Novel Techniques to Improve the Performance and the Energy of Vector Architectures

Adrián Barredo Ferreira

Barcelona, 2021

Advisors: **Adrià Armejach Sanosa**
Miquel Moretó Planas

Collaborators: **Jonathan C. Beard**
Juan M. Cebrián González
Marc Casas Guix
Rhadika Jagtap

A thesis submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy

in the Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Abstract

The rate of annual data generation grows exponentially. At the same time, there is a high demand to analyze that information quickly. In the past, every processor generation came with a substantial frequency increase, leading to higher application throughput. Nowadays, due to the cease of Dennard scaling, further performance must come from exploiting parallelism. Vector architectures offer an efficient manner, in terms of performance and energy, of exploting parallelism at data-level by means of instructions that operate over multiple elements at the same time. This is popularly known as Single Instruction Multiple Data (SIMD). Traditionally, vector processors were employed to accelerate applications in research, and they were not industry-oriented. However, vector processors are becoming widely used for data processing in multimedia applications, and entering in new application domains such as machine learning and genomics. In this thesis, we study the circumstances that cause inefficiencies in vector processors, and new hardware/software techniques are proposed to improve the performance and energy consumption of these processors.

We first analyze the behavior of predicated vector instructions in a real machine. We observe that their execution time is dependent on the vector register length and not on the source mask employed. Therefore, a hardware/software mechanism is proposed to alleviate this situation, that will have a higher impact in future processors with wider vector register lengths.

We then study the impact of memory accesses to performance. We identify that an irregular memory access pattern prevents an efficient vectorization, which is automatically discarded by the compiler. For this reason, we propose a near-memory accelerator capable of rearranging data structures and transforming irregular memory accesses to dense ones. This operation may be performed by the devices as the host processor is computing other code regions.

Finally, we observe that many applications with irregular memory access patterns just perform a simple operation on the data before it is evicted back to main memory. In these situations, there is a lack of data access locality, leading to an inefficient use of the memory hierarchy. For this reason, we propose to utilize the accelerators previously described to compute directly near memory.

Resumen

La tasa de generación de información aumenta cada año. Al mismo tiempo, existe una alta demanda para analizar dicha información en el menor tiempo posible. En el pasado, se recurría a aumentar la frecuencia de los procesadores para conseguir una mayor velocidad de procesamiento de los datos. En la actualidad, debido al fin de la ley de Dennard, la frecuencia deja de ser una opción y se apunta al paralelismo como la mejor alternativa. Las arquitecturas vectoriales ofrecen una manera eficiente, en términos de rendimiento y energía, de explotar el paralelismo a nivel de datos a través de instrucciones que operan sobre múltiples elementos al mismo tiempo, conocidas popularmente como SIMD. Tradicionalmente, los procesadores vectoriales se utilizaban para acelerar las aplicaciones en la investigación y no estaban orientados a la industria. Sin embargo, dichos procesadores están siendo cada vez más utilizados para el procesamiento de datos en aplicaciones multimedia. En esta tesis doctoral, se investigan las causas que pueden suponer la ineficiencia de las arquitecturas vectoriales, y se proponen mejoras a nivel de hardware y software con el fin de mejorar el rendimiento y el consumo de estos procesadores.

En primer lugar, se estudia el funcionamiento de las instrucciones vectoriales predicadas en una máquina real. Como resultado, se observa que el tiempo de ejecución y el consumo de dichas instrucciones es independiente de la máscara empleada, mientras que sí es dependiente de la longitud de los registros vectoriales que contienen los datos. Por tanto, se propone un mecanismo hardware/software para aliviar esta situación, que se agravará en el futuro con la aparición de procesadores con la longitud de los registros vectoriales más alta.

En segundo lugar, se analiza el impacto de los accesos a memoria por parte del procesador vectorial. En este caso, se comprueba que un acceso irregular a memoria impide una vectorización eficiente de las aplicaciones, que es descartada automáticamente por el compilador. Por tanto, en esta tesis se propone un acelerador cerca de memoria capaz de reordenar los datos y proporcionar accesos secuenciales a memoria mientras el procesador está computando otras regiones de la aplicación.

En tercer lugar, se propone utilizar los aceleradores previamente descritos como elementos de cómputo, dado que muchas aplicaciones acceden a memoria de manera irregular para realizar un cómputo muy sencillo en el procesador. Este movimiento de datos puede ser evitado si la operación es realizada cerca de memoria. El rendimiento de estos aceleradores es evaluado en aplicaciones de computación de altas prestaciones y en grafos, un campo de la ciencia muy afectado por esta situación.

Acknowledgments

Este documento representa el final de un duro camino que comenzaría en 2015 con una mudanza Santoña-Barcelona para la realización de un máster. Durante este tiempo, mucha gente ha estado a mi lado y me han ayudado tanto en el ámbito profesional como en el personal. Familia, amigos y compañeros de trabajo han sido claves y sin los cuales ahora mismo no estaría escribiendo estas líneas.

En primer lugar, quiero agradecer a mis padres Ángel y Gemma y a mi hermano Jorge su apoyo incondicional durante este tiempo. Sus consejos y sus visitas me han dado fuerzas en numerosas ocasiones para seguir adelante. Gracias también a mis abuelos Ángel y María Ángeles por ser un ejemplo de amor y sacrificio. Muy especialmente, me gustaría agradecer a mi pareja Alazne su amor y su infinito apoyo durante todo este tiempo. Gracias por creer en mí.

En segundo lugar, quisiera hacer mención a mis directores de tesis, Miquel Moretó y Adrià Armejach, que han tenido un paper muy importante en su elaboración. Siempre han estado ahí para ayudarme y motivarme. He aprendido mucho de ellos como personas e investigadores. Sus consejos han sido fundamentales en muchos momentos. También me gustaría agradecer a Ramón Beivide de la Universidad de Cantabria, por ofrecerme la oportunidad de estudiar y formarme en Barcelona. Sin tí este projecto no habría sido posible. Gracias también por haber sido un apoyo durante la realización del doctorado.

Gracias también a todos mis compañeros, que ya son amigos, del equipo RoMoL del Barcelona Supercomputing Center. Mención especial para Isaac, Vladimir, Constan, Emilio, Calvin y Xubin. Nunca se me olvidarán nuestros momentos tanto dentro como fuera de la oficina. Gracias Isaac por todos tus consejos y ayuda, pero sobre todo por tu paciencia con este teleco convertido a informático.

I would also like to thank the people at Arm Research in Cambridge (UK) for two excellent internships in the company. Special thanks to Radhika Jagtap for opening Arm's doors for me as well as for being a wonderful advisor during the first internship. Thanks to Jonathan Beard for being an excellent tutor during the second one. I'm also thankful to Nikos Nikoleris, Javier

Setoain, Giacomo Travaglini, Ilias Vougioukas and Stephan Diestelhorst for their friendship and technical expertise during that time.

I'm grateful to the pre-defense committee members, Jaume Abella, Dimitrios Chasapis and Milan Radulović, and external reviewers, Gilles Sassatelli and María Jesús Garzarán, for providing valuable feedback, which helped me to improve this thesis.

Finalmente, los agradecimientos oficiales: This work has been partially supported by the RoMoL ERC Advanced Grant (GA 321253), the European HiPEAC Network of Excellence, the Spanish Government (contract TIN2015-65316-P) and the European Union's Horizon 2020 research and innovation program under the Mont-Blanc 2020 project (grant agreement 779877). Adrián Barredo has been supported by the Spanish Government under Formación del Personal Investigador (FPI) fellowship number BES-2017-080635.

A todos, A tots, To all, Guztiei:
Gracias, Gràcies, Thank you, Eskerrik asko

Table of contents

Abstract i

Resumen iii

Acknowledgments v

Contents x

1 Introduction 1

1.1 Thesis Objectives and Contributions 4

1.1.1 Improving Predication Efficiency Through Compaction/Restoration of SIMD Instructions 4

1.1.2 PLANAR: A Programmable Accelerator for Near-Memory Data Rearrangement 5

1.1.3 REMOTE: A Programmable Near-Memory Compute Engine 5

1.2 Thesis Outline 6

2 Background 7

2.1 Vector Architectures 7

2.1.1 An Example of the Micro-architecture of a Vector Processor 8

2.1.2 Vector Processors in Supercomputers 8

2.1.3 Vector Architectures in Microprocessors 9

2.1.4 SIMD Extensions 10

2.1.5 Advantages of Vector Architectures 10

2.1.6 Disadvantages of Vector Architectures 11

2.1.7 Challenges of Vectorization 11

2.2 The Memory Wall 15

2.2.1 The Memory Hierarchy 15

2.2.2 DRAM Organization 16
2.2.3 Memory Controller 18
2.2.4 Prefetching 19
2.2.5 Limitations of the Memory Hierarchy 20
2.2.6 Processing In/Near Memory 21
2.3 Parallel Programming for Shared-Memory Systems 24
2.3.1 Parallel Processors 24
2.3.2 Parallel Programming Models 25
2.4 Runtime-Aware Architectures 27

3 Experimental Methodology **29**
3.1 Simulation Infrastructure 29
3.1.1 Simulator 29
3.1.2 Environment 31
3.2 Benchmarks 34
3.2.1 Benchmarks for the Divergence Proposal 34
3.2.2 Benchmarks for the Near-Memory Data Rearrangement Proposal . . 36
3.2.3 Benchmarks for the Near-Memory Computing Proposal 36
3.3 Metrics 39

4 The Efficiency of Predicated SIMD Instructions **41**
4.1 Introduction 41
4.2 The Predication Problem in SIMD Extensions 42
4.3 The Compaction/Restoration Mechanism 44
4.3.1 Overview 44
4.3.2 CR Hardware Components 46
4.3.3 CR in an Out-of-Order Processor 48
4.3.4 Detecting Compactable Instructions 49
4.3.5 Populating Dense Instructions 50
4.3.6 Compaction Phase 50
4.3.7 Execution of Compacted Instructions 50
4.3.8 Restoration Phase 51
4.3.9 Dense Register Forwarding 52
4.3.10 CR Case Study 53
4.3.11 Optimizing SIMD Legacy Code 56

4.3.12 Discussion . . . 56
4.4 Design Space Exploration . . . 57
4.4.1 Compaction and Restoration Latencies . . . 57
4.4.2 Timeout Policies . . . 59
4.4.3 Costly SIMD Instruction Ratio . . . 59
4.4.4 Effectiveness with Different Loop Lengths . . . 60
4.5 Area and Power Consumption of CR Units . . . 61
4.6 Evaluation . . . 61
4.6.1 Predicated SIMD Applications . . . 62
4.6.2 Optimizing AVX-2 Legacy Code . . . 63
4.6.3 Comparison with Other Proposals . . . 64
4.7 Conclusions . . . 65

5 PLANAR: A Programmable Accelerator for Near-Memory Data Rearrangement 67
5.1 Introduction . . . 67
5.2 Motivation . . . 69
5.3 PLANAR Design . . . 72
5.3.1 Modifications to Application Code . . . 74
5.3.2 Allocation of Memory and PLANAR . . . 75
5.3.3 Offloading of Rearrange Function . . . 75
5.3.4 Execution of Rearrange Functions . . . 76
5.3.5 Synchronization Between PLANAR and Host . . . 76
5.3.6 Release of PLANAR Devices and Memory . . . 77
5.3.7 PLANAR Execution Example . . . 77
5.4 Design Space Exploration . . . 80
5.4.1 Pipeline Width . . . 80
5.4.2 Number of Functional Units . . . 81
5.4.3 Cache Size . . . 81
5.4.4 Number of PLANAR Devices . . . 81
5.4.5 Synchronization Granularity . . . 82
5.4.6 Selected Configuration . . . 83
5.5 Evaluation . . . 83
5.5.1 Impact to the Memory Hierarchy . . . 86
5.5.2 Area and Power Overhead . . . 87
5.5.3 Comparison to Other Proposals . . . 89

5.6 Conclusions 91

6 Near Memory Compute Engine 93

6.1 Introduction 93

6.2 Motivation 94

6.3 Proposal 96

6.3.1 The REMOTE Device 97

6.3.2 Changes to the Application 100

6.3.3 Changes to the Runtime System 100

6.4 Design Space Exploration 101

6.4.1 Pipeline width 101

6.4.2 Number of functional units 102

6.4.3 Latency of functional units 102

6.4.4 Frequency 103

6.4.5 Selected configuration 104

6.5 Evaluation 104

6.5.1 Profiling of the Applications 104

6.5.2 Results with REMOTE 105

6.5.3 Impact to the Memory Hierarchy 108

6.5.4 Host Core vs REMOTE Performance and Area Comparison 110

6.5.5 Comparison to Other Proposals 110

6.6 Conclusions 113

7 Conclusions and Future Work 115

7.1 Conclusions 115

7.2 Future Work 117

7.3 Publications 118

Bibliography 121

List of figures 143

List of tables 149

Chapter 1
Introduction

Moore's Law predicted that the number of transistors in a chip would double every two years. For decades, increasing the number of transistors was the common solution to improve the Instructions per Cycle (IPC) metric of the processors. In this scenario, deeper pipelines, branch predictors and cache memories greatly increased the instruction throughput of the processors.

For a long time, Moore's Law had two fundamental outcomes: (1) more features and functionality in an integrated circuit, and (2) higher operating frequencies with the same power density. Both of these outcomes contributed to the processor's overall performance increase. Frequency scaling was generally transparent to the programmer and algorithms were expected to execute faster with every new generation of processor.

At the beginning of the twenty first century, in what is commonly named as the cease of Dennard scaling [59, 32], thermal and power issues made it unfeasible to continue increasing the processor's operating frequency. This phenomenon is popularly known as the *Power Wall*, and it was reached around 2005 as shown in Figure 1.1. While Moore's Law still held true, the free performance scaling finally came to an end [175]. The industry had to shift its focus on using the extra available transistors to achieve better performance through explicit parallelism.

Parallelization techniques can be broadly categorised as instruction-level (ILP), thread-level (TLP) and data-level (DLP) [70]. When it is possible to exploit it, DLP is by far the most efficient form of parallelism [86]. DLP is defined as applying the same operation to multiple data elements. DLP can be exposed to the hardware by means of vector computations [20, 67], where a Single Instruction operates over Multiple Data streams (SIMD).

Vector machines appeared in the early 1970s and dominated supercomputer designs for two decades [155, 55, 26, 191]. These designs exploited DLP with long vectors of thousands of bits. Such vector designs are less popular nowadays, although the NEC's SX-Aurora processor has been recently announced featuring 16,384-bit vectors [139].

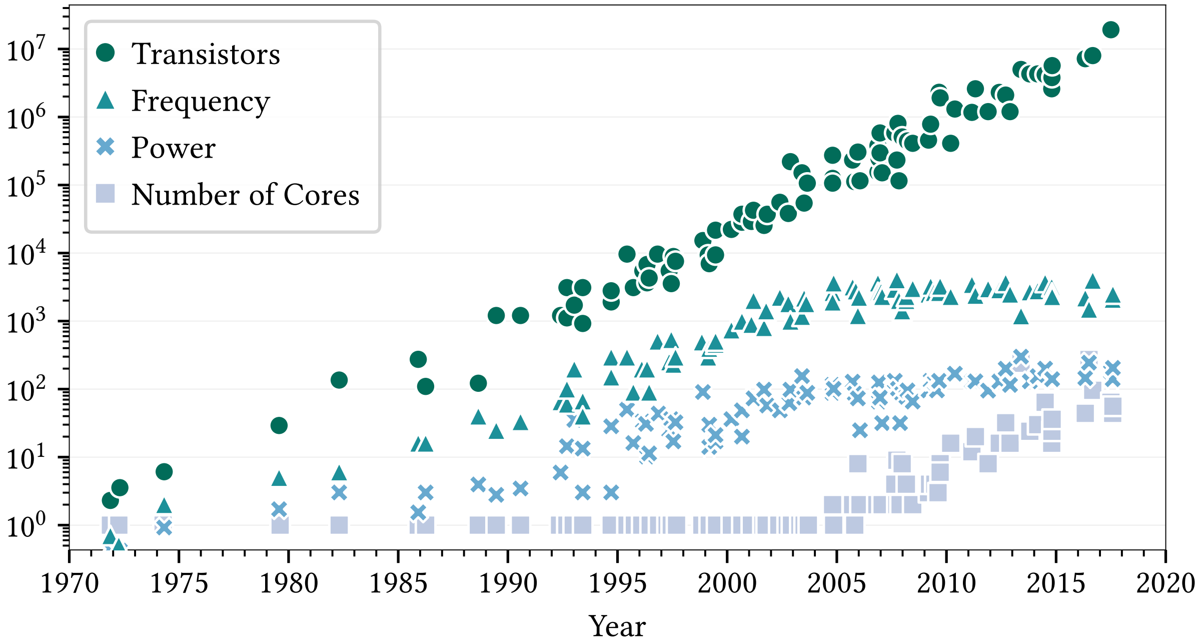


Figure 1.1: Historical trends of important metrics in computing systems.

Transistor count is presented in thousands, frequency in Hz and power in W. Original data up to 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten. Data from 2010 to 2017 collected by K. Rupp [154].

SIMD extensions to scalar Instruction Set Architectures (ISA) appeared in the late 1990s to improve the efficiency of multimedia applications, using short vectors of 128 bits [93, 74]. Such SIMD extensions have become ubiquitous in today's computer architectures [94, 17, 173, 6]. Processors with longer SIMD vector lengths have appeared in the last years, such as the 512-bit SIMD implementations from Intel [169, 94] and Fujitsu [201]. Nowadays, DLP exploitation is not limited to SIMD extensions. GPUs are alternative architecture designs that benefit from DLP with a massive amount of threads executing the same instruction in a lock-step model.

The effectiveness of a vector architecture depends on its ability to vectorize large quantities of code [86]. Although efficient vectorization by the compiler has captured industry's attention for several years, still some challenges remain unsolved, such as horizontal operations, data structure conversion or divergence control [91].

On the other hand, memories have been an integral part of computers ever since the appearance of the first concept for a programmable computing machine in 1837 [22]. Since they store both instructions and data, fast memory accesses are essential in order to achieve good application performance.

The differences in the technologies employed to manufacture processors and memories have widely increased the performance gap between these two resources. This effect, known as the *Memory Wall* [197], is clearly seen on Figure 1.2. In order to reduce this gap, the concept of cache memories came to light. Caches offer shorter access latency compared to main memory, but they are more costly in terms of area and power per byte of storage.

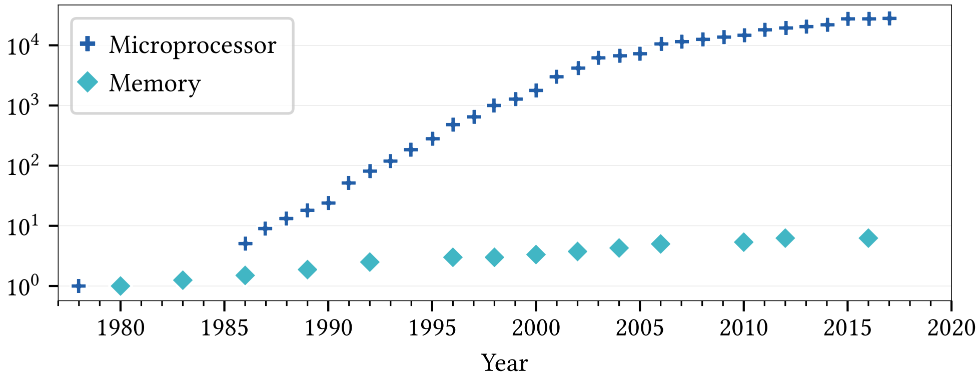


Figure 1.2: Evolution of relative processor and memory performance.

Data collected and plotted by Hennessy and Patterson [86]

Cache memories are a good alternative for applications that have locality of reference [171, 182] which means that cache memories with low latencies can meet data demands while prefetchers can act in parallel in the background to hide memory access latency. Deep cache hierarchies are the natural result of this trend, providing low-latency data access to high performance out-of-order processing elements.

However, recent trends show that we have effectively plateaued on the effectiveness of data prefetchers [132]. Trends also demonstrate ineffectiveness for irregular sparse patterns [207]. As a result of sparsity and irregular reuse distances, some studies have measured utilization of transmitted bandwidth as low as 20% for some applications [29]. Past studies have also proved that a significant fraction of the data brought into the last level cache goes unused before eviction [172]. This issue has a bigger impact on multi-core systems, where shared resources exist and every core competes for the memory bandwidth.

Performance is not the only factor affected by data motion. Approximately two thirds of the energy required to compute is consumed by data movement, specifically by the memory and interconnect [33]. In addition, irregular and sparse patterns preclude harnessing data-level parallelism (DLP) via vector instructions. Supplying SIMD units with high-bandwidth, low-latency data is critical to their efficiency [140].

In order to reduce the data movement through the memory hierarchy, the concept of Processing In/Near Memory (PINM) was born. The idea relies on placing computing resources close to where the data resides. Recent processing in memory proposals are summarized by Zhang *et al*. [204] and Balasubramonian *et al*. [23]. They can be split into two categories: (1) processing elements in memory (PIM), which require a special technology for the memory (i.e., Hybrid Memory Cube) [2, 138, 90, 62], and (2) near memory processing (PNM), where the compute elements are close to memory, such as in the memory controller [142, 124, 170, 206].

1.1 Thesis Objectives and Contributions

The main goal of this thesis is to improve the performance and the energy consumption of the chips, focusing on the vector architectures. To the best of our knowledge, vector architectures will be a key component in future processors, where huge amounts of data will require a fast and efficient processing. Nowadays, many challenges related to vector architectures remain unsolved and tackling these problems will be important for the next generations of processors.

Moreover, targeting the Memory Wall, we study techniques to make a more efficient use of the memory hierarchy and to reduce data movement on chip in applications that do not benefit from it. We propose solutions from the Processing In Memory (PIM) domain to do so.

In particular, we focus on three main problems: (1) alleviating the inefficiency in performance and energy of predicated vector instructions, (2) transforming irregular memory access patterns into sequential ones, to reduce data movement on chip and enable efficient vectorization, and (3) operating near memory, in those applications that present irregular memory accesses and low arithmetic intensity.

1.1.1 Improving Predication Efficiency Through Compaction/Restoration of SIMD Instructions

In the first contribution of this thesis, we target the inefficiency of predicated SIMD instructions. These instructions are a particular type of vector instructions. They have an additional operand called mask. The mask operand is a vector register which contains as many bits as the number of elements in the other vector source registers. Only if a bit is active in the mask, the operation has to be done to the elements in that position in the other vector operands.

Using real hardware, we observe that the execution time and energy consumption of these instructions is independent of the mask operand. We believe this situation will represent a problem in future processors. For this reason, we propose a hardware/software mechanism, the Compaction/Restoration (CR) mechanism. In our proposal, the elements in a vector register whose corresponding mask bit is active are extracted and inserted into a new register. This process is named *Compaction* and it is performed for the predicated vector instructions of the same Program Counter (PC). In the best case scenario, the new registers are completely populated. These new operands will access the vector functional unit instead of the original instructions. After the operation is done, the results are moved into the active positions of the original destination vector registers. This process is called *Restoration*.

The CR mechanism can be accomplished with minimal hardware support, and loops can be marked as good CR candidates at compile time (i.e., several predicated instructions in every iteration). This information is combined with runtime information (e.g., number of active elements in the mask) to decide whether CR should be activated for the current application.

1.1.2 PLANAR: A Programmable Accelerator for Near-Memory Data Rearrangement

The second contribution of this thesis is a near-memory accelerator that performs data-layout transformations. The goal of this proposal is to reduce the data-movement on chip and to allow the compiler to provide an efficient vectorization.

Our accelerator, called PLANAR, is a novel hardware approach that performs data rearrangements near memory, converting sparse data into dense. PLANAR rearranges data enabling an efficient use of memory bandwidth by *host* cores. Moreover, PLANAR decouples access and execute, allowing the overlap of rearrangements performed by the accelerator and computation done by *host* cores. As a result, PLANAR allows applications to take better advantage of the memory hierarchy by exploiting locality of dense data, and unlocks additional performance due to better prefetching and vectorization.

PLANAR is programmed via simple library calls that can be inserted by the programmer or the compiler. PLANAR has virtual memory support and it does not require a specific memory technology (i.e., 3D-stacking) to operate.

1.1.3 REMOTE: A Programmable Near-Memory Compute Engine

The third contribution of this thesis consists of an accelerator that performs computation near-memory. It targets applications with irregular memory accesses that do not benefit from the memory hierarchy. In these applications, a deep cache hierarchy can hurt performance as every cache level increasses main memory access latency.

This accelerator, called REMOTE, is a novel hardware approach whose simple and programmable design leads to performance and energy gains in applications that suffer from irregular memory accesses. Contrary to PLANAR, REMOTE targets applications that access memory in an irregular manner but the computation is so simple that the operation could be directly done near memory rather than in the *host* core. As a result, applications benefit from a higher memory bandwidth and a reduction in the data movement on chip.

REMOTE is programmed via pragmas that can be inserted by the programmer or the compiler. Moreover, the runtime system is in charge of scheduling codes to the accelerators depending on their availability. REMOTE has virtual memory support and it does not require a specific memory technology (i.e., 3D-stacking) to operate.

1.2 Thesis Outline

The contents of this thesis are organized as follows:

Chapter 2 presents the background of the relevant hardware and software components in the context of the work developed for this thesis.

Chapter 3 introduces the simulation infrastructure used for the experiments described in the thesis, and the description of the benchmarks used for the evaluation of the proposed designs.

Chapter 4 proposes the Compaction/Restoration (CR) mechanism, explaining its functionality and its integration to an out-of-order processor. It describes the hardware required and performs a thorough evaluation, obtaining performance and energy numbers.

Chapter 5 presents the PLANAR accelerator. It describes the proposal as well as the programming application interface. The effects to the memory hierarchy are explained and performance and energy numbers are also provided.

Chapter 6 presents the REMOTE device. It describes the proposal and the changes to the application and to the runtime system. An exhaustive design space exploration is performed to obtain the most optimal hardware configuration and a detailed evaluation with a wide range of applications is presented.

Chapter 7 summarizes the contributions presented in this dissertation and provides possible directions for future work.

Chapter 2
Background

This chapter presents the background of the relevant hardware and software components in the context of the work developed for this thesis. First, vector architectures are introduced, describing their design, their operation and the main causes of performance and energy degradation. In particular, we focus on horizontal instructions, irregular memory accesses and divergence control. Further, the chapter provides a state of the art on the work done to optimize these causes of performance degradation. Second, the memory hierarchy is introduced, describing its main components, such as: caches, prefetchers, memory controllers and main memory. Next, the concept of processing in/near memory is presented and the most relevant works from the state-of-the-art are described. Finally, parallel processors and parallel programming models are described. We also present the concept of runtime-aware architectures.

2.1 Vector Architectures

Vector architectures use vector instructions to operate on the values of the vector registers, which hold multiple values rather than a single-value as in common scalar registers. In particular, every position in a vector register is called *lane*. Vector architectures are known to be very energy efficient and yield high performance whenever there is enough data-level parallelism (DLP) [120]. This phenomenon is commonly referred to as *Single Instruction Multiple Data streams (SIMD)* in Flynn's taxonomy [71]. For example, a scalar addition instruction would take values from two scalar registers A and B, and produce a result that would be stored in scalar register C, as Figure 2.1 (left) shows. A vector addition instruction would take two vectors A and B of vector length (VL) elements, and produce a final vector C of the same size, as in Figure 2.1 (right). In the Figure, the VL is 4.

The high potential of vector architectures is useful in applications that involve comparing or processing large blocks of data. Examples of these applications are multimedia process-

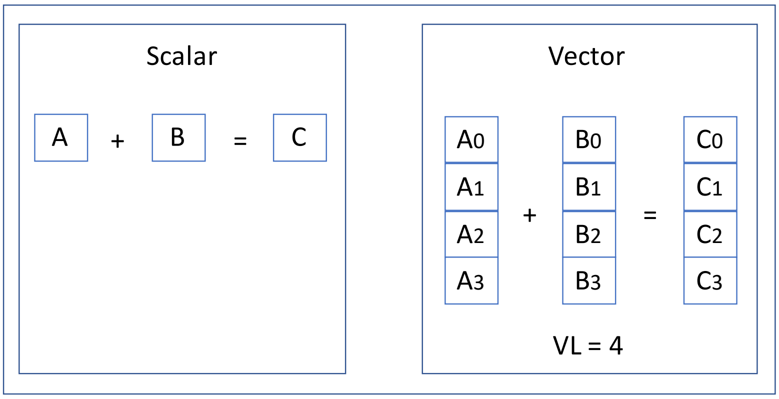


Figure 2.1: Comparison of a scalar and vector instruction.

ing (compression, graphics, audio synthesys, image processing), standard benchmark kernels (matrix multiply, FFT, convolution, sort), lossy compression (JPEG, MPEG video and audio), cryptography (RSA, DES/IDEA, SHA/MD5) and databases (hash/join, data mining, image/video serving).

2.1.1 An Example of the Micro-architecture of a Vector Processor

In order to better understand the concept of vector architectures, the micro-architecture of the VIRAM vector processor [111] is shown in Figure 2.2, focusing on the vector hardware and the memory system. VIRAM is a complete, load-store, vector instruction set defined as a coprocessor extension to the MIPS architecture [85]. It includes a vector register file with 32 entries that may store integer or floating-point elements. The vector registers contain four 64-bit element lanes which are connected to independent 64-bit functional units. The four lanes receive identical control signals on each clock cycle. The use of parallel lanes is a fundamental concept in the micro-architecture that leads to advantages in performance, design complexity, and scalability. Assuming sufficiently long vectors, VIRAM achieves high performance by executing in parallel multiple element operations for each vector instruction. Vector load and store instructions bypass SRAM caches and access DRAM main memory directly. DRAM is multi-banked to allow multiple data accesses concurrently, as vector instructions have a higher memory bandwidth utilization than scalar instructions.

2.1.2 Vector Processors in Supercomputers

Vector processors were frequently used in the past for large scientific and engineering applications. The first vector architectures in early 70s were memory-based with instructions that

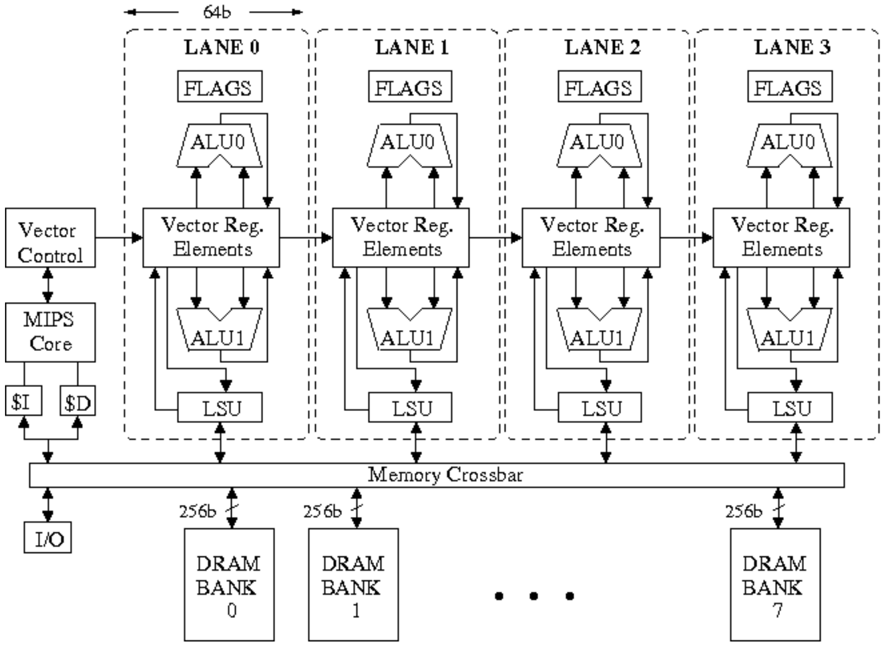


Figure 2.2: The micro-architecture of the VIRAM vector processor, from [111]

operate on memory-resident vectors [88, 191]. Later, Cray released the first commercially successful supercomputers [155]. They were register-based, providing arithmetic instructions that operate on vector registers, while separate vector load and store instructions move data between vector registers and memory.

The Japanese manufacturers, Fujitsu (VP50, VP100, VP200, VP400), Hitachi (S810) and NEC (SX) have been very successful in building vector processors for supercomputing [161]. For example, the Earth Simulator (ES) was a highly parallel vector supercomputer system based on NEC SX-6 architecture [82]. It was the fastest supercomputer in the world from 2002 to 2004. ES was replaced by the Earth Simulator 2 (ES2) in 2009, that is based on the NEC SX-92 architecture [203]. More recently, NEC's SX-Aurora processor was announced with 16,384-bit vectors [139].

2.1.3 Vector Architectures in Microprocessors

Due to the high performance of vector supercomputers, computer architects decided to incorporate the design of vector architectures to microprocessors. In the late 90s, Espasa *et al.* predicted that vector architectures would have a great potential for the future [67].

Some of examples are Torrent-0 and VIRAM [19, 111]. Torrent-0 is a vector microprocessor designed for multimedia, neural networks, and other digital signal processing tasks while

VIRAM is a vector coprocessor for the scalar MIPS processors. CODE [110], the successor of VIRAM, overcomes the limitations of conventional vector processors, such as: the complexity of a multiported centralized register file, the difficulty of implementing precise exceptions for vector instructions, and the high cost of on-chip vector memory systems.

Espasa showed that vector processors can improve their performance and hide latency by applying techniques such as decoupling, out-of-order execution, and multithreading [66]. As part of this effort, Espasa *et al.* developed Tarantula [65], a vector extension to the Alpha architecture.

2.1.4 SIMD Extensions

More recently, vector architectures have been added to scalar ISAs in the form of SIMD extensions. They appeared in the late 1990s to improve the efficiency of multimedia applications, using short vectors of 128 bits [74] and they have become ubiquitous in today's computer architectures. SIMD extensions are a particular case in vector architectures, where the width of the vector functional unit is equal to the size of the vector registers and the data is computed in parallel for the whole register. They also provide weaker memory units than the original vector machines and they do not support all gather/scatter memory operations. SIMD extensions to scalar ISAs tend to be less general-purpose, less uniform and more diversified [151].

SIMD extensions with wider vector registers have appeared in the last years, such as the 512-bit implementations from Intel [169] and Fujitsu [201], and the Arm Scalable Vector Extension (SVE) that allows up to 2,048-bit vectors [173].

2.1.5 Advantages of Vector Architectures

Vector processors and SIMD extensions have several advantages with respect to scalar ISAs:

- A single vector instruction specifies N operations, where N represents multiple operations. It dramatically reduces the pressure to the fetch and decode pipeline stages, what represents a bottleneck in conventional processors, particularly in terms of power consumption [133, 143].

- These N operations are independent. It allows simultaneous execution of all operations.

- Reduced control logic complexity. Hardware needs only to check for data hazards between two vector instructions once per vector operand. Therefore, the dependency checking logic required between two vector instructions is approximately the same as the required between two scalar instructions, but now many more operations can be in flight.

- Vector memory instructions can amortize a high overall latency, because a single access is initiated for the entire vector rather than for a single word.
- Vector memory instructions have a known access pattern. A memory system can implement important optimizations if it has accurate information on the address stream.

2.1.6 Disadvantages of Vector Architectures

Vector processors and SIMD extensions have several disadvantages with respect to scalar ISAs:

- They only work well if there is enough DLP. If the application does not contain enough DLP, the vector units will be underutilized or even idle [91].
- Vector architectures are more area and energy efficient than scalar-based microarchitectures [120]. However, the vector functional units are power-hungry and they should be disconnected in case they are not used.
- Vector memory accesses consume more bandwidth than scalar memory operations [66].
- In the event of sparse data, vector architectures are more inefficient in terms of power and energy than their dense counterparts. In these situations, just a small percentage of the data requested by a vector memory instruction is used, and later computed. Different compressed formats have been proposed to increase the performance and energy of vector architectures when dealing with sparse data to outperform scalar architectures [31, 35, 34].

2.1.7 Challenges of Vectorization

Many applications can potentially benefit from vectorized execution for better performance and higher energy efficiency [86]. Ultimately, the effectiveness of a vector architecture depends on its ability to vectorize large quantities of code [160]. However, the code vectorization faces certain challenges that require solutions, such as: horizontal operations, divergence control, and irregular memory accesses. These challenges are described below:

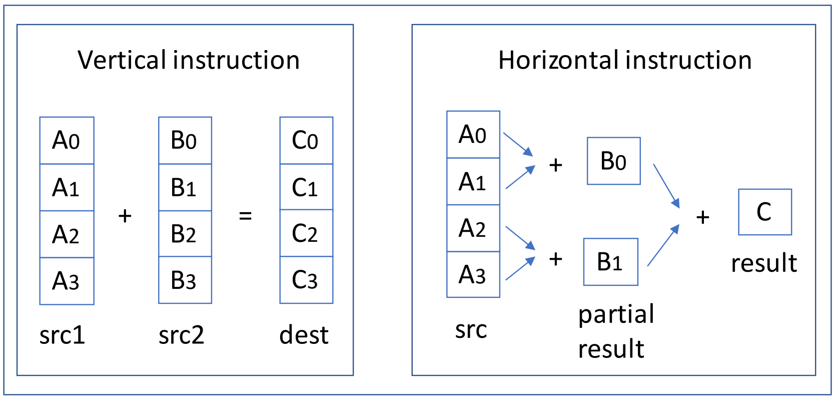


Figure 2.3: Comparison of a vertical (addition) and horizontal (reduction) instruction.

2.1.7.1 Horizontal instructions

A vector processor is able to perform two kinds of operations on a vector [92]:

- Vertical operations: operate on two vectors of the same width, and the result has the same width (e.g., a vector addition). These instructions can operate simultaneously on the source operands, since every element in the vector registers is independent of each other. If the vector functional unit has the same width as the vector operands, the whole operation could be performed in the same cycle, in the best case.

- Horizonal operations: combine multiple data items from the *same* vector. They fundamentally differ from other vector instructions in that they introduce data dependencies between different elements of the vector. They access particular lanes in the source vector operand and the result can be a scalar value. The most common horizontal instruction is a vector addition reduction, where the output is the addition of all the lanes of the vector source register. Horizontal instructions are more costly than the vertical ones, both in terms of time and hardware. However, these operations are needed so frequently in real algorithms, that they have to be part of the vector architectures.

Both instruction types are shown in Figure 2.3. In particular, a vector addition is depicted as a vertical instruction, and a vector reduction as an horizontal operation.

Much effort has been made to analyze the impact of horizontal instructions and to study other possible alternatives. Corbal *et al.* [54] demostrate that although reductions account for a small percentage of total instructions (less than 5%), their impact on final application performance can be much larger (up to 40% degradation in jpeg decode). They claim that given the current trend towards ever-increasing clock frequencies and hyper-pipelining, the latencies of horizontal operations are bound to increase. They propose two solutions to the horizontal instruction problem, and in particular to the vector reductions, such as using *packed accumulators* and *Matrix ISAs*.

2.1.7.2 Irregular memory accesses

Vector architectures exploit data-level parallelism by operating on several data elements at the same time using a single vector instruction. Due to the reduced number of instructions, vector architectures reduce the pressure to the fetch and decode pipeline stages [86]. However, they require the memory subsystem to provide data transfers with high bandwidth or the vector units will remain idle, with the energy waste it implies [120]. This problem is studied by Sebot *et al.* [162], that shows that memory was the main bottleneck for 7 of the 9 applications they optimized for SIMD. For this reason, it is important that either the application is optimized to avoid irregular memory accesses at runtime or that the memory subsystem is fast enough.

On the one hand, many proposals focus on optimizing the original application. For instance, Abel *et al*. [25] study the interactions between memory and the vector units as the memory access pattern changes. In particular, they demonstrate the impact of the data layout in memory to performance and the importance of software prefetching to reduce memory latency. Moreover, they study the performance benefits of splitting the data structures in subsets that fit the processor caches for multimedia applications. Targeting the same issue, Krishnaiyer *et al*. [112] demostrate how software prefetch and non-temporal store instructions may hide memory latencies for the Intel Xeon Phi coprocessor [168], which contains 512-bit SIMD units. They show that these instructions can be automatically generated by the compiler.

On the other hand, many proposals focus on hardware approaches to accelerate the memory subsystem for SIMD extensions. For instance, multi-bank or parallel memory structures are proposed to allow multiple data accesses concurrently [107, 176, 48]. The approach of Geng *et al.* [77] describes the design of a memory system based on a smart memory controller that automatically loads data according to the access pattern.

2.1.7.3 Divergence control

Vector architectures have vertical operations that perform the same operation on all the vector register elements. This situation happens when all the elements belong to the same execution flow. However, there may be situations where every lane in a vector register follows its own execution flow. For instance, in the event of an *if/else* conditional block, some lanes may execute the *if* portion while the other lanes may execute the *else* portion depending on the branch condition for each lane. This phenomenon is known as *control flow divergence* [183].

The control flow divergence problem appears frequently when executing vectorized codes [91]. Previous studies indicate that at least 10% of the most common vectorizable loops have divergence control issues [41]. The divergence problem is targeted both at application level and at micro-architectural level. From the software standpoint, Harrison *et al.* and Pichon *et al.* propose reordering techniques and implement math libraries so that divergence is minimized [84] [147]. From the micro-architectural perspective, the work of Smith *et al.* [166] analyzes different mechanisms to implement divergence control in the context of vector instruction sets. For example, they compare the performance of gathering only the active elements in each conditional block versus compressing data in memory and only loading the required elements. One of the proposals, Register compress/expand, compresses the active elements of a long vector into a dense one using new instructions, such as *IOTA*. It is supported in multiple vector supercomputers [155, 55, 181, 189]. Other works, such as the one of Kumar *et al.* [113], disable the lanes that do not execute on each conditional block, so that the power consumption on the vector functional unit is reduced.

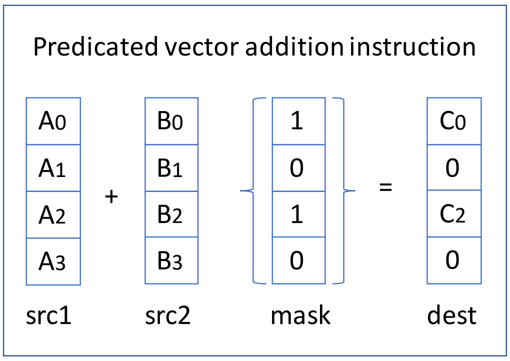


Figure 2.4: Predicated vector addition instruction.

