Ref. Ares(2022)4806244 - 30/06/2022

Prototype of performance and energy models

Deliverable No:	D1.2
Deliverable Title:	Prototype of performance and energy models
Deliverable Publish Date:	30 June 2022
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 1
- 1.1 Objectives of This Deliverable 1
- 1.2 Work Performed 2
- 1.3 Deviations and Counter Measures 2
- 1.4 Resources 2

2 Sparse-aware roofline modeling and analysis 3
- 2.1 Analysis of sparse kernels in Original Roofline Model 5
  - 2.1.1 Sparse matrix-vector multiplication 5
  - 2.1.2 Original Roofline Model Analysis 5
- 2.2 Sparse CARM: Improving roofline insightfulness for sparse computations 8
  - 2.2.1 Adapting CARM to SpMV 9
  - 2.2.2 Impact of sparse matrix characteristics in Sparse CARM 12
  - 2.2.3 Improving SpMV performance by reordering 13
- 2.3 Mansard Roofline Model: Sparse Kernels Analysis 23
  - 2.3.1 Mansard Roofline Model 23
  - 2.3.2 Characterization of Sparse Kernels 29
  - 2.3.3 Case Study: Second-Order Epistasis Detection 32

3 Performance and Energy-efficiency Modeling of Graphcore Intelligent Processing Unit 35
- 3.1 Roofline modeling of the in-tile execution 35
- 3.2 Modeling Impact of Inter-Tile Communication: Exchange phase 38
- 3.3 Roadmap for IPU Roofline development 40

4 Data Movement Analysis 41
- 4.1 Inter-Thread Communication Analysis 41
  - 4.1.1 ComDetective 41
  - 4.1.2 Implementation 42
  - 4.1.3 Communication Count Analysis 44
  - 4.1.4 ComDetective^{+} Roadmap 46
- 4.2 Cache Partitioning 46
  - 4.2.1 Profiling Tool 47
  - 4.2.2 Results 47

5 Digital SuperTwin 48
- 5.1 Probing 49
- 5.2 Constructing Digital Twin 49
- 5.3 Digital SuperTwin Roadmap 51

6 Conclusions 52

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 the ongoing research developments regarding the sparse computation-aware performance and energy-efficiency modeling of device architectures along with analysis/profiling tools. These models aim at encapsulating necessary information regarding the performance and energy-efficiency upper-bounds that are realistically exploitable by different classes of sparse applications, thus they can be used to guide optimizations and detect the potential execution bottlenecks with respect to compute and memory resources (e.g., caches and DRAM). In addition, communication modeling and analysis tools aim at identifying horizontal and vertical data movement within the memory hierarchy. The data movement profiling information enables us to optimize communication, data placement and cache partitioning. Finally, the developed performance, communication and energy models provide feedback to the Digital Twin to construct unified models.

1.2 WORK PERFORMED

In this deliverable, an extensive analysis, validation and characterization of different sparse computation kernels was performed in the state-of-the-art insightful models based on the roofline principles, in particular, in the Original Roofline Model (ORM) and Cache-aware Roofline Model (CARM). For this purpose, sparse matrices from SuiteSparse Matrix Collection with different sparsity patterns and characteristics were evaluated, across a set of custom-build algorithms and standard implementations from vendor-specific high-performance libraries, including Sparse Matrix-Vector Multiplication (SpMV) and Sparse Matrix-Matrix Multiplication (SpMM) from Intel MKL. A novel ORM analysis based on best- and worst case memory traffic estimates is also derived. A methodology based on micro-benchmarking is proposed to improve the CARM insightfulness by scaling the performance upper-bounds according to the characteristics of the sparse kernels. The proposed model is experimentally evaluated by considering different reordering algorithms applied to diverse sparse matrices in single- and multi-core execution scenarios.

Furthermore, to address the main drawbacks of the SoA roofline models, the Mansard Roofline Model (MaRM) is proposed, which uncovers a minimum set of architectural features that must be considered to provide insightful, but yet accurate and realistic, modeling of performance upper bounds for modern processors. This model encapsulates the retirement constraints due to the amount of retirement slots, Reorder-Buffer and Physical Register File sizes, and it is employed to characterize SpMV and SpMM kernels from Intel MKL, as well as to guide the optimization of the second-order epistasis detection algorithm (use-case application in the scope of the SparCity project). The obtained results show that the characterization in the proposed models is in line with the Intel TopDown VTune analysis.

Moreover, the roofline modeling principles are also applied when uncovering the performance and energy-efficiency upper-bounds of the Graphcore Intelligent Processing Unit (IPU). For this purpose, different strategies are considered to model different phases of the IPU execution, i.e., in-tile execution and inter-tile communication (exchange). The proposed in-tile execution roofline model is experimentally validated through micro-benhmarking, which shows complete matching between the theoretical model and the experimental data in all modeled domains, i.e., performance, power consumption and energy-efficiency.

In terms of data movement tools, the inter-thread communication detection tool is extended to AMD x86 architectures and tested on a number of benchmarks for its accuracy, sensitivity to sampling interval and time/memory overheads. Moreover, cache partitioning is applied to an iterative conjugate gradient method based on a CSR SpMV kernel to reduce L2 cache misses. The initial results show that even though the matrices that are already reordered for optimal cache reuse can still benefit from cache partitioning to further reduce cache misses. Lastly, all the performance, power and communication models developed in this WP are communicated to the WP4 team to influence the design of performance metrics in Digital Twin.

1.3 DEVIATIONS AND COUNTER MEASURES

There was no deviation from the work plan.

1.4 RESOURCES

As also envisioned in the project proposal, it is expected that the herein elaborated modeling approaches will undergo further improvements and developments, which will be reported in subsequent deliverables (D4.2, D1.4 and D1.5), as well as maintained and regularly updated on the respective SparCity Github repositories.

2 SPARSE-AWARE ROOFLINE MODELING AND ANALYSIS

The State-of-the-Art (SoA) roofline models, such as, Original Roofline Model (ORM)[1] and its hierarchical variant,[2] Integrated Roofline Model (IRM),[3] and Cache-Aware Roofline Model (CARM)[4] are widely used for application characterization and optimization, especially when considering sparse computations.[5] The popularity of these models arise from their ability to relate application behavior and device upper-bounds in a simple and insightful way. These models focus on the peak compute performance and maximum achievable memory bandwidth, which provides a visualization of the main bottlenecks that hinder application execution, and its potential to fully explore system capabilities.

In particular, CARM characterizes performance upper-bounds for a given architecture, with respect to the arithmetic intensity (AI), *i.e.*, the amount of performed computations over the total amount of requested data ($bytes$).[6] By considering that memory operations and computations can be simultaneously executed in modern out-of-order processors, the overall execution is limited either by the time to perform computations or by the time to serve memory requests. Hence, CARM contains three distinct regions: memory bound region (slanted roof), compute bound region (horizontal roof) and a "mixed" region, where applications can be both memory and/or compute bound. For each memory level, the slanted roof intersects the horizontal roof at a single point, *i.e.*, the ridge point.[7]

As it can be observed in Figure 1a, CARM memory region includes all memory levels (L1, L2, L3 and DRAM) in a single plot, each limited by its corresponding slanted roof. The maximum attainable performance in this region is limited by the L1 cache bandwidth, while the remaining levels offer a lower attainable performance, due to the reduction in the sustainable bandwidth when data is fetched further away from the core. The right part of the model, delimited by the maximum FP performance (F_p), represents the compute bound region.

When characterizing the application behavior, CARM decouples the bottlenecks limiting the application execution, allowing to select suitable optimization techniques to be applied. For example, if an application is at the left side of the ridge point (kernel "M" in Figure 1a), its execution is limited by memory accesses and can be improved by applying memory-related optimizations. On the other hand, an application positioned at the right side of the ridge point (kernel "C" in Figure 1a) is limited by arithmetic operations and its execution can be improved by

[1]Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". *Commun. ACM* 52.4 (2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.

[2]Douglas Doerfler et al. "Applying the roofline performance model to the Intel Xeon Phi Knights Landing processor". *Proceedings of the International Conference on High Performance Computing*. Springer. 2016, pp. 339–353.

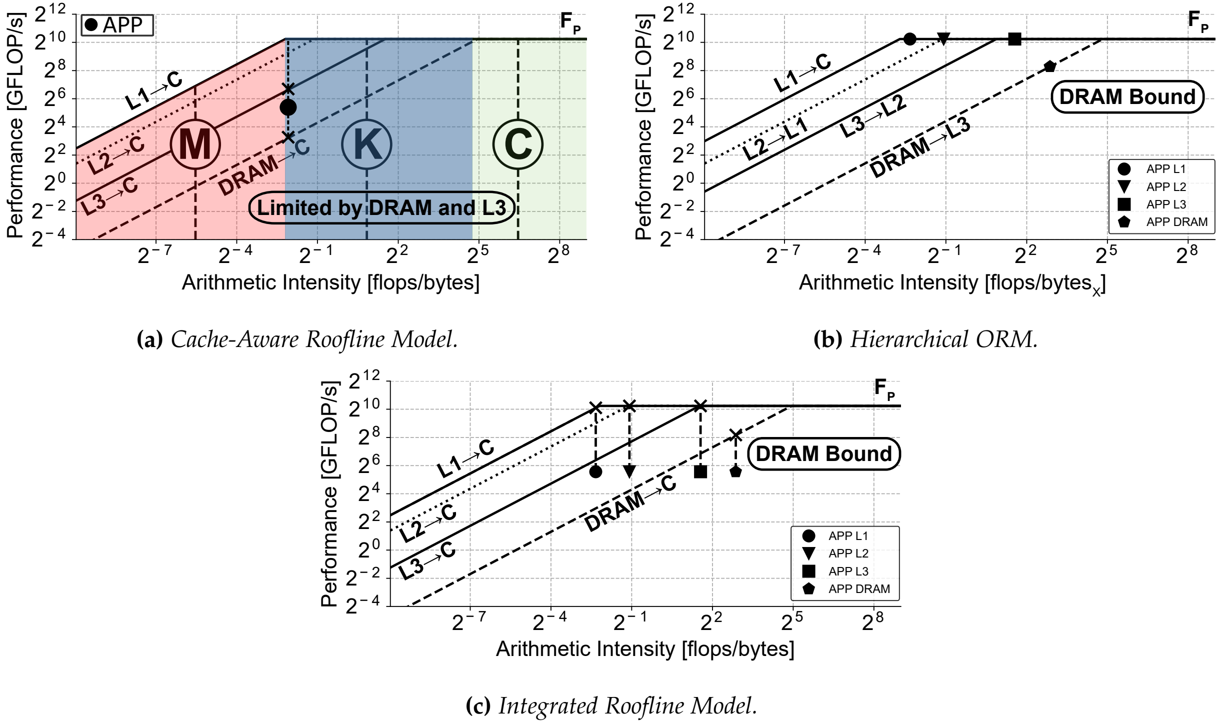
[3]Tuomas Koskela et al. "A novel multi-level integrated Roofline model approach for performance characterization". *Proceedings of the International Conference on High Performance Computing*. Springer. 2018, pp. 226–245.

[4]Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. "Cache-aware Roofline model: Upgrading the loft". *IEEE Computer Architecture Letters* 13.1 (2013), pp. 21–24; Diogo Marques et al. "Performance analysis with Cache-Aware Roofline model in Intel Advisor". *Proceedings of the International Conference on High Performance Computing & Simulation*. IEEE. 2017, pp. 898–907.

[5]N. Srivastava et al. "MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product". *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 766–780. DOI: 10.1109/MICRO50266.2020.00068; N. Srivastava et al. "Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations". *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 689–702. DOI: 10.1109/HPCA47549.2020.00062; Jiajia Li et al. "A Sparse Tensor Benchmark Suite for CPUs and GPUs". *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2020, pp. 193–204.

[6]Ilic, Pratas, and Sousa, "Cache-aware Roofline model: Upgrading the loft".

[7]Ilic, Pratas, and Sousa, "Cache-aware Roofline model: Upgrading the loft"; Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. "Beyond the Roofline: Cache-Aware Power and Energy-Efficiency Modeling for Multi-Cores". *IEEE Transactions on Computers* 66.1 (2016), pp. 52–58.



(a) *Cache-Aware Roofline Model.*

(b) *Hierarchical ORM.*

(c) *Integrated Roofline Model.*

Figure 1 *State-of-the-art Roofline Modeling approaches.*

code vectorization or parallelization. Finally, an application placed in the "mixed" region (kernel "K" in Figure 1a) may be limited by computations and/or memory transfers, depending on their instruction mix and on the memory levels exercised by the application.

Moreover, by plotting a vertical line at the AI of the application, as shown in Figure 1a, it is possible to uncover the main sources of performance degradation. The intersections of this vertical line and the CARM roofs represent the potential execution bottlenecks that might limit application performance. The intersections right above and below the application point are identified as the main sources of performance degradation and should be the main target of optimization. In the example of Figure 1a, the main bottlenecks of the application are accesses to L3 and DRAM memories (see black dot in Figure 1a).

Compared to ORM and IRM, while the compute region is evaluated equally in all three models, their modeling of the upper-bounds in the memory subsystem differs. Hierarchical ORM (Figure 1b) considers the bandwidth between memory levels, and its AI corresponds to the amount of performed computations over the amount of data requested by memory level "x" ($\mathtt{bytes}_x$).[8] Due to this property, a single application (kernel) is represented by "x" points in Hierarchical ORM, one for each memory level. Similar to CARM, the memory region of the Hierarchical ORM contains several roofs, each one representing a memory level. The main execution bottleneck corresponds to the minimum of the intersections between the AIs of the "x" points with their correspondent roofs (*e.g.*, DRAM bandwidth in Figure 1b). Finally, IRM (Figure 1c) uses the modeling approach of CARM in the memory subsystem *i.e.*, it considers the sustainable bandwidth seen from the core for each memory level, while adopting ORM methodology for application characterization and bottleneck detection (*e.g.*, in Figure 1c, DRAM is the main execution bottleneck).

[8] Williams, Waterman, and Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures".

2.1 Analysis of sparse kernels in original roofline model

Performance models such as the Original Roofline Model (ORM)[9] are often used to describe the upper limits of achievable performance for computational kernels on a particular hardware. While such models are simple and effective, they sometimes fail to predict the performance that is observed in practice, especially in the case of kernels that are dominated by irregular memory accesses. More detailed performance models are therefore being explored in the context of the SparCity project, precisely because irregular memory access patterns are an inherent feature of the kind of sparse computations that SparCity is aimed at. One such detailed performance model has been recently developed based on a cache tracing approach.[10] This has so far been applied to better understand the performance of irregular kernels, in particular different variants of Sparse Matrix Vector Multiplication (SpMV).

This section briefly describes a common sparse matrix-vector multiplication kernel and an ORM analysis based on best- and worst case memory traffic estimates. The shortcomings of this simple model illustrates the need for more detailed approaches.

2.1.1 Sparse matrix-vector multiplication

A commonly used sparse matrix-vector multiplication kernel based on the compressed sparse row (CSR) storage format is shown in Algorithm 1. This kernel computes $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{A}\mathbf{x}$, or

$$y_i \leftarrow y_i + \sum_{k=r_i}^{r_{i+1}-1} a_k x_{j_k}, \qquad \text{for } i = 0, 1, \ldots, M-1, \tag{1}$$

for an M-by-N matrix $\mathbf{A} = \left(a_{i,j}\right)_{i,j=0}^{M-1,N-1}$, a source vector $\mathbf{x} = \left(x_j\right)_{j=0}^{N-1}$ and a destination vector $\mathbf{y} = (y_i)_{i=0}^{M-1}$. The matrix $\mathbf{A}$ is assumed to be given by K nonzero entries $(i_k, j_k, a_k)_{k=0}^{K-1}$, such that i_k and j_k are row and column offsets, respectively, of the nonzero matrix value $a_k := a_{i_k,j_k}$. Moreover, $K \ll M \times N$ due to the matrix being sparse. It is also assumed that the nonzeros are sorted in ascending order of their row offsets, i_k. The row pointer r_i designates the position of the first nonzero of the ith row and $r_{i+1} - 1$ points to the last nonzero of the ith row.

The kernel in Algorithm 1 consists of two nested loops, where the outer loop runs over the rows of the matrix and the inner loop iterates over the nonzeros of the current row. Two floating point operations, one multiplication and one addition, are performed for each matrix nonzero. While the matrix nonzero values (`a`) and column indices (`colidx`) are accessed in a streaming fashion, accesses to the source vector `x` are irregular and unpredictable. Finally, OpenMP is used to partition the rows of the matrix and distribute the work among threads in a shared memory parallel fashion.

2.1.2 Original roofline model analysis

Algorithm 1 appears straightforward to analyse using the ORM (see also Vuduc et al.[11] for a similar analysis of an SpMV kernel that incorporates a register blocking optimisation). However,

[9] Williams, Waterman, and Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures".

[10] James D. Trotter, Johannes Langguth, and Xing Cai. "Cache simulation for irregular memory traffic on multi-core CPUs: Case study on performance models for sparse matrix–vector multiplication". *Journal of Parallel and Distributed Computing* 144 (2020), pp. 189–205. issn: 0743-7315. doi: 10.1016/j.jpdc.2020.05.020.

[11] Richard Vuduc et al. "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply". *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. Baltimore, Maryland: IEEE Computer Society Press, 2002, pp. 1–35. doi: 10.1109/SC.2002.10025.

```
1 void csrspmv(
2   int num_rows, int * rowptr, int * colidx,
3   double * a, double * x, double * y)
4 {
5   #pragma omp for
6   for (int i = 0; i < num_rows; i++) {
7     double z = 0.0;
8     for (int k = rowptr[i]; k < rowptr[i+1]; k++)
9       z += a[k] * x[colidx[k]];
10    y[i] += z;
11  }
12 }
```

Algorithm 1: Matrix-vector multiplication for a sparse matrix in compressed sparse row (CSR) storage format. OpenMP is used to partition the rows of the matrix and distribute the work among threads in a shared memory parallel fashion.

we will soon see that the irregular and data dependent accesses to the source vector `x` make matters more complicated, since they cannot be predicted ahead of time. On the surface, every iteration of the inner loop reads 20 bytes from memory (line 9), 4 bytes due to the column indices (`colidx`) and 8 bytes each due to the values of `a` and `x`, whereas every iteration of the outer loop writes 8 bytes to memory (line 10) for the destination vector `y`. For simplicity, we ignore writes to the destination vector, which anyway become negligible as long as the average number of nonzeros per row is large enough. Thus, concentrating on the inner loop, which performs two floating-point operations per iteration, the arithmetic intensity is $\frac{1}{10}$ flop/B.

As an example, we consider a dual socket system with two Intel Xeon Platinum 8168 CPUs, each with 24 cores operating at 2.5 GHz, and a theoretical memory bandwidth of 256 GB/s. An ORM for this system is shown in Figure 2. In this example, the arithmetic intensity of Algorithm 1 implies an upper bound of 160 Gflop/s, assuming that the 20 bytes needed during each iteration of the inner loop are loaded in 1.5 cycles.[12] However, this is only possible with an already warm cache and a matrix with no more than about 2 700 nonzeros per CPU core (i.e., fewer than 130 000 nonzeros in total). Otherwise, the matrix cannot fit in the 32 KiB L1 cache of each core. Moreover, any matrix with more than about 5.8 million nonzeros will not even fit in the 66 MiB shared L3 cache and must therefore be read entirely from main memory. Although, since the L3 cache in the Skylake-X microarchitecture is non-inclusive,[13] it is arguably more appropriate to consider the 114 MiB combined size of the L2 and L3 caches instead, meaning that matrices with more than 10 million nonzeros will not fit.

In the case of cold caches or a sufficiently large matrix, 12 bytes must be read from main memory for each nonzero to obtain its value and column index. If we assume for now that the source vector resides entirely in cache,[14] the arithmetic intensity becomes $\frac{1}{6}$ flop/B. With a theoretical memory bandwidth of 256 GB/s, the upper limit on performance becomes 42.7 Gflop/s.

[12] The Skylake-X microarchitecture can serve two loads per cycle from the L1 cache.

[13] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-040. Intel Corporation, 2018, Ch. 2.

[14] Even in an ideal scenario, where the entire 1.5 MiB L1 cache is dedicated to storing the source vector, it will only fit in the cache for matrices with up to 196 608 columns. Similarly, the source vector cannot fit entirely in the L3 cache for matrices with more than 8.7 million columns. Within the combined size of the L2 and L3 caches, the source vector will not fit for matrices with more than 14.9 million columns.

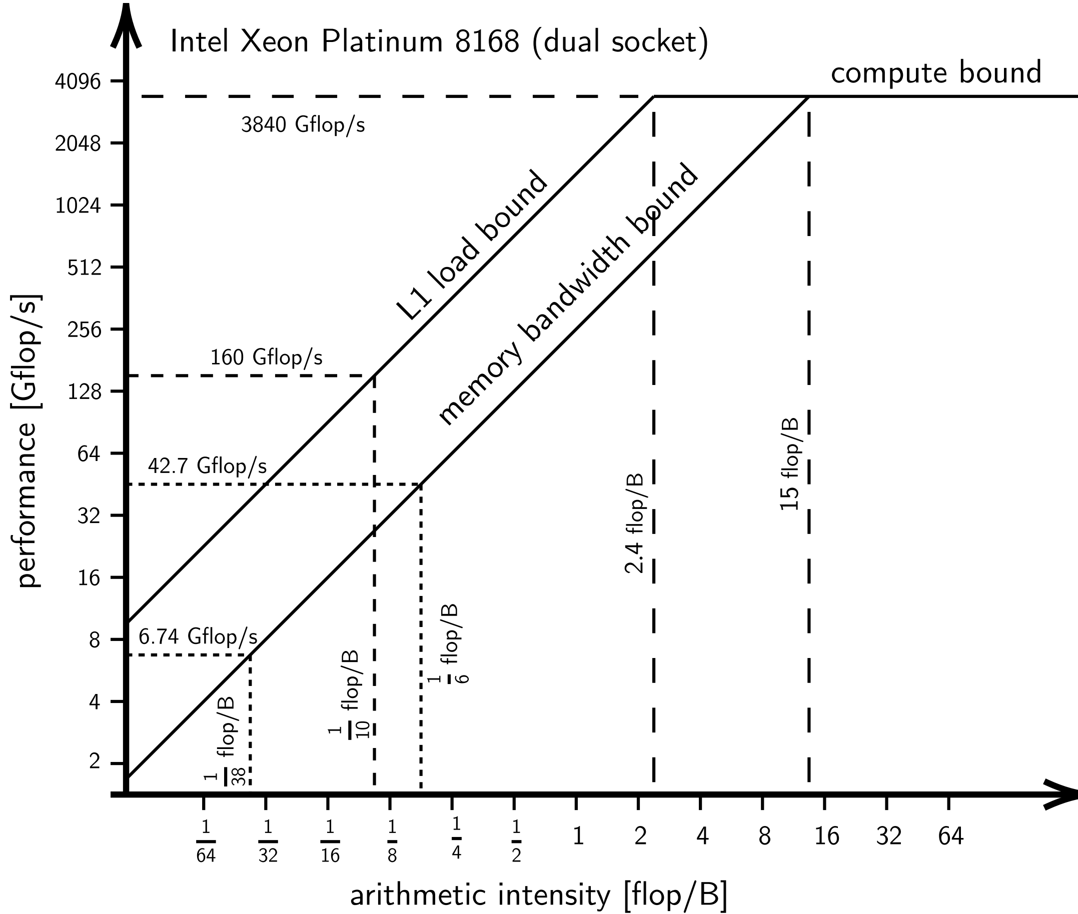


Figure 2 *Original roofline model for sparse matrix-vector multiplication with the CSR storage format (see Algorithm 1) on a dual socket Intel Xeon Platinum 8168 (Skylake-X) multicore CPU system.*

Note that this estimate is somewhat optimistic, since the actual memory bandwidth achieved in practice, as measured, for instance, by the STREAM benchmark,[15] is commonly found to be only about 70–80 % of the theoretical maximum. A more conservative estimate would therefore be an upper performance limit of about 30 Gflop/s.

Now, even if the matrix is too large or the cache is cold, the source vector x may still benefit from reusing cached data. Whether or not the source vector values are reused in practice depends on details of the caching algorithm, as well as the irregular memory access pattern induced by the column indices of the matrix nonzeros and the order in which the nonzeros are arranged. The best case, as described above, yields an arithmetic intensity of $\frac{1}{6}$ flop/B. But the worst case occurs when every access to the source vector must bring an entire cache line of 64 bytes from main memory, even though only a single 8-byte value is needed from the cache line in question. As a result, a total of 76 bytes must be read from main memory for every matrix nonzero. The arithmetic intensity is reduced to $\frac{1}{38}$ flop/B, and the performance is limited to merely 6.74 Gflop/s, as shown in Figure 2.

