MEEP

MareNostrum Experimental Exascale Platform

D5.3 – APPLICATIONS PORTED (FULL SOFTWARE-STACK)

Version 1.0

Document Information

Contract Number	946002
Project Website	https://meep-project.eu
Contractual Deadline	31/12/2022
Dissemination Level	Public (PU)
Nature	Report (R)
Authors	Xavier Teruel (BSC), Roger Ferrer (BSC), Xavier Martorell (BSC), Jorge Ejarque (BSC), Pere Verges (BSC), Julian Morillo (BSC), Manuel Rodrigues (BSC), Aaron Call (BSC).
Contributors	Rosa M. Badia (BSC), Josep Ll. Berral (BSC).
Reviewers	John D. Davis (BSC), Eduard Ayguadé (BSC).

The MEEP project has received funding from the European High-Performance Computing Joint Undertaking under grant agreement No 946002. The JU receives support from the European Union s Horizon 2020 research and innovation programme and Spain, Croatia, Turkey.

Change Log

Version	Author	Description of Change
v0.1	Xavier Teruel	Initial draft structure.
v0.2	Manuel Rodrigues, Julian Morillo	Runtimes, and benchmark descriptions. Performance Methodology updates
v0.5	Various Authors	Including contents for: compilers, containers, libraries, and offload-mode. Document structure changes.
v0.7	Various Authors	Including contents for: introduction, compilers, operating system, benchmarks. Drafting executive summary. Table of releases.
v0.9	Various Authors	Completing Operating System, Executive summary, Multi-devices, and Conclusions. Minor edits.
v1.0	Various Authors	Applying internal review feedback

ACKNOWLEDGEMENTS

The MEEP project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 946002. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Croatia, Turkey.

The partners in the project are BARCELONA SUPERCOMPUTING CENTER - CENTRO NACIONAL DE SUPERCOMPUTACION (BSC), FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING, UNIVERSITY OF ZAGREB (UNIZG-FER), & THE SCIENTIFIC AND TECHNOLOGICAL RESEARCH COUNCIL OF TURKEY, INFORMATICS AND INFORMATION SECURITY RESEARCH CENTER (TÜBİTAK BILGEM).

The content of this document is the result of extensive discussions within the MEEP © Consortium as a whole.

Index

1. Executive Summary

This document presents all the current releases of the MEEP Software Stack. It includes all the levels involved in the toolchain as well as the target benchmarks we plan to evaluate at the end of the project. As described in previous deliverables, the ACME EA platform can be used following two different approaches: 1) A stand-alone processor/accelerator booting Linux (ie, self-hosted); 2) A supporting accelerator device attached to a host (ie, offloading). The document focuses on the self-hosted mode, while we also explore the opportunities of the offload mode. When no explicitly specified, the contents will refer to the self-hosted mode.

The Operating System, compiler, runtimes, and containerization support were already introduced in deliverable *D5.2 Linux with initial host interface release, based on the requirements document* [33]. The set of benchmarks was also introduced in deliverable *D5.1 Benchmark suite of HPC applications* [32].

The integration of all these components aim to allow ACME EA programmers to exploit all the system capabilities, being able to deploy their own use cases, and to obtain useful information to infer the performance behaviour. As described in the Description of Action (DoA): *"all the application that have been identifed are ported to run on top of the emulation platform"*; and for each application's entry we also describe the metric of interest and its evaluation methodology (from DoA: *"The final phase will focus on application performance evaluation and debugging"*).

Following sections present a brief summary of the status of these components at the current stage of the project.

1.1. Operating System

The Operating System presents the updates on the support for communication with the driver for Ethernet over PCIe and the driver for Ethernet over QSFP. The PCIe uses the Xilinx QDMA driver and the Xilinx Open NIC driver, both deployed on the host side; and the Xilinx Linux kernel, on the RISC-V side. The QSFP driver allows FPGA to FPGA communication and it has been implemented based on the driver developed in the EPI project, with a DMA-based solution and the ability of scatter-gather.

1.2. Compiler support

The compiler includes the contribution to the RISC-V Vector Extensions, which target the VPU accelerator and the Systolic Array extensions. With respect to the Vector Extension we have explored to main lines: one based on assessing whether prefetching techniques are feasible to inform the CPU about memory accesses of the vector code and another one exploiting loop transformations to improve the use of the vector registers. Although prefetching hints looked like to be a reasonable mechanism to convey memory accesses (specifically about the vector length) information to the vector processor, the results obtained from our implementation suggest this is not an effective way to inform the CPU about the memory characteristics of vectorised code. The loop transformation techiques are still on development and we aim to impact on the locality characteristics of the computational kernels.

The Systolic Array extensions present a set of new custom instructions targeting the Systolic Array accelerators. It includes a set of new registers, and new computational and memory operations.

Both extensions have been implemented in the LLVM compiler and distributed as a source code repository as well as a RPM Fedora package. The compiler distribution also includes the OpenMP runtime library used to provide parallelization services to the OpenMP applications.

1.3. Runtimes and libraries

The Linux distribution includes several libraries to complete the HPC-AI ecosystem: the MPICH MPI library, the COMPSs/PyCOMPSs workflows, the TensorFlow Lite and Apache Spark frameworks, and the BLIS and NumPy libraries. All of them available as Fedora installable packages.

COMPSs [30] is a task-based programming model and runtime system to implement parallel distributed workflows. Supported applications are executed in a master-worker mode, where the workflow is executed in the master process and the tasks are executed in the worker processes.

Apache Spark [11] is an open-source unified analytics engine for large-scale data processing. It provides an interface for programming cluster with implicit data parallelism and fault tolerance.

Either COMPSs or Apache Spark relies on top of the Java Virtual Machine (JVM), consequently we have also included this component as part of the software stack.

TensorFlow [9] is a free and open-source software library for Machine Learning (ML) and Artificial Intenlligence (AI) applications. TensorFlow Lite [12] provides the inference engine and it is designed focusing on edge environments.

BLIS [38] is the linear algebra library we recommend in the MEEP ecosystem. We have adapted it in order to exploit the vector capabilities of the system by extending the OpenMP annotation to also target SIMD directives. We also put forward an exploration of this library with an offload mechanism to execute BLIS services in environments that are characterized with one or multiple accelerators.

NumPy [35] is a Python package that has support for scientfic computing. It provides support for different multidimensional objects, and mathematical functions. NumPy leverages the optimizations implemented in the aforementioned custom BLIS library.

1.4. Containerization support

With respect to the containerization support, we have selected three container engines to validate our work: Moby, Podman and Singularity. Moby is the open source version of the Docker stack, which is the most popular container engine nowadays. Podman, also very popular, because it has a compatible interface with Docker. Finally, Singularity is the most popular container engine in the HPC field because it allows traditional HPC resource managers and devices.

1.5. Benchmark descriptions

We also layout a set of benchmarks that are used to analyse their behaviour on the available MEEP environments. These benchmarks range from system benchmarks, such as Stream, EPCC-OpenMP and EPCC-OpenMP/MPI to common HPC benchmarks: HPL, HPCG, FFTXLIB, Cloud-Microphysics and Advection-MPDATA [32].

In the Data Analytics side, we include the TensorFlow Lite models, which are a set of Neural Networks (NN) representative of the current Data Analytics architectures. Among the set of models we found: MNIST, VGG-19, NesNet50, and MobileNet. Besides the TensorFlow models we also evaluate the Epistasis application running on top of the Apache Spark framework. The application can be configured by means of different parameters which allow to run vectorial and non-vectorial code, change the number of nodes, the problem size (and its internal partitions), etc.

In the Workflow benchmarking side, we have two different workloads. One based on the Distributed Computing Library (Dislib), another based on the Hyperdimensional Computing framework. Both use cases leverages the COMPSs/PyCOMPSs runtime and will allow to test the behaviour of this kind of applications using the MEEP architectures.

1.6. Offload mode and multi-devices

This section refers to the offload mode.

We have implemented a prototype infrastructure supporting OpenMP offload between the Intel Host, acting as the application runner, and the RISC-V on the FPGA, acting as the device accelerator. Thus, the LLVM compiler is invoked to generate x86_64 code for the Host and RISC-V rv64imafdc code for the accelerator (i.e. the target regions).

The support for OpenMP *target* on the Host side is implemented as a plugin to the *libomptarget* library. In our case, we have adapted the plugin developed by FORTH in the EPI project to work with the RISC-V accelerator on the FPGA.

One of the possible scenarios considered earlier in the MEEP project was that a single node could offer many accelerators where work could be offloadd to. This led us to identify a gap in OpenMP support for offloading. Aligned to this, we have proposed an extension to OpenMP in which we introduce a new OpenMP construct called `target spread`. Instead of receiving a single `device` clause, the spread construct has a `devices` clause which represents the set of devices that will execute the offloaded region.

1.7. Software distribution: releases

One important aspect of the current software reporting period is to make all the software stack publicly available for downloading by means of releases. In the MEEP project the OS will be distributed as binary images which can be installed on the development board.

Once the users have a booting Operating System running on the ACME EA platform, they will be able to use other software components by means of three different types of releases: 1) Source code repositories; 2) RPMs packages; and 3) Containerized images.

2. Introduction

This document presents all the initial releases of the Software Stack components for the MareNostrum Experimental Exascale Platform (MEEP). It includes all the levels involved in the toolchain (i.e., the Operating System, the compiler, and containerization support); as well as the applications, benchmarks and kernels we plan to evaluate at the end of the project (i.e., system, HPC, Data Analytics, and workflows).

The Operating System, compiler and containerizaton support were already introduced in deliverable *D5.2 Linux with initial host interface release, based on the requirements document*. In this document we will report the status at this stage of the project.

The set of benchmarks was already introduced in deliverable *D5.1 Benchmark suite of HPC applications*. In this document we will establish the objectives we plan to reach using them in the MEEP Project (i.e., performance evaluation or co-design with hardware/compiler). Also, for each of the componets targeting the performance evaluation, we will describe the set of metrics we want to acquire and which specific aspect of performance we want to test: memory, compute, multi-thread, multi-process, or vectorial will be the most meaningful ones. We will finally report any modification/porting we have introduced in these codes in order to adapt them for the purposes of the study or the execution on the ACME platforms.

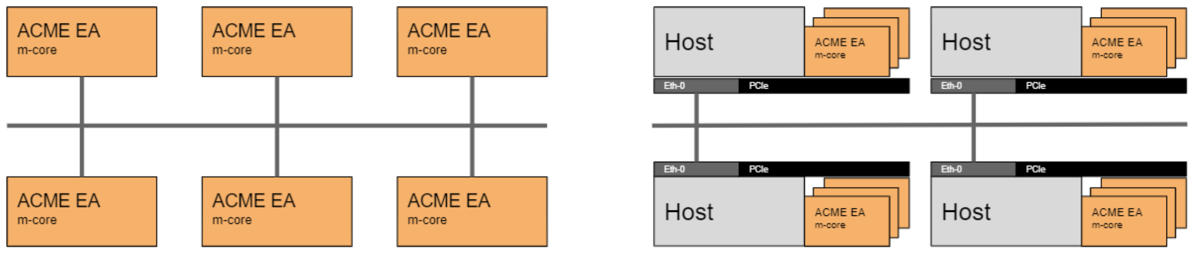
2.1. Type of releases

One important aspect of the current software reporting period is to make all the software stack publicly available for downloading. This deliverable will describe, for each of the presented software items, how they will be released.

The most important element on the Software Stack is the Operating System. It includes the OSBI and the File System based in the Fedora distribution. In the MEEP project they will be available as binary images which can be installed on the development board. The released OS will also contain the fundamental packages recommended to work on top of the ACME EA platform. These files can be found on the MEEP OS Layer, which also describes how these files can be installed.

Once the users have a booting Operating System running on the ACME EA platform, they will be able to use other software components by means of three different types of releases:

- Source code repositories: from where users may download the code and build it in their own platform.
- RPMs packages: that users may install or update from the repository sourced in their OS Fedora distribution.
- Docker images: that users may execute to use specific pre-configured software components (eg, TF Lite).



ACME EA: as a RISC-V self-hosted accelerator. **ACME EA:** as a RISC-V accelerator attached to a host.

Figure 1: MEEP Execution Modes: self-hosted vs offload.

2.2. Execution modes

The ACME EA platform can be used following two different approaches (see *D5.2 Linux with initial host interface release, based on the requirements document*; Section 2.2):

1. A stand-alone processor/accelerator booting Linux (ie, self-hosted). In this execution mode, the ACME EA becomes part of the HPC cluster;

2. A supporting accelerator device attached to a host. In this case the host becomes part of the HPC cluster, and it offloads parts of the computation to the ACME EA device.

Figure 1 illustrates these two approaches and how the HPC cluster is organized around the ACME EA computational system. As described in the previous deliverable, the main objective of the MEEP project is to target the self-hosted accelerator but it will also explore the offload execution mode and the opportunities this approach enables.

The rest of this document is organized as follows: Sections 3 to 8 refer to the self-hosted mode (ie, Operating System, Compiler, Runtimes/Libraries, Containerization support, Performance methodology, and Benchmarking), Section 9 describes all the components related with the MEEP offload-mode, and Section 10 presents the conclusions and summarizes all the software releases.

3. Operating System

The information about the Operating System has been already presented in MEEP Deliverable D5.2: Linux with initial host interface release, based on the requirements document [33]:

- Linux kernel boot and the boot flow process for ACME
- The Buildroot, Debian and Fedora portings
- The ACME memory map, Pmem disk, and Tun-on-Map basic networking

In this deliverable we present the updates on the support for communications with the driver for Ethernet over PCIe and the driver for 10/100Gbit Etherner over QSFP. More details of the Ethernet implementation can be found on Section 3 of the MEEP Deliverable D6.3: Emulated accelerator second release with full capacity of inter-accelerator communication [34].

3.1. Driver for Ethernet over PCIe

The FPGA infrastructure for ACME includes the IP dealing with the QDMA transactions. This infrastructure was initially only used to transfer the operating system and the filesystem image to the board. Later on, we used it from the user-level to implement the Tun-on-mmap communications, allowing a first implementation of Ethernet over PCIe.

The next development has been to move that communications infrastructure inside the kernel. This has been done in both sides, the host and the RISC-V. In order to do this, we used the following platforms:

- [Host side] The Xilinx QDMA driver source code (obtained from Xilinx DMA IP Drivers repo)
- [Host side] The Xilinx Open NIC driver source code (obtained from Xilinx Open NIC Driver repo)
- [RISC-V side] The Xilinx Linux kernel source code (obtained from Xilinx Linux repo)

On the FPGA infrastructure we have included a memory area in the I/O space that provides a non-cachable zone for data exchange between the QDMA driver on the host side, and the RISC-V. This infrastructure is described in Section 4.1 of the MEEP Deliverable D6.3: Emulated accelerator second release with full capacity of inter-accelerator communication [34].

3.1.1. Integrating the ONIC driver on QDMA

On the host side, we have taken advantage of Xilinx publishing the Open NIC driver, to use it as the basic structure to incorporate it on the QDMA driver. The new QDMA driver infrastructure developed in the MEEP project includes the support for Ethernet over PCIe.

In order to implement this new feature inside the QDMA driver, we have incorporated parts of the Open NIC driver, specifically:

- The creation of the Ethernet device.
- Enabling the DMA transfers of data from/to the kernel-mapped memory.
- The implementation of the Tun-on-mmap protocol from inside the kernel.

This code is available in the MEEP QDMA driver in this repository: MEEP QDMA+ONIC driver.

3.1.2. Developing the RISC-V driver counterpart

The RISC-V driver counterpart has been implemented based on the Xilinx Open NIC source code, by replacing the access to the DMA system to the use of the shared memory area in I/O space.

Being fully in the I/O space we ensure that the memory accesses from the host side through the QDMA+ONIC driver and the RISC-V accesses through the in-kernel /dev/mem device are coherent, and there are no cache-related issues.

This code has been incorporated in the Xilinx Linux version on the MEEP Lagarto Openpiton SDK repository.

3.2. Driver for 10Gbit Ethernet over QSFP

Providing Ethernet on the QSFP connection involves the RISC-V system running on the FPGA, that will be connected to another FPGA board in a point-to-point connection, or to a local switch.

The driver running on Linux on the RISC-V side has been implemented based on the driver developed in the EPI project, with a DMA-based solution and ability for scatter-gather. The driver accesses 2 types of data. On the one side, it uses DMA descriptors mapped onto non-cachable memory, ensuring that the DMA engine works properly.

On the other side, the driver receives and interacts with data buffers from the Linux kernel, on regular cacheble memory. As the Openpiton infrastructure is not providing cache flushing for coherency with memory accesses coming from the DMA engine, we have implemented a simple memory filling routine to try to flush the cache of previously accessed data. This solution is used right before setting the DMA up for transfering a packet, and it provides a temporary solution while we find another option to use.

This code is available in the MEEP Lagarto Openpiton SDK repository.

The Ethernet IP for the QSFP connection is described in Section 4.2 of the MEEP Deliverable D6.3: Emulated accelerator second release with full capacity of inter-accelerator communication [34].

3.3. Driver for 100Gbit Ethernet over QSFP

We have recently verified that the same driver that we use for 10Gbit Ethernet will support 100Gbit Ethernet communications. The only difference is that the 100Gbit IP hardware requires an additional initialization that will be implemented in the 10Gbit Ethernet driver, allowing 100Gbit transfers.

4. Compiler support

This section describes all the development carried out during the MEEP project aimed at supporting the MEEP architecture. This work includes contributions for the RISC-V Vector Extension, which targets the VPU accelerator component of MEEP, and the Systolic Array extensions. This work is mainly based on the LLVM infrastructure.

4.1. Infrastructure

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies under open source permissive licences. The most commonly known components of LLVM are the Core libraries (commonly known as LLVM) and the Clang C/C++ front end.

LLVM is based around the idea of a common intermediate representation called LLVM IR. This representation is powerful enough to cover a larger number of analyses and transformations that can be reused among different architectures. As a practical compiler, though, LLVM includes other representations that are used in specific parts of the compilation process. Clang has its own AST (Abstract Syntax Tree), the Codegen library of LLVM Core uses a low level representation called Machine IR, and the MC library of LLVM Core uses an even lower-level representation for encoding (assembly) and decoding (disassembly) instructions.

The work done on MEEP is built on top of the compiler developed in EPI SGA-1 which was extended in that project to support the RISC-V Vector Extension version 0.7.1 as implemented by the microarchitecture of the Vector Processor.

4.2. RISC-V Vector Extension optimisations

This section describes contributions that were done in the MEEP project with the goal to improve the code generation and the applicability of the RISC-V Vector Extension.

4.2.1. Prefetching

The RISC-V Vector Extension has been designed so it can adapt many implementation scenarios. This led to a design that is vector-length agnostic: the ISA does not prescribe a specific size for the vector registers. At the same time it provides enough functionality so it is possible to use the same sequence of vector instructions in implementations with different physical vector length.

An outcome of the work developed in the EPI project was a loop vectorisation strategy that is fully vector length agnostic. The compiler emits a vectorised loop that requests the CPU to process, using vector instructions, as many elements as the remaining iterations. This is called the *application vector length*. The CPU returns the available vector length it can honour based on the specific vector register size of the implementation.

The RISC-V Vector Extension defines what values the vector length returns but the code emitted

Application	Cycles NP	Cycles P	△Cycles (%)	#Insns NP	#Insn. P	△#Insns (%)	R Miss NP	R Miss P	Delta R Miss (%)
Blackscholes	35709	35077	-1.77%	71109	72237	1.59%	251	243	-3.19%
Somier	49101	48920	-0.37%	73472	73536	0.09%	373	301	-19.30%
SpMV	4727	5359	13.37%	3516	3672	4.44%	172	183	6.40%
Matmul	3115	3364	7.99%	905	1040	14.92%	96	106	10.42%

Table 1: Results without prefetch (NP) and compiler introduced prefetch (P).

by the compiler is fully agnostic. So a plausible scenario for an architecture that extends the RISC-V Vector Extension is to further this idea and let the CPU choose the vector length it deems ideal for a given iteration. As a way to communicate to the CPU what memory is going to be used by a loop, we looked at prefetch instructions, which in addition to do prefetch by themselves (of a future memory access). Not that the instructions are used in this context as hints to the CPU and not necessarily as a mechanism to enforce the memory prefetch.

The RISC-V community proposed a new set of instruction extensions called the RISC-V Base Cache Management Operations ISA Extensions (Zcmo). For the purpose of this exploration we only implemented support for the `prefetch.r` instruction in the compiler. We then modified LLVM so the IR intrinsic `llvm.prefetch` could emit this instruction. For the purpose of evaluation we modified the Coyote emulator so it could recognize the instruction and emulate a prefetch from the cache.

Then we implemented a simplified version of the approach described in [13] so the compiler inserted `llvm.prefetch` in loops containing set vector length instructions and memory references. This was evaluated against a small set of benchmarks to assess the feasibility of the technique.

The results are summarised in Table 1 and they yield mixed, inconclusive, results. All of them expose, expectedly, extra instructions executed, even if moderately like in the case of the Somier application. Some applications show a small improvement in number of cycles, which suggests their performance improves while others show a medium increase in cycles. The variations of the cycles correlates with the change of read misses ("R Miss" column) although this correlation is not totally clear. For instance, Somier reports a relatively large reduction of read misses but those do not translate into a much more improved performance, specially given that the number of extra instructions in the prefetch version is small.

It seems reasonable to conclude that the software prefetching mechanism based on the evaluated software prefetching algorithms, might not be the most suitable way to convey memory access information from software to the hardware.

4.2.2. Loop transformations for temporal locality

MEEP architecture features a vector ISA that provides a large (32) number of registers. Traditionally, to keep the functional units busy, applications need to make sure the register pressure is high while reducing the memory accesses. Minimising the number of memory accesses and trying to maximise the register utilisation is a similar problem to exploit temporal locality as much as possible.

Loop transformations are known to be able to dramatically impact the locality characteristics of computational kernels. We want to see if the application of these techniques is also useful to minimise memory accesses.

We used the Somier application as a case study. It simulates a 3D grid of of springs (like a three dimensional box spring). For each time step, using the position of the nodes, elastic forces are computed, then accelerations, velocities and then the new positions. The three magnitudes, position, forces, acceleration and velocities, are stored independently, and computed, for each node, one after the other. Listing 1 is a high level description of the simulation run by Somier.

```
foreach (t : timestamp) {
   foreach (p : nodes) {
     forces[p] <- compute_forces(position[p])
   }
   foreach (p: nodes) {
     accels[p] <- compute_accelerations(forces[p])
   }
   foreach (p: nodes) {
     velocs[p] <- compute_velocities(accels[p], t)
   }
   foreach (p: nodes) {
     positions[p] <- compute_positions(velocs[p], t)
   }
}
```

Listing 1: High-level scheme of the simulation implemented in Somier

Before implementing them in the compiler, we wanted to determine if the loop transformations would be favourable. Because the nodes are laid out in a 3D grid, each loop for nodes in Listing 1 is actually a 3-nested loop. The first thing we did was to linearise the three loops, called loop flattening, into a single loop that traverses all the nodes. Then we applied loop fusion. So we ended with a scheme like in Listing 2.

```
foreach (t : timestamp) {
   foreach (p : nodes) {
     forces[p] <- compute_forces(position[p])
     accels[p] <- compute_accelerations(forces[p])
     velocs[p] <- compute_velocities(accels[p], t)
     positions[p] <- compute_positions(velocs[p], t)
   }
}
```

Listing 2: Somier high level scheme, after loop fusion

After this change, though, the code is still using memory accesses to store and then load later. This exposes temporal locality so the cache can resolve these accesses. But we would like to avoid involving the memory system here. So we implemented a pass in the compiler that can remove those clearly redundant memory accesses.

Listing 3 shows the final loop. Figure 2 shows the reduction of loads in each iteration visible in a trace generated by vehave, a trap-based emulator developed in the EPI-SGA1 project. The trace shoes we are able to remove 6 of the vector loads (`vle64`) in every iteration.

```
foreach (t : timestamp) {
   foreach (p : nodes) {
     reg_forces := compute_forces(position[p])
     forces[p] <- reg_forces
     reg_accels := compute_accelerations(reg_forces)
     accels[p] <- reg_accels
     reg_velocs := compute_velocities(reg_accels, t)
     velocs[p] <- reg_velocs
     positions[p] <- compute_positions(reg_velocs, t)
```

```
    }
}
```
Listing 3: Somier high level scheme, with reduced load accesses

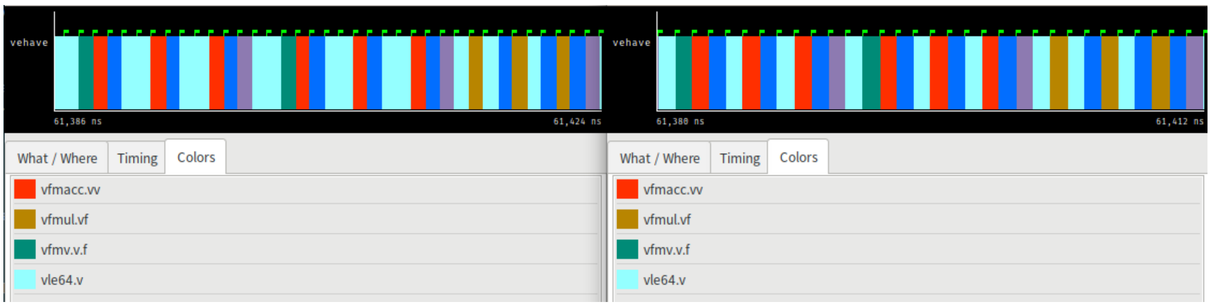


Figure 2: At the left, the original loop in Listing 2. At the right the loop in Listing 3.

4.3. Systolic Array

The MEEP architecture includes in its design two systolic arrays that act as accelerators. However, rather than presenting the accelerators like compute engines that must be accessed via I/O, MEEP chooses the path of integrating them in the ISA.

For this purpose we had to extend the RISC-V ISA with custom instructions that would allow operating the systolic array. This extension has been designed with some degree of flexibility in mind and it is inspired in some way by the RISC-V Vector Extension

- The ISA provides a set of 32 systolic array registers per systolic array.
- Operations use the systolic array register as explicit operands of the systolic array instructions.
- The ISA defines two *operational lengths*.
- Systolic array operations, including memory accesses, receive the operational lengths as implicit operands.
- The ISA defines a generic operation that each Systolic Array maps to the implemented function.

We implemented this extension in the LLVM compiler as assembly and disassembly support for the new instructions and systolic registers introduced. This required extending the MC layer of LLVM, whose task is to assemble (encode) and disassemble (decode instructions). Given the low level nature of the work carried out by the systolic arrays, there is no plan to implement a C/C++ intrinsic interface for this extension.

Refer to the Appendix A for a description of the current specification of this extension as implemented by the compiler

4.4. Multi-devices support

The multi-devices proposal, described in more detail in Section 9.2.1, was implemented in the clang C/C++ front-end of LLVM. Some minimal changes were also needed in the OpenMP runtime of OpenMP. The implementation is available at MEEP Compilers repository.

5. Runtimes and Libraries

In this section we present the additional packages, included in the Linux distribution, to complete the HPC-AI ecosystem: the MPICH MPI library, the COMPSs/PyCOMPSs workflows, the TensorFlow Lite and Apache Spark frameworks, and the BLIS and NumPy libraries. All of them available as Fedora installable packages.

5.1. Message Passing Interface

The Fedora 33 distribution comes with the MPICH MPI library as an installable package. MPICH is version 3.3.2. We install the package by default when generating the Fedora filesystem image.

We tested the MPI implementation with the HPCC - HPC Challenge benchmark, version 1.5.0. The benchmark has been run with 1, 2 and 4 cores on a single Ariane-based node and we found no issues while running this MPI application.

Running MPI is achieved with the command: mpirun -iface lo -np 2 <application-binary> <application arguments>

The MPI runtime is available at MEEP Runtimes webpage

5.2. OpenMP runtime

The OpenMP support was added through LLVM. The LLVM compiler and OpenMP support library (libomp.so) were imported from the EPI project.

The LLVM compiler allows to run OpenMP applications in the host server (Intel-based), and the RISC-V on the FPGA. On the RISC-V, we have run the STREAM benchmark to test the OpenMP support.

Additionally, we have implemented a prototype version of the OpenMP offload to allow the Intel host to spawn parallelism onto the RISC-V cores. This implementation is presented in section 9.

5.3. COMPSs runtime

In this section, we present how the COMPSs runtime has been ported to support ACME EA platforms. COMPSs is a task-based programming model and runtime system to implement parallel distributed workflows. Despite the core of the COMPSs runtime is written in Java it also offers bindings for C++ and Python (PyCOMPSs). These bindings interact with the runtime using Java Native Interface. COMPs/PyCOMPSs workflows are executed in a master-worker mode, where the workflow is executed in a master process and the tasks are executed in the worker processes which can be spawned in the same or different computing nodes. The spawn of the processes is performed by Secure Shell and the communication between master and worker nodes are performed by TCP/IP. The COMPSs runtime is integrated with Extrae in order to generate execution traces for performance analysis.

To enable the execution of COMPSs/PyCOMPSs application in the ACME EA prototype, the following main dependencies must be supported in the RISC-V 64 bits architecture and the ACME EA has to support profiling with Extrae and networking through TCP/IP protocol. The details about how the networking has been provided in the ACME EA prototype is explained in Section 3, and the profiling support in ACME EA is explained in Section 7.1 Regarding the first topic, the Python interpreter and the Secure Shell client and servers are available in the base Fedora 33 distribution, but the problematic dependency in this case was the Java support.

At the beginning of the Project COMPSs was requiring Java 8, however there is not a Java Virtual Machine (JVM) version 8 working for a RISC-V 64 bit architecture. In the Fedora distribution, we found a limited JVM for version 11. It is a ZeroVM implementation which does not include all the features and with limited performance because it do not include the Just-In-Time compiler. In the OpenJDk community there is a project for porting the full OpenJDK JVM for RISC-V 64 bits architecture which was starting in Java version 14. So, the main effort to port COMPSs to the ACME EA platform and other RISC-V architectures was devoted to support newer Java versions. It was a tedious task because there was a major change in the Java releases since version 9. Java EE and some other features included in the Java distribution where removed and ported to external projects, and the organization of the JVM libraries has also changed affecting the C and Python bindings. All these changes where initially ported to COMPSs version 2.10.1 and consolidated in version 3.0 and 3.1.

Source code of the COMPSs runtime including RISC-V 64 bits support can be found in COMPSs github repository and the RPM packages can be found in the MEEP RPM repository.

5.4. TensorFlow Lite framework

In this section we present how we ported the TensorFlow Lite runtime to make it work on Sifive HiFive Unmatched board and ACME-EA platforms successfully. TensorFlow Lite can be build either with bazel - which currently does not have support for RISC-V - or with CMake. Bazel was preferred as it is the main build system for the TensorFlow community. However it did not have support for RISC-V. Support was planned and some proposals to enable it were publicly made. We spend some effort on attempting to enable support following the suggested guidelines but eventually we found it was taking too long and switched to CMake.

To port TensorFlow lite we have first modified the basic build scripts to activate the following flags on GCC compiler: $ARMCC\_FLAGS = " - funsafe - math - optimizations"$ and add the following options to the cmake command line: *-DCMAKE_SYSTEM_NAME=Linux -DCMAKE_-SYSTEM_PROCESSOR=riscv64 -DTFLITE_ENABLE_XNNPACK=ON*.

The default CMake target builds a C library, so it was only possible to run benchmarks based on C. But our benchmarks were written in python. So we had to also compile and install a pip package for python. To build tensorflow lite as a pip package we have added the riscv64 option on the script *tensorflow/tensorflow/lite/tools/pip_package/build_pip_package_with_cmake.sh*. The modification consists on adding a switch case for riscv64 and correctly setting the variable *WHEEL_PLATFORM_NAME* to riscv64.

Most of the effort was devoted to finding the combination of flags that worked adequately. As it happened enabling some flags avoided some compilation errors however induced errors in other pieces of the software stack. After investigations we found out the combination of flags and operating system packages that needed to be installed so the runtime compiled successfully. Thus, additionally we need to install the following packages on the operating system:

- pytind11. It is needed to set appropriately the $INCLUDE\_PATH$ environment variable.
- python3-dev
- libboost-all-dev
- glibc2.33

TensorFlow Lite is offered as a RPM package containing the runtime with the modifications made on MEEP context. It can be found at: MEEP Toolchain webpage

5.5. Spark framework

Spark heavily relies on JVM for its core and encountered the same problems regarding the Java requirements in COMPSs. Once that was solved and java runtime was enabled no more modifications were needed to make Spark runtime work on MEEP systems. Spark is released as a RPM package containing the runtime and can be found at MEEP Toolchain webpage.

5.6. BLIS library

BLIS stands for BLAS-like Library Instantiation Software [38] and is the library employed to give applications the linear algebra functionalities that they required. In the context of the MEEP project, we explored BLIS in two different versions: Self-hosted and Offload (for more information on MEEP execution modes, we refer the reader to deliverable *D5.2*, *MEEP execution modes section*). Moreover, we provide the BLIS Self-hosted version to users, however the BLIS offload version was targeted only as an exploration task, hence, there is no BLIS offload release. The description/results of this exploration can be found at Section *The MEEP offload-mode*, sub-sections *BLIS single-device approach* and *BLIS multi-device approach*.

Description

The BLIS self-hosted version provides and explores the capabilities of executing this library natively in the accelerator. Here, we focus on two major features: 1) parallelism offered by the OpenMP programming model and also 2) the capabilities of the compiler to issue vector instructions when encountering SIMD directives.

Regarding the first feature, we are relying in the infrastructure already present and that applies the OpenMP programming model for exploring the parallelism capabilities of the platform. