On the co-design of scientific applications and long vector architectures

Constantino Gómez

Barcelona, 21 Mar, 2022

Advisors: **Filippo Mantovani,**
Marc Casas Guix

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

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

The landscape of High Performance Computing (HPC) system architectures keeps expanding with new technologies and increased complexity. To improve the efficiency of next-generation compute devices, architects are looking for solutions beyond the commodity CPU approach. In 2021, the five most powerful supercomputers in the world [82] use either GP-GPU (General-purpose computing on graphics processing units) accelerators or a customized CPU specially designed to target HPC applications. This trend is only expected to grow in the next years motivated by the compute demands of science and industry.

As architectures evolve, the ecosystem of tools and applications must follow. The choices in the number of cores in a socket, the floating point-units per core and the bandwidth through the memory hierarchy among others, have a large impact in the power consumption and compute capabilities of the devices. To balance CPU and accelerators, designers require accurate tools for analyzing and predicting the impact of new architectural features on the performance of complex scientific applications at scale. In such a large design space, capturing and modeling with simulators the complex interactions between the system software and hardware components is a defying challenge. Moreover, applications must be able to exploit those designs with aggressive compute capabilities and memory bandwidth configurations. Algorithms and data structures will need to be redesigned accordingly to expose a high degree of data-level parallelism allowing them to scale in large systems. Therefore, next-generation computing devices will be the result of a co-design effort in hardware and applications supported by advanced simulation tools.

In this thesis, we focus our work on the co-design of scientific applications and long vector architectures. We significantly extend a multi-scale simulation toolchain enabling accurate performance and power estimations of large-scale HPC systems. Through simulation, we explore the large design space in current HPC trends over a wide range of applications. We extract speedup and energy consumption figures analyzing the trade-offs and optimal configurations for each of the applications. We describe in detail the optimization process of two challenging applications on real vector accelerators, achieving outstanding operation performance and full memory bandwidth utilization. Overall, we provide evidence-based architectural and programming recommendations that will serve as hardware and software co-design guidelines for the next generation of specialized compute devices.

Keywords

HPC, Co-design, Parallelism, OpenMP, MPI, Next-Generation Architectures, SpMV, NEC Vector Engine, Long-Vector Architectures, Performance Optimization, HPCG, RISC-V

Acknowledgements

This thesis would have never been possible without the help of many friends and colleagues. My words in this section go to them.

First and foremost, I would like to thank my advisors Filippo Mantovani and Marc Casas for their guidance through my PhD. For being patient with me, for all the support and doing everything in their hand to bring this thesis to a good end. To Erich Focht, for the invaluable contributions to this thesis. I can not be more grateful for dedicating so much of his time to share his vast knowledge on linear algebra and vector architectures with me. To César and Francesc for bringing their best engineering work to the simulators.

I also would like to thank my pre-defense committee, Jaume Abella, Oscar Palomar and Borja Pérez, and to the external reviewers, Alexandra Jimborean and Estela Suárez, for providing valuable comments helping to improve this thesis.

Thanks to all the research colleagues with whom I shared this endeavor. To those who *Rode the Moore's Law* with me, specially to Adrián, Calvin, Luc, Xubin, Isaac and Vladimir. For the long talks, the laughs and good times together, for the help and support when I needed it. I will keep fond memories of those moments for the years to come. To the *Cache Miss* gang, Cristóbal Ortega, Pedro Benedicte, David Trilla and Albert Segura, whom accompanied me during these 12 years at UPC, for everything we shared in and out of the campus.

A mis amigos más cercanos, a mi hermano y mis padres, por estar siempre ahí, por apoyarme tanto, pase lo que pase. Os quiero.

A todos vosotros, de nuevo, gracias.

Declaration

I declare that the work contained in this thesis has not been submitted for any other degree or professional qualification except as specified. Some of the proposed techniques and results presented in this thesis has been published in the following papers:

- "Design Space Exploration of Next-Generation HPC Machines"
  Constantino Gómez, Francesc Martínez, Adrià Armejach, Miquel Moretó, Filippo Mantovani, Marc Casas
  International Parallel and Distributed Processing Symposium, May 2019 (IPDPS '19)
  DOI: 10.1109/IPDPS.2019.00017

- "Efficiently running SpMV on long vector architectures"
  Constantino Gómez, Filippo Mantovani, Erich Focht, Marc Casas
  Symposium on Principles and Practice of Parallel Programming, February 2021 (PPoPP '21)
  DOI: 10.1145/3437801.3441592

- "Optimizing HPCG using the new RISC-V Vector Extension"
  Constantino Gómez, Filippo Mantovani, Erich Focht, Marc Casas
  This work has been submitted for publication.

Constantino Gómez Crespo

Table of contents

List of figures

List of tables

Chapter 1

Introduction

There is a continued demand in industry and science for higher and efficient computing capabilities. To satisfy not only the performance demands but also the energy consumption and cost of production constraints, the landscape of High-Performance Computing system architectures keeps expanding with new technologies and increased complexity. The use of commodity server-grade processors as the common choice to design these systems is moving into a more specialized landscape. Processor trends are evolving in different directions, such as, leaner core designs [33], larger core counts per socket [81], wide vector units [78], or with integrated memory like high-bandwidth memory (HBM) modules via silicon interposer technologies [23]. Moreover, other trends in the semiconductor industry such as the blossoming of the IP (Intelectual Property) licensing model, which helps to reduce the costs and time-to-market of new chip designs, and *pure-play* foundries, capable of offering the best CMOS technology, are also favoring the appearance of new companies selling custom computing products in competitive markets. Consequently, the design space for next-generation HPC machines is expanding.

In this quest for efficiency, accelerated computing is becoming more and more relevant. The limitation to the performance improvements imposed by the slow-down of Moore's law applied to general purpose CPUs has made HPC providers looking for solutions that can complement the computational power beyond the use of CPUs. The convenient programability of general purpose CPUs comes at the cost of a significant energy overhead on each operation. Figure 1.1 inspired by the study of Horowitz in [36], shows the energy cost break down of a floating-point add operation in a 45 nm process CPU with scalar instructions. Of the total 70 pJ spent in the CPU executing the instruction, only a 1.4% of energy is used to perform the addition in the floating-point unit. The rest is distributed between fetching the instruction from the instruction cache, accessing the register file and activating the control logic. On top of that, the energy cost

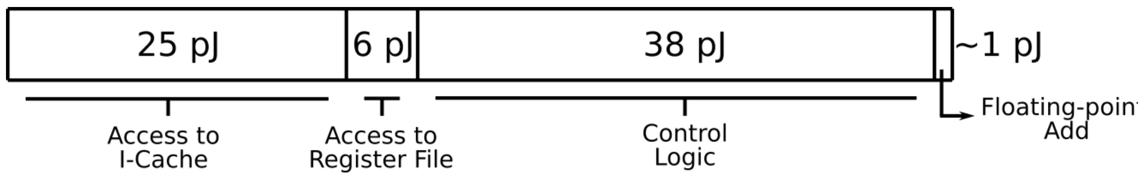


Figure 1.1: Energy cost break down of executing a scalar Floating-Point Add intructions in a 45 nm CPU.

of moving data from main memory into the register file is significantly higher than the the cost of arithmetic instructions. Therefore, it is possible to achieve large improvements in energy efficiency by increasing the effective computation per instruction. By balancing the architecture to the needs of a particular set of applications, offering an optimized memory hierarchy and special purpose instructions, accelerators attain, with the same power budget, several orders of magnitude more performance than CPUs.

The most visible example of accelerators in HPC are GP-GPU based systems, that populate 3 places within the first 5 most powerful supercomputers in the world (Top500, November 2021 list [83]). GP-GPUs however, are not the only approach to acceleration: the use of vector or SIMD (Single Instruction Multiple Data) extensions are also very common options in HPC systems. Vector and SIMD architectures are capable, with a single instruction, of performing operations on vectors instead of single scalar values. This approach helps mitigate the excessive control energy overhead exposed in the previous figure. Beside the AVX-512 [58] SIMD extension by Intel, we can find on the market the first CPU implementing the Arm SVE extension (Fujitsu A64FX, ranked first in the Top500 as of November 2021) and the NEC SX-Aurora vector engine (referred as VE from now on), a discrete accelerator leveraging vector CPUs able to operate with registers of up to 256 double-precision elements. It should not either be ignored the RISC-V architecture which recently ratified v1.0 of the V-extension [72], boosting vector computation from the academic world and the open-source community.

Understanding how HPC applications perform under such different design points is crucial to determine the best architecture for computing clusters. In that regard, microarchitecture simulators [59, 29, 74, 42] become an essential tool allowing architects to test the behaviour of applications running in a certain system that has yet to be built. However, capturing and modeling the complex interactions between the hardware components present in the next generation of HPC systems does not come without challenge. They must be fast, accurate and flexible. There is a trade-off between accuracy and speed in simulators, as modeling large multicore nodes with variety of hardware components has a very high computational cost. To alleviate that, simulators may apply techniques to leverage certain characteristics of the workloads. For example, sampling exploits the regularity in execution patterns of HPC workloads and reduces simulation time by modeling only a fraction of the full stream of instructions then extrapolating the rest [30]. While it is not expected for any simulator to be all-in-one solution, they must be flexible enough to interface with other state-of-the-art models and simulators. If possible, they should integrate the performance results with power dissipation, energy consumption and area figures. The desired goal is a toolchain capable of providing insightful information used by architects in the design of new hardware.

Leveraging the full potential of new architectures in applications does not come without a challenge. For example, in simple dense linear algebra codes (e.g., dot product), vector architectures may obtain straightforward benefits in performance with a trivial loop vectorization. But porting complex routines to run fast in vector accelerators often requires profound changes to algorithms and data structures with respect to prior general-purpose CPU implementations. Such profound changes are usually beyond the reach of compilers. Programmers often rely on manual tuning of the codes and the use of low-level intrinsics. The effort applied into designing specialized hardware to deliver high performance in demanding scientific applications, should be also applied in return, to rewrite such codes to make efficient use of the new architectural features. Given the wide variety of kernels and applications with different computational characteristics, a significant research effort is needed in the topic.

Developing fast next-generation systems will not be possible without a tightly coupled **hardware** and **software** optimization process supported by advanced **simulation tools**. In this thesis, we approach the co-design challenge of long vector architectures and HPC applications considering those three aspects. We contribute to the hardware design effort by exploring the performance efficiency trade-offs of the main state-of-the-art hardware components in high-performance systems. We extend the simulation infrastructure as needed to capture the latest compute trends and complex interactions between the application, the runtime system and hardware components. As a result of our simulation experiments, we identify the critical aspects of a compute system for performance and provide architectural design recommendations to build efficient vector accelerators, especially for memory-intensive applications, in the field of HPC. On the software side, we dive into the optimization process of challenging kernels and applications targeting long vector architectures. Finally, we provide thorough details about the performance impact of each optimization, and release open-source implementations for two of the current most relevant vector architectures for benefit of the scientific community.

1.1 Thesis Objectives and Contributions

This thesis brings the following contributions in the area of co-design of long vector architectures and HPC applications.

1.1.1 Design space exploration of next-generation HPC architectures

The first contribution of this thesis is to provide a set of hardware and software co-design recommendations for next-generation large-scale HPC systems. We undertake an extensive simulation-based design space exploration that considers the most relevant design trends we are observing in current HPC systems, providing estimations of performance, power and energy consumption. The goal of this work is to understand the individual impact of each of the multiple layers that compose the current complex HPC architectures.

To develop this study, we follow a recently introduced multi-level simulation methodology (MUSA) [29]. MUSA enables fast and accurate performance estimations and takes into account inter-node communication, node-level architecture, and system software interactions. We extend MUSA in multiple ways to perform a detailed evaluation of the impact of a set of identified key design trends on relevant HPC applications. In particular, we add support for a larger set of OpenMP pragmas, multiple target architectures, a model for vectorization, performance and power modeling of emerging memory technologies, and processor power estimations. The resulting infrastructure can target a wide range of HPC applications, and provide performance and power estimations of the trends that are currently dominating in HPC.

After extending MUSA, we perform large-scale simulations comprised of 256 MPI ranks, each representing a compute node, and up to 64 cores per node adding a total of 16,384 cores. Our design space exploration methodology analyzes current HPC trends by factoring in the relevant architectural parameters, for which we derive over 800 different architectural configurations. We simulate each of the 5 selected representative HPC applications with these configurations and provide estimations for performance, power and energy-to-solution.

1.1.2 Optimizing SpMV on Long Vector Architectures

As a second contribution, we extend the state-of-the-art SELL-C-σ sparse matrix format by introducing a set of new optimizations. The goal of this contribution is to produce a highly efficient SpMV implementation targeting long vector architectures.

Previous work proposals like SELL-C-σ [48] and ELLPACK Sparse Block [54] matrix formats improve both storage requirements and locality when running SpMV on SIMD CPUs. Our work demonstrates that, although some of these approaches are very good abstractions to represent and manipulate sparse matrices, there are many unexploited opportunities to further improve their performance on long vector architectures. Focusing on that, we develop, evaluate and discuss the performance consequences of several SpMV optimizations using the SELL-C-σ on long vector architectures. As a result, we improve the SELL-C-σ baseline performance by 12% on average.

Our implementation reaches 117 GFlops saturating 78% of the peak memory bandwidth in a NEC SX-Aurora Vector Engine (VE) [46]. This accelerator leverages a vector instruction set that goes beyond the SIMD approach (e.g., AVX-512 in x86 CPUs) with 16-kbit long vector registers and instructions. This is the first time that the SELL-C-σ format is implemented and evaluated on an architecture leveraging long vectors and predication. Our implementation shows how to deal with sparse data structures on emerging vector ISAs such as the Arm SVE or the RISC-V vector extension. In addition, we compare our novel approach for long vector architectures with other state-of-the-art approaches targeting SpMV in cutting-edge multi-core CPU and GPU devices. We demonstrate that our approach is **3.02× and 1.72×** faster, respectively, than these approaches since it achieves an outstanding average SpMV efficiency equivalent to 4.19% of the peak performance.

1.1.3 Efficient HPCG vectorization

For our third contribution, we push further in our effort to improve efficiency of scientific applications in long vector architectures. In this case, we focus on the optimization of HPCG on two emerging vector architectures leveraging large vector register size: the VE by NEC and a RISC-V accelerator implementing the 'V' vector extension (RISC-VV from now on).

First, we revisit previous HPCG vectorization proposals to develop a new open implementation using public vector APIs. Leveraging the learnings from our previous contribution, we apply unrolling and low-locality management techniques to the SpMV and Symmetric Gauss-Seidel numerical kernels. Second, we discuss our experiments in real and simulated accelerators obtaining individual performance estimations for each optimization in two accelerator designs featuring a vector processing unit (VPU from now on) of 512-bit and another with 2048-bit. Finally, aiming to understand the design flexibility introduced by having a detached VPU and architectural vector lengths, we employ a RISC-VV simulator to quantify the benefit of such vectorized and optimized version of HPCG for different vector lengths.

1.2 Thesis Outline

The contents of this thesis are organized as follows:

In Chapter 2 we review the prior work on the topics related to this dissertation, beginning with the background on HPC trends, vector accelerators and architectural simulator challenges. Further, we explore the different sparse matrix formats for efficient SpMV computation. Finally, we introduce the HPCG benchmark and provide an algorithm description followed by a breakdown of all the proposals from previous work on the optimization of HPCG for long vector architectures. In Chapter 3 we present a large-scale design space exploration study that considers the most relevant design trends we are observing today in HPC systems. In it, we simulate five hybrid (MPI+OpenMP) applications over 864 architectural proposals based on state-of-the-art and emerging HPC technologies and provide a detailed performance and power trade-off analysis quantifying the individual impact of hardware components. In Chapter 4 we show our contributions to efficiently run the SpMV numerical kernel targeting aggressive long vector architectures like the NEC Vector Engine. First we provide implementation details for each of the proposed optimizations for the SELL-C-σ format and SpMV algorithm, and second, we discuss their effects in performance over a wide set of sparse matrices. In Chapter 5 we describe our efforts optimizing the HPCG benchmark for long vector architectures. Along that, we provide two open-source implementations for RISC-VV and NEC Vector Engine devices and evaluate the performance in both platforms. Finally, we propose two accelerators configurations co-designed to run HPCG with high efficiency. Chapter 6 summarizes the contributions presented in this thesis.

Chapter 2

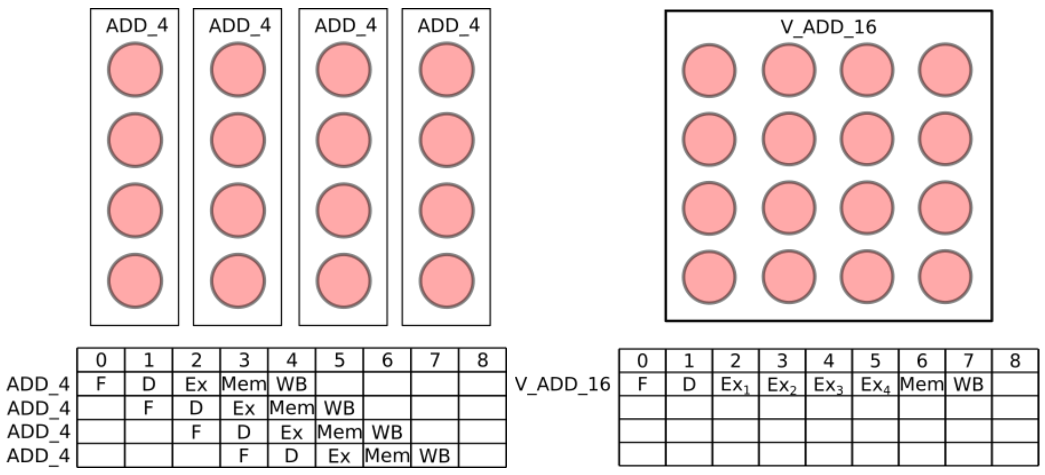
Background

This chapter presents the background concepts necessary to put into context the research developed in this thesis. To do so, in Section 2.1, we focus on the hardware, discussing the current trends in HPC systems, the ideas supporting the use of long vector accelerators and the challenges simulating next-generation complex architectures. In Section 2.2, we focus on the necessary changes applied to algorithms and data structures in order to achieve high performance in such new computing devices. We overview the state-of-the-art sparse matrix formats for SpMV used in accelerators and, after that, we breakdown previous efforts by the scientific community optimizing HPCG targeting long vector architectures.

2.1 HPC systems

2.1.1 Trends and challenges

The design space for next-generation HPC machines is expanding. First, the trend to use commodity server processors as the common choice is changing towards processors with leaner core designs that feature different microarchitectural characteristics. For example, Cray has already deployed Isambard [32], a system with 10,000+ Armv8 cores; and now supports ARM-based processors (including the Cavium ThunderX2) across their main product line. Second, vector architectures with larger lengths than the ones employed in recent years are starting to be considered again. In this regard, Arm recently introduced the Scalable Vector Extensions (SVE) that support up to 2048-bit vectors and per-lane predication. Third, several memory technologies are becoming more relevant in the HPC domain, for example: die-stacked DRAM like the one employed in Knights Landing [76], or High-Bandwidth Memory (HBM) already used in a number of GPUs.



	0	1	2	3	4	5	6	7	8
ADD_4	F	D	Ex	Mem	WB				
ADD_4		F	D	Ex	Mem	WB			
ADD_4			F	D	Ex	Mem	WB		
ADD_4				F	D	Ex	Mem	WB	

(a) SIMD.

	0	1	2	3	4	5	6	7	8
V_ADD_16	F	D	Ex_1	Ex_2	Ex_3	Ex_4	Mem	WB	

(b) Vector.

Figure 2.1: Chronograms representing the instruction flow operating on 16 FP elements (red circles) using a SIMD(a) and Vector(b) instructions. The example assumes a FP compute unit with a throughput of 4 FP elements per cycle and in-order execution.

The most visible example of this technological trend is that the five most powerful supercomputers in the world [82] use either GP-GPU accelerators or a customized CPU specially designed to target HPC applications.

The advent of these trends and technologies leads to a large design space for next-generation HPC machines that needs to be carefully considered. There is a clear opportunity to co-design hardware and software by mapping application requirements to the available hardware ecosystem that these trends are opening.

2.1.2 Long vector architectures

The use of vector or SIMD extensions is becoming commonplace in HPC systems. Some examples are the Fujitsu A64FX, ranked first in the top500 and the NEC SX-Aurora Vector Engine, a discrete accelerator leveraging vector CPUs able to operate with registers of up to 256 double-precision elements. The main difference between conventional SIMD architectures and vector architectures is the detachment between the number of bits/elements that the vector processing unit can compute in parallel (referred as VPU length from now on) and the Architectural Vector Length (*AVL*), which is the maximum number of bits/elements that can be processed with a single instruction. In SIMD architectures (e.g: Intel AVX [58]) the VPU length and the *AVL* is the same. In vector architectures, the *AVL* can be longer than the VPU length. In such case, the VPU computes the result of the operations in several batches of VPU length elements until completion. By reducing the number of instructions fetched, decoded and scheduled, this approach is designed to be more energy-efficient than the more common SIMD alternative [50]. The chronogram in Figure 2.1 shows a simple in-order pipeline example executing the same FP operations in SIMD and vector architectures. While the number of cycles remains equal in this example in both cases, the vector approach allows more instructions to execute concurrently in the pipeline and requires less branching instructions in loop structures, thus increasing the throughput of useful work in real applications.

The RISC-V 'V' extension RISC-V is an open and royalty-free architecture quickly rising in popularity. Due to its modular design, it is a suitable architecture for both general and specialized compute devices. The 'V' extension [72] (from now on RISC-VV) released in September 2021, enables the use of vector instructions in RISC-V. Similar to SVE, RISC-VV is a vector agnostic architecture meaning we can execute the same code in CPUs or accelerators implementing different VPU length. The ISA features instructions supporting operations on scalar, vector and matrix elements, on FP and fixed-point data types. The Xuantie910 [11] from Alibaba is one of the first commercial implementations of the RISC-VV architecture and its designed to accelerate their cloud applications.

In this thesis we focus on the NEC VE and the RISC-VV vector architectures. These architectures permit aggressive implementations that exploit the inherent high data-level parallelism found in many state-of-the-art compute-intensive workloads in fields like HPC, AI, etc.

2.1.3 MUSA and MUSA-RISCV simulators

In this thesis, we employ two different simulation toolchains to produce the performance modeling results. On one hand, we extend the MUSA architectural simulator to obtain the large design-space exploration results we present in Chapter 3. Below, we provide details about the status of the MUSA [29, 30] workflow before our work extending it. On the other hand, we use the MUSA-RISCV simulator to obtain the results used in the performance figures in Chapter 5, dedicated to HPCG optimization. The MUSA-RISCV simulator toolchain differs greatly from the original MUSA. We describe the details of such new toolchain in Section 5.4.

Given the increasing complexity of HPC systems, the ability to predict and fine-tune application performance for selected hardware designs that are deemed of paramount importance to system architects. Architectural simulators are used for that purpose. Prior work [29] introduced MUSA, a comprehensive multi-scale methodology to enable fast and accurate performance estimations of large-scale HPC machines. The methodology captures inter-node communication as well as intra-node micro-architectural and system software interactions by leveraging multi-level traces. These traces also allow for different simulation modes and execution replay to quickly extrapolate results of entire hybrid applications running on large-scale systems with tens of thousands of cores. Methodologies like MUSA can therefore help understand current and future scientific application performance on systems not yet available on the market and identify the best design points.

MUSA employs two components: (i) a tracing infrastructure that captures communication, computation and runtime system events; and (ii) a simulation infrastructure that leverages these traces for simulation at multiple levels.

Tracing: The initial step is to trace an application's execution at multiple levels. Given our targeted hybrid programming model, we start tracing each MPI process representing a rank with a single thread. This trace file contains coarse-grain information about the MPI communication phases as well as high-level computation information of the runtime system events.

In addition, MUSA requires instruction-level instrumentation for computational phases, such as the operation code, the program counter and the involved registers and memory addresses. Such detailed instrumentation is deferred to a separate native execution due to its higher overhead that might alter application behavior. Hence, when tracing in detailed mode, the timestamps taken in the initial coarse-grain trace are used to correct any deviation in the behavior of the application introduced in the detailed trace step.

This tracing methodology generates traces that allow simulations even if the characteristics of the simulated computational node (e.g., the number of cores, the memory hierarchy) or the communication network change. As a result, it is possible to perform architectural analysis of a large design space using the same set of traces, reducing trace generation time and storage requirements.

Simulation: The methodology initially identifies the different computation phases for each rank using the initial coarse-grain trace, which are independent and can be simulated in parallel. Each of these rank-level computation phases is simulated with the specified number of cores, and parameters of the microarchitecture and the memory hierarchy. MUSA is able to simulate an arbitrary number of cores per rank; to accomplish this, MUSA injects runtime system API calls by using the runtime system events recorded in the trace, effectively simulating the runtime system, including scheduling and synchronization for the desired number of simulated cores.

After the computation phases have been simulated, MUSA replays the execution of the communication trace events in order to simulate the communication network and generate the final output trace of the simulation. During this process, the durations of the computation phases are replaced by the results obtained in the simulations, and the communication phases are simulated using a network simulator. At the end of this process the entire simulation is complete and the output trace is generated for visualization and inspection.

2.2 Scientific kernels and applications

2.2.1 Sparse Matrix formats for SpMV

In this section, we introduce relevant state-of-the-art sparse matrix formats that set the grounds to build our long vector architecture targeted optimizations for SpMV.

