D4.2– First report on codesign
Version 1.0

Document Information

Contract Number	101143931
Project Website	www.pop-coe.eu
Contractual Deadline	M12
Dissemination Level	PU - Public
Nature	R - Document, Report
Authors	William Jalby (UVSQ)
Contributors	Radim Vavrik (IT4I@VSB), Xavier Teruel (BSC), Jesus Labarta (BSC), Kevin Camus (UVSQ), Stefan de Souza (USTUTT), Tobias Dollenbacher (RWTH)
Reviewers	Christoph Niethammer (USTUTT)
Keywords	Codesign, Kernels, EPI

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101143931. The JU receives support from the European Union's Horizon Europe research and innovation programme and Spain, Germany, France, Portugal, and the Czech Republic.

Change Log

Version	Author	Description of Change
v0.1	Radim Vavrik	Initial version of the document
v0.2	William Jalby	EPI methodology and evaluations
v0.3	Radim Vavrik	Codesign resources development
v1.0	Xavier Teruel	EPI methodology and codesign resources updated

Contents

Executive Summary 5

1 Introduction 6

2 Kernels development 7
2.1 GPU affinity 7
2.2 Pils 7
2.3 SPMXV 7
2.4 LBC 8

3 Technical pages development 9
3.1 Patterns and best-practices 9
3.1.1 Poor GPU data transfer rate 9
3.1.2 Appropriate process/thread mapping to GPUs 10
3.1.3 Load imbalance in hybrid programming (MPI+X) 11
3.1.4 Using DLB to compensate load imbalance 12
3.2 Programming models 13
3.2.1 HIP 13
3.2.2 OpenMP offloading 13
3.3 Projects collection 14

4 EPI Methodology 16
4.1 RISC V 16
4.2 RHEA 18

5 Evaluation and analysis 20
5.1 JUKRR kloop 20
5.2 JuKRR on RISC-V 20
5.3 JuKRR on RHEA 21
5.4 SPMXV Initial performance assessment 23
5.5 SPMXV on RISC-V 24
5.5.1 SpMV Elapsed time study 25
5.5.2 Extrae study 26
5.5.3 Architectural study 28
5.5.4 Conclusions 32
5.6 SPMXV on RHEA 33
5.6.1 Experimental Conditions 33
5.6.2 Timing Tests: Single Core/Multicore 34
5.6.3 Compiler Analysis 35
5.6.4 Conclusion and Future Work 36
5.7 LBC General performance assessment 37
5.8 LBC on RHEA 39
5.8.1 Experimental Conditions 39
5.8.2 Compiler Issues 39
5.8.3 Multicore runs 41
5.8.4 Conclusions and Future work 42

6 Conclusions and Future work 44

Acronyms and Abbreviations 46

List of Figures 47

List of Tables 48

Executive Summary

This document reports the activities and results accomplished in terms of tasks T4.1 and T4.2 during the first twelve months of the project.

First, the four kernels developed in both tasks are presented, and their main characteristics are listed.

Second, a few lessons taken from the audit effort are detailed: four case studies related to patterns and best practices, two notes related to programming models and finally one summary of our collaboration with other CoE.

Third, our methodology for cooperating with EPI projects (RISC V and Rhea) is presented. This covers the experimental methodology developed to generate results and analyze them as well as the various hardware platforms and software tools (simulators, compilers, . . .) used.

Fourth, our kernel evaluation results are presented. The two kernels selected (Sparse Matrix Vector multiply and Lattice Boltzmann Computations), turned out to be very challenging both from a hardware point of view and also from a compiler point of view. In particular, both kernels clearly revealed many deficiencies of current compiler technology and uniformly across several major compiler providers (ARM, INTEL and GCC/GFortran). Our findings have been shared with both RISC V and RHEA teams. In particular, with RHEA, our kernels have been tested on their in-house compiler. In both cases, our detailed performance analysis led to very fruitful interactions and will drive further work to improve the performance of these two kernels.

From a deliverable standpoint, Table 1 shows the defined KPIs to be reached until M12 of the project and their current state.

KPI	M12 Goal	M12 Reached
Kernels created	3	4
Technical pages created	7	7
Kernels evaluated	2	3

Table 1: KPIs for the MS5 Codesign 1 milestone

All the indicators were reached with a slight overachievement.

1 Introduction

The overall objective of WP4 is to enable codesign with Exascale architectures and applications in general, and the EPI in particular, while continuously improving the performance assessment methodology and the corresponding tools. The particular objectives of the codesign tasks T4.1 and T4.2 are:

- To create a set of resources that could be leveraged by application-, programming model- and system designers to guide its own development.
- To provide valuable guidance for the EPI hardware and software ecosystem based on the kernels and applications analysed in the project.

Task 4.1 Codesign resources started at M6 and will continue till the end of the project. Within the consortium, the partners IT4I@VSB, BSC, USTUTT, INESC-ID, and RWTH participate in it. The task synthesizes potential HPC application performance problems, develops corresponding solutions, and makes these available to the HPC community. It includes three main activities: (1) the development of new kernels that codesigning users may leverage for their codesign activities, (2) the continuous update of the technical content included in the POP codesign website; and (3) POP codesign website extension and maintenance. The main sources for technical pages (articles on the POP codesign website) and kernels are the HPC applications we work with in WP3 in form of performance assessments and second-level services.

The activities in this task are not limited to the creation of new content but also include the enhancement of the existing ones, so the task continuously refactors existing technical page descriptions where needed, includes new experiments to the already uploaded kernels, especially those used in the EPI codesign task.

Task 4.2 EPI codesign started at M1 and will be pursued till the end of the project by the partners UVSQ, BSC, USTUTT, and RWTH.

This task provides input/feedback from the application perspective to the developers of the hardware/software platform within EPI, the European Processor Initiative, and the two European Pilots (the EUPILOT and EUPEX). Within EPI, two target architectures have been selected: RISC V accelerators (project led by BSC) and RHEA (processor developed by SiPearl). Both projects have different development strategies and therefore we used different methodologies to carry out codesign activities. For the RISCV codesign, applications of interest will be executed on simulators, SDVs (Software Development Vehicles) or early prototypes to provide direct feedback to the development groups, addressing both the system performance and the system software. The Sipearl project is using an ARM Neoverse V1 core for which, implementations are already available (AWS Graviton 3) with a full software stack (compilers, libraries). For codesign activities on RHEA, we decided to focus our efforts first on the compiler side and second on testing not only Graviton 3 but also close by ARM cores (Neoverse V2 and Neoverse N1). To get a full performance picture of the competition, we also systematically tested recent INTEL X86 processors.

The developed kernels and technical pages are published in their full detail on the codesign section of the project website[1]. The shortened descriptions of the achieved results follow in the next sections. At the moment, the SPMXV and LBC kernels are accessible through these links:

- `https://gitlab.pop-coe.eu/kernels/spmxv/`
- `https://gitlab.pop-coe.eu/kernels/lbc/`

[1]https://co-design.pop-coe.eu

The document is structured in six major sections. After Section 1 is devoted to the introduction, section 2 presents the main kernels developed both in T4.1 and T4.2. Section 3 will describe the effort c arried o ut o n Technical P age D evelopment. S ection 4 w ill p resent codesign methodologies used in the 4.2 task. Then Section 5 will review the results obtained through the detailed evaluation of the 4.2 kernels. Finally, conclusion and future work will be described in section 6.

2 Kernels development

Between Tasks 4.1 an,d 4.2, The rationale for kernel selection is different a nd complementary. In Task 4.2, kernels are code fragments (potentially artificial) w hich w ere s elected b ased on well-known performance issues that were repeatedly observed during performance analysis. In Task 4.2, instead, we decide to select full algorithms (here Sparse Matrix-Vector Multiply and Lattice Boltzmann computation) which have been chosen because they are widely used in various scientific c omputations. B y t he w ay, d ue t o t his h eavy u se, s ome o f t he a lgorithms c an have some specific implementation in vendor libraries (MKL, ARMPL).

2.1 GPU affinity

This kernel (program) reproduces performance issues due to GPU affinity. It re peatedly performs a single precision α times x plus y (SAXPY) operation:

$$Y \leftarrow \alpha x + y$$

It uses OpenMP target offloading to repeatedly perform this operation on the GPU. Further, it uses MPI to divide huge vectors into smaller parts and computes them in parallel. Each MPI process uses one device to compute its partial result.

2.2 Pils

Pils is a synthetic code which allows to play with multiple MPI processes, each of them with a different workload parallelized with OpenMP. The program allows you to configure the workload per MPI process, the grain of the OpenMP parallel region, the number of steps to execute, etc.

This program is an isolated version of the *Pils* test included within the DLB distribution. Further information on https://pm.bsc.es/dlb/.

2.3 SPMXV

The SPMXV kernel implements the multiplication $y \leftarrow Ax$ of a sparse matrix A and a vector x. The algorithm for the SpMXV operation is outlined in Figure 1.

The memory bandwidth is the limiting factor in the SpMXV kernel, as by using a sparse matrix data structure the memory accesses become more random. The sparse matrices used in this kernel are stored in the CSR (Compressed Sparse Row) format. To compute the resulting value for matrix row r, the corresponding matrix values have to be loaded. CSR is efficient in the aspect that for the sparse matrix-vector multiplication, the matrix values are stored consecutively in memory, i.e., the n values of an arbitrary row r are stored consecutively as $(a(i), \ldots, a(i+n-1))$. Hence, reading from the matrix has high spatial locality, but again every element is read only once. The elements of the right-hand side vector x also have to be loaded, but here the memory access pattern depends on the matrix structure. For every

```
**for** $i = 0$ **to** $n - 1$ **do**
    $sum \leftarrow 0$
    $rowbeg \leftarrow Arow[i]$
    $rowend \leftarrow Arow[i + 1]$
    **for** $nz = rowbeg$ **to** $rowend - 1$ **do**
        $sum \leftarrow sum + Aval[nz] \cdot x[Acol[nz]]$
    **end for**
    $y[i] \leftarrow sum$
**end for**
```

Figure 1: SPMXV kernel

matrix element $a(i)$, the column is stored in a separate vector named `col` at the same position i, again resulting in consecutive access to col. However, access to x is random, as the position i of $x(i)$ is determined by the value of `col`(i). Consequently, the reuse of the elements in the vector `col` from the cache is only successful for blocks/bands in the matrix, resulting in a low spatial and temporal locality of this part of the operation. Finally, SPMXV is by far the most time-consuming operation in a GMRES (or similar iterative) method.

The main parameters for the SPMXV kernel are the size of the matrix and the vector as well as the structure of the used matrix. While the size increases the outer loop, the number of non-zero elements in a row determines the iteration count of the inner loop. Depending on the structure of the matrix this may vary and therefore result in a load imbalance.

The kernel can be found here: https://git-ce.rwth-aachen.de/hpc-public/epi-spmxv/

2.4 LBC

The Lattice Boltzmann kernel simulates a 3D vortex shedding around a cylindrical geometry at a low Reynolds number. The computations are performed using double floating-point precision. As a particle method, the kernel discretizes space into a 3D structured grid (a lattice), time according to the CFL condition, and velocity into a number (here, 19) of fixed flow directions (see Figure 2). This results in each lattice holding a particle distribution function (PDF) $F(\text{velocity}, \text{space})$ that describes the flow and allows one to recover variables such as velocity and momentum using simple integrals over these distribution functions and known weights. This results in the well-known *D3Q19* model, where the grid is traversed, and the new distribution function values are then calculated only depending on old grid values. The three main components of the kernel are first the streaming part, where the distribution functions from 19 neighboring lattice sites are pulled, second the calculation of macroscopic variables, and third the collision step, where an operator is executed, calculating the new distribution function and storing the new grid.

The code is written in Fortran. The distribution function is stored as a 4D array in one of two forms: $F(q, i, j, k)$ or $F(i, j, k, q)$, where q denotes velocity and the other three coordinates denote a lexicography numbering of the grid. Since Fortran is column major the first layout presents more of a 'structure of array' type pattern where each lattice point stores the velocity vectors contiguously, as opposed to the second layout, which includes the spatial coordinates in the most rapidly changing dimension.

The code is parallelized with MPI, and uses a simple blockwise domain decomposition. Code and some documentation are available at the Git repo: https://code.hlrs.de/SPMT/lbc-pop3/.

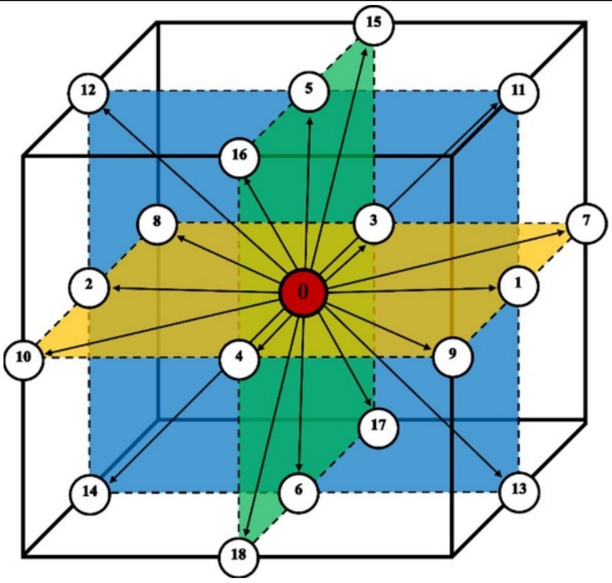


Figure 2: LBC Computational pattern

3 Technical pages development

One of the goals of the codesign resources task is to publish the finished reports (outputs of the assessments) to the codesign website. Only the reports that are approved by the owner of the assessed code through the post-assessment questionnaire can be published. We publish the reports in batches every 6 months to minimize the overhead - the first set of 3 approved POP3 reports was published in M7 and we expect to publish another set of new reports in M13 after the recent questionnaires submission.

In addition to the newly published technical pages described below, the back-end functionality of the codesign website was enhanced, too. To understand the relations among the content of the website, consider the following hierarchy. Each kernel page (named program in the context of the website) of the programs collection contains links to 2 or more program version pages, typically 1 page for an original version showing a performance issue (a pattern) and 1 page for the optimized version implementing the solution of the issue (a best-practice). Originally, each of the version pages included a link to a separate experiment page containing specific performance details of the version, e.g., POP performance metrics evaluation, profiles, or trace visualizations. Both the new technical pages development and some of the topics developed during the previous phase of the project exposed the need for side-to-side comparison of the performance data from both versions on the same page. This feature was newly implemented and already used, e.g., for the Parallel File I/O or the Pils kernels.

3.1 Patterns and best-practices

The following patterns and best-practices are the generalized descriptions of performance issues and their corresponding solutions related to the kernels 2.1 and 2.2.

3.1.1 Poor GPU data transfer rate

Most CPUs are organized in multiple NUMA domains. Accessing different parts of the memory from a certain core of such a CPU therefore has different performance. We call this affinity.

Since each available GPU on a system is connected to a NUMA domain this effect also is observable in data transfers from and to the GPU. Depending on the location of the data on the host side and the core that is handling the data transfer, both bandwidth and latency can vary significantly.

If we launch one MPI process per available GPU without specifying where the MPI process should run, the trace of such an application may look like the trace in Figure 3.

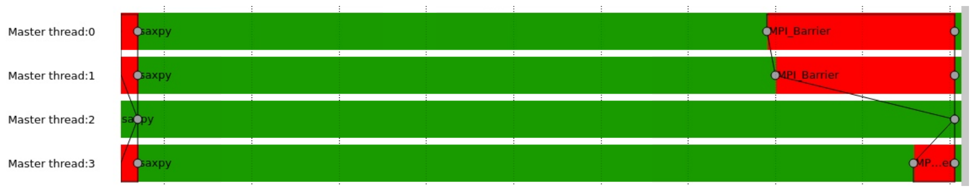


Figure 3: An example of a trace manifesting the poor GPU data transfer rate as a load imbalance issue.

We can see that the first process finishes faster even though the computational load is the same. This is due to the faster CPU-to-GPU transfers that are available on the MPI process with rank 0 as it was executed on the NUMA domain the used GPU is connected to. However, the MPI processes with rank 2 and 3 execute on socket 0 while the associated GPUs are connected to socket 1. This heavily increases the runtime and results in a load imbalance.

3.1.2 Appropriate process/thread mapping to GPUs

Using the correct CPU cores to handle data transfers to and from the GPU can have a significant impact on the performance. Especially with large data transfers the bandwidth differs heavily between different NUMA domains and GPUs. This is due to the fact that on most systems GPUs are connected to a NUMA domain and therefore have a NUMA affinity. Choosing appropriate cores to handle the GPU transfers can be achieved on multiple levels and depends on the programming paradigm used.

- During the job configuration, job scheduling system (e.g. SLURM, PBS) parameters may be used to set the correct affinities.
- Environment variables may be used to configure runtimes such to map cores and GPUs appropriately.
- Wrapper commands can be used to restrict the execution to certain cores (e.g., taskset).
- Directly in the code calls to libraries such as nvpl can be used to obtain the necessary information. Then the developer can handle the mapping directly in the code (e.g., with the OpenMP `device()` clause)

Which of these possibilities to ensure a correct mapping between CPU cores and GPUs should be used depends on the hardware resources and the runtimes used. Some clusters provide detailed information about this topic and how to configure jobs on it. If this information is not given by the provider, it can be obtained with vendor tools such as nvidia-smi or rocm-smi. These tools can provide information about the hardware topology and help identify GPU affinity.

When binding the launched MPI processes to the appropriate NUMA domains that correspond to the used GPUs, we can see that the load balance improves, as shown by Figure 4. The total runtime also improves by 20% in this example and the next iteration starts earlier.

