D5.8 Final Benchmark Results for Innovative Architectures

Document Identification			
Status	Final	**Due Date**	30/01/2022
Version	1.0	**Submission Date**	11/02/2022

Related WP	WP5	**Document Reference**	D5.8
Related Deliverable(s)	D3.1, D3.2, D3.3, D3.4, D3.5, D4.2, D4.3, D4.4, D5.5, D6.2, D6.4, D6.5, D6.6	**Dissemination Level (*)**	PU
Lead Participant	USTUTT	**Lead Author**	Sergiy Gogolenko
Contributors	USTUTT PSNC ICCS SZE PLUS	**Reviewers**	Konstantinos Nikas (ICCS) Nabil Ben Said (MOON)

Keywords:

New promising technologies, HPC, benchmarks, scalability, efficiency, exascale, Global Challenges, Global Systems Science

Document Information

List of Contributors	
Name	Partner
Sergiy Gogolenko	USTUTT
Lukasz Szustak	PSNC
Krzesimir Samborski	PSNC
Marcin Lawenda	PSNC
Nikela Papadopoulou	ICCS
László Környei	SZE
Mátyás Constans	SZE
Gregor Bankhamer	PLUS

Document History			
Version	Date	Change editors	Changes
0.1	13/07/2021	Sergiy Gogolenko (USTUTT)	TOC
0.2	03/10/2021	Sergiy Gogolenko (USTUTT)	Contributions to Chapter 2 from USTUTT
0.3	13/11/2021	Sergiy Gogolenko (USTUTT)	Contributions to Chapter 3 from USTUTT
0.4	15/12/2021	Sergiy Gogolenko (USTUTT)	Merged contributions from PLUS and ICCS
0.5	23/12/2021	Sergiy Gogolenko (USTUTT)	Added Intro, Conclusions, and Annexes
0.6	17/01/2022	Sergiy Gogolenko (USTUTT)	Version for internal review
0.7	04/02/2022	Sergiy Gogolenko (USTUTT)	Addressing review comments. Final version to be submitted.
1.0	11/02/2022	Francisco Javier Nieto de Santos (ATOS)	FINAL VERSION TO BE SUBMITTED

Quality Control		
Role	Who (Partner short name)	Approval Date
Deliverable Leader	Sergiy Gogolenko (USTUTT)	05/02/2022
Quality Manager	Marcin Lawenda (PSNC)	07/02/2022
Project Coordinator	Francisco Javier Nieto de Santos (ATOS)	11/02/2022

Table of Contents

List of Tables

List of Figures

List of Acronyms

Abbreviation / Acronym	Description
3D	Three-dimensional
ABSS	Agent-based social simulations
AMD	Advanced Micro Devices, Inc.
AP	Agent-parallel
API	Application programming interface
ARM	Advanced RISC Machines
ASP	Agent-space-parallel
BF16	Brain Floating Point 16-bit format
ccNUMA	Cache coherent NUMA
CDNA	Compute DNA, data centre compute GPU by Advanced Micro Devices
CFD	Computational fluid dynamics
CH	Contraction hierarchies
CI/CD	Continuous Integration / Continuous Deployment
CoeGSS	Cluster of Excellence for Global System Science
CPU	Central processing unit
CRP	Customizable route planning
CSV	Comma-separated values
CUDA	Compute unified device architecture
D	Deliverable
DL	Deep learning
DRAM	Dynamic random-access memory
EC	European Commission
FAQ	Frequently Asked Questions
FLOPS	Floating point operations per second
FP	Floating point format
FPGA	Field-programmable Gate Array
FS	Files system
GB	Gigabyte
GC	Global Challenge
GCC	GNU compiler collection
GIS	Geographic information system
GPC	GPU processing clusters
GPGPU	General-purpose computing on graphics processing units

Abbreviation / Acronym	Description
GPU	Graphics processing unit
GSS	Global system science
GUI	Graphical User Interface
HBM2	High bandwidth memory, second generation
HDD	Hard disk drive
HLRS	High Performance Computing Center Stuttgart
HPC	High Performance Computing
HTTPS	HyperText Transfer Protocol Secure
IEEE	Institute of Electrical and Electronics Engineers
INT	Integer format
IO	Input/output
IPC	Instructions per cycles
ISO	International Organization for Standardization
JIT	Just in time
KPM	Kernel-Polynomial-Method
LA	Linear algebra
M	Month
MIG	Multi-instance GPU
MIG	Migration
MLD	Multilevel Dijkstra
MPI	Message Passing Interface
NUMA	Non-uniform memory access
NVLink	Nvidia's multi-lane near-range communications link
NVM	Non-volatile memory
NVRAM	Non-volatile random-access memory
OS	Operating system
OSM	OpenStreetMap
OSRM	Open Source Routing Machine
P2P	Point-to-point
PBS	Portable Batch System
PCIe	Peripheral component interconnect express
PSNC	Poznan Supercomputing and Networking Center
PUE	Power Usage Effectiveness
PUNCH	Partitioning with natural cuts
Q&A	Questions and Answers
RAM	Random-access memory

Abbreviation / Acronym	Description
REST	Representational State Transfer
RISC	Reduced instruction set computer
ROCm	Radeon open compute
SLURM	Simple Linux Utility for Resource Management
SM	Streaming multiprocessors
SME	Small- and Medium-scale Enterprise
SNA	Social network analysis
SPEC	Standard performance evaluation corporation
SR-IOV	Single root input/output virtualization
SSD	Solid-state drive
SSH	Secure shell
SSL	Secure Socket Layer
SVD	Singular value decomposition
T	Task
TB	Terabyte
TF32	TensorFloat 32-bit format
TPCs	Texture processing clusters
UAP	Urban air pollution
URL	Uniformed Resource Locator
VM	Virtual Machine
VNNI	Vector neural network instructions
VVUQ	Verification, validation, and uncertainty quantification
WP	Work package

Executive Summary

D5.8 presents the final version of the HiDALGO benchmark suite and reports results of its execution on five testbeds composed of the state-of-the-art CPUs and accelerators from major vendors such as AMD, ARM (Huawei), Intel, and Nvidia. This work concludes activities on the evaluation of the innovative HPC architectures for the domain of Global Challenges (GC) conducted in the frame of HiDALGO project.

Our benchmark suite complements mature generic benchmark suites such as HPL, HPCG, or SPEC, and benchmark suites for global systems science applications such as CoeGSS benchmark suites in several way. It introduces a simple and portable methodology for collecting and reporting benchmark results, a wide set of automation scripts, as well as a rich set of benchmarks for CPUs and GPUs that cover computationally expensive data pre-/post-processing and simulation kernels in the data flows of HiDALGO pilots.

In conjunction with WP3, information from this report can be used not only to understand applications' behaviour better and improve the general scalability, but also to build a co-design baseline.

1 Introduction

1.1 Purpose of the document

This deliverable (D5.8) is prepared in the context of Task 5.4 of WP4, which pursues two major goals: (i) to analyse promising technologies and (ii) to develop a benchmark suite in order to improve application and system qualification in general. Specifically, D5.8 describes the final version of the HiDALGO benchmark suite and presents the final benchmark results, which have been executed on cutting-edge CPU and GPU testbeds. The HiDALGO benchmark suite differs from alternative benchmark suites in a way that it accounts for the specifics of the data flows in the use cases. Along with results obtained in WP3, results of this deliverable act as a co-design baseline.

1.2 Relation to other project work

Since we designed the HiDALGO benchmark suite taking into account all software components of the use cases as described in Chapter 3, this work is closely related to all technical work packages working on the use cases' software. In WP4 and WP6, we developed and reported the algorithms and the data flows of the use cases [2-8]. These were used to define the kernels of the HiDALGO benchmark suite. In WP3, we deal with the implementation, optimization and data management of the new tools [9-13]. As D5.8 reports performance results on the new hardware and software solutions, this information can be used to improve implementation and optimization of the tools in the frame of WP3. Moreover, results from D3.1 [9], D3.4 [12] and D3.5 [13] provided a baseline for identifying hotspots of the data flows of pilots, which serve as the kernels of the HiDALGO benchmark suite. Similarly to some WP3 reports, D5.8 presents results for measuring performance. Nevertheless, in contrast to WP3, this deliverable focuses on comparing the hardware platforms, while the main focus of WP3 deliverables is put on the software optimization. As a result, benchmarks of D5.8 complements reports of WP3 and act as a co-design baseline in strong cooperation with WP3.

Last but not least, this text is the second of a series of reports focusing on the Task 5.4 of WP4 (D5.5 [1], D5.8). This deliverable continues discussion of the novel architectures conducted in the D5.5 report [1] and extends it with the benchmark results for the novel testbeds.

1.3 Structure of the document

The rest of the document is organized as follows. **Chapter 2** presents architecture of the testbeds analysed in this report: after summarizing the trends in innovative HPC architectures covered in D5.5, we discuss core computational units of the evaluated testbeds – CPUs and GPUs – and outline testbed configurations. **Chapter 3** describes structure, components, and implementation of the HiDALGO benchmark suite. In particular, we motivate choice and specify implementation of the kernels of the benchmark suite, as well as explain benchmarking methodology and implementation of the automation mechanisms. We end this chapter by comparing the HiDALGO benchmark suite with alternatives such as SPEC and CoeGSS benchmark suites. **Chapter 4** presents and analyses results of running the HiDALGO benchmark suite on the testbeds from **Chapter 2**. **Chapter 5** concludes this deliverable and provides a summary. This document ends with three Annexes. **Annex A** contains short instructions about the automated deployment of the HiDALGO benchmark suite on the testbeds. **Annex B** holds visual representation for the system topology of the studied testbeds. **Annex C** presents auxiliary results that go beyond the methodology of the HiDALGO benchmark suite, yet help to understand major benchmark results better.

2 System components and configurations of the testbeds

We start with a short summary of the trends in innovative HPC architectures covered in details in D5.5 [1]. Next, we provide an overview of the building blocks of the testbeds studied in this deliverable such as processors, accelerators and memory. We finish this section providing an overview of the testbed configurations.

2.1 Recap of the trends in innovative HPC architectures

In D5.5 [1], we covered the most relevant trends in innovative HPC architectures. In particular, we conducted market analysis and comparison against the major metrics on such components as CPUs, accelerators, and RAM for all major vendors. Below, we recap the major findings.

Among the traditional CPU vendors for HPC, we discussed the state-of-the-art products of Intel and AMD. In order to reveal trends, the analysis contained comparison of top microprocessors year 2019 against year 2017 for each vendor. In our study of Intel's CPUs, we compared two typical x86-64 14nm server processors for scalable performance: Xeon Gold 6138 with a Sky Lake-SP (LGA 3647) microarchitecture against a more recent Xeon Gold 6248 with a Cascade Lake-SP microarchitecture. Despite significantly higher base core frequency (2.5GHz against 2.0GHz), Intel Xeon Gold 6248 introduced all benefits of Cascade Lake relative to Sky Lake microarchitecture such as doubled number of memory channels, 3D XPoint memory support, advanced vector extensions for Deep Learning Boost, e.g. VNNI (Vector Neural Network Instructions) logic, as well as mitigations for well-known Intel's hardware vulnerabilities – Meltdown and Spectre. As a result of all these improvements, SPEC benchmark results are higher for Cascade Lake processor than Sky Lake. In this deliverable, we benchmark a testbed with Xeon Platinum 8268 – a contemporary version of Intel's processor with the Cascade Lake-SP microarchitecture.

In the study of AMD's CPUs, we analysed two x86-64 server microprocessors: AMD Epyc Naples 7551 with a Zen microarchitecture against AMD Epyc Rome 7742 with a Zen2 microarchitecture. In the SPEC benchmark, AMD Epyc Rome 7742 outperformed not only AMD Epyc Naples 7551, but also Intel Xeon Gold 6248. AMD Epyc Rome 7742 demonstrated higher FLOPS performance, higher performance per watt, as well as higher bandwidth connection to InfiniBand and other fabric and storage adapters (NVMe SSDs, GPU Accelerators, and FPGAs) thanks to PCIe x16 v4.0. This deliverable reports benchmark results for a testbed with AMD Epyc Milan – a contemporary AMD processor with Zen3 cores.

Among the new players on the HPC server market, D5.5 contained the comparisons for the ARM-based server processors: Kunpeng 916 (Hi1616) with 32x ARM Cortex-A72 cores against Kunpeng 920 (Hi1620) with 64x TaiShan v110 cores (microarchitecture that implements ARMv8.2-A). In the SPEC benchmark, due to a higher number of cores, Kunpeng 920 outperformed Intel Xeon Gold 6248, but was almost twice slower than AMD Epyc Rome. On the other hand, ARM provided much better performance per watt ratio than the other families of processors. Besides full implementation of ARMv8.2-A and a few memory access improvements, Kunpeng 920 also featured compression and crypto offload engines. The latter can be useful for ABMS applications where hash function calls are commonplace.

The review in D5.5 covered such popular accelerator technologies as FPGA, vector co-processors, and GPGPU. It concluded that GPGPU is the leading accelerator technology in the HPC domain, as well as the most suitable accelerator technology for GSS applications. In order to reveal trends in GPGPU, we presented comparison of the top GPU cards of Nvidia (V100 PCIe and V100 SXM2) and AMD (MI50 and MI100) available in the market in 2020. This comparison demonstrated that AMD MI100 cards can be considered as a strong alternative for Nvidia V100 GPGPU systems due to a much higher FLOPS rate even though they generally have a simpler core architecture. As we show later in this text, the new generation of Nvidia cards – Ampere A100 GPUs – outperform AMD MI100 in both FLOPS rate and core architecture.

Since many GSS applications are memory- and I/O-bound, we separately included a review of the state-of-the-art memory technologies in D5.5. Besides overview of trends and leading solutions for volatile RAM memory such as DDR5 SDRAM (double-data-rate five synchronous dynamic random access memory), the review presented non-volatile RAM (NVRAM) solutions for bridging the performance gap between SDRAM and SSD cards with the focus on products based on the 3D Xpoint technology – Intel Optane PMem and Intel Optane SSD. We identified all these technologies as viable candidates for improving the performance of I/O-bound GSS applications.

2.2 General-purpose CPUs

Table 1 presents a short summary of the processors used in the testbeds studies in this deliverable. Below we provide more details on the novel features introduced in these CPUs.

Features	Intel Xeon(R) Platinum 8268	AMD EPYC Milan 7763	ARM Hi1620 (Kunpeng 920)
Microarchitecture	x86-64, Cascade Lake	x86-64, Zen3 (MILAN)	ARMv8.2, TaiSHan v110

Features	Intel Xeon(R) Platinum 8268	AMD EPYC Milan 7763	ARM Hi1620 (Kunpeng 920)
Fabrication Technology	14 m	7 nm	7 nm
Release Date	2019	2021	2019
Number of cores	2x 24	2x 64	2x 48
Number of threads	2x 48	2x 128	2x 48
Base Frequency (GHz)	2.90	2.45	2.6
Number of memory channels	2x 6	2x 8	2x 8
Memory support	DDR4-2933	DDR4-3200	DDR4-2933
Memory Bandwidth (GB/s)	2x 140.8	2x 204.8	2x 187.7
L1 instruction cache (KB per core)	32	32	64
L1 data cache (KB per core)	32	32	64
L2 cache (MB per core)	1	0.5	0.5
L3 cache (MB)	2x 35.75	2x 256	2x 48
Advanced Vector Extension	512-bit AVX-512	256-bit AVX2	128-bit NEON
Processor Interconnect	3x Intel UPI links	4x AMD Infinity links	3x HiSilicon Hydra ports
Thermal Design Power (W)	2x 205	2x 280	2x 158
Input and Output	2x PCIe 3.0 x48	2x PCIe 4.0 x128	2x PCIe 4.0 x40

Table 1: Microprocessors used in the study

2.2.1 Intel Xeon(R) Platinum 8268 (Altair node processor)

Xeon Platinum 8268 is a 64-bit 24-core processor designed for high-performance server use. It was introduced by Intel in early 2019 and can be found in PSNC's Altair supercomputer nodes. Based on the Cascade Lake x86 microarchitecture and manufactured with 14 nm lithography process this chip operates at 2.9GHz base frequency with a turbo boost frequency of up to 3.9 GHz with a TDP of 205W. When it comes to symmetric multiprocessing, it supports 8-way SMP via 3 Ultra Path Interconnect links and up to 1 TB of hexa-channel DDR4-2933 memory. Unusually this particular model features a larger non-default 35.75 MiB of L3 cache that would normally be found on a 26-core part.

2.2.2 AMD Epyc Milan (AMD x86)

The AMD MILAN 7763 CPU represents the third generation of EPYC server processors. The design of this CPU consists of a single central I/O hub (or I/O Die) through which all CPU components communicate. The CPU uses a collection of 8-core chiplets, called Core Complex Dies (CCDs), connected to the I/O Die through dedicated high-speed Infinity Fabric links. Through this die, a given CCD can communicate with other CCDs and the main memory and external devices connected by the PCIe bus.

This top-of-the-line MILAN CPU includes 8 CCDs. Every CCD consists of 8 cores and 32 MB of L3 cache. Every core is equipped with an L2 inclusive cache of 512 KB size and an L1-D cache of 32KB size. The total capacity of the L3 cache depends on the number of CCDs, and can maximally reach 256 MB for a single 64-core CPU. Each processor also includes the 8-channel memory controller providing DDR4 memory speeds up to 3200MHz, where each memory channel supports up to 2 DIMMs.