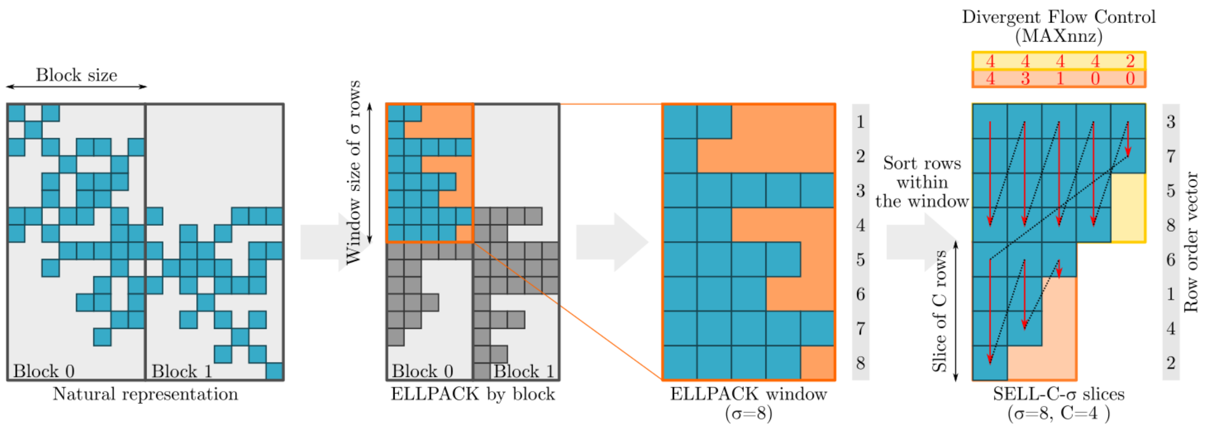


Figure 2.2: Data layout representation of SELL-C-σ with *Column Blocking* and *Divergence Flow Control* optimizations.

CSR The Compressed Sparse Row (CSR) is one of the most commonly used formats to represent sparse matrices. It stores the values and column indices of the number of non-zero (*NNZ*) elements in two separate arrays in row order. A third array keeps a pointer to the starting position of every row in those arrays. The main advantages of CSR are its large compression ratio for any type of matrix element distribution, and the possibility to sequentially accesses the A values and column indices. Its main disadvantages are lack of suitability for vector architectures, and poor locality of x accesses.

ELLPACK ELLPACK [43] is designed to perform efficiently in GPUs and vector architectures. Compared to CSR, it offers improved locality of memory accesses to x by storing and accessing non-zero elements in column order at the cost of additional storage. For a matrix A of size $M \times N$ with a maximum row size of K, it requires an array of size $M \times K$ to store A. The two left-most drawings in Figure 2.2 visually describe how a sparse matrix is compressed and represented in ELLPACK. The main downside of ELLPACK is that it only offers good performance and compression as long as the stored matrix has a regular *NNZ* elements per row.

Sliced ELLPACK SELLPACK [64] optimizes the ELLPACK format to improve storage efficiency and throughput for both regular and irregular matrices. In this format, matrix rows are ordered by their *NNZ*. Then, the matrix is logically divided into slices of a fixed number of rows. Rows are padded with zeroes to match the longest row within the same slice, as opposed to ELLPACK that matches the longes row in the whole matrix.

SELL-*C*-σ SELL-C-σ [48] introduces the idea of limiting the sorting window to avoid reordering the whole matrix. SELL-C-σ addresses two main issues: i) the large cost of sorting matrices with a large number of rows; and ii) the exploitation of the natural locality in accesses to x of adjacent rows, since applying a total sorting of the matrix can harm such locality. The sorting window size is defined by the σ parameter, which is typically a multiple of the maximum length of the SIMD or vector unit.

ELLPACK Sparse Block (ESB) ESB [54] introduces two additional optimizations to reduce bandwidth requirements: column blocking, and the use of a bit array to mask instructions. In an effort to improve the locality of accesses to x, ESB splits matrices into blocks of columns or *slices*. ESB uses a bit array data structure to store a one-bit mask for each column of the slice. A bit is set to one for each non-zero element in the column. This information is later used to mask the elements of the SIMD operations. It achieves large compression rates since it does not require zero-padding.

Figure 2.2 represents the logical steps to convert a sparse matrix to the SELL-C-σ (with blocking) format during the preprocessing phase. In this simplified example we consider the parameters: $C = 4$, $\sigma = 8$ and *num. blocks* = 2. The natural representation of the matrix is divided into two separate blocks of *Block size* columns. Then, the rows of each block are ordered by size within a so-called ELLPACK window, i.e., within the orange region containing contiguous chunks of σ rows. The rightmost drawing (SELL-C-σ slices) shows the final data layout within a window where two slices are highlighted in light-yellow and light-orange. Elements within each slice, including the zero-padding are stored contiguously in *column-major* order, if no optimization is applied. If the Divergent Flow Control optimization is applied (described in Section 4.3.2), the elements are stored in the same *column-major* order, but without any zero-padding. Red lines in the rightmost drawing of Figure 2.2 represent this order.

2.2.2 HPCG benchmark and optimization

The High-Performance Conjugate Gradient (HPCG) benchmark solves a symmetric sparse linear system representing the 3D semi-regular grid of a finite element problem. It uses the Conjugate Gradient method to precondition the sparse linear system on each iteration to approximate faster to a solution.

Algorithm 1 CG algorithm

```
Input: $x_0, A$
 1: $r_0 \leftarrow b - Ax_0$
 2: for $(i = 1; i < max\_iterations; i++)$ do
 3:     $z_i \leftarrow M^{-1} r_{i-1}$                          ▷ MG(SpMV, SymGS)
 4:     $rtz_i \leftarrow (r_{i-1}, z_i)$                          ▷ DOT
 5:     if $i = 1$ then
 6:         $p_i \leftarrow z_i$
 7:     else
 8:         $p_i \leftarrow z_i + (rtz_i / rtz_{i-1}) p_{i-1}$        ▷ WAXPBY
 9:     end if
10:     $\alpha_i \leftarrow rtz_i / (p_i, Ap_i)$                  ▷ SpMV, DOT
11:     $x_{i+1} \leftarrow x_i + \alpha_i p_i$                      ▷ WAXPBY
12:     $r_{i+1} \leftarrow r_i - \alpha_i Ap_i$                     ▷ WAXPBY
13:     if $||r_i||_2 \leq threshold$ then                 ▷ DOT
14:         break
15:     end if
16: end for
```

Algorithm 1 is the pseudocode representation of the Conjugate Gradient method used in HPCG. Next to each line, we annotate the corresponding main kernel. Vector scale and add (WAXPBY) and dot product (DDOT), are vector-vector operation kernels. Sparse Matrix-Vector multiplication (SpMV) and Symmetric Gauss-Seidel (SymGS), perform matrix-vector operations. SpMV and SymGS are the most time-consuming kernels of HPCG, roughly representing a 97% of the execution time [73]. Both kernels have also low arithmetic intensity and very similar access pattern, and often suffer from memory bandwidth performance bottlenecks. Additionally, using the default out-of-the-box (*Vanilla* from now on) implementation of both kernels, it is very challenging to produce an efficient vector code, even with manual vectorization. In order to maximize the HPCG performance, it is necessary to modify the code to improve memory accesses and vector unit utilization.

2.2.3 State-of-the-art HPCG optimizations

Following, we break down the two most relevant state-of-the-art HPCG tuning proposals targeting long vector architectures. A first proposal [44], demonstrates the benefits of using i) ELLPACK (*ELL*) matrix format [43], ii) *Hyperplane ordering* [2] (more often known as level-scheduling) and iii) selective caching in the on-chip scratch-pad memory available. In future references, we will refer to such set of optimizations as *Basic Vectorization*. The proposal is evaluated in the SX-ACE compute system [63] and achieves roughly 30 GFlops (a 11.9% performance efficiency). For reference, the SX-ACE is the previous generation of the VE.

A subsequent proposal [47], extends the work of the *Basic Vectorization*, and adds an additional set of optimizations to boost the HPCG performance in long vector accelerators. The most relevant changes are: iv) the use of a *L+U, Halo, Diagonal* matrix data structure decomposition, v) tweaking the SymGS algorithm to reduce the number of floating-point operations in half and vi) fusing kernel routines that operate over the same data consecutively, improving temporal locality of the accesses to the cache. In future references, we will refer to this set of optimizations as *Advanced Vectorization*. All optimizations together add to a total performance of 128.8 GFlops in a VE (model 10B) accelerator, a 5.99% performance efficiency.

Our work takes the *Advanced Vectorization* as a starting point and we build our contributions on top of it. In the following paragraphs, we provide details of the most relevant vectorization optimizations proposed by previous work. Further ahead in Section 5.3, we discuss and provide implementation details about our optimizations extending the state-of-the-art vectorizing HPCG.

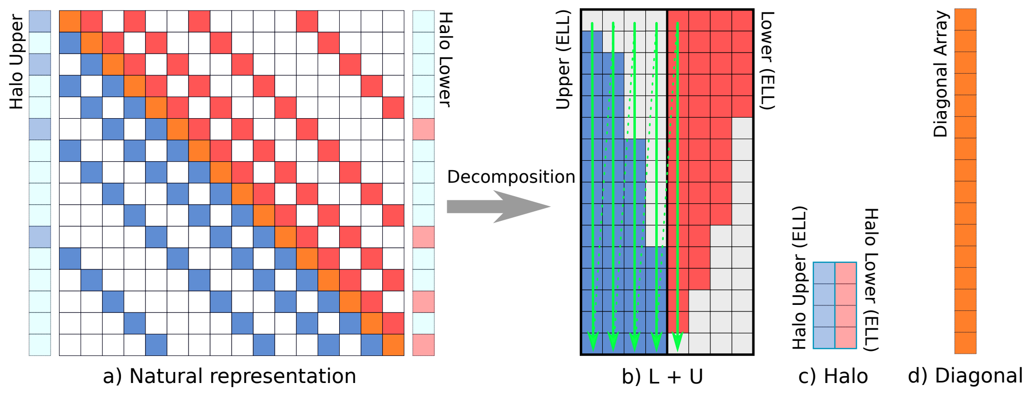


Figure 2.3: *A* matrix data layout decomposition into L+U, diagonal and Halo.

Data structures layout The two main data structures are the sparse matrix A and the vector x. Storing the matrix A in memory means storing the indices of the non-zero elements and their values. Optimizing the data layout of this information in memory is key to achieve performance. The default HPCG 3.1 code, stores the values of matrix A in a format equivalent to CSR (Compressed Sparse Row) where data is stored sequentially by rows. However, using *ELL* format is a better choice in this case. *ELL* is a well-known suitable format for vector architectures [43, 7]. In kernels like SpMV and SymGS it allows a straightforward use of vector instructions to perform operations on several rows in parallel. To support more complex optimizations, further layout modifications to the *A* data structures are required. Figure 2.3 shows a natural representation of a matrix similar to *A* in HPCG decomposed into *L+U, Halo, Diagonal*. In this new layout, the matrix *A* is split into Lower (*L*) and Upper (*U*) triangular matrices. Each triangular matrix is formated in *ELL* and then stored sequentially one after the other. The diagonal elements of the matrix are stored in a separate vector. The halo boundary elements of the matrix (*AH*) are stored in a separate *ELL* data structure also split in a lower-upper fashion. This new data reorganization enables total or partial SpMV and SymGS efficient operations on the matrix using a single data structure.

Level scheduling In addition to the data structure changes, further modifications are needed to obtain good parallelism and an efficient vectorization on SymGS. Some elements in the matrix might have compute dependences between them, making impossible to process them in parallel. A well-known technique to improve parallelism in SymGS is coloring. Coloring relaxes some dependencies between elements of the matrix, to create groups (called colors) of consecutive rows that can be computed concurrently, at the cost of slower convergence. Another approach is to group the rows based on the level-scheduling algorithm [2]. In this case, every group (called level) contains only elements that can be processed in parallel without breaking any compute dependency. Therefore not requiring additional iterations to converge to the solution with respect to the sequential approach. Even with strongest constraints, level-scheduling is suitable for highly regular sparse symmetric matrices (like the one in HPCG) where it is able to find enough row parallelism creating large *levels* to efficiently exploit wide vector units.

Algorithm tweaks There are several opportunities to increase performance by reducing the total number of floating-point operations required to solve the linear system. On one hand, once the matrix is stored in three separate data structures (*L+U, Halo, Diagonal* decomposition), it is possible to reformulate the SymGS algorithm to save half of the compute operations and loads from memory, which has a significant impact on the performance. This is a well-known technique also applied in several hardware vendor implementations [47, 68, 67, 55].

On the other hand, we can improve memory access locality by merging consecutive kernels that operate over the same data. For example, we fuse the SpMV and DDOT, and WAXPBY and DDOT operations (see Algorithm 1 lines 10 and 12 respectively). This last optimization requires a relatively small modification in the code and yields a 1.9% performance increment.

Kernel vectorization On the *Advanced Vectorization* implementation the SymGS and SpMV routines call to a vendor proprietary library from NEC. Such library is heavily tuned for the VE and achieves very high performance. Additionally, the simple WAXPBY and DDOT routines are replaced by modified versions including custom NEC compiler directives to facilitate a more efficient vectorization by the compiler. Since we can not access the code of such library, we replace the calls to SymGS and SpMV by our own implementation written with public API vector intrinsics for both the VE and RISC-VV [56, 72]. We also implement our own WAXPBY and DDOT kernel routines. We provide more detailed comments about the implementation and optimization of these four routines in Section 5.3.

Other optimizations The pre-process of HPCG is factored into the benchmark performance measurements. Meaning poorly optimized data structure transformations, for example, the ones required to obtain the *L+U, Halo, Diagonal* decomposition or reordering the matrix, might introduce a noticeable performance degradation in the final GFLOP count. For reference, in our experiments, adding an overhead of 1.5 seconds spent in pre-processing routines resulted in a performance degradation of 2.7%.

Additionally, the same work discusses the use of specific device optimizations, like using tiling of the shared caches. In the particular case of the VE, tiling the last level cache partitions the single shared coherence domain into two. In the context of an MPI application, where processes do not share memory, tiling reduces unnecessary cache conflicts and provides roughly 10% performance benefits running HPCG in the VE. Since this paper focus on optimizations applicable to long vector architectures in general, we do not include the tiling optimization in our experiments executed in the VE.

Chapter 3

A Design Space Exploration of Next-Generation HPC machines

This chapter develops the contributions introduced in Section 1.1.2. The chapter is organized as follows. We discuss the related work in Section 3.2. We describe our work extending MUSA methodology in Section 3.3.1. We introduce our experimental setup in Section 3.3.2. We perform the design space exploration of next-generation HPC systems in Section 3.4.

3.1 Introduction

In Section 2.1.1, we discuss about the rapid changes in technology in current compute systems. In this chapter, we undertake a design space exploration study that considers the most relevant design trends we are observing today in HPC systems. To perform this study, we follow a recently introduced multi-level simulation methodology (MUSA) [29]. MUSA enables fast and accurate performance estimations and takes into account inter-node communication, node-level architecture, and system software interactions. We extend MUSA in multiple ways to perform a detailed evaluation of the impact of a set of identified key design trends on relevant HPC applications. The contributions of this chapter are:

- We extend an end-to-end simulation methodology called MUSA. In particular, we add support for a larger set of OpenMP pragmas, multiple target architectures, a model for vectorization, performance and power modeling of emerging memory technologies, and processor power estimations. The resulting infrastructure can target a wide range of HPC applications, and provide performance and power estimations of the trends that are currently dominating in HPC.

- We perform large-scale simulations comprised of 256 MPI ranks, each representing a compute node, and up to 64 cores per node adding a total of 16,384 cores. Our design space exploration methodology analyzes current HPC trends by factoring in the relevant architectural parameters, for which we derive over 800 different architectural configurations. We simulate each of the 5 selected representative HPC applications with these configurations and provide estimations for performance, power and energy-to-solution.

- Through our extensive design space exploration, we provide hardware and software co-design recommendations for next-generation large-scale HPC systems.

3.2 Related Work

Subsystem simulators are common tools that allow us to obtain performance predictions and assist computer architects into designing specific parts of the HPC systems. Mubarak et al.[66] propose CODES a fast and flexible simulation framework to model state of the art Torus and Dragonfly networks at a large-scale. Compared to this, our work focuses on a multi-level simulation of the whole parallel system.

Early proposals [90, 31, 16] offer solutions targeting large-scale systems capable of simulating thousands of nodes, but their frameworks focus mainly on network events and they not model CPU components or system software interactions in detail.

Wang et al. [85], present a multi-level simulation framework that is capable of modeling in detail many parts of the system architecture including power estimations but is limited to single node applications.

SST is a multi-scale simulator often used in combination with other simulators to model distributed applications. In BE-SST, authors combine SST with coarse-grained behavioral emulation models abstracting from microarchitectural details in favor of simulation speed. Other implementations integrate SST with a highly accurate simulator but require too costly full system simulations to produce a wide set of experiments [38, 70].

With the same objective, application specific analytical models [62, 40] use a small set of parameters to predict performance for a single application on large systems. Once those models are created and validated they are able to predict performance accurately with negligible compute and time cost. The main downside of these models is that they have little flexibility; any significant change in the application or hardware architecture requires the model to be updated, refined and validated again. Our methodology focus on hardware microarchitectural exploration and iterative fast co-design; new features can be tested on all applications the moment they are included in the simulator with enough level of detail to study in depth hardware-software interactions.

3.3 Methodology

In order to be effective, we need to ensure the employed methodology captures the trends and the characteristics of the technologies under consideration, as well as main design constraints such as power consumption. In this chapter we significantly extend MUSA to support modeling of key aspects to enable accurate performance estimations of large-scale HPC systems that will make use of the technologies mentioned before. The following section describes these extensions that are later used to perform a comprehensive design-space exploration analysis of large-scale HPC systems.

3.3.1 Extending MUSA for Practical Design-space Exploration

We extend MUSA in multiple ways that not only allow us to capture the technological trends described before, but also enable exploration of a wider range of applications on different hardware architectures. The following paragraphs describe the functionalities added to MUSA for producing the results we present in this chapter.

Support for OpenMP `parallel for` *constructs:* The tracing infrastructure had support to trace applications based on tasks, both based on the OmpSs programming model[18] and the tasking features introduced in OpenMP 3.0 [15]. This was a limiting factor that did not allow us to trace applications of interest that use classic parallel for constructs, restricting exploration studies to a small set of application choices, thus narrowing co-design opportunities. For this reason, we extend the tracing infrastructure to support parallel loops as well as other common directives like *omp critical*. Therefore, MUSA is able to target a wide range of applications, making design-space exploration studies more robust and with a better coverage of application characteristics.

Support for multiple architectures: The tracing infrastructure was based on the dynamic binary translation tool PIN [60]. However, PIN is x86-specific and has some pitfalls that are not easy to overcome. For example, it is closed source code and comes with its own version of *libc*, which is not *c++11* compliant. This can lead to stiff restrictions when compiling applications that need full compliance with this standard, either directly or through any external library. To overcome these issues, we port the entire tracing infrastructure to DynamoRIO [19], which allows MUSA to support both x86_64 and Armv8 binaries while also removing all the restrictions that PIN imposes with its embedded *libc*. As we did with our PIN traces, we also validate that DynamoRIO traces are accurate. When compared to PIN traces on an Intel Xeon E5-2670, the differences in terms of loads, stores, and micro-instructions remained below 0.14%, 0.27%, and 2.40% respectively for all tested applications.

Support for vectorization: We consider a simple and practical model to enable simulations using different vector lengths while reusing the same application instruction trace. When our instruction tracing infrastructure finds a vector instruction, our internal decoder breaks them into scalar instructions with a special marker, effectively obtaining a trace with only scalar instructions. During the simulation step, if a vector length of 128 bits or wider has been requested, we fuse the marked instructions in order to simulate the specified vector length. For example, if a 128-bit vector length is specified, two arithmetic instructions are fused into one, while memory operations are also fused but its size is doubled to account for memory bandwidth. This mechanism trivially works for vector lengths of the same size or narrower than the one used during the tracing step. However, we also enable simulations with wider vector lengths by applying this fusion-driven mechanism to dynamic instructions corresponding to the same static instruction of the same basic block. We require a basic block to be executed several times in a row to apply the fusion mechanism. While this simple model may overestimate the vectorization impact, it is useful to indicate the potential that current HPC codes have for performance improvements when exposed to long SIMD register sizes.

Support for emerging memory technologies: A key component that significantly impacts the performance of current and future HPC machines is the employed memory technology. To enable MUSA to accurately model a wide range of emerging memory technologies, we add interfaces into our architectural simulator to support a fast and extensible external memory simulator, Ramulator [42]. Ramulator is able to model a wide range of commercial and academic DRAM standards, including: DDR, LPDDR, HBM, and Wide-IO. In addition, for most of these standards, Ramulator is also capable of reporting power consumption by relying on DRAMPower [10] as a backend. We integrate these tools into our toolflow, providing a robust infrastructure to obtain performance and power estimations for the memory subsystem.

Support for power estimations using McPAT: We also integrate the McPAT [51] modeling framework into our toolflow to obtain power estimations of the simulated multicore. We feed McPAT with the different architectural descriptions, as well as the simulation statistics, to obtain power estimations of the different cache levels and key core hardware structures.

The sum of these new features makes MUSA a robust tool to perform exhaustive design-space exploration studies that can cover a wide range of applications, potentially on different architectures, as well as hardware trends like vectorization and emerging memory technologies.

3.3.2 Design space exploration

This section describes the methodology we employ to carry out our design space exploration of HPC architectures.

Architectural Parameters

After reviewing the HPC systems landscape, we select a set of important compute node features in current and upcoming HPC architectures. These features expose relevant energy and performance trade-offs when considering different HPC workloads. We focus our exploration on six features: number of cores in a socket, out-of-order (OoO) capabilities of the core, memory technology, floating-point unit (FPU) vector width, CPU frequency and cache size.

Per each feature, we explore a set of possible values that are based on newly added Top500 systems or near-future announced systems [23, 65, 33, 75, 61]. Based on this information, we use 32 to 64 cores per socket with clock frequencies ranging from 1.5GHz to 3GHz. We define four types of core pipelines by modifying their OoO capabilities: a modest but floating-point capable, close to in-order, low power core, with three floating-point units and small issue width and buffers; two server-class cores in the medium high range; and an aggressive high-end configuration with an issue width of eight instructions, large hardware buffers, and up to four floating point units. To tweak the memory hierarchy we modify the L2 (private) and L3 (shared) cache sizes as well as the number of off-chip memory channels using DDR4-2333 technology. The private L1 cache size is fixed to 32kB per core. Lastly, we also explore the impact of using 128-bit, 256-bit and 512-bit wide floating point units. Table 3.1 shows a detailed list of all the parameters and values we explore and the names (labels) we will use to refer to them. We consider each possible combination of architectural configurations, running in total 864 simulations per application.

HPC Applications

To perform our experiments we use five HPC applications: HYDRO [49], a simplified version of RAMSES [80] that solves compressible Euler equations of hydrodynamics using the Godunov method; SP and BT multizone NAS benchmarks [84], which implement diagonal matrix solvers; LULESH [39], which implements a discrete approximation of the hydrodynamics equations; and Specfem3D, which uses the continuous Garlerkin spectral-element method to simulate forward and adjoint seismic wave propagation on arbitrary unstructured hexahedral meshes. For each application we adjust the input sets to potentially have enough parallelism when running on 256 MPI Ranks, one per node, and 64 cores per node, 16,384 cores in total.

L3:L2-caches	Size / associativity / latency	
Label	L3	L2
32M:256KB	32MB / 16 / 68	256kB / 8 / 9
64M:512KB	64MB / 16 / 70	512kB / 16 / 11
96M:1MB	96MB / 16 / 72	1MB / 16 / 13

Core OoO Label	ROB	Issue& commit	Store buffer	#ALU/ #FPU	IRF/ FRF
low-end	40	2	20	1 / 3	30/ 50
medium	180	4	100	3 / 3	130 / 70
high	224	6	120	4 / 3	180 / 100
aggressive	300	8	150	5 / 4	210 / 120

Other param.	Values
Frequency [GHz]	1.5, 2.0, 2.5, 3.0
Vector width [bits]	128, 256, 512
Memory [DDR4-2333]	4-channel, 8-channel
Number of Cores	1, 32, 64

Table 3.1: Simulation architectural parameters and values used in our design space exploration including: cache size, associativity and latency; and OoO details like Reorder buffer (ROB) and Integer/Float Register File (RF).

All applications use a hybrid programming model, either (MPI+OpenMP) or (MPI+OmpSs). OmpSs [18] and OpenMP [15] (since ver. 3.0) allow the annotation of parallel regions as tasks with input and output dependencies. During execution, OmpSs runtime system is in charge of scheduling task instances to the compute units. We compile all applications with GCC 7.1.0 and OpenMPI [24] 1.10.4. In GCC we use the -O3 optimization flag and force -msse4.2 in order to generate binaries with Intel SSE4.2 (128-bit width) SIMD instructions.

For the purpose of workload characterization and analysis discussion, in Figure 3.1 we show runtime memory access statistics that we obtain in our detailed hardware simulations.

Tracing and Simulation Infrastructure

For our experiments, we use a toolchain that implements the MUSA methodology. It allows us to obtain traces and to perform simulations at different levels of abstraction as we describe in Section 3.3.1. We obtain application traces using two lightweight tracing tools: Extrae [21] and DynamoRIO [19]. For each application we obtain two traces: a high-level trace (burst) using coarse-grained instrumentation and a low-level trace using fine-grained instrumented (detailed). The burst trace captures the events through the execution of the whole application. However, we only trace in detail a sample region of each application: in our applications, tracing one

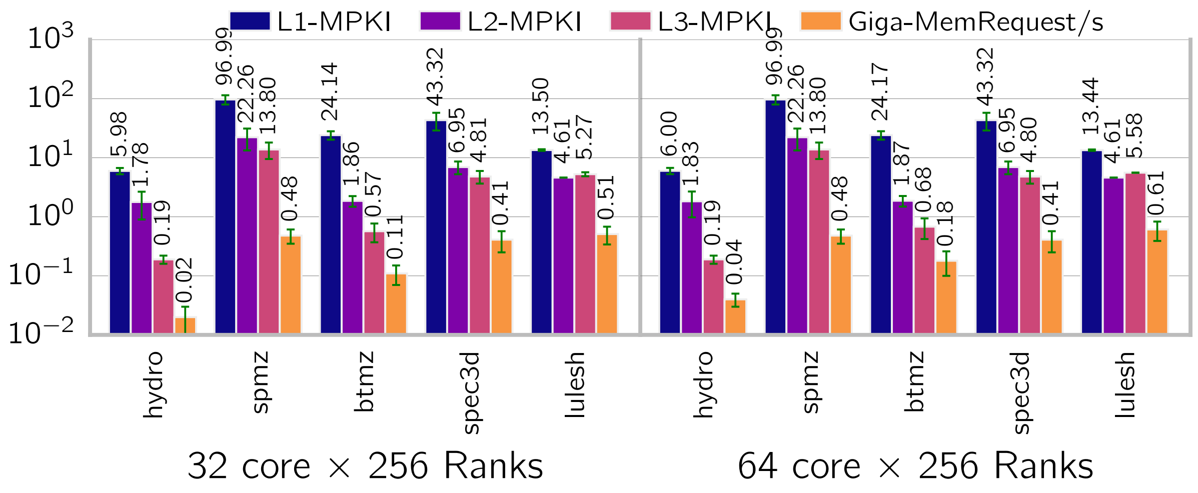


Figure 3.1: Application runtime statistics: Misses Per Kilo Instruction (MPKI) of caches and Billions of requests to main memory per second.

iteration (usually the second) of one MPI rank, is enough to capture a representative sample of the computational behavior of the application [29]. Obtaining the low-level traces takes between 1 and 5 hours depending on the application.