The ratio between the best and worst case performance described above is $\frac{38}{6} \approx 6.3$ times. In

[15] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Department of Computer Science School of Engineering and Applied Science, University of Virginia. 2013. URL: https://www.cs.virginia.edu/stream/.

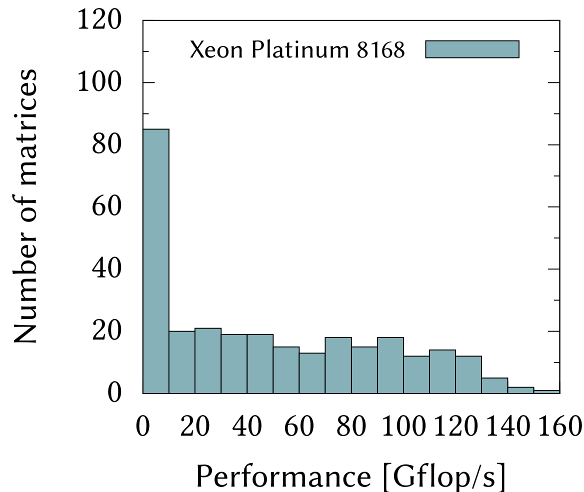


Figure 3 *Performance of sparse matrix-vector multiplication with the CSR storage format (see Algorithm 1) for 289 real matrices from the SuiteSparse Matrix Collection with more than one million nonzeros on a dual socket Intel Xeon Platinum 8168 (Skylake-X) multicore CPU system.*

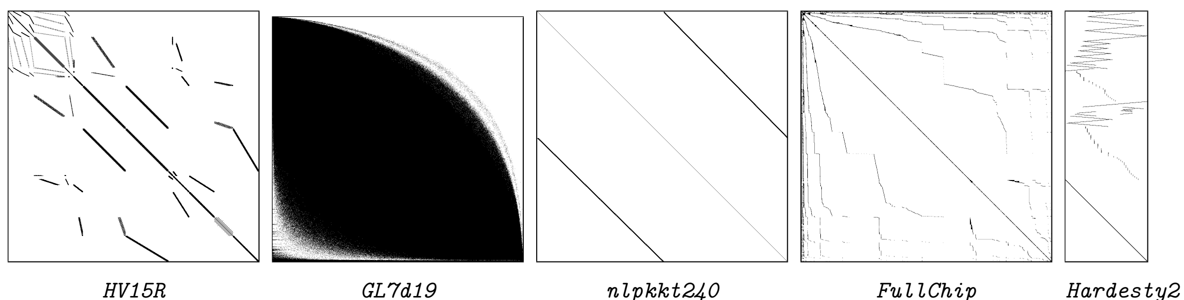


Figure 4 *Sparsity patterns of a few sparse matrices from the SuiteSparse Matrix Collection.*

reality, the performance of SpMV kernels, such as Algorithm 1, varies greatly depending on the matrix sparsity pattern (i.e., the location of the matrix nonzeros). This is illustrated by the spread in performance shown in Figure 3 for a selection of large, sparse matrices from the SuiteSparse Matrix Collection.[16] The performance is below 10 Gflop/s for almost a third of the matrices, but for the remaining matrices, the performance varies considerably, ranging all the way from 10 Gflop/s up to about 160 Gflop/s. To also demonstrate the diversity in the underlying sparsity patterns, Figure 4 shows the sparsity patterns of a few matrices that were used.

2.2 Sparse CARM: Improving Roofline Insightfulness for Sparse Computations

Cache-Aware Roofline Model (CARM)[17] provides insightful indication of performance upper-bounds of a micro-architecture with multiple levels in the memory hierarchy, which allows for visual application characterization, optimization and bottleneck detection. Similarly to other SoA roofline models, CARM is predominantly architecture-centric, i.e. it mostly considers architecture parameters (such as the maximum attainable memory bandwidth and peak compute throughput)

[16] Timothy Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–25. ISSN: 1557-7295. DOI: 10.1145/2049662.2049663.

[17] Ilic, Pratas, and Sousa, "Cache-aware Roofline model: Upgrading the loft".

when modeling the performance upper-bounds of a micro-architecture.

However, these performance maximums can only be attained by a small set of highly optimized and regular applications, or by carefully crafted assembly micro-benchmarks. In practice, the realistically attainable performance maximums do not only depend on the architecture capabilities, but also on the characteristics of the application (being executed on that architecture) and its ability to exploit those architecture maximums. For example, a naïve and non-vectorized SpMV kernel will never reached the maximum attainable performance that corresponds to the use of vector instructions.

To cope with these challenges and improve CARM insightfulness in these scenarios, we advocate herein a practical approach for CARM construction based on adCARM,[18] which specifically takes into account the application characteristics when determining the attainable maximums on a given architecture. The developed CARM-based model should also be able to adapt the characteristics of the input matrix, which can lead to an irregular memory access pattern, provoking performance degradation. This can cause even larger discrepancy between the application behavior and the modelled rooflines. Moreover, the kernel used for computations and the sparse matrix storage format must also be considered. With this aim, this work proposes a micro-benchmarking methodology to construct CARM for SpMV kernels, by relying on synthetic dense matrices stored in sparse formats, in order to obtain the maximum attainable bandwidth of the systems when performing SpMV computations. By using dense matrices, the micro-benchmarks sequentially access all positions in the vector, which indeed maximizes the memory bandwidth of a system. Finally, while the presented evaluation focuses on the MKL Single-Precision SpMV kernel (mkl_sparse_s_mv) and in the CSR format, the developed methodology can be extended to other kernels and formats.

Experimental results of this work were obtained in a Linux CentOS 7.5.1804 platform, with a eight-core Intel i7-7820X processor running at the fixed frequency of 3.60GHz, and 32GB of DRAM. All computation was performed using the Single-Precision sequential implementation of SpMV in the Intel® oneAPI Math Kernel Library[19] and hyper-threading and cache prefetching were disabled during testing.

2.2.1 Adapting CARM to SpMV

The first step to achieve a CARM-based approach for SpMV kernels is to determine which matrix dimensions correspond to the maximum attainable bandwidth of the micro-architecture when performing MKL SpMV. As the L1 cache is the memory level that guarantees maximum transfer rate between the core and the memory hierarchy, this preliminary test evaluates the performance of the MKL SpMV kernel with different sizes of involved data structures, such that the input sparse matrix (A, stored in CSR format), input vector (X) and output vector (Y) always fit in L1 cache. This evaluation was performed with a single-thread, where the single precision SpMV computation was executed multiple times to guarantee a total of at least 250ms of kernel runtime.

Preliminary bandwidth testing of several dense matrix dimensions, seen in Figure 5, shows that the matrices *20x64* and *32x64* as the dense parameters capable of delivering the highest performance when data structures involved in SpMV always fit in L1 cache. Considering the steady-state nature of these tests (i.e., SpMV kernel performed multiple times on warm caches), the streamed data structures in the SpMV kernel such as the CSR arrays (row pointer, column indices and non-zero values) and output Y vector, are only able to have reuse in a specific cache

[18] Diogo Marques et al. "Application-driven Cache-Aware Roofline Model". *Future Generation Computer Systems* 107 (2020), pp. 257–273. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2020.01.044. URL: https://www.sciencedirect.com/science/article/pii/S0167739X19309586.

[19] Intel Corporation. *Intel® oneAPI Math Kernel Library*. Intel. URL: https://software.intel.com/en-us/mkl.

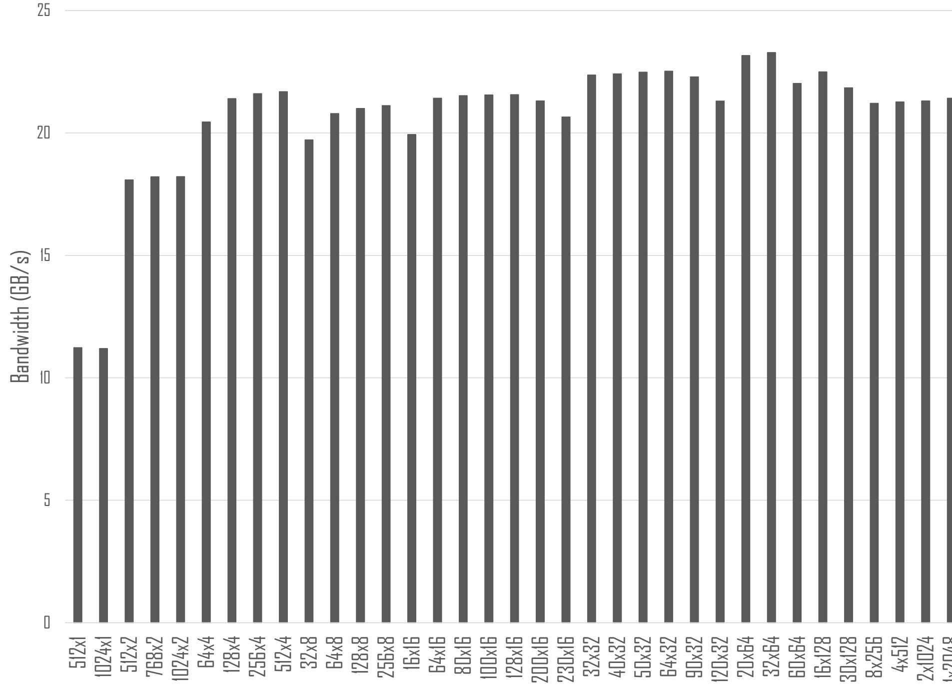


Figure 5 *Dense Preliminary Tests for Single Threaded MKL SpMV.*

level if the total memory occupied does not exceed the capacity of that cache level, while the input X vector, for a small column size (of 64), maintains locality at L1 cache level.

However, in order to extend this testing methodology to the other cache levels, while keeping the reuse of X vector in L1 cache, the amount of rows in the dense matrix (A) is increased while maintaining 64 columns, expanding the memory occupied by the data structures. It is also possible to extend this methodology to evaluate the scenarios when the reuse of X vector occurs beyond the L1 cache, since the X vector locality depends on whether a specific cache level is able to fully store it. For these dense tests, the previous test can be repeated by selecting more columns for A matrix (which result in increasing the amount of elements in X vector), thus exceeding the cache capacity to store the X vector, in order to test the attainable bandwidths for other cache levels when the X vector locality is in the L2 and L3 cache or DRAM.

Applying this testing methodology to the entire memory hierarchy, under single-thread execution leads to the bandwidth curves presented in Figure 6. As it can be observed, when X vector fits inside L1 cache (dense matrix with 64 columns), the bandwidth reduces as the size of the remaining structures surpass the size of each memory level. For example, when all structures fit in L1 cache, a maximum bandwidth around 23.3GB/s is achieved, while 19.82GB/s is attained for L2 cache accesses, i.e., when all other data structures (except the X vector) surpass the L1 cache size. Moreover, when considering different sizes of X, L2 bandwidth reduction also occurs when X fits in L2 cache (8704 columns), instead of L1 cache. Between the two execution scenarios, the bandwidth drops from 19.8Gb/s (when X fits in L1) to 17.8GB/s (when X fits in L2), confirming that the higher bandwidth is obtained when X elements are reused from the memory level closer to the core. On the other hand, for L3 cache and DRAM, the attainable bandwidth is similar for all the different X vector sizes, since in these memory levels, the latency in accessing other data structures is high enough to not be affected by the locality of X.

This micro-benchmarking principle is also adopted for multi-threaded execution, where each

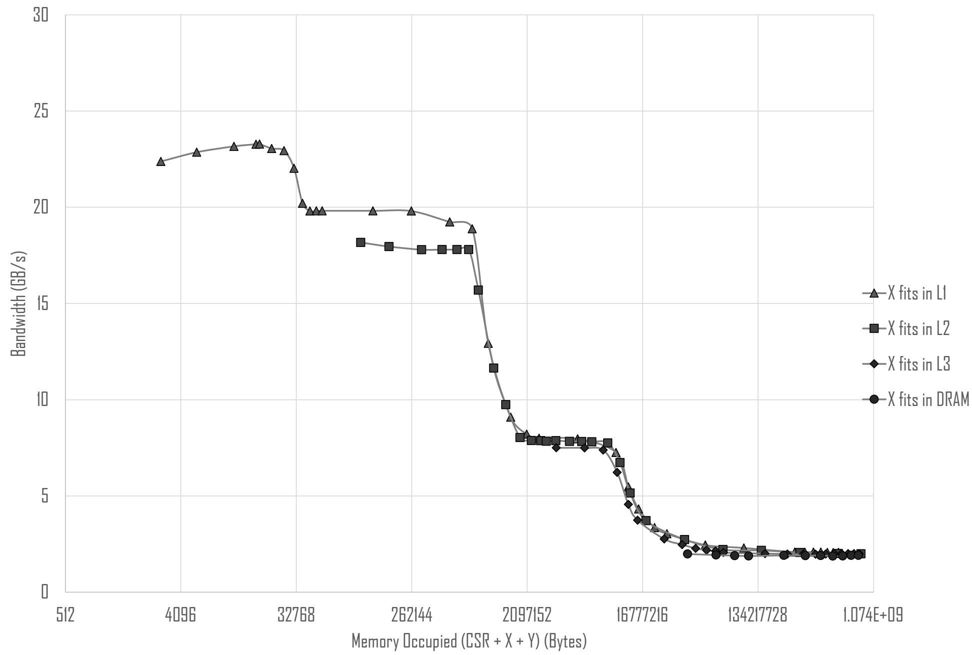


Figure 6 *Bandwidth Curve for Single Threaded MKL SpMV.*

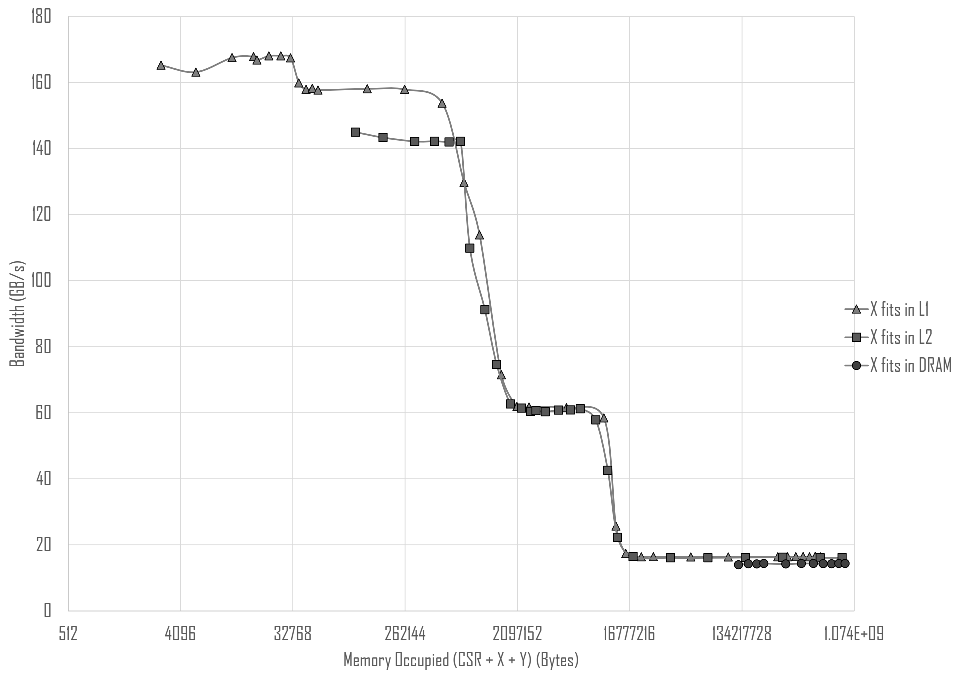
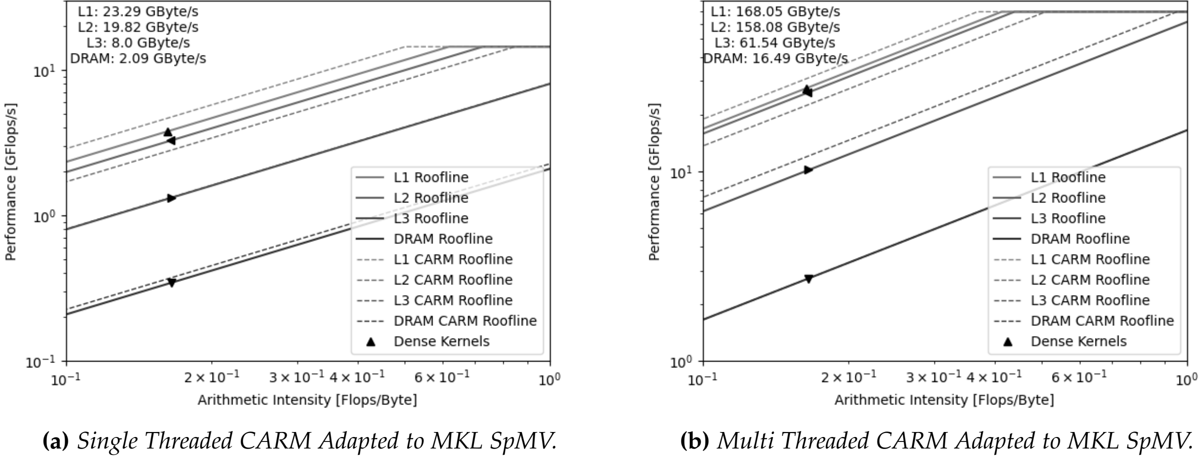


Figure 7 *Bandwidth Curve for Multi Threaded MKL SpMV (8 cores).*

thread shares the accesses to the sparse matrix A, while X and Y vectors are private to each thread. As it can be observed in Figure 7, the bandwidth in a multi-threaded (8 cores) environment shows similar behaviour as the single-threaded counterpart, with the memory bandwidth decreasing as



(a) *Single Threaded CARM Adapted to MKL SpMV.* **(b)** *Multi Threaded CARM Adapted to MKL SpMV.*

Figure 8 *SpMV Adapted Cache Aware Roofline Model.*

the accesses are served by memory levels further away from the core. As previously elaborated, the locality of X also affects the attainable L2 bandwidth, resulting in the bandwidth reduction of around 9% when X fits in L2 cache when compared to the scenario when X fits in L1 cache. Compared to the single-core tests, the bandwidth increases for all memory levels, achieving a maximum bandwidth of around 168, 158, 61.5, and 16.5 GB/s, for L1, L2, L3 and DRAM, respectively.

From the bandwidth values obtained in the tests presented in Figures 6 and 7, novel roofline models can be derived that better represent the performance upper-bounds of the SpMV kernel. The proposed roofline models are presented in Figures 8a and 8b, for single- and multi-threaded (8 cores) execution scenarios, respectively. Compared to the CARM roofs that correspond to scalar single-precision memory transfers (*i.e.*, the instructions used by MKL single-precision SpMV), the proposed rooflines have a lower L1 performance upper-bound given that the kernel is not only a streaming benchmark of the CSR and Y arrays but also contains the indirect access of the X vector, and these dependencies degrade the performance when access to the data is faster given the locality at the L1 cache level. When considering the proposed L2 rooflines, the performance is higher than the CARM roofs given that the locality of the X vector is preserved at the L1 while in the original roofs, streaming tests would have locality only in L2 cache. L3 and DRAM rooflines show little differences in performance.

2.2.2 Impact of Sparse Matrix Characteristics in Sparse CARM

Code in Listing 1 presents a naïve implementation of the SpMV computation kernel applied to a sparse matrix A represented in a CSR format, where vector *vals* contains the value of each row-major stored non-zero element, *cols* contains the column coordinate of each element, and *pointerB* and *pointerE* store the index of the first and last non-zero element in each row, respectively. Based on this implementation, the total number of loads and stores executed based on the characteristic of the matrix can be found in Table 1, and multiplying the sum of instructions by β_i (average number of bytes transfered per instruction), the total amount of bytes transferred to the core is obtained. Knowing that a SpMV kernel performs $2 \times$ NNZ floating-point operations, with NNZ being the number of non-zero elements in the sparse matrix, the Arithmetic Intensity (AI) (as perceived by CARM) can be derived from equations 3 and 4.

Memory Accesses		
Array	*Num LD*	*Num ST*
y	N_{rows}	N_{rows}
pointerB	N_{rows}	—
pointerE	N_{rows}	—
vals	N_{nnz}	—
cols	N_{nnz}	—
X	N_{nnz}	—

$$AI = \frac{\text{Floating Point Operations}}{\text{Memory Accessed (bytes)}} \quad (2)$$

$$AI = \frac{2 \times N_{nnz}}{\beta_i \times (3 \times N_{nnz} + 4 \times N_{rows})} \quad (3)$$

$$AI = \frac{2}{\beta_i \times (3 + 4 \times \frac{N_{rows}}{N_{nnz}})} \quad (4)$$

AI Limits	
Minimum $AI(\frac{N_{nnz}}{N_{rows}} = 1)$	$\frac{2}{\beta_i \times 7}$
Maximum $AI(\frac{N_{nnz}}{N_{rows}} = \infty)$	$\frac{2}{\beta_i \times 3}$

Table 1 *Arithmetic Intensity of Naive SpMV.*

```
for(int i = 0 ; i < n_rows ; i ++){
    y[i] = tmp;
    for(int j = pointerB[i]; j < pointerE[i]; j++){
        tmp += vals[j] * X[cols[j]];
    }
    y[i] = tmp;
}
```

Listing 1: Naïve SpMV Kernel

From these equations, it is possible to conclude that differently from most applications, CARM AI of SpMV depends on the characteristics of the input matrix, in particular, the number of non-zeros per row (NNZ/N_{rows}). In the case where there are no empty rows, the minimum AI corresponds to $NNZ/N_{rows} = 1$, while maximum AI occurs when $NNZ/N_{rows} \to \infty$. Moreover, since the performance in the memory region of the roofline models depends in the AI, the dependency between the AI and the characteristics of the input matrix indicates that different matrices have distinct performance upper-bounds.

This range of AI can be experimentally verified by relying on dense synthetic matrices with different dimensions. Figure 9 presents this experimental evaluation, where highlighted in grey is the AI range attainable using the MKL SpMV in single precision ($\beta_i = 4$). As it can be observed, the minimum AI corresponds to the dense matrices with 2 columns ($AI \approx 0.14152$). 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 ($AI \approx 0.16666$).

2.2.3 Improving SpMV performance by reordering

Given the memory access pattern of SpMV, the performance gains only occur if the accesses to vector X are improved, *e.g.*, X elements are reused in caches and/or accesses to X are coalesced. To achieve this, reordering techniques can be applied to the sparse matrix, in order to provide a more regular access pattern to X with improved data reuse. In order to evaluate the maximum performance gains that can be obtained with reordering, this work proposes a strategy that involves the creation of pairs of synthetic matrices. One of the matrices corresponds to the *worst*

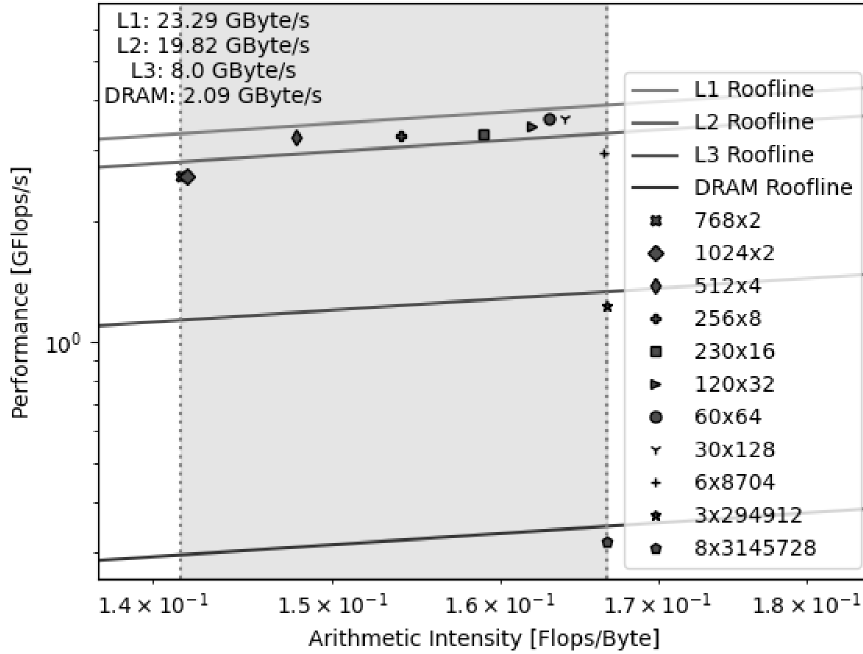


Figure 9 *AI Range Tests in MKL SpMV.*

case, which aims at minimizing the reuse of X values as much as possible. On the other extreme, the second matrix mimics the *best case* execution scenario, where the locality on X is maximized. In each pair, the best case scenario must correspond to a specific permutation of rows and columns of the worst case scenario.

