HiSeqSE.Sim [Bases: 15000 Mbases] [R: 150M; S: (100, 100, 100)] CGA (CNAG) Seconds | Mbytes | QUALITIES % Sen | % Spe | MiSeqSE.Sim [Bases: 15000 Mbases] [R: 50M; S: (300, 300, 300)] CGA (CNAG) Seconds | Mbytes | QUALITIES % Sen | % Spe | Illumina.SimSE.I500 [Bases: 15000 Mbases] [R: 30M; S: (500, 500, 500)] CGA (CNAG) Seconds | Mbytes | QUALITIES % Sen | % Spe |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bowtie2 v2.3.4 (16th) | very-fast | 1753 | 3606 | 98.9646 | 94.9326 | 1867 | 3645 | 99.3317 | 96.8288 | 8830 | 3802 | 99.7317 | 97.6296 |
| | default | 3393 | 3624 | 99.8326 | 96.0615 | 4061 | 3680 | 99.9798 | 98.0731 | 20535 | 3850 | 99.9957 | 98.4736 |
| | very-sensitive | 7411 | 3694 | 99.9254 | 96.2527 | 9750 | 3822 | 99.9891 | 98.1527 | 40311 | 4004 | 99.9906 | 98.5236 |
| BWA v0.7.17 (r1140) (16th) | default (300) | 2837 | 7845 | 100.0000 | 98.0689 | 3006 | 6739 | 100.0000 | 99.1908 | 3207 | 6767 | 100.0000 | 99.3284 |
| | c=500 | 2843 | 7855 | 100.0000 | 98.0689 | 2995 | 6739 | 100.0000 | 99.1908 | 3224 | 6766 | 100.0000 | 99.3284 |
| | c=1000 | 3104 | 8090 | 100.0000 | 98.0710 | 3124 | 6739 | 100.0000 | 99.1917 | 3366 | 6766 | 100.0000 | 99.3288 |
| | c=16000 | 7057 | 8830 | 100.0000 | 98.0738 | 5125 | 6769 | 100.0000 | 99.1922 | 4332 | 6772 | 100.0000 | 99.3290 |
| | c=20000 k=16 | 12728 | 9544 | 100.0000 | 98.0603 | 9138 | 7147 | 100.0000 | 99.1842 | 7271 | 6804 | 100.0000 | 99.3230 |
| CUSHAW2 v2.1.8-r16 (16th, K80 single-GPU) | default | 2770 | 4763 | 99.6433 | 96.1412 | 6248 | 5398 | 99.6021 | 97.8488 | 9418 | 6064 | 100.0000 | 98.5803 |
| | sensitive | 2744 | 4763 | 99.9507 | 96.3647 | 6260 | 5398 | 100.0000 | 98.2400 | 9433 | 6064 | 100.0000 | 98.5803 |
| GEM3-GPU git (06.18) (16th; K80 multi-GPU) | default CPU | 571 | 13724 | 99.8165 | 98.7672 | 747 | 11407 | 99.9797 | 99.7656 | 1014 | 11415 | 99.9943 | 99.8691 |
| | default GPU | 224 | 14309 | 99.8165 | 98.7672 | 204 | 13385 | 99.9797 | 99.7656 | 240 | 13322 | 99.9943 | 99.8691 |
| | sensitive CPU | 1573 | 14022 | 99.9922 | 98.9907 | 911 | 12417 | 99.9984 | 99.7944 | 1116 | 12120 | 99.9987 | 99.8802 |
| | sensitive GPU | 1163 | 15640 | 99.9922 | 98.9907 | 323 | 15640 | 99.9984 | 99.7944 | 296 | 14954 | 99.9987 | 99.8802 |
| HPG-aligner v2.0.1 (16th) | fast | 2031 | 28416 | 99.6441 | 98.9919 | 828 | 28294 | 99.3862 | 99.0682 | 568 | 28281 | 98.9732 | 98.5881 |
| | default | 2098 | 28423 | 99.6231 | 98.9920 | 1066 | 28297 | 99.2340 | 98.9616 | 783 | 28281 | 98.8160 | 98.4916 |
| | sensitive | 2093 | 28425 | 99.6231 | 98.9920 | 1433 | 28297 | 99.0484 | 98.7771 | 1154 | 28322 | 98.5056 | 98.1868 |
| nvBowtie v1.1.0 (K80 multi-GPU) | very-fast | 987 | 7945 | 99.8822 | 95.2311 | 816 | 8579 | 99.9855 | 97.9740 | 771 | 7964 | 99.9968 | 98.4271 |
| | default | 1313 | 7945 | 99.9424 | 95.8555 | 1151 | 8443 | 99.9966 | 98.1418 | 1156 | 7964 | 99.9996 | 98.5318 |
| | very-sensitive | 2046 | 7872 | 99.9353 | 95.7717 | 1995 | 8443 | 99.9959 | 98.1258 | 1908 | 7964 | 99.9996 | 98.5217 |
| SNAP v1.0 beta 18 (16th) | h50 | 812 | 37102 | 99.1970 | 95.8339 | 541 | 36410 | 99.9445 | 98.1461 | 524 | 36776 | 99.6776 | 98.2573 |
| | default (h300) | 2065 | 37102 | 99.6435 | 96.4105 | 736 | 36714 | 99.9702 | 98.2094 | 1016 | 36912 | 99.6899 | 98.2814 |
| | h1000 | 2162 | 37715 | 99.8162 | 96.5675 | 3200 | 35734 | 99.9766 | 98.2223 | 1844 | 37293 | 99.6916 | 98.2869 |
| | h2000 | 4662 | 37774 | 99.9252 | 96.6175 | 5722 | 36294 | 99.9781 | 98.2262 | 2924 | 37678 | 99.6920 | 98.2888 |
| SOAP3-dp GPU 2.3.r180 (16th; K80) | default | 854 | 12472 | 99.0700 | 98.4444 | 1172 | 20723 | 99.8334 | 99.5530 | 1718 | 25770 | 99.3503 | 99.0712 |
| NovoAlign V3.04.06 (16th) | default | 6221 | 8337 | 95.3074 | 95.2339 | 4770 | 8353 | 97.2817 | 97.2728 | 6102 | 8402 | 97.7374 | 97.7309 |