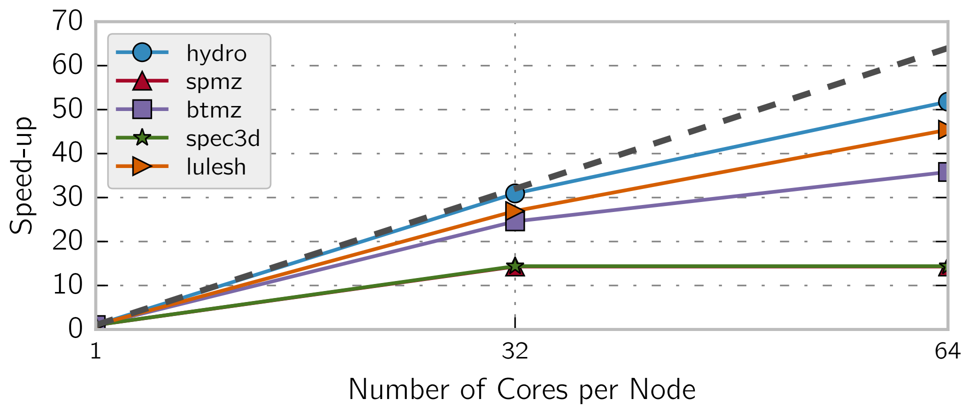
For detailed instruction-level simulations we use TaskSim [71] and Ramulator. Full application simulations are driven by Dimemas [5], which implements a high-level network model and is capable of integrating the accurate timing values that we obtain doing detailed simulations with TaskSim. Each of the 864 simulations takes between 2 and 50 hours depending on the application.

We obtain node power estimations using McPAT [51] and DRAMPower [10]. During TaskSim simulation Ramulator generates DRAM command traces that DRAMPower uses as input. Although Ramulator splits that trace into a different file for every channel and rank, DRAMPower does not support multi-rank DIMM simulation. Instead, we specify power parameters using single rank DDR4 datasheet from Micron [1]. In simulations with four channels we attach eight DIMMs for a total of 64 GB while in simulations with eight channels we attach 16 DIMMs for a total RAM of 128 GB. All power measurements take into account both static and dynamic power.

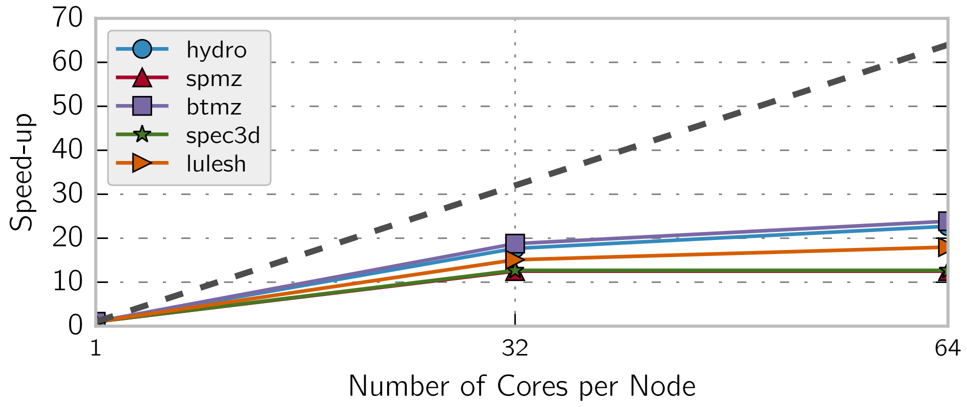
Dimemas and TaskSim, have been validated using the MareNostrum III supercomputer and three MPI+OpenMP codes obtaining a $< 10\%$ relative error [29, 25]. Previous work [42] describes how Ramulator is validated. We use the latest model adjustments in McPAT for state-of-the-art CMPs in order to improve accuracy to $< 20\%$ error [86]. DRAMPower claims to have $< 2\%$ error [22].

3.4 Evaluation

3.4.1 Scaling Analysis of Applications



(a) Single compute region of the application.



(b) Full application parallel region.

Figure 3.2: Scaling of applications using hardware agnostic simulations: a) measures a single representative compute region without MPI communications, b) measures the whole parallel region including MPI overheads.

Programming parallel applications to scale efficiently in next-generation systems where we expect to have a high number of distributed compute elements is a challenge. Understanding the code parallelization approach and scaling limits of applications is very valuable knowledge that helps to explain possible unexpected hardware interactions: for example, underusage of shared resources like memory buses and caches in memory intensive applications due to a low number of concurrent tasks in a processor. In this section, we show a scaling analysis of the five applications that are object in this study. We have been able to identify and quantify the source of parallel programming-related bottlenecks limiting scalability like MPI overheads or OpenMP task scheduling bottlenecks. We use the MUSA burst mode to obtain traces of

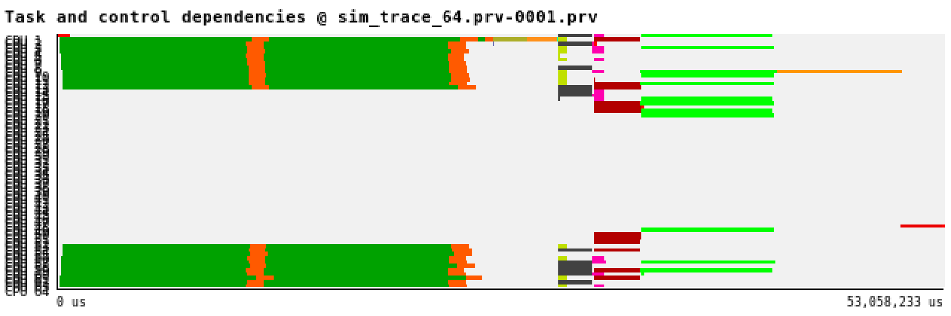


Figure 3.3: Low level task parallelism is the main cause for low parallel efficiency in Specfem3D. Many CPUs remain idle (in gray) during the whole execution, while few are populated with tasks (in color). X axis represents time, the Y axis represents the thread number.

the compute and MPI regions. These traces drive the simulation of applications in nodes with up to 64 cores. In burst mode, no details of the processor architecture are modeled and task execution time is not affected by penalties from shared cache contention or memory bandwidth exhaustion (see Section 2.1.3); we call this *hardware agnostic* simulation.

In Figure 3.2a we simulate a single representative compute region of each application; across applications, we observe an average parallel efficiency of roughly 70% at 32 cores, dropping rapidly to 50% at 64 cores. HYDRO is the only application whose main compute region scales over a reasonable $> 75\%$ parallel efficiency in systems with 64 cores per node. We analyze with visualization tools [69] traces containing task execution and task scheduling events generated during the simulation of applications in burst mode. In them we observe that the main overall source of performance losses at 32 and 64 cores for all codes is the lack of task level parallelism. Even using relatively large input sets, the main parallel compute regions in all applications except HYDRO do not have enough fine-grained task granularity to fill all cores simultaneously. In Figure 3.3 we can appreciate this behavior in Specfem3D, where most tasks (colored) are scheduled only in few of the threads while the rest remain idle (middle horizontal gray area). We also find important serialized execution segments in all applications except SPMZ. Lastly, thread level load imbalance is the main issue limiting LULESH scaling potential on 64 core configurations.

To further study the additional overhead of MPI communications on top of the code parallelization issues, we integrate all compute regions together with all MPI regions. Figure 3.2b shows the achieved scalability without considering initial data structure allocation and final I/O operations. Same as before, we do not consider shared caches or memory bus contention effects inside the node. We simulate a network with a bandwidth and latency similar to Marenostrum IV [61]. In this case, average parallel efficiency scales up to 49% and 28% when using configurations with 32 cores or 64 cores respectively. By analyzing MPI communication traces we observe that: *i)* message passing represents a minimal part of the total MPI overheads; *ii)* load imbalance, across different MPI ranks in the presence of synchronization barriers, causes a significant loss of performance in all applications except for HYDRO. In figure 3.4 we appreciate this behavior: colored in red the actual time spent point-to-point calls, in pink MPI_AllReduce regions causes all ranks to synchronize.

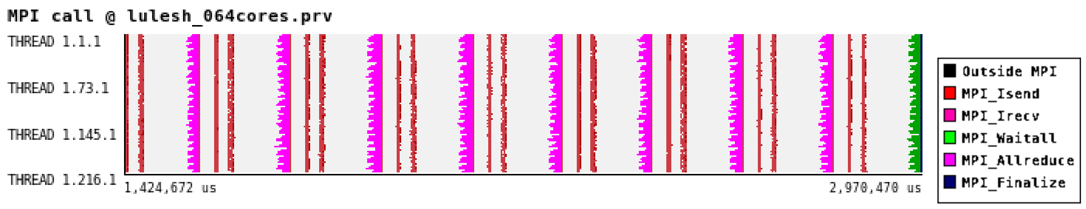


Figure 3.4: Visual timeline of a trace representing MPI and compute phases. Significant unnecessary time is spent in MPI_Barriers due to load imbalance in LULESH. X axis represents time, the Y axis represents the rank number.

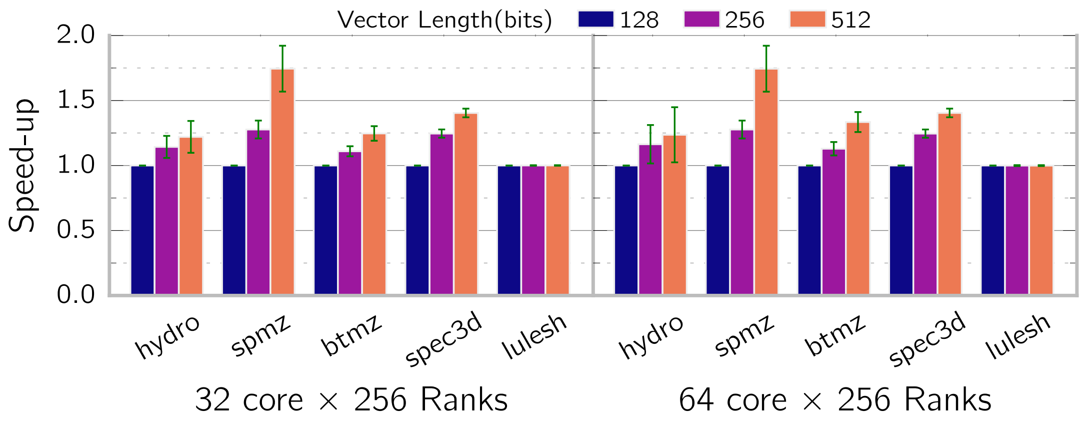
3.4.2 Hardware Exploration

Next, we present our design space exploration based on detailed hardware simulations for six main architectural components. For each component, we discuss the results that we provide in form of performance, power dissipation and energy-to-solution figures. To quantify the performance impact of each individual component, our plots represent the average values obtained by normalizing each simulation against another simulation that shares all the other architectural parameters except the one we are quantifying.

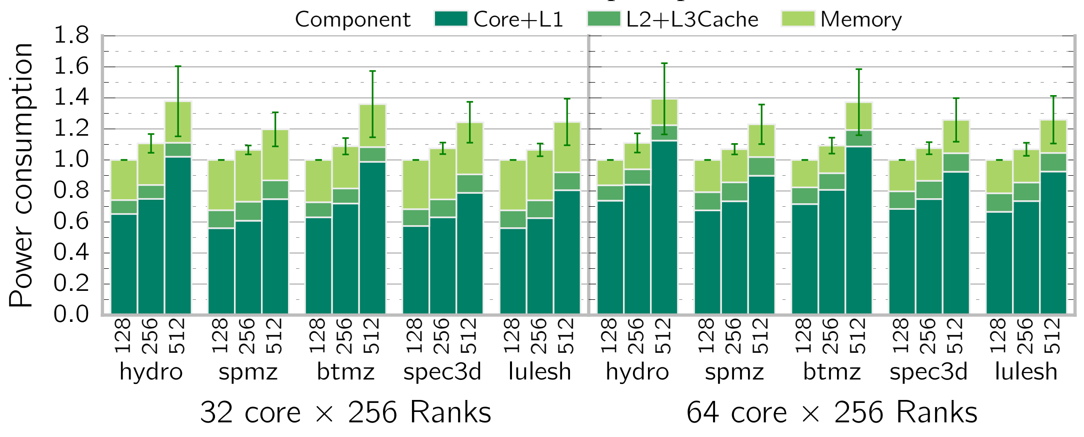
For example, in Figure 3.5a we measure the impact across applications of increasing the FPU vector width. Considering that we have a total of six parameters, a simulation instance would be $\{x,y,z,s,t,128bit\}$, where x,y,z, and t represent a specific value for each of the other architectural components parameters. Then, we normalize the execution time of each simulation with vector width = 256-bit against its baseline simulation that shares all the other architectural parameters, i.e., we would normalize $\{x,y,z,s,t,256bit\}$ against our baseline $\{x,y,z,s,t,128bit\}$. In this case, with a total of 864 simulations per application, we are averaging 96 samples per bar.

FPU vector width

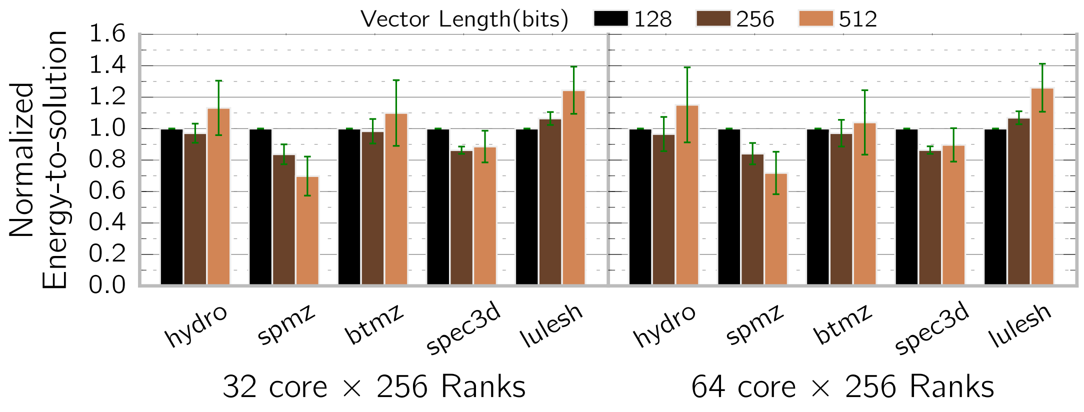
Figure 3.5 summarizes the performance-energy trade-off when we increase the vector Floating Point (FP) registers used for SIMD operations in each core. Results for 32 and 64 core configu-



(a) Performance Speedup



(b) Power dissipation



(c) Energy-to-solution

Figure 3.5: Average simulation results increasing FPU width up to 512-bits. Normalized to 128-bit configurations.

rations are very similar. Excluding LULESH, wider 512-bit FP units yield 20% (HYDRO) to 75% (SP-MZ) application performance speed-up; 40% on average.

As we describe in Section 3.3.1, the TaskSim approach to simulate SIMD instructions targets loops containing vector instructions that run for a large number of iterations. In codes with loops with a very short iteration count, like LULESH, our approach does not detect any potential for performance improvement by extending the vector size.

Measuring the *Core+L1* component in Figure 3.5b, we see that using 512-bit vector width translates into an average power increment across applications of 60% with respect to 128-bit units in each core. As expected, the core power consumption is relatively larger in compute-intensive applications like HYDRO and BTMZ than in memory bound counterparts. As a consequence, we appreciate a larger impact in the power consumption of applications that are compute-intensive, when increasing the vector units width. In terms of energy-to-solution, in all applications except LULESH, 256-bit configurations obtain 3% to 18% energy savings.

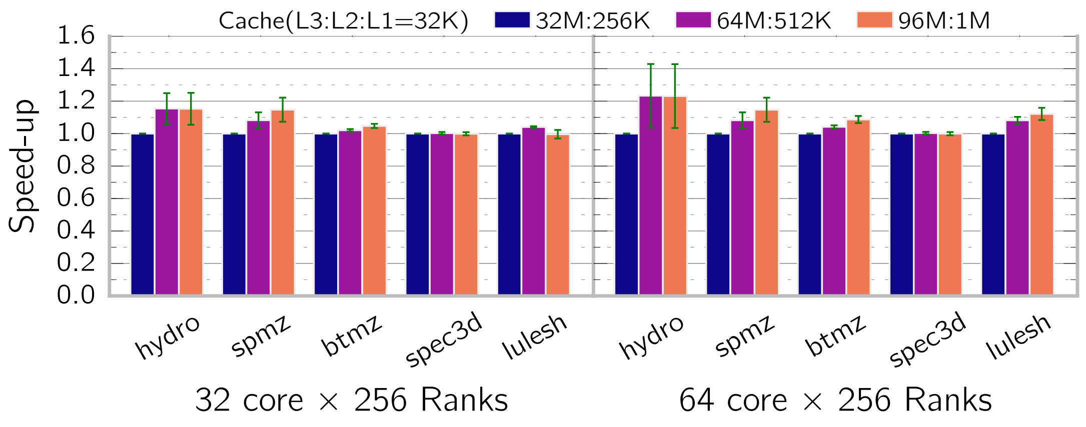
A careful analysis of the current auto-vectorization extracted from the applications under study shows that the auto-vectorization is enough to start obtaining gains from 256-bit vector units, but we lack manual vectorization or more efficient compiler auto-vectorization techniques in order to benefit from larger vectors.

We observe important timing differences when HYDRO is executed with L2 cache configurations smaller than 512KB. Configurations that perform more accesses to main memory are slower and have less throughput. In consequence, the stress to the individual compute units inside each core is also reduced. This allows to such units to scale better in relative terms when we upgrade them; absolute execution time will still be lower due to the slower additional memory accesses (see Fig. 3.6a). For the same reasons and in a similar case, when using smaller cache and low-end core configurations in BTMZ, widening the FP units from 128- to 256-bit yields a higher relative speedup compared to faster cores with bigger caches and better OoO capabilities.

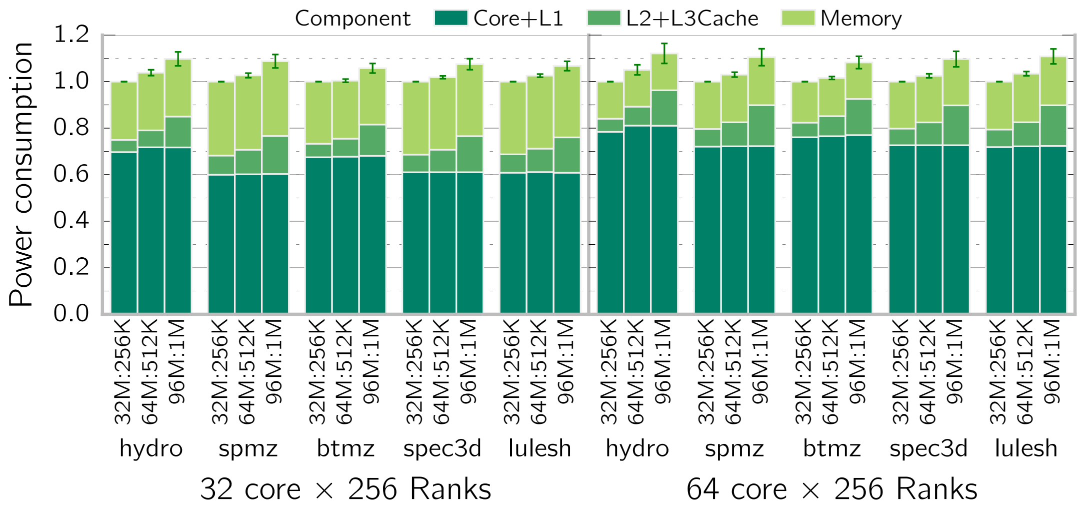
Cache sizes

Figure 3.6 shows how only modifying L2- and L3-cache sizes affects performance in our simulations; at 64 cores, upgrading to a cache configuration with 96MB:1MB (1.5MB:1MB per core) results in an 11% average speedup across applications.

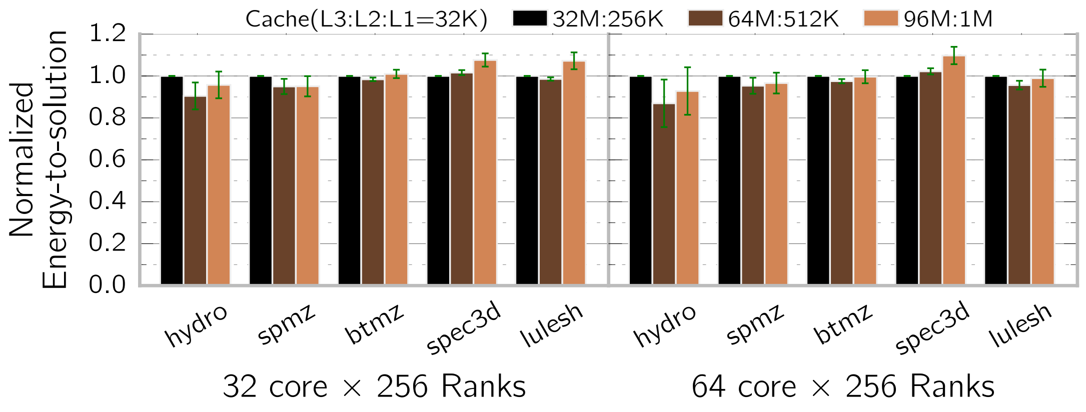
Fitting the workload dataset in cache has a huge impact on performance. In HYDRO we obtain a 4× drop in L2-cache MPKI when upgrading the L2-cache size from 256 kB to 512 kB per core, meaning that the main working set of HYDRO fits in less than 512 kB. This translates into a 21% average performance improvement for HYDRO. In the case of BTMZ and LULESH, we also obtain 9% and 12% speedup respectively. Specfem3D shows no differences across



(a) Performance



(b) Power dissipation



(c) Energy-to-solution

Figure 3.6: Average simulation results varying L3- and L2-cache parameters. Normalized to 32MB:256KB cache configs.

cache configurations: although it obtains locality benefits using larger caches those are minimal and it is not enough to compensate the increased latency per access to larger cache structures.

Figure 3.6b shows that with 64 cores, in configurations with 32MB, 64MB, and 96MB of L3-cache, the *L2+L3Cache* component represents respectively around 5%, 10% and 20% of the total power consumption. While the performance benefits when upgrading caches from 32MB:256KB to 64MB:512KB are significant in some applications; further upgrading from 64MB:512KB to 96MB:1MB at the cost of doubling the power budget of L2-/L3-cache is not justified due to smaller gains in terms of performance. Similar trends can be seen in the 32 cores configurations. Aside from Specfem3D, final energy reductions are minimal, around 5% on average for 64MB:512KB and $\sim$1% for 96MB:1MB.

Careful sizing of the L2 and L3 cache structures is necessary to minimize power consumption while achieving a good level of performance. Based on our results, we find that 1MB L3-cache and 512 kB L2-cache per core seem to offer the best trade-off. Note that we execute our applications without any manual or auto-tuning optimizations. Also, adding software optimizations, like cache blocking, to adequately fit application working sets into cache should be specially considered in systems with a high number of cores per socket to improve the energy consumption and reduce last-level cache miss overheads.

LULESH and HYDRO simulation results have a significant standard deviations of 5% and 15%, respectively. In LULESH, eight DDR4 channels compared to four DDR4 channels configurations achieve lower relative speedups when increasing L2- and L3-cache sizes. Although in absolute terms eight-channel configurations run faster, the relative improvement is lower. Also, variations in HYDRO are caused by bottlenecks in configurations with more than 2.5 GHz CPU frequency (see figure 3.9a).

Core Out-of-Order capabilities

In Table 3.1, we define four configurations to model different levels of core Out-of-Order (OoO) capabilities. The aggressive and low-end configurations are extreme cases that differ considerably from the usual state-of-the-art type cores found in HPC systems. Between those two, we simulate medium and high configurations to try to model cores with similar features to the ones we find in current server processors.

In terms of performance (see Figure 3.7a), for the majority of applications, low-end architectures are 35% slower than aggressive OoO configurations; 60% slower in the case of Specfem3D. Additionally, intermediate configurations suffer less than 5% slowdown in all applications except Specfem3D. We can appreciate similar results for 32 and 64 cores except in HYDRO and LULESH. As we mention in Section 3.4.1 applications with low parallelism leave many idle cores throughout the whole execution, therefore task distribution in 32 and 64 core configurations is the same.

The low-end pipeline configurations consume around 50% less power than its high-end counterparts (see Figure 3.7b), but as we discussed, these power savings come at a steep cost in terms of performance. On the other hand, intermediate *high* and *medium* OoO configurations consume 18% and 20% less power, respectively, across all applications, while still attaining performance that is close to the *aggressive* OoO pipelines. Therefore, the additional power that the aggressive cores consume is not translated into significant performance improvements, which makes the *high* and *medium* OoO configurations better design points for a good trade-off of performance and energy consumption (see Figure 3.7c). Additionally, we can observe that heavy memory constrained applications such as LULESH are able to obtain great energy savings since the impact in performance of the core compute capabilities is minor.