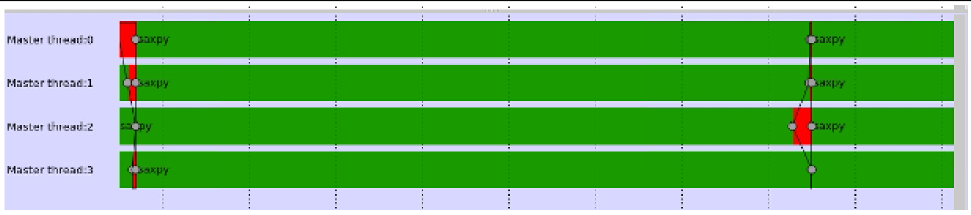


Figure 4: An example of a trace manifesting the improved load balance applying the appropriate process mapping to GPUs.

3.1.3 Load imbalance in hybrid programming (MPI+X)

Load imbalance is a longstanding issue in parallel programming and one of the primary contributors to efficiency loss in high-performance computing (HPC) systems. It occurs when the execution time required to complete the workload assigned to different processes varies, meaning some processes finish faster than others. While the most heavily loaded processes continue to compute, the less loaded ones are idle, waiting for the next synchronization point, thus wasting computational resources and reducing overall efficiency.

In Figure 5 we can see a trace showing the useful duration of two MPI processes with two threads each. The first process finalizes its execution early before the second one and its two threads are idling while the second process still executes its corresponding code.

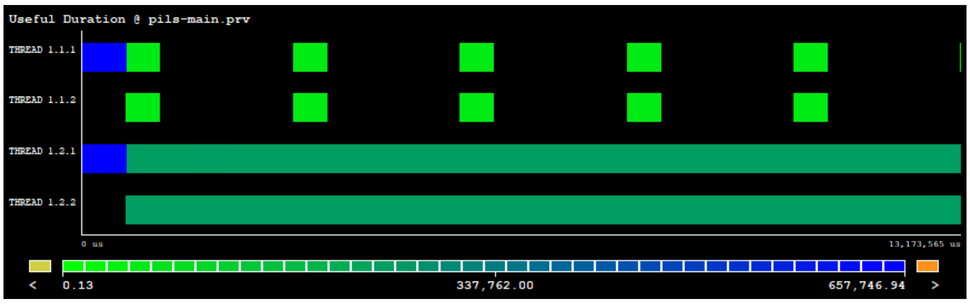


Figure 5: An example of a trace displaying useful duration executing MPI+OpenMP dlb-pils (2 MPI processes x 2 OpenMP threads).

Load imbalance can arise from various factors, and these can generally be categorized into three main sources: algorithmic factors, system functionality, and system variability.

- **Algorithmic Factors:** Load imbalance can be inherent to the algorithm itself, particularly due to uneven data partitioning or varying computational loads. A common example is uneven data partitioning, where the amount of data assigned to each process differs. Although mesh partitioning tools like Metis can be used to balance the data distribution, they often require heuristics to achieve optimal load balancing, which is not always straightforward. Additionally, even if a partition is well-balanced at the start, it may become imbalanced later in the execution. Computational imbalance can also arise in scenarios like sparse matrix calculations, where the distribution of non-zero values across processes is uneven.

- **System Functionality:** System-level factors can also introduce load imbalance. For instance, differences in Instructions Per Cycle (IPC) due to variations in data locality can lead to unequal workloads among processes. While code optimizations to improve data access patterns can mitigate this issue for specific hardware and input sets, such optimizations may not be effective across different architectures or scenarios.

- **System Variability:** Hardware and software variability can also significantly contribute to load imbalance. At the software level, factors like operating system noise or thread migration between cores can lead to imbalances. At the hardware level, issues such as network contention or manufacturing variations in components can cause discrepancies in processing speeds, further exacerbating load imbalance. Since these types of imbalances are difficult to predict and cannot be addressed through static methods, only dynamic mechanisms can help reduce or manage their impact on performance.

To address load imbalance, several strategies are commonly used, each targeting different aspects of the problem. Two major approaches are data-load balancing and computational load balancing, each with its advantages and limitations.

- **Data-Load Balancing:** This approach involves redistributing the data among processes to achieve a more even load distribution. It is one of the most widely used techniques in current solutions, such as repartitioning the mesh (e.g., PAMPA) or redistributing data (e.g., Adaptive MPI). However, data movement and mesh partitioning are costly operations, making this approach more effective for coarse-grain load imbalance, where large-scale redistribution can be tolerated. While effective, it can be computationally expensive and may not address finer-grained load imbalances efficiently.

- **Computational-Load Balancing:** In contrast, computational load balancing focuses on dynamically allocating more computational resources to processes with higher loads in order to balance the overall execution time. This strategy is particularly useful for fine-grained load balancing, as it allows for quicker adjustments compared to data movement. Since shifting computational resources is generally less costly than moving large data sets, this approach is better suited for situations where load imbalances are less predictable or fluctuate frequently

3.1.4 Using DLB to compensate load imbalance

The Dynamic Load Balancing (DLB) library manages resource allocation within a computational node to address load imbalances in parallel applications. It operates transparently to both the developer and the application, adjusting the number of threads assigned to different processes as needed. Since this solution is applied at runtime, it can dynamically resolve load imbalances that arise during execution.

DLB functions across all layers of the software stack, working in collaboration with each to optimize resource utilization. It requires two levels of parallelism: the inner level is used to enhance the resource efficiency of the outer le vel. In ty pical HP C ap plications, th is me ans parallelism at both the distributed memory level (e.g., across nodes) and the shared memory level (e.g., within a node). A common approach is to combine MPI (Message Passing Interface) for distributed memory systems with OpenMP for shared memory systems, allowing the strengths of both models to be leveraged.

Load imbalance can be particularly challenging in MPI applications because redistributing or moving data between processes is not straightforward. Despite this, MPI remains one of the most widely used programming models in HPC. DLB, however, is designed to be easily extended to support other parallel programming models. The integration of DLB with MPI is transparent to the application and user, leveraging MPI's PMPI interception mechanism. Coordination with OpenMP is based on their standard public API.

DLB supports various load-balancing policies, with one of the most common being LeWI (Lend When Idle). LeWI operates by ”lending” computational resources (CPUs) from an MPI process when it is idle, such as when waiting on a blocking call to another MPI process on the same node. Meanwhile, other processes on the same node that are still computing can ”borrow” these idle resources.

Under this policy, each CPU is owned by a single process, and ownership does not change during the lifetime of the process. However, the owner can lend the CPU to other processes. Importantly, only the owner of a CPU can reclaim it once it has been lent.

To use DLB, developers must include the DLB API in their code. While most DLB operations are automatically handled by the interception mechanism, certain tasks, such as calling `DLB_Borrow()`, may require explicit programming. Additionally, developers must link with the DLB library during the compilation process and preload it before running the application.

Figure 6 shows a trace executing the same program and inputset as in Figure 5 but in this case DLB mitigates the imbalance problem by lending the unused threads from the first process to the second one (additional threads in the trace).

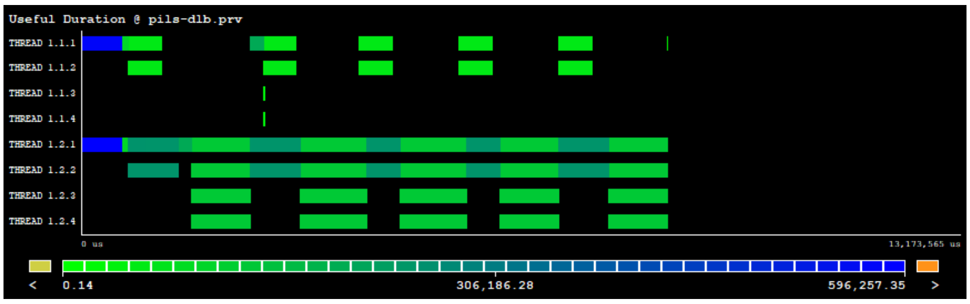


Figure 6: An example of a trace displaying useful duration executing MPI+OpenMP dlb-pils (2 MPI processes x 2 OpenMP threads) with DLB enabled.

3.2 Programming models

The following programming models were briefly characterized and listed on the website to cover the need originating from the new assessments and kernels being currently developed.

3.2.1 HIP

Cross-platform GPU programming without vendor lock-in enables greater cost efficiency and makes applications more future-proof against potential hardware changes. Applications developed relying on the Heterogeneous-computing Interface for Portability (HIP) (`https://github.com/ROCm/HIP/`) are capable of targeting AMD and NVIDIA GPUs through the vendor-specific ROCm and CUDA platforms. Its API is designed to closely resemble CUDA, which eases conversion from existing applications, either manually or relying on the HIPIFY translation tools, as well as the development of new applications from scratch by minimizing the learning curve for developers. Another feature of using HIP is that applications can be profiled and debugged using AMD/NVIDIA tools.

3.2.2 OpenMP offloading

Since OpenMP version 4.0, the standard has introduced support for heterogeneous systems, which consist of a host architecture and one or more external accelerator devices. The host architecture is where the program begins its execution, while the target accelerators (such as

GPUs) are external devices attached to the host, capable of executing portions of the computation. As a key feature, the OpenMP offload model enables performance portability across different HPC clusters by abstracting the user from device-specific architectures.

OpenMP facilitates offloading tasks to these accelerators using the target construct, which allows both data and code to be transferred from the host to the target device for execution. Additionally, OpenMP provides a set of specialized API routines for managing operations specific to devices, such as querying device information, handling data management, and managing thread hierarchies. The standard also includes environment variables that can be set at runtime to configure how the device executes kernels.

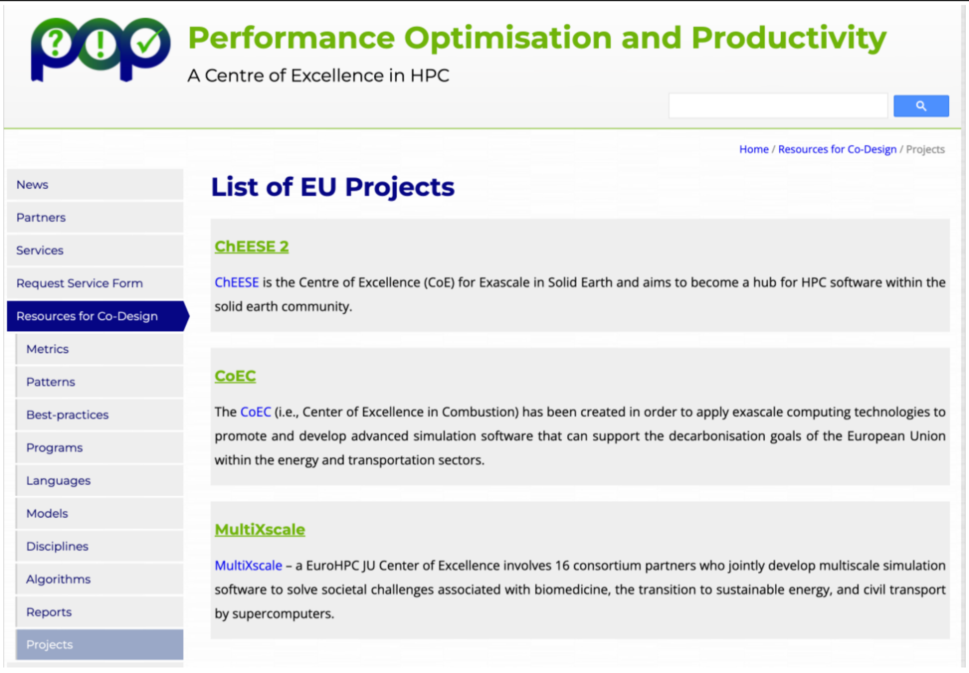
The typical workflow for executing kernels on a device involves three main steps: 1) The host maps its data to the target device's memory environment; 2) The host offloads OpenMP target regions to the device for execution, potentially reusing the data environment to execute multiple regions; and 3) The host retrieves the computed results from the device and transfers the data back to the host.

3.3 Projects collection

The one of main goals of the POP3 CoE is close collaboration with the other EuroHPC CoEs and projects. With the growing number of performance studies coming from campaigns, meaning that they target the specific CoE's codes, there is a need for assigning the published reports to those projects for several reasons. E.g., the same code is developed and assessed under multiple projects - the link helps with the identification of the particular report. Some readers visiting the codesign website might be interested in the subset of reports that belong to a particular campaign or CoE. The following projects are currently identified:

- CEEC
- ChEESE 2
- CoEC
- EXCELLERAT P2
- MultiXscale
- SPACE

Figures 7, 8, and 9 illustrate the design and integration of the projects collection into the codesign website.



List of EU Projects

ChEESE 2

ChEESE is the Centre of Excellence (CoE) for Exascale in Solid Earth and aims to become a hub for HPC software within the solid earth community.

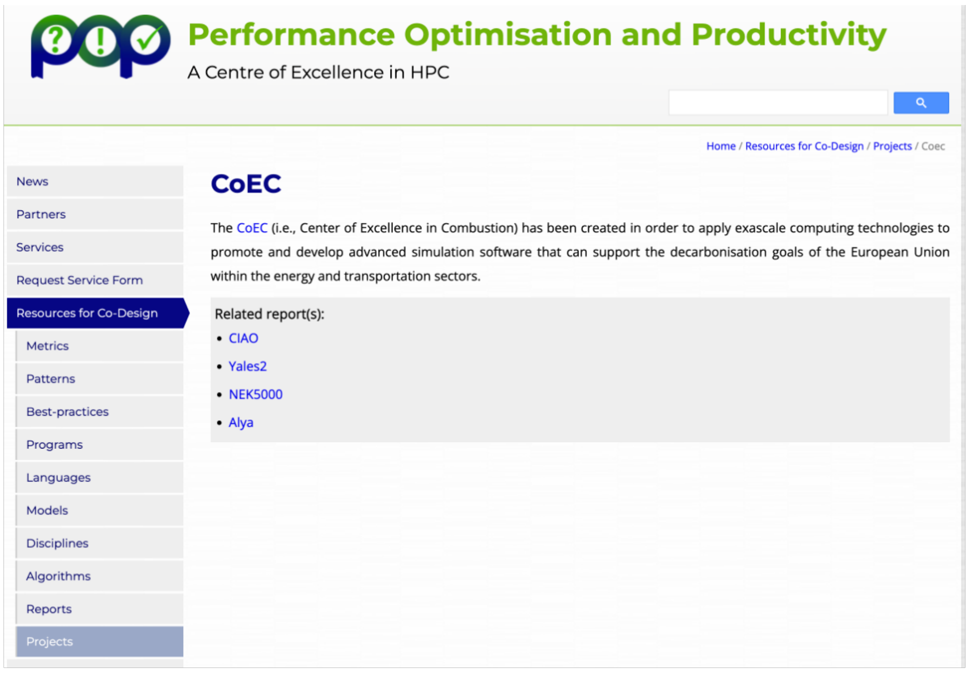
CoEC

The CoEC (i.e., Center of Excellence in Combustion) has been created in order to apply exascale computing technologies to promote and develop advanced simulation software that can support the decarbonisation goals of the European Union within the energy and transportation sectors.

MultiXscale

MultiXscale – a EuroHPC JU Center of Excellence involves 16 consortium partners who jointly develop multiscale simulation software to solve societal challenges associated with biomedicine, the transition to sustainable energy, and civil transport by supercomputers.

Figure 7: The new Projects collection listing the EU projects with a connection to the published assessment reports.

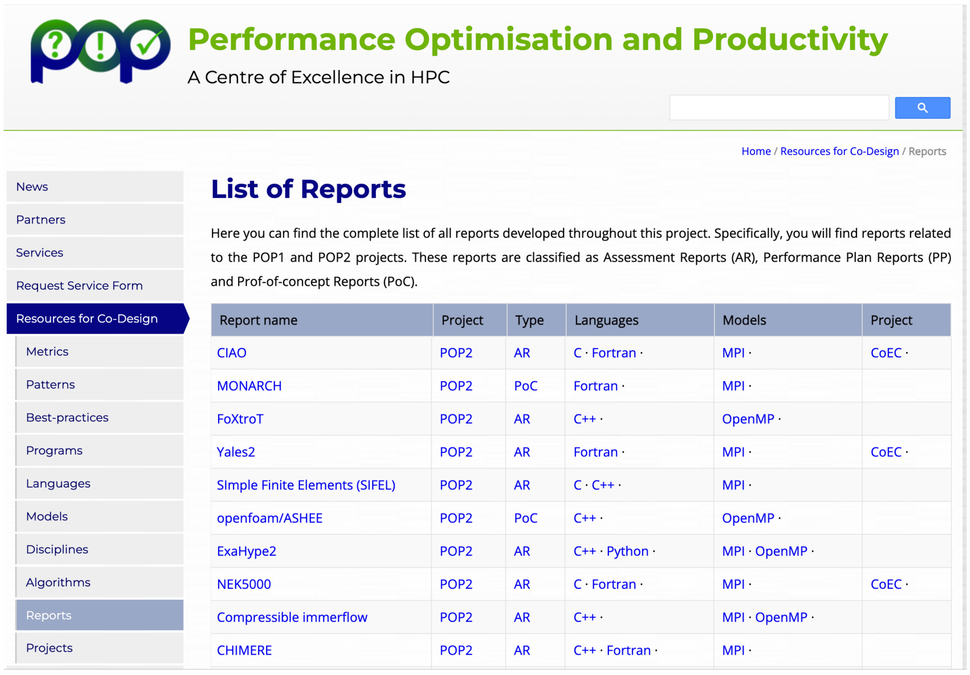


CoEC

The CoEC (i.e., Center of Excellence in Combustion) has been created in order to apply exascale computing technologies to promote and develop advanced simulation software that can support the decarbonisation goals of the European Union within the energy and transportation sectors.

Related report(s):
- CIAO
- Yales2
- NEK5000
- Alya

Figure 8: The detail of an EU project item from the new Projects collection listing the related code assessment reports.



List of Reports

Here you can find the complete list of all reports developed throughout this project. Specifically, you will find reports related to the POP1 and POP2 projects. These reports are classified as Assessment Reports (AR), Performance Plan Reports (PP) and Prof-of-concept Reports (PoC).

Report name	Project	Type	Languages	Models	Project
CIAO	POP2	AR	C · Fortran ·	MPI ·	CoEC ·
MONARCH	POP2	PoC	Fortran ·	MPI ·	
FoXtroT	POP2	AR	C++ ·	OpenMP ·	
Yales2	POP2	AR	Fortran ·	MPI ·	CoEC ·
SImple Finite Elements (SIFEL)	POP2	AR	C · C++ ·	MPI ·	
openfoam/ASHEE	POP2	PoC	C++ ·	OpenMP ·	
ExaHype2	POP2	AR	C++ · Python ·	MPI · OpenMP ·	
NEK5000	POP2	AR	C · Fortran ·	MPI ·	CoEC ·
Compressible immerflow	POP2	AR	C++ ·	MPI · OpenMP ·	
CHIMERE	POP2	AR	C++ · Fortran ·	MPI ·	

Figure 9: The table with the published assessment reports extended by the Project column linking the reports with their projects.

4 EPI Methodology

4.1 RISC V

The EPI RISC-V methodology is defined by three different components: the target platforms (including the performance reference platforms), the tool-chain (all the components in the software stack used in the codesign analysis), and the methodology plan (a sequence of studies to carry out the analysis).

In terms of the RISC-V platforms, we are using:

- **HiFive Unmatched**: This scalar core (four cores per socket) running at 1.2 GHz is our development platform and reference for performance studies. It shares the file system through NFS with the other platforms.
- **Pioneer MILK-V**: The nodes of this system are based on the SOPHON SG2042 processor and constitute a more recent design with up to 64 cores per socket, supporting the RVV0.7 ISA at 2.0 GHz with a very short vector length of 128 bits. This makes it a great comparison platform for a very different design point to our EPI long vector architecture.
- **EPAC_1.5@FPGA**: This is the EPAC text chip produced in EPI-SGA2. The chip implements the RVV0.7 ISA with MAXVL of 256 doubles (16K bits). We can run a full Linux image with access to the shared file system through NFS. We can run it at 333 MHz or 1 GHz. The memory bandwidth is limited due to an issue in the chip's PHYs. The system is fully functional and in a sense experimenting with the two frequencies we can evaluate the behavior of the core micro-architecture under long memory latencies. Being the memory latency tolerance one of the key design visions of the EPI (when combined

with long vector executions), this constitutes a very interesting evaluation point in the design space.

- **FPGA-SDV**: This system implements the current RTL of the EPAC core on an FPGA running at 50 MHz. This emulator runs Linux at sufficient frequency for a reasonable interactive performance. We can certainly argue that the high memory bandwidth is provided by real DRAM devices while the processor clock is only 50 MHz. We should be also aware that the memory controller is run at 50Mhz, thus resulting in a memory subsystem only partially over-dimensioned. Additionally, the environment supports changes in the memory subsystems such as throttling the memory bandwidth or dilating latency by adding the desired number of cycles. The platform provides relatively detailed observability of micro-architectural signals at different levels, including graduating program counter and vector instruction at the top of the reorder buffer, L2 cache hits and misses, and data transfers between scalar and vector core for example. Waveforms for half a million cycles can be captured.

The different platforms constitute a great infrastructure to evaluate and compare a fairly wide set of design points and gain insight useful for improving the EPI design. It would certainly be also interesting to have detailed micro-architectural observability for systems other than the FPGA-SDV. This is the very first insight deriving from the EPI experience and the study presented below.

Concerning the tool-chain, we are using:

- **LLVM/Clang 12.0.0**: The compilation chain developed within the EPI project. At the same time it is an enabler to generate the binaries we will use in the codesign experiments.
- **LLVM/Clang 20.0**: The new compilation chain used within the EPI project to specifically generate RVV 1.0. That tool-chain is used exclusively in the RAVE@QEMU studies when targeting the new vector specification for RISC-V
- **Extrae 4.2.0**: The generic BSC Tools instrumentation package is available in all the platforms and can be used to get Paraver traces at the level of code regions/functions, including hardware counter information. These traces include a lot of detailed metrics, which can be analyzed and visualized using Paraver.
- **Ila2prv (latest: 2024-11)**: This tool works on the FPGA-SDV waveforms, translating them to the Paraver format supporting extremely detailed analyses being applied to them.
- **RAVE@QEMU (latest: 2024-11)**: This QEMU plugin can generate Paraver traces with instruction sequences (vector and/or scalar) and information such as vector lengths, registers used or memory address patterns. Even if it does not include actual timing, just the count, or sequence of instructions is very useful for architectural codesign.

Finally, the methodology plan is currently defined by:

- **Elapsed time study**, it gives as the result the timing and some aggregated information derived by the application itself (e.g., computed throughput, bandwidth, etc.). In order to fairly compare the different platforms, we report if possible the normalized performance per cycle of the platform.
- **Extrae study**, it provides a deeper level of detail. It may contain traces, histograms, and aggregated POP metrics.

- **Architectural study** After this initial exploration, the codesign analyst may decide to continue the analysis following two different paths (not mutually exclusive):
  - **RAVE@QEMU**, it offers details on how the program behaves on a generic RISC-V architecture. Running on top of RAVE@QEMU the programmer simulates the vector instructions on top of a RISC-V QEMU virtual machine. The resulting trace provides insights into the functional behaviour of the compiler-generated code.
  - **Waveform, ILA traces**, it provides the specific behaviour of the RISC-V instructions on top of the codesign target architecture.

4.2 RHEA

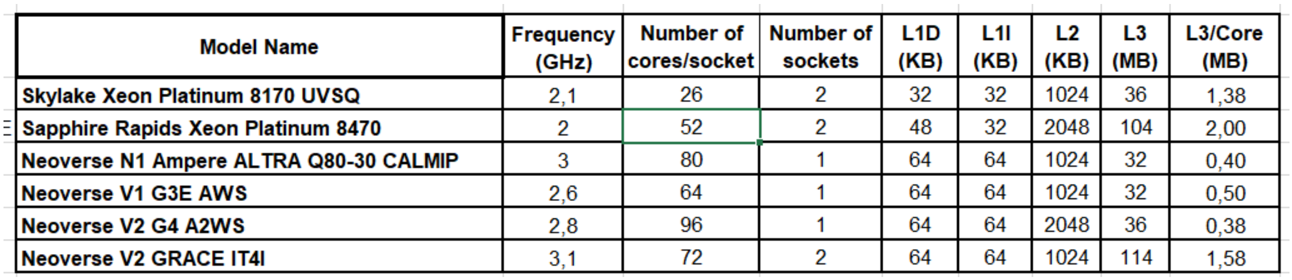
High compiler quality, i.e., the performance of the generated CPU code, is of primary importance in particular for newcomers on the market such as SiPearl. SiPearl is developing the processor RHEA based on an ARM Neoverse V1 core interfacing with a combined High Bandwidth Memory (HBM) and traditional DDR5 memory system. As a companion effort, SiPearl is also building a software stack environment including a specific c ompiler b ased o n L LVM to take full advantage of RHEA architecture.

Our goal is to support SiPearl's effort i n t he c ompiler a rea, i n p articular i dentifying worthwhile optimizations as well as counterproductive ones and correlate them with context. For that goal, we develop specific t echnology t o q uantitatively a nd c omparatively assess t he quality of the generated CPU code. The main goals are first t o i dentify w eaknesses (mistakes) i n the CPU code generation process and second to compare different c ompiler o utputs, T hen a s a final s tep, w e t ry t o b ackport b etween c ompilers, o ptimizations w hich h ave b een m issed: for example, for the same loop, Compiler A might have failed to vectorize (due to a poor data dependence analysis) while compiler B (using a more advanced analysis) might have succeeded. The comparative analysis has to be carried out not only between compilers but also between options/flags f or t he s ame compiler.

For achieving this detailed compiler output analysis, we rely on MAQAO toolset which will provide us first with detailed profiling analysis at 3 levels (whole application, function and loop). This 3-level analysis is essential because it allows us to detect in comparative studies the main location of performance difference b etween c ompiler o ptions o r c ompilers. U nfortunately, it does not reveal the main issues: compiler mistakes leading to CPU code with low performance. For this latter objective, we used MAQAO detailed binary analysis combining static analysis, simplified s imulations a nd m easurements. S uch a n a nalysis d etects c ompiler d eficiencies and estimates related performance impact.

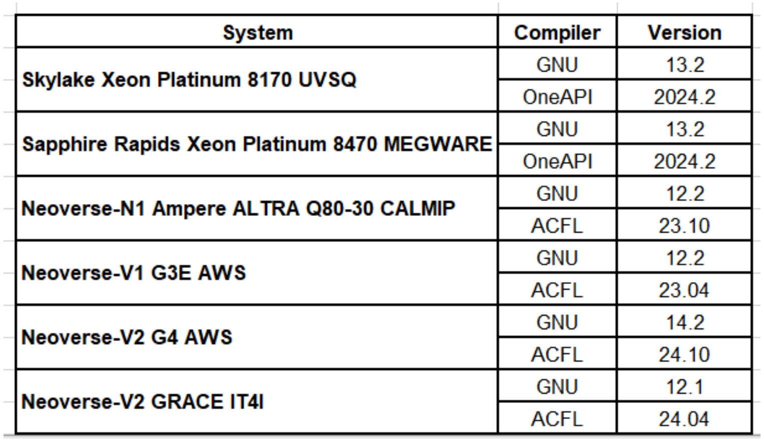
In our testing, initial performance testing has been carried out by the kernel author on a limited number of systems: such results will be reported in the subsections “Initial Performance Assessment”. Then we will perform systematic performance testing on various ARM-based platforms (Graviton 3E, Graviton 4, Ampere Altra, Grace) with different c ompilers: GCC/G-FORTRAN, ACFL (ARM Compiler for Linux based on LLVM). For each compiler, several options were also tested but for the sake of simplicity only `-O3` and `-Ofast` will be reported. The three instruction sets (NEON, SVE, SVE2) have been systematically tested and evaluated. Additionally, two X86-64 systems were used as references using the new OneAPI compilers (combining LLVM and INTEL technology) and the old intel classic compilers which have remarkable vectorization capabilities. All of the system details (number of cores, cache sizes) and compilers (names and versions) used are listed in Figures 10 and 11.

All of the kernels (source code) and performance results have been shared with SiPearl. SiPearl's compiler group tested their in-house compiler technology on the kernels and sent us back the obtained results. This cooperation effort will be pursued during the whole project.



Model Name	Frequency (GHz)	Number of cores/socket	Number of sockets	L1D (KB)	L1I (KB)	L2 (KB)	L3 (MB)	L3/Core (MB)
Skylake Xeon Platinum 8170 UVSQ	2,1	26	2	32	32	1024	36	1,38
Sapphire Rapids Xeon Platinum 8470	2	52	2	48	32	2048	104	2,00
Neoverse N1 Ampere ALTRA Q80-30 CALMIP	3	80	1	64	64	1024	32	0,40
Neoverse V1 G3E AWS	2,6	64	1	64	64	1024	32	0,50
Neoverse V2 G4 A2WS	2,8	96	1	64	64	2048	36	0,38
Neoverse V2 GRACE IT4I	3,1	72	2	64	64	1024	114	1,58

Figure 10: Hardware configurations tested



System	Compiler	Version
Skylake Xeon Platinum 8170 UVSQ	GNU	13.2
	OneAPI	2024.2
Sapphire Rapids Xeon Platinum 8470 MEGWARE	GNU	13.2
	OneAPI	2024.2
Neoverse-N1 Ampere ALTRA Q80-30 CALMIP	GNU	12.2
	ACFL	23.10
Neoverse-V1 G3E AWS	GNU	12.2
	ACFL	23.04
Neoverse-V2 G4 AWS	GNU	14.2
	ACFL	24.10
Neoverse-V2 GRACE IT4I	GNU	12.1
	ACFL	24.04

Figure 11: Compiler versions tested

5 Evaluation and analysis

5.1 JUKRR kloop

The Juelich KKR code family (JuKKR) is a collection of codes developed at Forschungszentrum Juelich implementing the Korringa-Kohn-Rostoker (KKR) Green's function method to perform density functional theory calculations. Since the KKR method is based on a multiple scattering formalism it allows for highly accurate all-electron calculations. For more details see https://co-design.pop-coe.eu/programs/jukkr-kloop/index.htm

Given the availability of the JuKRR code in the POP codesign resources repository, we decided to use it as a starting point for the EPI codesign activity in POP3.

5.2 JuKRR on RISC-V

Being an application unknown to the POP3 RVV codesign team, we started trying to run it in MareNostrum 5 as a first step to identify its structure and basic characteristics. The code is written in FORTRAN, the build system is based on CMake and uses OpenMP for parallelism within a node. We obtained Extrae traces of a first version of the code. A timeline view of the code is presented in Figure 12. As can be seen, long parallel regions were not yet parallelized and MKL seems to be used intensively in other regions. A second version of the code was also run showing a fairly different behaviour (Figure 13). In this case, the whole run is far more parallel but the analysis of the cache misses revealed the code fitted in the L2 cache of Marenostrum, which was interpreted as the kernel being an oversimplification of a more realistic code. Taking also into account that the FORTRAN compiler development efforts in EPI target the RVV 1.0 ISA and that the RVV 1.0 FPGA-SDV was not yet available we decided to focus our codesign studies effort on the SPMXV code (see Section 5.5), leaving JuKKR for later analysis on the RVV platforms.

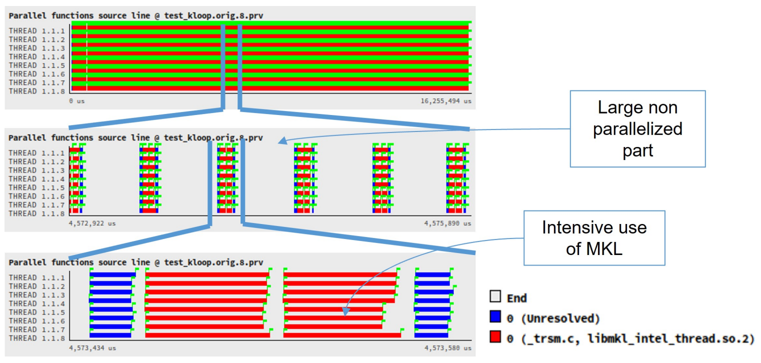


Figure 12: Timeline showing the structure (parallel functions) of the OpenMP parallelization at Marenostrum 5 for the first version of JuKRR.

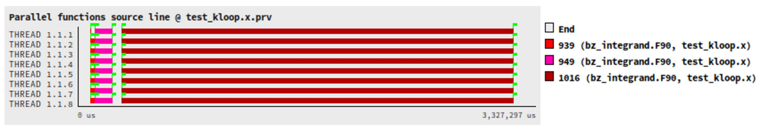
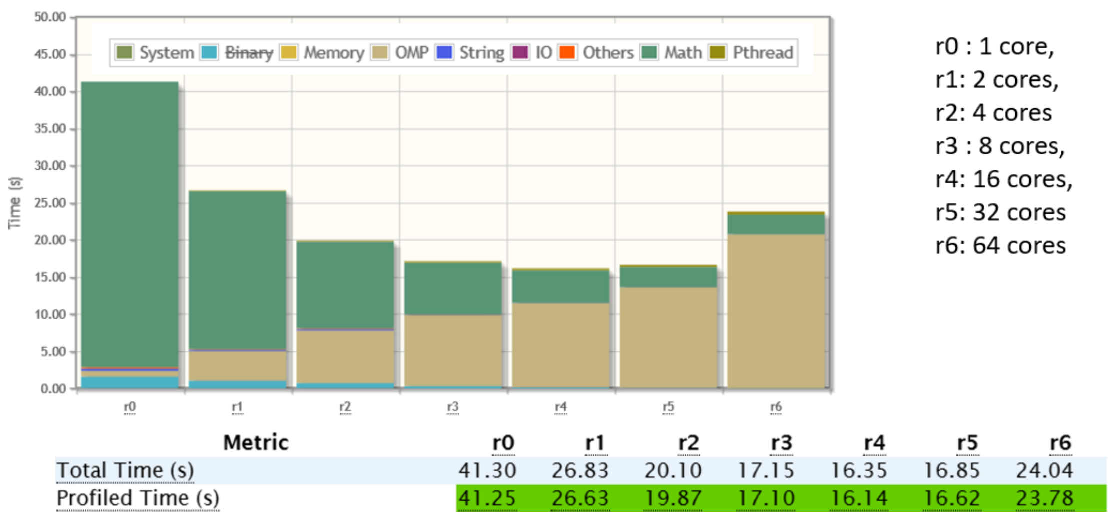

Parallel functions source line @ test_kloop.x.prv
THREAD 1.1.1
THREAD 1.1.2
THREAD 1.1.3
THREAD 1.1.4
THREAD 1.1.5
THREAD 1.1.6
THREAD 1.1.7
THREAD 1.1.8
0 us
3,327,297 us
End
939 (bz_integrand.F90, test_kloop.x)
949 (bz_integrand.F90, test_kloop.x)
1016 (bz_integrand.F90, test_kloop.x)

Figure 13: Timeline showing the structure (parallel functions) of the OpenMP parallelization at Marenostrum 5 for the refactored version of JuKRR.

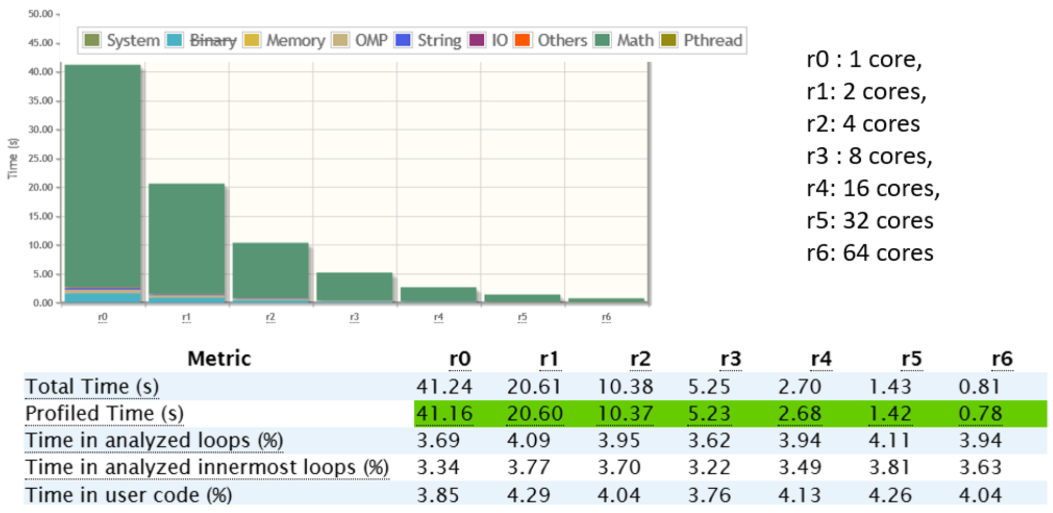
5.3 JuKRR on RHEA

The original JuKRR code has a very peculiar behaviour. As mentioned in the original POP2 JUKRR analysis, it scales up very poorly. Our runs on a Graviton 3E with 64-core showed exactly the same behaviour: a 64-core run hardly achieves a speedup of 1,5 over a single-core run (see Figure 12). Furthermore, the time spent in OpenMP is regularly increasing when increasing the number of cores.



Metric	r0	r1	r2	r3	r4	r5	r6
Total Time (s)	41.30	26.83	20.10	17.15	16.35	16.85	24.04
Profiled Time (s)	41.25	26.63	19.87	17.10	16.14	16.62	23.78

Figure 14: Performance of Original JuKRR: scalability test on a 64 cores Graviton 3E



Metric	r0	r1	r2	r3	r4	r5	r6
Total Time (s)	41.24	20.61	10.38	5.25	2.70	1.43	0.81
Profiled Time (s)	41.16	20.60	10.37	5.23	2.68	1.42	0.78
Time in analyzed loops (%)	3.69	4.09	3.95	3.62	3.94	4.11	3.94
Time in analyzed innermost loops (%)	3.34	3.77	3.70	3.22	3.49	3.81	3.63
Time in user code (%)	3.85	4.29	4.04	3.76	4.13	4.26	4.04

Figure 15: Performance of Optimized JuKRR: scalability test on a 64-core Graviton 3E

The optimized version developed during the POP2 project moves the parallel region to an upper level. This version performs much better (see Figure 15: the speedup on a 64-core Graviton 3E is now over 54. With this optimized version, the time spent in OpenMP is now negligible. Now, on the other hand, more than 90 percent of the time is spent in ZGEMM (dense matrix multiply operation) performed using the ARMPL library. So to some extent, the optimized JuKRR kernel ends up being simply a test of Dense Linear Algebra. Furthermore, the matrix sizes are large enough so the library achieves good performance.

5.4 SPMXV Initial performance assessment

The Sparse Matrix-Vector (SPMXV) kernel analyzed in this section corresponds to the implementation presented in Section 2.3. The structure of this code corresponds to a set of nested loops traversing the matrix layout, collecting the elements in a single row (expressed in CSR format) and computing the result via the reduction of a scalar value.

The SPMXV kernel was tested on two systems:

- A dual-socket Intel Xeon Platinum 8468 (Sapphire Rapids) system (2x48 cores, 2.1 GHz)
- A single socket Nvidia Grace CPU (1x72 cores, 3.0 GHz)

With approximately 5 percent, both systems reached a similar fraction of their theoretical peak performance (see Figure 17. The theoretical peak performance of the systems was calculated using their available FMA streams: Ppeak= Cores × 64bit FMA instructions per cycle × frequency As figures 16 and 17 show, the achieved maximal performance does not depend upon the compiler except for the Intel CPU, where the GCC compiler lags behind the Intel compiler (explanations are given in the detailed analysis).

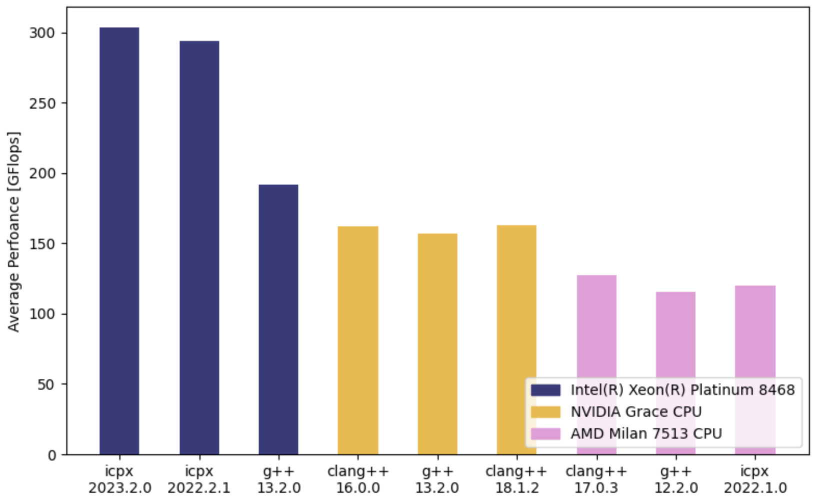


Figure 16: GFlops performance of SPMXV kernel

Further, the code was evaluated against BLAS like libraries that implement SPMXV on the Grace CPU. The tested libraries, ARM PL and Nvidia PL, reached around 3% of the peak performance on the Grace CPU with the given matrices. The developed SPMXV kernel outperforming the architecture-specific libraries indicates the lack of highly optimized SPMXV libraries on the ARM Neoverse V2 architecture so far. However, other libraries that implement SPMXV have not yet been evaluated.

Finally, the behaviour of the kernel was tested as the size of the problem increased (see Figure 18. The similar cache sizes on both machines lead to a similar scaling behaviour, with a performance drop at the L3 cache size line. However, with very large matrix sizes where the matrix does not fit into the L3 cache, the Grace CPU provides better performance compared to the Sapphire Rapids.

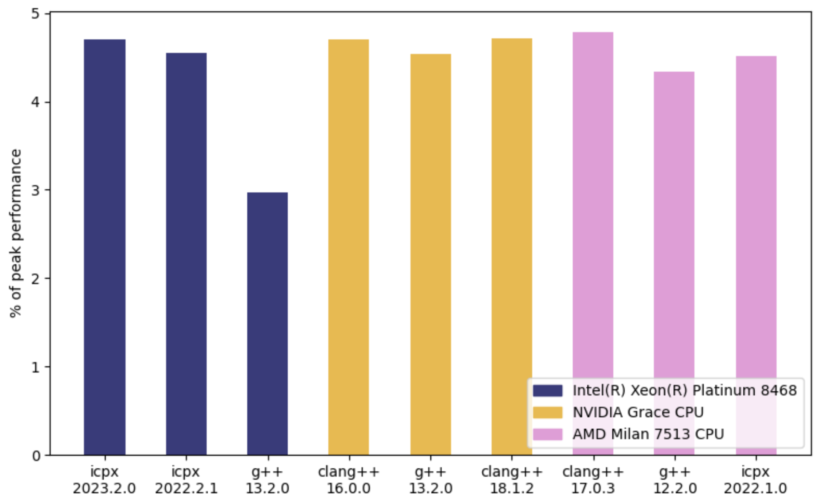


Figure 17: Fraction of peak performance for SPMXV kernel

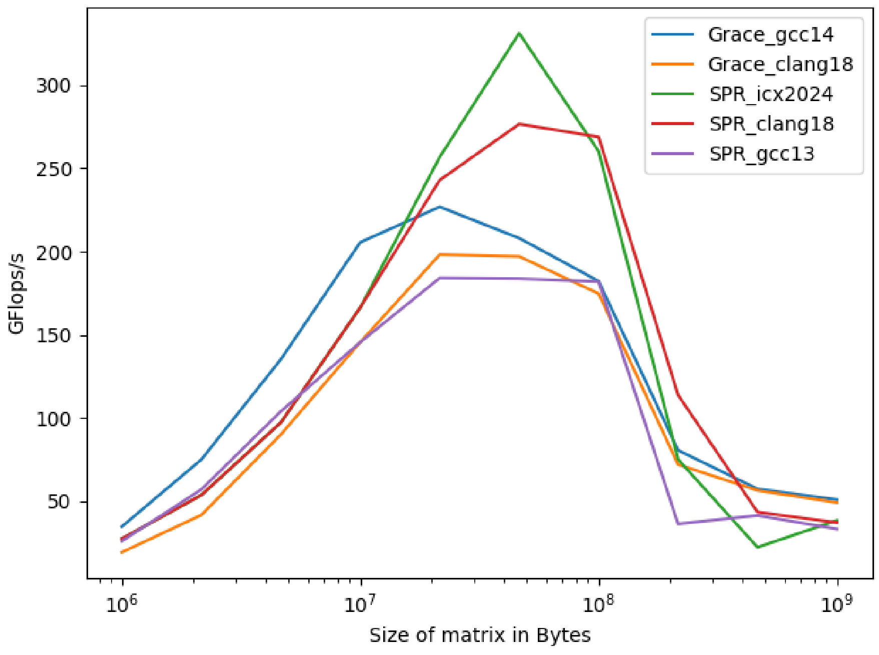


Figure 18: Impact of Datasize on SPMXV performance

5.5 SPMXV on RISC-V

In this section, we present the complete study of the SPMXV kernel for the EPI RISC-V architecture. The nature of this kernel allows us to explore different aspects of this code (e.g., gathering sparse elements or computing partial reductions) for the RISC-V vector extensions. In the following subsections, we will explore the elapsed time behaviour and provide further insights using Paraver traces for different RISC-V architectures. The study will also include the architectural profile of this kernel execution allowing to reach the assembly level of the ISA.

5.5.1 SpMV Elapsed time study

Following the described methodology, the first step was to compile and run the source code on the different p latforms. To r un o n t he H iFive a nd P ionner t he M akefile ha d to be ad apted to use the proper compiler and compiler flags as well as l ibraries. The first run of the same binary on the EPAC1.5 failed not finding t he O penMP L ibrary. A fter l oading t he p roper module the run was successful. We interpret this experience as another codesign insight in the sense that supporting standard environments (e.g., compiler, modules, ...) is a very important design target.

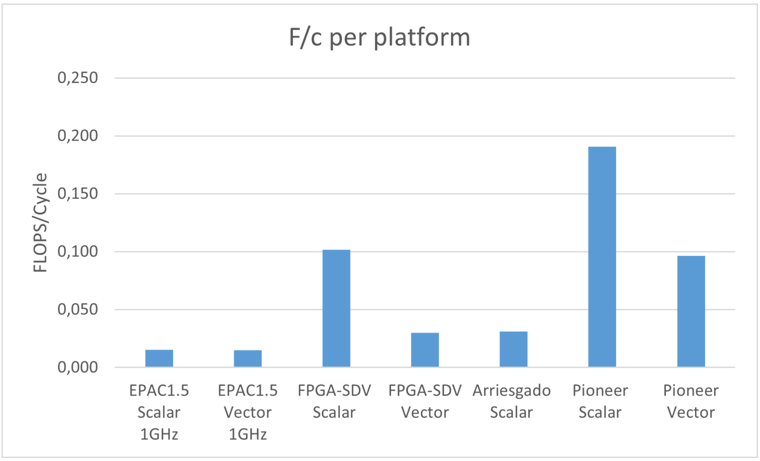


Figure 19: Normalized FLOPS per cycle for the different versions of the code (scalar/vector) and platform

The normalized performance (FLOPS/cycle) for the different platforms is reported in Figure 19. The best scalar execution performance is achieved by the Pioneer system. It is followed by the FPGA-SDV system, which has around half its performance. The HiFive and EPAC1.5 fall clearly behind. In the case of the EPAC1.5 we speculate that the high relative memory latency has an important impact towards such low performance.

Regarding vector code, the performance shrinks by half compared to the scalar case for the Pioneer, and about one-third in the case of the FPGA-SDV. In the case of the EPAC1.5 at 1 GHz, the performance is similar to the scalar performance.

The fact that the NNZ (Number of Non-Zero elements) is only 15, far below the design target of EPAC would reinforce the interest in vectorization approaches that exploit longer vector lengths for this problem.

Although the results are bad for the vector run, we need to understand why and get further insight into how the design can be improved beyond moving towards longer vector lengths.

We also studied in terms of elapsed time, the performance when scaling the number of cores in the available multicore platforms. The scaling results are shown in Figure 20. The results show a change in the slope at 4 cores in the Pioneer platform and again at 16 cores, which might be linked to the internal micro-architecture. Anyhow, the increase in performance with core count seems to indicate that memory bandwidth saturation, something one typically would suspect for an algorithm with low computational intensity, is not a real problem. A single core not being able to use a relevant fraction of the memory bandwidth neither for a scalar nor vector code is something we consider undesirable and something the EPI architecture should target.

Figure 21 shows the same numbers in terms of absolute performance in FLOPS/cycle where we can compare the different p latforms. In the figure, we can observe that the vector code at the FPGA-SDV with a single core results in about the same normalized performance as the 4 cores

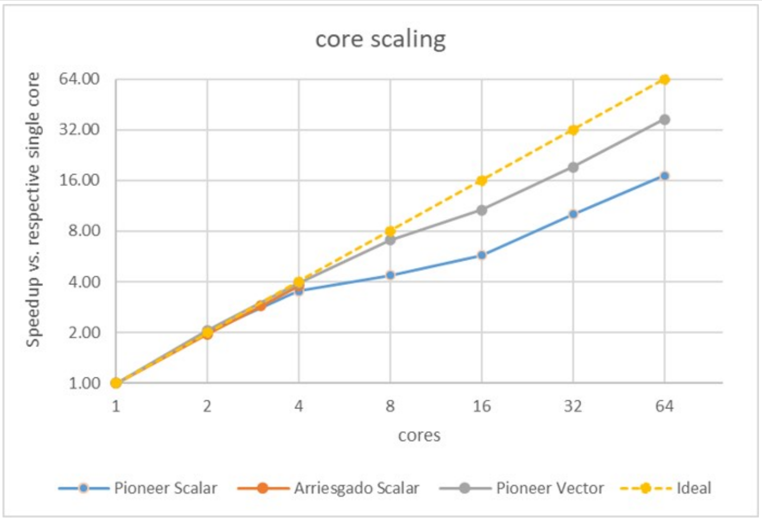


Figure 20: Scalar code OpenMP speedup with number of cores for the three multicore platforms

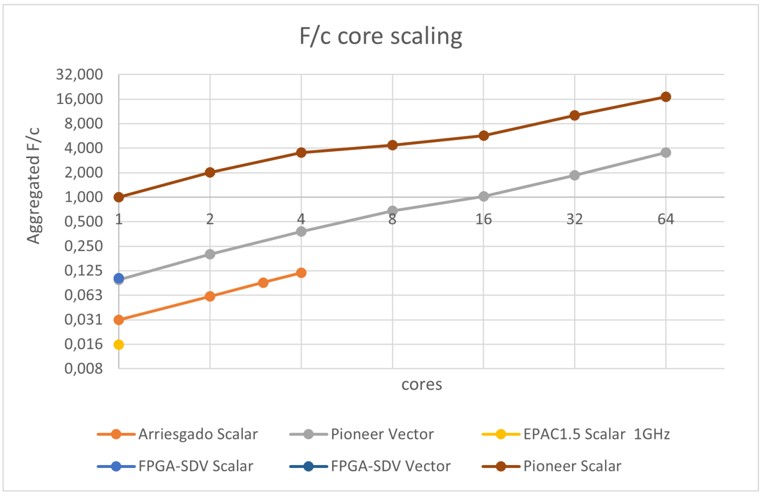


Figure 21: Aggregated FLOPS per cycle when scaling the number of cores in the different platforms

SiFive platform. This number should actually be larger, something we will further investigate later in this document.

A final experiment we performed measures the impact of increasing memory latency on both scalar and vector versions. This experiment can only be done in the FPGA-SDV platform. The result is presented in Figure 22. It shows again the better performance of the scalar code under the fast memory subsystem, something that will be investigated later. What we nevertheless want to highlight at this point is the lower sensitivity of the vector EPAC architecture to memory latency, one of the fundamental objectives of the EPI design.

5.5.2 Extrae study

The next level of the evaluation uses Extrae traces to get a deeper insight into the behaviour. We instrumented the main phases of the code including reading the matrix, initializing data structures and each of the invocations of the SpMV kernel. Besides that, we typically activate

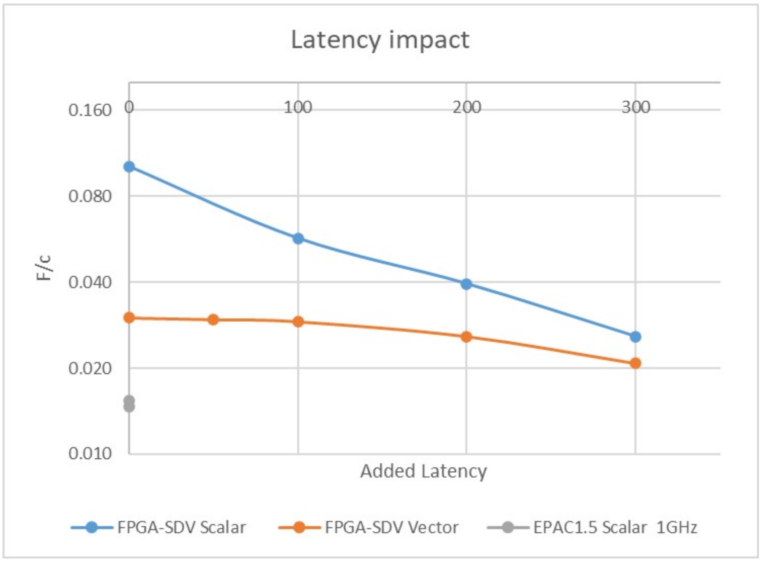


Figure 22: Impact of additional memory latency on FLOPS per cycle for the scalar and vector version of the code in the FPGA-SDV

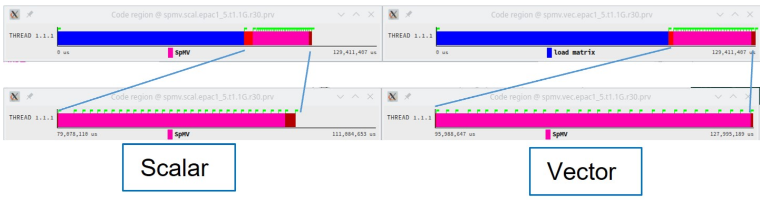


Figure 23: Timeline (full run and zoom on the execution of 30 SpMVs) for the scalar and vector version of the code at the same time scale.

the sampling mechanism every 10 milliseconds so that we can observe behaviour changes along the run if they appear. The traces can be obtained for the scalar and vector codes on all platforms. In Figure 23 we compare the timelines of phases for the scalar and vector runs at EPAC1.5 1GHz, showing a somewhat faster scalar code. Quantitative metrics derived from the traces are shown in table 24. We can see the number of instructions executed by the scalar code is about 3.3 larger than by the vector code. Considering we expect every element computation (matrix row by corresponding vector elements) to be vectorized and the known NNZ we would probably have expected a ratio closer to 15. We need to investigate the binary code generated for the vector code by the compiler. Also curious is that only 17 percent of vector instructions are memory-related, while the source code has very low arithmetic intensity. The scalar IPC is about 4.9 better than the vector IPC. Again, to break even we would expect the ratio to be 15. The utilization of the vector unit (VPU) reported by the hardware counter is nevertheless very high (0.93) even with less than 7.4 percent of instructions.

A similar comparison between the scalar runs on EPAC1.5 at 1 GHz and the EPAC FPGA-SDV at 50 MHz is shown in Figure 25 and table 26. EPAC1.5 is 4.21x faster than the FPGA-SDV even if the clock is 20x faster. A slightly higher number of instructions in the FPGA-SDV platform may indicate the hardware counter mechanism is also counting the scalar instructions executed by the sampling software within Extrae. This is a potential codesign input for the tools/platform development/maintenance teams, although we do not consider it to be a very important perturbation while producing very useful information.

	duration	Instr	Cycles	IPC	Vec. Instr	VPU %	L1_MPKI	TLB MPKI	Arith mix Vec	VPU FP MIPS	VPU FMA MIPS
EPAC1.5 Scalar 1GHz	760.8602	83274674	743042224	0.112	0	0.00	47.13	2.02			
EPAC1.5 Vector 1GHz	1054.045	25031164	1074528228	0.023	1840824	0.93	408.30	4.16	0.83	4.14	0.69

Figure 24: Metrics for the Scalar SpMV code computed from the traces in the FPGA-SDV and EPAC1.5 platforms

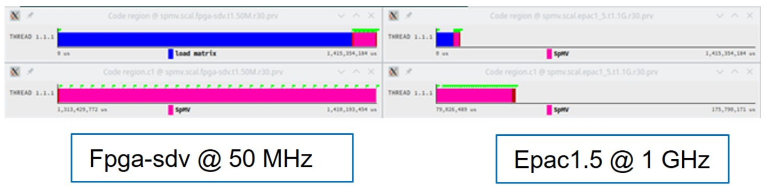


Figure 25: Timeline for the vector version of the code on the FPGA-SDV and EPAC1.5 platforms at the same time scale.

The IPC is significantly better in the FPGA-SDV case, most probably linked to the fairly large memory latency in the EPAC1.5. The impact of this latency could be further studied using the memory dilation features of the FPGA-SDV. This is done later in the document with the waveform analyses.

We have not really analyzed the cache and TLB MPKIs. That is one of the aspects that we should cover in future work for this study.

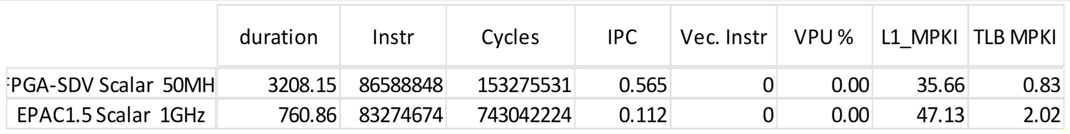
A final observation clearly apparent in the traces (Figures 23 and 25) is the very slow matrix read. This matches the experience in the previous timing runs and suggests the importance of implementing reverse offloading capabilities (further elaboration later in the document).

5.5.3 Architectural study

The next level of analysis is to look at waveforms (i.e., how the internal processor signals are propagated over time) with cycle-level accuracy for half a million cycles at the FPGA-SDV. For every run, we trigger the acquisition 9 times in sequence after the program prints the reading of the matrix has finished. With this mechanism, we obtain 9 waveform traces, which is enough to capture the behaviour in the SpMV computation. In fact, we obtain in some cases also waveforms from the initialization, which were also vectorized and can be used for further codesign insight beyond the main computational kernel.

Figure 27 shows the PC along time of the graduating instruction for the scalar code run at EPAC-SDV, Even if we have no access to the actual scalar instruction being executed, we can identify the loop structure of the code and its correlation to external memory subsystem activity like L2 cache hits and misses. The figure shows two loops, with 15 iterations of the innermost for each of the outer iterations. This matches the source code structure where the innermost loop corresponds to one iteration of the row computation and the outermost iteration corresponds to one element (Row by vector). We can really see the impact of cache misses, representing a global latency of 75 cycles and stalling the instruction graduation for that time.

The corresponding traces for the vector code are shown in Figure 28. Now we can see the instruction at the Top of the Reorder Buffer (ROB) and a bunch of other metrics for one iteration of the innermost vectorized loop which should correspond to one element (row by vector computation). Many important observations are apparent from the figure. We certainly see that the vector length of 15 is used in some of the main instructions, particularly loads and gathers. Longer vector lengths are nevertheless used for other instructions. This derives



	duration	Instr	Cycles	IPC	Vec. Instr	VPU %	L1_MPKI	TLB MPKI
PGA-SDV Scalar 50MH	3208.15	86588848	153275531	0.565	0	0.00	35.66	0.83
EPAC1.5 Scalar 1GHz	760.86	83274674	743042224	0.112	0	0.00	47.13	2.02

Figure 26: Metrics for the Scalar SpMV code computed from the traces in the FPGA-SDV and EPAC1.5 platforms

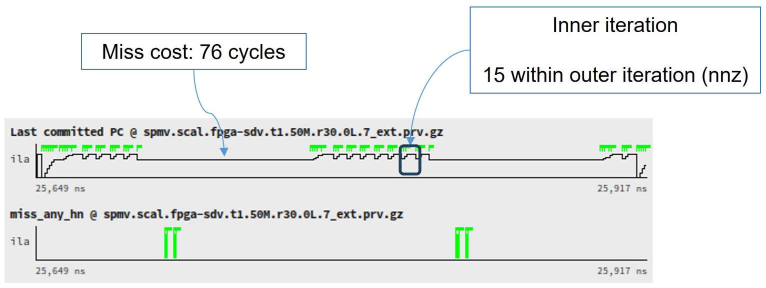

Miss cost: 76 cycles
Inner iteration
15 within outer iteration (nnz)
Last committed PC @ spmv.scal.fpga-sdv.t1.50M.r30.0L.7_ext.prv.gz
ila
25,649 ns
25,917 ns
miss_any_hn @ spmv.scal.fpga-sdv.t1.50M.r30.0L.7_ext.prv.gz
ila
25,649 ns
25,917 ns

Figure 27: Program counter and cache hits/misses timelines of one iteration of the row loop in the waveform for the scalar execution at the EPAC-SDV.

from the basic mechanism to implement reductions by the compiler. Being generic in case the overall reduction to perform (in our case nnz in a row) is larger than the architecture MAXVL (MAXimum Vector Length) it has to allocate a full register where to serialize intermediate reductions and perform the final reduction at full vector length.

We also see vector lengths of 256 and 512 elements clearly pointing to mixed use of 64-bit and 32-bit values.

In our case, the algorithmic vector length is less than MAXVL and thus operating at a longer vector length is a certain overkill. We certainly see that the reduction instruction itself takes an important percentage of the execution time.

Additionally, a `vwcvtu`[2] instruction takes a huge percentage of execution time. This instruction, as well as others (vid, merge, vmsltu, ...), are individually less expensive but still they represent a huge aggregated percentage of elapsed time.

With this codesign insight, we are pushing the compiler team to modify the generic code generation mechanism to avoid additional instructions and extended vector lengths.

About memory vector instructions we observe that vector loads execute in parallel with some of the instructions added by the compiler for the reduction support. This explains the reduced memory latency sensitivity observed in figure 22. Curiously, some of the data accesses by these load instructions are hits while others are misses. This is explained by the vector length, where one load instruction requests 15 DP elements while a cache line has 8. Being a small vector length and not multiple of the cache line results in those effects. For a longer vector length we would expect larger proportions of the accesses to be misses and these are the ones most people would expect to be the memory bandwidth bottleneck for the SpMV code.

The gather instruction (see Figure 28: vlxe) does not overlap with other arithmetic instructions. Its time at the top of the Re-Order Buffer (ROB) is larger than the time taken by loads even if accesses are hits. The limited injection capability[3] of the gather implementation is the fundamental limitation.

A final observation analyzing the traces is that the code uses 10 out of the 32 architected

[2]One of the RISC-V widening add instructions.
[3]Only one element per cycle can be at best injected into the register bank

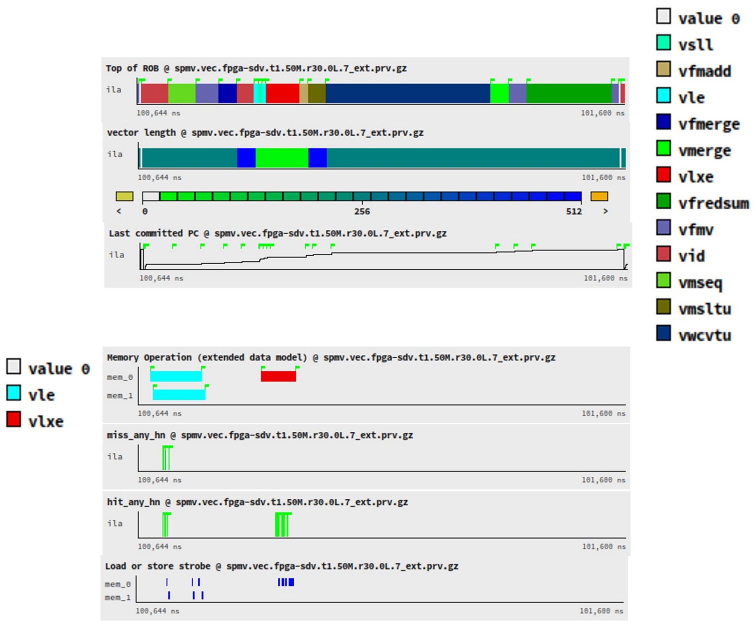


Figure 28: Several timelines for one iteration of the innermost loop of the vector execution. From top to bottom: instruction at the top of the ROB, vector length, program counter, memory operation, L2 misses, L2 hits and data being delivered to the vector unit.

vector registers. This value seems a bit high given the fundamental objects referenced at the algorithmic level and is probably originated by the code generation scheme for reductions and mixed data types. Even so, this provides useful codesign input for the compiler as it indicates that unrolling could be used to increase the utilization of the register state and build larger basic blocks that would allow for better instruction schedules and thus improve the overlap between arithmetic and memory instructions.

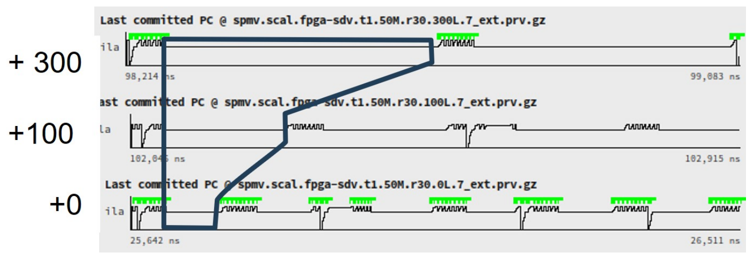


Figure 29: PC waveforms for 0, 100 and 300 additional cycles at the memory controller, showing how that translates into a stall in the case of cache misses.

To better observe the impact of memory latency, we obtained the corresponding traces for the case of 0, 100 and 300 additional latency in the memory subsystem. The results are shown in Fig 29 for the scalar case and Figure 30 for the vector case. We observe that the scalar run is highly sensitive to the additional latency, paying its full cost on every miss. In the vector runs, the additional latency is only paid when it goes beyond the duration of the overlapped non-memory instructions.

The above observations and comments have been presented to the compiler team of the EPI project. The information proved useful for them and some of the topics have been included in

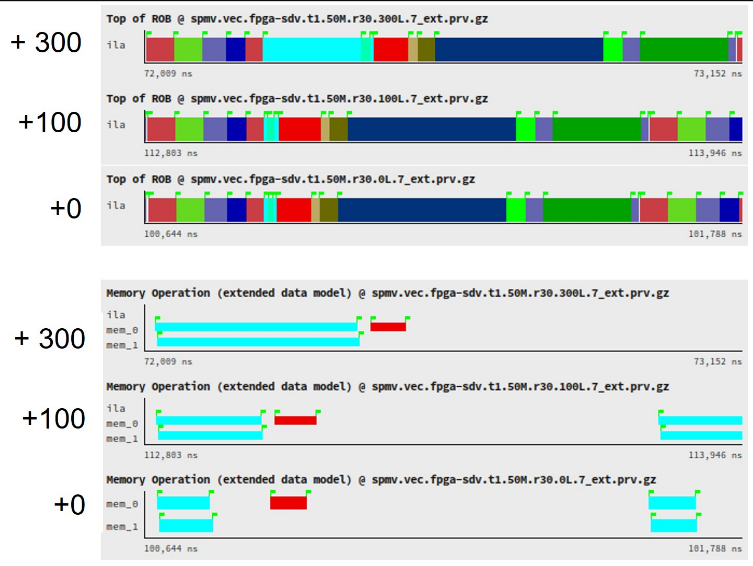


Figure 30: Top of the ROB and memory operations execution timelines for 0, 100 and 300 additional cycles at the memory controller. Only for the 300 cycles case, the additional latency translates into an elongation of the vector load instructions when at the top of the reorder buffer.

their ongoing compiler extensions road map[4]; while other suggestions will be taken over for the RVV1.0 target, but no backport will be done for the RVV0.7 ISA.

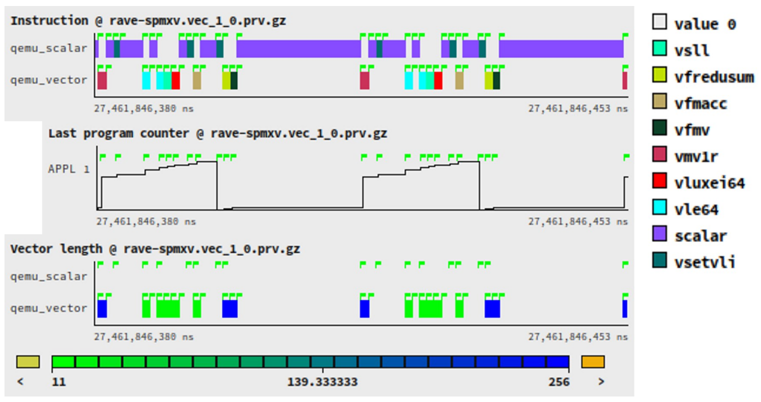


Figure 31: Two iterations of instructions in the innermost vector loop. Timelines from top to bottom show the actual vector instruction or indication of a scalar instruction, the last vector program counter and the vector length.

To test how much the RVV1.0 eases code generation and how much of the discussed potential improvements are already available in the RVV1.0 compiler we recently performed an additional study based on the use of the RAVE 1.0 software emulation platform. Figure 31 shows a trace of the instructions sequence. It is apparent that the number of vector instructions per iteration has drastically been reduced. We now clearly identify the two vector loads, the gather, FMA and reduction, plus a `vmv1r` and a `vfmv` instruction. Also interesting is that these last three instructions execute with a vector length of 256, while the memory and FMA instructions execute with a vector length of 15. The number of vector registers used is five, leaving

[4]Resource and scheduling constraints

opportunities for a large unrolling factor. These observations will now be discussed with the compiler team.

Unfortunately, the FPGA-SDV implementing RVV 1.0 is not yet available to actually measure timing and waveform traces detailing the cost of the individual instructions. We hope this to be soon available and will include further analyses in later deliverables.

A final very useful observation in this trace is that the long burst of scalar iterations between two vector sequences of instructions is of 17 instructions. This is the look-ahead that the scalar core should be able to tolerate in order to issue the vector instructions for the next iteration while the previous vector instructions are still in operation. This is very useful codesign insight for micro-architecture designers.

One of the observations we have mentioned several times is that the vector length in the parallelization approach followed is very low (only 15) which even if fixing s ome o f t he code generation issues detected would not efficiently use a microarchitecture designed for significantly longer vectors (256). From our point of view, this is codesign insight in the direction of adapting the algorithm to expose longer vectors. This can be done with other matrix formats. In a previous work, we explored the potential of the shell-c-sigma format supporting longer vector lengths. The basic idea of the approach was to operate on a block of rows at a time (typically matching the MAXVL of the architecture) and serializing the reduction. It was also written with intrinsics, avoiding the additional instructions included by the compiler. The result indicated a very good potential of the EPAC architecture including compiler techniques to be used such as unrolling and the potential to improve the micro-architecture in terms of performance of the gather instructions.

5.5.4 Conclusions

From the experience of the SpMV study at the RISC-V platforms we extract some overall insights and suggestions. At the ISA and micro-architecture level, we consider that evolving the current implementation should include:

- Adopt the RVV1.0 standard as a way to improve the support for loops with mixed data types.
- Rework the gather implementation to increase the effective injection rate of values into the register bank, in particular for the frequent case of sufficient spatial and temporal locality in the access pattern.

In terms of platforms, it would be very nice if we could:

- Extend the waveform level acquisition capabilities to other platforms. It is clear that this is impossible for the internals of chips procured from external vendors. It is very difficult in platforms not designed and produced by design teams to which our codesign analysts have access. Nevertheless, in our case, we could include signals at external interfaces (eg. memory requests). In the particular case of the EPAC1.5 test chip this is certainly possible.

At the compiler level we recommend to:

- Elaborate alternative code generation schemes for reductions, trying to reduce the number of additional instructions used in the single generic reduction scheme generated by the compiler as of today. In particular, the detection of problem vector lengths below the architectural vector length can be used to avoid a large number of instructions.

- Elaborate the instruction scheduler algorithms to advance the loads and computations that produce the indices for gather instructions in order to overlap other possible operations or loads with the latency of such operations.

- Elaborate the possibility to more aggressively apply loop unrolling to improve the register bank utilization and increase the potential overlap among memory operations and computation.

At the applications and libraries we consider that it would be good:

- To promote the use of other data layouts beyond the CSR. This could be either as native layout used by the applications or by carrying out temporary data conversion operations previous to repeated invocations of the basic operations. This situation with invariant matrix structure and values is typical for the SpMV kernel within iterative solvers and would amortize the conversion cost over a large number of invocations of the operation.

- Promote the use of numbering algorithms that directly support the implementation of the previous suggestion as native layout of the application.

- Do not rely on the existence of a vector reduction instruction in the ISA if possible. In general promote loop interchange approaches such that the innermost dimension is vectorizable with long lengths while serializing the reduction operation on such long vectors.

In terms of infrastructure, tools and methodology we suggest:

- To implement other variants of the SpMV algorithm based on the multiple rows approach but implemented in C++ to check how the compiler generates code for it.

- Use other more realistic matrices, with variable numbers of non-zeros per row and representing engineering problems.

- Use native compilers from other infrastructures in the holistic evaluation at the different levels on the FPGA-SDV platform

- Developing a more precise mechanism to synchronize the Extrae and waveform acquisition mechanism would be extremely useful to perform precise multi-scale analyses in codes with more phases and variability between them.

5.6 SPMXV on RHEA

5.6.1 Experimental Conditions

The matrix (resp. vector) size used in our experimentation campaign was around 58 MB (resp. 4 MB): with such sizes, neither matrix nor vector could be entirely stored in the L1 or L2 cache, however, the vector will fit into the L3 (allowing temporal locality) while the matrix will exceed the L3 cache sizes for most of our systems under test except SPR and Grace, which had large enough L3 caches to hold both the whole matrix and vector. With respect to cache behaviour, every matrix element is accessed with stride 1 (perfect spatial locality) but only used once (no temporal locality), cache size impact will be limited.

5.6.2 Timing Tests: Single Core/Multicore

Figure 32 presents the performance numbers obtained on a single core for our various systems and compilers under test. These results should be considered carefully because first they correspond to unrealistic use of the system and second they introduce a strong bias in the performance analysis: a single core is using the full system memory hierarchy. However, such tests allow us to detect differences in compiler behaviour and also in the quality of the generated code. All in all, both Neoverse V2 (Grace and G4) achieve similar performance for both GCC and ACFL (Arm Compiler For LINUX), with GCC benefitting from the `-OFast` option. Neoverse V1 is slightly behind but faster than Sapphire Rapid. The older systems (Skylake and Neoverse N1) are lagging behind. On the compiler front, GCC with the `-Ofast` option provides systematically (except on SPR) a performance gain over GCC with `-O3`. On all systems, GCC with `-Ofast` performance matches the native compilers: ICX on Intel and ACFL on Ampere.

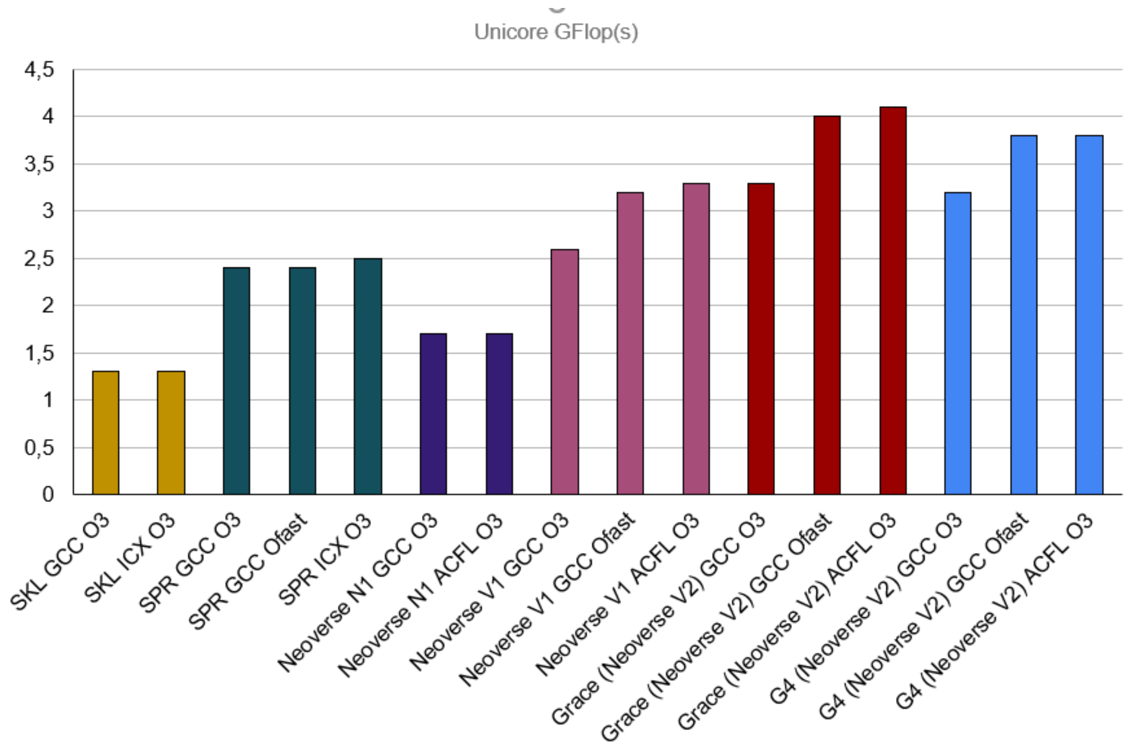


Figure 32: Performance in GLOPS unicore configuration. Higher is better.

In multicore measurements (see Figure 33), Grace and Sapphire Rapid systems have two clear advantages: first, the larger number of cores and second, their scalable memory system, which has good scalability properties. Sapphire Rapid reveals some interesting multicore characteristics. First, hyperthreading provides a performance boost, hinting that memory latency is one of the major performance bottlenecks of SMPXV. Second, GCC achieves significantly lower performance than ICPX while on unicore both compilers achieved similar performance: more detailed performance analysis indicated that the OpenMP GCC Library (GOMP) was adding a large overhead. The three older systems (SKL, Ampere and G3) are clearly lagging behind with respect to performance, mainly due to their lower core counts. Interestingly, there is a major difference in compiler behavior between GRACE and G4: on GRACE, GCC performance (both `-O3` and `-Ofast`) is lower than ACFL while on G4 all compilers achieve similar performance (similar situation as the unicore case).

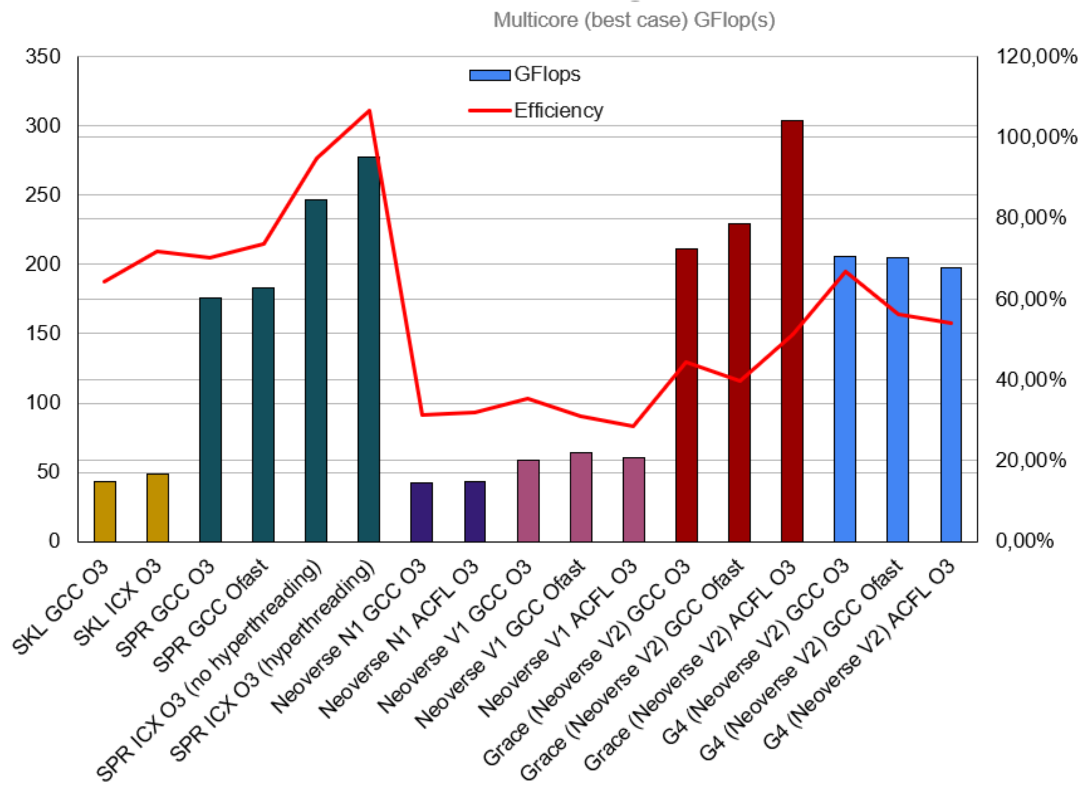


Figure 33: GFLOPS (histogram) and efficiency (red curve) for SPMXV Multicore runs. Higher is better.

The efficiency curve shown in Figure 33 confirms the very good SPR scalability : efficiency is above 90 percent with a remarkable peak at 107 percent with hyperthreading enabled. G4 has better efficiency (60 percent) than Grace (40 percent) essentially due to a smaller number of cores. Skylake exhibited good scalability with efficiencies of around 60 percent while G3 and Ampere exhibited poor efficiency (around 30 percent).

Refining the analysis by normalizing by the number of cores and frequency (see Figure 34) sets back GRACE at a lower level than G4, which ends up leading the pack. Finally, the most interesting part is again Sapphire Rapids, which turns out to be the fastest when using normalization. Furthermore, parallelism used at the outermost loop level did not introduce major overhead except for GCC on Sapphire Rapid.

5.6.3 Compiler Analysis

Although the SPMXV loop seems to be extremely simple, it offers a unique combination of challenges for the compilers. The outermost loop (on i) is fully parallel and has a very large iteration count (number of elements in array x). The innermost loop on nz is much more complex to optimize: first, the loop iteration count is variable and, in general, small (less than 15 in our test case), second the loop corresponds to reduction (scalar dot product), and finally the access to array x is indirect. The only simple characteristic is access to the matrix values which has perfect spatial locality (stride 1 access), but all data is exactly used once.

Compilers have a large panel of code generation possibilities which in fact, have been used by various compilers throughout our systematic testing (see Figure 11). First, compilers could stay with a scalar loop which could be further optimized by unrolling two or four times. How-

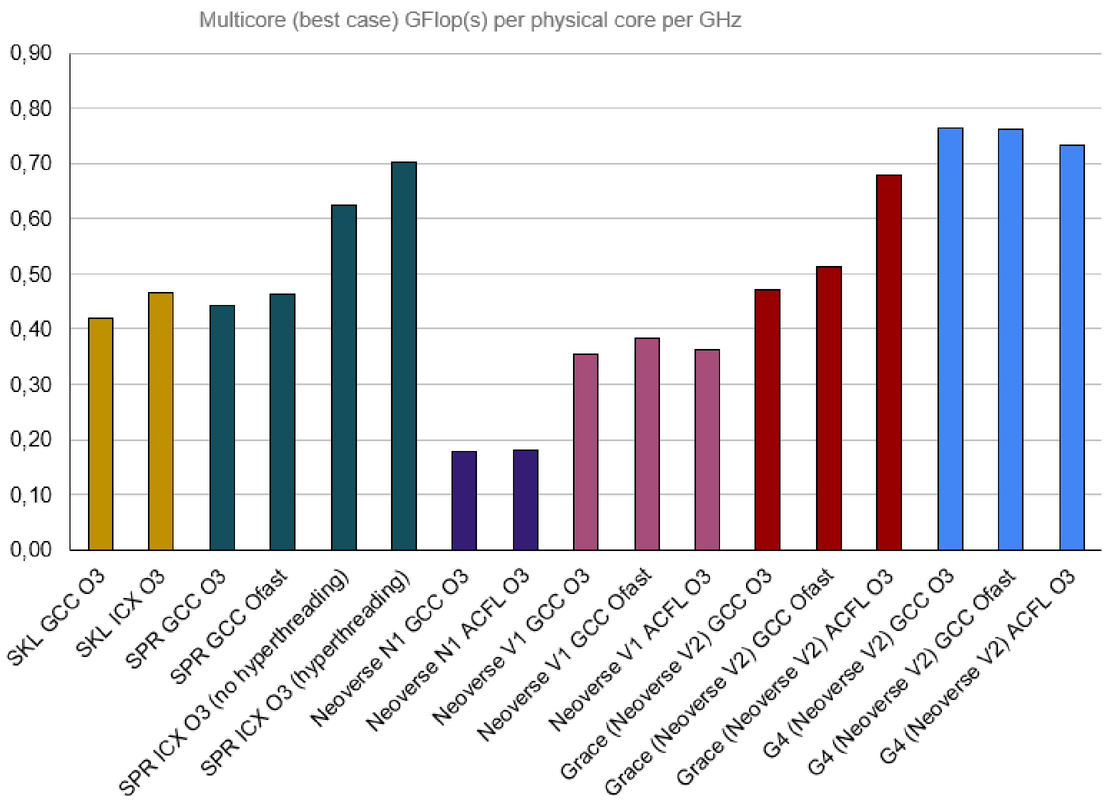


Figure 34: Multicore run GFLOPs normalized (divided) by number of cores and frequency. Higher is better

ever, Unrolling use will require then a final reduction step. Second, compilers could vectorize the loop, requiring the use of scatter/gather instructions (which are costly) then dealing with peel/tail and finally performing the final reduction stage. Third, compilers could perform partial vectorization, only vectorizing loads and/or the FP multiplies. Finally, the peel/tail loop could be suppressed by using masked instructions available in SVE and AVX 512. For that simple loop nest, compilers have a large amount of freedom degrees which should be carefully evaluated and selected by appropriate cost models.

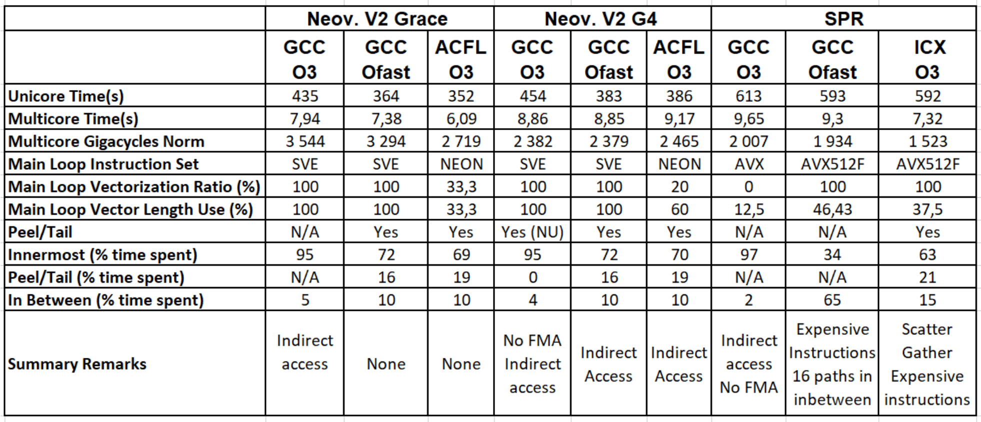
All in all, it is interesting to see that the use of shorter vectors (NEON) and partial vectorization finally gave the best performance on both GRACE and G4.

5.6.4 Conclusion and Future Work

MAQAO analysis results for all of the runs made on the various systems (hardware and software are available at https://datafront.exascale-computing.eu/public/spmxv/.

In the previous subsections, we presented a detailed performance analysis of the SPMXV kernel on different systems (ARM and X 86-64) using different compilers (ICPX, ACFL and GCC) and different compiler options. All of these results have been shared with SiPearl compiler team which tested their own in-house compiler.

The main takeaway of this analysis is the difficulty of generating efficient code for reduction patterns in particular for short ones. Choosing the right vector length (not necessarily the largest one) should be carefully evaluated by compilers in their cost models.



	Neov. V2 Grace			Neov. V2 G4			SPR		
	GCC O3	GCC Ofast	ACFL O3	GCC O3	GCC Ofast	ACFL O3	GCC O3	GCC Ofast	ICX O3
Unicore Time(s)	435	364	352	454	383	386	613	593	592
Multicore Time(s)	7,94	7,38	6,09	8,86	8,85	9,17	9,65	9,3	7,32
Multicore Gigacycles Norm	3 544	3 294	2 719	2 382	2 379	2 465	2 007	1 934	1 523
Main Loop Instruction Set	SVE	SVE	NEON	SVE	SVE	NEON	AVX	AVX512F	AVX512F
Main Loop Vectorization Ratio (%)	100	100	33,3	100	100	20	0	100	100
Main Loop Vector Length Use (%)	100	100	33,3	100	100	60	12,5	46,43	37,5
Peel/Tail	N/A	Yes	Yes	Yes (NU)	Yes	Yes	N/A	N/A	Yes
Innermost (% time spent)	95	72	69	95	72	70	97	34	63
Peel/Tail (% time spent)	N/A	16	19	0	16	19	N/A	N/A	21
In Between (% time spent)	5	10	10	4	10	10	2	65	15
Summary Remarks	Indirect access	None	None	No FMA Indirect access	Indirect Access	Indirect Access	Indirect access No FMA	Expensive Instructions 16 paths in inbetween	Scatter Gather Expensive instructions

Figure 35: Analysis of code generated by different compilers for SPMXV

Additional work will be carried out to define an efficient code generation strategy for reductions in function of the reduction size. Second, the cost of the GOMP (GCC Open MP) library has to be further explored and understood. Third, the parallelization strategy used (using fixed chunk size) although satisfactory in most of our experiments should be further analyzed by using different p arallelization s trategies. L ast b ut n ot l east, a f ull a lgorithm r ewrite relying on a major data format change (column storage) should be investigated. This data structure change will suppress the reduction but will require zero padding. This zero padding will incur an acceptable cost if the number of non-zero elements deviates too much from the average. Otherwise, padding will generate too many useless operations.

UVSQ used SPMXV as a test kernel in an engineering student project: students had to perform a ”normalized” (i.e. following strict experimental protocol) performance analysis of running SPMXV kernel on their own laptop. The resulting studies (around 40) gave us some additional insight of SPMXV on a large system panel.

5.7 LBC General performance assessment

The LBC code was mainly tested on the following two reference platforms:

1. A Sapphire Rapid with 2 sockets and 56 cores per socket, using `Gfortran 13.3.0` with `-march=native` and `-Ofast` flags

2. A GRACE processor with one socket and 72 cores using `Gfortran 11.4` with `-mcpu=native` and `-Ofast` flags

Figure 36 shows LBC behavior with weak scaling on a GRACE system.

Let’s consider a roofline performance estimate using the number of *Particle Distribution Function (PDF)* loads/stores and flops encountered in every lattice update. Since ARM Neoverse offers a write streaming mode we will assume that a write-allocate does not occur during the store phase of the loop. This means that for every lattice point update we require 19 PDF loads, 19 PDF stores, and 18 index loads of the velocity vectors (which show the relative offset of neighbouring lattice points). Assuming double precision floats (8 bytes) for the PDFs and (4 byte) integers for the indices, the memory traffic required is then $2 \times 19 \times 8 + 18 \times 4 = 376$Bytes. The kernel consists of roughly 240Flops. A simple roofline model prediction based on bandwidth numbers achieved from benchmarking a STREAM triad kernel then gives us a Performance of

324GBytes/s × 240Flops/376Bytes = 206.80Gflops/s or 861.70MLUP/s (Million Lattice Updates Per Second). Since this metric considers the number of lattice points updated per second, it aligns with the computational objective of Lattice Boltzmann Methods and is a better indicator of performance.

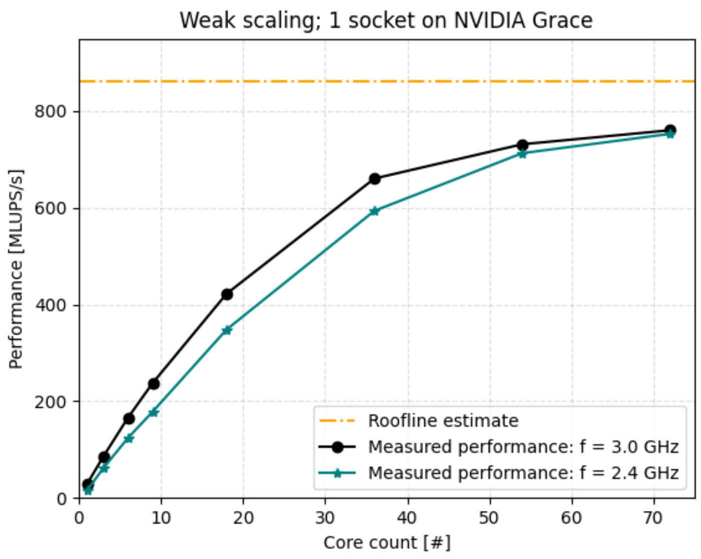
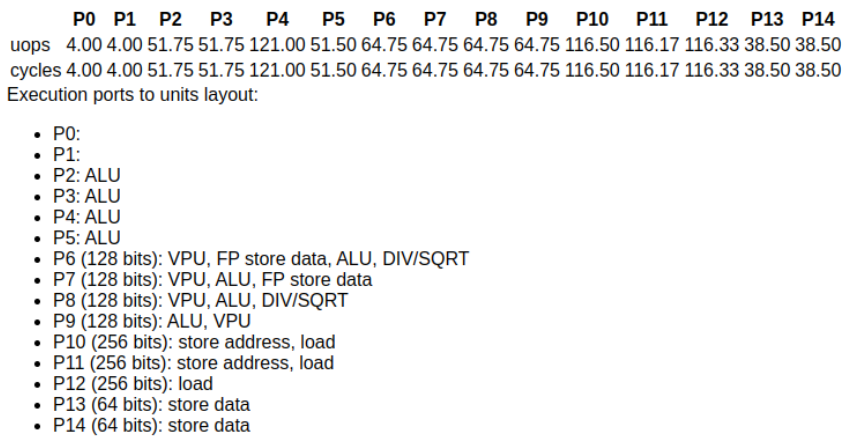


Figure 36: LBC Weak scaling performance on Grace system: the domain size and core count were changed in the inner-most (z) dimension: starting with a problem size of 256x256x64 elements for 1 core, until 256x256x4608 for 72 cores.

For single-core performance, let us consider the MAQAO CPU port model (see Figure 37). The ARM Neoverse documentation specifies the execution units present on the chip, from which this port model is constructed. Using assembly code disassembled from the application binary, the Code Quality Analysis module is able to assign micro-ops and latency cycle values to each of the execution units. Assuming perfect Out-of-Order execution, the static analysis calculates a critical path with a length of 121 cycles. The analysis here claims that the integer ALU port $P4$ presents the main bottleneck. At least for a single core, this means that the kernel is not primarily hindered by data load/stores but address calculation stemming from the streaming step of the Lattice Boltzmann kernel. Since we now know how many ALU cycles we need for a single lattice update, inserting the clock frequency of the core should give us a performance indicator. This would give us estimates of 3.0 GHz / 121 cy/LUP = 24.8 MLUP/s and 2.4 GHz / 121 cy/LUP = 19.8 MLUP/s for operating frequencies of 3.0 GHz and 2.4 GHz respectively. The single-core performance measurements [36] showed values of 28.6 MLUP/s and 23.2 MLUPs which seem to be in the ballpark, and surprisingly a little bit higher.



	P0	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14
uops	4.00	4.00	51.75	51.75	121.00	51.50	64.75	64.75	64.75	64.75	116.50	116.17	116.33	38.50	38.50
cycles	4.00	4.00	51.75	51.75	121.00	51.50	64.75	64.75	64.75	64.75	116.50	116.17	116.33	38.50	38.50

Execution ports to units layout:

- P0:
- P1:
- P2: ALU
- P3: ALU
- P4: ALU
- P5: ALU
- P6 (128 bits): VPU, FP store data, ALU, DIV/SQRT
- P7 (128 bits): VPU, ALU, FP store data
- P8 (128 bits): VPU, ALU, DIV/SQRT
- P9 (128 bits): ALU, VPU
- P10 (256 bits): store address, load
- P11 (256 bits): store address, load
- P12 (256 bits): load
- P13 (64 bits): store data
- P14 (64 bits): store data

Figure 37: Static CPU analysis of the LBC kernel on an NVIDIA GRACE core. The execution ports are listed, with the number of cycles and micro-ops they are assigned calculated for each of them.

5.8 LBC on RHEA

5.8.1 Experimental Conditions

In our experiments we used a large grid size 512 x 512 x 768, choosing subdomain size in such a way that the workload (number of grid points) is equally distributed across the cores. The grid size is so large that it does not fit in any L3 cache considered.

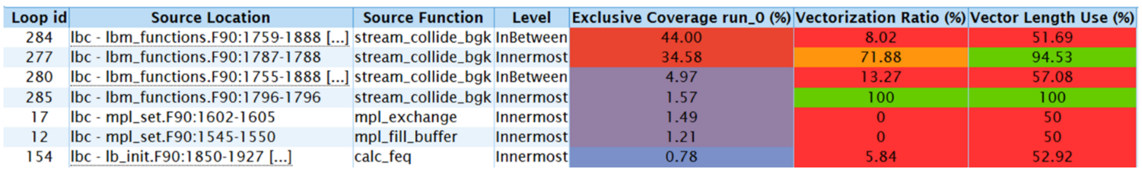
5.8.2 Compiler Issues

The code structure of the LBC kernel is shown in Listing 1. The overall code structure is extremely challenging for compilers because, beyond the classical three-loop nest, there are two additional loops located in the third loop level. These additional loops make the main computations hard to vectorize because they are no longer in the innermost loop level, which is the standard target for compiler vectorization. Most compilers concentrate their effort on vectorizing these loops. Unfortunately, their low iteration count (at most 19) prevents any significant gain from v ectorization. Worse, t he i teration c ount o f 1 9 a lso p revents t he compiler from fully unrolling the loop. Last but not least the first loop (denoted COPY IN) has a variable loop count.

Listing 1: Kernel structure of the LBC code

```
DO k=1,lb_domlx(3) !! Domain size along 3rd dimension
  DO j=1,lb_domlx(2)  !! Domain size along 2nd dimension
    DO i=1,lb_domlx(1)  !!  Domain size along 1st  dimension
      DO r=i,19  !! 19 iterations at most with variable iteration start
      ftmp(r)=fIn(NDX(r,i-cx(r,1),j-cx(r,2),k-cx(r,3)))  !! COPY IN
      ENDDO
      ! ...
      ! Regular computations
      ! ...
      DO r=1,19  !! Exactly 19 iterations
        fOut(NDX(r,i,j,k)) = ftmp(r) - omega*(ftmp(r) - fEq(r)) !! COPY OUT
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

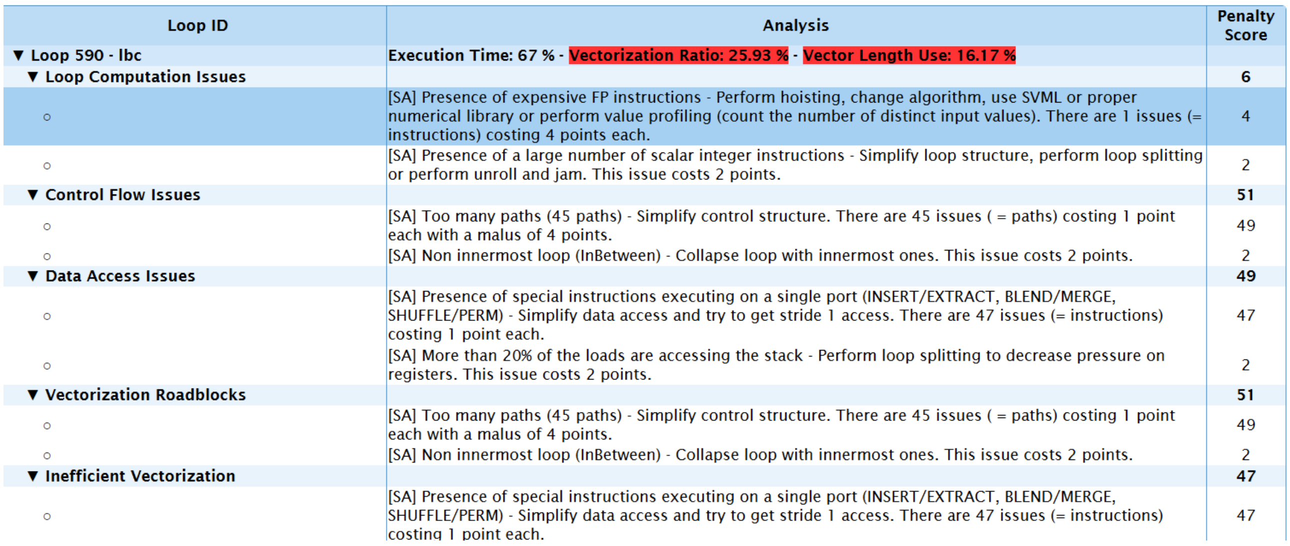
Code compiled with Gfortran with the `-O3` optimization level on a Graviton 4 clearly shows the issue of non-vectorization of the main computation (see Figure 38).



Loop id	Source Location	Source Function	Level	Exclusive Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
284	lbc - lbm_functions.F90:1759-1888 [...]	stream_collide_bgk	InBetween	44.00	8.02	51.69
277	lbc - lbm_functions.F90:1787-1788	stream_collide_bgk	Innermost	34.58	71.88	94.53
280	lbc - lbm_functions.F90:1755-1888 [...]	stream_collide_bgk	InBetween	4.97	13.27	57.08
285	lbc - lbm_functions.F90:1796-1796	stream_collide_bgk	Innermost	1.57	100	100
17	lbc - mpl_set.F90:1602-1605	mpl_exchange	Innermost	1.49	0	50
12	lbc - mpl_set.F90:1545-1550	mpl_fill_buffer	Innermost	1.21	0	50
154	lbc - lb_init.F90:1850-1927 [...]	calc_feq	Innermost	0.78	5.84	52.92

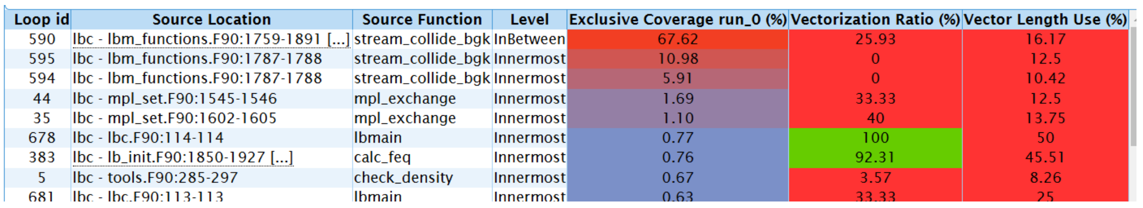
Figure 38: MAQAO analysis of LBC loops execution time and vectorization running on a 96-core Graviton, using Gfortran with -O3 as compiler. The hottest loop (44 percent exclusive coverage) is "in-between" and not vectorized at all. The two "innermost loops" achieve decent vectorization levels

Now, the more aggressive compiler (Ifort on X86-64) does not perform much better as can be seen in Figures 39 and 40.



Loop ID	Analysis	Penalty Score
▼ Loop 590 - lbc	Execution Time: 67 % - Vectorization Ratio: 25.93 % - Vector Length Use: 16.17 %	
▼ Loop Computation Issues		6
○	[SA] Presence of expensive FP instructions - Perform hoisting, change algorithm, use SVML or proper numerical library or perform value profiling (count the number of distinct input values). There are 1 issues (= instructions) costing 4 points each.	4
○	[SA] Presence of a large number of scalar integer instructions - Simplify loop structure, perform loop splitting or perform unroll and jam. This issue costs 2 points.	2
▼ Control Flow Issues		51
○	[SA] Too many paths (45 paths) - Simplify control structure. There are 45 issues (= paths) costing 1 point each with a malus of 4 points.	49
○	[SA] Non innermost loop (InBetween) - Collapse loop with innermost ones. This issue costs 2 points.	2
▼ Data Access Issues		49
○	[SA] Presence of special instructions executing on a single port (INSERT/EXTRACT, BLEND/MERGE, SHUFFLE/PERM) - Simplify data access and try to get stride 1 access. There are 47 issues (= instructions) costing 1 point each.	47
○	[SA] More than 20% of the loads are accessing the stack - Perform loop splitting to decrease pressure on registers. This issue costs 2 points.	2
▼ Vectorization Roadblocks		51
○	[SA] Too many paths (45 paths) - Simplify control structure. There are 45 issues (= paths) costing 1 point each with a malus of 4 points.	49
○	[SA] Non innermost loop (InBetween) - Collapse loop with innermost ones. This issue costs 2 points.	2
▼ Inefficient Vectorization		47
○	[SA] Presence of special instructions executing on a single port (INSERT/EXTRACT, BLEND/MERGE, SHUFFLE/PERM) - Simplify data access and try to get stride 1 access. There are 47 issues (= instructions) costing 1 point each.	47

Figure 39: MAQAO Analysis of compiler issues for the hottest loop (67 percent execution time) running on a 52 cores Skylake using IFort with `-O3`. The compiler succeeded in merging the loops and performing partial vectorization (25,93 % vectorization ratio) but at the cost of a very complex control flow.



Loop id	Source Location	Source Function	Level	Exclusive Coverage run_0 (%)	Vectorization Ratio (%)	Vector Length Use (%)
590	lbc - lbm_functions.F90:1759-1891 [...]	stream_collide_bgk	InBetween	67.62	25.93	16.17
595	lbc - lbm_functions.F90:1787-1788	stream_collide_bgk	Innermost	10.98	0	12.5
594	lbc - lbm_functions.F90:1787-1788	stream_collide_bgk	Innermost	5.91	0	10.42
44	lbc - mpl_set.F90:1545-1546	mpl_exchange	Innermost	1.69	33.33	12.5
35	lbc - mpl_set.F90:1602-1605	mpl_exchange	Innermost	1.10	40	13.75
678	lbc - lbc.F90:114-114	lbmain	Innermost	0.77	100	50
383	lbc - lb_init.F90:1850-1927 [...]	calc_feq	Innermost	0.76	92.31	45.51
5	lbc - tools.F90:285-297	check_density	Innermost	0.67	3.57	8.26
681	lbc - lbc.F90:113-113	lbmain	Innermost	0.63	33.33	25

Figure 40: MAQAO analysis of vectorization ratio for the hottest loops running on a 52 cores Skylake using `IFort -O3`. The compiler succeeded in merging the loops and performing partial vectorization (25,93 percent vectorization ratio) but at the cost of a very complex control flow.

5.8.3 Multicore runs

Due to the poor code quality generated by the different c ompilers w e h ave t ested s o f ar, we limited our experiments to 4 systems: Skylake, Ampere, Graviton G3 and Graviton G4.

A first a nalysis w as p erformed t o m easure t he i mpact o f M PI a nd l oad d istribution: i n all of our experiments, parallelism overhead remained under 10 percent showing a limited cost of MPI communication primitives and a good load distribution.

Figure 41 presents the total execution timing running on full systems. First, compiler choices and/or options have a very limited impact on execution time. Second, G4 is the overall winner but essentially due to a larger core count in particular when compared with G3 or Skylake. Figure 42 which normalizes performance with respect to core count sets back G3, G4 and Skylake on the same level. Overall, the normalized timings show clearly the low performance of NEOVERSE N1.

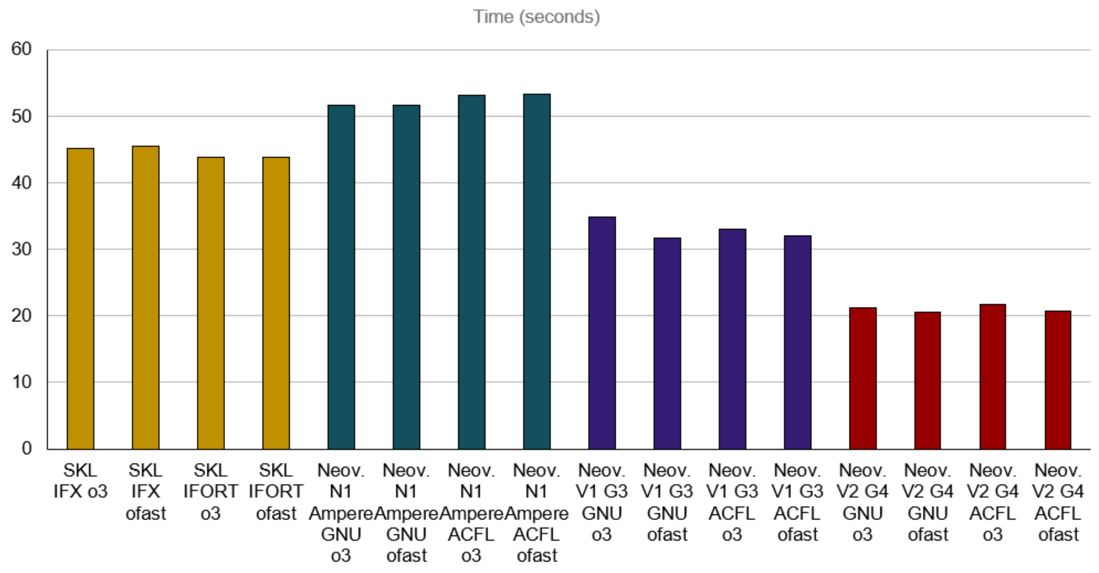


Figure 41: LBC full system (using all available cores) runs on different systems and different compiler options. Lower is better.

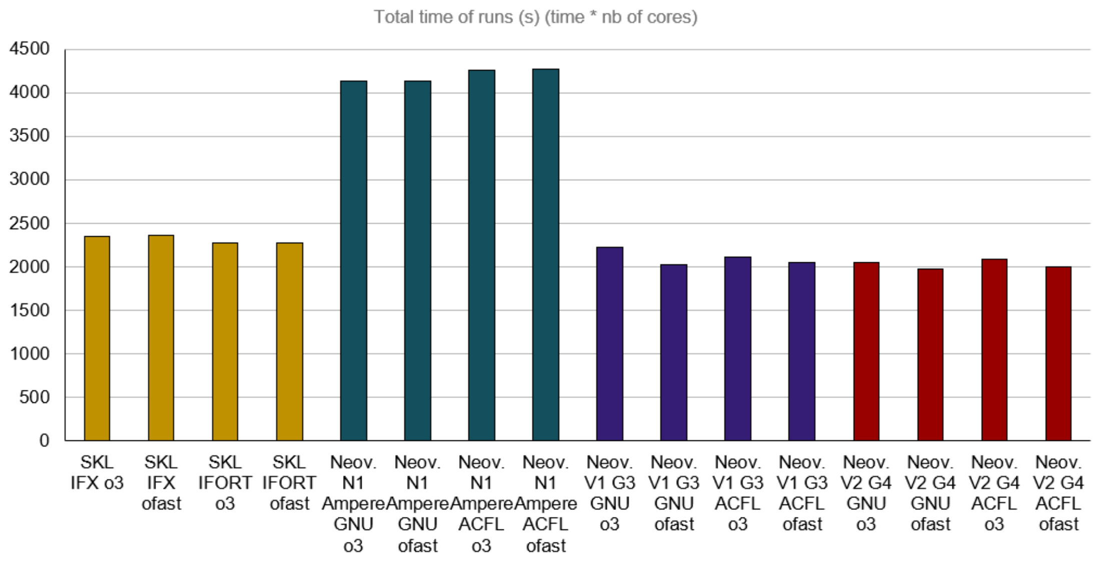


Figure 42: LBC full system (using all available cores) runs on different systems and different compiler options: scaled timings i.e. multicore execution times are multiplied by number of cores used. Lower is better.

5.8.4 Conclusions and Future work

MAQAO analysis for all of the runs made on the various systems (hardware and software) are available at https://datafront.exascale-computing.eu/public/LBC/.

In the previous subsections, we presented a performance analysis of the LBC kernel on different s ystems (ARM a nd X 86-64) u sing d ifferent co mpilers (I FC, AC FL an d GC C) and different c ompiler o ptions. A ll o f t hese r esults h ave b een s hared w ith S iPearl c ompiler team which tested their own in-house compiler.

The main takeaway of this analysis is the inability of current compilers to deal efficiently/vectorize with complex loop nests.

Additional work will be carried out to help the compiler by using directives for enabling loop splitting and/or forcing vectorization. That effort will be pursued by rewriting source code to facilitate compiler optimizations.

6 Conclusions and Future work

This document reported the activities and results accomplished in terms of tasks T4.1 and T4.2 during the first twelve months of the project.

First, the four kernels developed in both tasks are presented, and their main characteristics are listed. Second, a few lessons taken from the audit effort are detailed: four case studies related to patterns and best practices, two notes related to programming models and finally one summary of our collaboration with other CoE.

Third, our methodology for cooperating with EPI projects (RISC V and Rhea) is detailed. This covers the experimental methodology developed to generate results and analyze them as well the various hardware platforms and software tools (simulators, compilers, ..).

Fourth, our kernel evaluation results are presented. The two kernels selected (Sparse Matrix-Vector multiply and Lattice Boltzmann, computations) selected, turned out to be very challenging both from a hardware point of view and also compiler point of view. In particular, both kernels clearly revealed many deficiencies of current compiler technology and uniformly across several major compiler providers (ARM, INTEL and GCC/GFortran). Our findings have been shared with both RISC V and RHEA teams. In particular, with RHEA, our kernels have been tested on their in-house compiler. In both cases, our detailed performance analysis led to very fruitful interactions and will drive further work to improve the performance of these two kernels.

Clearly, there will be some additional carried out on the first two kernels selected: in particular, LBC will be evaluated/analysed with respect to RISC V

For the coming periods, we already identified two kernels (K-Means (provided by RWTH) and FFTW kernels (UVSQ/Sipearl).

With the growing number of published technical pages, especially patterns and best practices, we identified a need for an advanced listing of the pages. To fulfil the goal of improving the navigability of the codesign website, we plan to address the issue by designing and developing a hierarchical structure of the technical pages. The preliminary idea is to group the pages by a common affected performance metric, a common programming model or another keyword, or to have a hierarchy of generalized and specialized pages, e.g., describing a phenomenon under specific conditions.

On top of the goals defined in the DoA, we brought the idea of creating a knowledge base of issues (FAQs, caveats, troubleshooting) related to the usage of POP tools and methodology during assessments and second-level services. Those might be gathered, e.g., during particular technical meetings, or independently during an analyst work. The reason for such a database would be to enable a collaborative solving of issues, sharing the solution among the tools users, and as a codesign input for the tools developers. The selected issues might be published on the codesign website or through an alternative channel. The whole concept and its technical aspects will be thoroughly discussed during the following technical meetings.

Table 2 shows the defined KPIs for this activity including our achieved current state as well as the targets for the next milestone until M24 of the project. As can be seen, we are on track and even achieved a bit more than planned already.

KPI	M12 Goal	M12 Reached	M24 Goal
Kernels created	3	4	6
Technical pages created	7	7	15
Kernels evaluated	2	3	5

Table 2: KPIs for the current MS5 and the next MS11 Codesign milestones.

From the point of view of the RISC-V EPI codesign, the outcome has been presented to the LLVM compiler team. Their feedback so far was very positive and our results are very useful for them in terms of exploring the different approaches included in the study[5].

[5]Conclusions for the compiler team: 1) exploring the scheme for reductions, 2) instruction scheduler, and 3) loop unrolling.

Acronyms and Abbreviations

- BSC: Barcelona Supercomputing Center
- CA: Consortium Agreement
- CAdv: Customer Advocate
- DoA: Description of Action (Annex 1 of the Grant Agreement)
- EC: European Commission
- FZJ: Forschungszentrum Jülich GmbH
- D: deliverable
- GA: General Assembly / Grant Agreement
- HLRS: High Performance Computing Centre (University of Stuttgart)
- HPC: High Performance Computing
- IPR: Intellectual Property Right
- INESC-ID: Instituto de Ennenharia de Sistemas e Computadores, Investigacao e Desenvolvimento em Lisboa
- IT4I: Technical University of Ostrava
- KPI: Key Performance Indicator
- M: Month
- MS: Milestones
- PEB: Project Executive Board
- PM: Person month / Project manager
- POP: Performance Optimization and Productivity
- R: Risk
- RV: Review
- RWTH Aachen: Rheinisch-Westfaelische Technische Hochschule Aachen
- TERATEC: TERATEC
- USTUTT (HLRS): University of Stuttgart
- UVSQ: Universite de Versailles Saint-Quentin-en-Yvelines
- WP: Work Package
- WPL: Work Package Leader

List of Figures

1 SPMXV kernel 8
2 LBC Computational pattern 9
3 An example of a trace manifesting the poor GPU data transfer rate as a load imbalance issue. 10
4 An example of a trace manifesting the improved load balance applying the appropriate process mapping to GPUs. 11
5 An example of a trace displaying useful duration executing MPI+OpenMP dlb-pils (2 MPI processes x 2 OpenMP threads). 11
6 An example of a trace displaying useful duration executing MPI+OpenMP dlb-pils (2 MPI processes x 2 OpenMP threads) with DLB enabled. 13
7 The new Projects collection listing the EU projects with a connection to the published assessment reports. 15
8 The detail of an EU project item from the new Projects collection listing the related code assessment reports. 15
9 The table with the published assessment reports extended by the Project column linking the reports with their projects. 16
10 Hardware configurations tested 19
11 Compiler versions tested 19
12 Timeline showing the structure (parallel functions) of the OpenMP parallelization at Marenostrum 5 for the first version of JuKRR. 20
13 Timeline showing the structure (parallel functions) of the OpenMP parallelization at Marenostrum 5 for the refactored version of JuKRR. 20
14 Performance of Original JuKRR: scalability test on a 64 cores Graviton 3E . . . 21
15 Performance of Optimized JuKRR: scalability test on a 64-core Graviton 3E . . . 21
16 GFlops performance of SPMXV kernel 23
17 Fraction of peak performance for SPMXV kernel 24
18 Impact of Datasize on SPMXV performance 24
19 Normalized FLOPS per cycle for the different versions of the code (scalar/vector) and platform 25
20 Scalar code OpenMP speedup with number of cores for the three multicore platforms 26
21 Aggregated FLOPS per cycle when scaling the number of cores in the different platforms 26
22 Impact of additional memory latency on FLOPS per cycle for the scalar and vector version of the code in the FPGA-SDV 27
23 Timeline (full run and zoom on the execution of 30 SpMVs) for the scalar and vector version of the code at the same time scale. 27
24 Metrics for the Scalar SpMV code computed from the traces in the FPGA-SDV and EPAC1.5 platforms 28
25 Timeline for the vector version of the code on the FPGA-SDV and EPAC1.5 platforms at the same time scale. 28
26 Metrics for the Scalar SpMV code computed from the traces in the FPGA-SDV and EPAC1.5 platforms 29
27 Program counter and cache hits/misses timelines of one iteration of the row loop in the waveform for the scalar execution at the EPAC-SDV. 29

28 Several timelines for one iteration of the innermost loop of the vector execution. From top to bottom: instruction at the top of the ROB, vector length, program counter, memory operation, L2 misses, L2 hits and data being delivered to the vector unit. 30
29 PC waveforms for 0, 100 and 300 additional cycles at the memory controller, showing how that translates into a stall in the case of cache misses. 30
30 Top of the ROB and memory operations execution timelines for 0, 100 and 300 additional cycles at the memory controller. Only for the 300 cycles case, the additional latency translates into an elongation of the vector load instructions when at the top of the reorder buffer. 31
31 Two iterations of instructions in the innermost vector loop. Timelines from top to bottom show the actual vector instruction or indication of a scalar instruction, the last vector program counter and the vector length. 31
32 Performance in GLOPS unicore configuration. Higher is better. 34
33 GFLOPS (histogram) and efficiency (red curve) for SPMXV Multicore runs. Higher is better. 35
34 Multicore run GFLOPs normalized (divided) by number of cores and frequency. Higher is better 36
35 Analysis of code generated by different compilers for SPMXV 37
36 LBC Weak scaling performance on Grace system: the domain size and core count were changed in the inner-most (z) dimension: starting with a problem size of 256x256x64 elements for 1 core, until 256x256x4608 for 72 cores. 38
37 Static CPU analysis of the LBC kernel on an NVIDIA GRACE core. The execution ports are listed, with the number of cycles and micro-ops they are assigned calculated for each of them. 39
38 MAQAO analysis of LBC loops execution time and vectorization running on a 96-core Graviton, using Gfortran with -O3 as compiler. The hottest loop (44 percent exclusive coverage) is “in-between” and not vectorized at all. The two “innermost loops” achieve decent vectorization levels 40
39 MAQAO Analysis of compiler issues for the hottest loop (67 percent execution time) running on a 52 cores Skylake using IFort with `-O3`. The compiler succeeded in merging the loops and performing partial vectorization (25,93 % vectorization ratio) but at the cost of a very complex control flow. 41
40 MAQAO analysis of vectorization ratio for the hottest loops running on a 52 cores Skylake using `IFort -O3`. The compiler succeeded in merging the loops and performing partial vectorization (25,93 percent vectorization ratio) but at the cost of a very complex control flow. 41
41 LBC full system (using all available cores) runs on different systems and different compiler options. Lower is better. 42
42 LBC full system (using all available cores) runs on different systems and different compiler options: scaled timings i.e. multicore execution times are multiplied by number of cores used. Lower is better. 42

List of Tables

1 KPIs for the MS5 Codesign 1 milestone 5
2 KPIs for the current MS5 and the next MS11 Codesign milestones. 44