In order to maximize the reuse of elements in the X vector, the matrix for the *best case* scenario contains several dense blocks in the diagonal, each with $p \times q$ elements. This organization aims at guaranteeing the maximum data locality and reuse of X at the level of each block. In particular, when processing a dense block, the q elements of X are fetched when processing the first row, and subsequently reused for each p row processed. To allow for maximum reuse, the value of q is selected such that the part of X accessed in each block fits in L1 cache, and it is a multiple of cache line size. With this structure for the best matrix, the worst case matrix must contains q non-zero elements per row, and p non-zero elements per column, in order to be transformed into the best case through row and column permutation.

An example of the *worst case* matrix creation is presented in Figure 10, which considers blocks of a size 1×64. To avoid reuse of elements within a cache line, each of the 64 non-zero elements (in a single row) must be separated by at least a cache line size (e.g., 16 elements in single precision). This fact requires choosing a column size that allows for the even distribution of the elements (e.g., 4096 columns allows that each non-zero element is separated by $4096/64 = 64$ positions). To also avoid cache line re-utilization between rows when accessing X vector, each non-zero element of the next row is shifted by one cache line to the right. This is repeated until all the cache-lines are exhausted, i.e., the maximum amount of cache lines that can be allocated between two consecutive elements in a single row. For example, when considering 4096 columns and 64 positions between each non-zero in a single row, a total of 4 rows are needed to repeat this pattern (since 64 positions correspond to 4 cache lines, i.e. $64/16 = 4$). This distribution can

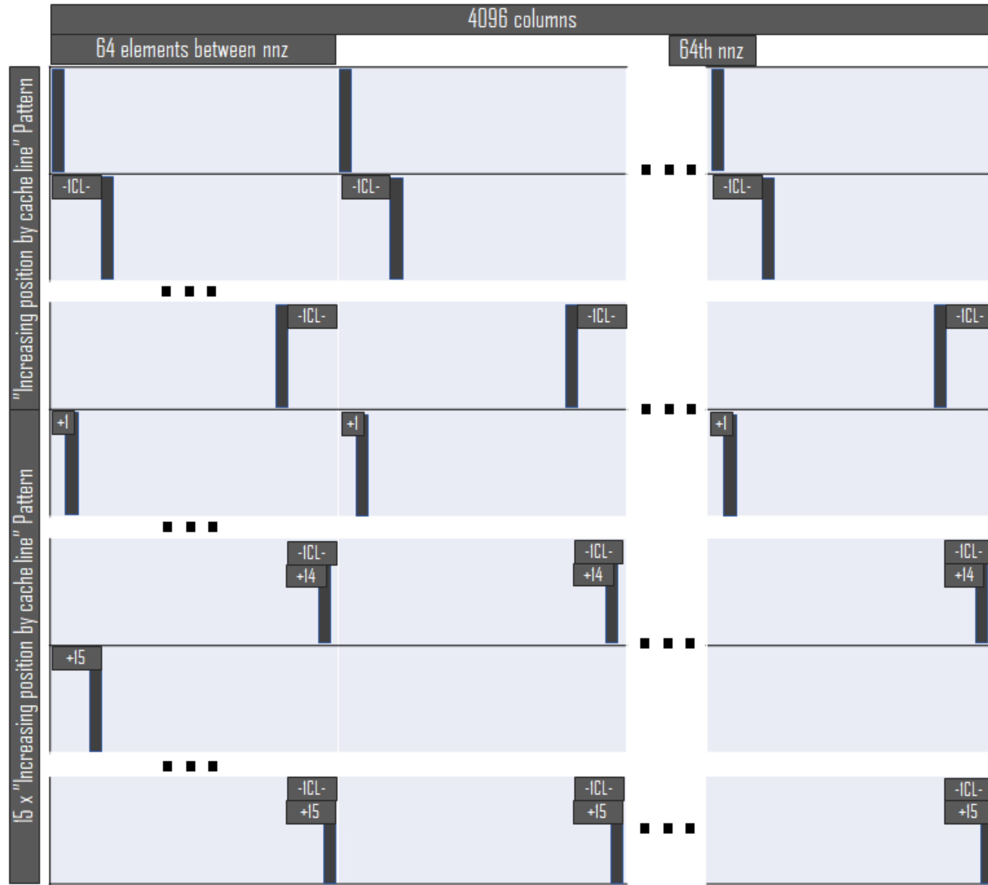


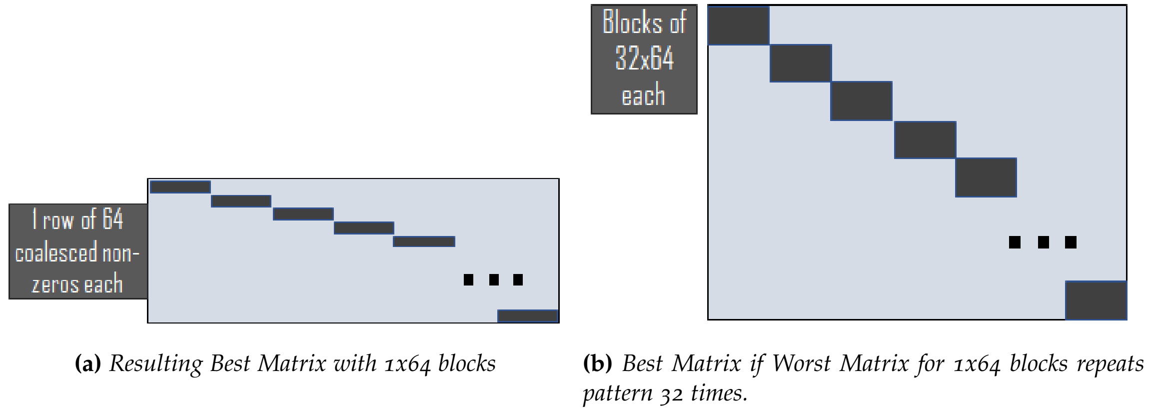
Figure 10 *Worst Matrix Example for 1x64 blocks*

be observed in Figure 10 within the first block of 4 rows described as "*Increasing position by cache line* Pattern".

This pattern for block of rows is repeated for the next group of 4×64 non-zero elements, where the position of each non-zero corresponds to the one from the previous block of rows, but incremented by one position. This distribution can be observed in Figure 10 being repeated for the next 15 blocks of rows described as "15 x *Increasing position by cache line* Pattern". Each of the blocks of 4 rows increase the position of each non-zero element by one until all elements of the X vector are accessed by the *worst case* matrix. This allows for the reordering of this non-zero distribution into a pattern as shown in Figure 11a. Having this distribution being repeated 32 times enables the reordering into the best case matrix with the diagonal of 32x64 blocks (see Figure 11b).

Using this strategy, for a specific dense block dimension of the best case matrix (i.e., $p \times q$), the only other parameter to choose is the amount of columns, which should preferably be divisible by the column size of the blocks for even distribution in the worst case matrix. The NNZ and row size of the matrix depends on the amount of blocks that fit in the best case matrix diagonal with the chosen block dimensions and column size.

Figure 12 presents the evaluation of several best and worst case matrices in the Sparse CARM, for single threaded workloads with single precision data. Since according to the preliminary testing presented in Figure 5 the maximum bandwidth is achieved for a block size of 32×64, this



(a) *Resulting Best Matrix with 1x64 blocks*

(b) *Best Matrix if Worst Matrix for 1x64 blocks repeats pattern 32 times.*

Figure 11 *Resulting best matrices depending on Worst matrix layout.*

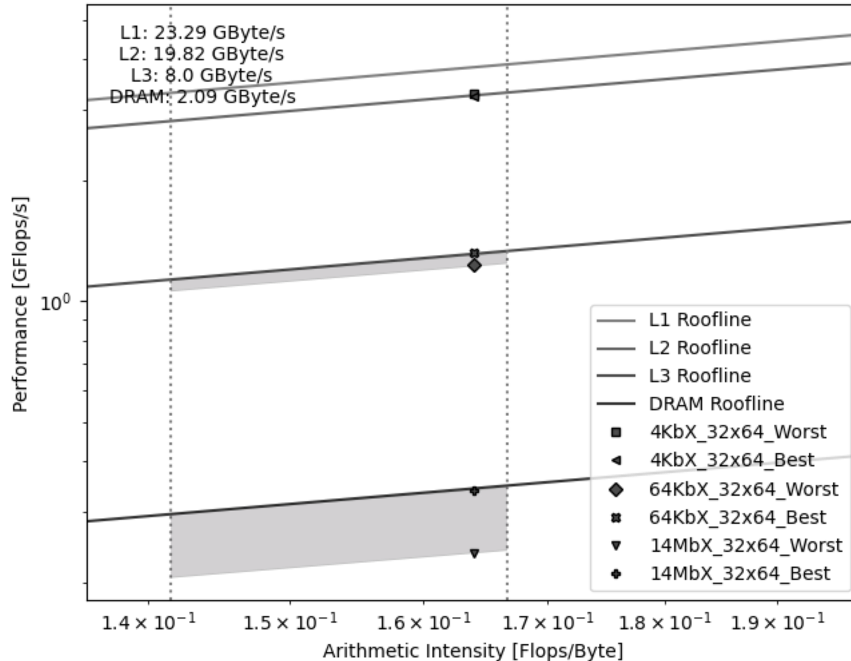
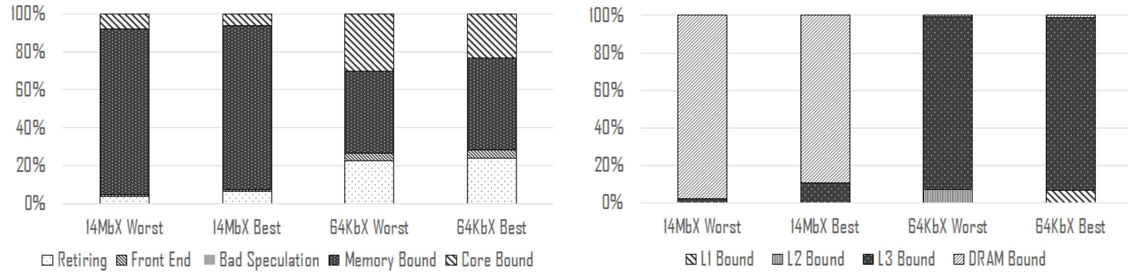


Figure 12 *ST-Best case and Worst case Matrices for 32x64 blocks*

experimental evaluation is performed with $p = 32$ and $q = 64$ (Figure 11b).

As it can be observed in Figure 12, the groups of matrices that fit in DRAM (14MbX) and L3 (64KbX) memory levels reveal that significant performance improvements can be achieved between the worst and the best cases. However, the group fitting in L2 cache (4KbX), which also are the smallest matrices that are able to be created for a block of 32×64 using this strategy, show little to no performance improvement. This effect occurs due to data re-utilization between the rows that have the non-zero elements only one column apart from each other, reducing the gap in cache reuse between the worse and best case matrices.

To further verify the obtained results, the Intel VTune Top-Down breakdown is presented in Figure 13. When analyzing the first level of the Intel VTune Top-Down results presented in Figure



(a) *Top Down VTune Analysis of ST Best Worst Matrices* **(b)** *Memory Bound VTune Analysis of ST Best Worst Matrices*

Figure 13 *VTune Analysis of Synthetic Worst/Best Matrices in single-thread.*

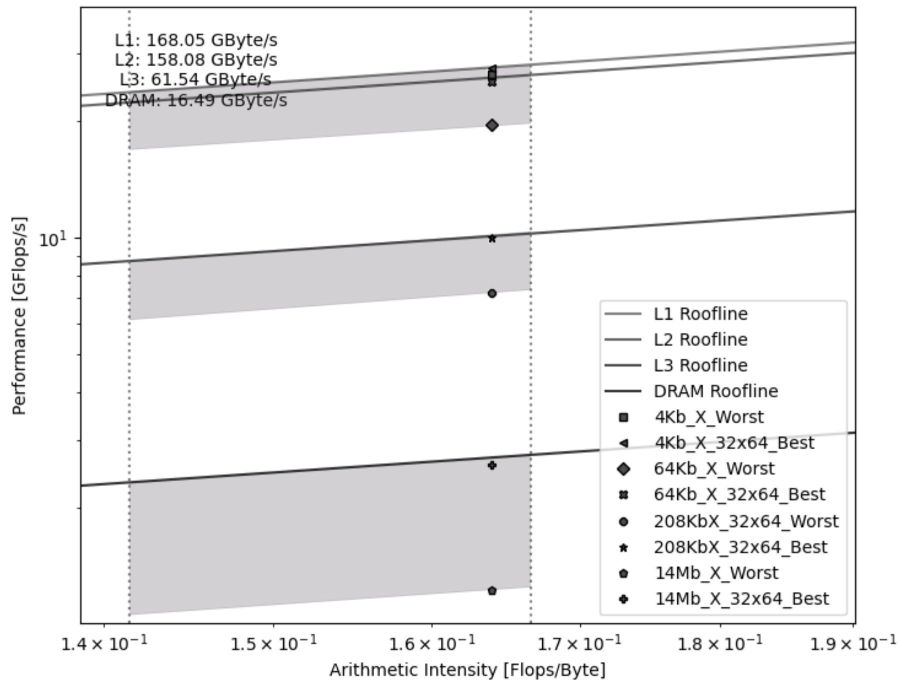
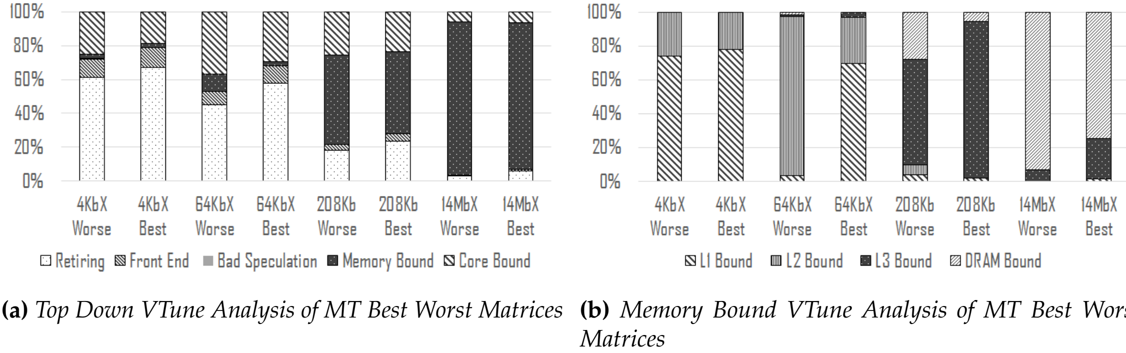


Figure 14 *MT-Best case and Worst case Matrices for 32x64 blocks*

13a, it is possible to verify that there is an increase in the retiring component from the worst case scenario to the best case matrix. Moreover, the memory bound breakdown of Top-Down (Figure 13b) shows an increase of the L3 Bound component for the 14MbX best case matrix, and the appearance of the L1 Bound component for the 64KbX matrix. Thus, it is possible to conclude that the performance improvement achieved by reordering the worst matrices arises from better cache utilization, due to a more regular access pattern.

This strategy is also applied to a multi-threaded (8 cores) scenario, by performing static row partitioning of the input sparse matrix among threads, with shared access to X and Y vector. As it can be observed in Figure 14, similar to the single-thread test, there is a clear improvement in the SpMV performance when reordering the worst case matrix, especially for L3 cache (208KbX) and



(a) *Top Down VTune Analysis of MT Best Worst Matrices* **(b)** *Memory Bound VTune Analysis of MT Best Worst Matrices*

Figure 15 *VTune Analysis of Synthetic Worst/Best Matrices in multi-thread.*

DRAM (14MbX). For example, from worst case to the best case, a maximum speedup of around 2.10x for DRAM matrices, and 1.39x for L3 matrices can be expected. With this row partitioning, each thread will only transfer 1/8th of the matrix, which will change the locality of some of the matrix groups. This effect can be observed for the 64KbX matrices, which were located below the L3 bandwidth roof for the single-threaded tests, while in the multi-threaded tests they are positioned below the L2 roofline, thus indicating a maximum speedup of 1.28x for matrices that have locality in this cache level.

The increase in the performance is once more due to better locality and regularity in the memory accesses to vector X. As observed in the first level of Top-down analysis (see Figure 15a), there is an increase in the retiring component for all tested matrices when moving from the worst case to the best case execution scenarios, indicating that after reordering the core is being more utilized. In the memory bound breakdown (Figure 15b), it is also possible to observe the effects of reordering in the accesses to the memory hierarchy. In the case of the 4KbX and 64KbX matrices (that aim at exercising the reordering upper-bounds for L1 and L2 caches, respectively), there is an increase in the L1 bound component after reordering, thus leading to increased performance. In the case of the larger matrices, *i.e.*, 208KbX and 14MbX, the contribution of the L3 bound increases greatly after the reordering, while the DRAM bound component reduces. In all four test cases, both Top-Down and Sparse CARM show that reordering the input matrix can provide significant speedups by just improving the accesses to vector X, leading to higher utilization of components closer to the core.

In order to represent the possible speedups for matrices with locality focused on each specific cache-level, the ranges of performance gain obtained with the best and worse case matrices for both single and multi-threaded scenarios are highlighted in grey in Figures 12 and 14, respectively.

Given that the previous evaluation with synthetic matrices has shown the potential to achieve significant speedups in SpMV by reordering the input sparse matrix, this work also provides an evaluation of the MKL SpMV performance when using real matrices reordered with state-of-the-art algorithms, for single and multi-threaded execution. The considered reordering algorithms are Reverse Cuthill-McKee (RCM),[20] Approximate Minimum Degree (AMD),[21] Nested Dissection (ND),[22] a partial implementation of GrayRO,[23] and on two matrices, the reordering algorithms

[20] E. Cuthill and J. McKee. "Reducing the Bandwidth of Sparse Symmetric Matrices". Association for Computing Machinery, 1969.

[21] 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.

[22] Alan George. "Nested Dissection of a Regular Finite Element Mesh". *SIAM Journal on Numerical Analysis* 10.2 (1973), pp. 345–363.

[23] Haoran Zhao et al. "Exploring Better Speculation and Data Locality in Sparse Matrix-Vector Multiplication on

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 2 *Real matrices retrieved from SuiteSparse.*

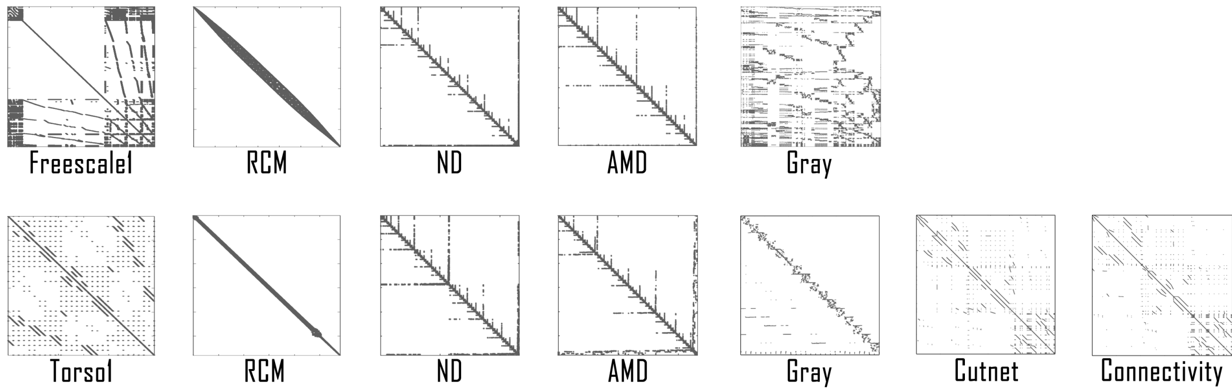


Figure 16 *Reordering Algorithms applied to Matrices*

included in the Patoh partition library:[24] cutnet and connectivity. A set of eleven matrices from SuiteSparse are considered for evaluation, as presented in Table 2. 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. Examples of how different reordering algorithms affect the disposition of the non-zero elements can be seen in Figure 16, *e.g.* in the matrices where RCM was used, it can be seen how the non-zero elements are more distributed along the diagonal and this reduction in the spread of the elements may provide performance benefits due to more coalesced accesses to the X vector.

Figure 17 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, Fullchip attained a speedup of 1.08x with ND, while RCM provided 1.17x speedup for Stanford. Further insights can be obtained through the Intel VTune Top-Down. As it can be observed with the Top-Down method for Freescale1, Stanford and FullChip (Figure 18), Freescale1 reordering with ND and Standford reordered with RCM

Intel Xeon". *2020 IEEE 38th International Conference on Computer Design (ICCD)*. 2020.

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

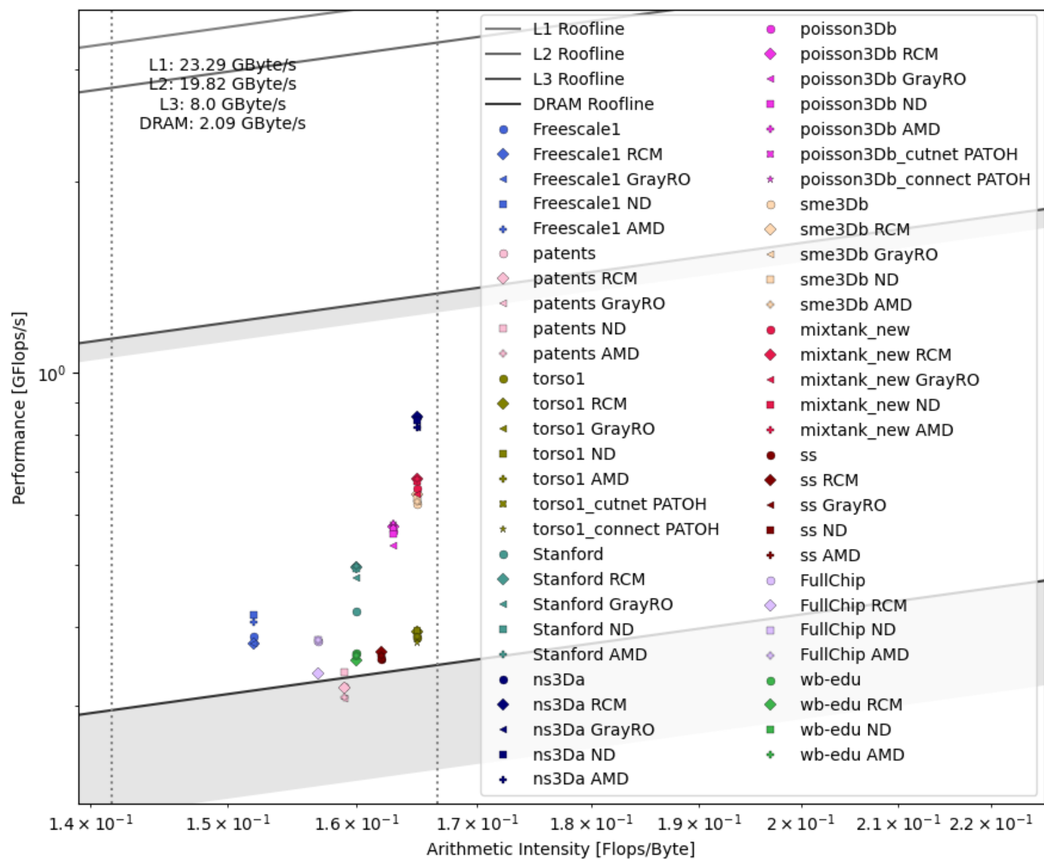
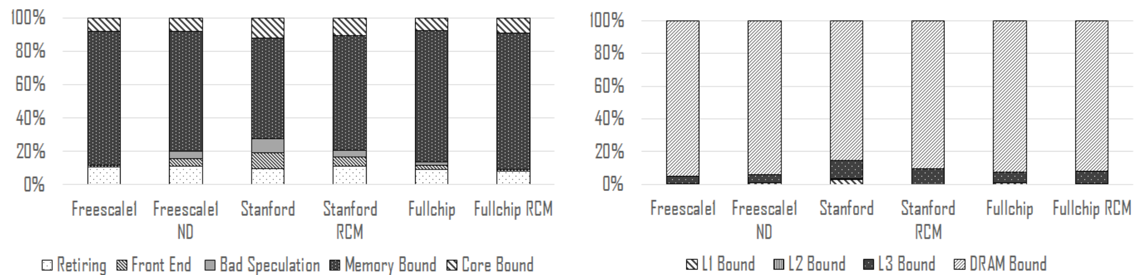


Figure 17 *ST Real and Reordered Matrices*



(a) *Top Down VTune Analysis of ST Real Matrices* **(b)** *Memory Bound VTune Analysis of ST Real Matrices*

Figure 18 *VTune Analysis of some real and reordered matrices in single-thread.*

have an increase in the retiring contribution, and a reduction in the memory bound component. This indicates that the bottlenecks after reordering are slightly more related to the core, which explains their performance increase of 8% for Freescale1 and 17% for Stanford. On the other hand, FullChip with RCM has a reduction in the retiring component together with an increase in the memory bound and core bound, which leads to a performance decrease of 11%.

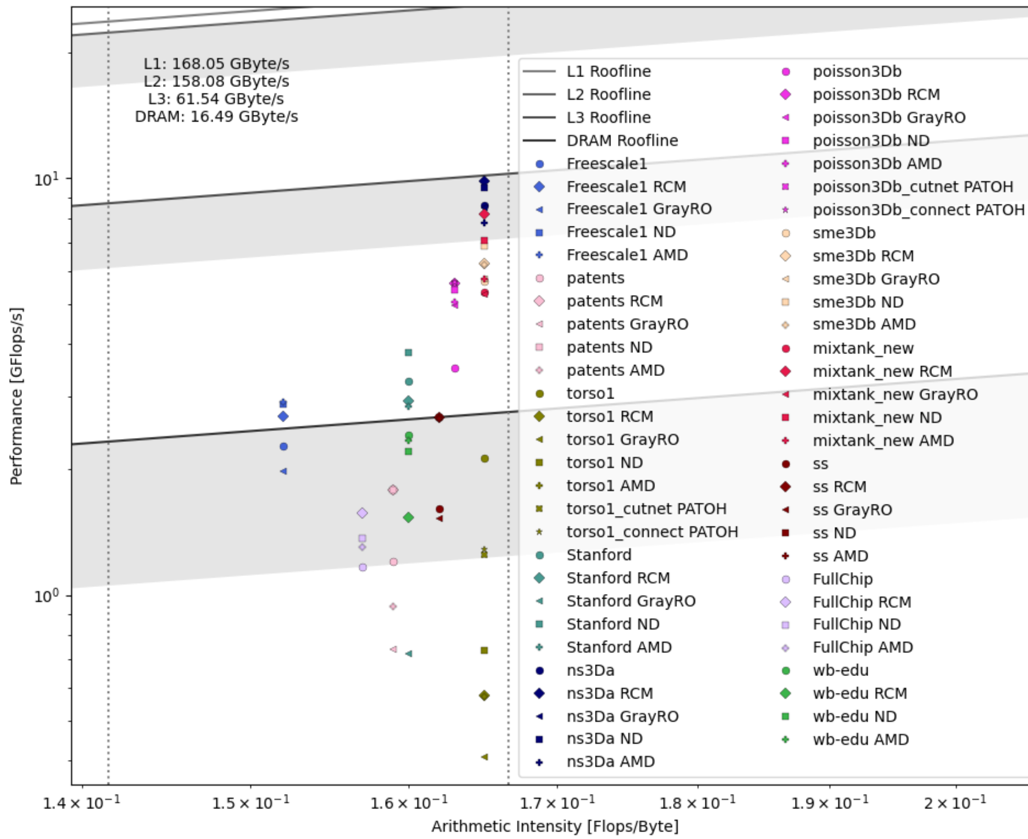
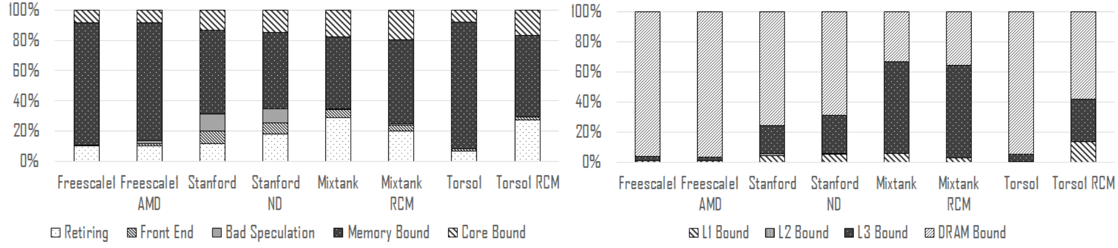


Figure 19 *MT Real and Reordered Matrices*

The same matrices and reordering algorithms were also tested in multi-threaded fashion for 8 threads total, 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 19) show a similar trend to the single-thread experiments, since the reordering algorithms can either provide speedups or slowdowns depending on the matrix. Moreover, in multi-thread execution is also necessary to take into account the impact of load balancing in the performance of the application. Since the experiments proposed in this work focused on an even partition of the rows between threads, the number of non-zero elements performance by each thread may differ, resulting in data imbalance, degrading performance. By observing the average core utilization of each matrix, presented in Figure 21, 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 other in may provoke serious imbalance issues. For example, matrix Poisson3Db has an increase of the average core utilization from 3.6 to 6.2 when using RCM. On the other hand, Torso1 has a reduction from 5.3 to 1.35 in the core utilization when using RCM. This explains their characterization in the Sparse CARM, since Poisson3Db has a speedup of 1.6x and torso1 a slowdown of 0.27x.

All these effects can also be observed in the Top-Down method, presented in Figure 20.



(a) *Top Down VTune Analysis of MT Real Matrices* **(b)** *Memory Bound VTune Analysis of MT Real Matrices*

Figure 20 *VTune Analysis of some real and reordered matrices in multi-thread.*

For example, applying AMD to Freescale1, results in performance gain, although no noticeable changes occur in the Top-Down metrics. Thus, the speedup in this matrix is mainly due to the improved load balancing attained after reordering, as the average core utilization increases from 6.31 to 7.95. In contrast, reordering Stanford with ND algorithm improves the locality of the accesses to X vector, indicated by the increasing in the retiring and L3 bound components. Despite ND causing load imbalance (average core utilization drops from 5.92 to 4.7), the performance of this matrix still improves after reordering. For Mixtank_new matrix, applying RCM deteriorates locality of accesses to X vector, as can be seen by the increase of the memory bound component and L1 bound indication, which still results in a performance gain, due to improvement of load balancing from 3.92 to 6.87. Analysing the effect of RCM applied to torso1, locality of X is greatly improved as the memory bound component lowers significantly while retiring increases, and observing the changes in memory bound analysis, the DRAM bound indication reduces with increased L1 and L3 components. Despite this, as previously mentioned, torso1 reduces the core utilization to 1.35 when applied RCM reordering algorithm, resulting in slowdown due to inferior resource utilization.

Highlighted in grey in Figures 17 and 19 are the performance improvement intervals expected when reordering is applied, which are obtained from the synthetic worse and best case matrices by considering different matrix sizes that guarantee locality at a specific cache level. As it can be observed in Figure 17, the majority of the real matrices in the single-threaded scenario are positioned above the DRAM roof and below the performance improvement interval for the L3 cache, which indicates that tested matrices are only partially exploiting locality in L3 cache, while still being limited by the DRAM accesses. In the multi-threaded scenario (see Figure 19), some matrices are positioned within the performance intervals for both L3 cache and DRAM. However, one can also observe that some matrices are placed below the highlighted performance improvement interval, *e.g.* RCM, ND and Gray reordering of torso1 matrix are positioned below the DRAM interval. Besides the potential poor data reuse, in these cases, the sparse CARM characterization also suggests the load balancing issues, which prevents the computation performed on the reordered matrix to exploit the performance limits corresponding to the 8 cores execution.

Observing analysed results on reordering as a performance improvement method for SpMV, synthetic best and worst case matrices are able to represent possible performance gains depending on the memory locality. While multi-threaded testing exhibits higher performance gains compared to single-threaded, when utilizing real matrices to tests these margins, load balancing proves to be an impediment in correlating improved locality through reordering and performance optimization, as the former can also affect how balanced the workload is split among working threads. Future work on this subject will focus on ways to represent load balancing in CARM for multi-threaded tests as well as the usage of partitioning techniques combined with reordering

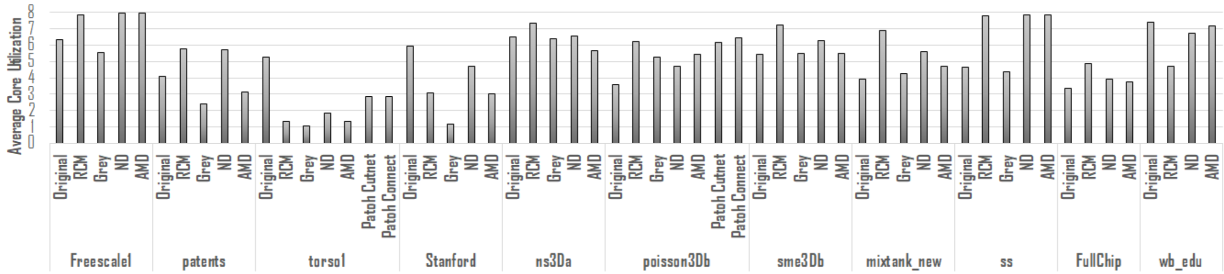


Figure 21 *Average Core Utilization for Multi-threaded testing*

algorithms for a more clear analysis on how performance is impacted by changes in cache locality without load balancing being a factor.

2.3 MANSARD ROOFLINE MODEL: SPARSE KERNELS ANALYSIS

When modeling the performance upper-bounds, SoA roofline models[25] may oversimplify the back-end of the micro-architectures by just focusing on a subset of functional units and the maximum attainable bandwidth of different memory hierarchy levels. As such, those models do not consider other hardware components that may limit the performance in any Out-of-Order (OoO) processor, especially the ones related to the retirement of instructions, such as, number of Retirement Slots (RS), Reorder Buffer (ROB) and Physical Register File (PRF). Instead, these models evaluate the performance upper-bounds by only considering the isolated performance limits of the different hardware resources, thus giving the illusion of infinite retirement and OoO windows. Moreover, when concurrently executing non-memory and memory instructions, the limited capacities of ROB and PRF can constraint the number of in-flight memory requests, preventing applications from achieving maximum memory bandwidth and increasing the impact of memory latency, especially when accessing the "slower" memory levels. This can hinder the ability of roofline models to provide accurate characterization of applications that suffer from latency issues. This is the case of several sparse kernels, that due to their irregular memory patterns are highly likely to have bottlenecks related to the latency of Last Level Cache (LLC) and DRAM.

2.3.1 MANSARD ROOFLINE MODEL

To address the main drawbacks of the SoA roofline models, the Mansard Roofline Model (MaRM)[26] considers all the instructions retired by an application, representing performance as Instructions Retired per Cycle (IPC). Since MaRM uses the instruction domain instead of the operations domain, its AI differs from standard roofline models. In MaRM, the AI is defined as the number of non-memory instructions (I_{NM}) over the number of memory instructions (I_M). Thus, the ridge point of memory level 'y' (R^y), *i.e.*, the point where memory transfers and computations are completely overlapped in time, is represented as the performance of the non-memory instructions (IPC_{NM}) over the maximum sustainable bandwidth of memory level 'y' ($IPC^y_{M,Max}$), where $y \in \{L1, L2, \ldots, LLC, DRAM\}$.

For representing performance as IPC, it is indispensable to incorporate the retirement limits

[25]Williams, Waterman, and Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures"; Ilic, Pratas, and Sousa, "Cache-aware Roofline model: Upgrading the loft".

[26]Diogo Marques, Aleksandar Ilic, and Leonel Sousa. "Mansard Roofline Model: Reinforcing the Accuracy of the Roofs". *ACM Trans. Model. Perform. Eval. Comput. Syst.* 6.2 (2021). ISSN: 2376-3639. DOI: 10.1145/3475866. URL: https://doi.org/10.1145/3475866.

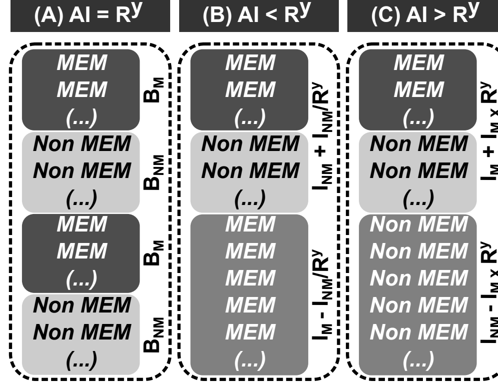


Figure 22 *Execution scenarios for different application types.*

of OoO Central Processing Units (CPUs). Due to the limited number of retirement slots, micro-architectures can only retire a limited number of instructions per cycle. Hence, the maximum attainable performance ($P_a^y(AI)$) is limited by the maximum dispatch rate (D_M) of the micro-architecture such that

$$P_a^y(AI) \leqslant D_M, \tag{5}$$

where D_M was 4 instructions per cycle in previous Intel micro-architectures, and 5 instructions per cycle in the newest Sunny Cove core architecture.

To include the impact of the ROB in MaRM, the three execution scenarios in Figure 22 are considered. These execution scenarios are representative of applications or application kernels with different AIs (expressed in $\frac{I_{NM}}{I_M}$), which according to roofline approaches are able to attain the maximum performance in different regions of the models. Scenario **A** fully overlaps memory and non-memory instructions, *i.e.*, $AI = R^y$; in Scenario **B**, applications are memory bound ($AI < R^y$); and Scenario **C** portrays workloads in the compute bound region ($AI > R^y$).

At the ridge point (scenario A), the ROB has several blocks containing B_M memory instructions and $B_{NM} = B_M \times R^y$ non-memory instructions. For example, an application with an average of 2 loads per store has $B_M = 3$ and $B_{NM} = 3R^y$. Since the amount of in-flight memory requests (IF_M) corresponds to the total number of memory instructions in the ROB, IF_M can be calculated as

$$IF_M = B_M \times \left\lceil \frac{ROB_{eff}}{B_M + B_{NM}} \right\rceil = B_M \times \left\lceil \frac{ROB_{eff}}{B_M \times (R^y + 1)} \right\rceil, \tag{6}$$

where ROB_{eff} is the effective ROB size.

When accessing high latency memory levels, the blocks $B_M + B_{NM}$ that fit in the ROB can only start retiring after the first memory request is completed. Thus, the execution time of the blocks $B_M + B_{NM}$ contained in the ROB is approximately the latency of the corresponding memory level 'y' ($Lat^y(IF_M)$), thus the performance of the micro-architecture at the ridge point ($IPC_R^y(IF_M)$) is given by

$$\begin{aligned} IPC_R^y(IF_M) &\approx \frac{IF_M \times (R^y + 1)}{Lat^y(IF_M)} = \\ &= (R^y + 1) \times IPC_M^y(IF_M), \end{aligned} \tag{7}$$

where $IPC_M^y(IF_M)$ is the effective memory bandwidth of the level 'y' restricted by the number of in-flight requests. This parameter is related to $Lat^y(IF_M)$ through Little's Law ($IPC_M^y(IF_M) \times$

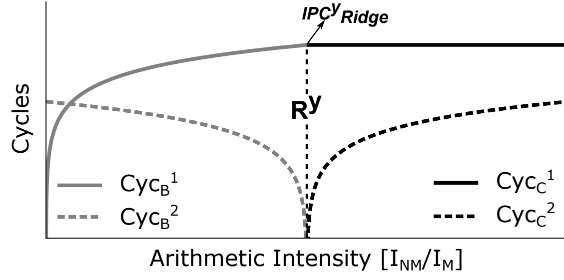


Figure 23 *Illustration of cycles of the proposed MaRM.*

$Lat^y(IF_M) = IF_M)$.[27]

Memory bound (scenario B) and compute-bound (scenario C) applications also aim at overlapping non-memory and memory instructions, but since their AI is different from R^y they consist of two components each. The first one overlaps memory and non-memory instructions in the same fashion as scenario A. The second component only contains instructions of a single type. Since in scenario B, the AI is lower than R^y, there are more memory instructions than non-memory instructions. Thus, the first component contains all non-memory instructions overlapped with $\frac{I_{NM}}{R^y}$ memory transfers, corresponding to a total of $I_{NM} + \frac{I_{NM}}{R^y}$ instructions, while the second component contains the remaining memory instructions, *i.e.*, $I_M - \frac{I_{NM}}{R^y}$. Since the overlapping component is organized as in scenario A, it is executed at the rate of $IPC_R^y(IF_M)$. In the second component, instructions are retired at the maximum sustainable memory bandwidth ($IPC^y_{M,Max}$), since there are enough requests in the ROB to attain the maximum bandwidth. Hence, the number of cycles necessary to execute a memory-bound application (Cyc_B) is given by:

$$\begin{aligned} Cyc_B^y(AI, IF_M) &= \frac{I_{NM} + \frac{I_{NM}}{R^y}}{IPC_R^y(IF_M)} + \frac{I_M - \frac{I_{NM}}{R^y}}{IPC^y_{M,Max}} = \\ &= \frac{AI \times I_M(1 + \frac{1}{R^y})}{IPC_R^y(IF_M)} + \frac{I_M \times (1 - \frac{AI}{R^y})}{IPC^y_{M,Max}}. \end{aligned} \tag{8}$$

On the other hand, the first code portion of compute-bound applications (scenario C) contains all memory instructions overlapped with $I_M \times R^y$ non-memory instructions, *i.e.*, a total of $I_M + I_M \times R^y$ instructions, retired at the speed of $IPC_R^y(IF_M)$. The second component contains $I_{NM} - I_M \times R^y$ non-memory instructions retiring at the performance of the non-memory instructions (IPC_{NM}). Thus, the number of cycles necessary to execute the application C (Cyc_C) can be calculated as:

$$\begin{aligned} Cyc_C^y(AI, IF_M) &= \frac{I_M + I_M \times R^y}{IPC_R^y(IF_M)} + \frac{I_{NM} - I_M \times R^y}{IPC_{NM}} = \\ &= \frac{R^y \times I_M(1 + \frac{1}{R^y})}{IPC_R^y(IF_M)} + \frac{I_M \times (\frac{AI}{R^y} - 1)}{IPC^y_{M,Max}}. \end{aligned} \tag{9}$$

Equations 8 and 9 are illustrated in Figure 23, each represented by their respective components, *i.e.*, Cyc_B^1 and Cyc_B^2 from $Cyc_B^y(AI, IF_M)$, and Cyc_C^1 and Cyc_C^2 from $Cyc_C^y(AI, IF_M)$. There is a relation between the components of both equations. Regarding the first components, Cyc_C^1 is always constant and equal to Cyc_B^1 at the ridge point, while the second components of the equations have the same absolute value, *i.e.*, $|Cyc_B^2| = |Cyc_C^2|$. Thus, from the analysis of Figure 23,

[27]John DC Little. "A proof for the queuing formula: L= λ W". *Operations research* 9.3 (1961), pp. 383–387.

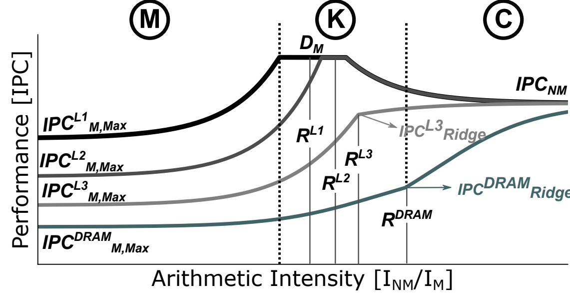


Figure 24 *Illustration of MaRM.*

Equations 8 and 9 can be unified in a single equation such that the application execution time (Cyc_{App}) is given by:

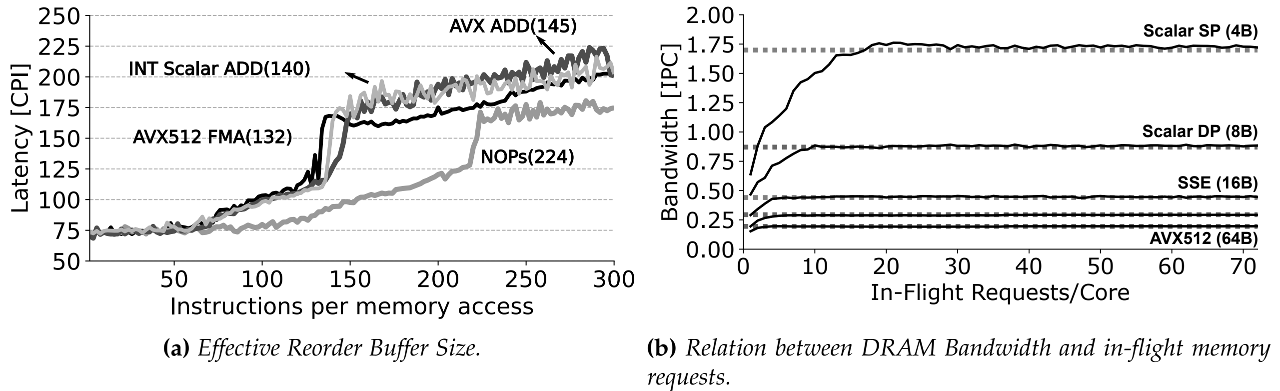
$$Cyc_{App}=I_M\times\left(\frac{\min(AI,R^y)(1+\frac{1}{R^y})}{IPC^y_R(IF_M)}+\frac{|\frac{AI}{R^y}-1|}{IPC^y_{M,Max}}\right). \tag{10}$$

Finally, from Equations 5 and 10, it is possible to derive the MaRM performance ($IPC^y_a(AI, IF_M)$):

$$IPC^y_a(AI, IF_M) = \min\left(\frac{I_M + I_{NM}}{Cyc_{App}}, D_M\right). \tag{11}$$

Figure 24 illustrates the proposed MaRM. As it can be observed, MaRM includes in the memory-bound region the entire memory hierarchy, with each level represented by their maximum sustainable bandwidth. The compute-bound region of the model is limited by the performance of the non-memory instructions. In contrast to SoA roofline models, the roofs in MaRM have a format similar to a "hill", and the memory roofs are no longer diagonal lines. Since MaRM performance cannot surpass the maximum retirement rate of the micro-architecture, flat regions may occur in the model, indicating areas where the application is limited by the number of retirement slots. This is observed for the L1 roof in Figure 24. While in CARM the ridge point corresponds to the minimum AI that allows attaining maximum performance for any memory levels, in MaRM this does not occur for L3 and Dynamic Random Access Memory (DRAM). Due to the high latency of these memory levels, their effective bandwidth depends on the amount of concurrent memory requests. Thus, the ridge points of the L3 cache and DRAM do not correspond to the point where maximum performance is achieved when accessing those memory levels. In fact, the performance continues to increase beyond the ridge point due to the growing contribution of the compute instructions, asymptotically approaching to the maximum performance of the compute units. On the other hand, for L1 and L2 caches, the micro-architecture is able to attain the maximum sustainable bandwidth for the entire range of AI, until reaching the ridge point. Hence, the ridge point in MaRM inherits the properties of CARM for L1 and L2 caches.

As it can be observed in Figure 24, MaRM contains three main regions: memory region, where application performance is limited by the memory bandwidth of each memory level; compute region, delimited by the maximum retirement rate of the system; and mixed region (K), where the bottlenecks can be either related to the memory accesses or the maximum achievable performance. The bottleneck identification is performed by plotting a vertical line at the application AI. The intersections right above and below the application dot correspond to the main sources of inefficiencies. Depending on the region where the application is located, a set of optimizations can



(a) *Effective Reorder Buffer Size.*

(b) *Relation between DRAM Bandwidth and in-flight memory requests.*

Figure 25 *Benchmarking results for Mansard Roofline Model.*

be derived to improve the execution time. In the memory region, the optimization should focus on improving the memory accesses, while in the compute region vectorization methods can be employed to improve application execution. In the mixed region, techniques from both memory and compute regions might be used according to the bottlenecks identified at each optimization phase.

In the mixed region of the roofs correspondent to the "slower" memory levels, applications are expected to be limited by both memory bandwidth and latency. Compared to the SoA roofline models, this property is exclusive to MaRM, and arises from the ROB impact to the bandwidth of the memory subsystem and retirement of instructions. This effect becomes more relevant as the AI approaches the ridge point. Moreover, since MaRM represents performance as IPC, the model is oblivious to the vector width. For this reason, certain optimizations lead to lower execution time with reduced IPC. For example, in current Intel micro-architectures, an application heavily dominated by scalar Integer (INT) instructions may be closer to the retirement roof of 4RS. However, by vectorizing the INT instructions, the IPC of the application will tend to the roofs correspondent to the retirement rate of SIMD ALU (3RS in most recent Intel CPUs).