| Mapper | Configuration | Illumina.SimSE.I1000 [Bases: 15000 Mbases] [R: 15M; S: (1000, 1000, 1000)] CGA (CNAG) Seconds | Mbytes | QUALITIES % Sen | % Spe | H.Sapiens.HiSeqPE.Sim [Bases: 15000 Mbases] [R: 150M; S: (100, 100, 100)] CGA (CNAG) Seconds | Mbytes | QUALITIES % Sen | % Spe | H.Sapiens.MiSeqPE.Sim [Bases: 15000 Mbases] [R: 50M; S: (300, 300, 300)] CGA (CNAG) Seconds | Mbytes | QUALITIES % Sen | % Spe |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bowtie2 v2.3.4 (16th) | very-fast | 25405 | 4392 | 99.8253 | 98.3799 | 2797 | 3709 | 99.8128 | 97.4813 | 6903 | 3834 | 98.6336 | 96.9530 |
| | default | 80832 | 4392 | 99.9945 | 98.9008 | 3882 | 3752 | 99.9612 | 97.7218 | 7584 | 3902 | 98.7317 | 97.2916 |
| | very-sensitive | 127756 | 4584 | 99.9779 | 98.9203 | 6950 | 3864 | 99.9850 | 97.7899 | 11801 | 4109 | 98.7361 | 97.3560 |
| BWA v0.7.17 (r1140) (16th) | default (300) | 3885 | 6710 | 100.0000 | 99.5089 | 3132 | 7929 | 99.9984 | 98.4102 | 3041 | 6697 | 99.9998 | 99.0843 |
| | c=500 | 3868 | 6710 | 100.0000 | 99.5089 | 3132 | 7915 | 99.9984 | 98.4102 | 3040 | 6696 | 99.9998 | 99.0843 |
| | c=1000 | 4004 | 6710 | 100.0000 | 99.5089 | 3439 | 8142 | 99.9985 | 98.4117 | 5274 | 6985 | 99.9998 | 99.0847 |
| | c=16000 | 4189 | 6710 | 100.0000 | 99.5090 | 7360 | 8843 | 99.9987 | 98.4138 | 5165 | 6866 | 99.9998 | 99.0844 |
| | c=20000 k=16 | 6269 | 6712 | 100.0000 | 99.5061 | 13015 | 9520 | 99.9987 | 98.4110 | 9124 | 7231 | 99.9998 | 99.0835 |
| CUSHAW2 v2.1.8-r16 (16th, K80 single-GPU) | default | Execution Fails | | | | 3235 | 4957 | 98.8854 | 96.8531 | 6754 | 5855 | 99.9638 | 98.6261 |
| | sensitive | | | | | 3167 | 4943 | 98.9341 | 96.9005 | 6755 | 5823 | 99.9638 | 98.6261 |
| GEM3-GPU git (06.18) (16th; K80 multi-GPU) | default CPU | 1337 | 11453 | 99.9992 | 99.9431 | 720 | 11505 | 99.9879 | 99.5840 | 813 | 11445 | 99.9969 | 99.8857 |
| | default GPU | 267 | 13470 | 99.9992 | 99.9431 | 361 | 14068 | 99.9879 | 99.5840 | 334 | 13696 | 99.9969 | 99.8857 |
| | sensitive CPU | 1374 | 12219 | 99.9996 | 99.9474 | 1856 | 13201 | 100.0000 | 99.5598 | 920 | 13234 | 99.9999 | 99.8940 |
| | sensitive GPU | 282 | 14840 | 99.9996 | 99.9474 | 1427 | 17737 | 100.0000 | 99.5598 | 867 | 16769 | 99.9999 | 99.8940 |
| HPG-aligner v2.0.1 (16th) | fast | 400 | 28242 | 98.2068 | 97.7900 | 2063 | 28298 | 99.8165 | 99.6090 | 855 | 28298 | 99.8196 | 98.6081 |
| | default | 474 | 28227 | 97.9814 | 97.6214 | 2144 | 28437 | 99.8151 | 99.6097 | 1102 | 28437 | 99.8277 | 98.6185 |
| | sensitive | 690 | 28255 | 97.6799 | 97.3307 | 1478 | 28741 | 99.8151 | 99.6097 | 1478 | 28741 | 99.8306 | 98.6154 |
| nvBowtie v1.1.0 (K80 multi-GPU) | very-fast | Max read size to 512 | | | | 2765 | 10265 | 99.9682 | 97.6379 | 2982 | 11407 | 99.6265 | 98.2358 |
| | default | | | | | 3768 | 10265 | 99.9908 | 97.7444 | 4501 | 11405 | 99.6288 | 98.2708 |
| | very-sensitive | | | | | 5619 | 10265 | 99.9904 | 97.7170 | 8023 | 11407 | 99.6286 | 98.2622 |
| SNAP v1.0 beta 18 (16th) | h50 | 811 | 36546 | 91.2564 | 90.2964 | 631 | 37033 | 98.4991 | 96.4709 | 473 | 36453 | 99.0575 | 97.7325 |
| | default (h300) | 2238 | 36761 | 91.2637 | 90.3036 | 589 | 37133 | 98.9956 | 96.9212 | 384 | 36579 | 99.2138 | 97.8828 |
| | h1000 | 3565 | 36761 | 91.2643 | 90.3065 | 1930 | 37444 | 99.9847 | 97.8512 | 3180 | 36442 | 99.4686 | 98.1370 |
| | h2000 | 3849 | 37707 | 91.2643 | 90.3085 | 2029 | 37625 | 99.9877 | 97.8541 | 3577 | 36841 | 99.4688 | 98.1372 |
| SOAP3-dp GPU 2.3.r180 (16th; K80) | default | 18640 | 39496 | 99.4802 | 99.3928 | 1190 | 13277 | 99.5868 | 99.3930 | 2665 | 23088 | 17.7143 | 17.4396 |
| NovoAlign V3.04.06 (16th) | default | 9611 | 8572 | 98.2350 | 98.2303 | 12149 | 8266 | 96.8175 | 96.7965 | 178403 | 8681 | 97.8633 | 97.8579 |