The CPU design permits simultaneous multithreading (SMT), which results in 128 logical cores for the 64-core CPU. The architecture of MILAN processors offers full AVX2 support enabling single-cycle AVX2 calculations. Every core includes two floating-point units built as four pipes equipped with 2x fadd and 2x Fmul engines. The AMD MILAN 7763 CPU is clocked at the base frequency of 2.45GHz. The maximum boost for a single core can reach up to 3.5 GHz. There is no restriction for CPU frequency when using AVX2 instructions, and the clock speed depends on temperature and voltage requirements regardless of the instructions used.

2.2.3 Huawei Kunpeng 920 (ARM Hi1620)

As mentioned in the Section 2.1, D5.5 contains a high level overview of Huawei Kunpeng 920 processors and the comparison with their predecessor. Thus, we refer the interested reader to D5.5 for further details on the innovative features introduced in Huawei Kunpeng 920.

2.3 Accelerators

Table 2 provides a summary of the characteristics of GPUs used in the testbeds studied in this deliverable. Text below describes architectural novelties introduced in these GPUs.

Features	Nvidia A100-SXM4-40GB	AMD Instinct MI100
Compute units	128	120
FP32 cores	8,192	7,680
Peak INT8 (TFLOPS)	624	184.6
Peak FP16 (TFLOPS)	312	184.6
Peak FP32 (TFLOPS)	19.5/156 (tensor core)	23.1
Peak FP64 (TFLOPS)	9.7/19.5 (tensor core)	11.5
Bus Interface (GB/s)	64 (PCIe Gen4) and 600 (NVLink)	64 (PCIe Gen4)
Global memory (GB)	40 (HBM2)	32 (HBM2)
L2 cache (MB)	40	8
Shared memory per MP (KB)	48	-
Memory interface	5,120-bit (HBM2)	4,096-bit
Memory bandwidth (GB/s)	1,555 (HBM2)	1,200
Max thermal design power (W)	250	300
Board Form Factor	full-height, up to 8 slots	full-height, dual slot
Compute APIs	CUDA, OpenCL, OpenACC	ROCm, OpenCL, OpenACC

Table 2: GPUs used in the study

2.3.1 Nvidia Ampere A100

Architecture-wise Nvidia Ampere A100 GPU is composed of HBM2 memory controllers and 7 GPU processing clusters (GPCs), with 7 or 8 texture processing clusters (TPCs) per GPC and 2 streaming multiprocessors (SMs) per TPC. In total, it gives 128 SMs per full GPU, each with 64 FP32 CUDA Cores per SM and 4 third-generation Tensor Cores per SM [14-16].

The new Ampere SM adds many new capabilities to the features originally introduced in the Volta and Turing SM architectures. The major improvements include the innovative third-generation Tensor Core, support of multi-instance GPU (MIG) virtualization for shared GPU use, larger HBM2 memory and L2 caches, faster multi-node and multi-GPU interconnects, as well as significant changes in CUDA.

The third-generation Tensor Core enhances operand sharing and improves efficiency, as well as adds support for new DL and HPC data types together with a new Sparsity feature. The list of the new and better supported data types includes new TensorFloat-32 (TF32) Tensor Core operations, new Bfloat16 (BF16) Tensor Core instructions operating at the same throughput as FP16, IEEE-compliant FP64 processing, Tensor Core acceleration of INT8, INT4, and binary round out support for DL inferencing. The new types allow to boost performance significantly. In particular, with the new TF32 operations, Nvidia reports 10x faster processing than V100 FP32 FMA. New Sparsity support enables exploiting a fine-grained structured sparsity in DL networks. This is achieved through 2:4 sparse matrix definitions that prune every four-entry vectors to two non-zero values.

In order to facilitate fine-grained workload provisioning for smaller workloads, the third-generation Tensor Cores implement a new multi-instance GPU (MIG) virtualization feature which allows the Tensor Core GPU to be securely partitioned into up to 7 separate GPU Instances for CUDA applications with separate and isolated paths through the entire memory system. In addition, single root input/output virtualization (SR-IOV) allows to share and virtualize a single PCIe connection for multiple processes or VMs.

In order to increase number of links per GPU and raise GPU-GPU communication bandwidth, Ampere GPUs use the new Nvidia NVSwitch and the third-generation of Nvidia high-speed NVLink interconnect. The latter also implements an improved error-detection and recovery. For multi-node communication, these GPUs are fully compatible with Nvidia Magnum IO, as well as Mellanox InfiniBand and Ethernet interconnects. In particular, they support PCIe Gen 4 for fast network interfaces, such as 200 Gbit/s InfiniBand.

A100 cards hold 40MB of L2 cache and 40GB of HBM2 memory with a class-leading 1555 GB/s of memory bandwidth. They support asynchronous copy and hardware-accelerated barriers in shared memory.

The new version of CUDA for Ampere cards introduced task graphs model for submitting work. A task graph defines series of operations (i.e., memory copies and kernel launches), connected by dependencies. It enables define-once and run-repeatedly execution flow where predefined task graph allows the launch of any number of kernels in a single operation.

For more detailed information on the Nvidia Ampere A100 we refer to [14].

2.3.2 AMD Instinct MI100

Deliverable D5.5 provides an overall tabular summary for GPGPU cards like AMD MI50 and MI100 [1]. In this elaboration we deliver more detailed description of the card AMD MI100 card discussing its improved performance over previous generation MI50.

AMD Instinct MI100 is based on next-generation CDNA architecture optimized for computing operations. In comparison MI50 is on Vega 20 architecture, which is specific for consumer cards targeting graphics rendering performance. Both accelerators are installed on AMD cloud testbed, which contains nodes with 8 cards each. We will now compare the two AMD cards.

Both AMD Instinct cards are fabricated at 7nm FinFET process and draw up to 300 Watts of power. MI100 features a total of 7680 stream processors operating at 1.5GHz and its FP64 performance peaks at 11.5 TFLOPs. Finally, MI100 card provides 32GB of HBM2 memory delivering 1.23TB/s of bandwidth. In turn the MI50 peaks at 10 TFLOPs, consists of 3840 stream processors at 1.725GHz, while having 16GB of HBM2 memory peaking at 1TB/s of bandwidth.

The newer architecture comes with AMD matrix core technology, which optimizes matrix datatype calculations. Thanks to it, MI100 comes with 7x greater peak FP16 performance than the MI50 (at 185 TFLOPs compared to 26.5 TFLOPs) [17].

Although it fits into a standard PCIe Gen4 x16 slot, The Instinct MI100 card is using Infinity Fabric Link. It is an additional interface for GPU-to-GPU communication much like Nvidia’s NVLink. This provides more performance than PCIe Gen4 interface and is designed to link up to four GPUs in a virtual GPU called "hive". MI100 GPUs can be configured with up to two fully-connected quad GPU hives, each providing up to 552 GB/s of P2P I/O bandwidth for fast data sharing. MI50s can also be connected with Infinity Fabric, but it can only be hooked into a ring topology, which increases the latency between devices [18].

Set side by side with Nvidia's accelerators, AMD offerings provide roughly similar performance to power efficiency ratio. Comparing to Volta V100, the Instinct MI100 offers a 19.5% uplift in FP64 and an 18.5% uplift in FP32 performance. However, in FP16 performance, the newer and more power-demanding Nvidia Ampere A100 has a 69% advantage over the Instinct MI100 [19].

2.4 Testbed configurations

Table 3 summarizes information about hardware and software configuration of the testbeds used to evaluate modern general-purpose CPUs. In this study, we evaluate state-of-the-art CPUs on 3 testbeds, each corresponding to the CPU described in subsection 2.2.

Feature	intel8268	amd7763	armHi1620
Node merchant name	HUAWEI CH121 V5	GIGABYTE H262-Z62-00	Atlas 800 (Model: 3000)
CPU	Intel Xeon(R)	AMD EPYC Milan	ARM Hi1620

Feature	intel8268	amd7763	armHi1620
	Platinum 8268		(Kunpeng 920)
RAM	192/384GB	512GB (8-Channel RDIMM/LRDIMM DDR4)	256 GB
SSD/HDD	2 x SAS 1.6 TB (OS and HOME discs) 2 x SSD 1.4 TB (scratch discs)	240G (Intel SSDSC2KB24)	2 x SAS 1.6 TB (OS and HOME discs) 2 x SSD 1.4 TB (scratch discs)
Interconnect	Infiniband EDR	HDR200 IB Mellanox MT28908 [ConnectX-6]	Infiniband EDR
OS	CentOS 7 (Core)	CentOS 8.3.2011	Ubuntu 18.04.5 LTS (Bionic Beaver)
FS	GPFS	BeeGFS 7.2.1	GPFS
MPI	OpenMPI 4.1.1	OpenMPI 4.1.1	OpenMPI 3.3a2
C/C++/Fortran	GCC 9.3.0	GCC 9.3.0	GCC 7.5.0
Python	Python 3.9.2	Python 3.7.9	Python 3.7.10
Access	PSNC, Eagle:altair	AMD Milan cloud	PSNC, SSH tunnel, proxy Eagle

Table 3: Hardware and native software configuration of the CPU testbeds used in the study

intel8268 testbed. In order to study the performance of Intel Xeon Platinum 8268 CPUs, we use nodes of PSNC's Altair clusters composed of HUAWEI CH121 V5 servers connected via the InfiniBand EDR interconnect. We refer to these nodes as intel8268 testbeds. Every intel8268 testbed offers two 24-core Cascade Lake CPUs and 192 or 384 GiB DDR4-2933 of main memory. This testbed come with CentOS 7.9 operating system and Lustre 2.12 file system. intel8268 testbeds can be accessed from the `altair` queue of Eagle.

amd7443 testbed. We study the performance of AMD EPYC Milan 7763 processors by utilizing the resources of the GIGABYTE H262-Z62-00 server of the external AMD cloud. We refer to it as amd7763 testbed. Besides two top-of-the-line 64-core CPUs, this testbed is equipped with 512 GiB DDR4-3200 of main memory and configured with NPS4 (Numa Per Secocet) setup exposing 2x4 NUMA domains per a single node. The underlined testbed comes with CentOS 8.3 operating system, BeeGFS 7.2.1 file system, and offers the InfiniBand HDR200 interconnect.

armHi1620 testbed. An Atlas 800 Model 3000 server hosted at PSNC is used to study the performance of ARM Hi1620 (Kunpeng 920-4826) processors. We refer to this server as armHi1620 testbed. A single node offers two 24-core ARMv8.2-based CPUs, 256GiB of DDR4-2933, and two 1.2TB 2,5" SAS Seagate HDD. This testbed is connected through Quad-Port (2x10GbE + 2x1GbE) card, and comes with Ubuntu 18.04 operating system.

Respectively, Table 4 summarizes information about hardware and software configuration of the 2 GPU testbeds used in the study.

	nvA100	amdMi100
Node merchant name	HPE Apollo 6500 ProLiant Gen10 Plus XL675d	Supermicro AS-4124GS-TNR
GPU	8 x Nvidia A100 SXM4 40GB PCIe	8 x AMD Instinct MI100 32 GB PCIe
CPU	2 x AMD EPYC 7702 Rome	2 x AMD EPYC 7742 Rome
RAM	1TB (8 x 128GB LRDIMM @ 3200 MT/s, HPE DDR4)	512GB (16x 32GB Samsung 3200 MT/s, DDR4)
SSD/HDD	14TB + 2 x 1.92TB (NVMe PCIe KCD6XLUL1T92)	1TB NVMe PCIe Samsung SM981/PM981
Interconnect	HDR200 IB Mellanox MT28908 [ConnectX-6]	HDR200 IB Mellanox MT28908 [ConnectX-6]
OS	CentOS 8.2.2004	Ubuntu 19.04
FS	Lustre 2.12.6ddn43	ext4
GPGPU API	CUDA 11.4.0	AMD ROCm v4.5
MPI	OpenMPI 4.1.1	MPICH version 3.3a2
C/C++/Fortran	GCC 9.3.0	GCC 7.5.0
Python	Python 3.8.3	Python 3.6.9
Access	HLRS, Hawk:rome-ai	AMD Mi100 cloud

Table 4: Hardware and native software configuration of the GPU testbeds used in the study

nvA100 testbed. In order to study the performance of Nvidia Ampere A100 solution, we use the GPU accelerated nodes of HLRS' Hawk clusters, which are composed of HPE Apollo 6500 Gen10 Plus XL675d servers connected via the Dual Rail InfiniBand HDR200 interconnect [20,21]. For the sake of brevity, we refer to these HPE Apollo 6500 Gen10 Plus XL675d servers as nvA100 testbeds. nvA100 testbeds can be accessed from the `rome-ai` queue of Hawk. Each nvA100 testbed has 8 Nvidia Ampere A100 SXM4 40GB PCIe cards each. Besides Nvidia

Ampere A100 GPUs, nvA100 testbeds include 2 64-core AMD EPYC 7702 Rome CPUs, 1TB of HPE DDR4 SmartMemory, 14TB of DDN EXAScaler storage with IME (Infinite Memory Engine) burst buffer, and 2 NVMe PCIe SSD cards with 1.92TB each. Our nvA100 testbeds come with CentOS 8.2 operating system and Lustre 2.12 file system. The default software environment includes GCC 9.3.0 compilers, Python 3.8.3 interpreter, and CUDA 11.4.0 for writing GPGPU applications.

amdMi100 testbed. To examine the performance of AMD Instinct MI100 GPUs, we use a system of external AMD Accelerator Cloud, which is composed of Supermicro 4124GS-TNR servers connected via the InfiniBand HDR200 interconnect. Every server includes two 2x AMD EPYC 7742 Rome, 512 GiB of DDR4-3200 main memory, 1TB NVMe PCIe SSD, and offers up to 8x AMD Instinct MI100 GPUs. We refer to every server as amdMi100 testbed. The amdMi100 testbeds come with Ubuntu 19.04 operating system.

3 HiDALGO Benchmark Suite

In this chapter, we present the structure of the HiDALGO benchmark suite. We start by discussing the computationally expensive parts in the simulation data flows of pilots. Next, we describe the purpose and implementation for each of these kernels. Afterwards, we present the implementation of the benchmark itself. Namely, we describe the methodology and tools for collecting measurements and reporting results, as well as the scripts that automate the process of deployment and running the benchmark. We finish this chapter by comparing the HiDALGO benchmark suite with other benchmarks suitable for GSS on HPC, such as SPEC and CoeGSS benchmark [22,23].

3.1 Structure of the HiDALGO Benchmark Suite

The HiDALGO benchmark suite is composed of three components: kernels to benchmark, automation scripts, and codes that implement benchmarking methodology. We host codes of the HiDALGO benchmark suite at GitLab repository:

https://gitlab.com/eu_hidalgo/hidalgo_bench_suite.

The core component of the benchmark is a set of kernels, which reflect the computationally expensive parts in the simulation data flows of our pilots. All kernels are put in the subfolders of the folder `tests`.

In the HiDALGO benchmark, we put a strong emphasis on automating deployment of the benchmark, running benchmark, and collecting metrics for all kernels and testbeds. The main automation scripts are implemented in `Ansible` and `Spack` and can be found in the folder `ansible`.

Last but not least, the HiDALGO benchmark relies on the metrics collected by three tools – `time`, `perf`, and `mpiP`. In the CPU benchmark, we collect various metrics on 16 cores and on a full testbed for each kernel. This allows to draw initial conclusions about the overall performance of the testbeds, as well as about the performance of the cores. In GPU benchmarks, we fully utilize a single GPU card. In order to harmonize and facilitate collection of the metrics and analysis of the benchmarking results, we develop a set of scripts in Python and bash. These scripts are located in the folder `tests/common`.

In the next sections, we cover these three components of the benchmark in depth. The last section is devoted to comparing our benchmark with the existing alternatives.

3.2 Kernels of the HiDALGO Benchmark Suite

3.2.1 Hotspots in the simulation data flows of pilots

In Chapter 2 of D5.5 [1], we covered simulation data flows of pilots. Namely, we discussed an overall idea on how applications spread across processes and communicate with each other, feed models with data, synchronize statuses and write results down. Data flows of all pilots contain simulation components, as well as pre- and post-processing components. In this section, we identify compute-intensive parts of the data flows, which provide the baseline for defining kernels of the HiDALGO benchmark suite discussed in the next section.

In order to forecast movements and destinations of people displaced by conflict, migration use case follows an agent-based modelling approach. The overall process includes several phases. In the first phase, the data from all sources is collected, cleaned up, and prepared for the simulation runs. Most of the computational efforts are spent on the GIS pre-processing step known as location graph extraction. In location graph extraction, the target is to produce a reduced form of the route network, called location graph, from the raw input OpenStreetMap (OSM) data. In the second phase, we feed pre-processed data to the ensemble run of parallel Agent-Based Social Simulations (ABSS) performed by ABSS framework called Flee. Flee runs agent-based simulations and reports aggregate agent totals. In the last phase, outputs of Flee are post-processed by the VVUQ tools. Compared to the fists two phases, the latter phase does not require significant computational resources.

The data flow of the UAP use case is driven by the traffic and CFD simulations. Similarly to the migration use case, UAP requires GIS pre-processing. Nevertheless, the scale of the GIS data is relatively small compared to the migration pilot. In the simulation phase, the CFD simulation takes significantly more computation time compared to the traffic simulation. In order to benefit from different computing platforms, SNA pilot experiments with several CFD simulation tools including OpenFOAM, Fluids-GPGPU, and RapidCFD-GPGPU. CFD simulations produce a large amount of outputs for further post-processing, where principal component analysis (PCA) for mode reduction is one of the most expensive post-processing routines. PCA implementation is based on the singular value decomposition (SVD). There are many implementations of SVD for different hardware platforms. In our studies, we rely on the implementations from the TensorFlow and Magma libraries which allow to compute SVD with different GPU accelerators.

The data flow of the SNA use case also includes several computationally expensive tasks such as approximation of the eigenvalue histogram, as well as simulation of the message spread in social media. The first task is implemented in the KPM tool, while the second task is carried out by the SN-Simulator tool.

