Scientific evaluation report of the SparCity framework

Deliverable No:	D5.3
Deliverable Title:	Scientific evaluation report of the SparCity framework
Deliverable Publish Date:	31 March 2024
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

*Until M21

CONTENTS

1 INTRODUCTION

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

Sparse computations are commonly found at the heart of many important applications, but at the same time, it is challenging to achieve high performance when performing such sparse computations. SparCity delivers a coherent collection of innovative algorithms and tools for enabling 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 main objective of Deliverable 5.3 is to provide a scientific evaluation of the software libraries, tools, and methods of the SparCity framework, which have been developed or extended during year 3 of the project. These will be evaluated with respect to the functionality and usability, as well as accuracy when applicable. (The elements of the SparCity framework that were finalized during the first two years of the project have already been evaluated in the preceding Deliverables 5.1 & 5.2. The evaluation of these will therefore not be repeated in this deliverable.)

1.2 WORK PERFORMED

The content of this deliverable is an evaluation of the scientific results from SparCity in three categories: *methods* (Section 2), *tools* (Section 3), and *libraries* (Section 4). Each element in the three categories will have a brief introduction, a summary of its functionality, and an evaluation of its usability and accuracy (if relevant). The evaluation is typically carried out through real-world examples of usage and effect.

The five academic partners of SparCity (Koc, Sabanci, Simula, INESC-ID, and LMU) have contributed substantially and collaboratively to the various elements. The industrial partner, Graphcore, has contributed with technical support as well as frequent and in-depth discussions with the other partners during the first 20 months of the project.

1.3 DEVIATIONS AND COUNTER MEASURES

There was no noteworthy deviation from the research plan that is related to the development and application of the SparCity framework.

2 SPARCITY METHODS

2.1 ML-BASED RECOMMENDATION METHODS

Sparse matrix-vector multiplication (SpMV) is a key kernel in many applications from different domains, and it often turns out to be a performance bottleneck for these applications. The performance of an SpMV kernel varies widely among instances of the same size, because it depends on several factors such as the sparsity pattern and the storage format for the matrix, in addition to the architecture and memory hierarchy of the processor. This has given rise to techniques that alleviate this problem by selecting the optimal storage formats, algorithms, and reorderings for a given combination of input matrix and architecture.

2.1.1 ML-BASED SPARSE MATRIX FORMAT SELECTION

Our early work has investigated the format selection problem.[1] In contrast to existing work in the area, we focused on the portability and explainability of the ML-based recommendations. Our results indicate that ensemble learning techniques such as Random Forest or XGBoost yield excellent accuracy, as well as portability of results. Transfer learning is highly effective here, providing competitive accuracies with retraining times which are considerably less than that of the original times. In contrast to earlier work, we found that approaches based on CNNs are less viable when dealing with large instances. The features used in the ensemble learning approach are order-invariant, which limits the applicability of this approach to other recommendation problems.

2.1.2 ML-BASED SPMV ALGORITHM SELECTION

For a given SpMV format, there are multiple different algorithms that have different advantages and disadvantages depending on the instance. A key characteristic is the load-balancing strategy for multicore processors. Here, we distinguish between 1D (row-based) and 2D (row- and column-based) load balancings. While 1D algorithms are standard, sophisticated 2D algorithms only recently became more common.[2]

We performed a large-scale comparison of the 1D and 2D algorithms using all larger instances from the SuiteSparse collection.[3] Since the number of instances in SuiteSparse is small, we enhance the test set with 17 large matrices. We split the test set by the number of nonzeroes into four groups: *very small* matrices having less than 10^6 nonzeroes, *small* with 10^6 to 10^7, *medium* with 10^7 to 10^8 nonzeroes, and *large* matrices having more than 100 million nonzeroes. Performance results showed that especially processors with large core counts benefit from the 2D approach. The experiment also included reorderings, which are discussed in the next subsection.

Note that the overhead of using the 2D algorithm is very small and can be amortized over multiple runs. Therefore, using ML to predict the better algorithm, while possible, is unlikely to improve the time to solution when repeated SpMV operations are performed.

[1] Sunidhi Dhandhania et al. "Explaining the Performance of Supervised and Semi-Supervised Methods for Automated Sparse Matrix Format Selection". *50th International Conference on Parallel Processing Workshop*. 2021, pp. 1–10.

[2] Duane Merrill and Michael Garland. "Merge-Based Sparse Matrix-Vector Multiplication (SpMV) Using the CSR Storage Format". *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2016.

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

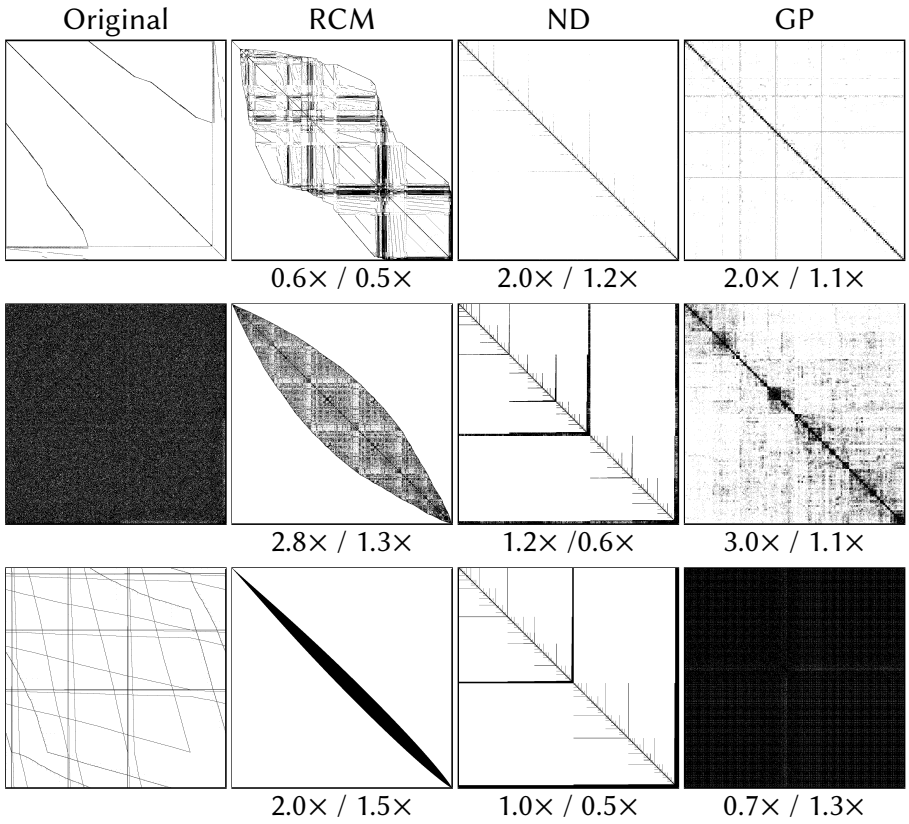


Figure 1 *Matrices reordered with Reverse Cuthill-McKee (RCM), Nested Dissection (ND) and graph partitioning (GP). The numbers below represent speedup (or slowdown) of SpMV on 64-core AMD Epyc Milan and 36-core Intel Ice Lake CPUs, respectively.*

2.1.3 ML-based sparse reordering performance prediction

In a typical CSR matrix with four-byte indices and eight-byte values, SpMV consumes a little more memory bandwidth than 12 bytes per nonzero when all vector elements are cached. If instead one cache line must be fetched for every vector value, the memory bandwidth requirement increases to 76 bytes on a typical CPU, thus causing a slowdown of more than 6× in the extreme case. Memory access latency may make this effect even worse.[4]

This problem can be alleviated by using Reverse Cuthill-McKee (RCM)[5] or other reordering algorithms that improve cache locality. Doing so can also indirectly improve load balance. However, when using 1D SpMV algorithms, reordering for cache reuse can also worsen load imbalance. This happens with matrices that have a widely varying number of nonzeroes per row and a relatively even distribution of longer and shorter rows among the cores. Figure 1 illustrates the effects of different reorderings.

A typical example is Kronecker graphs which are generated for the Graph500 benchmark.[6] In this case, a random ordering of the rows generally ensures a good load balance for 1D algorithms.

[4]Johannes Langguth et al. "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes". *Journal of Parallel and Distributed Computing* 76 (2015), pp. 120–131. DOI: 10.1016/j.jpdc.2014.10.005.

[5]Elizabeth Cuthill. "Several Strategies for Reducing the Bandwidth of Matrices". *Sparse Matrices and their Applications*. Springer, 1972, pp. 157–166.

[6]Richard C Murphy et al. "Introducing the Graph 500". *Cray Users Group (CUG)* 19 (2010), pp. 45–74.

However, this load balance will typically disappear when applying an RCM reordering, since the bandwidth minimization does not consider load balance and may, e.g., cluster denser rows.

For this reason, we focused first on predicting the performance gains of using the RCM algorithm combined with the 2D SpMV method. This problem differs from the format selection problem because the features that are commonly used for SPMV format selection are not sensitive to the row order of the matrix. Thus, they are not usable when predicting the effects of matrix reorderings since they are identical for all possible orderings of a given matrix. Therefore, we proposed two new order-dependent features based on simplified simulations of the cache: the *group reuse rate* and the *cache reuse rate*. Both are relatively simple and, unlike CNNs, can be computed efficiently even for large graphs. Furthermore, they do not lose information with increasing matrix size, which is a problem with CNNs since they have to scale large matrices to fit their input size.

The first feature is called *group reuse rate*. It attempts to capture spatial locality through a simple, single-line cache model for a straightforward SpMV algorithm. We assume the single available cache line consists of N consecutive elements which are always loaded simultaneously from memory. We assume that matrix nonzeros are accessed in row-major order. For the first nonzero of the matrix, a load is triggered which moves N consecutive vector elements into the cache, starting with the element corresponding to the position of the nonzero. For each subsequent nonzero, if the corresponding vector element is in the cache, a reuse event is triggered and we move on to the next nonzero. Otherwise, a new load event is triggered, as described above. For simplicity, our model of the group reuse rate does not assume that cache lines begin and end at multiples of the cache line size N. Although this is different from how CPU caches operate in reality, the model works well enough in practice.

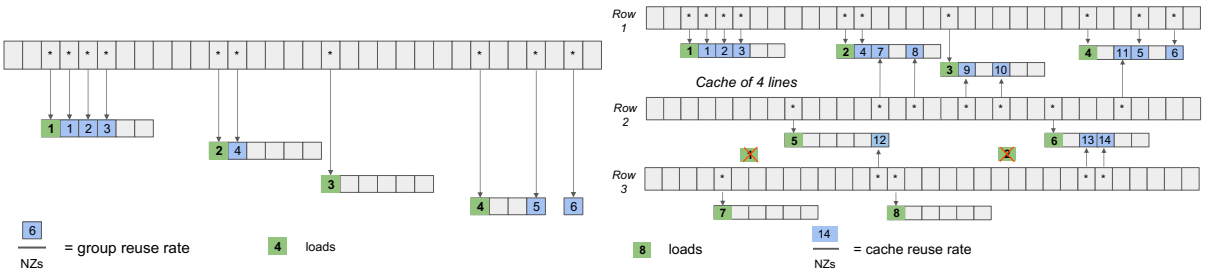


Figure 2 *An example of the group reuse rate (left) and the cache reuse rate (right).*

After all the nonzeros have been accessed, the group reuse rate is obtained by simply dividing the number of reuse events by the number of nonzeros in the matrix. Since each cache line has one load and up to $N-1$ reuse events, the best possible reuse rate is $(N-1)/N$.

We also introduce a second type of feature which we call *cache reuse rate*. It represents a simplified multi-line cache access simulation implemented for a simple SpMV algorithm. As with the *group load and reuse rates*, a cache line consists of N elements, but now there are M cache lines organized in an ordered list. Again, the nonzeros in the matrix are accessed consecutively, and the first element triggers a load event.

For each subsequent nonzero, if the corresponding vector element is currently present in any of the already loaded cache lines, a reuse event is triggered and the cache line that contained the cached element is moved to the top of the cache line access list. If the vector element is not currently in the cache, a load event will be triggered, loading another cache line which is placed on top of the cache line access list. If at least M cache lines have been loaded, the line on the bottom of the access list must be evicted before a new cache line can be loaded. This is part of a

Table 1 *Prediction quality of the random forest classifier for medium and large instances. Left: performance metrics. Right: Percentage of maximum performance reached by the different reordering strategies.*

	large	medium
True positives	159	457
True negatives	26	9
False positives	11	95
False negatives	0	2
Accuracy	0.94	0.83
Precision	0.94	0.83
Sensitivity	1	1
Specificity	0.7	0.09
F1 score	0.97	0.9

	Prediction	Always RCM	Never RCM
Large			
Epyc 7763	99.38	96.77	73.23
Epyc 7601	99.27	97.65	77.15
Epyc 7413	99.78	96.07	72.34
Epyc 7302P	99.08	94.19	79.02
Xeon 6130	99.07	94.8	78.78
Xeon 8360Y	99.97	99.28	79.29
Medium			
Epyc 7763	84.42	84.16	74.8
Epyc 7601	97.4	98.19	89.08
Epyc 7413	83.67	82.3	71.98
Epyc 7302P	96.54	97.4	83.5
Xeon 6130	96.87	97.23	79.16
Xeon 8360Y	98.35	98.06	90.93

single-load event. In this manner, we simulate a simple fully associative least recently used (LRU) policy. Once the simulation is finished, the cache reuse rate is obtained by dividing the number of reuse events by the number of nonzeros in the matrix. An example of both features is shown in Figure 2. By default, we compute this feature with N set to 64 and M to 65536.

Based on the structure-dependent features, we develop a qualitative performance prediction model in order to determine whether a given matrix can be reordered to increase SpMV performance. Thus, we have to solve a classification problem with two classes. For this purpose, we use a standard Random Forest classifier. The Random Forest classifier not only performs well for small datasets, but it can also handle input features of different scales without pre-normalization of the feature values. We train two classifiers, one for large matrices with more than 100 million nonzeroes, and one for medium-sized matrices with 10 to 100 million nonzeroes. There is no need for a classifier for smaller instances since they can run entirely out of cache.

To verify the accuracy of the classifiers, we show the standard classification performance metrics in Table 1 on the left. The classifier for the large matrices shows very good accuracy. However, for the medium-size matrices, the classification performance is lower.

In Table 1, right side, we show the effect of applying the predictions. We weigh every matrix by the number of nonzeroes in order to reflect that mispredictions on larger matrices are more costly. Clearly, RCM is beneficial for most, but not all matrices, and the classifier correctly predicts that in almost all cases. Furthermore, applying RCM when it is not needed is very costly. While we used a slow sequential code, based on the fastest parallel implementation of RCM,[7] reordering takes roughly as much time as 300 SpMV iterations. About 14% of the instances do not benefit from RCM, and in these cases using our system saves this cost.

Thus, we have presented a first classifier that can successfully predict whether applying a reordering is beneficial for SpMV performance, a question that often has a higher impact than the format selection problem which was studied earlier.

[7] Ariful Azad et al. "The reverse Cuthill-McKee algorithm in distributed-memory". *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 22–31.

Table 2 *Sparse matrix reordering algorithms used in this study*

Short Name	Reordering Algorithm	Description
RCM[9]	Reverse Cuthill–McKee	bandwidth reduction via breadth-first graph traversal
AMD[10]	Approximate minimum degree	local greedy strategy to reduce fill by selecting sparsest pivot
ND[11]	Nested dissection	recursive divide-and-conquer using vertex separators to redu
GP[12]	Graph partititoning	multi-level recursive graph partitioning with METIS using ed
HP[13]	Hypergraph partititoning	column-net hypergraph partitioning with PaToH using cut-ne
Gray[14]	Gray code ordering	splitting of sparse and dense rows and Gray code ordering

2.1.4 EXTENDED REORDERING PERFORMANCE EXPERIMENTS

Alternatives to the Reverse Cuthill-McKee (RCM) algorithm were also studied and the results published.[8] The paper contains the details of this investigation. To provide an overview, we categorise reordering algorithms into 1) bandwidth-reducing orderings, 2) fill-reducing orderings 3) graph and hypergraph partitioning-based orderings, and 4) other orderings.

Bandwidth-Reducing Orderings: Well-known examples of such orderings include the Cuthill-McKee (CM) algorithm[15] and the method described by Gibbs et al..[16] The CM ordering attempts to reduce the matrix bandwidth through a breadth-first search of the undirected graph corresponding to a symmetric sparse matrix. The vertices of the graph, which correspond to rows and columns of the matrix, are ordered by choosing a starting vertex (e.g., by finding a pseudo-peripheral vertex[17]) and then traversing the graph in breadth-first search order, where the vertices at each level are sorted in ascending order by degree. In the end, after traversing the entire graph, the ordering may be reversed to obtain the more commonly used Reverse Cuthill-McKee (RCM)[18] ordering.

Fill-Reducing Orderings: Minimum degree orderings[19] arise in the context of reducing fill-in during sparse Cholesky factorisation. The elimination graph of a sparse symmetric matrix consists of a vertex for every row, as well as edges between any pair of vertices a and b for which row a has a nonzero above the diagonal in column b. At each step of the factorisation, one row is eliminated by removing a vertex and its edges in the elimination graph, and replacing it with a clique consisting of the former neighbours of the vertex. The new edges that are created by forming such a clique lead to fill-in of the Cholesky factor. The minimum degree algorithm is a graph-based heuristic to find node orderings with low amounts of fill by always selecting a vertex

[8] James D Trotter et al. "Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2023, pp. 1–13.

[15] E. Cuthill and J. McKee. "Reducing the Bandwidth of Sparse Symmetric Matrices". *Proceedings of the 1969 24th National Conference*. Association for Computing Machinery, 1969, pp. 157–172. DOI: 10.1145/800195.805928.

[16] Norman E. Gibbs, William G. Poole, and Paul K. Stockmeyer. "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix". *SIAM Journal on Numerical Analysis* 13.2 (1976), pp. 236–250. ISSN: 00361429.

[17] Alan George and Joseph W. H. Liu. "An Implementation of a Pseudoperipheral Node Finder". *ACM Transactions on Mathematical Software* 5.3 (1979), pp. 284–295. DOI: 10.1145/355841.355845.

[18] Wai-Hung Liu and Andrew H Sherman. "Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices". *SIAM Journal on Numerical Analysis* 13.2 (1976), pp. 198–213.

[19] Alan George and David R. McIntyre. "On the Application of the Minimum Degree Algorithm to Finite Element Systems". *SIAM Journal on Numerical Analysis* 15.1 (1978), pp. 90–112. ISSN: 00361429. URL: http://www.jstor.org/stable/2156565; Alan George and Joseph WH Liu. "The evolution of the minimum degree ordering algorithm". *SIAM Review* 31.1 (1989), pp. 1–19; Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. "Algorithm 837: AMD, an Approximate Minimum Degree Ordering Algorithm". *ACM Trans. Math. Softw.* 30.3 (2004), pp. 381–388. ISSN: 0098-3500. DOI: 10.1145/1024074.1024081.

of least degree.

Another commonly used fill-reducing ordering is Nested dissection (ND),[20] which is based on computing a vertex separator for the undirected graph of a symmetric sparse matrix. The two subgraphs that arise from removing the separator are ordered first, while rows and columns corresponding to the separator are moved to the end of the matrix. This process is applied recursively for the two subgraphs. The underlying motivation for the ND ordering is that it incurs low fill-in for sparse Cholesky factorization if the separators are small. Since the method relies on graph partitioning, it can be grouped under graph partitioning-based orderings as well.

(Hyper)graph partitioning-based orderings: Graph partitioning can be used to define an ordering by directly partitioning a matrix into a given number of parts, then grouping rows and columns by their assigned parts. This approach is frequently used in a distributed-memory setting to perform work division of sparse matrix operations, and the same idea can be applied to the shared-memory case. METIS[21] is a well-known graph partitioning tool that can be used to partition large irregular graphs. It is based on the multilevel paradigm which consists of the graph coarsening, initial partitioning, and uncoarsening phases. The aim of the partitioning is to minimize a partitioning objective, while obeying a load balancing criteria.

Hypergraph partitioning may similarly be used for reordering. PaToH[22] is a commonly-used hypergraph partitioning tool which is known to reflect the actual communication volume requirement of parallel SpMV. Hypergraphs are the generalization of graphs, in which the hyperedges (nets) can be incident to any number of vertices instead of exactly two vertices in simple graphs. The hypergraph partitioning problem is the task of dividing a hypergraph into roughly balanced parts such that the cutsize is minimized. Other reorderings based on hypergraph partitioning include the separated block diagonal form proposed by Yzelman and Bisseling.[23]

Other Orderings: A number of alternative matrix orderings have been proposed with the goal of improving data locality in SpMV, including approaches based on the travelling salespesrson problem[24] and space-filling curves.[25] One particular method, which we call Gray ordering,[26] is motivated by microarchitectural concerns to reduce branch mispredictions and improve data locality for SpMV. First, to improve branch prediction, rows with similar nonzero density are grouped together (density reordering). Second, to improve locality, a bitmap-based reordering is applied, where each row is segmented into multiple sections of nonzeros (to construct the row

[20] Alan George. "Nested Dissection of a Regular Finite Element Mesh". *SIAM Journal on Numerical Analysis* 10.2 (1973), pp. 345–363. DOI: 10.1137/0710032; J. R. Gilbert and R. E. Tarjan. "The Analysis of a Nested Dissection Algorithm". *Numer. Math.* 50.4 (1987), pp. 377–404. ISSN: 0029-599X. DOI: 10.1007/BF01396660.

[21] George Karypis and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 359–392. DOI: 10.1137/S1064827595287997.

[22] U.V. Catalyurek and C. Aykanat. "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication". *IEEE Transactions on Parallel and Distributed Systems* 10.7 (1999), pp. 673–693. DOI: 10.1109/71.780863.

[23] A. N. Yzelman and Rob H. Bisseling. "Cache-Oblivious Sparse Matrix–Vector Multiplication by Using Sparse Matrix Partitioning Methods". *SIAM Journal on Scientific Computing* 31.4 (2009), pp. 3128–3154. DOI: 10.1137/080733243.

[24] Ali Pinar and Michael T. Heath. "Improving Performance of Sparse Matrix-Vector Multiplication". *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. Portland, Oregon, USA: Association for Computing Machinery, 1999. DOI: 10.1145/331532.331562; D.B. Heras et al. "Modeling and improving locality for the sparse-matrix–vector product on cache memories". *Future Generation Computer Systems* 18.1 (2001), pp. 55–67. ISSN: 0167-739X. DOI: 10.1016/S0167-739X(00)00075-3.

[25] Leonid Oliker et al. "Effects of Ordering Strategies and Programming Paradigms on Sparse Matrix Computations". *SIAM Review* 44.3 (2002), pp. 373–393. DOI: 10.1137/S00361445003820.

[26] Haoran Zhao et al. "Exploring Better Speculation and Data Locality in Sparse Matrix-Vector Multiplication on Intel Xeon". *2020 IEEE 38th International Conference on Computer Design (ICCD)*. 2020, pp. 601–609. DOI: 10.1109/ICCD50377.2020.00105.

Table 3 *Hardware used in our experiments.*

	Skylake	Ice Lake	Naples	Rome	Milan A	Milan B	TX2	Hi1620
CPUs	Intel Xeon Gold 6130	Intel Xeon Plat-inum 8360Y	AMD Epyc 7601	AMD Epyc 7302P	AMD Epyc 7413	AMD Epyc 7763	Cavium TX2 CN9980	HiSilicon Kunpeng 920-6426
Instr. set	x86-64	x86-64	x86-64	x86-64	x86-64	x86-64	ARMv8.1	ARMv8.2
Microarch.	Skylake	Ice Lake	Zen	Zen 2	Zen 3	Zen 3	Vulcan	TaiShan v110
Sockets	2	2	2	1	2	2	2	2
Cores	2×16	2×36	2×32	1×16	2×24	2×64	2×32	2×64
Freq. [GHz]	1.9–3.6	2.4–3.5	2.7–3.2	1.5–3.3	2.5–3.5	2.5–3.5	2.0–2.5	2.6
L1I/core [KiB]	32	32	64	32	32	32	32	64
L1D/core [KiB]	32	48	32	32	32	32	32	64
L2/core [KiB]	1024	1280	512	512	512	512	256	512
L3/socket [MiB]	22	54	64	16	128	256	32	64
Bandwidth [GB/s]	256	409.6	342	204.8	409.6	409.6	342	342

bitmaps), which are then labeled and ordered based on the Gray code.[27] In general, the matrix is first split into dense and sparse submatrices according to the number of nonzeros in each row, while the density and bitmap reorderings are applied depending on the characteristics of each submatrix.

The hardware used in our experiments is shown in Table 3. All codes are compiled with GCC 11.2.0 with the `-O3` and `-march=native` options on each node, and the test systems are running Ubuntu 18.04.6.

Our evaluation relies on the SuiteSparse Matrix Collection.[28] We apply six reorderings (see Table 2) to 490 matrices that are square, non-complex and have between 1 million and 1 billion nonzeros. On converting the matrices to CSR format, column offsets are stored as 32-bit integers and nonzero values as double precision floating point numbers. In the case of symmetric matrices, whenever an offdiagonal nonzero is encountered, two nonzeros are inserted into the CSR representation, one in the upper and another in the lower triangle of the matrix.

Each SpMV run is repeated 100 times, and we take the maximum performance among the runs. This represents the peak performance of a system with a warm cache and is less susceptible to noise than the average. Note that for smaller matrices used in our evaluation, some or all of the data may fit in last-level cache. For example, the AMD Epyc 7763 has the largest last-level cache at a total of 512 MiB. Only 77 matrices have more than 45 million nonzeros, which is the minimum size needed to exceed the capacity of the last-level cache if matrices are stored in CSR format.

Figures 3 and 4 provide a comprehensive analysis of the relationship between matrix reordering algorithms and their performance on SpMV. Through a large sparse matrix dataset using six broadly used reordering algorithms on eight state-of-the-art multicore architectures, we showed

[27] Sardar Anisul Haque and Shahadat Hossain. "A Note on the Performance of Sparse Matrix-vector Multiplication with Column Reordering". *2009 International Conference on Computing, Engineering and Information*. 2009, pp. 23–26. DOI: 10.1109/ICC.2009.40.

[28] Timothy A. Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". *ACM Trans. Math. Softw.* 38.1 (2011). ISSN: 0098-3500. DOI: 10.1145/2049662.2049663.

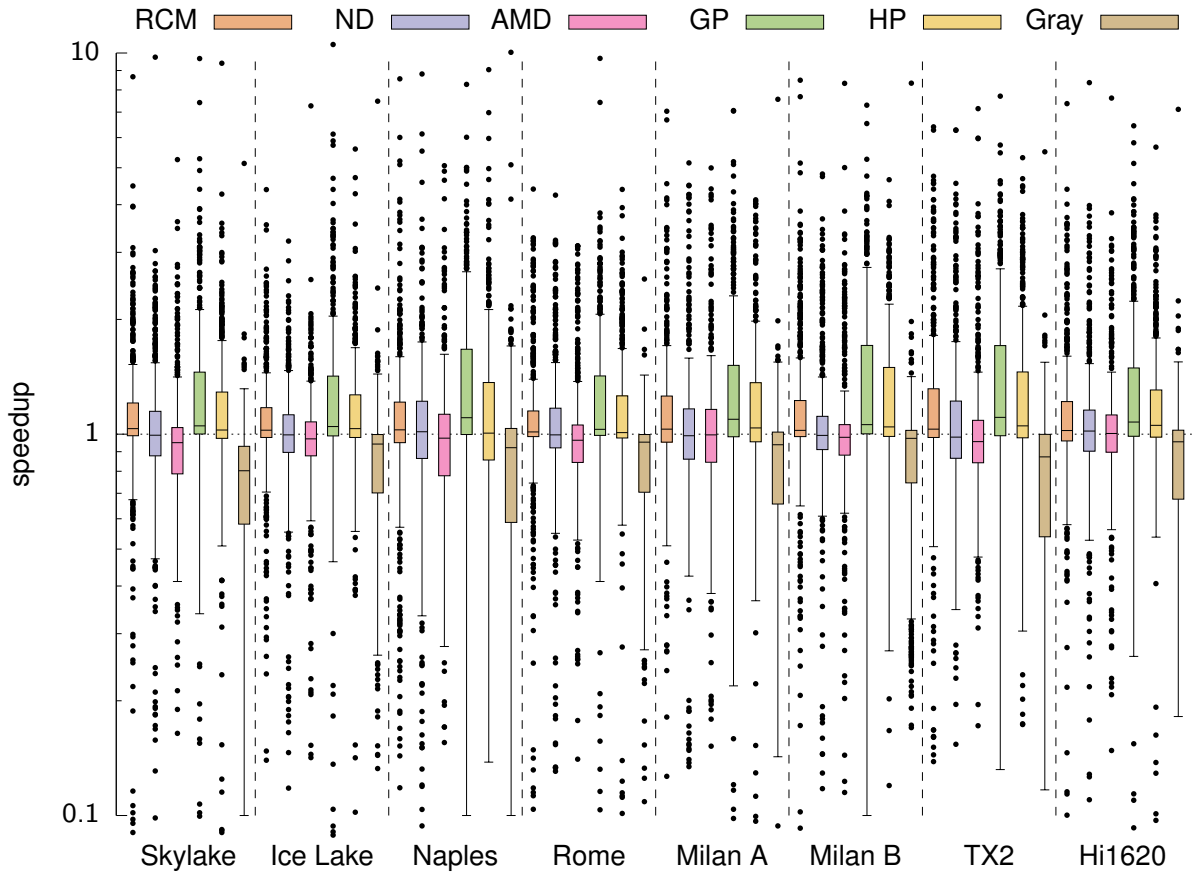


Figure 3 *Speedup of sparse matrix-vector multiplication using 1D algorithm after reordering. For each box, the middle line represents the median and endpoints represent the lower and upper quartiles.*

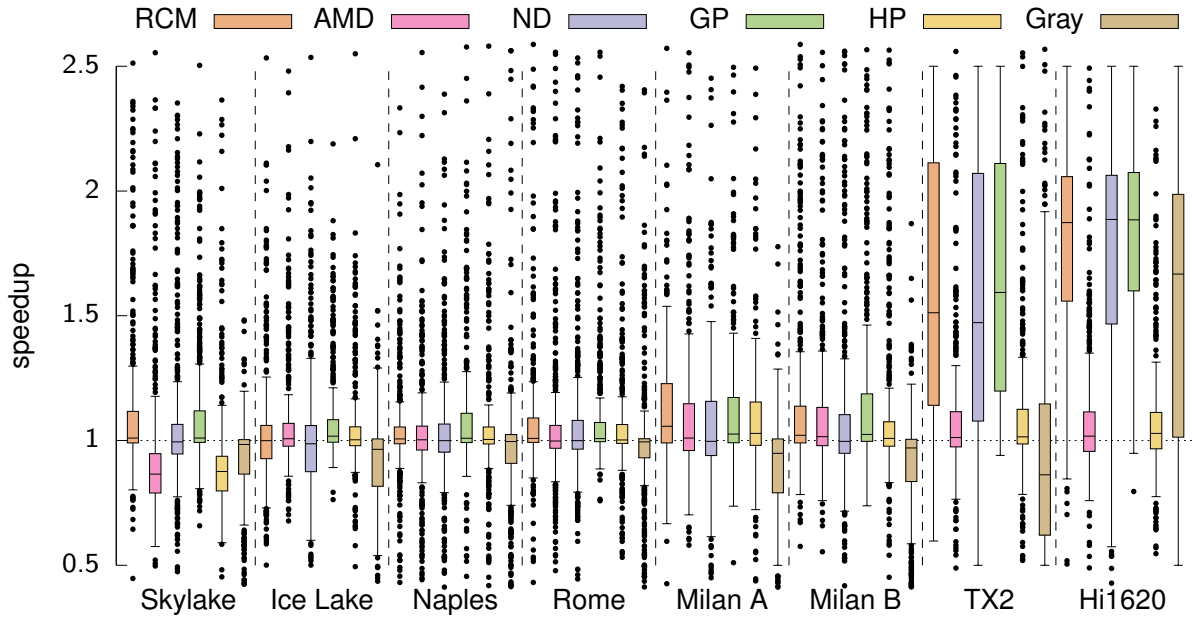


Figure 4 *Speedup of the nonzero-balanced CSR SpMV kernel (2D algorithm) after reordering.*

how the effectiveness of reordering relates to various factors, such as reordering algorithm, load balancing concerns, and off-diagonal nonzero counts. The results indicate that graph partitioning is generally the preferred reordering method. Furthermore, reorderings are far more relevant on ARM based processors than on x86. The techniques discussed in Section 2.1.3 can be applied to all reordering methods tested here.

2.2 AUTOMATED KERNEL FUSION

Kernel fusion refers to the optimization technique in computer programming where multiple computational operations, known as kernels, are combined or fused into a single, more efficient unit of execution by reducing the overhead associated with preparing and launching sequential kernels, as well as enforcing more cohesive access to resources such as memory, resulting in better cache utilization. Kernel fusion is done automatically or manually. Automated kernel fusion is a process where software tools or compilers automatically combine multiple computational kernels into a single optimized unit. In contrast, manual kernel fusion is done ad-hoc and by developers, via combining two or more kernels manually into an optimized unit.

Kernel fusion has been extensively performed on CPUs[29] to achieve increases in performance and reduction in energy usage. However, GPUs are the leading accelerator in modern High Performance Computing (HPC) systems, equipping 7 of the 10 leading Top500[30] supercomputers in the world today. GPUs are frequently used to solve scientific and computational problems, as they offer hundreds to thousands of cores in contrast with dozens on a CPU, albeit slower and with more limitations. Kernel fusion has been applied extensively in GPUs as well.[31]

In recent years, graphs with billions of vertices have been used to represent a diverse number of real-world problems such as scientific computations, social networks, machine learning and biology. Speeding up graph processing algorithms can improve the efficiency of these computations. As such, many graph processing algorithms are proposed and the area is well studied in recent years.[32]

[29]Hongzheng Chen et al. "Krill: a compiler and runtime system for concurrent graph processing". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–16; Peitian Pan and Chao Li. "Congra: Towards efficient processing of concurrent graph queries on shared-memory machines". *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE. 2017, pp. 217–224; Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. "Multilyra: Scalable distributed evaluation of batches of iterative graph queries". *2019 IEEE International Conference on Big Data (Big Data)*. IEEE. 2019, pp. 349–358; Jilong Xue et al. "Seraph: an efficient, low-cost system for concurrent graph processing". *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 2014, pp. 227–238; Jin Zhao et al. "GraphM: an efficient storage system for high throughput of concurrent graph processing". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–14.

[30]Erich Strohmaier. "TOP500 supercomputer". *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. Tampa, Florida: Association for Computing Machinery, 2006, 18–es. DOI: 10.1145/1188455.1188474. URL: https://doi.org/10.1145/1188455.1188474.

[31]Guibin Wang, YiSong Lin, and Wei Yi. "Kernel fusion: An effective method for better power efficiency on multithreaded GPU". *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*. IEEE. 2010, pp. 344–350; Mohamed Wahib and Naoya Maruyama. "Scalable kernel fusion for memory-bound GPU applications". *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2014, pp. 191–202; Jiří Filipovič et al. "Optimizing CUDA code by kernel fusion: application on BLAS". *The Journal of Supercomputing* 71.10 (2015), pp. 3934–3957.

[32]Yunming Zhang et al. "Graphit: A high-performance graph dsl". *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–30; Chen et al., "Krill: a compiler and runtime system for concurrent graph processing"; Julian Shun and Guy E Blelloch. "Ligra: a lightweight graph processing framework for shared memory". *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2013, pp. 135–146; Pan and Li, "Congra: Towards efficient processing of concurrent graph queries on shared-memory machines"; Yangzihao Wang et al. "Gunrock: A high-performance graph processing library on the GPU". *Proceedings of the 21st ACM SIGPLAN*

Within the SparCity project, we introduced an automated framework for concurrent graph processing kernel fusion in GPUs with the following contributions: We developed and implemented of an automated kernel fusion framework. We tested four graph jobs using the framework, namely Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), PageRank (PR) and Label Propagation (LP). We developed a meta-compiler to enable memory-efficient shared data structures for computational jobs, as well as facilitation of static polymorphism. We performed extensive evaluation of the framework with the implemented jobs on two different platforms with different combinations of up to 100 parallel jobs, demonstrating about 2% to 10% increase in runtime performance with kernel fusion.

2.2.1 KERNEL FUSION FRAMEWORK

To enable kernel fusion within GPU, we developed a framework to streamline creation, dispatching and fusion of different GPU jobs. The framework is created in C++, with supporting scripts in other languages. Graphs are represented as adjacency lists using the **AdjacencyGraph** class. They are read from files and stored in CPU and GPU memory as adjacency lists (two arrays for nodes and edges respectively). GPU jobs inherit from the base class **AbstractJob**, which denotes whether or not a job is active, what its current frontier is, and whether or not it is finished. **AbstractJob** also assists in initialization of jobs, by streamlining memory allocation and movement between CPU and GPU using helper methods.

Each job that inherits from **AbstractJob** needs to explicitly define 4 sections:

1. Auxiliary data (metadata): constants or variables needed by the job, usually involving several integer variables such as starting node index, as well as at least one array the size of the graph (e.g., visited nodes, distance, etc.).

2. Constructor: sets the initial variables of a job, such as the starting node, number of iterations, etc.

3. Initializer: is used to allocate and initialize job data in the GPU and assign their respective data pointers.

4. GPU-specific kernel: used to do the actual computation (in each iteration).

Auxiliary data needs to be defined in a specific order at a certain section of the job definition, as the meta-compiler relies on these definitions to construct the **FusionJob** class (as explained below). The GPU kernel receives the GPU thread id and the input graph. It has to perform both its core computation function, as well as update the job's frontier in the graph (i.e., what the next active nodes are going to be), and mark whether the job is completed or not. The framework also includes a **RunJobs** function that iterates over a list of jobs, performs kernel fusion (if enabled) and iterates over all jobs via a GPU kernel called **IterJobs** until all jobs are done. **RunJobs** is also in charge of collecting runtime and performance metrics from CPU, GPU, fusion algorithm, etc.

IterJobs is the GPU entry point of the framework, executed once per graph node. It is in charge of polymorphically calling all job kernels for each graph node, and passing them their respective metadata. **IterJobs** also checks whether jobs are still active and whether they are still not finished, before dispatching them. All jobs that are active and not completed will be dispatched, in order, during **IterJobs**. Finally, the framework includes **Fusion**, which is in charge of fusing job kernels based on their data access patterns to improve performance. Another

symposium on principles and practice of parallel programming. 2016, pp. 1–12; Xue et al., "Seraph: an efficient, low-cost system for concurrent graph processing".

component in the framework is an accurate timer used to profile different sections and activities and provide detailed time measurements.

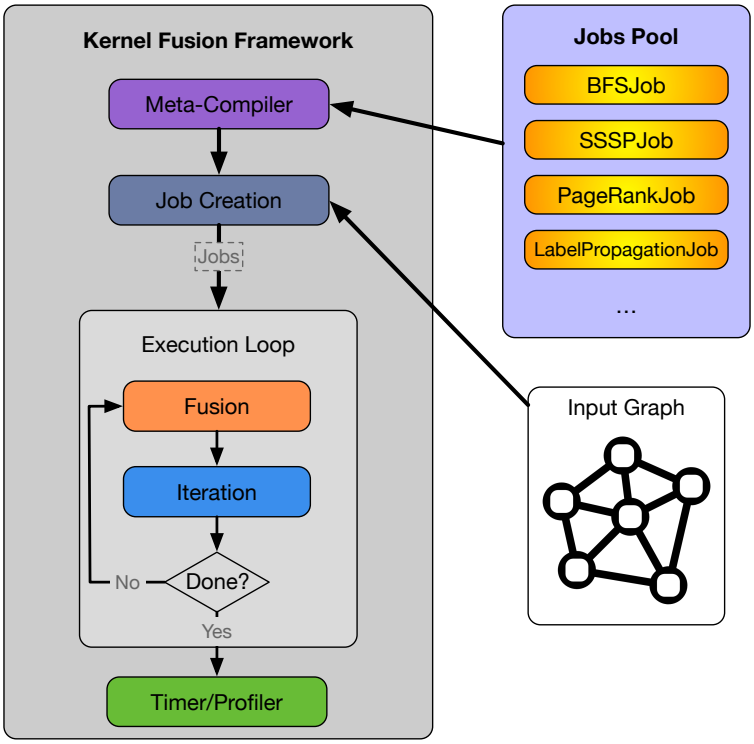


Figure 5 *The architecture of the kernel fusion framework. The defined jobs in the job pool are compiled by the meta-compiler, and then multiple jobs are created based on an input graph. The created jobs are then passed on to the execution loop, which fuses and runs them until they are all done.*

Figure 5 provides an overview of the described architecture, while Figure 6 provides an overview of the framework and how it works. Line ① defines the input graph, loads it from the input file (into both the CPU and the GPU), and assigns it to a variable. Line ② creates several jobs based on the defined criteria. Each created job is automatically initialized by allocating necessary memories, and setting initial metadata such as starting node, distance arrays, etc. Line③ is the start of simplified **RunJobs** function, where jobs are executed until all of them are done. Line④ performs kernel fusion for the jobs (if enabled). The result of this fusion is that some jobs will be active (for the current iteration), while others will be inactive. Line⑤ finally launched the GPU kernel **IterJobs** that runs the jobs in the GPU.

The rest of the listing covers the IterJobs GPU kernel. This kernel is executed as many times as there are nodes in the input graph. Line⑥ assigns the id of the current GPU thread to the variable **id**. The subsequent line ensures that **id** is not larger than the number of nodes in the graph. Line⑦ – the starting line of the loop that is executed as many times as the number of jobs – makes sure that the job is active in this iteration. Without fusion, all jobs are always active. With fusion, only jobs that are selected by the fusion algorithm are active. Line⑧ skips any job that is already finished. Line⑨ makes sure that the current job in the current execution for the current thread id has an active frontier, i.e., nodes that are actively being worked on. Line ⑩ and subsequent lines are auto-generated by the meta-compiler to enable polymorphism, and run the job-specific iteration function. This is further expanded on in Section 2.2.1.

```
❶ AdjacencyGraph graph = LoadGraph(file);
❷ FusionJobs jobs[] = CreateJobs(job_count, graph);
❸ while (not AllJobsDone()):
    ❹ if (isFusionEnabled) Fusion(jobs);
    ❺ IterJobs<<<graph.size/1024, 1024>>>(jobs, graph);

__global__ IterJobs(FusionJob jobs[], AdjacencyGraph graph):
  ❻ id = blockIdx.x * blockDim.x + threadIdx.x;
  if (id >= graph.size) return;

  for (i = 0; i < jobs.count; i++):
    ❼ if (not jobs[i].active) continue;
    ❽ if (jobs[i].done) continue;
    ❾ if (not jobs[i].frontier[id]) continue;
    ❿ // MetaCompiler Generated Below:
    if (jobs[i].job_type == JobTypes::BFS)
      ((BFSJob)jobs[i]).iter(id, graph);
    else if (jobs[i].job_type == JobTypes::SSSP)
      ((SSSPJob)jobs[i]).iter(id, graph);
    else if (jobs[i].job_type == JobTypes::PageRank)
      ((PageRankJob)jobs[i]).iter(id, graph);
    else
      print("Unsupported job type!");
```

Figure 6 *The simplified overall flow of the fusion framework.*

Fusion Polymorphism. Since the framework needs to run different jobs at runtime (as a list of jobs), and because it is a compiled C++ program, it needs to utilize polymorphism. However, this polymorphism is partially performed on the CPU (initialization, fusion) and partially on the GPU (kernels, fusion).

This work investigated three avenues for implementation of this polymorphism: C++ virtual functions and inheritance (native polymorphism), Runtime Type-Information (RTTI) based polymorphism, i.e., adding extra variables and code to jobs that defines their runtime types and behaviors, and meta-compiler static polymorphism, i.e., generating code based on available types and only passing minimal job type information at runtime via switch statements.

Native polymorphism enjoys somewhat limited support in CUDA. Specifically, it requires objects to be allocated on CUDA heap directly, otherwise virtual function tables will not be *on device* (i.e., in GPU memory). However, the default heap is quite limited, and can only be resized once per program, to a maximum of about 80% of GPU memory, i.e., it cannot be sized down later. Furthermore, our evaluations show that there is a significant performance hit for using native C++ polymorphism in CUDA. For these reasons, we decided to use other alternatives.

RTTI-based polymorphism requires specific compiler flags and passing of additional variables and data to the GPU, as well as allocation of objects on GPU heap within GPU code. Due to these limitations, this approach was also not desirable.

Finally, as a solution we decided to use a meta-compiler that adds small sections of code based on available GPU jobs and then compiles them to native code.

Meta-Compiler. The meta-compiler scans the code for all defined jobs that inherit from **AbstractJob**, and creates a list of available jobs as a C++ Enum. For example in Figure 7 this enum is then used in each class's constructor to explicitly set its type in a member variable called **job_type**, as apparent in ②. The meta-compiler subsequently adds a small section of switch statements inside **IterJobs** that dispatches the appropriate job kernel based on the type of the job instance. Finally, and most importantly, the meta-compiler needs to create a new **FusionJob** structure, that includes metadata of all available jobs in as little memory as possible, as this data is allocated and passed to the GPU in bulk for all jobs at the same time (for performance reasons).

```
❶ enum JobTypes {
    Abstract,
    BFS,
    SSSP,
    PageRank,
  };

❷ class BFSJob: public AbstractJob {
 public:
      JobTypes job_type; // Holds type of job for static polymorphism.
      BFSJob(u64 root) : root(root) {
          job_type = BFS;
      }
```

Figure 7 *The C++ enum of all jobs generated by the meta-compiler, as well as its usage in constructor of jobs to define their runtime type.*

FusionJob Structure. To generate the code for FusionJob structure, the meta-compiler needs to scan all available jobs and determine which metadata each one uses. The order of usage of this metadata, as well as their variable names and variable types are then used to create a class called **FusionJob** that uses C++ unions to minimize memory footprint, while allowing access to all these variables and their respective types in each job.

Kernel Fusion. As depicted in Figure 5, before each iteration, the framework performs kernel fusion (if enabled). Kernel fusion can be disabled to compare runtime performance with and without fusion. Additionally, the framework distinctively reports total job completion time, time spent on GPU, and time spent on fusion, as well as the time spent in each of the fusion steps.

Algorithm. The fusion algorithm finds the shared frontier of all jobs, i.e., the union of all the nodes that all jobs will work on in the next iteration. It then finds the subset of this shared frontier that is maximally used by all remaining jobs. Finally, the algorithm marks jobs that will work on this maximal shared frontier in the next iteration, as active, and all other jobs as inactive. This algorithm ensures that memory accesses to nodes on this maximal shared frontier will be shared by several jobs, resulting in effective caching in memory access which can significantly speedup job execution.

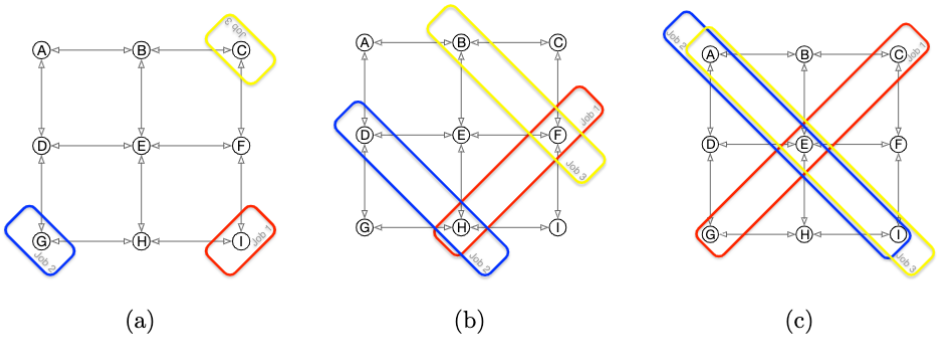


Figure 8 *The expanding frontier of 3 jobs running in parallel*

Figure 8 shows 3 jobs traversing the same graph from three different starting nodes, node I, G and C, respectively. In the next iteration (b), the jobs have expanded to access nodes (F, H) for job 1, (D, H) for job 2 and (B, F) for job 3. Notice that no node is shared between all three jobs, whereas node H is shared between job 1 and job 2, and node F is shared between jobs 1 and 3. As such, fusion can deactivate job 3, and run jobs 1 and 2, making them access node H simultaneously, resulting in one less uncached memory access.

In the next iteration (c), each job has an expanding frontier that encompasses 3 nodes. This is a typical trend in graph traversal, in the first few iterations, the frontiers are quite small, but they exponentially expand to include a significant portion of the entire graph, and then gradually dwindle in size to reach zero once the traversal is completed. As apparent from the Figure, in the third iteration, nodes (A, E, I) are shared between jobs 2 and 3, while node E is shared between all three jobs. If the fusion algorithm picks jobs 2 and 3 to run, there will be 3 memory accesses for traversing 3 nodes by 2 jobs (i.e., 6 node accesses), pushing execution of job 1 to the next iteration with its own distinct 3 memory accesses; whereas if the fusion algorithm picks all 3 jobs, there will be 5 memory accesses for traversing 5 nodes by 3 jobs (i.e., 9 node accesses). The latter selection results in a total of 5 memory accesses for completion of iteration 3 of all jobs, in contrast with 6 memory accesses in the former.

Our evaluations show that accessing the same memory regions by concurrent jobs can result up to 500 times faster runtime compared to sporadic memory access by different jobs. This speedup is achieved by running all jobs in the same order, starting from the same node and traversing the graph in identical patterns, in contrast with each job starting at a different point or performing a different traversal.

Finding the maximal shared frontier is a computationally intensive algorithm. Frontiers of all jobs need to be scanned first – each of which is potentially as large as the entire graph – to find the shared frontier among all jobs. To find the *maximal* shared frontier within the previously discovered shared frontier, the fusion algorithm needs to count how many times each node in the shared frontier is accessed by each job. Once the nodes with maximum access count are discovered, jobs that have these nodes in their frontier need to be listed, and subsequently marked as active (while all other jobs are deactivated).

As such, the framework implements the part of the fusion algorithm that finds the maximal shared frontier, inside the GPU itself (to follow a more CPU-independent execution model[33]), avoiding the need to copy frontiers of each job back to the CPU, and reducing the footprint of

[33] Ismayil Ismayilov et al. "Multi-GPU Communication Schemes for Iterative Solvers: When CPUs are Not in Charge". *Proceedings of the 37th International Conference on Supercomputing*. 2023, pp. 192–202.

fusion from around 10% of job execution time to between 1% to 2% of job execution time.

2.2.2 EVALUATION

In this section, datasets, environment setup and job lists are described, then multiple experiments are performed to evaluate the effectiveness of the proposed GFuse framework.

Table 4 *List of graphs used in evaluations dataset.*

	Graphs	Vertex Count	Edge Count	Diameter
1	LiveJournal	4,847,571	68,993,773	16
2	Patents	6,009,555	16,518,948	22
3	Higgs	456,631	14,855,875	9
4	Pokec	1,632,803	30,622,564	11
5	Youtube	1,134,890	2,987,624	20
6	Wiki-Talk	2,394,385	5,021,410	9

The graph dataset used for evaluations is listed in Table 4. Six widely used graphs – with varying sizes and structures – were selected. All of these graphs are extensively used in previous works[34] , and their sizes range from 500k nodes to 5 million nodes in size.

Four popular graph processing algorithms – widely used by previous work and in the industry – were selected and implemented on top of the framework as jobs. The implementations are straightforward and not particularly optimized, following the job definition patterns defined in the framework. These algorithms include Breadth-First Search (BFS), Bellman-Ford Single-Source Shortest Path (SSSP), PageRank (PR) and Label Propagation (LP) algorithm, as listed in Table 5.

To demonstrate the efficacy of the automated kernel fusion framework in different work load scenarios, 4 different job sets were devised.These job sets can be observed in Table 6. Job sets are all homogeneous meaning all parallel jobs are of the same algorithm although with different parameters such as starting nodes.

As this work is the first to automatically fuse graph processing algorithms together on a GPU, to show the performance gain of the proposed fusion method, all evaluations compare the same execution with and without the fusion algorithm. To this end, each experiment is repeated once with fusion enabled, and once with fusion disabled.

As previously mentioned, all experiments are also performed on each platform with an increasing job count that goes from 1 parallel job up to 100. However, not all platforms have enough GPU memory to contain the metadata necessary for 100 parallel jobs. In those cases, the job count is incremented until the GPU memory is exhausted. In the evaluations, the platform with the smallest GPU memory (8GB) is able to run 65 instances of the most demanding job (memory-wise) in parallel. Experiments were also repeated to ensure spikes and anomalies are not temporal.

Two distinct platforms were used to perform our extensive evaluations. The first, a scientific computation server containing one instance of NVIDIA A30 GPU with 32 GB of GPU RAM, Intel Xeon Gold 6258R CPU (2.70GHz) with 32 GB of RAM. The second, an Amazon Web Services (AWS) G5.xlarge instance[35] with one instance of NVIDIA A10 GPU with 24 GB of GPU RAM, AMD EPYC 7R32 CPU with 16 GB of RAM.

[34] Chen et al., "Krill: a compiler and runtime system for concurrent graph processing"; Pan and Li, "Congra: Towards efficient processing of concurrent graph queries on shared-memory machines".

[35] Amazon Web Services. *Amazon EC2 G5 Instances*. https://aws.amazon.com/ec2/instance-types/g5/. 2023.

Table 5 *Graph processing algorithms selected and implemented as jobs for evaluation.*

	Algorithm	Description
1	BFS	Breadth first search
2	SSSP	Single-source shortest path
3	PageRank (PR)	Measuring relative importance
4	Label Propagation (LP)	Labelling and clustering of a graph

Table 6 *Job sets used in experiments and evaluations.*

Job Set	Algorithms
Homo 1	BFS X N
Homo 2	SSSP X N
Homo 3	PR X N
Homo 4	LP X N

NVIDIA driver version 510.47.03 and CUDA version 12.1 were used. The same source code was used on all instances, and compiled without any flags, which is the **-O3** optimization flag by default on the NVIDIA Compiler (**nvcc**).[36] Table 7 summarizes the platforms used in experiments and evaluations.

2.2.3 RESULTS

In this section, several experiments used to analyze the performance of the proposed automated fusion method in different job sets are presented. For the purpose of brevity and clarity only a slide of runtime performance of job sets are depicted in Figure 9, 10, 11.

Figure 9 illustrates runtime performance of homogeneous job set Homo1 for all input graphs with increasing job counts starting from 1 up to 100, executed in parallel. It is noteworthy to mention that the speedup starts when the job count is over 20. The average speedup and slowdown for different input graphs ranges from 10% speedup for Higgs (12.5% for job counts over 20) to 2.5% slowdown for Youtube (3.1% for job counts over 20). The job set Homo2 which include multiple number of SSSPs yield very similar results and for the purpose of brevity is not included.

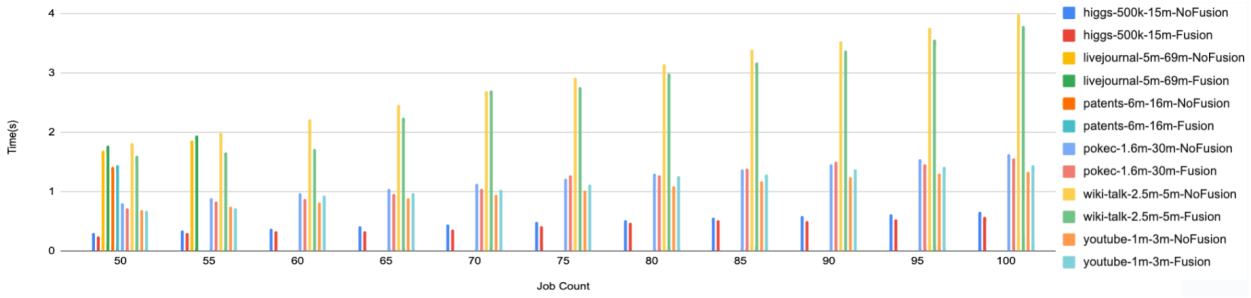


Figure 9 *Homogeneous job set Homo1 speedup result*

As depicted in Figure 10 , the speedup for the job set of Homo3 starts when the job count is over 25 with the notable exception of Youtube in which speedup starts when the number of jobs are over 60. Different input graphs have varying average speedup/slowdowns ranging from

[36] Nvidia Corporation. *NVIDIA CUDA Compiler Driver NVCC*. 2022. URL: https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_Data_Center_GPU_Driver_Release_Notes_510_v1.0.pdf.

Table 7 *Platforms used for evaluations.*

Name	GPU + RAM	CPU + RAM
Intel Cascade Lake Server	NVIDIA A30 32GB	Intel Xeon 6258R 32GB
AWS G5.xlarge	NVIDIA A10 24GB	AMD EPYC 7R32 16GB

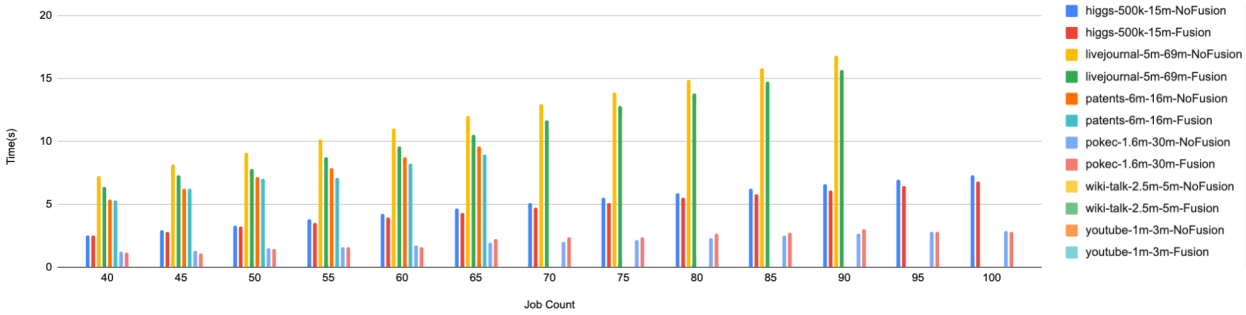


Figure 10 *Homogeneous job set Homo3 speedup result*

2%-3% of average speedup for Higgs and Wiki-Talk, 2% of average slowdown for Youtube and 8%-12% of speedup for Pokec depending on the job count.

The job set Homo4 as depicted in Figure 11, has consistent slowdowns with the noteworthy exception of Youtube graph that yields 5.5% average speedup. For the rest of input graphs this job set has varying slowdowns ranging from 2% to 8%.

As immediately observable in the figures, not all runtimes with fusion are better than their counterparts without fusion. This slowdown is specially pronounced when job count is small Additionally, slowdown is observed in certain algorithms such as Label Propagation on graphs that have a significantly large number of edges compared to the number of nodes, such as Higgs, Youtube and LiveJournal Figure 9, 10, 11.

As apparent in the figures, the runtime curve for fused jobs are generally below the curve for jobs without fusion for almost all graphs and algorithms in homogeneous job sets. This result is expected, and can be because similar jobs are more likely to have similar access patterns,[37] leading to similar frontier sets (both in size and shape) resulting in reduced memory access and improved runtime.

A notable exception to this improved performance is the Label Propagation algorithm in Figure 11.Most of the fusion curves have *decreased* performance in contrast with non-fusion curves, with the notable exception of the Youtube graph. This slowdown is because the Label Propagation algorithm needs to iterate all edges of each frontier node, to find the intersection of that node's neighbors with the starting node. As each node may have hundreds of thousands of edges, cache utilization quickly becomes ineffective when many concurrent LP jobs are executed, resulting in degraded performance.

Another visible anomaly is the *spikes* in runtime performance of Label Propagation on the Youtube graph (Figure 11). The fusion curve seems to be generally below the non-fusion curve, but has several spikes that sometimes go above the non-fusion curve (circa job count 70). Our hypothesis is that the shared frontier becomes too large to effectively utilize GPU memory cache at this job count, and as such fusion adds overhead instead of improving performance. To ascertain that these anomalous spikes are not due to temporal evaluation results, these experiments were repeated three more times.

[37] Chen et al., "Krill: a compiler and runtime system for concurrent graph processing".

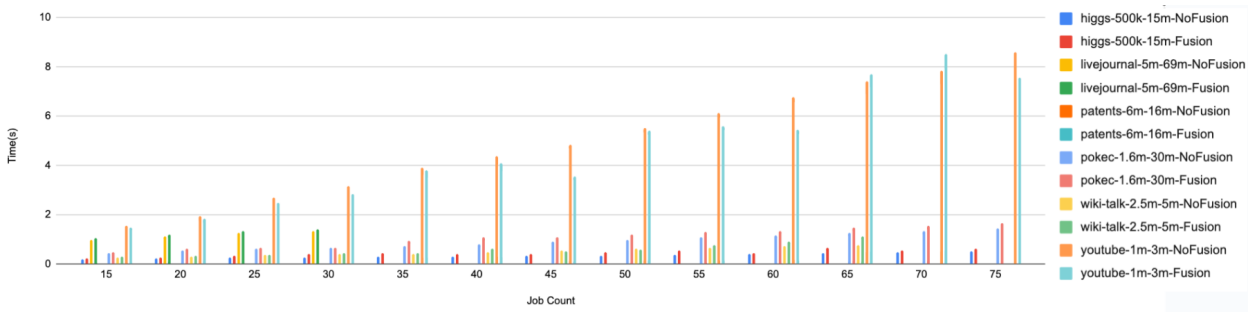


Figure 11 *Homogeneous job set Homo4 speedup result*

As mentioned before, for job counts below 20, there is usually negative speedup, as there isn't a significant shared frontier to fuse jobs together. However, as job count goes beyond 20 and towards 50, speedup increases significantly. Around 55 job count is where speedup starts to dip, although the average speedup is still about 10% thereafter.

Fusion Overhead. To measure overhead incurred by GFuse, it is first important to analyze how fusion can cause overhead. There are two aspects by which fusion adds overhead to the running time of jobs. The first is the running time of the fusion algorithm itself. The proposed framework uses two GPU kernels and a minimal CPU function – all three of which are heavily optimized – to perform kernel fusion. As such, the overhead caused by the fusion algorithm is generally negligible, averaging 61 milliseconds in contrast with an average job running time of 1.84 seconds on the first platform, 90 milliseconds in contrast with an average running time of 9.57 seconds on the second platform and 99 milliseconds in contrast with an average running time of 6.81 seconds on the third platform, resulting in 3.3%, 0.9% and 1.5% average fusion algorithm overheads.

Note that as apparent from Figure 9 and Figure 10, as the number of jobs and the overall execution time increases, the fusion algorithm overhead as a proportion reduces. This proportional reduction is the reason behind an average of 3.3% average fusion algorithm overhead on platform 1, that reduces to less than 1% in platform 2, which has a larger memory and supports many more parallel jobs.

The second aspect by which fusion adds overhead to the running time is by increasing the number of iterations required to finish all jobs, as certain jobs are deactivated and stalled by the fusion algorithm to maximize the active shared frontier. For example, running 100 homogeneous SSSP jobs on the **Youtube** graph with fusion, requires 42 iterations to complete, whereas the same job without fusion finishes in only 21 iterations. This overhead is significantly harder to measure and predict, as it is based on the shared frontier of jobs at runtime, depending on the runtime parameters of each job.

3 SPARCITY TOOLS

3.1 SPARSE-AWARE ROOFLINE MODELING

Sparse computation has been a topic of research for several years, due to its importance in fields such as graph analytics, artificial intelligence, data science and scientific computation.[38] Data in these applications is mostly comprised of zeros, thus it is represented with sparse matrices, where only non-zero elements are stored using diverse formats to reduce memory footprint. However, even for the most widely used sparse algorithms and formats, such as Sparse Matrix Vector Multiplication (SpMV) with Compressed Sparse Row (CSR), it is not trivial to identify the bottlenecks and improve their performance in current computing systems. This is mainly due to the diverse impact to performance, power consumption and energy efficiency when unpredictable and irregular memory access patterns dictated by the specific sparse matrix characteristics are coupled with increasing complexity of modern hardware.

For this purpose, performance models can be used to identify the main execution bottlenecks and give hints about optimization paths. One simple and insightful model that facilitates this process is Cache-Aware Roofline Model (CARM).[39] CARM offers intuitive analysis on the application bottlenecks through visual representation of the upper-bounds of the micro-architecture and application performance. However, when tackling sparse computation, this simplicity can lead to inaccurate characterization and optimization hints, due to the inability of the model to consider the realistic requirements of this type of computations, e.g., irregularity and indirect memory accesses.

In order to improve the applicability of CARM[40] to sparse computations, we proposed a novel micro-benchmarking methodology supported by a tool to achieve accurate and precise performance upper-bounds of current CPUs when performing sparse computations. The proposed methodology is used to build a sparse-aware CARM, which represent the limitations of the micro-architecture for the SpMV computation in a more accurate way. Rooflines are obtained based in micro-benchmarking with synthetic sparse matrices, specifically constructed to exercise the various component of the architecture. The proposed model retains the simplicity of the original model by relating the application performance with the hardware upper-bounds, while it significantly improves its insightfulness and applicability by providing additional hints for sparse-specific scenarios, such as possible reuse of involved data structures.

The proposed methodology to experimentally assess the SpMV performance upper-bounds takes in account two key aspects: the SpMV algorithm implementation and the disposition of non-zero elements in the sparse matrix. For the former, we focus our analysis on the hand-tuned SpMV kernel, developed in x86 assembly and operating on the sparse matrices in the most commonly used CSR format. For the latter, we developed a set of synthetic sparse matrices to exercise different memory access and data reuse patterns with the aim of fully exploiting the memory hierarchy and compute resources of modern CPUs. These achievements are extensively documented in Deliverables 1.2, 1.4, 1.5 and 4.2.

This methodology was experimentally verified in a computing platform with an octa-core Intel i7-7820X and Linux CentOS 7.5.1804, by varying the number of rows, non-zeros per row and ensuring the all data structures involved in SpMV fit in the L1 cache. It was observed that

[38] Xiaoping Li, Yadi Wang, and Rubén Ruiz. "A survey on sparse learning models for feature selection". *IEEE transactions on cybernetics* 52.3 (2020), pp. 1642–1660.

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

[40] Ibid.

the lowest bandwidth was achieved for the matrix with 512 rows and 1 NPR (512×1), which increases towards 8 NPR, thus utilizing the SpMV algorithm segments with higher unrolling factor, reaching a maximum for the 128×16 matrix. To extend this evaluation to other memory levels, two approaches are adopted that allow preserving the locality of x vector in a specific level: *i)* increase the number of rows while maintaining the number of columns (e.g., with 16 columns x is in L1 cache); and *ii)* change the number of columns to provoke different locality of x vector.

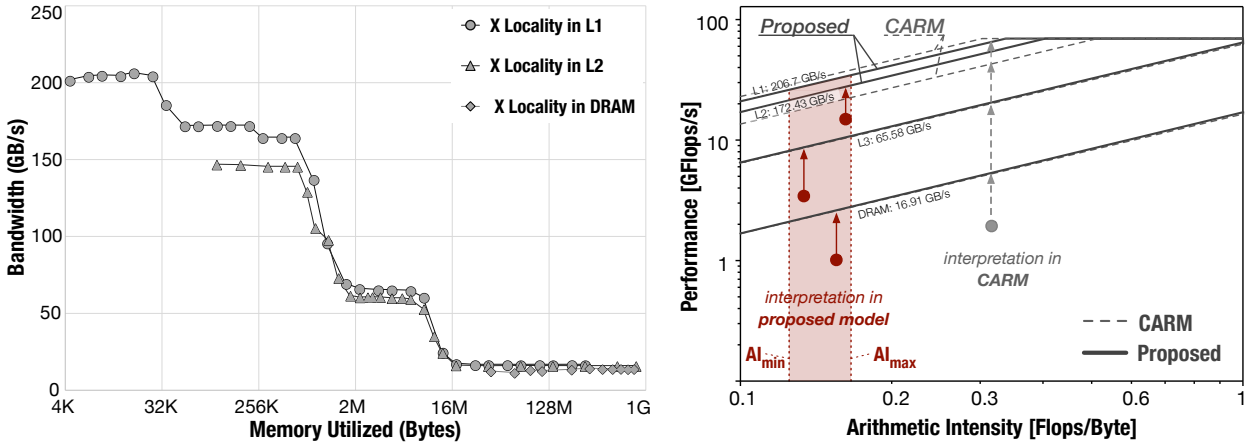


Figure 12 *Bandwidth variation (left) and sparse CARM (right) for multi-threaded SpMV.*

Figure 12 (left) shows the bandwidth variation for multi-threaded SpMV execution, as a function of the memory occupied by x and y vectors, and all CSR structures. For a given x vector locality, it can be seen that increasing the data size until exceeding the capacity of each memory level, causes a reduction in the maximum attainable bandwidth, since deeper memory levels are associated with higher access latency. The highest memory bandwidth is attained when the x vector fits in the L1 cache (for 16 columns, 206.7GB/s). When testing scenarios where x fits in other memory levels, e.g., in the L2 cache with 8702 columns, the resulting attainable L2 bandwidth suffers a bandwidth reduction. This difference in bandwidth is also noticeable for deeper memory levels (such as L3 and DRAM), where the increased access latency significantly reduces the impact of the locality of the x vector on overall bandwidth.

Based on these findings, a novel sparse-aware CARM is proposed, which is capable of more accurately characterizing sparse computation kernels and their ability to exploit the micro-architecture compute and memory resources, when compared to the state-of-the-art CARM.[41] Sparse-aware CARM is derived based on the bandwidth evaluation conducted and presented in Fig. 12 (right), depicting the performance upper-bounds of the SpMV kernel, when the x vector locality is preserved in the L1 cache.

When compared to the original CARM roofs (see dashed roofs in Fig. 12), the proposed sparse-aware CARM (solid roofs) achieves lower maximum attainable performance for the L1 cache. This is mainly due to indirect accesses to the x vector and memory accesses to multiple arrays, which prevent the theoretical L1 bandwidth to be reached. However, proposed L2 roof is higher than the one in original CARM, since the locality of x vector is preserved in the L1 cache (original CARM L2 roof is obtained by maintaining locality only in L2 when streaming the data). Despite the x vector L1 data locality, the L3 and DRAM rooflines only slightly differ to the ones in original CARM, which indicates that indirect accesses with higher latency to the cols vector (stored in L3/DRAM) diminish the potential performance benefits from x vector L1 locality.

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

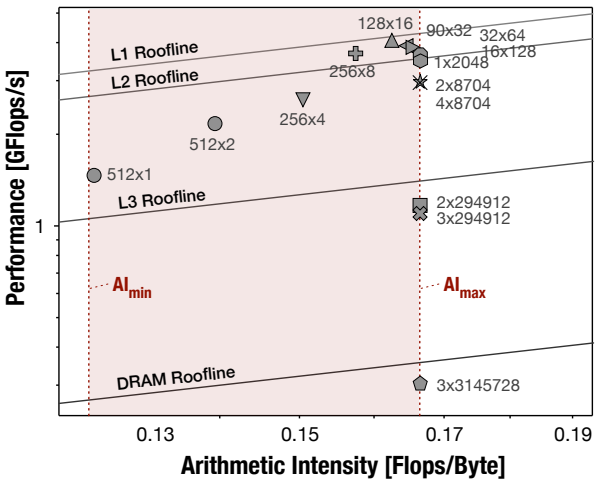


Figure 13 *AI variation with NNZ per row.*

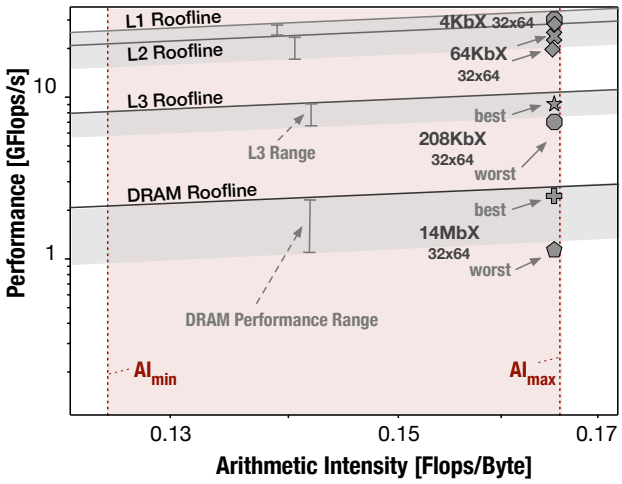


Figure 14 *Best-Worst cases in Sparse-CARM.*

Given the different micro-benchmarking and model construction principles, the proposed model also has fundamentally different interpretation methodology when compared to the original CARM. As shown in Fig. 12 (see gray dot with dashed arrow), in the original CARM the execution bottlenecks and optimization hints are derived by observing all the roofs intersected at the application AI, always suggesting the potential to exploit the maximum architecture performance (either corresponding to the L1 bandwidth or FP performance). In other words, the optimization strategy is based on surpassing all roofs positioned above the application point. Although this method might be adequate for some general-purpose kernels with working set potentially fitting into L1 cache, it is certainly not sufficient to provide in-depth characterization of the considered sparse kernel.

In contrast, the rooflines in the proposed model are representative of both micro-architecture and sparse application features, since they are built via bandwidth micro-benchmarking where all data structures are stored in the respective memory level and accessed in a sequential and coalesced manner. Hence, for a warm-cache scenario, the maximum attainable performance with a sparse matrix whose data structures only fit in a specific cache level cannot exceed the performance of the corresponding memory roofline in the sparse-aware CARM (i.e., the roofline immediately above the application point, as shown in Fig. 12). As such, the optimization path is restricted to matrix reordering, where row and column permutations may yield improved accesses and better reuse of x vector data, thus providing higher performance.

The vertically dotted lines and shaded region in sparse-aware CARM, shown in Fig. 12, represent the theoretical Arithmetic Intensity (AI) range of the x86 SpMV kernel. Figure 13 presents the experimental evaluation of this range by relying on a set of dense synthetic matrices with different dimensions. As elaborated before, the minimum AI is achieved with sparse matrices of 1 column (AI=0.125), then the AI shifts to the right as the Number of Non-Zeros (NNZ) per row increases, until reaching near-theoretical AI maximum with high column counts (AI≈0.16666).

Since reordering is the most important optimization approach to increase the SpMV performance (and potentially improve the x vector locality), we focus first on uncovering the realistically attainable ranges of performance improvements via reordering. For this, we create pairs of synthetic sparse matrices to mimic different execution scenarios: *i) worst case* matrix, which aims at minimizing the reuse of the x vector; and *ii) best case* matrix, which attempts to maximize the x locality. These matrices are obtained through a specific row and column permutation with dimensions and access patterns dictated by the cache specifics, and contain a set of diagonal

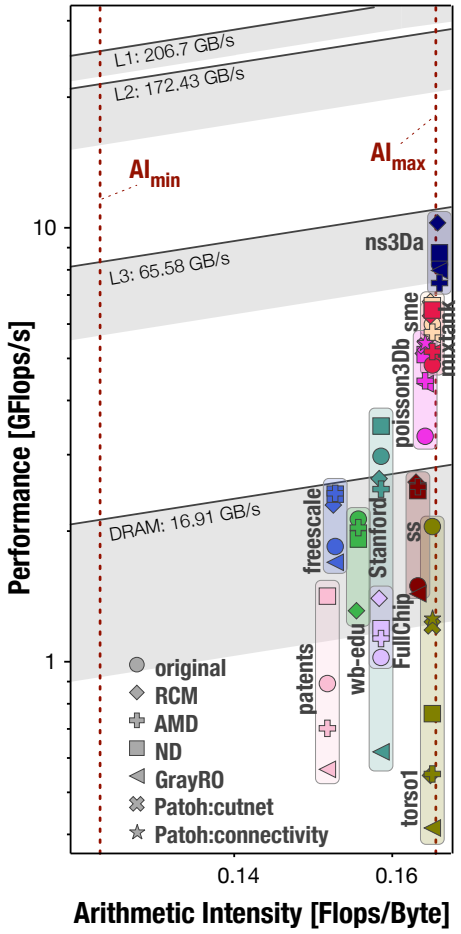


Figure 15 *Matrix reordering*

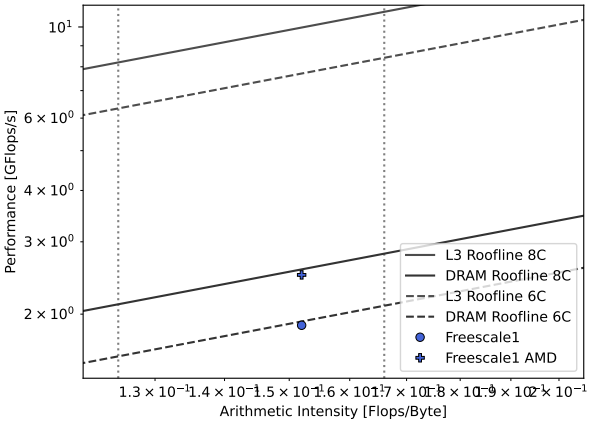


Figure 16 *Freescale in 6C/8C models.*

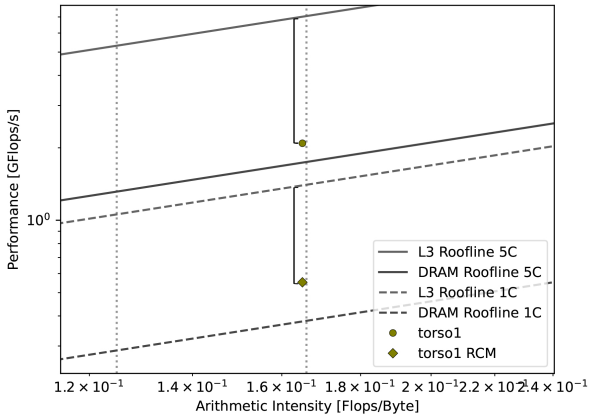


Figure 17 *Torso1 in 5C/1C models.*

dense blocks (each assigned to a specific core).[42] Fig. 14 presents the evaluation of several best and worst case matrices in the sparse-aware CARM. As it can be seen, performance improvements are achieved between the worst and the best cases for all memory levels, being the most notable for the groups of matrices that fit in DRAM (14MbX), L3 (208KbX) and L2 (64KbX), which may yield speedups of 2.13×, 1.3× and 1.23×, respectively.

Reordering real matrices: As presented in Fig. 15, we extend the experimental evaluation to a set of 11 real sparse matrices from Suite Sparse, to which up to six different reordering algorithms are applied, i.e., RCM, AMD, ND, cutnet and connectivity from Patoh library and GrayRO.[43] This set of matrices are real, general and non-complex, with diverse number of rows, columns and non-zero elements, covering a wide range of execution scenarios. As presented in Fig. 15, some reordering methods provide performance improvements for certain matrices (e.g., all for `poisson3Db`, all except GrayRO for `freescale`), but they might provoke performance degradation (e.g., all for `torso1` or RCM for `wb-edu`). This effect is not surprising since some reordering methods are not developed with data locality in mind, e.g., RCM is a fill reducing method.

Another noteworthy observation is that all considered matrices are placed within the modeled

[42] Afonso Silva Mendes Coutinho. "CARM-based approach for sparse computation characterisation". MA thesis. Instituto Superior Técnico, Universidade de Lisboa, 2022.

[43] Ibid.

AI range, but not all of them are positioned within the best-worst case performance ranges (see grey regions below the rooflines). This is mainly due to the impact of reordering algorithms to the load balancing in multi-threaded execution. For example, AMD improves the average core utilization for `freescale1` from 6.19 (original) to 7.98, while RCM provokes its reduction for `torso1` from 5.32 to 1.46. This explains their characterization in the sparse-aware CARM, since `freescale1`-AMD obtains a speedup of 1.32×, while `torso1`-RCM incurs a slowdown of 0.26×. However, unbalanced execution may also impact the quality of insights derived from the sparse-aware CARM, since the application points are analyzed against the roofs that are not representative of that execution scenario. For example, in Fig. 15, `torso1`-RCM execution is no longer strictly associated with the x vector memory accesses across all 8 cores in parallel.

Adapted sparse-aware CARM: To counter-balance this issue by retaining the cache locality focus of the sparse-aware CARM, its analysis is extended to consider different core utilizations. These variants of the proposed model are obtained by applying the previously elaborated micro-benchmarking methodology to different number of cores (lower core counts should deliver lower memory bandwidth). This adapted sparse-aware CARM allows to improve the model insightfulness and isolate the cache locality analysis, thus providing the means to characterize the performance variations due to changes in x accesses and minimize the impact of load balancing.

For example, in Fig. 15, the initial 8-core sparse-aware CARM characterization suggests that `freescale1`-AMD achieves significantly better DRAM bandwidth utilization and data locality for x vector accesses. However, as presented in Fig. 16, the characterization in adapted sparse-aware CARM (based on the average core utilization), reveals that both original and AMD `freescale1` matrices are placed in the same relative position regarding the DRAM roof. This suggests that there are no changes in the main execution bottlenecks after reordering, which fully corroborates with the conducted VTune Top-Down analysis.[44] Similar discrepancy in analysis can be observed in Fig. 17, where the adapted sparse-aware CARM is applied for characterization of `torso1` RCM, which showcases the locality improvements due to reduced relative distance to its respective roof.

Overall, the adapted sparse-aware CARM offers additional insights on where the application optimization should focus. For example, if a specific matrix is represented on top of a memory roof that corresponds to its average core utilization, the next optimization step should focus on improving the load balancing (if the core utilization is lower than the maximum). If the application attains good average core utilization, but it is positioned significantly below the corresponding roof, then the techniques to improve the accesses to the x vector can be applied. Finally, in the case of a kernel that is represented below the roof in a model that does not correspond to its maximum core utilization, further optimization can be focused on both accesses to the x vector and load balancing.

Optimization for energy efficiency: We also investigate applicability of the proposed sparse-aware CARM methodology to explore the optimization space for improving the SpMV energy efficiency. For this purpose, the realistic upper-bounds for performance, power consumption and energy-efficiency are obtained by running the previously introduced set of synthetic matrices with increasing NNZ per row to iterate over the complete AI range and for a range of core frequencies. To validate model usability, the previously tested `freescale1`-AMD and `ss`-RCM are selected because they attain near perfect load balancing, with core utilization close to the maximum, thus allowing to showcase the benefits of the proposed strategy by limiting the impact of other execution factors.

[44] Coutinho, "CARM-based approach for sparse computation characterisation".

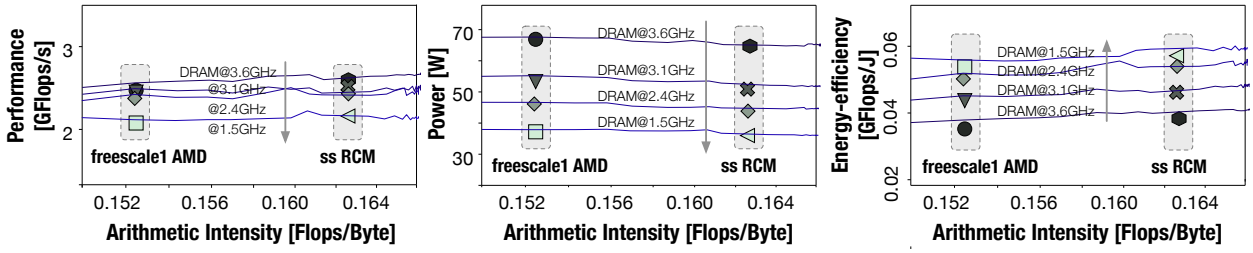


Figure 18 *SpMV performance, power and energy-efficiency variation with core frequency.*

Figure 18 presents the performance, power and energy efficiency analysis of these matrices for different core frequencies, by relying on the curves obtained for the memory levels which limit their performance (DRAM in this case). As it can be observed, the reordered matrices achieve close-to-modeled values in all three domain (see the difference between the point and the corresponding curve). Since the matrices are DRAM-bound, there is no significant variation in performance for different core frequencies (DRAM is clocked in a separate domain), while the minimum power consumption is obtained for the minimum frequency. As such, similar performance with reduced power results in the positive implications regarding energy efficiency, which is the highest for the minimum frequency.

3.2 A64FX CACHE PARTITIONING PROFILER

The sector cache is a feature found in the Fujitsu A64FX CPU, designed to enhance cache performance by allowing for hardware-supported partitioning of the L1 and L2 caches. This feature enables the CPU to divide the cache into sectors, each with its own set of cache lines, providing more control over data placement and cache utilization. By partitioning the cache into sectors, the sector cache feature allows for better management of data placement based on access patterns and memory requirements. This can help reduce cache pollution, improve cache hit rates, and optimize data access for specific applications or workloads.

In the context of SpMV on the A64FX CPU, the sector cache feature can be leveraged to improve performance by controlling the placement of data related to the computation, optimizing cache utilization, reduce cache misses, and enhance overall performance in SpMV operations with irregular and indirect memory access patterns.

We present a performance modelling tool[45] based on reuse analysis to describe cache behavior in SpMV. Our tool introduces a novel method for modelling cache behavior in SpMV on the Fujitsu A64FX CPU, considering the sparsity pattern and dimensions of the input matrix. By analyzing the impact of the sector cache on cache misses and performance under various configurations, we demonstrate the effectiveness of the sector cache in reducing cache pollution and improving performance for SpMV operations. Additionally, we highlight the significance of understanding cache behavior in optimizing performance for sparse matrix computations on modern architectures like the A64FX. The tool is available under `https://github.com/sparcityeu/spmvrd`.

Key findings include the classification of matrices based on dimensions to determine the benefits of using the sector cache, detailed measurements showing a correlation between reduced demand cache misses and speedup in SpMV, and the proposal of a method for accurate cache

[45]Sergej Breiter, James D Trotter, and Karl Fürlinger. "Modelling Data Locality of Sparse Matrix-Vector Multiplication on the A64FX". *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023, pp. 1334–1342; Sergej Breiter et al. "A Profiling-Based Approach to Cache Partitioning of Program Data". *International Conference on Parallel and Distributed Computing: Applications and Technologies*. Springer. 2022, pp. 453–463.

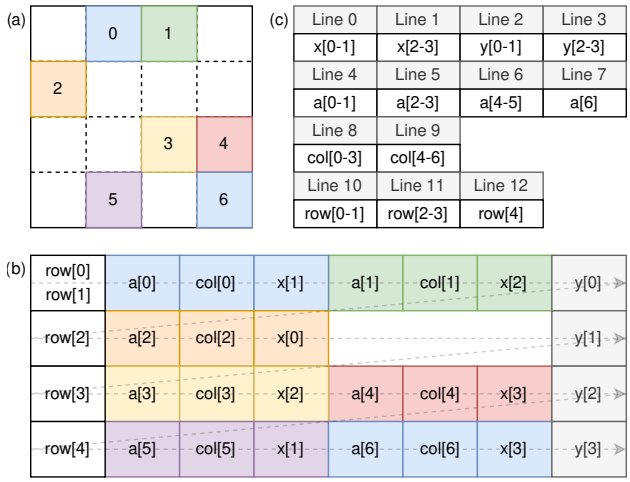


Figure 19 *(a) Sparse matrix pattern with 7 nonzeros (b) Access pattern of CSR SpMV (c) Cache memory layout of involved data structures* `x`*,* `y`*,* `a`*,* `colidx` *and* `rowptr` *assuming a cache line size of 16 bytes and alignment to cache line boundaries.*

miss predictions incorporating cache partitioning effects. We conclude that the sector cache can significantly enhance performance in SpMV on the A64FX processor.

Functionality The method proposed in the paper "Modelling Data Locality of Sparse Matrix-Vector Multiplication on the A64FX" to model cache behavior in sparse matrix-vector multiplication (SpMV) on the Fujitsu A64FX CPU is based on reuse analysis. This method aims to provide a comprehensive understanding of cache behavior by considering the sparsity pattern and dimensions of the input matrix.

The method utilizes reuse analysis to assess cache behavior in SpMV computations. By analyzing the reuse distance from the matrix sparsity pattern, the method can predict cache misses accurately, particularly focusing on the challenging aspect of estimating cache misses due to references to the x-vector, whose locality depends on the matrix sparsity pattern.

Fig. 19 shows an example of the approach described above. The access pattern of the SpMV with an example matrix (Fig. 19 (a)) is shown in Fig. 19 (b). Cache line numbers assigned to the elements of the data structures are shown in Fig. 19 (c). Each data structure is assumed to be aligned to a cache line boundary (i.e., 256 bytes for the A64FX). Finally, reuse distances can be computed from the resulting access pattern.

Accuracy The accuracy of the model in predicting cache behavior in SpMV on the A64FX CPU is a crucial aspect evaluated in the study. The model's accuracy is assessed based on its ability to predict cache misses resulting from irregular and indirect memory access patterns inherent in SpMV computations, particularly focusing on the challenging aspect of estimating cache misses due to references to the x-vector.

1. **Evaluation Metrics**: The accuracy of the model is typically evaluated using metrics such as Mean Absolute Percentage Error (MAPE) and Standard Deviation of the Absolute Percentage Error. These metrics provide insights into the deviation between the predicted cache misses and the actual cache misses observed during performance measurements.

2. **Comparison with Performance Events**: To validate the accuracy of the model, predictions of cache misses due to x-vector accesses are compared with performance event measurements.

This comparison helps in assessing the model's ability to predict cache behavior accurately and optimize cache utilization for improved performance in SpMV operations.

3. **Factors Influencing Accuracy**: The accuracy of the model may depend on various factors, including the complexity of the sparsity pattern, the dimensions of the input matrix, and the effectiveness of the sector cache configurations in reducing cache pollution. Understanding these factors is essential for evaluating the model's accuracy in predicting cache behavior.

4. **Error Analysis**: The model's accuracy is further analyzed by examining cases where the prediction error is higher, identifying factors that contribute to inaccuracies in cache miss predictions. By analyzing these errors, researchers can refine the model and improve its accuracy in predicting cache behavior for SpMV on the A64FX CPU.

We evaluated the accuracy on a set of 490 matrices from the suite sparse matrix collection with an average prediction error for L2 cache misses of < 3 % in sequential SpMV and < 4 % in parallel SpMV using 48 threads. These low error rates indicate that the model can effectively predict cache behavior in SpMV computations on the A64FX CPU.

3.3 SUPERTWIN

To our knowledge, there exists no work on using digital twins to model HPC systems; the literature can be investigated in three contexts; monitoring frameworks, profiling methods, and digital twin ontologies. To systematically collect and analyze information from performance metric sources, several frameworks have been developed, e.g., LDMS,[46] HPC-Toolkit,[47] Ganglia,[48] Nagios,[49] and PerfAugur.[50] E2EWatch[51] specializes in system-wide monitoring using Linux metrics and focuses on anomaly classification and detection. ClusterCockpit,[52] a more recent tool, reports performance metrics from distributed systems to InfluxDB and offers monitoring dashboards and job history queries. However, these tools have limitations, such as supporting only preselected, a fixed set of metrics and lacking a comprehensive knowledge representation and linked-data capabilities.

Performance Co-Pilot (PCP)[53] is a metric collection, transport, and storage tool that can be configured to sample every available metric counter on hardware and kernel, and energy usage of a system by Running Average Power Limit (RAPL)[54] and `perf` interfaces. It supports varying

[46] Brandt. *Lightweight Distributed Metric Service (LDMS): Run-time Resource Utilization Monitoring.* English. Tech. rep. SAND2013-6521C. Sandia National Lab. (SNL-CA), Livermore, CA (United States); Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), 2013. URL: https://www.osti.gov/biblio/1106397 (visited on 09/27/2021); Agelastos. *The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications.* English. Tech. rep. SAND2014-19868C. Sandia National Lab. (SNL-NM), Albuquerque, NM (United States); Sandia National Lab. (SNL-CA), Livermore, CA (United States), 2014. DOI: 10.1109/SC.2014.18. URL: https://www.osti.gov/biblio/1315267 (visited on 09/27/2021).

[47] Adhianto. "HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs Http://Hpctoolkit.Org". *Concurr. Comput.: Pract. Exper.* 22.6 (2010), pp. 685–701. ISSN: 1532-0626.

[48] Ganglia. *Monitoring system*. 2022. URL: http://ganglia.sourceforge.net/ (visited on 12/12/2022).

[49] Nagios. *Nagios*. https://www.nagios.org/. Accessed: 2022-12-12. 2022.

[50] Roy. "PerfAugur: Robust diagnostics for performance anomalies in cloud services". *2015 IEEE 31st International Conference on Data Engineering*. 2015, pp. 1167–1178. DOI: 10.1109/ICDE.2015.7113365.

[51] Aksar. "E2EWatch: An End-to-End Anomaly Diagnosis Framework for Production HPC Systems"". *Euro-Par 2021: Parallel Processing*. Springer International Publishing, 2021, pp. 70–85.

[52] *Cluster Cockpit*. https://www.clustercockpit.org/. Accessed on 30 Sep 2023.

[53] *Performance Co-Pilot*. https://pcp.io/. Accessed on 30 Sep 2023.

[54] Vincent M. Weaver et al. "Measuring Energy and Power with PAPI". *2012 41st International Conference on Parallel Processing Workshops*. 2012, pp. 262–268. DOI: 10.1109/ICPPW.2012.39.

sampling rates with negligible overhead, without the code compilation and instrumentation. SuperTwin leverages PCP to offer a robust and full-fledged analysis framework capable of enabling the creation of digital twins.

Digital twins for HPC systems differ from those for other physical entities due to the abundance of sensors, with each sensor, such as a hardware register or PMU, capable of reporting thousands of metrics through re-programming. Treating processes as unique components further adds to the heterogeneity within the HPC system. DTDL (Digital Twins Definition Language), a derivation of JSON-LD, consists of six metamodel classes that explain the context of digital twin components. These classes encompass *Interface*, *Telemetry*, *Properties*, *Commands*, *Relationship*, and various data schemes. In DTDL, each *Interface* represents a standalone (sub)twin, encompassing descriptions of its *Properties*, *Telemetry*, and *Relationships*. SuperTwin combines these components to hierarchically model an HPC system's structure, considering each component (e.g., node, socket, CPU, GPU, memory subsystem, etc.) as a distinct digital twin. The notion that each interface stands as an individual (sub)twin is a core principle extensively leveraged in SuperTwin.

The Roofline Model,[55] and its numerous variations,[56] including the Cache-Aware Roofline Model (CARM),[57] have emerged as invaluable tools to evaluate the computational capabilities of contemporary processors and pinpointing potential performance limitations.[58] SuperTwin incorporates CARM due to its ability to accurately characterize the entire system by considering all memory levels. However, the current literature primarily relies on a single tool, adCARM,[59] for CARM generation, which is tailored for Intel architectures, leaving a gap in support for AMD systems. In this work, an extension is introduced to support AMD systems under the SuperTwin framework. Furthermore, this work also addresses another gap in the area of Roofline modeling in general; real-time CARM visualization during execution. SuperTwin introduces the novel tool, the *live-CARM panel*, which takes performance-counter data and automatically calculates CARM-related metrics, displaying them in conjunction with other metrics to give users an immediate idea of how their application performs relative to architectural limits. This panel is a prime example of what can be achieved by leveraging all the capabilities of SuperTwin.

SuperTwin relies on a comprehensive knowledge base and linked-data capabilities. The Knowledge Base (KB), is used by each SuperTwin function as a parameter. It is dynamic, evolving to capture and link additional telemetry and metadata as they become available. This allows the twin to continue its operations in a live fashion without a procedural change and

[55] Nan Ding and Samuel Williams. "An Instruction Roofline Model for GPUs". *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2019, pp. 7–18. DOI: 10.1109/PMBS49563.2019.00007.

[56] Tuomas Koskela et al. "A novel multi-level integrated roofline model approach for performance characterization". *High Performance Computing: 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24-28, 2018, Proceedings 33*. Springer. 2018, pp. 226–245; Jee Whan Choi et al. "A roofline model of energy". *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, pp. 661–672; 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.

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

[58] Douglas Doerfler et al. "Applying the roofline performance model to the intel xeon phi knights landing processor". *High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^ 3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, 2016, Revised Selected Papers 31*. Springer. 2016, pp. 339–353; Didem Unat et al. "ExaSAT: An exascale co-design tool for performance modeling". *The International Journal of High Performance Computing Applications* 29.2 (2015), pp. 209–232. DOI: 10.1177/1094342014568690. URL: https://doi.org/10.1177/1094342014568690.

[59] Diogo Marques et al. "Application-driven cache-aware roofline model". *Future Generation Computer Systems* 107 (2020), pp. 257–273.

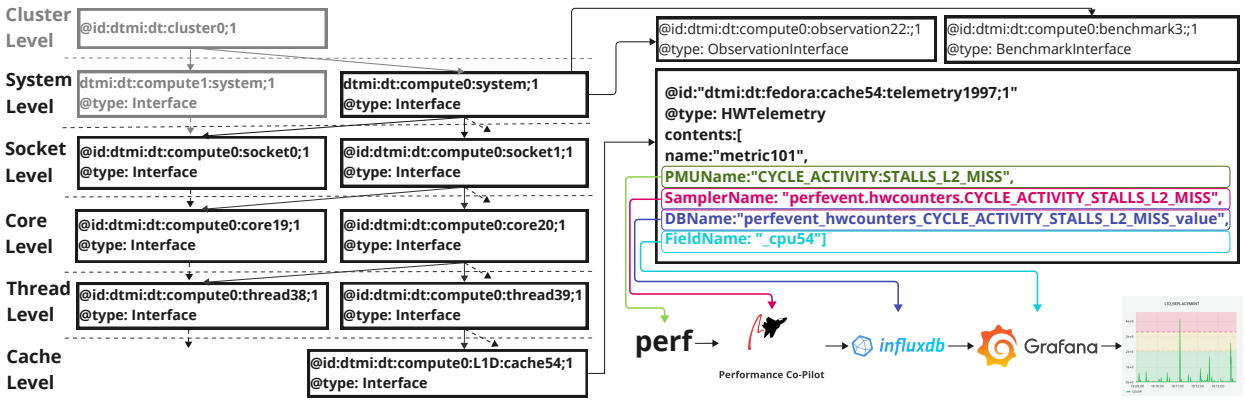


Figure 20 *Knowledge Base of* SuperTwin.

comprehend the factors influencing system performance in real time. An example KB is shown in Fig. 20.

The Knowledge Base: Capturing the target system and its component hierarchy, the KB can be parsed to acquire any information from topology to database parameters. There are two types of metrics to be sampled from an HPC system. The first type is *SWTelemetry*, i.e., software and system state-related metrics such as the number of processes, CPU, and memory load. These metrics are set to be *always sampled* with a low frequency. The second type is *HWTelemetry*, sampled from PMUs during kernel executions with high frequency. Sampling different metrics with varying frequencies yields a need for metadata associated with the host system's metadata. While time-series databases are tailored for telemetry data, they cannot keep much (linked) metadata. On the contrary, managing time-series data via a document database is impractical.[60] For this reason, SuperTwin's KB uses two types of databases with links between them. To this end, while InfluxDB stores the sampled SWTelemetry and HWTelemetry, MongoDB stores the knowledge base as JSON-LD extended with entries for each computation. To associate the computations with telemetry, pointers to InfluxDB are used to recall corresponding metrics.

Employing a tree-structured KB enables fully automated performance monitoring, anomaly detection and dashboards with meticulously selected metrics, tailoring various *views*. These views, namely (a) *Focus View*, (b) *Level View*, and (c) *Subtree View*, allow for a dynamic and versatile performance data exploration. Multiple views enable fine- and coarse-grain investigations into the component and system performance. Overall, SuperTwin can visualize data from different components and systems in tandem allowing for comprehensive analysis and comparison, further enriched by the inclusion of various views using Grafana visualization tool.

- **The focus** (i.e., component) view offers a dashboard that visualizes active metrics from a single component, e.g., a socket, core, thread, network, disk, or process, providing a focused lens on individual element performance. This view can be extended to focus on the path from the root (whole system) to the focused component to investigate the root cause of anomalous behaviors

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

(a) *Focus view for an individual cache*

(b) *Subtree view for a node*

Figure 21 *Focus- and subtree-view dashboards, automatically generated by* SuperTwin .

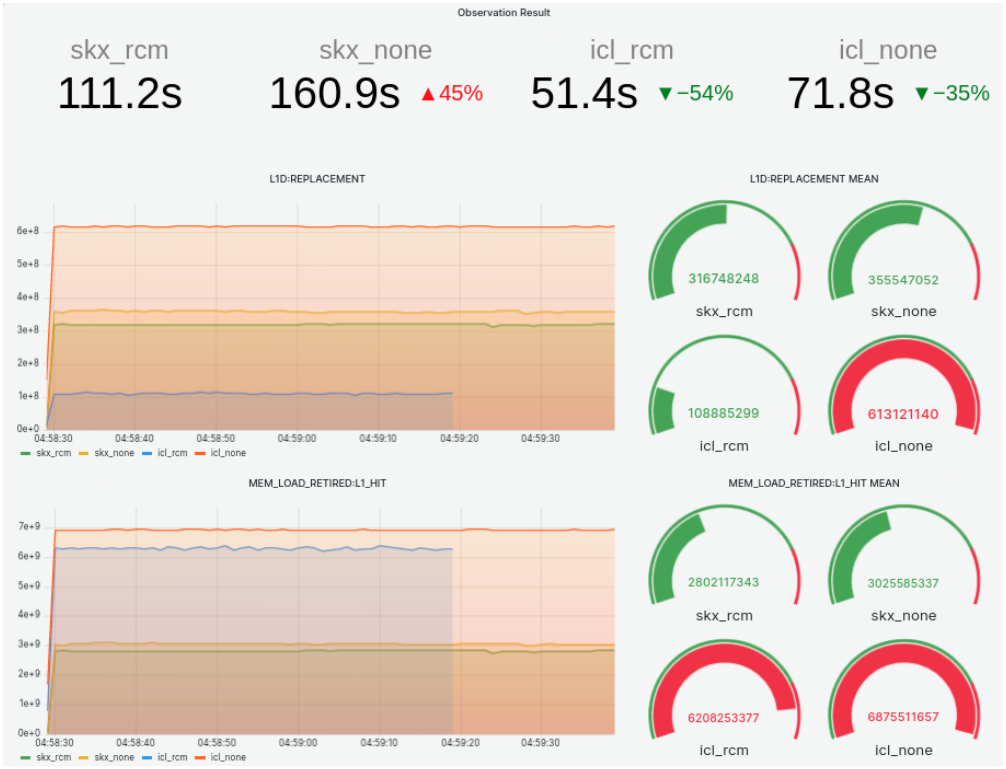


Figure 22 *Level-view for different matrix orderings.*

or performance drawbacks. That is the path navigating from a component perspective to a more generalized system perspective is analyzed, aiding in tracing and isolating performance issues. An example focus-view dashboard is given in Fig. 21(a) for an individual cache.

- **The subtree** (i.e., (sub)system) view seeks to *zoom* into performance events, starting from an arbitrary node and extending to all connected leaf nodes, moving from a general perspective to a more specific one, i.e., from a single socket to all cores/caches. The detail intensifies as the path moves from the root (subsystem) to the leaf (components at the bottom of the KB hierarchy), facilitating a deeper dive into specific performance events and data. An example subtree-view dashboard for a single server is given in Fig. 21(b).

- **The level** (i.e., type) view generates a dashboard that visualizes multiple instances from the same type, such as a group of threads, disks and even processes. This view allows the *isolation* of a single type, which corresponds to a level in the KB tree, treating them individually or in comparison to components within the same or a different system, whereas the linked-data capabilities enable the automatic visualization of component performance across different machines. For instance, the level-view dashboards for different processes running SpMV (each with a different reordering of the same matrix) is given in Fig. 22.

KB Lifecycle: The knowledge base is not a static object. It captures more about the system it represents as time passes by attaching new entries. To initialize the KB, SuperTwin uses its *probing tool*. To comprehensively capture the structural details of a system, including component specifications, inter/intra-relationships, and their associated performance metrics, a detailed probing is required. SuperTwin targets each hardware component that can be monitored, produce metrics or affect the overall system performance. Furthermore, it captures their relationships

in a lightweight and adaptable fashion. SuperTwin's probing relies on widely available Linux tools to gather data. The system, network, and memory information are collected via `lshw`. The CPU, memory/cache topology metadata are collected by parsing `likwid-topology` from `likwid` tools[61] and `cpuid` instruction. When available, disk info is probed from `/sys/block/*/device` and `SMART`[62] utility. PMU information is collected with `libpfm4` library, which can recognize model-specific registers and their events of virtually every x86 and ARM processor available on the market. Upon probing available PMU metrics via `libpfm4`, and software telemetry via PCP, are filtered and mapped with the components.

In the initial KB, every single component that performs computation, communication, or I/O is represented with an `Interface`. Furthermore, each relationship among these components is encoded into these interfaces with a `Relationship`. The available metrics for the components are filtered and encoded as SWTelemetry and HWTelemetry. This makes precisely pinned executions and automated queries possible. To keep the KB dynamic and continuously link the system components to performance data, SuperTwin uses `Interfaces` and attaches their instances (i.e., entries) to KB. For instance, as mentioned above, processes are monitored via per-process kernel metrics. JSON-LD interfaces are serialized with given parameters into a run-time object. Except for the `ProcessInterfaces`, all classes/interfaces have their values assigned as constants during the generation phase. In contrast, a `ProcessInterface` is re-instantiated each time it is invoked, reflecting the dynamic nature of processes. For performance events, SuperTwin has two other interface classes:

- `BenchmarkInterface`, and `BenchmarkResult` as a helper class, is designed to record benchmark results. SuperTwin is able to perform *Cache Aware Roofline Model* (CARM), *STREAM*[63] and *High Performance Conjugate Gradient*[64] (HPCG) benchmarks homogeneously using the `BenchmarkInterface`. It contains the source codes of these benchmarks in its codebase and similar to the probing phase, it first copies these codes to the target system. If required, based on the information in KB, SuperTwin first compiles the benchmarks on the target system using the benchmark's preferred compiler if it exists, e.g., `icc` or `gcc`. After the benchmark, SuperTwin parses the results and creates a `BenchmarkInterface` with the corresponding `BenchmarkResult`.

- `ObservationInterface` entries encode sampled hardware performance events and system metrics, executed commands, generated affinity, time and other relevant metadata. Using the parameters in KB, queries can be generated to automatically retrieve data through these entries. A basic `ObservationInterface` entry is shown in Listing 1. The queries automatically generated by SuperTwin to analyze the `BenchmarkEntry` in Listing 1 are given in Listing 2.

Performance DB: For long-term data management, thanks to its modular design, SuperTwin operates a *global performance database*, SuperDB. Unlike local instances, SuperDB employs cloud instances of MongoDB and InfluxDB. With a global performance database, SuperTwin aims to accumulate performance metrics from a wide array of systems to enhance architectural

[61] Thomas Röhl et al. "LIKWID Monitoring Stack: A Flexible Framework Enabling Job Specific Performance monitoring for the masses". *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 2017, pp. 781–784. DOI: 10.1109/CLUSTER.2017.115.

[62] The Smartmontools Team. *Smartmontools*. Accessed on 5th October 2023. URL: https://www.smartmontools.org/.

[63] John McCalpin. "Memory bandwidth and machine balance in high performance computers". *IEEE Technical Committee on Computer Architecture Newsletter* (1995), pp. 19–25.

[64] Jack Dongarra and Michael A Heroux. "Toward a new metric for ranking high performance computing systems". *Sandia Report, SAND2013-4744* 312 (2013), p. 150.

research and train robust machine learning models, particularly leveraging Large Language Models (LLMs) which can exploit the rich metadata collected to be trained as an assistant for performance engineering. The users of SuperTwin have the option to report their performance telemetry readings and the system's knowledge base to the performance database, alongside their local instances.

```
{
  "@type": "ObservationInterface",
  "@id": "278e26c2-3fd3-45e4-862b-5646dc9e7aa0",
  "displayName": "rcm_rma10_mt",
  "time": 48.667,
  "command": "./spmv -f rma10.mtx -o rcm -t 4",
  "modifier": "likwid-pin -q -c S0:0-1@S1:0-1",
  "no_threads": 4,
  "involved_threads": [0,1,22,23],
  "sampled_sw_metrics": ["kernel.percpu.cpu.idle", "mem.numa.alloc.hit", "mem.numa.alloc.
      miss"],
  "sampled_hw_metrics": ["RAPL_ENERGY_PKG", "INSTRUCTION_RETIRED", "FP_ARITH:
      SCALAR_DOUBLE", "MEM_LOAD_RETIRED:L1_HIT"],
  "dashboard": "http://localhost:3000/d/-PiOFZEVz/pmus-278e26c2-3fd3-45e4-862b-5646
      dc9e7aa0?time=1681499308500&time.window=17000"
}
```

Listing 1: An example `ObservationInterface` entry which is used to retrieve sampled metrics. A report is generated on the fly and added to the entry before appending to KB.

```
SELECT "_cpu0", "_cpu1", "_cpu22", "_cpu23" FROM "kernel_percpu_cpu_idle" WHERE tag="278
    e26c2-3fd3-45e4-862b-5646dc9e7aa0"
SELECT "_node0", "_node1" FROM "mem_numa_alloc_hit" WHERE tag="278e26c2-3fd3-45e4-862b
    -5646dc9e7aa0"
SELECT "_cpu0", "_cpu1", "_cpu22", "_cpu23" FROM "
    perfevent_hwcounters_fp_arith_scalar_double" WHERE tag="278e26c2-3fd3-45e4-862b-5646
    dc9e7aa0"
SELECT "_node0", "_node1" FROM "perfevent_hwcounters_RAPL_ENERGY_PKG" WHERE tag="278e26c2
    -3fd3-45e4-862b-5646dc9e7aa0"
```

Listing 2: Queries automatically generated by SuperTwin for the `BenchmarkInterface` entry given in Listing 1.

In SuperDB, the `ObservationInterface` of SuperTwin evolves into two versions within the performance database context: `TS ObservationInterface` and `AGGObservationInterface`, where the latter statistically summarizes data using various aggregations, e.g., min, max, mean, to manage high data volumes. The users require a local SuperTwin instance to access SuperDB, visualize performance data, and automatically generate dashboards and reports. Without SuperTwin, they can only download selected data for ML training. Future adaptations may include appending source code and binary executables to the collected metadata, facilitating the training of models that can optimize code and predict performance and potential inefficiencies.

Adding Compute Devices to SuperTwin: The integration of a computing device, i.e., FPGA, GPU, etc., into the KB is handled similarly to other hardware components within a system. Initially, an in-depth probing of the target devices is done using widely available tools. For instance, in the case of Nvidia GPUs, this investigation uses `nvidia-smi` to find available GPUs, their models, bus and process information. `/sys/class/drm/` is used for NUMA location, and `DeviceQuery` for the hardware specifications such as the number of SMs, shared memory, and cache sizes. The latest GPUs lack the capability for real-time hardware telemetry reporting without modifications to the source code. To address this, we have employed `pcp-pmda-nvidia`

for collecting SWTelemetry, essentially capturing every metric supported by NVML. Regarding HWTelemetry, we leveraged the approach used in benchmark executions. SuperTwin is tasked with creating a wrapper script for initiating the kernel launch and configuring ncu to record hardware performance events during runtime. Following the completion of these executions, SuperTwin analyzes the output from ncu, integrating these comprehensive performance metrics into the KB through the ObservationInterface. An example for (a subset of) an Interface encoding a GPU device in KB is given in Listing 3

```
"dtmi:dt:cn1:gpu0;1": {
    "@type": "Interface",
    "@id": "dtmi:dt:cn1:gpu0;1",
    "@context": "dtmi:dtdl:context;2",
    "contents": [
      {
        "@id": "dtmi:dt:cn1:gpu0:property0;1",
        "@type": "Property",
        "name": "model",
        "description": "NVIDIA Quadro GV100"
      },
      {
        "@id": "dtmi:dt:cn1:gpu0:property1;1",
        "@type": "Property",
        "name": "memory",
        "description": "34359 Mb"
      },
      {
        "@id": "dtmi:dt:cn1:gpu0:property12;1",
        "@type": "Property",
        "name": "numa node",
        "description": 0
      },
      {
        "@id": "dtmi:dt:cn1:gpu0:telemetry1337;1",
        "@type": "SWTelemetry",
        "name": "metric4",
        "SamplerName": "nvidia.memused",
        "DBName": "nvidia_memused",
        "fieldName": "_gpu0",
      },
      {
        "@id": "dtmi:dt:cn1:gpu0:telemetry1404;1",
        "@type": "HWTelemetry",
        "name": "metric137",
        "PMUName": "ncu",
        "SamplerName": "gpu__compute_memory_access
        _throughput",
        "DBName": "ncu_gpu__compute_memory_access
        _throughput",
        "FieldName": "_gpu0",
        "description": "Compute Memory Pipeline :
        throughput of internal activity within
        caches and DRAM",
      }}
```

Listing 3: An example GPU Interface entry which is used to monitor GPU devices on the system and profile kernel executions.

The Mechanics of SuperTwin: SuperTwin is designed to run on a *host* that can be different than the *target* system. The host runs the SuperTwin daemon as well as the tools with heavy workloads, e.g., InfluxDB, MongoDB, and Grafana. The target only runs the PCP samplers and reports telemetry to the host when requested. In Figure 23, step ⓪ reads the environment

variables such as the IP addresses of InfluxDB and MongoDB instances and Grafana token to the SuperTwin daemon. In step ①, the probing module is copied to the target system to generate a JSON file containing the system information which, in ②, is copied back to the host to generate the KB. The information collected from all the tools, components, and third-party tools SuperTwin manages is fused for KB generation. Once the KB is generated, it is inserted into MongoDB in step ③. Step ③ re-occurs every time KB changes or SuperTwin is restarted. When this phase is completed, the framework becomes fully functional using only this data structure.

In Figure 23, two SuperTwin scenarios are shown; the first is sampling software emitted metrics to monitor system state (Scenario A), and the other is capturing the hardware perfor-mainglunce events during kernel execution. In step (A1), using KB, SuperTwin configures the PCP collectors and samples system-related metrics, such as CPU and memory usage, NUMA-related events, and energy spent. In (A3), a sampler on the target is requested for this telemetry. Since the query parameters are already encoded in KB, steps (A1) and (A2) can happen at the same time. That is the dashboards are already generated on the host when the target starts reporting.

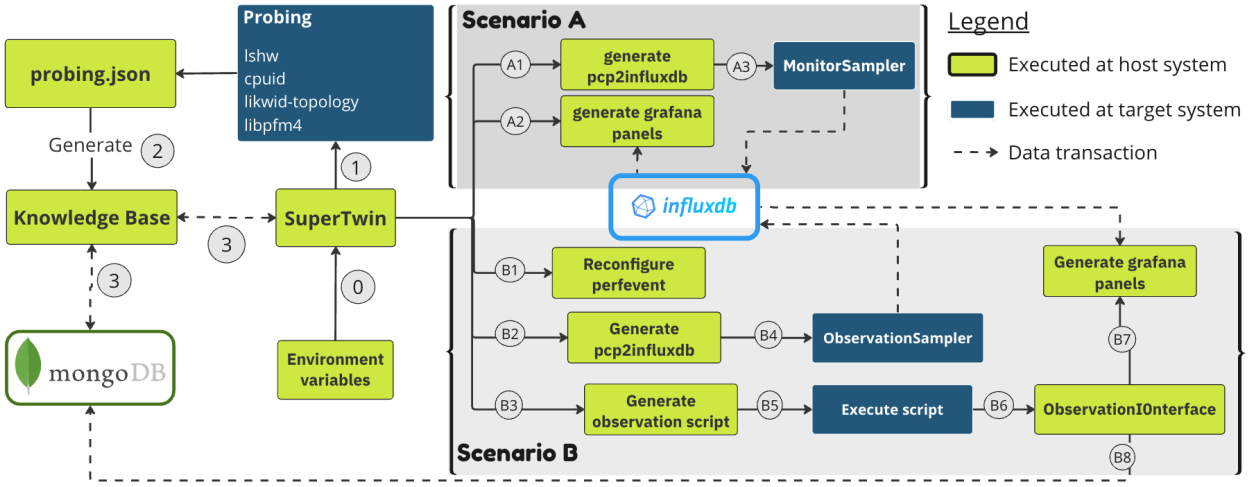


Figure 23 *Two scenarios within the* SuperTwin *framework*

In Scenario B, SuperTwin samples hardware events reported from the PMUs. In this case, it focuses on an execution on the target and the components on which the execution takes place. Therefore, SuperTwin requests an executable and its command-line parameters. Once these are provided, the PMUs are configured to report the requested metrics in step (B1). That is SuperTwin configures the sampler in the same way as step (A1). After the PMUs are configured, it generates a script to run the requested kernel on the target system. This script bounds the threads to the cores using one of the *balanced*, *compact*, *numa balanced*, *numa compact* strategies based on the probed target system topology. Then it samples performance events, executes the script to run a kernel on a target and stops the sampling as the kernel is halted. An `ObservationInterface` is generated to encode the execution metadata, collected metrics and the unique observation ID associated with the time-series data in InfluxDB. In step (B8), the `ObservationInterface` is appended to the system's KB. This `ObservationInterface` entry is later used to recall the performance data for visualization or analysis purposes.

Abstraction Layer: To perform its actions and to effectively monitor PMU events on diverse target systems, each hosting CPUs across various vendors and micro-architectures, SuperTwin leverages an *Abstraction Layer*. The monitoring units and their reported events can significantly

vary among different micro-architectures and from vendor to vendor. For instance, Intel has four general-purpose programmable counters/per-core to count performance events (eight if is not shared with a second thread in the core), whereas AMD has two internal counters, one for each sampling flag. Intel provides 62 sub-events corresponding to 12 events, each accompanied by mask values. Similarly, AMD offers support for events similar to Intel. As an example, similarities and differences of events for Intel Cascade and AMD Zen3 are listed in Table 8. A detailed comparison between Intel and AMD PMUs can be found in.[65]

Event	Intel Cascade	AMD Zen3
Energy	RAPL_ENERGY_PKG RAPL_ENERGY_DRAM	RAPL_ENERGY_PKG RAPL_ENERGY_DRAM
Retired Inst.	INSTRUCTIONS_RETIRED	RETIRED_INSTRUCTIONS
Tot. Mem. Op.	MEM_INST_RETIRED:ALL_LOADS + MEM_INST_RETIRED:ALL_STORES	LS_DISPATCH:STORE_DISPATCH+ LS_DISPATCH:LD_DISPATCH
L3 Hit	**Not Supported**	LONGEST_LAT_CACHE:MISS + LONGEST_LAT_CACHE:RETIRED

Table 8 *Intel vs. AMD PMU events: the same, similar, different, and exclusive event names for the same generic event, respectively.*

To facilitate the monitoring of PMU events in a platform-agnostic manner, an abstraction layer is implemented for SuperTwin. This layer effectively maps generic event names to concealed hardware-specific PMU event names, enhancing the system's versatility and ease of use. We have established a set of common events, such as L1_CACHE_DATA_MISS, FP_DIV_RETIRED, and RAPL_ENERGY_ PKG, that are *assumed to be* supported by all the commodity CPUs. The rest of the events are left to the user's discretion. For further flexibility and scalability, SuperTwin utilizes configuration files to establish a straightforward mapping of common events to corresponding hardware events. The structure of a configuration file is as follows:

```
[pmu_name | alias]
<generic_event>:<hardware_event_1> [op]
[op] : ((+|-|*|/)(<hw_event> | <const>)) [op]
```

Following the pattern delineated, it is possible to generate a configuration file for "any" hardware by specifying the events intended for monitoring. Upon registering the desired configuration files within SuperTwin, the application proceeds to configure the PCP of the target system using the registered configuration files when needed. Additionally, users can access event information in a CPU agnostic manner within the program using `pmu_util.get(...)` method. An example is given below;

```
>pmu_utils.get(HW_PMU_NAME, COMMON_EVENT_NAME)
>pmu_utils.get("skl", "TOTAL_MEMORY_OPERATIONS")
>[
  "MEM_INST_RETIRED:ALL_LOADS",
  "+",
  "MEM_INST_RETIRED:ALL_STORES"
 ]
```

[65] Muhammad Aditya Sasongko et al. "Precise Event Sampling on AMD Versus Intel: Quantitative and Qualitative Comparison". *IEEE Transactions on Parallel and Distributed Systems* 34.5 (2023), pp. 1594–1608. ISSN: 1558-2183. DOI: 10.1109/TPDS.2023.3257105.

Although this example belongs to the Intel CPU outlined in Table 9, SuperTwin's configuration mapping via its abstraction layer offers versatility. Users can create mapping files for a wide range of CPUs, including Intel, AMD, PowerPC, ARM, and others, as long as they are supported by the *libpfm4* library which is the core library that enables PCP to monitor PMU events in CPUs. As SuperTwin configures PCP on the target, it creates empty and *zero-overhead* dashboards on Grafana, which are simply JSON files. Last, but not least, the abstraction layer seamlessly generates the formulas for the events the user is interested in. This changes from vendor to vendor as well as for every architecture even when the events are the same. An abstraction layer is necessary in modern tools to handle this diversity for performance profiling.

Cache-aware Roofline Model in SuperTwin: For an intuitive visualization framework, SuperTwin supports the construction of a tailored CARM model for Intel and AMD microarchitectures. It is enriched with a set of custom micro-benchmarks in `x86` assembly, designed to experimentally assess the realistically attainable maximum performance of a given system, i.e., the sustainable bandwidth for different levels of memory hierarchy and the peak throughput of computational units. In order to assess the different metrics necessary to construct the CARM roofs, such as bandwidth and peak flops, we rely on the Time Stamp Counter (TSC) to measure the number of clock cycles elapsed, detection of CPU operating frequency, and predefined amount of memory and compute operations contained in a specific microbenchmark executed. The microbenchmarks support various instruction set architecture (ISA) extensions, including scalar, SSE, AVX2 and AVX512, along with multithreaded measurements. This allows for further customization of SuperTwin's CARM plot based on the prevalent ISA extension or a specific thread count utilized in the tested applications.

Thanks to Knowledge Base, CARM microbenchmarks are automatically configured for a target system, taking into account cache sizes and available ISAs. To reduce the overheads associated with extensive benchmarking of all possible combinations of thread counts, SuperTwin generates a subset of the most representative thread counts for the microbenchmark executions. Finally, the KB is also used to store all the microbenchmarking results for each tested system, thus allowing for a re-construction of the CARM plot without the need to re-run all the microbenchmarks.

Besides the construction of a CARM plot for a target system, SuperTwin also provides the CARM-based visualization of the application execution progress at run-time (live monitoring feature). This functionality is achieved by automatically configuring PMU events based on the underlying architecture of a system, in order to accurately calculate the live Arithmetic Intensity (AI) and live-GFLOPS of the system. These PMU-based metrics are sampled on a time-stamp basis and used to plot the application points in real time on the generated CARM for the target system. This generated panel is referred to in the framework as the live-CARM panel, which offers a unique feature of SuperTwin by delivering real-time feedback on a target system's utilization relative to architectural constraints determined by the already constructed CARM. This dynamic functionality is achieved through the formulation of specialized expressions based on hardware events, enabling the calculation of GFLOPS and Arithmetic Intensity (AI) tailored to diverse Intel and AMD microarchitectures.

The amount of GFLOPS is determined by mapping and adding all of the available floating-point operation events of the target system, using the PMU remapping capabilities of SuperTwin. As for the AI, this metric requires the already calculated GFLOPS, as well as the total amount of memory bytes transferred to/from the processing cores, which calculation varies across different generations of Intel and AMD systems. In general, they are inferred from the ratios of different floating point instructions (scalar, SSE, AVX2, AVX512), which are applied to the total amount of

store and load events measured in the target system.

The live-CARM panel also automatically retrieves the micro-benchmarking results (to construct the CARM plot of the target system) from the Knowledge Base. By tightly coupling the application's live metrics with the CARM plot in the SuperTwin panel, we facilitate the observation of the relative performance of an application in real-time, when compared to the theoretical limits of the architecture it is running on. Furthermore, SuperTwin auto-generates graphs for any hardware metric configured by the user, which display the values of selected metrics for the different cores of the target machine, including a cumulative sum of the events across all cores.

Experimental Results: In the host system, we used Grafana v9.4.7, InfluxDB 1.8, MongoDB 6.0.6. For micro-benchmarks, we used `likwid-bench` v5.2.2. Specifications of the target systems used in the experimental setting are presented in Table 9.

CSL		*ZEN3*	
OS	CentOS Linux release 7.9.2009 (Core) x86_64	OS	Ubuntu 22.04.3 LTS x86_64
Kernel	3.10.0-1160.90.1.el7.x86_64	Kernel	6.2.0-33-generic
CPU	Intel Xeon Gold 6258R @2.7GHz (28c/56t)	CPU	AMD EPYC 7313 @3GHz (16c/32t)
Arch	Cascade Lake	Arch	Zen3
Mem	64GB DDR4 @ 3200 MHz	Mem	128GB DDR4 @ 2933 MHz
Env.	pcp 6.1.0-1	Env.	pcp 5.3.6-1

Table 9 *Specifications of platforms used in the experiments.*

The PCP agents include `pmcd`, which manages other agents and reports their readings; `perfevent`, which samples PMU readings via Linux `perf` interface; `pmdalinux`, reporting software-sourced system state metrics like memory usage; and `pmdaproc`, which reports per-process metrics like I/O and memory usage. CPU usage measurements use the `proc.psinfo.utime` and `proc.psinfo.stime`, whereas memory measurements use the `proc.psinfo.rss` metric. Notably, regardless of the reported metrics or sampling frequency, all agents maintain constant memory usage. `pmdaproc` uses more memory due to a larger instance domain. Except for `pmdaproc`, all agents are efficient in resource usage. Overall, SuperTwin employs 0 per-process metrics and uses approximately 20 `pmdalinux` metrics, and 2 `pmdaperfevent` metrics at 1-second intervals.

During the hardware performance event samplings, both PCP run on the target system and performance monitoring registers are sampled. Therefore kernel run-time may be affected negatively. To measure the effect of sampling on a target system, we ran the same micro-benchmarks from previous tests with and without sampling and measured the change in their completion times. The overhead caused by sampling can be seen in Figure 24. Surprisingly, negative overheads are observed, which we explain as overhead added by sampling is smaller than the variance observed between different runs of the same kernel. This is understandable since the positive overheads are also measured at 0.01%. A similar negative overhead is also reported by[66] even in a much bigger distributed setting. However, a meaningful skew towards positive overhead is observed with increasing frequency.

Monitoring Live Performance Events: To showcase the live monitoring capabilities of SuperTwin, we execute two state-of-the-art algorithms for Sparse Matrix Vector Multiplication (SpMV),

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

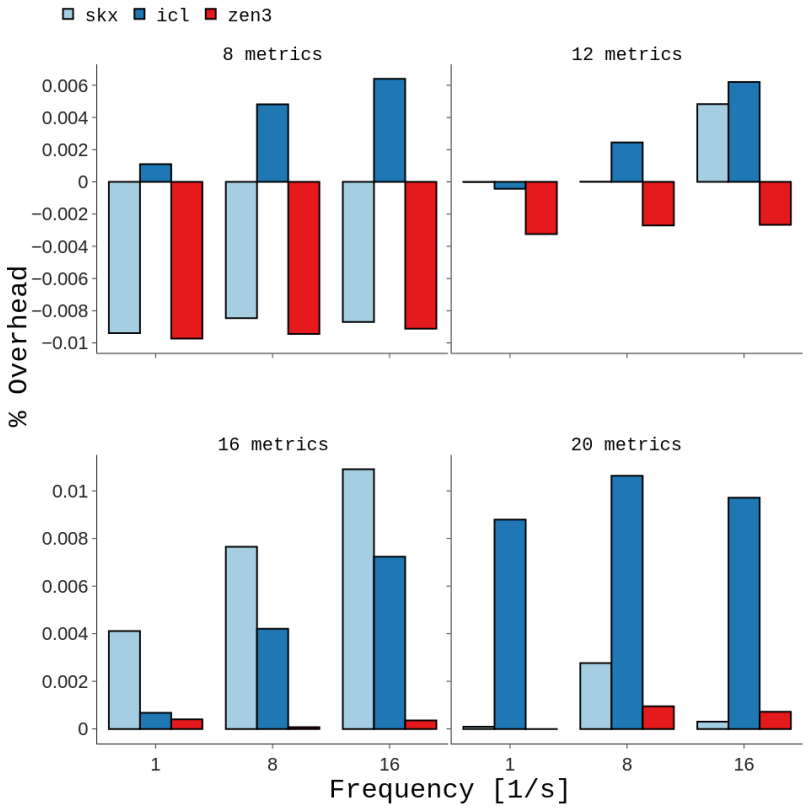


Figure 24 *Overhead caused by profiling six* `likwid-bench` *kernels (executions repeated 5 times, the run-times averaged).*

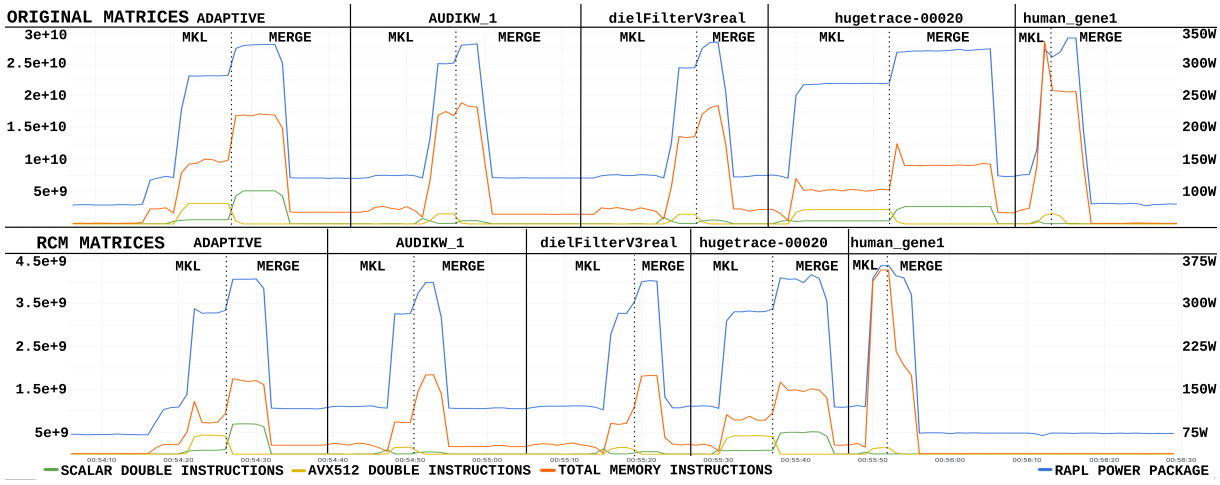


Figure 25 *Monitoring live performance events during SpMV execution on Intel CSL system*

i.e., Intel MKL[67] and Merge,[68] on the Intel CSL system presented in Table 9. We selected five

[67] Endong Wang et al. "Intel Math Kernel Library". *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*. Springer International Publishing, 2014, pp. 167–188. DOI: 10.1007/978-3-319-06486-4_7. URL: https://doi.org/10.1007/978-3-319-06486-4_7.

[68] Duane Merrill and Michael Garland. "Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format". *Acm Sigplan Notices* 51.8 (2016), pp. 1–2.

Name	Group	Rows	Cols	Nnz
adaptive	DIMACS10	6,815,744	6,815,744	27,2M
audikw_1	GHS_psdef	943,695	943,695	77,7M
dielFilterV3real	Dziekonski	1,102,824	1,102,824	89,3M
hugetrace-00020	DIMACS10	16,002,413	16,002,413	48,0M
human_gene1	Belcastro	22,283	22,283	24,7M

Table 10 *Sparse matrices used in the experiment.*

sparse matrices from the SuiteSparse collection,[69] as presented in Table 10, which cover a range of matrices from different scientific domains, characteristics, dimensions, and number of non-zero elements. Both SpMV algorithms are performed on the original (unaltered) matrices, as well as on their reordered versions using Reverse Cuthill-McKee (RCM).[70] For each combination of the sparse matrices, algorithms and reordering, the performance data is collected at runtime.

The obtained results are presented in Figure 25, when running the original (top part) and RCM-reordered (bottom part) matrices, and by subjecting each sparse matrix to the Intel MKL, followed by the Merge SpMV algorithm. For all cases, a set of PMU events were collected, these include `SCALAR_DOUBLE_INSTRUCTIONS`, `AVX512_ DOUBLE_INSTR.`, `TOTAL_MEMORY_INSTR.`, and `RAPL_POWER_PACKAGE`, with their evolution during the algorithm execution is depicted in Figure 25. As can be observed, there is a noticeable difference in the overall execution time required to process all five original (top) and reordered (bottom) matrices, where the reordered ones took about 22% less time for processing. This effect indicates the positive influence of reordering on improved data locality, which subsequently results in substantial performance improvements.

By focusing on the evolution of collected PMU events presented in Figure 25, one can observe that the `AVX512_DP_FP` events are only manifested during the Intel MKL execution, while the `SCALAR_DP_ FP` appear during the Merge algorithm runs. This is due to the ability of MKL SpMV implementation to take advantage of the Intel CPU's AVX512 capabilities, while Merge SpMV only exercised the scalar units (note the drop in AVX512 and the increase in scalar FP instructions at the vertical dashed lines, i.e., the points in time when MKL finishes and Merge starts its execution). We can also observe that during the MKL execution, the measures for `RAPL_POWER_PACKAGE` and `TOTAL_MEMORY_INSTRUCTIONS` are lower than for Merge. This corroborates the fact that the codes using higher SIMD ISA may provoke reduced instruction counts when compared to their scalar counterparts (e.g., AVX512 load/store instructions involve 64-byte data transfer versus scalar memory instructions that operate on 8 bytes of data). This phenomenon, as well as data locality in different memory levels achieved with different algorithms and reordering, can provoke significant power consumption variations, as shown in Fig. 25.

Monitoring Live CARM: To showcase the live-CARM feature in SuperTwin, we further analyze the performance differences between MKL and Merge SpMV algorithms, as well as three *likwid* benchmarks on the Intel CSL system (see Table 9).

- SpMV Execution Profiling: Figure 26 presents the live-CARM panel during the execution of both Intel MKL SpMV and Merge SpMV for the `hugetrace-00020` (see Table 10) in its original and RCM-reordered form. The live-CARM timestamps belonging to each execution phase are identified by the colored square that contains them, namely: *pink square* – Intel MKL; and *orange square* – Merge execution, while for both algorithms the *blue* and *green*

[69] Tim Davis. *Sparse Matrix Collection*. Accessed on 5th October 2023. URL: https://sparse.tamu.edu/.

[70] Elizabeth Cuthill and James McKee. "Reducing the bandwidth of sparse symmetric matrices". *Proceedings of the 1969 24th national conference*. 1969, pp. 157–172.

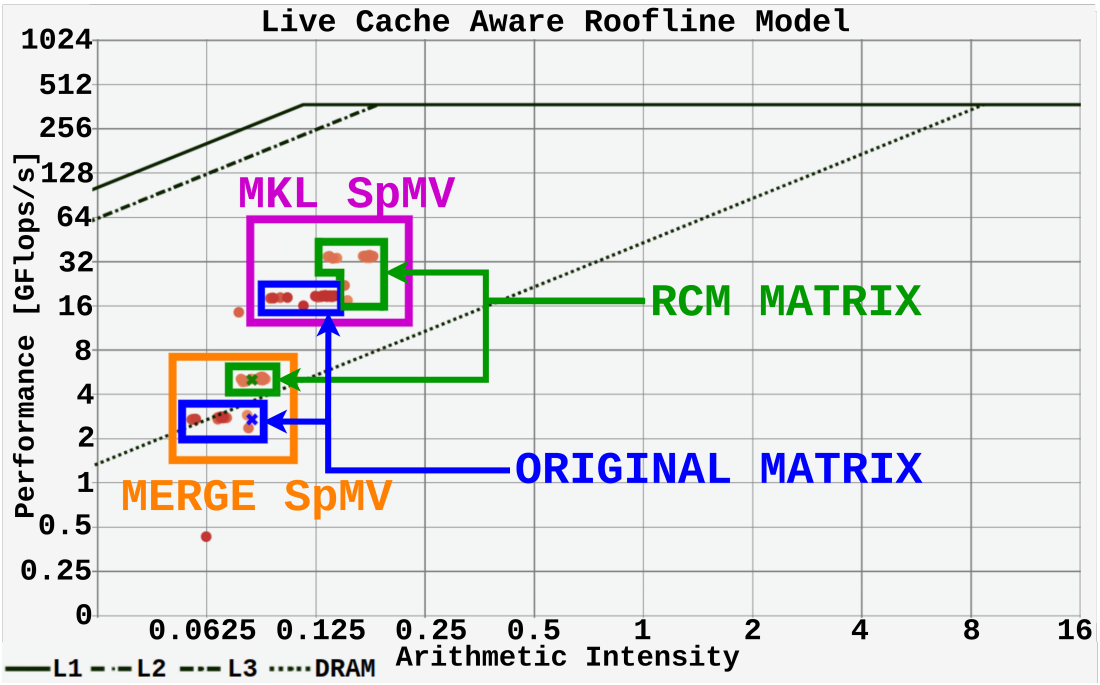


Figure 26 *Live-CARM during SpMV execution*

squares denote the executions corresponding to the original and RCM-reordered matrix, respectively. As can be observed in the CARM plot, for each algorithm, the RCM reordering yielded higher performance, while we can also observe that the Intel MKL SpMV provides higher performance than the Merge SpMV (mainly due to its ability to exploit AVX512 SIMD capability). Furthermore, this study showcases how the Live-CARM dashboard can be used to make intuitive and insightful performance analyses across different applications and their execution phases during the run-time, as it allows pinpointing the data locality in different memory levels.

- Benchmark Execution Profiling: Live-CARM can also be used to profile benchmarks, by directly comparing the execution of a benchmark against the live-CARM roofs, i.e., the performance upper-bounds attainable on a target platform for different memory levels and compute units. This analysis provides a general idea on the ability of executed applications to fully exploit the capabilities of underlying hardware resources. For this purpose, various benchmarks from the *likwid* tool[71] (Triad, PeakFlops, and DDOT) were considered, with corresponding live-CARM reports presented in Figure 27.

  The Triad benchmark (see orange points enclosed with green box) is a memory-bound benchmark with a theoretical AI of 0.625, which is accurately captured by the live-CARM in Fig. 27. As can be seen, the performance of this kernel approaches the L2 roof, but it is unable to surpass it since the workload size does not fit in the 32Kb L1 cache. The PeakGflops benchmark (red dots enclosed with the dark blue box) is designed to reach the peak FP performance. With a theoretical AI of 2, this benchmark reports a performance very close to the one obtained with the CARM microbenchmarks (the application points aligned with the horizontal live-CARM roof in Fig. 27). Finally, similarly to Triad, the

[71] Thomas Röhl et al. "Overhead Analysis of Performance Counter Measurements". *2014 43rd International Conference on Parallel Processing Workshops*. 2014, pp. 176–185. DOI: 10.1109/ICPPW.2014.34.

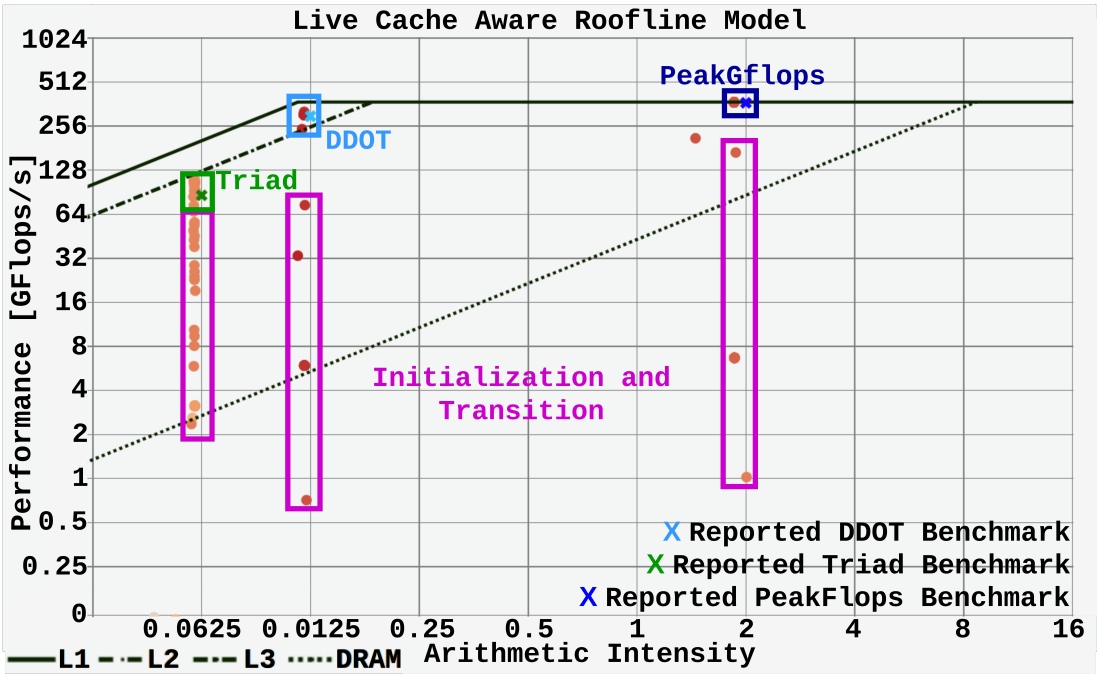


Figure 27 *Live-CARM during Likwid benchmarks execution*

DDOT benchmark, is a memory-bound kernel that utilizes smaller problem sizes, thus able to fit in the L1 cache. As presented in Fig. 27 (see red dots with a light blue box), the theoretical DDOT AI of 0.125 is accurately captured by the live-CARM, with the performance surpassing the L2 roof, and approaching the maximum performance of the architecture.

3.4 SPARSEVIZ

SparseViz is a low-code library designed to enhance the visualization of sparse matrices and tensors without the need for direct coding. It provides a user-friendly inter- face and tools that allow researchers and developers to easily explore sophisticated sparse data structures and identify the reasons for performance bottlenecks. Combined, SparseViz and SparseBase offer a comprehensive ecosystem for handling sparse data, emphasizing ease of use and integration with existing computational frameworks. In this way, advanced data processing techniques become available to a wider audience.

SparseViz is built upon the principle of simplifying the interaction with sparse data structures. It offers a suite of functionalities that cater to the needs for understanding the performance of a function running on sparse data. It not only facilitates the efficient representation and manipulation of sparse data but also extends its capabilities to various operations such as implementing orderings, executing kernels, and visualizing them. It has been designed with extensibility in mind, allowing for continuous development and integration of new features, i.e., orderings and kernels.

Key Features: SparseViz's capabilities are designed to address various challenges and needs encountered in working with sparse data. Here are some of its key features:

- **Visualization Tools:** The main functionality of SparseViz is its capability to visualize sparse data structures. With this one can understand complex sparsity patterns and see their impact on performance.

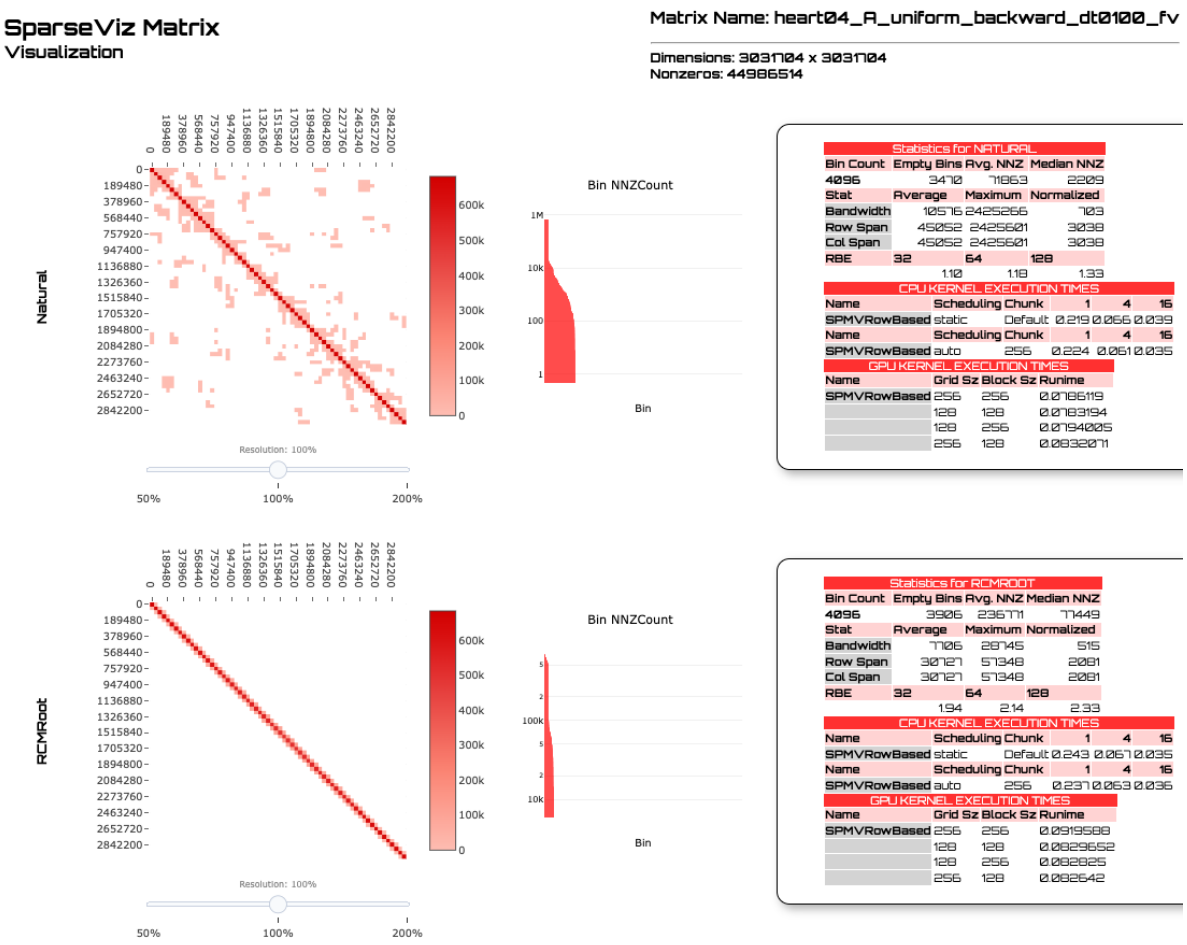
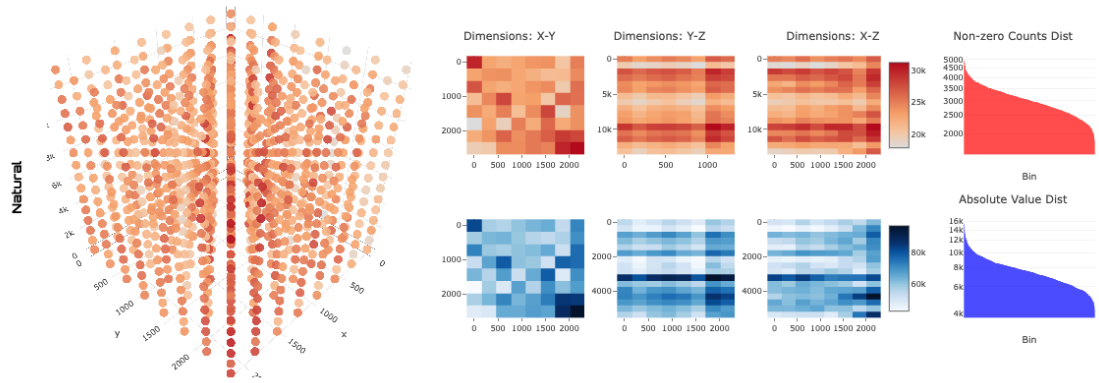


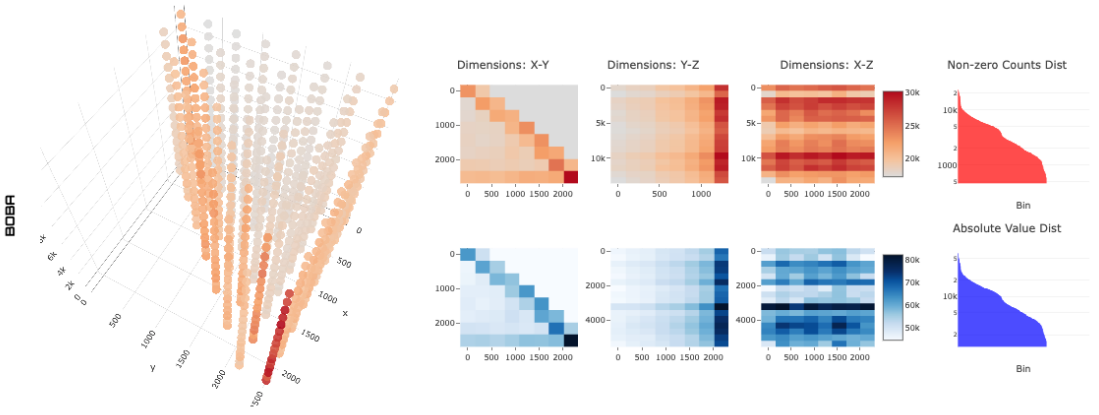
Figure 28 *A cardiac simulation matrix in the Natural and RCM order.*

- **Representation of Sparse Data Structures:** SparseViz excels in the efficient handling and representation of sparse matrices and tensors, making it easier to work with large, sparse datasets.
- **Flexible Ordering Systems:** SparseViz allows for customizable ordering of sparse data structures. This feature is particularly useful for optimizing data (to increase cache-hit ratio) for specific algorithms or processing techniques.
- **Automated Kernel Execution:** With SparseViz, users can execute their customized kernels on sparse data. This functionality is essential for understanding the efficiency of the implemented orderings in practical terms.
- **Efficient Storage Solutions:** The library offers optimized storage solutions for sparse data. This ensures that large datasets are not only stored efficiently but are also easily retrievable for future use.

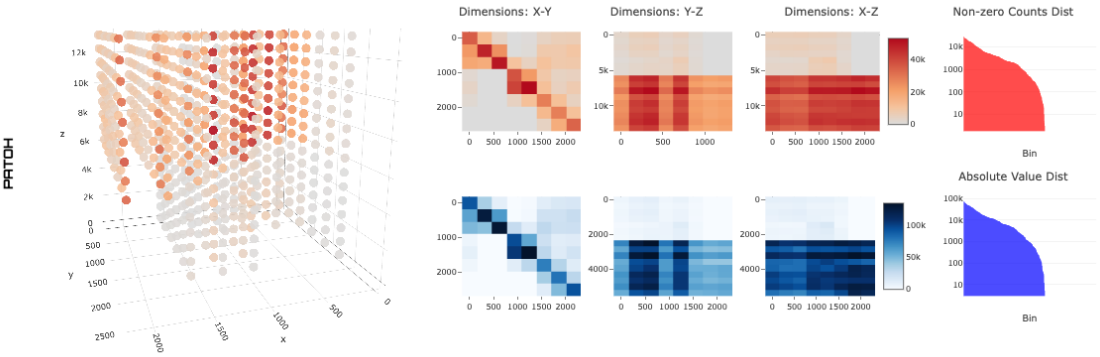
Figure 28 shows a matrix coming from one of our use cases, Cardiac Simulation. With SparseViz, one only needs to enter the matrix file location, the orderings s/he wants (Natural and RCM in the figure) and the CPU/GPU kernel names s/he wants to execute on the ordered matrices (CPU/GPU SpMV with different thread counts and size of Block/Grid). Metrics that



(a) *Tensor in Natural order.*



(b) *Tensor in RCM order.*



(c) *Tensor in PaToH order.*

Figure 29 *The first three modes of the NIPS tensor are ordered with different algorithms.*

define the compactness of the sparsity structure obtained after the orderings, such as bandwidth, average row/column nonzero span, number of empty bins etc., are also reported along with the kernel execution times. SparseViz is also able to report the values collected from HW counters obtained via `perf`.

In addition to matrices, SparseViz can also work with tensors. Figure 29 shows the visualiza-

tion of the NIPS tensor with three different orderings; Natural (a), RCM (b), and PaToH (c). This tensor corresponds to the papers published in the NIPS conference from 1987 to 2003. The modes represent paper-author-word-year with dimensions $2,482 \times 2,862 \times 14,036 \times 17$ and the values are counts of words. In this visualization, we have extracted the first three modes. The 3D scatter plots on the left show the 3D view of the nonzeros organized in multiple 128×128 bins. The rightmost charts show the nonzero distribution among these bins. The middle heatmaps show the aggregated nonzero distribution when the tensor is viewed from XY, XZ, and YZ dimensions.

3.5 SPARSE MATRIX/TENSOR GENERATOR

A smart sparse matrix and tensor generator is developed that mimics real sparse matrices and tensors. The aim is to obtain a large number of sparse matrices and tensors to study ML models, increase the size of the open datasets, and quickly evaluate proposed methods and algorithms without storing the matrix or tensor. The generator imitates real matrices and tensors using their substantial features.

A strong advantage of the matrix and tensor generator is that it utilizes size-independent features that can be used easily to generate instances with different sizes, such as coefficient of variation, imbalance, and density. For matrices, more matrix-related features are also included as options for the user, such as bandwidth, profile, and symmetricity.

The sparse matrix generator (genMat) and the tensor generator (genTen) are available at https://github.com/sparcityeu/genMat and https://github.com/sparcityeu/genTen, respectively.

Since the matrix and tensor generator algorithms are similar, and tensors are the generalization of matrices, the results are presented for the tensor case. The effectiveness of the tensor generator is validated by means of the feature quality and the decomposition performance of the generated tensors.

Table 11 shows the comparison of the generated tensors with their original versions (real tensors) in terms of some important features. Since the number of nonzeros is the most significant feature that a generator must obey, our generator applies some scalings during calculations to catch the given density. The success of the generator in terms of obeying the given density is seen in both levels of nonzero slice, nonzero fiber, and nonzero density. As can be seen in the table, the resulting densities, i.e. the nonzero count of the generated tensors, are at least 0.96 times smaller or at most 1.05 times larger than the ones of the respective original tensors.

Figure 30 depicts the success of the generated tensors in resembling the real tensors in terms of tensor decomposition performance. The performance of the generated tensors is compared with the performance of the naive random tensors, which have the same sizes and nonzero counts but the nonzero locations are uniformly random. The SPLATT tool is used for applying the CPD decomposition. The experiment is conducted on a 2-socket AMD EPYC 7352 CPU, using a single thread. The runtime results of both the naive random and the generated tensors are normalized with respect to the runtime of the original tensors. Therefore, the normalized values closer to 1.0 are interpreted as more successful in terms of resembling the original tensor performance. As can be seen in the figure, the generated tensors are superior to the naive random ones for 14 out of 16 cases, which validates the success of the generator.

3.6 PARTITIONING UTILITY API

We created a website called sparseutils.com to support the scientific community by providing easy access to pre-partitioned matrices from the SuiteSparse Matrix Collection and tools for matrix operations. This website is designed to help researchers and engineers optimize computations

Table 11 *Comparing the features of the original tensors with their generated versions.*

Name	Coefficient of Variation						Density								
	Fiber per Slice			Nonzero per Fiber			Nonzero Slices			Nonzero Fibers			Nonzeros		
	Org	Gen	Ratio	Org	Gen	Ratio	Org	Gen	Ratio	Org	Gen	Ratio	Org	Gen	Ratio
LBNL-network	8.0	4.5	0.56	26.2	12.1	0.46	2.1E-06	2.1E-06	1.00	8.4E-10	9.5E-10	1.13	4.2E-14	4.4E-14	1.05
NIPS	0.2	0.2	0.98	0.0	0.0	1.00	8.3E-04	8.3E-04	1.00	3.1E-05	3.0E-05	0.98	1.8E-06	1.8E-06	0.98
uber	0.2	0.2	0.94	1.0	1.0	0.94	1.0E+00	1.0E+00	1.00	1.4E-01	1.3E-01	0.93	3.8E-04	3.8E-04	0.99
chicago-crime-comm	0.4	0.3	0.82	0.5	0.4	0.82	9.5E-01	9.5E-01	1.00	3.2E-01	2.7E-01	0.83	1.5E-02	1.4E-02	0.96
chicago-crime-geo	0.3	0.3	0.94	0.1	0.0	0.00	1.0E-01	1.0E-01	1.00	2.8E-04	2.9E-04	1.01	8.9E-06	8.9E-06	1.01
vast-2015-mc1-3d	0.5	0.5	1.00	0.0	0.0	1.00	1.0E+00	1.0E+00	1.00	1.4E-02	1.4E-02	0.99	6.9E-03	6.9E-03	0.99
vast-2015-mc1-5d	0.0	0.0	0.00	0.0	0.0	0.00	6.9E-03	6.9E-03	1.00	6.9E-05	6.9E-05	1.00	7.8E-07	7.8E-07	1.00
DARPA1998	13.1	8.2	0.63	23.1	14.0	0.61	8.0E-01	8.1E-01	1.02	1.5E-04	1.6E-04	1.03	2.4E-09	2.4E-09	1.00
enron	4.1	3.6	0.87	1.8	1.4	0.76	4.4E-03	4.4E-03	1.00	3.7E-06	3.7E-06	0.99	5.5E-09	5.7E-09	1.05
NELL-2	3.3	3.1	0.94	0.9	1.1	1.25	1.0E+00	1.0E+00	1.00	3.0E-03	3.1E-03	1.01	2.4E-05	2.4E-05	0.99
freebase_music	24.4	20.2	0.83	0.1	0.0	0.00	9.7E-01	1.0E+00	1.03	1.8E-07	1.9E-07	1.04	1.1E-09	1.1E-09	1.03
flickr-3d	3.3	3.2	0.97	1.0	1.0	0.99	1.0E+00	1.0E+00	1.00	3.1E-06	3.1E-06	1.00	7.8E-12	7.9E-12	1.01
flickr-4d	1.0	1.0	0.99	0.0	0.0	1.00	3.1E-06	3.1E-06	1.00	7.8E-12	7.9E-12	1.01	1.1E-14	1.1E-14	1.01
freebase_sampled	24.0	19.4	0.81	0.1	0.0	0.00	9.1E-01	9.1E-01	1.00	9.2E-08	9.6E-08	1.05	1.7E-10	1.8E-10	1.04
delicious-3d	2.8	2.7	0.99	1.4	1.0	0.71	1.0E+00	1.0E+00	1.00	5.1E-06	5.1E-06	1.00	6.1E-12	6.1E-12	1.00
NELL-1	13.6	10.8	0.80	7.5	4.5	0.60	1.0E+00	1.0E+00	1.00	2.8E-06	2.8E-06	1.01	9.1E-13	9.2E-13	1.01

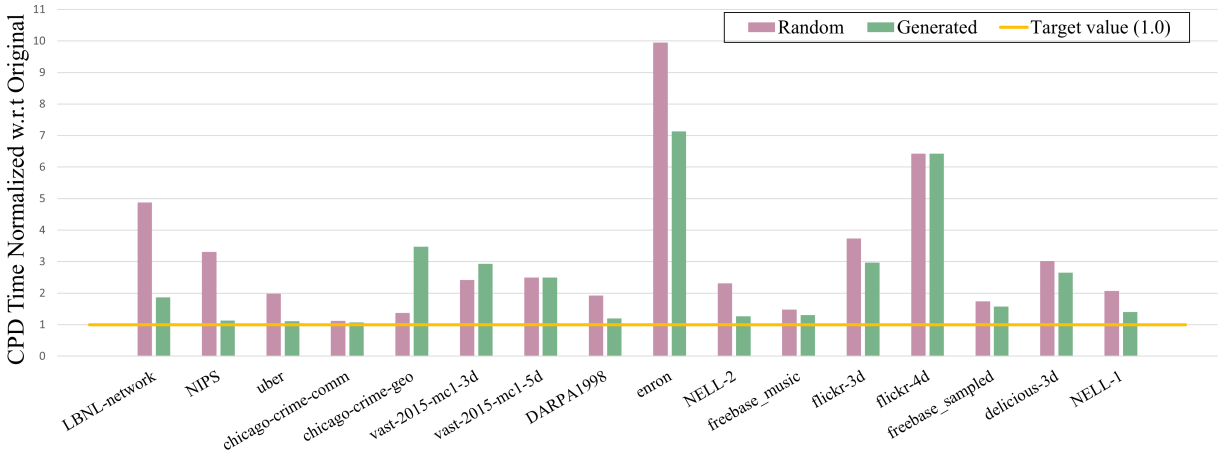


Figure 30 *CPD (SPLATT) runtime comparison of the naive random and the generated tensors using single thread. The values are normalized with respect to the runtime obtained for the respective original tensor.*

across various applications by allowing them to download and work with partitioning vectors.

Our goal is to provide a centralized resource where users can:

- Explore and download partition vectors for the matrices in the SuiteSparse Matrix Collection.
- Access tools for common tasks like partitioning matrices, working with partition vectors, and matrix reordering.

The SuiteSparse Matrix Collection is a large set of sparse matrices that arise in real applications. The Collection is widely used by the numerical linear algebra community for the development and performance evaluation of sparse matrix algorithms.

We aimed to offer pre-partitioned data from the SuiteSparse Matrix Collection because making researchers partition these matrices each time they need them wastes computational resources. Partitioning, especially of large matrices, is not only challenging but also demands significant

computing power. The Collection is useful for those in numerical linear algebra to develop and test sparse matrix algorithms, as it provides real-world matrices for more accurate and consistent experiments. By providing these matrices pre-partitioned, we aim to save valuable time and resources, facilitating more efficient and effective research.

Our collection consists of partitioning vectors for matrices in the SuiteSparse collection that have more than one hundred thousand non-zero values. For each matrix that satisfies this criterion, we used different partitioners (such as PaToH and METIS) to partition them into 2, 4, 8, 16, 32, 64, and 128 pieces respectively. During the partitioning process, we used all the available partitioning objectives offered by these partitioners. For example, PaToH offers two different partitioning objectives, compact and cutpart. We partitioned each matrix into powers of two for these two metrics respectively. If applicable, we used a constant seed of 1 for each partitioning task for consistency. We also recorded meta-data related to each partitioning operation such as the weights of each part, the maximum imbalance of the partitions, and the time it took to partition the matrix.

The website we created to share our collection is divided into three main sections: Partitions, Utilities, and About.

Partitions page contains the list of SuiteSparse matrices that we partitioned. On this page, using the filter bar, users can specify the properties of the matrices of which they want to download the partitioning vectors, such as specifying the maximum and minimum number of columns, rows, and non-zero value counts, or filtering by particular keywords. Then, they can choose the partitioner and partition count that they want to have. Users also have the option to download the entire collection of partitions if they choose.

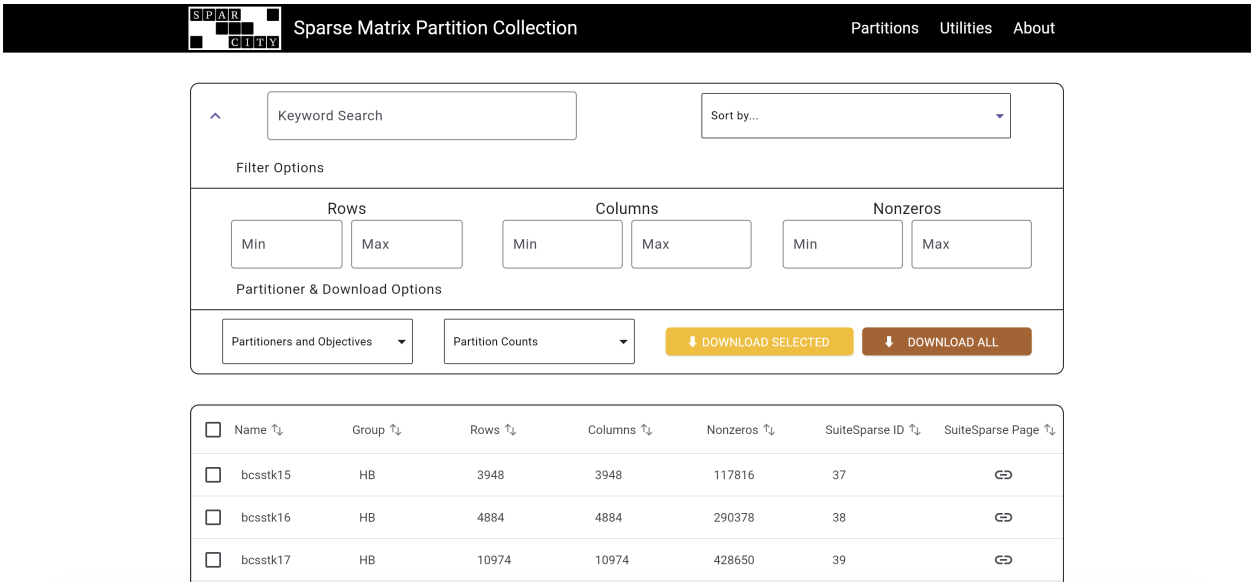


Figure 31 *Partitions page*

Utilities page contains the tools implemented for helping with tasks such as partitioning data or reordering matrix rows. As of the time of this writing, we provide two different utilities to the users. The first utility contains our codes utilizing partitioners such as PaToH and Metis, that we

used to partition the Suitesparse collection. We have instructions on how to compile and run our programs. These programs essentially act as wrappers for the partitioners. Our goal is to make these partitioners easier and more straightforward to use.

We also provide a reader and converter tools for the partitioning vectors. With the help of these tools, users can convert a partitioning vector to a permutation vector, and reorder the corresponding matrix rows with respect to the permutation vector. These programs are provided as plain C source code, and instructions on how to compile and use them are written in their GitHub repositories. They can work independently with each other.

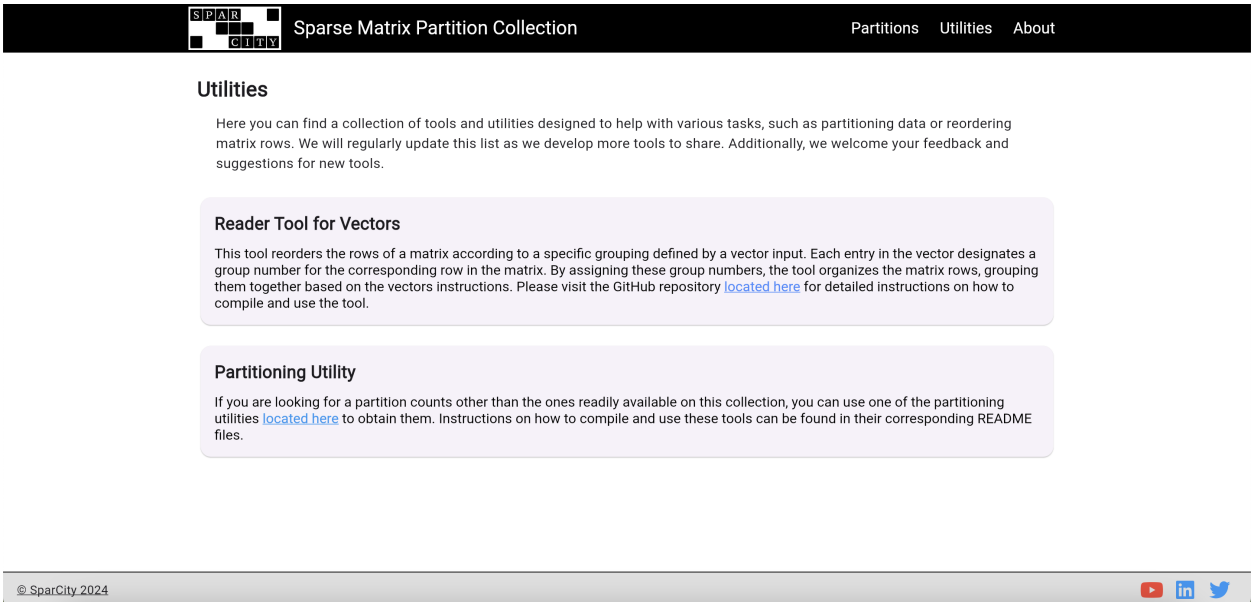


Figure 32 *Utilities page*

About page contains information about the properties of the data that is hosted on the website, how to use the filtering and sorting functionalities, information about how the partitioning vectors and tools were implemented, and lastly, the funding structure of the project.

We plan to continuously update our website using the feedback coming from our prospective users and include the utilities that will be developed in the future. We are also planning to extend our partitioning collection with matrices other than the ones present in the SuiteSparse Matrix collection.

The front end of the website was built using Flutter SDK. Python and Flask were used to develop the back end server, and is currently hosted on Heroku. Amazon S3 storage is being used to store the partitioning vectors. The utilities provided in the website are stored on their respective public GitHub repositories.

4 SPARCITY LIBRARIES

4.1 SPARSEBASE

Sparsebase is a C++ library designed for High-Performance Computing (HPC) applications, focusing on handling sparse data efficiently. It encompasses functions for encapsulating, preprocessing, and managing input/output operations on sparse data seamlessly and optimally, serving as a foundation for workflows involving such data types. The library's design is centered on achieving the following objectives: delivering the fastest code to users, presenting an intuitive API for ease of use, and ensuring the library's extensibility and maintainability. With its versatility to integrate into various sparse data workflows, the library prioritizes flexibility, notably in the choice of data types utilized for storage. This emphasis on adaptability is reflected in Sparsebase's highly templated structure, empowering users to employ data types that align with their specific workflows. Furthermore, Sparsebase provides utilities for securely manipulating these data types, enhancing the robustness of the library.

As described above, SparseBase is capable of performing various operations from reading sparse file formats such as Matrix Market files to extracting a wide range of sparse matrix/graph features. To be able to perform those operations, SparseBase heavily relies on formats. Formats, store sparse data structures and are the fundamental data structures of the library. Tensors, matrices, and graphs are all represented as Formats. Fig 33 is a code snippet showing a sample usage of **Sparsebase**. In the snippet, a matrix market file is read into a matrix, and then that matrix is converted, reordered, and permuted.

```
1  // Read the matrix market file into a Coordinate (COO) object
2  COO<int, int, float>* coo =
3              IOBase::ReadMTXToCOO<int, int, float>(filename);
4
5  // Convert the format object into a Compressed Sparse Row (CSR)
6  CSR<int, int, float>* csr = coo->Convert<CSR>();
7
8  // The context to use for carrying our reordering and permutation
9  CPUContext cpu;
10
11 // Carry out RCM reordering on the CSR object on the CPU.
12 //Allow converting input if needed
13 int* permutation =
14             ReorderBase::Reorder<RCMReorder>({}, csr,{&cpu}, true);
15
16 // Permute the original matrix object using the generated reordering
17 CSR<int, int, int>* new_csr =
18         ReorderBase::Permute2D<CSR>(permutation, csr, {&cpu}, true);
```

Figure 33 *A matrix market file is first read into a Coordinate (COO) object (Line 2), then it is converted to a Compressed Sparse Row (CSR) representation (Line 5). Afterwards, a Reverse Cuthill McKee (RCM) reordering is generated for it (Line 11), and the resulting permutation vector is used to transform the matrix (Line 14).*

Sparsebase is designed to seamlessly integrate into any workflow involving sparse data structures. To ensure this adaptability, formats were devised to store data as raw C++ arrays without

any additional auxiliary data or specific storage prerequisites. These arrays can be readily accessed by the user through a straightforward getter function. This facilitates swift retrieval of data from Sparsebase into their workflows, devoid of any additional code overhead. Furthermore, this storage approach guarantees optimal efficiency in terms of memory usage.

Formats can exist on different architectures. We specify an architecture in the library through a Context class. Currently, the library has two context classes: *CPUContext* for data on the CPU, and *CUDAContext* for data on a CUDA GPU.

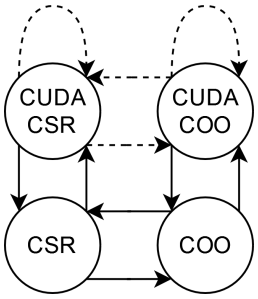


Figure 34 *Conversion graph. Each node is a Format object. Continuous lines are conversions, and dashed lines are conditional conversions.*

Another crucial functionality of SparseBase involves Conversions, which directly interacts with the Formats. Sparse data structures are renowned for their diverse representations, which vary in memory efficiency, access patterns, and stored data. SparseBase accommodates this diversity in representations by offering an intuitive interface for converting between format objects. It achieves this by conceptualizing Formats and conversions as a directed multi-graph. Each format class serves as a node in this graph, and a conversion from format A to format B constitutes a directed edge from A to B. This framework allows for *multi-step* conversions. For instance, in the conversion graph illustrated in Fig 34, if we aim to convert a *COO* object to a *CUDACSR* format, we can simply invoke a single call to the library with the desired destination format. The library will then automatically determine the path from *COO* to *CSR* to *CUDACSR* and execute the conversion accordingly, providing us with the *CUDACSR* object. This streamlined process is immensely advantageous, as it obviates the need to provide conversions to and from every other format when adding a new format to the library. Merely furnishing a conversion to and from a format in the strongly connected component of the conversion graph is sufficient.

There are also I/O operations in the SparseBase. Almost every workflow involving sparse data structures requires some I/O operations. Sparsebase provides an easy-to-use interface for optimally reading and writing sparse data to disk. It can read some of the most common file formats like the matrix market file format (`.mtx` file extension), edge lists, and the tensor file format (`.tns` file extension) and, graph format (**.graph**).

4.1.1 REORDERING

Sparse matrix reordering is a fundamental process in the world of high-performance computing. Reordering involves rearranging the initial structure of rows and columns of a sparse matrix in order to achieve specific objectives without altering its mathematical content. In this context, reordering plays a crucial role in optimizing the performance, efficiency, and memory usage of operations involving these types of matrices.

The distribution of non-zero elements across different sparse matrices is more often than not irregular and without a predictable pattern. Many sparse operations are then hard to optimize

due to the irregular access to memory. By using reordering algorithms, the distribution of non-zero in a sparse matrix can be optimized to address data locality and load-balancing issues commonly found in various sparse kernels.

Different reordering algorithms target different objectives. In this context, sparsebase offers a variety of algorithms with different objectives, each one with its own benefits. AMD and Nested Dissection are two reordering methods that focus on the reduction of fill-ins, described as the non-zeros that appear during Cholesky decomposition. RCM aims to reduce the bandwidth of the sparse matrix, while Rabbit finds community patterns and orders the rows and columns based on the community hierarchy of its graph representation. Sparsebase also provides a range of reordering algorithms that cater to more specific objectives beyond those currently listed. SlashBurn can be used as a compressing method for sparse matrices, with the focus in placing non-zeros closer together. BOBA is characterized as a lightweight reordering algorithm, effectively improving reordering time while maintaining a good final permutation. Gray ordering has a heavy focus in placing rows with similar non-zero distributions closer together.

The reordering functions accept four arguments. The first argument is a structure containing the reordering parameters specific to each algorithms. The second is the matrix structure, which can be in CSR, COO or CSC format. The third argument is the context in which the reordering will be performed. Finally, the fourth argument is a flag indicating whether the input matrix format should be converted to the required format for the reordering to take place. Most functions perform reordering on matrices in CSR format, except for BOBA, which operates on matrices in COO format.

The output of these reordering functions is the inverse permutation vector of the matrix, which is an array containing the new order of rows and/or columns of the sparse matrix. Each element of the permutation vector corresponds to the index of a row and/or a column. In an inverse permutation vector, each element represents the new index of a row or column, corresponding to the original index of that row or column. Therefore, if index 1 corresponds to row/column 1, the element in index 1 of the inverse permutation vector represents the new position of row/column 1 after reordering. To achieve the new reordered matrix structure, there are functions available in SparseBase that can be used to apply the inverse permutation vector to the original matrix structure. Below is an example illustrating the utilization of the Degree Reordering Algorithm. The only parameter in this scenario is a flag that determines whether we opt for reordering in ascending or descending order:

```
DegreeReorderParams params(true);
int*order = ReorderBase::Reorder<DegreeReorder>(params, csr, {&cpu}, false);
```

From the code above we obtain the inverse permutation vector. However, this vector alone isn't the final objective of the reordering process. We must proceed to transform the original matrix structure into its reordered version by applying the inverse permutation vector in the following way:

```
CSR<int,int,float>*new_csr = ReorderBase::Permute2D<CSR>(order, csr, {&cpu}, false);
```

This function returns the newly reordered matrix structure. It takes the same four arguments as the reordering functions, with the only difference being the first one: instead of passing the reordering parameters, we provide the inverse permutation vector. It's worth noting that various reordering algorithms come with different parameters. Typically, these algorithms have a default setting for their parameters. Table X provides a listing of the different reordering functions alongside their respective parameters.

4.1.2 PARTITIONING

Solving large-scale graph optimization problems often necessitates balanced partitioning due to the inherent challenges posed by the sheer size and complexity of the graphs involved. By breaking down a massive graph into smaller, manageable pieces, each segment can be processed independently, allowing for parallelization and distributed computing. However, this approach introduces potential suboptimality stemming from the interactions among different partitions. Additionally, the presence of links or connections between distinct parts of the graph can significantly impact both the runtime of the computation and the associated network communication costs. Thus, there is a strong motivation to achieve balanced partitioning with minimal cut size, ensuring efficient processing of graph data while mitigating the adverse effects of partitioning on overall performance and resource utilization.

That is why a strong partitioner called Mt-KaHyPar is integrated into SparseBase alongside other partitioning algorithms such as Patoh and METIS. Mt-KaHyPar stands out as a robust solution for the challenging task of graph and hypergraph partitioning, essential for various computational domains. With its fast and high-quality partitioning algorithms it can meet the diverse needs of users. One of the key strengths of Mt-KaHyPar lies in its scalability, demonstrated by its ability to efficiently utilize up to 64 threads without sacrificing solution quality. This scalability is crucial for handling increasingly large datasets and leveraging modern parallel computing architectures effectively. Furthermore, Mt-KaHyPar provides deterministic partitioning algorithms, ensuring consistent solutions for the same input and random seed. This deterministic behavior enhances reproducibility, facilitating experimentation and comparison across multiple runs, a critical requirement in research and development environments.

Mt-KaHyPar also addresses the need for partitioning graphs and hypergraphs into a large number of blocks with configurations tailored for scenarios where fine-grained partitioning is necessary. This capability is particularly relevant in applications requiring intricate control over partition granularity, accommodating scenarios where number of blocks exceed 1024. Additionally, the tool incorporates optimized data structures specifically designed for graph partitioning, resulting in significant speedups compared to conventional approaches. These optimizations contribute to improved efficiency and reduced computational overhead, making Mt-KaHyPar a preferred choice for large-scale partitioning tasks. Mt-KaHyPar supports various objective functions, including cut-net, connectivity, sum-of-external-degrees, and Steiner tree metrics. Each objective function offers distinct advantages and can be chosen according to specific application requirements, such as minimizing communication overhead or optimizing wire-lengths in VLSI design. Moreover, the tool facilitates the mapping of (hyper)graphs onto target graphs, a critical aspect in many applications. By optimizing the Steiner tree metric, Mt-KaHyPar minimizes the total weight of all Steiner trees induced by the nets of the hypergraph on the target graph, enabling efficient allocation of resources and minimizing communication costs in distributed systems.

In conclusion, Mt-KaHyPar emerges as a comprehensive and versatile tool for graph and hypergraph partitioning, offering a combination of speed, scalability, and quality. Its integration into frameworks like SparseBase expands its utility, allowing users to leverage its capabilities alongside other partitioning algorithms. With its diverse features and customizable objective functions, Mt-KaHyPar is well-equipped to address the partitioning challenges posed by modern computational applications.

4.1.3 FEATURE EXTRACTION

After examining workflows that involve sparse data, we note two observations: 1) Users typically extract features from their data in batches rather than individually, and 2) the computation of

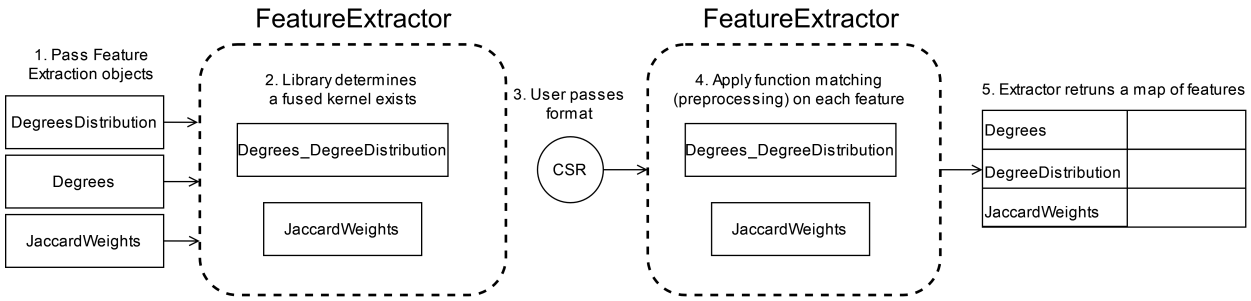


Figure 36 *Visualization of the feature extraction process shown in Şekil 35. 1) User passes the features they want to extract (Lines 5-7 in Şekil 35). 2) Library realizes a fused kernel exists and fuses degree and degree distribution. 3) User extracts these features for an input (Line 10 in Şekil 35), and 5) the library returns a map from each feature's ID to its output.*

various features often involves multiple shared calculation steps. We integrate a feature extraction module into Sparsebase to leverage these observations.

```
// FeatureExtractor is an implementation of ClassMatcherMixin
FeatureExtractor<int, int, int, double> engine;

// We add whichever features we wish to extract
engine.Add(DegreeDistribution<int, int, int, double>{});
engine.Add(Degrees<int, int, int>{});
engine.Add(JaccardWeights<int, int, int, double>{});

// Extractor fuses the first two
// features and executes two kernels only
auto raws = extractor.Extract(features, csr);

// Fetch whichever features are needed
// through specific IDs for Feature classes
int * degree_dist = std::any_cast<int*>(
    raws[DegreeDistribution<int, int, int, double>
                        ::get_feature_id_static()]
    );
feature_type * jac_w = std::any_cast<int*>(
    raws[JaccardWeights<int, int, int, double>
                        ::get_feature_id_static()]
    );
```

Figure 35 *Extracting degrees and degree distribution*

Firstly, we define a special kind of preprocessing classes called **Features**. Each one of these classes extracts one or more features efficiently. Examples of such classes are **DegreeDistribution** which can extract the degree distribution of the vertices in a graph, **Degrees**, which extracts the degree of every vertex in a graph, and **Degrees_DegreeDistribution** which will overlap the computation of the former two classes and extract both of their outputs in one go.

The second part of this system is the **ClassMatcherMixin** class. This class can take multiple

Feature classes, figure out if any of them can be replaced by their fused counterparts, and allow the user to fetch them all at once efficiently. In other words, it allows the user to specify the individual kernels they want, and then exploits any possibility for overlapping feature computations. The code snippet in Fig 35 demonstrates the usage of concrete *ClassMatcherMixin* class called *FeatureExtractor* which is specific for *FormatOrderTwo* classes. Fig 36 visualizes this process.

4.2 MPI COMMUNICATION OFFLOADING

In the reporting period we have developed a library, named *libmmcso* (MPI Multithreaded Communication Software Offloading), for transparent communication optimization of MPI applications. Using our library, internal locking within the MPI library in multithreaded hybrid MPI applications can be avoided. Furthermore, internal progress within the MPI library can be guaranteed for asynchronous communication operations (i.e., `Isend` or `Irecv`) and one-sided operations (i.e., `Put` or `Get`). This can reduce overheads from resource competition and improve the overlap of communication and computation.

Both objectives, avoiding internal locking and providing progress guarantees, are achieved by offloading the communication of an MPI application to one or more dedicated communication threads (*offload threads*). This, however, comes at the cost of compute resources due to the reservation of resources for the communication thread(s).

4.2.1 SOFTWARE ARCHITECTURE

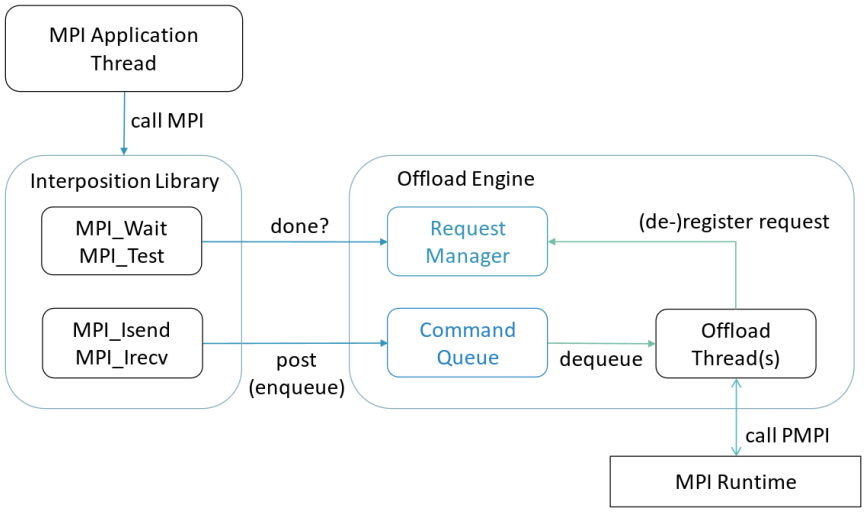


Figure 37 *High-level software architecture of* `libmmcso`.

Our approach is realized as an interposition layer residing between the MPI application and the MPI library, and is compatible with common MPI implementations, such as IntelMPI, OpenMPI, and MVAPICH. It can be used transparently by MPI applications using the standard MPI interfaces. Code changes or recompiling is not necessary to use our library.

Fig. 37 shows the high-level software architecture. MPI calls made by threads of an MPI application are intercepted by the interposition library. They are enqueued into a command queue, instead of executed immediately. The offload thread dequeues commands, if available, and performs the actual MPI operations using MPI's *PMPI* interface.[72] Additionally, the offload

[72]Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. 2021. URL: https:

thread manages pending MPI requests and provides the application thread with an `MPI_Request` object. The application thread can check this object for completion of the MPI call. Pending MPI requests are checked for completion periodically by the offload thread, which forwards the updated status to the application thread via the `MPI_Request` object.

4.2.2 MICROBENCHMARKS

To evaluate the benefits of our communication offloading libary, we have developed a microbenchmark suite, measuring overlap of computation and communication for point-to-point MPI operations. Each thread of an MPI process issues a nonblocking `MPI_Irecv` followed by a nonblocking `MPI_Isend`. Next, each thread performs a certain amount of work (sleeps for a certain amount of time), before the thread issues two `MPI_Wait` calls, waiting for completion of the nonblocking MPI calls. The communication happens between pairs of threads with the same thread ID of the involved MPI processes. We measure the time spent in MPI calls (*overhead*) and the ratio of time spent doing work divided by the total time (*overlap*) in dependence of the message size that is communicated. This experiment is similar to the microbenchmark described by Vaidyanathan et al.[73] to evaluate the effectiveness of asynchronous progress and to quantify the overhead of issuing nonblocking MPI calls.

Experimental Setup Our experiments are performed on the SuperMUC supercomputer. Each compute node of the SuperMUC is equipped with a 48-core Intel Xeon Platinum 8174 (Skylake) CPU. The CPUs have 2 sockets (NUMA-domains) and 24 cores per socket.

For evaluation we use the following inter-node communication scenario using four MPI processes on two compute nodes. Communication is performed between threads of one compute node to another. The threads (max. 23) of the same MPI process are pinned to dedicated cores in the same NUMA-domain. One core per NUMA-domain is dedicated to the offloading thread of the MPI application. We compare the performance of two standard MPI implementations (IntelMPI and OpenMPI) using `MPI_THREAD_MULTIPLE` to our communication offloading approach.

Inter-Node Performance Fig. 8 shows the inter-node communication overhead and overlap of computation and communication using our microbenchmark with scenario (2).

Again, the offloading approach has lower overhead for small messages below about 100 kB. For larger messages, only Intel MPI using messages between 100 kB and 1 MB with 23 threads is significantly better in terms of overhead. The achieved overlap using the offloading strategy is again equally good or better in all of our test cases.

4.2.3 PARTITIONED COMMUNICATION

Partitioned communication is based on the functionality for persistent communication, partitioned point-to-point communication is included in MPI4.0.[74] The idea is to provide a simple way to combine multithreading with message passing while avoiding the performance implications of fully multithreaded concurrent send/receive operations. Partitioned communication extends the usual `(buffer, count, datatype)` descriptor for the data to be transmitted or received with a fourth parameter denoting the number of partitions. The 4-tuple `(buffer, partitions, count, datatype)` specifies that `buffer` contains `partitions` many partitions, each holding

`//www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`.

[73] Karthikeyan Vaidyanathan et al. "Improving concurrency and asynchrony in multithreaded MPI applications using software offloading". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12.

[74] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*.

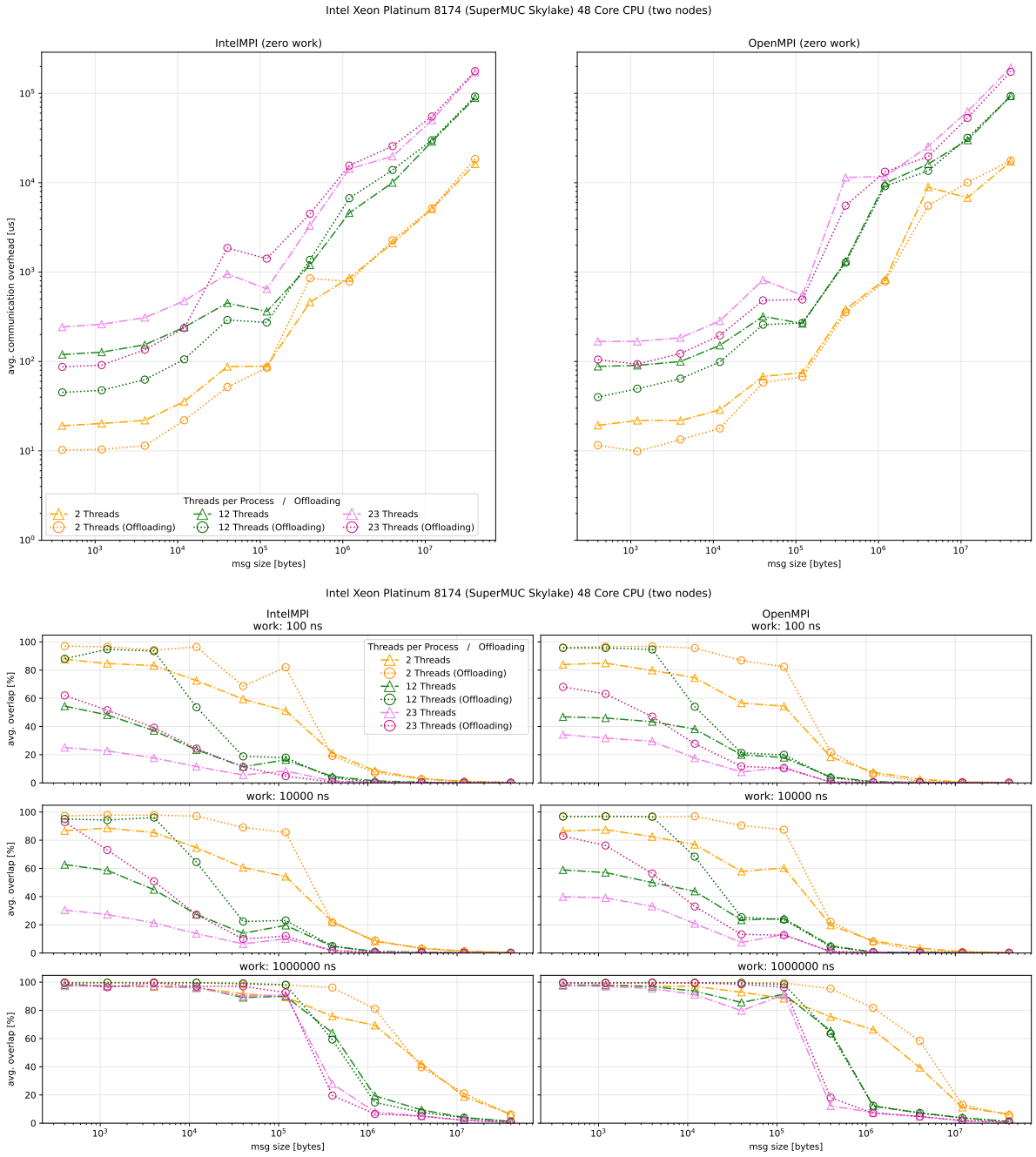


Figure 38 *Top: inter-node communication overhead (lower is better) using our microbenchmark with zero work and four MPI processes with 1, 12, and 23 threads per MPI process with and without communication offloading for varying message size. Bottom: inter-node communication overlap of computation and communication (higher is better) using our microbenchmark and four MPI processes with 1, 12, and 23 threads per MPI process with and without communication offloading for varying message size and amount of work.*

`count` elements of `datatype`. The communication model is then extended with the functions `MPI_Pready` and `MPI_Parrived` specifying the partition number. `MPI_Pready` is used on the sender side to indicate that a particular partition of the send buffer is ready to be sent. `MPI_Parrived` on the other hand can be used on the receiver side to query if the data in a particular partition

has fully arrived. these functions may be called by different threads, but they do not impose a requirement for the MPI library to immediately send the data that was marked as ready. Instead, an MPI implementation may aggregate message partitions as it sees fit and to guarantee the completion of the send and receive operation, `MPI_Wait` or `MPI_Test` have to be used.

Holmes et al.[75] have proposed partitioned collective operations. Here we follow their exposure with some simplifications, which we believe will allow for a simpler and more efficient implementation. The basic concept in partitioned collective operations is the same as with partitioned point-to-point operations. The same sequence of init-start-wait-free procedures is employed. For each persistent collective operation there is a partitioned variant, except for barrier which moves no data.

A simplified list of proposed partitioned collectives is shown in Fig. 41. Each `(buffer, count, datatype)` triple is replaced by a `(buffer, partitions, count, datatype)` 4-tuple, `count` now again specifying the number of elements *per partition*. While in partitioned point-to-point operations sender and receiver may have different partitioning, we propose identical partitioning for collectives for reasons of simplicity and efficiency of implementation. I.e., where MPI requires that all invocations of a collective call have the same number of elements and datatype, we require that they have the same number of partitions as well. Note that this is different from the proposal of Holmes et al.,[76] where different partition counts are allowed. Fig. 39 shows an example where a different number of partitions is specified in each process (not allowed by our proposal) and Fig. 40 shows an example where all processes provide the same number of partitions (as mandated by our proposal).

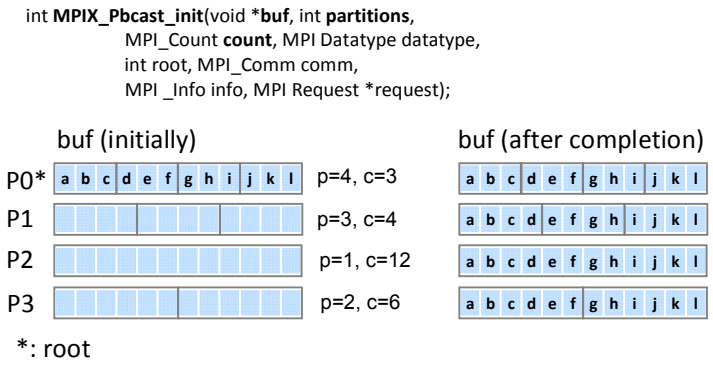


Figure 39 *Setting up a partitioned broadcast operation. Each participating process specifies a different number of partitions (p) and a corresponding number of elements per partition (c).*

Fig. 41 shows a partial list of collective operations in their partitioned form. The functions `MPI_Pready` and `MPI_Parrived` stay the same as in the case of point-to-point operations. In the case of reduce-type operations, it should be noted that the proposed interface allows for operations on partially received data and thus unlocks more overlap potential compared to the regular (non-partitioned) variants of the reduce operation. The reduce functions supported by these operations also include user-defined functions.

Besides the collective operations shown in Fig. 41, partitioned communication is also available for the "v" and "w" variants and for neighborhood collectives.

[75] Daniel J Holmes et al. "Partitioned collective communication". *2021 Workshop on Exascale MPI (ExaMPI)*. IEEE. 2021, pp. 9–17.

[76] Ibid.

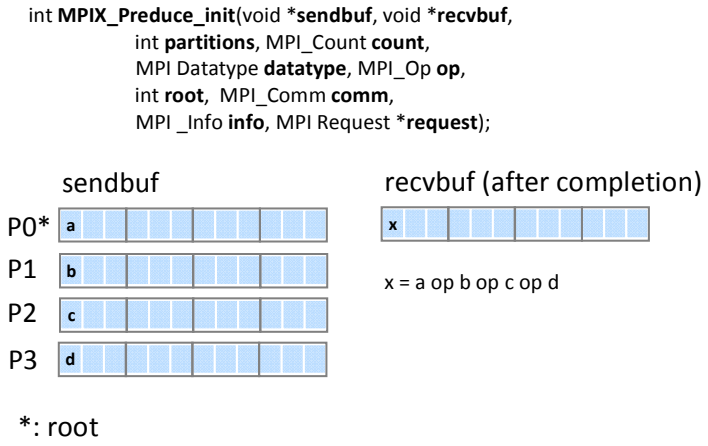


Figure 40 *Setting up a partitioned reduce operation. Here each participating process specifies the same number of partitions to enable a simplified implementation.*

Combining Software Offloading with Partitioned Communication Partitioned communication, as defined for point-to-point operations by the MPI4.0 standard and extended for collective operations as described above, can be combined with software offloading. This could enable certain optimization opportunities, such as for example computation on partially received messages (partitions), in partitioned point-to-point and collective operations. A prototypical implementation of partitioned reduce, broadcast, and neighborhood collectives is planned in the future.

4.2.4 SPARSE COMPUTATION USE CASE

As future work, we plan to implement the proposed partitioned collective scheme using our communication offload approach and test the benefits on a sparse computation use case. This use case is the simulation of cardiac electrophysiology using a hybrid (MPI plus threads) approach, developed by the Simula partner, employing an unstructured mesh using an explicit method. The code for this application uses an iterative computational loop with `MPI_Neighbor_alltoall_v` outside of parallel region (executed by the master thread). Local SpMV (sparse matrix-vector product) computation is performed in each iteration by multiple threads.

In each iteration, parts of the (updated) vector (depending on the unstructured mesh) must be communicated to neighbor processes, according to the sparsity pattern given by the unstructured mesh, where the number of neighbors can vary and communication can be overlapped with computation. Previous investigations have indicated that using MPI communication operations from multiple threads (`MPI_THREAD_MULTIPLE` can lead to degraded performance due to locking and contention for communication queues in MPI. We plan to investigate if the halo exchange can be more efficiently be implemented in terms of (partitioned) P2P communication plus software offloading or with (partitioned) `MPI_Neighbor_alltoall_v` collective plus software offloading.

```
int MPIX_Pbcast_init(void *buf, int partitions, MPI_Count count,
                     MPI_Datatype datatype, int root, MPI Comm comm,
                     MPI_Info info, MPI_Request *request);

int MPIX_Pgather_init(const void *sendbuf, int sendparts,
                      MPI_Count sendcount, MPI_Datatype sendtype,
                      void *recvbuf, int recvparts,
                      MPI_Count recvcount, MPI_Datatype recvtype,
                      int root, MPI Comm comm, MPI_Info info,
                      MPI_Request *request);

int MPIX_Pscatter_init(const void *sendbuf, int sendparts,
                       MPI_Count sendcount, MPI_Datatype sendtype,
                       void *recvbuf, int recvparts, MPI_Count recvcount,
                       MPI_Datatype recvtype, int root,
                       MPI Comm comm, MPI_Info info, MPI_Request *request);

int MPIX_Pallgather_init(const void *sendbuf, int sendparts,
                         MPI_Count sendcount, MPI_Datatype sendtype,
                         void *recvbuf, int recvpart, MPI_Count recvcount,
                         MPI_Datatype recvtype, MPI Comm comm,
                         MPI_Info info, MPI_Request *request);

int MPIX_Palltoall_init(const void *sendbuf, int sendparts,
                        MPI_Count sendcount, MPI_Datatype sendtype,
                        void *recvbuf, int recvpart, MPI_Count recvcount,
                        MPI_Datatype recvtype, MPI_Comm comm,
                        MPI_Info info, MPI_Request *request);

int MPIX_Pallreduce_init(const void *sendbuf, void *recvbuf,
                         int partitions, MPI_Count count,
                         MPI_Datatype datatype, MPI_Op op,
                         MPI_Comm comm, MPI_Info info,
                         MPI_Request *request);
```

Figure 41 *A selected subset of proposed partitioned collective communication operations, based on the the work of Holmes et al.*[77]

5 CONCLUSIONS

Over the three years of the SparCity project, we have managed to create an extensive framework of methods, software tools and libraries. As shown in this deliverable, as well as in the preceding Deliverables 5.1 & 5.2, the SparCity framework covers many topics that can enhance sparse computations involving sparse matrices, graphs and tensors. The related scientific results have been published as research papers and open-source libraries and database. There is also good reason to expect that some of the tools and libraries will be further developed by the respective SparCity partners, because these are in line with their research profiles and future ambitions. Overall, we consider the SparCity framework as a timely and useful input to the scientific community, with also the potential of being adopted by other parties of interest.

REFERENCES

Adhianto. "HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs Http://Hpctoolkit.Org". *Concurr. Comput.: Pract. Exper.* 22.6 (2010), pp. 685–701. ISSN: 1532-0626.

Agelastos. *The Lightweight Distributed Metric Service: A Scalable Infrastructure for Continuous Monitoring of Large Scale Computing Systems and Applications*. English. Tech. rep. SAND2014-19868C. Sandia National Lab. (SNL-NM), Albuquerque, NM (United States); Sandia National Lab. (SNL-CA), Livermore, CA (United States), 2014. DOI: 10.1109/SC.2014.18. URL: https://www.osti.gov/biblio/1315267 (visited on 09/27/2021).

Aksar. "E2EWatch: An End-to-End Anomaly Diagnosis Framework for Production HPC Systems"". *Euro-Par 2021: Parallel Processing*. Springer International Publishing, 2021, pp. 70–85.

Amazon Web Services. *Amazon EC2 G5 Instances*. https://aws.amazon.com/ec2/instance-types/g5/. 2023.

Amestoy, Patrick R., Timothy A. Davis, and Iain S. Duff. "Algorithm 837: AMD, an Approximate Minimum Degree Ordering Algorithm". *ACM Trans. Math. Softw.* 30.3 (2004), pp. 381–388. ISSN: 0098-3500. DOI: 10.1145/1024074.1024081.

Azad, Ariful et al. "The reverse Cuthill-McKee algorithm in distributed-memory". *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, pp. 22–31.

Brandt. *Lightweight Distributed Metric Service (LDMS): Run-time Resource Utilization Monitoring*. English. Tech. rep. SAND2013-6521C. Sandia National Lab. (SNL-CA), Livermore, CA (United States); Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), 2013. URL: https://www.osti.gov/biblio/1106397 (visited on 09/27/2021).

Breiter, Sergej, James D Trotter, and Karl Fürlinger. "Modelling Data Locality of Sparse Matrix-Vector Multiplication on the A64FX". *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023, pp. 1334–1342.

Breiter, Sergej et al. "A Profiling-Based Approach to Cache Partitioning of Program Data". *International Conference on Parallel and Distributed Computing: Applications and Technologies*. Springer. 2022, pp. 453–463.

Catalyurek, U.V. and C. Aykanat. "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication". *IEEE Transactions on Parallel and Distributed Systems* 10.7 (1999), pp. 673–693. DOI: 10.1109/71.780863.

Chen, Hongzheng et al. "Krill: a compiler and runtime system for concurrent graph processing". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–16.

Choi, Jee Whan et al. "A roofline model of energy". *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, pp. 661–672.

Cluster Cockpit. https://www.clustercockpit.org/. Accessed on 30 Sep 2023.

Coutinho, Afonso Silva Mendes. "CARM-based approach for sparse computation characterisation". MA thesis. Instituto Superior Técnico, Universidade de Lisboa, 2022.

Cuthill, E. and J. McKee. "Reducing the Bandwidth of Sparse Symmetric Matrices". *Proceedings of the 1969 24th National Conference*. Association for Computing Machinery, 1969, pp. 157–172. DOI: 10.1145/800195.805928.

Cuthill, Elizabeth. "Several Strategies for Reducing the Bandwidth of Matrices". *Sparse Matrices and their Applications*. Springer, 1972, pp. 157–166.

Cuthill, Elizabeth and James McKee. "Reducing the bandwidth of sparse symmetric matrices". *Proceedings of the 1969 24th national conference*. 1969, pp. 157–172.

Davis, Tim. *Sparse Matrix Collection*. Accessed on 5th October 2023. URL: https://sparse.tamu.edu/.

Davis, Timothy A. and Yifan Hu. "The University of Florida Sparse Matrix Collection". *ACM Trans. Math. Softw.* 38.1 (2011). ISSN: 0098-3500. DOI: 10.1145/2049662.2049663.

Dhandhania, Sunidhi et al. "Explaining the Performance of Supervised and Semi-Supervised Methods for Automated Sparse Matrix Format Selection". *50th International Conference on Parallel Processing Workshop*. 2021, pp. 1–10.

Ding, Nan and Samuel Williams. "An Instruction Roofline Model for GPUs". *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2019, pp. 7–18. DOI: 10.1109/PMBS49563.2019.00007.

Doerfler, Douglas et al. "Applying the roofline performance model to the intel xeon phi knights landing processor". *High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^ 3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, 2016, Revised Selected Papers 31*. Springer. 2016, pp. 339–353.

Dongarra, Jack and Michael A Heroux. "Toward a new metric for ranking high performance computing systems". *Sandia Report, SAND2013-4744* 312 (2013), p. 150.

Filipovič, Jiří et al. "Optimizing CUDA code by kernel fusion: application on BLAS". *The Journal of Supercomputing* 71.10 (2015), pp. 3934–3957.

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

Ganglia. *Monitoring system*. 2022. URL: http://ganglia.sourceforge.net/ (visited on 12/12/2022).

George, Alan. "Nested Dissection of a Regular Finite Element Mesh". *SIAM Journal on Numerical Analysis* 10.2 (1973), pp. 345–363. DOI: 10.1137/0710032.

George, Alan and Joseph W. H. Liu. "An Implementation of a Pseudoperipheral Node Finder". *ACM Transactions on Mathematical Software* 5.3 (1979), pp. 284–295. DOI: 10.1145/355841.355845.

— "The evolution of the minimum degree ordering algorithm". *SIAM Review* 31.1 (1989), pp. 1–19.

George, Alan and David R. McIntyre. "On the Application of the Minimum Degree Algorithm to Finite Element Systems". *SIAM Journal on Numerical Analysis* 15.1 (1978), pp. 90–112. ISSN: 00361429. URL: http://www.jstor.org/stable/2156565.

Gibbs, Norman E., William G. Poole, and Paul K. Stockmeyer. "An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix". *SIAM Journal on Numerical Analysis* 13.2 (1976), pp. 236–250. ISSN: 00361429.

Gilbert, J. R. and R. E. Tarjan. "The Analysis of a Nested Dissection Algorithm". *Numer. Math.* 50.4 (1987), pp. 377–404. ISSN: 0029-599X. DOI: 10.1007/BF01396660.

Haque, Sardar Anisul and Shahadat Hossain. "A Note on the Performance of Sparse Matrix-vector Multiplication with Column Reordering". *2009 International Conference on Computing, Engineering and Information*. 2009, pp. 23–26. DOI: 10.1109/ICC.2009.40.

Heras, D.B. et al. "Modeling and improving locality for the sparse-matrix–vector product on cache memories". *Future Generation Computer Systems* 18.1 (2001), pp. 55–67. ISSN: 0167-739X. DOI: 10.1016/S0167-739X(00)00075-3.

Holmes, Daniel J et al. "Partitioned collective communication". *2021 Workshop on Exascale MPI (ExaMPI)*. IEEE. 2021, pp. 9–17.

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.

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

Ismayilov, Ismayil et al. "Multi-GPU Communication Schemes for Iterative Solvers: When CPUs are Not in Charge". *Proceedings of the 37th International Conference on Supercomputing*. 2023, pp. 192–202.

Karypis, George and Vipin Kumar. "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs". *SIAM Journal on Scientific Computing* 20.1 (1998), pp. 359–392. DOI: 10.1137/S1064827595287997.

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". *High Performance Computing: 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24-28, 2018, Proceedings 33*. Springer. 2018, pp. 226–245.

Langguth, Johannes et al. "Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes". *Journal of Parallel and Distributed Computing* 76 (2015), pp. 120–131. DOI: 10.1016/j.jpdc.2014.10.005.

Li, Xiaoping, Yadi Wang, and Rubén Ruiz. "A survey on sparse learning models for feature selection". *IEEE transactions on cybernetics* 52.3 (2020), pp. 1642–1660.

Liu, Wai-Hung and Andrew H Sherman. "Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithms for sparse matrices". *SIAM Journal on Numerical Analysis* 13.2 (1976), pp. 198–213.

Marques, Diogo et al. "Application-driven cache-aware roofline model". *Future Generation Computer Systems* 107 (2020), pp. 257–273.

Mazloumi, Abbas, Xiaolin Jiang, and Rajiv Gupta. "Multilyra: Scalable distributed evaluation of batches of iterative graph queries". *2019 IEEE International Conference on Big Data (Big Data)*. IEEE. 2019, pp. 349–358.

McCalpin, John. "Memory bandwidth and machine balance in high performance computers". *IEEE Technical Committee on Computer Architecture Newsletter* (1995), pp. 19–25.

Merrill, Duane and Michael Garland. "Merge-Based Sparse Matrix-Vector Multiplication (SpMV) Using the CSR Storage Format". *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2016.

— "Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format". *Acm Sigplan Notices* 51.8 (2016), pp. 1–2.

Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

Milenković, Katarina. "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/.

Murphy, Richard C et al. "Introducing the Graph 500". *Cray Users Group (CUG)* 19 (2010), pp. 45–74.

Nagios. *Nagios*. https://www.nagios.org/. Accessed: 2022-12-12. 2022.

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

Nvidia Corporation. *NVIDIA CUDA Compiler Driver NVCC*. 2022. URL: https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_Data_Center_GPU_Driver_Release_Notes_510_v1.0.pdf.

Oliker, Leonid et al. "Effects of Ordering Strategies and Programming Paradigms on Sparse Matrix Computations". *SIAM Review* 44.3 (2002), pp. 373–393. DOI: 10.1137/S00361445003820.

Pan, Peitian and Chao Li. "Congra: Towards efficient processing of concurrent graph queries on shared-memory machines". *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE. 2017, pp. 217–224.

Performance Co-Pilot. https://pcp.io/. Accessed on 30 Sep 2023.

Pinar, Ali and Michael T. Heath. "Improving Performance of Sparse Matrix-Vector Multiplication". *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. Portland, Oregon, USA: Association for Computing Machinery, 1999. DOI: 10.1145/331532.331562.

Röhl, Thomas et al. "LIKWID Monitoring Stack: A Flexible Framework Enabling Job Specific Performance monitoring for the masses". *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 2017, pp. 781–784. DOI: 10.1109/CLUSTER.2017.115.

Röhl, Thomas et al. "Overhead Analysis of Performance Counter Measurements". *2014 43rd International Conference on Parallel Processing Workshops*. 2014, pp. 176–185. DOI: 10.1109/ICPPW.2014.34.

Roy. "PerfAugur: Robust diagnostics for performance anomalies in cloud services". *2015 IEEE 31st International Conference on Data Engineering*. 2015, pp. 1167–1178. DOI: 10.1109/ICDE.2015.7113365.

Sasongko, Muhammad Aditya et al. "Precise Event Sampling on AMD Versus Intel: Quantitative and Qualitative Comparison". *IEEE Transactions on Parallel and Distributed Systems* 34.5 (2023), pp. 1594–1608. ISSN: 1558-2183. DOI: 10.1109/TPDS.2023.3257105.

Shun, Julian and Guy E Blelloch. "Ligra: a lightweight graph processing framework for shared memory". *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2013, pp. 135–146.

Strohmaier, Erich. "TOP500 supercomputer". *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. Tampa, Florida: Association for Computing Machinery, 2006, 18–es. DOI: 10.1145/1188455.1188474. URL: https://doi.org/10.1145/1188455.1188474.

Team, The Smartmontools. *Smartmontools*. Accessed on 5th October 2023. URL: https://www.smartmontools.org/.

Trotter, James D et al. "Bringing Order to Sparsity: A Sparse Matrix Reordering Study on Multicore CPUs". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2023, pp. 1–13.

Unat, Didem et al. "ExaSAT: An exascale co-design tool for performance modeling". *The International Journal of High Performance Computing Applications* 29.2 (2015), pp. 209–232. DOI: 10.1177/1094342014568690. URL: https://doi.org/10.1177/1094342014568690.

Vaidyanathan, Karthikeyan et al. "Improving concurrency and asynchrony in multithreaded MPI applications using software offloading". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12.

Wahib, Mohamed and Naoya Maruyama. "Scalable kernel fusion for memory-bound GPU applications". *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2014, pp. 191–202.

Wang, Endong et al. "Intel Math Kernel Library". *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*. Springer International Publishing, 2014, pp. 167–188. DOI: 10.1007/978-3-319-06486-4_7. URL: https://doi.org/10.1007/978-3-319-06486-4_7.

Wang, Guibin, YiSong Lin, and Wei Yi. "Kernel fusion: An effective method for better power efficiency on multithreaded GPU". *2010 IEEE/ACM Int'l Conference on Green Computing and*

Communications & Int'l Conference on Cyber, Physical and Social Computing. IEEE. 2010, pp. 344–350.

Wang, Yangzihao et al. "Gunrock: A high-performance graph processing library on the GPU". *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 2016, pp. 1–12.

Weaver, Vincent M. et al. "Measuring Energy and Power with PAPI". *2012 41st International Conference on Parallel Processing Workshops*. 2012, pp. 262–268. DOI: 10.1109/ICPPW.2012.39.

Xue, Jilong et al. "Seraph: an efficient, low-cost system for concurrent graph processing". *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 2014, pp. 227–238.

Yzelman, A. N. and Rob H. Bisseling. "Cache-Oblivious Sparse Matrix–Vector Multiplication by Using Sparse Matrix Partitioning Methods". *SIAM Journal on Scientific Computing* 31.4 (2009), pp. 3128–3154. DOI: 10.1137/080733243.

Zhang, Yunming et al. "Graphit: A high-performance graph dsl". *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–30.

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, pp. 601–609. DOI: 10.1109/ICCD50377.2020.00105.

Zhao, Jin et al. "GraphM: an efficient storage system for high throughput of concurrent graph processing". *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–14.

6 HISTORY OF CHANGES

Version	Author(s)	Date	Comment
0.1	Xing Cai	15.03.2024	Initial draft skeleton
0.2	Johannes Langguth	21.03.2024	Added mostly old text in 2.1. Will be modernized.
0.3	Beyza Cavusoglu	23.03.2024	Added SparseBase section
0.4	Didem Unat	24.03.2024	Added Graph Fusion section
0.5	Sinan Ekmekcibasi	25.03.2024	Completed SparseBase section.
0.6	Sergej Breiter	26.03.2024	Added input about A64FX cache profiler.
0.7	Tuğba Torun	27.03.2024	Added input about sparse matrix/tensor generator.
0.8	Emre Duzakın	27.03.2024	Added input about partitioner utility API.
0.9	Karl Fuerlinger	27.03.2024	Added input about PI communication offloading.
1.0	Kamer Kaya	30.03.2024	Added input about SuperTwin and SparseVis.
1.1	Johannes Langguth	30.03.2024	More input about ML-based recommendation methods.
1.2	Xing Cai	31.03.2024	Finalized the entire deliverable.

Table 12 *Document History of Changes*