Another scenario is the optimization of a compute-bound application dominated by Scalar instructions. This workload will be limited by the horizontal roof corresponding to the performance of these instructions (for example, 2RS for Floating-Point (FP) scalar instructions), indicating that there is no room for optimization, although the application can be easily sped-up by using vector instructions. Thus, when a scalar application is already on top of the computational roof that resembles the maximum throughput that its instruction mix can achieve, the recommendation is to attempt the code vectorization.

While the drop in IPC might look counter intuitive, it increases MaRM intuition regarding the ability to evaluate the vectorization efficiency. For example, when the throughput of scalar ad vector instructions is equal (*e.g.*, FP AVX512), three scenarios can occur after vectorization: **1)** IPC remains constant, indicating that the vectorization attained maximum efficiency and the execution time reduced proportionally to the vector size; **2)** the IPC reduced, hinting that the contribution of memory accesses and/or vectorization overheads increased and the vectorization did not attain 100% efficiency; and **3)** the IPC is equal to 1/vector_width, corresponding to the worst case scenario where the vectorization did not resulted in any benefit and the execution time remains almost the same.

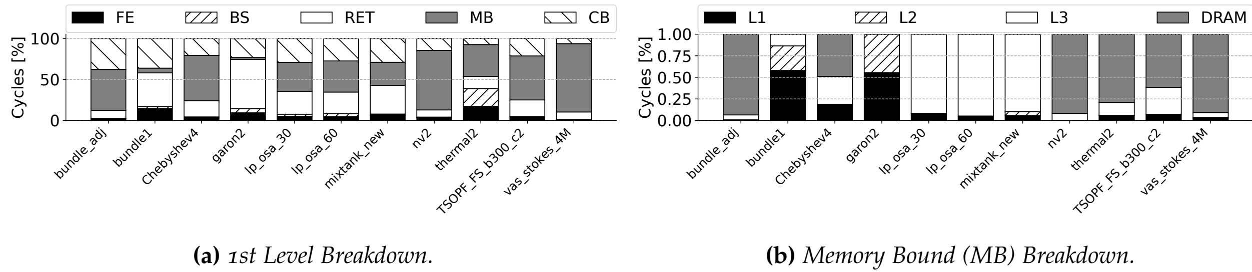
MaRM depends on the relation between the effective ROB size with the number of in-flight requests and the attainable memory bandwidth. This relation can be obtained through micro-benchmarking. Regarding the effective ROB size, the micro-benchmark is constructed such that a

Table 3 *SuiteSparse matrices used for the evaluation.*

Matrix	#NNZ	#Rows	#Columns
Bundle_adj	20,207,907	513,351	513,351
bundle1	770,811	10,581	10,581
Chebyshev4	5,377,761	68,121	68,121
garon2	373,235	13,535	13,535
Lp_osa_30	604,488	4,350	104,374
Lp_osa_60	1,408,073	10,280	243,246
mixtank_new	1,990,919	29,957	29,957
thermal2	8,580,313	1,228,045	1,228,045
nv2	37,475,646	1,453,908	1,453,908
TSOPF_FS_b300_c2	8,767,466	56,814	56,814
vas_stokes_4M	131,577,616	4,382,246	4,382,246

significant increase in the execution time occurs at the ROB limit. The experimental evaluations of the effective ROB size for No-Operation (NOP), 512-bit-AVX (AVX512) Fused Multipy-Add (FMA) instructions, Advanced Vector Extension (AVX) Additions (ADD) instructions and scalar integer additions (INT ADD) are presented in Figure 25a. Since NOPs do not use any register during execution, the latency drastically increases approximately at the size of the ROB, *i.e.*, 224 entries. For AVX512 FMA instructions, the memory latency increases when there are 132 instructions per memory access, *i.e.*, around 78.6% of the vector PRF capacity (168 entries), while for AVX ADD instructions, the effective ROB size is close to 145 entries. This non-ideal behavior indicates the existence of additional bottlenecks in different components of the core pipeline. The results also show that the effective ROB size depends on the instruction used when accessing a register file, as it is the case of the AVX512 FMA and AVX ADD instructions. Similar scenario occurs for the INT ADD test, with an effective ROB size around 140 entries.

To relate the effective micro-architecture bandwidth and the number of in-flight memory requests, the developed benchmarking exploits the organization of instructions in the ROB, by managing the number of memory requests that are simultaneously performed in this component. Figure 25b contains the relation between in-flight memory requests and the DRAM bandwidth for a ratio of 2 loads per 1 store (2LD/ST). As it can be observed, the curves for different memory transfer sizes have similar characteristics. For small amount of in-flight memory requests, the bandwidth increases in a non-linear fashion. Once the number of concurrent accesses is high enough to fully hide the memory latency, the bandwidth converges to the maximum sustainable bandwidth, corresponding to the dashed horizontal lines. For example, for the DRAM test with Scalar Double-Precision (DP) instructions, the effective bandwidth for 1 in-flight group of 2LD + 1ST per core is around 0.5, while the maximum sustainable bandwidth is attained around 10 in-flight groups of 2LD + 1ST per core, converging to an IPC around **0.9**. In the scenario, SoA roofline models would overestimate DRAM bandwidth approximately by 2x. It is also possible to verify that the bandwidth and the number of in-flight requests vary with the data size. For example, for L3 test the AVX512 instructions attain the maximum sustainable bandwidth of **2.95** around **7** groups of concurrent 2LD + 1ST requests, while for Scalar SP instruction it attains an IPC of **18.85** only when 48 groups of 2LD + 1ST are simultaneously in the ROB.



(a) *1st Level Breakdown.* **(b)** *Memory Bound (MB) Breakdown.*

Figure 26 *Top-Down Method of MKL SpMV.*

2.3.2 CHARACTERIZATION OF SPARSE KERNELS

To evaluate the capability of MaRM to provide accurate characterization of different sparse kernels, we relied on Intel MKL[28] implementation of two most commonly used sparse operations, namely: sparse matrix-vector (SpMV) and sparse matrix-matrix (SpMM) multiplication. For this analysis, a set of SuiteSparse[29] matrices were considered, which cover a wide range of number of non-zeros, rows and columns, as summarized in Table 3. The experimental evaluation was conducted on a Intel Xeon 6140 Gold (SKL-X), with 18 cores. Turbo boost, prefetching and hyper-threading were turned off during the evaluation.

The MKL SpMV and MKL SpMM kernels are characterized in MaRM and CARM, and their insights are compared with the ones provided by Top-Down Method, obtained from Intel VTune.[30] Top-Down classifies application bottlenecks into five main categories, namely: frontend (FE), bad speculation (BS), retiring (RET), core bound (CB) and memory-bound (MB). RET and CB indicate that the application performance is limited by the retirement rate of the micro-architecture and port utilization. FE and BS correspond to performance penalties from instruction fetch/decoding, and branch misprediction, respectively. MB highlights issues related to memory accesses.

SPARSE MATRIX-VECTOR (SPMV) CHARACTERIZATION

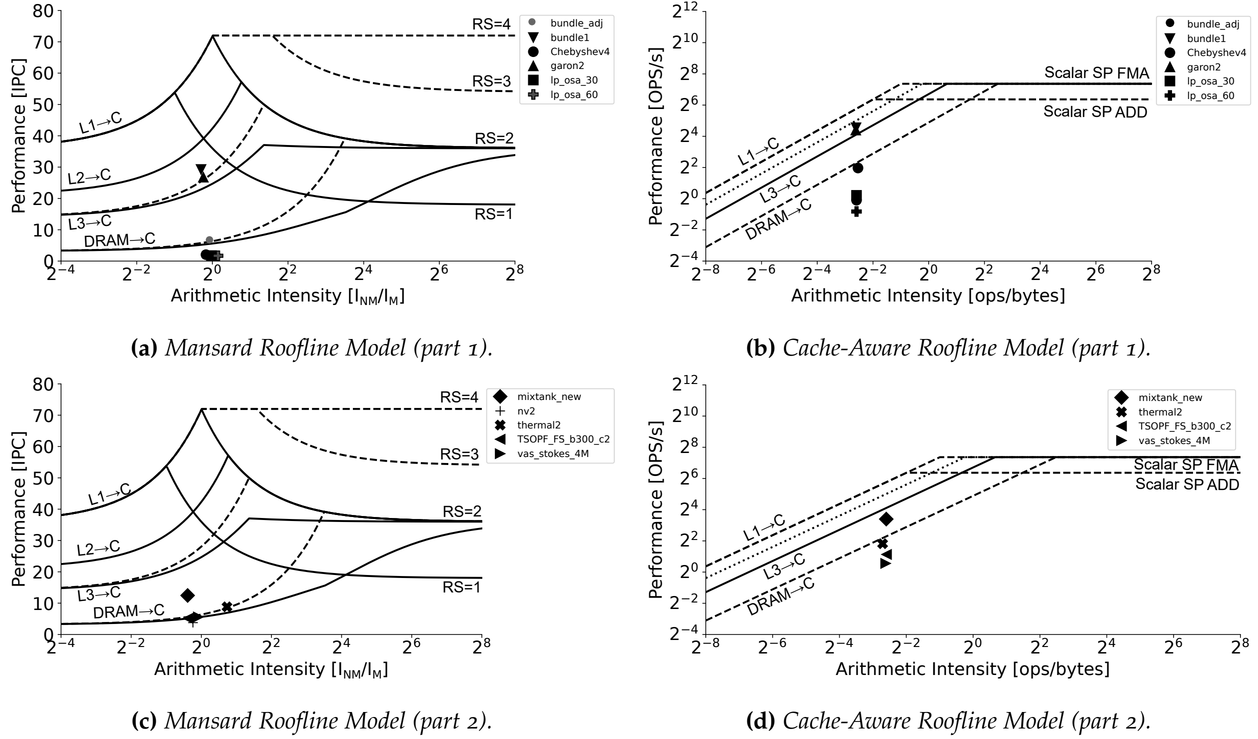
The Top-Down results for MKL SpMV are presented in Figure 26. As it can be observed from the first level of Top-Down (Figure 26a), all the matrices have significant contributions from retiring, core bound and memory bound. In the case of `bundle_1` and `garon2`, since their memory bound component is almost negligible, these matrices are expected to be limited by hardware components closer to the core, *e.g.*, private caches. For the remaining matrices that have a significant impact from memory-bound component, it is also essential to evaluate the memory bound breakdown provided by Top-Down (Figure 26b), indicating which memory level is the main bottleneck. From the memory breakdown, it is possible to conclude that several matrices are mainly limited by DRAM (`bundle_adj`, `nv2`, `thermal2`, and `vas_stokes_4M`), while `lp_osa_30`, `lp_osa_60` and `mixtank_new` are limited by L3 cache. `Chebyshev4` and `TSOPF_FS_b300_c2` have bottlenecks in several memory levels, mainly from L1, L3 and DRAM.

The characterization of these matrices in MaRM and CARM is presented in Figure 27. When comparing the hints obtained from Top-Down method and the bottlenecks pinpointed by both

[28] Endong Wang et al. "Intel math kernel library". *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.

[29] Scott P Kolodziej et al. "The suitesparse matrix collection website interface". *Journal of Open Source Software* 4.35 (2019), p. 1244.

[30] Ahmad Yasin. "A top-down method for performance analysis and counters architecture". *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2014, pp. 35–44; Intel Corporation. *VTune Profiler*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. [Online; visited June-2022].



(a) *Mansard Roofline Model (part 1).*

(b) *Cache-Aware Roofline Model (part 1).*

(c) *Mansard Roofline Model (part 2).*

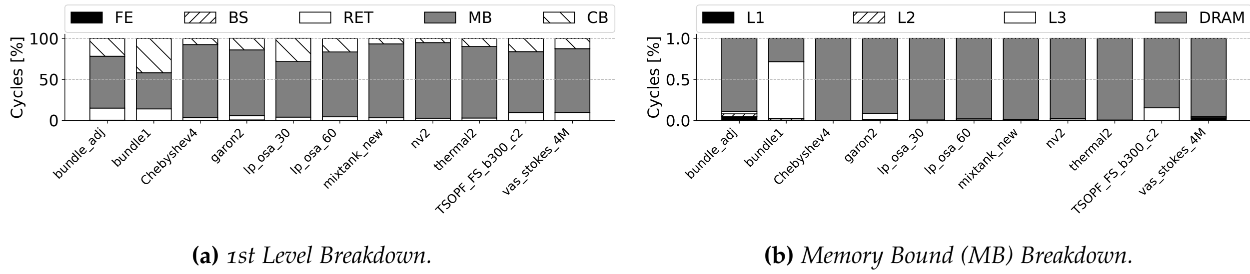
(d) *Cache-Aware Roofline Model (part 2).*

Figure 27 *MKL SpMV in Mansard and Cache-Aware Roofline Models.*

roofline models, it is possible to observe for some scenarios a better characterization in MaRM than when using CARM. This is the case of `bundle_adj` and `thermal2`, which are placed above DRAM in MaRM, indicating bottlenecks related to DRAM but also with contributions from components closer to the core that allow to surpass DRAM roof, which corroborates with Top-Down. On the other hand, this matrix is below DRAM in CARM. For `bundle1` and `garon2`, while they have a similar characterization in both models (between L2 and L3 caches), MaRM provides a different perspective by placing these matrices close to the retiring roof of $RS = 1$. This hints that these matrices can be limited by retiring, which is according to Top-Down method. From CARM, it is only possible to conclude that these matrices are fully limited by memory. Moreover, while `vas_stokes_4M` and `TSOPF` are below DRAM roof in CARM, these matrices are placed on top of DRAM roof in MaRM, which is expected given the DRAM bound nature indicated by Top-Down method. Finally, `lp_osa` and `mixtank_new` matrices have similar characterization in both roofline models. In the case of `lp_osa` matrices, according to Top-Down these matrices should be limited by L3 cache, which does not corroborate with any of the insights provided by both roofline models. The second-order effects behind this characterization will be further investigated in this project.

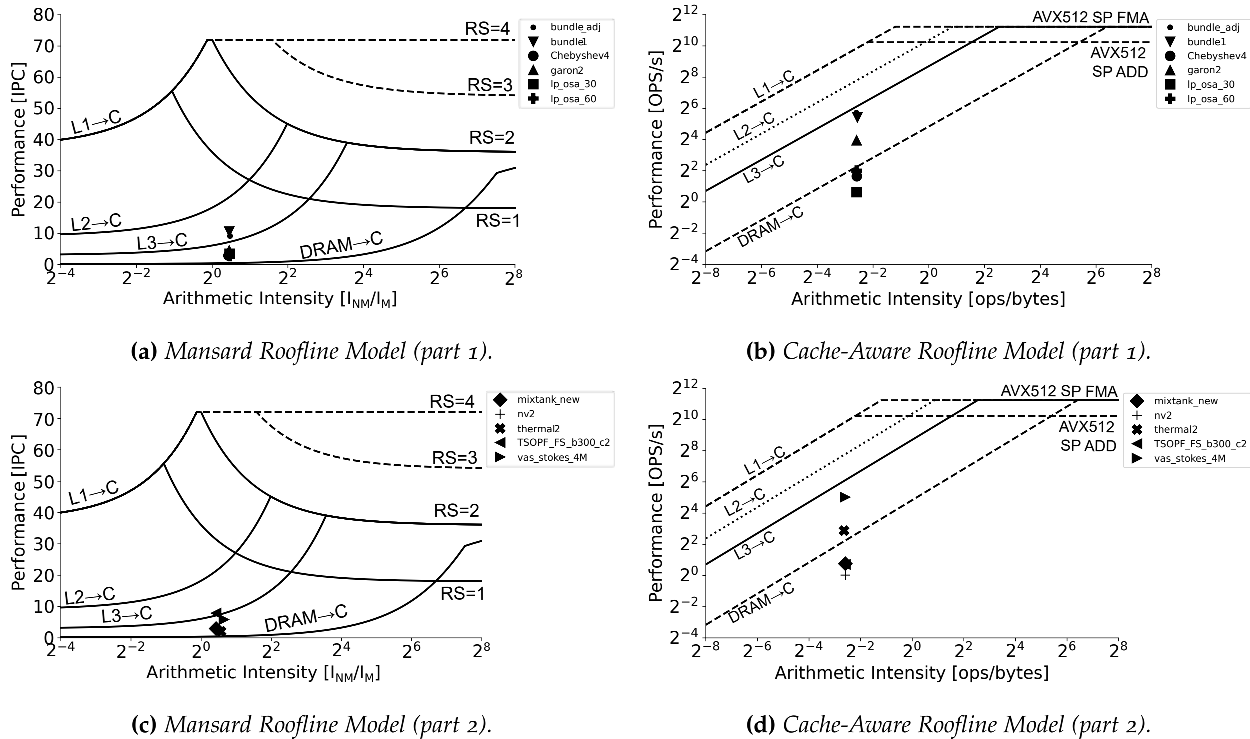
SPARSE MATRIX-MATRIX (SPMM) CHARACTERIZATION

The Top-Down results for MKL SpMM are presented in Figure 28. Differently from MKL SpMV, according to the first Top-Down level (Figure 28a), most of the matrices are limited by memory, with small contributions from core bound. The main exceptions are `bundle_adj`, `bundle1` and `lp_osa_30`, which also have significant contributions related to retiring and core bound. Regarding the memory breakdown (Figure 28b), most of the memory bound components arises from DRAM bottlenecks, with `bundle1` and `TSOPF` matrices having also bottlenecks that arise from L3 cache.



(a) *1st Level Breakdown.* **(b)** *Memory Bound (MB) Breakdown.*

Figure 28 *Top-Down Method of MKL SpMM.*



(a) *Mansard Roofline Model (part 1).* **(b)** *Cache-Aware Roofline Model (part 1).*

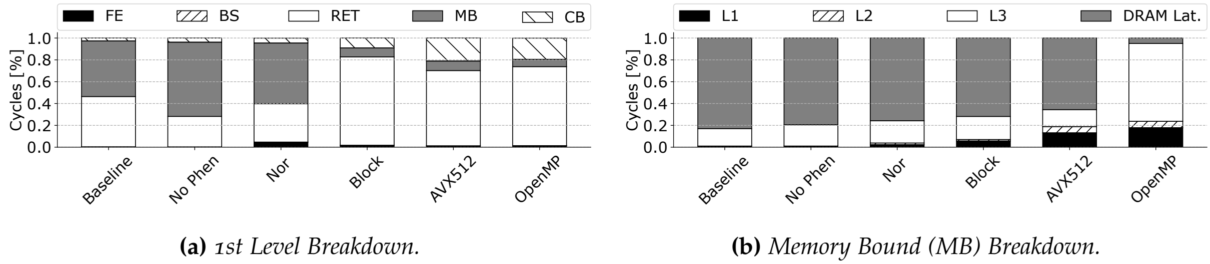
(c) *Mansard Roofline Model (part 2).* **(d)** *Cache-Aware Roofline Model (part 2).*

Figure 29 *MKL SpMM in Mansard and Cache-Aware Roofline Models.*

From the characterization in Top-Down, it is expected that most of the matrices are on top or slightly above the DRAM roof (due to the small core bound contributions). `Bundle1` and `bundle_adj` are the only matrices that are limited by components closer to the core, such as private caches.

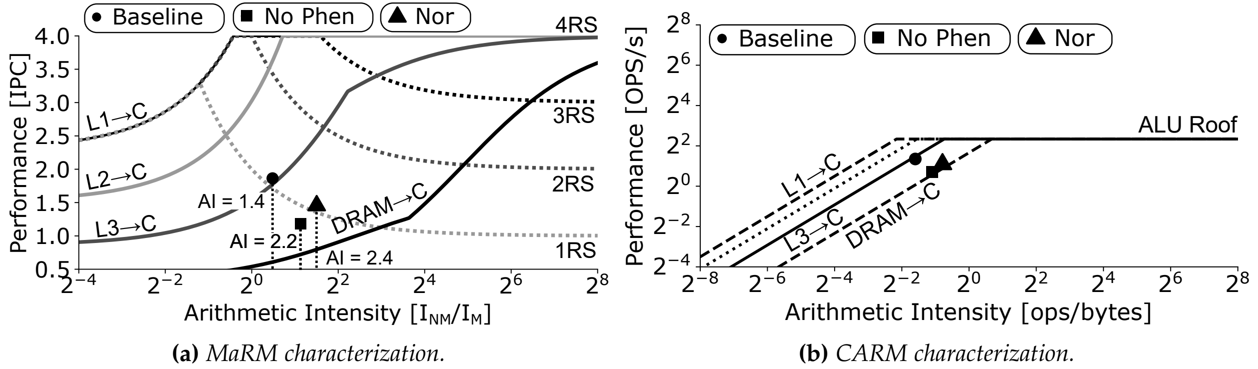
As observed in the characterization with different roofline models (Figure 29), MaRM indicates that `bundle1` and `bundle_adj` are limited by L3 cache, which is inline with the Top-Down characterization (by also taking into account significant retiring and core bound components).

On the other hand, CARM places these matrices between L3 and DRAM, where bottlenecks related to retiring and core bound are unlikely to occur. Furthermore, most of the remaining matrices are placed below the DRAM roof in CARM, which is not expected due to its DRAM bound nature with small contributions from core bound. These matrices are placed between L3 and DRAM roofs in MaRM, which provides a more accurate characterization of their bottlenecks when considering the Top-Down insights.



(a) *1st Level Breakdown.* **(b)** *Memory Bound (MB) Breakdown.*

Figure 30 *Top-Down Method of Epistasis Detection Algorithm.*



(a) *MaRM characterization.* **(b)** *CARM characterization.*

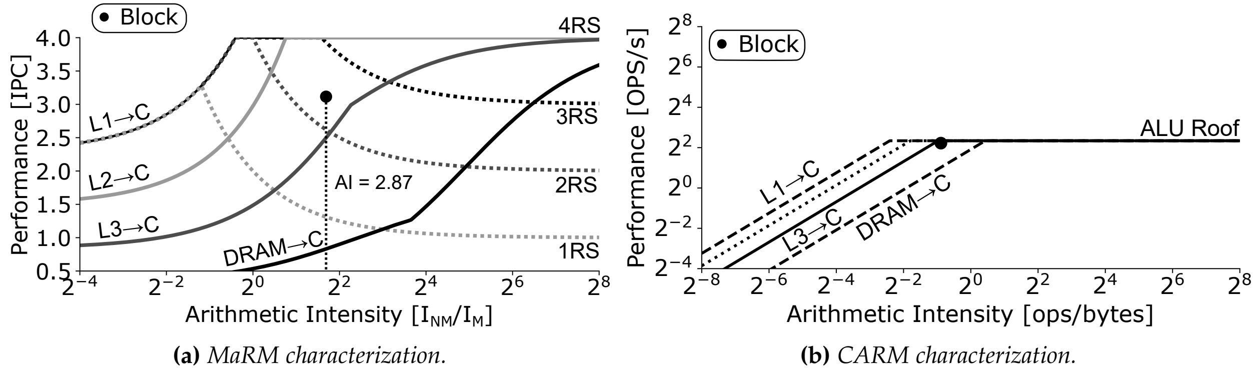
Figure 31 *Characterization of the Baseline, No Phen and Nor versions of the Epistasis Detection algorithm in MaRM and CARM.*

2.3.3 Case Study: Second-Order Epistasis Detection

To showcase the ability of MaRM to provide accurate insights when optimizing applications, an epistasis detection algorithm is optimized by following the hints provided by MaRM. It is worth noting that epistasis detection represents one of the core use-case applications in the SparCity project. This algorithm is widely used in bioinformatics to uncover the Single-Nucleotide Polymorphism (SNP) combination that is most likely to cause a disease or trait in a given dataset. The baseline algorithm contains a set of bitwise operations and population count (popcount) instructions,[31] which are not commonly included in any of the INT or FP SoA roofline models. The input dataset of this algorithm is a matrix organized with the SNPs in rows and patients in columns. Each SNP is represented by three binary arrays, that express the genotypes. A phenotype is also associated with each patient, indicating if the patient has the disease (case) or does not have the disease (control). In this work, the dataset contains **10040 SNPs and 104448 patients**, *i.e.*, more than 50 million pairwise combinations of SNPs need to be evaluated.

The characterization of the baseline algorithm in MaRM and CARM is presented in Figures 31a and 31b, respectively. In both models, the baseline algorithm is mainly limited by the L3 cache and placed in the mixed region of the models, *i.e.*, close to the compute roof of 1RS in MaRM and on top of L3 cache in CARM. These insights corroborate the results in Figure 30a, obtained with the Top-Down Method, which shows that the execution is mainly limited by memory accesses and retiring. Since the application is memory bound and mainly limited by L3 cache, the user must focus on memory related optimizations, such as, improving the memory access pattern or on reducing the amount of memory accesses. This is the case of the *No Phen* and *Nor* optimizations.

[31] Ricardo Nobre et al. "Exploring the Binary Precision Capabilities of Tensor Cores for Epistasis Detection". *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2020, pp. 338–347.



(a) *MaRM characterization.*

(b) *CARM characterization.*

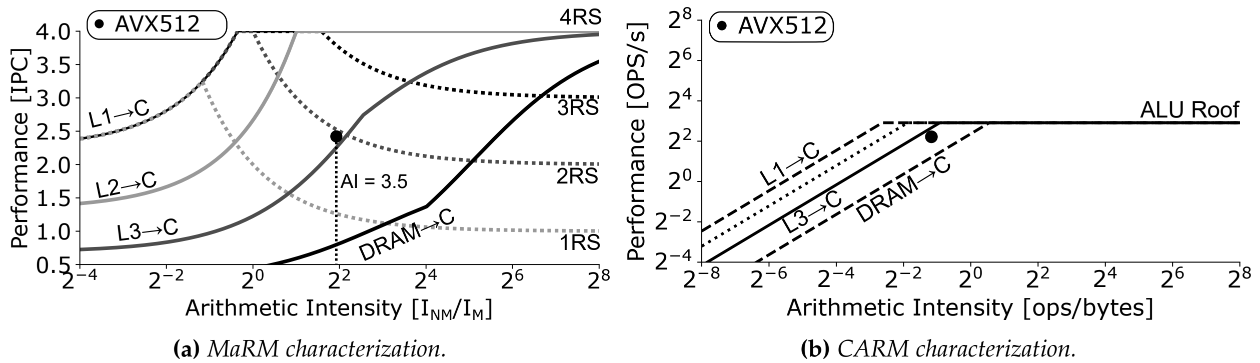
Figure 32 *Characterization of the Block version of the Epistasis Detection algorithm in MaRM and CARM.*

In the first optimization (No Phen) the phenotype is discarded by separating the dataset into cases and controls. The second optimization (Nor) only uses genotypes 0 and 1 for each patient, while genotype 2 is obtained by applying nor operation over the two remaining genotypes. Both these optimizations allow reducing the the number of memory accesses.

As it can be observed in Figure 31a, these optimization techniques resulted in an increase of the arithmetic intensity of the application, through the reduction of the memory instructions performed. However, it is also possible to verify that the IPC of these two versions is lower than the baseline, although the execution time reduced 1.42x for the *No Phen* version and 1.72x for the *Nor* version, in comparison to the Baseline application. This effect occurs due to the reduction of the total instructions performed by the applications. Compared to the baseline algorithm, the retired instructions for *No Phen* version reduced 2.27x, while for *Nor* version this reduction was around 2x. Since the decrease in instructions is higher than the time/cycles improvement, the applications attain a lower IPC and become more memory bound. However, this effect is not exclusive to MaRM. As it can be observed in Figure 31b, the reduction of the number of compute operations (around 4x) resulted in a performed drop in CARM, since the improvement of the execution time is lower than the operations reduction. For these two versions, the characterization between both models also differs. While MaRM also places the applications close to the 1RS, which is responsible for the retiring contribution according to the Top-Down method, CARM pinpoints DRAM as the main bottlenecks and does not hint any retiring contribution. Moreover, MaRM is also able to hint the DRAM latency issues pinpointed by the Top-Down analysis (Figure 30b) for these two application versions; the application is placed in the region around the DRAM ridge point, where the memory bandwidth is only a fraction of the maximum sustainable bandwidth of the micro-architecture.

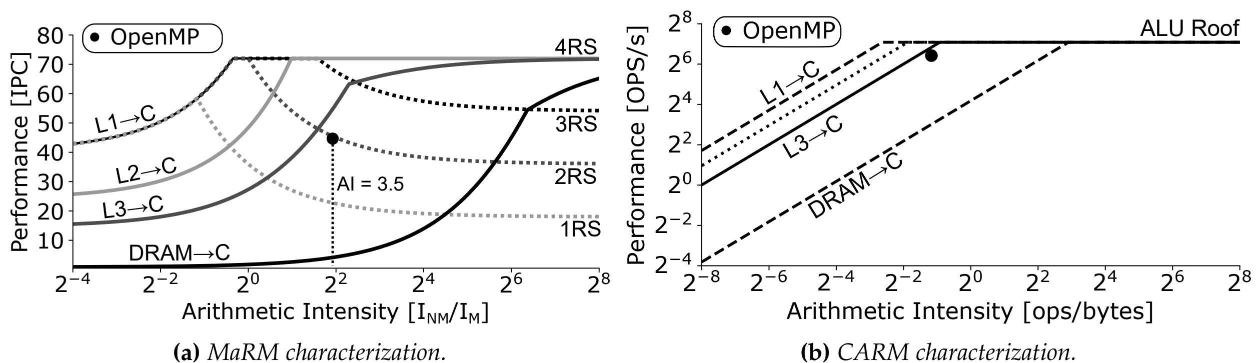
Since the *Nor* application has a low retirement rate and it is placed below L3 cache roof, to boost application execution it is necessary to further improve the memory accesses. This task is performed by introducing cache blocking techniques, improving the memory access pattern and resulting in the *Block* version of the application, which attained a speedup around 3.6x when compared to the *Baseline* version. As it can be observed in Figure 32a, in MaRM, the *Block* version is placed between the compute roofs 2RS and 3RS, hinting its compute-bound nature. Similarly, as shown in Figure 32b, CARM also indicates that the *Block* version is compute bound. Top-Down provides the same hints, characterizing the application as limited by retiring, with small contributions from the core bound and the memory (in particular L3 cache and DRAM Latency as shown in Figure 30b).

The *Block* version of the application only contains scalar instructions and it is limited in the MaRM by the higher retirement roofs. Thus, it is recommended to vectorize the application, which



(a) *MaRM characterization.* **(b)** *CARM characterization.*

Figure 33 *Characterization of the AVX512 version of the Epistasis Detection algorithm in MaRM and CARM.*



(a) *MaRM characterization.* **(b)** *CARM characterization.*

Figure 34 *Characterization of the OpenMP version of the Epistasis Detection algorithm in MaRM and CARM.*

is performed through the utilization of AVX512 intrinsics, allowing to attain a time speedup of 5.1x compared to the *Baseline* version. MaRM (Figure 33a) places the *AVX512* version of the algorithm on top of the 2RS roof, indicating that it is completely limited by the retirement units, which is corroborated by the Top-Down analysis, that identifies as the main bottlenecks the core and the retirement. On the other hand, CARM characterization (Figure 33b) is inaccurate, since the kernel is placed below the L3 cache roof. It is important to notice that MaRM is able to accurately characterize this kernel due to its modeling approach that considers all the instructions retired by the application. Given that the Intel Xeon Gold 6140 does not support vectorized popcounts, the use of extract instructions was required, which are not accounted for in current roofline models. Additionally, it is also possible to observe a drop in the IPC between *Block* and *AVX512* versions. This effect results from the limited number of ports that support AVX512 instruction in Skylake-SP micro-architecture. Hence, it is possible to conclude that the *AVX512* algorithm is able to attain the maximum retirement of the micro-architecture for AVX512 instructions. Moreover, since the *AVX512* version attained the maximum IPC allowed by AVX512 instructions, MaRM hints that the vectorization attained close to maximum efficiency.

Finally, since the single-threaded algorithm is already vectorized and completely limited by the retiring roofs in MaRM, the application is parallelized by using the OpenMP programming model. This parallel version is limited by the 2RS roof in MaRM (Figure 34a) and attains a speedup of 18.5x compared to the AVX512 version and 94x when compared to the baseline version. The super-linear speedup between the *AVX512* and *OpenMP* versions results from the higher utilization of the L3 cache in the *OpenMP*, as it is observed in Figure 30b. As it can be

observed in Figure 34b, CARM continues to indicate that the application is completely limited by L3 cache, which does not corroborate Top-Down analysis.

3 PERFORMANCE AND ENERGY-EFFICIENCY MODELING OF GRAPHCORE INTELLIGENT PROCESSING UNIT

The Graphcore Intelligent Processing Unit (IPU)[32] is a massively parallel device that aims at improving the performance of artificial intelligence workloads. The IPU can be seen as a distributed memory system, organized in independent tiles, each with its own local memory. The most recent Graphcore IPU, *i.e.*, the Colossus™MK2 GC200 contains 1472 tiles, each supporting 6 parallel threads, with a local memory of 624KB. Thus, the GC200 IPU provides up to 8832 parallel application threads and a total memory of around 900MB.

Each tile in the IPU contains an Accumulating Matrix Product (AMP) unit, which delivers up to 64 multiply and accumulate operations per cycle. Thus, at a nominal frequency of 1.33GHz, the GC200 IPU has a maximum floating-point performance of 250 TFlop/s. At this frequency, the IPU is also to serve memory requests at a maximum memory bandwidth of 47.5 TByte/s.

The tile execution in the IPU is based on the bulk synchronous parallel (BSP) model, where the execution is divided into three distinct phases: 1) in-tile execution, 2) sync, and 3) exchange. In phase 1, the tile performs the computations, and the memory accesses to their private memories. Since the execution across the tiles might be imbalanced, during phase 2 the tiles wait for the tile with the highest execution time and sync at a barrier. Once all tiles are synced, data is exchanged between tiles in phase 3. In the GC200 IPU, the tiles can communicate data between them at a maximum rate of 8 TByte/s.

Since the IPU execution is based on the BSP model, modeling the performance, power consumption, and energy efficiency of this device requires to consider the upper bounds of the memory accesses and computations for phase 1, and the maximum exchange bandwidth for phase 3. Given that the in-tile execution can be either limited by the memory accesses or the computations, roofline models are a good fit for this task. As for phase 3 (exchange), this work aims at extending the roofline modeling methodology to consider not only the upper bounds of the functional units and memory transfers but also the bottlenecks related to the communication between tiles and IPUs.

3.1 ROOFLINE MODELING OF THE IN-TILE EXECUTION

As previously referred, Roofline models[33] are insightful tools that provide an intuitive and simple relation between application behavior and micro-architecture upper bounds for performance, power consumption, and energy efficiency. These models rely on the assumption that memory transfers and computations are executed concurrently in current processors, due to their out-of-order nature. In this scenario, execution can be either limited by the memory hierarchy or by the computation capabilities of the functional units. With the execution limited either by the time to serve the memory requests (T^{β}) or by the time to perform the computations (T^{ϕ}), the execution time according to roofline models is given by:

$$T = \max\left\{T^{\beta}, T^{\phi}\right\} . \tag{12}$$

[32]Simon Knowles. "Graphcore". *2021 IEEE Hot Chips 33 Symposium (HCS)*. 2021, pp. 1–25. DOI: 10.1109/HCS52781.2021.9567075.

[33]Ilic, Pratas, and Sousa, "Cache-aware Roofline model: Upgrading the loft"; Williams, Waterman, and Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures".

For an IPU system with maximum memory bandwidth B (in byte/s) between the tiles and their local memories, and maximum computational performance F_p (in flop/s), the time to transfer β bytes is given by $T^{\beta} = \beta/B$, while the time to perform ϕ computations is given by $T^{\phi} = \phi/F_p$. Hence, the maximum attainable performance of an application (F_a) can be expressed as:

$$F_a(AI) = \frac{\phi}{T} = \min\{B \times \mathit{AI}, F_p\} , \tag{13}$$

where AI is the arithmetic intensity and corresponds to the total amount of computations performed over the total amount of bytes transferred.

The roofline modeling principles can also be applied to the power consumption domain.[34] When modeling power consumption, it is necessary to consider three components: the power consumption relative to the memory accesses (P^{β}), the power consumption of the computations (P^{ϕ}), and the constant power of the chip (P^{q}), due to components that are always active or shared between computations and memory transfers (*e.g.* register file). Given that P^{q} is always present when performing either computations or memory transfers, the power consumption of their correspondent components can be given by the sum between the constant power and the variable power consumption of each component, *i.e.*, $P^{\beta} = P^{q} + P^{\nu,\beta}$, and $P^{\phi} = P^{q} + P^{\nu,\phi}$. Based on these parameters, the power consumption of the IPU tiles can be calculated as:

$$\begin{aligned} P(AI) = \frac{E}{T} &= \frac{E^{q} + E^{\nu,\beta} + E^{\nu,\phi}}{T} = P^{q} + \frac{P^{\nu,\beta} \times T^{\beta}}{T} + \frac{P^{\nu,\phi} \times T^{\phi}}{T} = \\ &= P^{q} + P^{\nu,\beta} \times \frac{\beta/B}{\phi} \times F_a(AI) + P^{\nu,\phi} \times \frac{\phi/F_p}{\phi} \times F_a(AI) = \\ &= P^{q} + P^{\nu,\beta} \times \min\left\{1, \frac{F_p}{AI \times B}\right\} + P^{\nu,\phi} \times \min\left\{\frac{AI \times B}{F_p}, 1\right\} , \end{aligned} \tag{14}$$

where E is the total energy, E^{q} is the constant energy, $E^{\nu,\beta}$ is the variable energy of the memory transfers, and $E^{\nu,\phi}$ is the energy of the computations.

Based on the performance and power consumption, the total energy and energy-efficiency models can be derived. The IPU energy is defined as:

$$E(AI) = P(AI)T = \phi \left[\frac{P^{q}}{\min\{B \times AI, F_p\}} + \frac{P^{\nu,\beta}}{B \times AI} + \frac{P^{\nu,\phi}}{F_p}\right] , \tag{15}$$

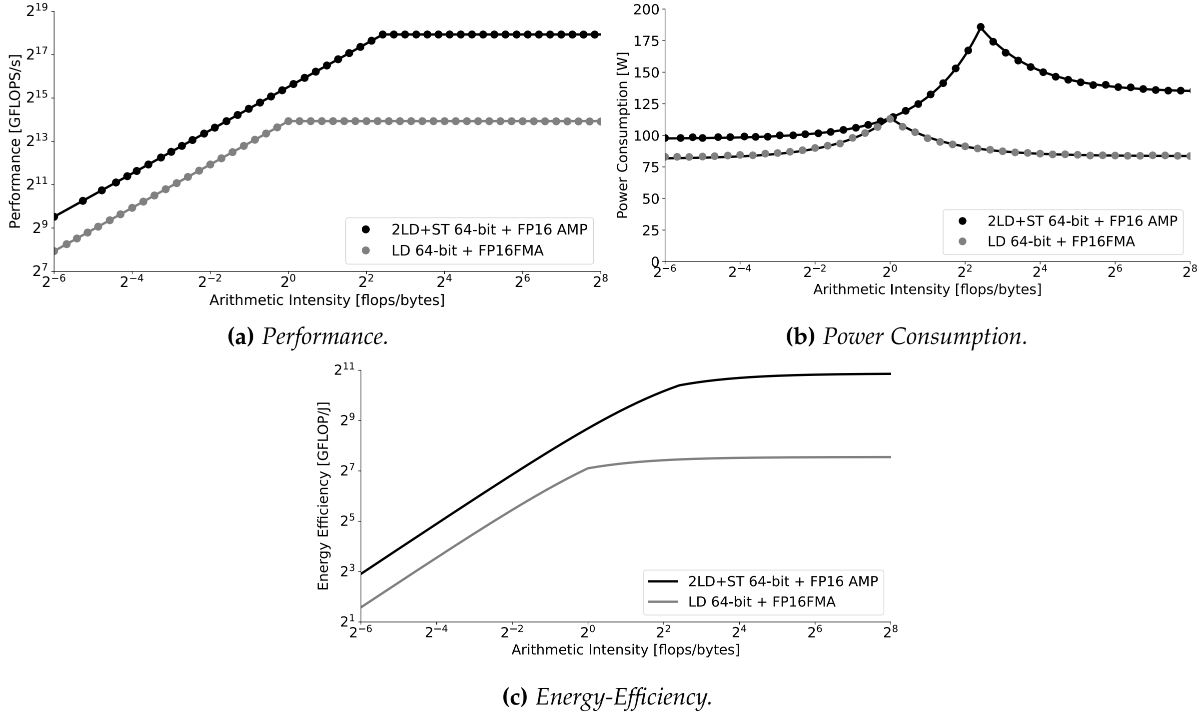
while the energy-efficiency is given by:

$$\epsilon(AI) = \frac{F_a(AI)}{P(AI)} = \frac{\phi}{E(AI)} = \frac{B \times AI \times F_p}{P^{q} \max\{F_p, B \times AI\} + P^{\nu,\beta} F_p + P^{\nu,\phi} B \times AI} . \tag{16}$$

The experimental validation of the performance and power consumption models for the in-tile execution is presented in Figures 35a and 35b, respectively. As it can be observed in the performance roofline model (Figure 35a), the measurements follow closely the theoretical curve, even when considering different sets of instructions, *i.e.*, 2 loads and 1 store (2LD+ST) and FP16 AMP instructions, and 64-bit loads and FP16 fused-multiply and add (FMA).

As it can be observed in Figure 35a, both tests have similar curves. The memory region (slanted roof) is limited by the maximum bandwidth achieved by each type of memory instructions, while

[34] Ilic, Pratas, and Sousa, "Beyond the Roofline: Cache-Aware Power and Energy-Efficiency Modeling for Multi-Cores".



(a) *Performance.* **(b)** *Power Consumption.*

(c) *Energy-Efficiency.*

Figure 35 *IPU Roofline Models.*

the compute region (horizontal roof) is limited by their correspondent FP performance, *i.e.*, 250 TFlop/s for the test that exercises the AMP units, and 15,7TFlop/s for the test containing FMA instructions. Since different instructions provide different performance limits, when characterizing applications, it is essential to select the roofline model that adapts to the workload characteristics. For example, if an application does not use the AMP unit and uses FP16 FMAs instead, the compute roof to consider must the one that represents the performance limits of FP16 FMA. This allows maximizing the usability and insightfulness of the model.

Similar to the performance model, the validation of the power consumption roofline model (Figure 35b) also closely follows the theoretical curve. In this model, the maximum power consumption is achieved at the ridge point, since it corresponds to the arithmetic intensity where the time to perform the memory transfers and computations is equal, thus their contribution to power consumption is maximized. When the arithmetic intensity reduces, the power consumption drops and tends to the power consumption of the memory transfers. Similarly, when the arithmetic intensity increases, the power reduces until achieving the power consumption that corresponds to the usage of the computation units. Moreover, when comparing performance and power consumption rooflines, it is also possible to observe that higher power consumption is coupled with a higher performance since the test containing 64-bit 2LD+ST transfers and FP16 AMP instructions achieves higher power consumption that the test with 64-bit loads and FP16 FMAs.

Finally, the energy-efficiency model obtained from both the previous models can also be observed in Figure 35c. This model has a similar shape to the performance. However, since the ridge point corresponds to the point where maximum power consumption is achieved, it does not correspond to the point of maximum energy efficiency. The maximum energy efficiency is attained when AI goes to infinity and it is equal to $\frac{F_p}{P\phi}$. For lower arithmetic intensity, the efficiency is limited by a slanted roof.

3.2 MODELING IMPACT OF INTER-TILE COMMUNICATION: EXCHANGE PHASE

By considering the exchange phase of the BSP model, the application execution time in the IPU corresponds to the time of the "slowest" tile. Thus, for the tile i with in-tile execution time of T_i and communication time of T_i^c, the overall execution time of an IPU with N tiles (T^{IPU}) is given by:

$$T_{IPU} = \max\left[T_0 + T_0^c, T_1 + T_1^c, ..., T_{N-1} + T_{N-1}^c\right] . \tag{17}$$

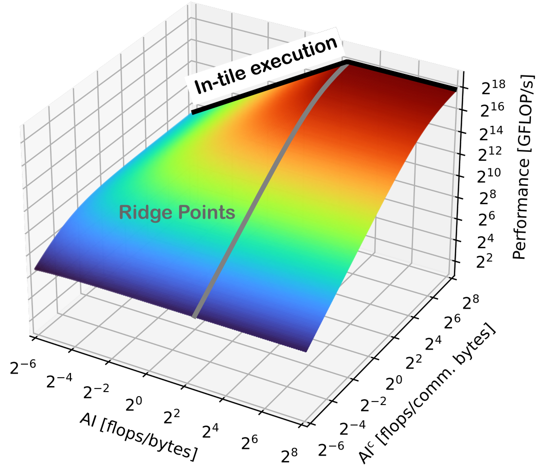
From the point-of-view of roofline modeling methodology, the model must consider the performance upper-bounds of the system, which in that case of the IPU corresponds to a balanced application execution. In this scenario, all the tiles spent the same time in the in-tile execution and exchange phases, *i.e.*, $T_0 = T_1 = ... = T_{N-1} = T$ and $T_0^c = T_1^c = ... = T_{N-1}^c = T^c$. Hence, $T_{IPU} = T + T^c$ and the maximum attainable performance of the IPU (F_a^{IPU}) can be calculated as:

$$\begin{aligned} F_a^{IPU} &= \frac{\phi}{T_{IPU}} = \frac{\phi}{T + T^c} = \\ &= \frac{\phi}{\frac{\phi}{F_a(AI)} + T^c} = \frac{\phi}{\frac{\phi}{F_a(AI)} + \frac{\beta^c}{B^c}} = \\ &= \frac{1}{\frac{1}{F_a(AI)} + \frac{\beta^c}{\phi B^c}} = \frac{F_a(AI)}{1 + \frac{F_a(AI)}{AI^c B^c}} , \end{aligned} \tag{18}$$

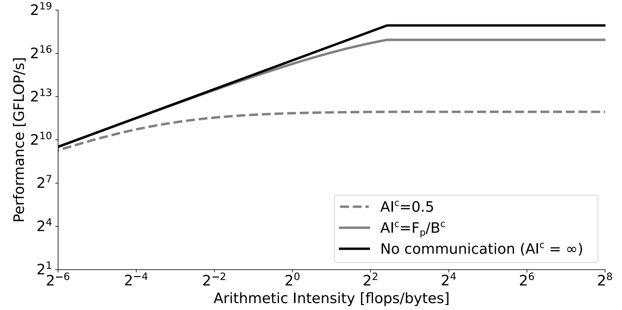
where β^c is the amount of bytes transferred during the exchange phase, B^c is the communication bandwidth, and AI^c is the is the communication arithmetic intensity and corresponds to the ratio between computations and exchange bytes.

The same principles can be applied to the power consumption, by considering an additional component P^c related to the communication between tiles, such that, $P^c = P^{\nu,c} + P^{q,c}$, where $P^{\nu,c}$ and $P^{q,c}$ are the variable and constant power consumption of the communication, respectively. In this scenario, the power consumption of the IPU (P^{IPU}) is given by:

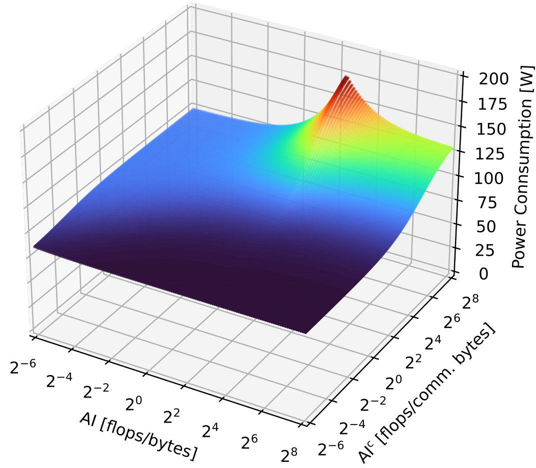
$$\begin{aligned} P_{IPU} &= \frac{E_{IPU}}{T_{IPU}} = (E^q + E^{\nu,\beta} + E^{\nu,\phi} + E^{\nu,c}) \times \frac{F_a^{IPU}}{\phi} = \\ &= (E^{q,\phi,\beta} + E^{q,c} + E^{\nu,\beta} + E^{\nu,\phi} + E^{\nu,c}) \times \frac{F_a^{IPU}}{\phi} = \\ &= (E^{q,\phi,\beta} + E^{\nu,\beta} + E^{\nu,\phi} + E^c) \times \frac{F_a^{IPU}}{\phi} = \\ &= (P^{q,\phi,\beta} \times T + P^{\nu,\beta} \times T^\beta + P^{\nu,\phi} \times T^\phi + P^c \times T^c) \times \frac{F_a^{IPU}}{\phi} = \\ &= \left(P^{q,\phi,\beta} \times \frac{\phi}{F_a(AI)} + P^{\nu,\beta} \times \frac{\beta}{B} + P^{\nu,\phi} \times \frac{\phi}{F_p} + P^c \times \frac{\beta^c}{B^c}\right) \times \frac{F_a^{IPU}}{\phi} = \\ &= \frac{P^{q,\phi,\beta}}{1 + \frac{F_a(AI)}{AI^c B^c}} + P^{\nu,\beta} \times \frac{\min\left(1, \frac{F_p}{AI \times B}\right)}{1 + \frac{F_a(AI)}{AI^c B^c}} + P^{\nu,\phi} \times \frac{\min\left(\frac{AI \times B}{F_p}, 1\right)}{1 + \frac{F_a(AI)}{AI^c B^c}} + \frac{P^c}{1 + \frac{AI^c B^c}{F_a(AI)}} , \end{aligned} \tag{19}$$



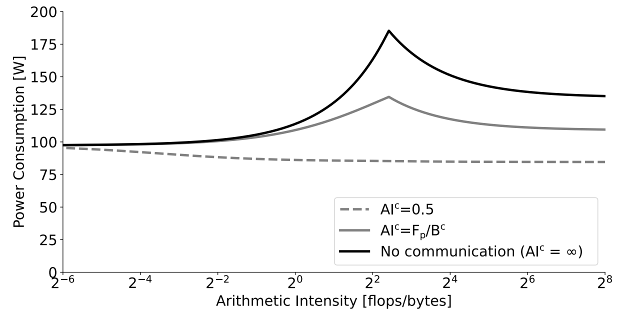
(a) *Performance - 3D Plot.*



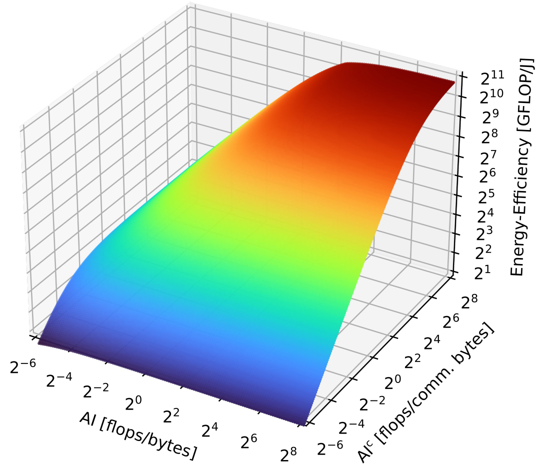
(b) *Performance - 2D Plot.*



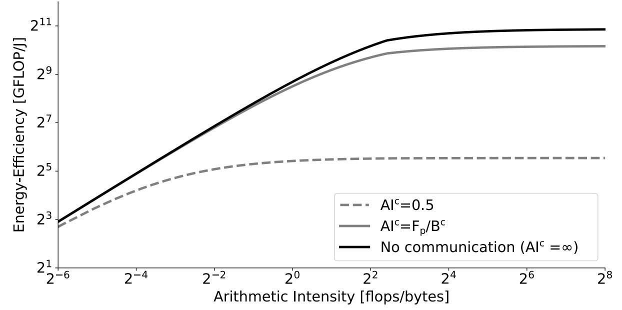
(c) *Power-Consumption - 3D Plot.*



(d) *Power-Consumption - 2D Plot.*



(e) *Energy-Efficiency - 3D Plot.*



(f) *Energy-Efficiency - 2D Plot.*

Figure 36 *IPU Roofline Models with effect of inter-tile communication.*

where $P^{q,\phi,\beta}$ is the constant power consumption of the in-tile execution phase, *i.e.* P^q of Equation 14, and E_{IPU} is the energy of the IPU. Similarly, the energy of the IPU (E_{IPU}) can be calculated

as:

$$E_{IPU} = P_{IPU} \times T_{IPU} = P_{IPU} \times \frac{\phi}{F_a^{IPU}} = \\ = \phi \times \left[\frac{P^{q,\phi,\beta}}{\min(AI \times B, F_p)} + \frac{P^{\nu,\beta}}{AI \times B} + \frac{P^{\nu,\phi}}{F_p} + \frac{P^c}{AI^c B^c} \right], \quad (20)$$

and the energy-efficiency is given by

$$\epsilon_{IPU} = \frac{F_a^{IPU}}{P_{IPU}} = \frac{\phi}{E_{IPU}} = \frac{AI \times B \times F_p}{P^{q,\phi,\beta} \max(AI \times B, F_p) + P^{\nu,\beta} \times F_p + P^{\nu,\phi} \times AI \times B + P^c \frac{AI \times B \times F_p}{AI^c B^c}}. \quad (21)$$

Figure 36 presents the performance, power consumption and energy-efficiency models for the IPU, and their variation with AI and AI^c. As it can be observed in Figure 36a, the maximum attainable performance of the in-tile execution phase (black roofline curve) is achieved when the AI^c tends to high values. As AI^c decreases, *i.e.*, the impact of communication increases, the maximum performance that can be achieved for the entire range of AI decreases, as most of the time is spent in the inter-tile communication. This effect can be seen in Figure 36b, for three different AI^c. When there is no data exchange between IPU tiles, *i.e.*, $AI^c = \infty$, the attainable performance of the IPU is equal to $F_a(AI)$, which corresponds to the performance of in-tile execution phase. As AI^c increases, *i.e.*, the amount of exchanged data per floating point operation increases, there is a reduction in the maximum attainable performance of the IPU. For example, an $AI^c = 0.5$ leads to a decrease in the maximum performance from around 2^{18} to 2^{12} GFLOP/s.

Regarding power consumption (Figure 36c), the maximum power consumption is attained for higher values of AI^c, which corresponds to the power consumption of the in-tile execution. As the amount of communication increases, the power consumption decreases and tends to a constant value equal to the power consumption of the communication between tiles. This effect can also be observed in Figure 36d, where as the AI^c decreases, the power consumption of the IPU also decreases. In fact, an AI^c of 0.5 leads to a power consumption almost constant and equal to the power consumption of the components involved in the exchange phase.

Finally, energy-efficiency (Figure 36e) has a behaviour similar to the performance curve, which maximum efficiency achieved for higher values of AI^c. Moreover, as the exchange phase contribution increases (lower AI^c), the energy-efficiency also decreases, mainly due to the big reduction in performance that also occurs with the increased impact of communication. This is also observed for the three different AI^c represented in the 2D plot of Figure 36f.

3.3 ROADMAP FOR IPU ROOFLINE DEVELOPMENT

While the scenario here considered assumes that the application execution is balanced, such scenario might be unlikely to occur when deploying real-world applications in the IPU. This can reduce the usability and characterization accuracy of the model when targeting this type of workloads. Hence, the next challenges to address in the scope of the SparCity project are the incorporation of exchange bottlenecks related to the communication between tiles in applications that suffer from imbalance. With this aim, a set of micro-benchmarks will be developed, in order to evaluate the maximum capabilities of the communication interface within the IPU under distinct scenarios, such as different communication patterns and system congestion.

4 DATA MOVEMENT ANALYSIS

Data movement is a major factor that affects the performance of parallel applications.[35] In the context of share-memory multithreaded applications, most of them happen in the forms of cache misses or cache line transfers across multiple cores. Due to the prevalence of this problem, tools to detect inter-thread communications and cache partitioning were developed.

4.1 INTER-THREAD COMMUNICATION ANALYSIS

To detect inter-thread communications in multithreaded code with low overheads, we extend ComDetective[36] to AMD architectures, which was previously developed and tested on Intel architectures. ComDetective captures inter-thread communications in the forms of communication matrices, and attributes the detected communications to their locations in source code. Using the information generated by ComDetective, programmers are able to perform performance tuning on their code, for example, by means of thread mapping or code refactoring. In this work, we modify it to leverage Instruction Based Sampling (IBS) facility[37] when running on AMD machines to sample memory loads and stores in detecting communications. We refer to the modified version of the tool as ComDetective$^+$.

In our experimental study, we firstly verify the accuracy of ComDetective$^+$ by using the microbenchmarks in.[38] We also perform sensitivity analysis to evaluate the impacts of different sampling intervals and different debug register counts on the accuracy of ComDetective$^+$. After that, we evaluate the runtime and memory overheads of the tool by running it on eight PARSEC benchmarks.[39] In the experiments, ComDetective$^+$ displays high accuracy while incurring 2.8$\times$ runtime and 1.92$\times$ memory overheads. Our next step is to utilize ComDetective$^+$ on a set of sparse matrices and evaluate their inter-thread communication when executing sparse solvers such as SpMV or SpTRSV.

4.1.1 COMDETECTIVE

ComDetective was first introduced for Intel architectures in.[40] Since the release of the Ryzen chips, AMD processors have become more widely used.[41] With high core and thread counts per CPU and lower power consumption than their Intel counterparts, processors from AMD Ryzen series have become very attractive for HPC applications. By 2021, around 73 supercomputers around the world already employed AMD processors with four out of the top ten most powerful supercomputers utilizing these processors.[42]

[35] D. Unat et al. "Trends in Data Locality Abstractions for HPC Systems". *IEEE Transactions on Parallel and Distributed Systems* 28.10 (2017), pp. 3007–3020. ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2703149.

[36] 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.

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

[38] Sasongko et al., "ComDetective: A Lightweight Communication Detection Tool for Threads".

[39] C. Bienia et al. "The PARSEC benchmark suite: Characterization and architectural implications". *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008, pp. 72–81.

[40] Sasongko et al., "ComDetective: A Lightweight Communication Detection Tool for Threads".

[41] Jack Howarth. *AMD vs Intel 2022: Which Should be Your First Gaming CPU?*. https://www.wepc.com/cpu/compare/amd-vs-intel-gaming/. 2022.

[42] Inc. Advanced Micro Devices. *AMD Processors Accelerating Performance of Top Supercomputers Worldwide*. https://www.amd.com/en/press-releases/2021-11-16-amd-processors-accelerating-performance-top-supercomputers-worldwide. 2021.

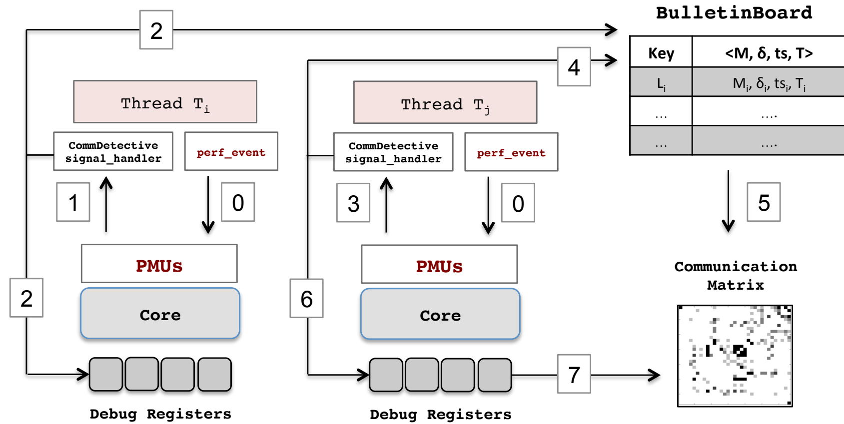


Figure 37 *One possible execution scenario of* ComDetective*: 0) Every thread configures its PMU to sample its stores and loads. 1) Thread T_i's PMU counter overflows on a store. 2) T_i publishes the sampled address to* `BulletinBoard` *if no such 'recent' entry exists and tries to arm its watchpoints with an address in the* `BulletinBoard` *(if any). 3) Thread T_j's PMU counter overflows on a load. 4) T_j looks up* `BulletinBoard` *for a matching cache line. 5) If found, communication is reported. 6) Otherwise, T_j tries to arm watchpoints on a cache line randomly selected from the* `BulletinBoard`*. 7) T_j accesses an address on which it already set a watchpoint, the debug register traps, communication is reported.*

ComDetective is a profiling tool that leverages hardware PMUs and debug registers to detect communication among threads. PMUs are employed to sample memory accesses in each thread of a profiled multithreaded application, and debug registers are used to detect memory accesses that are involved in communication with the memory accesses sampled by PMUs.

An example workflow of how ComDetective detects an inter-thread communication is shown in Figure 37. In the beginning of profiling, ComDetective, which works in the address space of the profiled process, sets each thread to configure its PMUs to sample memory loads and stores. When a sampled memory access to address m_0 occurs in a thread t_0, t_0 retrieves the sampled address m_0 and extracts the offset address c_0 of its cache line. c_0 is queried in a globally shared data structure called `BulletinBoard` to see if an address from the same cache line c_0 has 'recently' been published by another thread t_1 in the `BulletinBoard`. If such an address exists in the `BulletinBoard`, a communication is detected between t_1 and t_0. However, if such an address cannot be found in the `BulletinBoard`, t_0 randomly selects a 'recent' entry published by another thread t_2, such that $t_2 \neq t_0$, from the `BulletinBoard`, and arms its debug registers to detect communication between t_0 and t_2. If any of the debug registers traps, it means there is a memory access in t_0 to the same cache line that was accessed by t_2, and therefore a communication or cache line transfer happens from t_2 to t_0. In addition to attempting to detect communication, t_0 also tries to publish its sampled address m_0 to the `BulletinBoard` if the type of the memory access sampled by t_0 is a store access and there has not been any 'recent' sample on cache line c_0 in the `BulletinBoard`.

4.1.2 Implementation

To profile multithreaded applications in AMD machines, ComDetective$^+$ leverages IBS features.[43] However, to configure and sample IBS using `perf_event_open` system call, certain

[43]Drongowski, *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors*.

BIOS software that is not always available by default is needed[44].[45] Since this BIOS software is not always available, we have to rely on an open source Linux kernel module named AMD_IBS_Toolkit.[46] For ease of prose, we refer to this Linux kernel module as IBS driver. This IBS driver allows a user application to configure IBS and retrieve samples from IBS. Two flavors of sampling can be performed using IBS – fetched instruction sampling and executed micro-operation sampling. Using its `ioctl` interface, this driver allows user applications to enable or disable IBS counters and set up sampling period. To further support the profiling tools, we modify the IBS driver by introducing additional capabilities to it. This extra modification of the code that runs in kernel space makes the development of ComDetective$^+$ more laborious than the development of its Intel counterpart, which can already take advantage of `perf_event_open` system call to program PEBS.

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 CPU 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.

Figure 38 displays the workflow of the IBS driver during profiling. When ComDetective$^+$ 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.

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' model-specific registers (MSRs) in that CPU core to a ring buffer. Since ComDetective$^+$ only needs 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.

[44]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.

[45]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.

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

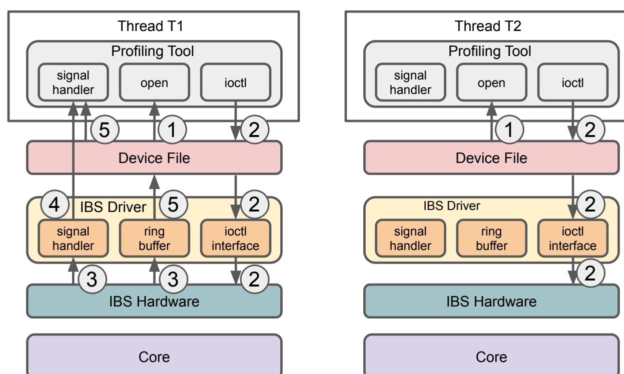


Figure 38 *One possible workflow scenario of the IBS driver: 1) Every thread calls `open` system call to get the file descriptor of the device file that corresponds to the core it is running on. 2) Every thread uses `ioctl` system call on the file descriptor to configure the sampling period of IBS, sets 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. 3) Thread T_1'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. 4) The interrupt handler sends an OS signal to the thread that triggered the interrupt, i.e. thread T_1. 5) A signal handler that runs in T_1's address space handles the OS signal, and reads the device file to retrieve the sampled data.*

In AMD machines, ComDetective$^+$ interfaces with the modified IBS driver to configure the settings of IBS sampling, and retrieve data from valid samples. In each IBS sample, this tool reads 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.

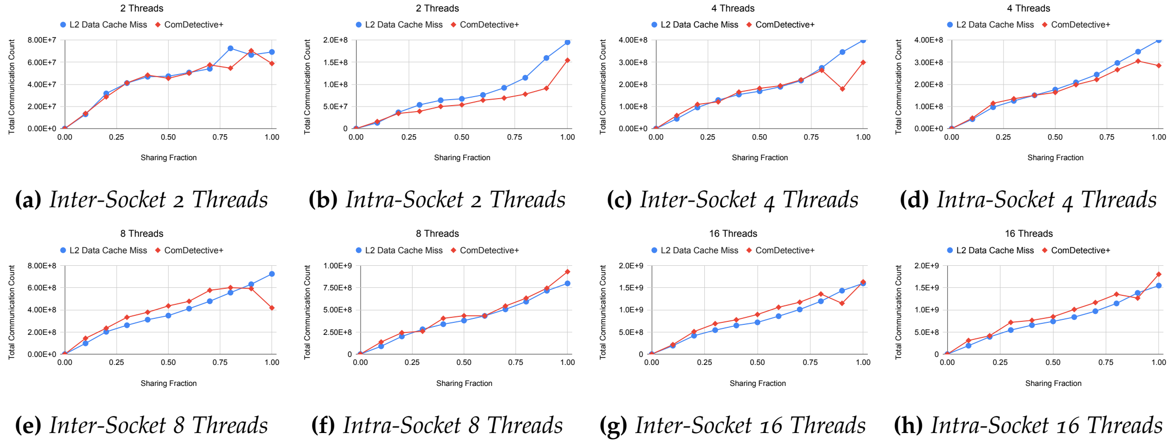
4.1.3 Communication count analysis

The accuracy of the communication analysis tool is evaluated in terms of its abilities to capture communication patterns, to differentiate true sharing from false sharing, and to capture total communication counts correctly. In evaluating the tool's accuracy, we employ the microbenchmarks in.[47] Furthermore, we also evaluate the sensitivity of ComDetective$^+$ under different thread counts, different sampling intervals, and different debug register counts. This section also reports the overheads of the tool by running it on a number of benchmarks from the PARSEC benchmark suite. 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-9.3.0 toolchain in this machine. Unless otherwise stated, the default sampling interval is 50K, and the default number of debug registers that we use in each core is four.

In this experiment, we evaluate the accuracy of ComDetective$^+$ in capturing total communication counts. The ground truth for this experiment is L2 data cache miss counts measured using Linux perf as these counts reflect the total communication counts that occur in the *Write-Volume* benchmark from[48] that we use in this experiment.

The results show the estimated communication counts captured by ComDetective$^+$ when

[47] Sasongko et al., "ComDetective: A Lightweight Communication Detection Tool for Threads".
[48] Ibid.

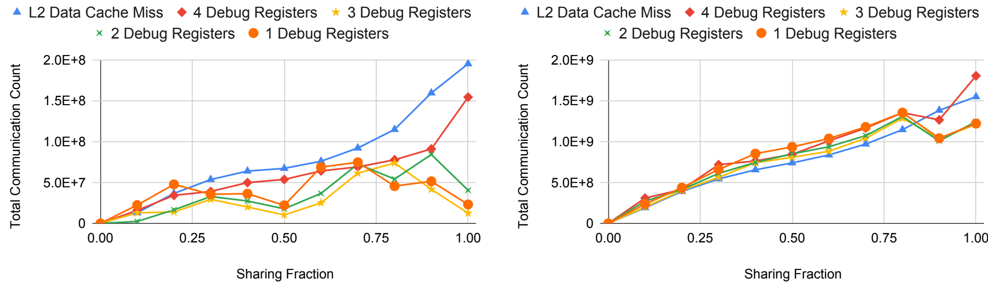


(a) *Inter-Socket 2 Threads* **(b)** *Intra-Socket 2 Threads* **(c)** *Inter-Socket 4 Threads* **(d)** *Intra-Socket 4 Threads*

(e) *Inter-Socket 8 Threads* **(f)** *Intra-Socket 8 Threads* **(g)** *Inter-Socket 16 Threads* **(h)** *Intra-Socket 16 Threads*

Figure 39 *Total communication counts for different sharing fractions in the AMD machine.*

profiling the benchmark running with different thread counts and different socket mappings. As shown in the figure, the estimated counts are close to the ground truth in nearly all cases. The exceptions are the cases with high sharing fraction, i.e. 0.8 or higher, and low thread counts, i.e. 2, 4, and 8 threads.



(a) *2 Threads Intra Socket* **(b)** *16 Threads Intra Socket*

Figure 40 *Debug register sensitivity results of* ComDetective^{+}

Sensitivity to Debug Register Count

We performed sensitivity analysis on the accuracy of the ComDetective^{+} with respect to the number of used debug registers. In this experiment, we used the tool to profile the communication volume benchmark running on 2 and 16 threads that are mapped to the same socket. To evaluate the impact of different debug register count on the accuracy of the profiling tool, we employed the tool to profile the synthetic benchmark running with different sharing fractions and different number of debug registers. Figure 40 presents the results. As shown in the figures, when the number of threads is 16, the accuracy of the tool remains high regardless of the number of debug registers used. However, when there are only two threads, ComDetective^{+} is the closest to the ground truth only when the number of debug registers used is 4.

Overhead Analysis

We evaluate the overheads of ComDetective^{+} by running it on eight PARSEC benchmarks,[49] i.e. *blackscholes*, *bodytrack*, *dedup*, *fluidanimate*, *freqmine*, *streamcluster*, *swaptions*, and *vips* with 50K sampling interval. The average runtime overhead over these benchmarks is 2.8×, and

[49] Bienia et al., "The PARSEC benchmark suite: Characterization and architectural implications".

the average memory overhead is 1.92×, which are still much lower than the overheads of cycle-accurate simulators and binary instrumentation-based tools. One binary instrumentation-based communication analyzer that we evaluated as a comparison is Numalize[50][51] whose runtime overhead is more than 16× and memory overhead is nearly 2000×. Consequently, the overheads of ComDetective$^+$ make it practical to be used for real-life applications.

4.1.4 ComDetective$^+$ roadmap

ComDetective$^+$ is currently working on Intel and AMD x86 architectures. From this point, we would like to pursue two direction of research. Firstly, we would like to extend the communication monitoring tool to other architectures such as ARM, GPUs, possibly IPUs. Secondly, we will utilize the tool on a number of sparse matrices and leverage it for performance analysis. In addition, we plan to integrate it to the Digital SuperTwin toolset so that it can work seamlessly with the rest of the probing features of Digital SuperTwin.

4.2 Cache partitioning

Cache partitioning is an optimisation technique that allows to virtually split a cache into multiple partitions. One of the use cases for cache partitioning is to improve cache behaviour e.g. by separating reusable data from non-temporal data in different partitions. The potential benefit of cache partitioning with two partitions can be modeled using reuse distance by dividing the address space into two parts D and $\overline{D}$.

Reuse distance is a metric that allows to analyse the cache behaviour of an application. The metric is independent of the cache size and only depends on the addresses of the sequence of memory references (*trace*) made during application execution. Based on the reuse distance $RD(x)$ of a trace, the number of cache misses occurring during application execution can be computed for fully associative caches with LRU policy for arbitrary cache capacities C (Equation 22). For set-associative caches with (pseudo-)LRU policy, the number of cache misses can be approximated using reuse distance.

$$\text{miss}(C) = \sum_{RD(x) \geqslant C} RD(x) \tag{22}$$

The application trace is split into one trace containing all memory references belonging to D and another trace including the remaining references. To model the number of cache misses when data in D is isolated in a cache partition, the reuse distances of the resulting two split traces are calculated to construct two partitioned reuse distance histograms. Equation 22 can then be modified to estimate the number of cache misses for various cache partition sizes C_0 and C_1 ($C_{total} = C_0 + C_1$) when the data in D is placed in a partition with cache capacity C_0 (Equation 23).

$$\text{miss}(C_0, D) = \sum_{RD(x) \geqslant C_0} RD_D(x) + \sum_{RD(x) \geqslant C_1} RD_{\bar{D}}(x) \tag{23}$$

The optimal cache partitioning w.r.t. D is the partitioning scheme that minimizes the total number of cache misses.

[50] Matthias Diener et al. "Characterizing communication and page usage of parallel applications for thread and data mapping". *Performance Evaluation* 88-89 (2015), pp. 18–36.

[51] Matthias Diener et al. "Communication in Shared Memory: Concepts, Definitions, and Efficient Detection". *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2016.