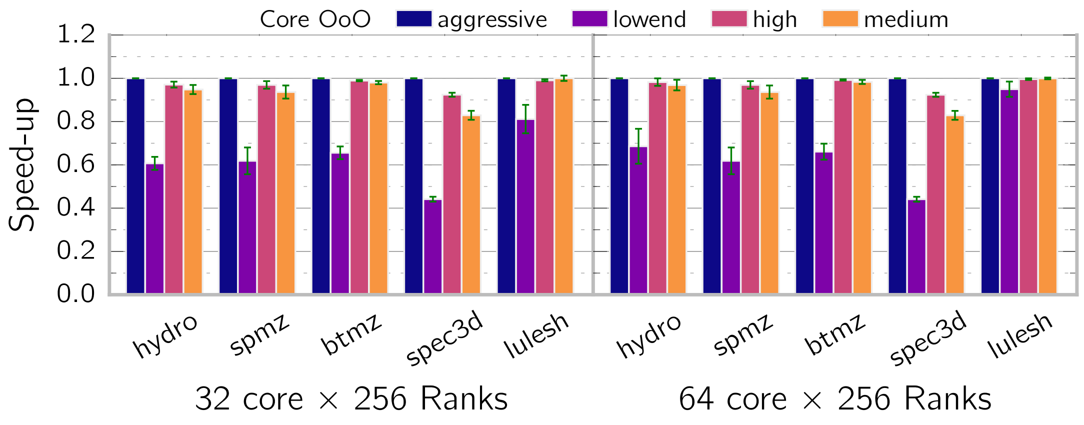
Memory channels

We evaluate memory configurations with four and eight DDR4 memory channels.

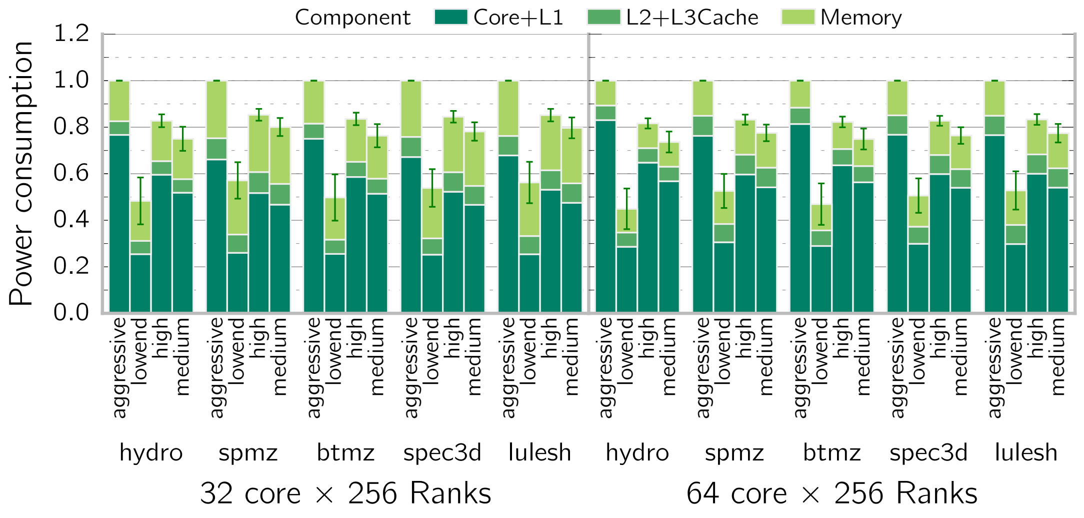
Figure 3.1 compares several memory metrics of our applications running on 64 core processors. Of all five, we find that only Specfem3D and LULESH have considerable high bandwidth requirements. Although Specfem3D requires more memory bandwidth than LULESH when simulating on a single core processor, at 64 cores it is the other way around: LULESH presents much better performance scaling and the usage of memory bandwidth scales accordingly. In Figure 3.8a we observe that despite the high bandwidth requirements of Specfem3D, increasing the number of memory channels from four to eight does not yield performance benefits. On the other hand, LULESH achieves up to 60% average speedup at 64 cores. Due to its very high memory bandwidth utilization, only LULESH takes advantage of having extra memory channels.

Upgrading from four to eight memory channels and populating them with DRAM DIMMs increases total DRAM power consumption by almost 100%. Nonetheless, in 64 core configurations, the impact of the memory component of this extra DIMMs over the overall total node power consumption is roughly 10%. As we see in Figure 3.8c LULESH, as a memory bound application, obtains on average 30% energy savings with eight channels.

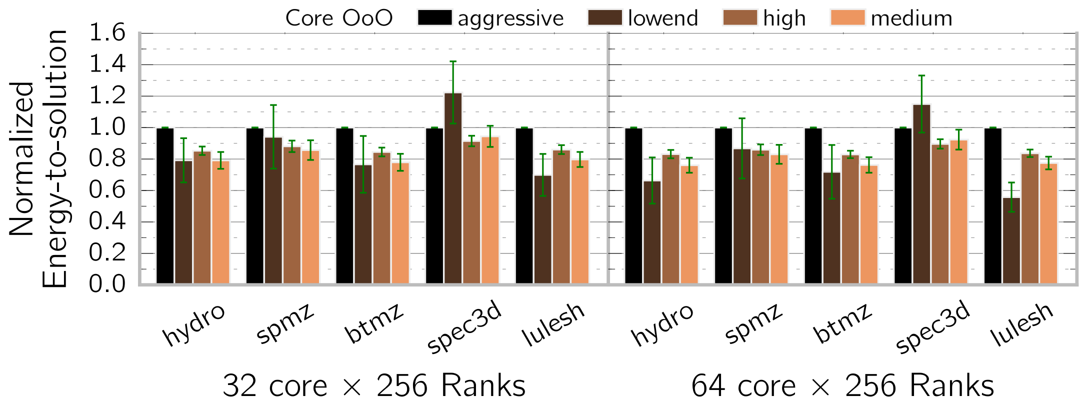
As a side note, memory bandwidth consumption is another computational characteristic that is expected to change significantly on improved versions of applications with better parallelized codes capable of scaling efficiently up to 64 cores. For example, if SPMZ was able to scale up to 64 cores with reasonable efficiency, it would demand more memory bandwidth than our four channel configurations are able to provide and we would obtain clear benefits on eight channel configurations. Another remark to consider is that, with the same micro-architectural



(a) Performance

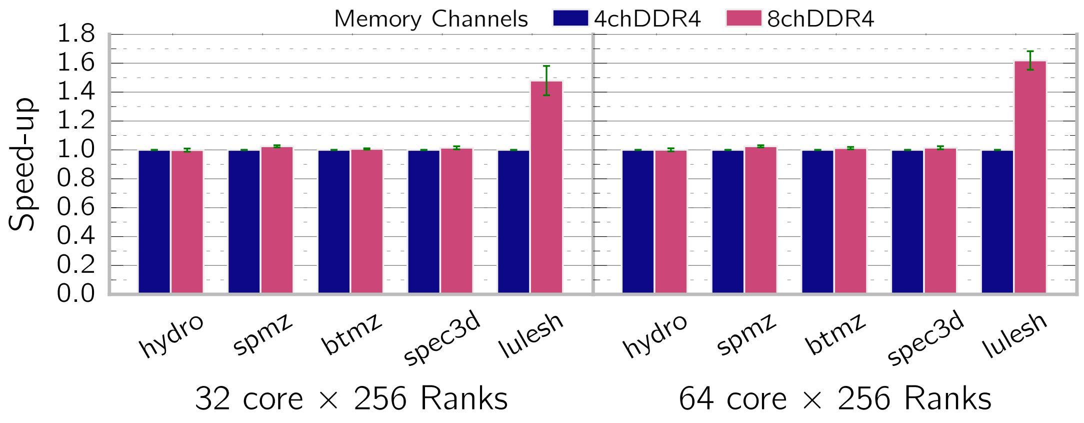


(b) Power dissipation

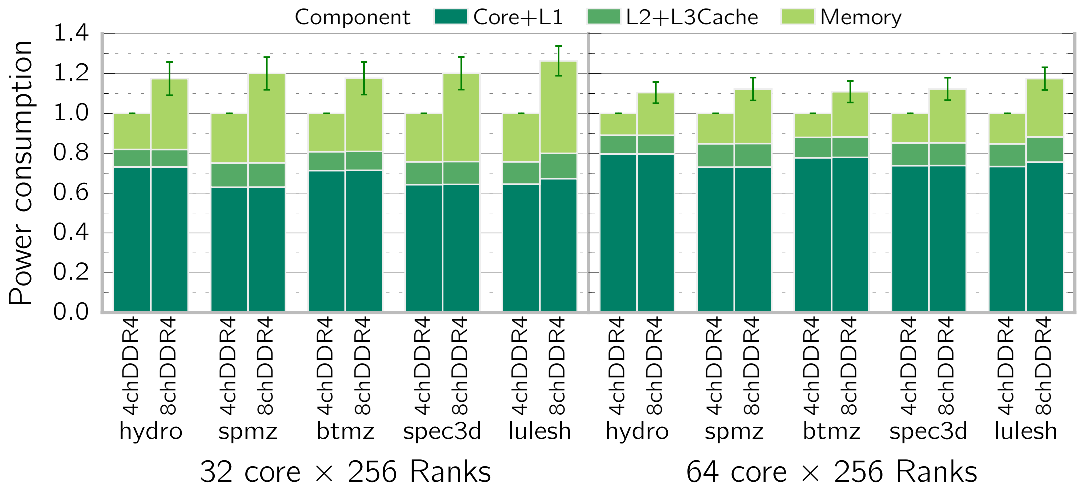


(c) Energy-to-solution

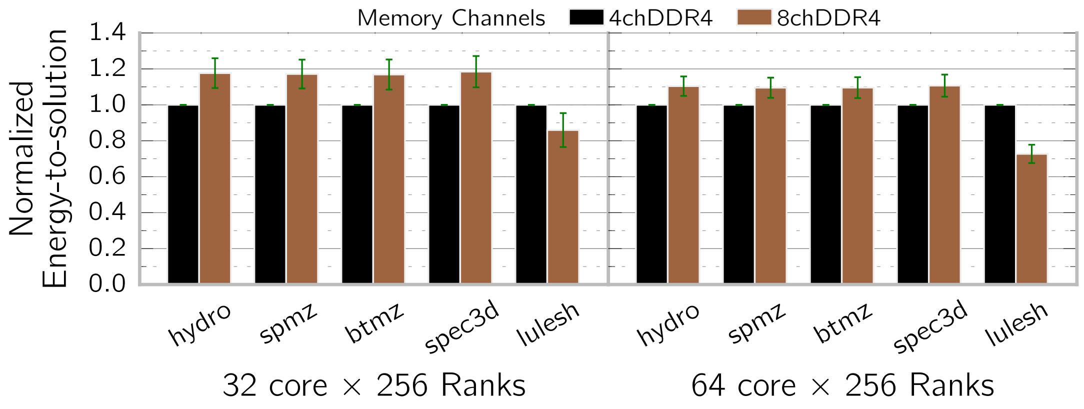
Figure 3.7: Average simulation results varying core OoO structures. Normalized to aggressive configurations.



(a) Performance



(b) Power dissipation



(c) Energy-to-solution

Figure 3.8: Average simulation results increasing the number of memory channels. Normalized to four channel configurations.

configuration, the power consumed by memory can vary up to 20% from one application to another. If we compare HYDRO with applications with similar core occupancy such as LULESH, we observe that the total processor power consumption varies up to 5% in 64 cores configurations. Applications with many idle cores due to not having enough task level parallelism, like SPMZ at 64 cores, vary up to 20% difference in power consumption compared to HYDRO.

Frequency

We evaluate CPU frequency values from 1.5 to 3.0 GHz. For each step in frequency, we provide McPAT with adequate voltage parameters to scale up voltage accordingly to 22nm process technology. It is important to note that TaskSim uses the same frequency for all the components of the chip, so the frequency at which L1, L2, and L3 caches run is the same as the CPU clock.

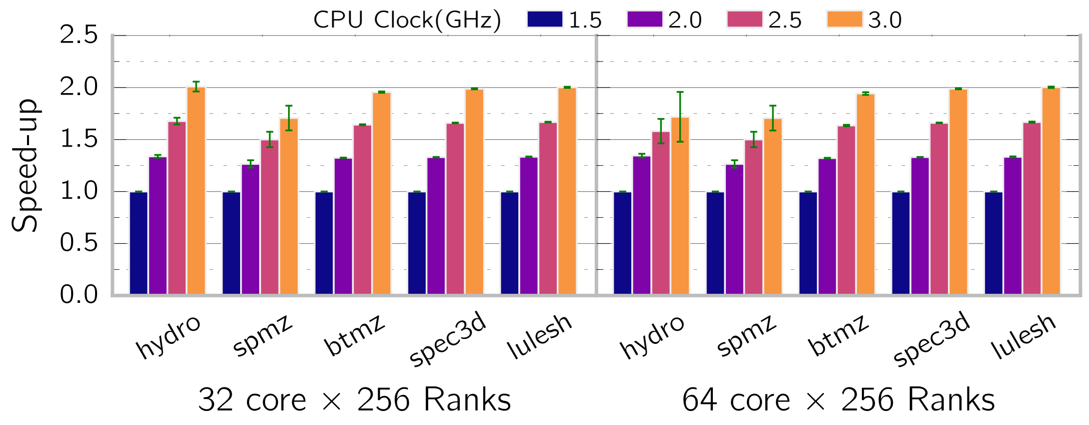
Figure 3.9a shows that all applications except HYDRO scale their performance linearly as frequency increases. HYDRO encounters a scheduling bottleneck at high frequencies, tasks are too small and the threads are not able to schedule tasks fast enough. This is a TaskSim limitation as OpenMP/OmpSs runtime event timings are taken from the original trace. In real systems, this issue is not expected to happen.

Regarding power, Figure 3.9b shows that when comparing 1.5 to 3.0 GHz there is a 2$\times$ increase in performance and a 2.5$\times$ increase in power consumption. It is almost linear so we can consider that adding 1% in performance will increase by 1.25% the power consumption. Commonly known, scaling up frequency is a good way to obtain higher performance but it has a high power consumption cost. Frequency is a key aspect to consider and balance the different clock frequencies of the different hardware components in a compute node.

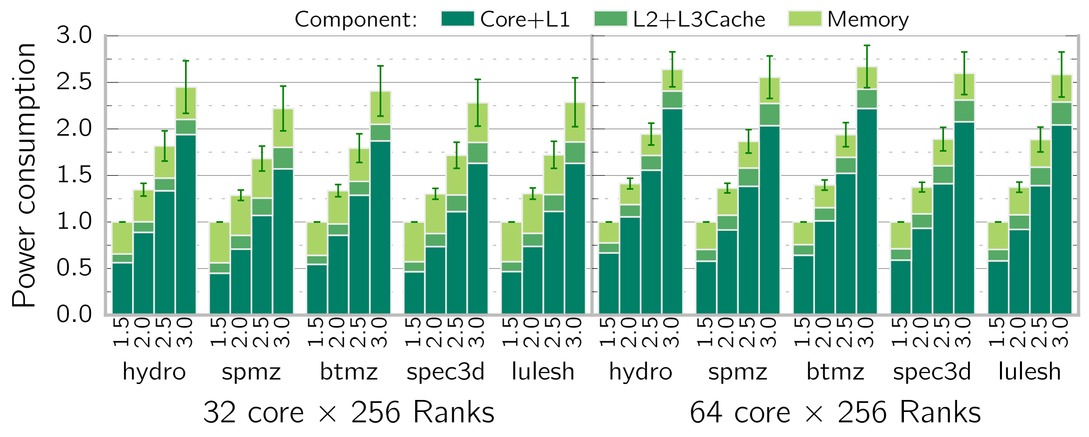
3.4.3 Principal Component Analysis

In Section 3.4.2, we provide average performance and standard deviation to measure the individual impact of each architectural feature. Even if bar plots are a convenient and straightforward way to visualize our experimental campaign, they do not provide insight about the performance tradeoffs between different architectural parameters. To explore those, we use Principal Component Analysis (PCA).

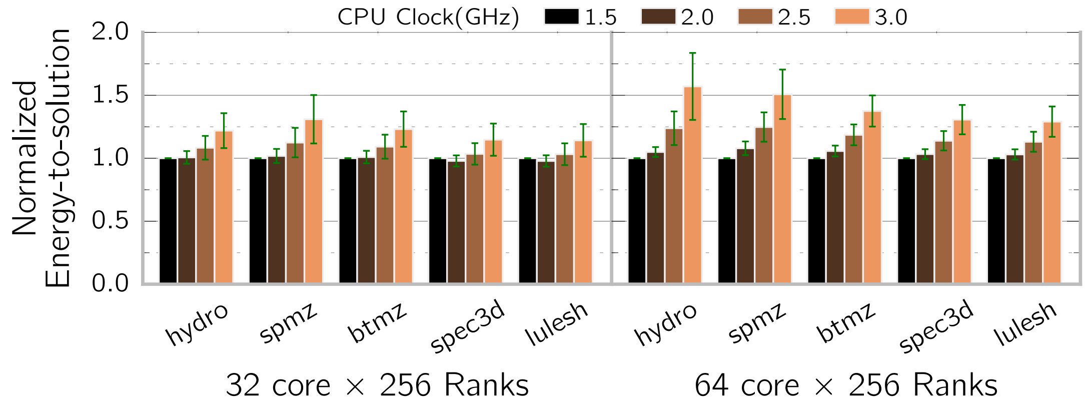
For each application, we find the principal components considering five variables: OoO capacity, number of memory channels, SIMD width, cache size and the number of cycles of that simulation.



(a) Performance



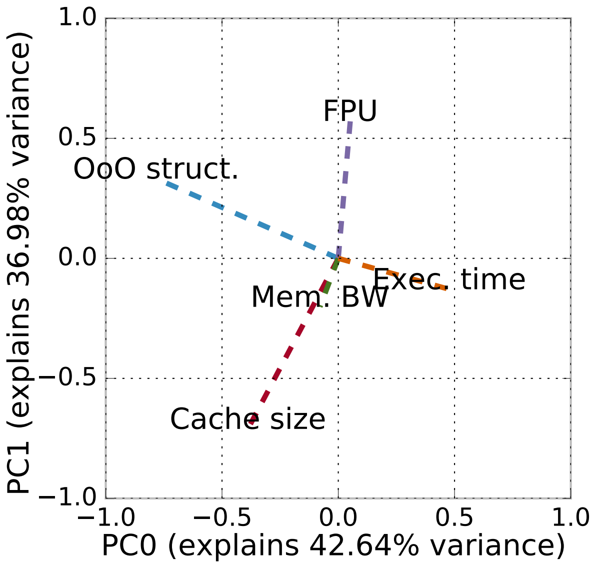
(b) Power dissipation



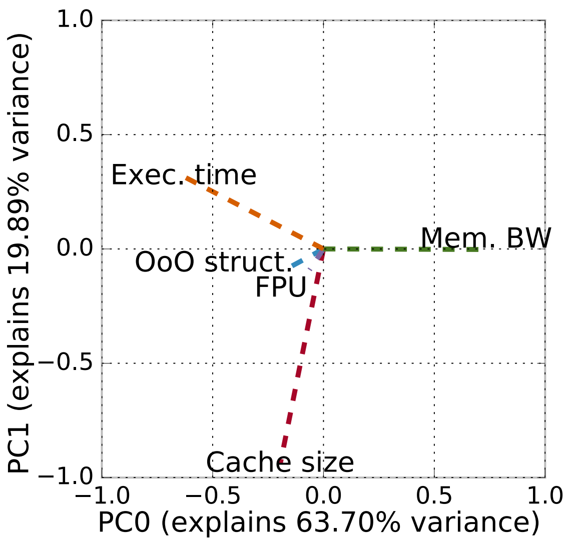
(c) Energy-to-solution

Figure 3.9: Average simulation results increasing CPU Clock Frequency. Normalized to 1.5GHz configurations.

Next, we discuss our results when applying PCA to HYDRO and LULESH. Other applications show similar trends and insights. We consider just the 2GHz CPU clock frequency and 64-core simulations.



(a) HYDRO



(b) LULESH

Figure 3.10: PCA results: performance correlation between different architectural paremeters.

Figure 3.10 shows the two most relevant principal components (PC) labeled as *PC0*, in the x-axis, and *PC1* in the y-axis. For the case of LULESH, PC0 explains more than 60% of the variance and it clearly shows that the memory bandwidth parameters evolve in a similar way as the total number of cycles. They evolve in an opposite way in the sense that an increase in memory bandwidth implies a reduction in the total amount of CPU cycles that LULESH requires. The cache size parameter has also a non-negative PC0 value, which means that it also related to the total cycles parameters for the case of LULESH, although not as much as the memory bandwidth. The other two considered parameters, OoO capacity and SIMD width, have no contribution to the PC0 variable for the case of LULESH, which means that the total cycles evolution is not related to them whatsoever.

In the case of HYDRO, the PC0 axis stands for 42.64% of the variance. Both the OoO capacity and the total number of cycles are major contributors to the PC0, which implies that they evolve in a tight and opposite way. The larger are the OoO capacities, the smaller becomes the total number of cycles for the case of HYDRO.

SPMZ specific config.				
Label	**Core OoO**	**FP Unit**	**Cache(L3:L2)**	**Memory**
DSE Best.	Aggressive	512-bit	96MB:1M	8-Ch. DDR4
Vector+	High ↓	1024-bit ↑	64MB:512kB ↓	4-Ch. DDR4 ↓
Vector++	High	2048-bit ↑	64MB:512kB	4-Ch. DDR4

LULESH specific config.				
Label	**Core OoO**	**FP Unit**	**Cache(L3:L2)**	**Memory**
DSE Best.	High	512-bit	96MB:1M	8-Ch. DDR4
MEM+	Medium ↓↓	64-bit ↓↓	64MB:512kB ↓	16-Ch. DDR4 ↑
MEM++	Medium	64-bit	64MB:512kB	16-Ch. HBM ↑

Table 3.2: Application-specific architectural configurations. All of them in a 64 core setting running at 2GHz.

3.4.4 Unconventional Configurations

We test two additional pairs of unconventional configurations for the SPMZ and the LULESH codes (see Table 3.2). We select these two applications as the results shown in Section 3.4.2 clearly show that SPMZ and LULESH are very sensitive to the SIMD register size and the memory bandwidth, respectively. In particular, results in Section 3.4.2 show how SPMZ benefits significantly from increasing the SIMD widths while other parameters such as the size of the OoO structures, cache and memory bandwidth have a minor impact in its performance.

Taking into account these observations, we simulate parallel executions of SPMZ considering architectures with increasing SIMD widths of 1024- (*Vector+* configuration) and 2048-bits (*Vector++* configuration) while keeping the rest of the architectural features settings that give the best possible performance-power tradeoff. LULESH is a heavily memory-bound application which does not benefit from floating point computing capacity. We test both high-bandwidth 16-Channel DDR4 (*MEM+*) and HBM (*MEM++*) configurations. For all results presented in this section, we compare the unconventional configurations against the best performing configuration in terms of execution time of Section 3.4.2 (*BEST-DSE*) running on 64 cores at 2GHz.

As we see in Figure 3.11, compared to *BEST-DSE*, the *Vector+* configuration for the case of SPMZ achieves a performance increase of 1.13× with a similar increase in power while the more aggressive *Vector++* configuration obtains a performance benefit of 1.43× but incurs in

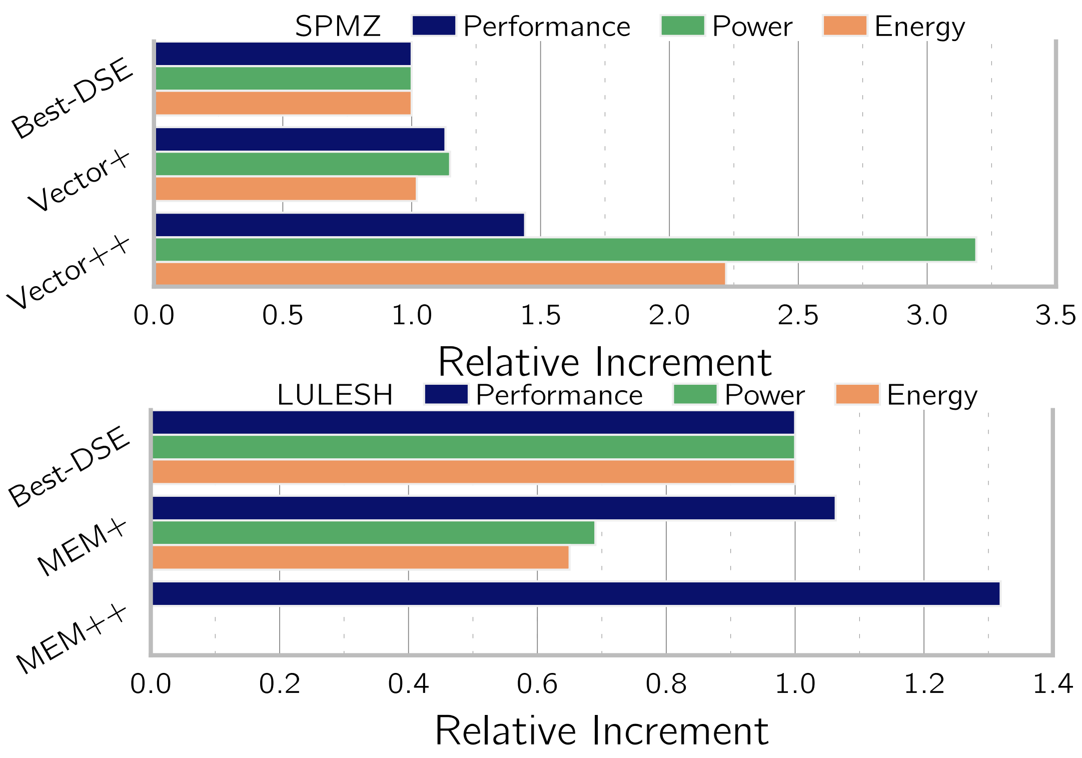


Figure 3.11: Performance, power and energy-to-solution of application specific configurations. Normalized to the best result in Section 3.4.2 .

additional power consumption equivalent to 3.14× the baseline. Overall, the *Vector++* suffers a pronounced 2.5× increase in energy-to-solution.