Based on the analysis of the data flows of the HiDALGO pilots, we defined 5 CPU and 3 GPU kernels for the HiDALGO benchmark suite. These are summarized in Table 5 and Table 6.

Kernel	Domain	Language	Pilot	Purpose
Flee	Simulation/ABSS	Python	MIG	Simulation of human migration
OpenFOAM	Simulation/CFD	C++	UAP	CFD simulation of UAP
SNSimulatior	Simulation/ABSS	Python	SNA	Simulation of message spread in social media
LGE	Pre-process/GIS	C++	MIG	Location graph extraction
KPM	Pre-process/LA	Python	SNA	Kernel-Polynomial-Method for approximation of the eigenvalue histogram

Table 5: List of the CPU kernels in the HiDALGO benchmark suite

Kernel	Domain	Language	Pilot	Purpose
Fluids-GPGPU	Simulation/CFD	C++	UAP	CFD simulation for UAP
RapidCFD-GPGPU	Simulation/CFD	C++ (CUDA)	UAP	CFD simulation for UAP
SVD	Post-process/LA	C	UAP	Singular value decomposition

Table 6: List of the GPU kernels in the HiDALGO benchmark suite

3.2.2 Simulation components of the benchmark

3.2.2.1 Migration use case

Flee kernel. Flee is an ABSS framework developed by BUL, which performs the agent-based social simulations for the Migration use case. We focus on P-Flee, the parallel algorithm of the framework. The Flee code calculates the movement of displaced agents on a daily basis with a total number of simulation days *D*. The simulation starts with an initial number of agents *N*, while at every new timestep, *d* agents can be added. The environment is modelled by an attributed weighted graph of routes between locations, i.e., the location graph, denoted as *G*. Flee implements two parallelization schemes, one that replicates the location graph and distributes agents evenly among processes (agent-parallel, AP), and another (agent-space-parallel, ASP) that distributes both locations and agents among processes, with each process being responsible for the update of its "local" locations.

In our benchmark, the Flee kernel takes the following inputs:

- Flee parallelization mode: We focus on the ASP version of the Flee algorithm, as prior evaluation has shown that it is effective. It also allows us to evaluate the performance of Flee on a single node with a smaller location graph as input.
- Location graph *G*: We use a synthetic location graph of 100 nodes with a vertex degree of 4, denoted as 10-10-4.
- Number of agents *N*, *d*: We perform a simulation with 2 million initial agents (*N*), adding 10 thousand agents (*d*) per time step.
- Simulation days *D*: We perform a simulation with 10 time steps, equivalent to 10 days.

3.2.2.2 Urban Air Pollution use case

The most computationally intensive part of the UAP Pilot is the CFD module. The task here is to compute air flow from weather data and have it interacted with traffic emitted pollution. The city model, coupled weather and traffic data is accumulated, converted to a format readable by the implementing software. After simulation, the appropriate physical state data is stored for post processing. On novel CPU architectures, the original source code version v20.12 of OpenFOAM is recompiled, which is currently used in production for CFD simulations. For GPU architectures, a new, internally developed GPGPU code named "fluid solver" is introduced and tested, which is specifically aimed for GPU usage. A GPU variant of OpenFOAM named RapidCFD is also tested to assess the porting possibilities of the OpenFOAM version to GPUs.

OpenFOAM kernel. The OpenFOAM kernel uses the open source CFD toolset OpenFOAM with the community versions 20.06 or 20.12. The software has been extended with modules to handle the special data formats of the UAP Pilot. Also, most used simulation tools are instrumented and tweaked for our use case.

The module runs in a 3-step way. First, UAP format data is converted to OpenFOAM format: the 3D simulation domain and time dependent atmospheric and traffic data is converted to

OpenFOAM `polyMesh` and `dict` formats to be read by the simulation tools afterward. Next, domain decomposition is issued for as many number of domains, as the number of processors used. The third part will be run in parallel, which includes renumbering the mesh with `renumberMesh` to get physically close cells closer in index, too. The physical simulation is started with calculating a steady state with `simpleFoam` for the starting time: boundary conditions and emission rates are fixed. Next, a `changeDictionary` tool is executed to set up transient parameters within the simulation data. Finally, the actual time dependent simulation is done with pimpleFoam.

The simulation expects the following *inputs*: the 3D model of the simulation domain in Fluent `.msh` format and the wind profiles for the boundary condition of the flow field in `.wind` format. At least one pollutant is recommended as traffic emission in UAP table format, although additional point sources can be added in the pointSources file. Custom background emission profile can be set with the UAP table format. Sampling locations are set in the probeLocationsFull file. Further simulation parameters can be configured in the in the `uap_foam.cfg` file.

Output format is configured in the `uap_foam.cfg` file, that include `.ensight`, `.vtk` and `.csv` formats for 3D outputs. Probe location values are saved in ascii format, while configurable cutting plane outputs are in `vtp` format. Outputs may include flow field velocity as vector, pressure, and pollutant concentration for the configured pollutants.

Fluids kernel. `fluid_solver` is a 3 dimensional computational fluid dynamics solver, based on a cell-based finite volume method, that can simulate complex flow problems. It is compiled as C++, but mostly follows a C99 coding style (with the exception of operator overloads) in order to retain a sane ABI (Application Binary Interface). It is first and foremost a command line tool which, given a boundary condition, configuration, and tetrahedral mesh, can simulate the problem and write result as raw data (state matrices), probe data (useful for acoustic analysis for instance) or visualization ready files (`.vtk` format for Visualization Toolkit). It is a finite-volume, cell-based CFD solver that works on tetrahedral meshes. It currently supports a FOM (full-order model) and ROM (reduced-order model) mode which runs orders of magnitudes faster (x1000, x2000 faster). It currently solves the compressible Navier-Stokes equations, mainly using the Explicit-Euler method. It also has an emission module, where concentration can be computed for an arbitrary amount of substances.

In order to run the solver in the Reduced-Order-Model (ROM) mode, a training step is required. During this training step, multiple Full-Order-Models (FOMs) are computed, the flow states are placed into a matrix (rows correspond to cell indices, and columns correspond to timesteps), and the SVD (Singular Values Decomposition) of those flow values are computed. Using the result of that SVD, we use a POD in order to get reduced-order modelling capabilities. The precision of this modelling can be adjusted dynamically, based on the number of considered singular values (corresponding to the dimension of the reduced-order model).

There are currently two versions. A multicore CPU version that runs on a single node via OpenMP, and a CUDA version that runs on Nvidia GPUs. The CUDA version only relies on cuBLAS, everything else is written as simple CUDA kernels Most of the computations in our solver involve iterating over a huge number of vertices/faces/cells (approximately 10 million), where we compute the flux value for each face/cell, repeatedly executing the same routines over and over for each of them. This is a perfect fit for us, for our GPU version, since that is by definition, what GPUs are best at. Thus, our most performance/resource intensive CUDA kernels in `fluid_solver`, are the 3 CUDA kernels that handle flux computations; for the vertices, faces, and cells.

For benchmarking purposes, we always consider a full-day (24 hours). If it would take too much time to simulate, we run the solver for less time, and estimate the results for a full-day instead. It is only needed to do this for the FOMs on CPUs, since the ROMs run extremely fast, and the FOMs run decently fast enough on GPU architectures.

There are many parameters needed in order to run `fluid_solver`, many of which we keep constant while benchmarking. We list below the values that have the most significant impact while benchmarking:

- `CFL`: Courant-Friedrichs-Lewy number for the FOM. Before the ROM is started, there is a pre-simulation FOM phase, of which the duration is given by the parameter `steady_state_time`. Keeping this value equal to 0.85 is recommended.
- `ROM_CFL`: Courant-Friedrichs-Lewy number for the ROM, which can be in the order of thousands (1000, 2000), due to the overall stability of the reduced-order model.
- `reduced_r` : dimension of the ROM, which can go as low as 10. It determines the precision of the reduced model.
- `vtk_save_rate`: Save rate for `.vtk` files.
- `state_save_rate`: Number of simulation states to save. These are saved to a custom binary format that can later be used to compute wind-field and other statistics.

All these values determine a compromise between speed and accuracy, thus they are the values we are actually interested in.

During our benchmarks, we kept `reduced_r` constant, while modifying `ROM_CFL`. Modifying the `ROM_CFL` greatly improves the runtime of our solver, since it directly influences the required amount of time needed to simulate a set amount of time. If the CFL value is twice as high, we roughly expect the solver to simulate twice as fast. Of course, this might not perfectly be the case, but in our experience, it roughly scales as such. Precision barely seems to degrade for higher CFL values (sometimes, the opposite being the case: precision might even be better), up to a certain threshold (ex. 2500, 3000).

RapidCFD kernel. One of the ways to use GPUs for the UAP CFD module is to implement the OpenFOAM functionality using one of the OpenFOAM variants that use GPU. Three of these variants were considered: using PETSc solver via the `PETSc4Foam` module, using the GPU solvers in `foam-extend` or using RapidCFD, a variant that is built for GPUs directly. The first two variants only use GPUs for solving the discretized system of linear equations, so data must be transferred from and to the GPU for every system solved. In RapidCFD, however, all calculations are done on the GPU, so RapidCFD is chosen to be tested to assess, if it is worthwhile to port UAP to.

In the test, the solver simpleFoam is tested and run in the well-known test case `motorBike`, that is available in all OpenFOAM variants and versions. As RapidCFD only implements solvers, all other utilities, like domain decomposition, mesh generation or mesh import utilities are used from OpenFOAM version 2.3.1, which is compatible with RapidCFD on the data structure level. For the benchmarks, the mesh generation routine of the original tutorial was modified to get various size meshes. These were generated separately, converted to Fluent `.msh` format, and were reread with `fluent3DMeshToFoam` at the start of the simulation. Afterwards, the potentialFoam and simpleFoam solvers were executed from the RapidCFD variant, which executed the simulation on the GPU.

As it was used only for test purposes, only the mesh file was considered an input, and the simulation state after the final iteration was considered output.

3.2.2.3 Social Network use case

SNSimulator kernel. At the heart of the Social Network pilot lies the so-called SN-Simulator, which has the task of simulating the spread and flow of messages in some given input network. On a high level, the SN-Simulator works as follows. First, it needs to be supplied with a set of input files that contain the network structure and some additional information about the dissemination behaviour of the messages (further description follows below). Next, multiple messages are initiated in the network and for each such message the SN-Simulator determines which neighbouring nodes need to adopt this message and aid in the spread of this message (i.e. retweet the message in the context of twitter). Finally, some metrics about this simulation process, for example the average number of retweets, are calculated and stored. The goal is for this simulation process is to closely mimic the message-spreading behaviour of real-word social networks.

Throughout our benchmarks we will focus on determining the efficiency of the second task, the simulation of the message spread itself. As all of the SN-Simulator, it is written completely in Python and relies on the python packages `numpy`, `scipy` and `pandas`. At the core of this simulation lies the function `edge_sample` which is invoked each time a tweet/message arrives at a node in the network (in the following we will use the terms tweet and message interchangeably). This function is called each time a tweet arrives at a user in the network and is used to determine which neighbouring users in the network will retweet this tweet, allowing it to spread further through the network. Recently we made a significant improvement to this hotspot by using the JIT compiler of the package `numba` in order to increase its performance (see Section 5.3 of D3.5 [13] for a detailed description and analysis).

In order to simulate the spread of messages, multiple *inputs* must be provided. Below, we list all the required input parameters and datasets that were used as part of the benchmark:

- Graph: A network, modelled as a graph, must be provided in either `.metis` or `.npz` format. Used as network in which the flow of messages is simulated.
- Simulation Features: A list of tweets. For each tweet a so-called feature class is specified. Each feature class describes certain tweets, for example, there is a feature class for tweets that contain a link and originate from a verified twitter user.
- Sources and Samples: Two parameters which determine the amount of tweets that are simulated. For each feature class (in our data-set we usually have 100-200 such classes), we simulate sample tweets per source.

With sources and samples we can control the overall amount of work performed by the simulation. In our experiments we fix these values to 400 and 1000, respectively. As input graph and tweets we use data that we crawled from twitter. We consider three different topics, each consists of a set of tweets about a certain topic. Additionally, each data-set includes a follower-followee relationship graph of twitter users talking about this topic.

- neos: A smaller data-set, which consists of roughly 4,500 tweets about the austrian neos political party. It was acquired during the 2019 austrian elections.
- fpoe: A larger data-set, which was acquired at the same time as neos. However, in this case it consists of approximately 19,000 tweets about the FPÖ political party.
- covid19: A large data-set, which consists of roughly 375,000 tweets about the covid19 social distancing regulations. It was acquired at the beginning of 2020.

These data-sets can be found on the Hidalgo CKAN. The input graphs can be found in https://ckan.hidalgo-project.eu/dataset/metis-outer-graphs and the link to the corresponding simulation features is https://ckan.hidalgo-project.eu/dataset/simulation-features-version-2.

The simulation *outputs* a `.csv` file, which contains metrics for each feature class of tweets. It lists `mean_retweets`, the average amount of times a tweet was retweeted, and `retweet_probability`, the probability that a tweet of some user was retweeted at least once by one of the user's followers. Additionally, our simulation also outputs an error metrics: for each feature class the deviation in `mean_retweets` and `retweet_probability` from the corresponding values of the ground-truth data is specified. This ground-truth information can be calculated from the simulation features input file which we described above.

3.2.3 Pre- and post-processing components of the benchmark

3.2.3.1 Migration use case

LGE kernel. Automated location graph extraction (LGE) is the most computationally expensive part in the pre-processing phase of the migration and epidemiology use cases. The algorithm for location graph extraction consists of two steps – computing distance matrix and triangular pruning [24].

In our benchmark, we compute distance matrix by utilizing Table Service of the Open Source Routing Machine (OSRM). This service calculates pairwise distances between locations of interest via state-of-the-art batched shortest path algorithms for routing – multilevel Dijkstra's (MLD) and contraction hierarchies (CH) – in a time complexity of $\mathcal{O}((|E_G|+L_G \log L_G)L)$, where L denotes the number of locations, while E_G and L_G correspond to the number of edges and vertices in the original route network. Both algorithms consist of pre-processing and query phases. The pre-processing phase attempts to annotate and simplify the complicated route network in order to drastically reduce the duration of further shortest-path and batched shortest-path queries. MLD belongs to the family of separator-based shortest-path techniques. Conceptually, it differs from the celebrated customizable route planning (CRP) algorithm only by the hierarchical partitioning approach used in the pre-processing phase: canonical CRP applies patented graph partitioning with natural cuts (PUNCH) approach, while MLD opts for inertial flow approach. Contraction hierarchies is a classic hierarchical shortest-path algorithm, widely discussed in the literature. In our benchmark, the distance matrix is calculated using the contraction hierarchies (CH) algorithm. Pre-processing phase of the CH algorithm consists of two stages – extraction of the inputs for routing with `osrm-extract` utility and contraction of the inputs with `osrm-contract` utility. `osrm-contract` utility produces OSRM-file which is further used by OSRM library to compute distance matrix.

As soon as the OSRM-file is ready, we use it to compute distance matrix and apply triangular pruning to this matrix for obtaining resulting network. Triangular pruning implements a multi-threaded version of the route pruning algorithm for undirected graphs described in [24], as well as in D6.5 [7]. This algorithm has $\mathcal{O}(L^3)$ complexity. The codes of the location graph extraction routine are available from the folder `tests/lge/src`. In our benchmark, we compute distance matrix via the direct C++ interface of the core OSRM library instead of the default REST API to access its services. Our tests rely on OSRM version 5.24.

As a result, the whole process of obtaining location graph utilizes three application which correspond to three different tasks:

- `osrm-extract` for extracting route network for the given network connectivity profile (in this benchmark we use `car.lua` profile from the default example list of the OSRM profiles),
- `osrm-contract` for annotating and simplifying the extracted route network according to the CH algorithm,
- `lge-extract` for extracting location graph from the simplified route network.

All three applications solve relatively independent tasks. In order to reflect that, we introduced three separate subkernels for these steps into the benchmark. Moreover, in our benchmark, we measure separately the elapsed time for computing distance matrix and for triangular pruning. Note that in contrast to location graph extraction (`lge-extract`), pre-processing phase (`osrm-extract` and `osrm-contract`) should be normally executed once per geo-region, so that pre-processed files can be reused for producing new location graphs on the same geo-region.

LGE kernel requires two types of *inputs*: OSM maps and the list of location. These inputs come from the folder `data`. They are organized in the separate subfolders for each of the tested geographic region (see Table 7). We deliberately excluded Europe and Asia from the benchmark due to the long-time of contraction for these continents. Each subfolder contains file `map.osm.pbf` with the OSM map and CSV file `location.csv` with the list of locations.

Region	Location tags	Number of locations	Map file
South Sudan (conflict)	city, camps	51	africa/south-sudan-latest.osm.pbf
South Sudan	city, camps	1810	africa/south-sudan-latest.osm.pbf
Germany	city, town, village	40204	europe/germany-latest.osm.pbf
Ukraine	city, town, village	26903	europe/ukraine-latest.osm.pbf
Africa	city, town	9807	africa-latest.osm.pbf

Region	Location tags	Number of locations	Map file
Antarctica	city, town	4	antarctica-latest.osm.pbf
Australia & Oceania	city, town	1693	australia-oceania-latest.osm.pbf
Central America	city, town	1948	central-america-latest.osm.pbf
South America	city, town	8591	south-america-latest.osm.pbf
North America	city, town	9951	north-america-latest.osm.pbf

Table 7: Inputs for the LGE kernel

Files `location.csv` are directly available at `data`. Information of the `location.csv` files corresponds to the settlements from the OSM maps tagged with place equal to city, town, or village. These files contain 3 columns: OSM ID of the settlements, as well as their latitude and longitude. Preparation of these files was automated by running the following steps:

- reduce the full input OSM map by extracting information about settlements to the smaller OSM file using `osmosis`:

```
export GEO_REGION=africa/south-sudan
osmosis -read-pbf data/$GEO_REGION/map.osm.pbf \
      -tf accept-nodes place=city,town,village,hamlet \
      -tf reject-relations -tf reject-ways -lp \
      -write-xml places.osm
```

- convert the reduced OSM file to the `location.csv` with the Python script `bin/osm2locations_csv.py`.

