Complete performance and energy models

Deliverable No: D1.4
Deliverable Title: Complete performance and energy models
Deliverable Publish Date: 31 March 2023

Project Title: SparCity: An Optimization and Co-design Framework for Sparse Computation
Call ID: H2020-JTI-EuroHPC-2019-1
Project No: 956213
Project Duration: 36 months
Project Start Date: 1 April 2021
Contact: sparcity-project-group@ku.edu.tr

List of partners:

Participant no.	Participant organisation name	Short name	Country
1 (Coordinator)	Koç University	KU	Turkey
2	Sabancı University	SU	Turkey
3	Simula Research Laboratory AS	Simula	Norway
4	Instituto de Engenharia de Sistemas e Computadores, Investigação e Desenvolvimento em Lisboa	INESC-ID	Portugal
5	Ludwig-Maximilians-Universität München	LMU	Germany
6	Graphcore AS	Graphcore	Norway

CONTENTS

1 INTRODUCTION

The SparCity project is funded by EuroHPC JU (the European High Performance Computing Joint Undertaking) under the 2019 call of Extreme Scale Computing and Data Driven Technologies for research and innovation actions. SparCity aims to create a supercomputing framework that will provide efficient algorithms and coherent tools specifically designed for maximizing the performance and energy efficiency of sparse computations on emerging High Performance Computing (HPC) systems, while also opening up new usage areas for sparse computations in data analytics and deep learning.

Sparse computations are commonly found at the heart of many important applications, but at the same time it is challenging to achieve high performance when performing the sparse computations. SparCity delivers a coherent collection of innovative algorithms and tools for enabling both high efficiency of sparse computations on emerging hardware platforms. More specifically, the objectives of the project are:

- to develop a comprehensive application and data characterization mechanism for sparse computation based on the state-of-the-art analytical and machine-learning-based performance and energy models,
- to develop advanced node-level static and dynamic code optimizations designed for massive and heterogeneous parallel architectures with complex memory hierarchy for sparse computation,
- to devise topology-aware partitioning algorithms and communication optimizations to boost the efficiency of system-level parallelism,
- to create digital SuperTwins of supercomputers to evaluate and simulate what-if hardware scenarios,
- to demonstrate the effectiveness and usability of the SparCity framework by enhancing the computing scale and energy efficiency of challenging real-life applications.
- to deliver a robust, well-supported and documented SparCity framework into the hands of computational scientists, data analysts, and deep learning end-users from industry and academia.

1.1 OBJECTIVES OF THIS DELIVERABLE

The objective of this deliverable is to document sparse-aware performance, power and energy-efficiency models for devices with emerging architectures. Through the identification of the potential for performance and energy-efficiency of different sparse computation workloads, the purpose of these models is to detect compute and memory bottlenecks and suggest optimizations. These optimizations can include modifications to the kernel performing sparse computations that is under study, suggesting specific core and memory frequencies through exploiting Dynamic Voltage and Frequency Scaling (DVFS), and/or in regard to hardware design decisions that can be taken to improve execution given a metric of interest. Another objective is to develop profiling tools for identifying and analysing data movement. Tools capable of performing inter-process/inter-thread communication profiling, reuse distance analysis and cache partitioning/mapping are of utmost importance to achieve efficient mapping of the different types of sparse computations to emerging hardware architectures and devices. Finally, advances in the development of Digital SuperTwin targeting supercomputing platforms, which can seamlessly

integrate the proposed models and tools, enable a high-throughput and precise collection of data and more insightful modeling, which paired with advanced visualization capabilities results in a more straightforward identification of how to take action.

1.2 WORK PERFORMED

In this deliverable, we provide an extensive study focused on the strengths and usability of the proposed extensions to the Cache-Aware Roofline Model (CARM) for efficient sparse computing (sparse CARM), for which a new construction methodology and a novel micro-benchmarking strategy have been proposed as part of Deliverables 1.2 and 4.2 of the SparCity project. In addition, in this deliverable, we include in-depth explanations on the complete model interpretation methodology and usability of the proposed model, and we report the outcomes of its evaluation in the context of the use of matrix reordering techniques applied on synthetic and real-world sparse matrices. We also provide a novel roofline-based evaluation strategy that aims at assessing the variation in performance, power consumption and energy efficiency for a range of sparse kernels' Arithmetic Intensity (AI)s when scaling the operating frequency of the CPU cores. Furthermore, a performance analysis of an SpMV hand-tuned assembly implementation has been performed on a RISC-V microarchitecture. Also relying on CARM, the range of arithmetic intensities of the used algorithm has been thoroughly examined taking into account the cache dynamics of the targeted microprocessor. The compiled knowledge and modeling techniques can be used to identify the demands of sparse computations on current platforms with RISC-V processors and/or to guide the design of novel more capable and domain-specific processors for efficient sparse computations.

Sparse Matrix-Matrix Multiplication (SpMM) methods relying on different sparse formats, and considering different sparsity levels, have been profiled using different combinations of core and memory frequencies on a state-of-the-art GPU across both traditional CUDA cores and AI-oriented tensor cores. The different SpMM alternatives have been cross-compared, using as a baseline a state-of-the-art General Matrix Multiplication (GEMM) implementation, which has also been tuned with DVFS. Several applications rely on contractions between tensors of other orders, such as Tensor Times Matrix (TTM), which represents a contraction between a multi-dimensional tensor of arbitrary order and a second-order tensor. We derived approaches tackling sparse TTM and examined their performance bounds at processing both real and synthetic inputs on CPU and GPU microarchitectures. In addition, this deliverable compiles insights resulting from the exploration for sparse matrix operations of DVFS, a key mechanism of today's computing devices to enable achieving high levels of performance and energy-efficiency.

We extended the profiling tools elaborated in Deliverable 1.2 to target AMD x86 microarchitectures. Low-overhead inter-thread communication (ComDetective) and reuse distance (ReuseTracker) analysis is achieved using the instruction-based sampling (IBS) facility and debug registers present in AMD processors. These tools have relevant features not available in other profiling tools, such as the capability of taking into account true and false sharing and measuring reuse distance in private and shared caches at a low overhead. Those features have been implemented through modification of a Linux kernel module that allows interfacing with the IBS hardware. The tools have been experimentally evaluated on a set of representative benchmarks, achieving high accuracy at a lower overhead than cycle accurate simulators and code instrumentation tools. Furthermore, we analyzed the impact of the cache partitioning methods proposed in Deliverable 1.2 in the context of SpMV execution targeting the A64FX processor, which has an embedded cache partitioning mechanism named sector cache. Partitioning and mapping program objects to specific cache partitions is performed at runtime through the use of

compiler directives. Experimental evaluation demonstrated that using the sector cache allowed in most cases to improve execution in regard to performance and memory bandwidth utilization when processing matrices from cardiac electrophysiology. Relying on dynamic binary instrumentation, we developed a profiling tool that predicts the number of cache misses, with and without the use of the sector cache, from reuse distance histograms. The predictions produced by the profiling tool integrating the proposed cache partitioning methods have been demonstrated to closely match real measurements.

The initial stages of designing a prototype of SuperTwin, a digital replication framework, had been presented in Deliverable 1.2. In this deliverable, we document new developments for this framework, to which several novel and advanced features have been added. The current version of SuperTwin has additional augmentation and semantical query abilities, as-well new benchmarking capabilities, automatic generation of dashboards with support for real-time and per-request monitoring, and CARM modeling with integrated visualization. SuperTwin has been evaluated in regard to the throughput and integrity, as part of a study performed on four different systems under different sampling frequencies and with a varying amount of reported events.

1.3 DEVIATIONS AND COUNTER MEASURES

There was no deviation from the work plan.

1.4 RESOURCES

As also envisioned in the project proposal, the herein elaborated modeling approaches had undergone further improvements and developments over the initial ones reported in the previous deliverables. As such, it is expected that further extensions of the proposed models and tools will be reported in subsequent deliverables, as well as maintained and regularly updated on the respective SparCity Github repositories.

2 CACHE-AWARE ROOFLINE MODELING FOR EFFICIENT SPARSE COMPUTING AND HARDWARE CO-DESIGN

2.1 SPARSE CARM: IMPROVING ROOFLINE INSIGHTFULNESS FOR SPARSE COMPUTATIONS

In order to enable accurate measurement of performance and power consumption upper-bounds of the micro-architecture when performing sparse computations, we proposed a novel micro-benchmarking strategy in Deliverable 4.2 of the SparCity project. In Deliverable 1.2, we also proposed a new methodology for sparse-aware CARM construction, which relies on the proposed micro-benchmarking strategy and is capable of more accurately characterizing sparse computation kernels and their ability to utilize the micro-architecture computation resources. The sparse-aware CARM is also able to provide more precise insights regarding the bottlenecks of sparse computations, as well as, to identify the best optimization steps that should be considered to improve their performance. In this deliverable, we provide an extensive study that demonstrates the insightfulness and usability of the sparse-aware CARM, while we also reveal the complete construction and interpretation methodology behind the proposed model. Furthermore, an in-depth validation and characterization is performed using a set of synthetic and real-world

sparse matrices from standard matrix collections (*e.g.* Suite Sparse[1]) in both single- and multi-threaded execution scenarios. For this evaluation, we also apply several of the most commonly used reordering techniques for sparse matrices: Reverse Cuthill-McKee (RCM), Approximate Minimum Degree (AMD), Nested Dissection (ND), GrayRO based on Zhao et.al.'s work[2] and two algorithms from the Patoh library.[3] By using the proposed model to visualise the changes in cache locality, we aim at verifying the ability of these reordering schemes to better utilize the computational power available in the micro-architecture, as well as their potential to provide performance improvements. A characterization methodology is also proposed in order to tackle load balancing issues that may arise in multi-threaded execution scenarios, which allows for further improvements in the model's insightfulness. Finally, a novel roofline-based evaluation methodology is adopted in order to assess the variation in performance, power consumption and energy efficiency for a range of kernel's AIs when scaling the operating frequency of the CPU cores.

2.1.1 SPARSE-AWARE CARM FOR PERFORMANCE

To validate the proposed sparse-aware CARM, we conducted an extensive experimental campaign on a Linux CentOS 7.5.1804 platform, with an eight-core Intel i7-7820X processor running at the fixed frequency of 3.60GHz with 32KB of L1 Data cache, 1MB of L2 cache, 11MB of L3 cache and 32GB of DRAM running at 2133MHz, with prefetching and hyper-threading deactivated. The proposed sparse-aware CARMs for this execution platform are presented in Figures 1 and 2, which are obtained for Sparse Matrix Vector Multiplication (SpMV) computation where the vector elements fit in the L1 cache and for single- and multi-threaded (8 cores) execution scenarios, respectively. For this evaluation, an open-source SpMV kernel, in x86 assembly, was specifically developed (as documented in Deliverable 4.2), which attains performance close to the corresponding Intel MKL kernel, while facilitating the algorithm analysis.

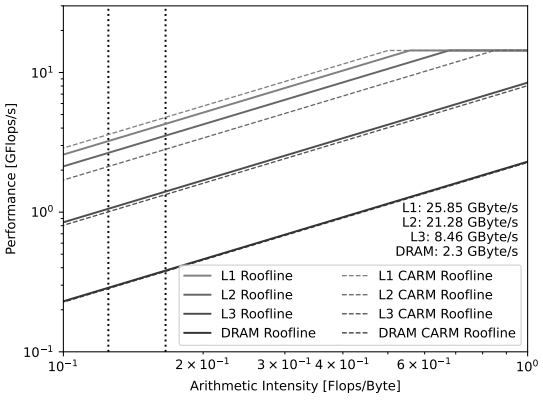


Figure 1 *Single-Threaded SP Sparse CARM*

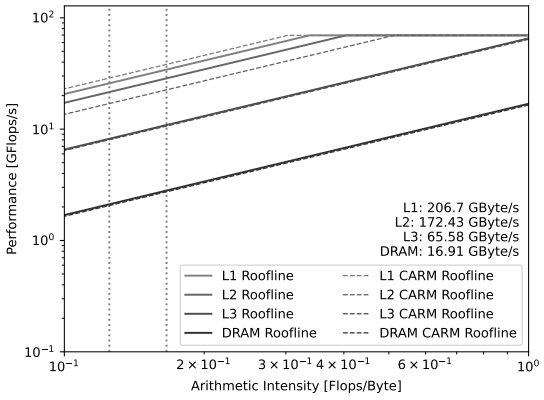


Figure 2 *Multi-Threaded SP Sparse CARM*

As it can be observed in Figures 1 and 2, compared to the dashed original CARM roofs achieved with scalar and single-precision instructions, used in the tested x86 assembly SpMV

[1]Timothy A Davis and Yifan Hu. "The University of Florida sparse matrix collection". *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–25.

[2]Haoran Zhao et al. "Exploring Better Speculation and Data Locality in Sparse Matrix-Vector Multiplication on Intel Xeon". *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE. 2020, pp. 601–609.

[3]Ümit V Çatalyürek and Cevdet Aykanat. "Patoh (partitioning tool for hypergraphs)". *Encyclopedia of parallel computing*. Springer, 2011, pp. 1479–1487.

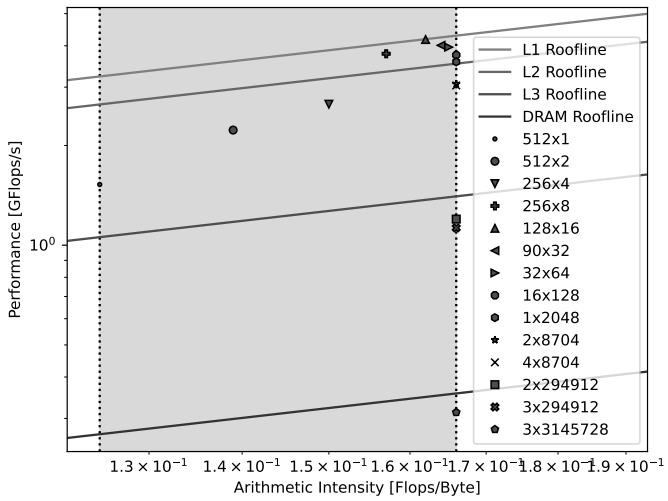


Figure 3 *AI Variation according to NNZ per row in Single Threaded Execution.*

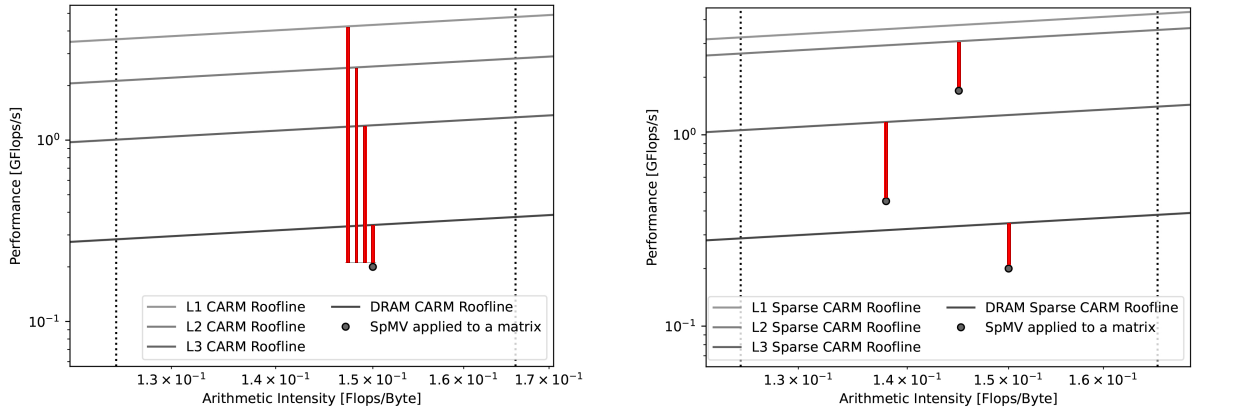
kernel, the proposed sparse-aware CARM achieves a lower L1 performance. This reduction in the maximum attainable performance for the L1 cache is mainly due to indirect accesses to the vector elements and memory accesses to multiple arrays. The dependencies between all those transfers lead to performance degradation when data is being retrieved from the L1 cache. When considering the proposed L2 rooflines, the performance is higher than the CARM roofs given that the locality of the vector elements is preserved at the L1 cache (while in the original roofs, streaming tests would have locality only in L2 cache). L3 roofline is slightly higher than the CARM, however the diminished difference in performance entails that the higher latency in accessing the vector with column indexes, with locality in L3, lowers the possible performance benefits of having the vector elements continually stored in the L1 cache, due to the dependencies related to indirectly accessing the former. New DRAM rooflines also show little differences in performance, further justifying this performance impact.

The vertically dotted lines shown in Figures 1 and 2 represent the theoretical AI range of the x86 Assembly SpMV kernel (see Deliverable 4.2). This range can be experimentally verified by relying on dense synthetic matrices with different dimensions. Figure 3 presents this experimental evaluation, where highlighted in grey is the AI range attainable using the x86 assembly SpMV kernel in single precision ($\beta_i = 4$). As it can be observed, the minimum AI corresponds to the dense matrices with 1 column (AI $= 0.125$). As the number of non-zero elements per row increases, the AI shifts to the right, approaching and stabilizing at values close to the theoretical maximum, as it can be observed for matrices with more than 8704 columns (AI ≈ 0.16666).

2.1.2 Analysis and usability of sparse-aware CARM

The CARM derives its insight on attainable performance from the representation of the applications relative to the rooflines of the model, thus facilitating the optimization process. However, the sparse-aware CARM and corresponding rooflines are created based on different principles, given that sparse computation and corresponding performance not only depends on the utilized kernel but also on the sparse matrix to compute upon, thus providing different insights from the original model. As the attained rooflines are not only architecture-based but also related to the used kernel, *i.e.* using bandwidth values retrieved from micro-benchmarking (as documented

in Deliverable 4.2), the adapted model is now a representation of how the computation upon a specific sparse matrix is able to utilize the performance capabilities of the micro-architecture, bound by the characteristics and inefficiencies of the sparse algorithm utilized. There is also another difference concerning the attainable performance and its association with the warm-cache nature of the performed testing. The rooflines were built from micro-benchmarking bandwidth values obtained using dense matrices stored in Compressed Sparse Row (CSR) for warm-cache SpMV computation, where all involved data structures are stored in the respective memory level, while the vector elements fit in the L1 cache. Given that all accesses to matrix data are performed in a sequential and cohesive manner and that total data involved in the computation is always the same, the optimization path is restricted to matrix reordering, where row and column permutation vectors are applied to the matrix in order to optimize the accesses to the vector. This means that, for a warm-cache scenario, a sparse matrix, whose SpMV involved data structures only fit inside a specific cache level, cannot exceed the performance of the corresponding memory roofline, as optimization through reordering only affects the accesses to the vector elements.



(a) *CARM representation of a SpMV kernel applied to a sparse matrix and perceived attainable performance.*

(b) *Sparse CARM representation of different matrices computed upon and perceived attainable performance.*

Figure 4 *Comparison between the two CARM representations in a sparse computation scenario.*

Figure 4 graphically represents the attainable performance portrayed by the original CARM approach and the sparse-aware CARM approach. An example of a SpMV kernel computing upon a matrix is presented as grey dots in both models, and the red lines represent the attainable performance through optimization, where the original model portrays an misleading representation of the application. This is due not only to the rooflines not being built considering limitations of the sparse computation, but also the represented attainable performance by applying recommended optimization strategies for memory bound kernels is unlikely to maximize the applications' use of the computational power of the micro-architecture. Comparatively, profilling using the Sparse CARM portrays a realistic scenario regarding maximum performance in each memory level, with the kernel limitations properly considered, and attainable performance is limited by the memory level where the total size of the involved data structures are located.

2.1.3 Sparse CARM analysis: reordering algorithms and load balancing

Given that the results presented in Deliverable 1.2 have shown the potential to achieve performance gains with Intel MKL SpMV by reordering the input sparse matrices, we extend herein

this performance evaluation by considering the x86 assembly SpMV code. For this purpose, we focus on using real sparse matrices, representing them in the Sparse CARM, and applying state-of-the-art reordering algorithms, in single and multi-threaded execution. The considered reordering algorithms are RCM,[4] AMD,[5] ND,[6] the reordering algorithms included in the *Patoh* partition library:[7] cutnet and connectivity applied to two matrices, and a reordering algorithm created based on the work proposed by Zhao et. al,[8] named GrayRO. A set of eleven matrices from Suite Sparse[9] are considered for evaluation, as presented in Table 1. This set of matrices are real, general and non-complex, and have a diverse number of rows, columns and non-zero elements, covering a wide range of execution scenarios.

Matrix Name	*Rows*	*Cols*	*NNZ*	*Size (KBytes)*
Freescale1	3428755	3428755	17052626	≈ 173400
patents	3774768	3774768	14970767	≈ 161190
torso1	116158	116158	8516500	≈ 67900
Stanford	281903	281903	2312497	≈ 21370
ns3Da	20414	20414	1679599	≈ 13360
poisson3Db	85623	85623	2374949	≈ 19560
sme3Db	29067	29067	2081063	≈ 16600
mixtank_new	29957	29957	1990919	≈ 15900
ss	1652780	1652780	34753577	≈ 290880
Fullchip	2987012	2987012	26621983	≈ 242990
wb-edu	9845725	9845725	57156537	≈ 561920

Table 1 *Real matrices retrieved from SuiteSparse.*

Figure 5a presents the characterization of the matrices with different reordering algorithms in the Sparse CARM for single-threaded execution. As it can be observed, the considered reordering methods do not guarantee performance improvements for all matrices. In fact, several reordered matrices suffer from performance reduction (*e.g.* Fullchip RCM and Freescale RCM), while other have small gains in performance. For example, Freescale1 attained a speedup of 1.05x with ND, while RCM provided 1.21x speedup for Stanford.

When taking into consideration the insight provided by the proposed modelling approach, the maximum performance (in warm-cache conditions) that the SpMV kernel can achieve when computing upon a specific sparse matrix is the roofline located directly above its representation. This means that matrices Fullchip and SS which are represented on top of the DRAM roofline (as are some of their reordered counterparts), have already highly optimized accesses to the vector elements, meaning that any further optimization is likely to yield small performance gains, and it might even result in the performance degradation, as can seen by the Fullchip RCM case. However, matrices represented below this point are able to benefit from the improved locality

[4]Wai-Hung Liu and Andrew H. Sherman. "Comparative Analysis of the Cuthill–McKee and the Reverse Cuthill–McKee Ordering Algorithms for Sparse Matrices". *SIAM Journal on Numerical Analysis* 13.2 (1976), pp. 198–213. DOI: 10.1137/0713020.

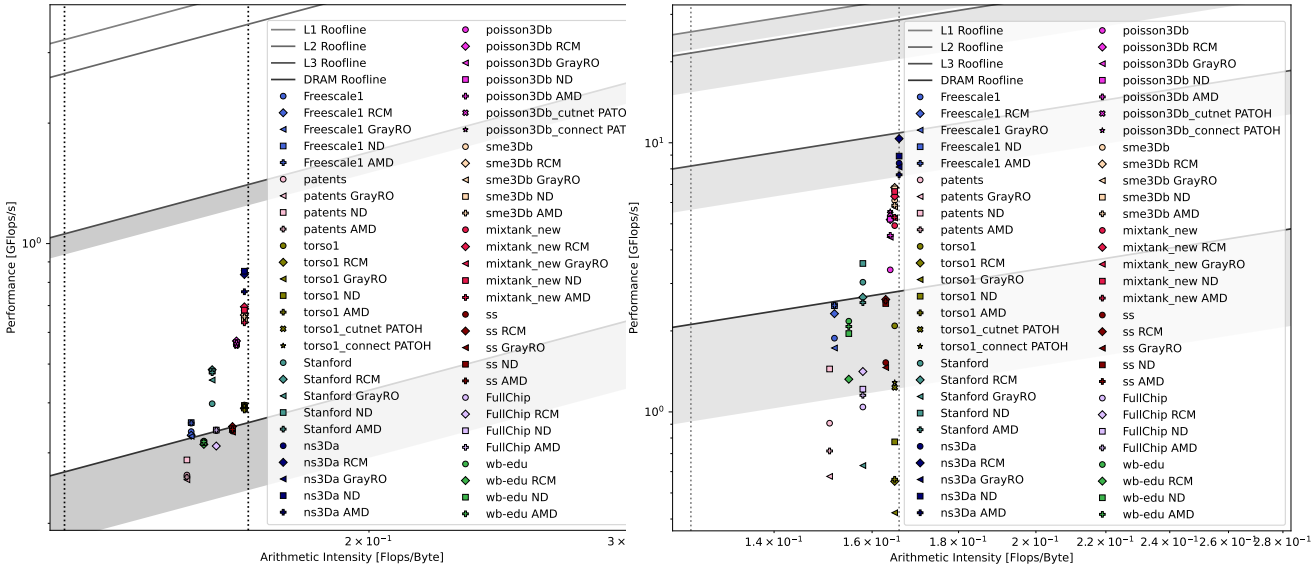
[5]Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. "An Approximate Minimum Degree Ordering Algorithm". *SIAM Journal on Matrix Analysis and Applications* 17.4 (1996), pp. 886–905. DOI: 10.1137/S0895479894278952.

[6]Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. "Generalized Nested Dissection". *SIAM Journal on Numerical Analysis* 16.2 (1979), pp. 346–358. DOI: 10.1137/0716027.

[7]Çatalyürek and Aykanat, "Patoh (partitioning tool for hypergraphs)".

[8]Zhao et al., "Exploring Better Speculation and Data Locality in Sparse Matrix-Vector Multiplication on Intel Xeon".

[9]Davis and Hu, "The University of Florida sparse matrix collection".



(a) *Single-threaded execution.* **(b)** *Multi-threaded execution.*

Figure 5 *Real and Reordered Matrices represented in Sparse-CARM.*

of the memory accesses to the vector elements, meaning that their performance can further be improved towards their respective roofline.

The same matrices and reordering algorithms were also tested in multi-threaded fashion (8 cores execution), with each thread being assigned a partition based on equal distribution of the rows with shared access to the X and Y vector. The representation of these matrices in the multi-threaded SpMV adapted CARM (see Figure 5b) show a similar trend to the single-thread experiments, since the reordering algorithms can either provide speedups or slowdowns depending on the matrix. However, some of these results are also highly dependent on load balancing of the multi-threaded execution. Since the experiments tested in this work focused on an even partition of the rows between threads, the number of non-zero elements computed by each thread may differ, resulting in data imbalance, thus provoking the performance degradation, as threads with less workload will idle at the end of their execution.

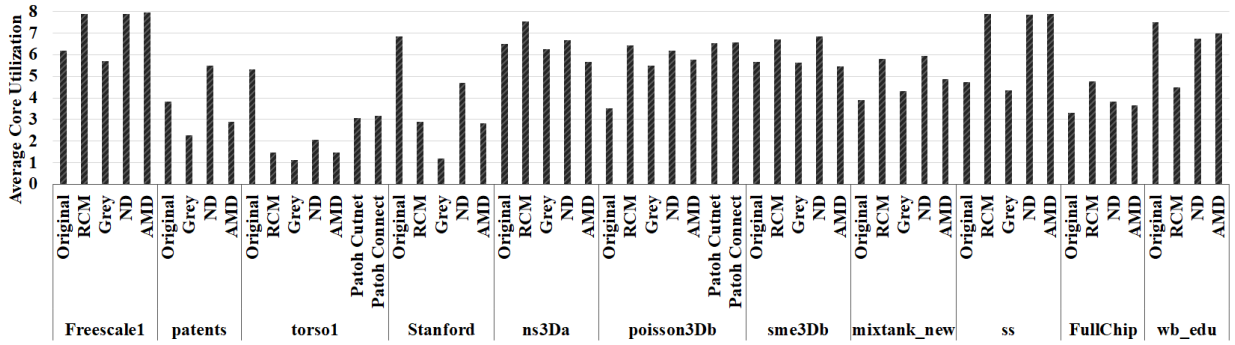
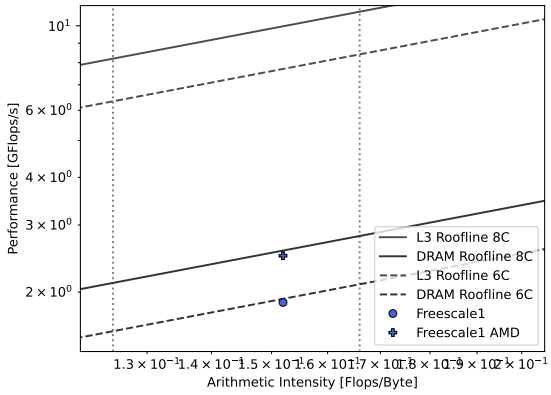


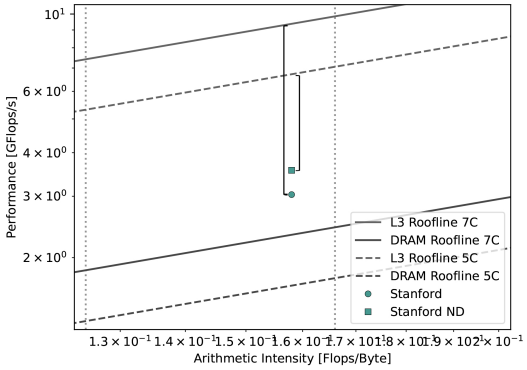
Figure 6 *Average Core utilization for each used matrix in Multi-threaded execution.*

By observing the average core utilization of each matrix and reordered versions, presented in Figure 6, it is possible to observe that reordering algorithms may result in an increase of the load balancing (all 8 cores are being utilized), while in others may provoke serious imbalance

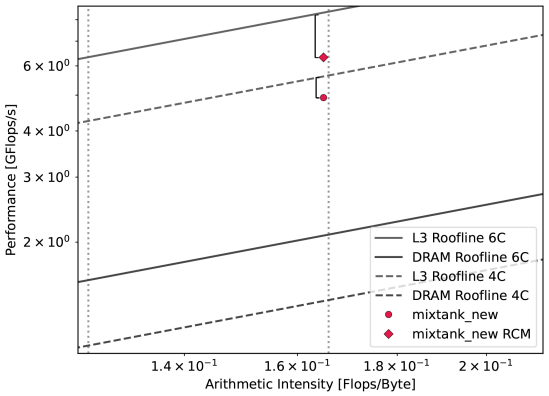
issues. For example, matrix Poisson3Db has an increase of the average core utilization from 3.5 to 6.4 when using RCM as a reordering algorithm. On the other hand, Torso1 has a reduction from 5.32 to 1.46 in the core utilization when using the same RCM algorithm. This explains their characterization in the Sparse CARM, since Poisson3Db has a speedup of 1.54x and Torso1 a slowdown of 0.26x. However, the performance impact of load balancing affects the insight provided by the Sparse CARM, as the variation of the representation of the matrices is no longer uniquely associated to an improvement in accessing the vector elements. To tackle this issue, the sparse-aware CARM analysis is extended in order to consider different core utilizations when targeting multi-thread execution.



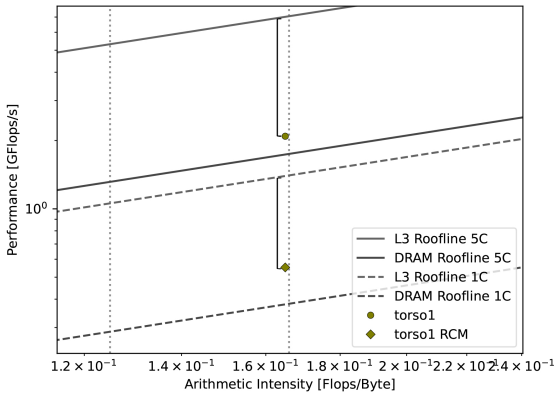
(a) *Freescale1 and Freescale1 AMD represented with 6C and 8C rooflines respetively.*



(b) *Stanford and Stanford ND with 7C and 5C rooflines respetively.*



(c) *Mixtank_new and Mixtank_new RCM represented with 4C and 6C rooflines respetively.*



(d) *Torso1 and Torso1 RCM represented with 5C and 1C rooflines respetively.*

Figure 7 *Matrices and reordered versions represented with Sparse CARM rooflines according to their average core utilization.*

As can be seen in Figure 7, when the previously tested matrices are profiled in models that consider their average core utilization, it is possible to characterize the performance variations due to changes in locality of accesses to the vector elements, without the impact of load balancing. For example, the computation upon Freescale1, with an average core utilization of 6.2, and the AMD reordered version, with an improved load balancing reaching close to an average core

utilization of 8, when analysed using the VTune Top-down method, showcased no changes, indicating similar locality when accessing the vector elements, which was not fully corroborated by the 8 core sparse-aware CARM. However, when characterizing these matrices in sparse-aware models that consider their core utilization (Figure 7a), it is possible to verify that both Freescale1 and Freescale1 AMD are placed in the same relative position regarding the DRAM roof. This shows that there was no change in the main execution bottlenecks after reordering, which fully corroborates the findings of the Top-Down analysis.

Similar behaviour occurs for Stanford and Stanford ND, where VTune analysis indicates an improvement in the locality of the accesses to the vector elements, and comparing the representation of each kernel in Figure 7b relative to their rooflines, the reordered version is represented closer to the L3 roofline, which corroborates the previous statement. For a scenario where the reordering lowers the locality of the accesses to the vector elements, the analysis of the Mixtank_new and Mixtank_new with RCM shows that the latter is placed at a lower relative position to their corresponding roofline compared to the original one, as can be seen in Figure 7c. The torso1 locality improvements showcased by the VTune analysis are also clearly seen in Figure 7d by the improved relative representation of the reordered matrix to its respective roofline.

Figures 8, 9, 10, 11, 12, 13, 14 and 15 showcase all tested matrices and reordered counter parts represented on respective Sparse CARM models that correspond to their average core utilization. As can be observed, profiling the matrices in this manner, improves on the insight obtained from their representation related to the new rooflines. For example, observing the poisson3Db matrices represented in Figure 9, we can derive that the RCM reordering improved the locality of accesses to the vector elements when compared to the original matrix and the GrayRO reordered version, knowing that their representation is now less correlated to load balancing issues. The same analysis can be made between two reordered matrices, corresponding to different load balance scenarios, such as Stanford in Figure 9 and Stanford ND in Figure 11, in which the latter is positioned closer to its respective L3 roofline, showcasing a better scenario in terms of locality of accesses to the vector elements.

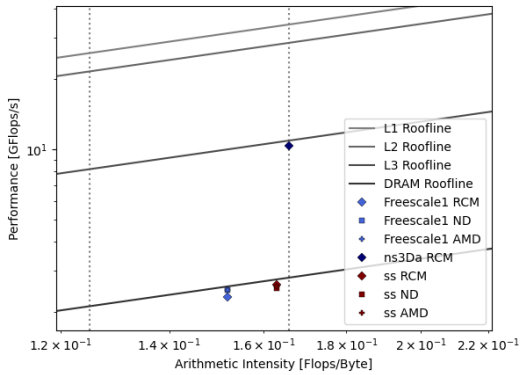


Figure 8 *Matrices with average core utilization ≈ 8 profiled in 8 Thread SpMV CARM.*

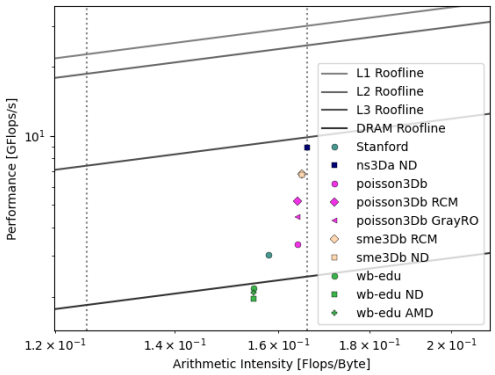


Figure 9 *Matrices with average core utilization ≈ 7 profiled in 7 Thread SpMV CARM.*

Based on these results, in order to profile the use of the available computational power when executing the SpMV computation upon matrices in multi-threaded fashion while also excluding the load balancing issue, each case should be profiled according to their respective average core utilization, so as to isolate the cache locality as the major performance impediment and assess whether further optimization on the matrix structure through reordering is necessary or even

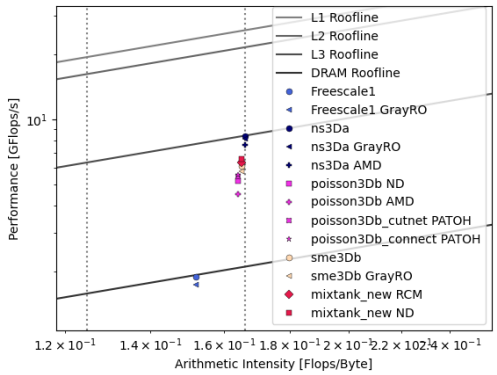


Figure 10 *Matrices with average core utilization ≈ 6 profiled in 6 Thread SpMV CARM.*

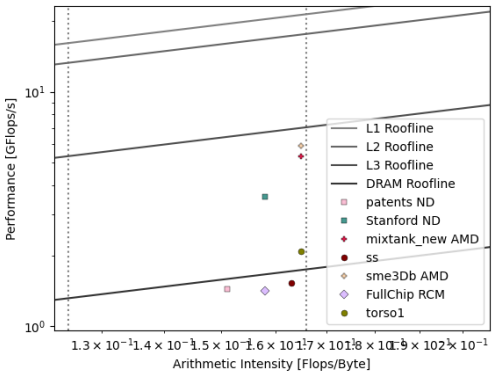


Figure 11 *Matrices with average core utilization ≈ 5 profiled in 5 Thread SpMV CARM.*

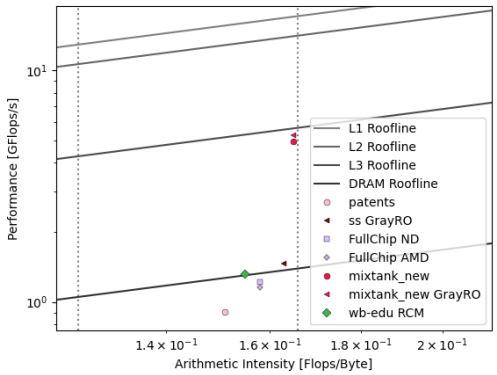


Figure 12 *Matrices with average core utilization ≈ 4 profiled in 4 Thread SpMV CARM.*

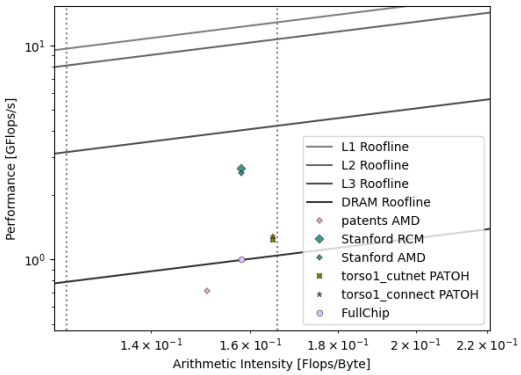


Figure 13 *Matrices with average core utilization ≈ 3 profiled in 3 Thread SpMV CARM.*

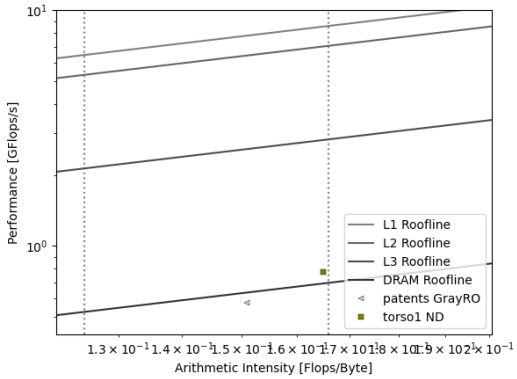


Figure 14 *Matrices with average core utilization ≈ 2 profiled in 2 Thread SpMV CARM.*

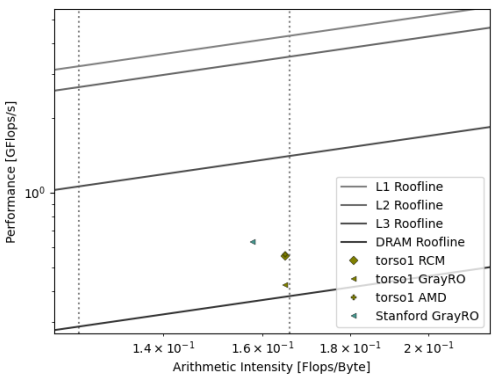


Figure 15 *Matrices with average core utilization ≈ 1 profiled in 1 Thread SpMV CARM.*

worth the pre-processing considering the maximum attainable performance.

With this, the multi-threaded performance model offers additional insights on where the

optimization process should focus. For example, if a specific matrix is represented on top of a memory roof that corresponds to its average core utilization, the next optimization step should focus on improving the load balancing if the average core utilization is less than the total number of cores (maximum core utilization). If the application has good core utilization but is below the roof right above its representation, then reordering techniques can be applied to improve the accesses to the vector elements. Finally, in the case of a kernel that is represented below the memory roof in a model that does not correspond to its maximum core utilization, further optimization can be focused on both the accesses of the vector elements and load balancing.

2.1.4 ENERGY EFFICIENCY ANALYSIS AND OPTIMIZATION

Focusing on power consumption and energy-efficiency of sparse computation, we also provide herein a roofline-based analysis according to the benchmarking methodology presented in Deliverable 4.2. With this aim, an evaluation of the variation of performance, power consumption and energy-efficiency according to the AI of the kernel, *i.e.*, the Number of Non-Zeros (NNZ) per row of the input matrices, is performed.