**Figure 10.6:** Performance and Accuracy comparison with the most relevant mappers on the state-of-the-art using simulated data

| Mapper | Configuration | HiSeqSE.Real [Bases: 15000 Mbases] [R: 150M; S: (100, 100, 100)] | | | MiSeqSE.Real [Bases: 15000 Mbases] [R: 50M; S: (300, 300, 300)] | | | IonTorrentSE.Real [Bases: 15000 Mbases] [R: 79M; S: (4, 188, 2716)] | | | MoleculoSE.Real [Bases: 1660 Mbases] [R: 0.43M; S: (30, 3817, 19497)] | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | CGA (CNAG) Seconds | Mbytes | QUALITY % Sen | CGA (CNAG) Seconds | Mbytes | QUALITY % Sen | CGA (CNAG) Seconds | Mbytes | QUALITY % Sen | CGA (CNAG) Minutes | Mbytes | QUALITY % Sen |
| Bowtie2 v2.3.4 (16th) | very-fast | 1771 | 3606 | 96.0296 | 1466 | 3636 | 60.7852 | 1312 | 5818 | 28.3227 | 134251 | 49969 | 98.9969 |
| | default | 3469 | 3622 | 96.9959 | 3163 | 3669 | 62.2318 | 3667 | 5937 | 29.4939 | 380706 | 50408 | 99.1221 |
| | very-sensitive | 7310 | 3690 | 97.2748 | 6829 | 3793 | 62.5437 | 7146 | 6121 | 29.7112 | 794802 | 51188 | 99.1323 |
| BWA v0.7.17 (16th) | default (300) | 2890 | 8517 | 99.4004 | 3766 | 7556 | 98.8656 | 3462 | 8971 | 57.4209 | 2305 | 6582 | 99.6276 |
| | c=500 | 2876 | 8402 | 99.4004 | 3766 | 7569 | 98.8656 | 3451 | 8987 | 57.4209 | 2296 | 6578 | 99.6276 |
| | c=1000 | 3012 | 8690 | 99.4004 | 3867 | 7565 | 98.8656 | 3541 | 9002 | 57.4210 | 2309 | 6578 | 99.6276 |
| | c=16000 | 3416 | 9076 | 99.4004 | 3921 | 7573 | 98.8656 | 3733 | 9058 | 57.4210 | 2327 | 6578 | 99.6276 |
| | c=20000 k=16 | 4877 | 9640 | 99.4435 | 6897 | 10716 | 98.9310 | 10432 | 19196 | 57.6958 | 3121 | 6584 | 99.6286 |
| CUSHAW2 v2.1.8-r16 (16th, K80) | default | 2428 | 4802 | 96.6252 | More than 80h of execution | | | Execution Fails | | | Execution Fails | | |
| | sensitive | 2389 | 4800 | 98.8269 | | | | | | | | | |
| GEM3-GPU git (06.18) (16th; K80) | default CPU | 648 | 11441 | 98.6821 | 1237 | 11521 | 98.2439 | 1659 | 12373 | 53.3166 | 2501 | 12108 | 99.2157 |
| | default GPU | 285 | 13419 | 98.6821 | 638 | 15945 | 98.2439 | 1178 | 15623 | 53.3166 | 313 | 25718 | 99.2157 |
| | sensitive CPU | 6001 | 11849 | 98.9966 | 62109 | 13329 | 98.2124 | 4104 | 14204 | 53.4058 | 2565 | 22956 | 99.6235 |
| | sensitive GPU | 5426 | 17978 | 98.9966 | 59611 | 15119 | 98.2124 | 3572 | 16221 | 53.4058 | 357 | 25956 | 99.6235 |
| HPG-aligner v2.0.1 (16th) | fast | 2198 | 28458 | 99.1352 | 1058 | | 94.3202 | 886 | 28491 | 39.5140 | 367 | 29681 | 99.0647 |
| | default | 2278 | 28459 | 99.1381 | 1230 | 28281 | 94.5441 | 999 | 28497 | 39.5975 | 325 | 29687 | 99.0843 |
| | sensitive | 2258 | 28463 | 99.1381 | 1471 | 28282 | 94.2318 | 1020 | 28483 | 39.6060 | 354 | 29587 | 99.0710 |
| nvBowtie v1.1.0 (K80) | very-fast | 1035 | 8257 | 96.9891 | 717 | 8555 | 53.6904 | Max read size to 500nt | | | Max read size to 500nt | | |
| | default | 1361 | 8257 | 97.2238 | 1157 | 8664 | 53.9722 | | | | | | |
| | very-sensitive | 2166 | 8257 | 97.2625 | 2279 | 8555 | 54.0155 | | | | | | |
| SNAP v1.0 beta 18 (16th) | h50 | 812 | 38277 | 96.3302 | 952 | 36618 | 33.4575 | 1103 | 36600 | 8.8983 | 547 | 36554 | 81.5051 |
| | default (h300) | 3610 | 38477 | 97.2014 | 6379 | 36313 | 33.4750 | 4762 | 36255 | 8.9459 | 975 | 36249 | 81.5582 |
| | h1000 | 6485 | 38181 | 97.4879 | 10748 | 36756 | 33.4817 | 4098 | 36301 | 8.9634 | 1437 | 35903 | 81.5822 |
| | h2000 | 10268 | 38467 | 97.6543 | 13464 | 37148 | 33.4836 | 6100 | 36520 | 8.9733 | 2509 | 35683 | 81.5950 |
| SOAP3-dp GPU 2.3.r180 (16th; K80) | default | 503 | 12887 | 97.0102 | 1402 | 21043 | 85.6903 | Max read size to 1024nt | | | Max read size to 1024nt | | |
| NovoAlign V3.04.06 | default | 65072 | 8596 | 91.1972 | 697864 | 8472 | 93.5828 | 1464887 | 8498 | 27.2728 | 51932 | 8670 | 98.1982 |