Alternatively, one could use osmctools to replace `osmosis`. Nevertheless, our experience shows that osmctools extracts more redundant information compared to `osmosis`.

Before running benchmark kernels, files `map.osm.pbf` for the target regions should be downloaded from https://download.geofabrik.de by running the script `bin/fetch_osm.sh`. Parameters of the maps are summarized in Table 8 .

Region	Size	Nodes	Ways	Relations	Restrictions
South Sudan	97MB	23203258	2129495	16	3
Germany	3.76GB	354371881	57866890	138394	126519
Ukraine	642MB	86802157	10650703	14613	19466
Africa	4.83GB	898583514	104069873	11954	18813

Region	Size	Nodes	Ways	Relations	Restrictions
Antarctica	31MB	5068020	140385	0	0
Australia & Oceania	929MB	137625485	10174012	9204	38376
Central America	483MB	70910074	10548086	1817	8609
South America	2.62GB	425337345	32438176	23937	146476
North America	10.91GB	1457364460	127856913	108879	395706

Table 8: Parameters of the input maps for the LGE kernel

Most of the *outputs* are produced by the application `osrm-extract`, which generates 22 separate files with the information necessary for routing (such as names, edges, geometry maneuvers, restrictions, turns, etc). Based on these files, `osrm-contract` produces OSRM file with annotated and simplified route network. `lge-extract` gets OSRM file and the list of locations as input. It outputs the location graph.

3.2.3.2 Urban Air Pollution use case

SVD kernel studies the performance of the singular value decomposition (SVD). In our studies, we rely on the SVD implementations from the Magma library which computes SVD in hybrid CPU+GPU mode, as well as from the TensorFlow framework which also allows to utilize GPU accelerators. Respectively, we define two subkernels TensorFlow and Magma. The TensorFlow subkernel is implemented in Python. In order to compute SVD with TensorFlow, we call the function `linalg.svd()`. The Magma subkernel is implemented in pure C. It uses the function `magma_dgesvd()` to compute SVD for double precision floating point matrix.

In both cases, we feed the functions with randomly generated square matrices of various sizes and compute all their singular values without storing matrices *U* and *V*. In TensorFlow subkernel, we generate the input matrix with the function `random.uniform()` from the core TensorFlow library, while, in Magma subkernel, we use the function `rand()` from `stdlib.h`.

3.2.3.3 Social Network use case

The main task of the SNA pilot is the simulation of message flow in given networks. This requires the input of a social network in form of a graph, which is either (i) sampled from real data (e.g. the twitter relationship graphs we use in the benchmarks of Section 3.2.3), or (ii) synthetically generated. In case the second method is desired it is important to establish that the generated synthetic networks also possess properties of real world social networks. Various metrics can be used for this comparison, for example, edge degree distribution or clustering coefficient. An especially powerful but computationally demanding metric is the eigenvalue spectrum of the graph Laplacian.

KPM Eigenvalue kernel approximates the eigenvalue spectrum of the normalized Laplacian matrix of a given graph. More precisely, it computes an approximation of the histogram of eigenvalues with the so-called Kernel-Polynomial-Method (KPM) [25]. In essence, this approach allows the calculation of the aforementioned histogram via a large number of matrix-vector multiplications, where the matrix is the graph Laplacian and the vector is randomly generated.

The application is written in Python and relies on the `numpy` and `scipy`. Most notably we rely on the sparse matrix-vector multiplication algorithms supplied by `scipy`.

Besides the *input* graph of which the eigenvalue histogram should be computed, three additional parameters that determine the accuracy of the computed histogram need to be specified:

1 Graph: The input graph. Given either in `*.metis` format or in `numpy`'s `*.npz` format. Upon executing the application with a `*.metis` graph for the first time, a `*.npz` file is generated which makes future reading of the input more efficient. For the benchmarks of this section, we relied on the friendship relationship graph of the pokec social network (see https://snap.stanford.edu/data/). It consists of roughly 1M nodes. As concrete input we used a sub-graph of size 50,000 of this graph which was extracted via breath-first search and can be found on the Hidalgo CKAN (https://ckan.hidalgo-project.eu/dataset/pokec-relationship-graphs).

2 Intervals: The number of histogram bins that need to be computed. The computational effort increases linearly with increasing number of intervals.

3 Samples and Degree: Parameters which control the accuracy of the KPM. Empirical test showed that the values of samples>100 and degree>50 are required to achieve good results.

It is important to note that the majority of computational effort is spent performing matrix-vector multiplications. Most of these multiplications can be performed completely independent from other multiplications. However, all these multiplications rely on the same matrix (the normalized graph Laplacian). In order to avoid contention when multiple cores access this matrix in shared memory at the same time, we recently employed ccNUMA optimizations. The idea is to store multiple copies of the matrix in the shared memory of each node to improve performance for processors that have many cores and NUMA domains. See Section 5.3 of D3.5 [13] for a thorough discussion and analysis of this optimization. This improvement may be enabled with the optional parameter `numa_cores`, which specifies the number of cores that share a NUMA domain. The parameter value depends on the processor architecture and should be set to the number of CPU cores per NUMA domain. This number may be displayed via `lscpu`.

The application *outputs* the histogram of eigenvalues of the given graph. More precisely, it states the number of eigenvalue that lie in each of the intervals many bins. In case the input graph is specified in `*.metis` format a `*.npz` file is generated which makes reading the input in subsequent runs more efficient.

3.3 Automation scripts for deployment of the benchmark and collecting results

Automation scripts are an integral part of the HiDALGO benchmark suite. In the HiDALGO benchmark, we automate deployment and installation of the benchmark kernels and their dependencies on the testbeds with Ansible and Spack. Our scripts reuse Ansible roles for Spack and external Spack packages developed in Task 4 of WP6. Section 4.2.1 of D6.6 [8] details the benefits of automated deployment with Ansible and Spack, as well as describes the Ansible roles for installing software with Spack. The main purpose of using Spack is to improve performance of the kernel dependencies by their platform-aware installation from sources, overcome technical limitations for installing software with the conventional package managers (e.g., lack of Internet access and sudo rights, different vendor-specific software, etc), and make the benchmarks reproducible. Notably, Spack allows to get the same versions for the software dependencies of the benchmarked kernels on all testbeds.

Automated deployment scripts are available from the folder `ansible`. In particular, along with the inventory files for the testbeds, this folder contains:

1 subfolder `spack-configs` with Spack configs,

2 playbook `hidalgo_bench_deploy.yaml` for deploying Spack with the HiDALGO benchmark software environments, and

3 playbook `hidalgo_bench_install.yaml` for installing the software environments with Spack.

Content of the `spack-configs` subfolder follows guidelines of the Extreme-scale Scientific Software Stack (E4S) project and includes: the custom Spack packaging scripts missing in the default Spack distribution (e.g., `libOsrm`), the configurations for the testbeds (`config.yaml`, `compilers.yaml`, and `packages.yaml`), and the software stacks of the benchmark. We define separate software stacks for the CPU kernels (`cpu.yaml`) and for the GPU kernels (`gpu.yaml`).

Besides automated deployment and installation scripts, the folder `ansible` contains Ansible playbooks for collecting benchmark results and information about the hardware and the installed software of the testbeds. These playbooks allow to report results in a uniform and precise way, as well as to improve reproducibility. For example, playbook hidalgo_bench_fetch_setup.yaml fetches files spack.lock and hardware.xml from the testbeds. File spack.lock corresponds to the concretized software environments of the HiDALGO benchmarks installed via Spack on the testbed. It comprises exhaustive information about the software packages including their versions, compilation settings, etc. File hardware.xml is produced by the tool lstopo from the package hwloc. This file includes detailed data about the hardware of the testbed. Annex B contains visual representation of these data for the studied testbeds.

In order to simplify work with the above mentioned Ansible scripts, we introduced a wrapper bin/hidBS.sh written in bash. Under the hood, this script makes the following call of the Ansible playbook:

```
ANSIBLE_ROLES_PATH=$ANSIBLE_ROLES_PATH:./core_scripts/ansible/roles \
            ansible-playbook -i ./inventory \
            ./hidalgo_bench_${HIDALGO_BENCH_CMD}.yaml \
            -extra-vars"hidbench_res_dir=results/{{inventory_hostname}}" \
            -l $HIDALGO_BENCH_HOSTS
```

Annex A contains quick start instructions on how to use the automation scripts of the HiDALGO benchmark suite.

3.4 Methodology and tools for collecting measurements and reporting results

For this deliverable, due to the diversity of the available platforms and the number and diversity of HiDALGO kernels to be evaluated, we design an evaluation methodology with the following targets:

- use high-level, command-line tools, that are available on any different system architecture;
- use tools that require zero to minimal adaptation for their setup and execution on different systems;
- collect metrics that are indicative of the applications' performance and resource utilization, where we consider the resources to be the CPU, memory, interconnection network and filesystem;
- collect metrics that can offer a fast-and-accurate comparison of different architectures against an application's performance.

Following the aforementioned targets, we select the following tools for our measurements:

- The Linux `time` command: This program can time or measure the resource usage (memory, I/O, inter-process communication) for any simple program. It is handy for parallel applications as it transparently measures the resource usage for all threads and any child processes.
- The Linux `perf` command: This program can gather performance counter statistics for any simple program. Based on the available performance counters of each architecture, it is able to accurately profile applications and identify performance hotspots.
- The `mpiP` lightweight profiler: This library can collect statistical information about MPI functions. Its ease of use lies in the fact that it intercepts MPI calls and does not require application instrumentation for profiling.

Table 9 lists the metrics we collect and the respective tool to collect its metric. Note that we limit our set of metrics to the intersection of available metrics for different systems. For example, one can easily use perf to collect counter statistics for the last-level-cache utilization, but the respective performance counters are not available across the evaluated architectures, either because the hardware does not offer such capabilities, or because the site administrator has opted to limit access to such capabilities for security reasons.

Metric	Unit	Description	Tool
Elapsed real time	Seconds	The end-to-end execution time indicates the performance	`time`
Instructions	-	CPU instructions can indicate differences between compilers and architectures for the same program	`perf`
Cycles	-	The CPU cycles indicate the performance of the application irrespective of the clock frequency	`perf`
Instructions per Cycle (IPC)	-	The IPC approximately indicates processor performance	`perf` (derived)
Maximum Resident Set Size (RSS)	KBytes	The maximum RSS indicates the memory footprint of the application	`time`
MPI Time	(%)	The percentage of the execution time spent on MPI functions indicates the communication intensity of the application	`mpiP`

Metric	Unit	Description	Tool
Filesystem inputs	-	The number of filesystem inputs indicates the I/O intensity of the application	`time`
Filesystem outputs	-	The number of filesystem outputs indicates the I/O intensity of the application	`time`

Table 9: Metrics collected per application

In order to simplify measuring performance with the above-mentioned tools, we developed a set of wrapping bash functions. These functions are defined in the file `tests/common/bin/functions/time_functions.sh`. These functions can be called from bash scripts as follows:

- At first, `time_functions.sh` should be included and inputs should be specified:

   a. if test is located in the tree of HiDALGO benchmark suite, the minimum setting looks as follows:

```
current_dir=`dirname "${BASH_SOURCE-$0}"`
HBS_TEST_DIR=`cd "${current_dir}/.."; pwd`
. $HBS_TEST_DIR/../common/bin/functions/time_functions.sh
```

   It uses the default format of inputs and outputs. Namely, the name of the kernel corresponds to the name of the folder which holds the script, the outputs are stored in the subfolder `results` of the folder with the test, etc.

   b. the default behaviour can be customized by a set of environment variables. E.g.:

```
HBS_KERNEL=dist_matr_ch
HBS_SUBKERNEL=contract_$OSRM_PROFILE
HBS_DATASET=$GEO_REGION
HBS_TESTCASE=
HBS_TIMESTAMP=$(datetime '+%H%M%S')

HBS_OUTPUT_DIR=.
HBS_TIME_FILE=$HBS_OUTPUT_DIR/summary.csv
HBS_TIME_STDOUT_FILE=test.log
HBS_TIME_STDERR_FILE=test.err
```

2 Afterwards, the benchmarked application should be prefixed with call of the function that wraps the target performance measurement tool (`hbs_time`, `hbs_perf`, or `hbs_mpip` respectively). E.g.:

```
hbs_time osrm-contract $WORK_DIR/$OSM_FILE_BASENAME.osrm
```

As a result, each call of the wrapping function produces log-files with stdout and stderr of the benchmarked application, as well as appends a record to one of the benchmark summary files in a CSV format.

Benchmark summary files hold records composed of header and result parts. The header part is the same for all wrapping functions and includes the following fields:

- testbed name (e.g., "`armHi1620`"),
- number of used cores in the test (by default it uses the output of `nproc`),
- kernel name (e.g., "`lge`"),
- subkernel name (e.g., "`extract_car`"),
- input data set (e.g., "`africa`"),
- input data subset (by default "`0`"),
- timestamp when subkernel was launched in the format "`+%Y%d%m%H%M%S`".

The result part depends on the tool for measuring performance. In case of `hbs_time`, the following fields are reported by default:

- status of the execution (0 if succeeded),
- elapsed real (wall clock) time used by the process in seconds,
- maximum resident set size of the process during its lifetime in KB,
- number of file system inputs,
- number of file system outputs.

This default behaviour is achieved by specifying the format string "`%x,%e,%M,%I,%O`" for `/usr/bin/time` and can be changed by updating an environment variable `HBS_TIME_METRICS` with the new format string.

In case of `hbs_perf`, the summary file reports the following performance counters by default:

- instructions,
- cycles.

This list can be changed by specifying an environment variable `HBS_PROF_METRICS`.

In case of `hbs_mpip`, the summary file contains only one result field which corresponds to the percentage of MPI time.

Measurement collection is automated through the usage of the HiDALGO benchmark suite. Using the three tools – `time`, `perf`, and `mpiP`, – and the respective scripts, three `.csv` files are produced, `hbs_time.csv`, `hbs_perf.csv` and `hbs_mpiP.csv`. All results are stored in the HiDALGO GitLab repository: https://gitlab.com/eu_hidalgo/benchmarking/. Python scripts are provided to analyse data with `pandas` and plot results directly from the `.csv` file in a uniform manner for each application, comparing the different metrics on different systems in the form of barplots. These scripts are available from files `hbs_read.py` and `hbs_plot.py` in the folder `tests/common/python`. In particular, one can read the `pandas` dataframe with the mean values for all metrics collected for `lge-extract` subkernel of LGE kernel with `europe/ukraine` dataset, using the following script:

```
from hbs_read import *
df = read_summary_data(HBSReader('.'),
                       hbs_kernel = 'lge', hbs_subkernel='lge-extract',
                       hbs_dataset = 'europe/ukraine')
```

These results can be further visualized in the form of barplots using the script:

```
from hbs_plot import *
prep_sns()
plot_summary_data(df)
```

3.5 Comparison with another benchmarks

HiDALGO benchmark suite is one of several benchmark suites that can be applied to evaluate hardware platforms for large scale Global System Science (GSS) applications. As an example, in this section we provide details about SPEC and CoeGSS benchmark suites [22,23]. Nevertheless, in contrast to other GSS benchmark suites, HiDALGO benchmark suite is tightly connected to and carefully accounts for the requirements of its use cases.

SPEC CPU suites are popular benchmark suites for testing CPU performance by measuring the run time of several programs. SPEC CPU 2017 package is the latest version of SPEC CPU which is composed of four suites [23]. These suites contain only 10 out of 43 benchmarks which can be applicable to solving subtasks in the data flows of HiDALGO use cases. Moreover, these benchmarks do not cover many computationally expensive parts of the data flows listed in Section 3.2.1. Table 10 summarizes this information. Finally, SPEC CPU 2017 package compares platforms based on elapsed time and throughput (or work per unit of time) only, which does not allow to draw solid conclusions and reason deeply about performance of the testbeds.

Kernel	Domain	Language	Pilot	Purpose
`x05.mcf_r`	Pre-process/GIS	C	MIG	Route planning
`x20.omnetpp_r`	Simulation/ABSS	C++	SNA	Discrete Event simulation - computer network
`x19.lbm_r`	Simulation/CFD	C	UAP	Fluid dynamics
`x21.wrf_r`	Simulation/CFD	Fortran, C	MIG, UAP	Weather forecasting
`x27.cam4_r`	Simulation/CFD	Fortran, C	UAP	Atmosphere modelling

Table 10: Accordance between the kernels of the SPEC CPU 2017 benchmark suite and HiDALGO use cases (x corresponds to 5 for SPECrate suite and 6 for SPECspeed suite)

In contrast to SPEC CPU, CoeGSS benchmark suite is designed specifically for GSS and GC applications [22]. Many of its kernels can be a good alternative to the simulation kernels of the HiDALGO benchmark suite. Nevertheless, this benchmark suite lacks pre- and post-processing kernels relevant for HiDALGO use cases. Table 11 reflects correspondence between the kernels of CoeGSS benchmark suite and HiDALGO use cases. Moreover, similarly to SPEC CPU, the kernels of CoeGSS benchmark suite can be launched on CPU testbeds only.