From all the proposals, predicated execution is considered the most effective and compiler-friendly. A predicated vector instruction is a vector instruction that contains an additional source register, called *mask register*, where every lane has one bit. If the bit is one, the instruction operation for that lane, in the other source registers, has to be performed. In this case, we consider the elements in that lane position as *active*. If the bit is zero, the operation does not have to be performed and the elements in that lane position are considered *inactive*. Figure 2.4 depicts a high-level overview of how a predicated vector addition instruction is executed. Generally, the percentage of active elements in a mask register is known as the *mask density*.

If the number of active elements is high, the mask density is *dense*, if not it is *sparse*. In the example, the mask density is 50%, as only half of the elements in the mask are active.

2.2 The Memory Wall

The increase of the annual data generation rate is leading to changes in the computing paradigm and, in particular, to the notion of moving computation to data in what we call the Processing In/Near Memory (PINM) approach. A traditional computing architecture is shown in Figure 2.5. Computing units may include the CPU, GPU, and other accelerators such as a digital signal processor (DSP). Data are transferred between the computing units and main memory through the memory-hierarchy levels.

For many applications, the bottleneck of data processing for a traditional computing architecture is the bandwidth and latency of memory data transfers [205, 86]. One alternative to mitigate this problem is to exploit PIM technology.

In this chapter, the memory hierarchy is introduced, as well as the situations where the processors do not benefit from it. Finally, the PIM domain is presented. We describe the technology behind PIM and different proposals from the state-of-the-art.

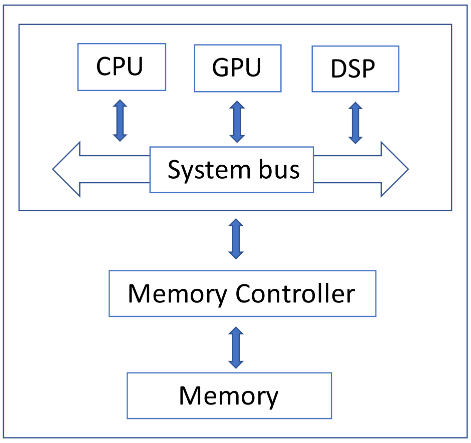


Figure 2.5: Traditional computing architecture.

2.2.1 The Memory Hierarchy

The advances in process technology have led to an ever-increasing gap between processor and memory speeds, as shown in Figure 1.2. This phenomenon is popularly known as the Memory Wall [197] and it was the main reason to add cache memories to computing systems. Caches

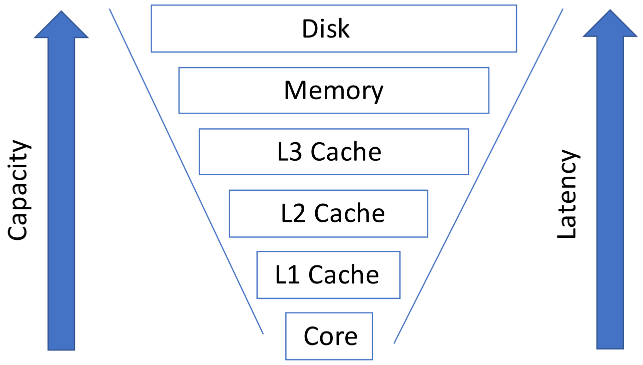


Figure 2.6: The memory hierarchy.

are a small and fast storage that keeps frequently accessed data. They offer a lower access latency compared to main memory. However, they consume more power per unit of storage.

In today's processors, it is common to see a multi-layered cache hierarchy, which employs several caches of increasing sizes and latencies. The first level of cache is typically pretty small but fast enough to keep up with the processor's demands. Instructions and data often exhibit different access patterns. To tailor the cache design to these specific needs, modern processors split instruction and data caches. Figure 2.6 depicts a high-level overview of the memory hierarchy, with three levels of cache.

The appearance of multi-core processors had an impact on the cache hierarchy as well. In general, the L1 cache is private to the core, while the upper levels can be either private or shared. Figure 2.7 illustrates the most common memory hierarchy designs encountered in modern processors. Single-core (i) and low-performance multi-cores (ii) usually employ a two-level cache hierachy where the second level is shared. More advanced designs employ three-level cache hierarchies (iii), while some of the recent processors add an L4 cache (iv) implemented in DRAM technology as a separate die on the same package.

2.2.2 DRAM Organization

Dynamic random-access memory (DRAM) is widely used as a computer's main memory. Each DRAM memory cell is made up of a transistor and a capacitor within an integrated circuit, and a data bit is stored in the capacitor. Since transistors always leak a small amount of current, capacitors will slowly discharge, causing information stored in it to drain. For this reason, DRAM has to be refreshed every few milliseconds to retain stored data.

A common DRAM organization in the modern systems is shown in Figure 2.8. Processors offer multiple memory interfaces (channels). Each channel can hold one or two DIMM modules,

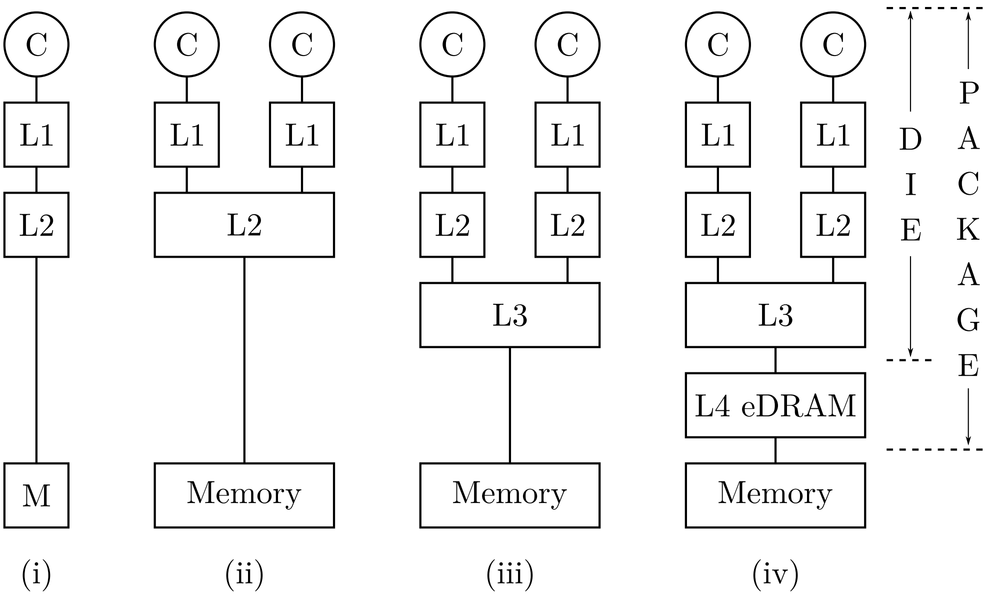


Figure 2.7: Typical memory hierarchy architectures

each of which can have up to two ranks. Within a rank, the memory is further subdivided into banks, and banks into sub-arrays. Sub-arrays within the bank can operate simultaneously. However, since all components on the same channel share physical commands, addresses and data buses, the access needs to be serialized. Devices residing on different channels can function independently from each other.

Due to the technological properties of a DRAM cell, the content of a row first needs to be loaded into a buffer before it can be accessed. Every memory request is decomposed into three commands: (i) *Activate* reads a row from a sub-array into the row buffer. (ii) *Read/Write* accesses the selected column inside the row buffer. (iii) *Precharge* writes the contents of the row buffer back into the corresponding row of the sub-array. Since an activate command destroys the original data in the row, the row buffer always needs to be written-back before a new row is activated. Depending on the row buffer status, the latency of a memory access can vary significantly. For example, if a requested row buffer already holds the necessary data, only the *Read* command needs to be issued. On the other hand, if the row-buffer holds the content of another row, the executed command sequence is *Precharge - Activate - Read*.

A memory block corresponding to a cache line is distributed across banks in a rank. Therefore, to serve a LLC cache miss, memory controller simultaneously issues appropriate commands to the banks within the selected channel, device and rank. Modern DRAM designs expose a great level of parallelism. In order to obtain higher bandwidths and maximize row

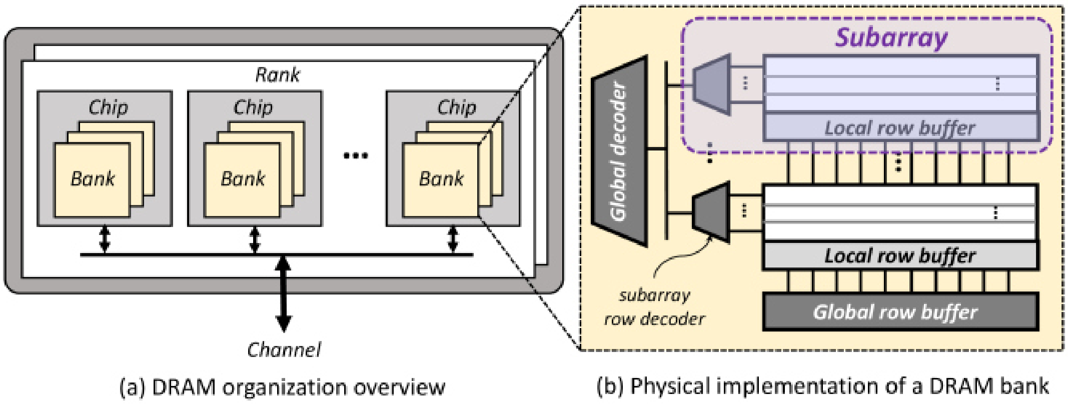


Figure 2.8: DRAM organization, obtained from [149].

buffer hit rates and bank/rank parallelism, it is necessary a careful design of the memory controller. Many techniques have been developed with this goal in mind [145, 184, 153].

2.2.3 Memory Controller

A memory controller is a component that lies between the processor and main memory and manages the flow of data in and out of DRAM. Until early 2010s, the memory controller was part of a separate chip typically called the North-Bridge. Since the appearance of the Intel Sandy Bridge and AMD Sledgehammer architectures, enabled by the increasing number of transistors on chip, the memory controllers have become a part of the processor die.

Due to the complexity of DRAM memory-access protocols, the large number of timing parameters, the innumerable combinations of memory system organizations, the different workload characteristics, and different design goals, the design space of a DRAM memory controller is really wide.

Figure 2.9 illustrates some basic components of an abstract DRAM memory controller. The memory controller accepts requests from one or more cores and one or more I/O devices and provides the arbitration interface to determine which request agent will be able to place its request into the memory controller.

Once a transaction wins arbitration and enters into the memory controller, it is mapped to a memory address location and converted to a sequence of DRAM commands. The sequence of commands is placed in queues that exist in the controller. The queues may be arranged as a generic queue pool, where the controller will select from pending commands to execute, or the queues may be arranged so that there is one queue per bank or per rank of memory. Then,

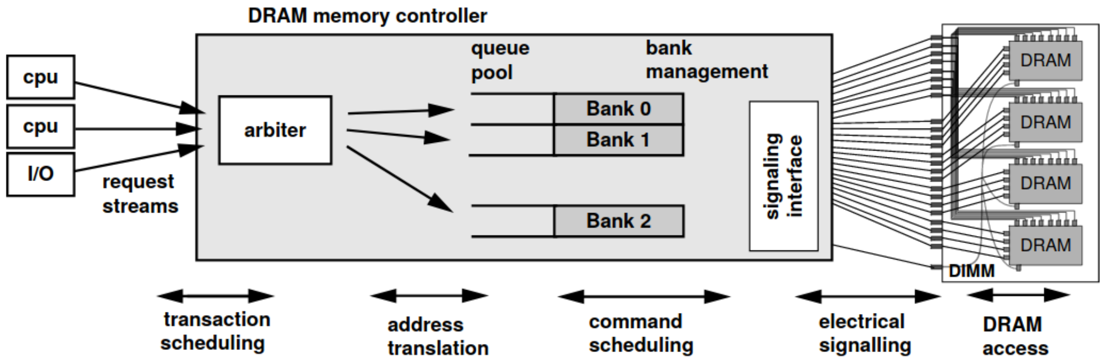


Figure 2.9: Abstract DRAM memory controller, obtained from [98].

depending on the DRAM command scheduling policy, commands are scheduled to the DRAM devices through the electrical signaling interface.

2.2.4 Prefetching

To overcome the Memory Wall, computer architects have resorted to the memory hierarchy, which relies on the memory access locality to reduce the memory access latency. Unfortunately, many important workloads exhibit adverse memory access patterns that do not benefit from the memory hierarchy. As such, processors often spend much time idling upon a demand fetch of memory blocks that miss in higher cache levels. One way to hide memory access latency is to prefetch [68]. Prefetching means to predict future memory accesses and issuing requests for the corresponding memory blocks in advance of explicit accesses. However, late or inaccurate prefetches waste energy and, in the worst case, can hurt performance.

To hide latency effectively, a prefetching mechanism must: (i) predict the address of a memory access, (ii) predict when to issue a prefetch, and (iii) choose where to place prefetched data (and, potentially, which other data to replace).

Prefetching can be controlled by hardware or software. In software prefetching [42], explicit prefetch instructions are provided and the compiler or programmer are responsible to place them in the correct code regions in the application. In hardware prefetching, changes to the software are not needed, and a new hardware is attached to the memory hierarchy. The prefetching unit monitors memory accesses and looks for common patterns. Predicted addresses are placed into the prefetch queue, which is only checked when no processor requests are waiting. Prefetching requests look like read requests to the memory hierarchy. Prefetchers trade memory bandwidth for latency. Commercial processors have multiple prefetchers, which are usually closer to the core, as it is easier to detect memory access patterns.

The prefetching mechanism has been deeply studied. For example, Lee *et al.* [118] study the benefits and limitations of hardware and software prefetching. Similarly, Byna *et al.* [36] discuss various issues that have to be considered in designing a prefetching strategy for multi-core processors. Due to the advantages of prefetching, it is now being widely used in high-performance processors, for example, Intel Xeon [100, 60], and IBM POWER [156].

2.2.5 Limitations of the Memory Hierarchy

The memory hierarchy and prefetching have been proposed as solutions to mitigate the effects of the Memory Wall. In this scheme, small cache memories with low latencies can meet data demands while prefetchers can act in the background to hide memory access latency. Deep cache hierarchies are the natural result of this trend, providing low-latency data access to high-performance out-of-order processing elements. Applications that exhibit locality of reference can benefit from this hardware organization [171, 182]. However, recent trends show that we have effectively plateaued on the effectiveness of data prefetchers [132] and that they have become ineffective for irregular memory access patterns [207].

As a result of sparsity and irregular reuse distances, some studies have measured utilization of transmitted bandwidth as low as 20% for some applications [29]. Srinivasan *et al.* prove that a significant fraction of the data brought into the last level cache goes unused before eviction [172]. The data moved throughout the memory hierarchy that is not used is popularly known as the *dark bandwidth* [28]. Dark bandwidth results into more energy, higher memory access latency and less usable memory bandwidth.

Making matters worse, many applications that exhibit poor prefetching behavior have dependent or indirect access loads [64], meaning that every cache level adds to the overall round-trip access latency. This issue has a bigger impact on multi-core systems, where resources are shared and every core competes for the memory bandwidth.

Performance is not the only factor affected by data motion. Approximately two thirds of the energy required to compute is consumed by data movement, specifically by the memory and interconnect [33].

There are a few options to increase bandwidth utilization and reduce data movement. Some researchers suggest that byte-level addressing is the key to improve bandwidth utilization. However, this type of system is impractical from an engineering perspective. Another alternative is Processing In/Near Memory (PINM), where the computing elements are placed next to where data resides. In the following sections, we will discuss the PINM technology and proposals from the state of the art.

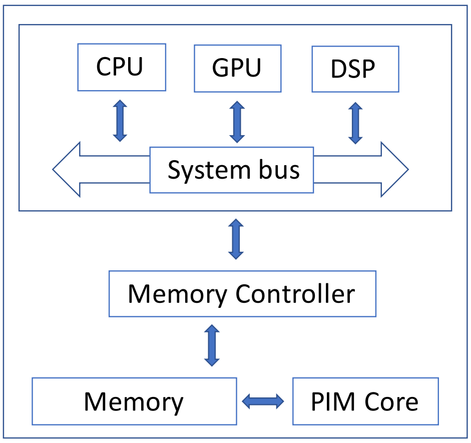


Figure 2.10: Processing In Memory concept.

2.2.6 Processing In/Near Memory

In a traditional computing architecture, data is moved towards a CPU independently of where it resides, as depicted in Figure 2.5. However, applications with irregular memory accesses do not benefit from the memory hierarchy available in a traditional computing architecture. With the evolution of emerging DRAM technologies, Processing In/Near Memory (PINM) has now become of great interest to academia as well as different industries [180, 204]. Figure 2.10 illustrates the PINM concept.

PINM is usually split into two categories, (i) Processing In Memory (PIM), where the new compute engine is tightly integrated with the memory and usually requires a specialized memory technology, and (ii) Processing Near Memory (PNM), where compute logic is placed near memory to exploit low latency and high bandwidth of near-memory data accesses.

2.2.6.1 Processing Near Memory

Near-memory computing refers to bringing logic or processing units closer to memory. Notwithstanding the closer integration, processing units still remain distinct from memory arrays. Near-memory computing has been explored at various levels of the memory hierarchy. For example, Wei *et al.* [192] propose an in-order processor connected to the memory controller. It contains a scratchpad, scalar and vector units and 64x64 bit memory to perform bit-level operations. It is programmed through memory-mapped operations. Solihin *et al.* also add a processor to the memory controller [170]. In this case, the new hardware performs correlation prefetching, although the prefetching scheme may be customized depending on the application. The Scatter-Add proposal [56] extends the memory controller to enable parallel execution of atomic operations, that serialize execution in data-parallel architectures. Similarly, Zhang *et al.* [206] present the Impulse memory controller. It performs gather/scatter operations as the core requested memory belonging to particular regions. It receives a rearrange function and transforms a sparse data structure into a dense structure. The work of Beard [29] also performs data structure rearrangement but contrary to Impulse, this operation can be done ahead of time and be overlapped by computation in the host core. His proposal, The Sparse Data Reduction Engine, may be placed anywhere in the cache hierarchy.

Lockerman *et al.* [124] propose Livia, a tiled-multicore system where every tile has a chunk of L3. Every tile contains an out-of-order core with L1 and L2 caches, and an in-order core (an accelerator) connected to the L2 and L3. These in-order cores were shared by all the tiles. Depending on the location of the data, the task is migrated from the original out-of-order core to the tile where the data resides and it is computed in the accelerator.

Ozdal *et al.* [142] connect hardware accelerators to the DRAM. They contain special hardware to deal with graph applications, and they may be combined to provide parallelism.

Other proposals modify the SRAM cell to support simple operations in the cache [105, 108]. For example, the work of Aga *et al.* [1], Compute Caches, implements operations such as copy, search, compare and logical operators in the caches.

Finally, the NYU supercomputer [81] supports atomic operations inside network switches.

2.2.6.2 Processing In Memory

The technological advances of chip fabrication, (e.g., 3D-stacked memory designs), help to pack much more DRAM cells on a single chip. Moreover, they allow a better integration between memory and compute logic. For example, the High Bandwidth Memory (HBM) [117, 104] and the Hybrid Memory Cube (HMC) [144] are commercial implementations of a 3D-stacked memory connected to a logic layer via through-silicon vias (TSV) [177]. Figure 2.11 depicts a high level overview of a 3D-stacked memory connected to a logic layer.

Since their introduction, many proposals based on these designs exploit the in-memory computation capabilities. The Active Memory Cube [138] uses HMC as the base of its design. It offers a significant amount of parallelism by having multiple *lanes* in the logic layer with scalar and vector units. Tesseract [2] is also implemented on the HMC. It consists of 512 in-order cores that communicate with each other using a message passing protocol. It targets graph applications. GraphPIM [136] proposes to execute graph workloads directly on HMC using new HMC atomic operations.

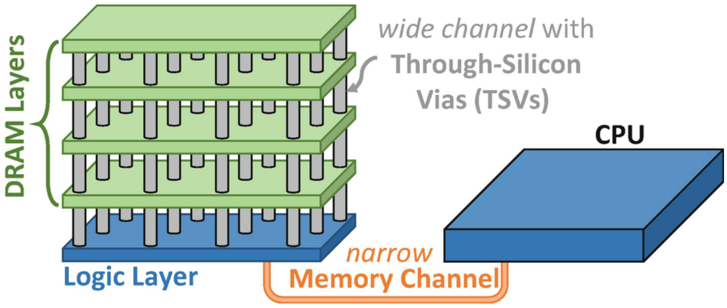


Figure 2.11: High-level overview of a 3D-stacked DRAM based architecture, obtained from [79].

The data rearrangement engine (DRE) [123] performs in-memory data restructuring to accelerate irregular, data-intensive applications. Authors add some logic to the HMC's logic layer to perform in-memory operations and propose an API to program the new hardware.

In TOM [90], a host GPU is interconnected to multiple 3D-stacked memories that have small lightweight GPU cores. Authors present a compiler framework which automatically identifies possible offloading candidates and which uses a mapping scheme which ensures data and code co-location.

Lee *et al.* [119] analyze the architectural behavior of search applications. In particular, they focus on the *k*-nearest neighbors algorithm. Their study reveals a high percentage of vector operations and memory reads, which confirms that vector operations are important for search applications and that they are bound by high data movement. Based on the observation, they integrate specialized vector processing units in the HMC's logic layer and propose instruction extensions to leverage those hardware units.

The Mondrian Data Engine [62] consists of a mesh of HMC with tightly connected Arm cores in the logic layer. Authors demonstrate that a hardware and software co-design is needed to achieve an efficient performance in PIM systems.

Similarly, Liu *et al.* [122] propose an heterogeneous PIM architecture to train deep neural network models. In particular, the logic layer of their 3D-stacked memory comprises programmable Arm cores and large fixed-function units. A runtime system dynamically maps and schedules the kernels, based on an online profiling.

Singh *et al.* [165] evaluate these works and observe that certain challenges currently prevent a wide adoption of these designs. While they provide notable performance improvements over the traditional paradigms, a lack of programming model support and the resulting increase in application complexity are still open issues in the current state of the art.

2.3 Parallel Programming for Shared-Memory Systems

2.3.1 Parallel Processors

Traditionally, software has been written in a serial/sequential fashion, where instructions are executed one after another and only one may execute at any moment in time. Improvement in computer performance was implemented through clock rate ramping in order to provide faster execution of the instructions. However, increasing the clock frequency hit a wall (see Figure 1.1). As a consequence, and enabled by Moore's Law, computer architects decided to keep packing more transistors on a single chip but to use them to pack multiple processors. Not long afterwards, the first multi-core processors were introduced, as shown in Figure 2.12, where multiple cores collaborate to solve a computational problem.

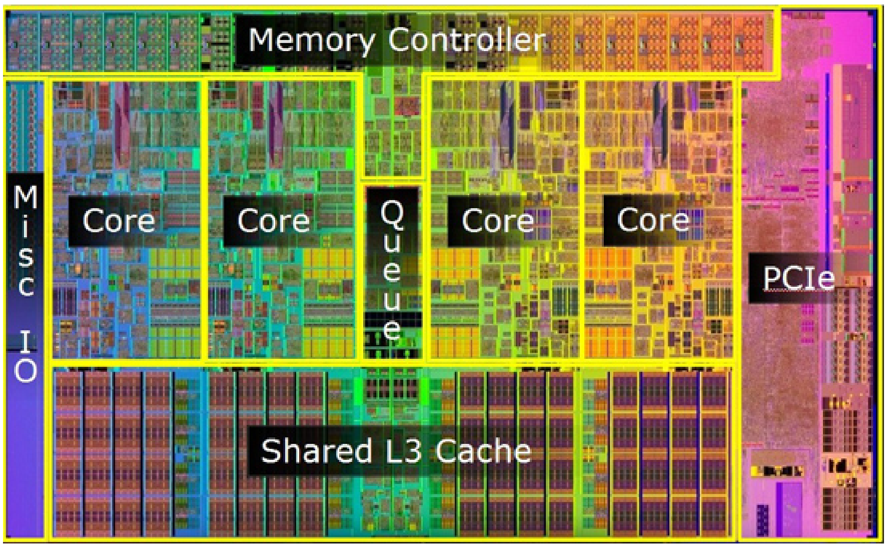


Figure 2.12: Multi-core processor. Four cores are connected to the same L3 cache. Obtained from [131].

Some recent designs employ heterogeneous architectures, which combine low-power, slower cores with high-performance cores. Arm's bigLITTLE [10] architecture is an example of an heterogeneous processor design. In this case, the cores in the big cluster are only activated if the workload is really demanding.

In order to program a multi-core processor it is necessary a new programming model. Parallel programming models are explained in the next section.

2.3.2 Parallel Programming Models

Parallel programming models exist as an abstraction of hardware and memory architectures. In fact, these models are not specific and do not refer to particular types of machines or memory architectures. Parallel programming models represent the way in which the software must be implemented to perform a parallel computation. Each model has its own way of sharing information with other processors in order to access memory and divide the work. Popularly, two main parallel programming models exist:

- Message passing. This programming model is usually applied in the case where each processor has its own memory (i.e., distributed memory systems). The programmer is responsible for determining the parallelism and data exchange that occurs through the messages, in a shared network, whenever a synchronization is needed, as shown in Figure 2.13.

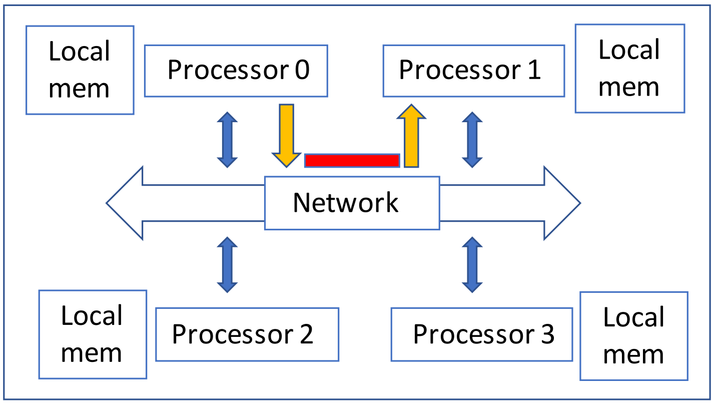


Figure 2.13: Example of a distributed system with four processors, where every processor has a local memory. If they want to communicate, they need a message passing protocol. In this case, processor 0 sends a message to processor 1.

  For example, the Message Passing Interface (MPI) [53] is a specification for the developers and users of message passing libraries. It was created in 1980 and supports both point-to-point and collective communication. MPI remains the dominant parallel programming model used in high-performance computing today [174]. The MPI API provides a set of functions to let two processors communicate. If processor 0 wants to send a message to processor 1, it will use the *MPI_send* function, whereas processor 1 will utilize the *MPI_receive* function.

- Shared memory. In this programming model, processors share a common address space, which they read and write to asynchronously. Various mechanisms such as locks/semaphores are used to control access to the shared memory, manage contention and to prevent race conditions and deadlocks. All processors see and have equal access to shared memory, as Figure 2.14 illustrates.

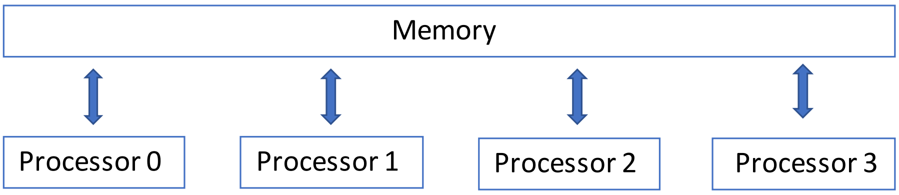


Figure 2.14: Example of shared memory system with four processors. If they need to synchronize, they use the shared memory.

For example, the OpenMP standard [141], created in 1996, allows parallel programming in a shared memory system in a *fork-join* fashion. In this case, neither communication nor data distribution is needed. For loops are a common target for parallelization in parallel codes, achieved by using *#pragma parallel for* annotation before a for loop in OpenMP. The supporting library automatically creates threads and distributes the loop iterations among them. To support intra-thread synchronization, programmers can use *atomic* constructs to guard the access to a certain variable. The specification offers a customizable scheduling policy to achieve the best load balancing among threads by using the directive *schedule*. Aside from loops, it is possible to manually define sections that are executed by different threads using *pragma omp parallel*.

An alternative to the fork-join paradigm is the use of *task* as a unit of parallel work. Tasks are viewed as portions of the serial code that can execute asynchronously with other tasks while respecting the synchronization points between them. The programmer splits the sequential code into tasks and defines the dependencies between them. During the execution, the main task creates user-defined tasks until it arrives to a explicit synchronization primitive, such as *taskwait* in OpenMP, which pauses the main task until all the children tasks complete. Upon its creation, a task is added into a task dependency graph (TDG) as *pending*. When all dependencies of a task are fulfilled, meaning that all the predecessor tasks completed their execution, the task is considered a *ready* task and it can be assigned to a thread to be executed. Examples of task-based programming model are OpenMP 3.0 [141] and OmpSs [63].

Other popular parallel programming models exist, such as Threading Building Blocks (TBB) [150] and Transactional Memory (TM) [87]. Similar to OpenMP, TBB breaks computation down into tasks that can run in parallel. The library manages and schedules threads to execute these tasks. On the other hand, TM requires hardware support that has recently been adopted by major hardware vendors like Intel [200], IBM [188], and Arm. TM implementations can differ significantly as there are many implementation choices [16, 163, 179, 83].

2.4 Runtime-Aware Architectures

The evolution in processor manufacturing has led to complex hardware designs. This makes efficient programming of such systems more difficult. Historically, the hardware and software design has been decoupled to ease the programmability and provide code portability. However, in order to achieve a good performance it is important to fully exploit hardware resources. To that end, the hardware implementation details need to be known at the software level.

In order to target this situation, Valero *et al.* [185] propose the concept of *Runtime-Aware Architectures* where hardware and software are managed by an intermediate layer, the *runtime system*. It manages the hardware and software and provides a set of optimization techniques that are not feasible in the current computer designs. Moreover, Casas *et al.* [43] explore the potential of the runtime system-level information in the hardware and software design. This may ultimately lead to a better overall performance, lower energy consumption and reduced programming complexity of future systems.

Many recent works have focused on studying and optimizing the runtime system. For example, Chasapis *et al.* propose a job scheduling algorithm for power-restricted NUMA systems [49]. Castillo *et al.* [44] design a runtime-assisted management of the frequency of the cores depending on the criticality of running tasks. Alvarez *et al.* [5, 4] propose a runtime-guided management of scratchpad memories. Sanchez *et al.* [158, 157] perform partitioning of the task-dependency graph to reduce the data movement in NUMA systems. Caheny *et al.* [40, 38, 39] present a runtime optimization to reduce cache coherence traffic in NUMA systems and to deactivate coherence for data that does not need it. Finally, Jaulmes *et al.* [101, 102, 103] analyze the utility of runtime systems for reliability.

Chapter 3
Experimental Methodology

This chapter presents the methodology followed in this thesis. The first section describes the simulator used for the evaluation and details of the simulated architectures. The second section introduces the benchmarks employed. Finally, the third section briefly presents the metrics used to evaluate the proposals developed in this thesis.

3.1 Simulation Infrastructure

3.1.1 Simulator

The *gem5* simulator [30, 126] has been used to model the hardware extensions proposed in this thesis. *gem5* is an execution-driven multi-core full system simulator that can do a cycle accurate execution of a complete operating system. *gem5* supports various ISAs with different CPU and memory models ranging from pure functional ones to highly detailed and cycle accurate.

gem5 supports checkpointing and KVM Emulation [159] to accelerate system and benchmark initialization using less detailed CPU and memory models. In this thesis we employ the checkpointing capabilities so that simulations start right at the parallel sections.

The experiments in this thesis have been done using three different configurations as listed in Tables 3.1, 3.2 and 3.3. In the second proposal, the PLANAR devices reside outside of the coherent network, so specific flush/invalidate requests are needed to maintain the data coherence between the *host* cores and the accelerators. In the third proposal, the *host* cores and the devices belong to the same coherent network.

In Chapter 4 we employ the x86-64 ISA in the simulator, while Chapters 5 and 6 are based on the Armv8 ISA. The reason for this is that, at the time the thesis started, Arm had not released the Scalable Vector Extension (SVE) [173]. This situation made us resort to Intel's SIMD extensions to test our hardware proposals. Nevertheless, due to an internship to

Arm Limited in 2017 and to the release of SVE, we moved from x86-64 to Armv8. All the experiments are run with the most detailed configurations available for each architecture trying to resemble a real system.

During the development of this thesis, we needed to extend *gem5* with the Intel's SIMD extensions. We did the following contributions to the official *gem5*:

- We re-implemented the Streaming SIMD Extensions (SSE) [93], which operates on 128-bit vector registers. In particular, we compared the statistics of *gem5* with the performance counters of a real machine and we discovered a huge increment on the μoperation count in the simulator. We realised that there was a lack of vector registers in *gem5*, and that the μoperations of SSE were modelled as two 64-bit scalar μoperations. For this reason, we implemented a vector register file and adapted the original SSE instructions to work with proper vector operands.

- We implemented the Advanced Vector Extensions [94] (AVX2 and AVX512) to operate with 256 and 512-bit vector registers. This process was done following closely the architecture as explained in the Intel's manuals. In particular, every addressing mode and feature, such as predication, was considered during the decoding of these vector instructions in *gem5* and modelled as in the manual description. Overall, this *gem5* extension accounts for 500K+ lines of code, 7400+ macro instructions and 2000+ μinstructions.

- We modified the context switch mechanism to properly save/restore vector registers.

- We accurately modelled 42 different SIMD instruction types, with the corresponding issue and execution latencies reported by Fog [72].

- We have adopted the configuration of different x86 processors, such as a latency and a throughput-oriented implementation based on the Icelake (ICE) [187] and the Knights Landing (KNL) [169].

- We created a semi-automatic validation framework which allowed us to compare the statistics provided by *gem5* to a real machine. This tool was used to find several sources of error that altered the expected behavior of the simulated processor, which we later documented and corrected. This work was accepted and published in a journal:

  *J. M. Cebrián, **A. Barredo**, H. Caminal, M. Moretó, M. Casas and M. Valero, "Semi-automatic Validation of Cycle-Accurate Simulation Infrastructures: The Case for gem5-x86", 2020 Future Generation Computer Systems.*

Table 3.1: Configuration of *gem5* simulations for the first proposal.

Chip details	
Cores	1 single-threaded out-of-order x86 core, 2GHz
Core details	
Fetch, decode, rename width	4 insts/cycle
Dispatch, issue, commit width	4 insts/cycle
Branch target buffer	1 way, 2048 entries
Branch predictor, Branch target buffer	Bimode, 8K+8K entries
Fetch Buffer, Decode Buffer	16B, 56-μops
Fetch, Load and store queues	32 entries, 90 entries, 72 entries
Physical registers	200 integer + 360 floating point
Issue queue, re-order buffer	196 entries, 320 entries
Functional units	1 Int ALU + 3 Int/FP/SIMD ALU
Instruction latencies (int)	add (1c.), mul (4c.), div (22c.)
Instruction latencies (FP)	add (5c.), mul (5c.), div (22c.)
Instruction latencies (Icelake SIMD)	add (3c.), mul (5c.), div (14c., 8c. issue), sqrt (16c., 10c. issue)
Instruction latencies (KNL SIMD)	add (6c.), mul (10c.), div (30c., 16c. issue), sqrt (40c., 20c. issue)
L1 instruction cache	32KB, 8-way, 1 cycle access lat.
L1 data cache	32KB, 8-way, 4 cycle access lat.
L2 unified cache	4MB, 16-way, 12 cycle access lat.
CR structures	
Compaction Unit	1 pipelined unit, 2 stages
Restoration Unit	1 pipelined unit, 2 stages
Dense Ticket Table	64 entries, 8 bits per entry
Compactable Instruction Table	160 entries, 170 bits per entry

The energy consumption and the microprocessor area are evaluated using McPAT [121]. McPAT is an integrated power, area and timing simulator for multi-core architectures built on top of CACTI [135, 134, 24]. It models various processor components, such as cores, including the functional units, caches, on-chip interconnections and memory controllers. We add the model of a vector functional unit (VFU) to perform a power analysis in our first contribution. In particular, we scale the model of the scalar functional unit by a factor. As we simulate the AVX-512 ISA in *gem5*, we consider this factor to be 8 (i.e., 8 double-type elements fit in a single 512-bit vector register). The accuracy of the built-in models is improved by incorporating the changes suggested by Xi *et al.* [198]. *gem5* is extended with appropriate counters to record the necessary statistics corresponding to the hardware components.

3.1.2 Environment

In the first contribution, the simulated system is an Ubuntu v16.04 with a Linux kernel v4.9.4. Benchmarks are compiled with GCC v5.5 using the “-O2” optimization flag. We do not employ

Table 3.2: Configuration of *gem5* simulations for the second proposal.

Chip details	
Cores	8 single-threaded out-of-order Arm cores, 2GHz
Core details	
Fetch, decode, rename width	4 insts/cycle
Dispatch, issue, commit width	8 insts/cycle
Branch target buffer	1 way, 2048 entries
Branch predictor	Bimode, 8K+8K entries, RAS 16 entries
Load and store queues	48 entries, 48 entries
Physical registers	256 integer + 256 floating point
Issue queue, re-order buffer	92 entries, 192 entries
Functional units	3 Int ALU + 2 FP/SIMD ALU
Instruction latencies (int)	add (1c.), mul (3c.), div (12c.)
Instruction latencies (FP)	add (5c.), mul (4c.), div (9c.)
L1 instruction cache	48KB, 3-way, 64B/block, 1 cycle access lat.
L1 data cache	32KB, 2-way, 64B/block, 2 cycle access lat.
L2 banked unified cache	2MB, 16-way, 64B/block, 12 cycle access lat.
Prefetcher	Stride prefetcher
Memory details	
Type	DDR4 2400
Channel	2 channels, 16GB/s per channel
PLANAR details	
Number of devices	8
μCore	in-order core, single-threaded, 2GHz
Functional units	1 Int ALU
Instruction latencies (Int)	add (3c.), mul (3c.), div (9c.)
L1 instruction μcache	1KB, 2-way, 64B/block, 1 cycle access lat.
L1 data μcache	1KB, 2-way, 64B/block, 2 cycle access lat.
Translation lookaside buffer (μTLB)	8 entries

Table 3.3: Configuration of *gem5* simulations for the third proposal.

Chip details	
Cores	8 single-threaded out-of-order Arm cores, 2GHz
Core details	
Fetch, decode, rename width	4 insts/cycle
Dispatch, issue, commit width	8 insts/cycle
Branch target buffer	1 way, 2048 entries
Branch predictor	Bimode, 8K+8K entries, RAS 16 entries
Load and store queues	48 entries, 48 entries
Physical registers	256 integer + 256 floating point
Issue queue, re-order buffer	92 entries, 192 entries
Functional units	3 Int ALU + 2 FP/SIMD ALU
Instruction latencies (int)	add (1c.), mul (3c.), div (12c.)
Instruction latencies (FP)	add (5c.), mul (4c.), div (9c.)
L1 instruction cache	64KB, 3-way, 64B/block, 1 cycle access lat.
L1 data cache	32KB, 2-way, 64B/block, 2 cycle access lat.
L2 unified shared cache	512KB, 16-way, 64B/block, 12 cycle access lat.
L3 unified shared banked cache	16MB, 16-way, 64B/block, 20 cycle access lat.
Prefetcher	Stride prefetcher in L1 and L2
Memory details	
Type	DDR4 2400
Channel	2 channels, 16GB/s per channel
REMOTE details	
Number of devices	1/2/4/8/16/32
μCore	in-order core, single-threaded, 2GHz
Functional units	2 Int ALU, 1 FP ALU
Instruction latencies (Int)	add (6c.), mul (6c.), div (18c.)
Instruction latencies (FP)	add (12c.), mul (12c.), div (12c.)
L1 instruction μcache	1KB, 2-way, 64B/block, 1 cycle access lat.
L1 data μcache	1KB, 2-way, 64B/block, 2 cycle access lat.
Translation lookaside buffer (μTLB)	8 entries

"-O3" since it enables auto-vectorization and manually vectorizing our applications with Intel's intrinsics [95] provides better performance. In the second and third contributions, the simulated system is an Ubuntu v18.04 with a Linux kernel v4.15. Benchmarks are compiled with GCC v7 using the "-O3" flag.

We use the cluster *arvei* (now *sert*) at the *Departament d'Arquitectura de Computadors* in the *Universitat Politècnica de Catalunya* to run our experiments on real machines. The cluster consists of 4,111 cores with x86_64 processors from different manufacturers and generations. We utilize the newest nodes, in particular, the AMD EPYC 7101p [7] at 2.80GHz and the Intel Xeon E5-2630L v4 [96] at 2.20GHz. The software stack comprises an Ubuntu v18.04 with a Linux kernel v5.3.0-61-generic.

3.2 Benchmarks

The benchmarks used for the evaluation of the proposals in this thesis are selected among HPC benchmarks, graph applications and other kernel codes to cover a wide range of algorithms. The codes in the first proposal are single-threaded, but the ones for the second and third contributions are parallel as they were written in OpenMP [27] and OmpSs [63] programming models. Most of the benchmarks are chosen from larger collections, such as the ParVec Benchmark suite [45], Coral-2 benchmarks [114], NAS Parallel Benchmarks [47], the PERFECT suite [115], the GraphBIG suite [137] and the BSC Application Repository [46].

Some of the applications are manually vectorized by the author of this thesis to exploit better the SIMD resources. The remaining of this section describes the benchmarks corresponding to each proposal, including proposal-specific changes introduced in each code, the input parameters and some code properties relevant to each contribution.

3.2.1 Benchmarks for the Divergence Proposal

Table 3.4 lists the benchmarks employed in the divergence proposal and their description. They are manually vectorized using Intel intrinsics, as compilers frequently try to minimize the number of predicated instructions and, since our proposal targets the predication issue, we need that the compiler generates codes with this type of instructions. The inputs of these applications are images, signals and arrays which do not exceed 8MB of total memory footprint.

Figure 3.1 shows the instruction breakdown of the main loop in the region of interest of each benchmark. In particular, we differenciate SIMD instructions (regular, high-latency

Table 3.4: Benchmarks used to evaluate the proposal about divergence.