| Mapper | Configuration | HiSeqPE.Real [Bases: 14800 Mbases] [R: 150M; S: (0, 98, 100)] | | | MiSeqPE.Real [Bases: 14703 Mbases] [R: 50M; S: (0, 294, 300)] | | |
|---|---|---|---|---|---|---|---|
| | | CGA (CNAG) Minutes | Mbytes | QUALITY % Sen | CGA (CNAG) Minutes | Mbytes | QUALITY % Sen |
| Bowtie2 v2.3.4 (16th) | very-fast | 2972 | 3715 | 96.1792 | 3919 | 3798 | 50.5516 |
| | default | 4203 | 3758 | 96.8460 | 5854 | 3860 | 51.0307 |
| | very-sensitive | 7263 | 3881 | 97.0835 | 7937 | 4053 | 51.1266 |
| BWA v0.7.17 (16th) | default (300) | 3390 | 8596 | 98.3239 | 4006 | 7581 | 98.4101 |
| | c=500 | 3394 | 8593 | 98.3239 | 4007 | 7581 | 98.4101 |
| | c=1000 | 3552 | 8783 | 98.3240 | 4088 | 7589 | 98.4100 |
| | c=16000 | 3962 | 9155 | 98.3241 | 4170 | 7598 | 98.4101 |
| | c=20000 k=16 | 5542 | 9745 | 98.3433 | 7318 | 10764 | 98.5181 |
| CUSHAW2 v2.1.8-r16 (16th, K80) | default | 3698 | 4900 | 80.2309 | More than 80h of execution | | |
| | sensitive | 3172 | 4900 | 82.1077 | | | |
| GEM3-GPU git (06.18) (16th; K80) | default CPU | 834 | 11634 | 98.4038 | 2404 | 12335 | 97.6324 |
| | default GPU | 297 | 15603 | 98.4038 | 1884 | 41619 | 97.6324 |
| | sensitive CPU | 6171 | 12895 | 98.6289 | 62844 | 15901 | 97.6313 |
| | sensitive GPU | 5603 | 17653 | 98.6289 | 61146 | 113189 | 97.6313 |
| HPG-aligner v2.0.1 (16th) | fast | More than 80h of execution | | | 2287 | 11559 | 91.9627 |
| | default | | | | 2287 | 11562 | 92.8672 |
| | sensitive | | | | 3752 | 11562 | 93.2819 |
| nvBowtie v1.1.0 (K80) | very-fast | 2858 | 10860 | 96.9891 | 1622 | 11562 | 54.0648 |
| | default | 3917 | 10860 | 97.2236 | 2287 | 11559 | 54.1500 |
| | very-sensitive | 5721 | 10860 | 97.2626 | 3752 | 11562 | 54.1665 |
| SNAP v1.0 beta 18 (16th) | h50 | 1127 | 38248 | 95.7284 | 923 | 36502 | 31.5260 |
| | default (h300) | 1617 | 38293 | 96.1234 | 3470 | 36317 | 31.5577 |
| | h1000 | 5553 | 38495 | 96.6152 | 14770 | 36428 | 31.5946 |
| | h2000 | 5798 | 38631 | 96.6207 | 14870 | 36528 | 31.5972 |
| SOAP3-dp GPU 2.3.r180 (16th; K80) | default | 857 | 13598 | 98.3492 | 2409 | 22153 | 33.3928 |
| NovoAlign V3.04.06 | default | 190661 | 9181 | 92.0261 | 659880 | 8757 | 86.8555 |

**Figure 10.7:** Performance and Accuracy comparison with the most relevant mappers on the state-of-the-art using real data

# 11
# Conclusions

*"Every real story is a Neverending story... but that's another story and shall be told at another time"*

The Neverending Story - **Michael Ende**

This chapter presents the experiences gained and conclusions derived from this thesis. We also describe the viable open lines that can be considered in the future in order to provide further strategies.

## 11.1 Conclusions

This thesis' cornerstone is the use of heterogeneous compute platforms and their applicability to genomics workloads. During the last years, the computational systems used for bioinformatic analysis have increasingly become more heterogeneous, offering a direct chance of applying the findings exposed in this thesis and evaluate their impacts for the scientific community.

Currently, we are in a unique moment where the intersection between the bioinformatics and artificial intelligence fields is fostering new advances. Some examples include the use of neuronal networks to improve the accuracy of the results generated in the base calling and variant calling stages of the sequencing pipelines. Given that heterogeneous systems (accelerators) are strongly established in artificial intelligence and deep learning, there is a unique interest to evaluate and develop current bioinformatics algorithms associated with sequencer technologies.

New computational systems are also increasingly more heterogeneous and tightly integrated, therefore every day they will be more present in current production systems. This gives the current work more future relevance.

This project has improved the state-of-the-art on performance and accuracy for the genomics sequencing downstream process. The thesis has identified, characterised and analysed the performance of most relevant and computationally expensive algorithms used in real genomics pipelines for sequence mapping and sequence alignment. We have made several proposals and delivered a solid software pipeline accelerated by GPU computer architectures.

The presented work has provided a full-pledge GPU mapper that is running in production at the National Center of Genomic Analysis (CNAG). Therefore, the proposals have been validated using real data against the state-of-the art CPU and GPU with a broadly accepted methodology. When comparing with the current de facto state-of-the-art BWA-MEM the results presented in the current work achieve an improved performance of 20x to 40x faster than the throughput on the reference data.

Not only a full integration of the proposals has been performed, but a general CUDA library (GEM-cutter) has also been created. GEM-cutter accelerates the most widely adopted bioinformatic primitives for mapping and alignment in a fully transparent way for the library users without the need of in depth CUDA GPU API knowledge.

We conclude that GPUs and heterogeneous systems are a feasible alternative to the more traditional CPU-based applications, providing cost-reduction benefits and an improvement on the turn-around of the data analysis. Also, we consider that in the next future, the algorithmic

proposals of this work will fit well in upcoming GPU architectures.

We believe that when all these systems are broadly available, patients are expected to benefit from the presence of cost-effective alternatives for diagnosis.

Finally, the preparation of scientific publications, assistance to congresses (both national and international) and the realisation of Internships and Institution visits have provided invaluable knowledge and experience, which have inevitably impacted the work exposed in the current thesis. Furthermore, direct individual contributions have also been made in each of the visited institutions.

## 11.2 Future lines

Several future research lines can arise when considering the contributions of the current thesis. Over the last years, there has been a trend in the scientific community towards the concept of pan-genomics [52], in which the reads are simultaneously aligned against several genome references (e.g., Haplotype-aware graphs). In future works, the knowledge acquired optimising the FM-index for GPUs would be interesting in order to explore if this could be applied to the most recent graph indexes based on FM-index.

After the work of offloading methods from GEM3-CPU to GPU performed in this thesis, we identified that the three most consuming phases pending to be accelerated are (1) Smith and Waterman Gotoh Banded algorithm [53], (2) Sorting of the candidates to perform seed chaining and (3) the current GEM3 custom CPU memory allocator.

Interestingly, part of the work of the Internship at NVIDIA in 2016 was based on the Smith-Waterman-Gotoh algorithm [53], so it would be interesting to revisit the ideas having in mind the last Nvidia GPU architectures and fully-integrate the contributions from that project in GEM3-GPU, and by extension into GEM-Cutter.

Historically, sorting algorithms on GPUs performs very well, and therefore it would be interesting to explore how to port and integrate the sorting presented on the seed chain algorithm.

GEM3 includes a custom memory allocator that was developed with the non-batched workflow in mind, which has an striking performance difference when running the batched (GPU) and non-batched (CPU) GEM3 pipeline. It would be desirable to review the current implementation to explore more suitable approaches for the batched pipeline.

The GPU accelerated proposals of this thesis utilise a fine grain parallelization approach, a well as cooperative thread warp techniques. These early ideas are closely related to the ones of the recently proposed NVIDIA model called Cooperative Groups. The work from this thesis was made before the Cooperative Groups model was published, and therefore potential future works to port these ideas to the NVIDIA model could take profit of their features instead of the ad-hoc implementation currently present in GEM-cutter.

Also, at the moment of developing this thesis, CUDA Managed Memory allocators had not been yet released by Nvidia. Thus, all the GEM3-GPU memory is managed by a low-level CUDA API malloc. For performance and portability reasons, it would be interesting to port and analyse all the thesis implementations using the more recent CUDA allocators.

Regarding the last advances in GPU devices and architectures, there are a few topics that could be interesting to consider extending from the current thesis. We believe that

the exploration of the new characteristics of recent GPU architectures (such as Ampere), could be highly favourable towards the GEM3-GPU performance. This would include the asynchronous copy using the internal SM's DMAs, exploring multi-instance GPU (MIG) partitioning or a better usage of larger main memories (currently up to 80 GBytes), which would allow to redesign more efficient indexes. At instruction level, horizontal reductions could help to obtain more efficient FM-index primitives; and match instructions to improve the k-mer count primitive improving their binning process.

Broader open-lines could include the evaluation of new embedded architectures, such as the Nvidia Arm Jetson, as it could be more suitable for clinics environments. In addition, other potential lines would be the acceleration of basic compression libraries (BCL), as well as the exploration of the new promising noisy long reads technologies on heterogeneous systems.

## 11.3 List of publications

The work from this thesis have been peer-viewed and published on the following high impact international congresses and scientific journals:

1. **A.Chacón, P.Erencia, A.Espinosa, JC.Moure, P.Hernández. ”Suffix-Array and FM-index analysis in Multi-ManyCore” in XXIII Jornadas de Paralellismo, pp 255-260, 2012.**

   A comparative analysis of performance between Suffix-array and FM-Index was performed for CPU Quad-Core and GPU Nvidia Fermi. The results showed speedups between 2.3x and 6x after implementing specific short-alfabet optimizations. The experimentation also revealed that both searching algorithms are limited by memory latency.

2. **A.Chacón, JC.Moure, A.Espinosa, P.Hernández. ”n-step FM-index for faster pattern matching.” in Procedia Computer Science, Vol 18, pp 70–79, 2013. DOI: 10.1016/j.procs.2013.05.170**

   Presented a variation of the FM-index called “n-step FM-index” which can be applied for exact sequencing. This is a two dimensional FM-index structure allowing backward navigation giving steps of n symbols at a time. This allows a reduction of the computational work and an increase in the temporal locality of the data access pattern (however this increases the amount of data required for the index). Speedups ranging between 1.4x and 2.4x were reported if no DRAM limitation was present. The proposal provided an alternative for pseudo-random memory access algorithms to be redesigned to scale in current and future computer systems.