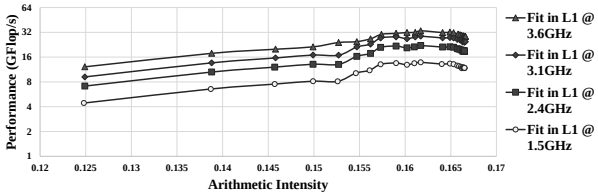


Figure 16 *Performance variation according to Core Frequency when all data fits inside L1 cache.*

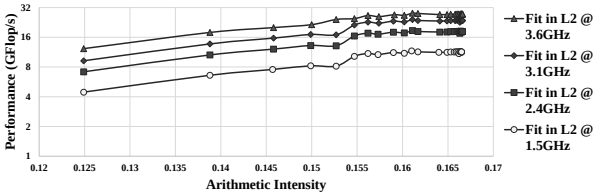


Figure 17 *Performance variation according to Core Frequency when all data fits inside L2 cache.*

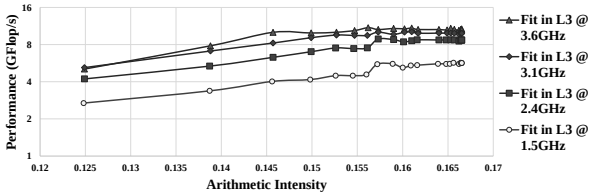


Figure 18 *Performance variation according to Core Frequency when all data fits inside L3 cache.*

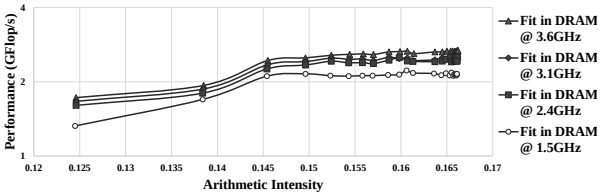


Figure 19 *Performance variation according to Core Frequency when all data fits inside DRAM.*

In order to explore possible optimization options to improve the energy efficiency for SpMV computation, the micro-benchmarking procedure proposed in Deliverable 4.2 was executed with lower core frequencies, exercising the possible AI ranges of the x86 assembly kernel. Figures 16, 17, 18 and 19 showcase how the performance is negatively affected by lowering the core frequency. For the cache levels closest to the core, such as L1 and L2, the decrease in performance is proportional to the frequency decrease, since these memory levels operate at the same speed as the core. For example, the performance loss measured between the maximum performance tests using 3.6GHz and 1.5GHz (a 58.2% decrease), was 58.6% (from 33.42 to 13.85 GFlop/s) and 58.5% (from 28.10 to 11.65 GFlop/s) for L1 and L2 respectively. For the L3 and DRAM focused testing, the relative performance decrease is lower than the one obtained for the private caches, due to the separate frequency domains at which both these memories work. For example, the performance

loss measured between maximums from testing performed in each of the memory levels, when lowering the frequency from 3.6GHz to 1.5GHz (58.23% frequency drop), was 48% (from 10.96 to 5.7 GFlop/s) and 17.1% (from 2.68 to 2.22 GFlop/s) for L3 and DRAM respectively.

Regarding the effects of frequency scaling to the power consumption, it is possible to observe in Figures 20, 21, 22 and 23, that differently from performance, the decrease in power consumption is lower than the ratio between frequencies. For example, the maximum power loss when lowering the frequency from 3.6GHz to 1.5GHz is 29.7%, 30.2%, 35.5% and 55.9% for L1, L2, L3 and DRAM focused tests respectively. Moreover, it is possible to verify that for lower frequencies, the power consumption of the Dynamic Random Access Memory (DRAM) focused tests is higher than the ones obtained for the remaining memory levels, as the former averages 36.8 W and for example, the L1 test averages 29.49 W, which is likely correlated to the fixed frequency at which the external memory works. In this scenario the power consumption in the package domain gets dominated by the active components in the memory controller instead of the other memory levels which are working at a very low frequency.

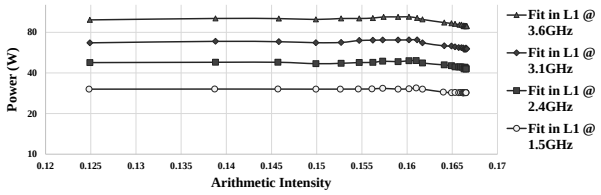


Figure 20 *Power variation according to Core Frequency when all data fits inside L1 cache.*

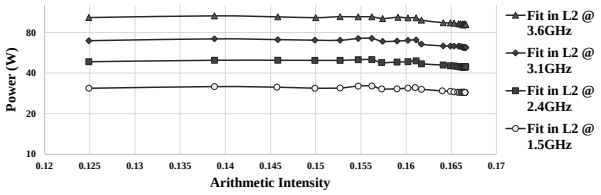


Figure 21 *Power variation according to Core Frequency when all data fits inside L2 cache.*

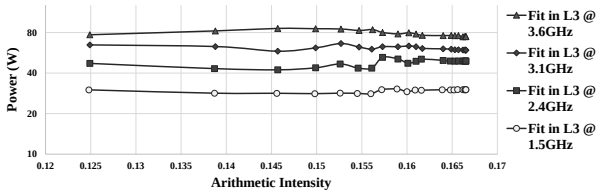


Figure 22 *Power variation according to Core Frequency when all data fits inside L3 cache.*

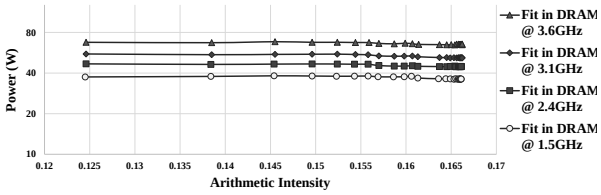


Figure 23 *Power variation according to Core Frequency when all data fits inside DRAM.*

Figures 24, 25, 26 and 27 showcase the variation of energy efficiency according to the AI when tested with different core clock speeds. As it can be observed, it is possible to attain better energy efficiency when lowering the frequency, due to higher reduction in power consumption than the one that occurs in performance. In particular for L1 and L2 caches (Figures 24 and 25), there is an increase in the maximum efficiency of 35.2% and 32.3% respectively when lowering the core frequency from 3.6 GHz to 2.4GHz. However, when further reducing the frequency from 2.4GHz to 1.5GHz, there is a slight decrease in the efficiency, showcasing that for the L1 and L2 caches, the minimum frequency is not coupled with the maximum efficiency. On the other hand, L3 and DRAM tests (shown in Figures 26 and 27), indicate that reducing the core frequency always leads to improved energy efficiency. For example, when decreasing the frequency from 3.6GHz to 1.5GHz, the energy-efficiency increases 46% and 32.4% respectively. However for lower AI values, tests fitting inside L3 show diminished returns in lowering the clock speed until 1.5GHz

as testing using 2.4GHz returns the highest efficiency values in this range.

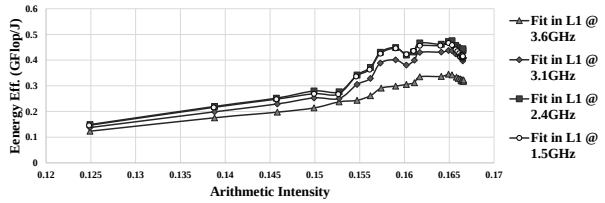


Figure 24 *Energy Efficiency variation according to Core Frequency when all data fits in L1 cache.*

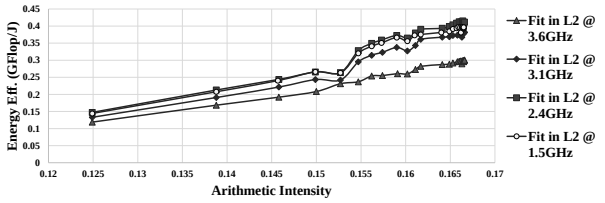


Figure 25 *Energy Efficiency variation according to Core Frequency when all data fits in L2.*

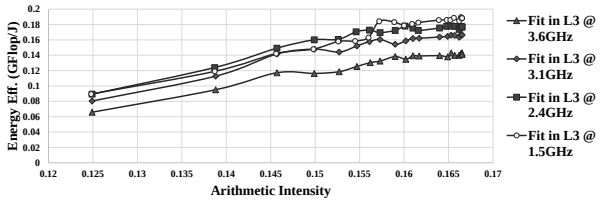


Figure 26 *Energy Efficiency variation according to Core Frequency when all data fits in L3.*

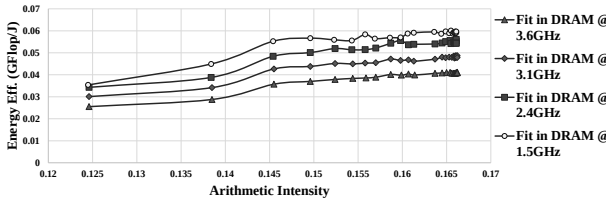


Figure 27 *Energy Efficiency variation according to Core Frequency when all data fits in DRAM.*

In order to verify the applicability of this roofline-based analysis to the optimization of power consumption and energy efficiency, some of the previously tested real matrices were evaluated by considering the performance, power consumption and energy-efficiency curves with varying core frequencies. To simplify the analysis, the selected matrices have almost perfect load balancing, with core utilization being close to the maximum cores used. Nevertheless, this analysis can be easily applied to imbalanced matrices by extending the micro-benchmarking methodology from Deliverable 4.2 to different number of threads. Figures 28, 29 and 30 present the analysis of performance, power and energy efficiency respectively, of Freescale1 AMD, ss RCM and ns3Da RCM for different core frequencies, by relying on the curves obtained for the memory levels which limit their performance.

The performance and power variation (seen in Figures 28 and 29) in the computation of the matrices presents a very similar behaviour regarding the relative loss compared to the previous tests throughout the AI range of the kernel. As can be seen, the performance drop is less noticeable for the matrices limited by the DRAM compared to the other matrix which was able to be stored inside the L3 cache, due to the DRAM frequency remaining unchanged. Likewise, analysing the power variation, all tested matrices reach similar values in the same AI of the curves.

When observing the energy efficiency variation (presented in Figure 30), for all of the tested matrices, lowering the core frequency increases energy efficiency. Furthermore, the matrices that are focused on DRAM, Freescale1 AMD and ss RCM, are placed very close to the perceived best cases for energy efficiency for any of the tested frequencies, meaning that improving their efficiency was only attained by lowering the clock speed of the core.

The portrayed scenario indicates that in situations where optimizing for energy efficiency is more important than improving performance, lowering the core clock frequency is a simple method to further increase the efficiency of the SpMV computation. Looking back at the multi-threaded testing performed in real matrices, the Freescale1 and AMD reordered version (as seen

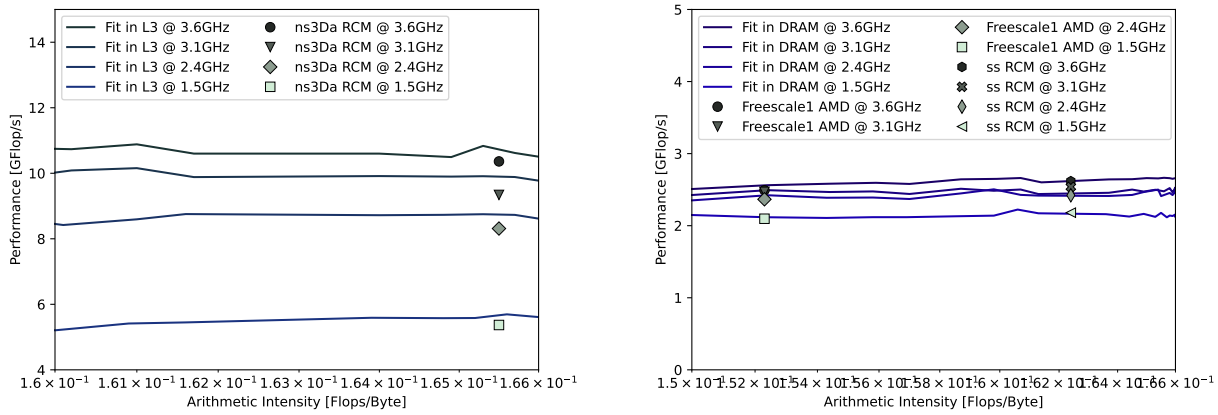


Figure 28 *Variation in Performance of SpMV applied to real matrices when changing core frequency.*

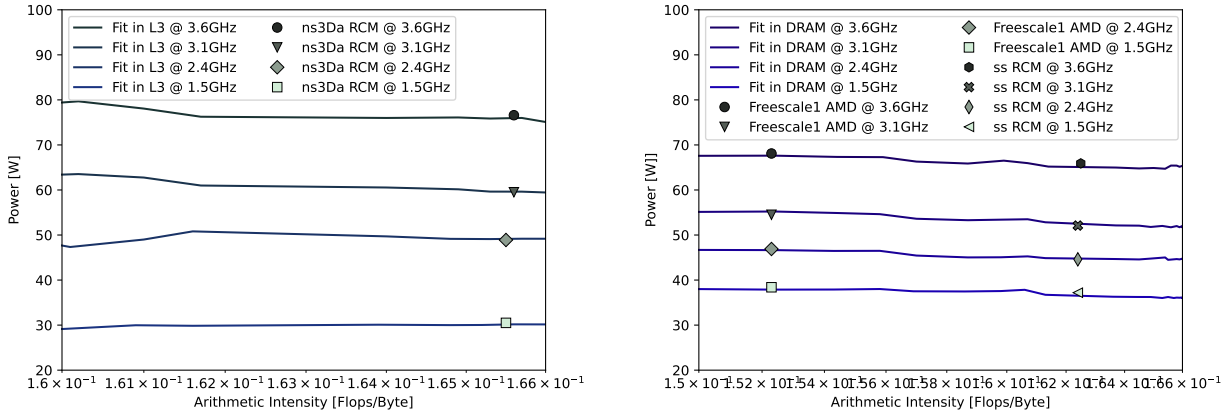


Figure 29 *Variation in Power consumption of SpMV applied to real matrices when changing core frequency.*

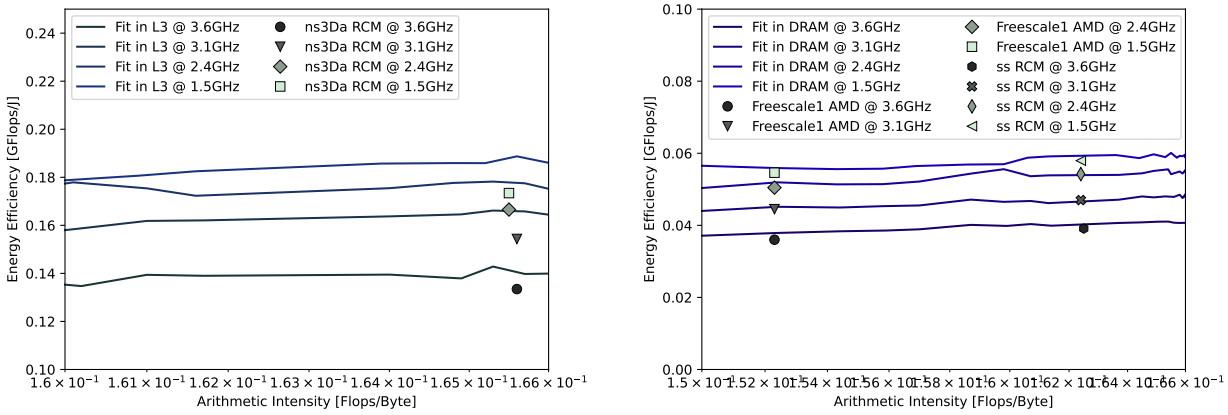


Figure 30 *Variation in Energy Efficiency of SpMV applied to real matrices when changing core frequency.*

in Figure 7a) showcase a scenario where the memory accesses to the vector elements are already highly optimized, given their relative position to the corresponding roofline, providing a perfect scenario where lowering the clock speed of the processor, with little change in overall execution time, can increase the energy efficiency of the computation.

2.2 ROOFLINE-BASED HARDWARE SCALING FOR EFFICIENT SPARSE COMPUTING

The demand for efficient computing systems has seen a recent increase with the end of Dennard's scaling and Moore's law. In response to this challenge, the RISC-V instruction set architecture (ISA) has emerged as a promising solution. RISC-V is based on Reduced Instruction Set Computing (RISC) principles, which have gained significant popularity in both small battery powered systems and high-performance computing. Its open-source nature and lack of licensing requirements have also led to its rapid growth in adoption, both in the academic and commercial spaces. Performance analysis and modelling tools are crucial for an emerging architecture such as RISC-V, as it enables software to better leverage architectural resources, and aids in designing efficient hardware that meets application demands.

To this end, a hand-optimized SpMV implementation in RISC-V assembly is presented, which is then used to do performance analysis on a simulated RISC-V architecture. The cache dynamics of SpMV execution are explored, identifying ways the hardware may be scaled to improve performance. Through the use of the Cache-Aware Roofline Model, the algorithm's range of arithmetic intensities is analyzed, aiding in the hardware design process through the identification of application demands and bottlenecks.

2.2.1 SPMV IMPLEMENTATION

The proposed SpMV implementation operates on Compressed Sparse Row (CSR) format matrices, with the matrix and vector being composed of 64-bit floating-point values, while the row offset and column index arrays use 32-bit integers. The optimized implementation relies on the unrolling of the inner loop that iterates through the elements in one row, with an unrolling factor of 8. When iterating through the elements in each row, the number of remaining non-zero values is determined, and used to calculate a target jump address. If 8 or more values remain, the execution jumps to the start of the unrolled section. If there are fewer than 8 values remaining, the program jumps within the unrolled portion so that as many elements as remaining are processed.

2.2.2 ARCHITECTURE AND METHODOLOGY

The aforementioned implementation was tested using gem5,[10] a computer architecture simulator with a number of parameterizable CPU models. The Minor CPU, an in-order model with a four-stage pipeline is used, initially parameterized with a 32kB L1 cache with 4 ways, and a 128kB L2 cache, both with 64 byte lines, running at 2GHz.

2.2.3 EXPLORING CACHE DYNAMICS

In order to explore the performance impact the caches have on the execution, two synthetic sparse matrices were generated, referred to as the best- and worst-cases. Both matrices are of size (D, D) where D is the number of 8 byte elements that the L1 cache plus an extra line can contain, in this case 4104. The matrices are given this size so that the vector does not entirely fit in the L1 cache, which allows dynamics such that the reuse of the vector's data and cache eviction to be explored.

[10] Jason Lowe-Power et al. "The gem5 Simulator: Version 20.0+". en. *arXiv:2007.03152 [cs]* (2020). arXiv: 2007.03152. URL: http://arxiv.org/abs/2007.03152 (visited on 12/28/2021).

The worst-case matrix aims to make no reuse of the vector data stored in the L1 cache, evicting previously cached data with every access. Since the L1 cache is 4-way associative, 5 accesses to distinct addresses mapping to the same cache line must be made in order to evict the vector's previously cached data. As the 32kB L1 cache has 8kB per set, and we are indexing a vector of 64-bit or 8 byte elements, the column indexes of each row are generated at an increment of 1024, leading to 5 accesses mapping to the same cache address per row. In other words, each row of the spare matrix contains 5 non-zero elements, spaced 1024 columns apart. This causes the desired eviction on every access of the vector, ensuring it is not stored in the L1 cache.

The best-case matrix has the opposite goal, which is maximizing the reuse of cached data. In order to better establish a direct comparison, the number of accesses per row is maintained constant, i.e., the number of non-zero elements per row is still 5. However, the column indexes of these non-zero elements instead allow for contiguous access to the elements of the vector. Since each 64 byte cache line can contain 8 elements of the vector, each read always accesses the same cache line, which should be cached in L1 after the very first element is read. This way there is maximal data reuse, given the accessed vector elements are always in the L1 cache after the first read.

The two matrices are processed, with their properties and performance results shown in Table 2. As expected, the best-case matrix significantly outperforms the worst-case matrix in performance, despite the two differing exclusively in the column indexes. This highlights the cache's impact on performance, even when it is impossible to cache the entire set of data.

	Worst-case	Best-case
Dimensions	(4104, 4104)	(4104, 4104)
NNZ	20520	20520
FLOP	45144	45144
Bytes	459652	459652
Performance (GFLOP/S)	0.072	0.119

Table 2 *Properties and performance of the best- and worst-case matrices*

In order to mitigate the issues with cache eviction seen in the worst-case scenario, the architecture is reparametrized with two approaches, altering the L1 cache's dimensions and its associativity. Both matrices are processed in the new architecture, with the results presented in Table 3. As illustrated by the results, both modifications of the cache achieve the same result, fully mitigating the decrease in performance seen in the worst-case matrix, as cache eviction is prevented and the vector is stored in the L1 cache for both cases.

	Size = 48kB	Associativity = 8
Worst-case performance (GFLOP/S)	0.119	0.119
Best-case performance (GFLOP/S)	0.119	0.118

Table 3 *Performance of the best- and worst-case matrices following changes to the L1 cache*

While both changes result in nearly the same performance, it should be noted that they would differ significantly in resource usage when implemented. Depending on the power and area budgets of the design, a designer might prefer the increase in associativity for its much more modest resource usage. It is also of note that the synthetic nature of the two matrices may lead to

the results not translating fully to real data – if accesses to the vector are not sufficiently sparse, an increase in associativity may not be sufficient to fully mitigate cache evictions, in which case an increase of the cache size may be necessary for optimal performance.

2.2.4 EXPLORING THE RANGE OF ARITHMETIC INTENSITIES

By analysing the presented SpMV implementation, it is possible to determine the number of floating-point operations performed and bytes transferred during execution from the characteristics of the matrix. With these, it is possible to calculate the arithmetic intensity of the workload as follows:

$$AI = \frac{FLOPs}{bytes} = \frac{R + 2 \cdot nnz}{4 + 20 \cdot nnz + 12R}, \tag{1}$$

where R represents the number of rows and nnz the number of non-zero values of the matrix. By manipulating and simplifying the expression, we can determine the arithmetic intensity exclusively as the ratio between non-zero values and rows, such that:

$$AI = \frac{2\frac{nnz}{R} + 1}{\frac{4}{R} + 20\frac{nnz}{R} + 12} \approx \frac{2\frac{nnz}{R} + 1}{20\frac{nnz}{R} + 12}, \text{ when } R >> 1. \tag{2}$$

By defining the limits of the aforementioned ratio, we can determine the range of arithmetic intensities achievable. For the sake of applicability, the minimum ratio is limited to 1, thus the matrices with empty rows are excluded from this analysis. Since there is no upper limit to the ratio of non-zeros and rows, it can be concluded that the range of arithmetic intensities covered by the adopted SpMV kernel is $\left[\frac{3}{32}, \frac{1}{10}\right[$. It is possible to create a matrix exhibiting any arithmetic intensity within this range by selecting the ratio between the number of rows and the number of non-zero values.

In this vein, two sets of matrices of varying arithmetic intensities and sizes are created, following the same principles as the previously presented best-case matrix. Table 4 shows the characteristics of each set of matrices, the "L1 set" and "L2 set", named after the cache level they target.

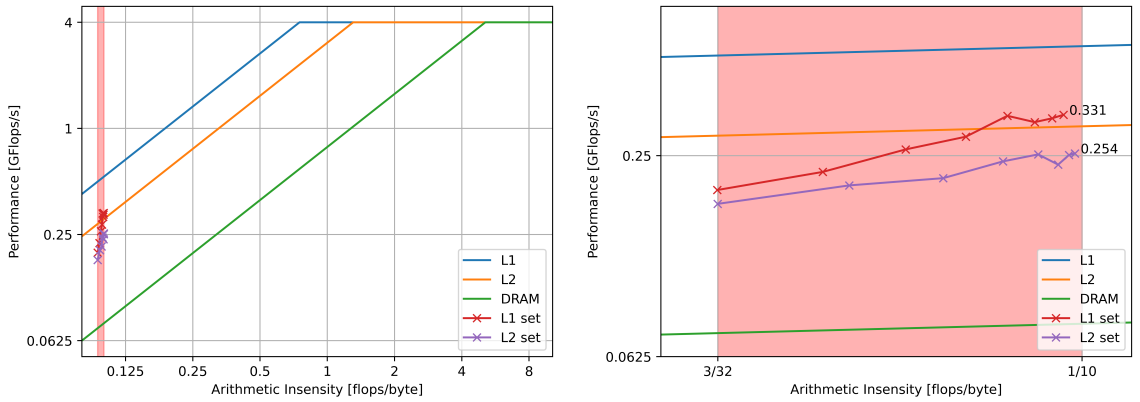
	L1 set	L2 set
Target Size (kB)	24	64
NNZ	1228	3276
Columns	614	1638

Table 4 *Characteristics of the two sets of matrices*

As all matrices in each set share the same number of non-zero values and columns, the change in the $\frac{nnz}{R}$ ratio (which governs the arithmetic intensity) is achieved by varying the number of rows. The non-zero values are evenly distributed among the rows for optimal uniformity, with matrices that have fewer rows becoming more dense.

The system is first benchmarked in order to build its Cache-Aware Roofline Model, after which the sets of matrices are processed. Each matrix is processed twice in order to level the cache data and obtain steady-state performance figures (closer to the CARM's rooflines), which are measured during the second execution. Figure 31 shows the architecture's CARM, along with the performance of both sets plotted on it.

The results of both sets show an increase in performance as the arithmetic intensity increases, consistent with what is expected due to the memory-bound nature of the range in study. The set targeting the L1 cache is capable of delivering a higher performance than the one targeting the L2



(a) *Full view of the system's CARM*

(b) *The CARM plot zoomed in on the relevant arithmetic intensity range, annotated with each set's maximum performance*

Figure 31 *Performance of the two matrix sets plotted on the system's CARM*

cache, as expected due to the higher L1 bandwidth, and resulting higher roofline. However, it is clear both performance curves are positioned significantly below their theoretical maximums, i.e., their respective rooflines. One of the reasons for this behaviour lies in the in-order nature of the processor, which execution paradigm differs from CARM's assumption that memory and floating-point operations execute in parallel. Additionally, the algorithm is significantly more complex than the synthetic benchmarks used to measure the memory bandwidth and peak performance with which the CARM is built, with other dynamics at play such as frequent conditional branching and data dependency. Despite the results not fully correlating with the model, the CARM can still be used to expeditely determine the performance upper limits of the arithmetic intensity range being operated in.

2.2.5 Improving SpMV efficiency with roofline-based architecture scaling

In this part of the study, we focus on exploring the possibility of improving the SpMV efficiency by following the hardware utilization insights given by the CARM. In a nutshell, we aim at scaling the capability of different hardware resources in order to meet the demands of the application. While in the analysis of cache dynamics the goal was the improvement of performance through the reparameterization of the cache, the objective now is to scale down the architecture to reduce the hardware's footprint, while maintaining a similar performance level.

As Figure 31a shows, the SpMV kernel is positioned deeply in the memory-bound region, and the attainable AIs range is very limited (see transparent shaded region), meaning this kernel will never be able to exploit the compute upper-bounds of this architecture (see the distance from the horizontal roof). Given these observations, the peak (compute) performance can be significantly reduced while the current performance (limited by memory bandwidth) can be kept, scaling the architecture to closely meet the application requirements. In this study, this scaling is achieved by replacing the floating-point unit with a non-pipelined implementation, where the unit's latency and its effect on performance is explored.

With the insights provided by the CARM, it is possible to determine the latency that minimizes the peak performance, while not reducing the attainable performance at a desired arithmetic intensity. Equation 3 describes the attainable performance according to the CARM, where B is the bandwidth of the memory level in study, I is the arithmetic intensity, and F_p is the peak

performance.

$$F_a(I) = \min\{B \cdot I, F_p\} \tag{3}$$

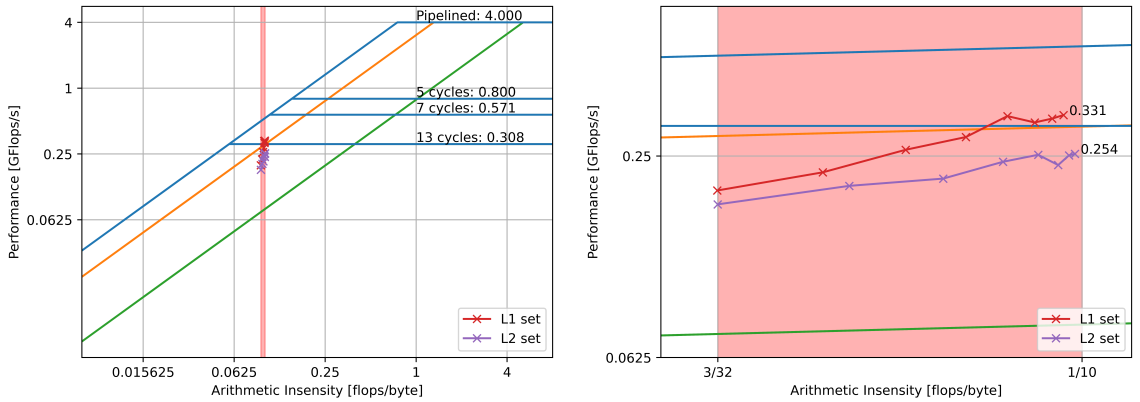
By setting a goal of preserving the performance across all memory levels at the upper limit of the arithmetic intensity range, we consider the memory level with the highest bandwidth, the L1 cache. Given that the range in study is memory-bound, the attainable performance is governed by the equation's left term, and we may calculate as (4):

$$F_a(\frac{1}{10}) = B_{L1\to C} \cdot \frac{1}{10} = 0.532 \text{ GFLOP/s} \tag{4}$$

We may then determine the maximum latency that results in a peak performance above this result (4) using Equation 5, where F_{po} denotes the system's original peak performance of 4 GFLOP/s:

$$\text{Latency} = \text{ceil}(\frac{F_{po}}{F_a(\frac{1}{10})}) = 7 \text{ cycles} \tag{5}$$

Two more latency values are tested in order to better study this parameter's impact on performance, these being 5 and 13 cycles. The former was chosen to provide a higher peak performance, while the latter was obtained by performing the process shown in Equations 4 and 5 for the L2 cache's bandwidth. The resultant rooflines, along with the original one, and the SpMV performance of the original system are presented in Figure 32.



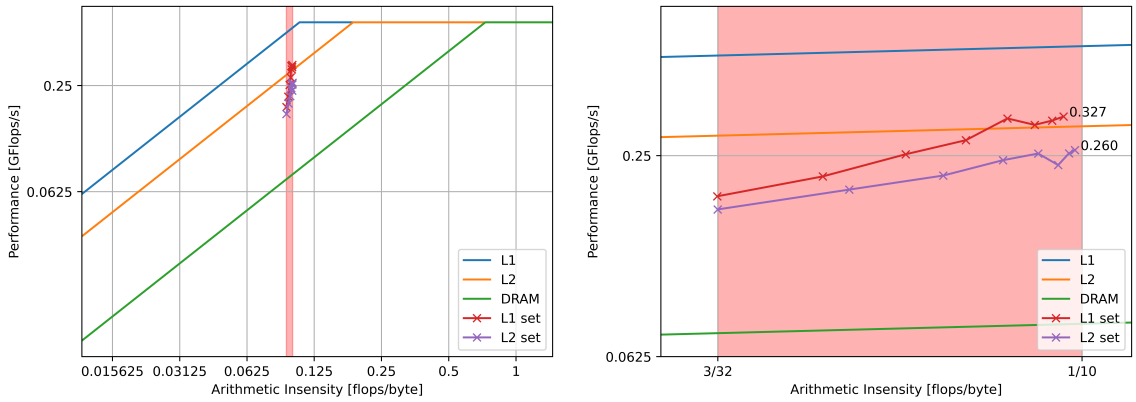
(a) *Full view of the CARM resulting from each of the latencies, annotated with the respective peak performance*

(b) *The CARM plot zoomed in on the relevant arithmetic intensity range, annotated with each set's maximum performance*

Figure 32 *Comparison between the rooflines of the 4 parameterizations and the SpMV on the original system*

As predicted, the ridge points of the 7-cycle system's L1 roofline and of the 13-cycle system's L2 roofline are immediately to the right of arithmetic intensity range's upper limit, preserving the respective performance upper-bounds. The compute-bound roof of the 13-cycle system is lower than the observed SpMV performance on the original system, and as such a drop in performance is expected. The SpMV performance is then measured on each of the systems, with the results shown below, beginning with the 7-cycle system in Figure 33.

As demonstrated by the results, despite the significant reduction to the system's peak performance, from 4 GFLOP/s to approximately 0.571 GFLOP/s, the performance of SpMV remains virtually unchanged. By determining the ratio between the achieved performance and the peak performance, we can evaluate if efficient usage of the hardware's resources is being made. In the



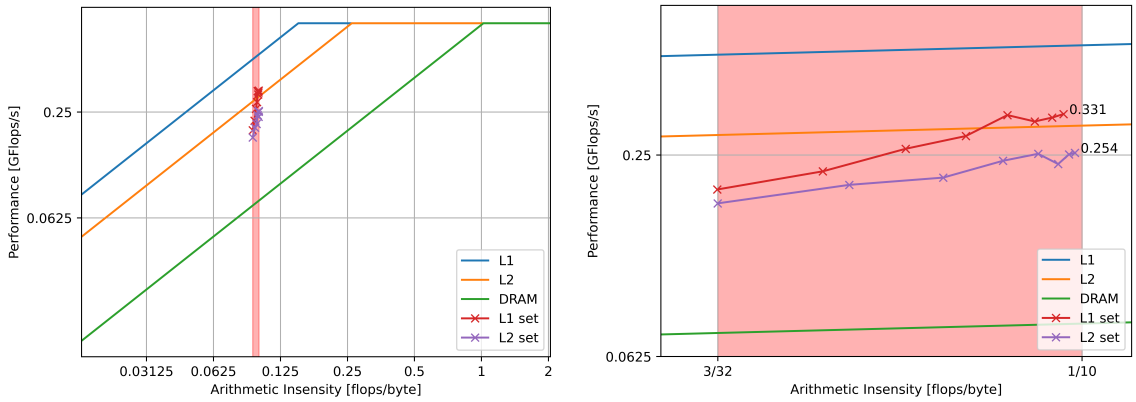
(a) *Full view of the system's CARM*

(b) *The CARM plot zoomed in on the relevant arithmetic intensity range, annotated with each set's maximum performance*

Figure 33 *Performance of the two matrix sets plotted on the system's CARM after the floating-point unit's reparameterization to a 7-cycle latency*

case of the L1 set, while in the original system the execution reaches 8.3% of the peak performance, the reparameterization increases it to 57.3%, bringing the architecture's capabilities much closer to the requirements.

The 5-cycle system is then tested, with the outcome presented in Figure 34. This results in a higher peak performance when compared to the 7-cycle reparameterization, at 0.8 GFLOP/s, and similarly it does not affect the SpMV performance. However, this also leads to a less efficient use of resources, with the ratio between achieved and peak performance sitting at 40.9%.



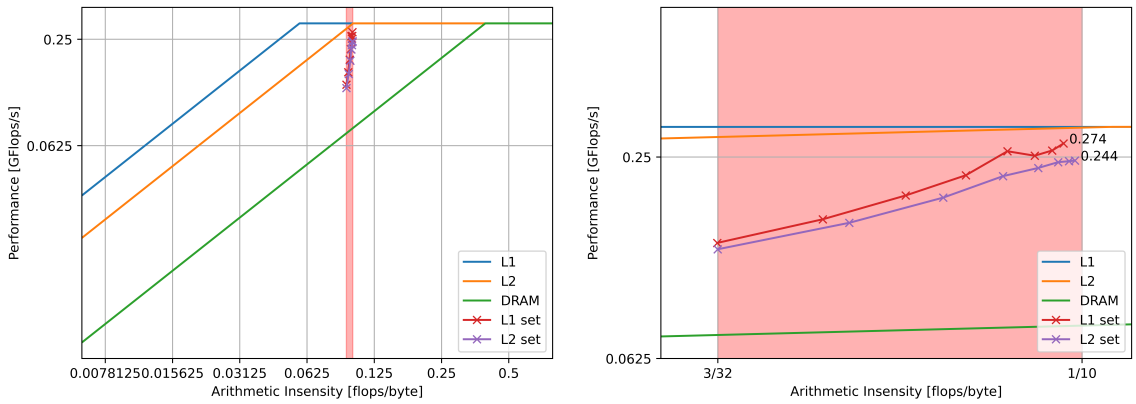
(a) *Full view of the system's CARM*

(b) *The CARM plot zoomed in on the relevant arithmetic intensity range, annotated with each set's maximum performance*

Figure 34 *Performance of the two matrix sets plotted on the system's CARM after the floating-point unit's reparameterization to a 5-cycle latency*

Lastly, the 13-cycle system is evaluated, with the results shown in Figure 35. The system now attains a peak-performance of 0.308 GFLOP/s, which is lower than the originally achieved performance of the L1 set. Consequently, this leads to a reduction in this set's performance by approximately 17.2%, as it is no longer able to benefit from the the L1 cache's high bandwidth. With the sharp reduction in peak-performance, the execution becomes bottlenecked by the floating-

point unit, bringing it closer to the L2 set. Despite the L2 set originally performing below the L2 roofline, a slight reduction in performance of 3.9% is also observed. This demonstrates some level of reuse of L1 data in this set, despite the performance in the original system staying below the L2 roofline. As explained previously, elements of the vector may be cached in L1 even if the entirety of the dataset does not fit in it, leading some sections of the algorithm benefiting from a higher memory bandwidth, and achieving higher performance due to their memory-bound nature. By reducing the peak performance, those sections instead become compute-bound.



(a) *Full view of the system's CARM*

(b) *The CARM plot zoomed in on the relevant arithmetic intensity range, annotated with each set's maximum performance*

Figure 35 *Performance of the two matrix sets plotted on the system's CARM after the floating-point unit's reparameterization to a 13-cycle latency*

While experimental results are not available, we can speculate about the power consumption of the new floating-point unit, comparing the original system with the first reparameterization, which has lowest-performing, presumably most efficient floating-point unit that does not harm SpMV performance. Seeing as the unit is no longer pipelined and has a 7-cycle latency, its input frequency is reduced by a factor of 7. This reduction is likely to provide a significant increase in energy efficiency due to the strong correlation between a digital circuit's power consumption and its frequency.

Other approaches to reduce the peak performance are available, such as providing the caches and the core with different clocks, and reducing the clock frequency of the latter, which would likely lead to a significant decrease in power consumption in the whole core. It should be noted that this may have a more pronounced negative impact on performance in this particular case, given that the execution is not exclusively limited by the memory system, and the reduction in processor frequency may exacerbate the slowdowns caused by branching and data dependency.

2.2.6 CONCLUSION

Overall, the results demonstrate the pronounced impact the processor's cache has on performance, particularly due to the memory-bound nature of the workload, which is evidenced with the aid of the CARM. Through an analysis of the algorithm's arithmetic intensity range, insights provided by the CARM allowed for an informed redesign of the architecture, leading to more power efficient execution. This study demonstrates the key role a performance modeling tool such as the CARM can play in hardware design, allowing the architecture to be tailored to the requirements of any workload, and improving the overall efficiency.

3 EXPLORING THE PROCESSING LIMITS OF SPMM AND TTM ON CPU/GPU SYSTEMS

3.1 ANALYSIS OF SPARSE MATRIX MULTIPLICATION ON A GPU DEVICE

SpMM kernels, which perform matrix multiplication between a sparse matrix and a dense matrix, are widely used in High Performance Computing (HPC) scientific and engineering applications. The processing of sparse matrices provides advantages such as requiring less storage space and the fact that computations can in some cases be performed significantly faster than with their dense counterparts. This is, respectively, the result of not storing (any or all of) the zero elements and only needing to compute with the non-zero elements of the sparse input matrix.

Modern accelerator devices such as Graphics Processing Units (GPUs), which are widely used as accelerators for both dense and sparse matrix multiplication kernels, provide Dynamic Voltage and Frequency Scaling (DVFS) capabilities to comply with power constraints and increase energy efficiency. Depending on the application load, different frequencies can be selected on-the-fly for the accelerator compute and memory components. These can be set independently to exploit workload characteristics. Particularly important is the amount of computations in relation to global memory loads/stores. For some compute-bound workloads it might be preferable in terms of power and energy efficiency to lower memory clocks while keeping core clocks high. In contrast, for memory-bound workloads, one might lower core clocks with no significant effect on performance.

In some cases, the use of SpMM kernels can exercise the available compute and memory resources differently than GEMM kernels. While the latter is typically compute-bound, and as a result often used as a benchmark to show peak performance of parallel accelerator devices, operations involving sparse matrices can be memory-bound to varying degrees. As a result, given an hardware platform or device, the optimal core and memory frequencies can depend on the representation format or algorithm used, on the particular inputs, i.e. matrix dimensions, sparsity/density levels (number of zeros to non-zeros, or vice-versa), and on the numerical precision used and hardware support for native efficient operation at the used precision.

We evaluate the potential of tuning the core and memory frequencies of an Ampere GPU accelerator device for optimizing the execution of SpMM (and GEMM). Cross-comparing between the use of different methods for multiplying sparse matrices, we analyse the effect of GPU core and memory frequencies on the achieved throughput, average power and energy efficiency (and resulting energy consumption), taking into account inputs with different sparsity/density levels, the use of different matrix representation formats, and the use of 32-bit (float) or 16-bit (half) floating-point precision for data representation.