Kernel	Domain	Language	Pilot	Purpose
IPF	Pre-process/ABSS	C	-	Population synthesis with iterative proportional fitting
ABM4Py/GG	Simulation/ABSS	Python	MIG, UAP	Graph-based ABSS for green growth model
Pandora/GG	Simulation/ABSS	C++	MIG, UAP	Naive ABSS for green growth model
CMAQ/CCTM	Simulation/CFD	Fortran, C	UAP	Community air multiscale quality modelling system
CM1	Simulation/CFD	C	UAP	Model for studying atmospheric processes
OpenSWPC	Simulation/CFD	C	-	Open-source seismic wave propagation code
HRWF	Simulation/CFD	Fortran, C	-	Hurricane weather research and forecasting model

Table 11: Accordance between the kernels of the CoeGSS benchmark suite and HiDALGO use cases

4 Benchmark findings

In this chapter, we outline results of the benchmark. Note that some performance measurement tools are not applicable to all HiDALGO benchmark kernels. In particular, we skip measuring with `mpiP` for LGE kernel and kernels targeting GPU testbeds, since these applications do not use MPI. In addition, some of the applications cannot be ported to all our testbeds. For example, Fluids and RapidCFD require CUDA, which is available on testbeds with Nvidia cards. In our case, it is only nvA100 testbed. Table 12 summarizes information about these restrictions.

Kernel	Testbeds	time	perf	mpiP
Flee	All CPU	√	√	√
OpenFOAM	All CPU	√	√	√
SNSimulatior	All CPU	√	√	√
LGE	All CPU	√	√	-
KPM	All CPU	√	√	√
Fluids-GPGPU	nvA100	√	-	-
RapidCFD-GPGPU	nvA100	√	-	-
SVD	All GPU	√	√	-

Table 12: HiDALGO benchmark matrix

For each application, we focus on comparing the performance and various metrics, on 16 cores and on a full node, where the respective core count varies with the capacity of each system.

Results of the benchmark are uploaded to the folder `deliverable_5_8` of the GitLab repository: https://gitlab.com/eu_hidalgo/benchmarking .

4.1 Migration use case

4.1.1 Experimental setup

All kernels of the migration use case are designed for running on CPUs. Table 13 outlines the software environments which we used on the CPU testbeds with this pilot. C++ codes were compiled with the GCC. In our tests, we applied GCC, Python, MPI, Mpi4py, Numpy, and

Pandas pre-installed by vendors of the testbeds. Boost and OSRM (along with their dependencies) were installed by Spack.

Software	intel8268	amd7763	armHi1620
GCC	9.3.0	9.3.0	7.5.0
CMake	3.18.4	3.19.3	3.18.4
Python	3.7.10	3.8.12	3.7.5
MPI	OpenMPI 3.1.3	OpenMPI 4.0.2	MPICH 3.3
Numpy	1.20.2	1.20.1	1.19.5
Mpi4py	3.0.3	3.0.3	3.0.2
Pandas	1.1.5	1.1.5	1.1.5
Boost	1.74	1.74	1.74
OSRM	5.22	5.22	5.22

Table 13: Software environment for the kernels of Migration use case on CPU testbeds (intel8268, amd7763 and armHi1620)

4.1.2 Results discussion

Flee kernel. We have performed a micro-scale simulation with Flee for 10 days (t=10). We have used a synthetically generated graph, and have simulated a case where the initial number of agents is 2M and 10000 new agents are added per time step. The results are presented in Figure 1, for a synthetic 10-10-4 graph, for three different systems.

We observe that Flee demonstrates the best performance in terms of execution time on amd7763, both when using 16 cores, and when using the full node. The latter is expected, as amd7763 hosts 128 cores, which is more than two times the 40 cores of intel8268. However, the execution time on armHi1620 also drops significantly, but performance is not bound by computation or memory, as indicated by the increase in IPC, in combination with the high percentage of time spent on MPI in this case. As communication occurs over shared memory, this effect could be the result of unoptimized MPI functions, therefore the performance on Kunpeng can eventually be comparable to that of amd7763 and both architectures are a good fit for Flee. Regarding I/O, a high number of filesystem inputs is observed on intel8268 on 16 cores, which is attributed to library initializations and not the application. Similarly, the high number of outputs on amd7763 is also attributed to the software stack and not the application, since it is not observable on other systems.

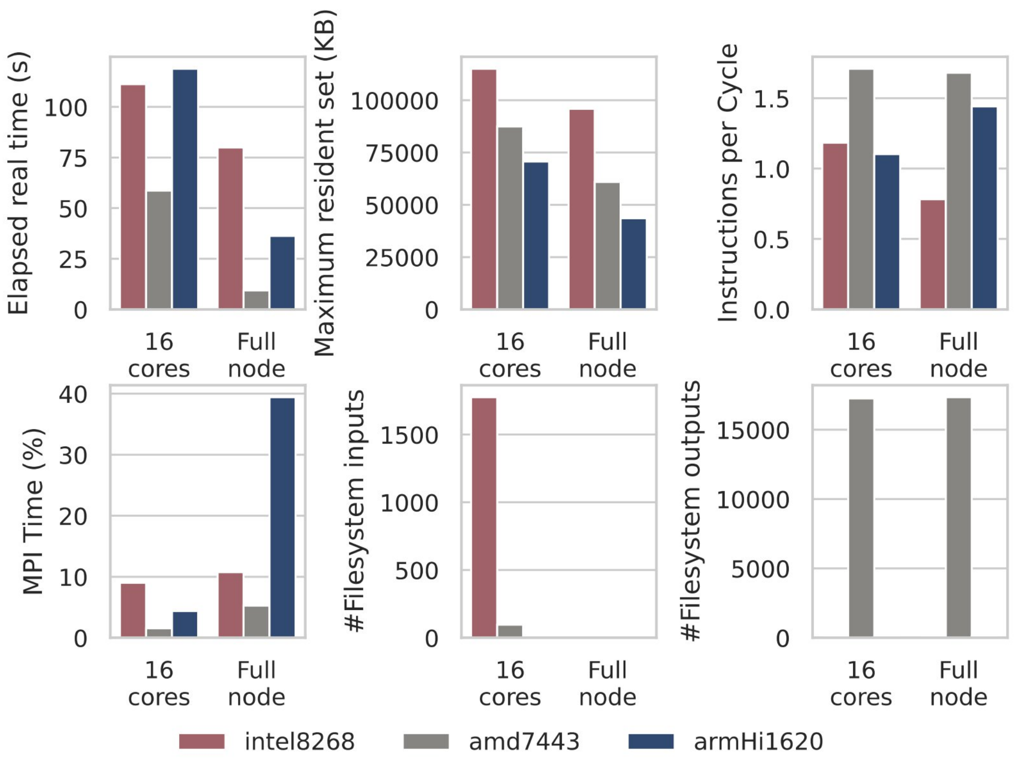


Figure 1. Flee evaluation on CPU testbeds using 16 cores and using all cores of the testbed.

LGE kernel. We have performed experiments for all three subkernels of the LGE kernel with the datasets listed in Table 7. All results are uploaded to the GitLab repository. In this section, we focus on the results for the dataset of Ukraine. In contrast to the small dataset of South Sudan or the big dataset of Germany, this dataset has an optimal size for benchmarking LGE kernel: it is sufficiently large to reveal performance trends for full testbeds, while the complete benchmark execution takes reasonable time. The dataset of Ukraine, the original OpenStreetMap extract hold 642MB in `*.osm.pbf` format. It contains route network with 86.8M nodes, which have to be reduced to the location graph with 26.9K locations corresponding to Ukrainian cities, towns, and villages.

Figure 2 presents results for the `osrm-extract` subkernel for CPU testbeds. amd7763 demonstrates the best execution time on both when using 16 cores, and when using the full node. Nevertheless, the execution time on amd7763 drops only slightly when increasing the number of used cores from 16 to 128. The worst execution time is observed for armHi1620. Moreover, it grows when we switch from 16 cores to the full testbed (96 cores) despite the dramatic increase in IPC. This effect could be explained by noting that `osrm-extract` is an I/O and memory bound application, in conjunction, that armHi1620 is less optimized for memory and I/O operations than other two testbeds. `osrm-extract` produces roughly the same large number of outputs for all testbeds, independent of the number of utilized cores. The RAM consumption is comparable with the number of outputs and increases as the number of utilized cores becomes larger.

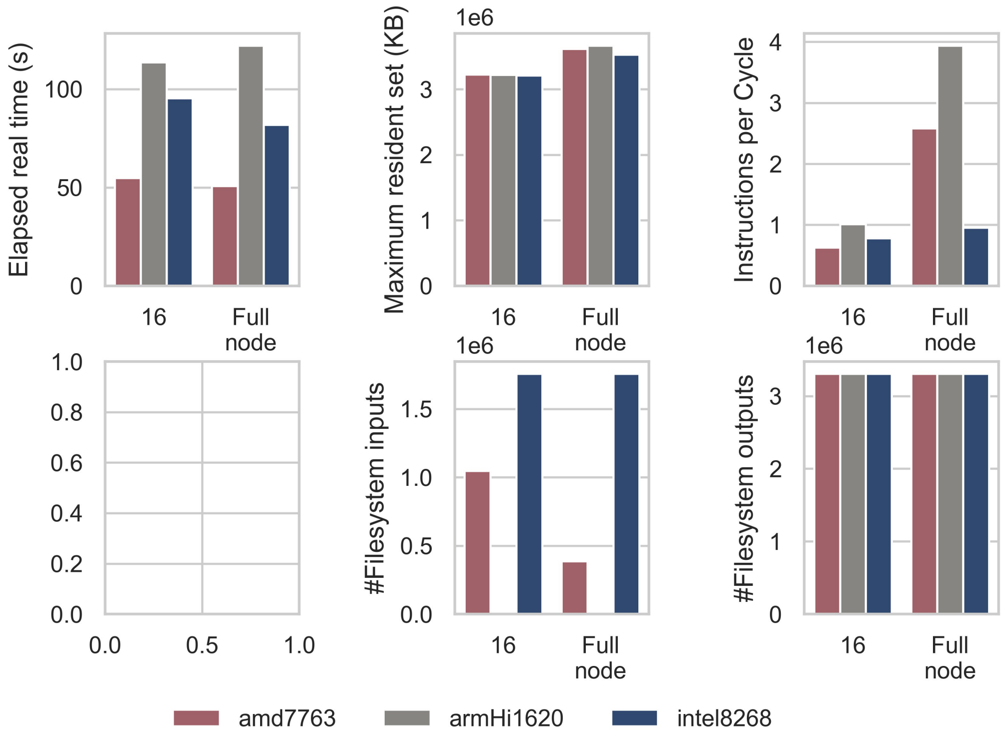


Figure 2. Evaluation of the osrm-extract subkernel of LGE kernel on CPU testbeds using 16 cores and using all cores of the testbed.[1]

Figure 3 illustrates results for the `osrm-contract` subkernel. In this case, the best execution time corresponds to the amd7763 testbed, while the worst corresponds to the intel8268 testbed. At the same time, intel8268 demonstrates the highest IPC. For all testbeds, execution time significantly drops when we utilize all cores available on the testbed, while IPC increases with the number of cores. Even though this subkernel produces large outputs and requires a lot of RAM, it is more computationally extensive than `osrm-extract` which allows to mitigate I/O bottlenecks of the armHi1620 testbed. Amount of outputs remains independent of the testbed and the number of cores. The RAM consumption is roughly twice higher than the amount of outputs and has a tendency to grow slowly with the number of cores.

[1]In Figure 2, Figure 3, and Figure 4, bar plots with MPIp results are left empty since LGE kernel is a multi-threaded application which does not use MPI for parallelism.

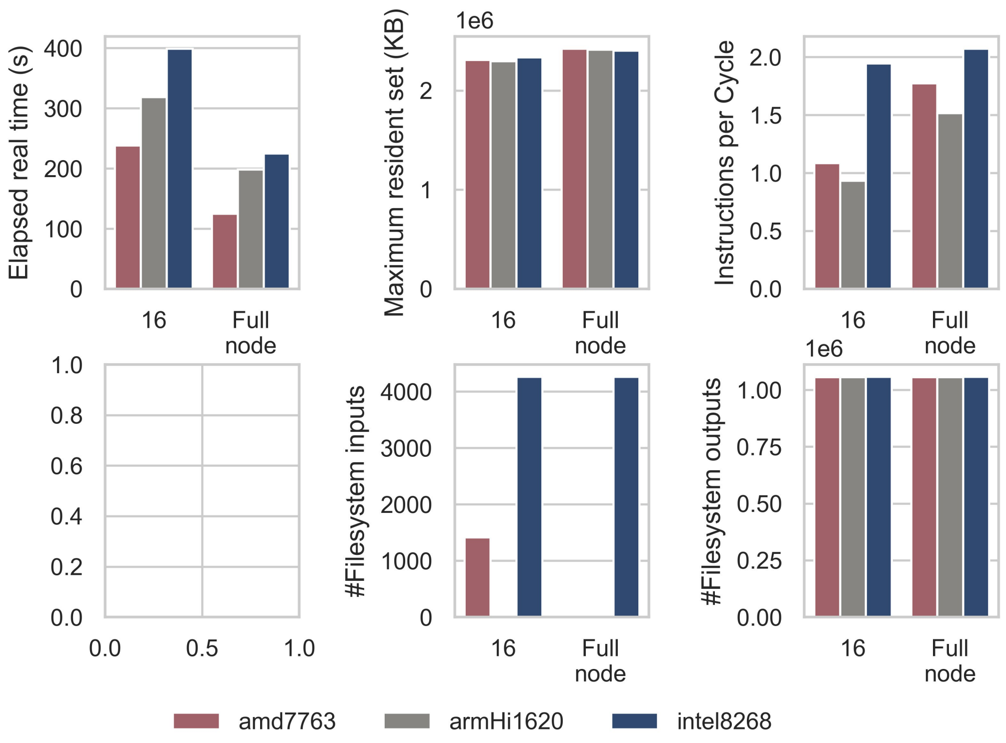


Figure 3. Evaluation of the osrm-contract subkernel of LGE kernel on CPU testbeds using 16 cores and using all cores of the testbed.

Figure 4 shows results for the `lge-extract` subkernel. In this case, intel8268 testbed outperforms amd7763 and armHi1620 in terms of execution time, while armHi1620 has the worst execution time. The RAM consumption is the highest among all LGE subkernels, stays the same for all testbeds regardless of the number of cores. The latter is expected, as the memory complexity of the algorithms is dictated by the size of the distance matrix, which stays in the RAM during the computations. Amount of outputs is very small compared to the RAM consumption.

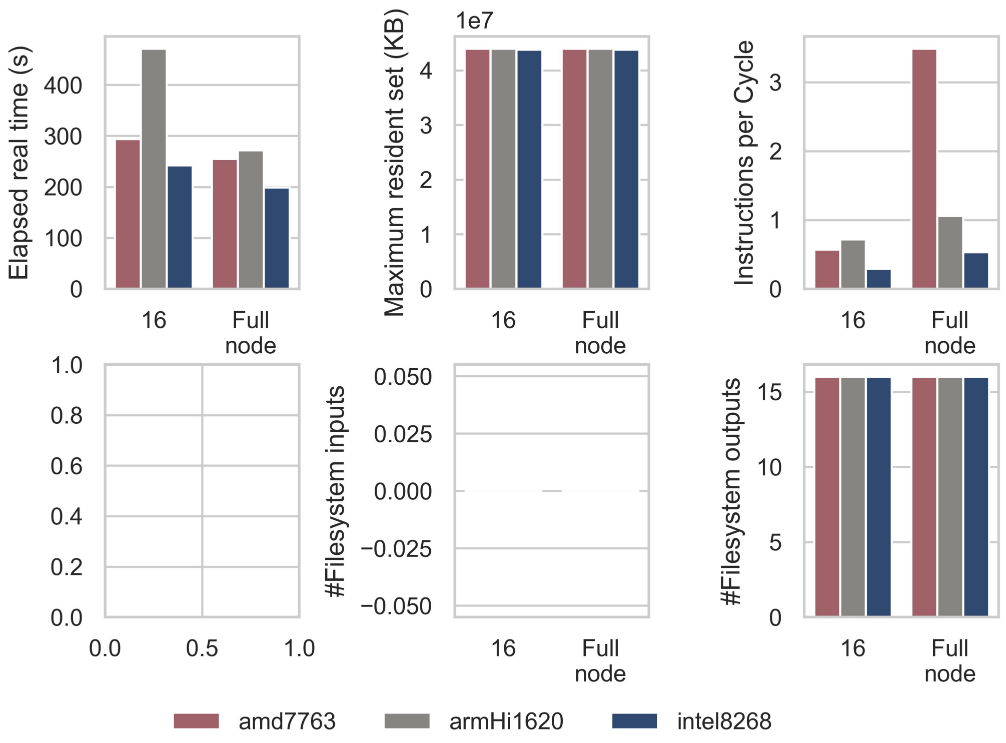


Figure 4. Evaluation of the lge-extract subkernel of LGE kernel on CPU testbeds using 16 cores and using all cores of the testbed.

4.2 Urban Air Pollution use case

4.2.1 Experimental setup

Urban Air Pollution use case is represented by one CPU kernel – OpenFOAM, – and three GPU kernels – Fluids-GPGPU, RapidCFD-GPGPU, and Magma. Moreover, Fluids and RapidCFD require CUDA runtime, and thus available only on nvA100 testbeds.

In our experiments, OpenFOAM kernel was compiled with GCC and linked against MPI pre-installed by vendors of the CPU testbeds.

Software	intel8268	amd7763	armHi1620
GCC	10.2.0	3.19.3	7.5.0
CMake	3.18.4	9.3.0	3.18.4
MPI	OpenMPI 4.1.0	OpenMPI 4.0.2	OpenMPI 4.0.2
OpenFOAM	v 20 12	v 20 12	v 20 12

Table 14: Software environment for the kernels of UAP use case on CPU testbeds (intel8268, amd7763 and armHi1620)

On our GPU testbeds, we also used versions of GCC, CUDA, and ROCm pre-installed by vendors. The only exception is RapidCFD kernel which can be installed only with CUDA up to version 11.1. Due to the implementation of the libraries and technical restrictions of the testbeds, we did not install Magma, RapidCFD, and Fluids on the testbed with AMD MI100 cards. Moreover, TensorFlow was installed with PyPI using packages `tensorflow-gpu` and `tensorflow-rocm` respectively.

Software	nvA100	amdMi100
GCC	9.2.0	7.5.0
CMake	3.16.4	3.21.4
Magma (with GPGPU API)	2.5.4 (CUDA 11.4)	-
TensorFlow	2.4.3 (CUDA 11.4)	2.4.3 (ROCm 4.5)
RepastHPC (with GPGPU API)	x.y.z (CUDA 11.1)	-
Fluids (with GPGPU API)	x.y.z (CUDA 11.4)	-

Table 15: Software environment for the kernels of UAP use case on GPU testbeds (nvA100 and amdMi100)

4.2.2 Results discussion

OpenFOAM kernel. The OpenFOAM kernel was investigated on all novel CPU architectures. The smallest mesh size of the model of Győr was used with 728k cells and only the third step of the kernel was looked at, which has only applications run in parallel. The weather data was acquired from ECMWF from the day of 20th June, 2020. Also, a realistic simulation data was used for traffic and calculation for the emission. For the steady state calculations with `simpleFoam` 600 iterations were made and one hour real time was simulated by `pimpleFoam` for the transient part. We report metrics on these two applications.

Figure 5 illustrates the performance markers of the steady state simulation with `simpleFoam`. The amd7763 performs the fastest, however it cannot gain any more speedup over 16 cores on a full node. On both other architectures runtime improves significantly. The maximum largest resident set is the largest on the amd7763 and it increases slightly for a full node. Other architectures have smaller maximum set size, moreover this value decreases for full node. The proportional time spent on MPI communication is around 20% for 16 cores and jumps to 40-80% for the full node. Filesystem I/O is consistent among the architectures, filesystem inputs was not measured on armHi1620, however.

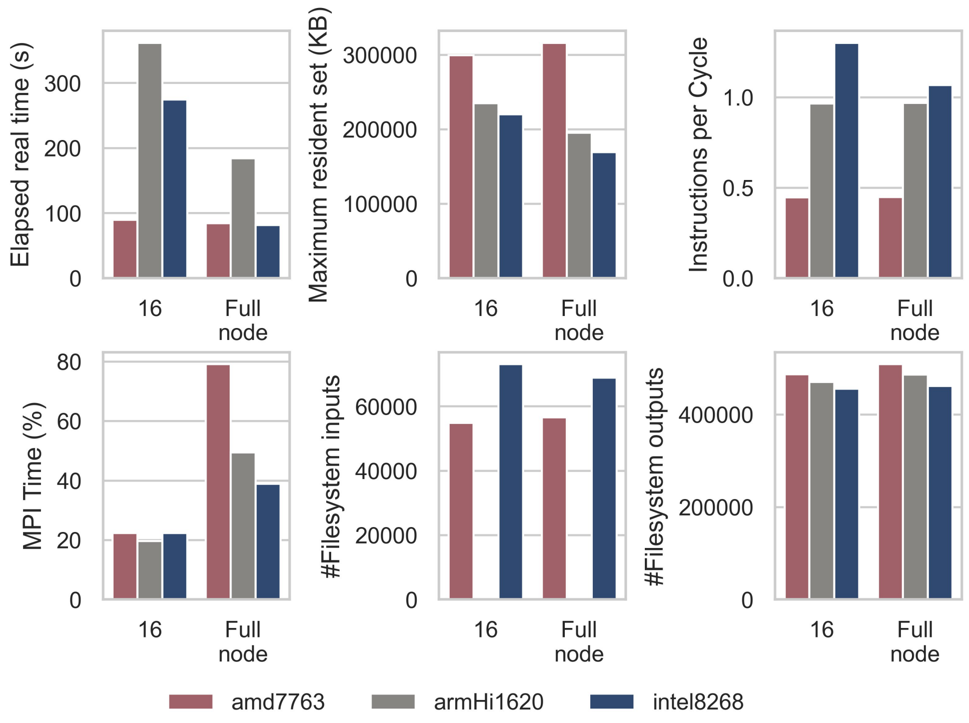


Figure 5. Evaluation of the submodule simpleFoam on CPU testbeds using 16 cores and using all cores of the testbed.

Figure 6 illustrates the performance markers of `pimpleFoam`. Here, runtime on the amd7763 gets a significant improvement on runtime, too. Memory usage, IPC and IO outputs marker behaviour is the same as in the steady state simulation, except for IO inputs, which are a fraction of the previous one. MPI Time % shows an inverse behaviour to number of cores on the node. The amd7763 is the fastest, has the smallest room for improvement, however, because of the high proportion of MPI communication.

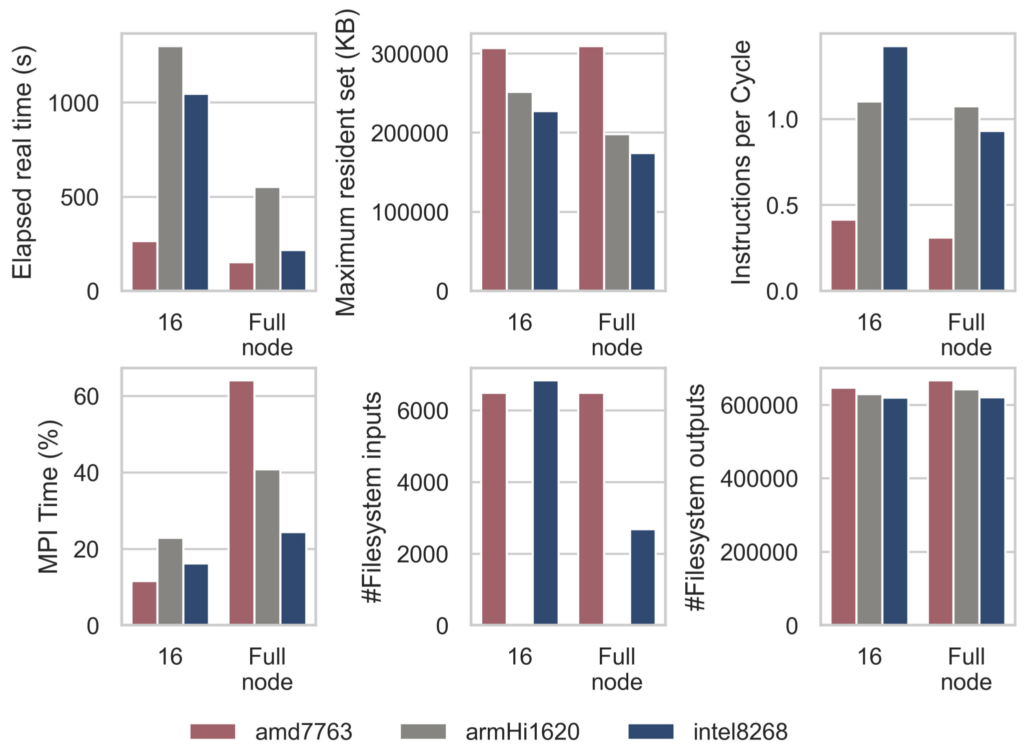


Figure 6. Evaluation of the submodule pimpleFoam on CPU testbeds using 16 cores and using all cores of the testbed.

Fluids kernel. We ran our benchmarks for `fluid_solver` on nvA100 testbed for the dataset corresponding to the city of Antwerp. We studied 3 test cases: one for FOM and two for ROM. In the case of the ROM, we tested for two different `CFL` values, 1000 and 2000. We kept the dimension of the reduced order model space (`r`) constant. Table 16 presents the results. Our benchmarks show that utilizing Nvidia A100 cards leads to significant reduction in the runtime. The test cases performed at least 10 times faster if GPU was utilized.

Test case	nvA100 (no GPUs)	nvA100 (single GPU)
FOM	820109	30221
ROM(`r=10, CFL`=1000)	2842	38
ROM(`r=10, CFL`=2000)	1500	19

Table 16: Evaluation of the Fluids kernel on nvA100 testbed disabling all GPUs and enabling single GPU.

RapidCFD kernel. In this investigation, the performance experiments are conducted to examine the capabilities of novel GPUs and assess the performance of the selected `simpleFoam` solver from RapidCFD. The latest version of RapidCFD is compiled for platforms equipped with Nvidia V100 GPU (as a part of the Altair cluster) and Nvidia A100 GPU (nvA100 testbed). We use CUDA 10.2 and CUDA 11.1 for Nvidia V100 and Nvidia A100 accelerators, respectively, to accomplish the compilation process. At the same time, we observe a compilation issue of RapidCFD with the newest versions of CUDA >=11.2 that fully support the Nvidia Ampere GPU microarchitecture. In addition, we test the performance of the `simpleFoam` solver on GPU in comparison to the CPU version of this solver available in OpenFOAM 2.3.1. For this purpose, we use intel8268 testbed.

Table 17 summarizes the concluded performance measurements. This table outlines the execution time of the simpleFoam solver obtained for different computing testbeds and various mesh sizes. We observe that the simpleFoam solver run on Nvidia A100 GPU allows achieving slightly better performance than on Nvidia V100 GPU for relatively larger mesh sizes and comparable performance for smaller sizes. However, the simpleFoam solver from OpenFOAM executed on Intel8268 testbed offers better performance than the RapidCFD version in all performed tests.

Test case (mesh size)	Intel8268 (OpenFOAM)	nvV100 (RapidCFD)	nvA100 (RapidCFD)
12M	1656.8	1966.3	1748.5
4M	497.5	887.5	896.4
2M	182.5	614.4	688.3

Table 17: Evaluation of the simpleFoam solver available in RapidCFD and OpenFOAM using intel8268 and nvA100 testbeds.

SVD kernel. We have performed SVD on double precision floating point square matrices of sizes ranging from 1000 to 32000 (see Annex C). Figure 7 presents the results for both GPU testbeds and both subkernels when the matrix has size 32000. Since we use neither MPI nor I/O for this kernel, we do not report these metrics on the figure.

We observe that Magma computes SVD order of magnitude faster than TensorFlow. The latter is expected, as Magma involves both CPUs and GPU in computations, while TensorFlow computes SVD only with GPU card. It is also indicated by the higher IPC for Magma compared to TensorFlow. The execution time of TensorFlow on nvA100 testbed is lower than on amdMi100. It can be explained by higher peak FP64 performance of Nvidia A100. At the same time, the difference in execution time is less dramatic than the difference in peak FP64 performance. The memory consumption of both subkernels is roughly the same. TensorFlow requires slightly more memory which can be related to the implementation of the library.

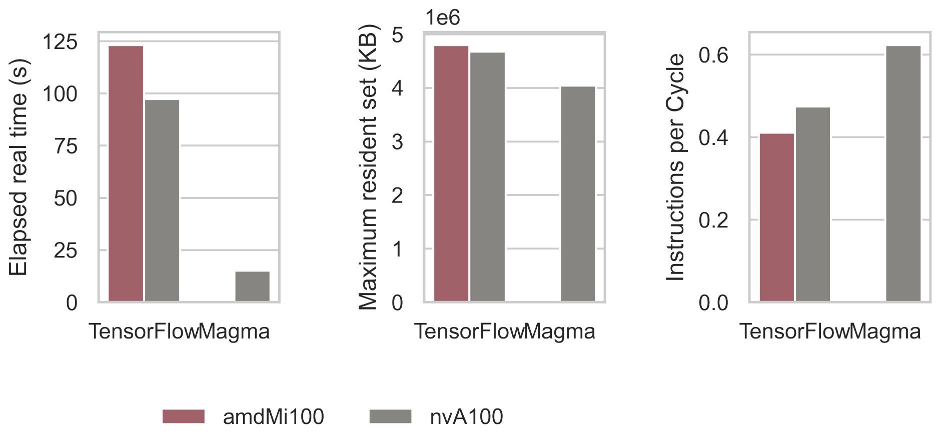


Figure 7. Evaluation of the SVD on GPU testbeds.

4.3 Social Network use case

4.3.1 Experimental setup

Social Network use case is represented by two kernels – SNSimulatior and KPM, – both of which are designed for running of CPUs. In case of kernels for SNA pilot, we used MPI, Python, and Python libraries pre-installed by vendors of the testbeds.

Software	intel8268	amd7763	armHi1620
Python	3.7.10	3.8.12	3.6.9
MPI	OpenMPI 3.1.3	OpenMPI 4.0.2	MPICH 3.3
Numpy	1.20.2	1.20.1	1.19.5
Mpi4py	3.0.2	3.0.3	3.0.2
Scipy	1.6.2	1.7.1	1.5.4
Numba	0.53.1	0.54.1	0.53.1
Pandas	1.1.5	1.1.5	1.1.5

Table 18: Software environment for the kernels of SNA use case on CPU testbeds (intel8268, amd7763 and armHi1620)

4.3.2 Results discussion

SN-Simulator kernel. We have performed a social network simulation with the SN application, for the neos_20201110 dataset, using 400 sources, 1000 samples and a random seed of 12. The results are presented in Figure 8, for three different systems.

We observe that SN demonstrates its best performance in terms of execution time on amd7763 testbed, however it is scalable both on intel8268 testbed and on armHi1620 testbed. Its IPC remains unchanged with the number of cores, thus the application can benefit from parallelism. However, its IPC on armHi1620 testbed is relatively low, thus this architecture is not a good fit for this specific application.

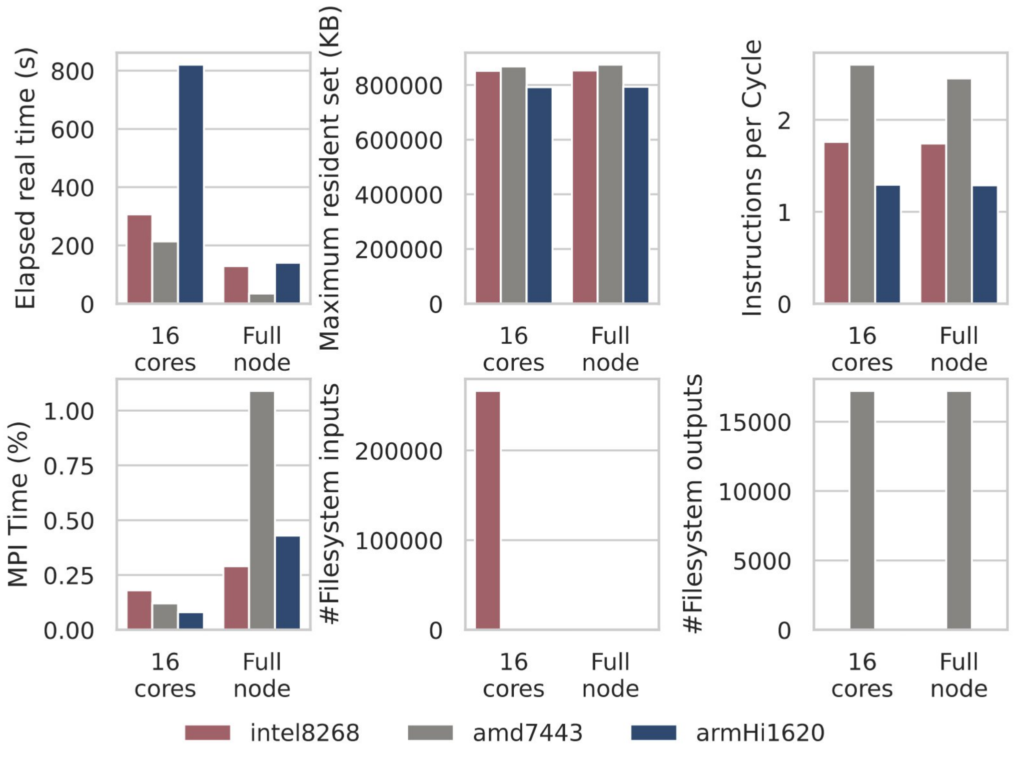


Figure 8. SN-Simulator evaluation on CPU testbeds using 16 cores and using all cores of the testbed.

KPM kernel. We have performed a computation of the network eigenvalues histogram with the Kernel Polynomial Method (KPM) for a subgraph of the pokec social network graph with a size of 50,000. We used 96 intervals, a degree of 62 and 512 samples. The results are presented in Figure 9, for three different systems.

We observe that the best performance in terms of execution time is achieved on amd7763 testbed. However, both on amd7763 testbed and on intel8268 testbed, using more cores on a single node has a negative effect on performance. The memory footprint of the application slightly increases when utilizing more cores, on all three cases, however, the application's IPC shows different behaviour on all three nodes: when adding more cores, a better IPC is observed on intel8268 testbed, a worse IPC is observed on amd7763 testbed, and an equal IPC is observed on armHi1620 testbed. What determines this behaviour is actually the application's communication, which leads to extreme increases in MPI time on amd7763 testbed and intel8268 testbed, but not on Kunpeng 920. Therefore, in this case, intel8268 testbed could be the most promising architecture in terms of per-core performance for the application's scalability, given the improvement in IPC, if the effects of communication are mitigated. However, in its current status, the application is mostly benefited from the faster cores of amd7763 testbed. A significant number of filesystem inputs is observed on the intel8268 testbed, however it does not impact performance. Since this is not observed across all systems, it is attributed to the software stack and not the application.

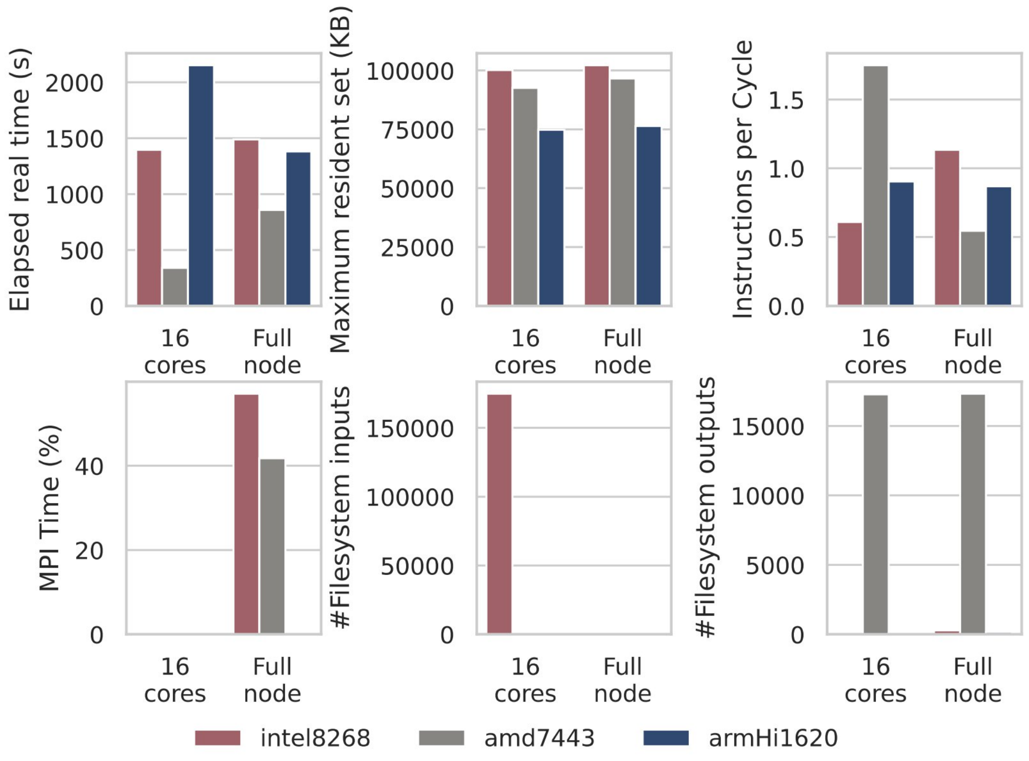


Figure 9. KPM evaluation on CPU testbeds using 16 cores and using all cores of the testbed.

5 Conclusion

In D5.8, we assess five contemporary processing unit architectures – 3 CPUs and 2 GPUs – that in the near future can be dominant in HPC environments. These processing units come from major CPU and GPU vendors: AMD, ARM (Huawei), Intel, and Nvidia. The assessment is fulfilled using the HiDALGO benchmark suite.

In contrast to the existing counterparts such as SPEC and CoeGSS benchmark suites, HiDALGO benchmark suite is suitable not only for CPU, but also for GPU testbeds. It incorporates a wide range of benchmarks for CPU and GPU testbeds that cover diverse numerical simulations from ABSS to CFD as well as computationally expensive data pre- and post-processing kernels related to linear algebra and GIS. These benchmarks are derived from the analysis of the hotspots in the data flows of HiDALGO pilots. HiDALGO benchmark suite reports a wider range of metrics compared to SPEC and CoeGSS benchmark suites, which allows to gain deeper understanding the bottlenecks in the testbeds and software. At the same time, the methodology for collecting metrics does not require instrumentation and relies on a simple set of tools – `/urs/bin/time`, `perf stat`, and `mpiP` – natively available at all *nix systems. Solid attention is put on reporting of the results and reproducibility. All results are collected in the CSV files and supplemented with Python scripts for analysis and visualization. Along with the measurements, we precisely report experimental setup: the hardware configuration as an XML output of `lstopo` and the exact software environment as a Spack lock file. Last but not least, the HiDALGO benchmark suite contains a broad set of automation scripts including Ansible playbooks for deployment, installation, fetching setups and results, etc. Simple and portable methodology, as well as powerful automation scripts make it easy to extend HiDALGO benchmark suite with the new benchmark kernels and testbeds.

Our benchmarks reveal that the individual Zen3 cores of AMD Milan CPUs are more powerful than Cascade Lake-SP cores of Intel Xeon Platinum 8268 and ARM Cortex-A72 cores of Kunpeng 920. At the same time, in many benchmarks Intel Xeon Platinum 8268 and Kunpeng 920 demonstrate better scalability than AMD Milan CPUs as the number of utilized cores grows. Moreover, in certain situations (e.g., triangular pruning step of location graph extraction), both Intel Xeon Platinum 8268 and Kunpeng 920 can outperform AMD Milan for sufficiently large number of cores.

Even though state-of-the-art AMD GPUs like MI100 provide a viable alternative to contemporary Nvidia cards like A100 in terms of performance, they are still a subject of many limitations related to porting the software. In particular, out of 3 GPU kernels from the HiDALGO benchmark suite, only one partially supports AMD GPUs, while all of them fully support compilation with CUDA.

Despite our efforts to include into the study as diverse and representative selection of the testbeds as possible, we were not able to get access to all testbeds with promising novel microarchitectures. In particular, we lack results for Intel processors with Cooper Lake cores which support BF16. This is a subject of future research.

References

[1] HiDALGO, "D5.5 – Innovative HPC Trends and the HiDALGO Benchmarks." 2021.

[2] HiDALGO, "D4.2 – Implementation Report of the Pilot Applications Year 1." 2019.

[3] HiDALGO, "D4.3 – Implementation Report of the Pilot Applications Year 2." 2020.

[4] HiDALGO, "D4.4 – Final Implementation Report of the Pilot and Future Applications." 2021.

[5] HiDALGO, "D6.2 – Workflow and Services Definition." 2019.

[6] HiDALGO, "D6.4 – Initial Report on Requirements, Components and Workflow Integration." 2019.

[7] HiDALGO, "D6.5 – Intermediate Report on Requirements, Components and Workflow Integration." 2020.

[8] HiDALGO, "D6.6 – Final Report on Requirements, Components and Workflow Integration." 2021.

[9] HiDALGO, "D3.1 – Report on Benchmarking and Optimisation." 2019.

[10] HiDALGO, "D3.2 – Initial Specifications for HPC Scalability Optimisation, HPDA Model Implementation, Data Management, Visualisation and Coupling Technologies." 2019.

[11] HiDALGO, "D3.3 – Intermediate Report on Implementation and Optimisation Strategies." 2019.

[12] HiDALGO, "D3.4 – Intermediate Report on Benchmarking, Implementation, Optimisation Strategies and Coupling Technologies."2020.

[13] HiDALGO, "D3.5 – Final Report on Benchmarking, Implementation, Optimisation Strategies and Coupling Technologies." 2021.

[14] Ronny Krashinsky, Olivier Giroux, Stephen Jones, Nick Stam, and Sridhar Ramaswamy, "Nvidia Ampere Architecture In-Depth." May 14, 2020. URL: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

[15] Nvidia Corporation, "Nvidia A100 40GB PCIe GPU Accelerator: Product Brief PB-10137-001_03." September 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/A100-PCIE-Prduct-Brief.pdf

[16] Nvidia Corporation, "Nvidia A100 Tensor Core GPU: Unprecedented acceleration at

every scale." 2021. URL: https://www.nvidia.com/en-us/data-center/a100/

[17] Advanced Micro Devices, Inc., "Introducing AMD CDNA Architecture: The All-New AMD GPU Architecture for the Modern Era of HPC & AI." 2020. URL: https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf

[18] Timothy Prickett Morgan, "AMD At A Tipping Point With Instinct MI100 GPU Accelerators." 2020. URL: https://www.nextplatform.com/2020/11/16/amd-at-a-tipping-point-with-instinct-mi100-gpu-accelerators/

[19] Hassan Mujtaba, "AMD Unveils Instinct MI100 CDNA GPU Accelerator, The World's Fastest HPC GPU With Highest Double-Precision Horsepower." 2020. URL: https://wccftech.com/amd-unveils-instinct-mi100-cdna-gpu-accelerator-the-worlds-fastest-hpc-gpu/

[20] Hewlett Packard Enterprise Development LP, "HPE Apollo 6500 Gen10 Plus System – Overview." 2021. URL: https://support.hpe.com/hpesc/public/docDisplay?docId=a00109734en_us&docLocale=en_US

[21] HLRS, "HPE Apollo (Hawk): Next-Generation HPC System @ HLRS." https://www.hlrs.de/systems/hpe-apollo-hawk/

[22] Damian Kaliszan, Norbert Meyer, Sebastian Petruczynik, Michael Gienger, and Sergiy Gogolenko, "HPC Processors Benchmarking Assessment for Global System Science Applications." Supercomputing Frontiers and Innovations, 6 (2), 2019. DOI: 10.14529/jsfi190202

[23] Standard Performance Evaluation Corporation (SPEC), "SPEC CPU® 2017 Overview: What's New?" URL: https://www.spec.org/cpu2017/Docs/overview.html

[24] Christoph Schweimer, Bernhard C. Geiger, Meizhu Wang, Sergiy Gogolenko, Imran Mahmood, Alireza Jahani, Diana Suleimenova, and Derek Groen. "A route pruning algorithm for an automated geographic location graph construction." Scientific Reports, 11(11547), 2021. DOI: 10.1038/s41598-021-90943-8

[25] Edoardo Di Napoli, Eric Polizzi, and Yousef Saad, "Efficient estimation of eigenvalue counts in an interval." Numerical Linear Algebra with Applications, 29 (1), 2016. DOI: 10.1002/nla.2048

[26] Linux Foundation, "`time` (1) – Linux manual page." URL: https://man7.org/linux/man-pages/man1/time.1.html

[27] Linux Foundation, "`perf-stat` (1) – Linux manual page." URL: https://man7.org/linux/man-pages/man1/perf-stat.1.html

[28] LLNL, "mpiP 3.5: A light-weight MPI profiler." URL: https://software.llnl.gov/mpiP/

Annex A. Quick start to the automated deployment with the HiDALGO benchmark

Ansible is a necessary prerequisite, thus, before running hidBS make sure you have Ansible on your control workstation. There are many ways to install Ansible. One of the simplest is via PyPI

```
# python -m venv ~/ansible
# . ~/ansible/bin/activate
python -m pip install -r ./ansible/requirements.txt
```

In order to make your work more comfortable, it is also recommended to register remote testbed credentials in Ansible. You can store user logins like this:

```
mkdir -p ./ansible/inventory/host_vars
echo                   "ansible_user:                    <your-psnc-atlas01-
login>" > ./ansible/inventory/host_vars/armHi1620.yaml
```

For the sensitive information like passwords, you should use either SSH keys or Ansible vaults. For cloning the HiDALGO benchmark repo use:

```
git clone https://gitlab.com/eu_hidalgo/hidalgo_bench_suite.git
git pull -recurse-submodules
```

As soon as Ansible is installed and repo is cloned, you can run hidBS following these steps:

- getting help information

  ```
  ./bin/hidBS.sh help
  ```

- fast deployment of Spack at testbeds and submission of software installation jobs with Ansible

  ```
  ./bin/hidBS.sh deploy
  ./bin/hidBS.sh install
  ./bin/hidBS.sh fetch_setup
  ```

  If you need only a specific testbeds (subset of inventories), you can place the names in the command line

  ```
  ./bin/hidBS.sh deploy  armHi1620,nvA100
  ./bin/hidBS.sh install armHi1620,nvA100
  ```

  - collecting information about platforms

  ```
  ./bin/hidBS.sh fetch_setup
  ```

  If you have hwloc on your local workstation, you can visualize topology of the testbed

```
lstopo -i ./hidBS/armHi1620/setup/hardware.xml
```

- cleaning up from the installation

```
./bin/hidBS.sh cleanup armHi1620,nvA100
```

Annex B. Topology of the testbeds

This annex holds figures with the system topology of each testbed as generated by `lstopo`.

CPU testbeds

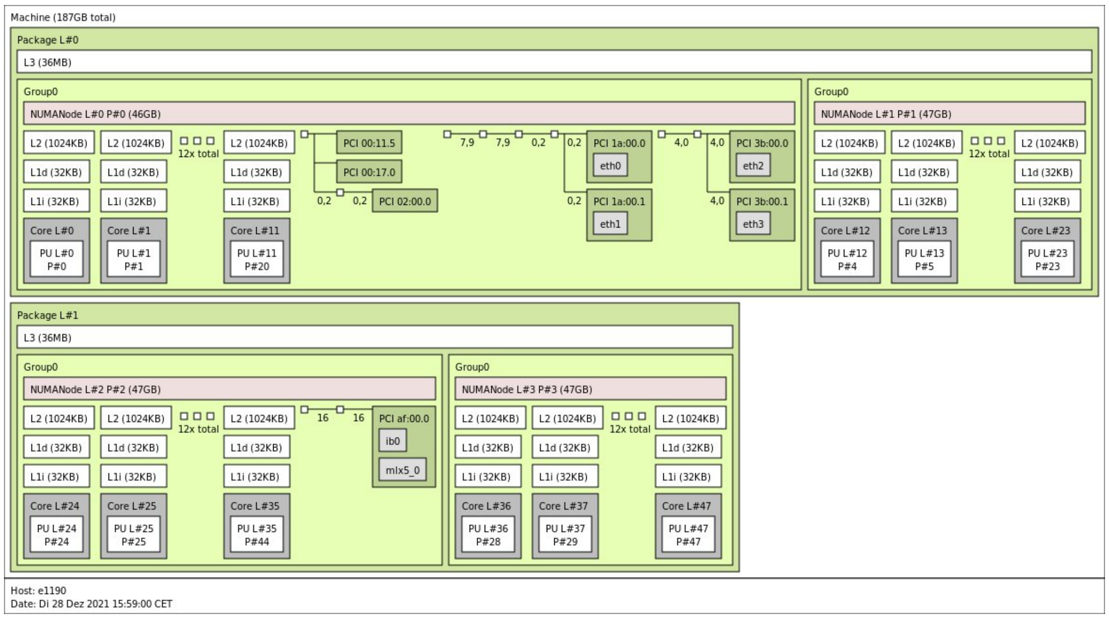


Figure 10. Topology of the intel8268 testbed.

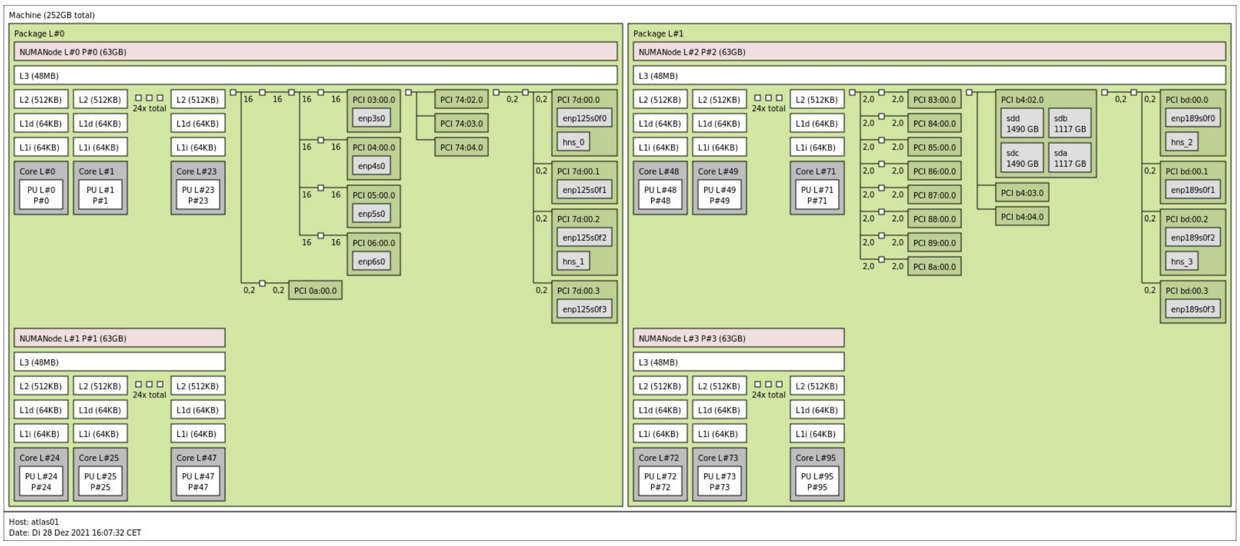


Figure 11. Topology of the armHi1620 testbed.

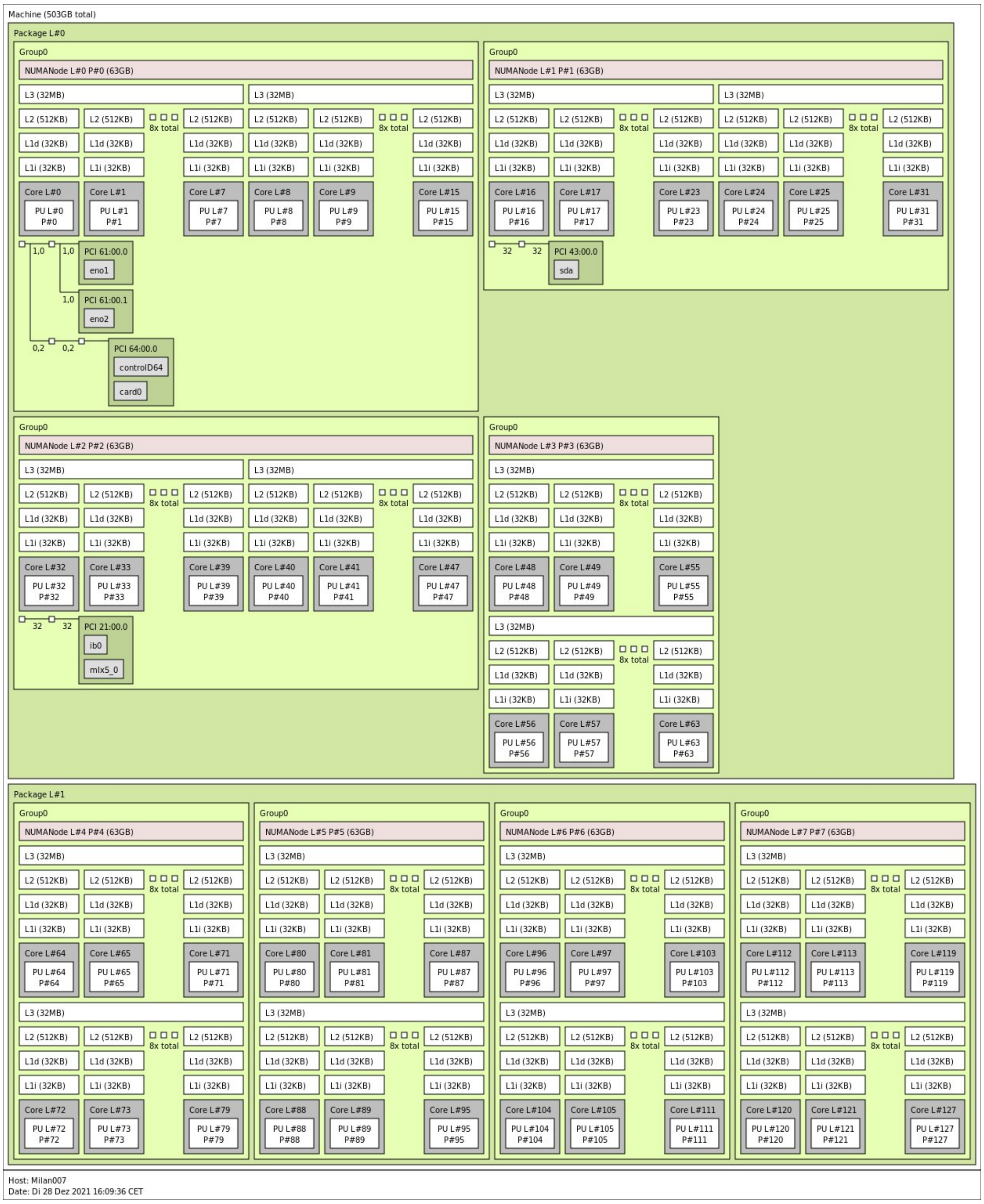


Figure 12. Topology of the amd7763 testbed.

GPU testbeds

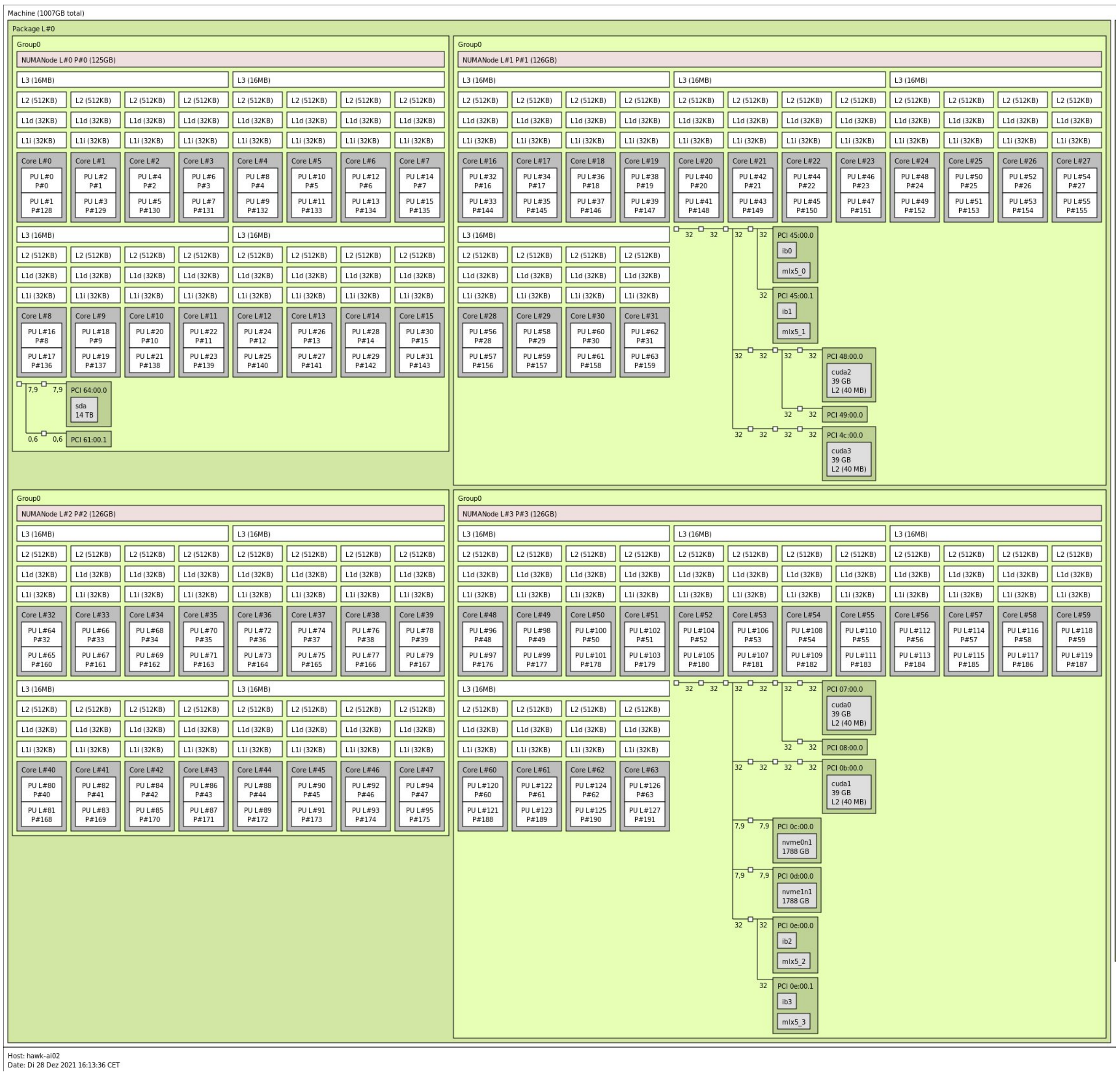


Figure 13. Topology of the nvA100 testbed (single package out of two identical).

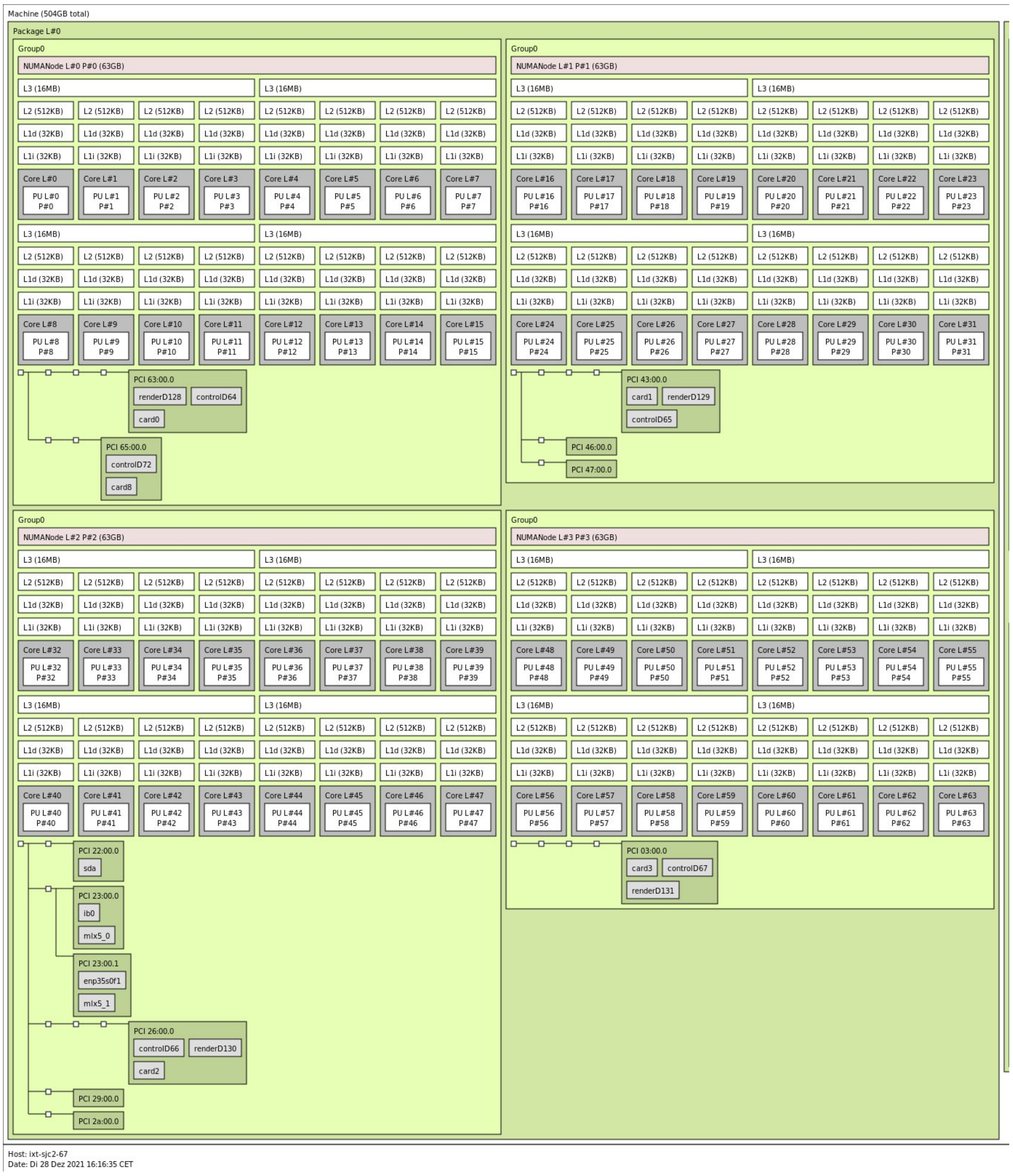


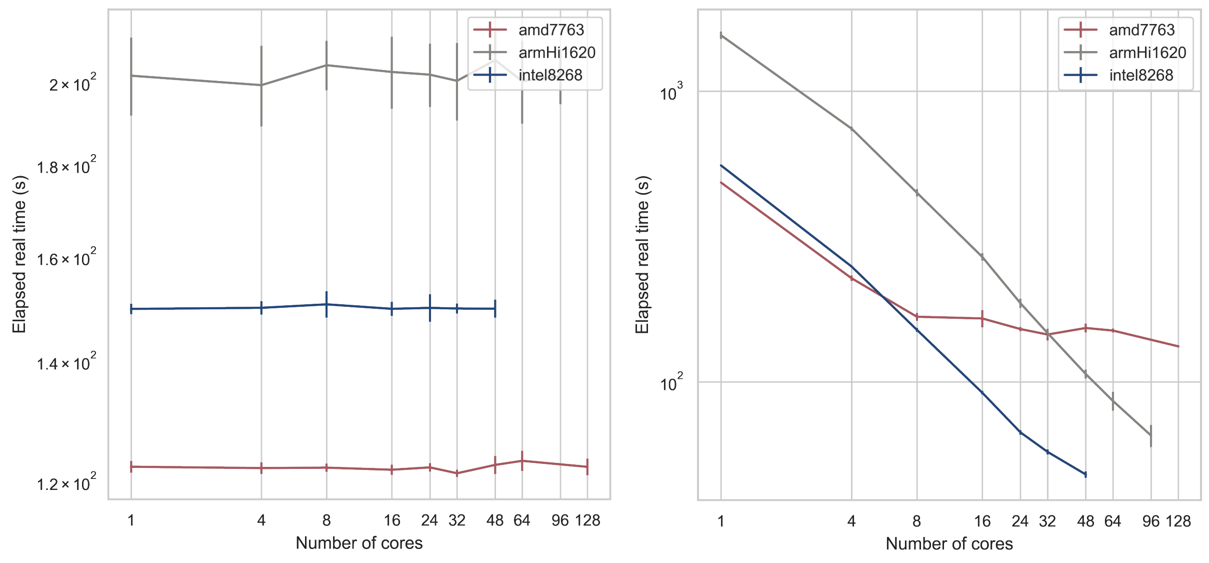
Figure 14. Topology of the amdMi100 testbed (single package out of two identical).

Annex C. Auxiliary benchmark results

In certain cases, in order to approve or refute unexpected conclusions about performance of the testbeds with different kernels, we require additional benchmarks that go beyond the scope of the methodology of the HiDALGO benchmark suite. Results of such auxiliary benchmarks are presented in this Annex.

Migration use case

Figure 4 shows that intel8268 testbed requires less time than amd7763 testbed to execute `lge-extract` subkernel of LGE kernel on `europe/ukraine` dataset. It can give a wrong impression that Cascade Lake-SP cores of Intel Xeon Platinum 8268 outperform Zen3 cores of AMD Milan. In order to disprove this hypothesis, we performed a number of auxiliary measurement for the `lge-extract` subkernel, which are summarized in Figure 15.



(a) Computing distance matrix

(b) Triangular pruning

Figure 15. Scaling of computing routing table and triangle pruning of LGE kernel with the number of cores on CPU testbeds.

Figure 15 demonstrates how two computational steps of the `lge-extract` subkernel – computing distance matrix and triangular pruning – scales with the number of cores. In the step of computing distance matrix, elapsed time remains invariant to the number of the utilized cores. This is due to the fact that OSRM library uses only one core to compute distance matrix. In this step, amd7763 testbed is significantly faster than intel8268 and arm1620. Similarly, in the step of triangular pruning, amd7763 outperforms intel8268 and arm1620 testbed for the small number of cores. In other words, for both steps, Zen3 cores of AMD

Milan perform better than competitors. Nevertheless, as the number of cores becomes more than 8, performance of amd7763 significantly drops, while performance of intel8268 and arm1620 scales almost linearly with the number of cores for up to the full capacity of cores in the testbeds. This scalability issue explains better results for intel8268 when the number of cores is equal 16.

Urban Air Pollution use case

Figure 16 demonstrates how elapsed time and IPC for SVD kernel change with the size of the problem on nvA100 testbeds. Magma performs SVD almost order of magnitude faster than TensorFlow. IPC for TensorFlow drops quickly as the size of the problem growth, while changes in IPC for Magma are relatively slow. Both these facts illustrate the main difference in utilizing computational resources by Magma and TensorFlow: TensorFlow computes SVD on GPU cards, while Magma involves additionally utilizes available CPUs.

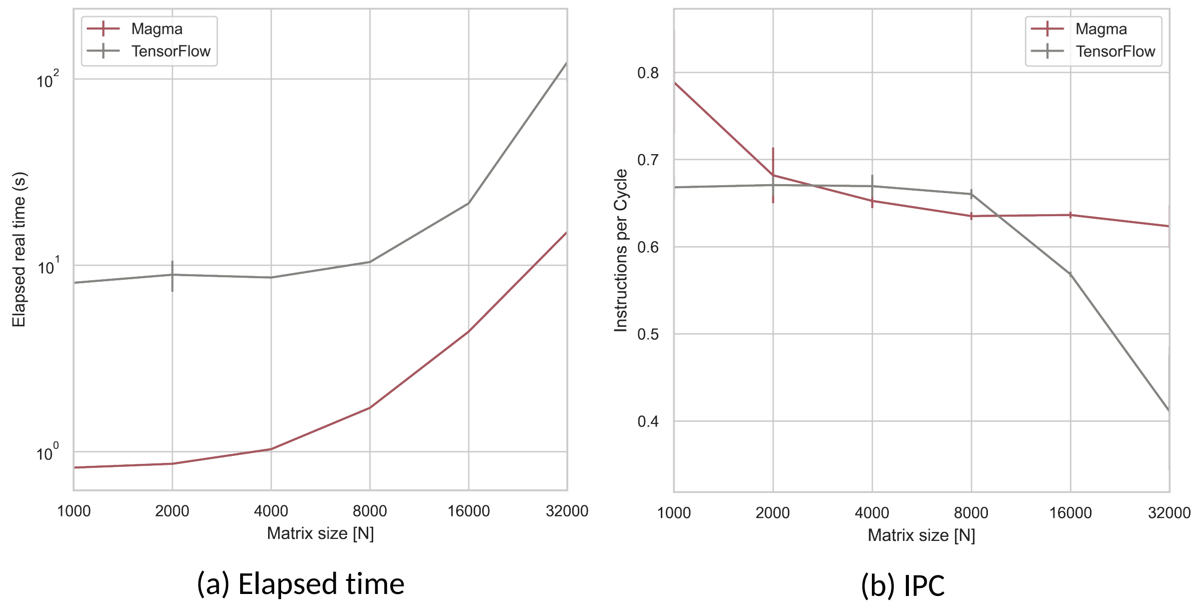


Figure 16. Change in the elapsed time and IPC with the size of the problem for SVD kernel on nvA100 testbed.