With respect to LULESH, we achieve a 47% reduction in energy-to-solution by using narrower FPUs units and a performance increase of 7% by doubling up the memory bandwidth. Moreover, if we consider very low latency memory (*MEM++*), we can further achieve up to 1.30× speedup over *DSE-BEST*. It is not possible to provide energy measurements regarding the *MEM++* configuration due to the lack of data describing the energy consumption of the HBM technology, although previous studies indicate that it would consume less power than the *MEM+* configuration [41].

Chapter 4

Efficiently Running SpMV on Long Vector Architectures

In the previous chapter, we analyze with simulations the interactions of specialized architectures to improve efficiency in HPC workloads. However, we observed the particular case of LULESH uncapable of taking advantage of wide SIMD units, thus, the most efficient hardware configuration required to downsize the VPU and increase the available memory bandwidth. As we note in Section 1, scientific computation with scalar units has a significant energy overhead and there is a large margin for improvement if we are able to exploit SIMD and vector units.

This chapter, takes on the issue of efficiently vectorizing challenging HPC codes on long vector architectures, in particular, our work optimizing the SpMV numerical kernel in the NEC Vector Engine. Previously in Section 2.2.1 we introduce the background of SpMV matrix formats. The rest of the chapter is organized as follows: Section 4.1 provides an introduction to the topic and the goals of this segment of the thesis; Section 4.2 contains an overview of other relevant related work; in Section 4.3 we explain in detail the added contributions of this chapter; in Section 4.4 we describe our experimental setup; in Section 4.5 we present and analyze the results of our experiments.

4.1 Introduction

The Sparse Matrix-Vector (SpMV) product is an ubiquitous kernel in the context of High-Performance Computing (HPC). For example, discretization schemes to solve Partial Differential Equations (PDE), like finite difference or finite element methods, produce linear systems with a highly sparse matrix. Such linear systems are typically solved via iterative methods, which require an extensive use of the SpMV kernel across the whole execution. In addition, emerging workloads from the data-analytics area require the manipulation of highly irregular and sparse matrices via SpMV. Therefore, the efficient execution of this fundamental linear algebra kernel is of paramount importance.

The performance of the SpMV $y = Ax$ operation is strongly correlated to several factors. First, accesses to data structures containing the A matrix and the y vector are typically regular, which means that they benefit from hardware resources like memory bandwidth capacity and structures like hardware pre-fetchers. The accesses on the x vector are driven by the sparsity pattern of A, which makes them irregular and hard to predict. In addition, x is the only data structure involved in SpMV where some degree of data reuse can be exploited, although the irregular nature of its accesses makes it hard to fully exploit these reuse opportunities. Another important performance aspect when parallelizing SpMV is the control flow divergence driven by the different number of non-zero entries of the A matrix. Such divergence can significantly limit the SpMV performance on high-end numerical accelerators like Graphic Processor Units (GPUs) or long vector architectures.

Many different approaches have been proposed to efficiently store sparse matrices and efficiently run SpMV. One of the most common approaches, Compressed Sparse-Row (CSR), efficiently stores sparse matrices and enables simple stride-1 memory access patterns on A and y. However, accesses on x are highly irregular. In addition, CSR suffers from flow divergence issues since each row-vector product depends on the number of non-zeros it contains. As such, CSR is not an appropriate format to manipulate sparse matrices on GPUs or long vector architectures. Its column counterpart, Compressed Sparse-Column (CSC), displays very similar issues.

Other approaches aim to mitigate the drawbacks of CSR by enlarging its storage requirements to increase the locality on x. SELL-C-σ [48] and ELLPACK Sparse Block [54] make use of row sorting and column blocking to improve both storage requirements and locality on x. This chapter demonstrates that, although some of these approaches are very good abstractions to represent and manipulate sparse matrices, there are many unexploited opportunities to improve their performance on long vector architectures.

The main contributions of this chapter are:

- We develop a highly efficient SpMV implementation for long vector architectures leveraging the state-of-the-art SELL-C-σ format. Our implementation reaches 117 GFlops saturating 78% of the peak memory bandwidth in a NEC SX-Aurora Vector Engine (VE) [46]. This accelerator leverages a vector instruction set that goes beyond the SIMD approach (e.g., AVX-512 in x86 CPUs) with 16-kbit long vector registers and instructions. This is the first time that the SELL-C-σ format is implemented and evaluated on an architecture leveraging long vectors and predication. Our implementation shows how to deal with sparse data structures on emerging vector ISAs such as the Arm SVE or the RISC-V vector extension.

- We implement, evaluate, and discuss the performance impact of several optimizations to run SpMV on long vector architectures. We improve the SELL-C-σ baseline performance by 12% on average.

- We compare our novel approach for long vector architectures with other state-of-the-art approaches targeting SpMV in cutting-edge multi-core CPU and GPU devices. We demonstrate that our approach is **3.02$\times$ and 1.72$\times$** faster, respectively, than these approaches since it achieves an outstanding average SpMV efficiency equivalent to 4.19% of the peak performance.

4.2 Related work

Since SpMV is a kernel of paramount importance in several algorithms of scientific computing, a large body of research about SpMV optimization on many architectures has been published in the last twenty years. Most recently, the research community is focusing on developing efficient SpMV implementations targeting emerging architectures with different degrees of parallelism, e.g., high number of cores and long SIMD or vector units. However, most of those studies are limited to 8-elements SIMD units in CPUs or 32-elements warps in GPUs [3, 54, 12, 48, 13, 52]. Even if these approaches consider state-of-the-art Intel CPUs with the AVX-512 extension and the latest GPUs, they fall short compared to the 256 double-precision elements vector platform, SX-Aurora, we consider in our study.

Several studies propose new formats capable of exploiting SIMD/vector units. They create blocks of contiguous elements by adding padding of zeroes [53, 52, 13]. However, as these formats rely on having big enough groups of NNZ elements close to each other to be efficient, they are not suitable for long vector architectures.

Both Beamer et al. [6] and Buono et al. [9] propose a two-step SpMV algorithm that shows good performance improvements for big-data matrices. Using the code provided by the authors of this previous work [6], we were able to reproduce the results in an x86 system similar to the one used in their study. However, we were not able to develop an efficient implementation using intrinsics close in performance to our SELL-C-σ code in the VE.

4.3 Contributions

In this section, we describe our contributions to improve the performance of SpMV. Section 4.3.1 focuses on format optimizations, while Section 4.3.2 describes our contributions to speedup SpMV on long vector architectures.

4.3.1 Implementing the SELL-C-σ format

SELL-C-σ is an efficient sparse matrix format for vector architectures. Because of that, we choose SELL-C-σ as a starting point to develop our SpMV implementation for VE. From there, we revisit and adapt some optimizations previously proposed in the literature extending them with new approaches targeting long vector architectures. In detail, we explore: *i)* the adequate sorting strategy based on the trade-off between performance and preprocessing overhead as the σ parameter increases; *ii)* the use of task-based parallelism and the impact of the task granularity in the scaling performance of SELL-C-σ; and *iii)* the impact of column blocking in matrices to improve locality on vector x.

Sorting strategies The SELL-C-σ format requires matrix rows to be sorted in terms of their *Number of Non-Zero* elements, *NNZ*, which may incur a significant overhead. We mitigate the cost of SELL-C-σ in terms of sorting by applying the Radix Sort algorithm. It has a worst-case cost of $\mathcal{O}(w \times n)$, where w is the number of digits of the largest value, and is well suited for sorting integers in long vector architectures [89]. Since reordering the whole matrix can be costly, SELL-C-σ splits the matrix into several subsets of rows, each one of them sorted independently. Each subset can be easily sorted in parallel by using basic OpenMP *parallel for* directives. We refer to the number of rows per subset as *reorder window*; represented with the σ parameter. We select an optimal σ size for our tests by running our SELL-C-σ implementation with the Divergent Flow Control optimization described in Section 4.3.2. We evaluate the trade-off between the performance gains of SELL-C-σ during the SpMV computation, and the additional time sorting the matrix. We use the optimal task partitioning for each matrix.

In some particular cases (see Figure 4.1), a bigger reorder window can boost performance up to 2×. Overall, a σ = 16K is enough to perform optimally with up to 30% improvement compared to σ = 256. In general, reordering the matrix in groups of 16K rows takes about 40% to 50% additional time during the preprocessing phase. In our experiments, converting most matrices to SELL-C-σ takes the equivalent in time of 10 to 15 iterations. We use the data structure, *row_order[num_rows]*, that keeps the initial row order of the matrix, to store back the results in y.

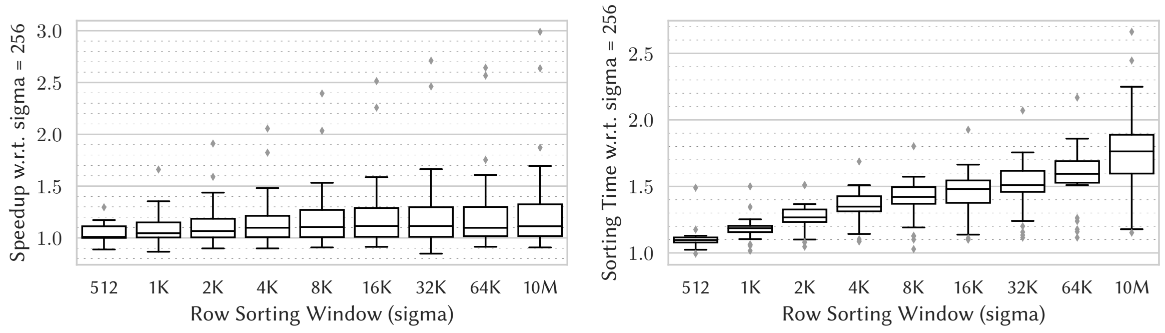


Figure 4.1: Performance increase and preprocessing overhead increasing the sigma window.

Task-Based Parallelism We orchestrate the parallel execution of our workloads by splitting them into several sequential pieces, called tasks, and letting an underlying runtime system to dynamically schedule them as computing resources become available. OpenMP supports this parallel execution model via task constructs [8]. In this context, the programmer must assign an appropriate amount of work to each parallel task to expose a significant amount of concurrency to the parallel hardware while, at the same time, avoid incurring too much overhead in terms of task creation or synchronization cost.

As the ideal task granularity depends on the input matrix, we perform a scalability test up to 8 OpenMP threads, with a different number of tasks ranging from 8, which would be the minimum for 8 cores, to 256 in power of two steps. Tasks deal with a similar amount of work in terms of matrix non-zero coefficients for all domain decomposition configurations. The results of these experiments indicate the level of task parallelism needed to achieve the best workload balancing without creating too many task instances. In general, we observe that most matrices achieve its best performance when the workload is divided between 8 to 64 tasks. Only a few experiments (around 10%) show better performance when the workload is divided between 128 and 256 tasks. In such cases, the benefits are less than 1% compared to dividing it by 8 to 64 tasks. For each matrix, we use the level of task parallelism that achieves the highest performance. We use such configurations in the performance experiments described in Section 4.5.1.

Column Blocking As we mention in Section 2.2.1, previous work [54] analyzes the effects of column blocking on SpMV. However, that study is limited to Intel Xeon Phi (KNC) which has a SIMD width of 512-bits. We extend our SELL-C-σ base implementation with a similar column blocking approach in order to test its suitability with longer vectors. We evaluate the impact of this optimization in Section 4.5.1.

4.3.2 Optimizations Targeting Long-Vector Architectures

In this section, we describe the optimizations we propose targeting long vector architectures like SX-Aurora and others. Our proposals to accelerate SpMV on long vector architectures are: *i)* the use of cache allocation to improve the reuse of x and deprioritization of store dependencies; *ii)* divergence flow control adapting the length of vector operations to avoid loading and computing *zero-padded* elements; *iii)* enabling loop unrolling in SELL-C-σ using partial loop fusion; *iv)* efficient computation of gather and scatter addresses with special instructions.

In Listing 4.1 we provide a detailed pseudo-code of our basic implementation of SELL-C-σ for the VE. The rest of the optimizations are built on top of this code. We optimize our codes using low-level intrinsics to leverage NEC specific architectural features such as arithmetics using long vectors and memory access policies.

Cache allocation and store relaxation policies In the context of SpMV, data structures corresponding to the matrix A have very different data access patterns and reuse properties than the vector x. Indeed, data structures containing the matrix non-zero elements and their corresponsing column indices, arrays *values* and *column_indices*, are accessed via a very simple stride-1 memory access pattern and never reused, while vector x is accessed randomly but often reused. One of the fundamental aspects to achieve good performance for SpMV is to exploit all opportunities for data reuse to alleviate the pressure on the memory bandwidth. Therefore, it is critical to reuse all vector x coefficients stored in the cache hierarchy as much as possible. Long vector architectures typically offer support to explicitly guide from the source code the cache replacement policy [46, 14]. We exploit such support to indicate to the architecture that cache lines containing pieces of both *values* and *column_indices* arrays must be evicted before than any other cache line. Therefore, we reduce the chances of evicting a cache line occupied by the vector x, which may be reused. We refer to this policy as *Non-Cacheable* load.

Another important aspect is the way we handle stores, which exploits the memory access pattern driven by the SELL-C-σ format. We use a register to accumulate the intermediate results within a *slice*. Once the pass over a certain slice finishes, the vector register holding intermediate results is stored to several memory addresses that will not be accessed again until many other slices are computed. Storing the whole vector register in memory is done via a scatter store instruction. Typically, dependencies on scatter store instructions are computed considering the whole range between the initial and final addresses they access, which can delay subsequent memory instructions that access addresses within this range, although not necessarily the same ones as the scatter store. Since we know that memory instructions immediately following the scatter do not access the same memory addresses, we instruct the

```
void sell_c_sigma_mv(mtx, x, y, num_rows) {
  int maxvl = 256;
  /* Outer loop: iterates over rows in the mtx */
  for (int row = 0; row < num_rows; row += maxvl) {
    int vl=((num_rows - row) < maxvl)? (num_rows - row): maxvl;
    vr res = vxor(res, res, vl);
    /* Set pointers to values and column indices */
    int sli = row / maxvl;
    int nnz_idx = mtx.slices_ptr[sli]
    double *values = &mtx.values[nnz_idx];
    int *col_indices = &mtx.column_indices[nnz_idx];
    /* Compute scatter addresses */
    vr y_sc_addr = vload(8, &mtx.vrow_order[row], vl);
    y_sc_addr = vshiftadd(y_sc_addr, 3UL, y, maxvl);
    /* Inner loop: iterates over columns in the slice */
    int swidth = mtx.slices_width[sli];
    for (int i = 0; i < swidth; i++) {
      /* Load mtx values and column index */
      vr val = vload(8, values, vl);
      vr col = vload(8, col_indices, vl);
      vr xgather_addr = vshiftadd(col, 3, x, vl);
      /* Gather and multiply */
      vr x_val = vgather(xgather_addr, vl);
      res = vmultiplyadd(res, x_val, val, vl);
      values += maxvl; col_indices += maxvl;
    }
    vscatter(res, y_sc_addr, vl);
  }
}
```

Listing 4.1: Basic Vector Engine SELL-C-σ implementation

hardware to not check dependencies across the scatter and some subsequent instructions. We refer to this policy as Store *Overtake*. A memory fence instruction is inserted to define where this relaxation period finishes. Modern vector architectures support this kind of memory scatter dependencies relaxation. [79].

Table 4.1 summarizes how we apply these policies in our SELL-C-σ implementations to increase performance.

Handling Flow Divergence by Adapting the Vector Length One major aspect when computing SpMV on long vector architectures is the management of flow divergence. It mainly arises from the varying number of non-zero elements per matrix row, which forces the loop iterating over matrix non-zero coefficients to produce different control flow scenarios per row.

Table 4.1: Policies applied to optimize SELL-C-σ.

Loop	Vector Mem. Access	Policy applied
Inner	Load Values	Non-Cacheable
Inner	Load Col. Indices	Non-Cacheable
Inner	Gather X coef.	< none >
Outer	Load Row Order	< none >
Outer	Scatter result to Y	Overtake

Therefore, vectorizing matrix rows containing very different numbers of non-zero elements wastes computing resources, particularly the ones assigned to rows with small amounts of non-zero coefficients.

One very popular technique to handle flow divergence is the use of predicated registers. They contain an array of bits specifying whether its corresponding vector element is zero or not. Predication is supported by commercial SIMD ISAs like AVX512 [58] or SVE [77] and is a valuable approach to let the compiler vectorize irregular loops. However, it requires a large amount of storage, since every single vector element needs its corresponding predicated mask, and many of its implementations do not avoid processing zeroed vector entries but just discard the output of these meaningless computations.

We propose a new approach to handle flow divergence on vector architectures that relies on the ratio between the number of vector elements v_{el} and the number of vector lanes v_{la}. We call this new approach *Divergent Flow Control* (DFC). Vector architectures have the capacity of processing vector elements in batches of v_{la}, that is, they require processing $\frac{v_{el}}{v_{la}}$ batches of v_{la} to finish the pass over the whole set of v_{el} elements. Our approach instructs the hardware to process just *MAXnnz* elements of the vector, which implies that the hardware will need to process just $\lceil \frac{MAXnnz}{v_{la}} \rceil$ batches of v_{la} elements, where *MAXnnz* is the maximum number of non-zeros over all rows involved in the vector operation. Importantly, this information can be defined in just 8 bits. For example, the SX-Aurora architecture, which is described in Section 4.4, has 256-element vector registers, hence the possible values are $1 \leq MAXnnz \leq 256$, which means that in a single byte we can encode the vector length that we need for each vector instruction.

As we describe in the previous section, we modify our code to enable this optimization by adding an extra data structure, the `active_lanes` vector, which contains the *MAXnnz* for every *column-wise* vector operation to do inside each slice. The rightmost drawing of Figure 2.2 shows a basic example of how *MAXnnz* are counted and stored in a vector. The computation of zero-padded elements in yellow is avoided.

Applying loop unrolling Loop unrolling is a well-know optimization to reduce control flow overhead and maximize the use of the register file. Our loop unrolling approach particularly maximizes locality on vector x by increasing the depth of vertical or *column-wise* vector operations while, at the same time, maximizes data reuse on the register file. Figure 4.2 shows how matrix elements are accessed in different order if unrolling is applied. This is a very natural optimization in our context since SELL-C-σ also targets the same kind of locality on vector x without writing back to memory partial results until all the rows in a slice are completed.

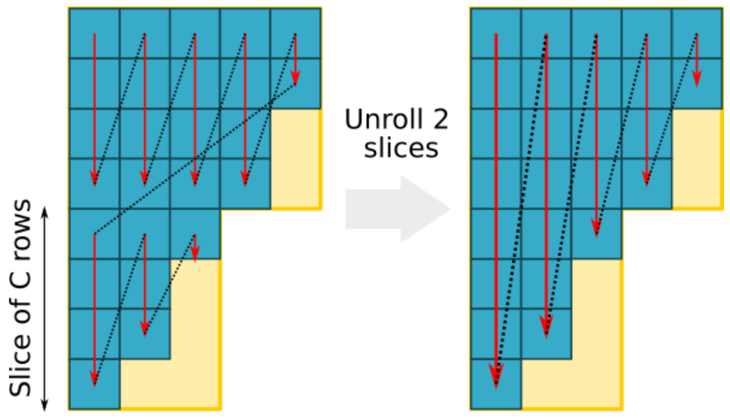


Figure 4.2: Matrix access pattern when unrolling is applied.

```
void sell_c_sigma_mv_unroll(mtx, x, y, num_rows) {
  int maxvl = 256;
  for (int row = 0; row < (num_rows - 511); row += 512) {
    int sli = row / maxvl;
    /* Set pointers to values and column indices */
    double *values1 = &mtx.values[slices_ptr[sli]];
    double *values2 = &mtx.values[slices_ptr[sli+1]];
    int *col_indices1 = &mtx.col_idx[slices_ptr[sli]];
    int *col_indices2 = &mtx.col_idx[slices_ptr[sli+1]];
    vr res1 = vxor(res1, res1, maxvl);
    vr res2 = vxor(res2, res2, maxvl);
    int swidth = mtx.slices_width[sli];
    int swidth2 = mtx.slices_width[sli + 1];
    int al1 = active_lanes[sli];
    int al2 = active_lanes[sli + 1];
    vr y_sc_addr = vload(8, &mtx.vrow_order[row], maxvl);
    y_sc_addr1 = vshiftadd(y_sc_addr1, 3, y, maxvl);
    vr y_sc_addr2 = vload(8, &mtx.vrow_order[row+maxvl], maxvl);
    y_sc_addr2 = vshiftadd(y_sc_addr2, 3UL, y, maxvl);
    /* Partial loop fusion: Slices 1 & 2 */
    for (int i = 0; i < swidth2; i++) {
      /* Compute slice 1 */
      int nl_1 = vactive_lanes[al1++] + 1;
      vr val1 = vload(8, values1, nl_1);
      vr col1 = vload(8, col_indices1, nl_1);
      vr xgather_addr1 = vshiftadd(col1, 3, x, nl_1);
      vr x_val1 = vgather(xgather_addr1, nl_1);
      res1 = vmultiplyadd(res1, x_val1, val1, nl_1);
      /* Compute slice 2 */
      int nl_2 = vactive_lanes[al2++] + 1;
      vr val2 = vload(8, values2, nl_2);
      vr col2 = vload(8, col_indices2, nl_2);
      vr xgather_addr2 = vshiftadd(col2, 3, x, nl_2);
      vr x_val2 = vgather(xgather_addr2, nl_2);
      res2 = vmultiplyadd(res2, x_val2, val2, nl_2);
      /* Advance pointers */
      values1 += nl_1; values2 += nl_2;
      col_indices1 += nl_1; col_indices2 += nl_2;
    }
    /* Finish computing the remaining vector ops in slice 1*/
    for (int i = swidth2; i < swidth; i++) {
      /* Same code as above but only for slice 1 */
    }
    vscatter(res1, y_sc_addr1, maxvl);
    vscatter(res2, y_sc_addr2, maxvl);
  }
}
```

Listing 4.2: Manually unrolling the SELL-C-σ by 2 slices

An efficient implementation of loop unrolling is not straightforward in our context. In Listing 4.2, we show an example including unrolling two slices of the algorithm. This example also implements the *DFC* optimization that we describe above. The main issue when unrolling slices is that each slice may have a different width. However, we know that within the same row order window, each slice has a width less or equal to the previous one. That is, in on our example $swidth1 \geq swidth2$.

It is possible to fuse from the *bottom-up* the inner loop iteration space of the slices as we show in lines 21 to 39 in Listing 4.2. We must add an additional loop to handle the remaining elements of slice 1. This loop is represented in line 41 in our example. Note that, if unrolling is applied, the sigma reorder window has to be a multiple of the number of rows the unroll covers.

The unroll size is limited by the number of vector registers. Since the number of local variables increase each time we unroll, it is important to declare them in the most immediate context where they are consumed, which makes it easier for the compiler to manage register dependencies, apply architecture-specific optimizations and avoid spilling. Our implementation is able to unroll up to 8 times without producing any spilling access on a vector architecture with 64 architectural registers available.

Efficient computation of gather/scatter addresses Gather and scatter instructions fetch and store, respectively, to the addresses provided in a vector. In our implementation, gather is used to access non-contiguous elements of the x vector, while scatter is used to store the results back to the corresponding element of the y vector. The addresses are generated by multiplying the index of the vector element we access by the corresponding data type size. (e.g., by 8 if we use 64-bit elements). To handle the integer arithmetic involved in address computation, a multiply instruction followed by an add operation can be replaced by a single shift and add instruction. Listing 4.2, (line 19) exemplifies how we apply this optimization.

To evaluate the impact of our proposals, we created five implementations which incrementally include the optimizations we describe above. The optimizations included in every implementation are specified in Table 4.2. In addition to those five, we also created an implementation of SELL-C-σ including the column blocking, sorting strategies, DFC and efficient computation of gather/scatter addresses, that we evaluate in a dedicated subsection of Section 4.5.1.

4.4 Methodology

In this section we introduce the hardware and software infrastructure used for our evaluation. The main system on which we evaluate our implementation is the **NEC SX-Aurora** (described in the following section), while for performance and energy comparison we consider:

- **Intel Xeon Platinum 8160 CPU** with 24 cores each running at 2.10 GHz. Each core houses 32 kB L1 and 1024 kB L2 data cache. The L3 cache is shared among the 24

Table 4.2: Optimizations applied on each of the implementations evaluated in the paper.

Optimization	SELLCS	SELLCS DFC	SELLCS U8-DFC	SELLCS U8-NC	SELLCS U8-NC-DFC
Sorting strategies	●	●	●	●	●
Task-Based Parallelism	●	●	●	●	●
Column Blocking					
Cache Allocation & Store relaxation pol.				●	●
Divergent Flow Control		●	●		●
Loop unrolling			●	●	●
Efficient gather/scatter address computation	●	●	●	●	●

cores and its size is 33792 kB. Also, each core is powered by an AVX-512 SIMD unit, allowing operating with registers of up to 512 bits (i.e., 16 floats, 8 doubles).

- **NVIDIA V100 (Volta) GP-GPU** with 84 Volta Streaming Multiprocessor (SM) running at a maximum frequency of 1.5 GHz. The 84 SMs share 6144 kB L2 cache and are connected to 4096 GB HBM2.

4.4.1 SX-Aurora Vector Engine

The NEC SX-Aurora Vector Engine (VE) is the latest incarnation of NEC's long vector architecture which combines SIMD and pipelining. Vector units and vector registers use a 32×64-bit wide SIMD front in an 8-cycles deep pipeline resulting in a maximum vector length of 256×64-bit elements or 512×32-bit elements. The VE10B processor used for this publication has been released in 2018. Its characteristics were presented at the IEEE HotChips 2018 [87], and the first performance evaluation was described in the same year [46]. Due to its 6 HBM2 8 high stacks the VE has a very high memory bandwidth of 1.22 TB/s out of the 48 GB on-chip RAM, shared by only 8 powerful cores. The VE10AE and VE10BE models released at the end of 2019 have improved the memory bandwidth to 1.35 TB/s but were not accessible for this publication's work.

Sparse matrix operations performance benefits of large memory bandwidth, but equally important are other characteristics of the processor like mechanisms for memory latency hiding or caches. Each of the 8 VE cores consists of a scalar processing unit (SPU) and a vector processing unit (VPU) and is connected to a common last level cache (LLC) of 16 MB. The core's bandwidth to the LLC is 406.9 GB/s, bidirectional, therefore the memory bandwidth (995 GB/s peak measured with STREAM) can be saturated by 4 cores. Each VPU has 64 architectural vector registers of 256×64-bit elements and the threefold amount implemented in hardware, used for register renaming. Three fused multiply-add vector units deliver a peak performance of 269 GLFOPS (double precision) per core at 1.4 GHz, and 307 GFLOPS for the VE10A model running at 1.6 GHz. The peak performance of the used VE variant is 2.15 TFLOPS, which is not impressive when compared to the latest GPGPUs, but the 0.56 byte/FLOP represent a well-balanced CPU with coarse-grain parallelism in 8 cores and fine-grain parallelism at vector level.

Vector Engines are integrated as PCIe cards into their host machines and offload the operating system functionality entirely to the host. They run in multitasking and multiprocessing mode and, like standard CPUs, programs can run entirely on the VE (native programming model), offload parts of code to the host machine (reverse offloading), or run on the host and offload compute kernels to the VE (accelerator, offload model). Heterogeneous programs can also be built with the hybrid MPI provided by NEC that connects processes running on the host and the VEs. In all cases programmers can use languages like C, C++, Fortran, and parallelize with MPI as well as OpenMP, while accelerator code can still use almost any Linux system call transparently.

The proprietary compilers from NEC support automatic vectorization aided by directives. They are capable of using most features of the extensive vector engine ISA [79] from high-level languages loop constructs. For the work presented in this paper, we needed even tighter control over VE features like vector masks generation and control, vector registers and LLC cache affinity of data. Therefore, we use the open-source LLVM-VE project [57], which supports intrinsics allowing full control over the generated code [56].

4.4.2 Experimental Setup

For our study, we select matrices frequently used in recent literature [13, 52, 48, 9] representing a wide range HPC application problems. Table 4.3 includes the list of matrices and some of its characteristics. All of them are available at SuiteSparse Matrix Collection repository[1].

For our SELL-C-σ implementations in the VE, we compile with LLVM-VE v1.8 and link NEC's proprietary compiler NCC v3.0.1. We also use the vendor-proprietary math libraries for performance comparisons between platforms: NLC 2.0, cuSPARSE v10.2, and MKL v2020.0 for the VE, NVIDIA V100 and Intel Xeon systems described above. All these libraries were released on 2019 or 2020. All codes are compiled with the *-O3* optimization level. For easy reproducibility, all the SpMV implementations, the benchmarking tool developed and the exact environment configuration files used for each system are provided in our repository[2].

[1]https://sparse.tamu.edu/

[2]https://repo.hca.bsc.es/gitlab/cgomez/spmv-long-vector

Table 4.3: Sparse matrices used in the evaluation.

Name	Row×Col	NNZ	NNZ/row	Density
scircuit	170K × 170K	958K	5	2.92E-05
mc2depi	525K × 525K	2.1M	3	5.71E-06
webbase-1M	1000K × 1000K	3.1M	3	3.00E-06
s4dkt3m2	90K × 90K	3.7M	41	4.53E-04
dense2	2K × 2K	4.0M	2000	1.00E+00
bmw7st-1	141K × 141K	7.3M	51	3.61E-04
torso1	116K × 116K	8.5M	73	6.28E-04
mip1	66K × 66K	10.3M	155	2.33E-03
fcondp2	201K × 201K	11.2M	55	2.73E-04
pwtk	217K × 217K	11.5M	52	2.39E-04
fullb	199K × 199K	11.7M	58	2.91E-04
halfb	224K × 224K	12.3M	55	2.45E-04
BenElechi1	245K × 245K	13.1M	53	2.16E-04
crankseg-2	63K × 63K	14.1M	221	3.46E-03
Si41Ge41H72	185K × 185K	15.0M	80	4.31E-04
TSOPF-RS-b2383	38K × 38K	16.1M	424	1.11E-02
msdoor	415K × 415K	19.1M	46	1.11E-04
bundle-adj	513K × 513K	20.2M	39	7.60E-05
ML-Laplace	377K × 377K	27.5M	73	1.94E-04
ldoor	952K × 952K	42.4M	44	4.62E-05
bone010	986K × 986K	47.8M	48	4.86E-05
af-shell10	1.5M × 1.5M	52.2M	34	2.25E-05
ML-Geer	1.5M × 1.5M	110M	73	4.85E-05
nlpkkt240	27M × 27M	760M	27	9.65E-07
12month1	12K × 837K	22.6M	1814	2.25E-03
spal-004	10K × 322K	46.1M	4524	1.46E-02

4.5 Evaluation

4.5.1 Performance of Long Vector Optimizations

In this section, we evaluate the performance impact of the different optimizations we describe in Section 4.3. We build them on top of our SELL-C-σ base implementation for the VE. As several studies show [12], SpMV performance is highly dependent on the sparse matrix structure. Therefore, our optimizations are not expected to deliver the same performance on all the matrices evaluated. To discuss our performance results, we consider several aspects like matrix size and density (see Table 4.3), and microarchitecture event information provided by the hardware Performance Monitoring Counters (PMCs). The PMCs available in the VE, report relevant metrics such as: total number of scalar and vector instructions executed, average vector instruction length, elapsed cycles, and cache miss ratios. We measure those and other PMCs right before and after the SpMV computation. We analyze these metrics in correlation with the performance obtained applying our optimizations.

Figure 4.3 shows GFLOP/s performance results considering all matrices described in Section 4.4.2 except the last two. We evaluate six different implementations of the SpMV kernel: *NLC, SELLCS, SELLCS-DFC, SELLCS-U8-DFC, SELLCS-U8-NC* and *SELLCS-U8-NC-DFC*. The *NLC* category represents results obtained with the math library developed by NEC which is particularly tailored for the VE.

Table 4.2 specifies which optimizations described in Section 4.3 are included in *SELLCS, SELLCS-DFC, SELLCS-U8-DFC, SELLCS-U8-NC* and *SELLCS-U8-NC-DFC*. Results in Figure 4.3, are executed in SX-Aurora running in a single VE, with optimal σ and task partitioning configurations as described in section 4.3. The performance measures the SpMV computation without any pre- or post-process.

There are missing values for the dense2 and nlptkk240 matrices in Figure 4.3. In the case of dense2, implementations containing the 8-slice unroll optimization require at least 2048 rows to execute correctly and this matrix has only 2000. In the case of nlptkk240, the implementation using NLC runs out of memory while trying to allocate the data structures. Despite these issues, we still include these matrices as we consider them valuable for evaluation purposes.

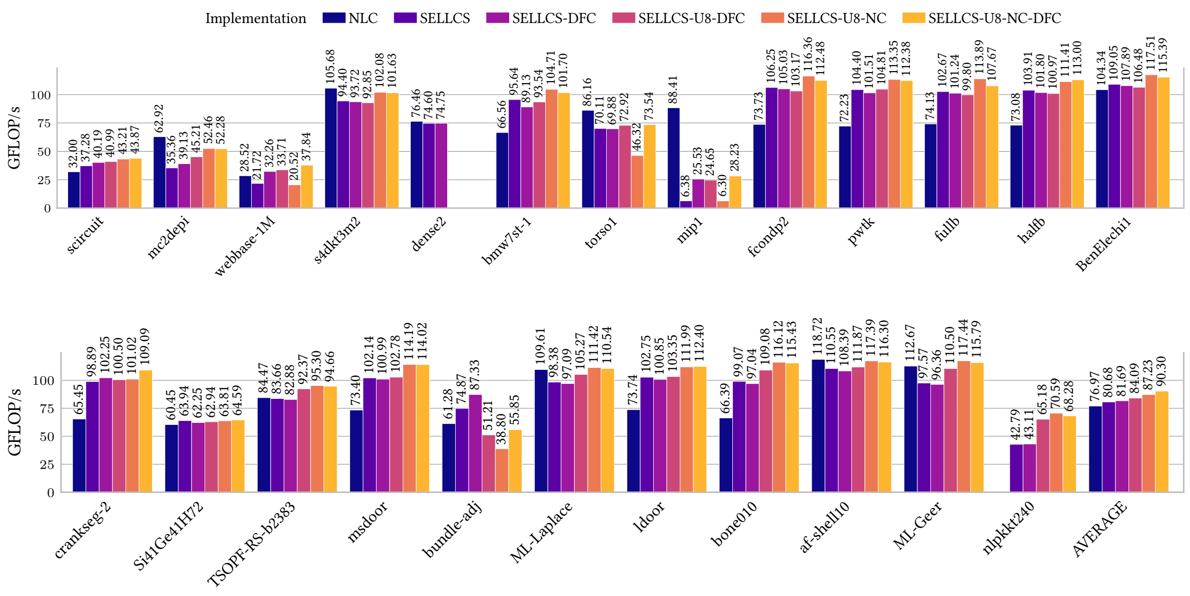


Figure 4.3: Performance comparison of NLC vs our SELL-C-σ implementations for regular matrices.

Cases with significant performance issues

In Figure 4.3 we observe a particular set of matrices where the performance of our SELL-C-σ implementation is clearly suboptimal compared to the NLC math library. Those matrices are: mip1 and torso1. In such cases, we identify issues when scaling the number of cores. Ideally, if we run a parallel workload using 1 or 8 cores, the sum of total instructions executed in each core should be roughly the same for every run. In the case of mip1 and torso1, when running on 8 cores, the total number of scalar instructions is one order of magnitude larger than the single-core run, while the number of vector instructions executed is roughly the same. This large increment of scalar instructions is introduced by the OpenMP runtime due to workload imbalance.

Divergent Flow Control evaluation

To understand the impact of the *DFC* optimization, we compare the performance of *SELLCS* with *SELLCS-DFC*. These two implementations only differ in the use of the *DFC* optimization. Only the second one includes it. On average, the overall performance gains of adapting each vector length instruction to *MAXnnz* are almost negligible. However, it has a large impact in some scenarios. For example, when considering webbase-1M, which represents a website connectivity matrix and has a very low non-zero element density, *SELLCS-DFC* is 50% faster than *SELLCS*. For this matrix, PMC data reveals that the average length of vector instructions when using *SELLCS* is 256, while it drops to 148 for *SELLCS-DFC*. The performance difference between *SELLCS-DFC* and *SELLCS* is explained by the vector length reduction achieved by *SELLCS-DFC*, driven by the ratio of *NNZ* to zero-padding in webbase-1M. We observe very similar behavior for the rest of the matrices that benefit from this optimization (e.g., bundle-adj). While *DFC* does not offer any benefit in regular matrices where an average vector length of 256 does not waste resources, it can be critical for performance in highly irregular matrices. As the minimum required length of vector instructions can be precomputed before running SpMV, *DFC* potential benefits can be easily predicted.

Unrolling by 8 evaluation

To evaluate the benefits of unrolling we compare the performance of *SELLCS-DFC* with *SELLCS-U8-DFC*, since they are equivalent with the only exception that *SELLCS-U8-DFC* implements the unroll optimization. *SELLCS-U8-DFC* unrolls its most inner loop 8 times to process eight slices in the same iteration. *SELLCS-U8-DFC* is in average $\sim$3% faster than *SELLCS-DFC*. For 13 out of 24 matrices, unrolling yields benefits ranging from 1% to 15%, while in the particular case of nlpkkt240 it brings a 51% performance increase.

Unrolling introduces a constraint during the partitioning of the slices into tasks. The way we implement it, requires that the first slice of the 8 unrolled slices is the *longest* of them. After that, every next slice must be equal or shorter than the previous. To avoid breaking this constraint, we do not allow slices from two different sorting windows to concur in the same *unrolled iteration*; the number of slices per task is always a multiple of 8. However, this forces our partitioner to create coarser than optimal tasks, which might end up causing load imbalance between cores. The drop in performance we observe in bundle-adj on the unrolled implementations is consequence of that: the first slices of the matrix contain a very uneven amount of non-zeros compared to the rest of it; to satisfy our partitioning constraint, those first slices are included in a single suboptimal large task.

Figure 4.3 shows that, in general, matrices with fewer rows barely benefit from the unrolling optimization while the largest ones can obtain noticeable benefits. PMC data indicates that these performance benefits come from a reduction in the time spent computing vector operations. For the ML-Geer and ML-Laplace matrices, unrolling optimization yields 3% and 3.5% improvements in LLC hit ratio, respectively. It also brings 14% and 7% improvement in vector load throughput for ML-Geer and ML-Laplace, respectively. These benefits in load throughput are not only due to improved locality of the accesses on x, but also in the vector ILP, since unrolling exposes more instructions to the hardware.

Cache allocation and store relaxation policies

We evaluate the performance impact of using techniques that control cache eviction policies when data is loaded, and the relaxation of instruction dependencies when data are stored. We compare technique *SELLCS-U8-NC-DFC* with *SELLCS-U8-DFC*. Figure 4.3 reports that two thirds of the matrices obtain improvements ranging between 5% to 12% when using *SELLCS-U8-NC-DFC* compared to *SELLCS-U8-DFC*. We examine in detail two cases that represent the behavior observed in matrices that benefit from this optimization. When Cache allocation and store relaxation policies are enabled, ldoor and pwtk obtain an increase in LLC hit ratio of 8% and 6%, and an increase of the vector loaded elements per cycle of 10% and 8%, respectively. We also observe a reduction of $\sim$50% of the L1 cache misses in both cases. These results show how preventing the eviction of the x vector is beneficial for performance. We do not observe any major drawback in performance for any of the matrices we tested, so cache allocation and store relaxation policies can be applied in any scenario without having to add complex logic to enable or disable it. We do not observe any correctness issue when using store relaxation policies.

Applying all optimizations

We obtain in average 90.3 GFLOPs across all matrices by enabling all optimizations, which constitutes a significant improvement of ∼12% and ∼17% compared to the baseline SELL-C-σ and NEC math library implementations, respectively. The significant performance increase that we obtain over the NEC proprietary software, which is specially tailored to SX-Aurora VE, demonstrates the relevance of our optimizations in long vector architectures.

Applicability of column blocking

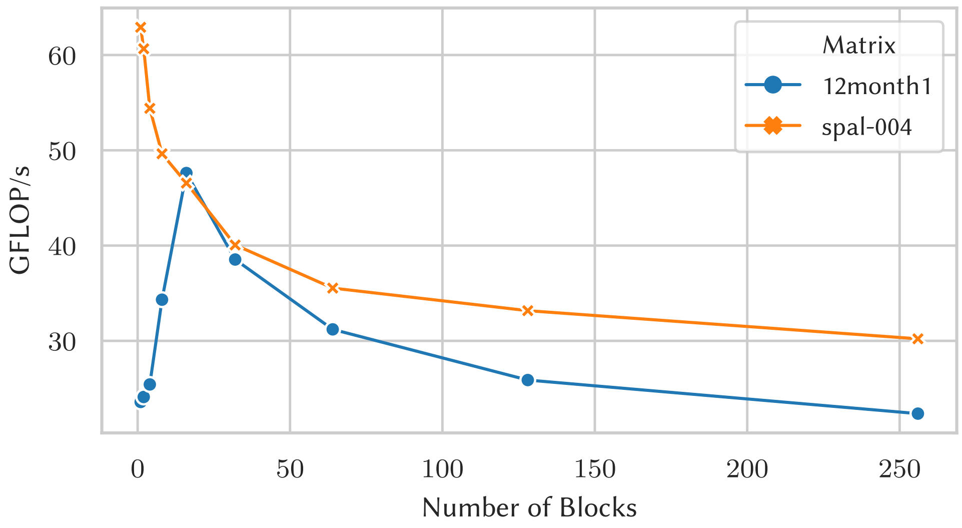


Figure 4.4: Column Blocking: *12month1* obtains up to 2× performance improvement.

We revisit the matrix column blocking optimization (see Figure 2.2) to evaluate the impact in long vector architectures. We evaluate the matrices used previously in the previous experiments (see Figure 4.3). For such matrices, we do not observe performance benefits increasing the number of blocks in which the columns are divided. Previous studies [54] with SIMD architectures, show that this optimization is only effective in *skewed* matrices with very long rows, where x vector locality is more critical. We test a second group composed of two additional matrices with such *skewed* shape: *spal_004* and *12month1*. If Figure 4.4, *12month1* achieves up to 2× performance if the matrix is divided in 16 blocks instead of one. In the case of *spal_004*, we obtain a steady degradation in performance as the number of blocks increases. Our results suggest that, while the applicability of this optimization in long vector architectures seem to be limited, it can have a huge performance impact in a few particular cases.

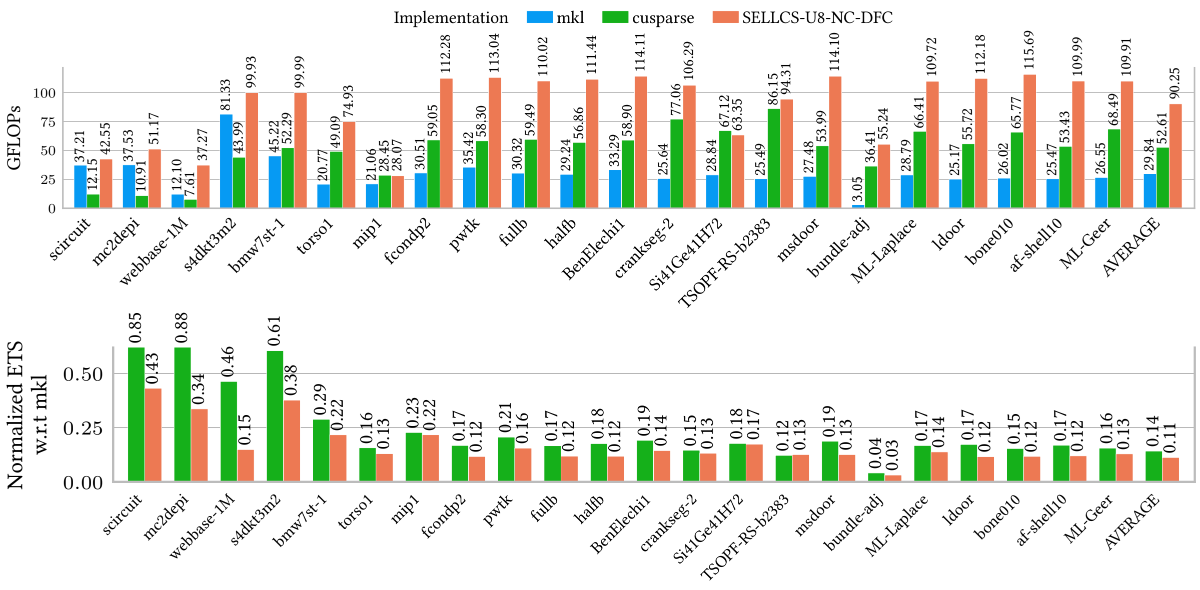


Figure 4.5: Performance and energy-to-solution comparison between different computing platforms.

4.5.2 Comparison with state-of-the-art HPC architectures

We compare the performance and the energy efficiency of our *SELLCS-U8-NC* implementation in the VE against the Intel and NVIDIA platforms we describe in Section 4.4 using their respective *MKL* and *cuSPARSE* math libraries. Both math libraries provide two methods to partition and schedule workload among cores. To ensure fairness, we obtain performance results for both methods and keep the best result per matrix. We report energy to solution metrics measuring the whole execution of the SpMV including matrix loading and preprocessing. The Intel platform reports *total energy* consumed by a run based on information provided by the *RAPL* monitoring counters. The nvidia-smi tool, provides information about the *power* drained by the GPU every second. In VE, we collect power information using a command-line program that reports instantaneous *current and voltage*. Power samples are then integrated over time to obtain the total energy consumption of each run. In all power measurement experiments, we run 600K iterations of the SpMV algorithm so the time spent loading the matrix from disk and pre-processing is less than 10% of the full duration. *MKL* experiments run with OMP_NUM_THREADS=48 to utilize all the cores in both sockets. *cuSPARSE* experiments use a single NVIDIA Volta V100 device.

The upper plot in Figure 4.5 shows our performance comparison of the three platforms. The lower bars represent the normalized energy-to-solution with respect to *MKL* results. On average, *SELLCS-U8-NC* achieves a 1.72*x* and 3.02*x* improvement over the *cuSPARSE* and *MKL*, respectively. In terms of energy efficiency, *SELLCS-U8-NC* consumes 22% less energy compared to *cuSPARSE*, and 9.09× less energy compared to *MKL*. Also, a careful reader can detect that the performance figures of the VE presented in the upper part of Figure 4.5 are slightly lower than the ones presented in Figure 4.3. The mismatch is on average below 5% and it is due to the instrumentation overhead introduced by the power monitoring on the VE. We excluded two matrices from this comparison: dense2, due to incompatibilities with the

Table 4.4: Percentage of the DP peak performance reached by SpMV on state-of-the-art HPC architectures.

	x86 Skylake		NVIDIA v100		NEC SX-Aurora	
	GFlops	% of peak	GFlops	% of peak	GFlops	% of peak
Peak DP	3200		7800		2150	
Best SpMV	81.33	2.54%	86.15	1.11%	115.69	5.38%
Worst SpMV	3.05	0.09%	7.61	0.09%	28.07	1.30%
Average SpMV	29.84	0.93%	52.61	0.67%	90.25	4.19%

unroll optimization; and nlpkkt240, due to memory management errors in the instrumentation libraries. In Table 4.4 we report the percentage of the double-precision peak performance achieved by each of the three architectures.

Chapter 5

Optimizing HPCG using the new RISC-V Vector Extension

The previous chapter shows how matrix data structures and routine implementations can be optimized to perform standalone SpMV as fast as possible in long vector architectures.

In this chapter, we take on the optimization of a more complex benchmark application such as HPCG. We introduce the background related to this topic in Section 2.2.3. The rest of the chapter is organized as follows: Section 5.1 we introduce the HPCG benchmark, its optimizations for vector architectures and the technological background; Section 5.2 contains an overview of other relevant related work; in Section 5.3 we explain in detail the added contributions of this chapter; in Section 5.4 we describe our experimental setup, the hardware platform and the simulator used for our evaluation; in Section 5.5 we present and analyze the results of our experiments.

5.1 Introduction

While the LINPACK [17] benchmark still drives the Top500 ranking, the High Performance Conjugate Gradient [34] (HPCG) is widely accepted as an alternative choice complementing LINPACK for evaluating the performance of large HPC systems. HPCG is recognized by the community as a complementary benchmark to LINPACK because it stresses different arithmetic intensity spots that appear to be representative of other relevant complex HPC workloads. For this reason, in this chapter we focus on the optimization of HPCG on two emerging vector architectures leveraging large vector register size: the VE by NEC and a RISC-V accelerator implementing the 'V' vector extension.

The outcomes of this work are:

- We provide the most optimized open implementation of HPCG for the NEC VE and evaluate its performance.
- We port the vectorized version of HPCG to RISC-V taking advantage of the vector extension.
- Using an architectural simulator of the RISC-VV accelerator, we quantify the benefit of a vectorized and optimized version of HPCG for different vector lengths.
- We perform our study on two designs, one with a vector processing unit (VPU from now on) of 512-bit and another with 2048-bit.

5.2 Related work

HPCG is one of the most used benchmarks for platform comparison. Many studies have focused on optimizing the benchmark to run efficiently on heterogeneous HPC systems with accelerators. Phillips et al. [68] presents an efficient HPCG implementation on CUDA suitable for GP-GPU-based systems. The paper showcases the use of custom SpMV and SymGS kernels, replacing the generic cuSPARSE library calls, to fully exploit the advantages of the *ELL* format. Ao et al. [67] and Liu et al. [55] demonstrate how to achieve high performance in HPCG by using the architectural features available in the Sunway TaihuLight and Tianhe-2 supercomputers respectively. These papers explore techniques such as host off-loading, fine-grain overlapping of computation and data accesses and the use of low-latency intra-node communication between CPUs to send and receive x vector data from boundary elements. Our work extended previous efforts targeting long vector architectures, we cover most of their tuning proposals in the background section [44, 47, 73]. We provide an open highly efficient implementation for both RISC-VV and VE architectures and discuss in detail the impact of the low-locality and unrolling optimizations in two accelerator designs.

5.3 Kernel vector optimizations

In this section, we discuss our contributions to improve the performance of the main HPCG matrix-vector kernels, SpMV and SymGS, in both the VE and RISC-V architectures.

We produce an open portable vector intrinsics HPCG implementation based on the *Advanced Vectorization* code we describe in the previous section. We will refer to our new implementation as *Open Vectorization*. Moreover, in the previous chapter, we show how applying unrolling and cache prioritization to SELL-C-σ SpMV algorithm in long vector accelerators can yield a positive performance impact close to 10% on average across a wide set of different sparse matrices [26]. We adapt those SELL-C-σ optimizations to support *ELL* in HPCG, in both SpMV and SymGS kernel routines, allowing us to be more efficient in our matrix-vector operations.

5.3.1 Low temporal locality data management

In Section 4.3.2 we describe the optimization opportunities in SELL-C-σ SpMV to manage data with low locality, using cache allocation policies, and how we leverage them. In HPCG, both SpMV and SymGS kernels have a similar memory access pattern to the matrix A, vector x and y vector, therefore, in the of the VE we apply equivalent cache allocation policies as in SELL-C-σ SpMV. The implementation however, differs in RISC-VV. The current RISC-VV specification [72] does not include low locality management support yet. We test cache bypassing in our RISC-VV architectural performance predictions by implementing it in the simulator. Different from the cache allocation policies in the VE, cache bypassing avoids storing any data flagged as with low locality in the cache.

Table 5.1: Memory access locality

Loop	Memory Access	Temporal locality
Outer	Load Y values (SymGS only)	Low
Inner	Load Values	Low
Inner	Load Col. Indices	Low
Inner	Gather X coef.	High
Outer	Store result to Y	Low

Cache bypassing and allocation implementation Similar to our work with SELL-C-σ, in Table 5.1, we categorize the memory accesses in each of the load and store instructions in the SpMV and SymGS kernels on HPCG. We introduce the low priority hints in the code and the simulator configuration accordingly. In the case of our RISC-VV code, we do not include any locality hint directly, instead, we provide the simulator with an explicit list of the load and store memory instructions whose accessed data should not be cached.

5.3.2 Unrolling

Unrolling the outer loop of vector *ELL*-based SpMV implementations can reduce control flow overhead and improve the usage of the register file. In HPCG, it is possible to unroll the outer loop of the SymGS and SpMV kernels. On top of the better register file usage, loading larger chunks of elements *column-wise*, increase the locality of the random accesses to vector *x*.

In Figure 5.1 we show how the matrix access pattern changes with the unrolling optimization. The red arrow and dotted lines indicate the element access order in two cases: without unrolling (left) and unrolling by two the outer loop (right). In this example, we assume an *AVL* of four elements. Without unrolling, the *column-wise* loads to the first four rows are issued before loading any element from the next four rows. Unrolling by 2 changes the order to access the elements of the eight rows *column-wise*, before loading elements from the next column.

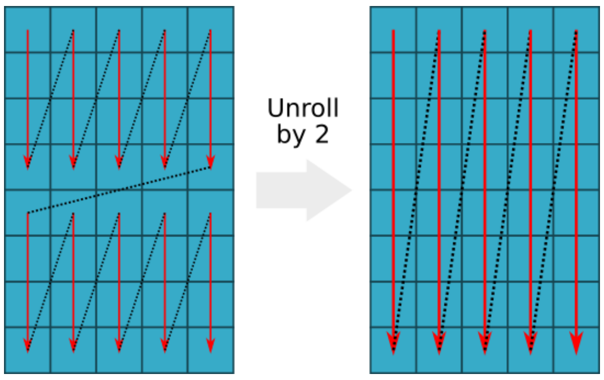


Figure 5.1: *A* matrix acccess pattern when unrolling is applied.

Unrolling implementation In HPCG we employ a different approach to implement the unrolling optimization compared to SELL-C-σ in Section 4.3.2. We implement unrolling in the VE code by using the convenient *#pragma clang loop unroll* directives available in LLVM. SpMV and SymGS use a low number of registers at the same time; up to three in the inner loop. In the case of the VE with 64 ISA registers, it is possible to unroll up to 20 times without spilling registers. We create vector register arrays with the same size as the unroll so each unrolled iteration works with an independent set of vector variables. In compile time, this translates to every unrolled iteration having its own exclusive set of vector registers. For this particular study, after testing several levels of unroll, unrolling by 4 yields the best performance results, for the rest of the paper we will use implicitly unrolling by 4 in our VE results. Epilogues are often needed when implementing unroll optimizations. We eliminate the necessity of using epilogues by dynamically adjusting the granted vector length for each instruction to avoid doing incorrect out-of-bounds accesses to the data structures. In our tests, this additional management comes at the cost of a negligible overhead.

This optimization can be also applied to RISC-VV. However, due to limitations in our compilation toolchain it requires a different implementation approach. In particular, the LLVM-VE compiler allows declaring arrays of *vector type* variables the same way we can declare arrays of *char type* to hold a text string; our RISC-VV compiler does not. We use a workaround using C code constructs and pre-processing macros. We replace every *#pragma clang loop unroll* with a custom macro.

5.3.3 WAXPBY and DDOT

These kernels are simple vector-vector operations. We replace the WAXPBY and DDOT routines using the NEC custom compiler directives (#pragmas) found in the *Advanced Vectorization* implementation [37] we describe in Section 2.2.3. We develop new intrinsics-based versions for both VE and RISC-VV architectures, covering the especial cases found in HPCG like for example, the calls to DDOT where the two input vectors are the same, etc. It is important to remark at the moment of this publication, the toolchain supporting the RISC-VV is limited and under development. Therefore, some of our implementation decisions are made according to those current limitations, e.g: the use of macros instead of LLVM pragmas to implement unrolling. Nonetheless, the underlying concepts applied should provide equal benefits if they were reimplemented more conveniently in the future.

For easy access, we made the code for the *Advanced Vectorization* and *Open vectorization* implementations, containing all our optimizations for the VE and RISC-VV, available in GitHub.[1]

5.4 Methodology

In this section we introduce the hardware and software infrastructure used in our VE experiments and RISC-V performance predictions. We obtain results corresponding to the RISC-VV-based accelerators from Section 5.5 using the **RISC-V simulation toolchain**.

5.4.1 SX-Aurora Vector Engine

The NEC VE accelerator we use for the experiments in this chapter is the VE 20B. It a very similar architecture to the VE 10B we describe in Section 4.4 with just improved memory bandwidth and CPU clock frequencies. We upgrade our hardware from the VE 10B to provide numbers based on the latest hardware available. The VE 20B has a memory bandwidth of 1.53 TB/s and the cores run at a 1.6 GHz frequency. As in our work with SpMV, we use the open-source LLVM-VE project [57] which supports intrinsics allowing full control over the generated code [56].

[1]https://github.com/efocht/hpcg-ve-open

5.4.2 RISC-VV simulation toolchain

To produce our results, we rely on a trace-based simulation toolchain provided by the EPI project [20]. We use the instruction emulator *Vehave* and the cycle-accurate architectural simulator *MUSA*

Tracing with Vehave Vehave is a user-space emulator for the V-extension of the RISC-V ISA that runs on RISC-V Linux. It allows a functional verification of a program that uses V-extension instructions or a code generator, such as a compiler, that emits V-extension instructions. Vehave has been used on the HiFive Unleashed board [35]. This is a RISC-V hardware platform capable to run Linux. Vehave emulates instructions by intercepting the illegal instruction exception that a CPU emits when it encounters an unknown/invalid instruction. Once an illegal instruction is found, Vehave decodes it and if it is a valid V-extension instruction it emulates it, else an error is propagated back. The program resumes once the emulation of the vector instruction is complete. Vehave relies on the LLVM libraries of the compiler which already supports the V-extension for the process of decoding the instructions. The output of Vehave is collected in a .trace file which stores in plain text extensive details about each vector instruction emulated and some quantitative figure of the scalar code executed before each vector instruction.

The tracing and simulation process is roughly $100\times$ slower than the native execution on the VE. Is not possible to perform a proper sweep in order to find a big enough and well-performing input size. After a small set of tests, we set the HPCG input parameters for our simulations at *nx=ny=nz=128*. This generates a matrix of 648 MB in size, with an x vector of 16 MB. We trace a single CG iteration, which with this input size, it takes a bit more than 24h to generate the trace. As every iteration roughly performs the same operations, one iteration is highly representative of the rest of the CG execution and valid for our measurements and analysis.

MUSA simulator While Vehave has full visibility on vector instructions and their scope in the code, it does not allow to collect performance information. The emulated time of vector instruction, in fact, can not be correlated with their realistic timing. For this reason, the MUSA simulator incorporates a timing model capability in the Vehave emulation infrastructure that uses the initial trace to drive a microarchitecture simulation. From this simulation, it enriches the trace by adding precise information in terms of instruction latency, memory hierarchy level where accesses are served, or commit cycle time. The MUSA timing model keeps all information present in the first trace unaltered and adds additional timing and performance data. On top of the data stored in the trace, MUSA considers different architectural abstractions to simulate fundamental aspects like memory hierarchy, instruction dependencies, or out-of-order pipeline. Using textual configuration files we can instruct MUSA to simulate several

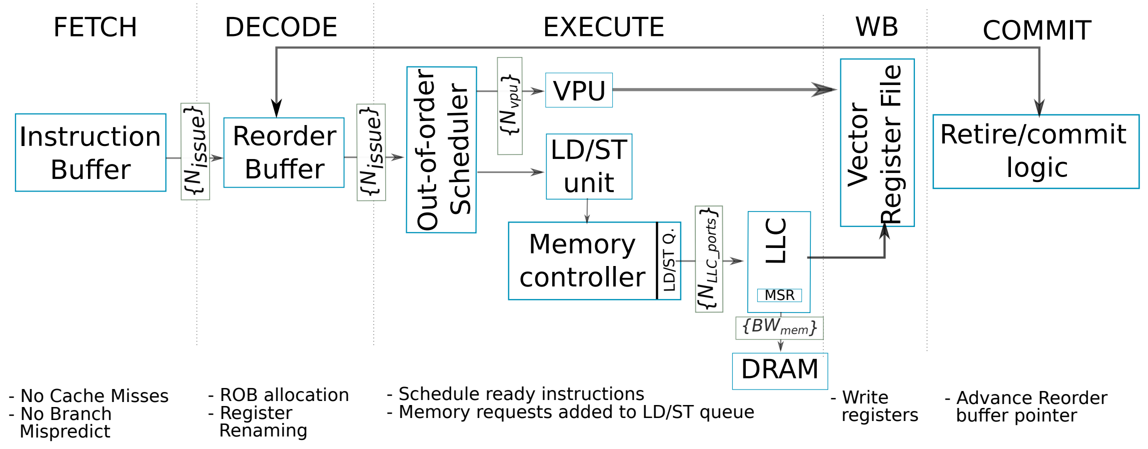


Figure 5.2: *MUSA* simulator architecture overview.

architecture configurations such as instruction latency for all the vector instructions, number and size of cache levels, memory access latency and size, number of reorder buffer entries and other similar information about the underlying architecture. The output of the MUSA simulator can be analyzed using Paraver as well [69], thoroughly visualizing all the architecture behaviour during the execution of a vector program coupled with timing information. This way we can estimate i) the performance of the vector implementation of our algorithms; ii) the effect of architectural changes on our code.

The block diagram in Figure 5.2, shows an overview of the simulated pipeline stages and modules of MUSA. Every instruction is simulated in five stages: Fetch, Decode, Execute, Write-back and Commit. The labels in the arrows between stages indicate the parameter determining the maximum throughput in terms of instructions, requests or bytes. For example, N_{issue} is the maximum number of instructions sent every cycle from the instruction buffer to the reorder buffer, and also, from the reorder buffer to the scheduler. If the arrow is not labeled, the throughput is infinite. N_{vpu} expresses the number of arithmetic vector instruction pipelines. Other simulation parameters not included in Figure 5.2 but listed in Table 5.2 are: $ROB_{entries}$ is the number of instructions the Reorder buffer can hold, C_{size} is the size of the LLC in bytes, MS_{regs} is the number of simultaneous in-flight requests to main memory, VRF_{regs} is the number of physical vector registers available. The annotations at the lower part of the figure describe the relevant behaviour of the components on that stage.

Simulated accelerators Using MUSA, we simulate two RISC-VV floating-point accelerators designs targeting different performance levels: i) a *low-end* design, with modest specifications and 512-bit wide VPU similar to SIMD units used in mainstream supercomputer CPUs [58, 88]; and ii) another *high-end* design, with a more aggressive 2048-bit wide VPU and 7× more

Table 5.2: Simulation parameters

Parameter	Label	Low-end	High-end
VPU throughput [64-bit elem/cycle]		8	32
Vector Registers	VRF_{regs}	40	64
Fetch/Decode/Commit [inst/cycle]	N_{issue}	2	8
ROB entries	$ROB_{entries}$	16	48
CPU to LLC ports	N_{LLC}	1	4
Cache Size [MB]	C_{size}	1	2
Memory bw. [Gbit/s]	Mem_{BW}	21	150
Miss Status Registers (MSR)	MS_{regs}	64	512

memory bandwidth than the first design. We list the rest of the accelerators key specifications in Table 5.2. The table contains the values we assign to each structure and interconnection in Figure 5.2. In both designs, we set the latency for each type of vector instruction based on other state-of-the-art hardware accelerators. We configure our accelerator memory hierarchy such that the vector load and store instructions access directly to the LLC. To simulate low locality data policies, we use the MUSA support for cache bypassing. We specify the list of PC's with vector load and store instructions that bypass the LLC and access directly to the DRAM.

5.5 Evaluation

Following, we evaluate the performance impact of our kernel optimizations targeting long vector architectures in HPCG described in Section 5.3. For that, we perform tests in real hardware using a VE and architectural simulations using the MUSA RISC-VV simulator, both introduced in Section 5.4.

To provide a comparison baseline, the first three blue bars in Figure 5.3 represent the performance results corresponding to VE HPCG implementations prior to this work. On the other hand, the bars in red represent the performance we obtain by progressively adding kernel optimizations. The bars corresponding to previous work, left to right, represent: the default HPCG 3.1 code without any kind of optimization (*Vanilla*); the *Basic Vectorization* (BV), executed in a previous VE 10B model [45]; and the *Advanced Vectorization* (AV) implementation (see Section 2.2.3) that achieves up to 120.8GFlops. We replace the heavily tuned proprietary kernel library with our *Open Vectorization* (OV) kernel library coded using vector intrinsics. The red bars, left to right, represent the performance obtained when progressively adding kernel optimizations to such implementation. In the end, we obtain 122.71 GFlops when both unroll and low-locality management optimizations are enabled. In the next sections, we break down the individual performance impact of each of these optimizations.

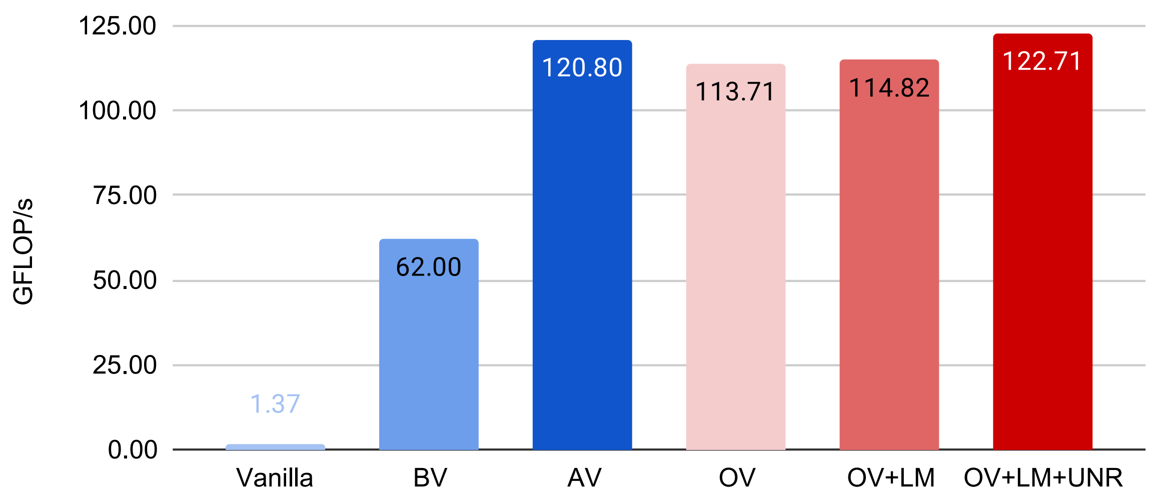


Figure 5.3: VE performance improvements across different levels of optimization.

5.5.1 Cache prioritization and bypassing

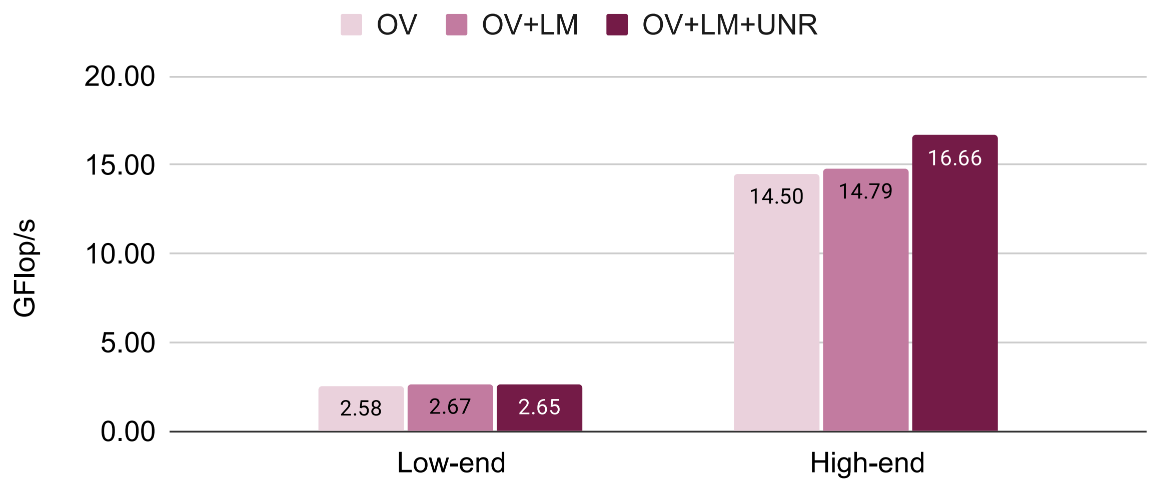


Figure 5.4: Performance impact of each kernel optimization in the simulated RISC-VV accelerators.

We discuss low locality data accesses optimization in Section 5.3.1. The *OV* and the *OV+LM* results in Figure 5.3 run the same code with the sole difference of using VE *non-cacheable* directives for locality management as we specify in Table 5.1. Comparing those results, we observe a 1.0% performance increase if this optimization is enabled. By measuring the performance counters available in the VE, we observe that such increase in performance can be linked to a reduction of 0.6% in the LLC misses and 50% less L1 cache misses during the execution of SymGS and SpMV.

In our RISC-VV accelerator performance predictions (see Figure 5.4), by comparing the results simulating with bypassing enabled (*OV+LM*) and disabled (*OV*), we obtain a 3.5% and 2.0% performance increase in the *low-end* and *high-end* designs respectively.

5.5.2 Unrolling

In Figure 5.3, we also evaluate the benefits of the unrolling optimization on the VE by comparing the *OV+LM* bar, without unrolling, and the *OV+LM+UNR* bar, which runs the same code as *OV+LM* with the sole addition of unrolling. If enabled, unrolling four times the outer loop of both SpMV and SymGS yields a 6.9% performance increase.

Back to Figure 5.4, we observe that the *low-end* accelerator experiences a negligible slowdown of 0.7% when unrolling is used. Note that, our simulation statistics show 97.6% memory bandwidth usage. Reaching that point, we do not expect further performance improvements on the *low-end* design, as it is not possible to bring data from memory faster. On the *high-end* accelerator, however, unrolling by four yields 12.6% performance increase, reaching a memory bandwidth usage of 86.2%.

5.5.3 Accelerator designs

Compared to the *low-end* design, the *high-end* counterpart achieves 6.24× more performance. The best performance results of the *low-end* and *high-end* designs have a 20.5 GB/s and a 129.3 GB/s average memory bandwidth usage respectively, which constitutes a 6.19× difference. This similarity in the performance and memory bandwidth metrics suggest there is a strong correlation between them. To take full advantage of the main memory bandwidth available, it is important to balance the architecture hardware structures and other datapath links in the hierarchy, e.g: CPU to LLC bandwidth. Analyzing the *high-end* simulation statistics, we find that even its memory hierarchy is well balanced, there are a significant amount of cycles where the front-end of the CPU is not issuing enough memory instructions to saturate the memory bandwidth. One approach to tackle this issue is to increase the reorder buffer size ($ROB_{entries}$ parameter in MUSA), so that more memory instructions can be executed concurrently. In our experiments, doubling the $ROB_{entries}$ from 48 to 96 lead to 16.2% performance improvements; further increasing up to 200 $ROB_{entries}$ did not yield additional benefits.

5.5.4 Vector Length efficiency

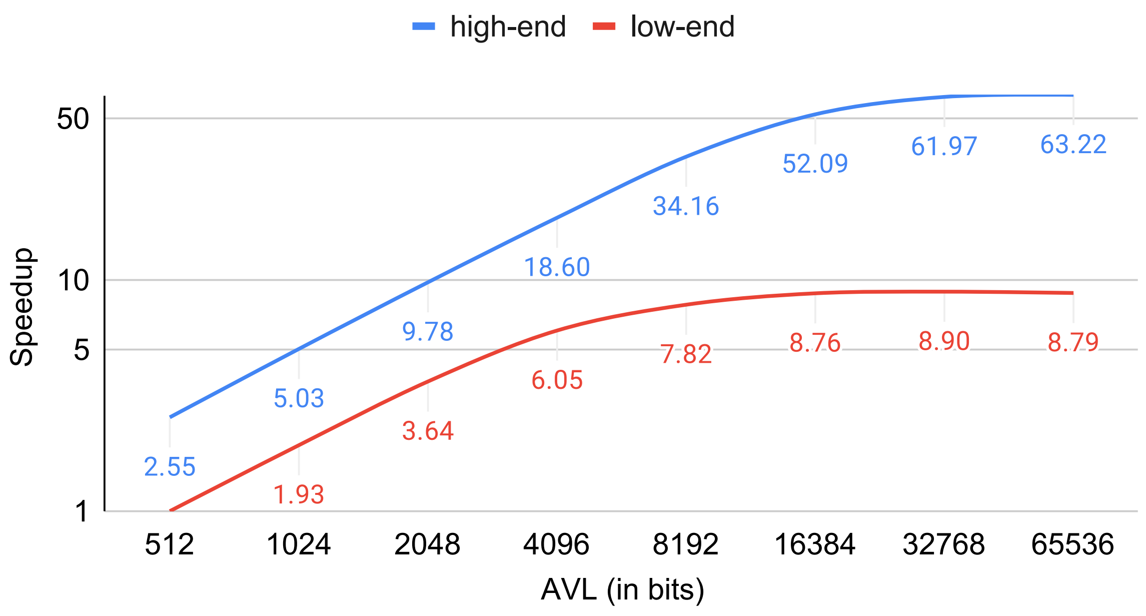


Figure 5.5: Performance impact increasing the *AVL* in our two accelerator designs.

In section 2.1.2, we comment on the potential advantages of having an (*AVL*) detached from the VPU length. Long vector architectures benefit the most from compute-intensive workloads with a high degree of data-level parallelism. But bear in mind, that not all applications might benefit from having an aggressive *AVL*. In some cases, finding enough parallelism may be challenging or rather impossible. On top of that, having a long VPU or *AVL* has a significant impact on the area and power consumption of the core. Thus, it is important to carefully size such hardware parameters according to the target application demands.

In figure 5.5, we present our experiments varying the *AVL* on HPCG on two accelerators. We simulate *AVL* values ranging from 512-bit to 65536-bit in power of two steps. Compared to 512-bit *AVL*, the *low-end* accelerator achieves a 6.05× speedup when scaling up to 4096-bit. After that, the performance benefits obtained until reaching 16384-bit *AVL* are moderate. Finally, the memory bandwidth of the device becomes the bottleneck of the application and the performance curve flattens. In the case of the *high-end* accelerator, we obtain performance benefits beyond the 16384-bit *AVL* used the experiments Section 5.5.1 and 5.5.2. In this case, we find that the performance stops scaling at 32768-bit *AVL*.

Technological trade-offs Previous studies use custom implementations of McPat [4] to model the chip area cost when increasing the VPU length. The trends observed in those studies suggest that doubling the VPU length roughly doubles its area too. A similar trend applies to the vector register file which roughly doubles its size every time the *AVL* doubles. This linear area increment, paired with the also close-to-linear benefits (until the memory bandwidth becomes the bottleneck) observed in Figure 5.5, indicate that the optimal VPU length and *AVL* design points for efficiency will vary depending on the target performance and application. In the case of low arithmetic applications like HPCG, this target performance will be also constrained by the memory bandwidth available on the device.

Chapter 6

Conclusions and Future Work

This last chapter summarizes the main conclusions of the thesis, highlighting the publications derived from it and commenting on possible future work research areas.

6.1 Conclusions

This dissertation has analyzed the trade-offs adjusting the architectural features in a compute node or accelerator to fit the needs of state-of-the-art HPC benchmark applications. We extend the MUSA end-to-end simulation methodology to perform a wide design space exploration campaign, analyzing parallel behavior and providing performance and power estimations for each application and architecture simulated. We derive several conclusions from this campaign that could be used to drive the design of next-generation HPC compute systems.

First, our scaling analysis shows significant parallel inefficiencies in most of the benchmarks tested. Even without taking into account message passing communication overheads and using an optimistic upper bound (hardware agnostic simulations), most applications are not capable of scaling over 75% parallel efficiency on 64 core CPU configurations. Second, our hardware explorations show how compute node configurations with 512-bit wide FP units yield 20% to 75% performance speed-up and an average power increase of 60% with respect to 128-bit wide configurations. Our study also shows that increasing the size of the SIMD units has a general benefit, e.g., may provide energy reductions for parallel codes that properly exploit both thread-level and SIMD level parallelism

With the continuous increase of static power consumption expected in next-generation HPC systems, underutilization of computing resources is the main way to hurt overall energy efficiency. Given the scaling issues and vectorization challenges observed, we insist in the co-design point of view, bringing to the spotlight the necessity of a large effort on the task-level and data-level parallelization of applications, to maximize the utilization of the available hardware resources.

As a result of the findings in the design space exploration, this thesis endeavors in the optimization of challenging applications such as the SpMV numerical kernel and the HPCG benchmark in long vector architectures. Specifically, we focused on a commercial accelerator by NEC leveraging a vector length of 16-kbits and the emulation of a RISC-V vector architecture that allows us to explore different vector lengths.

Our version of SpMV for the SX-Aurora long vector architecture shows roughly 17% performance over the highly optimized proprietary vendor implementation. When compared to other high-end architectures, our implementation performance on the VE is very competitive, on average winning over both standard libraries from the competing architectures Xeon Skylake Platinum and NVIDIA Volta: MKL and cuSparse. In terms of energy-to-solution, accelerators allow up to 9x speed-up compared to general-purpose CPUs, even when using SIMD extensions and highly optimized libraries. In addition, performance measurements indicate that long vector architectures match the requirements of memory bandwidth-demanding data-parallel workloads like SpMV.