4.2.1 PROFILING TOOL

Based on the aforementioned considerations, a profiling tool was developed to provide hints to programmers on parts of their application that may benefit from cache partitioning. An application's memory access trace, which characterises the application cache behaviour, is obtained using Intel Pin for binary instrumentation in order to calculate partitioned reuse distance histograms. The partitioned reuse distance histograms allow to estimate the number of cache misses occurring when isolating data structures in a cache partition. Therefore, the tool is able to model the potential cache reuse during sparse computations using cache partitioning.

4.2.2 RESULTS

Fujitsu's A64FX processor comes with a lightweight cache partitioning hardware feature called *sector cache* and was used to evaluate the developed tool-based approach using the NAS parallel benchmarks. Multiple applications in the NAS parallel benchmarks have been identified to profit from cache partitioning, one of them being the CG benchmark class C. CG is an iterative conjugate gradient method based on a CSR SpMV kernel (see (algorithm 1)). It is found that isolating the matrix data (`colidx` and a) in a cache partition with minimal space of the L2 cache improves reuse of the source vector x by protecting x from being thrashed by the streaming accesses to the matrix data `colidx` and a.

Table 4 shows the measured and predicted L2 cache miss reduction using the sector cache for the CSR SpMV kernel with the sparse matrix data of the CG benchmark class C as input using 1 and 12 cores (A64FX L2 is shared by 12 cores). The predicted cache miss reduction is in line with the actual measurements using hardware performance events. However, the potential benefit of cache partitioning for that particular sparsity pattern is low. In an ongoing further investigation in collaboration with Simula, a matrix with a more irregular sparsity pattern (*Lynx68_reordered*[52]) in conjunction with a ELLPACK format SpMV kernel was profiled with the tool. Even though this matrix is already reordered for optimal cache reuse, the tool indicates that cache partitioning can even further reduce occurring cache misses (see Table 4). Actual measurements of cache misses for this matrix using cache partitioning on the A64FX processor will be provided in the future. Also other matrices and storage formats are part of ongoing further investigation. For example the *SELL-C-sigma*[53] storage format was used by Alappat et al.[54] and the authors also showed that SpMV performance can be improved significantly using the sector cache feature of the A64FX processor in a SpMV kernel based on this storage format.

[52]Trotter, Langguth, and Cai, "Cache simulation for irregular memory traffic on multi-core CPUs: Case study on performance models for sparse matrix–vector multiplication".

[53]Moritz Kreutzer et al. "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units". *SIAM Journal on Scientific Computing* 36.5 (2014), pp. C401–C423.

[54]Christie Alappat et al. "Execution-Cache-Memory modeling and performance tuning of sparse matrix-vector multiplication and Lattice quantum chromodynamics on A64FX". *Concurrency and Computation: Practice and Experience* (2021), e6512.

Table 4 *Average measured and predicted L2 cache miss reduction of SpMV kernels using the sector cache of the A64FX processor.*

Cores	Matrix	Format	L2 miss reduction m. [%]	L2 miss reduction p. [%]
1	CG class C	CSR	0.92	1.37
12	CG class C	CSR	1.25	1.23
1	Lynx68_reordered	ELLPACK	-	4.44
12	Lynx68_reordered	ELLPACK	-	11.26

5 DIGITAL SUPERTWIN

A Digital Twin is a structured collection of information emitted from a real-world system. Digital twins capture both structural and sensory data from their physical counterparts and allow for in-depth historical analysis, real-time monitoring, simulation, and prediction of the entity it models.

Although there are several digital twin ontologies in the literature for modeling industrial machines,[55] cities[56],[57] smart buildings[58] and even earth,[59] up to our knowledge, there is little to no work on ontologies to describe computers as cyber-physical systems. Some of the well-known ontologies are; SOSA (Sensor, Observation, Sample, Actuator)[60] which is used to describe industrial pipelines and FOAF (Friend of a Friend) which is used to describe relations among people. Moreover, there are several vocabularies used to describe digital twins, such as RDF (Resource Description Framework) and OWL (Web Ontology Language). Ontologies using these vocabularies allow static information to be located and queried using web interfaces via SPARQL endpoints. These frameworks are widely used to represent web-based interactions. However, these ontologies are used to keep static data. In our case, we used DTDL (Digital Twin Definition Language) to design and implement the proof-of-concept of our SuperTwin which is a digital twin of a supercomputer and can be used to model and analyze the performance and energy consumption of sparse kernels.

DTDL is developed by Microsoft for IoT frameworks, therefore provides a more suitable basis for describing computers. It is made up of six metamodel classes that describe the context of digital twin components. These classes are; `Interfaces`, `Telemetry`, `Properties`, `Commands`,

[55]Charles Steinmetz et al. “Internet of Things Ontology for Digital Twin in Cyber Physical Systems”. *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*. 2018, pp. 154–159. DOI: 10.1109/SBESC.2018.00030.

[56]Tianhu Deng, Keren Zhang, and Zuo-Jun (Max) Shen. “A systematic review of a digital twin city: A new pattern of urban governance toward smart cities”. *Journal of Management Science and Engineering* 6.2 (2021), pp. 125–134. ISSN: 2096-2320. DOI: https://doi.org/10.1016/j.jmse.2021.03.003. URL: https://www.sciencedirect.com/science/article/pii/S2096232021000238.

[57]Ehab Shahat, Chang T. Hyun, and Chunho Yeom. “City Digital Twin Potentials: A Review and Research Agenda”. *Sustainability* 13.6 (2021). ISSN: 2071-1050. DOI: 10.3390/su13063386. URL: https://www.mdpi.com/2071-1050/13/6/3386.

[58]Qiuchen LuVivi et al. “Developing a Dynamic Digital Twin at a Building Level: using Cambridge Campus as Case Study”. *International Conference on Smart Infrastructure and Construction 2019 (ICSIC)*, pp. 67–75. DOI: 10.1680/icsic.64669.067. URL: https://www.icevirtuallibrary.com/doi/abs/10.1680/icsic.64669.067.

[59]Jensen Huang. *Nvidia to build earth-2 supercomputer to see our future*. 2022. URL: https://blogs.nvidia.com/blog/2021/11/12/earth-2-supercomputer/.

[60]Krzysztof Janowicz et al. “SOSA: A lightweight ontology for sensors, observations, samples, and actuators”. *Journal of Web Semantics* 56 (2019), pp. 1–10. ISSN: 1570-8268. DOI: https://doi.org/10.1016/j.websem.2018.06.003. URL: https://www.sciencedirect.com/science/article/pii/S1570826818300295.

`Relationships` and `Data Types`. In DTDL, every `Interface` is a digital twin on it's own with it's contents describing it's `Properties`, `Telemetry`, and `Relationships`. When combined, these enable to capture the hierarchical structure of a supercomputer and model every single component (e.g., CPU, GPU, memory subsystem, etc.) as a separate digital twin entity. Moreover, processes also can be modeled as digital twins and monitored via per-process kernel metrics. This approach, in turn, provides a fine-grained modeling of entities. For example, an L1 cache can be isolated from the system and analyzed against different settings, the same process can be monitored with different parameters and on different hardware, or a component's different firmwares can be tested against each other to find bugs or performance anomalies.

5.1 PROBING

To capture the structural information for the underlying system, a detailed probing is required. We probed the system and aimed to describe the system components with their specifications and their inter-/intra-relations. This ontology -although is still under development- aims to present each physical hardware component that produces performance metrics or affects the overall system performance. This probing must capture these relationships in a lightweight, easily adaptable, and generic way. To this end, we relied on widely available Linux tools to gather data. The system, network, and memory information are collected via `lshw` utility. The CPU, GPU (and other accelerators) as well as the memory/cache topology metadata are collected by parsing `likwid-topology` and `cpuid` tools. Disk info is probed from `/sys/block/*/device` and `SMART` utility when available. PMU information is collected with `libpfm4` library which can recognise model-specific registers and their events of virtually every x86 and ARM processor available on the market. Upon probing PMU information via `libpfm4`, every hardware performance metric core and uncore which `perfevent` can report are obtained. Some of the datapoints acquired while probing the system is presented at Table 5.

5.2 CONSTRUCTING DIGITAL TWIN

Using the information acquired via probing, the Digital Twin, a.k.a, SuperTwin, is generated. While constructing the "base digital twin", all the components on the physical system, their relations with other components, and every performance metric that the physical system can report are added to the digital twin independently. This approach could be seen in Algorithm 2. The system is considered as the node at depth $d = 0$, and all the other components are contained by it. For the components that also contain other components, such as CPUs, the same method is applied until no subcomponent is left. However, for every depth, every component has its metrics added as their telemetry if there is any metric in the metric namespace with their unique label. Moreover, note that at line 18, caches are created as they exist in the physical system. Instead of every thread having a cache subcomponent, there is a single cache component contained by every thread that shares it. To validate the generated JSON-LD presentation, Azure Digital Twin Explorer is used. A twin graph generated by Azure Digital Twin Explorer could be seen in Figure 41.

Although the number of the performance metrics a digital twin can report in theory is in the order of thousands, this is impractical due to the overhead of the sampling process. Hence, our base digital twin is instantiated into linked data models, which only have metrics that are configured to be sampled. Since in a computer's Digital Twin, both structural and sensory time-series (i.e., the telemetry) data will have importance for semantic queries to be executed, it is better to keep these data in an associated way. However, while time-series databases like InfluxDB are fast and efficient for processing time-series data, they can't keep much metadata

Figure 41 *Part of the twin graph generated by the Azure Digital Twin Explorer.*

Figure 42 *Management and linking of time-series and metadata.*

for knowledge management. On the contrary, inserting and querying time-series data into a document database such as MongoDB is impractical.[61] Therefore, the Digital SuperTwin requires two types of databases and an association between them. To this end, in our proposed design, while InfluxDB holds performance metrics as time-series data, MongoDB, which is a document store, holds the digital-twin instance (in JSON-LD format), the metadata of the monitored time interval, process, and/or framework. In addition, to associate them with telemetry data, a pointer to InfluxDB is stored, which is the place of the performance metrics regarding the MongoDB entry kept. This structure allows accessing per-observation data, w.r.t. the job type and component type, via queries on metadata.

[61]Friedemann, Marko, Wenzel, Ken, and Singer, Adrian. "Linked Data Architecture for Assistance and Traceability in Smart Manufacturing". *MATEC Web Conf.* 304 (2019), p. 04006. DOI: 10.1051/matecconf/201930404006. URL: https://doi.org/10.1051/matecconf/201930404006; Katarina Milenković et al. "Enabling Knowledge Management in Complex Industrial Processes Using Semantic Web Technology". English. *Proceedings of the 2019 International Conference on Theory and Applications in the Knowledge Economy*. 2019 International Conference on Theory and Applications in the Knowledge Economy, TAKE 2019 ; Conference date: 03-07-2019 Through 05-01-2020. 2019. URL: https://www.take-conference2019.com/.

Field	Probing	Field	Probing	Field	Probing
System	arch	CPU	model	Network	businfo
	os		number of sockets		firmware
	kernel		number of threads		ipv4
	motherboard		threads per core		link
	uuid		hyperthreading		model
GPU	id		tlb		serial
	L2 size		flags		speed
	SIMD width		min clock rate		vendor
	numa node		max clock rate		virtual
	clock rate		topology	Memory	clock
	compute capability	Cache	L1D associativity		model
	max registers per block		L1D cache group topology		size
	max threads per SP		L1D cache line size		slot
	max threads per block		L1D no sets		vendor
	memory size		L1D size	PMU	name
	memory bus width		L2 associativity		number of events
	memory clock rate		L2 cache group topology		number of counters
	name		L2 cache line size		max encoding
	number of SPs		L2 no sets		type
	shared mem per block		L2 size	PMU Event	PMU name
	surface alignment		L3 associativity		name
	texture alignment		L3 cache group topology		description
Disk	model		L3 cache line size		flags
	rotational		L3 no sets		Umask-*
	size		L3 size		modif-*

Table 5 *A subset of the data probed from the system. Except of `CPU, Cache` and `System`, system information is probed for every component of the same type.*

5.3 DIGITAL SUPERTWIN ROADMAP

SuperTwin is currently implemented as a prototype, and its development continues. From this point, several visualizations and extendability capabilities will be added. Firstly, dashboard generation will be implemented to facilitate both real-time and per-request monitoring and visual analysis. To increase extendibility, using the aforementioned individual digital twins approach, digital twin description will be generated using RDF vocabulary instead of DTDL, which can be converted back into DTDL. This migration will enable SuperTwin hardware components and their properties to be analyzed easily via web interfaces. After that, the focus will be on scaling SuperTwin from a single-node framework to a supercomputer framework. At the same time, machine learning models will be developed using acquired data on the sparse kernels that have been currently investigated by the consortium to see if the output of the ML models match with the state-of-the-art analysis.

```
1 def add_my_metrics(component):
2     for metric in available_metrics:
3         if(component.type == metric.type):
4             add_telemetry(component, metric)
5
6 def add_component(component, subcomponent):
7     add_to_twin(subcomponent)
8     add_my_metrics(subcomponent)
9     add_ownership(component, subcomponent)
10
11 def add_subcomponents(component, subcomponents):
12     for socket in system:
13         add_component(system, socket)
14         for core in socket:
15             add_component(socket, core)
16             for thread in core:
17                 add_component(socket, thread)
18                 for cache in cache_groups[thread]:
19                     add_component(thread, cache)
20
21 def create_twin(system_probing):
22     system = create_system()
23     add_subcomponents(system, cpus)
24     add_component(system, memory)
25     add_component(system, disks)
26     add_component(system, networks)
27     add_component(system, gpus)
28     add_component(system, proc)
29 }
```

Algorithm 2: Digital Twin is generated via both contextual and structural information probed from the system.

6 CONCLUSIONS

The Deliverable 1.2 elaborated the ongoing research developments conducted in the scope of the SparCity project that refer to the performance and energy-efficiency modeling and analysis/profiling tools, which are predominantly focused on tackling the challenges related with efficient sparse computing in different device architectures and systems.

For this purpose, an extensive analysis, validation and characterization of different sparse computation kernels was performed in the state-of-the-art insightful models, i.e., in the Original Roofline Model (ORM) and in the Cache-aware Roofline Model (CARM). To improve the CARM insightfulness, a micro-benchmarking methodology was proposed, which allows for scaling the performance upper-bounds according to the characteristics of the sparse kernels. Furthermore, a novel Mansard Roofline Model (MaRM) was also proposed, which incorporates a new set of architectural features related to the retirement constraints of modern processors to provide more realistic modeling of performance upper bounds when compared to the state-of-the-art models.

The proposed roofline models, analysis and methodology were rigorously evaluated for a large set of sparse matrices from SuiteSparse Matrix Collection by considering different sparsity patterns, characteristics and reordering algorithms, when characterizing SpMV and SpMM custom-build and Intel MKL sparse kernels, as well as when guiding the optimization of the second-order epistasis detection algorithm (use-case application in the SparCity project). The obtained results show that the proposed models allow for more precise characterization of the sparse computations than the state-of-the-art models, while also providing the insights that are in line with the Intel TopDown VTune analysis, for both single- and multi-core execution scenarios.

The roofline principles were also applied to the Graphcore Intelligent Processing Unit (IPU), in order to model the performance and energy-efficiency upper-bounds of this architecture. Since the IPU adopts a specific execution model, a new modeling approach was derived, which specifically considers different phases of the IPU execution, i.e., in-tile execution and inter-tile communication (exchange). For both execution phases, the performance, power consumption and energy-efficiency upper-bounds were modeled and experimentally validated with micro-benchmarking for in-tile execution, attaining a very high match with the theoretical model.

In addition, several communication modeling and analysis tools were proposed, which aim at identifying horizontal and vertical data movement within the memory hierarchy. These tools allow for detailed data movement profiling, which is crucial for communication optimization, data placement and cache partitioning. In particular, the support for AMD x86 architectures is provided in the extended inter-thread communication detection tool, with the tool's accuracy, sensitivity to sampling interval and time/memory overheads beeing experimentally verified across a large set of benchmarks. Moreover, a cache partitioning strategy based on reuse distance is proposed, coupled with a profiling tool that was developed to pinpoint parts of the application that may benefit from cache partitioning. The proposed work was applied to an iterative conjugate gradient method based on a CSR SpMV kernel, and the experimental results show that the improved reuse of the source vector and overall reduction in the L2 cache misses were attained in Fujitsu's A64FX processor. Finally, all the developed performance, energy-efficiency and communication models are expected to influence the design of the SparCity Digital Twin, which initial construction and probing mechanisms were documented herein.

REFERENCES

Advanced Micro Devices, Inc. *AMD Processors Accelerating Performance of Top Supercomputers Worldwide*. https://www.amd.com/en/press-releases/2021-11-16-amd-processors-accelerating-performance-top-supercomputers-worldwide. 2021.

Alappat, Christie et al. "Execution-Cache-Memory modeling and performance tuning of sparse matrix-vector multiplication and Lattice quantum chromodynamics on A64FX". *Concurrency and Computation: Practice and Experience* (2021), e6512.

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.

Bienia, C. et al. "The PARSEC benchmark suite: Characterization and architectural implications". *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008, pp. 72–81.

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

Corporation, Intel. *Intel® oneAPI Math Kernel Library*. Intel. URL: https://software.intel.com/en-us/mkl.

Cuthill, E. and J. McKee. "Reducing the Bandwidth of Sparse Symmetric Matrices". Association for Computing Machinery, 1969.

Davis, Timothy and Yifan Hu. "The University of Florida Sparse Matrix Collection". *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–25. ISSN: 1557-7295. DOI: 10.1145/2049662.2049663.

Deng, Tianhu, Keren Zhang, and Zuo-Jun (Max) Shen. "A systematic review of a digital twin city: A new pattern of urban governance toward smart cities". *Journal of Management Science and Engineering* 6.2 (2021), pp. 125–134. ISSN: 2096-2320. DOI: https://doi.org/10.1016/j.jmse.2021.03.003. URL: https://www.sciencedirect.com/science/article/pii/S2096232021000238.

Diener, Matthias et al. "Characterizing communication and page usage of parallel applications for thread and data mapping". *Performance Evaluation* 88-89 (2015), pp. 18–36.

Diener, Matthias et al. "Communication in Shared Memory: Concepts, Definitions, and Efficient Detection". *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2016.

Doerfler, Douglas et al. "Applying the roofline performance model to the Intel Xeon Phi Knights Landing processor". *Proceedings of the International Conference on High Performance Computing*. Springer. 2016, pp. 339–353.

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, Marko, Wenzel, Ken, and Singer, Adrian. "Linked Data Architecture for Assistance and Traceability in Smart Manufacturing". *MATEC Web Conf.* 304 (2019), p. 04006. DOI: 10.1051/matecconf/201930404006. URL: https://doi.org/10.1051/matecconf/201930404006.

George, Alan. "Nested Dissection of a Regular Finite Element Mesh". *SIAM Journal on Numerical Analysis* 10.2 (1973), pp. 345–363.

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

Greathouse, Joseph L. *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.

Howarth, Jack. *AMD vs Intel 2022: Which Should be Your First Gaming CPU?* https://www.wepc.com/cpu/compare/amd-vs-intel-gaming/. 2022.

Huang, Jensen. *Nvidia to build earth-2 supercomputer to see our future*. 2022. URL: https://blogs.nvidia.com/blog/2021/11/12/earth-2-supercomputer/.

Ilic, Aleksandar, Frederico Pratas, and Leonel Sousa. "Beyond the Roofline: Cache-Aware Power and Energy-Efficiency Modeling for Multi-Cores". *IEEE Transactions on Computers* 66.1 (2016), pp. 52–58.

— "Cache-aware Roofline model: Upgrading the loft". *IEEE Computer Architecture Letters* 13.1 (2013), pp. 21–24.

Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-040. Intel Corporation, 2018.

— *VTune Profiler*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. [Online; visited June-2022].

Janowicz, Krzysztof et al. "SOSA: A lightweight ontology for sensors, observations, samples, and actuators". *Journal of Web Semantics* 56 (2019), pp. 1–10. ISSN: 1570-8268. DOI: https://doi.org/10.1016/j.websem.2018.06.003. URL: https://www.sciencedirect.com/science/article/pii/S1570826818300295.

Knowles, Simon. "Graphcore". *2021 IEEE Hot Chips 33 Symposium (HCS)*. 2021, pp. 1–25. DOI: 10.1109/HCS52781.2021.9567075.

Kolodziej, Scott P et al. "The suitesparse matrix collection website interface". *Journal of Open Source Software* 4.35 (2019), p. 1244.

Koskela, Tuomas et al. "A novel multi-level integrated Roofline model approach for performance characterization". *Proceedings of the International Conference on High Performance Computing*. Springer. 2018, pp. 226–245.

Kreutzer, Moritz et al. "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units". *SIAM Journal on Scientific Computing* 36.5 (2014), pp. C401–C423.

Li, Jiajia et al. "A Sparse Tensor Benchmark Suite for CPUs and GPUs". *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2020, pp. 193–204.

Little, John DC. "A proof for the queuing formula: L= λ W". *Operations research* 9.3 (1961), pp. 383–387.

LuVivi, Qiuchen et al. "Developing a Dynamic Digital Twin at a Building Level: using Cambridge Campus as Case Study". *International Conference on Smart Infrastructure and Construction 2019 (ICSIC)*, pp. 67–75. DOI: 10.1680/icsic.64669.067. URL: https://www.icevirtuallibrary.com/doi/abs/10.1680/icsic.64669.067.

Marques, Diogo, Aleksandar Ilic, and Leonel Sousa. "Mansard Roofline Model: Reinforcing the Accuracy of the Roofs". *ACM Trans. Model. Perform. Eval. Comput. Syst.* 6.2 (2021). ISSN: 2376-3639. DOI: 10.1145/3475866. URL: https://doi.org/10.1145/3475866.

Marques, Diogo et al. "Application-driven Cache-Aware Roofline Model". *Future Generation Computer Systems* 107 (2020), pp. 257–273. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2020.01.044. URL: https://www.sciencedirect.com/science/article/pii/S0167739X19309586.

Marques, Diogo et al. "Performance analysis with Cache-Aware Roofline model in Intel Advisor". *Proceedings of the International Conference on High Performance Computing & Simulation*. IEEE. 2017, pp. 898–907.

McCalpin, John D. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Department of Computer Science School of Engineering and Applied Science, University of Virginia. 2013. URL: https://www.cs.virginia.edu/stream/.

Milenković, Katarina et al. "Enabling Knowledge Management in Complex Industrial Processes Using Semantic Web Technology". English. *Proceedings of the 2019 International Conference on Theory and Applications in the Knowledge Economy*. 2019 International Conference on Theory and Applications in the Knowledge Economy, TAKE 2019 ; Conference date: 03-07-2019 Through 05-01-2020. 2019. URL: https://www.take-conference2019.com/.

Nobre, Ricardo et al. "Exploring the Binary Precision Capabilities of Tensor Cores for Epistasis Detection". *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2020, pp. 338–347.

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.

Shahat, Ehab, Chang T. Hyun, and Chunho Yeom. "City Digital Twin Potentials: A Review and Research Agenda". *Sustainability* 13.6 (2021). ISSN: 2071-1050. DOI: 10.3390/su13063386. URL: https://www.mdpi.com/2071-1050/13/6/3386.

Srivastava, N. et al. "MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product". *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 766–780. DOI: 10.1109/MICRO50266.2020.00068.

Srivastava, N. et al. "Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations". *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 689–702. DOI: 10.1109/HPCA47549.2020.00062.

Steinmetz, Charles et al. "Internet of Things Ontology for Digital Twin in Cyber Physical Systems". *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*. 2018, pp. 154–159. DOI: 10.1109/SBESC.2018.00030.

Trotter, James D., Johannes Langguth, and Xing Cai. "Cache simulation for irregular memory traffic on multi-core CPUs: Case study on performance models for sparse matrix–vector multiplication". *Journal of Parallel and Distributed Computing* 144 (2020), pp. 189–205. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2020.05.020.

Unat, D. et al. "Trends in Data Locality Abstractions for HPC Systems". *IEEE Transactions on Parallel and Distributed Systems* 28.10 (2017), pp. 3007–3020. ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2703149.

Vuduc, Richard et al. "Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply". *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. Baltimore, Maryland: IEEE Computer Society Press, 2002, pp. 1–35. DOI: 10.1109/SC.2002.10025.

Wang, Endong et al. "Intel math kernel library". *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.

Williams, Samuel, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". *Commun. ACM* 52.4 (2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.

Yasin, Ahmad. "A top-down method for performance analysis and counters architecture". *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2014, pp. 35–44.

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)*. 2020.