3. **A.Chacón, S.Marco-Sola, A.Espinosa, P.Ribeca, and JC.Moure. ”Thread- cooperative, bit-parallel computation of Levenshtein distance on GPU” in Proceedings of the 28th ACM international conference on Supercomputing, pp Pages 103-112, 2014. DOI: 10.1145/2597652.2597677**

   A CUDA version of the Bit Parallel Myers algorithm was implemented and optimized, reaching a high efficiency by using a parallel cooperative strategy between the threads. This resulted in an improvement in the GPU performance by 20x when compared to the CPU performance.

4. **A.Chacón, S.Marco-Sola, A.Espinosa, P.Ribeca, and JC.Moure. ”FM-index on GPU: a cooperative scheme to reduce memory footprint.” in International Symposium on Parallel & Distributed Processing with Applications, pp 1-9, 2014. DOI: 10.1109/ISPA.2014.10**

   This work was based in a combination of a compact design of FM-index and a thread cooperative approach to adjust the ideal balance between compute and memory. The proposed structure was less-dependent memory bandwidth and allowed a more optimal use of the computational resources of the GPU across several GPU architectures.

5. **A.Chacón, S.Marco-Sola, A.Espinosa, P.Ribeca, and JC.Moure. ”Boosting the FM-index on the GPU: effective techniques to mitigate random memory access”, in Journal Transactions on Computational Biology and Bioinformatics, Volume 12, Issue 5, pp 1048-1059, 2015. DOI: 10.1109/TCBB.2014.2377716**

   This study shows that several strategies can be applied to deal with the GPU memory bottleneck: more compact indexes can be implemented by increasing the number of threads working cooperatively on larger memory blocks, and a k-step FM-index can be used to further reduce the number of memory accesses. This resulted in an implementation that was able to process about 2 Gbases of queries per second on the test platform, being about 8× faster than a comparable multi-core CPU version, and about 3× to 5× faster than the FM-index implementation on the GPU provided by the Nvidia NVBIO bioinformatics library

6. **S.Laurie, M.Fernandez-Callejo, S.Marco-Sola, J.Trotta, J.Camps, A.Chacón, A.Espinosa, M.Gut, I.Gut, S.Heath, and S.Beltran. ”From wet-lab to variations: robustness and speed of bioinformatics pipelines for WGS and WES”, in Journal Human Mutations, 2016. DOI: 10.1002/humu.23114**

   This is a publication in a bioinformatic journal in which all the development for the correct genomic validation of the GEM-mapper tool is exposed. All the production processes in CNAG, as well as the specific technologies and optimizations used are detailed. The work shows a bioinformatic pipeline in a hybrid CPU and GPU environment. The results show reduction in the processing times of one order of magnitude, with qualities that are comparable between other state-of-the-art tools.

## 11.4 Acknowledgements

I would like to extend my most sincere gratitude to everyone that has contributed, in any way, shape or form, to the work presented in this thesis. Every single person that I have encountered in the process of these past 10 years has imprinted a mark in this work, in one way or another. The current thesis has involved a huge amount of work and sacrifice, which would have not been possible should I have not been surrounded by people who supported me. Among those, several people deserves a special mention for their help over all these years.

To my advisors, Juan Carlos Moure, Toni Espinosa and also Santiago Marcos for introducing me to the wonderful field of Bioinformatics and GPU Architectures so many years ago right before my final degree project. Also, to the team at the Computer Architecture and Operating Systems Department (CAOS) at Universitat Autonoma de Barcelona. Who would have told me then that the project would become the research topic of a decade!

To my internship mentors Nuno Subtil and Jacopo Pantaleoni at NVIDIA. Not only for the incredible opportunity that they brought to me five years ago when they bet for me offering me an Internship at NVIDIA, but also for the warm welcome, all the advice and the friendship that has endured even after all these years. You have continued to help and mentor me way beyond the time of my internship.

To Jonathan Cohen, Michael Garland, Peter Ferguson, Baris Ozgul, Juango Noguera, Julien Demouth and Tom Bradley, for bringing me the opportunity of continuing my career at your amazing teams, and for allowing me the time to finalise the current thesis.

To my family, because none of this would have been possible if you have not been there for me at all times. Thank you for joining me in this crazy journey until the very end.

To my wife, for being the very best partner-in-crime that one can wish for. For the sleepless nights, the countless hours of revision and for always believing in me. We have been together through all this adventure, across four different countries and countless misfortunes, but also through a thousand small special moments. Thank you for transforming my world into a wonderful one to live in.

To all my friends, who are an extension of my own family for me and who make life worth living. Thank you for being by my side, no matter how far away or how long since my last visit.

From California, to Alex Ramirez, Isaac Gelado, Nuno Subtil and Monica Pimentel, Simon Migaz, Piotr Wojciechowski, Juango Noguera, Oscar Pelaez.

From Ireland, to Pedro Duarte and Diana Santos, Raúl Mateos and Slavka Madarova, Pancho Barat, Jose Marques, Francesco Maio, Dylan Malone, David Clarke, Zack Dickman,

Peter McColgan, Stephan Munz, Jan Langer, Javi Cabezas, Sneha Date and to the living memory of Johannes Kappauf.

From Cambridge, Roger Ferrer, Rekai Gonzalez, Miyo Takahashi, Elias Vougioukas, Rene De Jong, Jonas Svedas, Stephen Kyle, Vassilis Laganakos, Gem Dot, Kike Sedano and Dácil Carpio, Filippo Spiga, Javi Setoain, Paola Montes and Mimi Gonzalez.

From Barcelona, to Pedro Monzo and Carolina Fernandez, Oriol Mula and Ágata Franco, Jordi Cantó, Xavi Andiñach, Mingu Manubens, Ricard Molina, Jose Moreno, Oriol Puig, Josefina and Diego, Mario, Miguel and Kim. To my lifetime friends, Jaime Gallardo and Maria Carrillo, Juan Ruiz, Albert Rubio, Chefo, Ruchi, Charli and Lucas Ventura.

For all the shared memories across master degree and thesis: Javier Navarro, Mireia Sanchez, Tito Cruz, Pilar Gomez, Tomas Artes, Andres Cencerrado, Cesar Acevedo, Cesar Allande, Arindam Choudhury, Marcela Castro, Claudia Rosas, Pedro Erencia, Javier Panadero, Hugo Meyer, Noe Meyer, Daniel Hernandez, Hai Nguyen, Aprigio Bezerra, Carlos Rangel, Laura Espinola, Jorge Villamayor, Liu Zhengchun, Claudio Marquez, Abel Castellanos, Roberto Solar, Cecilia Caramillo and Joe Carrión, and to the living memory of Manuel Brugnoli.

To Oriol Mula, for being the most amazing teacher of the Study Room, as well as the best Best Man. For the help, the advice and the motivation across the years, way before this thesis. Also, for the effort in reviewing, correcting and annotating the thesis in the most altruistic way. There will be an extra shelf in your new house soon.

To Momo, for being the best cat ever.

# Bibliography


[1] N. National human genome research institute. (2020) Sequencing human genome cost. [Online]. Available: https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost

[2] A. Prabhakaran, B. Shifaw, M. Naik *et al.* Infrastructure for deploying gatk best practices pipeline. [Online]. Available: https://www.intel.la/content/www/xl/es/healthcare-it/solutions/documents/deploying-gatk-best-practices-paper.html

[3] J. Watson and F. Crick, "Molecular structure of nucleic acids: A structure for deoxyribose nucleic acid," *Nature*, vol. 171, no. 4356, p. 737–738, 1953. [Online]. Available: https://europepmc.org/articles/PMC3322381

[4] M. A. Hamburg and F. S. Collins, "The path to personalized medicine," *New England Journal of Medicine*, vol. 363, no. 4, pp. 301–304, 2010. [Online]. Available: https://app.dimensions.ai/details/publication/pub.1011021419

[5] L. Chin, J. N. Andersen, and P. A. Futreal, "Cancer genomics: from discovery science to personalized medicine," *Nature Medicine*, vol. 17, no. 3, pp. 297–303, 2011.

[6] E. Schadt, S. Turner, and A. Kasarskis, "A window into third-generation sequencing," *Human Molecular Genetics*, vol. 19, no. R2, pp. R227–R240, 09 2010. [Online]. Available: https://doi.org/10.1093/hmg/ddq416

[7] G. Myers, "Efficient local alignment discovery amongst noisy long reads," in *Algorithms in Bioinformatics*, D. Brown and B. Morgenstern, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 52–67.

[8] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows–Wheeler

transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, 01 2010. [Online]. Available: https://doi.org/10.1093/bioinformatics/btp698

[9] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, "A domain-specific architecture for deep neural networks," *Commun. ACM*, vol. 61, no. 9, p. 50–59, aug 2018. [Online]. Available: https://doi.org/10.1145/3154484

[10] M. Holtgrewe, "Mason - A read simulator for second generation sequencing data," *Technical Report FU Berlin*, 2010.

[11] J. Pantaleoni and N. Subtil. (2014, Jun.) NVBIO: a library of reusable components designed by NVIDIA corporation to accelerate bioinformatics applications using CUDA. [Online]. Available: http://nvlabs.github.io/nvbio/

[12] Laurie S, Fernandez Callejo M, Marco-Sola S, Trotta JR, Camps J, Chacón A, et al, "From wet-lab to variations: Concordance and speed of bioinformatics pipelines for whole genome and whole exome sequencing," *Human Mutations*, vol. 12, no. 37, pp. 1263–1271, September 2016.

[13] "Novoalign," http://www.novocraft.com/, accessed: 2010-09-30.

[14] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature methods*, vol. 9, no. 4, p. 357—359, March 2012. [Online]. Available: https://europepmc.org/articles/PMC3322381

[15] Y. Liu and B. Schmidt, "CUSHAW2-GPU: empowering faster gapped short-read alignment using GPU computing," *Design and Test of Computers*, 2013.

[16] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Computing Surveys (CSUR)*, vol. 39, no. 1, p. 2, 2007.

[17] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 2000, pp. 390–398.

[18] M. Burrows and D. Wheeler, "A block-sorting lossless data compression algorithm," in *Technical Report 124, palo Alto, CA*. DEC, 1994.

[19] T. Smith and M. Waterman, "Identification of common molecular subsequences," *J Mol Biol*, vol. 147, pp. 195–197, 1981.

[20] J. Tárraga, M. Pérez, J. M. Orduña, J. Duato, I. Medina, and J. Dopazo, "A parallel and sensitive software tool for methylation analysis on multicore platforms," *Bioinformatics*, vol. 31, no. 19, pp. 3130–3138, 06 2015. [Online]. Available: https://doi.org/10.1093/bioinformatics/btv357

[21] "nvbowtie," https://nvlabs.github.io/nvbio/nvbowtie_page.html, accessed: 2010-09-30.

[22] R. Luo, T. Wong, J. Zhu, C.-M. Liu, X. Zhu, E. Wu, L.-K. Lee, H. Lin, W. Zhu, D. Cheung, H. Ting, S. Yiu, S. Peng, C. Yu, Y. Li, R. Li, and T. Lam, "Soap3-dp: Fast, accurate and sensitive gpu-based short read aligner," *PLoS ONE*, vol. 8, 2013.