The last set of conclusions is gathered from our work with HPCG. Leveraging the learnings from the previous contribution, we study the performance boost given by the unrolling and low-locality management kernel optimizations adapted to the ELLPACK-based SpMV and SymGS kernels on HPCG. In our real hardware evaluation, enabling both kernel optimizations in the VE results in 7.9% additional GFlops compared to disabling them. The total 122.71 GFlops attained represent a 1.6% performance increase over the previous proprietary proposal. Using simulations, we estimate the performance of our open HPCG implementation in two accelerator designs leveraging the RISC-V vector extension, called *low-end* and *high-end*. On one hand, applying both optimizations running in the *low-end* accelerator yields a 2.7% speed-up. In such case, the simulation statistics reveal that our vectorized baseline already utilizes more than 95% of such DRAM bandwidth available, thus, making it difficult to obtain further significant gains. On the other hand, the *high-end* accelerator has a much higher memory bandwidth and the performance improves up to 14.9% when both optimizations are enabled.

Finally, the vector length scaling experiments support the idea that the increased flexibility introduced by having an architectural vector length detached from the VPU length is a powerful tool in the design space of compute accelerators and improves code portability.

6.2 Publications

This section lists the publications linked to the research presented in this thesis.

In the first place, we perform a large design space explorations to analyze the trade-offs of hardware components in next-generation systems. To achieve that, we extend the MUSA methodology, improving the simulation of hybrid parallel workloads, adding support for new memory technologies and accurate energy consumption estimations using third-party power models. This work, presented in Chapter 3, has been published at the International Parallel and Distributed Processing Symposium (IPDPS) [28]:

- "Design Space Exploration of Next-Generation HPC Machines"
  Constantino Gómez, Francesc Martínez, Adrià Armejach, Miquel Moretó, Filippo Mantovani, Marc Casas
  International Parallel and Distributed Processing Symposium, May 2019 (IPDPS ’19)
  DOI: 10.1109/IPDPS.2019.00017

We developed a very efficient SELL-C-σ SpMV for vector architectures as our second contribution. The detailed study in Chapter 4, showing the optimization process was accepted and published at the Symposium on Principles and Practice of Parallel Programming (PPoPP) [27]:

- "Efficiently running SpMV on long vector architectures"
  Constantino Gómez, Filippo Mantovani, Erich Focht, Marc Casas
  Symposium on Principles and Practice of Parallel Programming, February 2021 (PPoPP ’21)
  DOI: 10.1145/3437801.3441592

We made all the SpMV codes available under Apache 2.0 license in a repository:

- https://repo.hca.bsc.es/gitlab/cgomez/spmv-long-vector

Finally, the last contribution takes on a more challenging application, the HPCG. We propose vectorization optimizations for the main sparse kernels that compose the benchmark, achieving high throughput in both VE and RISC-V long vector architectures. Our work with HPCG developed in Chapter 5 has been submitted for publication.

- "Optimizing HPCG using the new RISC-V Vector Extension"
  Constantino Gómez, Filippo Mantovani, Erich Focht, Marc Casas

6.3 Future Work

During the development of this thesis, the interest in accelerators to build large scientific computing systems kept growing at a higher rate than ever. In addition, the use of custom hardware is becoming commonplace in mass consumer markets like the mobile phone industry to accelerate demanding workloads (e.g., speech recognition, video decoding, etc.). This explosion in the design space of hardware and applications opens many research opportunities opportunities. Overall, we can only expect in the future further co-design studies tackling other different software and hardware challenges in the field of high-performance accelerators and applications.

We think there is a main research path that our work could be directly extended to. We addressed two optimization challenges in the field of linear algebra: the SpMV algorithm and the HPCG benchmark application. However, regular grids, like the one HPCG solves, only represent a subset of the large spectrum of problems in computational physics.

Following this topic, a next step would be tackling the issue of optimizing mathematical methods solving irregular grids problems. When mapped to matrices and operations, it is possible to exploit such regularity to optimize and simplify otherwise complex routines. We discussed these opportunities in the context of HPCG in Section 2.2.3, and how they allow us to make efficient use of the vector units. Irregular grids might introduce stronger data dependences reducing the data-level parallelism, thus limiting the performance in long vector accelerators. It would be necessary then, a further study to asses the suitability of vector architectures for this type of problems.

In addition, to converge faster to a solution, linear iterative solvers perform factorization operations on the matrix on every step. Current state-of-the-art factorization implementations often make use of branching instructions that are difficult to vectorize (e.g., Incomplete *LU* factorization). Researching new techniques to handle such flow divergence efficiently on vector architectures, could bring large performance benefits in linear algebra solvers, as factorization represents a significant part of the total execution time.

Bibliography

[1] *8GB (x72, ECC, SR) 288-Pin DDR4 RDIMM - Micron Technology*. URL: www.micron.com/~/media/documents/products/data-sheet/modules/rdimm/ddr4/asf18c1gx72pz.pdf (visited on 03/27/2018).

[2] EDWARD ANDERSON and YOUCEF SAAD. "SOLVING SPARSE TRIANGULAR LINEAR SYSTEMS ON PARALLEL COMPUTERS". In: *International Journal of High Speed Computing* 01.01 (1989), pp. 73–95. DOI: 10.1142/S0129053389000056. eprint: https://doi.org/10.1142/S0129053389000056. URL: https://doi.org/10.1142/S0129053389000056.

[3] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. "Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs". In: *University of Tennessee, Tech. Rep. ut-eecs-14-727* (2014).

[4] Eishi Arima, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, and Mitsuhisa Sato. "Power/Performance/Area Evaluations for Next-Generation HPC Processors using the A64FX Chip". In: *2021 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS)*. 2021, pp. 1–6. DOI: 10.1109/COOLCHIPS52128.2021.9410320.

[5] Rosa M Badia, Jesús Labarta, Judit Gimenez, and Francesc Escale. "DIMEMAS: Predicting MPI applications behavior in Grid environments". In: *Workshop on Grid Applications and Programming Tools (GGF8)*. Vol. 86. 2003, pp. 52–62.

[6] Scott Beamer, Krste Asanović, and David Patterson. "Reducing Pagerank Communication via Propagation Blocking". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2017, pp. 820–831. DOI: 10.1109/IPDPS.2017.112.

[7] Nathan Bell and Michael Garland. *Efficient sparse matrix-vector multiplication on CUDA*. Tech. rep. Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[8] OpenMP Architecture Review Board. *OpenMP 5.0 Specification*. Tech. rep. November 2018. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[9] Daniele Buono, Fabrizio Petrini, Fabio Checconi, Xing Liu, Xinyu Que, Chris Long, and Tai-Ching Tuan. "Optimizing Sparse Matrix-Vector Multiplication for Large-Scale Data Analytics". In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS '16. Istanbul, Turkey: Association for Computing Machinery, June 2016, pp. 1–12. ISBN: 978-1-4503-4361-9. DOI: 10.1145/2925426.2926278. (Visited on 03/10/2020).

[10] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. "DRAMPower: Open-source DRAM power & energy estimation tool". In: *URL: http://www.drampower.info* 22 (2012).

[11] Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, Chunqiang Li, Yu Pu, Jianyi Meng, Xiaolang Yan, Yuan Xie, and Xiaoning Qi. "Xuantie-910: A Commercial Multi-Core 12-Stage Pipeline Out-of-Order 64-bit High Performance RISC-V Processor with Vector Extension : Industrial Product". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 2020, pp. 52–64. DOI: 10.1109/ISCA45697.2020.00016.

[12] Shizhao Chen, Jianbin Fang, Donglin Chen, Chuanfu Xu, and Zheng Wang. "Adaptive optimization of sparse matrix-vector multiplication on emerging many-core architectures". In: *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE. 2018, pp. 649–658.

[13] Xinhai Chen, Peizhen Xie, Lihua Chi, Jie Liu, and Chunye Gong. "An efficient SIMD compression format for sparse matrix-vector multiplication". In: *Concurrency and Computation: Practice and Experience* 30.23 (2018). e4800 CPE-18-0532.R1, e4800. DOI: 10.1002/cpe.4800. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4800. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4800.

[14] NEC CORPORATION. *SX-Aurora TSUBASA Architecture Guide Revision 1.1*. Tech. rep. 2018.

[15] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE Computational Science and Engineering* 5.1 (Jan. 1998), pp. 46–55. ISSN: 1070-9924.

[16] Wolfgang E. Denzel, Jian Li, Peter Walker, and Yuho Jin. “A Framework for End-to-End Simulation of High-performance Computing Systems”. en. In: *SIMULATION* 86.5-6 (May 2010), pp. 331–350. ISSN: 0037-5497. (Visited on 03/28/2018).

[17] Jack Dongarra and Piotr Luszczek. “LINPACK Benchmark”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1033–1036. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_155. URL: https://doi.org/10.1007/978-0-387-09766-4_155.

[18] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. “OmpSs: A proposal for programming Heterogeneous Multi-Core Architectures”. In: *Parallel Processing Letters* 21.02 (June 2011), pp. 173–193. ISSN: 0129-6264. (Visited on 03/27/2018).

[19] *DynamoRIO Dynamic Instrumentation Tool Platform*. URL: dynamorio.org/ (visited on 03/27/2018).

[20] *EPI RISC-V toolchain*. 2021. URL: %5Curl%7Bhttps://repo.hca.bsc.es/gitlab/epi-public/risc-v-vector-simulation-environment/-/wikis/BSC-RISC%E2%80%90V-Vector-Toolchain%7D.

[21] *Extrae | BSC-Tools*. URL: tools.bsc.es/extrae (visited on 03/27/2018).

[22] *Fast and Accurate DRAM Power and Energy Estimation for DRAM*. URL: uni-kl.de/en/3d-dram/tools/drampower/.

[23] *Fujitsu Reveals Details of Processor That Will Power Post-K Supercomputer*. URL: top500.org/news/fujitsu-reveals-details-of-processor-that-will-power-post-k-supercomputer/ (visited on 10/13/2018).

[24] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. “Open MPI: Goals, concept, and design of a next generation MPI implementation”. In: *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer. 2004, pp. 97–104.

[25] Sergi Girona, Jesus Labarta, and Rosa M. Badia. “Validation of Dimemas Communication Model for MPI Collective Operations”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin Heidelberg. 2000, pp. 39–46.

[26] Constantino Gómez, Filippo Mantovani, Erich Focht, and Marc Casas. “Efficiently Running SpMV on Long Vector Architectures”. In: PPoPP ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 292–303. ISBN: 9781450382946. DOI: 10.1145/3437801.3441592. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/3437801.3441592.

[27] Constantino Gómez, Filippo Mantovani, Erich Focht, and Marc Casas. “Efficiently Running SpMV on Long Vector Architectures”. In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 292–303. DOI: 10.1145/3437801.3441592.

[28] Constantino Gómez, Francesc Martınez, Adrià Armejach, Miquel Moretó, Filippo Mantovani, and Marc Casas. “Design Space Exploration of Next-Generation HPC Machines”. In: *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 54–65. DOI: 10.1109/IPDPS.2019.00017.

[29] Thomas Grass, César Allande, Adrià Armejach, Alejandro Rico, Eduard Ayguadé, Jesús Labarta, Mateo Valero, Marc Casas, and Miquel Moretó. “MUSA: a multi-level simulation approach for next-generation HPC machines”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016*, pp. 526–537.

[30] Thomas Grass, Alejandro Rico, Marc Casas, Miquel Moreto, and Eduard Ayguadé. “TaskPoint: Sampled simulation of task-based programs”. In: *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2016, pp. 296–306. DOI: 10.1109/ISPASS.2016.7482104.

[31] Eric Grobelny, David Bueno, Ian Troxel, Alan D. George, and Jeffrey S. Vetter. “FASE: A Framework for Scalable Performance Prediction of HPC Systems and Applications”. en. In: *SIMULATION* 83.10 (Oct. 2007), pp. 721–745. ISSN: 0037-5497. (Visited on 03/29/2018).

[32] *GW4 Isambard*. en-GB. URL: gw4.ac.uk/isambard/ (visited on 10/13/2018).

[33] FU Haohuan, LIAO Junfeng, YANG Jinzhe, WANG Lanning, HUANG Xiaomeng, YANG Chao, XUE Wei, QIAO Fangli, ZHAO Wei, YIN Xunqiang, HOU Chaofeng, GE Wei, ZHANG Jian, WANG Yangang, and YANG Guangwen. "The Sunway TaihuLight supercomputer: system and applications". In: *SCIENCE CHINA Information Sciences* 59.7, 072001 (2016).

[34] Michael Allen Heroux, Jack Dongarra, and Piotr Luszczek. "HPCG Benchmark Technical Specification". In: (Oct. 2013). DOI: 10.2172/1113870. URL: https://www.osti.gov/biblio/1113870.

[35] *HiFive Unleashed specifications*. URL: %5Curl%7Bhttps://www.sifive.com/boards/hifive-unleashed%7D (visited on 2021).

[36] Mark Horowitz. "1.1 Computing's energy problem (and what we can do about it)". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14. DOI: 10.1109/ISSCC.2014.6757323.

[37] *HPCG for Vector Engine by Erich Focht*. 2021. URL: %5Curl%7Bhttps://github.com/efocht/hpcg-ve-open%7D.

[38] Mingyu Hsieh, Kevin Pedretti, Jie Meng, Ayse Coskun, Michael Levenhagen, and Arun Rodrigues. "SST + Gem5 = a Scalable Simulation Infrastructure for High Performance Computing". In: *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*. SIMUTOOLS '12. Desenzano del Garda, Italy, 2012, pp. 196–201. ISBN: 978-1-4503-1510-4.

[39] Ian Karlin, Jeff Keasler, and JR Neely. *Lulesh 2.0 updates and changes*. Tech. rep. Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2013.

[40] Darren J Kerbyson, Henry J Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J Wasserman, and Mike Gittings. "Predictive performance and scalability modeling of a large-scale application". In: *Supercomputing, ACM/IEEE 2001 Conference*. IEEE. 2001, pp. 39–39.

[41] Joonyoung Kim and Younsu Kim. "HBM: Memory solution for bandwidth-hungry processors". In: *Hot Chips 26 Symposium (HCS), 2014 IEEE*. IEEE. 2014, pp. 1–24.

[42] Y. Kim, W. Yang, and O. Mutlu. "Ramulator: A Fast and Extensible DRAM Simulator". In: *IEEE Computer Architecture Letters* 15.1 (Jan. 2016), pp. 45–49. ISSN: 1556-6056.

[43] D R Kincaid, T C Oppe, and D M Young. "ITPACKV 2D user's guide". In: (May 1989).

[44] K. Komatsu, Ryusuke Egawa, H. Takizawa, and Hiroaki Kobayashi. “An Approach to the Highest Efficiency of the HPCG Benchmark on the SX-ACE Supercomputer”. In: 2015.

[45] Kazuhiko Komatsu and Hiroaki Kobayashi. “Performance Evaluation of SX-Aurora TSUBASA by Using Benchmark Programs”. In: *Sustained Simulation Performance 2018 and 2019*. Ed. by Michael M. Resch, Yevgeniya Kovalenko, Wolfgang Bez, Erich Focht, and Hiroaki Kobayashi. Cham: Springer International Publishing, 2020, pp. 69–77. ISBN: 978-3-030-39181-2.

[46] Kazuhiko Komatsu, Shintaro Momose, Yoko Isobe, Osamu Watanabe, Akihiro Musa, Mitsuo Yokokawa, Toshikazu Aoyama, Masayuki Sato, and Hiroaki Kobayashi. “Performance Evaluation of a Vector Supercomputer SX-Aurora TSUBASA”. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. Nov. 2018, pp. 685–696. DOI: 10.1109/SC.2018.00057.

[47] Kazuhiko Komatsu, Akito Onodera, Erich Focht, Soya Fujimoto, Yoko Isobe, Shintaro Momose, Masayuki Sato, and Hiroaki Kobayashi. “Performance and Power Analysis of a Vector Computing System”. In: *Supercomputing Frontiers and Innovations* 8.2 (Aug. 2021), pp. 75–94. DOI: 10.14529/jsfi210205. URL: https://superfri.org/index.php/superfri/article/view/387.

[48] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop. “A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units”. In: *SIAM Journal on Scientific Computing* 36.5 (Jan. 2014), pp. C401–C423. ISSN: 1064-8275. DOI: 10.1137/130930352. (Visited on 03/10/2020).

[49] Pierre-François Lavallée, Guillaume Colin de Verdiere, Philippe Wautelet, Dimitri Lecas, and Jean-Michel Dupays. “Porting and optimizing HYDRO to new platforms and programming paradigms-lessons learnt”. In: *Technical report, PRACE* (2012).

[50] Christophe Lemuet, Jack Sampson, Jean-Francois Collard, and Norm Jouppi. “The Potential Energy Efficiency of Vector Acceleration”. In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC ’06. Tampa, Florida: Association for Computing Machinery, 2006, 77–es. ISBN: 0769527000. DOI: 10.1145/1188455.1188537. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1145/1188455.1188537.

[51] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. “McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures”. In: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2009, pp. 469–480.

[52] Yishui Li, Peizhen Xie, Xinhai Chen, Jie Liu, Bo Yang, Shengguo Li, Chunye Gong, Xinbiao Gan, and Han Xu. “VBSF: a new storage format for SIMD sparse matrix–vector multiplication on modern processors”. en. In: *The Journal of Supercomputing* (Apr. 2019). ISSN: 1573-0484. (Visited on 03/10/2020).

[53] Weifeng Liu and Brian Vinter. “CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication”. In: *Proceedings of the 29th ACM on International Conference on Supercomputing*. Newport Beach, California, USA: Association for Computing Machinery, June 2015, pp. 339–350. ISBN: 978-1-4503-3559-1. DOI: 10.1145/2751205.2751209. (Visited on 03/10/2020).

[54] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. “Efficient sparse matrix-vector multiplication on x86-based many-core processors”. In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. Eugene, Oregon, USA: Association for Computing Machinery, June 2013, pp. 273–282. ISBN: 978-1-4503-2130-3. DOI: 10.1145/2464996.2465013. (Visited on 03/10/2020).

[55] Yiqun Liu, Chao Yang, Fangfang Liu, Xianyi Zhang, Yutong Lu, Yunfei Du, Canqun Yang, Min Xie, and Xiangke Liao. “623 Tflop/s HPCG run on Tianhe-2: Leveraging millions of hybrid cores”. In: *The International Journal of High Performance Computing Applications* 30.1 (2016), pp. 39–54. DOI: 10.1177/1094342015616266. eprint: https://doi.org/10.1177/1094342015616266. URL: https://doi.org/10.1177/1094342015616266.

[56] *LLVM VE intrinsics*. URL: %5Curl%7Bhttps://sx-aurora-dev.github.io/velintrin.html%7D%20-%20last%20accesses%20April%202020.

[57] *LLVM-VE github repository*. URL: %5Curl%7Bhttps://github.com/sx-aurora-dev/llvm-project%7D%20-%20last%20accesses%20April%202020.

[58] Chris Lomont. *Introduction to Intel Advanced Vector Extensions. Intel White Paper*. 2011.

[59] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhwi Jang, Reiley Jeyapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishnendra Nathella, Hoa Nguyen, Nikos Nikoleris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglini, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Éder F. Zulian. *The gem5 Simulator: Version 20.0+*. 2020. arXiv: 2007.03152 [cs.AR].

[60] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *SIGPLAN Not.* 40.6 (June 2005), pp. 190–200. ISSN: 0362-1340.

[61] *Marenostrum IV - User's guide*. URL: bsc.es/support/MareNostrum4-ug.pdf (visited on 03/27/2018).

[62] Vladimir Marjanović, José Gracia, and Colin W Glass. "Performance modeling of the HPCG benchmark". In: *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer. 2014, pp. 172–192.

[63] Shintaro Momose, Takashi Hagiwara, Yoko Isobe, and Hiroshi Takahara. "The Brand-New Vector Supercomputer, SX-ACE". In: *Supercomputing*. Ed. by Julian Martin Kunkel, Thomas Ludwig, and Hans Werner Meuer. Cham: Springer International Publishing, 2014, pp. 199–214. ISBN: 978-3-319-07518-1.

[64] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. “Automatically tuning sparse matrix-vector multiplication for GPU architectures”. In: *International Conference on High-Performance Embedded Architectures and Compilers*. Springer. 2010, pp. 111–125.

[65] *Mont-Blanc Prototype: Dibona*. en-US. URL: montblanc-project.eu/prototypes (visited on 10/13/2018).

[66] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns. “Enabling Parallel Simulation of Large-Scale HPC Network Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (Jan. 2017), pp. 87–100. ISSN: 1045-9219.

[67] Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj D Kalamkar, Md Mosotofa Ali Patwary, Vadim Pirogov, Pradeep Dubey, Xing Liu, Carlos Rosales, Cyril Mazauric, and Christopher Daley. “Optimizations in a high-performance conjugate gradient benchmark for IA-based multi- and many-core processors”. In: *The International Journal of High Performance Computing Applications* 30.1 (2016), pp. 11–27. DOI: 10.1177/1094342015593157. eprint: https://doi.org/10.1177/1094342015593157. URL: https://doi.org/10.1177/1094342015593157.

[68] Everett Phillips and Massimiliano Fatica. “Performance analysis of the high-performance conjugate gradient benchmark on GPUs”. In: *The International Journal of High Performance Computing Applications* 30.1 (2016), pp. 28–38. DOI: 10.1177/1094342015599239. eprint: https://doi.org/10.1177/1094342015599239. URL: https://doi.org/10.1177/1094342015599239.

[69] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. “Paraver: A tool to visualize and analyze parallel code”. In: *Proceedings of WoTUG-18: transputer and occam developments*. Vol. 44. 1. IOS Press. 1995, pp. 17–31.

[70] Ajay Ramaswamy, Nalini Kumar, Aravind Neelakantan, Herman Lam, and Greg Stitt. “Scalable Behavioral Emulation of Extreme-Scale Systems Using Structural Simulation Toolkit”. In: *Proceedings of the 47th International Conference on Parallel Processing*. ICPP 2018. Eugene, OR, USA: ACM, 17:1–17:11. ISBN: 978-1-4503-6510-9.

[71] Alejandro Rico, Felipe Cabarcas, Carlos Villavieja, Milan Pavlovic, Augusto Vega, Yoav Etsion, Alex Ramirez, and Mateo Valero. “On the Simulation of Large-scale Architectures Using Multiple Application Abstraction Levels”. In: *ACM Trans. Archit. Code Optim.* 8.4 (Jan. 2012), 36:1–36:20. ISSN: 1544-3566.

[72] *RISC-V V-extension 1.0.* URL: %5Curl%7Bhttps://github.com/riscv/riscv-v-spec%7D (visited on 2021).

[73] Daniel Ruiz, Filippo Spiga, Marc Casas, Marta Garcia-Gasulla, and Filippo Mantovani. "Open-Source Shared Memory implementation of the HPCG benchmark: analysis, improvements and evaluation on Cavium ThunderX2". In: *2019 International Conference on High Performance Computing Simulation (HPCS)*. IEEE, 2019, pp. 225–232. DOI: 10.1109/HPCS48598.2019.9188103.

[74] Daniel Sanchez and Christos Kozyrakis. "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems". In: *SIGARCH Comput. Archit. News* 41.3 (June 2013), pp. 475–486. ISSN: 0163-5964. DOI: 10.1145/2508148.2485963. URL: https://doi.org/10.1145/2508148.2485963.

[75] *Sierra | High Performance Computing*. URL: hpc.llnl.gov/hardware/platforms/sierra (visited on 10/13/2018).

[76] A. Sodani. "Knights landing: 2nd Generation Intel® Xeon Phi processor". In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. 2015, pp. 1–24.

[77] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker. "The ARM Scalable Vector Extension". In: *IEEE Micro* 37.2 (2017), pp. 26–39. DOI: 10.1109/MM.2017.35. URL: https://doi.org/10.1109/MM.2017.35.

[78] *SX-Aurora TSUBASA Architecture*. URL: nec.com/en/global/solutions/hpc/sx/architecture.html? (visited on 10/13/2018).

[79] *SX-Aurora TSUBASA Architecture Guide*. 2018. URL: %5Curl%7Bhttps://www.hpc.nec/documents/guide/pdfs/Aurora_ISA_guide.pdf%7D.

[80] R. Teyssier. "Cosmological hydrodynamics with adaptive mesh refinement - A new high resolution code called RAMSES". en. In: *Astronomy & Astrophysics* 385.1 (Apr. 2002), pp. 337–364. ISSN: 0004-6361, 1432-0746. (Visited on 03/23/2018).

[81] *ThunderX2 ARM Processors*. URL: cavium.com/product-thunderx2-arm-processors.html (visited on 10/10/2018).

[82] Top500. Nov. 2018. URL: https://www.top500.org/list/2018/11/.

[83] *Top500 list - November 2021*. URL: %5Curl%7Bhttps://www.top500.org/lists/top500/2021/11/%7D (visited on 2022).

[84] Rob F. Haopiang vanderWijngaart. “NAS Parallel Benchmarks, Multi-Zone Versions”. In: Phoenix, AZ, United States, June 2003. (Visited on 03/23/2018).

[85] J. Wang, J. Beu, R. Bheda, T. Conte, Z. Dong, C. Kersey, M. Rasquinha, G. Riley, W. Song, H. Xiao, P. Xu, and S. Yalamanchili. “Manifold: A parallel simulation framework for multicore systems”. In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2014, pp. 106–115.

[86] S. L. Xi, H. Jacobson, P. Bose, G. Wei, and D. Brooks. “Quantifying sources of error in McPAT and potential impacts on architectural studies”. In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 577–589.

[87] Yohei Yamada and Shintaro Momose. “Vector engine processor of NEC’s brand-new supercomputer SX-Aurora TSUBASA”. In: *Proceedings of A Symposium on High Performance Chips, Hot Chips*. Vol. 30. 2018, pp. 19–21.

[88] Toshio Yoshida. “Fujitsu high performance CPU for the Post-K Computer”. In: *Hot Chips*. Vol. 30. 2018.

[89] Marco Zagha and Guy E Blelloch. “Radix sort for vector multiprocessors”. In: *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. 1991, pp. 712–721.

[90] G. Zheng, G. Gupta, E. Bohm, I. Dooley, and L. V. Kale. “Simulating Large Scale Parallel Applications Using Statistical Models for Sequential Execution Blocks”. In: *2010 IEEE 16th International Conference on Parallel and Distributed Systems*. Dec. 2010, pp. 221–228. DOI: 10.1109/ICPADS.2010.98.