Benchmark	Description
Bilateral Filter (B-Filter)	It is a non-linear, edge-preserving, and noise-reducing smoothing filter for images. It replaces the intensity of each pixel with a weighted average of intensity values from nearby pixels [178].
Convolution (Convol)	A signal convolution [164].
Gaussian Blur (G-Blur)	An image being blurred by a Gaussian function [76].
K-means (Kmeans)	It partitions N observations into K clusters in which each observation belongs to the cluster with the nearest mean [109].
k-nearest neighbors (KNN)	It finds the distances between a query and all the examples in the data, selecting the specified number examples (K) closest to the query, then votes for the most frequent label [148].
Quadratic equation (Quadr)	It performs the quadratic equation to an input array.
Random Number Generator (RNG)	A Box-Muller number generator [61].
Sound distorter (S-Distort)	A form of audio signal processing used to alter the sound [50].
Distance Calculator (Stream)	A distance calculator based on Streamcluster [45].

predicated instructions, low-latency predicated instructions) and scalar instructions. Loops contain between 9 and 58 instructions. The predicated instruction percentage is between 21% (KNN) and 72% (B-Filter).

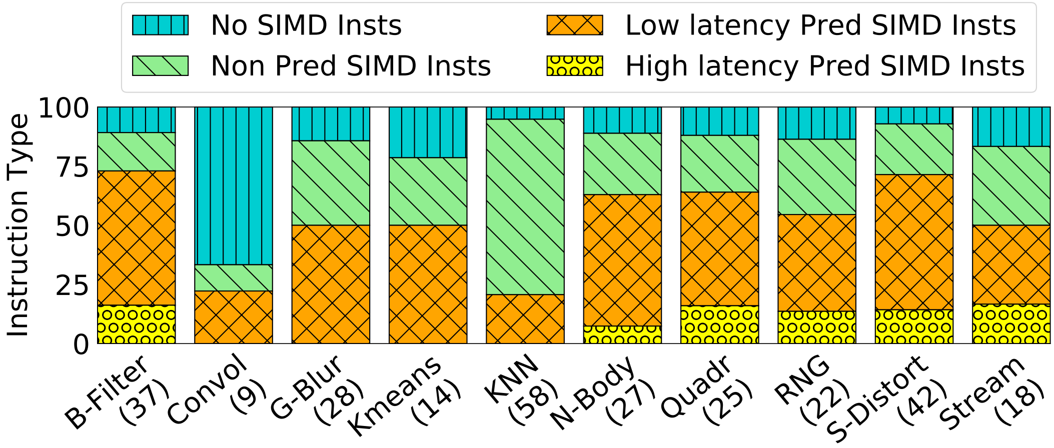


Figure 3.1: Loop iteration breakdown. In the X axis, the applications name and their number of instructions per iteration. In the Y axis, the instruction type percentage in every iteration.

3.2.2 Benchmarks for the Near-Memory Data Rearrangement Proposal

Table 3.5 lists the benchmarks employed in the near-memory data rearrangement proposal, their description and their input. The matrices are obtained from the University of Florida Sparse Matrix Collection [58], which has sparse matrices collected from a wide range of real applications. The table also contains the number of different rearrangements, split among the available devices. For example, if 8 devices are available, every rearrangement will be performed on 4 devices in the benchmark named “SymGS”. The evaluated benchmarks contain strided and irregular memory accesses, have a low cache block utilization and do not benefit from the memory hierarchy. The eliglible codes can be optimized by creating a new version of the data where data resides sequentially in memory.

The selected benchmarks are modified to work with the accelerators. This process involves: (i) to define the rearrange function; (ii) to replace the original irregular accesses of the original data structures to the dense one; and (iii) to add the device allocation, offload, and release calls. In most of the mentioned benchmarks, very few modifications are required to the original code: $\approx$20 lines of code for the rearrange function, the three device library calls, allocation via regular malloc/free of the dense data structure, and the code modification to access the new dense data.

3.2.3 Benchmarks for the Near-Memory Computing Proposal

Table 3.6 lists the benchmarks employed in the near-memory compute proposal, their description and their input. The matrices are obtained from the University of Florida Sparse Matrix Collection [58], which has sparse matrices collected from a wide range of applications.

In this case, we consider applications that contain irregular memory access patterns. We split these benchmarks in two categories: (1) graph applications, which have an elevated data movement on chip, do not benefit from the memory hierarchy and perform a simple computation on the data, and (2) HPC applications, which suffer from the same issue but benefit from the memory hierarchy to access particular data structures and have a higher arithmetic intensity than the graph applications. A profiling of these benchmarks is done in Section 6.5.

The selected benchmarks are modified to work with the accelerators. We identify the code regions which can be offloaded to the near-memory devices to obtain a performance benefit and to reduce data movement on chip. These regions are marked with a pragma and compiled so that the runtime system performs the code offloading.

Table 3.5: Benchmarks used to evaluate the proposal about data layout transformation.

Benchmark	Description	Input	# rearr.
Multigrid compute (CompMG)	An algorithm for solving differential equations using a hierarchy of discretizations, from HPCG [152].	Matrices: bcspwr10 (A), bcsstk15 (B), blckhole (C), circuit_1 (D), ex12 (E), lns_3937 (H), G30 (F), jan99jac100sc (G)	2
Extended box filtering approximation (EBOX)	An extended box filtering approximation of a Gaussian convolution for an image [78].	Stride: 8, 16 and 32. 400,000 double-type elements	4
Matrix-matrix block multiply (MatMul)	An optimized matrix-matrix multiplication with blocking support [80].	1x1 block of 400x400 elements, 2x2 blocks of 200x200 elements, 4x4 blocks of 100x100 elements, 2x2 blocks of 300x300 elements, 3x3 blocks of 200x200 elements, 6x6 blocks of 200x200 elements	1
Meabo	A multi-phased multi-purpose benchmark. Used for energy efficiency studies [12]. It accesses memory using a pseudo random indirection vector.	Phase2, 300,000 double-type elements	1
Spatter	Kernel used for timing scatter/gather kernels on CPUs and GPUs [116].	Distance: 1, 2, 4, 8, 16, 32, 64, 128, 256, random, 300,000 double-type elements	1
Sparse Matrix-Vector multiply (SpMV)	The sparse matrix is represented in the CSR format [31] and the vector is dense.	Matrices: A, B, C, D, E, F, G, H	1
STRIDE	Memory stress benchmark commonly used to characterize the memory system of HPC systems.	Distance: 1, 2, 4, 8, 16, 32, 64, 128. 320,000 double-type elements	1
Symmetric Gauss-Seidel smoother (SymGS)	Symmetric Gauss-Seidel smoother, from HPCG [152]. It performs a forward and backward triangular solve.	Matrices: A, B, C, D, E, F, G, H	2

Table 3.6: Description of the benchmarks used to evaluate the near-memory compute proposal.

	Benchmark	Description	Input
Graph applications	Breadth-First Search (BFS)	It traverses a graph. It starts at the tree root or some arbitrary node, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.	–dataset LDBC/output-100k/ –root 31
	pageRank	It counts the number and quality of links to a page to determine a rough estimate of how important the website is.	–dataset LDBC/output-100k/ –quad 0.001 –damp 0.85
	k-Core decomposition (kCore)	It removes all the vertices that have degree less than K from the input graph.	–dataset LDBC/output-100k/ –kcore 6
	Graph coloring (graphCol)	It is special case of graph labeling. It is an assignment of labels traditionally called "colors" to elements of a graph subject to certain constraints.	–dataset LDBC/output-100k/
	Shortest path (SPath)	It finds a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.	–dataset LDBC/output-100k/ –root 31
	Triangle count (trCount)	It finds a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.	–dataset LDBC/output-100k/
	Connected component (CComp)	It computes connected components for a given graph. Connected components are the set of its connected subgraphs. Two nodes belong to the same connected component when there exists a path between them.	–dataset LDBC/output-100k/
	Degree centrality (DCentr)	It measures the number of incoming and outgoing links from the node.	–dataset LDBC/output-100k/
HPC applications	Random access (randAcc)	It measures the rate of integer random updates of memory.	Table 256MB
	Histogram (hist)	Calculates a histogram of weighted averages using a 3D 27-point stencil over a $N \times N \times N$ cube represented by a dense 3D matrix of floating point numbers.	1D of 3D matrix: 220
	Meabo	A multi-phased multi-purpose benchmark. Used for energy efficiency studies [12]. It accesses memory using a pseudo random indirection vector.	Array of 9,000,000 double-type elements
	Spatter	Kernel used for timing scatter/gather kernels on CPUs and GPUs [116].	Stride 16, 32, 64. Array of 9,000,000 double-type elements.
	1D Particle in Cell	A kernel from a 1D particle-in-cell code used for kinetic simulations in physics [128].	83MB
	Sparse Matrix-Vector multiply (SpMV)	The sparse matrix is represented in the CSR format [31] and the vector is dense.	Matrices: italy_osm, tx2010, ecology1, webbase-1M
	Symmetric Gauss-Seidel smoother (SymGS)	Symmetric Gauss-Seidel smoother, from HPCG [152]. It performs a forward and backward triangular solve.	Matrices: italy_osm, tx2010, ecology1, webbase-1M

3.3 Metrics

The evaluation of the proposals in this thesis is performed by analyzing several performance metrics. *gem5* provides metrics that measure the execution time, depending on the number of cycles and the frequency of the simulated processor. It also gives statistics regarding the memory hierarchy and the network on chip.

The comparison relative to the baseline architectures is done by using equation 3.1.

$$Metric = \frac{Metric_{baseline}}{Metric_{proposal}} \tag{3.1}$$

To compare the proposals in the general case, the metric values corresponding to different benchmarks are aggregated to provide a single measure of performance. For the metrics defined as ratios, such as speedup, geometric mean is used (equation 3.2). Metrics that represent absolute values are averaged using arithmetic mean (equation 3.3).

$$Geometric\ mean = \sqrt[n]{value_1 \times value_2 \times \cdots \times value_n} \tag{3.2}$$

$$Arithmetic\ mean = \frac{value_1 + value_2 + \cdots + value_n}{n} \tag{3.3}$$

The cache missrate is obtained at the Miss Status Holding Registers (MSHR). This is because two consecutive cache misses to the same block address only generate one request to the next level in the memory hierarchy, after checking in the MSHR that the first miss is being handled. Thus, cache missrate is calculated using the equation:

$$Cache\ missrate = \frac{MSHR\ misses}{MSHR\ total\ accesses} \tag{3.4}$$

Cache misses are combined with the instruction count to form a compound metric of misses per kilo instructions (MPKI) using the following equation:

$$MPKI = \frac{Cache\ misses}{\frac{Total\ executed\ instructions}{1000}} \tag{3.5}$$

The performance in terms of power is obtained from McPAT, and is used to compute the energy using the following equation:

$$Energy = Power \times Execution\ time \tag{3.6}$$

Data movement on chip is calculated by performing the addition of the size of all the packets moved between the cores and main memory, using the following equation:

$$Data\ movement = \sum_{i=1}^{n_{cores}} Bytes\ transferred\ between\ core_i\ and\ L1D_i + \tag{3.7}$$

$$\sum_{i=1}^{n_{cores}} Bytes\ transferred\ between\ L1D_i\ and\ L2 + \tag{3.8}$$

$$Bytes\ transferred\ between\ L2\ and\ LLC \tag{3.9}$$

Memory bandwidth is obtained by performing the addition of the size of all the data moved between the memory controllers (MC) and main memory, and dividing it by the execution time of the application. It is compared to the maximum theoretical bandwidth. It is done using this equation:

$$Memory\ bandwidth = \frac{\sum_{i=1}^{n_{MC}} Bytes\ transferred\ between\ MC_i\ and\ Memory}{Execution\ time} \tag{3.10}$$

Chapter 4

The Efficiency of Predicated SIMD Instructions

4.1 Introduction

This chapter presents a hardware solution to target the inefficiency of predicated instructions in SIMD extensions. As explained in Section 2.1.7.3, predication is the most common approach to deal with *control flow divergence* in vector architectures. However, we observed that the performance and energy consumption of predicated SIMD instructions is independent on the number of active elements in the mask operands (mask density). Instead, current designs lead to performance and energy be proportional to the vector length (VL). We evaluated this problem in real hardware on an Intel Xeon Platinum 8160 processor [97].

With the current trend of doubling the register size every four years [86], SIMD implementations with VL-time performance will become extremely energy inefficient when executing predicated instructions. Thus, there is an urgent need towards SIMD implementations with mask density-time performance for predicated executions.

In order to target this issue, we propose a novel hardware mechanism, the Compaction/Restoration (CR) design. CR identifies code sections with SIMD instructions guarded by a mask, extracts the active elements from the predicated instructions belonging to different loop iterations, and compacts them into a single dense instruction. Such dense instructions are executed efficiently with density-time performance and energy. Finally, their results are restored to the original predicated SIMD instructions.

Moreover, CR improves the performance of unmodified legacy code by dynamically and transparently compacting several vector instructions into a wider register ISA.

CR could be combined with compiler information to detect code regions that benefit more from our proposal.

Next, we list the main contributions of this proposal:

- The CR hardware design to enable density-time performance and energy efficiency for predicated SIMD instructions. CR requires minimal hardware support to compact predicated instructions. A detailed design space exploration is performed to properly size the CR hardware structures.

- An exhaustive evaluation with a full system cycle-accurate simulator. Our evaluation shows that CR achieves an average of 11% speed-ups, while reducing dynamic energy consumption by an average of 16%.

- CR transparently executes unmodified legacy code with 256-bit Advanced Vector Extensions (AVX-2) [93] on a newer architecture with twice longer vectors. By using the 512-bit registers and VFUs in AVX-512 [94], CR achieves an average of 17% speed-ups on unmodified AVX-2 applications.

4.2 The Predication Problem in SIMD Extensions

In SIMD extensions, the latency and energy of predicated instructions depends on architectural vector length (VL), not on the number of elements to be executed. This situation has become a challenge for current and future processors, that will contain wider vector registers. Many studies have measured the mask density[1] on modern codes, and results into 18-20% on typical benchmarks [75, 183, 45]. Such a low mask density means that current SIMD extensions waste a significant portion of energy on unnecessary computations, and increase contention in the VFU, which can hurt performance.

To illustrate the divergence control problem for predicated SIMD instructions, we analyze the performance degradation and energy waste in a set of benchmarks[2] with different mask densities. The selected representative benchmarks contain AVX-512 instructions (VL=512 bits), with a wide range of SIMD instructions types including different percentages of predicated instructions.

For the selected benchmarks, Figure 4.1 shows the potential performance degradation and dynamic energy waste with several percentages of active elements in the masks. Mask densities range from 12.5% to 87.5% in increments of 12.5%. The processor employed in this evaluation has a configuration similar to Intel's Knights Landing (KNL) [169]. Knowing the execution

[1] Percentage of active elements in the mask register.

[2] Section 3.2.1 describes in detail the employed benchmarks.

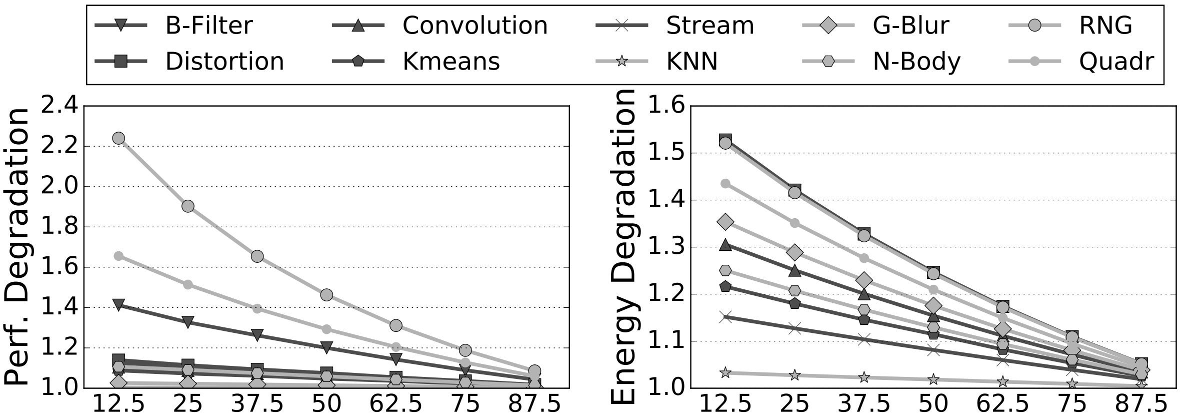


Figure 4.1: Performance and dynamic energy degradation for predicated SIMD applications with different mask densities.

time and energy of each vector instruction, we estimate the performance and energy for the mask density of each scenario. These results are estimated with respect to an ideal processor with density-time performance and energy efficiency.

Results with several mask densities show no significant difference in time for the evaluated configurations. Indeed, AVX-512 has a *VL-time performance* in the evaluated architecture, which is not optimal. As a result, performance significantly degrades with respect to an ideal density-time SIMD implementation. All benchmarks are sensitive to mask density, with performance degradations ranging from 4% (G-Blur) to 2.2× (RNG) with 12.5% mask densities.

In the case of dynamic energy, a density-time implementation reduces VFU energy linearly with the mask density. In benchmarks with a high percentage of predicated SIMD instructions such as S-Distortion or B-Filter, this translates into a significant waste in dynamic energy (up to 54% with 12.5% mask densities). On average, dynamic energy waste is 35% with 12.5% densities.

The results shown in Figure 4.1 make clear that significant fractions of energy and performance are wasted in current SIMD implementations with VL-time performance and energy efficiency. In the next section we introduce CR, a hardware proposal that achieves *density-time* performance and energy efficiency in predicated SIMD instructions without any code transformations.

4.3 The Compaction/Restoration Mechanism

The CR approach aims at achieving *density-time*[3] performance and energy efficiency in predicated instructions in SIMD extensions without user mediation. CR could be combined with information from the compiler to know which code regions could benefit more from the compaction/restoration mechanism.

4.3.1 Overview

CR targets SIMD extensions available in current processors (such as AVX [94]), where vector length (VL) is equal to the VFU width. CR creates a *dense* version of several dynamic predicated SIMD instructions for a certain program counter (PC). The active elements[4] of these instructions are selected and *compacted* into a dense instruction. Candidates for compaction delay execution until dense registers are full. In the best scenario, this dense instruction has source registers with all elements active and is executed instead of the original instructions. As a result, contention and the number of accesses to the VFU decreases. This is crucial for performance and energy efficiency, since VFU can add up to 75% of the total power dissipated by the core [167]. Once the dense instruction is executed, results are *restored* back to the original destination registers.

CR can be implemented in any architecture with predication support. Modern SIMD architectures with variable-length vectors, such as RISC-V [190] and Arm *Scalable Vector Extension* (*SVE*) [173] can also benefit from CR. These processors know the register length at runtime and CR needs the same information. In this proposal, we have deployed CR in an out-of-order processor with 512-bit VFUs. Section 4.3.2 describes the new hardware components to support CR, while Section 4.3.3 contains a detailed description of the changes required to an out-of-order pipeline to implement CR. Afterwards, we describe the different phases in the CR mechanism: i) detection of compactable instructions (Section 4.3.4), ii) compaction of dense instructions (Section 4.3.6), iii) execution of dense instructions (Section 4.3.7), and iv) restoration of compacted instructions (Section 4.3.8). Next, we present a case study with CR (Section 4.3.10). Finally, we describe how CR can be used to execute SIMD legacy code on newer and wider SIMD extensions (Section 4.3.11) and discuss other considerations (Section 4.3.12).

[3] Results are relative to the mask densities.

[4] Elements whose corresponding mask bits are true.

4.3.1.1 Basic Functionality Example

CR basic functionality is shown in Figure 4.2. In this case, there are two predicated instructions with 50% mask densities, corresponding to two loop iterations for the same PC. In particular, for the instruction 0, active lanes[5] are 0 and 3, and for the instruction 1, the active lanes are 1 and 2. CR decides to compact both instructions into a single dense instruction. After compaction, the dense instruction is executed and the result is restored to the corresponding lanes of the destination registers of the original instructions 0 and 1.

Figure 4.3 shows a time diagram of the same example comparing the baseline to CR. In the baseline, the second predicated instruction cannot access the VFU as it is busy executing operations of the same type. In CR, the execution of the first instruction is delayed until the dense registers become full (best case scenario). After compaction, only one instruction is executed reducing VFU contention. Finally, a pipelined restoration phase happens and results are committed.

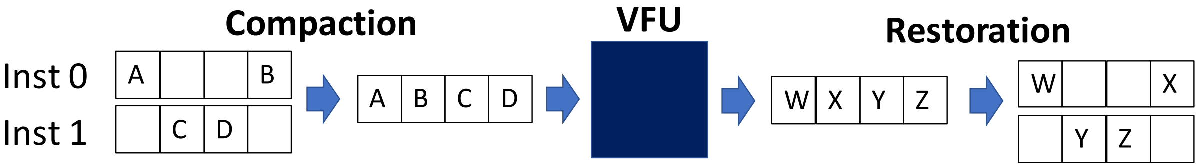


Figure 4.2: CR basic functionality. In this case, two intructions for the same PC with 50% mask densities are compacted and restored.

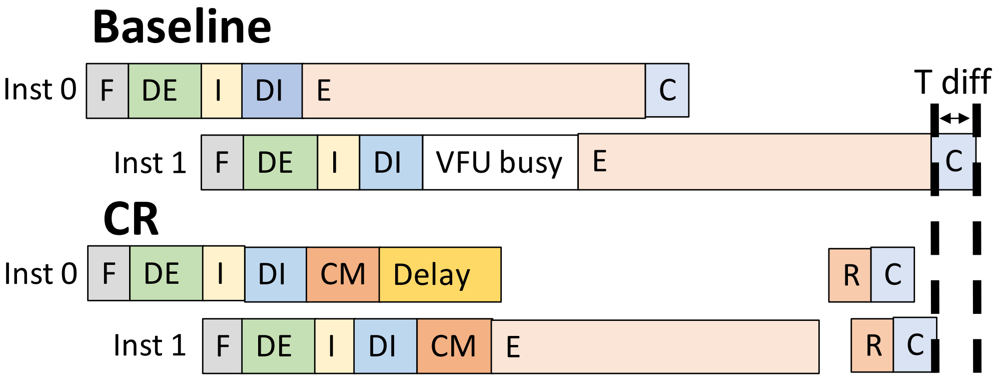


Figure 4.3: Time diagram comparing the execution of baseline vs CR in the pipeline. The pipeline stages are: Fetch (F), Decode (DE), Issue (I), Dispatch (DI), Execute (E), Commit (C), Compact (CM), Restore (R).

[5] Position in a vector register that contains an element.

4.3.2 CR Hardware Components

CR hardware components are described below.

1) The **Compactable Instruction Table** (CIT) is a direct-mapped table which contains the information regarding to dense instructions and their compactable instructions. It is needed to perform the Compaction (Section 4.3.6) and Restoration (Section 4.3.8) phases. Table 4.1 defines the functionality and size of every CIT entry. In this case, we target double-precision operations although finer-grain operations can be supported (e.g., machine learning). It would require more bits per entry but the chances of finding a non-true element would be higher, increasing CR efficiency. The number of CIT entries should be smaller than the maximum amount of in-flight instructions. In our design, CIT entries must be filled with at least one compactable. Thus, the maximum number of entries is $ROB\ Entries/2$ although we did not exceed half of its capacity.

Table 4.1: CIT entry fields, size in bits.

Dense instruction information		
Capacity	Number of elements the dense instruction may handle	4
Alloc Occupancy	Number of elements allocated by compactable instructions	4
Insert Occupancy	Number of elements inserted by compactable instructions	4
Last Insertion	Cycle the latest compacted instruction was inserted	6
isSquash/isTimeout	Whether dense instruction was squashed/timeout triggered	1
Insert_d	Whether dense instruction was inserted	1
Compactable Instruction Information		
Mask	Instruction mask bits	8
Dest Reg Idx	ROB entry where instruction is stored	8
Allocate	Whether instruction is allocated	1
Insert_c	Whether instruction is inserted	1

Next, we describe how and when the entry bits in the CIT are modified. First, we start with the CIT fields regarding the dense instruction. The “capacity” is updated when the dense instruction is created, with the number of lanes in the vector register for that instruction (i.e., it depends on the vector register length and the data type specified in the instruction). The “alloc occupancy” is modified as an instruction is marked as compactable, adding to the current value the number of active elements in the instruction, at the issue stage. If a new compactable instruction for this PC is encountered, and the “capacity” and “alloc occupancy” have the same value, the current dense instruction for this PC is full and a new dense instruction is required. The “insert occupancy” is updated when all the source operands of any of the compactable instructions for this PC are ready. The content of this field will be modified, adding to the current value the number of active lanes in the compactable instruction. In this case, the compaction phase can happen for this instruction and the "last insertion" is modified with the current cycle. If the "insert occupancy" and the "capacity" have the same value, the dense instruction for this PC can proceed to the execute stage. In case the dense instruction is squashed (e.g., branch missprediction) or a timeout is triggered, the "isSquash/isTimeout" is set to one. Finally, the "insert" field will be set to one after the compaction phase of one compactable instruction for this PC has finished. It ensures that the dense is executed in case of squash or timeout.

Second, we continue with the CIT fields regarding the compactable instruction. As the predicated instruction is marked as compactable, the "mask" is updated with the content of the mask register and the "dest reg idx" with the index of the destination register. Both fields are needed to perform the compaction and restoration phases. The "allocate" bit is also set to one to indicate that this entry is allocated. If a new compactable instruction for the same PC is encountered, if it is not full, the following entry will be accessed if it is, a new dense instruction will be required. Finally, after the source operands for the compactable instruction for this PC are ready, the "insert" bit will be set to one. This information is required so that the compaction/restoration phases occur for this particular compactable instruction.

2) The **Dense Ticket Table** (DTT) is a direct-mapped table which keeps track of the latest created dense instruction for every PC. It facilitates the accesses to the CIT, since there can be multiple dense instructions for the same PC waiting to be executed. The DTT holds a set of unique keys or *tickets*, representing CIT entry identifiers. Every dense and compactable instruction keeps a ticket to access the CIT. The number of DTT entries is limited by the number of instructions in every loop iteration, a maximum of 60 in our applications. By indexing DTT entries using the 10 lowest PC bits, we avoid conflicts. If no entry exists for a particular PC, a new one is created and a new ticket is chosen from the DTT. If a new dense instruction is created, the existing DTT entry for that PC gets a new ticket. Tickets are restored as the associated dense instructions commit. The ticket size is limited by the number of in-flight dense instructions (i.e., $\log_2 ROB\ Entries/2$ bits).

3) The **Compaction Unit** moves active lanes from source vector registers in compactable instructions to the assigned dense registers. It happens separately for every source register as they become ready. It receives a vector register and a mask as inputs and a dense register as output. Section 4.3.6 describes the Compaction phase and Section 4.4.1 explores its design space.

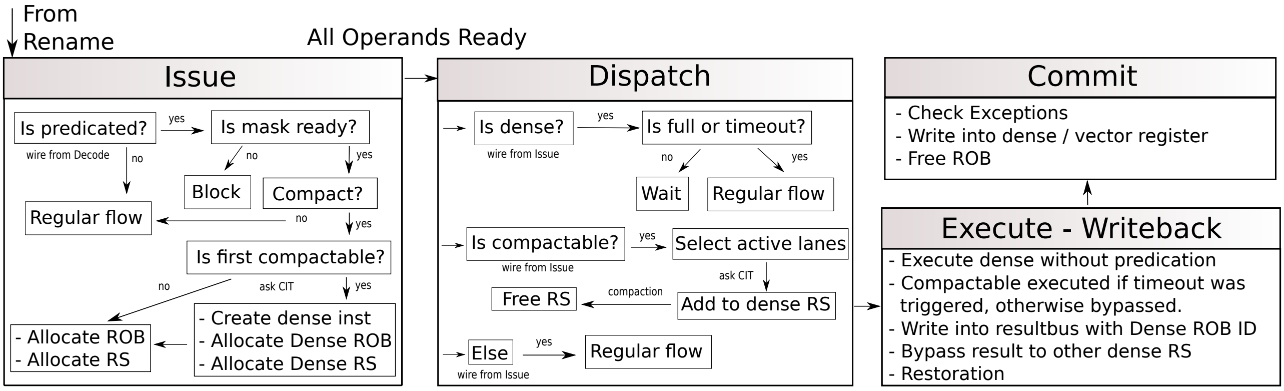


Figure 4.4: CR overview when incorporated to an out-of-order processor.

4) The **Restoration Unit** restores the results of an executed dense instruction back to the original destination registers. The dense destination register elements are moved to the corresponding active lanes of the destination registers. It receives a dense vector register and a mask as inputs and a vector register as output. Section 4.3.8 describes the Restoration phase, while Section 4.4.1 performs a design space exploration to size it.

4.3.3 CR in an Out-of-Order Processor

Next, the main functional changes to incorporate CR into a classic out-of-order processor are described. Figure 4.4 depicts the whole process in a state-diagram style.

1) **Decode**: In case a predicated instruction is found a signal is sent to Issue stage.

2) **Issue**: If the signal from Decode is active and the mask register is ready, a logic decides whether the instruction has to be compacted or not (see Section 4.3.4). If so, it is marked as compactable. Then, the DTT and CIT are accessed to know if it is the first compactable instruction for that PC or if existing dense instructions for that PC are already fully occupied. In case a new dense instruction is required, a dense instruction is created and its operands are renamed. The DTT creates and stores a new ticket, which is provided to the compactable instruction and employed to create a new CIT entry. In the CIT entry, the *Capacity* field is updated with the total number of lanes in the dense register. A reservation station (RS) and a re-order buffer (ROB) entry are allocated for the dense instruction. Also, a dense destination register is reserved in the Register Alias Table (RAT) to allow operand forwarding. Candidates to be compacted on it are given the DTT ticket after their mask operand becomes ready. Finally, the *Alloc Occupancy*, *Mask*, *Dest Reg Idx* and *Allocate* CIT fields are updated with the compactable instruction information.

3) **Dispatch**: As compactable operands become ready, the compaction occurs independently for every compactable instruction and their RS are freed. The *Insert Occupancy*, *Insert$_c$* and *Last Insertion* fields in the CIT are updated. Once dense operands are full, a timeout occurs, or a squash happens, the instruction becomes ready to execute.

4) **Execution**: The dense instruction is executed and compacted instructions are bypassed (Section 4.3.7). If the dense destination register is used by subsequent dense instructions, it is forwarded (Section 4.3.9).

5) **Writeback**: The dense instruction is written in the ROB and the restoration is performed to copy the results to the original destination registers (Section 4.3.8).

6) **Commit**: Dense and compacted instructions commit sequentially, ensuring speculation and exception handling are performed in-order.

4.3.4 Detecting Compactable Instructions

To have a simple CR implementation, we currently consider all loops as compaction candidates. However, in a preliminary analysis (Section 4.4) and in the evaluation (Section 4.6) we observe that several factors should be considered to enable an efficient CR mechanism: i) predicated instruction latency, ii) number of instructions per iteration, iii) inter-loop dependencies, iv) mask densities and v) processor events that hide CR latencies.

The first three factors can be statically determined and have important effects on performance. For instance, inter-loop dependencies cause an execution serialization. On the other hand, mask densities are fundamental and input-dependent (see Section 4.4). Finally, some processor events, such as cache misses, pause the core backend hiding CR latencies. A compiler may analyze the first three static factors and produce a hint to enable CR, if the two latter factors happen at runtime, for every loop (e.g., using a memory-mapped register).

Predicated SIMD instructions that fulfill all these factors cause a CIT allocation, becoming *compactable*. CR distinguishes between CIT *allocation* and *insertion*. Allocation is done in program order, while insertion may happen out of order. Allocation reserves the CIT entries which will be later filled in the insertion step. Insertion is performed as compactable instructions become ready. Ensuring program order in insertion is critical to enable *dense register forwarding* (described in Section 4.3.9).

4.3.5 Populating Dense Instructions

In order to populate a dense register, compactable instructions delay execution until it is full or a timeout triggers. The ROB is used as a *buffer* to obtain candidates for compaction. Some events, such as cache misses, pause the core backend until they are resolved. For this reason, regular processor behavior may hide the delayed execution and it may not affect performance in many situations (e.g., irregular memory accesses).

4.3.6 Compaction Phase

In this phase, active elements from compactable instructions in an RS are moved into the RS belonging to the dense. The CIT is accessed to obtain information about the compaction. It occurs as source operands of compactable instructions become ready, and after CIT insertion is done. The compaction phase does not require extra ports or buffers as the VFU already reads all inputs from the RS simultaneously. When compaction finalizes, compactable instructions are called *compacted*.

Figure 4.5 shows the compaction phase for one instruction which has a mask density of 50% (lanes 0 and 3). After accessing the CIT, the compaction unit extracts the active elements (A and D) and places them in the first lanes of an empty dense instruction.

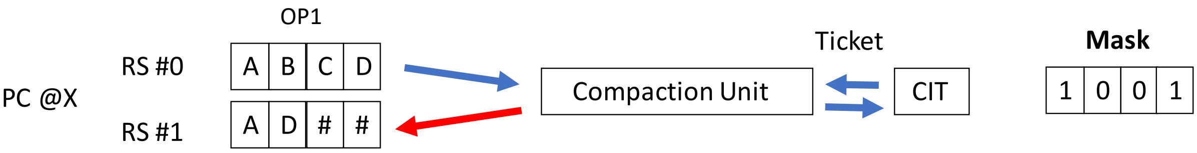


Figure 4.5: Compaction phase for one compactable instruction with a mask density of 50%.

4.3.7 Execution of Compacted Instructions

Once the dense instruction is ready in the RS, it is executed. Dense instructions can be ready due to three reasons: i) dense operands are completely populated; ii) a squash happens; or iii) a timeout is triggered.

The first case is the ideal scenario for CR, minimizing the number of SIMD ALU accesses as a result. In this case, compacted instructions are not executed (they are bypassed to the next pipeline stages). It also facilitates *dense register forwarding* to dependent instructions.

When a squash happens, CR removes allocated, but not yet inserted, compacted instructions from the CIT entry, forcing the dense to become ready to execute.

Finally, multiple timeout policies are incorporated into CR to avoid delaying too much the execution of predicated SIMD instructions.

Postponing the execution of predicated instructions increases the utilization of internal processor resources, potentially stalling the pipeline and slowing down the whole application. For this reason, two timeout policies are created. They stop the allocation/insertion of new CIT entries and trigger the dense instruction execution.

1) **Resource occupancy**. The lack of free hardware resources prevents instructions from entering into the pipeline, and thus, it may not allow dense operands to be completely populated. This situation may lead to performance degradation. For this reason, if resources are occupied above a certain threshold, the CIT forces the execution of dense instructions whose *Last Insertion* field is higher than a timeout. CR considers the occupancy in the reservation station (RS), the ROB, and the Load-Store Queue (LSQ).

2) **Circular dependencies**. A dense instruction could have allocated but not inserted compacted instructions waiting for dependencies to be freed. If the dependency is associated to another dense instruction, execution is blocked. For this reason, if the dense maximum commit time is exceeded execution is forced.

If a timeout is triggered, the remaining allocated but not yet inserted compactable instructions referring to that CIT entry will execute the ordinary way. Section 4.4.2 studies the impact of the timeout policies.

4.3.8 Restoration Phase

In the Restoration phase, the elements from dense destination registers are moved into the active lanes of the destination vector registers from the original compacted instructions. Restoration is performed in the Writeback stage, after the dense instruction is executed and after its result is placed on its ROB entry. It happens in parallel with the *dense register forwarding*. Restoration can be done in parallel for every compacted instruction. The CIT is accessed to get the information of every compacted instruction. The dense instruction keeps the ticket provided in the Compaction phase to know its corresponding CIT entry.

Figure 4.6 shows the restoration phase for a dense instruction. In this case, the dense instruction contains two compacted instructions, with mask densities of 50% (i.e., in the first instruction the active elements are lanes 0 and 3, and in the second, lanes 0 and 2). For every compacted instruction, the CIT is accessed to obtain the mask and index of the destination registers. First, the restoration unit will extract the first two lanes (E, F) from the dense destination register and will insert them in the lanes 0 and 3 of the destination register of the first compacted instruction. Second, the last two lanes will be extracted from the dense destination register (G, H) and will be inserted in the positions 0 and 2 of the second compacted instruction.

In the Restoration phase, multiple data values must be written to the ROB. This phase is usually out of the critical path of execution, as the dense version of the instruction executes. Thus, this phase can be handled by buffering writes to the ROB not requiring extra ports.

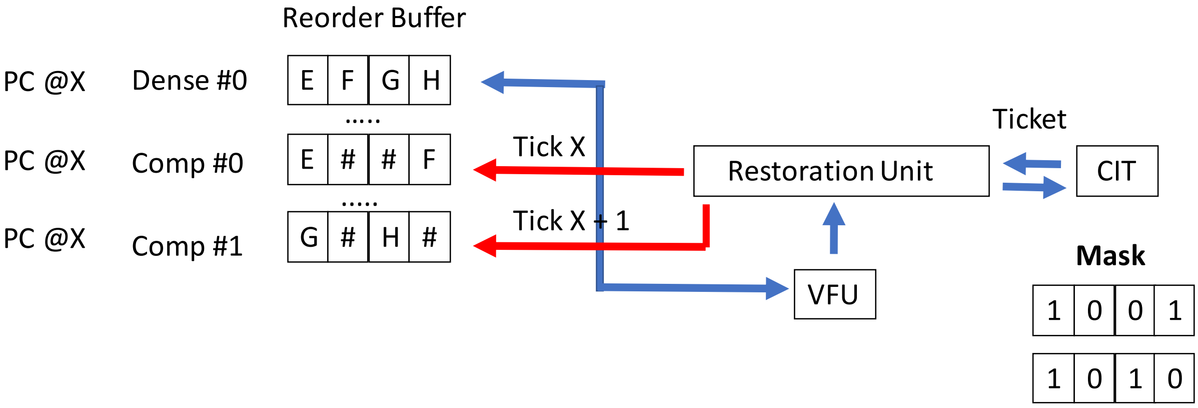
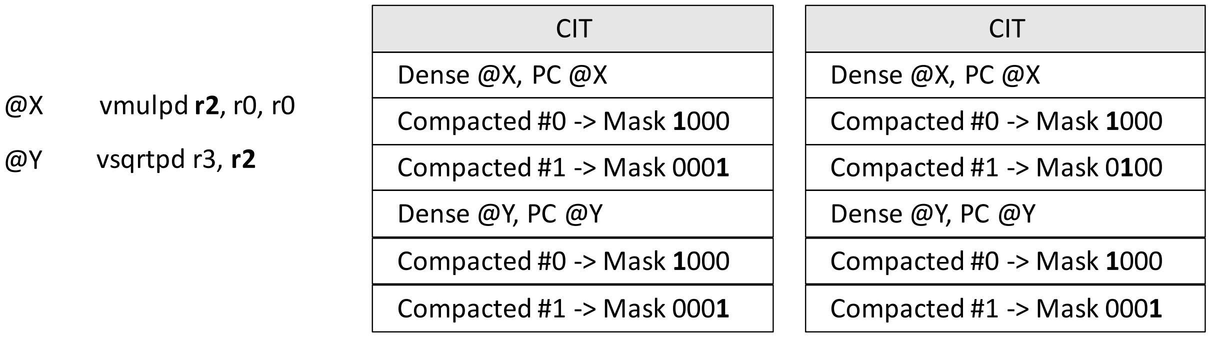


Figure 4.6: Restoration phase for one dense instruction containing two compacted instructions with a mask density of 50%.

4.3.9 Dense Register Forwarding

A dense register can be forwarded if it is fully occupied or if the dense instruction and its dependent ones share the same inserted compacted instructions positions. The $Insert_c$ CIT entry bit provides this information for every allocated compactable instruction. If not, the remaining uninserted compactable instructions will be compacted. An efficient dense register forwarding reduces CR latencies and hides the restoration process.

Figure 4.7 shows two cases of dense register forwarding, where the dense instruction @Y is dependent on instruction @X due to the dense register “r2”. In the left, the dense instructions @X and @Y contain compactable instructions that share the same active element positions. In this case, the dense instruction @Y does not require its instructions to be compacted, as the dense destination register from @X is in compacted form because the compactable instructions in instruction X were already compacted. For these instructions, there is dense register forwarding of 100%. In the right, the second compacted instruction from @X does



```
@X    vmulpd r2, r0, r0
@Y    vsqrtpd r3, r2
```

CIT
Dense @X, PC @X
Compacted #0 -> Mask **1**000
Compacted #1 -> Mask 000**1**
Dense @Y, PC @Y
Compacted #0 -> Mask **1**000
Compacted #1 -> Mask 000**1**

CIT
Dense @X, PC @X
Compacted #0 -> Mask **1**000
Compacted #1 -> Mask 0**1**00
Dense @Y, PC @Y
Compacted #0 -> Mask **1**000
Compacted #1 -> Mask 000**1**

Figure 4.7: Example of dense register forwarding. In the left, compaction is not needed, but it is on the right, as the mask for the second compactable instruction differs.

not have the same active element position as compared to the one from @Y. In this case, compaction for the first compactable instruction in @Y is not needed, but it must be done for the second one. For these instructions, there is dense register forwarding of 50%.

4.3.10 CR Case Study

To illustrate how the CR mechanism works, we refer to the code from Figure 4.8. It is used to describe the different phases in CR: activation, compaction, execution, and restoration. For the sake of simplicity, in this particular example, we assume a 128-bit vector length architecture. Thus, each vector register may hold 2 double precision elements. In this case, a vector multiplication (*vmulpd*, line 8), a subtraction (*vsubpd*, line 9), and a square root (*vsqrtpd*, line 7) represent the 3 predicated instructions in this loop. They are guarded by a mask register *k1* created in line 6. This mask is built by comparing each element in array *C* to a zero-filled vector. In this case, we assume that the compiler marks this loop as suitable for

```
1  for (i←0; i≤N_ELEMENT; i+=VL)
2      vmovapd   r2, &B[i]
3      vaddpd  r1, r2, <imm>
4      vmovapd r3, &C[i]
5      vmovapd r4, &D[i]
6      vcmppd  k1, r3, <zero>, <NE>
7      vsqrtpd r5 {k1}, r4
8      vmulpd  r5 {k1}, r5, r3
9      vsubpd  r1 {k1}, r1, r5
10     vmovapd &A[i], r1
```

Figure 4.8: SIMD loop in Intel's assembly.

