Profiling and instrumentation tools

Deliverable No: D4.1
Deliverable Title: Profiling and instrumentation tools
Deliverable Publish Date: 31 March 2022

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

List of partners:

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

CONTENTS

1 INTRODUCTION

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

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

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

1.1 OBJECTIVES OF THIS DELIVERABLE

The objective of this deliverable is to provide an overview of the tools for performance, power and energy monitoring on computing nodes equipped with different device architectures (such as CPU, GPU, and Graphcore IPU), as well as on HPC systems that combine several computing nodes. This step is important in order to define the most adequate platform-specific instrumentation frameworks to be used when assessing the application characteristics, determining the execution bottlenecks and evaluating their ability to fully exploit the capabilities of a given hardware. A special emphasis is paid on tools and frameworks for run-time computation and communication monitoring, also based on hardware counters.

1.2 WORK PERFORMED

In this deliverable, the features of several tools for performance, power and energy monitoring on different computing systems and devices, including multi-core CPUs, GPUs and Graphcore IPUs,

are presented. The deliverable starts with the introduction of some of the existing performance and monitoring events that can be used to profile and identify bottlenecks of sparse applications. Next, a set of profiling tools based on hardware counters is also described, as these tools are traditionally used to access the hardware counters in-built in current computing systems. Since some of these counters can also be used for communication profiling and execution monitoring, these type of tools is also targeted in this deliverable. Moreover, dynamic binary instrumentation approaches are also covered, as the information obtained from these methods can provide additional information to complement the one obtained from hardware and software events. Furthermore, frameworks for application analysis on Intel CPUs and GPUs, NVIDIA GPUs and Graphcore IPUs are also addressed in this deliverable. Finally, a performance model of sparse kernels based on cache simulation is presented.

1.3 DEVIATIONS AND COUNTER MEASURES

There was no deviation from the work plan.

1.4 RESOURCES

It is likely that in the course of the project, new profiling and instrumentation tools and frameworks will be added to the core list provided in this document (and/or their set of features and implementation will be improved).

2 PERFORMANCE AND ENERGY MONITORING EVENTS

Due to the ability of sparse computations to reduce the storage and computational requirements of real-world applications, these type of kernels have become increasingly relevant in several scientific fields, such as physics, mathematics and machine learning.[1] Sparse kernels rely on data formats tailored to efficiently store the non-zero entries of the input data,[2] which result in irregular memory access patterns, leading to performance and efficiency degradation. To tackle this issue, it is essential to correlate the application characteristics and the capabilities of the underlying hardware when performing sparse applications, allowing to uncover the main bottlenecks that affect their execution.

To this end, software and hardware events in-built in current operating and computing systems can be used to to profile application execution on different devices, and to extract metrics useful for the optimization of sparse applications. For example, for the Linux operating system, there are several tools to monitor the different components that compose the operating system (see Figure 1). These software events can provide insights regarding the utilization of different components in the software and hardware stack, *e.g.*, disk and network usages, which can become a bottleneck when deploying sparse applications in large scale systems. Through performance monitoring tools, additional information can be obtained from the hardware counters, which are contained in the chip package of modern processors, thus providing insights on the utilization of different

[1] Shail Dave et al. "Hardware Acceleration of Sparse and Irregular Tensor Computations of ML Models: A Survey and Insights". *Proceedings of the IEEE* 109.10 (2021), pp. 1706–1752. DOI: 10.1109/JPROC.2021.3098483; Thaha Mohammed et al. "DIESEL: A novel deep learning-based tool for SpMV computations and solving sparse linear equation systems". *The Journal of Supercomputing* 77.6 (2021), pp. 6313–6355; Amuthan A. Ramabathiran and Prabhu Ramachandran. "SPINN: Sparse, Physics-based, and partially Interpretable Neural Networks for PDEs". *Journal of Computational Physics* 445 (2021), p. 110600. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2021.110600. URL: https://www.sciencedirect.com/science/article/pii/S0021999121004952.

[2] Yue Zhao et al. "Bridging the gap between deep learning and sparse matrix format selection". *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 2018, pp. 94–108.

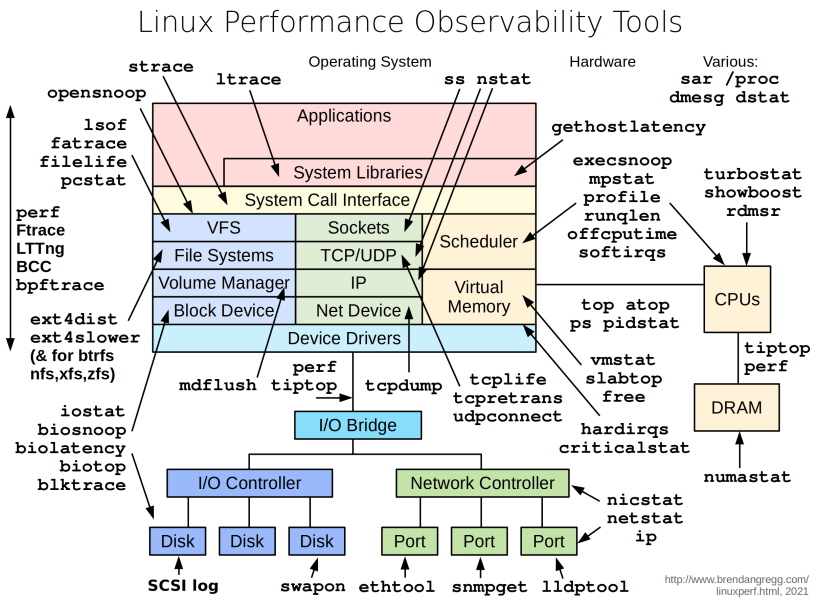


Figure 1 *Linux components and specific tools to monitor each component.*[3]

hardware components when performing sparse computations.

2.1 HARDWARE COUNTERS

Modern multi-core CPUs support an extensive set of hardware counters in order to measure events related to the performance and energy consumption of applications, e.g., Intel CPUs provide hundreds of different events that can be tacked.[4] The counters provided by the performance monitoring unit have a vast range of applicability, from measuring the amount of stalls that originate from accessing different hardware components, to the number of floating-point instructions performed by an application, and cache misses that result from each memory level. Since sparse computations are expected to be limited by the memory capabilities of computing devices, mainly due to the irregular accesses, hardware counters related to the memory requests are especially important to profile these workloads. Moreover, counters related to the floating point instructions should also be considered to assess the utilization of compute units. This task can be performed by using the counters presented in Table 1.

When evaluating the accesses to the memory hierarchy, the total amount of loads and stores, and the misses of each level can provide useful insights regarding cache utilization and efficiency of the memory accesses. While for measuring loads, stores and L1 data misses it is only necessary to access a single counter, in the case of L2 cache data misses and L3 misses, it is necessary to obtain data from two counters. In the case of L2 data misses, it is necessary to compute the difference between all the memory requests from the core that reference a cache line in the last level cache (LLC_REFERENCE) and the code reads that miss L2 cache (L2_RQSTS.CODE_RD_MISS). As for the LLC misses, we rely on the sum of two IMC uncore events,[5] which allow to measure all the access to the main memory, *i.e.*, DRAM reads (CAS_COUNT_RD) and DRAM writes (CAS_COUNT_WR). Other hardware counters provide

[4]R Intel. "and IA-32 Architectures. Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C". *Order Number* (64).

[5]Intel Xeon Processor Scalable Memory Family. "Uncore Performance Monitoring Reference Manual". *Intel Corporation, July* (2017).

measurements regarding the execution stalls that occur due to the misses in specific memory levels (*e.g.* CYCLE_ACTIVITY.STALLS_L1D_MISS) or regarding the cycles where there are outstanding misses. Given that stalls result in low execution performance and efficiency, considering the measurements obtained from these counters may allow to pinpoint the main execution bottlenecks of sparse computations.

Table 1 *Hardware counters for profiling sparse computations.*

Metric	Hardware Counters	Description
Cycles	CPU_CLK_UNHALTED.THREAD	Counts the number of core cycles while the logical processor is not in halt state.
Loads	MEM_INST_RETIRED.ALL_LOADS	Counts the number of retired loads.
Stores	MEM_INST_RETIRED.ALL_STORES	Counts the number of retired stores.
L1 Data Misses	L1D.REPLACEMENT	Counts the data line replacements that occur on L1D cache.
L2 Data Misses	LLC_REFERENCE-L2_RQSTS.CODE_RD_MISS	Number of data requests that miss L2D cache. Corresponds to the difference between every core request that references a cache line in LLC and the L2 code misses.
LLC Misses	CAS_COUNT.RD+CAS_COUNT.WR	Sum between all DRAM reads and all DRAM writes.
L1 Data Stalls	CYC_ACT†.STALLS_L1D_MISS	Stalls that occur due to outstanding loads that miss L1D cache.
L2 Stalls	CYC_ACT†.STALLS_L2_MISS	Stalls that occur due to outstanding loads that miss L2 cache.
L3 Stalls	CYC_ACT†.STALLS_L3_MISS	Stalls that occur due to outstanding loads that miss L3 cache.
Memory Stalls	CYC_ACT†.STALLS_MEM_ANY	Stalls that occur due to outstanding loads in the memory subsystem.
Cycles with misses on L1 Data	CYC_ACT†.CYCLES_L1D_MISS	Cycles while there are outstanding loads that miss L1D cache.
Cycles with misses on L2	CYC_ACT†.CYCLES_L2_MISS	Cycles while there are outstanding loads that miss L2cache.
Cycles with misses on L3	CYC_ACT†.CYCLES_L3_MISS	Cycles while there are outstanding loads that miss L3 cache.
Cycles with outstanding loads	CYC_ACT†.CYCLES_MEM_ANY	Cycles while there are outstanding loads in the memory subsystem.
FP Scalar Double	FP_AI_RET*.SCALAR_DOUBLE	Double-precision scalar FP instructions.
FP Scalar Single	FP_AI_RET*.SCALAR_SINGLE	Single-precision scalar FP instructions.
FP 128-bit SIMD Double	FP_AI_RET*.128B_PACKED_DOUBLE	Double-precision 128-bit packed FP instructions.
FP 128-bit SIMD Single	FP_AI_RET*.128B_PACKED_SINGLE	Single-precision 128-bit packed FP instructions.
FP 256-bit SIMD Double	FP_AI_RET*.256B_PACKED_DOUBLE	Double-precision 256-bit packed FP instructions.
FP 256-bit SIMD Single	FP_AI_RET*.256B_PACKED_SINGLE	Single-precision 256-bit packed FP instructions.
FP 512-bit SIMD Double	FP_AI_RET*.512B_PACKED_DOUBLE	Double-precision 512-bit packed FP instructions.
FP 512-bit SIMD Single	FP_AI_RET*.512B_PACKED_SINGLE	Single-precision 512-bit packed FP instructions.

† – CYCLE_ACTIVITY; * – FP_ARITH_INST_RETIRED

To measure the energy consumption on Intel CPUs, it is necessary to rely on the RAPL interface.[6] As shown in Table 2, RAPL supports different energy domains, each with their specific counter, that can provide different insights about the efficiency of sparse computations. The power plane 0 (PP0) counter allows to measure the energy from the processor cores and LLC, while the counter from the package domain encapsulates the entire processor package. While the

[6]Intel, "and IA-32 Architectures. Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C".

package counter is available in all types of Intel processors, the PP0 domain is usually specific to client platforms. The power plane 1 (PP1) domain targets the integrated processor graphics in-built on the chip package of the client processors, while the DRAM domain considers the energy consumption of the main memory. Finally, the platform energy counter considers the energy of the cores, integrated graphic, system agent and other hardware components, and it is only available in Skylake or more recent architectures. Its availability also depends on the BIOS support. In case it is not supported in the tested hardware, the counter returns the value of zero.

Table 2 *Counters supported by RAPL interface.*

Metric	Hardware Counters	Description
Power Plane 0 Energy	MSR_PP0_ENERGY_STATUS	Energy of processor cores and LLC.
Package Energy	MSR_PKG_ENERGY_STATUS	Energy of entire processor package.
Power Plane 1 Energy	MSR_PP1_ENERGY_STATUS	Energy of integrated processor graphics (only client processors).
DRAM Energy	MSR_DRAM_ENERGY_STATUS	Energy of DRAM (only server processors).
Platform Energy	MSR_PLATFORM_ENERGY_COUNTER	Energy of entire platform (only if BIOS and vendor hardware supports it).

2.2 SOFTWARE EVENTS

Along with hardware events, internal kernel information (vmstat, iostat, disk, network), and process level metrics are also provided by the operating system, e.g., see Figure 1 (taken from[7]). When a (sparse) application runs on a cluster, all of these components can resemble a bottleneck. Hence, by monitoring this information along with the hardware events, spotting performance issues of an application can be possible. Some of these metrics are provided in Table 3.

Table 3 *Some selected metrics from kernel performance metric counters which could relate to sparse application performance.*

Source and Description	Metric	Metric Description
/proc Kernel statistics	kernel.all.intr	Context switches metric from /proc/stat
	kernel.all.pressure.cpu.some.total	Total time processes stalled for CPU resources
	kernel.all.pressure.memory.some.total	Total time processes stalled for memory resources
	kernel.all.pressure.memory.full.total	Total time when all tasks stall on memory resources
	kernel.all.pressure.io.some.total	Total time processes stalled for IO resources
	kernel.percpu.interrupts.PMI	Performance monitoring interrupts for each core
	kernel.percpu.interrupts.TRM	Thermal event interrupts for each core
	kernel.percpu.interrupts.line*	Number of interrupts caused by each IO device
/proc/meminfo System memory statistics	mem.util.used	Used system memory
	mem.util.free	Free system memory
	mem.util.directMap4k	Amount of memory that is directly mapped in 4kB pages
	mem.util.directMap2M	Amount of memory that is directly mapped in 2MB pages
	mem.util.directMap1G	Amount of memory that is directly mapped in 1GB pages
	swap.pagesin	Pages read from swap devices due to demand for physical memory

[7] Brendan Gregg. 2021. URL: https://brendangregg.com/linuxperf.html.

	swap.pagesout	Pages written to swap devices due to demand for physical memory
/proc/meminfo NUMA statistics	mem.numa.util.free	Per-node free memory
	mem.numa.util.used	Per-node used memory
	mem.numa.alloc.hit	Per-node count of times a task wanted alloc on local node and succeeded
	mem.numa.alloc.miss	Per-node count of times a task wanted alloc on local node but got another node
	mem.numa.alloc.local_node	Per-node count of times a process ran on this node and got memory on this node
	mem.numa.alloc.other_node	Per-node count of times a process ran on this node and got memory on another node
/proc/vmstat Virtual memory statistics	mv†.kswapd_low_wmark_hit_quickly	Count of times low watermark reached quickly
	mv†.kswapd_high_wmark_hit_quickly	Count of times high watermark reached quickly
/proc/net/dev Network interface statistics	network.interface.in.bytes	Network recv read bytes per network interface
	network.interface.out.bytes	Network send bytes per network interface
/proc/diskstats Disk statistics	disk.dev.read	Per-disk read operations
	disk.dev.write	Per-disk write operations
	disk.dev.read_merge	Per-disk count of merged read requests
	disk.dev.write_merge	Per-disk count of merged write requests
/proc/<pid>/* Per process statistics	proc.psinfo.ngid	NUMA group identifier
	proc.psinfo.threads	Number of threads
	proc.psinfo.nvctxsw	Number of non-voluntary context switches
	proc.psinfo.processor	Last CPU the process was running on
	proc.psinfo.cmaj_flt	Count of page faults other than reclaims of all exited children
	proc.psinfo.maj_flt	Count of page faults other than reclaims
	proc.io.wchar	write(), writev() and sendfile() send bytes
	proc.io.rchar	read(), readv() and sendfile() receive bytes

† – mem.vmstat

3 PROFILING TOOLS BASED ON HARDWARE COUNTERS

The access to the hardware counters and the energy counters supported by the RAPL interface on Intel devices, it is usually required to have privileged access, thus they cannot be accessed from user space. To overcome this issue and to to ease the adoption of counters for profiling applications and modeling computing systems, monitoring tools, such as, Perf,[8] PAPI,[9] and LIKWID,[10] can be used to easily configure and access the counters. All these tools allow to access both the hardware counters for performance monitoring and the RAPL interface when targeting the energy consumption of a system.

3.1 PERF: PERFORMANCE ANALYSIS TOOLS FOR LINUX

Perf is included in the Linux kernel and relies on a command line interface to configure the hardware counters and performing the measurements. This tool not only supports hardware counters, but also tracepoints, kprobes and uprobes for dynamic tracing. To simply profile the entire execution of an application, *perf stat* command must be used. Moreover, the event accounts can also be recorded through the command *perf record*, which together with the commands *perf*

[8] Arnaldo Carvalho De Melo. "The new linux'perf'tools". *Slides from Linux Kongress*. Vol. 18. 2010, pp. 1–42.

[9] Dan Terpstra et al. "Collecting performance data with PAPI-C". *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[10] J. Treibig, G. Hager, and G. Wellein. "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments". *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*. 2010.

report and *perf annotate* allow to breakdown the measured events by process/function and/or to annotate the application source code with the event counts. This allows to detect which application kernels should be the focus for optimization. Besides these features, perf also allows to print the sampled functions in real time (*perf top*), as well as to perform different multi-threaded micro-benchmarks to evalute the capabilities of CPU devices (*perf bench*).

3.2 PERFORMANCE APPLICATION PROGRAMMING INTERFACE (PAPI)

PAPI profiles applications or specific regions of interest by manually inserting functions/markers in the application code. When using PAPI, the first step is to initialize the PAPI library (*PAPI_library_init*). When targeting multi-thread applications, it is also necessary to initialize the PAPI interface for multiple threads (*PAPI_thread_init*). After initializing PAPI, the second step is to select the counters to be read by each thread/core. This needs to be explicitly performed by the programmer in the application code, through the functions *PAPI_create_eventset* and *PAPI_add_event* to create a event set and add counters to it. The list of events supported by the computing system can be obtained from the executable files *papi_avail* and/or *papi_native_avail*. After adding the events to measure to the event set, the counting of the hardware counters must be started with *PAPI_start*. After starting the counting, to profile specific application kernels, *PAPI_read* calls can be inserted around the region of interest to read the current values of the hardware counters. Finally, at the end of the application code, the counting of the hardware counters must be stopped (*PAPI_stop*) and the envent set cleaned (*PAPI_cleanup_eventset*) and destroyed (*PAPI_destroy_eventset*).

3.3 LIKWID PERFORMANCE TOOLS

LIKWID is another alternative to Perf and PAPI. Similar to Perf, it allows to profile the entire application execution by using a command line interface (*likwid-perfctr*). This interface is also responsible to select and configure the hardware counters. However, LIKWID also includes a mechanism to profile specific regions of the application, by relying on the marker API. The first step is to initialize the LIKWID marker interface (*LIKWID_MARKER_INIT*). In the case of multi-thread applications, it is also necessary to call the function *LIKWID_MARKER_THREADINIT*. After this initialization step, the *LIKWID_MARKER_START* and *LIKWID_MARKER_STOP* must be placed before and after the region of interest. To reduce the overhead when performing the measurements, the user can attribute names to the different regions and register them before starting the measurements (*LIKWID_MARKER_REGISTER*). Finally, at the end of the application the LIKWID marker interface must be closed (LIKWID_MARKER_CLOSE). With this approach, the values obtained for each hardware event and in each core are presented in a command line output.

4 COMMUNICATION PROFILING AND MONITORING TOOLS

Communication is one of the main factor that prevents parallel applications from scaling to large number of cores. In the context of multi-threaded applications, data movement or communication takes place in forms of cache line transfers across multiple cores within or across sockets. Because of the criticality of communication in the performance of an application, in the SparCity project, we leverage communication monitoring tools developed by the KU partner. These tools are

namely ComDetective,[11] ReuseTracker[12] and ComScribe.[13] All these three tools are publicly available on the ParCoreLab git repository: https://github.com/ParCoreLab/ParCoreTools

4.1 COMDETECTIVE: A COMMUNICATION MONITORING TOOL

Inter-thread data movement is a vital performance indicator in multi-core systems. To detect inter-thread communications in multi-threaded codes with low time and memory overheads, the KU partner previously developed ComDetective,[14] a tool that captures inter-thread communications in the forms of communication matrices. The tool employs hardware performance counters (PMUs) to sample memory-access events and uses hardware debug registers to capture communicating pairs of threads. A PMU is a special on-chip hardware in each CPU core that can be used to monitor hardware events, such as memory loads, stores etc, or software events like page faults, while a debug register is a special register that can be programmed to monitor any memory address and trap the next access to that memory address.

ComDetective works by sampling memory accesses in each application thread using PMUs and publishing the sampled memory addresses on a global data structure called BulletinBoard. In addition to publishing sampled addresses to BulletinBoard, each sampling thread also attempts to detect inter-thread communication by comparing the cache lines of its sampled addresses with the cache lines of the addresses published on BulletinBoard. If there is a matching cache line, a communication is detected, otherwise, one address in BulletinBoard posted by another thread is randomly selected and its cache line is monitored by the debug registers in the currently sampling thread to trap a communication.

In addition to detecting communications, ComDetective also differentiates the communications into true sharing and false sharing. This can be useful for the application users if they would like to eliminate or reduce false sharing in their codes by making data structure changes. Furthermore, it also attributes the detected communications to their locations in source code and the data objects involved in the communications.

This tool works in both Intel and AMD architectures. To ensure precision of event sampling, it leverages Processor Event-Based Sampling (PEBS)[15] in Intel and Instruction-Based Sampling (IBS)[16] in AMD. The group's work on extending the tool to ARM-based multicore architectures is in progress.

The KU partner verified the accuracy of ComDetective using several microbenchmarks[17] that have known ground truths. These benchmarks were designed to have known ground truths for total number of communications, ratio of false sharing to true sharing, and distribution of communication volume across communicating thread pairs. The communications captured

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

[12] Muhammad Aditya Sasongko et al. "ReuseTracker: Fast Yet Accurate Multicore Reuse Distance Analyzer". *ACM Trans. Archit. Code Optim.* 19.1 (2021). ISSN: 1544-3566. DOI: 10.1145/3484199. URL: https://doi.org/10.1145/3484199.

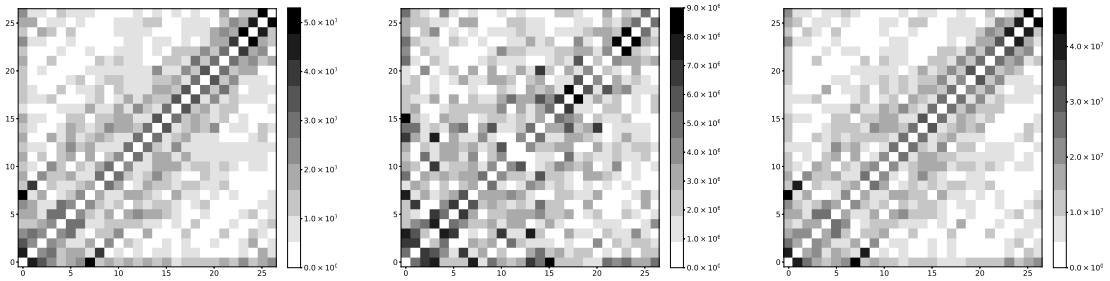
[13] Palwisha Akhtar et al. "ComScribe: Identifying Intra-node GPU Communication". 2021. DOI: 10.1007/978-3-030-71058-3_10.

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

[15] Intel. *Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide*. https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf. 2010.

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

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



(a) *LULESH* **(b)** *LULESH True Sharing* **(c)** *LULESH False Sharing*

Figure 2 *Communication matrices of LULESH (Left to Right: All, True and False Sharing). Darker color indicates more communication.*

by ComDetective from these benchmarks are close to these ground truths. Moreover, the KU partner also evaluated the time and memory overheads of ComDetective by running it on twelve PARSEC[18] and six CORAL benchmarks[19][20][21][22][23][24][25] in an Intel Broadwell machine. Figure 2 shows the communication matrices from LULESH, one of the CORAL benchmarks. Its average overheads are 1.30× for runtime and 1.27× for memory overheads under 500K sampling interval, which are much lower than the overheads of cycle-accurate simulators[26][27][28] and prior-art code instrumentation tools[29][30].[31]

4.2 REUSETRACKER: A REUSE DISTANCE ANALYSIS TOOL

Data locality is another important performance indicator in multi-core machines with multi-level caches. One widely used metric that measures data locality is reuse distance, which calculates the number of unique memory locations accessed between two memory accesses to a particular

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

[19]AMG. *Parallel Algebraic Multigrid Solver*. https://github.com/LLNL/AMG. 2017.

[20]Ulrike Meier Yang. "Parallel Algebraic Multigrid Methods High Performance Preconditioner". *Numerical Solution of Partial Differential Equations on Parallel Computers, LNCS 51* (2006), pp. 209–233.

[21]Quicksilver. *A proxy app for the Monte Carlo Transport Code, Mercury*. https://github.com/LLNL/Quicksilver.

[22]PENNANT. *Unstructured mesh hydrodynamics for advanced architectures*. https://github.com/lanl/PENNANT. 2016.

[23]miniFE. *MiniFE Finite Element Mini-Application*. https://github.com/Mantevo/miniFE.

[24]VPIC. *Vector Particle-In-Cell (VPIC) Project*. https://github.com/lanl/vpic.

[25]LULESH 2.0. *Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)*. https://github.com/LLNL/LULESH.

[26]Nick Barrow-Williams, Christian Fensch, and Simon Moore. "A communication characterisation of Splash-2 and Parsec". *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.* 2009.

[27]P.S. Magnusson et al. "Simics: A full system simulation platform". *Computer* 35.2 (2002), pp. 50–58.

[28]Eduardo Henrique Molina da Cruz et al. "Using Memory Access Traces to Map Threads and Data on Hierarchical Multi-core Platforms". *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. 2011.

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

[30]Arya Mazaheri, Felix Wolf, and Ali Jannesari. "Characterizing Loop-Level Communication Patterns in Shared Memory Applications". *Proceedings of the 2015 44th International Conference on Parallel Processing*. Beijing, China, 2015. DOI: 10.1109/ICPP.2015.85.

[31]Arya Mazaheri, Felix Wolf, and Ali Jannesari. "Unveiling Thread Communication Bottlenecks Using Hardware-Independent Metrics". *Proceedings of the 47th International Conference on Parallel Processing*. Eugene, OR, USA: ACM, 2018, 6:1–6:10. DOI: 10.1145/3225058.3225142. URL: http://doi.acm.org/10.1145/3225058.3225142.

memory location. To profile reuse distance in multi-threaded code with low runtime and memory overheads, the KU partner developed ReuseTracker.[32] This profiling tool leverages PMUs to sample memory accesses and uses debug registers to detect either a *reuse* or a cache line invalidation of the sampled memory location. ReuseTracker works in Intel by leveraging PEBS and in AMD by leveraging IBS. Similar to ComDetective, its extension to ARM-based multicore architectures is in progress.

ReuseTracker employs two different algorithms that profile reuse distance in private caches and shared caches, respectively. To profile reuse distance in private caches, each thread that encounters a PMU sample arms a debug register in every CPU core in the machine. If the next debug register trap occurs in the same CPU core as the PMU sample, a reuse in private cache is detected, otherwise, if the next trap happens in another core, a cache line invalidation at private cache level is detected. When a reuse is detected, the number of memory accesses between the PMU sample and the debug register trap is recorded into a time reuse distance histogram, which is then converted into stack reuse distance histogram using the method in.[33] In the algorithm that profiles reuse distance in shared caches, each thread that faces a PMU sample arms a debug register in every other core in the machine. A reuse in the same shared cache is detected if the next trap happens in another core that shares the same socket, and a cache line invalidation at shared cache level is detected when the next trap occurs in another core located in another socket. When a reuse at shared cache level is detected, the number of memory accesses in all cores that share the same socket is recorded in the time reuse distance histogram.

To evaluate the accuracy of ReuseTracker, the KU partner developed a microbenchmark that can be configured to generate a variety of reuse distance patterns. The private cache profiling algorithm has been evaluated on this microbenchmark, and its accuracy is 92% in an Intel Skylake machine under 100K sampling interval. The overheads of this tool have also been evaluated by running it on ten PARSEC benchmarks with 2.9× runtime and 2.8× memory overheads under the same sampling interval, which are much lower than the overheads of the other open source reuse distance analysis tools.[34]

4.3 COMSCRIBE: INTER-GPU COMMUNICATION DETECTION TOOL

Communication monitoring among GPUs can help reason about scalability issues and performance divergence between different implementations of the same application. ComScribe[35] is a tool that can identify communication among all GPU-GPU and CPU-GPU pairs in a single-node multi-GPU system. It can monitor data movement induced by both Peer-to-Peer (P2P) primitives of CUDA and collective communication primitives of NVIDIA's Collective Communication Library (NCCL). It employs the NVIDIA's profiling tool nvprof and Unix dynamic linker utility to monitor P2P communication and collective communication respectively to gather the necessary information. Then, the collected information is processed to quantify communication among GPUs and generate the communication matrices. In the SparCity project, we plan to leverage this tool to monitor inter-GPU communication in multi-GPU applications with this tool.

[32] Sasongko et al., "ReuseTracker: Fast Yet Accurate Multicore Reuse Distance Analyzer".

[33] Xipeng Shen, Jonathan Shaw, and Brian Meeker. "Accurate Approximation of Locality from Time Distance Histograms". 2006.

[34] Xiaoya Xiang et al. "HOTL: A Higher Order Theory of Locality". *SIGARCH Comput. Archit. News* 41.1 (2013), pp. 343–356. ISSN: 0163-5964. DOI: 10.1145/2490301.2451153. URL: https://doi.org/10.1145/2490301.2451153; *dcompiler/loca: Program locality analysis tools*. https://github.com/dcompiler/loca.

[35] Akhtar et al., "ComScribe: Identifying Intra-node GPU Communication".

5 SYSTEM-LEVEL MONITORING TOOLS

In order to systematically collect and store information from performance metric sources, several monitoring tools are developed and widely used in the literature. These tools aim to provide a wider picture on the system performance via monitoring multiple components of and building relations among them. These systems are used to facilitate intelligent job placement, run-time workload partitioning/adaptation and HPC hardware procurement planning.[36] Some of these tools are: LDMS,[37] Performance Co-Pilot,[38] Ganglia,[39] Nagios, HPC-Toolkit[40] and PerfAugur.[41] Among them Ganglia is proven to be scalable up to 2000 nodes but is used for general system monitoring, requires considerable number of installation dependencies, targets larger collection intervals (10s of seconds to 10s of minutes) and uses an aging tool for storage. Nagios also targets larger collection intervals (10s of seconds to 10s of minutes) and mainly used for failure alerts. PerfAugur is used to trace the cause of a system anomaly by finding common attributes that predicate an anomaly.[42] HPCToolkit is a suite of tools which can provide accurate measurements of program performance on a wide variety of systems from single host computers to large clusters. However it involves a binary analysis and re-compilation of the target code. LDMS and Performance Co-Pilot are metric collection, transport and storage systems which can be configured to sample every performance metric counter on hardware and kernel including RAPL, PAPI and perf interfaces. Moreover they support frequent and variable sampling rate on these performance metrics with negligible overhead and without requirement of recompile or source code instrumentation. This enables real time monitoring of HPC systems in cluster level, node level and process level in order to provide multiple-aspect insight of application performance.

5.1 LIGHTWEIGHT DISTRIBUTED METRIC SERVICE (LDMS)

LDMS is part of OVIS, a suite of HPC monitoring, analysis and feedback tools which is jointly developed by Sandia National Laboratories and Open Grid Computing. LDMS is based on daemons called `ldmsd` which can run on either sampler or aggregator modes. A sampler `ldmsd` daemon is created by running and configuring sampling plugins which sample PMUs. Each sampling plugin combines a specific set of data into a single metric set. An aggregator `ldmsd` daemon is created by running and `ldsmd` and configuring aggregator plugings. Each aggregator collect metric sets from samplers and/or other aggregators. Higher level aggregators can listen many lower level aggregators, aggregate and streams data into storage. LDMS can store sampled performance data on CSV files, D-SOS and InfluxDB. Increasing with number of sampled metrics, LDMS causes very little overhead on the system performance. It causes $\approx$ 0.01% CPU utilization,

[36] James M. Brandt, Thomas Tucker, and Ann C. Gentile. *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).

[37] Anthony Michael Agelastos et al. *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).

[38] Red-Hat. URL: https://pcp.io/.

[39] Ganglia. *Ganglia monitoring system.* URL: http://ganglia.sourceforge.net/.

[40] L. Adhianto et al. "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.

[41] Sudip Roy et al. "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.

[42] Agelastos et al., *The Lightweight Distributed Metric Service*.

$<$ 2MB memory, $<$ 4MB filesystem and 4KB network overhead for $\approx$ 200 metrics @1 second intervals.[43]

Although it has been widely used in the literature, and it works well under certain circumstances, LDMS is mostly used by a strictly related group, lacks documentation and still under development. Therefore it's hard to deploy, develop and maintain.

5.2 PERFORMANCE CO-PILOT

Performance Co-Pilot, which was initially released at 1995 by SCI and currently being developed by Red-Hat is a system performance analysis toolkit. PCP contains two types of components: PCP collectors and PCP monitors. PCP collectors are responsible for collecting and extracting performance data from various sources. This sources could be PMCs, PMUs or application performance logs. PCP collectors consists of two components; Performance Metrics Domain Agent (PMDA) and Performance Metric Collector Daemon (PMCD). PMDAs connects to performance sources and sample their values, then reports these values to PMCD. There currently 75 PMDAs available and apart from existing PMDAs, new PMDAs could be developed to connect any wanted performance metrics source using PMAPI library. To be able to report metrics from a host machine, there must be a PMCD which listen and control all PMDAs and answer requests of monitoring applications. Monitoring tools are used to display, manipulate and store performance metrics extracted from PMCDs. PMCDs can collect performance metrics from remote hosts or answer to remote monitoring tools in a distributed setting. Some of the PCP monitoring tools are; PMIE, an inference engine which could be used to automate system management tasks via predicate-action rules. PMLOGGER, archive manager which enables subsets of collected performance metrics to be replayed. PMCHART, a visualization tool which can generate on the fly charts from collected metrics. PMREP, a performance metrics reporter with highly customizable output format. PCP can export collected metrics to several databases such as; Elasticsearch, Graphite, InfluxDB, Redis and Apache Spark.

5.3 GRAFANA: OPEN SOURCE VISUALIZATION TOOL

Grafana is an open source visualization tool which provides dynamic dashboards, ad-hoc queries and alerting functions on time-series data. Since it's initial release at 2014, Grafana quickly become industry standard and reached 10M+ global users recently. Due to it's massive userbase, Grafana provides support for every popular database and provides a wide variety of visualization methods. Due to it's strong recognition and flexibility, Grafana is chosen as Digital SuperTwin front-end interface during development in order to prove usability of design concepts for Digital SuperTwin's own interface.

5.4 MEASUREMENT OVERHEAD

The overhead for a default configuration of Performance Co-Pilot is reported in Table 4. Since the default configurations of monitoring tools focused on overall system health and component state, main effort is put on reconfiguration of sampler processes to sample performance centric metricsets and development of custom samplers. On top of that, to be able to implement and test customized solutions for monitoring and to report reproducible results, a freely available research medium is required. To this end, virtual clusters using docker containers as compute node, which can perform MPI communications and managed by SLURM is realized. But since this framework is still under development, results acquired from this setting, along with the custom samplers and

[43] Agelastos et al., *The Lightweight Distributed Metric Service*.

Figure 3 *An example Grafana dashboard showing performance metrics collected by PCP and indexed with InfluxDB on a single host. Exhibiting node-level monitoring usage.*

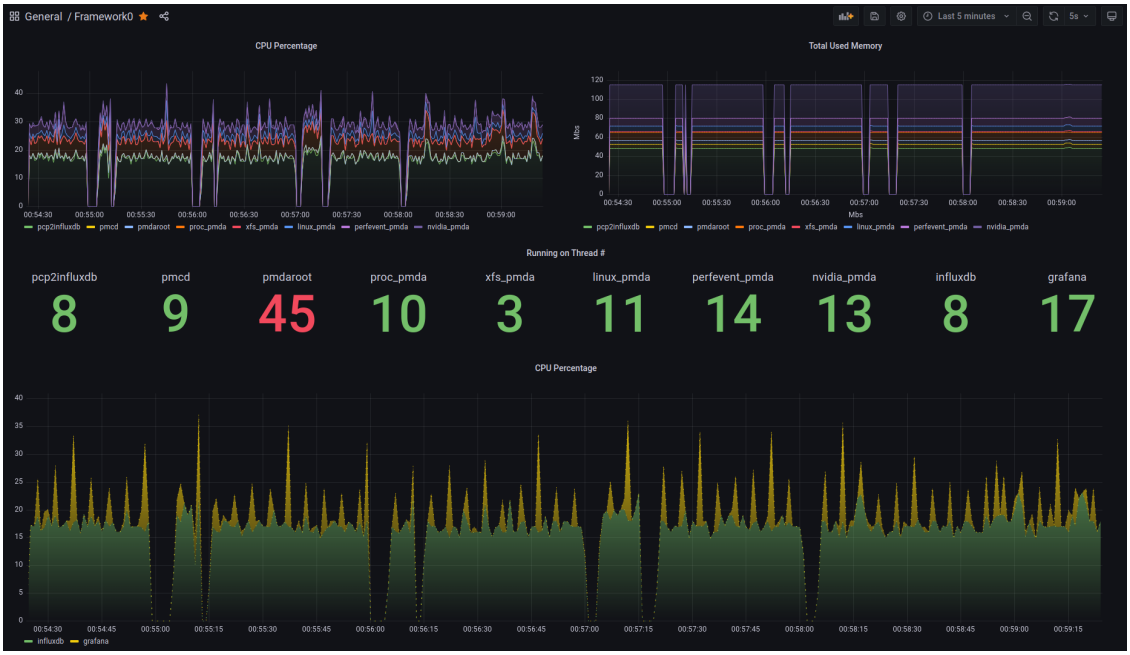


Figure 4 *An example Grafana dashboard showing statistics for processes used for creating Table 3. Exhibiting process-level monitoring usage.*

implementation of monitoring tool on physical clusters will be reported at the next deliverable.

Sampler / Reporter	pcp2influxdb	pmcd	pmdaroot	proc_pmda	xfs_pmda	linux_pmda	perfevent_pmda
CPU Percentage	19.1	0.1	0.1	7.3	0.09	1.6	0.1
Resident Set Size (MBs)	46.7	4.4	3.8	6.3	3.6	5.7	7.6

Table 4 *Overhead of Performance Co-Pilot agents on a single host machine with default installation. Among 2857 metrics collected by samplers, 107 are reported to InfluxDB. Average CPU usage and RSS of proccesses for one hour is reported.*

6 DYNAMIC INSTRUMENTATION AND CACHE PARTITIONING TOOLS

Hardware counters are extremely useful to extract metrics to understand the behavior of applications in current computing systems. However, they do not encapsulate all the information about the specifics of an application. For example, while the hardware counters on Intel CPUs allow to count the number of retired loads and stores, they do not provide information related to the size each memory access. Similarly, while there are counters for floating-point operations, no information can be inferred for integer operations. Both these specifics can be crucial when understanding the bottlenecks of sparse computations, as well as to achieve accurate application characterization.[44] To overcome these issues, dynamic instruction tools can be used to dynamically evaluate the instructions executed by the application to obtain profile regarding the type of instructions performed.

6.1 INTEL SDE, PIN AND GTPIN

For Intel CPUs, one of the most known tools is the Intel Software Development Emulator (SDE).[45] Intel SDE is a emulation tool built on top of Intel Pin,[46] which allows to debug programs using instructions before their release. Intel SDE also contains a histogram tool, which can be used to dynamically count the instructions executed by an application. This can be divided according to their ISA extension, type, data size, etc. Since the instrumentation is performed while Intel SDE discovers the execution path of the application, the results obtained are more reliable than the ones obtained by statically analyse the assembly code. Intel SDE also supports a marker API to only obtain the dynamic trace of specific applications kernels, by placing the *__SSC_MARK* macros around the code. Given that Intel SDE is able to categorize memory and compute instructions in different categories, *e.g.*, according to their size and data type, this tool can also be fundamental when applying specific modeling approaches, such as roofline modeling methodologies.

The binary instrumentation can also be performed on Intel GPUs by relying on GTPin,[47] an adaptation of the Intel Pin to Intel GPU architectures. GTPin features a binary instrumentation engine for the execution units in Intel GPUs, a set of sample tools and an API for developing additional analysis tools. GTPin instrumentation can be used to gather data for workloads in compute and graphics applications, enabling fast analysis of code running in the GPU. GTPin capabilities are used in various profiling tools, such as Intel Profiling Tools Interface for GPU,[48]

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

[45] Moshe Bach et al. "Analyzing parallel programs with pin". *Computer* 43.3 (2010), pp. 34–41.

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

[47] Intel Corporation. *GTPin*. https://software.intel.com/sites/landingpage/gtpin/index.html. [Online; visited March-2022].

[48] Intel Corporation. *Profiling Tools Interfaces for GPU (PTI for GPU)*. https://github.com/intel/pti-gpu. [Online; visited March-2022].

Intel Advisor and Intel VTune. It is supported in both integrated and discrete Intel GPUs.

6.2 Reuse distance analysis and cache partitioning for the ARM A64FX CPU

Due to high costs incurred when accessing main memory, data locality is an important consideration when optimizing numerical kernels. Caches keep small amounts of data close to the processor's compute units, and thus have the potential to speed up computation and reduce energy consumption. Programmers may exploit caches by employing optimization techniques such as blocking or tiling. To ease programmability, the cache is automatically managed by hardware and its contents are under implicit but not explicit control of a programmer. The implicit control a programmer may enact is imprecise and has limitations, because CPU hardware is complex, several subsystems may be interacting unpredictably and the precise functionality of these subsystems is often not documented.

Cache partitioning is an approach to give programmers significantly more explicit control over cache functionality. It allows the available cache space to be divided into a configurable number of partitions. Data structures (corresponding to one or more address ranges) may be assigned to a dedicated partition. This is especially promising in situations where data is repeatedly accessed and should be retained in cache but is evicted by intervening unrelated data accesses due to the cache's eviction policy (typically a variant of LRU - least recently used). In a partitioned cache each partition is managed with a separate eviction policy and by assigning the reusable data its own partition it can be shielded from evection by the unrelated data accesses.

A feature for cache partitioning as described above is not available in most mainstream CPUs but has been available in the SPARC64VIIIfx CPU[49] developed by Fujitsu for the K computer system and has also been introduced in the Fujitsu A64FX CPU powering the Fugaku system (#1 in the Top 500 list as of November 2021). In the A64FX CPU the feature is called *instruction-based way partitioning* and referred to as the *sector cache*. It works by dividing the "ways" dimension of the 4-way set-associative L1 data cache or 16-way set-associative L2 cache into two or more sectors. An extension to the Fujitsu compiler suite can be used to specify which data structures should be assigned to a dedicated sector.

While a potentially powerful feature, unless a developer knows an application's data reuse pattern in great detail, it will be a challenge to determine whether cache partitioning may be a profitable option and which data structures to isolate in dedicated sectors. To help developers with these questions, a tool was developed at LMU Munich[50] which can be used to derive hints about a sector cache configuration and identify candidate data structures to assign a dedicated sector. The tool works by monitoring and analysing the accessed memory locations during execution to compute reuse distance histograms under various hypothetical partitioning scenarios.

The tool is based on dynamic binary instrumentation using Intel Pin[51] and captures memory accesses during execution. Based on the stream of accessed memory locations, reuse distance histograms are computed. The reuse distance of a memory access for a particular location is the number of distinct memory locations referenced since the last access to this same location. The reuse distances can then be aggregated per data structure into reuse distance histograms, and these histograms can be used to derive hints about which data structures may benefit from a placement in a dedicated sector. If the reuse distances of a data structure are predominantely

[49] Toshio Yoshida et al. "SPARC64 VIIIfx: CPU for the K computer". *Fujitsu Sci. Tech. J* 48.3 (2012), pp. 274–279.

[50] Sergej Breiter. "Evaluating Sector Caches in High-Performance Computing". Master Thesis. Ludwig-Maximilians-Universität München, 2022.

[51] Luk et al., "Pin: building customized program analysis tools with dynamic instrumentation".

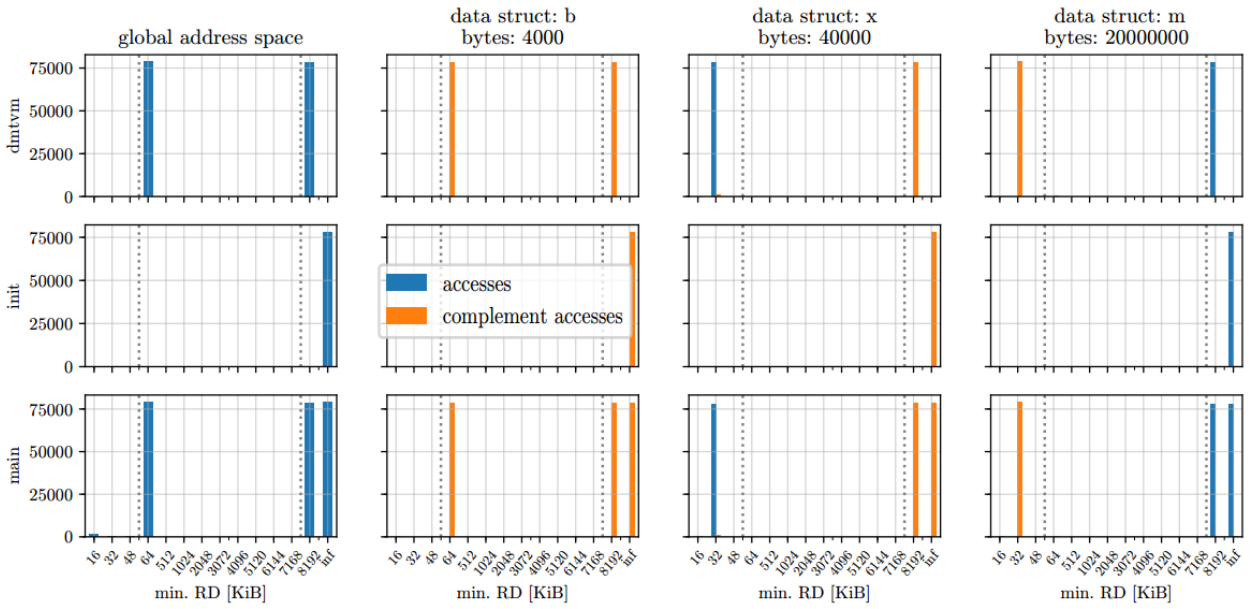


Figure 5 *Reuse distance histograms of the DMTVM data structure.*

region	data struct	cache level	nways	misses sc	misses nosc	reduction [%]
main	m	1	1	156824	236875	33.79
main	m	2	2	156583	157742	0.73
dmtvm	m	1	1	78447	157277	50.12
dmtvm	m	2	2	78237	78450	0.27

Figure 6 *The tool's output of the sector cache configuration for DMTVM. The number of L1D cache misses can be reduced significantly by isolating m in the function dmtvm in a partition with quota of one cache way.*

larger than the capacity of a cache, LRU implies that data is evicted just before being reused. By using two partitions, the reuse distance histogram is split into two histograms in which the main amount of accesses may now be smaller than the partition's size, thereby providing a benefit.

As an example for how the tool can be used, Fig.5 shows the output for the simple dmtvm (Dense Matrix Transposed Vector Multiplication) benchmark application. The source code consists of the *main* function, the *init* function and the *dmtvm* kernel itself, which is shown in Fig 7. The multiplication operates on the input matrix m, the multiplicand vector b and stores the result in the vector x. x is reused in every row of the multiplication, m is streamed and never reused and b can be ignored regarding memory accesses, because only one element of b is loaded per row. m interferes with the reuse of x and doubles the reuse distances of accesses to x, because equal amounts of data from x and m are required during the calculation of each row.

In the reuse distance histograms shown in Fig. 5, the number of the double precision floating-point matrix rows is set to 500 and the number of columns is set to 5000. This leads to the sizes of 4KB for b, 40KB for x and 20MB for m. Thus, x can fit in 3 ways of the L1D cache (total size 64 KB) and m does not fit in the L2 cache (total size 8 MB). In the main function (third row of histograms), accesses to the global address space (first column) have only reuse distances higher than the L1D capacity. The accesses with reuse distance 8 are attributable to the init function (second row) and are compulsory misses. Accesses with other reuse distances occur during the

```
#pragma scache_isolate_way L1=1
#pragma scache_isolate_assign m
#pragma omp for
  for ( int i = 0 ; i < nrow ; ++i ) {
    for ( int j = 0 ; j < ncol ; ++j ) {
      x[j] += m[i * ncol + j] * b[i] ;
}}
```

Figure 7 *Dense Matrix Transposed Vector Multiplication (dmtvm) source code and example sector cache configuration using Fujitsu compiler directives, called Optimization Control Lines (OCLs).*

multiplication (first row). Isolating b (second column) does not change the reuse distances and can be omitted as option. Isolating x (third column) shifts the reuse distances of accesses to x (blue) below the L1D capacity. Other accesses (orange) are not negatively affected. This is exactly as expected, because x is reused frequently and 40KB large. The isolation of m (fourth column) shifts the reuse distances of accesses to its complementary data structure below the L1D cache capacity.

The output of the analysis is shown in Fig. 6. The optimal sector cache configuration is the isolation of m in the smallest possible sector of each cache level which halves the L1D misses when applied. This is again as expected, because half of the accesses are made to x which can be reused when it is isolated from the interfering accesses to m. Isolating m in a small sector slightly decreases the cache misses compared to isolating x in a sector of appropriate size, because it has a minor positive affect on the reuse of b.

The predicted configuration has been tested on a Fujitsu A64FX system installed at the Leibniz Supercomputing Center (LRZ) and the measurements of the L1D and L2 misses on the A64FX CPU have been found to be in line with the predictions of the tool. The measured L2 misses in the dmtvm function remained unchanged at around 80000 when the sector cache was applied. On the other hand, the measured L1D misses went down from 142000 to 81000. Similar results can be observed for the L2 cache partitioning.

7 VENDOR-SPECIFIC FRAMEWORKS FOR APPLICATION ANALYSIS

Another approach to evaluate the execution of applications in current computing systems is the utilization of frameworks provided by the different vendors. For example, for Intel CPUs and GPUs, Intel Advisor can be used to obtain metrics related to the instruction mix, memory accesses and execution time, together with the Cache-Aware Roofline Model,[52] which is proposed by the INESC-ID partner and fully automated in this framework. Intel Vtune also encapsulates the Top-Down method,[53] which provides an extensive evaluation of the application execution to identify which hardware resources are limiting application performance. For NVIDIA GPUs, similar information can be obtained from the NVIDIA Visual Profile,[54] while the PopVision and Poplar tools can be used to analyze applications running in Graphcore IPU systems.

[52] Marques et al., "Application-driven cache-aware roofline model".

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

[54] NVIDIA Corporation. *Profiler User's Guide*. NVIDIA Corporation. 2022. URL: https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

7.1 INTEL VTUNE

Intel VTune Profiler is a framework developed by Intel that provides different methods for application analysis on CPUs, GPUs and FPGAs. With the information provided by Intel VTune, it is possible to optimize application performance on diverse systems. In this deliverable, we focus on two main approaches included in Intel VTune, namely: the Microarchitectual General Exploration[55] for Intel CPU applications, and the GPU offload analysis for workloads deployed on Intel GPUs. Microarchitectual General Exploration can be used to identify bottlenecks related to the different hardware components on the CPU architecture/system, by using hardware counters to pinpoint the performance issues. The GPU offload analysis supports DPC++ and OpenCL kernels and provides several performance metrics that allow to verify the parallelism efficiency of the application.

7.1.1 MICROARCHITECTUAL GENERAL EXPLORATION FOR INTEL CPUS

In a simplistic view, execution bottlenecks on current computing devices can be either attributed to the maximum performance limits of specific units or due to micro-architectural stalls. The Top-Down analysis[56] focuses on these aspects, identifying if in a given cycle the pipeline slots are empty or filled with a micro-operation (µop). This modeling method is integrated in Intel VTune, under the Microarchitectual Exploration analysis.[57] This analysis provides several metrics in respect to different parts of the micro-architecture and are derived by relying on the performance monitoring unit in-built in Intel processors. For this analysis, the events are collected in the Event-Based Sampling (EBS) mode. With EBS, each hardware counter is configured to provide a sample-after-value and once this value is reached, the counter increments, an interrupt is fired and the data collected. After data collection, the counter is reset and the process is repeated. Moreover, in recent micro-architectures, the Microarchitectual Exploration analysis collects around 60 hardware events. Since there is a limited number of hardware counters to use simultaneously, Intel VTune relies on multiplexing to obtain all the measurements in a single profiling run, which can reduce the accuracy of the analysis, especially for short lived kernels. To overcome this issue, multiple runs can be performed at the cost of higher profiling overhead. While the sampling methodology of Microarchitectual Exploration analysis allows it to characterize different hotspots of an application, by attributing the different samples to the loops and functions of an application, this analysis can also be applied to specific regions of interest through the use of the marker API, by placing `__itt_resume()` and `__itt_pause()` calls around the code region to profile.

As it can be observed in Figure 8, the Top-Down method integrated in Intel VTune is based on a hierarchical tree organization. In the scenario that the pipeline slot is empty, it is necessary to identify if the stall occurs due to the instruction fetching and decoding (frontend) or due to the availability of the data operands (backend). On the other hand, in case the pipeline slot contains a µop, the bottleneck can be either due to the retiring limits of the architecture (usually 4 µop on Intel processors) or due to the branch prediction engine (bad speculation). Depending on the identified bottleneck, different insights can be derived. For example, when an application is limited by the frontend, the user must focus on improving the code layout or reduce the code memory footprint. Issues related to bad speculation indicate that the code must be inspected to avoid indirect branches or error conditions that result in machine clears. Since the branch

[55] Intel Corporation. *Understanding How General Exploration Works in Intel® VTune™ Amplifier*. https://www.intel.com/content/www/us/en/developer/articles/technical/understanding-how-general-exploration-works-in-intel-vtune-amplifier-xe.html. [Online; visited March-2022].

[56] Yasin, "A top-down method for performance analysis and counters architecture".

[57] Intel Corporation, *Understanding How General Exploration Works in Intel® VTune™ Amplifier*.

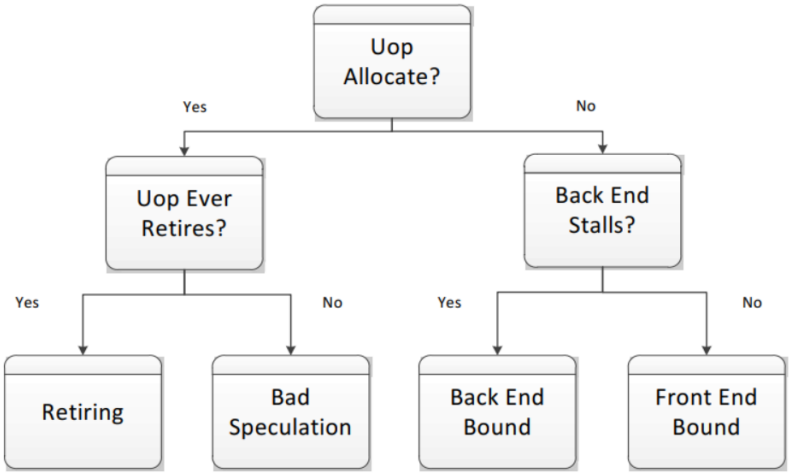


Figure 8 *Hierarchical organization of Top-Down analysis*[58]

prediction engine also influences the fetched instructions, reducing the bad speculation issues may also decrease the impact of the frontend bottlenecks. When an application is limited by retiring, it means that most of its µops are being executed at the maximum throughput of the micro-architecture. However, this does not mean that the performance of the application is maximized. For example, if a scalar application is entirely limited by retiring, vectorization techniques must be employed to further improve performance. Finally, backend issues can be either a result of core bound stalls or memory bound stalls. Core bound stalls can occur when certain execution units are not fully utilized or when there is competition for discrete execution units. Memory bound stalls result from accesses to the memory hierarchy, such as cache misses or latency-related issues, which is expected to be the main source of bottlenecks when profiling sparse computations. The memory bound stalls are further split according to each memory level contained in the memory hierarchy: L1 bound, L2 bound, L3 bound and DRAM bound. From each of the metrics it is also possible to identify if the performance issues result from latency or bandwidth, as well as other inefficiencies, such as false sharing at the L3 cache when executing in multiple-threads.

7.1.2 GPU OFFLOAD ANALYSIS

The primary analysis type in VTune used for GPU applications is the GPU Offload analysis. It allows to explore the application execution in the current platform consisting on CPU and GPU, correlate the activity in both devices, and identify if the application's performance is bounded by the CPU or GPU. For compute applications using DPC++ or OpenCL kernels, it allows to explore the efficiency of the application when using GPU hardware.

In the analysis configuration options, the user can customize the analysis, selecting the options to trace GPU programming APIs, collecting host stacks, analyze CPU-GPU bandwidth and show GPU performance insights. These allow, respectively, to analyze DPC++/OpenCL kernels, analyze call stacks from the CPU, analyze the data transfers between CPU and GPU to obtain the bandwidth, and get metrics to assess the efficiency of the GPU usage. The metrics available are the cycles where the EUs are active, stalled or idle (presented in percentage of total cycles), the EU thread occupancy, *i.e.*, the cycles when a thread is scheduled, and the total number of compute threads started. Upon completion of the analysis, a summary view is presented, showing the total time spent on compute tasks, and the execution time per task. From this, the user can

assess the balance between GPU execution and data transfers. This information can also be taken from the metrics Host-to-device and Device-to-host transfer. Further results can be seen in the Graphics tab, with a breakdown of the timings of all computing tasks, split between allocation, data transfers and execution. The data transfer sizes and instance counts are also presented. The user can easily identify computing tasks that need optimizing by locating the highlighted entries, that can have issues such as larger data transfer time than execution time. A timeline view of events is also available, showing when the CPU and GPU are busy at each instant of execution.

Following the GPU Offload analysis, the GPU Hotspots analysis in VTune allows to characterize the use of the GPU based on hardware metrics. It aims to identify performance issues in GPU workloads, to understand if they are due to inefficient memory accesses by the kernel, low occupancy of the GPU or wrong configuration of GPU threads and thread blocks. It offers deeper insights into the performance issues of GPU kernels relatively to the GPU Offload analysis.

The summary of this analysis contains data on the GPU time, EU cycles, occupancy, peak occupancy and the most active compute tasks. The amount of cycles where the EUs are active and the thread occupancy can provide insights on the application efficiency. In addition, the main cause of the performance degradation is also presented, *e.g.*, performance bound by GPU L3 Cache Bandwidth. Following this, the main kernels are shown in the "Hottest GPU Computing Tasks with Low Occupancy", and tips on how to improve these metrics are displayed. Additionally, the percentage of the elapsed time with FPU utilization is shown, along with a histogram of the bandwidth utilization. Depending on the chosen options for the analysis, the Graphics tab will display CPU and GPU usage data for each thread and show a list of various hardware metrics for the GPU, such as the local and global work sizes, total and average time, instruction mix, the throughput of compute operations, SIMD width of a compute task, number of cycles spent, average latency, etc. A memory hierarchy diagram is also available, featuring an overview of the architecture, with the bandwidths achieved in different memory levels and usage metrics indicated in each component.

7.2 INTEL ADVISOR

Intel Advisor is a tool for application analysis and development. Through a set of different analysis, the programmer can use Intel Advisor to assist on developing high-performance code for CPU and GPU. Intel Advisor is available in a GUI or command line interface.

The analysis available for CPU code are focused on achieving efficient parallelization, vectorization and memory accesses. For GPU code, the programmer can identify parts of the code that can be offloaded, and improve memory accesses and compute operations. In both CPU and GPU code, a Roofline analysis can be used to get insight into the application performance relatively to the hardware limits imposed by the machine, through the use of the Cache-Aware Roofline Model. All available analysis can be done from the Advisor GUI, by creating an Advisor project, selecting the application binary, the analysis type and all parameters. Alternatively, the command line interface can be used, and the results generated with it can also be visualized in the GUI or exported to other file formats. In order to avoid unnecessary overheads in analysis, Intel Advisor allows the application code to be instrumented in order to restrict the analysis to specific areas of interest. This can be done by simply using the Instrumentation and Threading Technology API, particularly by enveloping the code with the `__itt_resume()` and `__itt_pause()` calls.

When profiling a CPU application, the Survey analysis provides an overview of the performance of the different loops and functions calls in the application, and obtaining metrics, such as the elapsed time, the vectorization efficiency and speedup relatively to scalar execution. More detailed data can be obtained by performing a Trip Counts analysis, which counts the number of

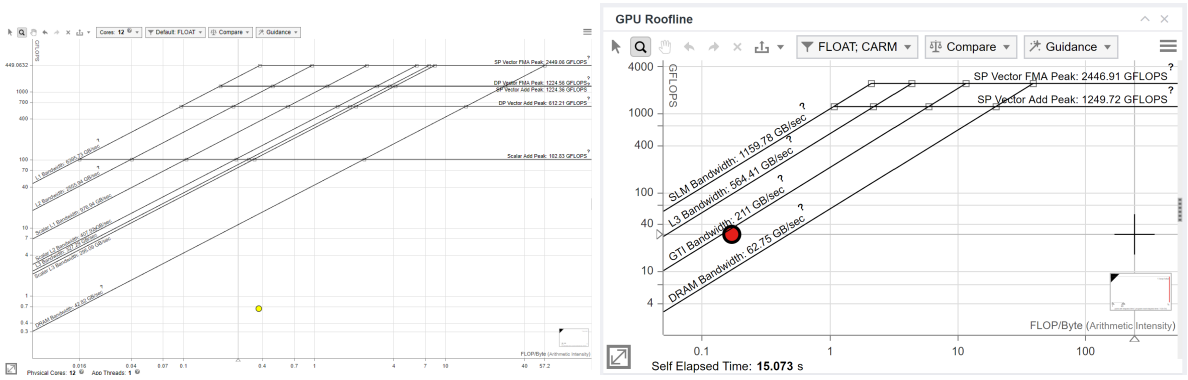


Figure 9 *CPU (left) and GPU (right) Rooflines on Intel Advisor.*

times each loop is executed. This analysis allows to use the FLOP option to obtain the data on floating point and integer operations. Using the results from the Survey and Trip Counts analysis with FLOP, Intel Advisor can create a Roofline chart.

As can be observed in Figure 9, the CPU Roofline analysis allows to visualize the Roofline model for the host CPU. The memory roofs are obtained for each memory level, *i.e.*, L1, L2, L3 and DRAM bandwidths. The compute roofs are obtained for both the vectorized and scalar peak throughputs for the ADD and FMA operations, and for both single and double precision. The user can select to view the Roofline model for any number of available cores in the machine, and for either integer or floating point operations. The application loops and functions are represented as dots in the chart, with varying sizes and colors depending on their importance. Depending on the arithmetic intensity of a given loop, this dot will be located in one of two different regions - the memory bound region, when strictly below the slanted memory roofs, or the compute bound region, when only below the horizontal compute roofs. When hovering the cursor on a dot, it is possible to view a dotted vertical line that shows the possible performance that the loop may obtain. Depending on which roofs the line intersects, the user can have a notion of the main bounds to performance, usually the roofs immediately above, and thus know which type of optimizations are necessary to achieve better performance. When considering sparse computations, for instance, it can be expected that a dot representing the main loop computing a sparse matrix and vector product to be located in the memory bound region, which indicates a need to organize data accesses in order to achieve better memory bandwidth, or block data in order to achieve better locality. Compute related optimizations, on the other hand, can consist on targeting the vector units available in the CPU, for example. In the case of codes already vectorized, the vectorization efficiency can be an improvement point.

For GPU applications, the Survey analysis obtains the timings for the different GPU kernels instead of loops and functions. The data on floating point and integer operations is also obtained through a Trip Counts with FLOP analysis, similarly to its CPU counterpart. When visualizing the GPU Roofline results, Advisor also presents metrics such as the utilization of GPU resources, data transfer times between CPU and GPU, or the percentage of time the GPU execution units are active, stalled or idle. In the GPU Roofline view, shown in Figure 9, depending on the target GPU, the memory roofs for the L3 cache, the Shared Local Memory, Graphics Technology Interface, and DRAM bandwidth are represented. For the compute roofs, the vector ADD peak throughputs are available for the available data types bit-width, 8, 16, 32, and 64. The developer can select to view the Roofline for integer or floating point data. Similarly to the CPU, the dots representing the kernels can be located in the memory bound or compute bound regions. When compute bound, as the GPU is based on vector processing units, the optimizations consist usually on

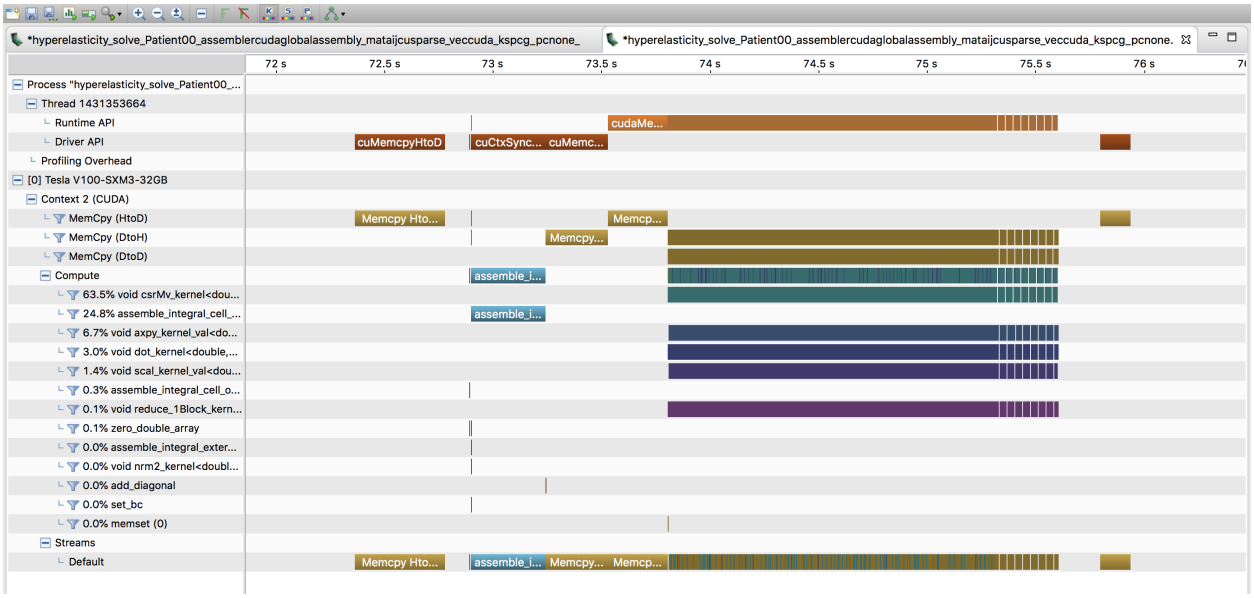


Figure 10 *Timeline of a GPU-accelerated application on an NVIDIA Tesla V100 GPU presented by the NVIDIA Visual Profiler.*

achieving better utilization of the SIMD lanes, *i.e.*, organize operations to run in parallel across a sub-warp/sub-group. Examples of memory-related optimizations can be a better organization of the data structures in order to achieve coalesced data accesses.

In both CPU and GPU Roofline analysis, Advisor allows the user to compare two or more results in the same Roofline view. This allows to verify the improvements between each version of the application, to assess the efficacy of the optimizations. Differences in performance and arithmetic intensity can be easily seen when using this feature, thus guiding the developer during the optimization process.

7.3 Profiling and Instrumentation on NVIDIA GPUs

For applications that employ NVIDIA GPUs to offload and accelerate computations, NVIDIA's CUDA Toolkit[59] comes with built-in profiling and instrumentation tools. In particular, the CUDA profiler, nvprof,[60] is designed to easily instrument CUDA API calls, including memory allocations, data transfers between host (i.e., CPU) and device (i.e., GPU), as well as kernel launches. With the help of the NVIDIA Visual Profiler, this information may be presented for offline analysis in the form of a visual timeline that quickly grants an overview of an application's GPU activity. An example is shown in Figure 10, which portrays a small extract from the execution of a GPU-accelerated finite element code on an NVIDIA Tesla V100 GPU.[61]

More detailed analysis can also be carried out at the level of individual kernels. This usually requires running nvprof with the `--analysis-metrics` option, which causes the CUDA runtime to record a great deal of useful information about kernel executions. This option uses hardware performance monitoring features of the GPU hardware to collect various performance metrics. Because kernels must be executed many times over to collect the required performance data, such

[59] NVIDIA Corporation. *CUDA Toolkit Documentation v11.6.1*. NVIDIA Corporation. 2022. URL: https://docs.nvidia.com/cuda/index.html.

[60] NVIDIA Corporation, *Profiler User's Guide*.

[61] NVIDIA Corporation. *NVIDIA Tesla V100 GPU architecture*. NVIDIA Corporation. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

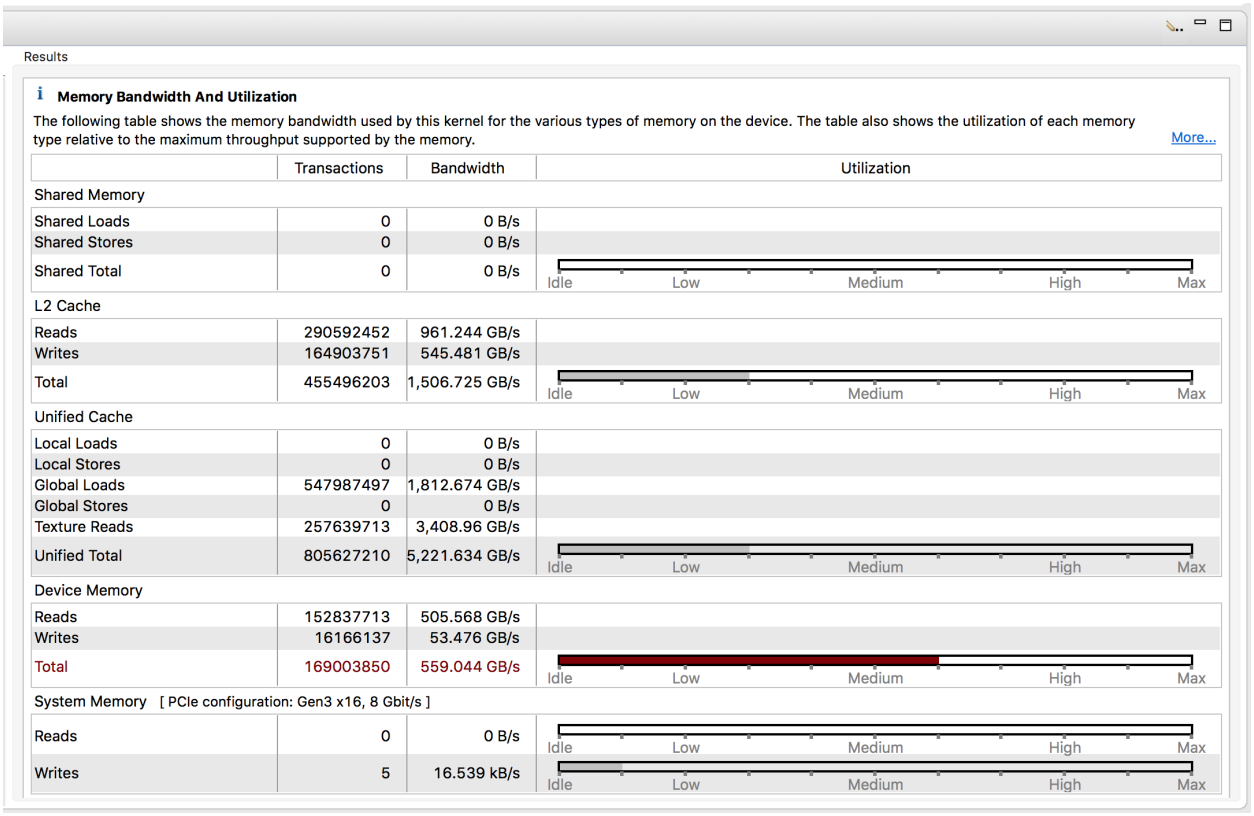


Figure 11 *Analysis of a kernel's memory usage presented by the NVIDIA Visual Profiler.*

profiling therefore incurs a considerable overhead.

Figure 11 shows an example of analysing the memory usage of a kernel from the GPU-accelerated finite element code mentioned above. The number of reads and writes are shown in addition to the achieved throughput for each of the GPU's different memory types, including shared memory, caches and device memory. Due to the use of an unstructured mesh, the kernel in question features highly irregular memory access patterns and is severely memory-bound. Nevertheless, the profiler's analysis indicates that a throughput of 559 GB/s (62 %) is attained out of the 900 GB/s theoretical maximum of the GPU device memory. This should be considered a high degree of bandwidth usage for a kernel with such irregular memory access patterns. If needed, additional memory statistics are also available to report memory traffic volumes and cache hit rates at different levels of the GPU's memory hierarchy.

In addition to metrics based on performance counters, nvprof may also act as a sampling profiler. This mode of operation is useful for correlating various performance issues with specific lines of kernel source or with specific instructions of the compiled kernel assembly code, as illustrated in Figure 12. This example points to specific source code lines where the kernel performs most of its irregular loads from memory. These irregular memory accesses prevent memory coalescing, and thus more transactions are needed compared to the ideal case of fully coalesced reads. Ultimately, such a performance analysis should guide one in the direction of memory optimisations, as described in the CUDA C++ Best Practices Guide.[62]

In recent versions of the CUDA Toolkit, nvprof and the NVIDIA Visual Profiler have been

[62] NVIDIA Corporation. *CUDA C++ Best Practices Guide*. NVIDIA Corporation. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

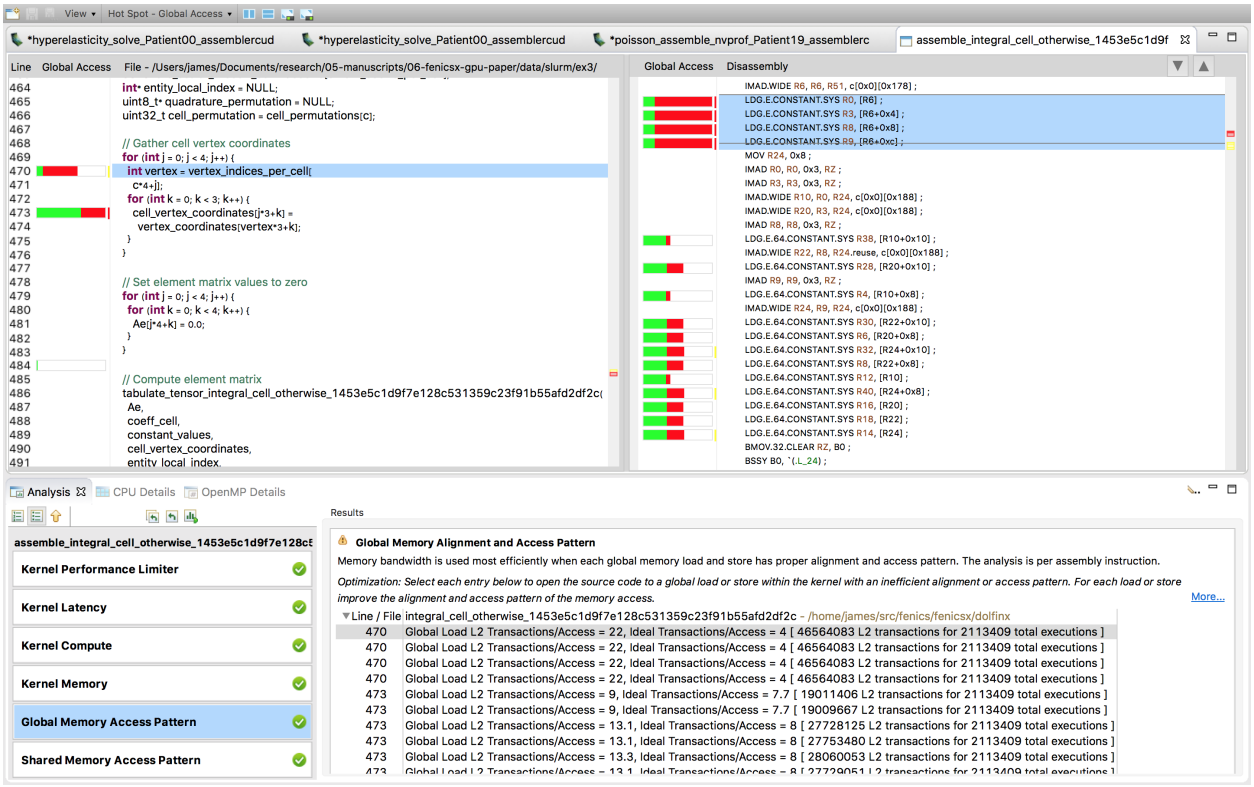


Figure 12 *Analysing kernel source and assembly code based on using nvprof as a sampling profiler to diagnose inefficient global memory accesses caused by irregular memory access patterns and a lack of memory coalescing.*

superseded by NVIDIA Nsight Systems and NVIDIA Nsight Compute, which offer essentially the same profiling and instrumentation capabilities for the successors of NVIDIA's Volta GPUs.

7.4 Tools for visual and programmatic analyses on Graphcore IPUs

Graphcore provides PopVision™, a set of tools for visual and programmatic analyses of applications run on IPU systems. These tools can be downloaded from Graphcore's support portal.[63] Readers are encouraged to explore the extensive documentation [64] and tutorials [65] on these tools and their usage. For the purposes of this report, we shall summarise the key forms of analyses that are possible with these tools at present. These include analysing Poplar™ graphs and their execution on IPUs, and analysing host side code in the context of the overall application run on an IPU system.

7.4.1 Analysing Poplar profiles

Poplar™ graphs run on IPUs can be deeply inspected to analyse and optimise for memory and performance using the PopVision™ Graph Analyser tool.[66] For instance, the tool can show tile memory usage across all the tiles on device in the IPU system on which a program is run. Some variables, e.g. accumulators that repeatedly get flushed, are not always live during program

[63] Graphcore's support portal: https://downloads.graphcore.ai/
[64] Profiling and debugging documentation: https://docs.graphcore.ai/en/latest/software.html
[65] PopVision™ tutorials: https://github.com/graphcore/tutorials/tree/master/tutorials/popvision
[66] PopVision™ Graph Analyser user guide: https://docs.graphcore.ai/projects/graph-analyser-userguide/

execution, and the Graph analyser is able to capture such information in liveness reports detailing operational memory usage across program steps. The tool also shows placement of variables per tile, including their size. Cycle and FLOP counts for vertices using these variables is also available for examining and tuning the performance of the program. It is also possible to analyse Poplar™ graphs programmatically using the PopVision™ Analysis Library (`libpva`),[67] which is part of the Poplar SDK and provides both a C++ and a Python interface for users to query profiles.

7.4.2 TRACING APPLICATIONS

Any IPU application can be instrumented to identify potential bottlenecks between the host and device. Graphcore's Poplar™ graph programming framework, and higher level frameworks such as PopART, Pytorch, and Tensorflow, make use of the PopVision™ Trace Instrumentation Library (`libpvti`)[68] to trace applications programmatically, also available to users via C++ and Python APIs. The default, framework provided tracing, and custom tracing instrumented by user, can be visually inspected using the PopVision™ System Analyser tool.[69] As an example, for a gradient descent based machine learning (ML) application, if user were to analyse the time taken by a single iteration involving data transfer between host and device and a gradient update, the user could create a custom trace channel and introduce trace points within the training loop. The application code run on the host would then log the iteration trace in a report available for later analyses. In addition to tracing an application's execution, `libpvti` also allows for capturing numerical data (series) from a run of the application, an example being logging the iteration loss, following the ML example just mentioned. This data series is displayed in real (wall) time alongside the application trace, enabling analyses of the impact of application execution on this series.

8 CACHE SIMULATION FOR IRREGULAR MEMORY TRAFFIC

This section describes profiling and instrumentation tools developed by the Simula partner, which is based on a cache tracing approach[70] that was recently developed to support detailed performance modelling of sparse kernels, in particular different variants of sparse matrix-vector multiplication, or SpMV. These tools include a cache tracing simulator for SpMV kernels on multicore CPUs for predicting cache and memory traffic in multilevel memory hierarchies with private and shared caches. We also give an example of using hardware performance monitoring features of Intel CPUs to accurately measure memory traffic throughout the memory hierarchy, which is a critical part of validating the cache tracing performance model.

8.1 CACHE TRACING FOR SPARSE MATRIX-VECTOR MULTIPLICATION

For sparse matrix-vector multiplication (SpMV) kernels, performance models can often fail to gauge the impact of irregular memory access patterns. In these cases, memory accesses inherently depend on the problem data itself, since they arise from the underlying matrix sparsity pattern (i.e., the locations of matrix nonzeros). Moreover, various realistic problems feature a wide range of disparate sparsity patterns, as illustrated in Figure 13.

[67] PopVision™ Analysis Library user guide: https://docs.graphcore.ai/projects/libpva/

[68] PopVision™ Trace Instrumentation Library user guide: https://docs.graphcore.ai/projects/libpvti/

[69] PopVision™ System Analyser user guide: https://docs.graphcore.ai/projects/system-analyser-userguide/

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

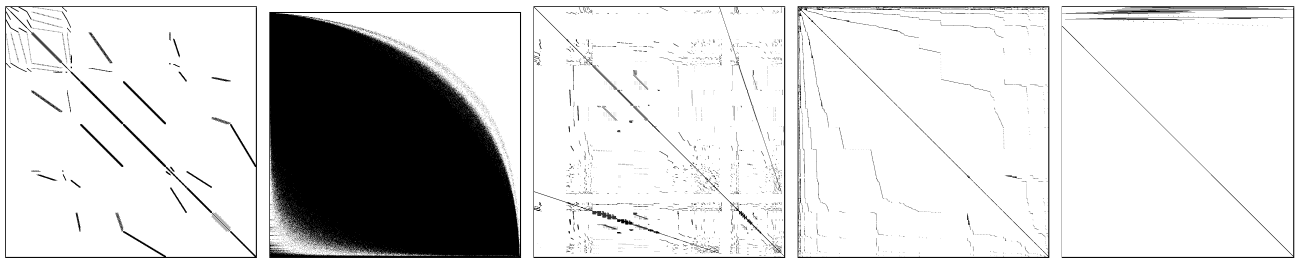


Figure 13 *Sparsity patterns of a few sparse matrices from the SuiteSparse Matrix Collection.*

In this section, a performance model based on cache tracing that explicitly accounts for the matrix sparsity pattern is presented. This is achieved by first obtaining a trace of the memory accesses that are induced by a particular kernel, and then using it to carry out a high-level simulation of the memory hierarchy of a specified multicore CPU system. The result of such a simulation is a set of cache and memory traffic estimates for each CPU core. By combining these estimates with the bandwidths of cache and memory (accounting for NUMA effects, if needed), we can identify bottlenecks in the memory hierarchy and predict the performance accordingly.

With the aim of applying the proposed performance model, we have developed tools for capturing memory traces of SpMV kernels. This includes kernels for several common sparse matrix storage formats, such as compressed sparse row (CSR), coordinate (COO), ELL and a hybrid format.[71] At the moment, these tools specifically target SpMV kernels and must be extended to encompass any new, irregular kernels one may wish to study. In the future, a more general approach could be to use instrumentation or hardware monitoring to automatically extract the required memory traces.

Once the memory trace has been obtained, we can supply it to our cache simulation tool to produce cache and memory traffic estimates at every level of the memory hierarchies of typical multicore CPUs. The simulation is based on a simplified model of such memory hierarchies, and it is related to the Ideal Cache Model,[72] but with some additions to simplify the overall simulation and to incorporate shared caches.

First, the cache simulation model assumes that caches are fully associative, and that a least recently used (LRU) policy is used when evicting cache lines to make room for new ones. The simulator accounts for cache lines that are brought to a cache as a direct result of load or store instructions issued by a CPU core. It does not account for any form of prefetching, unless it is explicitly incorporated into the memory trace itself. Furthermore, caches are assumed to be inclusive of higher-level caches (i.e., those closer to the CPU), meaning that a cache line residing in a particular cache must also be present in lower-level caches (farther from the CPU). For example, anything that is found in a first-level cache must also be present in second- and third-level caches. Although this assumption may not always be true in practice (e.g., the L2 and L3 caches in Intel's Skylake Server architecture are non-inclusive[73]), it greatly simplifies the overall modelling effort by allowing the cache simulation to be carried out independently for each cache. Alternatively, a non-inclusive cache can be treated as though it were larger, increasing its size by an amount equal

[71] Nathan Bell and Michael Garland. "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors". *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009. DOI: 10.1145/1654059.1654078.

[72] Matteo Frigo et al. "Cache-Oblivious Algorithms". *ACM Transactions on Algorithms* 8.1 (2012), 4:1–4:22. ISSN: 1549-6325. DOI: 10.1145/2071379.2071383.

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

to the size of those higher-level caches (closer to the CPU) not included in the current cache.

For a cache of a given size and with a given cache line size (64 bytes, in most cases), the simulation runs through the provided sequence of memory accesses to estimate the number of cache misses. The cache is represented by a least recently used (LRU) list that may hold a number of items equal equal to the cache size divided by the line size. The list is used to keep track of cache lines that were accessed most recently and to evict the oldest entries whenever new entries are placed at the front of the list. As the simulation progresses, it counts the number of cache misses that occur, and the final memory traffic volume is simply the cache line size multiplied by the number of cache misses.

For caches that are shared by several CPU cores, (e.g., the third-level caches of most multicore CPUs), the sequence of memory accesses observed by the shared cache is actually a combination of the underlying memory accesses generated by each individual core. In reality, these memory accesses occur in some unpredictable order, for example, due to differences between cores in thread scheduling or memory access latencies. Even executing the same kernel twice may produce different memory traces. To avoid needlessly complicating the cache simulation, we instead assume that memory accesses from different CPU cores are perfectly interleaved. The CPU cores are thus treated as if their memory requests are served one at a time in a round-robin fashion.

A known limitation of cache simulation methods in general is that they are relatively expensive, at least compared to executing the kernel itself. On the other hand, the simulation and performance modelling can be performed even without access to the multicore CPU system being modelled, or in scenarios where the data traffic cannot be quantified directly through hardware monitoring facilities.

8.2 CACHE TRACING RESULTS

The cache tracing performance model explained above is based on high-level simulations of the memory hierarchies of multicore CPU systems, including private as well as shared caches. The model has been validated for multicore CPUs based on the Intel Sandy Bridge and Skylake X architectures by comparing its memory traffic predictions to measurements of actual cache and memory traffic obtained from hardware performance counters on those systems.

More specifically, measurements of actual data traffic volumes are obtained from Performance Monitoring Units (PMUs) that are configured to report hardware performance events.[74] In our case, these events are accessed by providing event encodings to the `perf_event_open`[75] system call and using the returned file descriptor to read values of hardware performance counters. The relevant event encodings are acquired from the *libpfm4*[76] library by providing it with corresponding event names. Suitable hardware events were selected after using microbenchmarks to correlate hardware events with data transfers between different levels of the memory hierarchy, as described by Molka et al..[77]

Table 5 shows measurements of cache and memory traffic volumes for a standard, OpenMP-parallel SpMV kernel based on the compressed sparse row (CSR) storage format on a dual socket Intel Xeon Platinum 8168 system. On each CPU core, the events *l1-dcache-load-misses*, *l2_lines_in:any*

[74] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*. 325384-065US. Intel Corporation, 2017, Ch. 18–19.

[75] Vincent Weaver. *perf_event_open*. Linux programmer's manual, version 5.13, The Linux man-pages project (Eds. Michael Kerrisk and Alejandro Colomar). 2012.

[76] Stephane Eranian and Robert Richter. *perfmon2: improving performance monitoring on Linux*. http://perfmon2.sourceforge.net/.

[77] Daniel Molka et al. "Detecting Memory-Boundedness with Hardware Performance Counters". *Proceedings of the 8th ACM/SPEC on international conference on performance engineering*. ACM, 2017, pp. 27–38.

	L1 ← L2 [MiB]			L2 ← L3 [MiB]			L3 ← DRAM [MiB]		
matrix	meas.	est.	(error)	meas.	est.	(error)	meas.	est.	(error)
TSOPF_RS_b2383	194	187	(−3.6 %)	199	185	(−7.0 %)	187	185	(−1.1 %)
spal_004	1320	1054	(−20.2 %)	866	840	(−3.0 %)	535	528	(−1.3 %)
RM07R	525	488	(−7.0 %)	481	454	(−5.6 %)	441	447	(+1.4 %)
relat9	1504	1477	(−1.8 %)	882	844	(−4.3 %)	593	589	(−0.7 %)
HV15R	3683	3522	(−4.4 %)	3467	3368	(−2.9 %)	3346	3360	(−0.4 %)
GL7d19	3213	2714	(−15.5 %)	2425	2397	(−1.2 %)	483	473	(−2.1 %)
sx-stackoverflow	3254	2565	(−21.2 %)	2342	2173	(−7.2 %)	504	552	(+9.5 %)
FullChip	511	490	(−4.1 %)	544	462	(−15.1 %)	410	428	(+4.4 %)
Freescale1	389	381	(−2.1 %)	398	369	(−7.3 %)	316	332	(+5.1 %)
circuit5M	1196	1185	(−0.9 %)	1073	957	(−10.8 %)	908	927	(+2.1 %)
Hardesty3	744	734	(−1.3 %)	653	620	(−5.1 %)	632	618	(−2.2 %)

Table 5 *Measurements and estimates of cache and memory traffic (in MiB) for a SpMV kernel with the CSR storage format on a dual socket Intel Xeon Platinum 8168 system. For each CPU core, the hardware performance events "l1-dcache-load-misses", "l2_lines_in:any" and "offcore_response_0:any_data:any_rfo:l3_miss_local:l3_miss_miss_remote_hop1_dram:snp_any" are used to measure L1 ← L2, L2 ← L3, and L3 ← DRAM traffic, respectively. The measured cache and memory traffic volumes shown are the sum across all cores.*

and *offcore_response_0:any_data:any_rfo:l3_miss_local:l3_miss_miss_remote_hop1_dram:snp_any* are used to measure L1 ← L2, L2 ← L3, and L3 ← DRAM traffic, respectively. For comparison, estimates of cache and memory traffic produced by the cache tracing approach are also shown. Overall, the estimates from the cache simulation are close to the memory traffic measurements. The error is less than 20 % even for very challenging and irregular matrices, such as "GL7d19" and "sx-stackoverflow". Additional measurements and cache tracing results are found in a published research paper,[78] together with detailed explanations of discrepancies between measurements and cache tracing estimates, as well as further discussion of the accuracy and limitations of the cache tracing approach.

Since the end goal is to model the performance of SpMV kernels, the estimated cache and memory traffic volumes serve as intermediate results. Ultimately, cache and memory bandwidths are also needed to determine the time needed to transfer data from each memory hierarchy level. Although theoretical bandwidth numbers are sometimes provided by vendors, it is more common in practice to use microbenchmarks, such as STREAM,[79] to measure the needed bandwidths. In our current example, the estimated cache tracing performance is based on experimentally measured per-core bandwidths of 13.4, 12.5 and 10.1 GB/s for L1 ← L2, L2 ← L3 and L3 ← DRAM traffic, respectively. In addition, a bandwidth of 99.8 GB/s is used for the aggregate memory traffic for the NUMA domain of each socket on the dual socket Intel Xeon Platinum 8168 system. Data transfer times are found by dividing estimated cache or memory traffic volumes by the corresponding bandwidth, and the supposed performance bottleneck is simply the memory hierarchy level with the longest data transfer time.

Table 6 finally compares the measured performance of the CSR SpMV kernel to the performance predicted by the cache simulation approach for a selection of large, sparse matrices

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

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

				performance [Gflop/s]		
matrix	rows	columns	nonzeros	measured	cache sim	(error)
atmosmodl	1.49M	1.49M	10M	37.37	25.74	(-31 %)
HV15R	2.02M	2.02M	283M	30.68	31.96	(+4 %)
Freescale1	3.43M	3.43M	19M	16.50	13.65	(-17 %)
Freescale2	3.00M	3.00M	23M	17.75	23.67	(+33 %)
FullChip	2.99M	2.99M	27M	4.55	4.97	(+9 %)
circuit5M	5.56M	5.56M	60M	4.05	3.85	(-5 %)
circuit5M_dc	3.52M	3.52M	19M	19.31	19.07	(-1 %)
memchip	2.71M	2.71M	15M	21.83	19.87	(-9 %)
Hardesty3	8.22M	7.59M	40M	16.18	24.60	(+52 %)
ML_Geer	1.50M	1.50M	111M	32.04	31.97	(0 %)
Transport	1.60M	1.60M	24M	34.35	28.49	(-17 %)
tp-6	0.14M	1.01M	12M	11.41	7.78	(-32 %)
rajat31	4.69M	4.69M	20M	23.19	22.66	(-2 %)
dgreen	1.20M	1.20M	38M	22.73	13.21	(-42 %)
nv2	1.45M	1.45M	53M	14.42	12.59	(-13 %)
ss	1.65M	1.65M	35M	24.22	24.26	(0 %)
stokes	11.45M	11.45M	349M	14.51	22.18	(+53 %)
vas_stokes_1M	1.09M	1.09M	35M	17.06	22.52	(+32 %)
vas_stokes_2M	2.15M	2.15M	65M	15.80	22.47	(+42 %)
vas_stokes_4M	4.38M	4.38M	132M	14.14	22.55	(+59 %)
cage14	1.51M	1.51M	27M	28.47	25.96	(-9 %)
cage15	5.15M	5.15M	99M	25.81	24.39	(-6 %)

Table 6 *Measured and predicted performance (in Gflop/s) based on the cache tracing performance model for a SpMV kernel with the CSR storage format on a dual socket Intel Xeon Platinum 8168 system. The dataset consists of 23 unsymmetric, real matrices from the SuiteSparse Matrix Collection with more than* 10 *million nonzeros and more than one million columns.*

from the SuiteSparse Matrix Collection.[80] Since the predictions appear to largely agree with the measured performance, the proposed performance model captures the overall performance characteristics of the irregular kernel.

In some instances, such as "Hardesty3", "stokes", "vas_stokes_2M" and "vas_stokes_4M", the performance model overestimates the performance by about 50 %. Possible explanations may include less cache reuse than expected, for example, due to a substantial number of conflict misses, or lower effective bandwidth due to poor hardware prefetching. Recall that conflict misses are ignored by the cache simulator due to the assumption of fully associative caches, and the experimentally measured bandwidths used are based on streaming access patterns, where prefetching should be mostly successful. Conversely, in cases where the model underestimates the actual performance, there may be more cache reuse than expected, for example, if the underlying replacement policy is different than the supposed LRU model,[81] or due to caches being non-inclusive. In any case, further analysis is needed to understand these discrepancies between the model and the performance that is observed in practice.

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

[81] Andreas Abel and Jan Reineke. "Measurement-based modeling of the cache replacement policy". *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2013, pp. 65–74. DOI: 10.1109/RTAS.2013.6531080; Pepe Vila et al. "CacheQuery: Learning Replacement Policies from Hardware Caches". *PLDI '20: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, 2020, pp. 519–532. DOI: 10.1145/3385412.3386008.

9 CONCLUSIONS

The deliverable 4.1 focused on presenting several state-of-the-art tools that can be used to monitor the performance, power and energy on computing systems equipment with different processing devices that range from multi-core CPUs, to GPUs and Graphcore IPUs. First, hardware and software events for performance and energy monitoring were introduced, with a specific emphasis given to the events that might be able to provide more insights about the execution of sparse applications. Since most of the events supported by current operating systems and computing devices are accessed through profiling tools, the main features of the most commonly used tools, i.e., Perf, PAPI and LIKWID, were exposed in this deliverable. Moreover, communication profiling tools based on the precise sampling on hardware counters contained in Intel and AMD CPUs, such as ComDetective and ReuseTracker, were also described. As these tools can provide insightful information regarding the data communication between and inside threads, they allow for detecting bottlenecks that may occur when parallelizing sparse kernels. A similar approach for GPU-GPU or GPU-CPU pairs is also considered within the ComScribe tool, which is based on the communication protocols of NVIDIA. System-level monitoring tools can be used to aid the monitoring of the different events and data collection, especially when targeting large scale systems. To this respect, the main concepts of LDMS, Performance Co-Pilot and Grafana were included in this report. Dynamic binary instrumentation methods, *e.g.*, Intel SDE, Intel Pin and Intel GTPin, were also covered, as the data collected by these tools can provide hints about the execution units exercised by applications. As a use case of the utilization of dynamic instrumentation approaches, a custom tool for reuse distance analysis and cache partitioning on ARM A64FX CPU was also elaborated in the deliverable. Next, some of the frameworks provided by the different device vendors for performance analysis on their hardware were also described. For Intel CPUs and GPUs, the most common frameworks used in the state-of-the art are Intel Advisor, which includes the Cache-Aware Roofline Model, and Intel VTune that encapsulates the Top-Down performance analysis method. For NVIDIA GPUs, different metrics can be obtained from the NVIDIA Visual Profiler, while for Graphcore IPU, the Graphcore PopVision and Poplar tools are used instead. Finally, a cache simulation method for irregular memory traffic focused on sparse matrix-vector multiplication was also introduced. This tool considers the sparsity patterns of the matrix to provide more accurate traffic estimates between memory levels contained in the CPUs, which allows to pinpoint the main execution bottlenecks of SpMV kernel.

REFERENCES

Abel, Andreas and Jan Reineke. "Measurement-based modeling of the cache replacement policy". *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2013, pp. 65–74. DOI: 10.1109/RTAS.2013.6531080.

Adhianto, L. et al. "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, Anthony Michael et al. *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).

Akhtar, Palwisha et al. "ComScribe: Identifying Intra-node GPU Communication". 2021. DOI: 10.1007/978-3-030-71058-3_10.

AMG. *Parallel Algebraic Multigrid Solver*. https://github.com/LLNL/AMG. 2017.

Bach, Moshe et al. "Analyzing parallel programs with pin". *Computer* 43.3 (2010), pp. 34–41.

Barrow-Williams, Nick, Christian Fensch, and Simon Moore. "A communication characterisation of Splash-2 and Parsec". *IEEE International Symposium on Workload Characterization, 2009. IISWC 2009.* 2009.

Bell, Nathan and Michael Garland. "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors". *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009. DOI: 10.1145/1654059.1654078.

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

Brandt, James M., Thomas Tucker, and Ann C. Gentile. *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. "Evaluating Sector Caches in High-Performance Computing". Master Thesis. Ludwig-Maximilians-Universität München, 2022.

Cruz, Eduardo Henrique Molina da et al. "Using Memory Access Traces to Map Threads and Data on Hierarchical Multi-core Platforms". *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*. 2011.

Dave, Shail et al. "Hardware Acceleration of Sparse and Irregular Tensor Computations of ML Models: A Survey and Insights". *Proceedings of the IEEE* 109.10 (2021), pp. 1706–1752. DOI: 10.1109/JPROC.2021.3098483.

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

dcompiler/loca: Program locality analysis tools. https://github.com/dcompiler/loca.

De Melo, Arnaldo Carvalho. "The new linux'perf'tools". *Slides from Linux Kongress*. Vol. 18. 2010, pp. 1–42.

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

Drongowski, Paul J. *Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors*. https://pdfs.semanticscholar.org/5219/4b43b8385ce39b2b08ecd409c753e0efafe5.pdf. 2007.
Eranian, Stephane and Robert Richter. *perfmon2: improving performance monitoring on Linux*. http://perfmon2.sourceforge.net/.
Family, Intel Xeon Processor Scalable Memory. "Uncore Performance Monitoring Reference Manual". *Intel Corporation, July* (2017).
Frigo, Matteo et al. "Cache-Oblivious Algorithms". *ACM Transactions on Algorithms* 8.1 (2012), 4:1–4:22. ISSN: 1549-6325. DOI: 10.1145/2071379.2071383.
Ganglia. *Ganglia monitoring system*. URL: http://ganglia.sourceforge.net/.
Gregg, Brendan. 2021. URL: https://brendangregg.com/linuxperf.html.
Intel. *Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide*. https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf. 2010.
Intel, R. "and IA-32 Architectures. Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C". *Order Number* (64).
Intel Corporation. *GTPin*. https://software.intel.com/sites/landingpage/gtpin/index.html. [Online; visited March-2022].
— *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 248966-040. Intel Corporation, 2018.
— *Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*. 325384-065US. Intel Corporation, 2017.
— *Profiling Tools Interfaces for GPU (PTI for GPU)*. https://github.com/intel/pti-gpu. [Online; visited March-2022].
— *Understanding How General Exploration Works in Intel® VTune™ Amplifier*. https://www.intel.com/content/www/us/en/developer/articles/technical/understanding-how-general-exploration-works-in-intel-vtune-amplifier-xe.html. [Online; visited March-2022].
Luk, Chi-Keung et al. "Pin: building customized program analysis tools with dynamic instrumentation". *Acm sigplan notices* 40.6 (2005), pp. 190–200.
LULESH 2.0. *Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)*. https://github.com/LLNL/LULESH.
Magnusson, P.S. et al. "Simics: A full system simulation platform". *Computer* 35.2 (2002), pp. 50–58.
Marques, Diogo et al. "Application-driven cache-aware roofline model". *Future Generation Computer Systems* 107 (2020), pp. 257–273.
Mazaheri, Arya, Felix Wolf, and Ali Jannesari. "Characterizing Loop-Level Communication Patterns in Shared Memory Applications". *Proceedings of the 2015 44th International Conference on Parallel Processing*. Beijing, China, 2015. DOI: 10.1109/ICPP.2015.85.
— "Unveiling Thread Communication Bottlenecks Using Hardware-Independent Metrics". *Proceedings of the 47th International Conference on Parallel Processing*. Eugene, OR, USA: ACM, 2018, 6:1–6:10. DOI: 10.1145/3225058.3225142. URL: http://doi.acm.org/10.1145/3225058.3225142.
McCalpin, John D. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Department of Computer Science School of Engineering and Applied Science, University of Virginia. 2013. URL: https://www.cs.virginia.edu/stream/.
miniFE. *MiniFE Finite Element Mini-Application*. https://github.com/Mantevo/miniFE.

Mohammed, Thaha et al. "DIESEL: A novel deep learning-based tool for SpMV computations and solving sparse linear equation systems". *The Journal of Supercomputing* 77.6 (2021), pp. 6313–6355.

Molka, Daniel et al. "Detecting Memory-Boundedness with Hardware Performance Counters". *Proceedings of the 8th ACM/SPEC on international conference on performance engineering*. ACM, 2017, pp. 27–38.

NVIDIA Corporation. *CUDA C++ Best Practices Guide*. NVIDIA Corporation. 2022. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/.

— *CUDA Toolkit Documentation v11.6.1*. NVIDIA Corporation. 2022. URL: https://docs.nvidia.com/cuda/index.html.

— *NVIDIA Tesla V100 GPU architecture*. NVIDIA Corporation. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

— *Profiler User's Guide*. NVIDIA Corporation. 2022. URL: https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

PENNANT. *Unstructured mesh hydrodynamics for advanced architectures*. https://github.com/lanl/PENNANT. 2016.

Quicksilver. *A proxy app for the Monte Carlo Transport Code, Mercury*. https://github.com/LLNL/Quicksilver.

Ramabathiran, Amuthan A. and Prabhu Ramachandran. "SPINN: Sparse, Physics-based, and partially Interpretable Neural Networks for PDEs". *Journal of Computational Physics* 445 (2021), p. 110600. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2021.110600. URL: https://www.sciencedirect.com/science/article/pii/S0021999121004952.

Red-Hat. URL: https://pcp.io/.

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

Sasongko, Muhammad Aditya et al. "ReuseTracker: Fast Yet Accurate Multicore Reuse Distance Analyzer". *ACM Trans. Archit. Code Optim.* 19.1 (2021). ISSN: 1544-3566. DOI: 10.1145/3484199. URL: https://doi.org/10.1145/3484199.

Shen, Xipeng, Jonathan Shaw, and Brian Meeker. "Accurate Approximation of Locality from Time Distance Histograms". 2006.

Terpstra, Dan et al. "Collecting performance data with PAPI-C". *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

Treibig, J., G. Hager, and G. Wellein. "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments". *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*. 2010.

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

Vila, Pepe et al. "CacheQuery: Learning Replacement Policies from Hardware Caches". *PLDI '20: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, 2020, pp. 519–532. DOI: 10.1145/3385412.3386008.

VPIC. *Vector Particle-In-Cell (VPIC) Project*. https://github.com/lanl/vpic.
Weaver, Vincent. *perf_event_open*. Linux programmer's manual, version 5.13, The Linux man-pages project (Eds. Michael Kerrisk and Alejandro Colomar). 2012.
Xiang, Xiaoya et al. "HOTL: A Higher Order Theory of Locality". *SIGARCH Comput. Archit. News* 41.1 (2013), pp. 343–356. ISSN: 0163-5964. DOI: 10.1145/2490301.2451153. URL: https://doi.org/10.1145/2490301.2451153.
Yang, Ulrike Meier. "Parallel Algebraic Multigrid Methods High Performance Preconditioner". *Numerical Solution of Partial Differential Equations on Parallel Computers, LNCS 51* (2006), pp. 209–233.
Yasin, Ahmad. "A top-down method for performance analysis and counters architecture". *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2014, pp. 35–44.
Yoshida, Toshio et al. "SPARC64 VIIIfx: CPU for the K computer". *Fujitsu Sci. Tech. J* 48.3 (2012), pp. 274–279.
Zhao, Yue et al. "Bridging the gap between deep learning and sparse matrix format selection". *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 2018, pp. 94–108.