3.1.1 TARGETED DEVICE, REPORTED METRICS AND EXECUTION VARIABLES

Targeted device. All experiments have been performed on an Intel Core-i7 system with an NVIDIA RTX 3080 Ampere (10GB / GA102-200-KD-A1 variant) GPU accelerator. The targeted GPU has 64 Streaming Multiprocessors (SMs) 8704 CUDA cores (128 per SM) and 272 tensor cores (4 per SM). The GPU has a base core frequency of 1440 MHz and a boost core frequency of 1710 MHz. In the Ampere architecture, each CUDA core is capable of performing a native 32-bit (or 16-bit) fused-multiply add operation per clock cycle (1:1 FP16 to FP32 ratio). Thus, on the targeted GPU, the peak floating-point throughput on CUDA cores (calculated at boost frequency) is 29.77 TFLOP/s ($= 8704 \times 1710 \times 2$). On the Ampere GA102 GPU microarchitecture, each tensor core performs 64 fused multiply-add 16-bit floating-point operations with 32-bit

accumulation, totaling in a throughput of 59.54 TFLOP/s ($= 64 \times 272 \times 1710 \times 2$) considering the use of the whole device. Global memory is of the GDDR6X type and can go up 1188MHz (19 Gbit/s effective speed), reported as 9501MHz in the NVIDIA System Management Interface (`nvidia-smi`) command line utility, due to this type of memory sending 8 bits of data per clock cycle (quad data rate and PAM4 signaling). The bandwidth to global memory is 760.3 GB/s. All experiments have been performed relying on CUDA 12.0 and GPU Driver version 525.85.12.

Reported Metrics. We report the achieved throughput in Tera FLoating-point Operations Per Second (TFLOP/s) and the average power consumption in Watts (W). Throughput is calculated based on the execution time of the given matrix multiplication run and on the actual number of performed floating-point operations. Time measurements are performed relying on the absolute elapsed wall-clock time for executing matrix multiplication on the GPU, being obtained on the host with `clock_gettime(CLOCK_MONOTONIC, &time)`. Power measurements are performed relying on the NVIDIA Management Library (NVML), an API for monitoring and managing NVIDIA GPUs. Energy efficiency, which is calculated from the previous metrics, is reported in Giga FLoating-point Operations Per Joule (GFLOP/J). Notice that the cost of populating the input matrices, converting to/from matrix formats, or any other host code preceding/following the execution of matrix multiplication routines on the GPU are not taken into account in the performance, power and energy efficiency metrics. There is a minimum warm-up period of 5 seconds before any measurement is performed, which is accomplished by running the matrix multiplication kernel under study as many times as needed. Multiple executions are also performed during the measurement phase to increase the precision of the reported metrics. After the warm-up period, on the measuring phase, the kernel is again executed for a minimum time of 5 seconds, being the execution time for a single kernel execution estimated by dividing the elapsed time by the amount of kernel executions.

Execution Variables. We use the cuBLAS[11] and cuSPARSE[12] libraries for performing matrix multiplication targeting CUDA cores and (dense) tensor cores. These are the defacto libraries for GEMM and SpMM (and related) computations on NVIDIA GPUs. The versions used are those included in the CUDA software stack, being the default versions of the algorithms of each of these libraries the ones for for the experiments. Native hardware-accelerated SpMM is performed relying on CUTLASS[13] (version 2.11), which is a set of CUDA C++ abstractions for implementing matrix-multiplication and related computations. CUTLASS supports the Sparse Matrix Multiply-Accumulate (MMA) Ampere tensor cores, which cuBLAS and cuSPARSE do not.

To evaluate the impact frequency scaling, we performed runs with different GPU core and memory frequencies, set through the `nvidia-smi` tool before execution of a CUDA program performing matrix multiplication. For the GPU core we alternatively use a frequency of 345MHz, 690MHz, 1020MHz, 1365MHz or 1710MHz, which are the closest supported frequencies to 20%, 40%, 60%, 80% or 100% of the vendor-announced boost frequency. Memory is set alternatively to 810MHz, 5001MHz, 9251MHz or 9501MHz, the four highest frequencies supported by the driver.

As the starting point for all experiments, we rely on dense representations for representing both input matrices. For SpMM runs, the first matrix input is converted to a sparse representation at runtime. In the experiments using cuSPARSE to perform SpMM, we consider the Coordinate (COO), Compressed Sparse Column (CSC), CSR and Blocked-Ellpack (Blocked-ELL) formats,

[11] NVIDIA Corporation. *cuBLAS*. version 12.0. 2023. URL: https://developer.nvidia.com/cublas.
[12] NVIDIA Corporation. *cuSPARSE*. version 12.0. 2023. URL: https://developer.nvidia.com/cusparse.
[13] Andrew Kerr et al. *CUTLASS*. version 2.11.0. 2022. URL: https://github.com/NVIDIA/cutlass.

which are the formats supported for representing sparse matrices in this library. The experimental campaign considers the following different sparsity levels: ~ 50.00%, ~ 75.00%, ~ 87.50%, ~ 93.75%, ~ 96.88%, ~ 98.44%, ~ 99.22%, ~ 99.61%, ~ 99.81%, ~ 99.90%.

In the experiments using the CUTLASS library for sparse tensor-core accelerated SpMM, we use specially built matrices complying with the 2:4 fine-grained sparsity format. This format represents the first input matrix (the sparse matrix) as two smaller matrices — a compressed matrix (half the values of the dense representation) and a matrix with 2-bit indices, indicating the positions of the matrix elements in the dense matrix to be multiplied with those on the compressed matrix. To use the 2:4 fine-grained sparsity format, there must exist at least two zeros in each set of four contiguous values in any given matrix row (row-major representation). Notice that the use of this format is a strict requirement to enable using the sparse tensor cores.

The matrices are represented relying on the use of the 16-bit IEEE-754 floating-point (half) or the 32-bit IEEE-754 floating-point (float) numerical formats. Using cuBLAS, we evaluate the use of uniform-precision GEMM operation (on CUDA cores) and that of mixed-precision GEMM using 16-bit inputs with 32-bit accumulation (on CUDA cores and on Tensor cores). On cuSPARSE SpMM, we evaluate the use of the COO, CSR and CSR formats using uniform 32-bit precision. When using the (dense) tensor cores on cuSPARSE SpMM, which relies on the Blocked-ELL format, 16/32 mixed-precision is used. Finally, when using CUTLASS to make use of the sparse tensor cores, we also rely on mixed-precision with 16-bit inputs and 32-bit accumulation. Notice that direct operation with 32-bit data is not supported in tensor cores (either dense or sparse). When using 16-bit precision inputs we always rely on 32-bit accumulation (i.e. mixed precision), since that is most representative of real use cases. As a matter of fact, cuSPARSE does not support uniform 16-bit precision SpMM for the COO, CSC, CSR formats.

3.1.2 GEMM ON CUDA CORES AND ON TENSOR CORES

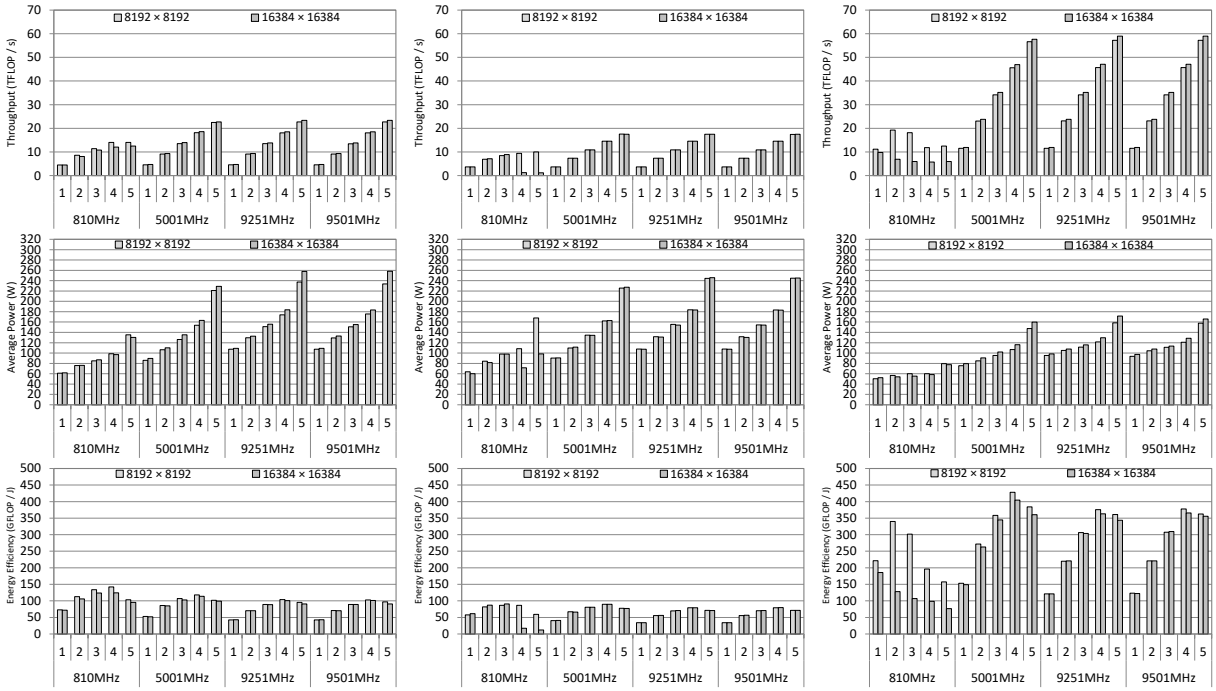
Fig. 36 depicts the throughput, power and energy efficiency achieved with different pairings of core and memory frequencies, for GEMM runs (M,N,K set to 8192 or 16384) using the cuBLAS library. Fig. 36(a) and Fig. 36(b) show these metrics for the use of 16-bit and 32-bit precision inputs on CUDA cores. The same metrics are showcased in Fig. 36(c) for the use of 16-bit precision inputs on tensor cores.

The achieved results indicated that close to the full potential of the cuBLAS GEMM kernels is being achieved, with no significant difference in the throughput achieved (TOPS/s) for processing matrices with 8192×8192 or 16384×16384 elements. Execution relying on cuBLAS uniform-precision 32-bit GEMM on CUDA cores achieved up to 23.368 TFLOP/s, which represents 78.50% of the vendor-announced throughput (29.77). Mixed-precision cuBLAS GEMM on CUDA cores resulted in worse throughput, achieving only 17.44 TFLOP/s. This can be attributed to the execution of more instructions, since mixed-precision is not natively supported in CUDA cores.

Higher throughput on CUDA cores (29.49 TFLOP/s, 99.07% of peak) has been achieved with uniform-precision 16-bit operation, i.e. 16-bit inputs and 16-bit accumulation (not represented in the charts). However, since this is not safe to use for many applications and not supported in several libraries (e.g. not supported in COO, CSC and CSR on cuSPARSE), uniform-precision 16-bit operation has not been included as part of the main experimental campaign.

Operation using tensor cores allowed to achieve a significantly higher throughput (up to 58.96), which represents 99.03% of the peak (59.54) at the used precision, i.e. 16-bit inputs with 32-bit accumulation. This is to expect, since tensor cores are hardware units especially designed for performing matrix multiplication operations at high throughput.

The reported average power consumption is comparable between uniform precision and



(a) *Uniform-precision 32-bit GEMM on CUDA cores.*

(b) *Mixed-precision 16/32-bit GEMM on CUDA cores.*

(c) *Mixed-precision 16/32-bit on tensor cores.*

Figure 36 *Throughput, power and energy efficiency for cuBLAS GEMM with different GPU core (1: 345MHz, 2: 690MHz, 3: 1020MHz, 3: 1365MHz or 5: 1710MHz) and memory (810MHz, 5001MHz, 9251MHz or 9501MHz) frequencies, with input matrices of size* 8192×8192 *or* 16384×16384 *elements.*

mixed-precision matrix multiplication on CUDA cores, 258.10W or 245.10W for processing 16384×16384 matrices at the highest clock/memory frequencies. However, the use of tensor cores, for the same core/memory frequencies, results in a much lower power consumption (165.88) than if relying on CUDA cores for the GEMM computations. This can be attributed to the specialization of these special-purpose acceleration units for the task at hand, since tensor cores are built for the sole purpose of multiplying matrices.

As expected, throughput is at its highest when pairing the highest GPU core and memory frequencies considered. However, depending on the metric(s) one wants to optimize, there are benefits in using lower GPU core and/or memory frequencies. The fact that, overall, throughput is significantly affected when using lower core frequencies (and not lower memory frequencies) can be attributed to the compute-bound nature of GEMM computations.

Core frequency variation resulted in a more significant impact on power consumption than memory frequency variation. Lowering the core frequency results in a drop in voltage, resulting in a drop in power consumption. This is especially the case when dropping from 1710MHz to 1365MHz, as the higher is the frequency a chip is operating at, the less efficient it becomes.

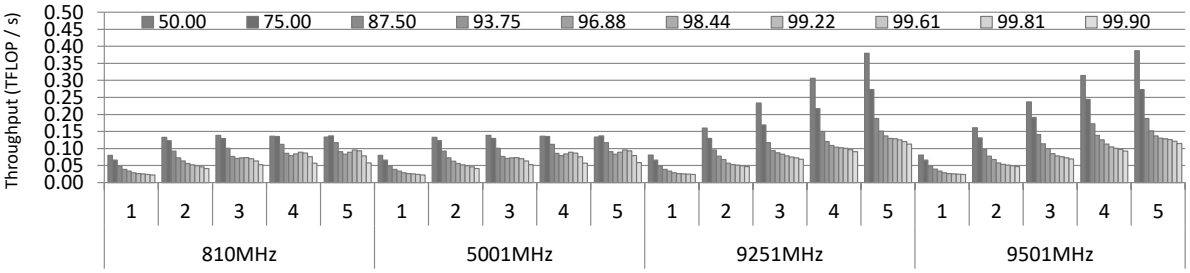
Dialing back the memory frequency can be useful for the resulting power consumption reduction; especially because it could often be performed with a negligible impact on performance. As a matter of fact, using a memory frequency of 5001MHz (52.64% of the maximum) is enough to sustain close to peak performance for all considered core core frequencies. Memory frequency only has a very noticeable impact on throughput when set to its lowest considered value (810MHz). This is to expect since at such point GEMM operations become memory bound.

Notice that these trends can be observed for all three considered execution modes. While

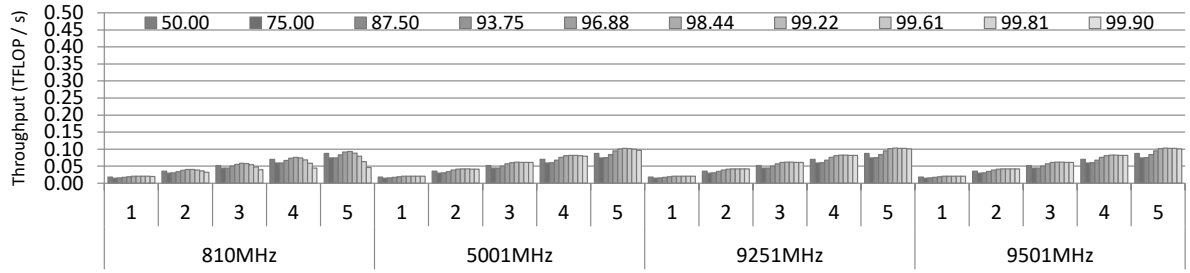
power consumption using a memory frequency of 9251MHz or 9501MHz is similar, for processing 16384×16384 matrices, an absolute reduction in power consumption of 12.73W up to 24.54W, 16.79W up to 20.30W, and 5.95W up to 17.66W has been registered when using a memory frequency of 5001MHz (compared to the maximum, i.e. 9501MHz). For GEMM on tensor cores, highest energy efficiency is achieved by setting memory frequency to 5001MHz. Execution on CUDA cores benefited from the use of an even lower memory frequency (810MHz) allowed to further improve energy efficiency. For the same matrix sizes, combining with tuning of the core frequencies resulted in an energy efficiency of up to 124.13 (1365MHz) for uniform-precision GEMM on CUDA cores, 90.84 (1020MHz) for mixed-precision GEMM on CUDA cores and 404.31 (1365MHz) for mixed-precision GEMM on tensor cores.

3.1.3 SPMM ON CUDA CORES

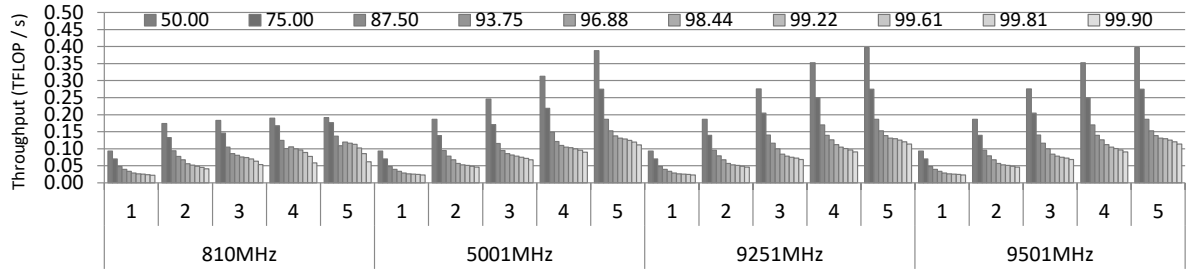
Fig. 37(a), Fig. 37(b) and Fig. 37(c) depict the throughput achieved with different core and memory frequencies, for cuSPARSE SpMM (M,N,K set to 16384) runs using the COO, CSC or COO matrix formats, respectively, and considering different sparsity levels. Notice that for the calculation of the throughput metric only the floating-point calculations effectively performed to multiply elements from the two input matrices have been considered.



(a) *Throughput using the COO sparse matrix format.*



(b) *Throughput using the CSC sparse matrix format.*



(c) *Throughput using the CSR sparse matrix format.*

Figure 37 *Throughput of cuSPARSE cuSpMM on CUDA cores with different GPU core (1: 345MHz, 2: 690MHz, 3: 1020MHz, 4: 1365MHz or 5: 1710MHz) and memory (810MHz, 5001MHz, 9251MHz or 9501MHz) frequencies, with input matrices of* 16384×16384 *elements.*

The throughput of SpMM in terms of TFLOP/s, up to 0.387 (COO), 0.103 (CSC) or 0.397 (CSR), is orders of magnitude lower than that of GEMM. This can be attributed to multiple factors. Data locality is lost in regard to fetching data from the second matrix (the dense input matrix), which does not allow SpMM to apply the same optimizations that are typically applied for GEMM kernels. Notice that skipping more computations means that in total less data is used form the dense matrix, making reuse of the data obtained through memory loads pertaining to that matrix less likely to occur. Moreover, in relation to GEMM, there are additional memory accesses to the column and/or row indices/pointers that are integral part of the sparse formats. Notice, however, that less TFLOP/s being achieved is not necessarily indicative of higher execution time. This is expected to be the case for high sparsity matrices, where the amount of NNZs is much smaller than the amount of elements in the correspondent dense representation of the same matrix.

SpMM runs with the COO or CSR formats are impacted in a similar manner in regard to the sparsity of the input matrices. For any given core and memory frequency setting, increasing sparsity results in a decrease in the achieved throughput. Matrices with higher sparsity have less non-zero elements, making SpMM work more on the memory latency, and as a result exercise less the available compute resources. SpMM using CSC displays a different behaviour, denoting less efficient operation. Nevertheless, throughput on all representations converges for high sparsity inputs, achieving 0.115 (COO), 0.101 (CSC) and 0.113 (CSR) TFLOP/s for ~ 99.90 sparsity. Further increasing sparsity (i.e. processing matrices with less NNZs) after a certain point (around ~ 99.22 sparsity) does not have a very pronounced effect on the achieved throughput.

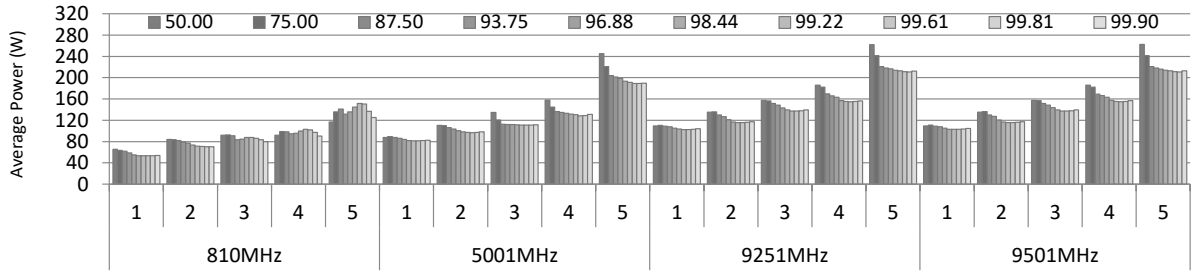
Overall, as is the case with GEMM, using higher core and memory frequencies tends to increase the throughput of SpMM. And, similarly to GEMM, memory clocks can be reduced down to 5001MHz without a very significant impact on throughput. However, in contrast with GEMM, for some sparsity levels (especially between ~ 96.88% and ~ 99.61%), when setting the memory frequency to 810MHz, SpMM is still able to achieve a throughput close to that of using the highest core and memory frequencies.

It is worthwhile to to notice that if using COO or CSR with the memory set to operate at 810MHz, the throughput of SpMM for low sparsity inputs (especially ~ 50%) is significantly impacted by core frequency settings. However, if using the CSC matrix format, sparse matrices with ~ 50% up to ~ 98.44% zeros can be processed with a similar throughput as that of using the highest memory frequency, even if using the highest core frequency. This is indicative that CSC is still working at the throughput, even when pairing such low memory frequency with high core frequencies, while COO and CSR work at the memory latency under the same conditions.

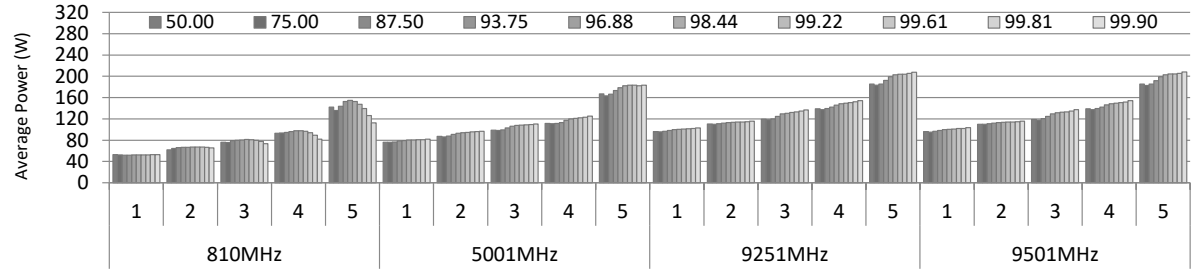
Fig. 38(a), Fig. 38(b) and Fig. 38(c) depict the average power consumption achieved with different core and memory frequencies, for cuSPARSE SpMM (M,N,K set to 16384) runs using the COO, CSC or COO matrix formats, respectively, and considering different sparsity levels.

In consonance with the throughput metric, the use of the COO or CSR formats resulted in a different behaviour in regard to average power consumption than if using the COO format. Notice that the only difference between COO and CSR is that in the case of the later the row indices are represented using the CSR format. The CSC formats differs from those two formats (COO and CSR) due to it being column-major and the column indices using the CSC representation, which can have an impact on the efficiency of an SpMM implementation. As a matter of fact, a given matrix represented as COO has the same layout in memory as the transpose in CSR.

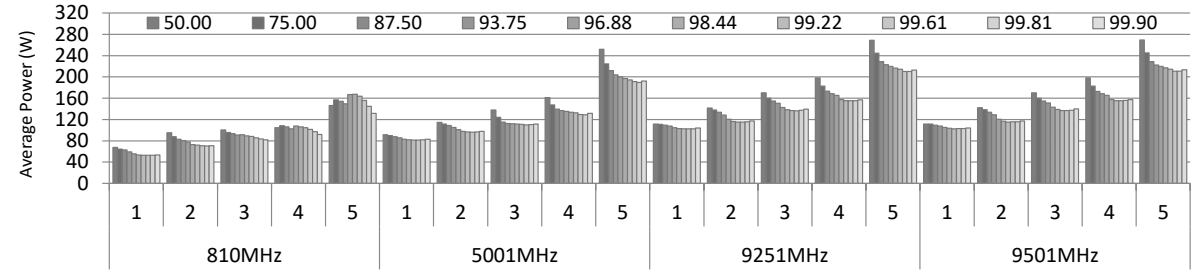
Overall, for COO and CSR, power consumption drops when increasing sparsity, indicating that SpMM is working at memory latency and in agreement to the fact that high sparsity resulted in fewer TFLOP/s having been achieved. For CSC, power consumption increases, indicating that SpMM is using more GPU resources when operating high sparsity matrices. The only exception is when GPU memory frequency is set to its lowest considered value (810MHz) in combination with



(a) *Average power consumption using the COO sparse matrix format.*



(b) *Average power consumption using the CSC sparse matrix format.*



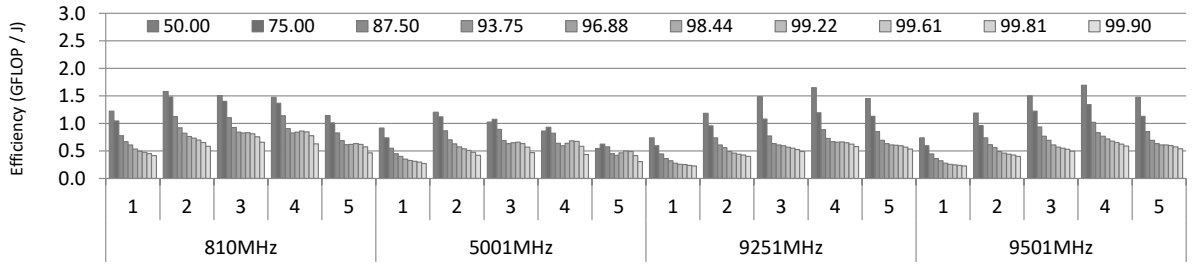
(c) *Average power consumption using the CSR sparse matrix format.*

Figure 38 *Average power consumption of cuSPARSE cuSpMM on CUDA cores with different GPU core (1: 345MHz, 2: 690MHz, 3: 1020MHz, 4: 1365MHz or 5: 1710MHz) and memory (810MHz, 5001MHz, 9251MHz or 9501MHz) frequencies, with input matrices of* 16384×16384 *elements.*

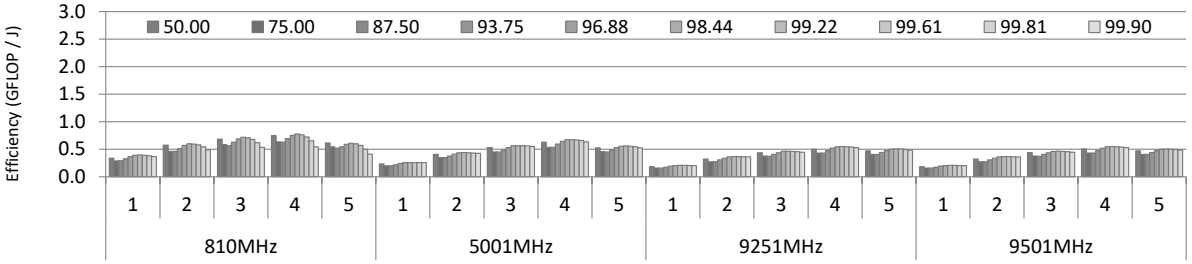
high GPU core frequencies, e.g. GPU core set to 1710MHz. In such scenario, SpMM operations seems to be limited by memory latency for the use of any of these sparse matrix formats.

Fig. 39(a), Fig. 39(b) and Fig. 39(c) show the energy efficiency achieved with different core and memory frequencies, for cuSPARSE SpMM (M,N,K set to 16384) runs using the COO, CSC or COO formats, respectively, taking into consideration different sparsity levels.

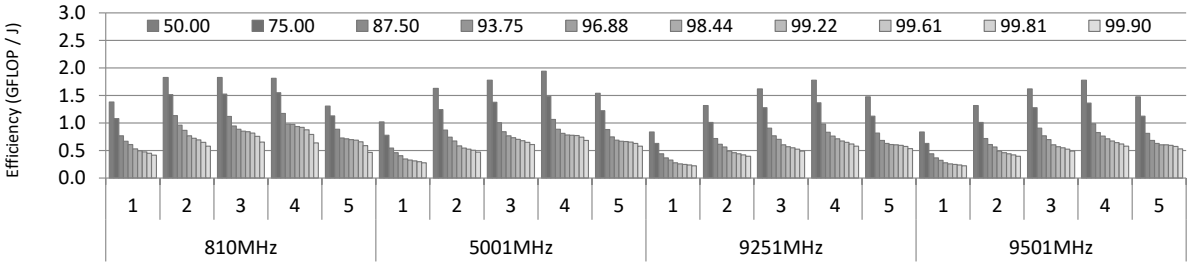
The COO and CSR formats placed higher than COO in terms of energy efficiency. The energy efficiency of SpMM using COO/CSR matrix formats decreased significantly with increases in sparsity. However, SpMM using COO or CSR formats to process high sparsity inputs still converges to higher energy efficiency (0.657 and 0.682 GFLOP/J, respectively) than CSC (0.633). In contrast, the efficiency of SpMM with CSC, is not as affected by the sparsity of the input sparse matrix. Notice that for COO the achieved throughput (see Fig. 37) varies in close to linear proportion to power consumption (see Fig. 38), while this is not the case for COO and CSR. For these two formats, performance drops much faster than power consumption with increases in sparsity. Hence the registered decrease in energy efficiency for high sparsity.



(a) *Energy efficiency using the COO sparse matrix format.*



(b) *Energy efficiency using the CSC sparse matrix format.*



(c) *Energy efficiency using the CSR sparse matrix format.*

Figure 39 *Energy efficiency of cuSPARSE cuSpMM using the COO, CSC and CSR matrix formats with different GPU core (1: 345MHz, 2: 690MHz, 3: 1020MHz, 4: 1365MHz or 5: 1710MHz) and memory (810MHz, 5001MHz, 9251MHz or 9501MHz) frequencies, with input matrices of* 16384×16384 *elements.*

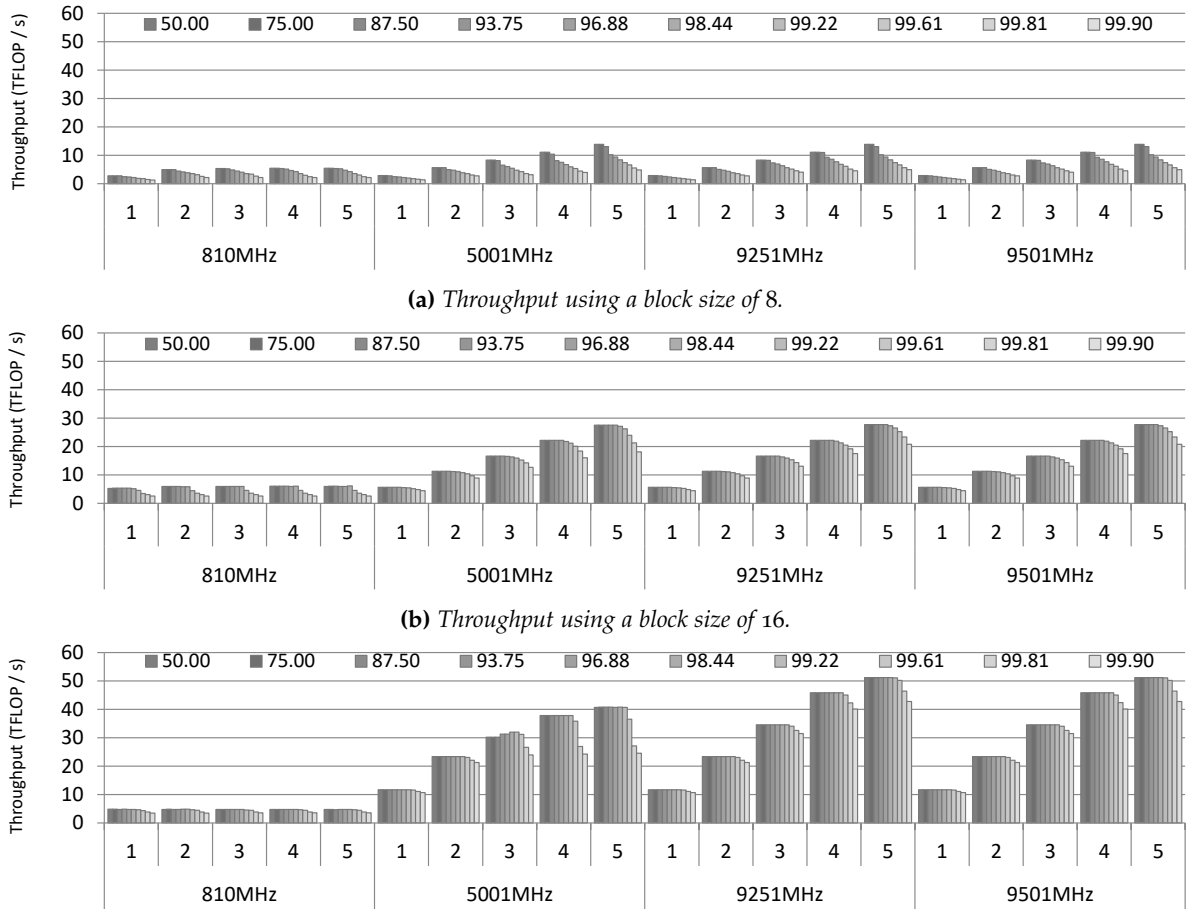
3.1.4 SPMM ON TENSOR CORES

The use of (dense) tensor cores is supported by the cuSPARSE library through the use of the Blocked-Ell format. Given a block size B, a compacted matrix represents only the blocks of $B \times B$ elements from the original matrix that do not have any non-zero element. In addition, an auxiliary matrix stores, for each row of blocks, the column indices of the blocks form the original matrix that are represented in the compacted matrix. This auxiliary matrix, which has $M \times c$ positions, where c is the maximum number of blocks with non-zero elements in the original matrix on any given row, is used to decide which data to load from the dense matrix.

Multiple factors need to be taken into account to maximize the matrix multiplication performance using this matrix format, which are influenced by the particular sparse matrices to process. In order to fully exercise the tensor cores, one must use a sufficiently large block size. However, compared to smaller block sizes, the downside is it becomes more difficult to find in the original input matrix blocks that only have zeros. Notice that the existence of these blocks is what allows a significant portion of floating-point operations related to matrix multiplication to be skipped.

Fig. 40 depicts the throughput achieved with different pairings of core and memory frequencies, for cuSPARSE SpMM (M,N,K set to 16384) runs using the Blocked-ELL format, considering the use of different block sizes (8, 16 or 32) and different sparsity levels. Notice that only the

operations performed to multiply elements from the sparse input and dense input have been considered for the calculation of this metric. Thus, this metric is directly proportional to the ratio of blocks represented in the column index matrix of the Blocked-Ell representation format.



(a) *Throughput using a block size of* 8.

(b) *Throughput using a block size of* 16.

(c) *Throughput using a block size of* 32.

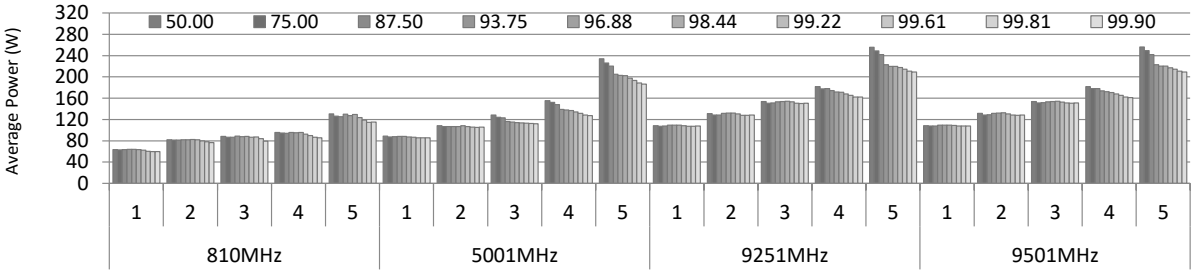
Figure 40 *Throughput using the Blocked-ELL sparse matrix format, setting the block size to* 8, 16 *or* 32, *with different pairings of GPU core (1:* 345*MHz, 2:* 690*MHz, 3:* 1020*MHz, 4:* 1365*MHz or 5:* 1710*MHz) and memory (*810*MHz,* 5001*MHz,* 9251*MHz or* 9501*MHz) frequencies, with matrices of* 16384 × 16384 *elements.*

Overall, the use of the Blocked-Ell format achieves a higher throughput when using the largest considered block size. Using a block size of 32 allowed to achieve a throughput of up to 51.18 TFLOP/s, representing 85.96% of the peak throughput of the GPU with tensor cores (59.54%). Relying on a block size of 8 (16), only up to 13.86 (27.74) TFLOP/s have been achieved.

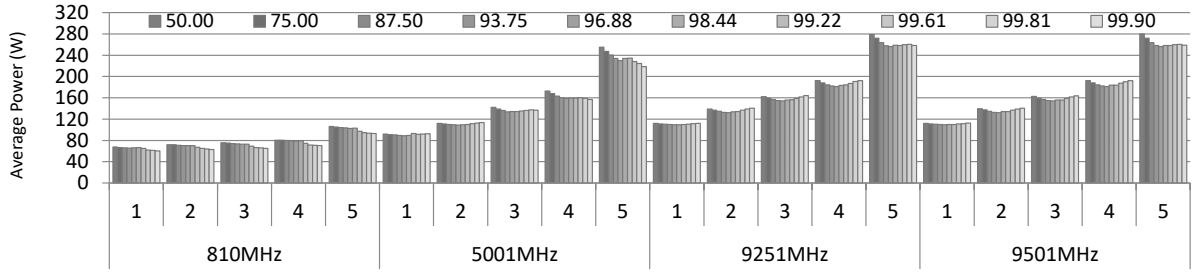
In order to extract the maximum matrix multiplication performance (i.e. the lowest execution time), one might have to use a block size resulting in less throughput (which only considers the computations being actually performed). For all block sizes considered (8, 16 or 32), when processing an input sparse matrix with ~ 50% sparsity, all blocks from the original matrix are represented in the Blocked-Ell compacted matrix. As a result, no operations are skipped in comparison to using GEMM. However, for the two highest levels of sparsity, ~ 99.81% and ~ 99.90, 11.78% and 6.08% of the blocks get to be used for computations if using a block size of 8. If using a block size of 32 (16), for the same sparsity levels, 86.58% (39.44%) and 63.43% (22.22%) of the blocks are mapped to computations. In particular, the use of a block size of 16 represents a

suitable trade-off between throughput and the percentage of blocks with NNZs, as it still results in avoiding a significant amount of operations.

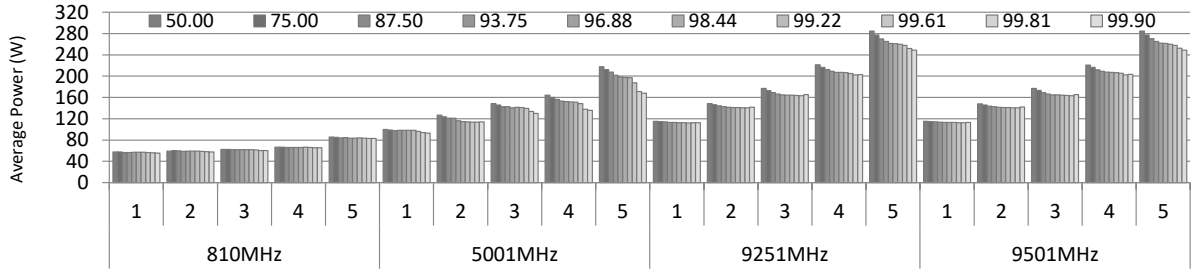
Another interesting thing to extract form the obtained experimental results is that, for any given block size, the achieved throughput when using 810MHz for the memory frequency is being capped at around the same level, for all core frequencies. While the achieved throughput scales very well with core frequency if using a memory frequency of 5001MHz or above, it stays more or less constant if using 810MHz as the memory clock. This behaviour is consistent with that of GEMM running on tensor cores.



(a) *Average power consumption using a block size of* 8.



(b) *Average power consumption using a block size of* 16.

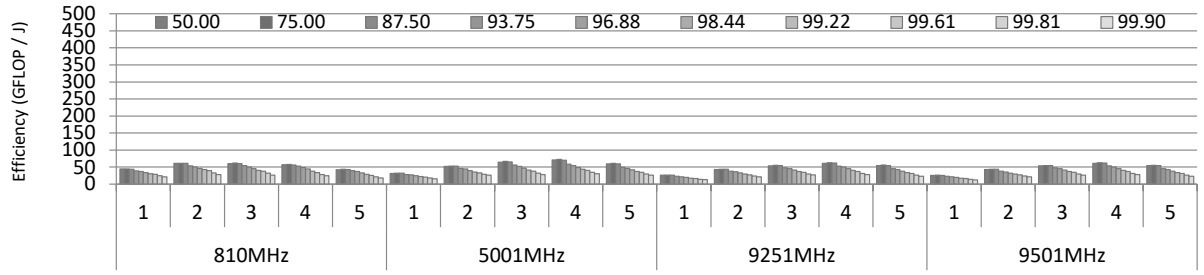


(c) *Average power consumption using a block size of* 32.

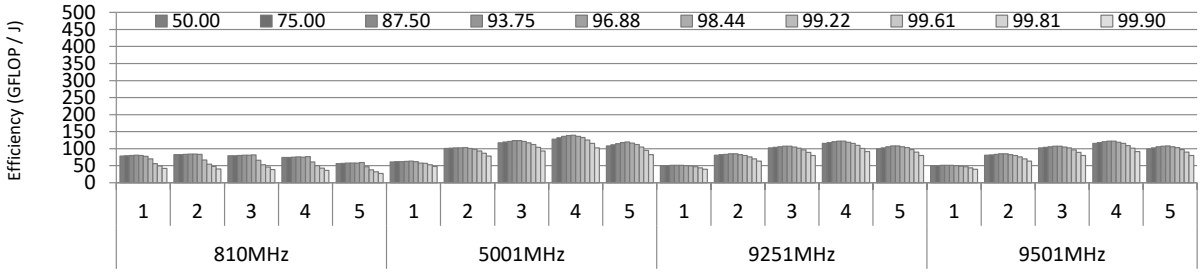
Figure 41 *Average power consumption achieved using the Blocked-ELL matrix format, setting the block size to* 8, 16 *or* 32, *with different pairings of GPU core (1:* 345*MHz, 2:* 690*MHz, 3:* 1020*MHz, 4:* 1365*MHz or 5:* 1710*MHz) and memory (*810*MHz,* 5001*MHz,* 9251*MHz or* 9501*MHz) frequencies for runs using cuSPARSE cuSpMM, with input matrices of* 16384×16384 *elements.*

Overall, the average power consumption is lower for runs with high sparsity inputs. This can be attributed to less data reuse when the ratio of blocks represented in the compressed matrix to the total of blocks in the original matrix is low, making the kernel more memory bound. This results in less efficient of the compute resources, hence the registered drop in power consumption.

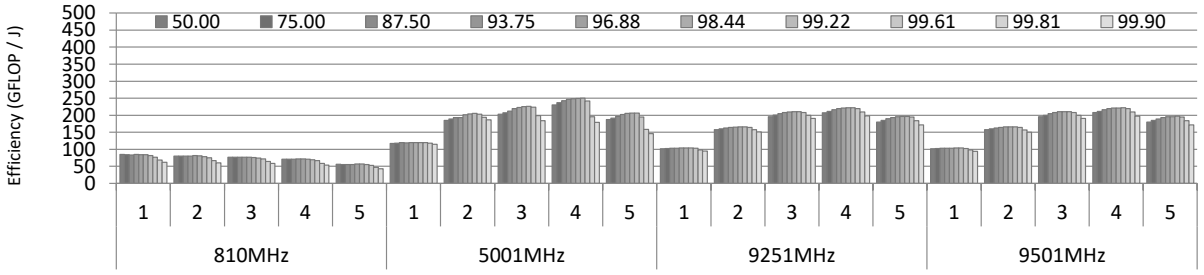
Fig. 42 depicts the energy efficiency achieved relying on different core and memory frequencies, for cuSPARSE SpMM (M,N,K set to 16384) using the Blocked-Ell format, considering the use of different block sizes (8, 16 or 32) and sparsity levels.



(a) *Energy efficiency using a block size of* 8.



(b) *Energy efficiency using a block size of* 16.



(c) *Energy efficiency using a block size of* 32.

Figure 42 *Energy efficiency with Blocked-ELL, using a block size equal to* 8, 16 *or* 32, *with different GPU core (1:* 345*MHz, 2:* 690*MHz, 3:* 1020*MHz, 4:* 1365*MHz or 5:* 1710*MHz) and memory (*810*MHz,* 5001*MHz,* 9251*MHz or* 9501*MHz) frequencies for SpMM runs with input matrices of* 16384×16384 *elements.*

The combination of core and memory frequencies most conducive to high energy-efficiency depends on the block size used. For all sparsities, if using a block size of 8 or 16, setting the core to 1365MHz and memory to 5001MHz results in highest energy-efficiency being achieved. However, if using a block size of 32 to process the two highest sparsity levels considered ($\sim$ 99.81% and $\sim$ 99.90%), setting the memory frequency to 9251MHz achieves higher energy efficiency (241.83 and 311.61) than if setting it to 5001MHz (225.57 and 282.26).

Overall, the most energy efficient Blocked-ELL runs have been achieved setting the block size to 32. This is a direct consequence of how throughput and power consumption are affected by the block size. A significantly higher throughput is achieved with a block size of 32 in relation to using a block size of 16, and especially 8. Since the average power consumption does not differ significantly when using different considered block sizes, this results in the block size achieving higher throughput also resulting in more energy-efficient SpMM runs.

3.1.5 SPMM ON SPARSE TENSOR CORES

In relation to Volta and Turing, which respectively have the first and second generation dedicated matrix acceleration hardware, the tensor cores in the Ampere GPU architecture add the capability of natively performing hardware accelerated SpMM. In order to use the sparse tensor cores on the

Ampere GPU architecture, one is required to use of a specialized representation (2:4 sparse matrix representation). In this representation at least two data values per each consecutive four data values have to be 0. The sparse tensor cores are fed tiles of matrix values, that are half the size in comparison to their representation in the original matrix and additional matrices identifying (with 2-bit indices) which 2 particular elements (out of 4) are to be used for computations. As a result, less multiply-add operations are actually performed in comparison to when performing dense-dense matrix multiplication, allowing to achieve higher matrix multiplication performance.

As part of the selection of an efficient sparse tensor-core GPU kernel, we explored different thread-block tiles, warp tiles and MMA operation tiles, as well as different alternative matrix layout configurations. This exploration took into account the different limitations of sparse/dense tensor cores. In comparison to a dense tensor core, a sparse tensor core operates on a larger tile in regard to the K dimension. On CUTLASS SpMM, the combination of parameters resulting in highest throughput includes the use of a thread-block tile with $128 \times 128 \times 64$ (M,N,K), a warp tile of $64 \times 64 \times 64$ and an MMA operation tile of $16 \times 8 \times 32$. In addition to tensor-core accelerated SpMM, since CUTLASS also supports GEMM on tensor cores, we opted to also test it in order to have an additional baseline. For the experiments with CUTLASS GEMM, the only modification in relation to the parameterization used for tensor-accelerated SpMM resides in the tile shapes used in the K dimension (half of that used for sparse tensor cores), making the number of MMA operations per warp and per thread-blocks the same as for the CUTLASS SpMM runs.

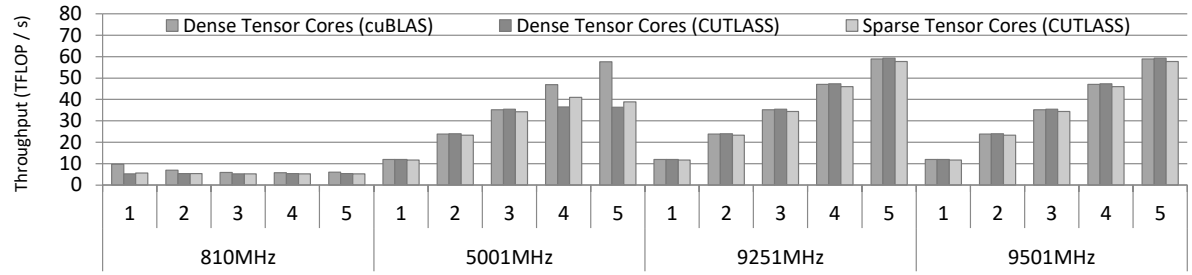
Fig 43 depicts the throughput, average power consumption and energy efficiency of GEMM using dense tensor core acceleration through cuBLAS and CUTLASS and that of SpMM using sparse tensor core acceleration through CUTLASS. Sparse tensor cores are rated as having double the peak throughput of dense tensor cores. However, as for the other formats and methods evaluated as part of this study, we are counting only the operations performed, i.e. skipped operations (half of those in relation to GEMM) are not counted for the calculation of the throughput metric.

GEMM on CUTLASS achieved similar throughput (58.96 TFLOP/s) to that of GEMM on cuBLAS (59.32 TFLOP/s), i.e. close to the peak throughput of the GPU for the operating precision used. The only exception is when one combines a high core frequency with a low memory frequency. For example, if using CUTLASS GEMM, it can be observed that when using a memory frequency of 5001MHz, there is no improvement in throughput from an increase in core frequency from 1020MHz to 1710MHz. As a result, if using cuBLAS, higher throughput is achieved using that memory frequency paired with core frequencies above 1020MHz.

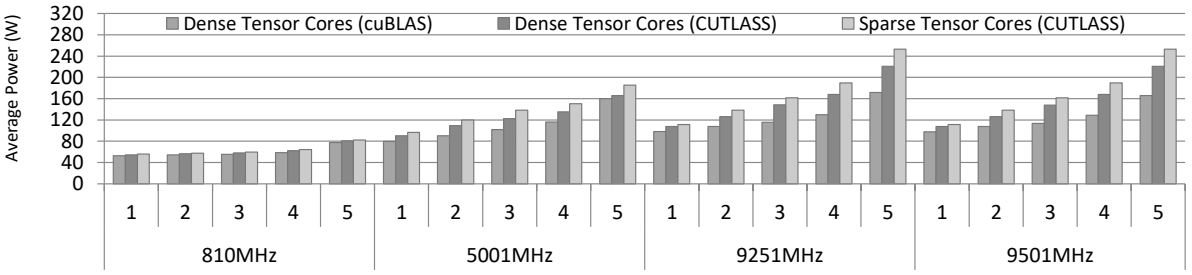
SpMM on CUTLASS achieved a similar throughput to that of GEMM across a wide range of core and memory frequencies. The exception is also for a 5001MHz memory frequency, for which case using a core frequency above 1365MHz does not result in higher throughput. For this particular case, SpMM operations using sparse tensor cores achieves a higher throughput than that of GEMM operations on CUTLASS using dense tensor cores.

Overall, GEMM on CUTLASS resulted in significantly higher power consumption than on cuBLAS, with SpMM on CUTLASS being even more demanding in that regard. This is especially the case when high clock frequencies are paired with high memory frequencies, allowing the hardware to be exercised to its potential. In the case of the SpMM, its extra power consumption in regard to GEMM on the same library can be attributed to sparse tensor cores using additional hardware to process the metadata matrix that is part of the 2:4 fine-grained sparsity format.

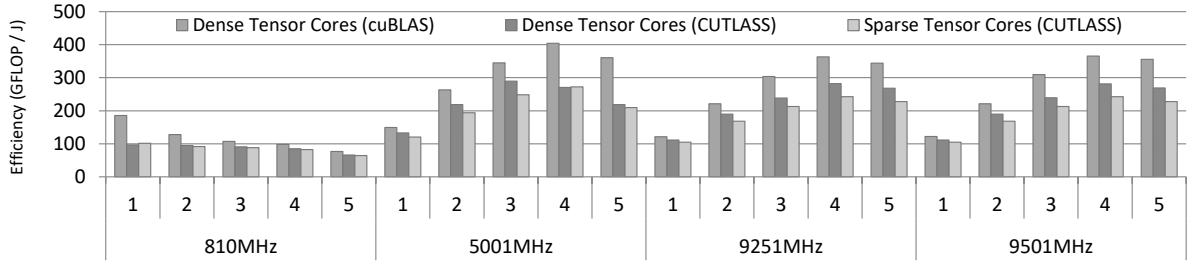
As a direct result of higher power consumption, dense tensor core acceleration on CUTLASS is less energy-efficient than on cuBLAS. Highest energy efficiency has been achieved on GEMM, both on cuBLAS (404.31 GFLOP/J) and CUTLASS (289.38 GFLOP/J), combining a memory frequency of 5001MHz with a core frequency of 1365MHz. On the sole basis of throughput in terms of amount of floating point operations processed per unit of time, SpMM on CUTLASS proved to be



(a) *Throughput.*



(b) *Average power consumption.*



(c) *Energy efficiency.*

Figure 43 *Throughput, average power consumption and energy efficiency of SpMM using sparse tensor cores (CUTLASS) and that of GEMM using dense tensor cores (cuBLAS or CUTLASS), with different GPU core (1: 345MHz, 2: 690MHz, 3: 1020MHz, 4: 1365MHz or 5: 1710MHz) and memory (810MHz, 5001MHz, 9251MHz or 9501MHz) frequencies, with input matrices of* 16384×16384 *elements.*

even less efficient in regard to throughput, which is a direct result of higher power consumption having been achieved. However, due to the skipped operations, processing a matrix that can be represented in the 2:4 matrix format, is still faster in SpMM on CUTLASS than in GEMM on CUTLASS (or cuBLAS), resulting in less energy use.

A significant challenge for using the sparse tensor cores is related to the requirements of the first input matrix (the sparse matrix), i.e. the fact that the used sparse matrix format (2:4 fine-grained sparsity) requires at least two data values per each consecutive four data values stored in row-major memory layout to be zero. One way to address this could be to add dummy columns (where needed) with only zeros. Using such an approach (possibly coupled with reordering in order to minimize the amount of dummy values inserted), at most the input matrices, i.e. the sparse input (before conversion to the 2:4 format) and the dense input, would need to be modified (adding columns of zeros) until they are double in size in relation to the original inputs.

In order to demonstrate what happens in such scenario, multiple runs of an SpMM kernel implemented using CUTLASS to make use of the sparse tensor cores were profiled, increasing K (from M,N,K, the dimensions of the matrix multiplication operation) from 16384 up to 32768 (representing the worst case), with increments of 1024. Fig. 44 represents the speedup achieved

with SpMM on CUTLASS, in relation to GEMM performed always with 16384 × 16384 matrices, on either cuBLAS or CUTLASS.

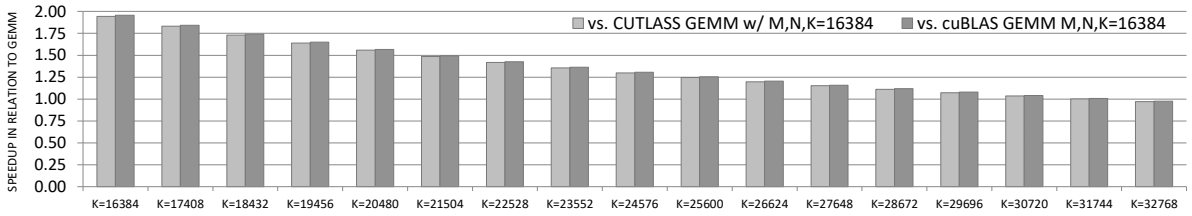


Figure 44 *Speedup of SpMM accelerated with sparse tensor cores on CUTLASS in comparison to GEMM accelerated with dense tensor cores on CUTLASS and on cuBLAS.*

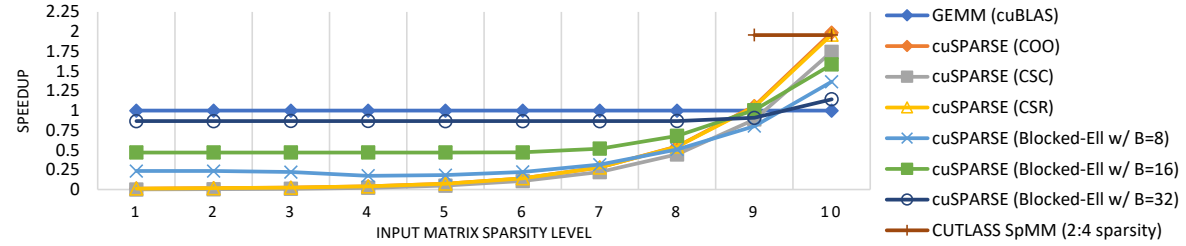
In comparison to processing matrices with 16384 × 16384 elements using the dense tensor cores, the use of the sparse tensor cores still resulted in a speedup > 1 for increments in K up to 31744, i.e. to close to double in comparison to the initial input matrices. This suggests that it is worthwhile to explore methods to enable the use of sparse tensor cores for matrices that otherwise can not be directly used due to not adhering to the 2:4 sparsity format requirements.

3.1.6 Comparison between the evaluated SpMM approaches

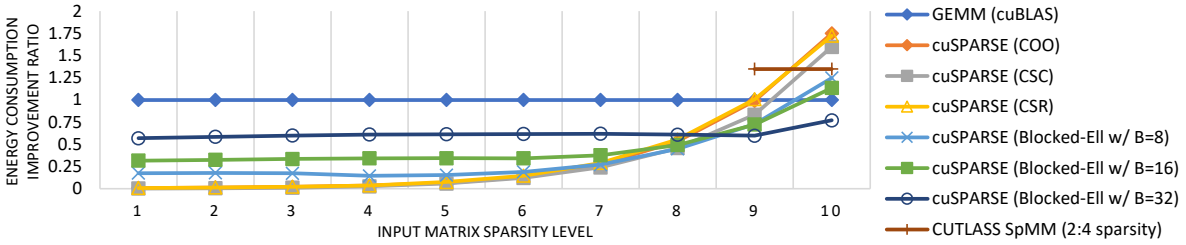
Depending on the inputs to process, different methods should be considered to maximize performance and/or minimize energy consumption. For low sparsity matrices, the use of highly efficient GEMM kernels (e.g. those on cuBLAS) can still be the preferable route. However, the experimental results clearly show that it can payoff to explore the use of SpMM if there is enough sparsity in at least one of the inputs of matrix multiplication. Individually tuning the core and memory frequencies on a modern GPU device had a profound effect on energy-efficiency on the different considered matrix multiplication methods. Thus, the achieved experimental results demonstrate that tailoring core/memory frequencies to the needs of each method it is a requirement to minimize (maximize) energy consumption (efficiency).

Fig 45(a) and Fig 45(b) represent the speedups and energy consumption improvements achieved for processing matrices with 16384 × 16384 elements, with the different SpMM methods considered as part of this study — SpMM on cuSPARSE using the COO, CSC, CSR or Blocked-ELL sparse matrix formats and SpMM on CUTLASS accelerated with sparse tensor cores relying on the 2:4 fine-grained format. The reported speedups are calculated in relation to mixed-precision GEMM on cuBLAS using the dense tensor cores, which proved to always be preferable in all fronts (throughput, power and energy efficiency) both to GEMM relying on CUDA cores on either considered precision (32-bit uniform or 16-bit/32-bit mixed-precision) and to GEMM relying dense tensor cores on CUTLASS. The energy consumption improvement metric is calculated dividing the energy consumption of the SpMM methods by that resulting from executing GEMM.

Overall, SpMM runs using the COO CSC or CSR formats significant increase in performance and energy efficiency (translating to energy consumption reductions) in relation to GEMM with each halving of the density of the input sparse matrix. In fact, after ~ 93.75% sparsity the registered speedups in relation to GEMM on cuBLAS between matrices with consecutive levels of sparsity is always above 1.61×. Considering the exploitation of the DVFS capabilities to maximize the energy-efficiency of each of the matrix multiplication methods, after the same level of sparsity SpMM using these sparse formats always improved efficiency by at least 1.62× with each sparsity increase. This results from the fact that there are half the amount of floating-point computations to be performed, since only those represented in the sparse matrix (i.e. the non-zero elements in



(a) *Speedup achived considering the use of the highest core (1710MHz) and memory (9501MHz) frequencies.*



(b) *Energy consumption improvement ratio with the use of core and memory frequencies specialized to each method.*

Figure 45 *SpMM in comparison to cuBLAS GEMM, with input matrices of* 16384 × 16384 *elements, considering different levels of sparsity. The matrix sparsities are the following: 1:* ~ 50.00%, *2:* ~ 75.00%, *3:* ~ 87.50%, *4:* ~ 93.75%, *5:* ~ 96.88%, *6:* ~ 98.44%, *7:* ~ 99.22%, *8:* ~ 99.61%, *9:* ~ 99.81%, *10:* ~ 99.90%.

the first matrix) are to be multiplied by values in the dense matrix.

Sparsity levels of ~ 99.61% or above resulted in SpMM achieving speedups > 1 using any of the sparse matrix formats that are processed using CUDA cores on cuSPARSE (COO, CSR or CSC) in relation to execution of a cuBLAS GEMM using CUDA cores. To achieve performance improvements with SpMM using these formats in relation to that of cuBLAS GEMM on tensor cores, the sparsity of the sparse input matrix must be higher. To surpass the matrix multiplication performance of cuBLAS GEMM on tensor cores, a sparsity of ~ 99.61% or above is required, having speedups up to 1.99× (using the COO format) been achieved when processing matrices with ~ 99.90% sparsity. Using the individually the core and memory frequency individually found to maximize energy-efficiency for each of the matrix multiplication methods resulted in energy consumption improving by a factor of 1.75× in relation to GEMM.

The use of (dense) tensor cores has been attained on cuSPARSE using the Blocked-ELL matrix format. In order to achieve speedups > 1 in relation to cuBLAS using tensor cores, SpMM using Blocked-Ell required the use of input matrices with sparsity of ~ 99.81 or above. This has been achieved using a block size of 16, which achieves a speedup of 1.58× over GEMM for processing the highest sparsity input. When striving to minimize energy consumption, the use of a block size of 16 resulted in energy consumption improving by a factor of 1.13× over GEMM. For matrices with sparsity between ~ 50% and ~ 99.61%, the use of a block size of 32 resulted in higher matrix multiplication performance and improved energy consumption.

Compared to using COO, CSC or CSR, which only use the CUDA cores of the Ampere microarchitecture, the performance and energy-efficiency of Blocked-ELL format is significantly higher for low sparsity matrices. This can be explained by the fact that the former are not very efficient at processing matrices with a high percentage of NNZs and Blocked-ELL makes use of tensor cores. However, for such sparsity levels, using GEMM on tensor cores still resulted in faster execution, being the improvement in regard to energy consumption even more pronounced.

When processing low sparsity matrices using the Blocked-ELL format, it was not possible to simultaneously achieve a significant reduction in the percentage of blocks with non-zero elements

and the use of a block size large enough to be conducive to high throughput. This can be partially attributed to the fact that the matrices used in the experiments have been generated with a random uniform distribution. As a result, the NNZs are spread thought the whole matrix, making the existence of clusters of NNZs rare, which results in fewer blocks free of NNZs, which are the ones that are responsible for accelerating SpMM with the Block-ELL format.

SpMM on CUTLASS using sparse tensor cores allowed to achieve 1.96× higher performance than GEMM on cuBLAS using (dense) tensor cores, which represents close to double the speedup achieved with SpMM on cuSPARSE compared to the same baseline (1.06×). The achieved improvement in regard to energy consumption, while still significant, is not as high (1.35×) due to the additional power consumption registered when using the sparse tensor cores. For ~ 99.81 sparsity, the use of sparse tensor cores allowed to achieve higher performance and improved energy consumption in relation to any other evaluated matrix multiplication method. However, while still competitive in regard to performance in relation to SpMM on cuSPARSE at the highest level of sparsity considered (~ 99.90), the later is expected to far surpass sparse tensor core accelerated SpMM on CUTLASS at processing matrices with higher levels of sparsity. This is due to the fact that, independently of the level of sparsity, considering the use of the sparse tensor cores results at best in skipping half of the floating point computations.

A particularity related to sparse tensor cores is that its direct application requires the sparsity pattern of the sparse input to comply with a specialized format (2:4 fine-grained sparsity). As a result, the kernel using sparse tensor cores was not able to process the input matrices with up to 99.22% sparsity that have been used across the different considered SpMM methods. Ongoing work includes the exploration of efficient approaches to modify matrices with different levels of sparsity and different sparsity patterns, so that SpMM can be accelerated with sparse tensor cores for processing matrices that otherwise do not comply with the required format.

3.2 IDENTIFYING THE TENSOR-TIMES-MATRIX UPPER-BOUNDS ON CPU AND GPU DEVICES

A Tensor Contraction (TC)[14] is an important tensor operation, which is analog to GEMM in the multidimensional realm of tensors. Two tensors are multiplied across their matching modes resulting in a tensor with the remaining modes. For example, a TC $C = A \cdot B$ with two fourth-order tensors $A \in \mathbb{R}^{I_0 \times I_1 \times I_2 \times I_3}$ and $B \in \mathbb{R}^{I_2 \times I_3 \times I_4 \times I_5}$, results in a tensor $C \in \mathbb{R}^{I_0 \times I_1 \times I_4 \times I_5}$. I_0 to I_3 are the dimensions of tensor A in modes zero to three, while I_2 to I_5 are the dimensions of tensor B in modes zero to three.

There are a few TCs that are notable and are often even treated independently of regular contractions. One of these is GEMM, which is a contraction between two second-order tensors. Others are Tensor Times Vector (TTV), which is a contraction between a Kth-order tensor and a first-order one, and TTM,[15] which is a contraction between a Kth-order tensor and a second-order

[14] Thomas Herault et al. "Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure". *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021, pp. 537–546. DOI: 10.1109/IPDPS49936.2021.00062; Jinsung Kim et al. "Optimizing Tensor Contractions in CCSD(T) for Efficient Execution on GPUs". *Proceedings of the 2018 International Conference on Supercomputing*. 2018, pp. 96–106. DOI: 10.1145/3205289.3205296; Jiawen Liu et al. "Athena: High-Performance Sparse Tensor Contraction Sequence on Heterogeneous Memory". *Proceedings of the ACM International Conference on Supercomputing*. 2021, pp. 190–202. DOI: 10.1145/3447818.3460355; Jiawen Liu et al. "Sparta: High-Performance, Element-Wise Sparse Tensor Contraction on Heterogeneous Memory". *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2021, pp. 318–333. DOI: 10.1145/3437801.3441581; Ryan Levy, Edgar Solomonik, and Bryan K. Clark. "Distributed-Memory DMRG via Sparse and Dense Parallel Tensor Contractions". *CoRR* abs/2007.05540 (2020).

[15] Jiajia Li et al. "Optimizing Sparse Tensor Times Matrix on Multi-core and Many-Core Architectures". *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. 2016, pp. 26–33. DOI: 10.1109/IA3.2016.010;

one.

For TTV, all fibers of the Kth-order tensor are multiplied with the vector resulting in a tensor of order $K-1$. For TTM, all fibers are multiplied with each column of the matrix. Since a fiber is a vector, this operation consists of several vector-matrix dot products, which generate new vectors with length equal to the number of columns of the matrix. Therefore, the resulting fiber length becomes equal to the number of columns in the matrix.

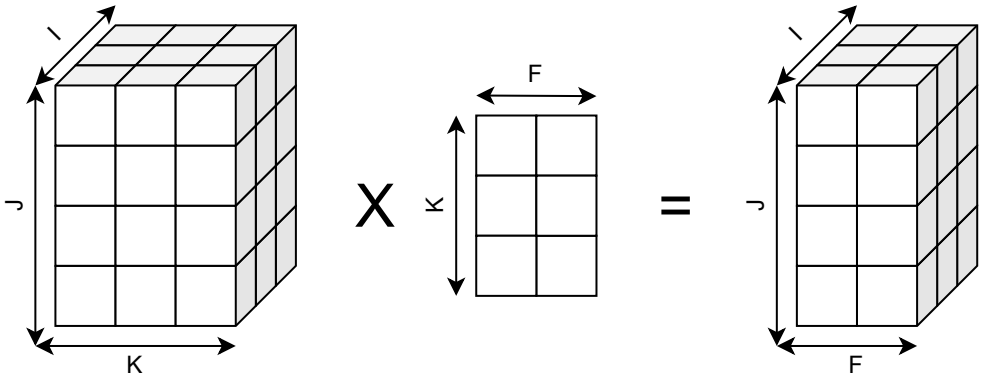


Figure 46 *Example of a dense TTM with a third-order tensor*

Figure 46 provides an example of a dense TTM with a third-order tensor. In the example, the tensor has $I \times J$ fibers of length K. Each of these fibers multiplies with each of the F columns of the matrix, resulting in a dense tensor with $I \times J$ fibers of length F. For sparse TTM the rationale is similar to the one presented in Figure 46, however only the fibers with one or more nonzero elements are computed. Also, the output of the operation is no longer a sparse tensor but instead a semi-sparse tensor meaning that not all of the output tensor's fibers are sparse. This happens because all fibers that have at least one nonzero element after the dot product with a dense column of the matrix generate a nonzero element for the output tensor. Since all columns of the matrix are dense, all fibers that have at least one nonzero element generate a dense fiber for the output tensor.[16]

There are several formats to store and perform computations over sparse tensors. One of the most used for HPC applications is the Compressed Sparse Fiber (CSF)[17] format. The general idea is that it implements a tree like structure, where each mode is a level and paths from root to leaf encode a nonzero coordinate.

Given the importance and the previously elaborated characteristics of TTM, the purpose of this study is to derive a set of data-parallel approaches for sparse TTM and analyse their efficiency in the state-of-the-art computing platforms, with CSF as the storage format for the tensor. The analysis consists of measuring the AI and performance of the proposed TTM approaches by relying on several data-sets, real and synthetic, on both the Central Processing Unit (CPU) and GPU architectures. With this analysis and with resort to the CARM,[18] it is expected to pinpoint the potential execution bottlenecks and uncover the performance and utilization limits when

Yuchen Ma et al. "Optimizing Sparse Tensor Times Matrix on GPUs". *J. Parallel Distrib. Comput.* 129.C (2019), pp. 99–109. DOI: 10.1016/j.jpdc.2018.07.018.

[16] Ma et al., "Optimizing Sparse Tensor Times Matrix on GPUs".

[17] Shaden Smith and George Karypis. "Tensor-Matrix Products with a Compressed Sparse Tensor". *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. 2015. DOI: 10.1145/2833179.2833183.

[18] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. "Cache-aware Roofline model: Upgrading the loft". *IEEE Computer Architecture Letters* 13.1 (2013), pp. 21–24.

performing TTM on different architectures.

3.2.1 Data-parallel tensor-times-matrix (TTM) processing

The development of data-parallel TTMs approaches was done in Intel's OneAPI DPC++, which is one of the most widely used Khronos SYCL implementations. SYCL provides a unified model, where developers program at a higher level than the native acceleration Application Programming Interface (API), but always have access to lower-level code that allows users to target any accelerator without having to change their source code. Besides its portability, when compared with other API such as CUDA, SYCL has proven to provide comparable performance.[19] Therefore it can be considered a future standard in heterogeneous programming, thus selected for the developments conducted in this work.

Due to their higher order, sparse tensors (even more than sparse matrices) are very prone to variability both in their shape and non-zero element distribution. Therefore, creating one kernel that is fully optimized for all scenarios is not a trivial task. As such, in order to minimize this problem, two different versions of data-parallel TTM were created, tested and compared.

Kernel V1: Element-centric TTM approach For TTM with a sparse tensor stored in CSF format, the first approach is to process the data in such a way that each thread computes one element of the output. Therefore, each thread requires access to one fiber and to one column of the matrix.

```
// FbrCnt * ColCnt threads -> one for each element of the output
cl::sycl::range<2> globalSize(fbrCnt, colCnt);
cl::sycl::range<2> localSize(1, colCnt);
cl::sycl::nd_range<2> numItems(globalSize, localSize);

cl::sycl::event e {
    q.submit([&](cl::sycl::handler &h) {
        h.parallel_for(numItems, [=](cl::sycl::nd_item<2> item) {
            const auto fbr { item.get_global_id(0) };
            const auto col { item.get_local_id(1) };
            auto tmp { 0.0f };

            // Load fiber boundaries: accFbrPtr[fbr] and accFbrPtr[fbr+1]
            for (auto ele { accFbrPtr[fbr] }; ele < accFbrPtr[fbr+1]; ++ele) {
                // Load in-fiber index and value
                const auto k { accKIdx[ele] - 1 };
                const auto val { accValues[ele] };

                // Load element of the column
                // Compute product and accumulate
                tmp += val * accMatrix[k * colCnt + col];
            }

            // Store fiber-column dot product to global memory
            accOutput[fbr * colCnt + col] = tmp;
        });
    })
};
```

Listing 1: TTM Kernel v1

[19] Goutham Kalikrishna Reddy Kuncham, Rahul Vaidya, and Mahesh Barve. "Performance Study of GPU applications using SYCL and CUDA on Tesla V100 GPU". *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 2021, pp. 1–7. DOI: 10.1109/HPEC49654.2021.9622813.

In kernel 1 (see Listing 1), threads are created in the same number as output elements (line 2) with each of them computing the dot-product between their assigned fiber and column. Therefore, each thread starts by loading its fiber boundaries (line 14) and then for all non-zero elements in that fiber loads both their in-fiber index as well as their value (lines 16-17). Hence, from the tensor, there are two loads for boundaries plus two more loads for each non-zero element in the fiber. From the matrix, for each non-zero element in the fiber there is one more load, to fetch the corresponding element in the column (line 21). As such, the total amount of loads from memory can be expressed as $2 + 2\times\text{FbrLen} + \text{FbrLen}$, where FbrLen denotes the number of non-zero elements in the fiber. Since each thread computes the dot-product between a fiber and a column, the number of operations performed is one multiply and one addition per non-zero element in the fiber (line 21), i.e., the total amount of Floating-Point Operations (FLOPs) is equal to $2\times\text{FbrLen}$. This operation generates one element of the output, which is subsequently stored in the output fiber, thus contributing to the only one store operation performed (line 25). Assuming all data types are 4-byte wide, the AI can be expressed as follows:

$$\text{AI}_{v1} = \frac{1}{4} \times \frac{2 \times \text{FbrLen}}{2 + 2 \times \text{FbrLen} + \text{FbrLen} + 1} = \frac{1}{2} \times \frac{\text{FbrLen}}{3 + 3 \times \text{FbrLen}} = \frac{1}{6} \times \frac{\text{FbrLen}}{\text{FbrLen} + 1} \tag{6}$$

According to the expression derived in Equation 6, the AI of each thread may be different depending on the amount of non-zero elements in the fibers that are assigned to the thread for processing. However, these AIs will always range between a minimum and a maximum value, which can be calculated. The minimum AI can be achieved when the fiber has the least possible number of non-zero elements, which is one. Thus, the minimum AI can be expressed as follows:

$$\min(\text{AI}_{v1}) = \min\left(\frac{1}{6} \times \frac{\text{FbrLen}}{\text{FbrLen} + 1}\right) = \frac{1}{6} \times \frac{1}{2} = \frac{1}{12} \tag{7}$$

The maximum AI, on the other hand, is achieved when the number of non-zero elements in the fiber is large enough such that $\frac{\text{FbrLen}}{\text{FbrLen}+1} \approx 1$. As such, the maximum AI can be expressed as:

$$\max(\text{AI}_{v1}) = \max\left(\frac{1}{6} \times \frac{\text{FbrLen}}{\text{FbrLen} + 1}\right) \approx \frac{1}{6} \times 1 = \frac{1}{6} \tag{8}$$

Kernel V2: Fiber-centric TTM approach Another approach to efficiently extract data-parallelism in TTM processing is to assign each thread to compute an entire fiber of the output (instead of a single element). In this approach, each thread still requires access to one fiber, but now it also requires access to the whole matrix instead of just a column (as previously elaborated in Kernel 1 with element-centric TTM processing).

```
1  // FbrCnt threads -> one for each fiber of the output
2  cl::sycl::range<1> globalSize(fbrCnt);
3  cl::sycl::range<1> localSize(wgSize);
4  cl::sycl::nd_range<1> numItems(globalSize, localSize);
5
6  cl::sycl::event e {
7      q.submit([&](cl::sycl::handler &h) {
8          h.parallel_for(numItems, [=](cl::sycl::nd_item<1> item) {
9              const auto fbr { item.get_global_id(0) };
10             float tmp[colCnt];
11
12             for (auto col { 0 }; col < colCnt; ++col) {
```

```
13                  tmp[col] = 0.0f;
14              }
15
16              // Load fiber boundaries: accFbrPtr[fbr] and accFbrPtr[fbr+1]
17              for (auto ele { accFbrPtr[fbr] }; ele < accFbrPtr[fbr + 1]; ++ele) {
18                  // Load in-fiber index and value
19                  const auto k { accKIdx[ele] - 1 };
20                  const auto val { accValues[ele] };
21
22                  // Load corresponding row of the matrix
23                  // Compute product and accumulate
24                  for (auto col { 0 }; col < colCnt; ++col) {
25                      tmp[col] += val * accMatrix[k * colCnt + col];
26                  }
27              }
28
29              // Store output fiber to to global memory
30              for (auto col { 0 }; col < colCnt; ++col) {
31                  accOutput[fbr * colCnt + col] = tmp[col];
32              }
33          });
34      })
35  };
```

Listing 2: TTM Kernel v2

In kernel 2, the number of threads created is the same as the number of output fibers (line 2), where each thread is responsible for computing the dot-products of the assigned fiber against all columns of the matrix. Therefore each thread also starts by loading its fiber boundaries (line 17) and then for all non-zero elements in that fiber it also loads both their in-fiber index as well as their value (lines 19-20). As such, from the tensor, there are two loads (for boundaries) plus two more loads for each non-zero element in the fiber. From the matrix, there are as many loads as columns in the matrix for each non-zero element of the fiber (lines 24-26). As a result, the total amount of loads is equal to $2 + 2\times\text{FbrLen} + \text{FbrLen}\times\text{ColCnt}$, where ColCnt denotes the total amount of columns in the matrix. Since each thread computes the dot-product between a fiber and all columns of the matrix (lines 24-26), the number of operations performed is one multiply and one addition per column per non-zero element in the fiber, i.e., the total amount of FLOPs is equal to $2\times\text{FbrLen}\times\text{ColCnt}$. Since each thread in Kernel V2 generates one fiber of the output, and the output fibers have as many elements as there are columns in matrix, then one store per column is required (lines 30-32). This brings the total amount of stores to be equal to ColCnt. Assuming all data types are 4-byte wide the AI can be expressed as follows:

$$\begin{aligned} \text{AI}_{v2} &= \frac{1}{4} \times \frac{2 \times \text{FbrLen} \times \text{ColCnt}}{2 + 2 \times \text{FbrLen} + \text{FbrLen} \times \text{ColCnt} + \text{ColCnt}} = \\ &= \frac{1}{2} \times \frac{\text{FbrLen} \times \text{ColCnt}}{2 \times (\text{FbrLen} + 1) + \text{ColCnt} \times (\text{FbrLen} + 1)} = \\ &= \frac{1}{2} \times \frac{\text{FbrLen}}{\text{FbrLen} + 1} \times \frac{\text{ColCnt}}{\text{ColCnt} + 2} \end{aligned} \tag{9}$$

According to the expression derived in 9, the AI varies depending on the number of non-zero elements in the fiber as well as on the number of columns in the matrix. Again, these AIs will also

always range between a minimum and a maximum value, which can be calculated in a similar manner to the one previously adopted when analyzing the AI ranges for kernel V1. As before, the minimum AI can be achieved when the fiber has the least non-zero elements (which is one), and the matrix has the least number of columns (which is also one and occurs when the matrix is a vector). As such, the minimum AI for the fiber-centric TTM approach can be expressed as:

$$\min(AI_{v2}) = \min\left(\frac{1}{2} \times \frac{\text{FbrLen}}{\text{FbrLen}+1} \times \frac{\text{ColCnt}}{\text{ColCnt}+2}\right) = \frac{1}{2} \times \frac{1}{2} \times \frac{1}{3} = \frac{1}{12} \tag{10}$$

The maximum AI, on the other hand, is achieved when the number of non-zero elements in the fiber is large enough such that $\frac{\text{FbrLen}}{\text{FbrLen}+1} \approx 1$ and the number of columns on the matrix is large enough such that $\frac{\text{ColCnt}}{\text{ColCnt}+2} \approx 1$. Correspondingly, the maximum AI for the fiber-centric TTM approach can be expressed as follows:

$$\max(AI_{v2}) = \max\left(\frac{1}{2} \times \frac{\text{FbrLen}}{\text{FbrLen}+1} \times \frac{\text{ColCnt}}{\text{ColCnt}+2}\right) \approx \frac{1}{2} \times 1 \times 1 = \frac{1}{2} \tag{11}$$

Element-centric TTM vs. Fiber-centric TTM We focus herein on providing the comparison between the Element-centric (Kernel V1) and Fiber-centric (Kernel V2) TTM data-parallel approaches by analyzing their ranges of attainable AI. Kernels 1 and 2 have the same minimum AI (1/12), however kernel 2 has a 3× higher maximum AI (1/2 vs. 1/6). It is important to reinforce that all calculations involving AI were done under the assumption that all fibers with no non-zero elements are not introduced in the computation. If such fibers were to be introduced, the maximum AI would remain unchanged, but the minimum AI would drop down to zero.

3.2.2 CPU/GPU TTM performance upper-bounds with synthetic sparse tensors

A direct performance comparison between the two previously elaborated data-parallel TTM approaches also depends on other factors (other than the kernels' AI ranges), such as the processing capabilities of the device in which the TTM computation are performed (e.g., multi-core CPU or GPU), as well as on the characteristics of the sparse tensor under evaluation. Therefore, the following study aims at describing the behaviour of kernels 1 and 2, as well as uncovering their performance upper-bounds, on two state-of-the-art compute devices, i.e., Intel Core i9-11900KB (CPU) and on Intel 11th Gen UHD Graphics, which is the integrated GPU on the aforementioned CPU. All experiments on both devices were performed under Intel's Devcloud environment.

For this purpose, we also construct a set of synthetic sparse tensors in such a way that the worst case and best case performance can be attained. These synthetic best-case and worst-case sparse tensors are constructed based on the AI, derived in Equations 6 and 9, as well as on the characteristics of the employed CPU and GPU devices. Furthermore, the performance for some real-world tensors is also evaluated to serve as a term of comparison.

Best Case Performance Analysis The greatest challenges in most approaches to sparse TTM are load balancing and data locality. While the first challenge can to some extent be mitigated with efficient algorithms, the second challenge is mostly dependent on the specific features of the sparse data-set used. For that reason the prime candidate to achieve maximum performance with both kernels 1 and 2 is to construct a semi-sparse tensor, i.e., a tensor that is sparse in all its dimensions except for one.

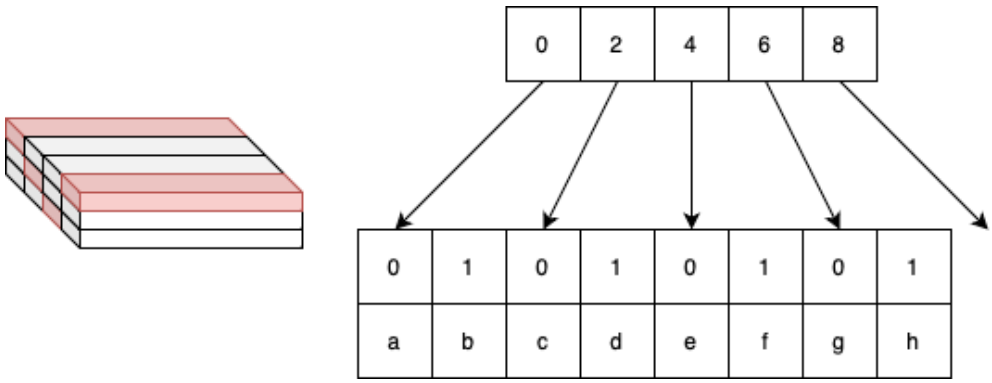


Figure 47 *Semi-sparse tensor and CSF's representation of its fibers*

This specific disposition, depicted in Figure 47, consists of a tensor with its horizontal mode dense, meaning all fibers fibers of this mode are either empty or fully dense. In other words, there are several dense fibers sparsely scattered across the tensor.

The distribution of the non-zero elements in the proposed best-case synthetic sparse tensor allows for both load balancing, since all fibers have the same length, as well as data locality, since all fibers access consecutively the rows of the matrix. To construct the best-case scenario, there are three parameters that must be considered: *i)* FbrCnt – the number of fibers with non-zero elements, *ii)* FbrLen – the number of non-zeros in each fiber and *iii)* ColCnt – the number of columns in the matrix. Note that for this specific kind of tensor, since fibers are dense, the number of rows of the matrix is the same as the number of non-zero elements in each fiber.

Two more parameters can be deduced from three aforementioned parameters, i.e., the number of threads created and the matrix size, which may slightly differ depending on the kernel used. For kernel 1, since every fiber-column dot product is assigned to a different thread, the number of threads created is FbrCnt $\times$ ColCnt. On the other hand, for kernel 2, each thread computes one fiber against the whole matrix, therefore the number of threads created is FbrCnt. Finally, the size of the matrix is defined by FbrLen $\times$ ColCnt.

CPU Analysis The CPU architecture under evaluation (Intel Core i9-11900KB) involves a memory hierarchy that includes a set of private (L1 and L2) and shared L3 caches. As such, there are different best cases depending on which of the cache levels is being addressed. Since the fastest cache is L1, the best case corresponds to the scenario when the matrix elements fit into L1. However, this requires the matrix to be very small which, according to Equations 6 and 9, causes the AI to be very low. As it can be observed, it is non-trivial to find the optimal trade-off between a cache fitting size and a high AI.

To determine this trade-off some empirical analysis is required, for which a starting point is required and must be chosen bearing in mind three crucial aspects: *i)* FbrCnt should be large enough, such that the workload is sufficiently large to provide the statistically relevant measures; *ii)* FbrLen is large enough, such that the AI of the kernel is close to its maximum; and *iii)* FbrLen $\times$ ColCnt is small enough, such that the matrix fits in the desired cache level. Let the following parameters be considered as a starting point: FbrCnt $= 131072$ for both kernels, then FbrLen $= 256$ for kernel 1 and FbrLen $= 512$ for kernel 2, and ColCnt $= 64$ for kernel 1 and ColCnt $= 256$ for kernel 2.

Figure 48 shows the behaviour of kernels 1 and 2 when changing just one of the parameters at a time, in this case the number of non-zero elements in the fiber. One can notice irregular spikes

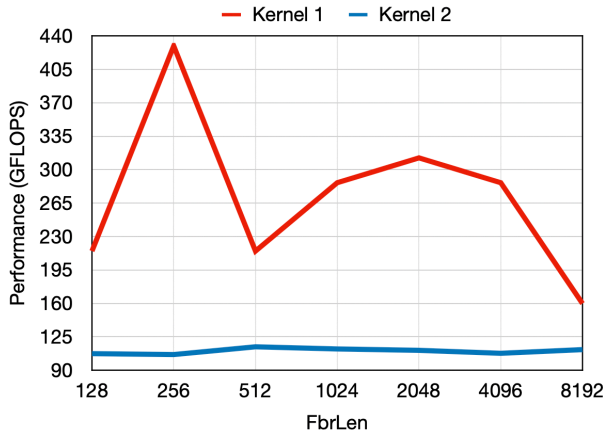


Figure 48 *CPU performance for different number of non-zero elements in the fiber*

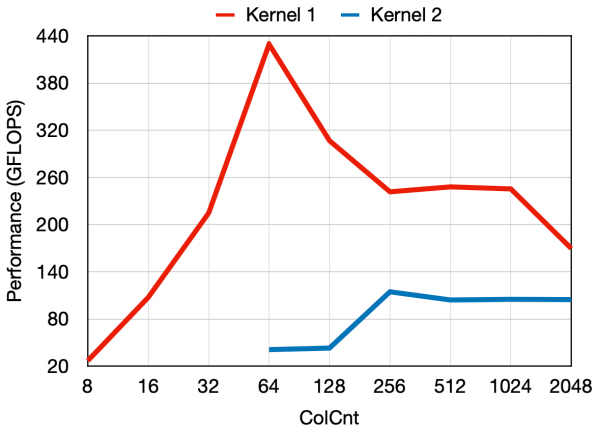


Figure 49 *CPU performance for different number of columns in the matrix*

and drops of performance in the obtained experimental results for kernel 1, while for kernel 2 the performance does not change drastically. The spikes are explained by having a larger FbrLen, since it provides higher AI and therefore allows for better performance. On the other hand, the drops are explained by the cache levels in the architecture, i.e., whenever the matrix becomes too large to fit in a cache level, the kernel becomes bound by the next level, which has lower bandwidth, thus making the TTM execution slower.

In Figure 49, when increasing the number of columns, kernel 1 displays increasing performance until ColCnt $= 64$ and then the performance starts to decrease, since the data-set no longer fits in L1 cache. A similar behaviour can be observed for ColCnt $= 1024$, but for L2 cache. Kernel's 2 AI also depends on ColCnt, but its behaviour is similar, performance increases until ColCnt $= 256$ and then slowly decreases as the number of columns is increasing.

Figure 50 provides the CARM characterization of kernels 1 and 2 for this specific CPU architecture. The gray zone represents the theoretical limits of AI calculated for kernel 2, while kernel's 1 AI bounds are also within these limits. It is possible to observe that the performance of both kernels is limited by the bandwidth of cache levels (kernel points are positioned between L2 and L3 rooflines), thus suggesting a high efficiency and data reuse in the synthetic tensors created to simulate the best case performance scenario. It is also worth noting that due to compiler optimisations, namely vectorization, kernel 1 achieves higher performance and AI than kernel 2.

GPU Analysis For the GPU analysis of proposed TTM approaches, we relied on Intel 11th Gen UHD Graphics, which is the integrated GPU on Intel Core i9-11900KB. Given a memory-bound nature of the TTM kernels, the best-case exploration strategy aims at ensuring that the AI of kernels is as high as possible. In addition, the parallelism should be high in order to fully exploit the massively parallel GPU architecture, while the matrix should fit in the GPU L3 cache, to avoid additional loads from the CPU.

For kernel 1, this means making FbrLen sufficiently large such that Equation 8 is verified, while ensuring FbrCnt $\times$ ColCnt large enough to keep the GPU units occupied and keeping FbrLen $\times$ ColCnt small enough to ensure that the data-set fits in the GPU L3 cache. A similar rationale is followed for kernel 2, where sufficiently large FbrLen and ColCnt should be provisioned to satisfy Equation 11, while large enough FbrCnt is required to maximize the occupancy of the GPU units. In addition, small enough FbrLen $\times$ ColCnt is needed to guarantee that the data-set fits in the L3 cache. As in the CPU case, a starting point is required to be chosen accord-

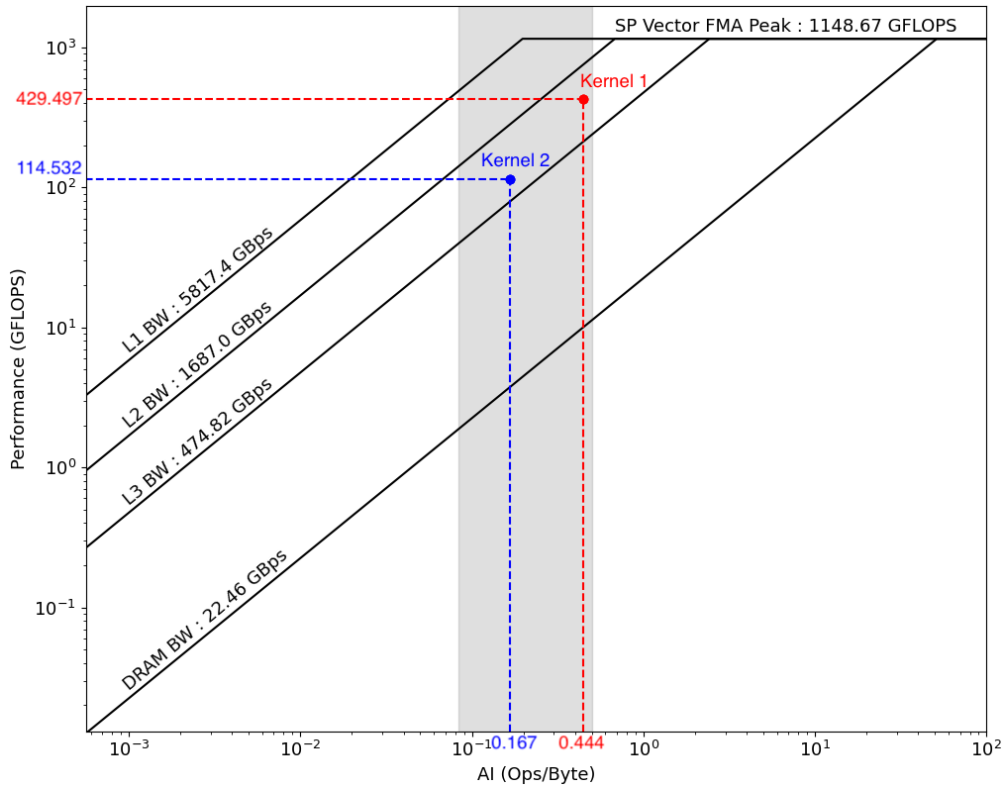


Figure 50 *Roofline model for CPU best case scenario with both kernels*

ing to these restriction, which is in this case set to: FbrCnt = 131072 which is enough to fully utilize the GPU's compute units, FbrLen = 512 which ensures AI very close to the theoretical maximum and ColCnt = 64.

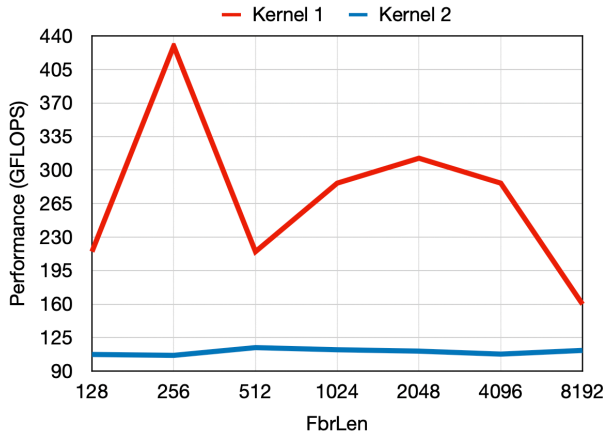


Figure 51 *GPU performance for different number of non-zero elements in the fiber*

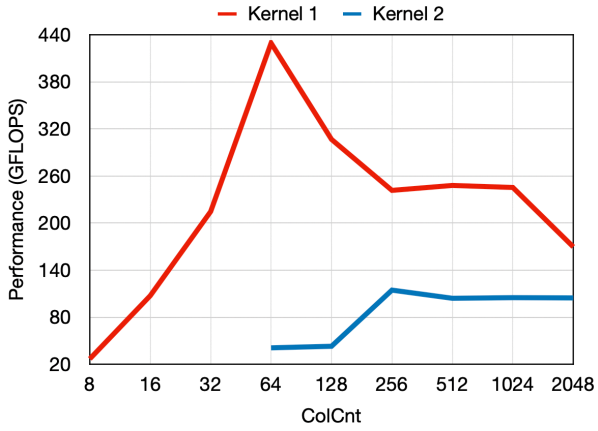


Figure 52 *GPU performance for different number of columns in the matrix*

Figures 51, 52 and 53 depict the behaviour of kernels 1 and 2 on the tested GPU device for different ranges of parameters on both kernels. In Figure 51, the performance of kernel 1 increases as the length of the fibers is increasing and starts stabilising around FbrLen = 256. However, when FbrLen is increased even further performance starts to drop drastically for kernel

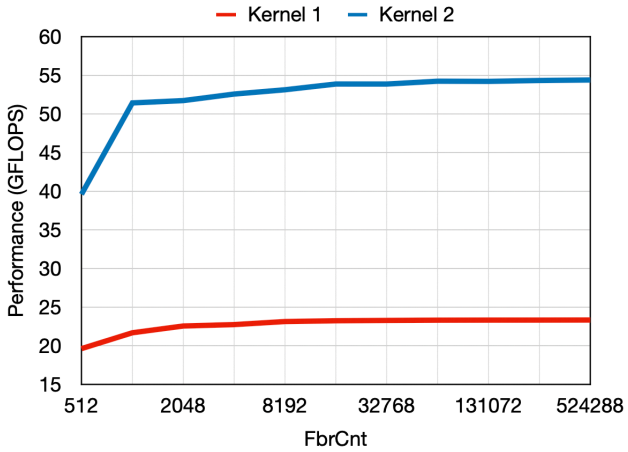


Figure 53 *GPU performance for different number of fibers with non-zero elements*

1. The reason for this behaviour lies in the fact that the matrix size is directly proportional to the FbrLen parameter, thus beyond the FbrLen $= 2048$ the matrix does not fit in GPU L3 cache. Such phenomenon is not observed for kernel 2, since each thread has a fiber assigned and therefore it efficiently streams through the tensor fibers. In Figure 52, the behaviour is very similar to the one observed in Figure 51. Performance increases until ColCnt $= 32$ for kernel 1 and ColCnt $= 64$ for kernel 2, where it maintains stable until ColCnt $= 256$ for kernel 1. After this point, any further increase in matrix size results in reduced performance for both kernels. In Figure 53, one can observe that the performance increases as the number of fibers is increasing and starts stabilising around FbrCnt $\geqslant 65536$, signalling that the GPU units are fully occupied. As previously referred, the fiber-centric TTM approach in kernel 2 creates less threads, concentrating more workload in each thread, which leads to having more factors affecting its AI and making its behaviour more irregular.

Finally, in Figure 54, the roofline model for the GPU architecture is presented. It is possible to observe that the performance is very close to the maximum achievable by the device for the corresponding AI in both kernels.

Worst Case Performance Analysis Following a similar reasoning as for the best case, it is also possible to determine the worst-case TTM processing scenario in order the uncover the lower bounds on the performance attainable with the proposed TTM kernels on both CPU and GPU architectures. The strategy followed herein aims at analysing the worst case scenario under the condition that the full utilization of processing resources is attained with a data distribution in the specifically created synthetic sparse tensor that hinders performance.

For this purpose, the data distribution can be modeled with four parameters: *i)* FbrCnt – the number of fibers with non-zero elements; *ii)* FbrLen – the number of non-zeros in each fiber; *iii)* ColCnt – the number of columns in the matrix, and *iv)* RowCnt – the number of rows in the matrix. It is important to notice that, unlike what happens in the semi-sparse tensor, the number of non-zero elements in the fiber do not match the number of rows in the matrix. Instead, RowCnt is now used to denote the effective size of the fiber, which is always greater than FbrLen.

Two more parameters can be deduced from these four parameters, which slightly differ depending on the kernel used. For kernel 1, since every fiber-column dot product is assigned to a different thread, the number of threads created is equal to FbrCnt $\times$ ColCnt. On the other hand, for kernel 2, each thread computes one fiber against the whole matrix, therefore the number of threads created is equal to FbrCnt. Finally, the size of the matrix is defined by RowCnt $\times$ ColCnt.

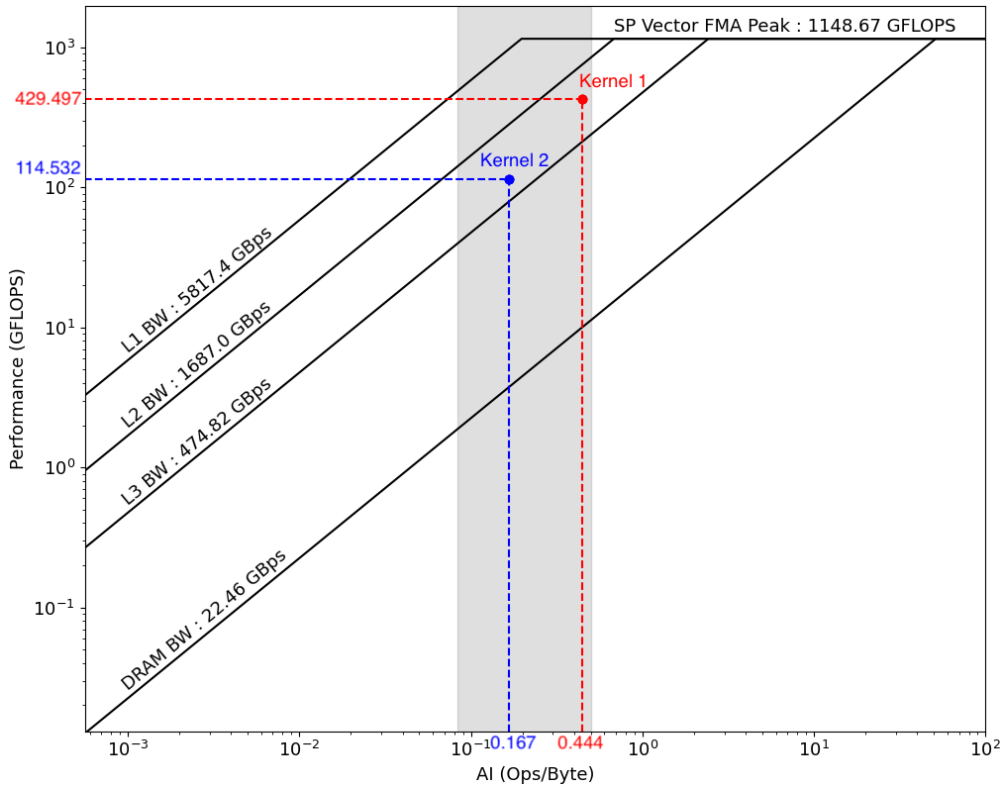


Figure 54 *Roofline model for GPU best case scenario with both kernels*

To facilitate the comparison of the results obtained with this scenario to the previously presented ones, both FbrCnt and ColCnt were kept the same, while only RowCnt was varied. This way, the workload is always kept at about the same size, with the only difference being the distribution of the non-zero elements across the fibers.

CPU Analysis For the CPU, instead of targeting specific cache levels, the idea behind the worst-case scenarios is to prevent any data reuse in the cache hierarchy, thus limiting the kernel performance to the lowest bandwidth available in the CPU memory hierarchy, i.e., DRAM.

To further accentuate the worst case performance scenario, a lower AI is also desirable. For kernel 1, this means making FbrLen $= 1$ such that Equation 7 is verified, while keeping RowCnt $\times$ ColCnt large enough to ensure that the tensor data does not fit in any cache level. For kernel 2, both FbrLen and ColCnt should be equal to one in order to satisfy Equation 10, while keeping RowCnt $\times$ ColCnt large enough, such that the data-set does not fit in any cache level.

Figure 55 provides one possible representation of a synthetic worst-case sparse tensor with long fibers, each with a single non-zero element. It is worth noting that the non-zero elements are displaced across fibers in such a way that they do not allow for any reuse of the matrix elements. To test and model the parameters of the proposed approach, we relied on Intel Core i9-11900KB CPU, for which setting RowCnt $= 524288$ proved to be large enough to display a minimal performance.

Figure 56 represents the roofline model for kernels 1 and 2 for Intel Core i9-11900KB CPU under the worst-case performance scenarios. In both cases, it is possible to observe that the objective of preventing data reuse in any of the cache levels is achieved, since both kernels are positioned near the respective DRAM roof. It is possible to observe that due to compiler

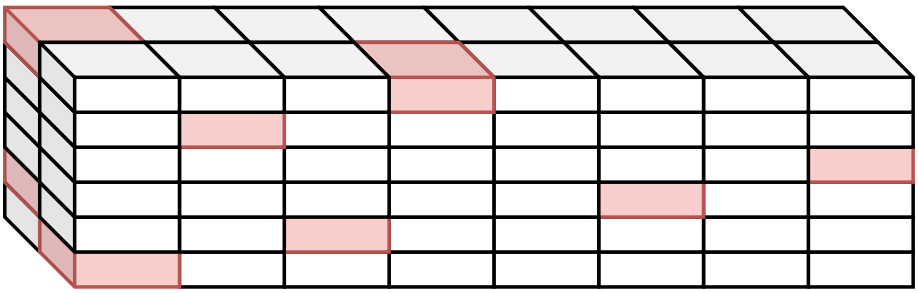


Figure 55 *Depiction of worst case tensor*

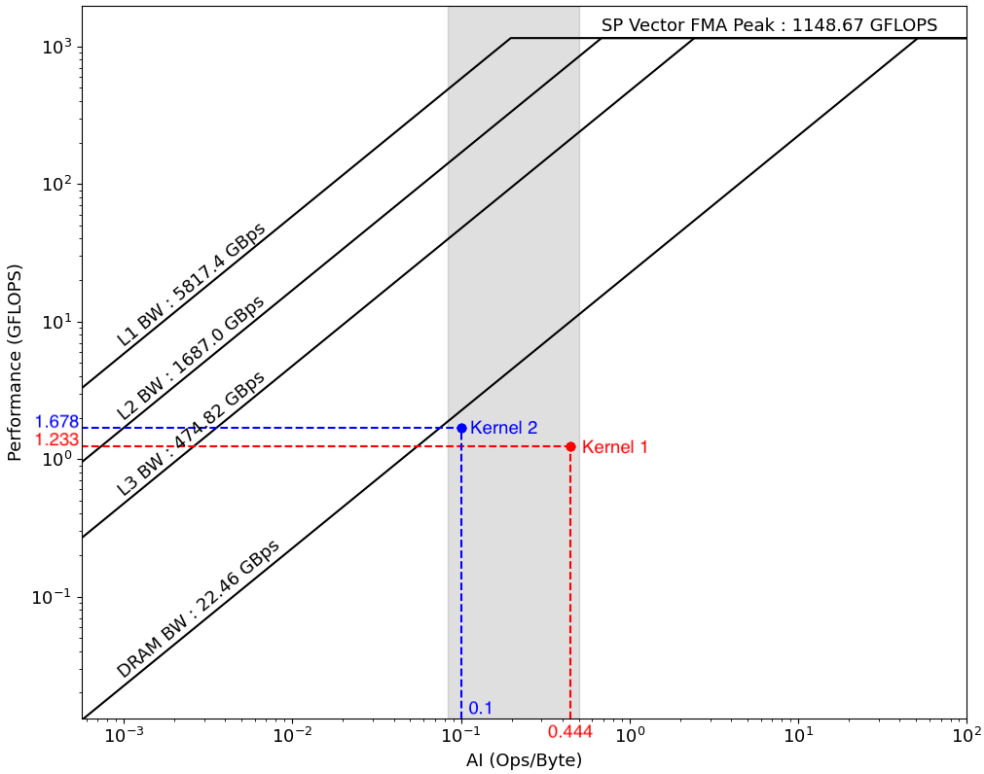


Figure 56 *Roofline model for CPU worst case scenario with both kernels*

optimisations, namely vectorization, kernel 1 achieves higher AI than kernel 2.

GPU Analysis In order to exercise the worst-case performance scenario on the GPU architecture, it is necessary to construct the synthetic sparse tensors that allow for the kernels' AI to be as low as possible, since TTM is memory bound on the GPU, while also ensuring that the matrix does not fit in the GPU L3 cache, thus enforcing constant loads from the CPU.

To create the worst-case synthetic sparse tensor for kernel 1, it is needed to ensure $\texttt{FbrLen} = 1$ such that Equation 7 is verified, while keeping $\texttt{RowCnt} \times \texttt{ColCnt}$ large enough such that the matrix does not fit in GPU L3 cache. For kernel 2, both FbrLen and ColCnt should be equal to one such that Equation 10 is satisfied, while keeping $\texttt{RowCnt} \times \texttt{ColCnt}$ large enough to prevent reuse of matrix elements in GPU L3 cache.

As in the best-case scenario, we conducted our experiments on the Intel 11th Gen UHD

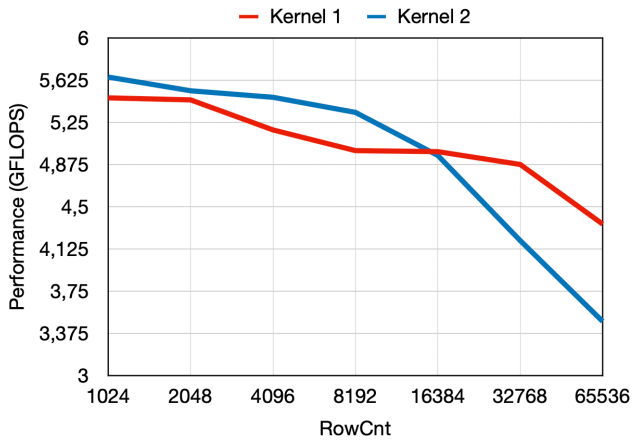


Figure 57 *GPU performance for different number of rows in the matrix*

Graphics, integrated GPU on Intel Core i9-11900KB, in order to test and model the worst-case parameters for this specific architecture. Figure 57 presents the performance variation of both kernels with respect to the different RowCnt values. As it can be observed in Figure 57, the performance is increasing with the increase of RowCnt. This happens because the size of the matrix is increasing, thus forcing the GPU to load data from outside the GPU L3 cache.

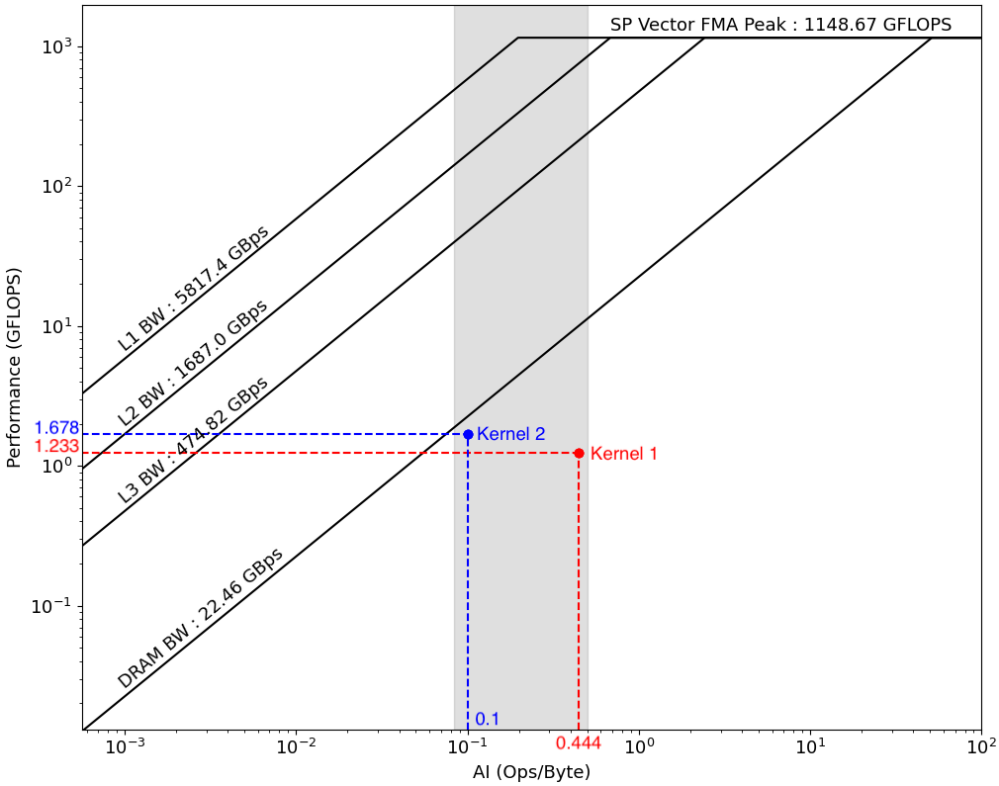


Figure 58 *Roofline model for GPU worst case scenario with both kernels*

Figure 58 represents the roofline model for kernels 1 and 2 for this specific architecture under the worst-case evaluation scenario. In both cases, it is possible to observe that the objective of being outside of the GPU L3 cache is achieved. However, it is important to notice that, since

ColCnt was kept at 64 columns (necessary to achieve a fair comparison), the AI of kernel 2 is higher than the minimum.

Analysing Performance with Real-World Tensors After having both the worst and best case scenarios discussed, and theoretically and experimentally verified, we focus herein on analysing the proposed two approaches for TTM data-parallel processing in real-world execution scenarios. For this analysis, the nell-2 and vast-3D tensors from the FROSTT data-set[20] were used, as described in Table 5. Since the majority of the tensor parameters are already predetermined by their structure, the only parameter that can be modified for the TTM computation over those real sparse tensors is the number of columns in the matrix, i.e., ColCnt.

	nell-2[21]	vast-3D[22]
FbrCnt	337 365	26 021 945
NNZ	76 879 419	26 021 945
Mode 0	12 092	165 427
Mode 1	9 184	11 374
Mode 2	28 818	2

Table 5 *Description of the datasets used*

CPU Analysis To explore the performance of real-world tensors on a multi-core CPU architecture and provide comparison with the best and worst cases elaborated in the previous sections, we relied on the Intel Core i9-11900KB CPU.

Given the characteristics of the **nell-2 tensor**, we can determine that it is expected to attain the AI very close to the theoretical maximum (an average FbrLen = 228), while the matrix has 28818 rows (imposed by the mode-2 dimension), thus it never fits in L1 cache.

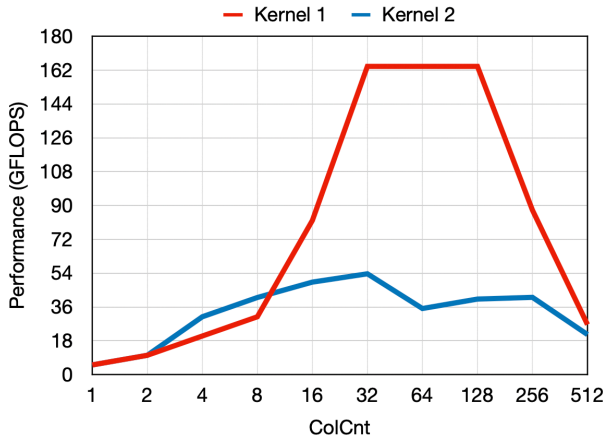


Figure 59 *CPU performance for different number of columns in the matrix*

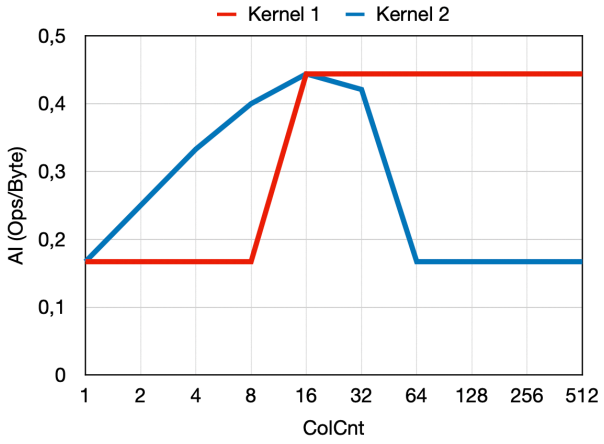


Figure 60 *AI on CPU for different number of columns in the matrix*

Figures 59 and 60 represent the performance and AI for both kernels for tensor nell-2 with the increasing number of columns in the matrix. For kernel 1, the AI should be fixed, since it only

[20] Shaden Smith et al. *FROSTT: The Formidable Repository of Open Sparse Tensors and Tools*. 2017. URL: http://frostt.io/.

depends on FbrLen and not on ColCnt, however from ColCnt = 16, the compiler is capable of vectorizing kernel loops and therefore provoking an increase in both the AI and the performance. On the other hand, for kernel 2, as expected, since the AI depends on the number of columns, it starts with the same value as in the previous kernel and, with the increase in number of columns, increases until very close value to the maximum calculated in Equation 11. However, since the kernel loops are never vectorized by the compiler, the AI ends up dropping for larger workloads. From ColCnt = 128, the performance starts reducing for kernel 1, while the AI is maintained, since the matrix does not fit anymore in any of the caches of the CPU. On kernel 2, performance increases with the AI as the problem is always memory bound for all cache levels but L1, where the matrix would never fit anyway. It is also important to notice that the performance drops mostly match with the sizes of the cache levels.

When compared with the performance of kernel 1 for the nell-2 tensor, the kernel 2 attains the lower performance. The main reason behind this behavior lies in the irregularity of data accesses and workload imbalance that were not present in the best case evaluation.

For the **vast-3D tensor**, which has all fibers with a single non-zero element, the AI should be very close to the theoretical minimum. Due to its mode-2 dimension, the matrix only has 2 rows, thus for all numbers of columns tested, it always fits in L1 cache.

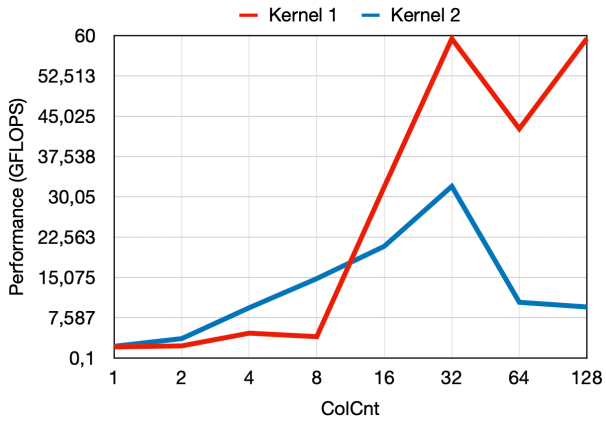


Figure 61 *CPU performance for different number of columns in the matrix*

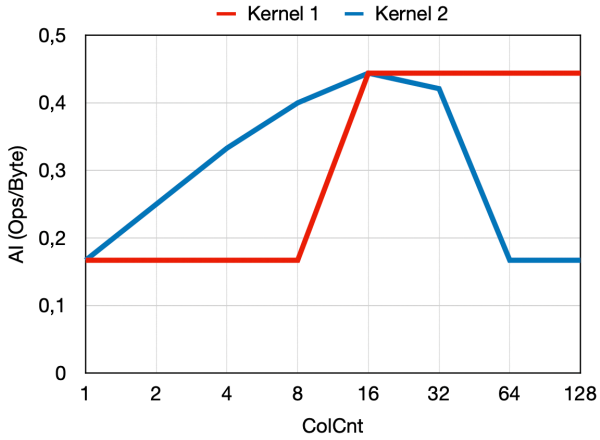


Figure 62 *AI on CPU for different number of columns in the matrix*

Figures 61 and 62 represent the performance and AI of both kernels for tensor vast-3D with the increase of number of columns in the matrix. As in the nell-2 tensor case, the expected constant AI for kernel 1 was not observed due to the compiler optimisations, which provokes an increase in both AI and kernel performance for a larger number of columns. This behaviour can be observed around ColCnt = 16, where a notable AI increase occurs due to the compiler's ability to vectorize the kernel loops. Since the matrix always fits in L1 cache the performance drop that happened in the nell-2 tensor for the same kernel does not happen with the vast-3D tensor. For kernel 2, although the AI was expected to be lower, it is not due to compiler optimization, it still behaves as expected. It starts with the same value as in the previous kernel and, with the increase in number of columns, increases until very close to the maximum calculated in Equation 11. However, since the kernel loops are never vectorized by the compiler, the AI ends up dropping for larger workloads. The performance also increases with the AI, which can be explained by most of the workload coming from the number of columns, since each fiber only has one element to compute.

GPU Analysis Resorting again to Intel 11th Gen UHD Graphics, which is the integrated GPU on Intel Core i9-11900KB, it is possible to test, for this specific architecture, these real-world tensors and compare them with the best and worst cases explored in the previous sections.

As previously referred, the **nell-2 tensor** has an average of $\mathtt{FbrLen} = 228$, thus its AI is very close to the theoretical maximum. In addtion, the tensor has 337365 fibers, thus GPU occupation is not an issue.

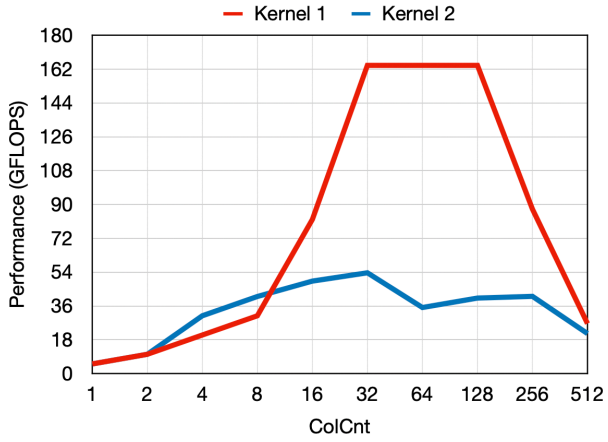


Figure 63 *GPU performance for different number of columns in the matrix*

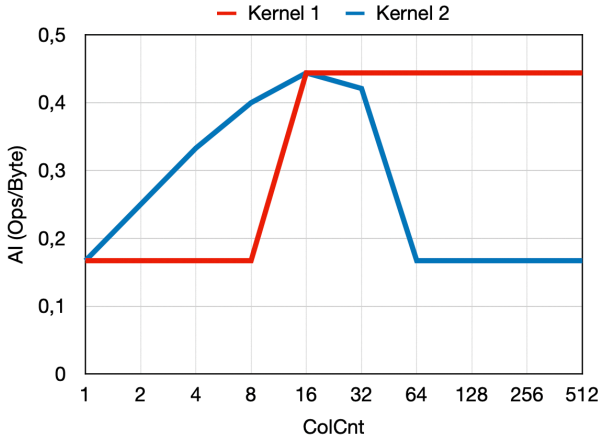


Figure 64 *AI on GPU for different number of columns in the matrix*

Figures 63 and 64 represent the performance and AI for both kernels for tensor nell-2 with the increase of number of columns in the matrix. For kernel 1, the AI is fixed as it only depends on the FbrLen and not on ColCnt. The main difference between this situation and the best case scenario, described in Section 3.2.2, is the irregularity and unbalance in the distribution of the non-zero elements over the fibers as well as the size of the matrix. For such reasons, the peak performance is achieved for $\mathtt{ColCnt} = 32$, instead of $\mathtt{ColCnt} = 64$, and it is slightly lower than the perfmance in the best case. For kernel 2, as expected, the AI starts with the same value as in the previous kernel but increases until very close value to the theoretical maximum calculated in Equation 11. The performance also increases with the AI as the problem is memory bound on the GPU.

When compared with the performance of kernel 1 for the same tensor, this kernel performs worst even though it performed better for the best case tensor. The main reason is the irregularity in data accesses and workload imbalance that were not present in the best case, when kernel 2 was the better option.

Tensor vast-3D, which has all fibers with a single non-zero element, should force the kernel to have an AI very close to the theoretical minimum. Due to a large number of fibers in this tensor (26021945), the GPU occupancy is not an issue.

Figures 65 and 66 represent the performance and AI for both kernels for tensor vast-3D with the increasing number of columns in the matrix. For kernel 1, the AI is fixed, as expected, since it only depends on FbrLen and not ColCnt. The main difference between this situation and the worst case scenario, described in Section 3.2.2, is RowCnt which causes the matrix to be much smaller in this tensor. For kernel 2, again as expected, the AI starts with the same value as in the previous kernel but increases with the increasing number of columns. The performance also increases with the AI as TTM is by nature memory bound on the GPU.

When compared, kernel 2 performs better than kernel 1 for tensor vast-3D. The main reason

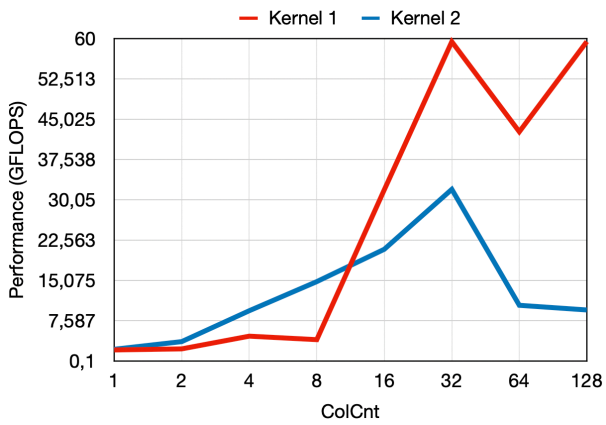


Figure 65 *GPU performance for different number of columns in the matrix*

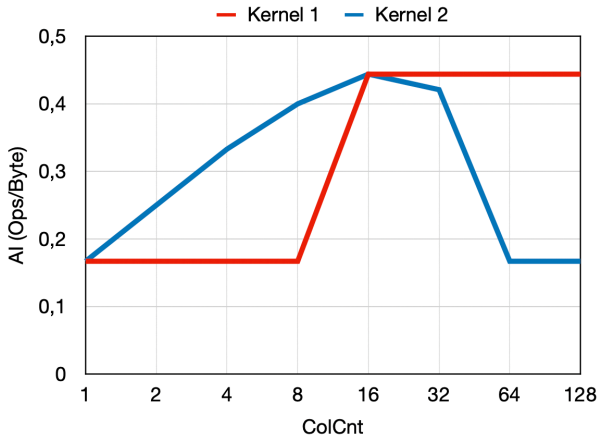


Figure 66 *AI on GPU for different number of columns in the matrix*

is the matrix size, since it is so small, thus it is efficient to compute whole rows as it is likely that the row is already cached. Finally, it is possible to create a roofline model sweeping the AI by changing the number of columns. This analysis, however, is only applicable for kernel 2, since in kernel 1 the AI is fixed, and for the GPU, since in the CPU the compiler tends to perform optimizations which make the AI's behaviour unpredictable. These can be found in Figures 67 and 68.

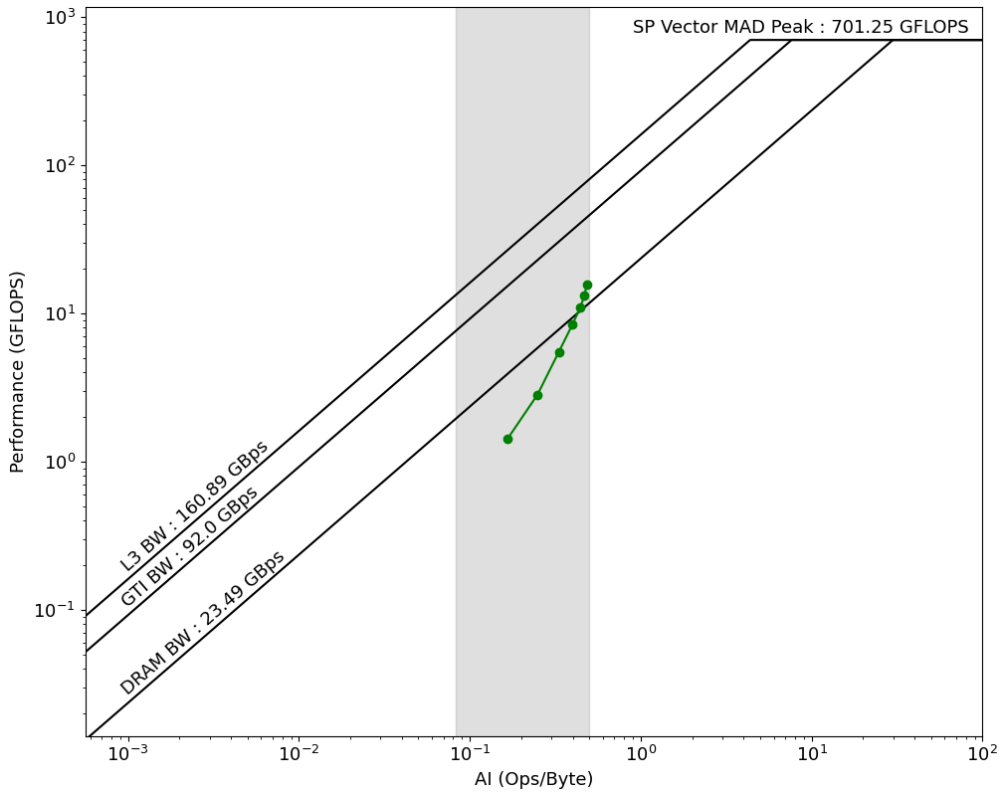


Figure 67 *Roofline model for kernel 2 with tensor nell-2 on the GPU*

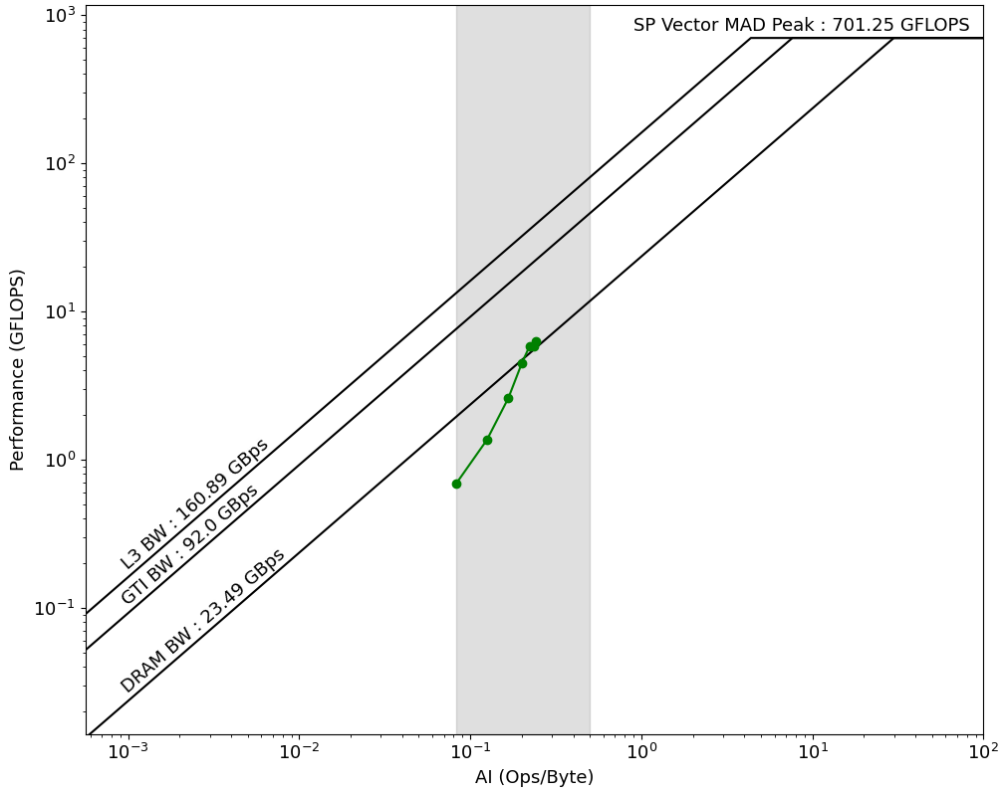


Figure 68 *Roofline model for kernel 2 with tensor vast-3D on the GPU*

4 COMMUNICATION AND PROFILING TOOLS FOR EMERGING MICROARCHITECTURES

4.1 EXTENDING COMMUNICATION ANALYSIS TOOLS TO AMD MULTICORES

We developed ComDetective, an inter-thread communication analyzer, and ReuseTracker, a reuse distance analyzer, that leverage the hardware features in AMD processors to support low-overhead profiling. Both tools employ the instruction-based sampling (IBS) facility and debug registers in AMD processors to detect inter-thread communication and data reuse. Different from prior arts, ComDetective differentiates the communication into true and false sharing, and ReuseTracker measures reuse distance in private and shared caches by also considering cache line invalidation with low overhead. Both tools can attribute the communications and reuses to source code lines. To our knowledge these tools are two of the few profiling tools designed specifically for AMD x86 architectures using IBS. These tools are timely and relevant considering the rise in numbers of AMD processor based data centers and HPC systems.

Even though the original communication detection and reuse distance algorithms remain unchanged, extending the Intel-based profiling tools to the AMD multicores is not a straightforward task and comes with many challenges. AMD IBS is very different from Intel PEBS in hardware design as it can be programmed to count and sample only instruction fetches or executed micro-operations. To target only specific events such as memory accesses for profiling, software-level filtering is needed to choose only memory accesses among all micro-operation samples to be taken as inputs for the profiling tool. Furthermore, AMD IBS requires certain BIOS software that

is not widely available[23] even in the cloud machines. As an alternative solution, we relied on a Linux kernel module[24] that allows us to program IBS hardware. Originally this kernel module is designed to count hardware samples and produces a log of samples. As the kernel module has a simplistic workflow, we extensively modified the kernel module in order to introduce additional functionalities necessary for supporting our profiling tools.

4.1.1 IBS DRIVER

We implement ComDetective and ReuseTracker on top of the open-source HPCToolkit performance analysis tools suite.[25] To profile multi-threaded applications in AMD machines, both tools leverage IBS features.[26] To configure and sample IBS, there are two possible options. The first option is to use `perf_event_open` system call. However, this option requires certain BIOS software, which is not always available by default[27].[28] The second option is to use a Linux kernel module. Since using `perf_event_open` and the BIOS software is not a reliable option, we rely on an open source Linux kernel module named AMD_IBS_Toolkit.[29] For ease of prose, we refer to this Linux kernel module as IBS driver. This IBS driver allows user application to configure IBS and retrieve samples from IBS. This is a different approach from the way precise event sampling is handled in Intel. Since the use of `perf_event_open` to program PEBS in Intel does not require any special firmware, ComDetective and ReuseTracker could always leverage `perf_event_open` to configure and sample PEBS.

Two flavors of sampling can be performed using IBS – fetched instruction sampling and executed micro-operation sampling. The IBS driver allows user applications to enable or disable IBS counters and set up sampling period using the `ioctl` interface of the driver. To further support the profiling tools, additional capabilities need to be introduced to the IBS driver. These capabilities are to i) allow user threads to register themselves as valid recipients of sampling signals, ii) have the interrupt handler record only samples from memory accesses that have valid effective addresses, iii) send OS signals from the interrupt handler to the user threads that trigger the IBS interrupts if the threads are already registered and the samples are from memory accesses with valid effective addresses.

Upon its installation, the IBS driver creates a number of character device files, each of which serves as an interface to the IBS hardware of each CPU core. For ease of reference, we call a character device file as a device file from this point on. The number of device files that are created is twice the number of logical cores in the machine, such that for each logical core there are two device files for sampling fetched instructions and executed micro-operations respectively. After creating the device files, the IBS driver also registers a function as an interrupt handler that will handle any hardware interrupt due to an IBS sample. Note that for the profiling tools we are interested in executed micro-operations because load and store operations that we need to

[23] Joseph L. Greathouse. *Re: Error : IBS profiling is disabled in your BIOS*. https://community.amd.com/t5/general-discussions/error-ibs-profiling-is-disabled-in-your-bios/td-p/55043. AMD Community; Joseph L. Greathouse. *Re: IBS not available on EPYC 7451 ?* https://community.amd.com/t5/server-gurus-discussions/ibs-not-available-on-epyc-7451/m-p/258228. AMD Community.

[24] Joseph L. Greathouse. *AMD Research Instruction Based Sampling Toolkit*. https://github.com/jlgreathouse/AMD_IBS_Toolkit. 2017.

[25] L. Adhianto et al. "HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs". *Concurrency Computation: Practice Experience* 22.6 (2010), pp. 685–701.

[26] Paul J. Drongowski. *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors*. https://pdfs.semanticscholar.org/5219/4b43b8385ce39b2b08ecd409c753e0efafe5.pdf. 2007.

[27] Greathouse, *Re: Error : IBS profiling is disabled in your BIOS* .

[28] Greathouse, *Re: IBS not available on EPYC 7451 ?*

[29] Greathouse, *AMD Research Instruction Based Sampling Toolkit*.

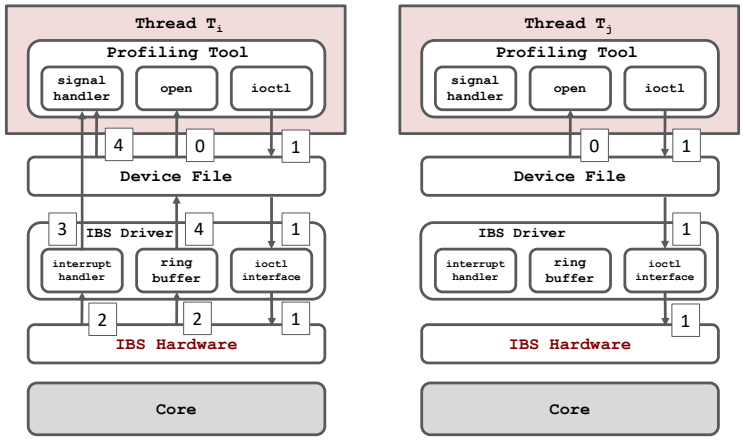


Figure 69 *One possible workflow scenario of the IBS driver: 0) Every thread calls* `open` *system call to get the file descriptor of the device file that corresponds to the core it is running on. 1) Every thread uses* `ioctl` *system call on the file descriptor to configure the sampling period of IBS, sets up the size of the ring buffer that will contain sampled data, registers its thread ID to the interrupt handler, and initializes the IBS counter. 2) Thread* T_i*'s IBS counter overflows, the interrupt handler handles the hardware interrupt triggered by the overflow, and the interrupt handler copies the sampled data from IBS' model-specific registers (MSRs) to the ring buffer. 3) The interrupt handler sends an OS signal to the thread that triggered the interrupt, i.e. thread* T_i*. 4) A signal handler that runs in* T_i*'s address space handles the OS signal, and reads the device file to retrieve the sampled data.*

monitor are subsets of micro-operations.

4.1.2 Interaction between IBS driver and profiling tools

Figure 69 displays the workflow of the IBS driver during profiling. When ComDetective or ReuseTracker begins profiling an application, each application thread, which also runs the profiling tool's code in its address space, opens a device file for sampling executed micro-operations that belongs to the logical core it is running on. By interfacing with the device file using the `ioctl` system call, each thread configures the sampling period of IBS in its core, sets up the size of the ring buffer that will contain sampled data, and activates IBS counter in its core. In addition to these configurations, we modified the IBS driver to allow a thread to register its thread ID so that the sampling interrupts whose sampled data are to be copied to a ring buffer by the interrupt handler are only those encountered by registered threads. Facilitated by this modification, each application thread also registers its thread ID in the beginning of profiling.

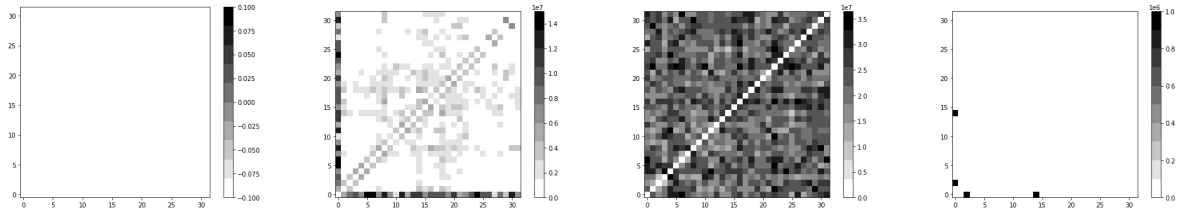
When an IBS counter overflow happens in a CPU core, a hardware interrupt is triggered and the interrupt handler is called by the IBS driver. This interrupt handler copies sampled data from IBS' special purpose registers in that CPU core to a ring buffer. Since both ComDetective and ReuseTracker only need memory access samples to profile multithreaded code, we modified the interrupt handler to allow only micro-operation samples that are memory accesses with valid instruction pointers and valid effective addresses to be copied to the ring buffer. For ease of reference, we refer to these samples as *valid samples*. To access the sampled data from the ring buffer, a user thread needs to read it from the device file that belongs to the CPU core.

By default, the IBS driver does not support signal delivery to user threads upon sampling interrupt. To enable profiling threads to get notified every time a valid sample occurs, we modified the IBS driver to send an OS signal to the user thread that triggers the interrupt. At a sampling interrupt, an OS signal will be sent to the user thread that causes the interrupt only if that thread has registered itself to the IBS driver. Upon handling a sampling signal, a profiling thread reads

the device file corresponding to the CPU core that encounters the interrupt to retrieve the sampled data.

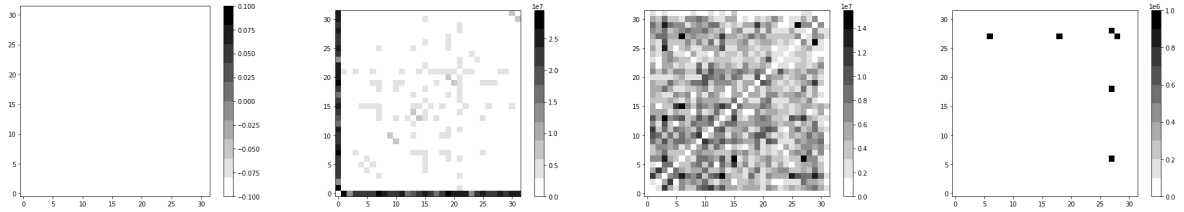
In AMD machines, both ComDetective and ReuseTracker interface with the modified IBS driver to configure the parameters of IBS sampling, and retrieve data from valid samples. In each IBS sample, both tools read the `IbsDcLinAd` and `IbsOpMemWidth` attributes of the sampled data to extract the sampled effective address and the width of the accessed memory region respectively. In addition to getting sampled addresses, ComDetective also checks the `IbsStOp` and `IbsLdOp` flags of each sample to see if a sampled memory access is a store or a load operation.

For time reuse distance computation in ReuseTracker, a separate hardware counter aside from IBS is used to count the number of memory accesses between a *use* and a *reuse*. This hardware counter is the counter for dispatched load or store micro-operations. When computing time reuse distance for intra-thread profiling, the time reuse distance is the subtraction of a CPU core's hardware counter value at *use* from the counter's value at *reuse*. In shared cache profiling, the time reuse distance is the subtraction of the sum of counter values of all CPU cores that share the same shared cache at *use* from the sum of counter values of the same CPU cores at *reuse*.



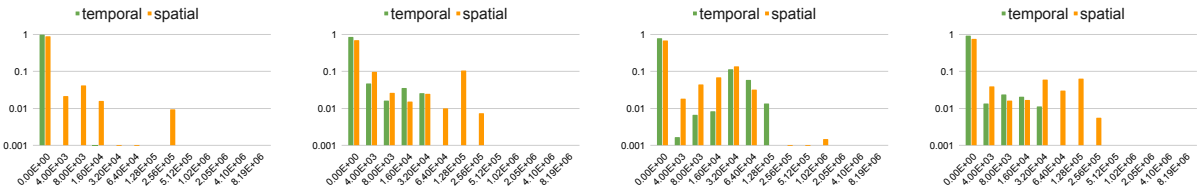
(a) *blackscholes* **(b)** *bodytrack* **(c)** *streamcluster* **(d)** *freqmine*

Figure 70 *Communication matrices of some PARSEC benchmarks in the Intel machine.*



(a) *blackscholes* **(b)** *bodytrack* **(c)** *streamcluster* **(d)** *freqmine*

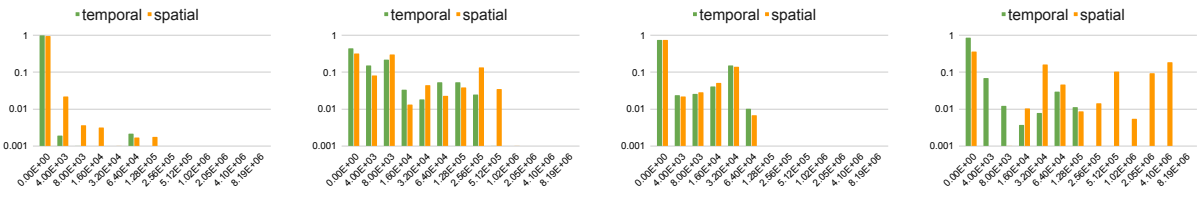
Figure 71 *Communication matrices of some PARSEC benchmarks in the AMD machine.*



(a) *blackscholes* **(b)** *bodytrack* **(c)** *streamcluster* **(d)** *freqmine*

Figure 72 *Histograms of reuse distance in private caches of some PARSEC benchmarks in the Intel machine.*

We perform experiments to evaluate the accuracy and overheads of the proposed tools on an AMD machine with two-socket EPYC 7352 processors. ComDetective exhibits high accuracy



(a) *blackscholes* **(b)** *bodytrack* **(c)** *streamcluster* **(d)** *freqmine*

Figure 73 *Histograms of reuse distance in private caches of some PARSEC benchmarks in the AMD machine.*

while introducing 5.14× runtime and 1.4× memory overheads. ReuseTracker also displays high accuracy, which is 95%, with 11.76× runtime and 1.46× memory overheads. These overheads are much lower than the overheads of existing simulators and code instrumentation-based tools. Lastly, we demonstrate the usage of the tools by having ComDetective and ReuseTracker facilitate the code refactoring of two data mining benchmarks to improve their performance by up to 29%.

4.1.3 Evaluation: ComDetective on AMD vs Intel

Our AMD machine is a 2-socket AMD EPYC 7352 CPU from Zen 2 microarchitecture family. There are 24 cores per socket with 2-way simultaneous multi-threading in this machine, and each core has its own local L1i, L1d, and L2 caches. We use Linux 5.11.0 and GNU-10.3.0 toolchain in this machine. The Intel machine is a 2-socket Intel Xeon Gold 6258R CPU from Cascade Lake microarchitecture family. There are 28 cores per socket and one thread per physical core in the machine. Each core also has its own private L1i, L1d, and L2 caches. The Intel machine runs Linux 5.11.0 and GNU-10.2.1 toolchain. Unless otherwise stated, the default sampling interval is 50K, the default number of debug registers that we use in each core is 4, and the default number of threads in each benchmark is 32 where each thread runs on its own physical core. In each experiment, the threads are distributed evenly across the two sockets, and the threads in each socket are bound to CPU cores with *compact* mapping[30] strategy by default.

We present and analyze the communication matrices of four PARSEC benchmarks, *blackscholes*, *bodytrack*, *streamcluster*, and *freqmine*, generated from the Intel and AMD machines using ComDetective. The sampling period that we use in this experiment is 500K, which is the default sampling period in ComDetective.[31]

Figures 70 and 71 present the communication matrices produced in the two machines with x and y axes show the core ids to which all threads are pinned. As can be seen in the two sets of figures, ComDetective captures the most frequent communication patterns in all benchmarks in both machines. As there is very little to no communication in *blackscholes*, the communication matrices generated by the AMD and Intel versions of the tool are empty. Both tools could also detect one-to-all communication patterns in *bodytrack* and *freqmine*, though the patterns in *bodytrack*'s matrices look more intense as the one-to-all communications happen more frequently in *bodytrack*. There is also a difference between the *freqmine*'s matrices generated in the Intel and the AMD machines. In the Intel machine, the one thread that communicates with all of the other threads is the thread pinned to core 0, while in the AMD machine, that one thread is the

[30] Compact mapping assigns the thread $t+1$ to a free thread context as close as possible to the thread context where the thread t is placed.

[31] Muhammad Aditya Sasongko et al. "ComDetective: A Lightweight Communication Detection Tool for Threads". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver, Colorado: Association for Computing Machinery, 2019. DOI: 10.1145/3295500.3356214. URL: https://doi.org/10.1145/3295500.3356214.

one pinned to core 27. From the matrices of *bodytrack* and *streamcluster*, we could also see that ComDetective detects more communications in the Intel machine than in the AMD machine. This result could be attributed to the fact that Intel PEBS can be programmed to sample only memory loads and memory stores, while AMD IBS introduces randomization and samples any kind of micro-operations. Therefore, under the same sampling period, i.e. 500K, Intel PEBS could get more memory access samples than AMD IBS.

4.1.4 Evaluation: ReuseTracker on AMD vs Intel

Next, we present and analyze the reuse distance histograms of four PARSEC benchmarks running on AMD and Intel machines using ReuseTracker. Figures 72 and 73 display the histograms of reuse distance at private cache level generated in the Intel and AMD machines. Most of the *reuses* in *blackscholes*, *bodytrack*, *streamcluster*, and *freqmine* are shown to be short in distances both in the Intel and the AMD machines. However, there are higher portion of reuses detected in mid-range distances in *bodytrack* and *streamcluster*. The patterns generated from the Intel and AMD machines are nearly the same except those from *freqmine*. There are more mid-range and longer-range spatial reuses detected by ReuseTracker in the AMD machine.

4.1.5 Conclusion

Data movement and locality are important factors that affect the performance of multithreaded applications. We developed COMDETECTIVE+ and REUSETRACKER+, profiling tools that leverage IBS facility on AMD and debug registers to detect inter-thread communications and measure reuse distance in multithreaded code running on AMD machines. In this work, we presented the implementation of the profiling tools and the Linux kernel module needed by the tools to interface with the IBS facility. We also reported an experimental study that evaluates the tools' accuracy, their sensitivity to different sampling intervals and debug register counts, and their overheads. In our experiments, COMDETECTIVE+ displays high accuracy in capturing total communication counts, differentiating true sharing from false sharing, and capturing communication patterns, while REUSETRACKER+ shows high accuracy in measuring reuse distance in private caches with and without cache line invalidations involved. Both tools also exhibit overheads that are lower than cycle accurate simulators and code instrumentation tools.

4.2 Investigating cache partitioning for SpMV on the A64FX processor

Given the preliminary results and methods for cache partitioning presented in Deliverable 1.2, we extend the results by an evaluation and analysis of the effect of cache partitioning in SpMV on the A64FX processor using several real-life matrices from cardiac electrophysiology.

First, we briefly explain cache partitioning on the A64FX. Second, we show the effect of cache partitioning in SpMV with performance results from measurements on the A64FX. Finally, we compare the measurements to predictions obtained from our profiling tool.

Cache partitioning allows dividing a cache into multiple partitions. Fujitsu's A64FX processor is equipped with a way-based hardware cache partitioning mechanism, named *sector cache*.[32] It enables partitioning the L1D and L2 caches and assigning a program's data objects (e.g. arrays) to partitions. The *partitioning policy* (partition sizes and data assignment) can be chosen dynamically at runtime and without flushing the cache. Partition sizes are set by allocating a number of

[32] *A64FX Microarchitecture Manual*. Version 1.5. Fujitsu Limited. 2021. URL: https://github.com/fujitsu/A64FX/blob/master/doc/.

cache ways to partitions (*sectors*). The data assignment is specified on each memory instruction, encoded in the otherwise unused top byte of the virtual address.

4.2.1 Using Fujitsu's C compiler for cache partitioning in SpMV on A64FX

The Fujitsu C Compiler (FCC) provides compiler directives to specify the partitioning policy in application code for the A64FX processor. Listing 3 shows an example of using the directives in SpMV (ELL format). `#pragma scache_isolate_way` specifies the number of cache ways allocated to sector 1. `#pragma scache_isolate_assign` specifies the data objects (arrays or pointers) assigned to sector 1. Other data is assigned to sector 0. Assigning the non-temporal matrix data (`a` and `colidx`) to a partition of minimal size increases the effective cache space and reuse of the vector `x` in this code.

```
#pragma procedure scache_isolate_way     L2=4
#pragma procedure scache_isolate_assign  a colidx
#pragma omp for
    for (int i = 0; i < num_rows; i++) {
        double yi = 0.0;
        for (int j = 0; j < rowsize; j++)
            yi += a[i*rowsize+j] * x[colidx[i*rowsize+j]];
        y[i] += ad[i]*x[i] + yi;
    }
```

Listing 3: SpMV in ELL format using the FCC sector cache compiler directives.

4.2.2 Experimental setup

We run the ELL SpMV code from Listing 3 on the A64FX processor with and without enabling the L2 sector cache using 1, 12, and 48 threads. The A64FX is a 48-core processor with private 64 KiB 4-way L1D caches and four 8 MiB 16-way L2 caches, each shared by 12 cores. Table 6 shows the 7 sparse matrices of increasing size used in this experiment. The matrices are square matrices with 16 non-zeroes per row.

Matrix	Rows, Cols	NNZs	Size [MiB]
hearto1	4,717	53,633	0.61
hearto2	210,101	2,937,795	33.62
hearto3	1,607,708	23,597,002	270.05
hearto4	3,031,704	44,986,514	514.83
hearto5	7,205,076	107,994,304	1,235.90
hearto6	23,595,379	357,427,713	4,090.44
hearto7	55,603,164	846,710,472	9,689.83

Table 6 *Matrices hearto1-hearto7 (square matrices, 16 NNZs per row) .*

4.2.3 Performance results

Table 7 shows the effect on performance and the memory bandwidth utilization using the sector cache. Except matrix *hearto1*, which fits into L2 cache, the sector cache improves performance.

For the larger matrices *hearto3-07*, the speedup that results from enabling the sector cache is about 6 % for a single thread, 13 % to 18 % when using 12 threads, and 15 % to 29 % when using

Matrix	Threads	No Sector Cache GFLOPs	No Sector Cache GiB/s (Mem)	With Sector Cache GFLOPs	With Sector Cache GiB/s (Mem)	Sector Cache Speedup
	1	2.46	0.04	2.11	0.08	0.86
hearto1	12	10.21	0.20	5.35	0.20	0.52
	48	9.23	1.18	3.36	2.98	0.36
	1	1.60	10.57	1.88	10.95	1.18
hearto2	12	15.62	82.10	17.91	73.90	1.15
	48	48.97	176.39	50.59	105.40	1.03
	1	1.40	9.58	1.48	9.88	1.06
hearto3	12	11.80	86.27	13.88	92.08	1.18
	48	41.13	266.62	52.13	274.13	1.27
	1	1.35	9.24	1.43	9.54	1.06
hearto4	12	11.00	83.66	12.97	90.15	1.18
	48	36.35	261.51	46.98	287.02	1.29
	1	1.28	8.90	1.37	9.20	1.06
hearto5	12	10.51	82.37	12.19	87.87	1.16
	48	32.55	251.14	40.87	277.55	1.26
	1	1.19	8.32	1.26	8.59	1.07
hearto6	12	10.09	81.73	11.56	86.29	1.15
	48	37.02	298.90	42.98	310.57	1.16
	1	1.14	8.05	1.21	8.29	1.06
hearto7	12	9.64	79.12	10.88	82.74	1.13
	48	36.04	296.47	41.62	314.04	1.15

Table 7 *Measured performance and memory bandwidth in ELL SpMV on A64FX with and without using the sector cache.*

48 threads. The sector cache also improves memory bandwidth utilization, typically by about 3 % for a single thread, 6 % for 12 threads, and 3 % to 11 % in the case of 48 threads.

4.2.4 CACHE PARTITIONING PROFILING TOOL

We have developed a profiling tool able to model the effect of cache partitioning in programs. The profiling tool uses Intel PIN[33] for dynamic binary instrumentation. It records reuse distance histograms for each of the program's data objects to estimate the number of cache misses in case a data object was assigned to a separate cache partition. The tool is not specifically tailored to the A64FX processor or SpMV programs. However, we can assess the effect of cache partitioning in SpMV on the A64FX using the profiling results.

Limitations There are several limitations to the approach and implementation of the profiling tool:

1. Conflict misses and additional misses due to prefetching or deviations from LRU replacement policy are not modeled because the cache miss model is solely based on reuse distance.
2. The profiling tool can only make predictions for a single (shared) cache.

[33]Chi-Keung Luk et al. "Pin: building customized program analysis tools with dynamic instrumentation". *Acm sigplan notices* 40.6 (2005), pp. 190–200.

Matrix	Threads	A64FX measurement			Tool prediction		
		L2 misses (no SC)	L2 misses (with SC)	Reduction [%]	L2 misses (no SC)	L2 misses (with SC)	Reduction [%]
hearto1	1	17	60	-256.43	2	2	0.00
	12	62	130	-109.43	14	14	0.00
hearto2	1	184,247	167,958	8.84	177,749	164,145	7.65
	12	185,405	166,866	10.00	175,885	164,234	6.62
hearto3	1	1,430,013	1,395,992	2.38	1,402,995	1,359,003	3.14
	12	1,586,047	1,469,304	7.36	1,511,264	1,394,574	7.72
hearto4	1	2,713,609	2,646,205	2.48	2,654,124	2,580,360	2.78
	12	3,079,859	2,851,425	7.42	2,917,656	2,697,232	7.55
hearto5	1	6,517,442	6,327,621	2.91	6,365,396	6,150,540	3.38
	12	7,490,161	6,900,145	7.88	7,077,854	6,554,938	7.39
hearto6	1	21,650,039	20,919,861	3.37	21,084,183	20,248,820	3.96
	12	25,309,557	23,320,884	7.86	23,708,561	21,976,032	7.31
hearto7	1	51,419,229	49,643,523	3.45	50,034,179	47,902,638	4.26
	12	60,698,083	55,946,986	7.83	56,591,793	52,480,861	7.26

Table 8 *Measured and estimated number of cache misses with and without used the sector cache and cache miss reduction.*

3. The tool can only profile x86 binaries because it uses Intel PIN for instrumentation.

Limitation (1) is inherent to reuse distance analysis and cannot be resolved without using a different approach. Limitation (2) can be resolved by extending the profiling tool implementation to allow multiple (shared) caches. Finally, limitation (3) can be resolved by implementing the reuse distance profiling with another method or framework that is compatible with other instruction sets.

Comparison of Measured and Estimated Cache Misses In this section, we show measurements of hardware performance events of L2 cache misses and compare the measurements to the cache miss prediction from our profiling tool. We use the same matrices during profiling, but use only up to 12 threads running in the locality domain of a single shared L2 cache. This is because the profiling tool is unable to make predictions for multiple shared caches yet.

Table 8 shows the measured and estimated number of L2 cache misses without using the sector cache as well as using the partitioning policy from Listing 3. The resulting measured and estimated cache miss reduction is also shown.

Matrix *hearto1* completely fits into L2 cache and the measurement confirms the tool's findings that using the sector cache in this case does not reduce the cache misses. For matrix *hearto2-07*, cache partitioning indeed improves cache behavior as indicated by the profiling tool. The predicted number is consistently below the real value. This is expected because of limitation (1) discussed in Section 4.2.4. However, the prediction for the total number of cache misses as well as the cache miss reduction is fairly accurate. The mean absolute percentage error (MAPE) of the predictions for the L2 cache misses for matrices *hearto2-07* is $\text{MAPE}_{\text{noSC}} = (4.08 \pm 1.73)$ % and $\text{MAPE}_{\text{SC}} = (3.83 \pm 1.57)$ %. For the miss reduction, we obtain $\text{MAPE}_{\text{reduction}} = (14.56 \pm 10.43)$ %.

5 DIGITAL SUPERTWIN

As reported in the previous deliverables, SuperTwin manages several other tools by generating and encoding information in a digital twin data structure called SuperTwin Description (STD). STD enables SuperTwin to make distinct information on distinct systems available to each other and create a linked-data system.

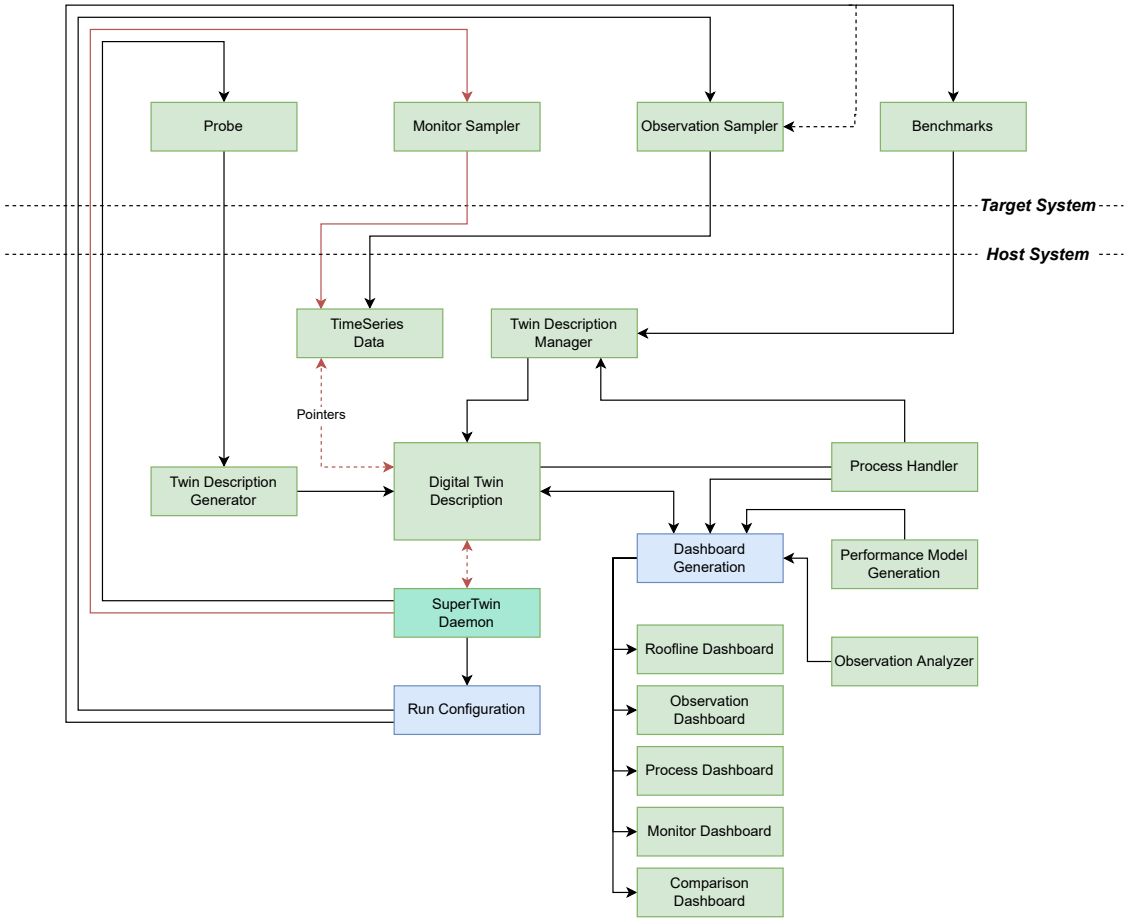


Figure 74 *Current structure for a summary of SuperTwin modules. Amber lines highlight the monitoring pipeline that is always on. Green nodes present functional modules, and blue nodes present configuration modules. Every other functional module on this figure is inherently invoked by the SuperTwin daemon. Monitor sampler is configured to perform low-frequency system status metric readings, that are mostly reported by the operating system, such as CPU and memory utilization, resource usage of individual processes, NUMA statistics, power usage etc. Observation sampler is configured to perform readings for hardware performance events that are reported by the PMUs. Such as cache misses, number and type of floating operations and instructions per cycle.*

5.1 SUPERTWIN DESCRIPTION

Digital Twin Description comprises several classes and relations between them, representing properties and hierarchy in a system. STD both; captures a semantic description of the target

system and enables a linked time-series structure similar to the framework proposed in[34] but with far richer metadata and contemporary metadata instantiations of events. For example, individual components, observations, and processes also have their digital twin descriptions with linked time-series data. This enables a fine-grain analysis of the behavior of applications to run on different systems with different software and hardware. For example, an L1 cache, a network interface, or a process can be isolated from the system, analyzed separately, or compared with its equivalent on a different system.

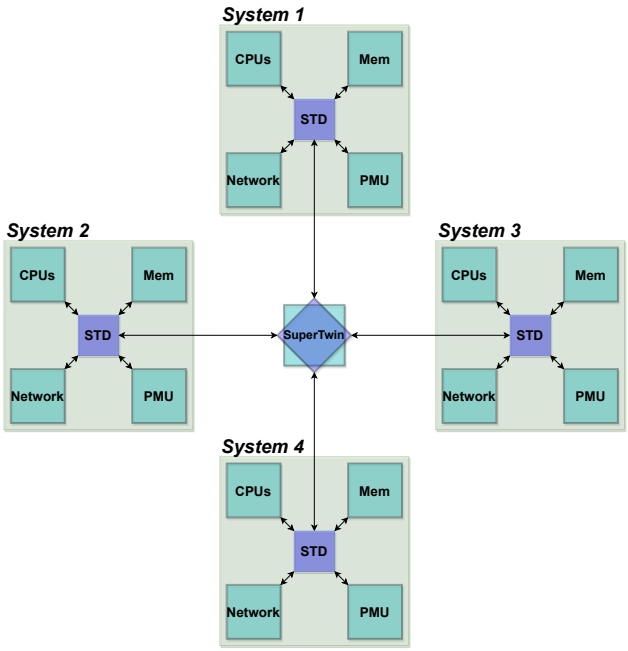


Figure 75 *Linked data approach of SuperTwin. Since the structure of metamodel classes of different STD instantiations for different systems are identical, and every STD instance has the highlighted property of recursive interfaces; every subcomponent of every SuperTwin instance can be isolated from their systems and compared on-the-fly.*

DTDL, as explained in previous deliverables have a recursive structure that allows components (interfaces in the context of twin description) to be other components' subcomponents which is important for describing a cyber-physical system. On top of that, DTDL models `Telemetry`, `Properties` and `Relationship` have exact correspondence between what they describe in DTDL and STD. However, since DTDL is designed with IOT systems in mind, its descriptions are more physical than cyber-physical and are meant to be static. For example, in the context of a smart home, a sensor is statically attached to a wall in a room and reports the same pre-determined metrics constantly. However, in the context of computers, the number of metrics, frequency of readings, and metrics themselves are subject to constant change. To this end, DTDL is modified, and new classes and properties are added to describe high-performance computing systems and create linked time-series data. The update made on DTDL to acquire STD ontology can be seen in Table 9. Among the newly added metamodels, `ObservationInterface` is inherently different from others. To be able to generate structured queries using other metamodels, `ObservationInterface` acts as a registry. For example, when an application is executed and

[34]Friedemann. "Linked Data Architecture for Assistance and Traceability in Smart Manufacturing". *MATEC Web of Conferences* 304 (2019), p. 04006. DOI: 10.1051/matecconf/201930404006.

registered within a `ObservationInterface`, the exact time that execution take place is known via *@id* field, since every reading is recorded to the time-series database with the observation's unique id as tag. Collected performance metric readings of individual threads are accessible via generating queries with *involved_threads* and *sampled_hw_metrics*. A system snapshot during the executions can be taken via a generated query that uses *sampled_sw_metrics*.

Property	*Description*
@type	**Interface**
@id	Unique identifier within digital twin for interface
contents	a set of Interface, Process Interface, ObservationInterface, SWTelemetry, HWTelemetry, Benchmark, Properties, Relationships
displayName	Name to be displayed when instantiated
dashboard	dashboard url, optional
@type	**SWTelemetry**
@id	Unique identifier within digital twin for this telemetry instance
name	index in telemetries
instance	instance name of reported component to be parameter in queries
samplerName	name of the metric to be referred during sampler configuration
DBName	name of the metric to be used in generation of queries
@type	**HWTelemetry**
@id	Unique identifier within digital twin for this telemetry instance
name	index in telemetries
instance	instance name of reported component to be parameter in queries
samplerName	name of the metric to be referred during sampler configuration
DBName	name of the metric to be used in generation of queries
PMUName	name of the metric as reported by libpfm4. To be used as parameter in perf event configuration
@type	**ProcessInterface**
@id	Unique identifier within digital twin for this process
displayName	Name of the process to be displayed when instantiated
PID	Current process ID. This may dynamically change
contents	a set of Properties, Relationships and SWTelemetry
dashboard	url of the dashboard that is generated exclusively for this process, optional
@type	**BenchmarkInterface**
@id	Unique identifier within digital twin for interface
contents	BenchmarkResult
displayName	Name of the benchmark to be displayed when instantiated
dashboard	url of the dashboard for the readings during benchmark, optional
@type	**BenchmarkResult**
@id	Unique identifier within digital twin for this telemetry instance
field	name of field for subkernels, optional
no_threads	number of threads used
involved_threads	involved thread indexes to be used in queries
modifier	modifications in pinning strategy or compilation
result	result of benchmark
unit	unit of benchmark result
sampled_sw_metrics	sampled software metrics during execution, to be used in queries, optional
sampled_hw_metrics	sampled hardware metrics during execution, to be used in queries, optional
dashboard	url of the dashboard that is generated specifically for benchmark field, optional
@type	**ObservationInterface**
@id	Unique identifier within digital twin for interface
displayName	Name to be displayed when instantiated
time	duration of observation
command	executed command
time	duration of observation
no_threads	number of threads used
involved_threads	involved thread indexes to be used in queries
sampled_sw_metrics	sampled software metrics during execution, to be used in queries
sampled_hw_metrics	sampled hardware metrics during execution, to be used in queries
modifier	any modification made to the environment, optionak
dashboard	dashboard url of collected metrics, optional

Table 9 *New metamodel classes added to DTDL to build STD.*

```
def add_my_metrics(component):
    for metric in available_metrics:
        if(component.type == metric.type):
            add_telemetry(component, metric)

def add_component(component, subcomponent):
    add_to_twin(subcomponent)
    add_my_metrics(subcomponent)
    add_ownership(component, subcomponent)

def add_subcomponents(component, subcomponents):
    for socket in system:
        add_component(system, socket)
        for core in socket:
            add_component(socket, core)
            for thread in core:
                add_component(socket, thread)
                for cache in cache_groups[thread]:
                    add_component(thread, cache)

def add_agents(component, subcomponent):
    for agent in pcp:
        resolve_process_state(agent)
        add_component(system, agent)

def generate_twin_description(system_probing):
    system = create_system()
    add_subcomponents(system, cpus)
    add_component(system, memory)
    add_component(system, disks)
    add_component(system, networks)
    add_component(system, gpus)
    add_component(system, proc)
    add_agents(system, pcp)
```

Listing 4: STD is generated via both contextual and structural information probed from the target system. The processes and framework component could also be represented in the STD. By leveraging resolve_process_state() method processes can be reinstated on-the-fly. Since the only change in their respective digital twins will be their PIDs; only the configuration of the sampling module will be affected and queries that will later be used to access readings will remain the same.

After acquiring system information via probing, STD is generated using this information. During the generation of STD, every single physical component that performs computations, communications, or I/O operations is presented with an `Interface`. Every hierarchical relationship between these components is encoded into the contents of these interfaces with a `Relationship` entry. Available metrics from these components are also filtered and encoded via `SWTelemetry` and `HWTelemetry`. Therefore, both precisely pinned executions and advanced semantic queries were made available. Those interfaces later use values to configure samplers and locate their values in the database. For example, using generated STD and run configuration module, an execution that will run on 8 threads on each socket that does not share an L1 cache can be launched; similarly, after the execution of a run, performance metrics from threads that share same L2 cache with a given thread can be queried.

Another unique contribution of SuperTwin is that processes also can be modeled as digital

twins and monitored via per-process kernel metrics as in Listing 4. JSON-LD objects are serialized, which means string values from the JSON object are instantiated with given parameters into a run-time object. In SuperTwin, there are two degrees of serialization. All models other than `ProcessInterface` are serialized and got their values assigned at the generation time; however, `ProcessInterface` is re-instantiated every time they are invoked and their metadata is changed due to the dynamic nature of processes.

```
def ConstructTwin(IP, user, password):
  name, prob_file = remote_probe(IP, user, password)
  mongodb_addr, influxdb_addr, grafana_addr = read_environment()
  influx_name, mongodb_name, grafana_name = create_datasources()
  mongodb_id = insert_twin_description(generate_twin_description(
    ↪ probe_file)) #Method in Listing 1
  #From this moment, SuperTwin description and object is available to
    ↪ interact
  SuperTwin.add_cache_aware_roofline_benchmark()
  SuperTwin.monitor_metrics = read_from_environment()
  SuperTwin.observation_metrics = read_from_environment()
  SuperTwin.monitor_pid = start_sampling()
  SuperTwin.generate_monitoring_dashboard()
  SuperTwin.configure_observation_events(observation_metrics)
  SuperTwin.observation(add_stream_benchmark())
  SuperTwin.observation(add_hpcg_benchmark())
  SuperTwin.generate_roofline_dashboard()
  SuperTwin.register_state(SuperTwin.run_time_variables)


def ReConstructTwin(IP) # or ReConstructTwin(Name):
  db_id = db_lookup(IP) # or Name
  SuperTwin = reconstruct(db[db_id][twin_description], db[db_id][
    ↪ run_time_variables])
```

Listing 5: Instatiation of a SuperTwin object from scratch

A SuperTwin object in Python uses generated STD as its main look-up table, for generating configurations for third-party frameworks and generating queries for visualization and analyses. After the generation of STD, every other SuperTwin module is callable and uses inherently STD for their operations and alters the STD, encoding information through time. SuperTwin object later can be re-constructed using STD and other run-time variables that are updated in the database throughout the execution.

5.2 SAMPLING FRAMEWORK

Performance Co-Pilot has metric samplers responsible for a metric domain. PMDAs are installed beforehand the sampling takes place, but they do not report values on their own. To sample metrics, a monitor framework needs to send requests to the target system PMCD. SuperTwin uses `pcp2influxdb` as a monitor framework. The monitor framework, however, has a configuration that specifies metrics to be collected, instance domains of these metrics, frequency of the sampling, database address, and bucket to write metrics. Software metrics can be sampled by querying their parameters from SuperTwin and running `pcp2influxdb` with a generated configuration file. However, `perfevent` PMDA must be re-configured every time requested metrics are changed to set PMUs report requested metrics. After probing the target system and acquiring MSRs and available events, parameters required to re-configure a remote PMU can be queried from STD.

SuperTwin also employs a `perf` reconfiguration module that queries this data from STD and reconfigures remote PMU automatically if a change in hardware telemetry in requested metrics is detected. After the reconfiguration, hardware telemetry can be sampled and recorded for corresponding observation. This process can be seen in Figure 76.

5.3 MONITORING

Software and hardware telemetry differ in their meaning and effect on the execution performance. Hardware metrics are direct measurements of performance events that took place within the CPU and are directly related to performance and could give definite reasoning for observed performance. For example, an exceptionally high L1 miss rate is thought to be primarily responsible for low performance and may not be resolved without source code optimization. On the other hand, software metrics do not give a cause directly related to the application but provide a picture of the system state during execution and could reveal system-related anomalies such as resource contention, thermal throttle, memory leak, or poor affinity. Therefore, in SuperTwin, software, and hardware telemetry are separated. This also allows the sampling of software and hardware metrics with different frequencies. This is called the monitoring part of profiling in the SuperTwin context. Monitoring data can be used to model the remote system as a whole, predict possible faults and mitigate them before they happen. A configuration for monitoring is generated just after the generation of STD and monitoring starts.

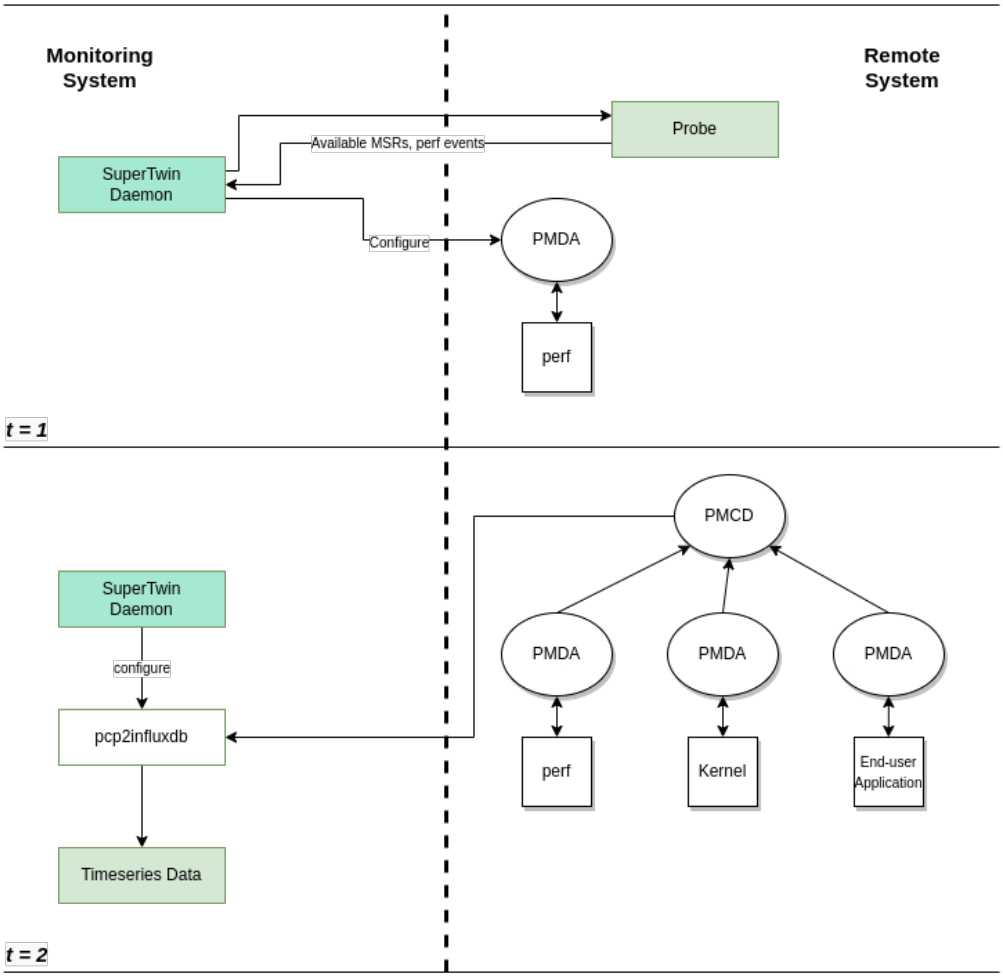


Figure 76 *To sample hardware metrics with* `perfevent` *PMDA. PMUs are re-configured beforehand the observation takes place.*

5.4 OBSERVATION

When kernels are executed with direct measurements of performance events, SuperTwin uses shell scripts as function wrappers. Whenever there is a need to set a directory, affinity, environment variables, or execute a binary, the Observation module, using the Run Configuration module as a helper tool, generates a shell script with instructions to execute, copies it to the remote system, and executes it. An example request and resulting script can be seen in Listings 6 and 7.

```
1 #!bin/bash
2
3 cd /home/sparcity/eu
4 likwid-pin S0:0-3,22-25@S1:0-3,22-25 ./mkl_spmv mixtank_old.mtx
```

Listing 6: Bash script generated to execute kernel at desired path alongside with affinity.

```
def observation_sampling(SuperTwin):
    config_lines = get_lines(SuperTwin.db_addr,
                             SuperTwin.db_name,
                             SuperTwin.hw_events)
    config_file = generate_configuration(config_lines)
    return config_file

def start_sampling(config_file):
    execute_local(pcp2influxdb -c config_file)
    return process

def observation(SuperTwin, path, command, input, threads):

    config_file = observation_sampling(SuperTwin)
    affinity = generate_binding(threads, "numa compact")
    bash_file = generate_bash_file(path, command, input, affinity)

    copy_to_remote(bash_file)
    sampler = start_sampling
    execute_remote("bash /tmp/st_files/gen_bash.sh")
    sampler.kill()
```

Listing 7: Pseudocode for executing observation at remote host with sampling using SuperTwin.

SuperTwin allows the profiling of any command executed on the system; this property also allows new frameworks to be easily integrated into the SuperTwin via compiled executables.

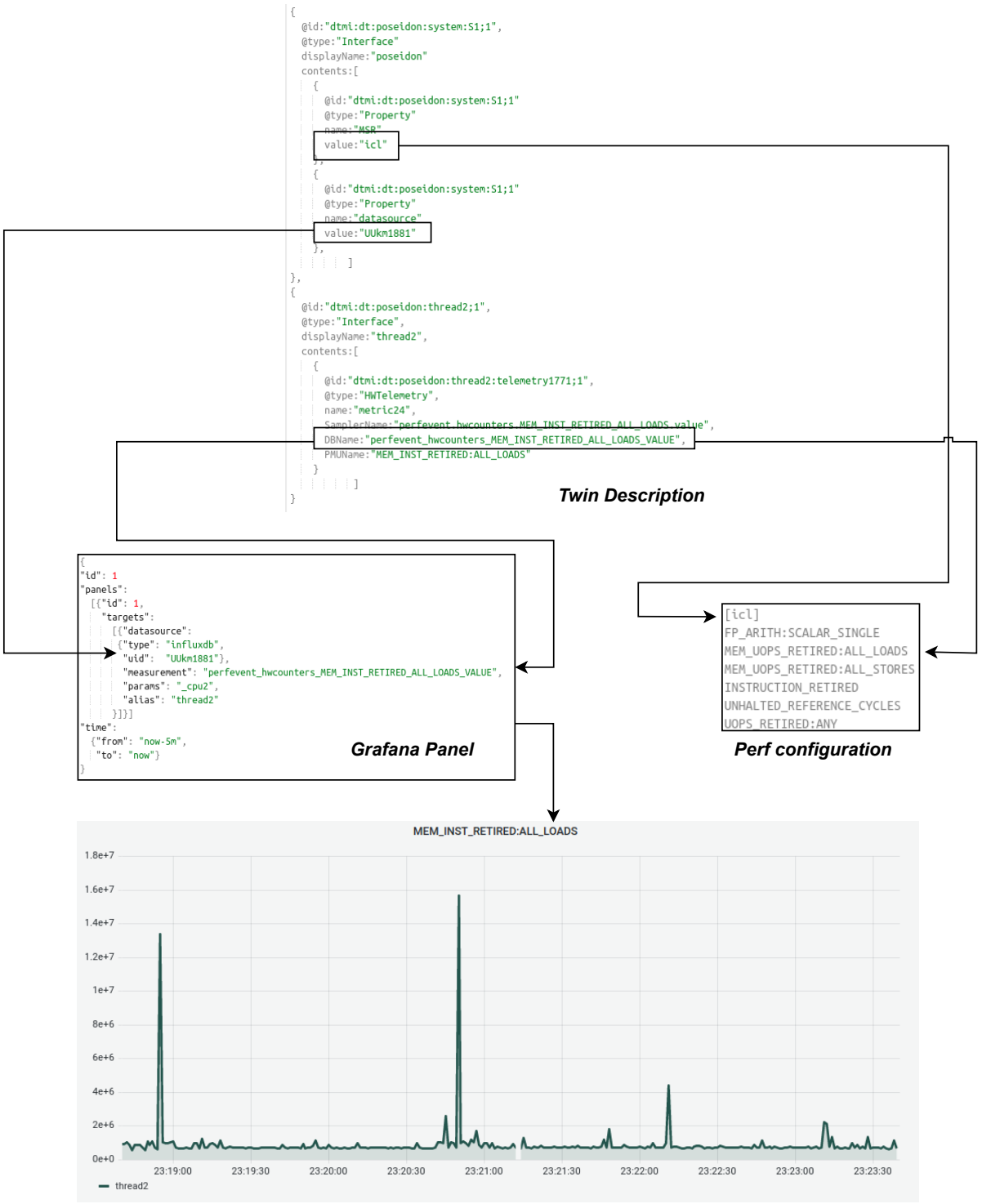


Figure 77 *A summarized version of SuperTwin dashboard generation pipeline. STD is generated with all components having their metrics, and their metrics representation in different frameworks as content. Structured queries then capture these values, create configuration files and run tools.*

Figure 78 *Generated Monitor dashboard for host Dolap. In the socket panels, threads sharing the same L1 cache are plotted consecutively, leveraging STD. At the time of the screenshot, a computation that is just launched in NUMA socket 0 but anomalously allocates memory from NUMA socket 1.*

5.5 GENERATION OF DASHBOARDS

SuperTwin can generate several types of dashboards on the fly after the STD is generated. SuperTwin's dashboard module exploits the fact that Grafana dashboards are serialized JSON files and easily generate Grafana queries, using parameters stored in STD interfaces. Generated dashboards are later uploaded to the local Grafana server and their addresses are encoded in the corresponding interface entry in STD. A brief example of JSON generation using STD can be seen in Listing 77.

Since the metadata and benchmark results are stored in STD, and the generation of performance models, dashboard panels, and dashboards are functions of SuperTwin, generated performance models and charts can be recalled at any time after the STD is created, any new observation is ready for cross-comparison as long as they have mutual metrics with previous observations.

5.6 BENCHMARKS

On top of CARM, widely used benchmarks STREAM and HPCG are also copied to the target system using architecture optimizations and the most recent versions. These benchmarks' makefiles are configured w.r.t. available maximum vector extension capabilities in the target system and compiled in place in order to ensure system performance is ideally measured. While taking measurements, the probed system topology is also taken into account and used in the generation of tailored scripts for different multi-threading and NUMA affinity settings. Benchmarks are counted as a part of probing but at the same time interpreted just as any other observations. Therefore monitoring metrics and observation metrics are also sampled during the execution of benchmarks and made available for future comparisons. Benchmarks are encoded in STD with

Figure 79 *Performance model dashboard generated by SuperTwin showing CARM model generated for threads that are multiples of two plus cores per socket, threads per socket, and total threads, along with STREAM and HPCG multicore scaling and architecture information. This information is also readily available and comparable to any other observation made by SuperTwin. Inconsistent results in STREAM and HPCG benchmarks are due to uncompleted feature of NUMA pinning for benchmarks.*

Figure 80 *Comparison dashboard generated with SuperTwin. After the execution of several distinct events at distinct times, metric timestamps are overlapped and presented to reveal common program phases in different settings. Augmented statistics are also provided alongside time-series data. In this example, linked data is used to make information from different events on the same host available to each other.*

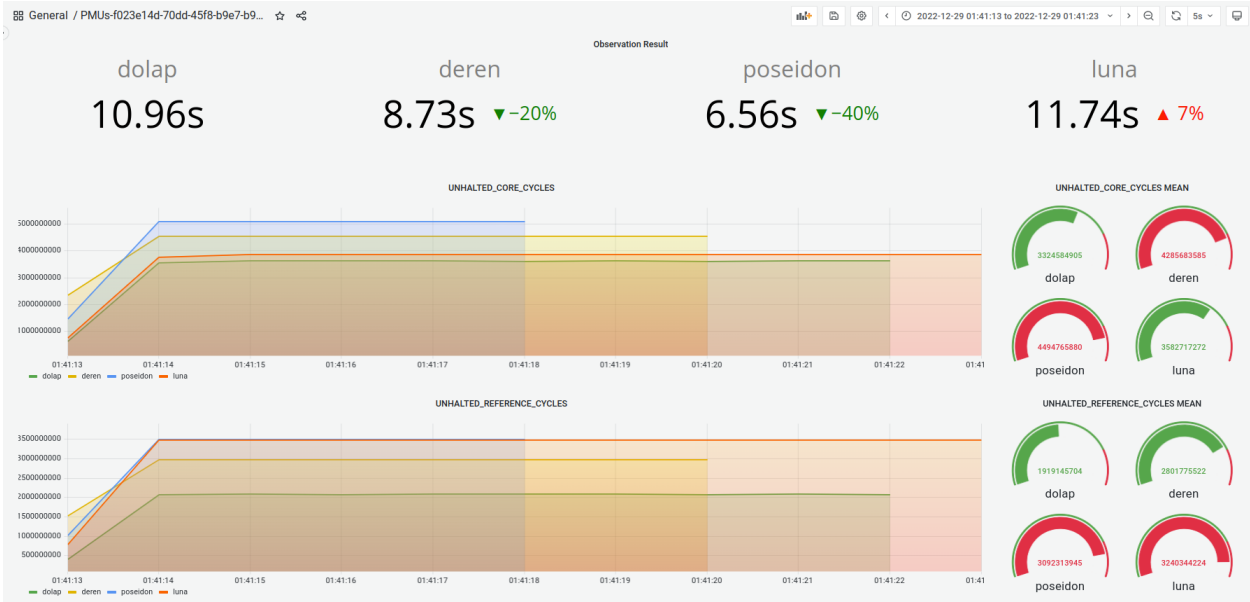


Figure 81 *Comparison dashboard generated with SuperTwin. After the execution of several distinct events at distinct times, and on different systems, metric timestamps are overlapped and presented to reveal common program phases in different settings. In this example, linked data is used to make information from different events on different hosts available to each other, as shown in Figure 75*

dedicated `Benchmark` and `Benchmark_Result` models to facilitate semantic queries. Benchmark entries for STD can be seen in Table9. An example encoding of a STREAM benchmark result with different multi-threading settings can be seen in Listing 8.

```
{
  {dtmi:dt:dolap:system:S1;1:
    {@type: "Interface",
     @id: "dtmi:dt:dolap:system:S1;1"
     @contents:
       [{@id: "dtmi:dt:dolap:benchmark:B1;1"
         @type: "benchmark",
         @name: "STREAM",
         @contents:
           [{@id: "dtmi:dt:dolap:benchmark_res:B1;1",
             @type: "benchmark result",
             @field: "triad"
             @threads: 1,
             @modifier: "likwid-pin -c 0",
             @result: 12816.9,
             @unit: "MB/s"},
            {@id: "dtmi:dt:dolap:benchmark_res:B2;1",
             @type: "benchmark result",
             @field: "triad"
             @threads: 2,
             @modifier: "likwid-pin -c 0-1",
             @result: 25071.5,
             @unit: "MB/s"}]
        }]
    }
  }
}
```

Listing 8: Benchmark results encoded in twin description

5.7 EVALUATION OF SUPERTWIN READINGS VIA PERFORMANCE CO-PILOT

SuperTwin aims to present and monitor every software and hardware component, with statistics of the past executions on a target system. It creates linked data, performs semantic queries, and generates live and historical dashboards and analyses. However, measurements need to be performed for all of the former to be meaningful, accurate, and lightweight. Measuring the performance of a system creates extra work to perform the measurement, affecting the correctness of measurements or, worse, decreasing the performance of the measuring system and/or execution. Due to this, a measurement on a target system should be as lightweight as possible and made sure not to affect the measured events.

SuperTwin performs performance measurements on target systems via Performance Co-Pilot which is developed by RedHat. To prove Performance Co-Pilot's suitability to SuperTwin use cases, an in-depth analysis of Performance Co-Pilot's performance, correctness, and effect on the target system, using SuperTwin configurations, is performed. To provide a complete picture of SuperTwin scenarios, a comprehensive analysis including system resource usage, remote report efficiency, the maximum resolution of monitor/performance events, correctness, and overhead of the Observation events are studied.

Dolap		Deren	
OS	Ubuntu 20.04.3 LTS x86_64	**OS**	Ubuntu 22.04.1 LTS x86_64
Kernel	5.4.0-135-generic	**Kernel**	5.15.0-47-generic
CPU	Intel Xeon Gold 6152 @3.7GHz x2 (44c/88t)	**CPU**	Intel i7-9700F @4.7GHz (8c/8t)
MSR	skx	**MSR**	skl
Mem.	1TB DDR4 @ 2666MHz	**Mem.**	64GB DDR4 @ 2133MHz
Env.	pcp 5.3.7-1	**Env.**	pcp 5.3.6-1
Poseidon		**Luna**	
OS	Ubuntu 20.04.4 LTS x86_64	**OS**	Ubuntu 18.04.6 LTS x86_64
Kernel	5.15.0-56-generic	**Kernel**	5.4.0-135-generic
CPU	Intel i9-11900K @5.1GHz (8c/16t)	**CPU**	Intel Xeon E5-1650 v2 @3.9GHz (6c/12t)
MSR	icl	**MSR**	ivb_ep
Mem.	16GB DDR4 @2666MHz	**Mem.**	16GB DDR3 @1866MHz
Env.	pcp 6.0.1-1	**Env.**	pcp 4.0.1-1

Table 10 *System specifications of hosts used in experiments.*

To provide a wider depiction, increase the confidence interval for the results, and assure the previously mentioned flexibility of SuperTwin, a test set including reasonably different systems, all different in capabilities with different MSRs, is used. `Dolap` is a recent and remarkably powerful high end server with 2 CPUs, 88 threads and 1TB of RAM. `Poseidon` is a performant and recent server, `Deren` is an upper-middle tier desktop for general use which have the desktop version of `Dolap` MSR. `Luna` is a 10-year-old and fairly weak machine included in a test set to analyze consistency in extreme cases. The specifications of the host machines used are summarized in Table 10.

5.7.1 Resource use of sampling

Since PCP employs several agents who collectively perform metric shipment operations, resource usage on the remote system may become overwhelming with the increasing number of sampled metrics and resolutions. To this end, CPU and memory usage of individual PCP agents that are used by SuperTwin are measured for the different number of sampled metrics and sampling frequencies. Measurements are performed for 10 minutes while the target systems are empty, and results are averaged. Results for sampling 50 metrics with varying frequencies are given in Figures 82, 83, 84, and 85 for Dolap, Deren, Poseidon, and Luna, respectively. The network is monitored as a whole for each system. I/O use of PCP agents was found to be negligible (<1 KB). Therefore, they are not included in the results. During the measurements, the host system had a 100 Mbit cabled connection with each system. The host system's disk performance was measured at 182 KB/s, and 1.2 MB/s for 512B and 8K block-sized writes, respectively.

For recall, from PCP agents, `pmcd` manages other agents and reports their readings to remote requesters. `perfevent` samples PMU readings via Linux `perf` interface, `pmdalinux` reports software sourced system state metrics such as CPU load, `pmdaproc` reports per process metrics, such as io and memory usage for each process on the system. Measurements for CPU usage are made using `proc.psinfo.utime` and `proc.psinfo.stime` metrics, for memory `proc.psinfo.rss` metric. The first observation that can be made instantly is, apart from the number of reported metrics or frequency of sampling, all agents are found to use a constant amount of memory. Higher memory usage of `pmdaproc` is due to the size of a much bigger instance domain. Other higher usages of system resources in Dolap, albeit having much more powerful component composition, is also due to much bigger instance domains in Dolap. For example, a `pmdaperfevent` metric has 8 instances in Deren while the same metric has 88 instances on Dolap. Similarly, a much higher number of running processes and system components results in higher system resource uses for Dolap. Apart from `pmdaproc`, all agents are found to be thrifty in system usage resources. These measurements were made without filtering on instance domains; instance domains can be

filtered to reduce thousands of instances to a couple of instances of interest. Also, the monitoring framework of SuperTwin uses no per-process metrics and uses ≈20 `pmdalinux` metrics and ≈2 `pmdaperfevent` metrics at 1 second intervals without user configuration.

An interesting observation is that even though Dolap has more processes (therefore instances in `pmdaproc` instance domain) than other baseline systems, `pmdaproc` uses slightly lower memory w.r.t. Poseidon and Deren and the memory consumption is similar to that of Luna. This can be due to both servers bearing Xeon CPUs, but it requires further investigation.

5.7.2 THROUGHPUT AND INTEGRITY OF REPORTED METRICS

PCP agents and network usage are scaled almost linearly for increased sampling frequency. They use resources consistently, as almost no deviation was observed during measurements, as seen in the error bars. This proper scaling also exists with metrics. Every system in the experiment set, except for Luna, scales proportionally when the number of collected data points is increased. On Luna, the reason for poor scalability is most probably due to a constant overhead for running the framework, since there is not much reporting loss, which will be explained later. However, one case in the test set hints that the PCP framework does not scale perfectly. In Figure 82, there is almost no difference in 4 and 8 reports per second, and the network traffic varies during measurement contrary to the rest of the entire test set. This behavior is also observed in other Dolap measurements with the exception of 10 metrics.

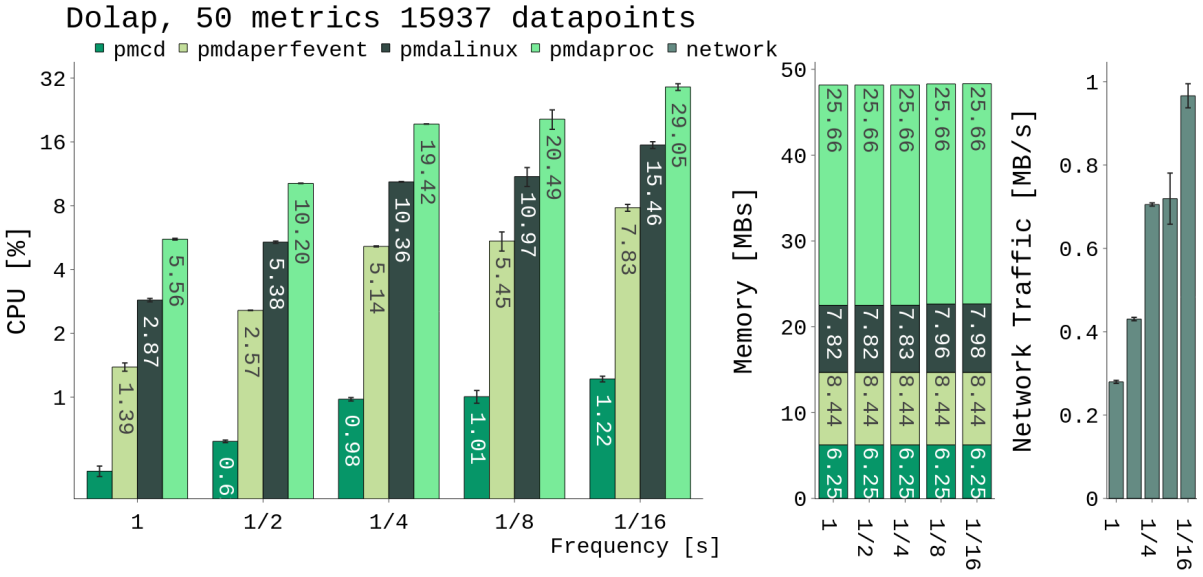


Figure 82 *System resource usage of metric shipment with kernel and PMU metrics on Dolap. Metric agents `pmdaperfevent`, `pmdalinux`, and `pmdaproc` report 24, 20, 6 metrics and 2112, 285, 13572 data points, respectively.*

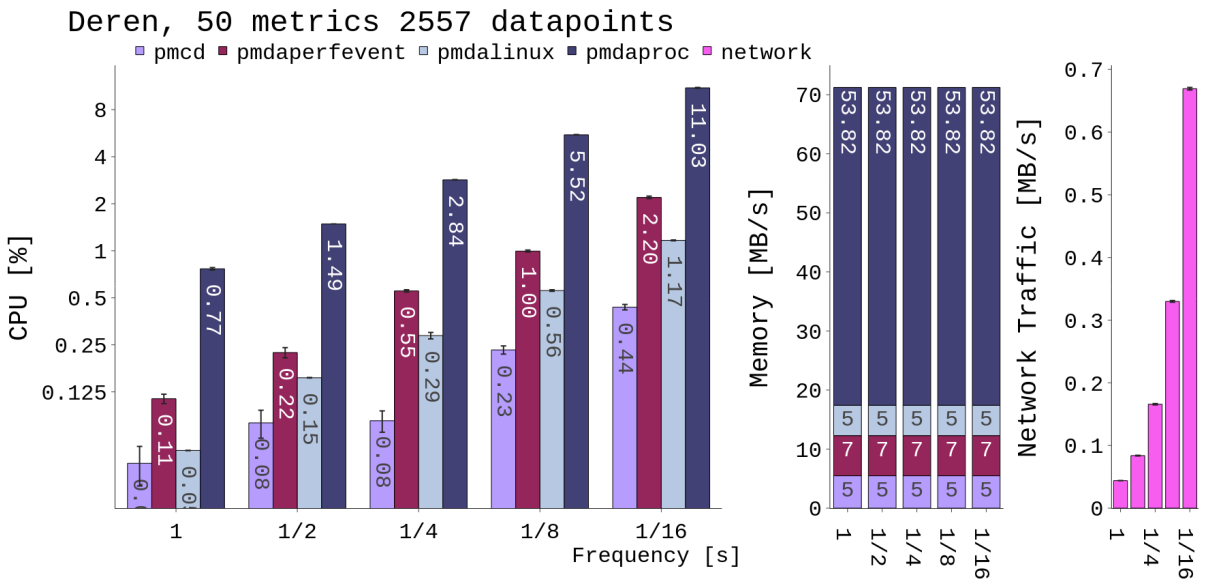


Figure 83 *System resource usage of metric shipment with both kernel and PMU metrics on Deren. Metric agents* `pmdaperfevent`*,* `pmdalinux` *and* `pmdaproc` *report 24, 20, 6 metrics and 192, 40, 2325 data points, respectively.*

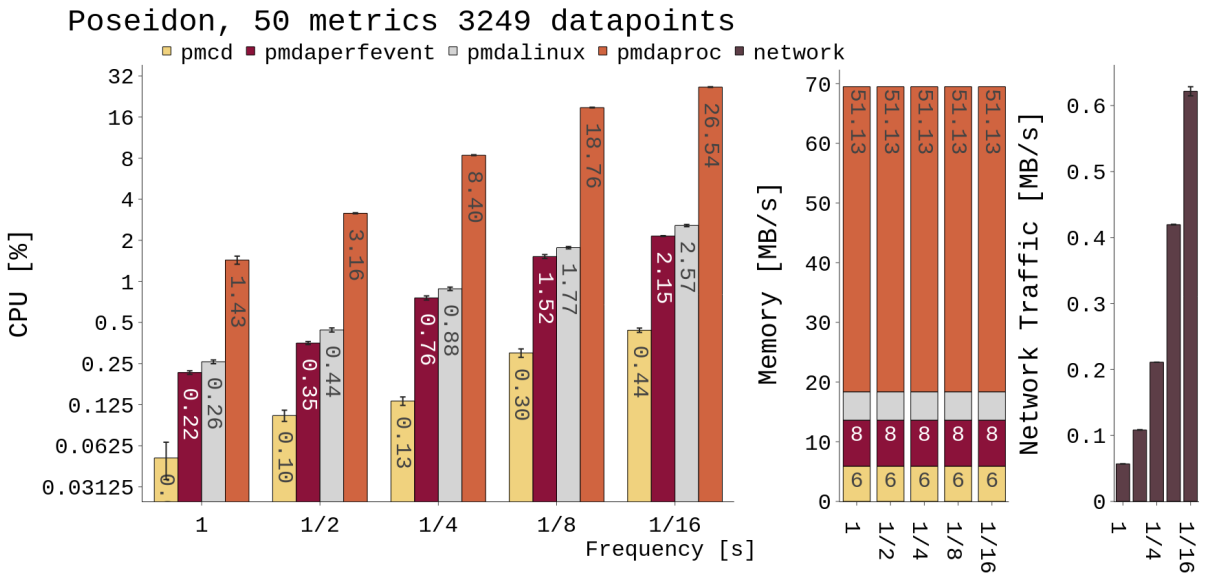


Figure 84 *System resource usage of metric shipment with both kernel and PMU metrics on Poseidon. Metric agents* `pmdaperfevent`*,* `pmdalinux`*, and* `pmdaproc` *report 24, 20, 6 metrics and 384, 60, and 2805 data points, respectively.*

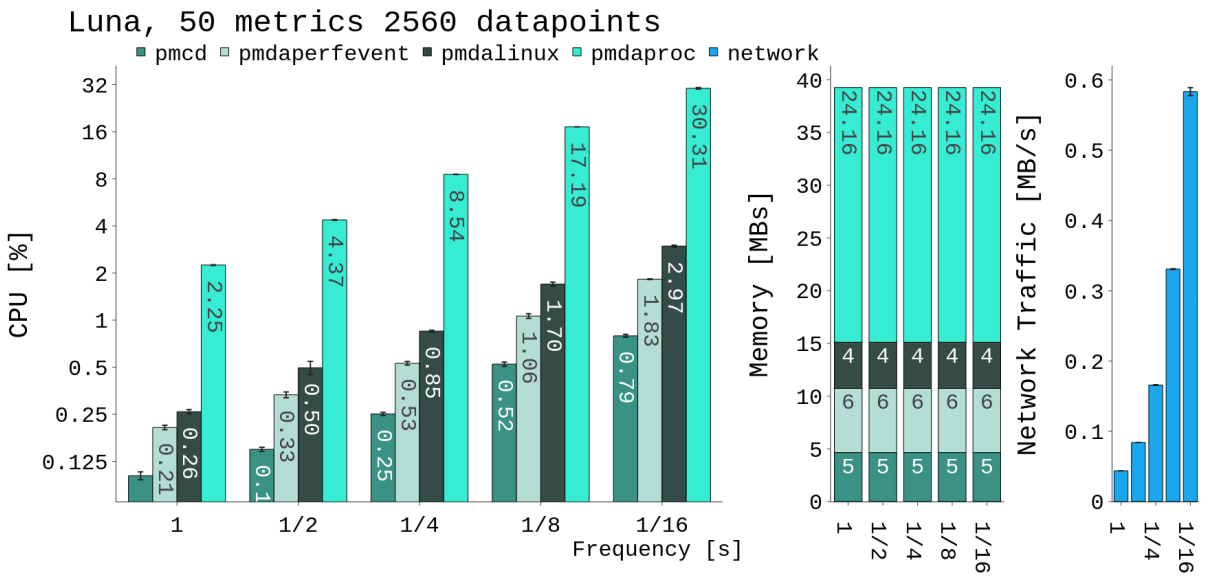


Figure 85 *System resource usage of metric shipment with both kernel and PMU metrics on Luna. Metric agents* `pmdaperfevent`*,* `pmdalinux`*, and* `pmdaproc` *report 24, 20, 6 metrics and 288, 52, 2205 data points, respectively.*

The under-utilization of the network, together with the non-linear increases in CPU usage, suggests that the framework is stalled and either struggles to sample or reports the performance metrics with the desired frequency. This is possible since communication is over the network, and there is no mechanism to buffer and resend missing metrics once more. Due to high frequency, at the time of re-sending, missing metrics are outdated by hundreds or thousands of new reports.

To further investigate this situation, we performed high-frequency readings and measured the actually reported data points and the ratio of loss with `procpmda` and `perfeventpmda`. On top of the missing values, we observed batched zero values in our database with very high frequencies of samplings. Since `procpmda` reports for all processes, and there will be many correct zero values, we measured the wrong zero values with `pmdaperfevent`. With `perfevent`, we sampled metrics that are highly unlikely to report zero; `UNHALTED_CORE_CYCLES`, `INSTRUCTION_RETIRED`, `UOPS_DISPATCHED` etc. Then, we count the number of zeros in the database after the measurement is completed.