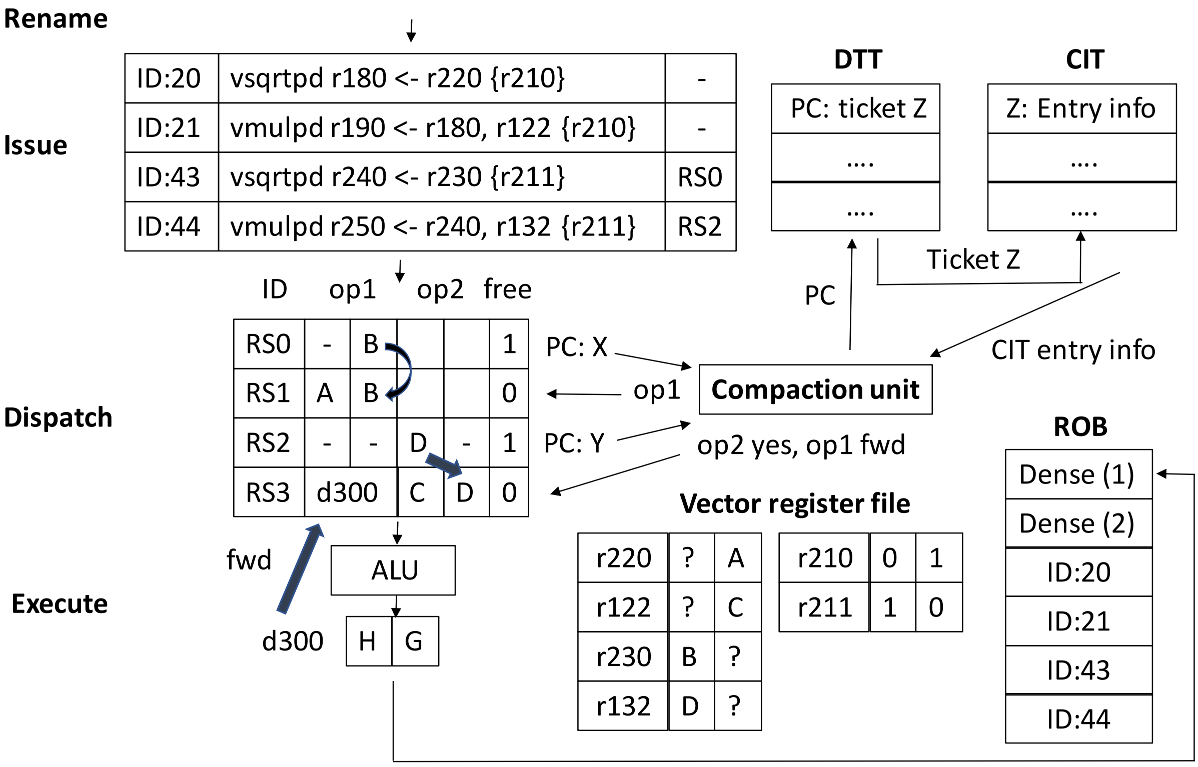
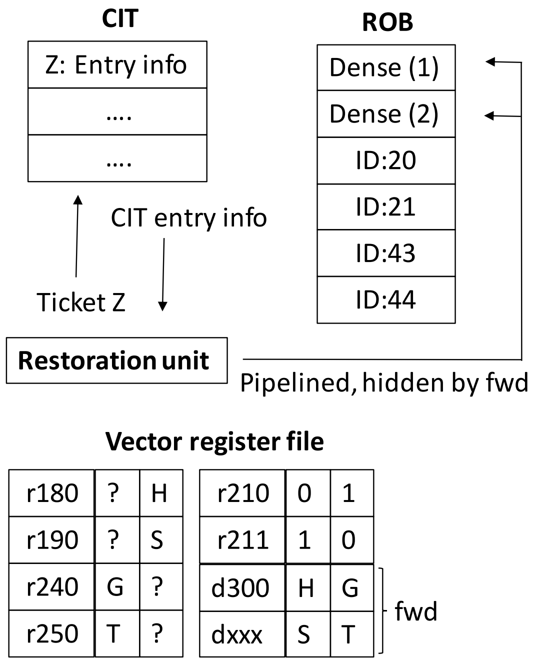


Figure 4.9: Example of the Compaction phase.

CR. Figures 4.9 and 4.10 show the compaction and restoration processes for the instructions *vsqrtpd* and *vmulpd*.

Activation Phase. At the Issue stage, there are two instances of these instructions (with identifiers 20, 21, 43 and 44). Mask registers *r210* and *r211* are read as they become ready. Since their mask density is low (50%), CR is enabled for this loop. Then, two dense instructions for these PCs are created and the CIT allocation is performed, allocating two CIT entries with *Capacity* 2. The *Alloc Occupancy*, *Mask*, *Dest Reg Idx* and *Allocate* fields are updated for every compactable, since the mask registers for every dynamic instrucion are ready and the Rename stage has been previously accessed. A ROB and an RS entry are allocated for each dense instruction. Two tickets are created and stored in DTT.

Compaction Phase. As operands become ready, the instructions are moved to the Dispatch stage. The CIT insertion is performed, updating the corresponding *Insert Occupancy*, *Insert$_c$* and *Last Insertion* CIT fields. After that, the compaction for the dense *vsqrtpd* instruction starts. This process is shown in Figure 4.9. In this case, the active element in register *r220* (A) is moved to the dense RS entry (RS1) using the CIT information. After that, the RS belonging to ID:20 is released. Similarly, in the next loop iteration, CR compacts the active element from register *r230* (B) into RS1. This dense instruction is ready for execution. The same process



r180	?	H
r190	?	S
r240	G	?
r250	T	?

r210	0	1
r211	1	0
d300	H	G
dxxx	S	T

Figure 4.10: Example of Restoration phase.

is done with instruction *vmulpd*, where the second operand is compacted moving the active lane in *r122* (C) and *r132* (D) to the dense instruction in RS3. However, the first operand in the compactable instruction is dependent of *vsqrtpd*, an already compacted one. The CIT notices this situation and skips its compaction, notifying that a dense register forwarding is going to happen. In particular, the register *d300*.

Execution Phase. The dense *vsqrtpd* instruction is executed as compaction is finished and its destination register *d300* is forwarded to the dense *vmulpd*, which will also be executed afterwards.

Restoration Phase. After execution, the restoration phase occurs for the dense instructions *vsqrtpd* and *vmulpd*. A brief overview is depicted in Figure 4.10. The CIT contains the information regarding every inserted compactable instruction for every dense. In *vsqrtpd*, the restoration unit reads the dense output *d300* and the original mask values from the instructions with ID 20 and 43, inserted in the CIT. Then, the restoration unit moves the *d300* elements to the destination entries in the ROB, performing an offset calculation depending on the mask values and the compacted instruction insertion order. For example, the register *r180* (instruction ID:20) receives the first element from the dense register *d300* (H) and it is placed in the second lane, where the mask register *r210* contains a true element. The register *r240* gets the second element (G), as the accumulated capacity is one, and it is placed in the first lane, specified by mask *r211*. The same process is done with the dense *vmulpd*, moving S and T to the second and first lanes of registers *r190* and *r250* respectively.

4.3.11 Optimizing SIMD Legacy Code

CR hardware can also be employed to optimize legacy SIMD codes on modern and wider processors. Many applications make use of hand-coded programs with SIMD intrinsics. Porting such codes to modern SIMD architectures is costly and time consuming. For this reason, many 256-bit or 128-bit SIMD codes are executed on 512-bit VFUs, underutilizing hardware capabilities. The CR mechanism can be employed to dynamically create dense instructions that compact two AVX-2 instructions into a single AVX-512 instruction.

This way, every SIMD instruction is a candidate for compaction as the CR mechanism is not restricted to predicated instructions. In this case, the mask density and the active element positions are known before-hand, as they are defined by the architecture (e.g., 50% if compacting two AVX-2 instructions into a single AVX-512 instruction). In such scenario, the compaction/restoration units complexity is reduced, enabling lower CR latencies than in the general CR case, and enhancing the CR mechanism efficiency.

This approach is transparent to the programmer and only requires a compiler to analyze the static factors described in Section 4.3.4 to determine if CR could improve performance. Section 4.6.2 evaluates the AVX-2 instruction compaction over AVX-512 using CR.

4.3.12 Discussion

The CIT is squashed in the event of a branch miss-prediction. Two scenarios must be considered: a) miss-predicted instructions created an entry within a dense instruction, but operands were not ready and thus, not compacted; and b) operands were ready and compacted. In the first case, the CIT would be waiting forever for this instruction. In the second case, a false version of the dense register would be created, since some lanes belong to miss-predicted instructions operands. The first scenario is handled by making the CIT aware of miss-predictions. The second scenario is not critical because results are written into miss-predicted ROB entries in the Restoration phase, but these results never commit.

Page faults need a special handling as they are attended at commit but a dense instruction may be blocking its attendance. A timeout is required to force the dense execution and of every instruction prior to it.

Precise exceptions are also feasible with CR. If an exception occurs while a dense instruction is executing, such as arithmetic overflow, the exception is *restored* to the corresponding compacted instruction to be handled.

A challenge to be faced in the future is the implementation of dense horizontal instructions. Horizontal instructions, such as *shuffles*, move a value from a particular vector register lane to another one. At the moment a dense register is created, the original element positions are lost so the operation cannot be done.

4.4 Design Space Exploration

Next, a design space exploration is done to size the CR hardware and to study the application impact on CR. In this case, we make use of a micro-benchmark hand-coded using Intel's AVX-512 intrinsics. It is parameterized so that the mask density, the percentage of costly instructions and the number of instructions in each loop iteration can be changed.

4.4.1 Compaction and Restoration Latencies

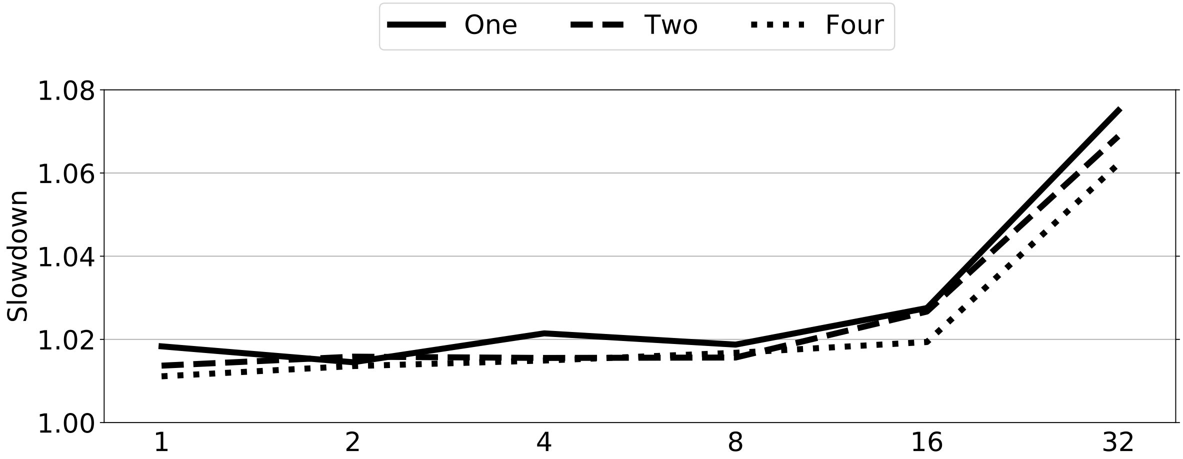


Figure 4.11: Compaction unit configuration slowdown on performance. Normalized to non-latency CR scenario. In the *x*-axis the different number of stages. Each line represents a different compaction unit count.

As explained in Section 4.3, CR requires four hardware components. The DTT and CIT sizes are defined by the ROB size. The compaction and restoration units are sized in this section. First, we start with the compaction unit design. Figure 4.11 shows the performance slowdown obtained when varying its number and operation latency. Performance is normalized to an ideal design with no CR latencies.

Figure 4.11 depicts the average results for the SIMD micro-benchmark executed with several mask densities. Increasing the number of compaction units from 1 to 4 provides less than 1.3% performance improvements. In contrast, when having more than 8 compaction stages, performance degrades. Thus, we select a compaction configuration with a single unit and two pipeline stages. It provides only a 1.4% performance degradation with respect to an ideal CR mechanism and a simple design.

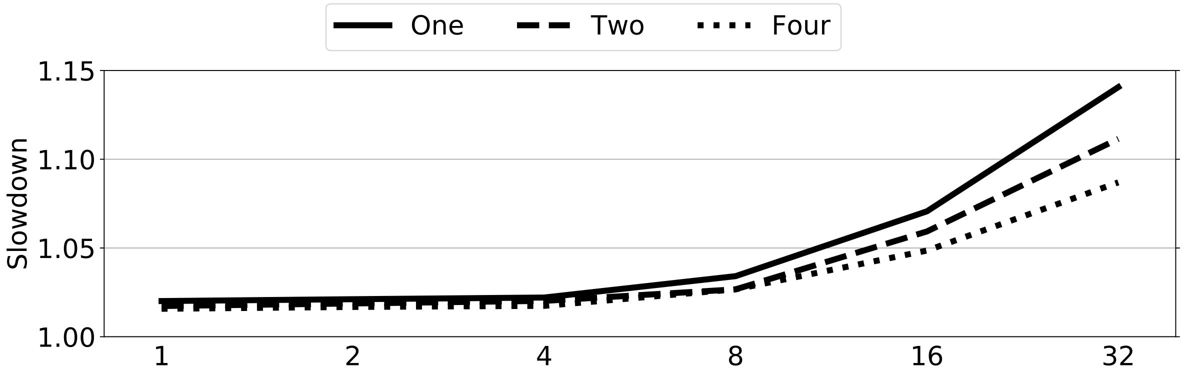


Figure 4.12: Restoration unit configuration slowdown on performance. One two-stage compaction unit latency considered. Normalized to non-latency CR scenario. In the *x*-axis the different number of stages. Each line represents a different restoration unit count.

Next, we explore the restoration unit design. Figure 4.12 shows the performance degradation with different restoration formats. For this experiment, use the selected compaction unit configuration. Results are normalized to an ideal design with no CR latencies. In this case, with 1, 2 and 4 stages, varying the number of units and restoration stages marginally degrades performance (less than 0.5% and 0.2% slowdowns, respectively). However, as 8-stage restoration units are reached, performance degrades drastically. A 14% performance degradation is achieved with 32-stage units, where increasing the unit number from 1 to 4 provides a benefit of up to 6%. As we are interested in reducing energy consumption, the final design is limited to a single two-stage unit. Thus, restoring a dense instruction with four compacted ones takes five cycles. This format combined with the selected compaction unit, has a 1.9% slowdown compared to an ideal scenario.

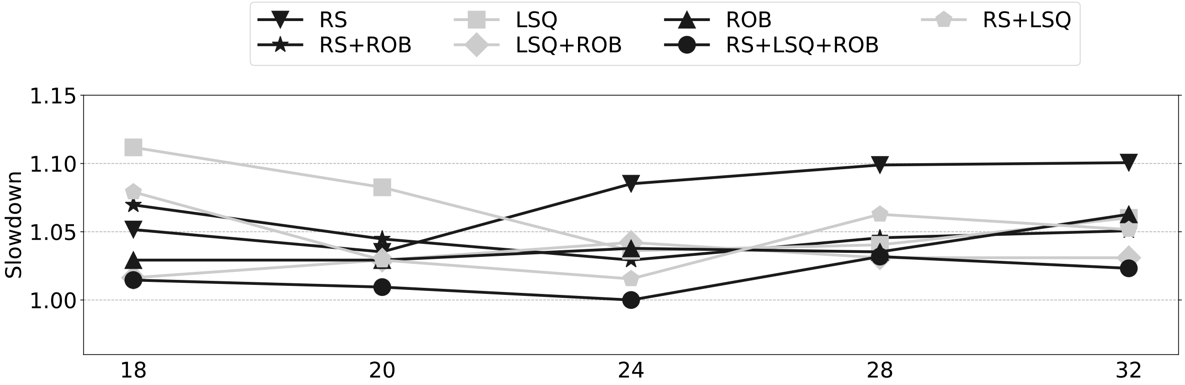


Figure 4.13: Timeout policy combinations impact on performance, normalized to the best scenario. The circular dependency policy is implicit in every scenario. In the *x*-axis the number of cycles for each timeout policy changes.

4.4.2 Timeout Policies

Next, we measure the impact of the timeout policies discussed in Section 4.3.7. In this case, the micro-benchmark is used with different timeout policies. Figure 4.13 depicts the performance degradation obtained by combining the original timeout policies, normalized to the best configuration. The timeout policies consider the occupancy in different resources (RS, LSQ and ROB) and different timeouts (from 18 to 32 cycles). All policies take into account circular dependencies as this is required for the correct execution of the benchmarks.

Selecting the optimal timeout policy is fundamental for CR, preventing the CPU from waiting too much for dense register population. Results show up to a 10% slowdown when only the issue queue is considered. The best outcome is obtained when considering all resources.

4.4.3 Costly SIMD Instruction Ratio

Next, we analyze the influence of the predicated instructions latencies to performance and energy. In this case, we consider the same base micro-benchmark where the ratio between low and high latency instructions increases, from 0% to 100%. All of them have the same memory access pattern and the same number of instructions per iteration. The mask density varies between 25-50%. Results are normalized to the 0% long latency instruction scenario.

Figure 4.14 shows performance and energy results. The higher the costly instruction ratio, the better the speedup and energy reduction. If instructions have a long latency, the dense register population and the CR latencies can be hidden by the execution and even lead to a performance benefit. For instance, in the case of a predicated square root with a 25% mask density scenario, the fact of delaying the execution of instructions from four iterations (50 cycles) and executing only one instruction (20 cycles) would be better than executing four instances of the same dynamic instruction (70 vs 80 cycles). Long latency SIMD instructions also permit higher timeout policy values, allowing more occupied dense registers, reducing the accesses to VFUs, and thus, generating higher dynamic energy benefits.

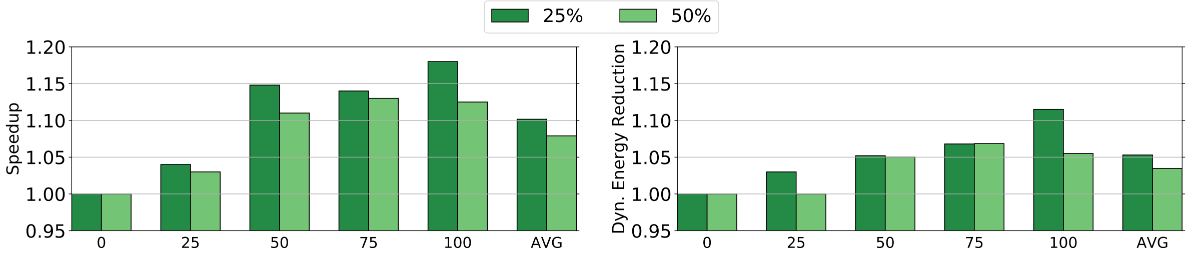


Figure 4.14: Impact of costly predicated SIMD instructions to performance (left) and dynamic energy (right). Normalized to the no-long latency instruction scenario. In the *x*-axis, the percentage of costly predicated instructions.

4.4.4 Effectiveness with Different Loop Lengths

Finally, we study the sensitivity of the CR mechanism to the loop instruction length. In this case, we consider the same SIMD micro-benchmark as in the previous sections. We use the same mask densities (25%, 50%) and different number of instructions per iteration in a processor with a 320-entry ROB.

Figure 4.15 shows the average number of predicated instructions compacted per dense instruction. With both mask densities, CR achieves a high number of compacted instructions with loops of 40 or less instructions. An increase in the number of instructions per loop iteration causes a higher ROB occupancy, preventing CR from doing an efficient population of dense registers. For example, moving from 20 to 60 instructions per iteration reduces the average compaction from 4 to 1.5 in a 25% mask density scenario.

Also, a higher mask density leads to more pressure on the ROB occupancy as a dense instruction is added more frequently (every 2 compacted instructions with 50% mask density; every 4 instructions with 25%). Consequently, loops with 160 instructions and 50% mask density can not be compacted with CR. In contrast, loops with 80 instructions and 25% mask density can be partially compacted with CR (1.45 instructions are compacted per dense).

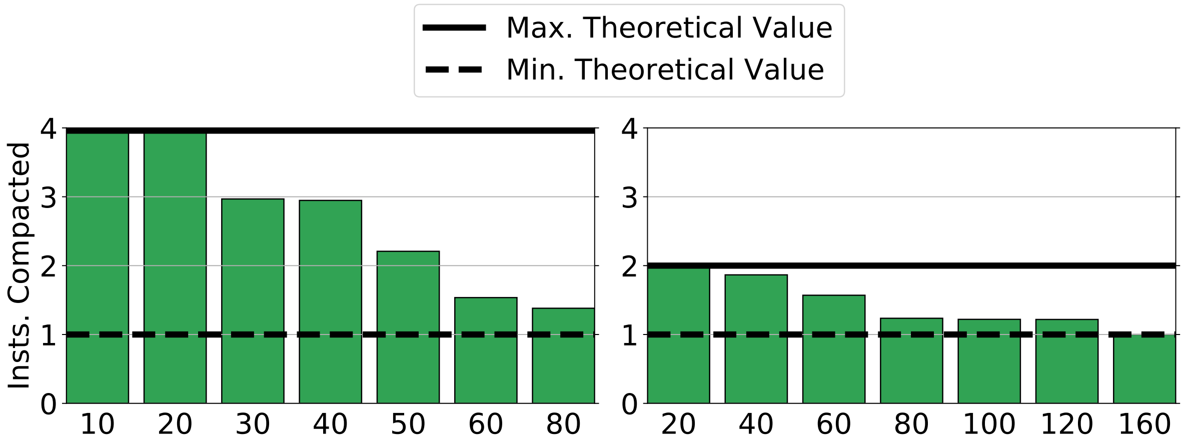


Figure 4.15: Average number of predicated instructions compacted per dense in CR. In the x-axis, the number of instructions per loop iteration. Masks: 25% (left) and 50% (right).

4.5 Area and Power Consumption of CR Units

The power consumption of the CR units is evaluated with McPAT [121] using a process technology of 22nm, a voltage of 0.6V and the default clock gating scheme. We incorporate the changes suggested by Xi Vaidya *et al.* [198] to improve the accuracy of the models. The CIT structure is modeled in CACTI 6.5 [135], adding the appropriate counters in gem5 to measure the extra power introduced by it.

CR units have been modeled in RTL [130, 37] with the configurations chosen in Section 4.4. Results for a 22nm technology show area requirements of 5000μm^2. It is almost three orders of magnitude smaller than a 512-bit ALU modeled in McPAT (4.45 mm^2). In terms of power, every unit consumes 11.25mW of peak power (combined leakage plus dynamic), almost two orders of magnitude smaller than the power of the 512-bit ALU computed by McPAT (0.92W).

This estimation is used in the next section to quantify the energy cost of the CR structures and to compare it to the baseline, while executing different benchmarks.

4.6 Evaluation

This section explains the performance and energy benefits of CR in real applications. We also describe the benefits of using CR to optimize legacy SIMD code.

4.6.1 Predicated SIMD Applications

The CR proposal is evaluated with ten different applications. As described in Section 3.1.1, we employ two processor configurations with different instruction latencies (ICE and KNL).

Fung and Vaidya *et al.* studied the mask densities in several applications [75, 183]. They showed they are usually input-dependent and range between $15-60\%$. Since input selection may strongly impact mask density and complicate the evaluation analysis, we consider values from 25% to 50% for all the codes. These values capture almost entirely the mask density range from the representative applications.

For each application, we perform an exploration with several timeout values and we select the configurations which provide the best performance outcomes. For the CR mechanism, we make use of the configuration determined in Section 4.4.

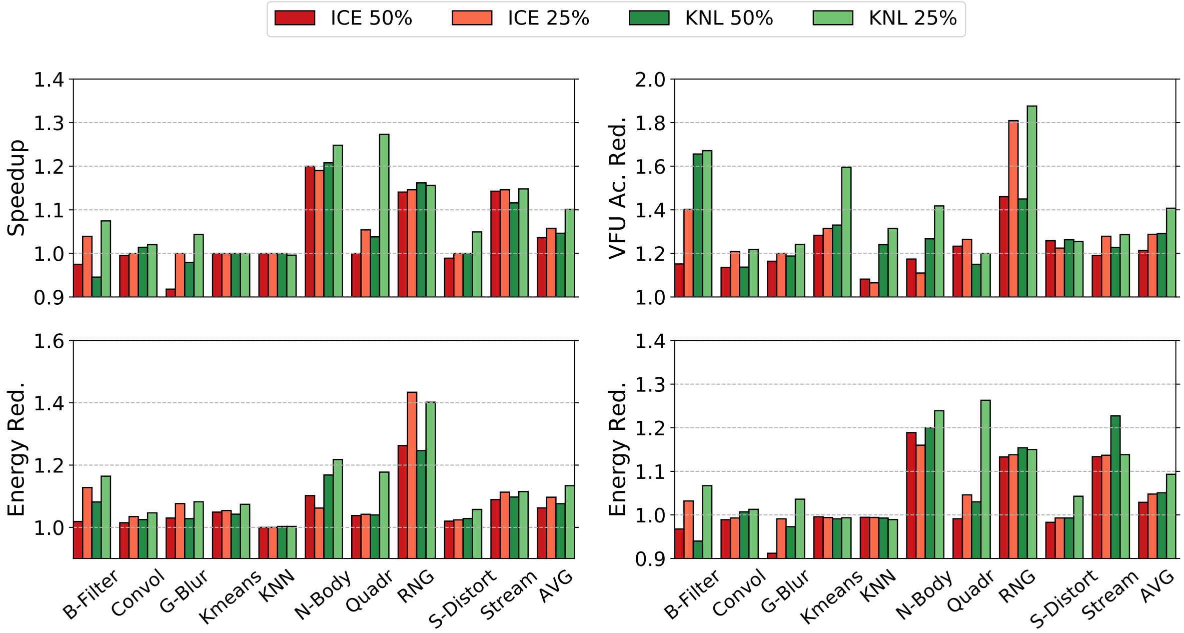


Figure 4.16: Performance (up left), VFU access (up right), dynamic energy (bottom left) reductions and leakage energy (bottom right) results of CR. Normalized to a non-CR scenario.

Figure 4.16 depicts the results in terms of speed-up, VFU access reduction and dynamic and leakage energy reductions. Energy reductions correspond to energy savings in the whole system. Results are normalized to a regular no-CR execution. On average, applications achieve between 3.6% and 10% speed-ups, between 21% and 41% VFU access reductions, and between 6.2% and 13.4% dynamic energy reductions. In all the experiments, the KNL configuration provides more opportunities to the CR mechanism as there is more contention in the VFU. Also, lower mask densities (i.e., 25%) lead to more compaction opportunities.

Significant speed-ups are obtained for some of the evaluated benchmarks. This is the case of N-Body and RNG, which contain a high percentage of long latency SIMD instructions per loop iteration (as shown in Figure 3.1). They achieve performance improvements up to 25% and 15%, respectively, and dynamic energy reductions up to 22% and 43%. This reduction in dynamic energy is a result of the significant reduction in VFU accesses (up to 42% and 87%, respectively). In the case of N-Body, the CR phases are hidden by the memory access requests and lead to better performance benefits.

B-Filter and S-Distort also contain long latency SIMD instructions. However, a higher number of instructions per loop iteration prevents an efficient population of dense registers. Only with a VFU contention increase in the KNL configuration, speed-ups reach a 7%.

The application memory access pattern is important for CR, since it can hide the dense register compaction/restoration. Convol, with an irregular access pattern and low-latency predicated instructions, is able of marginally improving performance and reducing dynamic energy consumption up to 5%. In contrast, Kmeans and KNN have a contiguous memory access pattern and no long latency predicated instructions. Kmeans is capable of reducing VFU accesses up to a 60%. However, the large amount of instructions and the low percentage of predicated instructions in KNN prevent CR from achieving performance benefits. KNN also contains horizontal operations, blocking dense register forwarding.

For all the applications, the long latencies of the KNL configuration enable higher VFU access reductions that lead to better dynamic energy results. In this configuration, there is a higher contention in the VFU than in the ICE one. As a result, a higher occupancy of dense registers is achieved. We have measured the *dense register forwarding*, in particular, at the lane level. If a dense register lane can be forwarded, the compaction phase can be avoided for that lane, reducing latency and energy consumption. For instance, 72% of dense lanes can be forwarded in BF, 65% in S-Distort, 73% in Kmeans, 46% in KNN, 77% in RNG and 46% in G-Blur.

4.6.2 Optimizing AVX-2 Legacy Code

The CR mechanism can also be used to optimize SIMD legacy code. Section 4.3.11 describes the motivation and the advantages of this approach. In this section, we explain the results of employing CR to compact two AVX-2 instructions into one AVX-512 instruction.

Figure 4.17 shows the results of CR with real applications compiled with AVX-2 support. Results are normalized to a regular execution without CR. In this case, we limit the original set of evaluated applications to seven, since three of them do not have a memory access behavior or the required percentage of SIMD instructions suitable for CR. A compiler may identify these static application characteristics and notify CR when to compact AVX-2 codes into wider SIMD extensions.

As expected, average results are better than in the scenario with predicated SIMD instructions. In the KNL configuration, speed-up and leakage energy reduction reach 17% on average, while dynamic energy reaches 16% reductions. In the ICE configuration, average results are more modest (5% and 12%). Both configurations achieve an average 35% reduction in VFU accesses.

The largest reductions in VFU accesses are achieved with B-Filter and RNG (between 60% and 73%). This translates into significant reductions in dynamic energy. RNG achieves a significant 56% performance improvement. N-body also reduces dynamic energy (between 10% and 12%). In contrast, KNN still suffers from the blocking of dense register forwarding due to horizontal operations and achieves minimal energy savings, even if VFU accesses are reduced by more than 10%.

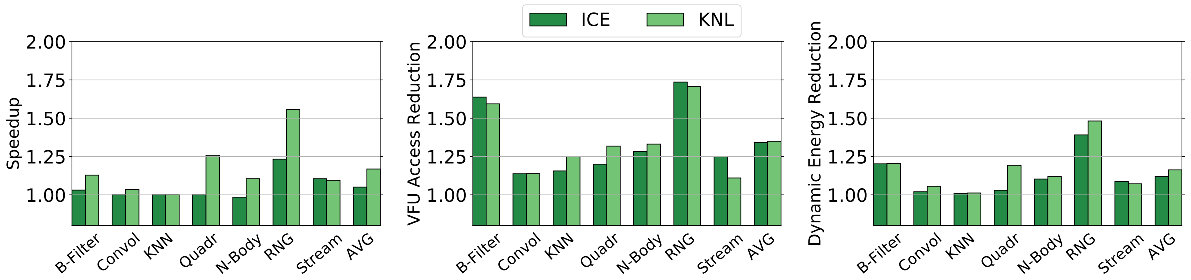


Figure 4.17: Results of AVX-2 legacy codes compacted into AVX-512 using CR. Normalized to a non-CR scenario. Speed-Up left, VFU access reduction (center) and dynamic energy reduction (right).

4.6.3 Comparison with Other Proposals

This section compares CR with Disable Inactive Lanes (DIL) [113], an alternative hardware proposal to reduce power consumption in the VFU. DIL reads the mask operands before executing predicated instructions and disables the lanes in the VFU with inactive elements. This solution reduces power consumption at the cost of increasing the complexity of the VFU design. However, DIL does not reduce the contention in the VFU and cannot be used to speed-up the execution of AVX-2 legacy codes. Interestingly, CR and DIL can be combined to further reduce the power consumption of CR when a timeout avoids instructions compaction.

The left chart of Figure 4.18 presents the average speed-up of CR, DIL and CR+DIL over a baseline without CR. As expected, DIL and CR+DIL do not improve performance over the baseline and CR, respectively. The right chart of Figure 4.18 presents the average energy reduction of the three techniques over a baseline without CR. DIL reduces energy between 5% and 8% as it reduces the dynamic power in the VFU. CR achieves higher energy reductions than DIL due to the increased performance in some of the benchmarks. However, in benchmarks in which CR provides no performance benefits (Convol, Kmeans, KNN), DIL achieves up to 18% energy reduction. Thus, CR+DIL provides the best energy results with average energy reductions between 6% and 13%.

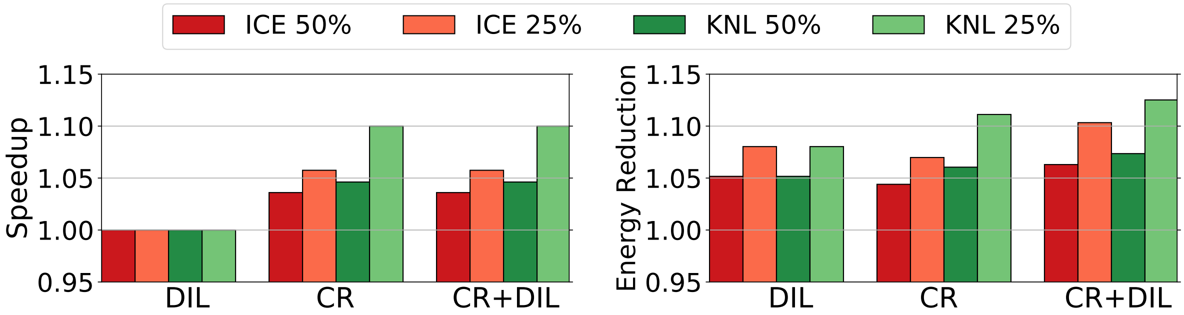


Figure 4.18: Speed-Up (left) and total energy reduction (right) of DIL, CR and CR+DIL normalized over a non-CR scenario.

4.7 Conclusions

Exploiting DLP in current processors with SIMD extensions is critical to improve performance and energy efficiency. When vectorizing applications, divergence control using predication is one of the most challenging obstacles to overcome.

Current SIMD extensions execute all elements in a predicated instruction independently of the values in the mask operand, wasting significant fractions of energy and performance.

In this approach, we propose the Compaction/Restoration (CR) hardware design, which is capable of achieving density-time performance and energy efficiency with predicated SIMD instructions. CR creates a dense instruction with several dynamic predicated instructions for a certain PC. The active elements of these regular SIMD instructions are *compacted* into a dense instruction. Then, dense instructions are executed and their results are *restored* to the original instructions. This is achieved without programmer intervention.

Our evaluation shows that CR improves performance by up to 25% and reduces dynamic energy consumption by up to 43% on real unmodified predicated applications.

Moreover, CR allows executing unmodified legacy code with short SIMD instructions (AVX-2) on newer architectures with wider vectors (AVX-512), achieving up to 56% performance benefits.

Chapter 5

PLANAR: A Programmable Accelerator for Near-Memory Data Rearrangement

5.1 Introduction

Memory latencies have not experienced the near-exponential improvements seen in processing speed and memory capacity [73, 69]. As a result, data access times increasingly limit system performance, a phenomenon known as the Memory Wall [197]. Deep cache hierarchies are the natural solution to this trend, providing low-latency data access to high-performance out-of-order processing units. Applications that have locality of reference benefit from cache hierarchies [171, 182], while prefetchers act in the background to hide memory access latency [132].

In the presence of sparsity and irregular reuse distances, studies show that data prefetching is not effective [207], utilization of transmitted bandwidth can be as low as 20% [29], and that most blocks in the last level cache are not reused before eviction [172, 28]. In addition, for applications with dependent or indirect access loads, every cache level increases the overall round-trip access latency [64]. Finally, irregular and sparse patterns preclude harnessing data-level parallelism via vector instructions that operate on multiple data values (SIMD) [125, 173, 190]. Data movement not only affects performance: approximately two-thirds of the energy required to compute is consumed by data movement, specifically by the memory and interconnect [33].

Data Layout Transformation (DLT) mechanisms have been proposed to tackle these problems. DLT aims to rearrange sparse data into a dense representation to improve locality and make better use of the memory hierarchy. Table 5.1 qualitatively compares multiple state-of-the-art proposals. A balanced design should fulfill three principles. First, a comprehensive design that scales well with multi-core systems by carefully choosing where rearrangements occur.

Table 5.1: Comparison with state-of-the-art DLT proposals.

Features	**Impulse [206]**	**DLT Acc. [89]**	**SPiDRE [29]**	**DRE [123]**	**PLANAR**
Full design	✓	✓	✗	✓	✓
Scalable design	✗	✓	✓	✗	✓
Non-blocking DLT	✓	✗	✓	✗	✓
Fine-grain sync.	✓	✗	✗	✗	✓
VM support	✓	✓	✓	✗	✓
Normal allocator	✗	✓	✓	✗	✓

Second, maximize system performance by providing non-blocking fine-grain rearrangements to hide DLT latency. Third, ease programmability for the DLT engine and target applications by providing virtual memory (VM) support and conventional memory allocation mechanisms. Previous proposals make compromises on these design principles hindering their adoption.

In this chapter we present a *ProgrammabLe Accelerator for Near-memory datA Rearrangement (*PLANAR*)*. PLANAR is located within the system-on-chip at the same level as the memory controllers, avoiding custom off-chip memory modifications that are difficult to adopt. Our design is non-blocking as it decouples access and execute, allowing overlap of data rearrangements and *host* core computation. In addition, we provide mechanisms for fine-grain synchronization between PLANAR and *host* cores to allow dense data to be consumed as it is rearranged, hiding rearrangement latency. PLANAR is programmable via simple library calls that can be inserted by a programmer or by a compiler pass. This simple programming interface is possible due to the fact that PLANAR has virtual memory support and employs well-known existing memory management mechanisms for the new dense data structures.

Moreover, PLANAR enables applications to take better advantage of the memory hierarchy by exploiting locality of dense data, and unlocks additional performance due to better prefetching and vectorization. On the latter, PLANAR allows compilers to optimize instruction emission for contiguous memory [194], which is critical to vector performance [140].

This chapter makes the following contributions:

- We introduce minimal functional changes to incorporate PLANAR into a system-on-chip with out-of-order cores. By locating PLANAR devices at the memory controller level we enable the design to (i) scale with multi-core systems, (ii) perform fine-grain non-blocking data rearrangements, (iii) operate with virtual memory support, and (iv) be off-chip memory technology agnostic. No solution in the state-of-the-art provides all such properties at once.

```
void stride_kernel(double *x, int *idx, ...){
    ....
    for (len = 0; i  < len; len++) {
        v1s1m3(); v1s2m3(); v1s3m3(); v2s2m3(); v2s2m4();
        v1s1i3(x, idx);
    }
}
void v1s1i3(double *x, int *idx, ... ){
    ....
    for( j = 0; j < irep; j++ ) {
        t1 = 1.0/(double)(j+1);
        for( i = 0; i < n; i++ )
            y[...] += t1*x[idx[i]];    //irregular accesses
    }
}
```

Figure 5.1: Original STRIDE code.

- A detailed evaluation using a full-system cycle-accurate simulator shows that a multi-core system with PLANAR achieves an average speed-up of 4.58× across a wide range of applications featuring sparse and irregular access patterns. This performance improvement is due to PLANAR reducing L1-D cache misses by an average of 89% and L1-D cache miss latency by an average of 53%. Overall, dynamic energy consumption is reduced by more than 40% in all benchmarks. PLANAR also enables additional vectorization of rearranged codes, increasing the average speed-up to 5.71×.

- We show that PLANAR outperforms software DLT techniques in Section 5.2 and two state-of-the-art hardware proposals, Impulse [206] and a DLT accelerator [89], in Section 5.5. Our quantitative comparison shows that, on average, PLANAR outperforms Impulse by 2.12× and the DLT accelerator by 2.23×. Thanks to non-blocking fine-grain rearrangements, PLANAR can hide DLT latency, allowing the *host* to consume dense data as it is rearranged.

5.2 Motivation

To explain the limitations of DLT techniques in software, and the advantages of performing DLT with PLANAR, we have chosen a representative case study based on the STRIDE benchmark [1]. STRIDE is a memory intensive benchmark that consists of a loop where every iteration executes six different kernels. In the original code, *v1s1i3* is the kernel with sparse memory accesses

[1] Section 3.2 describes the benchmark in detail.

```
1  void stride_kernel(double *x, int *idx, ...){
2      ....
3      for (len = 0; i  < len; len++) {
4          v1s1m3(); v1s2m3(); v1s3m3(); v2s2m3(); v2s2m4();
5          v1s1i3_sw_rearr(x, idx);
6      }
7  }
8  void v1s1i3_sw_rearr(double *x, int *idx, ... ){
9      ....
10     x_rearr = malloc(size);
11     for( i = 0; i < n; i++ )
12         x_rearr[i] = x[idx[i]];    //software rearrangement
13
14     for( j = 0; j < irep; j++ ) {
15         t1 = 1.0/(double)(j+1);
16         for( i = 0; i < n; i++ )
17             y[...] += t1*x_rearr[i];    //regular accesses
18     }
19     free(x_rearr);
20 }
```

Figure 5.2: Software-rearranged STRIDE code.

(see lines 8-15 in Figure 5.1). The memory access pattern is governed by the *idx* array which is populated with a configurable input stride[2].

The programmer could decide to replace the indirect memory accesses from *x* with sequential ones in an *x_rearr* array using a software DLT solution, as shown in Figure 5.2. This extra code should be placed just before the original loop in *v1s1i3* (see lines 10-12 in Figure 5.2). This software rearrangement is beneficial as *x* is accessed *irep* times in the baseline with strided accesses, and only once in this new version. As a result, execution time improves 22.1% on average for different stride values in the indirection vector *idx*.

In this chapter we present PLANAR, a hardware solution that performs near-memory data layout transformations. Figure 5.3 shows the pseudo-code of STRIDE compatible with PLANAR. The rearrange function (*offload* function in lines 1-7) performs the data layout transformation using the PLANAR devices. Several PLANAR devices can be allocated to do this transformation in parallel (line 12) and execute the rearrange function (line 13), extracting higher memory-level parallelism (MLP) than in the software rearrangement version. Finally, the PLANAR devices are released (line 6).