[23] Y. Liu, B. Schmidt, and D. L. Maskell, "CUSHAW: a CUDA compatible short read aligner to large genomes based on the Burrows–Wheeler transform," *Bioinformatics*, vol. 28, no. 14, pp. 1830–1837, 05 2012. [Online]. Available: https://doi.org/10.1093/bioinformatics/bts276

[24] M. Zaharia, W. Bolosky, K. Curtis, A. Fox, D. Patterson, S. Shenker, I. Stoica, R. Karp, and T. Sittler, "Faster and more accurate sequence alignment with snap," *ArXiv*, vol. abs/1111.5572, 2011.

[25] S. Marco-Sola, M. Sammeth, R. Guigo, and P. Ribeca, "The gem mapper: fast, accurate and versatile alignment by filtration," *Nature Methods*, vol. 9, no. 12, pp. 1185–1188, 2012.

[26] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: a unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[27] M. McCool, J. Reinders, and A. Robison, *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[28] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, and P. Dubey, "Fast: fast architecture sensitive tree search on modern cpus and gpus," *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010.

[29] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 235–246.

[30] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, “Dissecting the NVIDIA volta GPU architecture via microbenchmarking,” *CoRR*, vol. abs/1804.06826, 2018. [Online]. Available: http://arxiv.org/abs/1804.06826

[31] Y. Turakhia, G. Bejerano, and W. J. Dally, “Darwin: A genomics co-processor provides up to 15,000x acceleration on long read assembly,” *Association for Computing Machinery*, p. 199–213, 2018. [Online]. Available: https://doi.org/10.1145/3173162.3173193

[32] U. Manber and G. Myers, “Suffix arrays: a new method for on-line string searches,” *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[33] P. Weiner, “Linear pattern matching algorithms,” in *Switching and Automata Theory, 1973. SWAT’08. IEEE Conference Record of 14th Annual Symposium on*. IEEE, 1973, pp. 1–11.

[34] M. Abouelhoda, S. Kurtz, and E. Ohlebusch, “Replacing suffix trees with enhanced suffix arrays,” *Journal of Discrete Algorithms*, vol. 2, no. 1, 2004.

[35] P. Ferragina and G. Manzini, “Opportunistic data structures with applications,” in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 2000, pp. 390–398.

[36] Y. Liu and B. Schmidt, “Evaluation of GPU-based seed generation for computational genomics using burrows-wheeler transform,” in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, May 2012, pp. 684–690.

[37] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg, “Versatile and open software for comparing large genomes,” *Genome Biology*, vol. 5, no. 2, 2004.

[38] J. Treibig, G. Hager, and G. Wellein, “Likwid: A lightweight performance-oriented tool suite for x86 multicore environments,” in *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[39] U. Drepper, “What every programmer should know about memory,” *Red Hat, Inc (2007)*, 2007.

[40] Y. Ono, K. Asai, and M. Hamada, “PBSIM: PacBio reads simulator—toward accurate genome assembly,” *Bioinformatics*, vol. 29, no. 1, pp. 119–121, 2013.

[41] H. Li and R. Durbin, “Fast and accurate short read alignment with Burrows-Wheeler transform,” *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[42] H. Xin, S. Nahar, R. Zhu, J. Emmons, G. Pekhimenko, C. Kingsford, C. Alkan, and O. Mutlu, “Optimal seed solver: optimizing seed selection in read mapping,” *Bioinformatics*, vol. 32, no. 11, p. 1632–1642, 2016.

[43] G. Navarro, “A guided tour to approximate string matching,” *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, Mar. 2001. [Online]. Available: http://doi.acm.org/10.1145/375360.375365

[44] P. H. Sellers, “The theory and computation of evolutionary distances: Pattern recognition,” *Journal of Algorithms*, vol. 1, no. 4, pp. 359 – 373, 1980. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0196677480900164

[45] G. Myers, “A fast Bit-Vector algorithm for approximate string matching based on dynamic programming,” *J. ACM*, vol. 46, no. 3, pp. 395–415, May 1999. [Online]. Available: http://doi.acm.org/10.1145/316542.316550

[46] E. Ukkonen, “Finding approximate patterns in strings,” *Journal of algorithms*, vol. 6, no. 1, pp. 132–137, 1985.

[47] H. Hyyrö and G. Navarro, “Faster bit-parallel approximate string matching,” in *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching*, ser. CPM ’02. London, UK, UK: Springer-Verlag, 2002, pp. 203–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=647821.736383

[48] L. Langner, K. Reinert, and D. Weese, “Myers Bit-Vector Algorithm on GPU for SeqAn,” Master’s thesis, Freie Universität Berlin, 2011.

[49] D. Tristram and K. Bradshaw, “Evaluating the acceleration of typical scientific problems on the GPU,” in *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, ser. SAICSIT ’13. New York, NY, USA: ACM, 2013, pp. 17–26. [Online]. Available: http://doi.acm.org/10.1145/2513456.2513473

[50] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, “Effective compiler support for predicated execution using the hyperblock,” in *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2. IEEE Computer Society Press, 1992, pp. 45–54.

[51] A. McKenna, M. Hanna, E. Banks *et al.*, “The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data,” *Genome Research*, vol. 20, no. 1297-303, 2010.

[52] J. Eizenga, A. Novak, J. Sibbesen, and others., “Pangenome graphs,” *Annual Review of Genomics and Human Genetics*, vol. 21, no. 1, pp. 139–162, 2020, pMID: 32453966. [Online]. Available: https://doi.org/10.1146/annurev-genom-120219-080406

[53] O. Gotoh, “An improved algorithm for matching biological sequences,” *J Mol Biol*, vol. 162, no. 3, pp. 705–708, 1982.