Table 11 reports throughput of `procpmda`. On Dolap, there are ≈1100 processes running while the server is empty; therefore, `pmdaproc` has ≈1100 instances per metric. On Deren, there are ≈400 processes running while the server is empty. On Dolap, a loss jump was observed after 8 reports per second when the number of individual data points exceeds 30K per second, and another considerable jump is observed with 16 reports per second. After this point, despite slight increases with increasing demand, losses also increased, and reported individual data points remained between 30K and 40K data points per second. On Deren, losses exhibited a similar jump after 30K data points with 16 reports per second. Although 48K data points per second are achieved, with increasing frequency, losses are also increased, and maximum throughput remains around ≈40K data points. It is concluded that losses are affected by both the frequency and number of instances and reports that include fewer data points are slightly less prone to losses. The maximum throughput of `procpmda` is around 30K-40K data points per second, despite being subject to small changes w.r.t. host and number of instances.

The throughput achieved with `pmdaperfevent` can be seen in Table 12. Instead of sampling

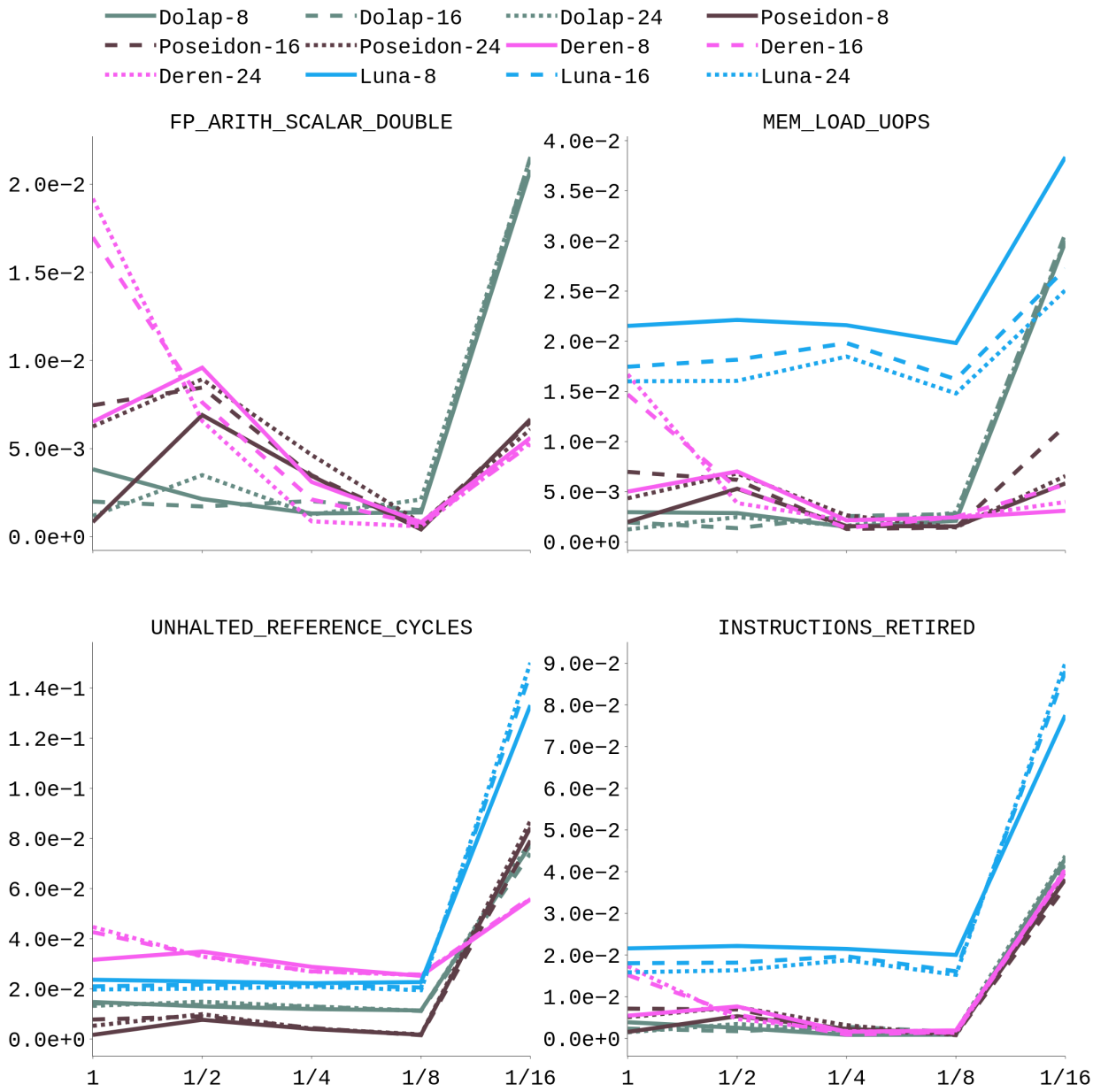


Figure 86 *Accuracy (y-axis) in terms of relative error of 4 different events counted and compared against values reported by* `likwid-bench` *kernels* `triad`, `stream`, `sum`, `peakflops`, `ddot`, `daxpy` *on 4 different systems. Calculated individual errors are further averaged into a single value. The x-axis shows the sampled values per second.*

Host	Frequency	# of metrics	Expected	Inserted	% Loss	Throughput
Dolap	2	4	8.56E+04	8.45E+04	**1.3**	8447.8
		5	1.07E+05	1.06E+05	**1.0**	10592.6
		6	1.29E+05	1.28E+05	**0.8**	12802.3
	4	4	1.72E+05	1.68E+05	**2.4**	16793.7
		5	2.13E+05	2.09E+05	**2.3**	20860.3
		6	2.57E+05	2.50E+05	**2.5**	25040.7
	8	4	3.49E+05	3.05E+05	**12.5**	30536.0
		5	4.22E+05	3.61E+05	**14.5**	36084.8
		6	5.16E+05	3.79E+05	**26.5**	37909.5
	16	4	6.92E+05	3.19E+05	**53.9**	31917.6
		5	8.45E+05	3.38E+05	**60.0**	33787.5
		6	1.03E+06	3.85E+05	**62.8**	38466.1
	32	4	1.37E+06	3.14E+05	**77.2**	31368.6
		5	1.69E+06	3.63E+05	**78.5**	36321.7
		6	2.07E+06	3.90E+05	**81.2**	38962.9
	64	4	2.72E+06	3.01E+05	**88.9**	30109.7
		5	3.38E+06	3.62E+05	**89.3**	36213.2
		6	4.11E+06	3.78E+05	**90.8**	37783.2
Host	*Frequency*	*# of metrics*	*Expected*	*Inserted*	*% Loss*	*Throughput*
Deren	2	4	3.16E+04	3.14E+04	**0.4**	3144.2
		5	3.99E+04	3.97E+04	**0.4**	3974.0
		6	4.76E+04	4.76E+04	**0.0**	4761.6
	4	4	6.32E+04	6.21E+04	**1.7**	6209.4
		5	7.98E+04	7.91E+04	**0.9**	7914.5
		6	9.54E+04	9.27E+04	**2.8**	9274.9
	8	4	1.26E+05	1.24E+05	**2.3**	12352.2
		5	1.60E+05	1.57E+05	**2.0**	15672.2
		6	1.91E+05	1.88E+05	**1.5**	18829.7
	16	4	2.54E+05	2.49E+05	**2.0**	24871.5
		5	3.18E+05	3.11E+05	**2.1**	31147.7
		6	3.82E+05	3.14E+05	**18.0**	31364.0
	32	4	5.08E+05	4.18E+05	**17.6**	41844.4
		5	6.35E+05	4.81E+05	**24.3**	48116.4
		6	7.64E+05	4.56E+05	**40.4**	45579.0
	64	4	1.02E+06	3.95E+05	**61.1**	39483.4
		5	1.26E+06	4.07E+05	**67.7**	40738.7
		6	1.53E+06	3.84E+05	**74.9**	38367.2

Table 11 *Number of data points expected and observed at the host database w.r.t. the number of metrics and sampling frequency. Throughput is inserted datapoints per second.*

and reporting of operating system files, `pmdaperfevent` samples PMUs, another bottleneck for maximum throughput. Therefore, we expect a lower value than `pmdaproc`. Similar to `pmdaproc`, with 16 reports per second, a massive jump in losses is observed with both systems, with the contribution of false zeros. Furthermore, the loss amount is correlated with the size of the instance domain. While being much more significant in Dolap, an increase in batch zeros and losses with correspondence with `pmdaproc` are observed. This further strengthens the previous conclusion that larger numbers of instances both in reports and high frequency are effective in losses.

Host	*Freq.*	*metrics*	*Expected*	*Inserted*	*Zeros*	*% Loss*	***% L+Zeros***	*Throughput*	*A. Throughput*
Dolap	2	4	7.04E+03	6.62E+03	0.00E+00	6.0	**6.0**	661.8	661.8
		5	8.80E+03	8.71E+03	0.00E+00	1.0	**1.0**	871.2	871.2
		6	1.06E+04	1.06E+04	0.00E+00	0.0	**0.0**	1056.0	1056.0
	4	4	1.41E+04	1.31E+04	2.22E+02	7.0	**8.6**	1309.4	1287.2
		5	1.76E+04	1.76E+04	0.00E+00	0.0	**0.0**	1760.0	1760.0
		6	2.11E+04	2.05E+04	0.00E+00	3.0	**3.0**	2048.6	2048.6
	8	4	2.82E+04	2.60E+04	5.84E+02	7.8	**9.8**	2597.8	2539.4
		5	3.52E+04	3.42E+04	7.72E+01	2.8	**3.0**	3423.2	3415.5
		6	4.22E+04	4.22E+04	0.00E+00	0.0	**0.0**	4224.0	4224.0
	16	4	5.63E+04	4.49E+04	1.34E+04	20.3	**44.0**	4491.5	3151.5
		5	7.04E+04	6.88E+04	1.73E+04	2.3	**26.8**	6881.6	5155.0
		6	8.45E+04	8.25E+04	2.00E+04	2.4	**26.0**	8247.4	6248.5
	32	4	1.13E+05	6.97E+04	3.04E+04	38.1	**65.1**	6969.6	3927.9
		5	1.41E+05	1.14E+05	5.32E+04	19.4	**57.2**	11352.0	6030.3
		6	1.69E+05	1.20E+05	5.02E+04	28.8	**58.5**	12027.8	7012.1
	64	4	2.25E+05	3.57E+04	1.61E+04	84.2	**91.3**	3569.3	1962.6
		5	2.82E+05	1.14E+05	5.32E+04	59.6	**78.5**	11387.2	6063.7
		6	3.38E+05	1.31E+05	5.91E+04	61.3	**78.8**	13073.3	7163.6
Host	*Freq.*	*metrics*	*Expected*	*Inserted*	*Zeros*	*% Loss*	***% L+Zeros***	*Throughput*	*A. Throughput*
Deren	2	4	6.40E+02	6.40E+02	0.00E+00	0.0	**0.0**	64.0	64.0
		5	8.00E+02	8.00E+02	0.00E+00	0.0	**0.0**	80.0	80.0
		6	9.60E+02	9.60E+02	0.00E+00	0.0	**0.0**	96.0	96.0
	4	4	1.28E+03	1.24E+03	0.00E+00	3.0	**3.0**	124.2	124.2
		5	1.60E+03	1.53E+03	0.00E+00	4.5	**4.5**	152.8	152.8
		6	1.92E+03	1.83E+03	0.00E+00	4.5	**4.5**	183.4	183.4
	8	4	2.56E+03	2.49E+03	0.00E+00	2.8	**2.8**	249.0	249.0
		5	3.20E+03	3.10E+03	1.60E+00	3.0	**3.1**	310.4	310.2
		6	3.84E+03	3.72E+03	0.00E+00	3.0	**3.0**	372.5	372.5
	16	4	5.12E+03	5.00E+03	4.61E+02	2.3	**11.3**	500.5	454.4
		5	6.40E+03	6.40E+03	5.27E+02	0.0	**8.2**	640.0	587.3
		6	7.68E+03	7.48E+03	5.89E+02	2.6	**10.3**	747.8	689.0
	32	4	1.02E+04	9.67E+03	3.69E+03	5.6	**41.6**	967.0	597.8
		5	1.28E+04	1.25E+04	4.67E+03	2.6	**39.1**	1246.4	779.9
		6	1.54E+04	1.49E+04	5.50E+03	2.9	**38.7**	1490.9	940.8
	64	4	2.05E+04	1.99E+04	1.05E+04	2.8	**54.3**	1991.0	936.2
		5	2.56E+04	2.19E+04	1.09E+04	14.4	**57.2**	2190.4	1095.6
		6	3.07E+04	2.70E+04	1.36E+04	12.2	**56.4**	2696.6	1338.1

Table 12 *Number of data points expected and observed at the host database w.r.t. the number of metrics and sampling frequency. Throughput is inserted data points per second.* ***L%+Zeros*** *is the ratio of false zeros subtracted from inserted values to the expected value. A.throughput is the number of correct data points inserted to the database per second.*

5.7.3 ACCURACY OF HARDWARE PERFORMANCE COUNTER SAMPLING

To measure the accuracy of hardware performance counting, we employed `likwid-bench` micro-benchmark, which executes the generated assembly code with adjustable size and time and reports performance events that have correspondence with hardware performance counters. From the reported values of `likwid-bench`, `Cycles` is calculated with `UNHALTED_REFERENCE_CYCLES`, the number of FLOPs is calculated with `FP_ARITH:SCALAR_DOUBLE` on Dolap, Deren and Poseidon and `FP_COMP_OPS_EXE:X87` on Luna.[35]

Data volume is calculated as `MEM_UOPS_RETIRED:ALL_LOADS+MEM_UOPS_RETIRED:ALL_STORES` on Dolap, Deren and Luna, and `MEM_INST_RETIRED:ALL_LOADS+MEM_INST_RETIRED:ALL_STORES` on Poseidon. The number of total instructions is calculated as `INSTRUCTION_RETIRED` on all hosts. UOPs is calculated with `UOPS_RETIRED_SLOTS` on all hosts except on Poseidon calculated with `UOPS_RETIRED_SLOTS`. Finally, `AI` is then calculated total FLOPs/total bytes for corresponding metrics. Micro-benchmark kernels `triad`, `sum`, `stream`, `peakflops`, `ddot` and `daxpy` executed on all hosts with varying frequencies and additional metrics other than previously mentioned in order to provide a deep analysis of accuracy. To focus on the L1 bandwidth on memory operations, kernels are executed with 100KB size, which completely fits in the cache on all hosts.

To present the setting with the highest accuracy, averaged relative errors for all kernels are examined and presented in Figure 86. It's found that Luna performs slightly worse than other hosts in terms of accuracy for all metrics, except for `UNHALTED_REFERENCE_CYCLES`. This is due to the fact that benchmarks are performed without fixing core frequency, and Deren, which is a much newer architecture, had much more fluctuations. However, results are presented this way due to still having a low error with varying core frequency, which is much more realistic for any other daily scenario. Moreover, Luna is found to be unable to report correct floating point operations. Luna's floating point operation reports seem to be unaffected by the executed kernels and always have $\approx$ 100% error. Nevertheless, since Luna is older than the other baseline, this error level is acceptable and a deeper analysis is not performed.

It's found that on every host frequency rating, 1/8 achieved the best accuracy. This is on par with findings from throughput, and higher errors coming with higher frequencies are thought to be a result of losses in reported data points. However, to show best and worst case scenarios and to show the impact of losses in high-frequency reporting, cases yielding the best and worst accuracy broken down to kernels are presented in Table 13 and 14 respectively. It's found that, as mentioned in,[36] architectures differ in accuracy for different events. While Poseidon chronically has the highest error with `UOPs` event in Table 13, it exhibits the lowest error in the floating point event. Among other hosts, Dolap and Deren are found to perform consistently with high accuracy, while Luna is found to have acceptable errors on all events other than the floating point. It's also found that the accuracy of measurements can be affected by the type of kernel, as in the case of `peakflops UOPs`. Poseidon achieves a thousand times less error than other kernels. Besides, Dolap and Luna also achieve the lowest error for `UOPs`. That may imply that PMUs are more accurate when counting the same type of events for a given metric. Another important finding is that, even with the worst-case accuracy, the calculated `AI` values are accurate enough to build roofline models. Still, the losses in sampling increased all errors of all hosts 4 to 10 times.

[35] Vincent Weaver, Dan Terpstra, and Shirley Moore. "Non-determinism and overcount on modern hardware performance counter implementations". 2013, pp. 215–224. DOI: 10.1109/ISPASS.2013.6557172.

[36] Ibid.

Dolap	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	1.01E-03	1.18E-02	8.65E-04	1.33E-03	1.35E-03	0.0625	0.0625
sum	1.16E-03	1.10E-02	1.50E-03	1.70E-03	1.80E-03	0.1249	0.1250
stream	2.32E-03	1.17E-02	1.15E-03	1.43E-03	2.26E-03	0.0833	0.0833
peakflops	1.61E-03	1.17E-02	4.27E-04	4.30E-05	4.87E-03	1.9935	2.0000
ddot	1.06E-04	1.08E-02	4.09E-04	2.66E-04	6.29E-04	0.1249	0.1250
daxpy	2.07E-03	1.16E-02	8.89E-04	1.74E-03	1.68E-03	0.0834	0.0833
Deren	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	5.80E-05	2.57E-02	2.65E-03	2.27E-03	1.80E-03	0.0624	0.0625
sum	5.63E-04	2.45E-02	4.05E-03	4.45E-03	3.90E-03	0.1246	0.1250
stream	1.76E-03	2.61E-02	3.59E-04	8.56E-04	5.79E-04	0.0831	0.0833
peakflops	8.28E-04	2.44E-02	6.15E-04	1.25E-03	4.45E-03	1.9896	2.0000
ddot	1.15E-03	2.57E-02	3.03E-03	2.85E-03	2.46E-03	0.1248	0.1250
daxpy	5.54E-04	2.48E-02	8.27E-04	2.70E-03	1.50E-03	0.0832	0.0833
Poseidon	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	1.18E-03	2.00E-03	1.08E-03	3.99E-01	1.19E-03	0.0624	0.0625
sum	9.91E-04	1.81E-03	1.20E-03	4.43E-01	1.18E-03	0.1247	0.1250
stream	1.89E-04	1.48E-03	1.55E-03	3.06E-01	2.31E-03	0.0831	0.0833
peakflops	4.41E-04	1.69E-03	1.52E-04	9.53E-04	2.48E-03	1.9942	2.0000
ddot	9.85E-04	2.10E-03	4.73E-04	2.21E-01	1.31E-03	0.1247	0.1250
daxpy	9.40E-04	2.21E-03	1.74E-04	3.07E-01	1.97E-04	0.0832	0.0833
Luna	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	1.00E+00	2.30E-02	1.53E-02	1.67E-02	1.93E-02	0.0000	0.0625
sum	1.00E+00	1.99E-02	1.56E-02	1.47E-02	1.53E-02	0.0000	0.1250
stream	1.00E+00	3.02E-02	2.80E-02	2.59E-02	2.78E-02	0.0000	0.0833
peakflops	1.00E+00	1.73E-02	1.46E-02	1.43E-02	9.14E-03	0.0000	2.0000
ddot	1.00E+00	2.34E-02	2.88E-02	2.86E-02	2.94E-02	0.0000	0.1250
daxpy	1.00E+00	2.25E-02	1.79E-02	1.69E-02	1.80E-02	0.0000	0.0833

Table 13 *Best case scenarios observed for the experiments summarized in Figure 86. Likwid-bench kernels are sampled with* 8 *metrics with frequency* 8/s*, executed* 10 *times, and average values are reported.*

Dolap	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	2.05E-02	7.55E-02	3.81E-02	1.29E-01	2.62E-02	0.0628	0.625
sum	2.01E-02	7.98E-02	4.35E-02	1.31E-01	3.26E-02	0.1266	0.125
stream	2.28E-02	7.81E-02	4.93E-02	1.31E-01	3.68E-02	0.0845	0.0833
peakflops	1.93E-02	7.98E-02	4.29E-02	1.33E-01	2.39E-02	2.0094	2.000
ddot	2.21E-02	7.20E-02	4.19E-02	1.34E-01	3.03E-02	0.1260	0.125
daxpy	2.01E-02	7.90E-02	4.31E-02	1.33E-01	2.97E-02	0.0841	0.0833
Deren	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	5.55E-03	5.57E-02	4.08E-02	2.48E-03	2.09E-03	0.0620	0.625
sum	5.54E-03	5.51E-02	4.04E-02	2.75E-03	1.53E-03	0.1241	0.125
stream	3.94E-03	5.74E-02	4.26E-02	6.32E-03	5.58E-03	0.0825	0.0833
peakflops	5.60E-03	5.40E-02	3.84E-02	6.97E-04	4.76E-03	1.9795	2.000
ddot	8.13E-03	5.33E-02	3.79E-02	9.21E-04	1.85E-03	0.1242	0.125
daxpy	4.91E-03	5.89E-02	4.19E-02	6.85E-04	2.86E-03	0.0826	0.0833
Poseidon	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	8.23E-03	8.74E-02	3.95E-02	4.02E-01	3.34E-03	0.0618	0.625
sum	7.28E-03	8.17E-02	3.40E-02	4.47E-01	5.48E-03	0.1247	0.125
stream	5.08E-03	8.52E-02	4.02E-02	3.10E-01	3.10E-05	0.0829	0.0833
peakflops	5.98E-03	9.07E-02	4.37E-02	5.91E-03	1.63E-02	1.9570	2.000
ddot	6.45E-03	8.13E-02	3.57E-02	2.25E-01	1.44E-03	0.124	0.125
daxpy	7.02E-03	7.97E-02	3.58E-02	3.11E-01	8.42E-03	0.0834	0.0833
Luna	*fp*	*cycle*	*inst*	*uops*	*bw*	*ai*	*ai real*
triad	1.00E+00	1.35E-01	9.51E-02	5.57E-02	3.00E-02	0.0000	0.625
sum	1.00E+00	1.36E-01	8.68E-02	3.43E-02	3.11E-02	0.0000	0.125
stream	1.00E+00	1.24E-01	7.87E-02	2.71E-02	3.03E-02	0.0000	0.0833
peakflops	1.00E+00	1.30E-01	8.35E-02	3.93E-02	3.10E-02	0.0000	2.000
ddot	1.00E+00	1.16E-01	2.97E-02	1.40E-02	8.58E-02	0.0000	0.125
daxpy	1.00E+00	1.56E-01	9.14E-02	3.84E-02	2.19E-02	0.0000	0.0833

Table 14 *Worst case scenarios observed for the experiments in Figure 86. Likwid-bench kernels are sampled with 24 metrics with frequency 16/s, executed 10 times and average values are reported.*

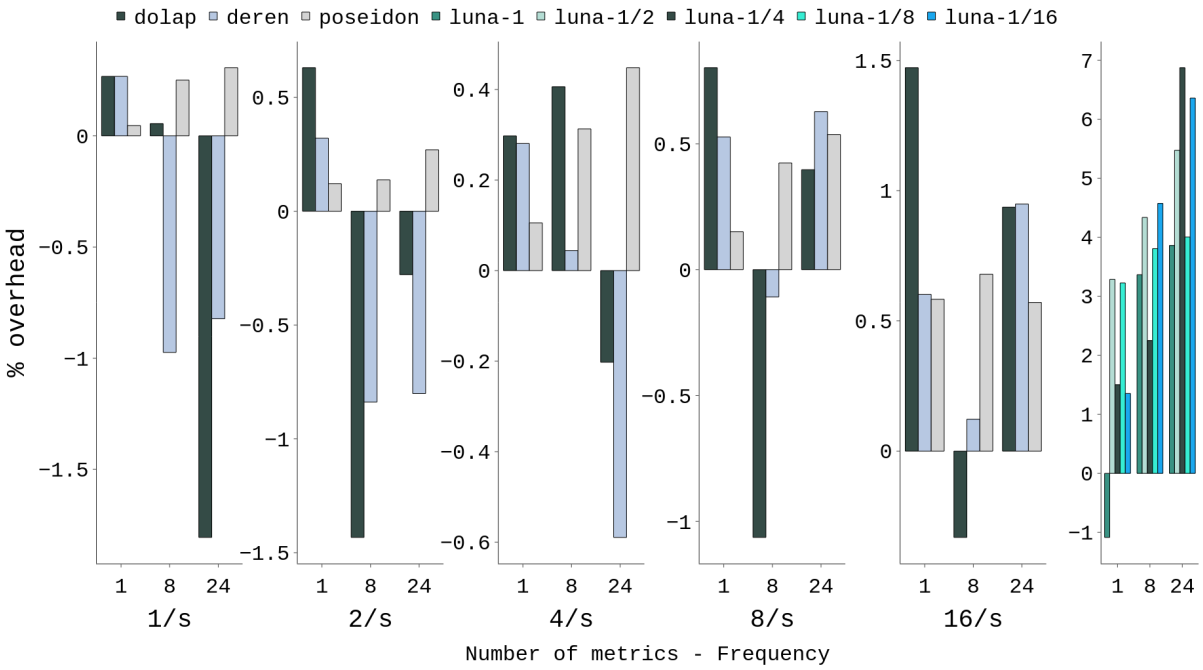


Figure 87 *Overhead of PMU sampling on* 4 systems *using PCP via SuperTwin. Values represent likwid-bench kernels* `triad`, `stream`, `sum`, `peak flops`, `ddot`, `daxpy` *executed 10 times each and averaged together with* 1, 8, *and* 24 *metrics sampled. Comparison is against the baseline in which no sampling takes place.*

5.7.4 OVERHEAD OF MEASUREMENTS

To measure the overhead of the PMU profiling, `likwid-bench` kernels are executed for 10 times. The runtimes without profiling and with a different number of metrics are reported for different sampling frequencies in Figure 87. The only system that consistently experiences overhead from PMU sampling is found to be Luna. This is understandable since Luna is an old architecture and has poor performance. Apart from Luna, negative overheads are observed which means the overhead added by sampling is smaller than the natural variance observed between different runs of the same kernel. A similar negative overhead is also reported in,[37] even in a much bigger distributed setting. However, a meaningful skew towards positive overhead is observed with increasing frequency. That hints that, in previously presented integrity results, PCP did not just skip samplings for high frequencies. It tries to sample events; however, it could not catch up with high frequency. Still, the overhead is very low, and negative overheads are present with runs where the frequency is 16/sec.

5.8 SUMMARY OF ACTIVITIES

Since the start of the development of SuperTwin, digital twin and linked time-series data approaches are proven to be worthy of attention since, their previous motivations on knowledge management, automated modeling, visualization, and comparison are adapted, implemented, and shown to function in a lightweight and accurate way for creating digital twins of supercomputers. With the added augmentation and semantical query abilities, they allow for automatically

[37] Andrzej Nowak and Georgios Bitzes. *The overhead of profiling using PMU hardware counters*. 2014. DOI: 10.5281/zenodo.10800. URL: https://doi.org/10.5281/zenodo.10800.

generated interlinked dashboards for every individual component for a target system and the proposed data structure is promising to be scaled to much larger systems. SuperTwin also stays promising for straightforward integration of external tools since the integrations of several benchmarks such as CARM, STREAM, and HPCG into SuperTwin are done with the same approach without giving up the homogeneity of the framework. Moreover, alongside the capabilities benchmarks added to the SuperTwin, SuperTwin also added capabilities to those benchmarks. For example, while CARM adds SuperTwin's ability to generate performance models, SuperTwin adds cache-aware roofline model ability of an automated generation of performance models for different threading settings and concerning NUMA domains, alongside with the capability to mark executed kernels on generated rooflines on-the-fly. Although automated reports of the effect of thread affinities on benchmarks are under development, it is shown that benchmarks can be tailored concerning metadata probed from the system and executed automatically with digital twins. This property also enables those results to be used as baselines for later measurements. Another important point is that the measurements of memory bandwidth, floating point operations, and metrics acquired by combining them are accurate enough to observe real-time deviations from performance models and generate application-specific performance models. This is now considered as a future work since SuperTwin already provides the required metadata. Moreover, the ability to automatically and quickly make a high volume of observations with minimal configuration and compare them, although yet to be realized, will prove worthy of architecture research and algorithm research.

6 CONCLUSIONS

Deliverable 1.4 focused on presenting performance and energy models and communication/profiling tools targeting sparse computation workloads on modern microarchitectures and systems. Sparse computing is at the core of several key applications with significant societal and/or economic impact, either in the form of sparse matrix multiplication or in the form of sparse tensor contractions involving arbitrary dimensional tensors with different levels of sparsity. Modern computing platforms are designed as a means to address the ever rising escalation in regard to the performance and energy-efficiency needs of these and other workloads. However, many of the contemporary microarchitectures and devices on modern computing platforms are yet to be fully exploited for performing sparse computations. Hence the importance of high performing and accurate models, profiling tools and digital replication software that allow us to predict the behavior of today's emerging systems and make better informed decisions when tasked with developing high-performance and energy-efficient methods for processing different representations of sparse data in the context of different computational problems.

We demonstrated the utilization of sparse-aware CARM, a novel modeling approach that strives to accurately characterize sparse computations and assesses the ability to make efficient use of a targeted microarchitecture. We provide detail explanations for the proposed model in regard to its interoperability, revealing its complete construction and interpretation methodology. In-depth validation and characterization relying on Sparse-aware CARM has been performed for a hand-tuned assembly SpMV kernel on an Intel x86 multi-core CPU using sparse matrices with synthetic and real-word data. Through the visualization of effects on cache locality and load balancing, we evaluated the ability of a set of matrix reordering schemes to improve the utilization of computational resources considering both single- and multi-threaded execution. Furthermore, for a range of kernel AIs and a range of CPU core frequencies, a novel method for roofline-based evaluation has been used, which can guide the optimization of applications in regard to performance, power consumption and/or energy efficiency. Relying on the previously referred approach for assembly implementation of SpMV, a performance analysis targeting a RISC-V microarchitecture has been performed, which recently emerged as a promising solution to the next generation energy-efficient computing platforms. A range of AIs has been analyzed in regard to the resulting effect on utilization of caches. The use of CARM evidenced the profound impact that certain aspects of RISC-V microprocessor architectures have on performance of SpMV, identifying bottlenecks, and as a result, opportunities for the design space exploration of RISC-V microprocessors in a direction of domain-specific architectures for efficient processing of sparse data.

In pair with SpMV operations, SpMM represents arguably one of the most popular types of operations over sparse data. A set of SpMM implementations targeting GPU devices have been explored on a modern GPU microarchitecture (NVIDIA Ampere). Making use of DVFS, a study cross-comparing different SpMM approaches has been performed for identifying the effects of selecting optimal combinations of GPU memory and core frequencies on performance, power consumption, energy-efficiency and resulting energy consumption. Both SpMM approaches relying on the general purpose cores on the GPU (CUDA cores) and specialized cores (dense and sparse tensor cores) have been considered. In comparison to the use of GEMM, the study clearly demonstrated that the processing of some sparse matrices can be significantly accelerated and more energy-efficient through the use of SpMM kernels, and that individual tuning of memory and core frequencies to each SpMM method can significantly improve execution in regard to these metrics. Furthermore, we also tackled computer workloads involving the use of sparse TTM, another notable type of tensor contraction processing sparse data, consisting in performing

the product between a multi-dimensional tensor and a matrix. A set of data-parallel TTM methods have been developed and tested on Intel CPU and GPU microarchitectures, identifying their upper bounds relying on CARM modeling. In order to inform the model, the AI intensity of the different methods and their performance have been evaluated taking into account the use of real and synthetic data.

Most of the profiling tools for CPUs target Intel microarchitectures. There is, however, an increasing presence of other CPU microarchitectures in HPC systems, such as AMD X86 Zen and Fujitsu ARM-based A64FX microarchitectures. This makes it important to create novel tools and/or extend and test existing ones on these novel emerging architectures. The proposed profiling tools for inter-thread communication and reuse distance analysis of multithreaded applications (ComDetective and ReuseTracker) have been extended to support low-overhead profiling on modern AMD microprocessors. The tools have been evaluated in regard to their overhead and accuracy under different scenarios, considering different sampling intervals and debug register counts. Overall, accurate results have been achieved with a low overhead in comparison to other means of collecting the same runtime data. This has been made possible on AMD CPUs due to the use of instruction-based sampling (IBS), which has been achieved by modifying a special Linux kernel module. The ARM-based Fujitsu A64FX GPU is another example of a very capable micro architecture that has been recently introduced to the market. This processor supports cache partitioning and mapping of program objects at runtime through a mechanism named sector cache. Experimental evaluation with matrices from cardiac electrophysiology showed that the use of sector cache improved performance and memory bandwidth, being the only exception when processing a matrix that already fits completely in L2 cache. A profiling tool has been developed to predict the improvement in regard to cache misses that one is expected to obtain if using the sector cache on a given program. Real measurements have been performed, which were closely matched by those predicted by the tool.

This deliverable documents the steps taken to take SuperTwin, a digital replication framework targeting supercomputer platforms, out of the prototype stage. Relying on the description of the target system and a large collection of integrated tools, SuperTwin can be used to accurately model, monitor and/or observe the system in a lightweight manner. In relation to the previously presented prototype, SuperTwin now has additional augmentation and semantical query abilities and additional benchmarking capabilities. Automatic generation of dashboards for real-time and per-request monitoring, as well as CARM-based modeling with integrated visualization, has also been added to SuperTwin. The relationship between SuperTwin and the tools integrated into the framework is a symbiotic one. For instance, in the case of CARM-based modeling, SuperTwin gains access to generation of state-of-the-art performance models, while the tool used to perform modeling automatically gains access the possibility of modeling for alternative configurations of the system (threading settings, NUMA domains, etc). SuperTwin has been validated in relation to throughput and integrity, varying the sampling frequency and the amount of events reported on four different computing platforms. The measurements related to throughput of floating-point operations and memory bandwidth have been demonstrated to be precise enough to instantaneously identify deviations from performance models.

REFERENCES

A64FX Microarchitecture Manual. Version 1.5. Fujitsu Limited. 2021. URL: https://github.com/fujitsu/A64FX/blob/master/doc/.

Adhianto, L. et al. "HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs". *Concurrency Computation: Practice Experience* 22.6 (2010), pp. 685–701.

Amestoy, Patrick R., Timothy A. Davis, and Iain S. Duff. "An Approximate Minimum Degree Ordering Algorithm". *SIAM Journal on Matrix Analysis and Applications* 17.4 (1996), pp. 886–905. DOI: 10.1137/S0895479894278952.

Carlson, Andrew et al. "Toward an Architecture for Never-Ending Language Learning." *AAAI*. Vol. 5. 2010, p. 3.

Çatalyürek, Ümit V and Cevdet Aykanat. "Patoh (partitioning tool for hypergraphs)". *Encyclopedia of parallel computing*. Springer, 2011, pp. 1479–1487.

Corporation, NVIDIA. *cuBLAS*. Version 12.0. 2023. URL: https://developer.nvidia.com/cublas.

— *cuSPARSE*. Version 12.0. 2023. URL: https://developer.nvidia.com/cusparse.

Davis, Timothy A and Yifan Hu. "The University of Florida sparse matrix collection". *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–25.

Drongowski, Paul J. *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors*. https://pdfs.semanticscholar.org/5219/4b43b8385ce39b2b08ecd409c753e0efafe5.pdf. 2007.

Friedemann. "Linked Data Architecture for Assistance and Traceability in Smart Manufacturing". *MATEC Web of Conferences* 304 (2019), p. 04006. DOI: 10.1051/matecconf/201930404006.

Greathouse, Joseph L. *AMD Research Instruction Based Sampling Toolkit*. https://github.com/jlgreathouse/AMD_IBS_Toolkit. 2017.

— *Re: Error : IBS profiling is disabled in your BIOS*. https://community.amd.com/t5/general-discussions/error-ibs-profiling-is-disabled-in-your-bios/td-p/55043. AMD Community.

— *Re: IBS not available on EPYC 7451 ?* https://community.amd.com/t5/server-gurus-discussions/ibs-not-available-on-epyc-7451/m-p/258228. AMD Community.

Herault, Thomas et al. "Distributed-memory multi-GPU block-sparse tensor contraction for electronic structure". *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2021, pp. 537–546. DOI: 10.1109/IPDPS49936.2021.00062.

Ilic, Aleksandar, Frederico Pratas, and Leonel Sousa. "Cache-aware Roofline model: Upgrading the loft". *IEEE Computer Architecture Letters* 13.1 (2013), pp. 21–24.

Kerr, Andrew et al. *CUTLASS*. Version 2.11.0. 2022. URL: https://github.com/NVIDIA/cutlass.

Kim, Jinsung et al. "Optimizing Tensor Contractions in CCSD(T) for Efficient Execution on GPUs". *Proceedings of the 2018 International Conference on Supercomputing*. 2018, pp. 96–106. DOI: 10.1145/3205289.3205296.

Levy, Ryan, Edgar Solomonik, and Bryan K. Clark. "Distributed-Memory DMRG via Sparse and Dense Parallel Tensor Contractions". *CoRR* abs/2007.05540 (2020).

Li, Jiajia et al. "Optimizing Sparse Tensor Times Matrix on Multi-core and Many-Core Architectures". *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*. 2016, pp. 26–33. DOI: 10.1109/IA3.2016.010.

Lipton, Richard J., Donald J. Rose, and Robert Endre Tarjan. "Generalized Nested Dissection". *SIAM Journal on Numerical Analysis* 16.2 (1979), pp. 346–358. DOI: 10.1137/0716027.

Liu, Jiawen et al. "Athena: High-Performance Sparse Tensor Contraction Sequence on Heterogeneous Memory". *Proceedings of the ACM International Conference on Supercomputing*. 2021, pp. 190–202. DOI: 10.1145/3447818.3460355.

Liu, Jiawen et al. "Sparta: High-Performance, Element-Wise Sparse Tensor Contraction on Heterogeneous Memory". *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2021, pp. 318–333. DOI: 10.1145/3437801.3441581.

Liu, Wai-Hung and Andrew H. Sherman. "Comparative Analysis of the Cuthill–McKee and the Reverse Cuthill–McKee Ordering Algorithms for Sparse Matrices". *SIAM Journal on Numerical Analysis* 13.2 (1976), pp. 198–213. DOI: 10.1137/0713020.

Lowe-Power, Jason et al. "The gem5 Simulator: Version 20.0+". en. *arXiv:2007.03152 [cs]* (2020). arXiv: 2007.03152. URL: http://arxiv.org/abs/2007.03152 (visited on 12/28/2021).

Luk, Chi-Keung et al. "Pin: building customized program analysis tools with dynamic instrumentation". *Acm sigplan notices* 40.6 (2005), pp. 190–200.

Ma, Yuchen et al. "Optimizing Sparse Tensor Times Matrix on GPUs". *J. Parallel Distrib. Comput.* 129.C (2019), pp. 99–109. DOI: 10.1016/j.jpdc.2018.07.018.

Nowak, Andrzej and Georgios Bitzes. *The overhead of profiling using PMU hardware counters*. 2014. DOI: 10.5281/zenodo.10800. URL: https://doi.org/10.5281/zenodo.10800.

Reddy Kuncham, Goutham Kalikrishna, Rahul Vaidya, and Mahesh Barve. "Performance Study of GPU applications using SYCL and CUDA on Tesla V100 GPU". *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 2021, pp. 1–7. DOI: 10.1109/HPEC49654.2021.9622813.

Sasongko, Muhammad Aditya et al. "ComDetective: A Lightweight Communication Detection Tool for Threads". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver, Colorado: Association for Computing Machinery, 2019. DOI: 10.1145/3295500.3356214. URL: https://doi.org/10.1145/3295500.3356214.

Smith, Shaden and George Karypis. "Tensor-Matrix Products with a Compressed Sparse Tensor". *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. 2015. DOI: 10.1145/2833179.2833183.

Smith, Shaden et al. *FROSTT: The Formidable Repository of Open Sparse Tensors and Tools*. 2017. URL: http://frostt.io/.

Weaver, Vincent, Dan Terpstra, and Shirley Moore. "Non-determinism and overcount on modern hardware performance counter implementations". 2013, pp. 215–224. DOI: 10.1109/ISPASS.2013.6557172.

Whiting, Mark et al. "VAST Challenge 2015: Mayhem at dinofun world". *Visual Analytics Science and Technology (VAST), 2015 IEEE Conference on*. IEEE. 2015, pp. 113–118.

Zhao, Haoran et al. "Exploring Better Speculation and Data Locality in Sparse Matrix-Vector Multiplication on Intel Xeon". *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE. 2020, pp. 601–609.

7 HISTORY OF CHANGES

Version	Author(s)	Date	Comment
0.1	Aleksandar Ilic (INESC)	06.03.2023	Initial draft for PO and reviewers
0.2	Didem Unat (KU)	14.03.2023	Communication modeling contributions
0.3	Ricardo Nobre (INESC)	14.03.2023	Inital draft: SpMM evaluation on GPU
0.4	Aleksandar Ilic (INESC)	18.03.2023	Sparse-aware CARM contribution
0.5	Fatih Taşyaran (SU)	21.03.2023	Digital SuperTwin contribution
0.6	Sergej Breiter (LMU)	22.03.2023	A64FX cache partitioning contribution
0.7	Kamer Kaya (SU)	23.03.2023	Digital SuperTwin final content
0.8	Aleksandar Ilic (INESC)	23.03.2023	Roofline-based HW scaling and CPU+GPU TTM
0.9	Ricardo Nobre (INESC)	25.03.2023	SpMM on GPU tensor cores finalized
1.0	Aleksandar Ilic (INESC)	27.03.2023	Pre-final version; Sent to proofreading
1.1	Johannes Langguth (Simula)	30.03.2023	Final version after proofreading

Table 15 *Document History of Changes*