[2] Section 3.2 describes the strides employed in the evaluation.

```
1  void offload(double *x, int *idx, double *x_rearr, ...){
2      // Rearrange function executed on PLANAR
3      for( i = start_idx; i < end_idx; i++ )
4          x_rearr[i] = x[idx[i]];
5      // Release device if last element
6      planar_release();
7  }
8  void stride_kernel(double *x, int *idx, ...){
9      ....
10     for (len = 0; i  < len; len++) {
11         x_rearr = malloc(size);
12         n_dev = planar_alloc(min, max);
13         offload«<n_dev»>(x, idx, x_rearr, size, ...);
14         v1s1m3(); v1s2m3(); v1s3m3(); v2s2m3(); v2s2m4();
15         v1s1i3_hw_rearr(x_rearr);
16         free(x_rearr);
17     }
18 }
19 void v1s1i3_hw_rearr(double *x_rearr, ... ){
20     ....
21     for( j = 0; j < irep; j++ ) {
22         t1 = 1.0/(double)(j+1);
23         for( i = 0; i < n; i++ )
24             y[...] += t1*x_rearr[i];    //regular accesses
25     }
26 }
```

Figure 5.3: PLANAR-rearranged STRIDE code.

This rearrangement can be done *ahead of time* while the *host* is operating on the first five kernels, thereby overlapping data rearrangements and *host* execution (see lines 14-15 in Figure 5.3). As a result, PLANAR effectively hides rearrangement latency Executing STRIDE with eight PLANAR devices provides average performance speedups of 2.77× and 3.39× over software-rearranged and the original versions, respectively.

PLANAR provides the required hardware support to enable fast data rearrangement near memory, converting sparse data to dense, resulting in a more efficient usage of the available bandwidth. This transformation is done while the *host* core performs useful computation, effectively decoupling access to memory and execution.

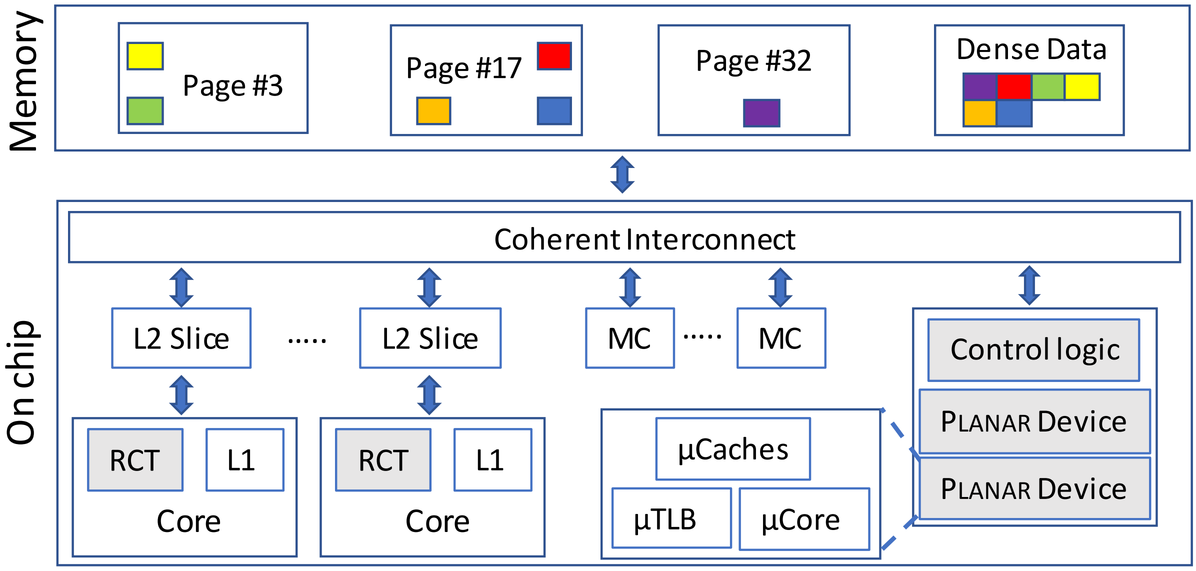


Figure 5.4: System overview with two PLANAR devices. Cores are augmented with a Rearrangement Control Table (*RCT*) to monitor ongoing rearrangements.

5.3 PLANAR Design

PLANAR targets applications with irregular memory access patterns, often due to the utilization of sparse data structures. In such applications, the memory subsystem is poorly utilized, leading to latency and bandwidth bottlenecks because of low cache block utilization [28] caused by disperse memory accesses that lead to high (but underutilized) traffic on data transfer networks (e.g., coherence bus, interconnects) [129].

Figure 5.4 shows a high level system overview with two PLANAR devices. PLANAR is implemented as a near-memory programmable accelerator connected to the main coherence bus with direct access to the memory controllers. Despite being programmable, it is a simple device that can be implemented as a microcontroller, as we do in this work. It is comparable to an Arm Cortex $M0+$, with the addition of a 64-bit ALU and minimal support for data caching and address translation.

The design enables accesses from the cores to bypass the PLANAR units when in normal operation, while allowing the PLANAR units to use the same memory controllers when commanded by the *host* core to reorganize data. In the figure, every core is augmented with a small *Rearrangement Control Table (RCT)* to monitor the status of ongoing transformations. The *RCT* has one entry per rearrangement in flight, within each *RCT* entry there is a slot for each

Table 5.2: Rearrangement Control Table (*RCT*) with the information about a rearrangement in flight being performed by 4 PLANAR devices.

PLANAR #ID	**First elem**	**Last elem**	**Max elem**
0	0	99	32
1	100	199	140
2	200	299	261
3	300	399	310

PLANAR device, and per-device sub-entries containing progress information for each ongoing rearrangement (three 64-bit entries to track virtual address range and status).

Table 5.2 depicts the *RCT* filled in with details of a data rearrangement being performed by four devices. The *first* and *last* elements represent the ranges in the final rearranged structure that correspond to that particular accelerator. The *max* element is the latest element that has been rearranged. As an example, if a hardware unit requests a memory access to element 120, the entry with ID 1 will be checked, because the element is contained in the range (100 – 199) and since the latest rearranged element is 140, the memory request will proceed. While we show the *RCT* as represented by a table of indices, it could conceptually be composed of address ranges or another logical identifier capable of specifying the location of a range of data.

PLANAR creates a new data structure whose elements are sorted the way they are to be accessed by the *host* core. This way, cache block and bandwidth utilization improves. In the best case, the process latency can be hidden if there is sufficient time between the rearrangement and the data access by the *host* core, allowing to overlap the rearrangement with computation. Multiple devices can be used to apply the same rearrange function, or multiple rearrange functions can be done in parallel by different devices. PLANAR works as an accelerator on behalf of a requesting core that sends commands to the PLANAR device. If there is computation to be done in the meantime, and suspends or computes until synchronization messages are received.

Figure 5.4 shows an example with two PLANAR devices. A *host* core has requested them to perform the near-memory data restructuring of the *sparse* elements in color from data pages 3, 17 and 32. The result is a dense version of the data placed into another data page. The *host* core may access this new dense structure via contiguous accesses instead of the original sparse accesses, reducing data movement and hiding latency. As an example, if the core only uses one 8B value from each of the cache blocks accessed (64B) the total payload needed would be 48B (six elements). In the original case, the core would have to access six different cache lines (i.e., 384B). However, with PLANAR the reduced payload would be a single cache line (assuming they are aligned), i.e., 64B with 48B of the transfer actually utilized. This represents an 83% reduction in data movement for this simple example case. Moreover, dense data presents additional opportunities to improve performance: (i) simple next-line prefetching schemes are efficient, and (ii) data level parallelism via vectorization is also easy to achieve.

The following sections provide the operational details of PLANAR, including the required modifications at application level, the different phases involved in a rearrangement, and finally a comprehensive step-by-step example of operation.

5.3.1 Modifications to Application Code

In our implementation, the programmer is responsible for providing a rearrange function to map the irregular data access to a dense data access. Most actions taken to offload to PLANAR units are handled either by the hardware or via library calls as shown in Figure 5.3.

- `planar_alloc()` takes a minimum and maximum number of devices to be allocated and returns the number of allocated devices. If allocation is not possible, the *host* core suspends until later notification is received when the minimum number of accelerators is available. PLANAR devices are simple and can have just a few in-flight memory requests; therefore, it is difficult for a single device to saturate memory bandwidth. Having several allocated devices leads to higher memory-level parallelism and to better utilization of the memory bandwidth.

- `offloadFunc<<<N>>>()` contains the code that is executed in the PLANAR devices, which includes the rearrange function. The *N* parameter between triple chevrons determines the number of devices to offload to, and it is used to calculate the start and end bounds of the rearrange loop for each device.

- `planar_release()` signals the device to finish.

Future versions could use compiler or pragma guided insertion. The information to produce a rearrange function is often known at compile time although data is often dynamic (e.g., loop bounds, indices), so a compiler with PLANAR support could enable transparent rearrangements as suggested by previous works [106, 146].

5.3.2 Allocation of Memory and PLANAR

A memory region is allocated so that the PLANAR devices may store the rearranged data on it. This allocation is needed to prevent the *host* core from accessing an outdated dense structure (i.e., from a previous rearrange task). This is done in line 11 in Figure 5.3.

To allocate PLANAR devices, the `planar_alloc` function is used, as there can be multiple *host* cores planning to use these devices at the same time. In our proposal, the cores can access a list of the available PLANAR devices from firmware table and dynamically choose a minimum and a maximum number of accelerators they want to employ. Each device is accessed via a memory mapped work queue, which could be virtualized by the operating system using many existing mechanisms [193].

If there are not enough available PLANAR accelerators, the *host* core suspends until later notification is received when the minimum number of accelerators is available. An alternative to suspending which we did not explore, but PLANAR is capable of doing, is to execute the rearrange code on the *host* core. It is left to future work to investigate the performance impact versus suspending.

5.3.3 Offloading of Rearrange Function

When `offloadFunc<<<N>>>` is called, a command data packet is created for each of the *N* PLANAR devices. The packet consists of pointers to the sparse and final dense data, and the start address (virtual program counter) of the rearrange function.

The boundaries of the dense data structure are used to split the workload among all the PLANAR devices (*N*) in charge of rearranging the same data structure. This data packet is the equivalent of two cache blocks of data (including a header of setup information for the *host* core). Approximately five cycles are needed to save this setup information. Subsequent transport of this setup information to the PLANAR devices is dependent on the topology of the interconnect and latency of cache write-back between the *host* and the PLANAR units. Further details about our configuration are given in Table 3.2. Once the command packet is sent, the *host* core updates the *RCT* entries of the PLANAR devices that have been allocated.

PLANAR is by definition an accelerator. As such it must communicate results with the *host* core. To do so, it makes use of a common coherent interconnect. PLANAR will often work on shared data with the *host* core, meaning that modified data could exist within the *host* caches. In order to maintain memory consistency between the *host* core and PLANAR, flush operations are triggered from PLANAR before the rearrangement starts. To achieve this, after receiving the command packet, PLANAR issues cache maintenance commands [11] to flush the sparse data address range from caches. They are issued from a state machine co-located with the PLANAR device. Once it finishes, the data rearrangement can start.

5.3.4 Execution of Rearrange Functions

Every device has received its rearrange function, data pointers and work boundaries in the offloading phase. Therefore, in this phase every PLANAR device accesses the sparse data, performing the irregular memory accesses, and populating the dense data structure. It is worth noting that PLANAR is designed to have virtual memory support. This can be accomplished by connecting the PLANAR devices to an input-output memory management unit (IOMMU), which provides virtual-to-physical address translation for the direct memory accesses (DMA) that PLANAR performs. The operating system also keeps track of the pages being accessed by the PLANAR devices. Whenever a rearrangement is happening, the involved data blocks are available to the *host* core in shared state but read only. This way, memory consistency is ensured.

5.3.5 Synchronization Between PLANAR and Host

Once a PLANAR device completely populates a set of cache blocks (number explored in Section 5.4) belonging to a dense data structure, a cache maintenance operation is issued to flush the blocks from the local cache, forcing a writeback to main memory.

A synchronization mechanism between PLANAR and the *host* core is needed to ensure the *host* only accesses data when it is ready. After the flush to a set of dense cache blocks is issued, a synchronization packet is created, containing the index of the last element in the last cache block. This packet is sent to the *host* core in order to update the corresponding *RCT* entry and to wake up the *host* in case it is suspended. The *RCT* keeps information for every rearrangement in flight from a *host* core, including the boundaries (virtual addresses) for every PLANAR device as well as the last rearranged element.

After the *RCT* is updated, a cache maintenance packet is sent to the core's cache to invalidate the set of cache blocks in case they are present in the caches. This is necessary as some hardware units, such as the prefetchers, may issue memory requests to these memory locations while PLANAR is operating, caching data that has not yet been rearranged. As explained in Section 5.3.4, data being rearranged is in read only state and it cannot be accessed by the *host* due to the *RCT*. For this reason, writebacks of these invalidated cache blocks are not needed. After invalidation, the *host* core or other hardware units can access a valid version of the dense data, located in main memory, as mediated by the *RCT*.

Whenever a load address is calculated in the execute stage of the *host* core, if the *RCT* contains valid entries it will be accessed and every virtual data range compared with the load address. If a match is found and the load address is part of the already rearranged region, the memory request can proceed. Otherwise, the load instruction is moved to a FIFO queue. The queue size is limited by the instruction window of the *host* core, since the *host* core will stall or suspend as it will not be able to proceed with the execution. Later PLANAR synchronization messages will notify the *host* core, which will check the FIFO queue, moving the instructions to the load-store queue as data becomes ready.

For example, Figure 5.5 shows a high-level synchronization between a *host* core and a PLANAR device. In this case, the device rearranges the sparse vector (X) from the SpMV benchmark to a dense form (X'). In the benchmark, the matrix is stored in CSR format [31] so the vector is accessed depending on the column index vector from the matrix. After the offloading, where the rearrange function, the data structure pointers and the loop bounds (start 1, end 20,000) are provided to the accelerator, the PLANAR device starts operating. During operation, the *host* core may block or even perform other computation. As the device finishes rearranging a set of dense cache blocks, a synchronization packet is issued to the *host* core, updating the RCT and allowing the *host* to consume it.

5.3.6 Release of PLANAR Devices and Memory

PLANAR devices are released via the `planar_release` call (see Figure 5.3, line 6). The PLANAR device sends a packet to the *host* and suspends, becoming available for future operations. Once the *host* core receives the packet, the related *RCT* entry is cleared. Finally, after the dense data is consumed by the *host*, it is freed (Figure 5.3, line 16).

5.3.7 PLANAR Execution Example

Figure 5.6 shows a detailed example of operation with PLANAR. It considers a single rearrangement performed by one PLANAR device. It shows four main hardware components (top), the *host* core, the coherent interconnect, the PLANAR accelerator and a representation of several main memory pages. From the core we show the view of the *RCT*, the data cache (D$), and the FIFO queue used to hold instructions that try to access data still not rearranged. From PLANAR,

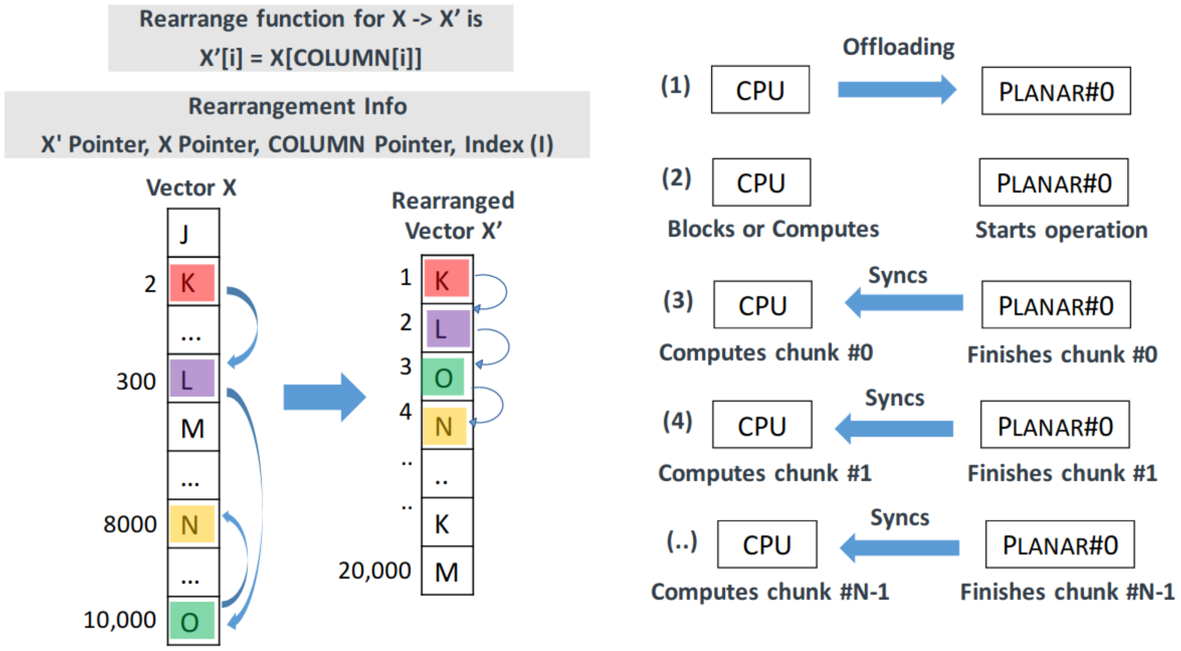


Figure 5.5: Synchronization example for a rearrangement that employs one PLANAR device.

we show the logic that, amongst other things, is in charge of processing command packets from/to all the devices, and from the device itself, the core (μcore) and data cache (μ\$).

In *Phase #1* (Section 5.3.2), the dense structure is allocated ❶ via a *malloc* call so that PLANAR has a memory region to store the dense data. The `planar_alloc` function triggers the allocation of PLANAR devices. In the example, all the devices are free and one device is requested, therefore device *ID0* is reserved for the current process (*PID 33*), allocating entry 0 in the *RCT* ❷.

In *Phase #2* (Section 5.3.3), offloading begins by sending a command data packet ❶ from the core to the PLANAR accelerator. This packet contains the information to program PLANAR, including the rearrange function with the appropriate loop bounds, which depend on the number of devices involved. After that, the core updates the corresponding *RCT* entry with the dense virtual address range and offset of rearranged elements ❷. When the control logic receives a command packet it uses a state machine to issue cache maintenance flush requests of the sparse data ❸. This ensures PLANAR devices will access the latest version from main memory. Finally, the control logic programs the PLANAR device to start the rearrangement ❹.

In *Phase #3* (Section 5.3.4), PLANAR starts executing the rearrangement, accessing the sparse data (@A) and writing to the dense data structure (@A') ❶. Subsequent accesses will fill a cache block with dense data ❷ ❸ ❹. The device continues executing the rearrange function, filling dense cache blocks until the operation completes.

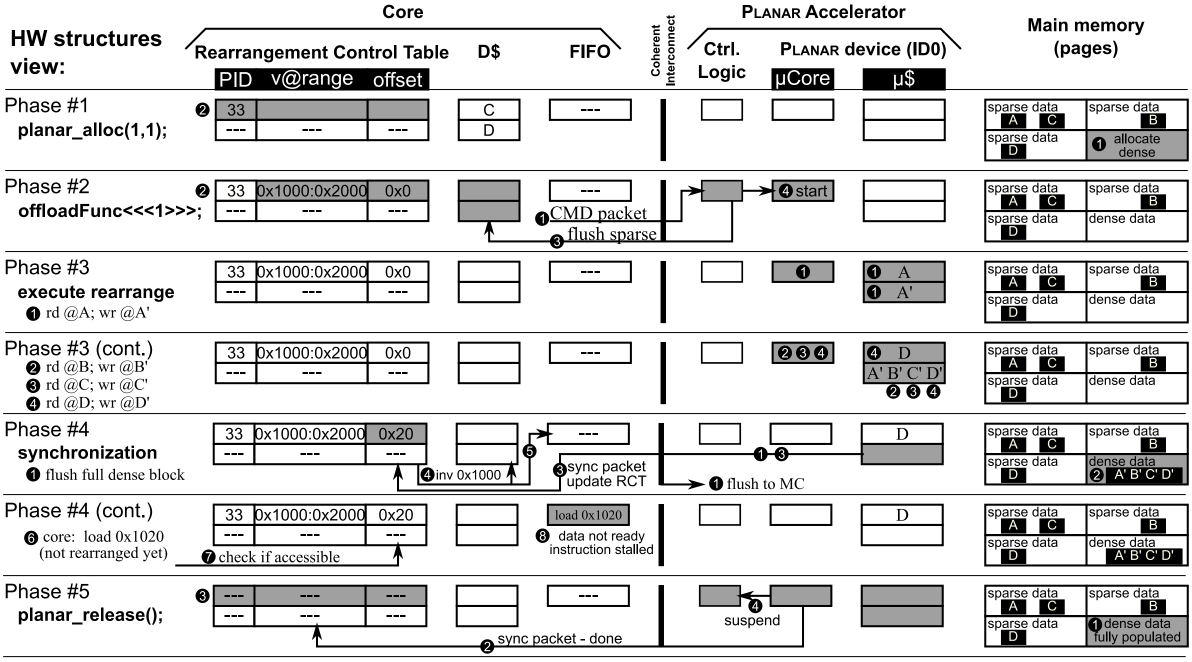


Figure 5.6: Execution example for a rearrangement that employs one PLANAR device.

In *Phase #4* (Section 5.3.5), the synchronization phase ensures that the *host* core obtains the results produced by the PLANAR device in a timely manner, while preventing the core from accessing data that has not yet been rearranged. Note that actions in this phase can happen in parallel with *Phase #3* actions. Therefore, to achieve this synchronization, for each dense cache block that is completely populated, a cache maintenance operation is issued to flush the block to the memory controller (MC) ❶. This data is eventually written to main memory in the dense data page ❷. Additionally, a synchronization packet is sent to the core to notify a new dense block is available, updating the corresponding *RCT* entry offset field ❸. To prevent the core from accessing stale data, an invalidation is sent to the cache hierarchy ❹, ensuring the dense data will be fetched from main memory. Finally, the FIFO queue is checked for stalled instructions to the now available rearranged cache block ❺, which would be able to proceed. The other mechanism present in this phase is triggered when the *host* core issues a load and there are in-flight rearrangements ❻. The *RCT* table is checked to see if the virtual target address conflicts with an in-flight range for the executing process. If that is the case the current offset determines if the rearranged data is ready to be consumed. If that is not the case, the load instruction is stalled and placed into the FIFO queue ❽. Eventually, the target address of the load instruction will be rearranged and the FIFO checked, allowing it to execute.

In *Phase #5* (Section 5.3.6), when the dense structure has been fully populated ❶, PLANAR is released. A packet is sent to the *host* core to indicate the operation has completed ❷, which clears the pertinent *RCT* entry ❸. In addition, the accelerator control logic is notified ❹ and the device suspends. Once the dense data is consumed, it is freed via a *free* function call.

5.4 Design Space Exploration

In this section, we perform a design space exploration to obtain the best configuration of PLANAR. PLANAR is envisioned as a simple microcontroller with in-order execution. For this reason, we explore the pipeline width, the data cache size, the synchronization granularity, and the number of functional units and PLANAR devices.

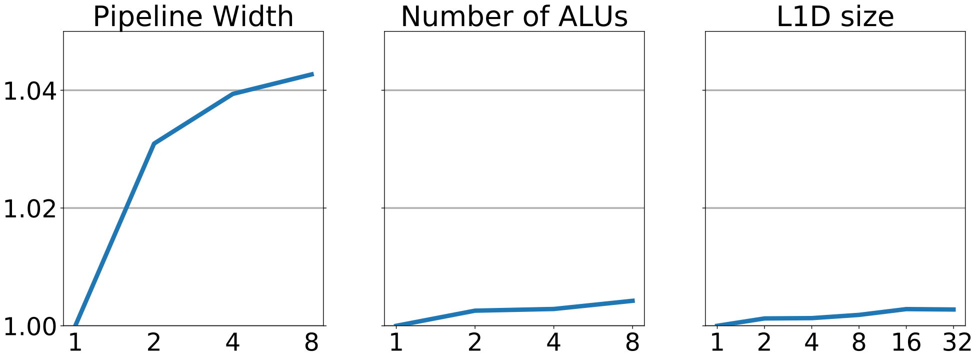


Figure 5.7: Normalized PLANAR design impact to performance in Spatter. Pipeline width (left), number of functional units (center) and L1-D cache size (right). In the *x*-axis the pipeline widths, number of functional units and cache size in *KB*.

5.4.1 Pipeline Width

Figure 5.7 (left) depicts the performance impact when changing the PLANAR pipeline width. Results are obtained using the average of all the inputs of the Spatter benchmark normalized to the single-issue scenario. When changing the input distance from 1 to 256 (see Section 3.2.2), Spatter provides a wide coverage of different irregular memory access patterns. Increasing the pipeline width from one to two provides between 2.0% and 4.0% improvements for the different inputs (3.1% on average). Further increasing the pipeline width provides diminishing improvements (3.9% and 4.3% on average for 4− and 8-wide pipelines, respectively). For small input distances, Spatter shows more cache locality. Thus, having a wider pipeline width in PLANAR provides higher performance benefits. As input distance increases, the latency of the memory requests hides the reduced performance of a narrow pipeline width.

5.4.2 Number of Functional Units

Figure 5.7 (center) shows the performance impact with respect to the number of functional units in PLANAR. Results are obtained using Spatter, normalized to one functional unit scenario. Increasing the number of functional units provides a marginal performance benefit, reaching an average 0.42% improvement with 8 units.

5.4.3 Cache Size

Figure 5.7 (right) depicts the performance impact with respect to L1-D cache size. Spatter results are normalized to the 1KB scenario. In this case, increasing the data cache size provides negligible performance benefits (0.28% on average with 32KB). This is expected as the rearrange function has a streaming memory access pattern with nearly no temporal locality. Only for small input distances, Spatter shows some cache locality, providing reduced benefits. As distance increases, the cache size does not provide any performance benefit. Regarding the L1-I cache, the rearrange function requires less than 100 instructions in the evaluated benchmarks. Thus, it does not exceed the 1KB capacity.

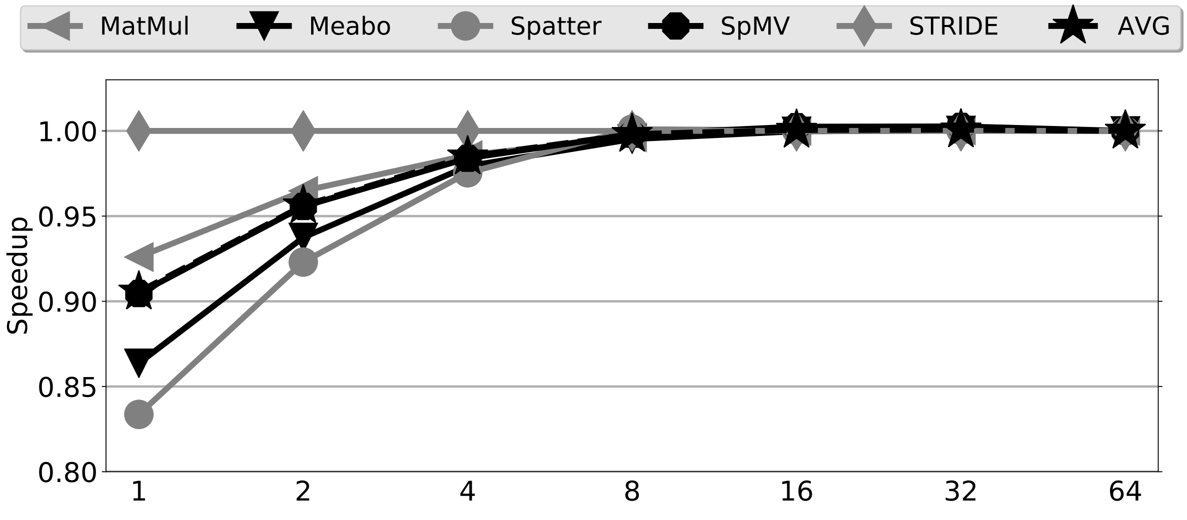


Figure 5.8: Performance relative to the number of PLANAR devices (*x*-axis), normalized to 64.

5.4.4 Number of PLANAR Devices

Figure 5.8 shows the impact of the number of PLANAR devices to performance. Due to hardware constraints, it is difficult for a single device to saturate the memory bandwidth, as not many outstanding memory requests are allowed per device. Results are obtained by performing the average across all inputs for the selected applications. We limit this study to the benchmarks that contain only a single rearrangement. This way we keep the number of devices per rearrangement constant. Results are normalized to the 64-PLANAR device scenario, which represents a *close-to-ideal* case in our simulation infrastructure. All benchmarks except STRIDE are sensitive to the number of PLANAR devices. With a single PLANAR device, performance degrades 9.5% on average with respect to 64 devices. Increasing the number of devices from 1 to 2 provides a 5.1% performance improvement, while moving from 2 to 4 provides an additional 3% improvement. With 8 devices all benchmarks are already within 1.0% the performance of 64.

5.4.5 Synchronization Granularity

During the description of the design we assume the RCT table is updated after every populated dense cache block (Section 5.3.5). However, synchronization between PLANAR and *host* can be done at a coarser granularity. We analyze scenarios where the *host* is notified it can consume rearranged data after 64 bytes, 4KB, and 8KB. Fine-grain synchronization lets the *host* consume dense data as PLANAR rearranges it, but increases synchronization traffic. As Figure 5.9 shows, coarser grain granularities of 4KB cause less than 1% performance slowdown on average. This is because after the first dense chunk finishes, PLANAR and *host* can overlap subsequent chunk rearrangements with compute over already rearranged chunks.

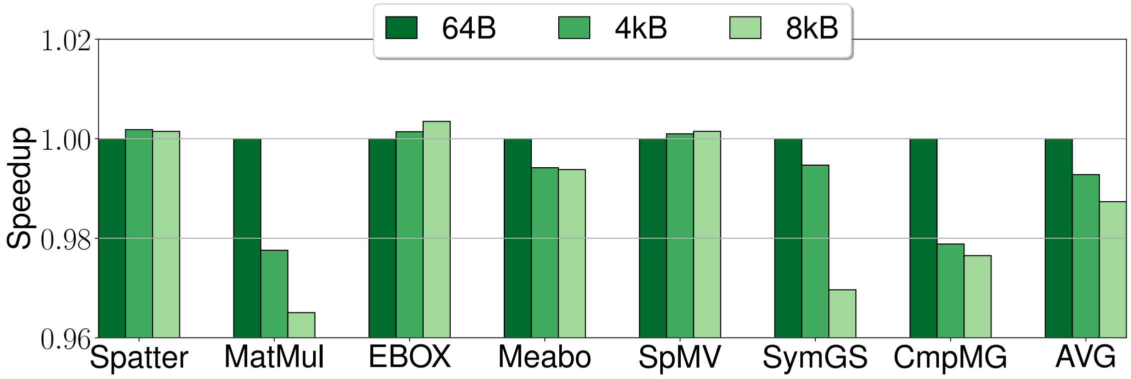


Figure 5.9: Performance with different synchronization chunk sizes normalized to 64B.

5.4.6 Selected Configuration

At the device level there is no significant improvement as the hardware complexity increases. For this reason, we choose a simple, low-power, dual issue in-order PLANAR accelerator, with a single integer functional unit, and a 1KB L1-D cache. The selected ratio of PLANAR devices with respect to off-chip bandwidth is one for every 4GB/s. Therefore, eight devices in our simulated system, as further increasing the number of devices provides negligible benefits. Finally, we chose a synchronization granularity of 4KB between PLANAR and *host* cores.

5.5 Evaluation

In this section, we analyze the performance impact of PLANAR. We use the applications described in Table 3.2.2. For each application we evaluate all the listed inputs and plot the average. We run simulations with one and eight threads to see their behavior. We also evaluate the impact of compiler auto-vectorization using the recently proposed Scalable Vector Extension (SVE) ISA [173]. SVE is vector length agnostic, meaning that a single binary can run on any target vector length [15, 14]. Therefore, we evaluate a scalar binary and an SVE-enabled binary with vector lengths of 128, 256, and 512 bits. Figure 5.10 shows speed-up for the evaluated benchmarks normalized to the scalar baseline system without PLANAR devices (*Baseline* + *Scalar*) for each core count. Figure 5.11 shows the average memory bandwidth usage.

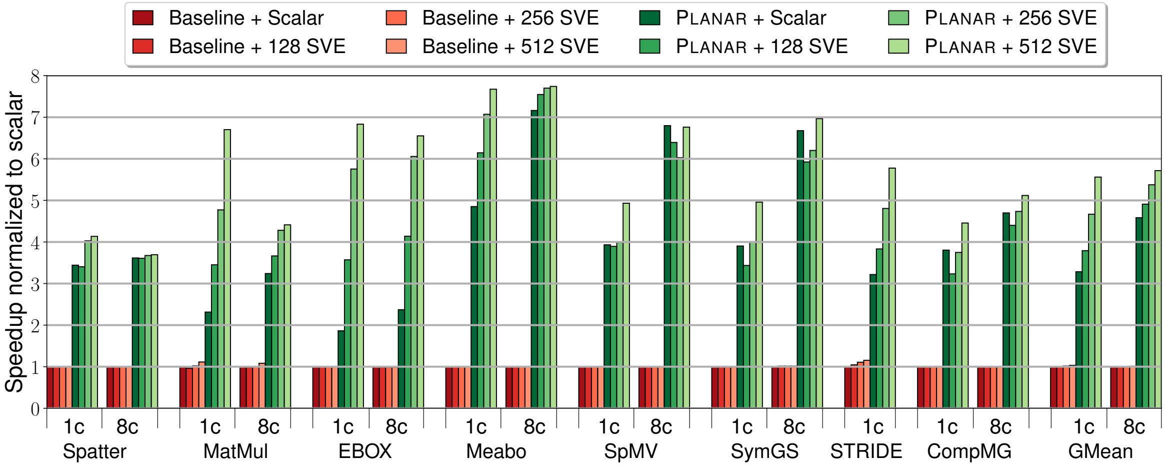


Figure 5.10: Speed-Ups with eight PLANAR devices for one and eight core runs. Both normalized to *Baseline* + *Scalar*.

In Spatter, a 3.44× speed-up is achieved when using PLANAR and a single thread without vectorization. However, benefits are input-dependent. Low distances lead to lower sparseness

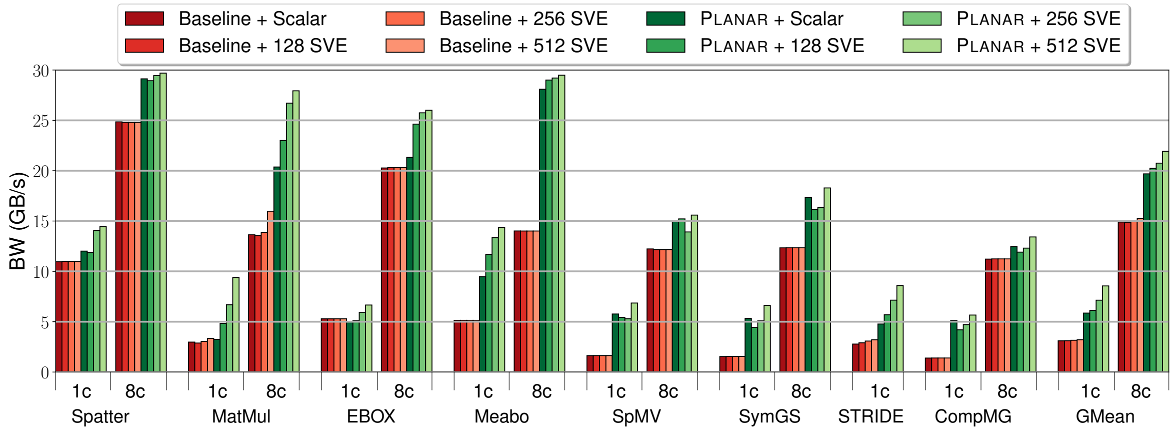


Figure 5.11: Average bandwidth usage at the memory controllers with eight PLANAR devices for one and eight cores.

and more cache locality. Higher distances affect execution time as there is a lower cache block utilization. Results also demonstrate that the original code is not auto-vectorized due to the irregular memory access pattern. However, PLANAR versions allow efficient auto-vectorization as memory accesses are now contiguous. Therefore, PLANAR unlocks further performance improvements through data-level parallelism, achieving 3.4×, 4.02× and 4.13× speed-up for 128, 256, and 512-bit SVE, respectively. Memory bandwidth is better utilized with PLANAR as cores now bring useful dense data into their caches, while sparse accesses are done near-memory. With eight threads, the speed-ups remain significant at 3.61× for scalar, with similar results for the vectorized versions. In this case vectorization is not improving performance significantly because with PLANAR we are able to saturate memory bandwidth, driving 29GB/s out of the 32GB/s peak.

In MatMul, sparse memory accesses appear when accessing the second matrix. In this case, PLANAR dynamically transposes one of the input matrices to create a contiguous memory access pattern from the *host* core standpoint. The bigger the blocks, the higher the distance between elements. Using multiple matrix block sizes, an average 2.31× speed-up is obtained on a single thread. In the baseline, vectorization provides a small performance benefit of 11%, as some phases of the application can be vectorized. Using PLANAR, SVE improves execution time as memory bandwidth is not a constraint. For instance, 512-bit SVE can drive an additional 6.05GB/s of memory bandwidth as the same baseline configuration, translating into a 6.70× speed-up. With eight threads, PLANAR speed-ups are 3.24× for scalar. However, vectorization for wide vectors offers diminishing returns as memory bandwidth saturates.

EBOX performs a Gaussian convolution by means of a filtering approximation. Filters take samples of the inputs to process the data. Consequently, PLANAR can be a good method to reorganize the input data and improve performance. In particular, EBOX extracts particular positions of an input and operates on them in two pairs (i.e., A[i] = p1*(B[-]-C[-]) + p2*(D[-]-E[-])). We have used PLANAR to create four dense structures, one for each element in the two pairs (i.e., B, C, D, E). As a result, we obtain a speed-up of 1.86× for scalar and up to 6.83× for 512-bit SVE with good vector performance scaling. In the multi-threaded scenarios the performance behavior is similar. Note that in eight thread runs PLANAR again provides better normalized speed-up compared to single-thread - 2.37× compared to 1.86×. This means that the overall design is well balanced in terms of compute, memory bandwidth and acceleration.

In Meabo, memory is accessed using a random indirection vector, which leads to non-existent locality and low cache block utilization. Single-thread runs with PLANAR obtain 4.85×, 6.14×, 7.07×, and 7.67× speed-up for scalar, 128, 256, 512-bit SVE. Using a dense structure makes a large difference in this benchmark as memory bandwidth is poorly utilized in the baseline: due to (i) the low amount of reuse, and (ii) the small amount of memory level parallelism the cores are able to extract, as stalls are common due to long latency misses and contention. With PLANAR the memory bandwidth utilization almost doubles both for single and multi-threaded scenarios, saturating it in the latter.

In SpMV and SymGS, the matrix is compressed in CSR format and the vector is accessed sparsely, jumping from one element to the other. This vector is rearranged by PLANAR. Performance is dependent on the vector access pattern. For this reason, the selected input matrices that define the vector access pattern are obtained from a wide variety of scientific domains. In SpMV the matrix is traversed forward, while in SymGS it is done forward and backwards, requiring two rearrange tasks. On average, a 3.9× speed-up is obtained for the scalar code on both applications. SVE 512-bit vectorization yields a 4.93× speed-up, while the baseline cannot be efficiently vectorized by the compiler. The performance gap is larger on eight thread runs with a 6.7× speed-up.

STRIDE is a memory-intensive application where the use of longer distances implies requesting memory more often, since fewer elements per cache block are accessed. In this particular benchmark, the *host* core and PLANAR can operate at the same time, competing for memory bandwidth resources. We evaluate multiple inputs to study this phenomenon and obtain an average speed-up of 3.21× in the scalar version. Even though some phases in this benchmark are auto-vectorized in the baseline code, the phase with sparse memory accesses is again not vectorized. For this reason, baseline reaches an improvement of 1.15× using 512-bit SVE, while the PLANAR version obtains 5.77× for the same configuration.

Lastly, CompMG performs recursive calls that contain several calls to SpMV and SymGS. For every CompMG call only two different rearrange tasks are required, as the rearrange task in SpMV is the same as the first rearrange in SymGS (i.e., the forward matrix traversal). PLANAR speed-ups are 3.8×, 3.32×, 3.74× and 4.45× for scalar, 128, 256, and 512-bit SVE. Using eight threads we observe a better speed-up than in the single thread case, such as a 5.19× in PLANAR with SVE 512-bit.

5.5.1 Impact to the Memory Hierarchy

Contiguous accesses to a dense data structure offer significant benefits compared to the original sparse access pattern, such as high cache block utilization and efficient data prefetching. Next lines demonstrate the impact PLANAR has on the memory hierarchy.

Figure 5.12 shows L1D miss reduction on *host* cores. The L1D is critical for the core's performance, and PLANAR rearrangements enable an average L1D miss reduction of 1.89× for one core. In PLANAR, all the elements contained in a rearranged cache block are referenced by the *host* core, whereas in the baseline, only one element is accessed in the worst case. The dense structure also causes a reduction of 53% in L1D miss latency. This is due to: (i) less touched cache blocks due to locality within a cache block, and (ii) memory access latencies being hidden due to better prefetching. Prefetching is less effective with irregular accesses, where data locality is difficult to exploit.

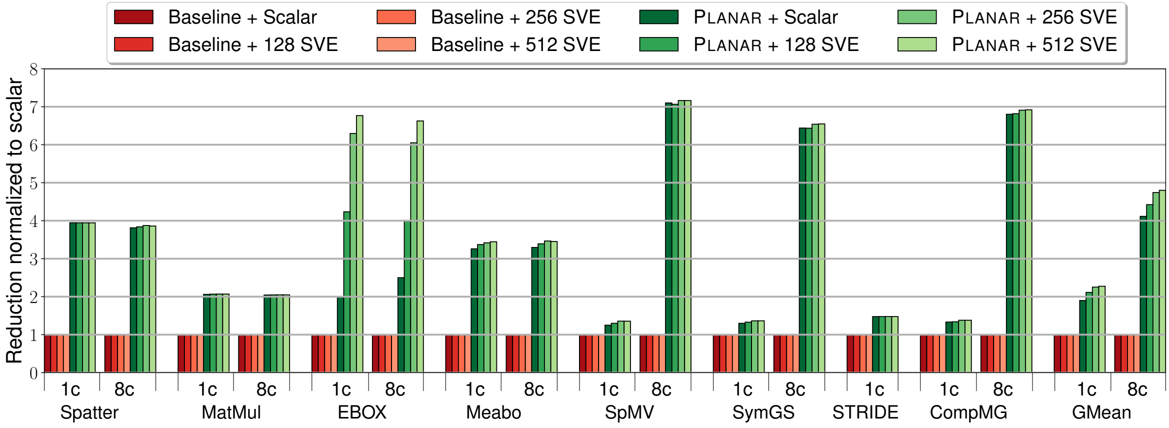


Figure 5.12: L1D miss reduction with 8 PLANAR devices for one and eight cores, both normalized to baseline scalar.

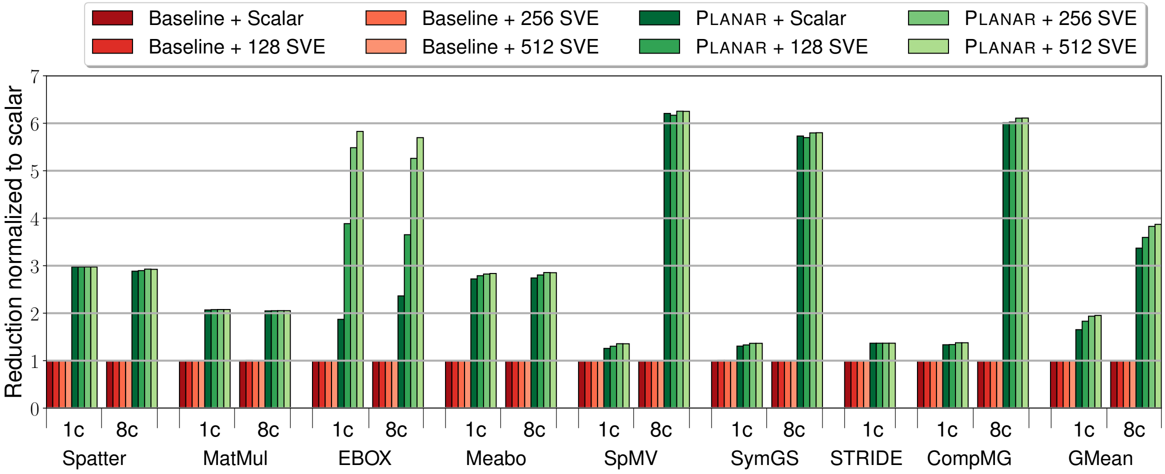


Figure 5.13: Byte reduction in the L1D-L2 bus with 8 PLANAR for one and eight cores, normalized to baseline scalar.

Figure 5.13 shows the data movement reduction in the L1D-L2 bus. The dense structure enables efficient cache block utilization and reduces cache pollution. On average, there is a 1.65× data movement reduction for one core.

In terms of DRAM accesses, one of the advantages of performing DLT is that subsequent accesses to the dense structure do not require accessing the intermediate data structure of the indirect memory access. In the baseline, both the intermediate and sparse structures are accessed. The latter may even have cache blocks accessed more than once, due to the significant cache pollution. Figure 5.14 depicts the normalized number of DRAM accesses. When using PLANAR, 1.6% of the total DRAM accesses are generated by PLANAR devices on average (up to 9.09% in CompMG). Overall, there is an average 41.89% DRAM access reduction. This reduction is primarily due to increased reuse, which can be observed indirectly through the increased L1D hit rate (see Figure 5.12) and L1D to L2 bandwidth reduction (see Figure 5.13). Despite writing the dense data structure back to memory, the actual accesses to DRAM are reduced because of better data cache utilization and far higher reuse of cached data. Writing data back to main memory, is of course, a compromise, however, it is one that reduces the repeated re-arrangement calls needed by alternatives such as Impulse, it also enables the dense data structure to be reused many times before being freed.

5.5.2 Area and Power Overhead

PLANAR devices can be compared to the Arm Cortex $M0+$ microprocessor, which is augmented with a 64-bit ALU (as discussed in Section 5.4). Using public data for an equivalent $M0+$ at

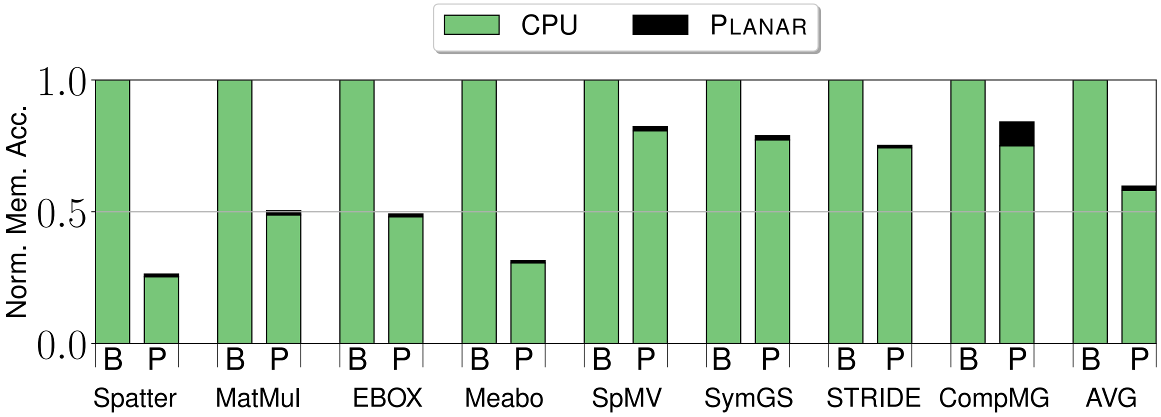


Figure 5.14: Normalized DRAM accesses for baseline (B) and PLANAR (P) on 8 cores with scalar codes.

40LP [13], the dynamic power is given as $5.3\mu W/MHz$ and the floor planned area as $.008mm^2$. To estimate the area of a single microcontroller, we scale these numbers, considering fin pitch, gate pitch, and interconnect pitch, using data from [57, 199, 196, 195, 18, 51] to arrive at a 12× area reduction when moving from 40nm to 7nm and an estimated reduction in power of 10×. Therefore, a system-on-chip could place 8 PLANAR devices, with their caches, using $< .25mm^2$ area of floor plan on chip. Equivalently, energy for this configuration would be $< .015W/GHz$.

We also estimate the area of the out-of-order core described in Table 3.2. We start from a similar production core at 20nm [127] and scale it to 7nm, which arrives at a $\approx 4mm^2$ in area. As a result, 8 PLANAR devices represent a 6.25% of the area of an out-of-order core. Similarly, 8 PLANAR devices represent a 1% of the dynamic power of an out-of-order core. Both results assume the devices run at 2GHz.

To estimate the dynamic energy and power consumption for our PLANAR proposal we used McPAT 1.3 [121] with the enhancements proposed by Xi *et al.* [198]. We performed this estimation, using a process technology node of 22nm, a supply voltage of 0.8V, and the default clock gating scheme. Figure 5.15 depicts the dynamic energy reduction for the applications with eight *host* cores. Overall, dynamic energy is reduced by more than 40% and up to 70% in Meabo. Energy savings are mainly due to reduced data movement across the memory hierarchy and reduced execution time (speed-up) as shown in Figure 5.10.

Figure 5.16 depicts the dynamic energy breakdown for the same applications. Compared to the baseline, PLANAR spends less DRAM energy. As explained in Section 5.5.1, PLANAR creates a dense structure and makes the applications more compute intensive form the host standpoint, as average data access latencies are lower.

The total dynamic power is higher in PLANAR. On average, the dynamic power increases with PLANAR by 2.41× in the cores, by 1.41× in the memory controller, and by 1.14× in DRAM. This is due to an increase in terms of activity per unit of time. However, taking into account the performance speed-ups of PLANAR, the overall energy spent is significantly reduced. As previously discussed, static power is barely affected when adding PLANAR.

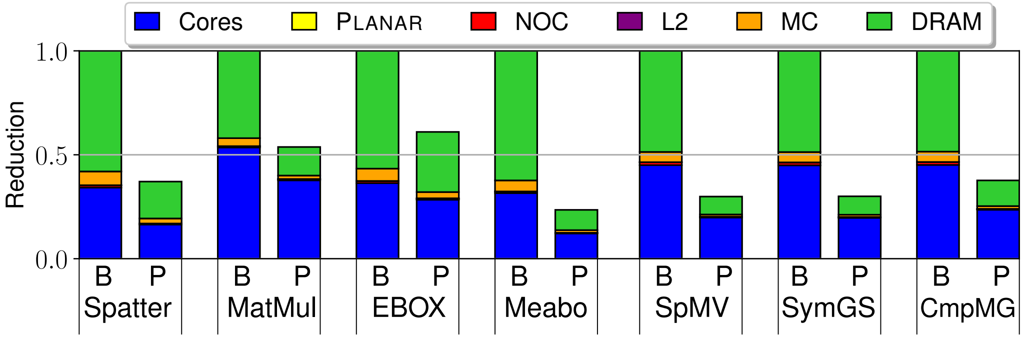


Figure 5.15: Dynamic energy reduction baseline (B) vs PLANAR (P) in scalar applications on eight cores.

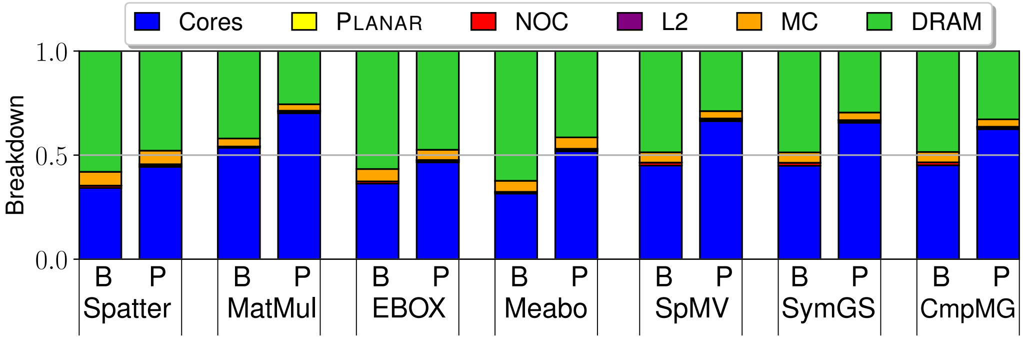


Figure 5.16: Dynamic energy breakdown baseline (B) vs PLANAR (P) in scalar applications on eight cores.

5.5.3 Comparison to Other Proposals

We compare PLANAR to Impulse [206]. Impulse is a hardware approach that creates a dense structure out of a sparse one. It performs data reordering in the memory controller as the *host* core accesses memory belonging to a *shadow address space*, which must be contiguous in physical memory. Thus, Impulse rearranges data *just in time*, not like PLANAR which is capable of rearranging data *before-hand* as the *host* is executing other code regions. Therefore, Impulse cannot hide the rearrangement latency. Moreover, in case the dense data is evicted and requested by the *host* again, Impulse will perform a new rearrangement, as it cannot assume that the original and new rearrage functions are the same. Finally, in a design with multiple memory controllers, the rearrange functions of different Impulse instances are not synchronized, limiting scalability.

Figure 5.17 depicts the performance comparison between PLANAR and Impulse for SpMV. PLANAR obtains an average 2.12× speed-up compared to Impulse. This is due to: (i) higher MLP of data rearrangements in PLANAR; (ii) more data reuse; and (iii) less data movement in the cache hierarchy as dense data is created just once. For instance, snoop traffic from the core to the L2 cache is 21× higher in Impulse.

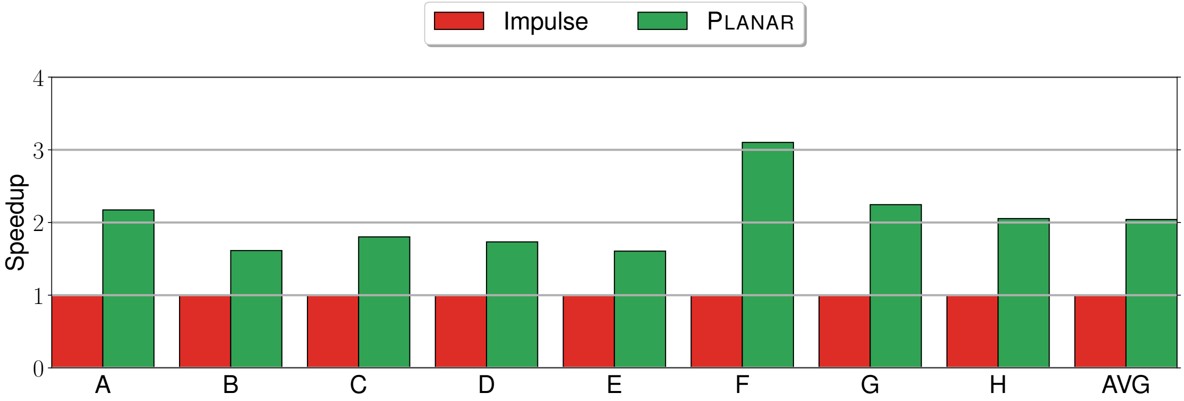


Figure 5.17: Performance of Impulse vs PLANAR in SpMV. In the *x*-axis, the matrix inputs from Table 3.2.2.

In addition, we compare to a more recent DLT accelerator proposal by Hoang *et al.* [89]. The DLT accelerator is tightly coupled to the *host* core, whereas PLANAR is connected at the memory controller level. Their proposal can bypass the cache hierarchy and directly access main memory, as PLANAR does, but requires an additional data bus. The accelerator does not allow the *host* to consume the dense data as the device rearranges it, and memory accesses are blocked on the *host* during DLT operation to maintain data consistency. Moreover, it supports a maximum of four parallel operations, contrary to PLANAR, where more devices can be added to the system, enabling additional parallelism.

Figure 5.18 depicts the performance comparison between PLANAR and the DLT accelerator for several benchmarks. We employ up to 8 PLANAR devices and also allow up to 8 simultaneous operations on the DLT accelerator. On average, PLANAR obtains a 2.23× speed-up compared to the DLT accelerator, with a maximum of 5.82× in SymGS. This is due to two main reasons: in PLANAR (i) the *host* is not blocked while devices are operating, as PLANAR effectively decouples rearrange and compute, and (ii) the *host* can consume dense data as it is rearranged, which hides rearrangement latencies.

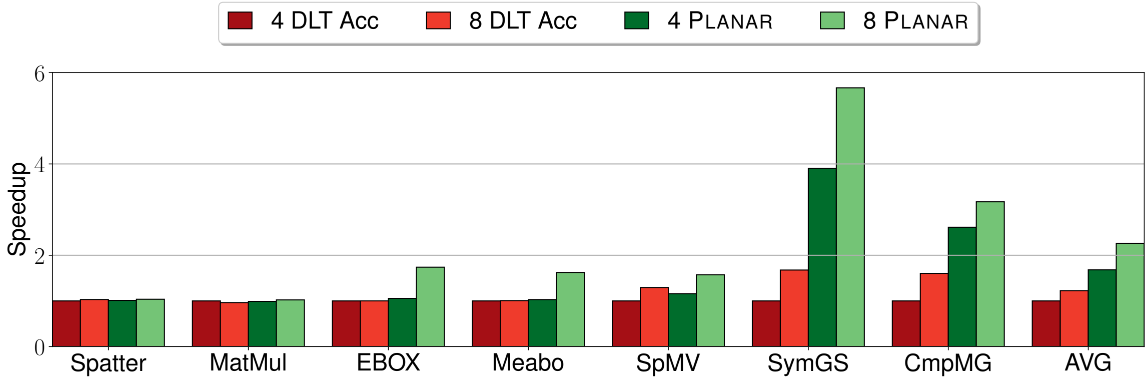


Figure 5.18: Performance of the DLT accelerator vs PLANAR for multiple applications.

5.6 Conclusions

Irregular memory accesses represent a challenge for current and future architectures. In this work, we present PLANAR, a programmable near-memory accelerator that rearranges sparse data into a dense representation. Contrary to prior proposals, the design of PLANAR scales with multi-core systems, hides operation latency by performing non-blocking fine-grain data rearrangements, and eases programmability by supporting virtual memory and conventional memory allocation mechanisms. Moreover, accesses to the dense structure expose locality, favouring prefetching and enabling efficient vectorization in applications with irregular memory accesses. No prior solution provides all such properties at once.

Our evaluation shows that PLANAR improves cache block utilization and reduces on-chip data movement. As a result, PLANAR improves performance for single-threaded runs by 3.28× and 5.56×, and for multi-threaded executions by 4.58× and 5.71×, for scalar and compiler-vectorized codes. Finally, we compare PLANAR to two state-of-the-art proposals, achieving 2.12× and 2.23× average performance improvements.

Chapter 6

Near Memory Compute Engine

6.1 Introduction

This chapter still targets the problems derived from irregular memory accesses described in Chapter 5. Processing In Memory (PIM) has been recently proposed as an alternative to deal with this issue. The idea relies on placing computing resources close to where the data resides. Recent processing in memory proposals are summarized by Zhang *et al.* [204] and Balasubramonian *et al.* [23]. In this field, most approaches requiere a special memory technology (e.g., 3D-stacking) to place the compute logic in the memory chip. Some popular examples are the Active Memory Cube [138] and Tesseract [2], which rely on the Hybrid Memory Cube (HMC) [144]. Moreover, most PIM proposals operate with physical memory rather than with virtual memory, and require a high programming effort to use them, a phenomenon known as the Programmability Wall [86].

In this chapter, we present a *pRogrammable nEar Memory cOmpuTe Engine (*REMOTE*)*, a novel hardware approach, located on-chip side and connected to the memory controller, which performs computation outside of the memory hierarchy and closer to where the data resides. REMOTE devices share the coherent network with the *host* processor. They operate with virtual memory and do not require contiguous physical memory allocation or uncached memory regions, as most prior proposals do [136, 62]. Contrary to many PIM proposals, our approach can be used on any system, no matter the memory technology. REMOTE devices are programmed in the application by means of pragmas [52] and used depending on their availability as mediated by the runtime system [185, 43].

This chapter makes the following contributions:

- The design, including area and power estimations of REMOTE, a programmable near-memory accelerator. We describe the functional changes to incorporate REMOTE into a system-on-chip with out-of-order cores. The programming interface and the changes to the runtime system are explained in detail. REMOTE is designed to work and scale with multi-core systems. It operates with virtual memory and it does not require custom memory allocators or uncached memory regions, as many existing proposals do.

- A comprehensive evaluation, including performance and energy results, shows that REMOTE is a good candidate to execute graph applications as it provides an average speed-up of 6.33× and 9.76×, with 16 and 32 accelerators, whereas the baseline reaches 5.76× with 8 out-of-order cores. REMOTE is also a good alternative to run HPC applications, outperforming the baseline in particular configurations.
- A profiling of the applications that benefit more from the REMOTE devices. The main aim is that the runtime system dynamically detects and offloads suitable parallel regions or tasks to REMOTE devices.
- A detailed comparison to three state-of-the-art hardware proposals. In particular, we compare to the Smart Memory Cube (SMC) [21], to the Programmable Prefetcher [3] and to a bigLITTLE configuration [186]. Our comparison shows that 8 REMOTE devices outperform the SMC by 1.41×, the Programmable Prefetcher by 1.82×, and 32 accelerators outperform a (8+16) bigLITTLE configuration by 1.29×.

6.2 Motivation

To explain the limitations and consequences of irregular memory access patterns, and the advantages of doing so with a mechanism such as REMOTE, we have chosen a case study using the PageRank benchmark. PageRank is a memory-intensive benchmark that traverses a graph and performs a simple computation on the vertices. In the original code (see Figure 6.1), line 11 represents a sparse memory access. The memory access pattern is governed by the "vit→edges()" array which depends on the graph connectivity.

We have executed this application in a system with 8 out-of-order cores, a shared memory hierarchy of three cache levels and an input that exceeds the capacity of the last-level cache (LLC). This execution takes 0.534 seconds, presents a 94.16% LLC missrate and has 125.27 misses per kilo-instruction (MPKI). Therefore, this application is dominated by data movement, as cores spend most part of their execution requesting data rather than performing useful computation. A metric that clearly demostrates this phenomenon is the IPC, that ranges between 0.0356 and 0.0425 out of 4 for all the cores. Moreover, prefetchers are inefficient

```
..............
#pragma omp parallel{
    ..............
    for (unsigned vid=start; vid<end; vid++){
        vertex_iterator vit = g.find_vertex(vid);
        float pr_push = damp *
            vit >property().old_pr / (double) vit >edges_size();
            for (iterator eit=vit >edges_begin();
                 eit!=vit >edges_end();
                 eit++){
            uint64_t dest = eit >target();
            vertex_iterator dvit = g.find_vertex(dest);
            #pragma omp atomic
            dvit >property().pr += pr_push;
        }
    }
    ..............
}
..............
```

Figure 6.1: Original PageRank code.

as there is not a predictable access pattern and they contribute to the existing cache block pollution.

Hardware mechanisms such as REMOTE, a simple in-order core connected to the memory controller, are key in this type of workloads. REMOTE devices operate closer to main memory taking advantage from the higher memory bandwidth. Figure 6.2 shows the PageRank benchmark adapted for REMOTE. The only difference is the "target(REMOTE)" clause in the pragma. This information makes the runtime system schedule this parallel region to the REMOTE devices, as they are a good candidate to execute this code instead of the *host* cores. We have executed this PageRank version on a system with up to 32 REMOTE devices. In this case, the execution takes 0.254 seconds (speedup of 2.102×) and reduces the number of bytes transferred from the cores to main memory by 1.46×.

As a result, we may conclude that REMOTE accelerators are a good alternative for this type of benchmarks, as they introduce a higher parallelism, lead to performance benefits and they are more energy efficient than the *host* cores.

Table 6.1 shows a comparison of the REMOTE proposal with other approaches from the state of the art. Contrary to them, REMOTE is a general purpose solution, as it targets any application suffering from irregular memory access patterns. It does not require a particular system distribution or a specific memory technology. Moreover, it is easy to program and the tasks are scheduled between the accelerators and the *host* cores by the runtime system as determined by the programmer or compiler. Finally, to demonstrate the potential of REMOTE, in Section 6.5 we perform a quantitative comparison with three hardware approaches from the state of the art.

```
..............
#pragma omp parallel target(REMOTE){
    ..............
    for (unsigned vid=start; vid<end; vid++){
        vertex_iterator vit = g.find_vertex(vid);
        float pr_push = damp *
            vit >property().old_pr / (double) vit >edges_size();
            for (iterator eit=vit >edges_begin();
                eit!=vit >edges_end(); eit++){
            uint64_t dest = eit >target();
            vertex_iterator dvit = g.find_vertex(dest);
            #pragma omp atomic
            dvit >property().pr += pr_push;
        }
    }
    ..............
}
..............
```

Figure 6.2: REMOTE version of the PageRank code.

6.3 Proposal

Many applications suffer from irregular memory access patterns. In such applications, the memory subsystem is poorly utilized, leading to latency and bandwidth bottlenecks because of a low cache block utilization [28] caused by dispersed memory accesses that lead to high (but underutilized) traffic on data transfer networks (e.g., coherence bus, interconnects) [129].

In this section we describe the design of the REMOTE devices, a novel hardware approach which targets these situations. We also explain the required modifications to the application and the runtime system, in order to program and schedule workloads to the accelerators.

Table 6.1: Comparison with state-of-the-art proposals.

Features	**Pref. [202, 99]**	**Prog. Pref. [3]**	**SPiDRE [29]**	**Acc. graph. [142]**	
General purpose	✓	✓	✓	✗	
No specific system/tech	✓	✓	✓	✓	
Easy to program	✓	✗	✗	✗	
Reduces data movement	✗	✗	✓	✓	
Dynamic scheduling	-	✗	✗	✗	
Host computes	✓	✓	✓	✗	
Quantit. comparison to SoA	✓	✓	✗	✗	
Features	**Livia [124]**	**Tesseract [2]**	**GraphPIM [136]**	**TOM [90]**	**REMOTE**
General purpose	✓	✗	✗	✓	✓
No specific system/tech	✗	✗	✗	✗	✓
Easy to program	✗	✗	✓	✗	✓
Reduces data movement	✓	✓	✓	✓	✓
Dynamic scheduling	✓	✗	✗	✓	✓
Host computes	Tasks split	✗	✗	✗	Tasks split
Quantit. comparison to SoA	✓	✗	✗	✗	✓

6.3.1 The REMOTE Device

Figure 5.4 shows the overall architecture of our design. It depicts a system with several high-performance cores sharing a three-level cache hierarchy. We add REMOTE devices and supporting hardware to perform address translations. In particular, we connect an Input-Output Memory Management Unit (IOMMU), which provides virtual-to-physical address translation for the direct memory accesses (DMA) that REMOTE performs. The REMOTE units are a set of in-order, low power, programmable cores attached to the memory controller that share the same coherent network with the cores. They are responsible for executing kernels dominated by irregular memory accesses.

The runtime system is in charge of scheduling codes to the REMOTE devices, which are marked as good offloading candidates by the compiler or programmer in the application. The REMOTE units run until completion of the kernels, which are typically only a few lines of code. During execution, they access directly the main memory skipping the cache hierarchy which is not useful in this type of applications. Finally, they sleep until the runtime system schedules additional tasks on them.

Attached to every REMOTE unit there is a small instruction cache and a small data cache. The amount of code footprint required for most applications is small, so instruction cache size requirements are minor: in the benchmarks described in Section 3.2 a maximum of 1 KB is fetched from main memory by the devices for the entirety of each application. The data cache is also small as REMOTE is aimed at applications with an irregular memory access pattern.

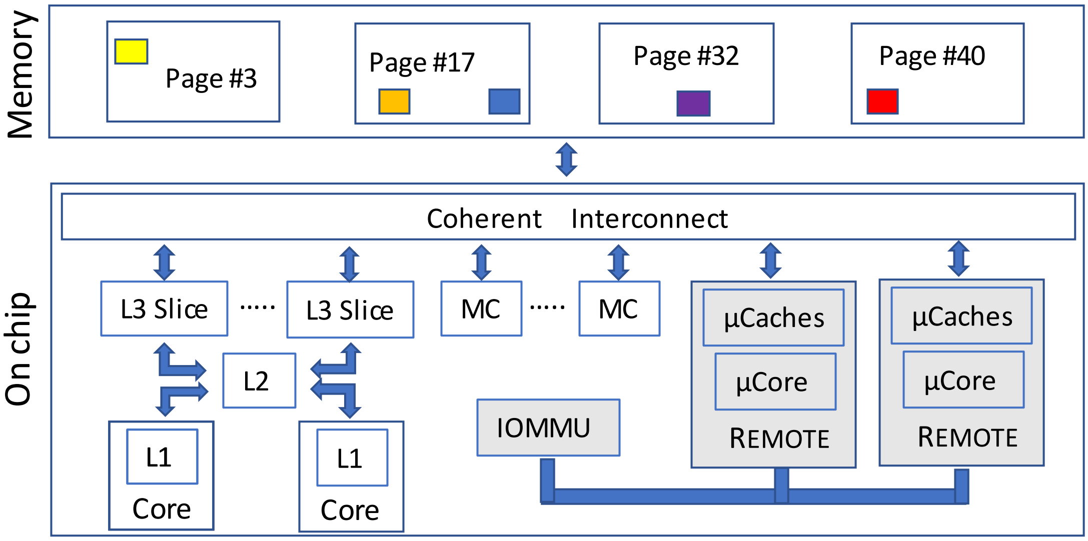


Figure 6.3: System overview with two REMOTE devices.

However, these applications also contain some small auxiliary structures that have spatial locality so, therefore, a modest data cache can make a large difference in these situations.

Figure 6.3 shows a system with two REMOTE devices. In this case, the application accesses data from memory pages 3, 17, 32 and 40. This represents an irregular access pattern, as only one element of these pages is accessed. The only exception is page 32, which is accessed twice to obtain two elements in the same cache block. In particular, if we consider 64-bit elements, 12.5% of the cache blocks are used in pages 3, 17, and 40, and 25% in page 32. Moreover, if both elements in page 32 are not accessed in a small period of time, it is possible that the cache block containing them is evicted, requiring a new memory access. In these situations: (i) prefetching schemes do not work properly, (ii) there is a low memory bandwidth utilization, and (iii) there is a high and under utilized data movement (i.e., cache pollution). REMOTE is a good alternative in these scenarios, as it operates closer to main memory, taking advantage of lower access latencies and not causing an elevated traffic on the chip. Moreover, due to its simple and efficient design, multiple devices may operate in parallel, exploiting a higher level of parallelism than the *host* cores.

6.3.1.1 Design

In order to obtain the best hardware design for REMOTE, Section 6.5 performs a detailed design space exploration. Finally, REMOTE results into an in-order core with a 1-width stage pipeline. We also include a small TLB of 8 entries to perform fast virtual-to-physical memory address translations. The final REMOTE hardware configuration is really simple and energy efficient, and it is summarized in Table 3.3.

Due to compatibility issues, the REMOTE units must execute the same ISA than the *host* cores. However, some ISAs provide a subset of instructions that have similar functionality but occupy less space, making the pipeline simpler, than the original ISA (e.g., Thumb in Arm [9]). The compiler may generate Thumb instructions as it identifies, or the programmer marks, a code region as suitable for REMOTE.

6.3.1.2 Virtual Memory Support

The REMOTE devices are designed to have virtual memory support. For this reason, we include an Input-Output Memory Management Unit (IOMMU) [8] that connects DMA devices that directly access main memory. It allows peripheral devices to use virtual memory using conventional memory allocation functions (i.e., contiguous physical memory not needed). Moreover, the IOMMU offers memory protection from malicious devices.

6.3.1.3 Operating System

REMOTE units are visible to the operating system, but the OS cannot schedule processes to them unless it is explicitly marked in the application code either by the programmer or compiler. In the event of a page table walk or a page fault, the IOMMU takes over and makes the *host* core perform the required accesses to the kernel. In these situations, the *host* cores perform a much faster context switch and kernel code execution than the REMOTE units due to a more complex hardware design. Thus, the REMOTE devices do not require privileged instructions.

6.3.1.4 Hardware Requirements

We perform an area and a power estimation of our REMOTE proposal using McPAT v1.3 [121] with the enhancements proposed by Xi *et al.* [198]. We perform this estimation, using a process technology node of 22nm, a supply voltage of 0.8V, and the default clock gating scheme. In our estimation, the area of a single REMOTE device corresponds to the 26.64% of an out-of-order core and the power of a single REMOTE device corresponds to the 4.97% of an out-of-order core. We also employ McPAT to obtain the total energy consumption of the hardware configuration listed in Table 3.3 when executing all the applications from Table 3.6. It is described in Section 6.5.

6.3.2 Changes to the Application

Our current implementation requires the programmer to annotate the code regions with irregular memory accesses, although future versions could resort to the compiler to do so. In particular, we conceive the REMOTE devices to be programmed through a shared memory programming model, such as OpenMP [52] or OmpSs [63]. In these scenarios, the programmer writes sequential code and adds annotations to define a parallel region to be executed by several threads, or to define *tasks* which are executed asynchronously following the synchronization rules defined by the dependences.

In order to program the REMOTE devices, the programmer just needs to add "*target(*REMOTE*)*" to the "*#pragma omp parallel*" clause or to the "*#pragma omp task depend(in/out/inout)*" clause. In the first case, the parallel region will be executed simultaneously by all the available REMOTE devices and in the second case, a task will be executed by one REMOTE unit. Our evaluation in Section 6.5 assumes that, in both cases, the code is executed in the accelerators, waiting for them if none is available. We do not consider executing a code targeting REMOTE in the *host* cores if all the devices are busy and any *host* core is idle. This is left to future work.

6.3.3 Changes to the Runtime System

We modified the Nanos++ runtime system [63] to perform kernel scheduling to the REMOTE devices. We believe these changes could be extended to any runtime system supporting a shared memory programming model. This process involves:

(1) To modify the software thread class to include a "isREMOTE" variable to differenciate *host* core software threads from REMOTE software threads.

(2) To include a new option to specify the total number of REMOTE accelerators in the system.

(3) To create a new class for REMOTE devices, similar to the class of the *host* cores, that binds software REMOTE threads to hardware REMOTE threads.

(4) To modify the work descriptor class with a new variable which specifies whether tasks are suitable for the *host* core threads or for the REMOTE threads. This variable is true if the "*target(*REMOTE*)*" clause is specified it the application, or false otherwise.

(5) To modify any of the existing scheduling policies and to add a new ready queue for the REMOTE threads. Once the task dependencies are freed, any task will be moved to any of these queues, depending on its work descriptor, and the *host* or REMOTE threads will pick it to be executed.

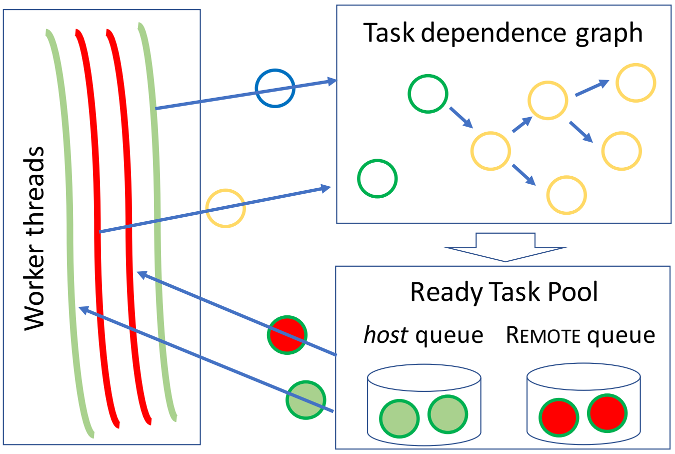


Figure 6.4: Nanos++ task flow over runtime structures with the changes to add REMOTE.

Figure 6.4 shows a scheme of the task flow where each circle represents a task. Each circle color is associated with a task state: yellow for a task being created or a submitted task, green for a ready task and blue for a finished task. First, a thread pushes the created tasks into the task dependence graph to determine the task order. Then, other threads "push" the finalized tasks into the task dependence graph to notify the successor tasks. In addition, this action removes the finished tasks from the graph and adds the tasks that become ready into the ready task pool. Ready tasks may be candidates for the *host* or REMOTE threads. Finally, the worker threads (*host* in green or REMOTE threads in red) try to acquire ready tasks from the ready tasks pool to execute them.

6.4 Design Space Exploration

In this section, we perform a design space exploration to obtain the best hardware configuration for REMOTE. REMOTE is envisioned as a simple core with in-order execution. For this reason, we study the impact to performance of the pipeline width, the number and the latency of the functional units and the clock frequency of the REMOTE devices.

6.4.1 Pipeline width

Figure 6.5 depicts the performance impact when doubling the REMOTE pipeline width and scaling the internal core resources. This means that a 1-width stage pipeline duplicates the internal hardware (i.e., doubles the buffers size in each pipeline stage) when moving to a 2-width stage pipeline. It is done to avoid bottlenecks during operation. In a 2-width stage pipeline every stage may execute two instructions simultaneously. Results are normalized to a scenario where the pipeline width of the accelerators is 1. Results demostrate that the pipeline width is not fundamental to achieve a good performance benefit in REMOTE executions. The highest benefit is 9.97% in RandAcc and the average is 1.62%.

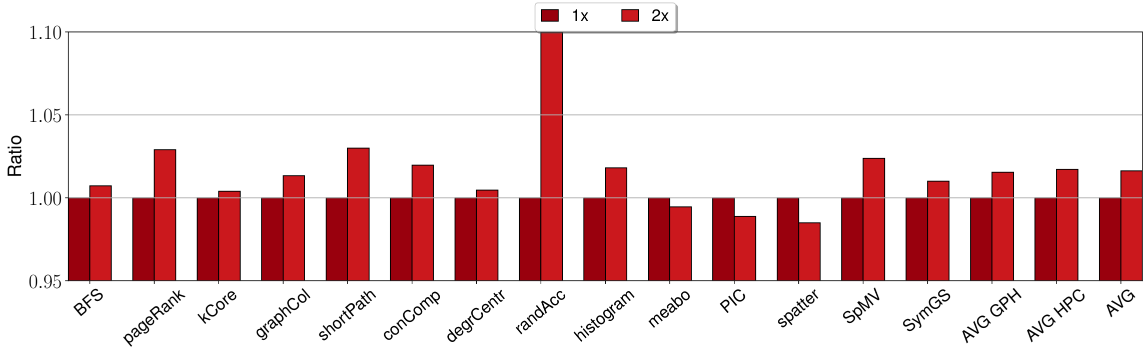


Figure 6.5: Impact of REMOTE's pipeline width (1 and 2) to performance. Results normalized to a pipeline of width one. In x-axis multiple graph and HPC bechmarks, including average numbers.

6.4.2 Number of functional units

Figure 6.6 shows the impact of the number of functional units in REMOTE to performance. In this case, we simulate all the benchmarks with up to 4 functional units and normalize the results to 1 functional unit. The functional units support integer and floating-point operations, including multiply and divisions. The benchmarks employed in our evaluation perform this type of operations near memory. Performance gains are higher on HPC benchmarks (up to 23.58% in Spatter and average of 9.12%, with 4 units), as the arithmetic intensity is higher than in graph applications (up to 7.83% in PageRank and average of 4%, with 4 units).

6.4.3 Latency of functional units

Figure 6.7 shows the performance impact with respect to instruction latency. In particular, the latencies are multiplied by 1×, 2× and 4×. In applications domained by irregular memory accesses, the main bottleneck corresponds to the memory access latency. This study identifies how much we can increase the instruction latency, saving energy, without affecting performance. When it comes to graph applications, this feature does not affect much, as the arithmetic intensity is low. For example, the highest performance degradation is 1.14% for conComp

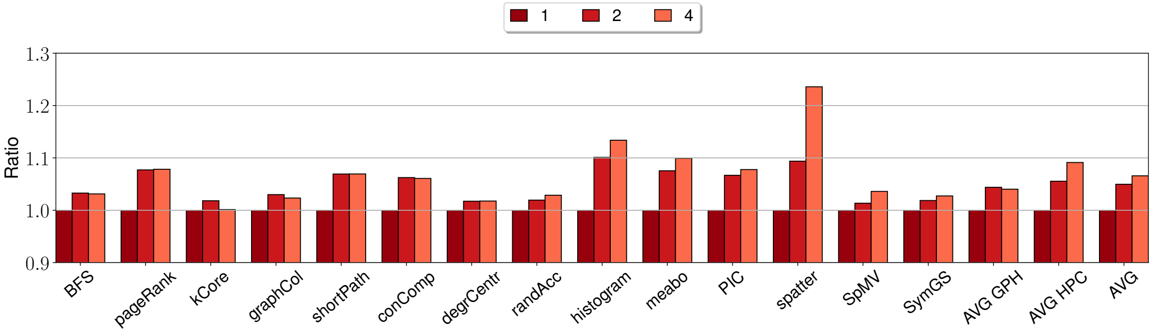


Figure 6.6: Impact of REMOTE's functional units (1, 2 and 4) to performance. Results normalized to 1 functional unit. In *x*-axis multiple graph and HPC bechmarks, including average numbers.

when latency is multiplied by 4×, while the average slowdowns are negligible. When it comes to HPC applications, the highest slowdown is 5.66% for histogram when latency is multiplied by 4×, the average result reaches 1.3% with a 4× latency. These results demonstrate that, for applications with irregular memory accesses, fast functional units are not fundamental to obtain performance gains and their latency can be increased to reduce area and power in REMOTE.

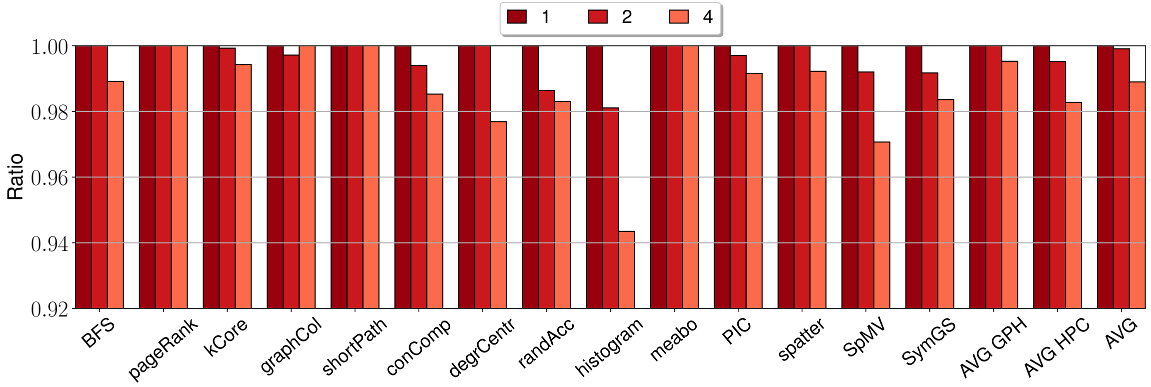


Figure 6.7: Impact of the latency of REMOTE's functional units (1×, 2× and 4×) to performance. Results normalized to 1× functional unit. In *x*-axis multiple graph and HPC bechmarks, including average numbers.

6.4.4 Frequency

Figure 6.8 shows the impact of the frequency of the devices to performance. In this case, we consider frequencies of 1, 2 and 4GHz. Results are normalized to 1GHz. Results are similar in graph (5.7% benefits at 2GHz and 9.67% benefits at 4GHz) and in the HPC applications (7%

benefits at 2GHz and 10.56% benefits at 4GHz). We conclude that increasing the frequency does not translate into linear performance benefits.

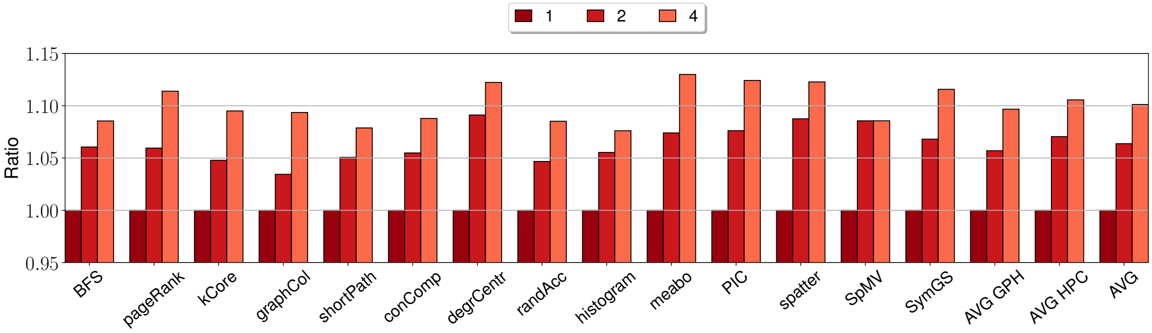


Figure 6.8: Impact of REMOTE's frequency (1, 2 and 4GHz) to performance. Results normalized to 1GHz. In *x*-axis multiple graph and HPC bechmarks, including average numbers.

6.4.5 Selected configuration

After doing the previous design explorations, we have chosen the configuration that provides the best performance at the optimal energy cost. In particular, we have chosen a 1-width REMOTE device with 2 functional units at a 2× latency, running at 2GHz.

6.5 Evaluation

6.5.1 Profiling of the Applications

Next, we profile the applications to see how REMOTE may improve their performance depending on their behavior.

Figure 6.9 shows the LLC missrate for several applications. We can observe that almost all applications present large missrates above 90%. However, as depicted in Figure 6.10, graph benchmarks present a much higher MPKI ratio when compared to HPC benchmarks, meaning that their execution is dominated by main memory accesses. On the other hand, HPC applications have higher arithmetic intensity (i.e., 8.55× higher, as shown in Figure 6.11). Therefore, HPC applications can take further advantage from agressive out-of-order cores, lowering potential speed-ups when using REMOTE devices.

Overall, these three factors (i.e., LLC missrate, MPKI rate and arithmetic intensity) are key to consider REMOTE devices as good candidates for the execution of these applications. In the next section, we evaluate the benefits in performance and energy when using REMOTE.

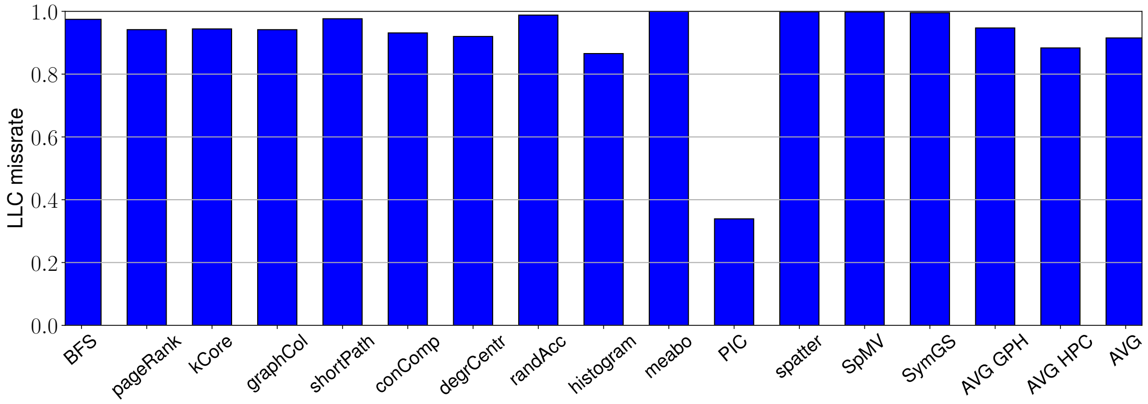


Figure 6.9: Missrate in the last-level cache. In *x*-axis multiple graph and HPC bechmarks, including average numbers.

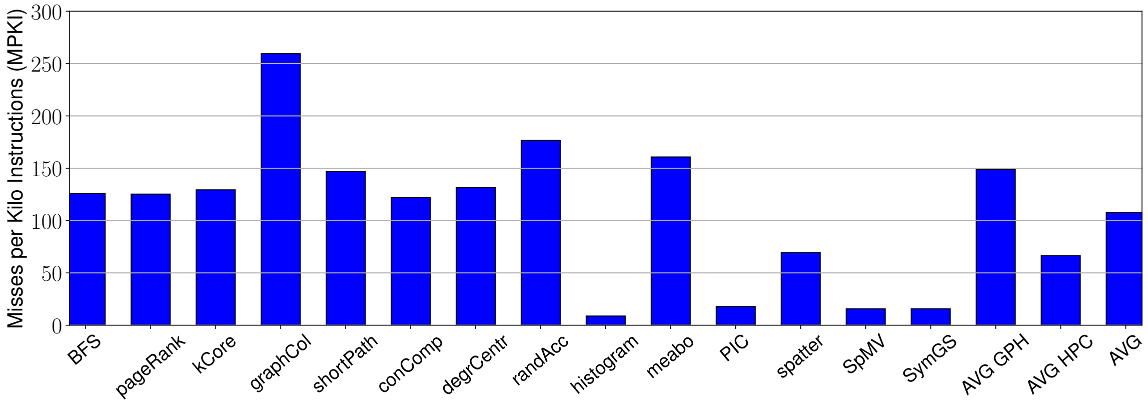


Figure 6.10: Misses Per Kilo Instruction (MPKI). In *x*-axis multiple graph and HPC bechmarks, including average numbers.

6.5.2 Results with REMOTE

Next, we show the results of executing the previously described benchmarks on REMOTE compared to the baseline.

On one hand, Figure 6.12 (up) shows the speed-ups of executing the applications either on the *host* cores or on the REMOTE devices. On average, the accelerators outperform the execution on 8 out-of-order cores when using more than 8 devices. 8 out-of-order cores obtain

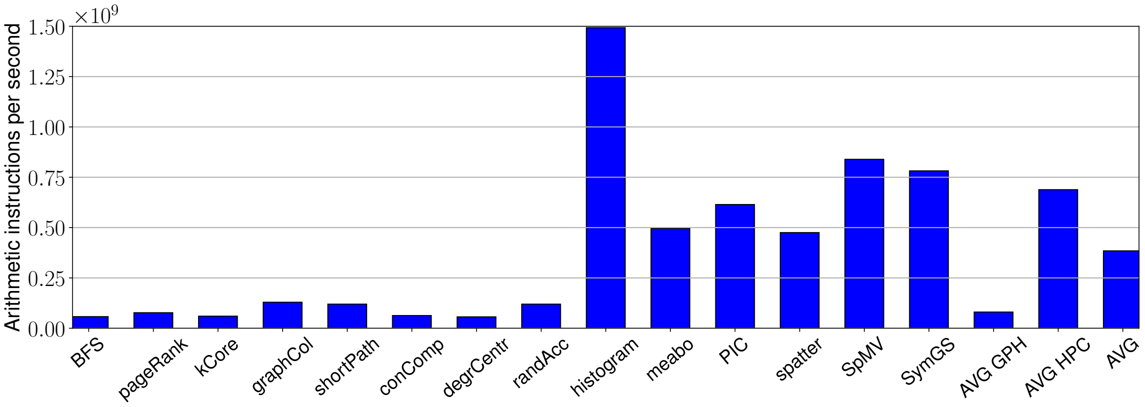


Figure 6.11: Arithmetic intensity. In *x*-axis multiple graph and HPC bechmarks, including average numbers.

a speed-up of 5.76×, while REMOTE obtains speed-ups of 6.33× and 9.77× with 16 and 32 devices, respectively. Therefore, in graph applications, REMOTE has a positive impact as it outperforms the baseline no matter the number of *host* cores and it does not get affected by scalability issues.

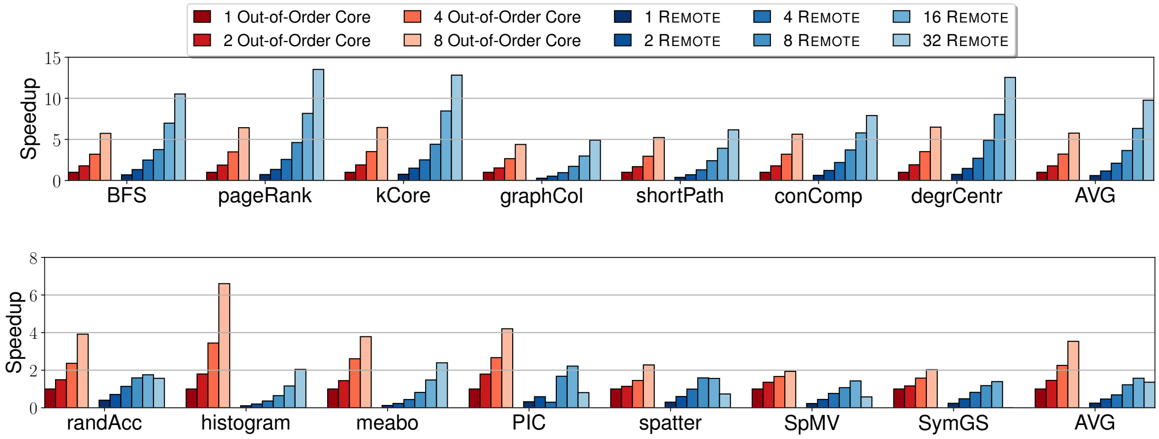


Figure 6.12: Speed-Up of graph (up) and HPC (down) benchmarks executed on the baseline and on REMOTE. Results normalized to 1 out-of-order core. In *x*-axis multiple graph benchmarks, including average numbers.

On the other hand, Figure 6.12 (down) shows the same results for HPC applications. In this case, the benchmarks contain more data structures that exhibit memory locality, and thus, they can take advantage from the cache hierarchy. Moreover, as depicted in Figure 6.11, HPC benchmarks have a higher arithmetic intensity than graph applications. In these situations, the complex out-of-order pipeline design from the *host* cores performs well, whereas the REMOTE devices have a simple in-order micro-architecture that cannot extract available ILP. On average, 8 REMOTE units outperform 1 *host* core by 22.4% and 16 devices outperform 2 *host* cores by 11.56%. With 32 REMOTE devices, scalability issues appear in several applications and 8 *host* cores perform better. Consequently, in HPC applications, REMOTE should only be used when there are not enough *host* cores and there are many accelerators available in the system. For instance, there are several combinations where REMOTE outperforms the baseline: in RandAcc, 4 devices are a 13.8% faster than 1 *host* core and 8 devices are a 9.63% faster than 2 *host* cores. In Spatter, 4 accelerators achieve the same performance as 1 *host* core and 8 devices are a 59% faster than 2 *host* cores. RandAcc and Spatter do not benefit from data locality in the cache hierarchy and that is the reason why the REMOTE outperforms the baseline. When it comes to the remaining HPC benchmarks, the baseline does benefit from data locality and mechanisms such as prefetching make it perform better than REMOTE. However, increasing the number of accelerators until a large value makes REMOTE outperform the baseline. For example, in Hist, 16 accelerators are a 15.9% faster than 1 *host* core and 32 are a 24.7% faster than 2. Despite the performance benefits, area and power constraints makes us select the baseline as a better candidate in these scenarios.

In terms of energy, Figure 6.13 shows the energy benefits when using the REMOTE devices. In particular, each bar contains dynamic numbers relative to 8 out-of-order *host* cores both for graph and HPC benchmarks.

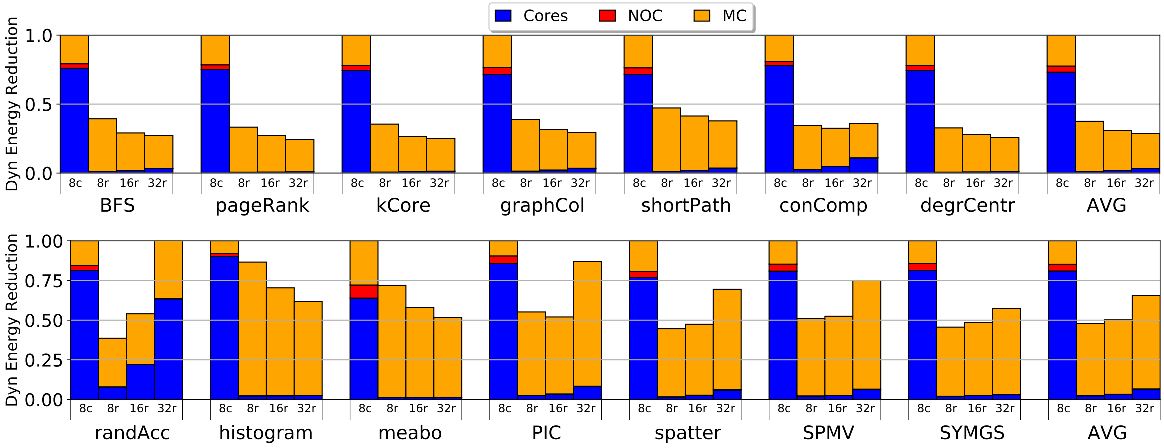


Figure 6.13: Relative dynamic energy of graph (up) and HPC (down) benchmarks executed on the baseline with 8 out-of-order cores ("8c") and on 8, 16 and 32 REMOTE ("8r", "16r", "32r"). Results normalized to 8 out-of-order cores. In *x*-axis several graph benchmarks with average numbers.

In graph applications the dynamic energy reductions are, on average, 62.5% for 8 devices, 69.06% for 16 devices and 71.1% for 32 devices compared to 8 *host* cores. These applications do not suffer in terms of performance scaling when increasing the number of accelerators, as shown in Figure 6.12 (up). In HPC benchmarks the dynamic energy reductions are, on average, 52% for 8 devices, 50% for 16 devices and 34.6% for 32 devices. In HPC applications, the scalability issues which appear with 16 and 32 devices in certain applications (e.g., randAcc, PIC in Figure 6.12 (down)), prevent higher dynamic energy reductions.

In graph and HPC benchmarks, energy reductions when using REMOTE configurations comes from employing simpler hardware to perform the computation, which in turn is more performant for graph applications. The NOC represents less than 1% of the total dynamic energy for all the applications. The core energy is also reduced. It represents the 80.98% of the total dynamic energy with 8 *host* cores and it is reduced to 10.03% with 32 REMOTE devices in HPC benchmarks. Finally, the memory controller represents most of the dynamic energy consumption, ranging from 88.8% to 95.11% overall in all the applications.

6.5.3 Impact to the Memory Hierarchy

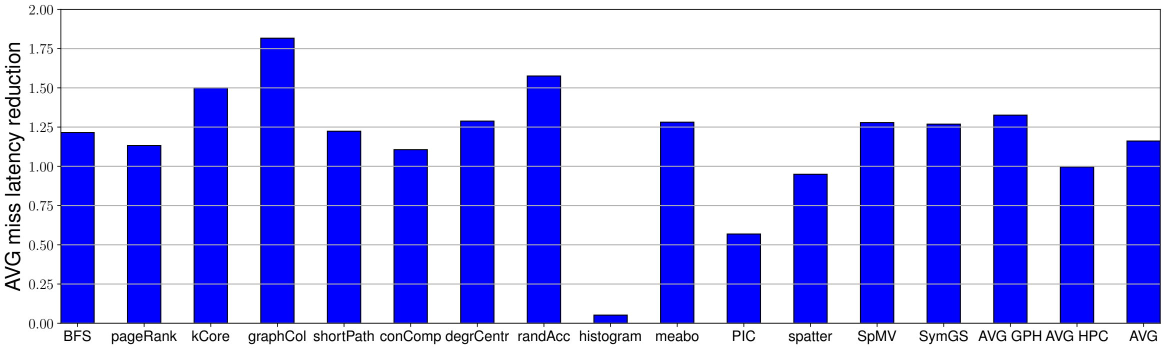


Figure 6.14: Average miss latency reduction, normalized to the baseline, when executing the applications on REMOTE. In *x*-axis multiple graph and HPC benchmarks, including average numbers.

Since REMOTE devices operate near memory, the number levels of the memory hierarchy that data has to traverse is reduced. In the case of the baseline, a cache block requested from memory traverses the memory controller, the L3, the L2 and L1D caches. When it comes to REMOTE, the L3 and the L2 are omitted, going from the memory controller directly into the L1D cache of the REMOTE device. As a result, the memory access time is reduced as the devices are closer to the data. Figure 6.14 shows the average memory access latency reduction seen either by the *host* core or the REMOTE devices, as they perform a memory request. On average, graph applications see a memory access latency reduction of 1.32× compared to the baseline, and HPC applications do not see any. This happens as some HPC benchmarks, such as RandAcc have a reduction of 1.57× and others, such as PIC, contain data access locality and take advantage of the deep memory hierarchy from the baseline. However, note that HPC applications that do reduce their average access latency, would still experience a cache miss ratio increase when using REMOTE devices.

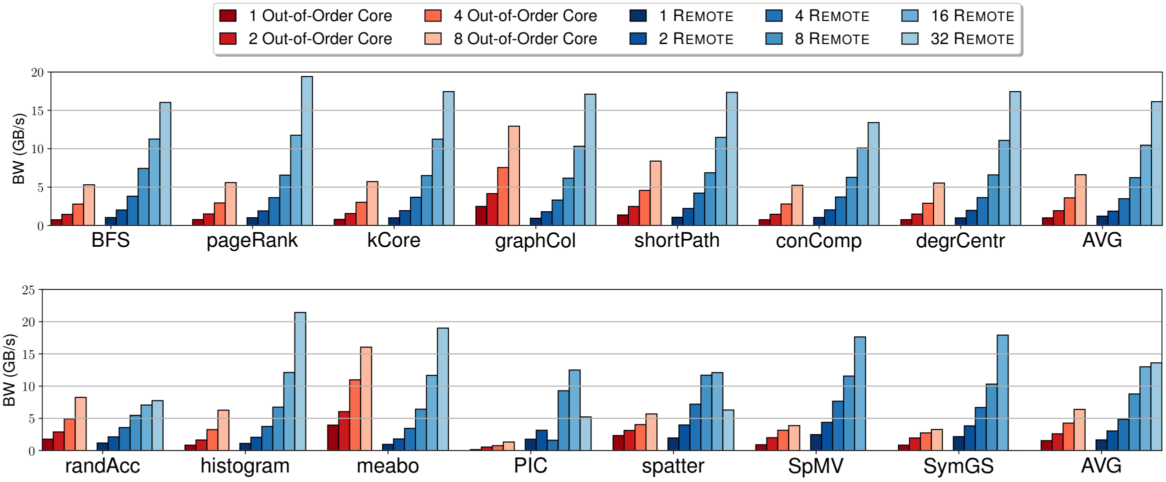


Figure 6.15: Memory bandwidth of graph (top) and HPC (bottom) benchmarks executed on the baseline and on REMOTE. Maximum theoretical bandwidth 32GB/s. In *x*-axis multiple graph benchmarks, including average numbers.

Figure 6.15 depicts the memory bandwidth utilization for graph and HPC applications. These values show that, in average, the baseline cannot exceed 6GB/s in both types of benchmarks while REMOTE reaches 16GB/s out of a theoretical peak of 32GB/s. Having more accelerators operating in parallel contributes to a higher exploitation of the memory bandwidth, ranging from 1.21GB/s with 1 device to 16.13GB/s with 32 devices in graph applications. These results correlate with the ones presented for performance, as a higher memory bandwidth utilization leads to higher speed-ups using REMOTE.

We may conclude that, despite the complex hardware design of the baseline which allows many simultaneous memory requests, the irregular memory accesses translate into an underutilized memory bandwidth and inefficient data movement on chip. Consequently, the near-memory design of the REMOTE devices tackles both limitations, providing benefits both in terms of performance and energy.

6.5.4 Host Core vs REMOTE Performance and Area Comparison

In this section, we compare the performance of executing several applications on the *host* cores with the performance obtained when executing the same benchmarks on the number of devices that fit in the same area as the *host* cores. As estimated in Section 6.3, the area of a single REMOTE device corresponds to the 26.64% of an out-of-order core. Approximately, 4 devices fit in the area of a single *host* core. For this reason, in Figure 6.16 we compare the performance of 16 *host* cores with 8 *host* cores + 32 REMOTE devices, which fit in the area of 16 out-of-order cores. We also show the results obtained when only the accelerators are employed, without intervention of the *host* cores.

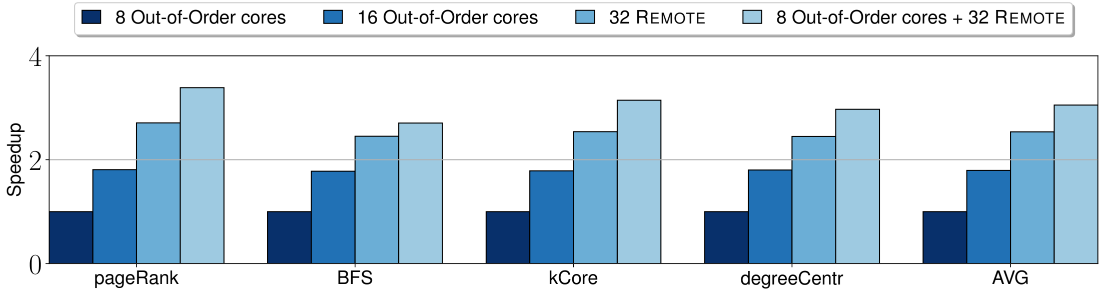


Figure 6.16: Relative performance results, normalized to 8 *host* cores. In *x*-axis multiple benchmarks, including average numbers.

Results demonstrate that using REMOTE accelerators leads to higher performance gains than employing *host* cores. In all the applications, replacing 8 *host* cores with 32 REMOTE devices provides higher benefits than using 16 *host* cores. On average, 16 *host* cores obtain 1.79× speed-ups, while 8 *host* cores + 32 REMOTE devices reaches a speed-up of 3.05×. In the event that the *host* cores are busy, using the 32 accelerators translates into 2.53× gains.

When it comes to energy, Figure 6.17 depicts the total energy reduction for all the configurations. On average, using 16 *host* cores represents an energy increase of 7.7%, while employing 8 *host* cores + 32 REMOTE devices obtains 63.75% energy reductions. In the case that only the accelerators are employed, energy reductions reach 90.12%.

6.5.5 Comparison to Other Proposals

In this section, we compare the performance of REMOTE with three other hardware proposals. In particular, the Smart Memory Cube (SMC) [21], the Programmable Prefetcher (PP) [3] and the bigLITTLE configuration [186].

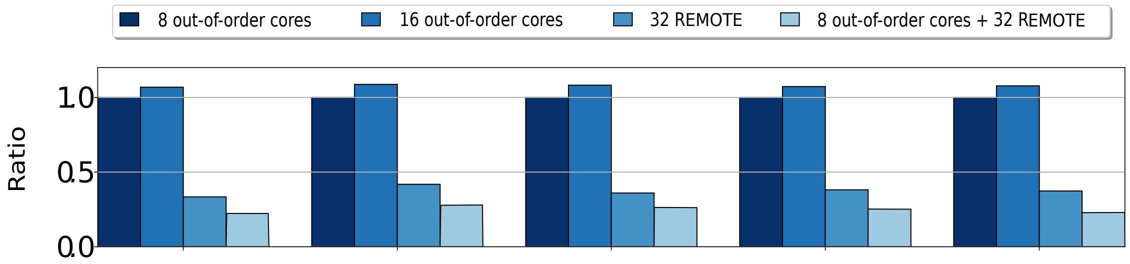


Figure 6.17: Total energy reduction results, normalized to 8 *host* cores. In *x*-axis multiple benchmarks, including average numbers.

Figure 6.18 presents the speed-up of the different proposals and the REMOTE approach with respect to a baseline with out-of-order cores using the PageRank benchmark while changing the graph connectivity. A higher graph connectivity implies a higher graph density, but the irregular memory accesses are still present in the execution.

On the one hand, results show that REMOTE is faster than the baseline beyond 4 devices as in Section 6.5.2. For instance, with 20% connectivity, 8 devices are 1.63× faster than 2 *host* cores, 16 devices are 1.44× faster than 4 *host* cores and 32 devices are 1.30× faster than 8 *host* cores. Moreover, results are more or less steady no matter the graph connectivity.

On other hand, the bigLITTLE configuration performs similarly to the baseline. It consists of the same number of out-of-order (big) cores but several in-order (LITTLE) cores are added. For instance, with 20% connectivity, 8 big cores with 16 LITTLE cores are 1.2% faster than 8 *host* cores. These LITTLE cores are more complex than the REMOTE accelerators, but they still suffer from the irregular memory accesses as they traverse the whole memory hierarchy. In bigLITTLE, the code scheduling represents a performance bottleneck in low graph connectivities but gets mitigated with higher graph densities. For example, employing 8 big cores plus 16 LITTLE cores the performance gains range from 3.82× to 6.02× with connectivities of 5% and 20%. These results prove that the bigLITTLE configuration provides negligible speed-ups and that it should only be considered with high graph densities.

When it comes to the PP, it only outperforms the baseline in low graph connectivities by 4%, and the results improve marginally with the number of prefetchers. Higher graph densities facilitate the predictions of the stride prefetcher employed in the baseline, and lead to performance degradations between 1 and 2% for the PP. In general, speed-ups are low due to the operation latency of the PP, which is summarized below.

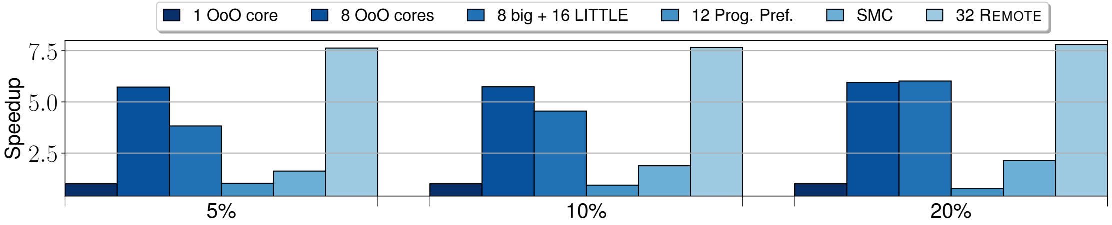


Figure 6.18: Comparison of the speed-up of three different proposals (the Smart Memory Cube [21], the Programmable Prefetcher [3] and the bigLITTLE configuration [186]) to the REMOTE approach using the PageRank benchmark. Results normalized to the performance of 1 out-of-order *host* core. In x-axis graph connectivities of 5%, 10% and 20%.

First, the PP examines the memory addresses of ongoing accesses to check whether the access corresponds to a data structure to be prefetched. If it does, the subroutine containing the code to make the prediction has to be assigned to a programmable prefetcher unit (PPU), which has to be awakened. Then, the PPU enqueues one or several prefetch requests, based on its prediction, and a translation for each is required. Moreover, the PP performs predictions based on prefetched addresses (e.g., to perform prefetches in the event of indirection vectors). The stride prefetcher is simpler, as it operates indexing with the PC of memory requests, not needing a specific subroutine for each data structure to be prefetched, tracking the memory access patterns and making a prediction.

In order to obtain higher speed-ups with the PP, a programming effort is needed to create a better data distribution. For example, in their paper [3] the authors use a different PageRank implementation which obtains higher speed-ups, but it is much more complex than ours and not easily adaptable to other proposals such as the SMC.

Finally, SMC consists of an in-order core integrated in the memory chip. It exploits better the memory bandwidth as it operates directly on it, reducing data movement on chip more than REMOTE does. The SMC is a 30% faster than 2 out-of-order cores and a 50.1% faster than 4 REMOTE devices. The synchronization between the *host* core and the SMC has to be managed by the operating system, introducing an operation latency, and taken into account by the programmer. Moreover, a flushing mechanism is required to keep data coherent between main memory and the *host* cores, which leads to important performance overheads. We may conclude that SMC is a good candidate to perform computation on memory, but it requires a special memory technology (i.e., 3D-stacking), more compute logic to outperform REMOTE and a larger programming effort.

6.6 Conclusions

Irregular memory accesses represent a challenge for current and future architectures. In this work, we present REMOTE, a programmable near-memory accelerator which performs computation outside of the memory hierarchy and closer to where data is located. Contrary to prior proposals, the design of REMOTE scales with multi-core systems and it does not require a specific memory technology (e.g., 3D-stacking). It targets the Programmability wall, by operating directly with virtual memory and by requiring simple pragma annotations to be added. Moreover, the runtime system schedules code to the accelerators depending on their availability.

In our evaluation, we study two different types of applications and demonstrate why they benefit more from REMOTE. As a result, in graph benchmarks REMOTE improves the 5.76× speed-up of 8 out-of-order cores reaching 6.33× with 16 devices and 9.76× with 32 devices. Other HPC applications outperform the baseline in particular configurations. Finally, we compare REMOTE to three existing hardware proposals, achieving 1.41×, 1.82× and 1.29× performance benefits.

Chapter 7

Conclusions and Future Work

This chapter summarizes the main conclusions and the contributions of this thesis and presents the future research lines opened by this work. Then it shows the list of publications produced during the realization of this thesis and acknowledges the financial support.

7.1 Conclusions

Vector architectures have become an important component of current processors in the form of SIMD extensions. They provide performance and energy benefits when it is possible to exploit data-level parallelism. However, they are affected by several issues, such as divergence control and irregular memory accesses. In current systems, these challenges remain unsolved and measures must be taken, as future processors will be further affected by these situations.

When it comes to the Memory Wall, memory access latency and data movement on chip lead to performance and energy degradations in the presence of irregular memory access patterns. Current solutions fail to successfully mitigate these problems, converting both issues in a challenge for the next generation of processors.

This thesis targets these challenges, proposing hardware/software solutions to better exploit the potential of vector architectures and to make a proper utilization of the memory hierarchy. Our hardware approaches can be combined with compile-time information, provided either by the programmer or the compiler, to utilize them whenever they may lead to performance benefits. Moreover, in our last contribution, the programmer may decide to delegate the responsibility of using these resources to the runtime system depending on internal decisions.

The first contribution of this thesis focuses on efficiently performing predication in SIMD extensions. We discovered that current implementations of predicated instructions have performance and energy consumptions dependent on the vector register length, and not on the number of active elements in the mask (mask density). For this reason, we propose a hardware approach that may be combined with compiler information, that achieves mask density-time performance and energy with minimal hardware support. Our evaluation shows that this contribution improves performance by up to 25% and reduces dynamic energy consumption by up to 43% on real unmodified applications with predicated execution. Our proposal will have a higher impact in next-generation processors, as the length of the vector registers doubles approximately every four years [86]. Moreover, this proposal allows executing unmodified legacy code with narrower vector instructions (AVX-2) on newer architectures with wider vectors (AVX-512), achieving up to 56% performance benefits.

The second contribution targets the issues derived from irregular memory accesses. Applications suffering from irregular memory accesses have a poor memory access locality, do not benefit from the memory hierarchy and have an inefficient vectorization, which is not considered by the compiler. Our proposal, called PLANAR, consists of a near-memory device, on the processor side and connected to the memory controller, that performs data-layout transformations converting sparse data structures into dense ones. These operations are non-blocking, as the *host* can compute and access memory while the device is operating, and fine-grained, as the accelerator synchronizes with the *host* as the dense structure is being populated, allowing it to be consumed. We demonstrate how this proposal reduces data movement on chip and enables an efficient code vectorization due to the new dense version of the sparse data. The hardware device, including a power and area estimations, and the software interface are described in detail in this document. As a result, PLANAR improves performance for single-threaded runs by 3.28× and 5.56×, and for multi-threaded executions by 4.58× and 5.71×, for scalar and compiler-vectorized codes, respectively. Finally, we compare PLANAR to two state-of-the-art proposals, achieving 2.12× and 2.23× average performance improvements.

The third contribution also targets the issues derived from irregular memory accesses. This approach called REMOTE operates near memory but, contrary to PLANAR, it does not require the *host* core to compute. REMOTE targets applications with irregular memory access patterns where the computation is so simple (e.g., single addition) that the operation could be directly done near memory rather than in the *host* core. REMOTE is programmed via pragmas and the runtime system is in charge of scheduling code to the accelerators. The REMOTE device directly operates with virtual memory and it does not require a specific memory technology (e.g., 3D-stacking). In our evaluation, we perform a profiling of the applications to see which perform better using REMOTE and a power and area estimation of the new hardware. As a result, REMOTE outperforms our baseline of 8 out-of-order cores by 1.57× with 16 devices and by 4× with 32 devices, on average, for graph applications. Finally, we compare REMOTE to three existing hardware proposals, achieving 1.41×, 1.82× and 1.29× performance benefits.

7.2 Future Work

The proposals presented in this thesis open the door to new research topics that could be explored in the future. Among others, three main research lines can be of great interest.

- *The Compaction/Restoration mechanism in multi-threaded systems.* The CR mechanism presented an evaluated in this thesis resorts to stalling the processor's pipeline in order to obtain CR candidates. Consequently, this translates into performance penalties and the saturation of the processor's internal resources, being both phenomenons currently tackled by means of timeouts. However, this limitation could be mitigated in multi-threaded systems. In such scenarios, CR could compact instructions with the same PC from several threads running the same program, avoiding the latency of waiting for CR candidates. In this contribution, we would explore the potential of CR in multi-threaded systems and propose new hardware/software techniques which would lead to higher performance and energy benefits.

- *The Compaction/Restoration mechanism with* PLANAR *to avoid horizontal operations.* The CR mechanism cannot be applied to horizontal predicated SIMD instructions. These instructions move the content of a vector register lane to a different one in the same register. Doing the compaction, the original lane position is lost, so CR cannot be utilized in these scenarios. However, we could employ PLANAR to reorganize data in memory so that horizontal instructions are not longer required in the processor and CR could be applied. In this contribution, we would study the situations where CR cannot be utilized and how our second proposal could alleviate this issue.

- *Runtime support to assist near-memory compute and DLT.* Our second and third proposals resort to a near-memory accelerator which can be used either to perform DLT or compute. In this contribution, we would combine both approaches and instruct the runtime system to determine when it is more suitable to perform each. It would require a deeper analysis of the applications and the new proposal would be software-oriented. This way, applications that suffer from irregular memory accesses and a high data movement on chip would be automatically benefited from having these near-memory accelerators, without programmer's intervention.

7.3 Publications

The contents of this thesis led to the following publications:

- (1) **A. Barredo**, A. Armejach, J. C. Beard and M. Moretó, "REMOTE: A Programmable Near-Memory Compute Engine", Currently under review.
- (2) **A. Barredo**, A. Armejach, J. C. Beard and M. Moretó, "PLANAR: A Programmable Accelerator for Near-Memory Data Rearrangement", 2021 International Conference on Supercomputing (ICS), Worldwide online event. To appear.
- (3) J. M. Cebrián, **A. Barredo**, H. Caminal, M. Moretó, M. Casas and M. Valero, "Semi-automatic Validation of Cycle-Accurate Simulation Infrastructures: The Case for gem5-x86", Future Generation Computer Systems, 2020.
- (4) **A. Barredo**, J. M. Cebrián, M. Moretó, M. Casas and M. Valero, "Improving Predication Efficiency through Compaction/Restoration of SIMD Instructions", IEEE International Symposium on High Performance Computer Architecture (HPCA), San Diego, CA, USA, 2020, pp. 717-728.
- (5) **A. Barredo**, J. C. Beard and M. Moretó, "SPiDRE: Accelerating Sparse Memory Access Patterns", Accepted as a poster paper in: 28th International Conference on Parallel Architectures and Compilation Techniques (PACT), Seattle, WA, USA, 2019, pp. 483-484.
- (6) **A. Barredo**, J. M. Cebrián, M. Moretó, M. Casas and M. Valero, "An Optimized Predication Execution for SIMD Extensions", Accepted as a poster paper in: 28th International Conference on Parallel Architectures and Compilation Techniques (PACT), Seattle, WA, USA, 2019, pp. 479-480.
- (7) **A. Barredo**, M. Moretó and J. C. Beard, "Hardware Acceleration of Sparse Data Rearrangement Near Memory", Accepted as a paper in: Arm Research Summit, Austin, TX, 2019.

The following publications are related but not included in this thesis:

- (1) J. M. Cebrián, A. Jimborean, **A. Barredo**, M. Casas, T. Balem, A. Ros and M. Moretó, "Compiler-Assisted Compaction/Restoration of SIMD Instructions", Currently under review.

- (2) J. Pavon, I. Vargas, **A. Barredo**, J. Marimon, M. Moretó, F. Moll, O. Unsal, M. Valero and A. Cristal, "VIA: A Smart Scratchpad for Vector Units With Application to Sparse Matrix Computations", IEEE International Symposium on High Performance Computer Architecture (HPCA), Seoul, South Korea, 2021.
- (3) **A. Barredo**, J. M. Cebrián, M. Moretó, M. Casas and M. Valero, "Efficiency analysis of modern vector architectures: vector ALU sizes, core counts and clock frequencies", The Journal of Supercomputing, 2019.
- (4) **A. Barredo**, J. M. Cebrián, M. Moretó, M. Casas and M. Valero, "Reconfigurable vector architectures", Accepted as a poster in: RoMoL project Final Workshop, Barcelona, Spain, 2018.
- (5) **A. Barredo**, J. M. Cebrián, M. Moretó, M. Casas and M. Valero, "Advanced Vector Architectures for Future Applications", Accepted as a paper in: 4th BSC Severo Ochoa Doctoral Symposium, Barcelona, Spain, 2017, pp. 73-75.

Bibliography

[1] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. “Compute Caches”. In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 481–492.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. “A scalable processing-in-memory accelerator for parallel graph processing”. In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. June 2015, pp. 105–117.

[3] S. Ainsworth and T. M. Jones. “An Event-Triggered Programmable Prefetcher for Irregular Workloads”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ASPLOS ’18. Williamsburg, VA, USA: Association for Computing Machinery, 2018, pp. 578–592. ISBN: 9781450349116.

[4] L. Alvarez, M. Moretó, M. Casas, E. Castillo, X. Martorell, J. Labarta, E. Ayguadé, and M. Valero. “Runtime-Guided Management of Scratchpad Memories in Multicore Architectures (PACT)”. In: *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. PACT ’15. 2015, pp. 379–391.

[5] L. Alvarez, L. Vilanova, M. Moreto, M. Casas, M. Gonzàlez, X. Martorell, N. Navarro, E. Ayguadé, and M. Valero. “Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures”. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. ISCA ’15. ACM, 2015, pp. 720–732. ISBN: 978-1-4503-3402-0.

[6] AMD. “3DNow! Technology Manual”. In: Motorola, 2000.

[7] AMD. *AMD EPYC 7101p*. 2020. URL: https://www.amd.com/en/products/cpu/amd-epyc-7401p (visited on 07/09/2020).

[8] N. Amit, M. Ben-Yehuda, and B.-A. Yassour. “IOMMU: Strategies for Mitigating the IOTLB Bottleneck”. In: *Proceedings of the 2010 International Conference on Computer Architecture (ISCA)*. ISCA’10. Saint-Malo, France: Springer-Verlag, 2010, pp. 256–274. ISBN: 9783642243219.

[9] Arm Limited. *The Thumb Instruction Set*. Accessed August 2020. 2005. URL: https://developer.arm.com/documentation/ddi0210/c/introduction/architecture/the-thumb-instruction-set.

[10] Arm Limited. *big.LITTLE Technology: The Future of Mobile*. White Paper. 2013.

[11] Arm Limited. “Arm Corex-A Series. Programmer’s Guide for Armv8-A”. In: 2015.

[12] Arm Limited. *Meabo*. Available at https://github.com/ARM-software/meabo. 2018.

[13] Arm Limited. *Arm Cortex-M0*. Accessed April 2019. URL: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0.

[14] A. Armejach, H. Caminal, J. Cebrian, R. Langarita, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, and M. Casas. “Using Arm’s scalable vector extension on stencil codes.” In: *The Journal of Supercomputing*. 2020, pp. 2039–2062.

[15] A. Armejach, H. Caminal, J. M. Cebrian, R. González-Alberquilla, C. Adeniyi-Jones, M. Valero, M. Casas, and M. Moretó. “Stencil codes on a vector length agnostic architecture”. In: *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018*. 2018, 13:1–13:12.

[16] A. Armejach, R. Titos-Gil, A. Negi, O. S. Unsal, and A. Cristal. “Techniques to Improve Performance in Requester-Wins Hardware Transactional Memory”. In: *ACM Trans. Archit. Code Optim.* 10.4 (Dec. 2013). ISSN: 1544-3566.

[17] *Arm NEON Technology*. Arm, Ltd. URL: https://developer.arm.com/technologies/neon.

[18] F. Arnaud, A. Thean, M. Eller, M. Lipinski, Y. Teh, M. Ostermayr, K. Kang, N. Kim, K. Ohuchi, J. Han, et al. “Competitive and cost effective high-k based 28nm CMOS technology for low power applications”. In: *IEEE International Electron Devices Meeting (IEDM)*. 2009.

[19] K. Asanović and J. Beck. *T0 Engineering Data*. 1997.

[20] K. Asanović. “Vector Microprocessors”. PhD thesis. 1998. ISBN: 0-591-99087-3.

[21] E. Azarkhish, D. Rossi, I. Loi, and L. Benini. “A Case for Near Memory Computation Inside the Smart Memory Cube”. In: *Workshop on Emerging Memory Solutions, DATE Conference 2016*. Dresden, Germany, 2016.

[22] C. Babbage. “On the Mathematical Powers of the Calculating Engine”. In: *The Origins of Digital Computers: Selected Papers*. Ed. by B. Randell. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 19–54. ISBN: 978-3-642-61812-3.

[23] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. “Near-data processing: Insights from a MICRO-46 Workshop”. In: *Micro* 34 (2014).

[24] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas. “CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories”. In: *ACM Trans. Archit. Code Optim.* 14.2 (June 2017), 14:1–14:25. ISSN: 1544-3566.

[25] M. Bargeron, T. Craver, M. Phlipot, M. Group, and I. Corp. “Applications Tuning for Streaming SIMD Extensions”. In: *Intel Technol J* Q2 (Sept. 2001).

[26] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. “The ILLIAC IV Computer”. In: *IEEE Transactions on Computers* C-17.8 (1968), pp. 746–757. ISSN: 0018-9340.

[27] A. Basumallik, S. Min, and R. Eigenmann. “Programming Distributed Memory Sytems Using OpenMP”. In: *2007 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2007, pp. 1–8.

[28] J. Beard and J. Randall. “Eliminating Dark Bandwidth: A Data-Centric View of Scalable, Efficient Performance, Post-Moore”. In: Oct. 2017, pp. 106–114. ISBN: 978-3-319-67629-6.

[29] J. C. Beard. “The Sparse Data Reduction Engine: Chopping Sparse Data One Byte at a Time”. In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS ’17. Alexandria, Virginia: Association for Computing Machinery, 2017, pp. 34–48. ISBN: 9781450353359.

[30] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. “The Gem5 Simulator”. In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964.

[31] S. Blackford. *Compressed row storage*. Accessed December 2019. 2000. URL: http://www.netlib.org/utk/people/JackDongarra/etemplates/node373.html.

[32] M. Bohr. "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper". In: *IEEE Solid-State Circuits Society Newsletter* 12.1 (2007), pp. 11–13.

[33] S. Borkar. "Role of interconnects in the future of computing". In: *Journal of Lightwave Technology* (2013).

[34] A. Buluç, S. Williams, L. Oliker, and J. Demmel. "Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication". In: *2011 IEEE International Parallel Distributed Processing Symposium (IPDPS)*. 2011, pp. 721–733.

[35] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks". In: *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA '09. Calgary, AB, Canada: Association for Computing Machinery, 2009, pp. 233–244. ISBN: 9781605586069.

[36] S. Byna, Y. Chen, and X. Sun. "A Taxonomy of Data Prefetching Mechanisms". In: *2008 International Symposium on Parallel Architectures, Algorithms, and Networks (i-span 2008)*. 2008, pp. 19–24.

[37] Cadence. *Genus Synthesis Solution*. Available at https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis.html. URL: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html.

[38] P. Caheny, L. Alvarez, S. Derradji, M. Valero, M. Moretó, and M. Casas. "Reducing Cache Coherence Traffic with a NUMA-Aware Runtime Approach". In: *IEEE Transactions on Parallel and Distributed Systems* 29.5 (2018), pp. 1174–1187. ISSN: 1045-9219.

[39] P. Caheny, L. Alvarez, M. Valero, M. Moretó, and M. Casas. "Runtime-assisted Cache Coherence Deactivation in Task Parallel Programs". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '18. 2018, 35:1–35:12.

[40] P. Caheny, M. Casas, M. Moretó, H. Gloaguen, M. Saintes, E. Ayguadé, J. Labarta, and M. Valero. “Reducing cache coherence traffic with hierarchical directory cache and NUMA-aware runtime scheduling”. In: *Proceedings of the 25th International Conference on Parallel Architecture and Compilation Techniques*. PACT ’16. 2016, pp. 275–286.

[41] D. Callahan, J. Dongarra, and D. Levine. “Vectorizing Compilers: A Test Suite and Results”. In: *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing (Supercomputing)*. Orlando, Florida, USA, 1988, pp. 98–105. ISBN: 0-8186-0882-X.

[42] D. Callahan, K. Kennedy, and A. Porterfield. “Software Prefetching”. In: vol. 19. Apr. 1991.

[43] M. Casas, M. Moretó, L. Alvarez, E. Castillo, D. Chasapis, T. Hayes, L. Jaulmes, O. Palomar, O. S. Unsal, A. Cristal, E. Ayguadé, J. Labarta, and M. Valero. “Runtime-Aware Architectures”. In: *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*. 2015, pp. 16–27.

[44] E. Castillo, M. Moreto, M. Casas, L. Alvarez, E. Vallejo, K. Chronaki, R. Badia, J. L. Bosque, R. Beivide, E. Ayguadé, J. Labarta, and M. Valero. “CATA: Criticality Aware Task Acceleration for Multicore Processors”. In: *Proceedings of the IEEE 30th International Parallel and Distributed Processing Symposium (IPDPS)*. IPDPS. 2016, pp. 413–422.

[45] J. M. Cebrian, M. Jahre, and L. Natvig. “ParVec: Vectorizing the PARSEC Benchmark Suite”. In: *Computing* (2015), pp. 1077–1100.

[46] B. S. Center. *BSC Application Repository*. 2020. URL: https://pm.bsc.es/projects/bar (visited on 07/09/2020).

[47] N. A. R. Center. *NAS Parallel Benchmarks*. 2020. URL: http://www.nas.nasa.gov/Software/NPB/ (visited on 07/09/2020).

[48] H. Chang, J. Cho, and W. Sung. “Performance Evaluation of an SIMD Architecture with a Multi-bank Vector Memory Unit”. In: *2006 IEEE Workshop on Signal Processing Systems Design and Implementation*. 2006, pp. 71–76.

[49] D. Chasapis, M. Moretó, M. Schulz, B. Rountree, M. Valero, and M. Casas. "Power Efficient Job Scheduling by Predicting the Impact of Processor Manufacturing Variability". In: *Proceedings of the ACM International Conference on Supercomputing (ICS)*. ICS '19. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 296–307. ISBN: 9781450360791.

[50] J. Chen, J. Benesty, and Y. Huang. "A Minimum Distortion Noise Reduction Algorithm With Multiple Microphones". In: *Audio, Speech, and Language Processing, IEEE Transactions on* 16 (Apr. 2008), pp. 481–493.

[51] K.-L. Cheng, C. Wu, Y. Wang, D.-W. Lin, C. Chu, Y. Tarng, S. Lu, S. Yang, M. Hsieh, C. Liu, et al. "A highly scaled, high performance 45 nm bulk logic CMOS technology with 0.242 μm2 SRAM cell". In: *2007 IEEE International Electron Devices Meeting*. 2007.

[52] J. Ciesko, S. Mateo, X. Teruel, X. Martorell, E. Ayguadé, and J. Labarta. "Supporting Adaptive Privatization Techniques for Irregular Array Reductions in Task-Parallel Programming Models". In: *OpenMP: Memory, Devices, and Tasks: 12th International Workshop on OpenMP, IWOMP 2016, Nara, Japan, October 5-7, 2016, Proceedings*. 2016, pp. 336–349.

[53] L. Clarke, I. Glendinning, and R. Hempel. "The MPI Message Passing Interface Standard". In: *Programming Environments for Massively Parallel Distributed Systems*. Ed. by K. M. Decker and R. M. Rehmann. Basel: Birkhäuser Basel, 1994, pp. 213–218. ISBN: 978-3-0348-8534-8.

[54] J. Corbal, R. Espasa, and M. Valero. "On the efficiency of reductions in μ-SIMD media extensions". In: *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2001, pp. 83–94.

[55] I. Cray Research. *Cray X-MP Series Model 48 Mainframe Reference Manual*. 1984.

[56] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J.-H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. "Merrimac: Supercomputing with Streams". In: *Supercomputing, 2003 ACM/IEEE Conference*. Nov. 2003, pp. 35–35.

[57] D. C. Daly, L. C. Fujino, and K. C. Smith. "Through the Looking Glass-The 2018 Edition: Trends in Solid-State Circuits from the 65th ISSCC". In: *IEEE Solid-State Circuits Magazine* (2018).

[58] T. A. Davis and Y. Hu. “The University of Florida Sparse Matrix Collection”. In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011). ISSN: 0098-3500.

[59] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. “Design of ion-implanted MOSFETs with very small physical dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[60] J. Doweck. “Inside Intel® Core microarchitecture”. In: Aug. 2006, pp. 1–35.

[61] G. Driss, A. Addaim, and A. M. Abdessalam. “Enhanced Box-Muller method for high quality Gaussian random number generation”. In: *International Journal of Computing Science and Mathematics* 9 (Jan. 2018), p. 287.

[62] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos. “The Mondrian Data Engine”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. ISCA ’17. Toronto, ON, Canada: Association for Computing Machinery, 2017, pp. 639–651. ISBN: 9781450348928.

[63] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. “OmpSs: A Proposal For Programming Heterogeneous Multi-Core Architectures”. In: *Parallel Processing Letters* 21.02 (2011), pp. 173–193.

[64] A. Elafrou, G. I. Goumas, and N. Koziris. “Performance Analysis and Optimization of Sparse Matrix-Vector Multiplication on Modern Multi- and Many-Core Processors”. In: *CoRR* (2017).

[65] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. “Tarantula: a vector extension to the alpha architecture”. In: *Proceedings 29th Annual International Symposium on Computer Architecture (ISCA)*. 2002, pp. 281–292.

[66] R. Espasa. *Advanced Vector Architectures. PhD. Thesis*. 1997.

[67] R. Espasa, M. Valero, and J. E. Smith. “Vector Architectures: Past, Present and Future”. In: *Proceedings of the 12th International Conference on Supercomputing (ICS)*. Melbourne, Australia, 1998, pp. 425–432. ISBN: 0-89791-998-X.

[68] B. Falsafi and T. F. Wenisch. “A Primer on Hardware Prefetching”. In: *A Primer on Hardware Prefetching*. 2014.

[69] A. Farmahini-Farahani, S. Gurumurthi, G. Loh, and M. Ignatowski. “Challenges of High-Capacity DRAM Stacks and Potential Directions”. In: *Proceedings of the Workshop on Memory Centric High Performance Computing*. 2018.

[70] M. J. Flynn. “Very high-speed computing systems”. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909.

[71] M. Flynn. “Flynn’s Taxonomy”. In: *Encyclopedia of Parallel Computing*. Ed. by D. Padua. Boston, MA: Springer US, 2011, pp. 689–697. ISBN: 978-0-387-09766-4.

[72] A. Fog. *Instruction Tables. Instruction latencies, throughputs and micro-operation breakdowns.* 2018. URL: http://www.agner.org/optimize/instruction_tables.pdf.

[73] G. Fuller. “Future lithography technology”. In: *Single Frequency Semiconductor Lasers*. 2017.

[74] S. Fuller. “Motorola AltiVec Technology”. In: Motorola, 1998.

[75] W. W. L. Fung and T. M. Aamodt. “Thread Block Compaction for Efficient SIMT Control Flow”. In: *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*. HPCA ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 25–36. ISBN: 978-1-4244-9432-3.

[76] E. Gedraite and M. Hadad. “Investigation on the effect of a Gaussian Blur in image filtering and segmentation”. In: Jan. 2011, pp. 393–396. ISBN: 978-1-61284-949-2.

[77] T. Geng, E. Diken, T. Wang, L. Jozwiak, and M. Herbordt. “An Access-Pattern-Aware On-Chip Vector Memory System with Automatic Loading for SIMD Architectures”. In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 2018, pp. 1–7.

[78] P. Getreuer. “A Survey of Gaussian Convolution Algorithms”. In: *Image Processing On Line* (2013).

[79] S. Ghose, K. Hsieh, A. Boroumand, R. Ausavarungnirun, and O. Mutlu. “The Processing-in-Memory Paradigm: Mechanisms to Enable Adoption”. In: *Beyond-CMOS Technologies for Next Generation Computer Design*. Ed. by R. O. Topaloglu and H.-S. P. Wong. Cham: Springer International Publishing, 2019, pp. 133–194. ISBN: 978-3-319-90385-9.

[80] S. Al-Ghuribi. “Matrix Multiplication Algorithms”. In: *International Journal of Computer Science and Network Security* (2012).

[81] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. “The NYU Ultracomputer-Designing an MIMD Shared Memory Parallel Computer”. In: *IEEE Transactions on Computers* C-32.2 (Feb. 1983), pp. 175–189.

[82] S. Habata, M. Yokokawa, and S. Kitawaki. “The Earth Simulator system”. In: *NEC Research and Development* 44 (Jan. 2003), pp. 21–26.

[83] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Vol. 5. Dec. 2010.

[84] A. Harrison and D. Joseph. “High Performance Rearrangement and Multiplication Routines for Sparse Tensor Arithmetic”. In: *SIAM Journal on Scientific Computing* (2018).

[85] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill. “MIPS: A microprocessor architecture”. In: *ACM SIGMICRO Newsletter* 13 (Jan. 1982), pp. 17–22.

[86] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. 6th. Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055, 9780128119051.

[87] M. Herlihy and J. E. B. Moss. “Transactional Memory: Architectural Support for Lock-Free Data Structures”. In: *SIGARCH Comput. Archit. News* 21.2 (May 1993), pp. 289–300. ISSN: 0163-5964.

[88] R. G. Hintz and D. P. Tare. “Control data star-100 processor design”. In: 1972.

[89] T. Hoang, A. Shambayati, and A. Chien. “A Data Layout Transformation (DLT) Accelerator: Architectural Support for Data Movement Optimization in Accelerated-centric Heterogeneous Systems”. In: 2016.

[90] K. Hsieh, E. Ebrahimi, G. Kim, N. Chatterjee, M. O’Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. “Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent near-Data Processing in GPU Systems”. In: *SIGARCH Comput. Archit. News* 44.3 (June 2016), pp. 204–216. ISSN: 0163-5964.

[91] J. N. Huber, O. R. Hernandez, and M. G. Lopez. “Effective Vectorization with OpenMP 4.5”. In: (Mar. 2017).

[92] C. J. Hughes. *Single-Instruction Multiple-Data Execution*. 2015.

[93] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 1: Basic Architecture.* 2012.

[94] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 2A: Instruction Set Reference.* 2015.

[95] Intel Corporation. *Intel Intrisics Guide*. 2020. URL: https://software.intel.com/sites/landingpage/IntrinsicsGuide/# (visited on 07/09/2020).

[96] Intel Corporation. *Intel Xeon E5-2630L v4*. 2020. URL: https://ark.intel.com/content/www/us/en/ark/products/92981/intel-xeon-processor-e5-2630-v4-25m-cache-2-20-ghz.html (visited on 07/09/2020).

[97] Intel Corporation. *Processor Intel® Xeon® Platinum 8160*. 2020. URL: https://ark.intel.com/content/www/es/es/ark/products/120501/intel-xeon-platinum-8160-processor-33m-cache-2-10-ghz.html (visited on 07/09/2020).

[98] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Jan. 2008. ISBN: 978-0-12-379751-3.

[99] A. Jain and C. Lin. "Linearizing Irregular Memory Accesses for Improved Correlated Prefetching". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. MICRO-46. Davis, California: Association for Computing Machinery, 2013, pp. 247–259. ISBN: 9781450326384.

[100] T. Jain and T. Agrawal. "The Haswell Microarchitecture - 4th Generation Processor". In: *International Journal of Computer Science and Information Technologies* 4(3) (Apr. 2013), pp. 477–480.

[101] L. Jaulmes, M. Casas, M. Moretó, E. Ayguadé, J. Labarta, and M. Valero. "Exploiting Asynchrony from Exact Forward Recovery for DUE in Iterative Solvers". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450337236.

[102] L. Jaulmes, M. Moreto, E. Ayguade, J. Labarta, M. Valero, and M. Casas. "Asynchronous and Exact Forward Recovery for Detected Errors in Iterative Solvers". In: *IEEE Transactions on Parallel and Distributed Systems* PP (Mar. 2018), pp. 1–1.

[103] L. Jaulmes, M. Moreto, M. Valero, and M. Casas. "A Vulnerability Factor for ECC-protected Memory". In: July 2019, pp. 176–181.

[104] JEDEC. *High Bandwidth Memory (HBM) DRAM*. Specification. JESD235C. Jan. 2020.

[105] S. Jeloka, N. Akesh, D. Sylvester, and D. Blaauw. "A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory". In: *IEEE Journal of Solid-State Circuits* 51 (Apr. 2016), pp. 1–1.

[106] W. Jin. *Feedback Compilation for Decoupled Access-Execute Techniques*. 2017.

[107] Jong Won Park. "Multiaccess memory system for attached SIMD computer". In: *IEEE Transactions on Computers* 53.4 (2004), pp. 439–452.

[108] M. Kang, E. Kim, M.-S. Keel, and N. Shanbhag. “Energy-efficient and high throughput sparse distributed memory architecture”. In: 2015 (July 2015), pp. 2505–2508.

[109] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. “An efficient k-means clustering algorithm: analysis and implementation”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 24.7 (2002), pp. 881–892.

[110] C. Kozyrakis and D. Patterson. “Overcoming the limitations of conventional vector processors”. In: *30th Annual International Symposium on Computer Architecture (ISCA), 2003. Proceedings.* 2003, pp. 399–409.

[111] C. Kozyrakis, D. Judd, J. Gebis, S. Williams, D. Patterson, and K. Yelick. “Hardwater/compiler codevelopment for an media processor”. In: *Proceedings of the IEEE* 89 (Dec. 2001), pp. 1694–1709.

[112] R. Krishnaiyer, E. Kultursay, P. Chawla, S. Preis, A. Zvezdin, and H. Saito. “Compiler-Based Data Prefetching and Streaming Non-temporal Store Generation for the Intel(R) Xeon Phi(TM) Coprocessor”. In: *2013 IEEE International Symposium on Parallel Distributed Processing (IPDPS), Workshops and Phd Forum*. 2013, pp. 1575–1586.

[113] R. Kumar, A. Martıénez, and A. González. “Vectorizing for Wider Vector Units in a HW/SW Co-designed Environment”. In: *10th IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, HPCC/EUC 2013, Zhangjiajie, China, November 13-15, 2013*. 2013, pp. 518–525.

[114] L. L. N. Laboratory. *CORAL-2 Benchmarks*. 2020. URL: https://asc.llnl.gov/coral-2-benchmarks/ (visited on 07/09/2020).

[115] P. N. N. Laboratory. *The PERFECT Suite*. 2020. URL: https://hpc.pnl.gov/PERFECT/ (visited on 07/09/2020).

[116] P. Lavin, E. J. Riedy, R. Vuduc, and J. Young. “Spatter: A Benchmark Suite for Evaluating Sparse Access Patterns”. In: *CoRR* (2018). arXiv: 1811.03743.

[117] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, J. H. Cho, K. H. Kwon, M. J. Kim, J. Lee, K. W. Park, B. Chung, and S. Hong. “25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV”. In: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC) 2014 IEEE International*. 2014, pp. 432–433.

[118] J. Lee, H. Kim, and R. W. Vuduc. “When Prefetching Works, When It Doesn’t, and Why”. In: *TACO* 9 (2012), 2:1–2:29.

[119] V. T. Lee, A. Mazumdar, C. C. del Mundo, A. Alaghi, L. Ceze, and M. Oskin. “Application Codesign of Near-Data Processing for Similarity Search”. In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pp. 896–907.

[120] Y. Lee, R. Avizienis, A. Bishara, R. Xia, D. Lockhart, C. Batten, and K. Asanović. “Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators”. In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 2011, pp. 129–140.

[121] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. “McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures”. In: *International Symposium on Microarchitecture (MICRO)*. New York, New York, 2009, pp. 469–480. ISBN: 978-1-60558-798-1.

[122] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao. “Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach”. In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 655–668.

[123] S. Lloyd and M. Gokhale. “In-Memory Data Rearrangement for Irregular, Data-Intensive Computing”. In: *Computer* (2015).

[124] E. Lockerman, A. Feldmann, M. Bakhshalipour, A. Stanescu, S. Gupta, D. Sanchez, and N. Beckmann. “Livia: Data-Centric Computing Throughout the Memory Hierarchy”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 417–433. ISBN: 9781450371025.

[125] C. Lomont. “Introduction to Intel Advanced Vector Extensions”. In: *Intel White Paper* (2011).

[126] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillón, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, M. Fariborz, A. F. Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, B. Hanindhito,

A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. S. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and É. F. Zulian. "The gem5 Simulator: Version 20.0+". In: *CoRR* abs/2007.03152 (2020). eprint: 2007.03152.

[127] H. T. Mair, G. Gammie, A. Wang, R. Lagerquist, C. Chung, S. Gururajarao, P. Kao, A. Rajagopalan, A. Saha, A. Jain, et al. "4.3 A 20nm 2.5 GHz ultra-low-power tri-cluster CPU subsystem with adaptive power allocation for optimal mobile SoC performance". In: *2016 IEEE International Solid-State Circuits Conference (ISSCC)*. 2016.

[128] F. H. McMahon. *The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range*. Tech. rep. UCRL-53745. Lawrence Livermore National Laboratory, Dec. 1986.

[129] J. Mellor-Crummey, D. Whalley, and K. Kennedy. "Improving memory hierarchy performance for irregular applications". In: *Proceedings of the 13th international conference on Supercomputing (ICS)*. 1999.

[130] Mentor. *Precision RTL Plus*. Available at https://www.mentor.com/products/fpga/synthesis/precision_rtl_plus/. URL: https://www.mentor.com/products/fpga/synthesis/precision_rtl_plus/.

[131] Mikhail. *SuperUser*. 2013. URL: https://superuser.com/questions/584900/how-distinguish-between-multicore-and-multiprocessor-systems.

[132] S. Mittal. "A survey of recent prefetching techniques for processor caches". In: *ACM Computing Surveys (CSUR)* (2016).

[133] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, W. Hoeppner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stehpany, and S. C. Thierauf. "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor". In: *IEEE Journal of Solid-State Circuits* 31.11 (1996), pp. 1703–1714.

[134] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0". In: *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. MICRO 40. 2007, pp. 3–14. ISBN: 0-7695-3047-8.

[135] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. *CACTI 6.0: A Tool to Understand Large Caches*. Tech. rep. 2009.

[136] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2017, pp. 457–468.

[137] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin. "GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450337236.

[138] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C. .-. Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. "Active Memory Cube: A processing-in-memory architecture for exascale systems". In: *IBM Journal of Research and Development* 59.2/3 (2015), 17:1–17:14.

[139] NEC. *Vector Supercomputer SX Series: SX-Aurora TSUBASA*. 2017. URL: http://www.nec.com/en/global/solutions/hpc.

[140] L. Oliker, A. Canning, J. Carter, J. Shalf, D. Skinner, E. Ethier, R. Biswas, J. Djomehri, and R. Van der Wijngaart. "Evaluation of cache-based superscalar and cacheless vector architectures for scientific computations". In: *SC'03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. 2003.

[141] OpenMP Architecture Review Board. *OpenMP Application Program Interface, v3.0*. 2008.

[142] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk. "Energy Efficient Architecture for Graph Analytics Accelerators". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 166–177.

[143] S. Palacharla, N. P. Jouppi, and J. E. Smith. "Complexity-Effective Superscalar Processors". In: *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*. ISCA '97. Denver, Colorado, USA: Association for Computing Machinery, 1997, pp. 206–218. ISBN: 0897919017.

[144] J. T. Pawlowski. "Hybrid memory cube (hmc)". In: *HOT CHIPS 23* (Aug. 2011).

[145] M. Peiron, M. Valero, E. Ayguadé, and T. Lang. "Vector Multiprocessors with Arbitrated Memory Access". In: *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*. ISCA '95. S. Margherita Ligure, Italy: Association for Computing Machinery, 1995, pp. 243–252. ISBN: 0897916980.

[146] G. Petrousis. "An Evaluation of Decoupled Access Execute on Armv8". MA thesis. Uppsala University, 2017.

[147] G. Pichon, M. Faverge, P. Ramet, and J. Roman. "Reordering Strategy for Blocking Optimization in Sparse Linear Solvers". In: *SIAM Journal on Matrix Analysis and Applications* (2017).

[148] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios. "K-Nearest Neighbors in Uncertain Graphs". In: *Proc. VLDB Endow.* 3.1–2 (Sept. 2010), pp. 997–1008. ISSN: 2150-8097.

[149] R. V. W. Putra, M. Hanif, and M. Shafique. *DRMap: A Generic DRAM Data Mapping Policy for Energy-Efficient Processing of Convolutional Neural Networks*. Apr. 2020.

[150] J. Reinders. "Intel Threading Building Blocks". In: 2007.

[151] G. Ren, P. Wu, and D. Padua. "A Preliminary Study on the Vectorization of Multimedia Applications for Multimedia Extensions". In: Oct. 2003, pp. 420–435.

[152] S. Report, J. Dongarra, and M. A. Heroux. "Toward a New Metric for Ranking High Performance Computing Systems". In: 2013.

[153] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. "Memory Access Scheduling". In: *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*. ISCA '00. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2000, pp. 128–138. ISBN: 1581132328.

[154] K. Rupp. *Microprocessor Trend Data*. https://github.com/karlrupp/microprocessor-trend-data. 2018.

[155] R. M. Russell. “The CRAY-1 Computer System”. In: *Commun. ACM* 21.1 (Jan. 1978), pp. 63–72. ISSN: 0001-0782.

[156] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke. “IBM Power9 Processor Architecture”. In: *IEEE Micro* 37.2 (2017), pp. 40–51.

[157] I. Sánchez Barrera, M. Casas, M. Moretó, E. Ayguadé, J. Labarta, and M. Valero. “Graph Partitioning Applied to DAG Scheduling to Reduce NUMA Effects”. In: *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP ’18. ACM, 2018, pp. 419–420. ISBN: 978-1-4503-4982-6.

[158] I. Sánchez Barrera, M. Moretó, E. Ayguadé, J. Labarta, M. Valero, and M. Casas. “Reducing Data Movement on Large Shared Memory Systems by Exploiting Computation Dependencies”. In: *Proceedings of the 2018 International Conference on Supercomputing (ICS)*. ICS ’18. ACM, 2018, pp. 207–217. ISBN: 978-1-4503-5783-8.

[159] A. Sandberg, N. Nikoleris, T. E. Carlson, E. Hagersten, S. Kaxiras, and D. Black-Schaffer. “Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed”. In: *2015 IEEE International Symposium on Workload Characterization*. 2015, pp. 183–192.

[160] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. “Can traditional programming bridge the Ninja performance gap for parallel computing applications?” In: *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 2012, pp. 440–451.

[161] W. Schönauer. “Scientific computing on vector computers”. In: *Special topics in supercomputing*. 1987.

[162] J. Sébot and N. Drach-Temam. “Memory Bandwidth: The True Bottleneck of SIMD Multimedia Performance on a Superscalar Processor”. In: May 2001.

[163] A. Seyedi, A. Armejach, A. Cristal, O. Unsal, I. Hur, and M. Valero. “Circuit design of a dual-versioning L1 data cache for optimistic concurrency”. In: Jan. 2011, pp. 325–330.

[164] J. Shi and J. M. F. Moura. *Graph Signal Processing: Modulation, Convolution, and Sampling*. 2019. eprint: 1912.06762.

[165] G. Singh, L. Chelini, S. Corda, A. J. Awan, S. Stuijk, R. Jordans, H. Corporaal, and A. Boonstra. “Near-Memory Computing: Past, Present, and Future”. In: *Microprocessors and Microsystems* (Aug. 2019).

[166] J. E. Smith, G. Faanes, and R. Sugumar. “Vector Instruction Set Support for Conditional Operations”. In: *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA)*. Vancouver, British Columbia, Canada, 2000, pp. 260–269. ISBN: 1-58113-232-8.

[167] A. Sodani. “Race to Exascale: Opportunities and Challenges”. In: Micro ’11 Keynote. 2011.

[168] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu. “Knights Landing: Second-Generation Intel Xeon Phi Product”. In: *IEEE Micro* 36.2 (2016), pp. 34–46.

[169] A. Sodani. “Knights landing (KNL): 2nd Generation Intel Xeon Phi processor”. In: *Hot Chips*. 2015.

[170] Y. Solihin, Jaejin Lee, and J. Torrellas. “Using a user-level memory thread for correlation prefetching”. In: *Proceedings 29th Annual International Symposium on Computer Architecture (ISCA)*. 2002, pp. 171–182.

[171] J. R. Spirn and P. J. Denning. “Experiments with program locality”. In: *Proc. of ACM Fall Joint Computer Conference, Part I*. 1972.

[172] J. R. Srinivasan. *Improving cache utilisation*. Tech. rep. University of Cambridge, Computer Laboratory, 2011.

[173] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. “The Arm Scalable Vector Extension”. In: *IEEE Micro* 37.2 (2017), pp. 26–39. ISSN: 0272-1732.

[174] S. Sur, M. J. Koop, and D. K. Panda. “High-Performance and Scalable MPI over InfiniBand with Reduced Memory Usage: An in-Depth Performance Analysis”. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC ’06. Tampa, Florida: Association for Computing Machinery, 2006, 105–es. ISBN: 0769527000.

[175] H. Sutter. “The Free Lunch Is Over A Fundamental Turn Toward Concurrency in Software”. In: 2013.

[176] J. K. Tanskanen, T. Sihvo, and J. Niittylahti. “Byte and modulo addressable parallel memory architecture for video coding”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 14.11 (2004), pp. 1270–1276.

[177] *The International Technology Roadmap For Semiconductors: Interconnect*. Tech. rep. 2009.

[178] C. Tomasi and R. Manduchi. “Bilateral filtering for gray and color images”. In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*. 1998, pp. 839–846.

[179] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. “EazyHTM: EAger-LaZY hardware Transactional Memory”. In: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2009, pp. 145–155.

[180] J. Torrellas. “FlexRAM: Toward an advanced Intelligent Memory system: A retrospective paper”. In: *2012 IEEE 30th International Conference on Computer Design (ICCD)*. 2012, pp. 3–4.

[181] K. Uchida and N. Kasuya. “FACOM Vector Processor”. In: (1983).

[182] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, et al. “Trends in data locality abstractions for HPC systems”. In: *IEEE Transactions on Parallel and Distributed Systems* (2017).

[183] A. S. Vaidya, A. Shayesteh, D. H. Woo, R. Saharoy, and M. Azimi. “SIMD Divergence Optimization Through Intra-warp Compaction”. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. Tel-Aviv, Israel, 2013, pp. 368–379. ISBN: 978-1-4503-2079-5.

[184] M. Valero, T. Lang, J. M. Llaberıéa, M. Peiron, E. Ayguadé, and J. J. Navarra. “Increasing the Number of Strides for Conflict-Free Vector Access”. In: *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*. ISCA ’92. Queensland, Australia: Association for Computing Machinery, 1992, pp. 372–381. ISBN: 0897915097.

[185] M. Valero, M. Moretó, M. Casas, E. Ayguade, and J. Labarta. “Runtime-Aware Architectures: A First Approach”. In: *Supercomputing frontiers and innovations* 1.1 (2014).

[186] E. Vasilakis, I. Sourdis, V. Papaefstathiou, A. Psathakis, and M. G. H. Katevenis. “Modeling energy-performance tradeoffs in Arm big.LITTLE architectures”. In: *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 2017, pp. 1–8.

[187] VentureBeat. *Intel confirms Ice Lake Core CPUs with 10nm+ process to followup its 8th-gen chips*. Available at https://venturebeat.com/2017/08/14/intel-confirms-ice-lake-core-cpus-with-10nm-process-to-followup-its-8th-gen-chips/. 2017. URL: https://venturebeat.com/2017/08/14/intel-confirms-ice-lake-core-cpus-with-10nm-process-to-followup-its-8th-gen-chips/.

[188] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. "Evaluation of Blue Gene/Q Hardware Support for Transactional Memories". In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. PACT '12. Minneapolis, Minnesota, USA: Association for Computing Machinery, 2012, pp. 127–136. ISBN: 9781450311823.

[189] T. Watanabe, T. Furukatsu, R. Kondo, T. Kawamura, and Y. Izutani. "The Supercomputer SX System: An Overview". In: *Proceedings of the Second International Conference on Supercomputing (ICS)*. 1987, pp. 51–56.

[190] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. "The RISC-v instruction set manual, volume I: Base user-level ISA". In: *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62* 116 (2011).

[191] W. J. Watson. "The TI ASC: A Highly Modular and Flexible Super Computer Architecture". In: *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I (AFIPS)*. Anaheim, California, 1972, pp. 221–228.

[192] M. Wei, M. Snir, J. Torrellas, R. B. Tremaine, T. M. Siebel, and N. Goodwin. "A Near-Memory Processor for Vector, Streaming and Bit Manipulation Workloads". In: 2005.

[193] J. Wiegert, G. Regnier, and J. Jackson. "Challenges for scalable networking in a virtualized server". In: *2007 16th International Conference on Computer Communications and Networks*. IEEE. 2007, pp. 179–184.

[194] M. J. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley, 1996.

[195] S.-Y. Wu, C. Lin, M. Chiang, J. Liaw, J. Cheng, S. Yang, M. Liang, T. Miyashita, C. Tsai, B. Hsu, et al. "A 16nm FinFET CMOS technology for mobile SoC and computing applications". In: *2013 IEEE International Electron Devices Meeting*. 2013.

[196] S.-Y. Wu, C. Lin, M. Chiang, J. Liaw, J. Cheng, S. Yang, C. Tsai, P. Chen, T. Miyashita, C. Chang, et al. “A 7nm CMOS platform technology featuring 4th generation FinFET transistors with a 0.027 μm2 high density 6-T SRAM cell for mobile SoC applications”. In: *2016 IEEE International Electron Devices Meeting (IEDM)*. 2016.

[197] W. A. Wulf and S. A. McKee. “Hitting the memory wall: implications of the obvious”. In: *ACM SIGARCH Computer Architecture News* 23.1 (Mar. 1995), pp. 20–24. ISSN: 0163-5964.

[198] S. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks. “Quantifying sources of error in McPAT and potential impacts on architectural studies”. In: *International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 577–589.

[199] R. Xie, P. Montanini, K. Akarvardar, N. Tripathi, B. Haran, S. Johnson, T. Hook, B. Hamieh, D. Corliss, J. Wang, et al. “A 7nm FinFET technology featuring EUV patterning and dual strained high mobility channels”. In: *2016 IEEE International Electron Devices Meeting (IEDM)*. 2016.

[200] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. “Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC ’13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789.

[201] T. Yoshida. “Introduction of Fujitsu’s HPC Processor for the Post-K Computer”. In: *Hot Chips*. 2016.

[202] X. Yu, C. J. Hughes, N. Satish, and S. Devadas. “IMP: Indirect memory prefetcher”. In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 178–190.

[203] T. Zeiser, G. Hager, and G. Wellein. “The world’s fastest CPU and SMP node: Some performance results from the NEC SX-9”. In: *2009 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. 2009, pp. 1–8.

[204] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. Greathouse, M. Meswani, M. Nutter, and M. Ignatowski. “A New Perspective on Processing-in-Memory Architecture Design”. In: *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*. MSPC ’13. Seattle, Washington: Association for Computing Machinery, 2013. ISBN: 9781450321037.

[205] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. "TOP-PIM: Throughput-Oriented Programmable Processing in Memory". In: *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 85–98. ISBN: 9781450327497.

[206] L. Zhang, Z. Fang, M. Parker, B. K. Mathew, L. Schaelicke, J. B. Carter, W. C. Hsieh, and S. A. McKee. "The Impulse Memory Controller". In: *IEEE Trans. Comput.* (2001). ISSN: 0018-9340.

[207] X. Zhuang and H.-H. Lee. "A hardware-based cache pollution filtering mechanism for aggressive prefetches". In: *2003 International Conference on Parallel Processing, 2003. Proceedings.* 2003.

List of figures

1.1 Historical trends of important metrics in computing systems. 2

1.2 Evolution of relative processor and memory performance. 3

2.1 Comparison of a scalar and vector instruction. 8

2.2 The micro-architecture of the VIRAM vector processor, from [111] 9

2.3 Comparison of a vertical (addition) and horizontal (reduction) instruction. . . 12

2.4 Predicated vector addition instruction. 14

2.5 Traditional computing architecture. 15

2.6 The memory hierarchy. 16

2.7 Typical memory hierarchy architectures 17

2.8 DRAM organization, obtained from [149]. 18

2.9 Abstract DRAM memory controller, obtained from [98]. 19

2.10 Processing In Memory concept. 21

2.11 High-level overview of a 3D-stacked DRAM based architecture, obtained from [79]. 23

2.12 Multi-core processor. Four cores are connected to the same L3 cache. Obtained from [131]. 24

2.13 Example of a distributed system with four processors, where every processor has a local memory. If they want to communicate, they need a message passing protocol. In this case, processor 0 sends a message to processor 1. 25

2.14 Example of shared memory system with four processors. If they need to synchronize, they use the shared memory. 26

3.1 Loop iteration breakdown. In the X axis, the applications name and their number of instructions per iteration. In the Y axis, the instruction type percentage in every iteration. 35

4.1 Performance and dynamic energy degradation for predicated SIMD applications with different mask densities. 43

4.2 CR basic functionality. In this case, two intructions for the same PC with 50% mask densities are compacted and restored. 45

4.3 Time diagram comparing the execution of baseline vs CR in the pipeline. The pipeline stages are: Fetch (F), Decode (DE), Issue (I), Dispatch (DI), Execute (E), Commit (C), Compact (CM), Restore (R). 45

4.4 CR overview when incorporated to an out-of-order processor. 48

4.5 Compaction phase for one compactable instruction with a mask density of 50%. 50

4.6 Restoration phase for one dense instruction containing two compacted instructions with a mask density of 50%. 52

4.7 Example of dense register forwarding. In the left, compaction is not needed, but it is on the right, as the mask for the second compactable instruction differs. 53

4.8 SIMD loop in Intel's assembly. 53

4.9 Example of the Compaction phase. 54

4.10 Example of Restoration phase. 55

4.11 Compaction unit configuration slowdown on performance. Normalized to non-latency CR scenario. In the *x*-axis the different number of stages. Each line represents a different compaction unit count. 57

4.12 Restoration unit configuration slowdown on performance. One two-stage compaction unit latency considered. Normalized to non-latency CR scenario. In the *x*-axis the different number of stages. Each line represents a different restoration unit count. 58

4.13 Timeout policy combinations impact on performance, normalized to the best scenario. The circular dependency policy is implicit in every scenario. In the *x*-axis the number of cycles for each timeout policy changes. 59

4.14 Impact of costly predicated SIMD instructions to performance (left) and dynamic energy (right). Normalized to the no-long latency instruction scenario. In the *x*-axis, the percentage of costly predicated instructions. 60

4.15 Average number of predicated instructions compacted per dense in CR. In the *x*-axis, the number of instructions per loop iteration. Masks: 25% (left) and 50% (right). 61

4.16 Performance (up left), VFU access (up right), dynamic energy (bottom left) reductions and leakage energy (bottom right) results of CR. Normalized to a non-CR scenario. 62
4.17 Results of AVX-2 legacy codes compacted into AVX-512 using CR. Normalized to a non-CR scenario. Speed-Up left, VFU access reduction (center) and dynamic energy reduction (right). 64
4.18 Speed-Up (left) and total energy reduction (right) of DIL, CR and CR+DIL normalized over a non-CR scenario. 65

5.1 Original STRIDE code. 69
5.2 Software-rearranged STRIDE code. 70
5.3 PLANAR-rearranged STRIDE code. 71
5.4 System overview with two PLANAR devices. Cores are augmented with a Rearrangement Control Table (*RCT*) to monitor ongoing rearrangements. . . 72
5.5 Synchronization example for a rearrangement that employs one PLANAR device. 78
5.6 Execution example for a rearrangement that employs one PLANAR device. . . 79
5.7 Normalized PLANAR design impact to performance in Spatter. Pipeline width (left), number of functional units (center) and L1-D cache size (right). In the *x*-axis the pipeline widths, number of functional units and cache size in *KB*. . 80
5.8 Performance relative to the number of PLANAR devices (*x*-axis), normalized to 64. 81
5.9 Performance with different synchronization chunk sizes normalized to 64B. . 82
5.10 Speed-Ups with eight PLANAR devices for one and eight core runs. Both normalized to *Baseline + Scalar*. 83
5.11 Average bandwidth usage at the memory controllers with eight PLANAR devices for one and eight cores. 84
5.12 L1D miss reduction with 8 PLANAR devices for one and eight cores, both normalized to baseline scalar. 86
5.13 Byte reduction in the L1D-L2 bus with 8 PLANAR for one and eight cores, normalized to baseline scalar. 87
5.14 Normalized DRAM accesses for baseline (B) and PLANAR (P) on 8 cores with scalar codes. 88
5.15 Dynamic energy reduction baseline (B) vs PLANAR (P) in scalar applications on eight cores. 89

5.16 Dynamic energy breakdown baseline (B) vs PLANAR (P) in scalar applications on eight cores. 89
5.17 Performance of Impulse vs PLANAR in SpMV. In the *x*-axis, the matrix inputs from Table 3.2.2. 90
5.18 Performance of the DLT accelerator vs PLANAR for multiple applications. . . 91

6.1 Original PageRank code. 95
6.2 REMOTE version of the PageRank code. 96
6.3 System overview with two REMOTE devices. 98
6.4 Nanos++ task flow over runtime structures with the changes to add REMOTE. 101
6.5 Impact of REMOTE's pipeline width (1 and 2) to performance. Results normalized to a pipeline of width one. In *x*-axis multiple graph and HPC bechmarks, including average numbers. 102
6.6 Impact of REMOTE's functional units (1, 2 and 4) to performance. Results normalized to 1 functional unit. In *x*-axis multiple graph and HPC bechmarks, including average numbers. 103
6.7 Impact of the latency of REMOTE's functional units (1×, 2× and 4×) to performance. Results normalized to 1× functional unit. In *x*-axis multiple graph and HPC bechmarks, including average numbers. 103
6.8 Impact of REMOTE's frequency (1, 2 and 4GHz) to performance. Results normalized to 1GHz. In *x*-axis multiple graph and HPC bechmarks, including average numbers. 104
6.9 Missrate in the last-level cache. In *x*-axis multiple graph and HPC bechmarks, including average numbers. 105
6.10 Misses Per Kilo Instruction (MPKI). In *x*-axis multiple graph and HPC bechmarks, including average numbers. 105
6.11 Arithmetic intensity. In *x*-axis multiple graph and HPC bechmarks, including average numbers. 106
6.12 Speed-Up of graph (up) and HPC (down) benchmarks executed on the baseline and on REMOTE. Results normalized to 1 out-of-order core. In *x*-axis multiple graph benchmarks, including average numbers. 106
6.13 Relative dynamic energy of graph (up) and HPC (down) benchmarks executed on the baseline with 8 out-of-order cores ("8c") and on 8, 16 and 32 REMOTE ("8r", "16r", "32r"). Results normalized to 8 out-of-order cores. In *x*-axis several graph benchmarks with average numbers. 107

6.14 Average miss latency reduction, normalized to the baseline, when executing the applications on REMOTE. In *x*-axis multiple graph and HPC benchmarks, including average numbers. 108

6.15 Memory bandwidth of graph (top) and HPC (bottom) benchmarks executed on the baseline and on REMOTE. Maximum theoretical bandwidth 32GB/s. In *x*-axis multiple graph benchmarks, including average numbers. 109

6.16 Relative performance results, normalized to 8 *host* cores. In *x*-axis multiple benchmarks, including average numbers. 110

6.17 Total energy reduction results, normalized to 8 *host* cores. In *x*-axis multiple benchmarks, including average numbers. 111

6.18 Comparison of the speed-up of three different proposals (the Smart Memory Cube [21], the Programmable Prefetcher [3] and the bigLITTLE configuration [186]) to the REMOTE approach using the PageRank benchmark. Results normalized to the performance of 1 out-of-order *host* core. In *x*-axis graph connectivities of 5%, 10% and 20%. 112

List of tables

3.1 Configuration of *gem5* simulations for the first proposal. 31
3.2 Configuration of *gem5* simulations for the second proposal. 32
3.3 Configuration of *gem5* simulations for the third proposal. 33
3.4 Benchmarks used to evaluate the proposal about divergence. 35
3.5 Benchmarks used to evaluate the proposal about data layout transformation. . 37
3.6 Description of the benchmarks used to evaluate the near-memory compute proposal. 38

4.1 CIT entry fields, size in bits. 46

5.1 Comparison with state-of-the-art DLT proposals. 68
5.2 Rearrangement Control Table (*RCT*) with the information about a rearrangement in flight being performed by 4 PLANAR devices. 73

6.1 Comparison with state-of-the-art proposals. 97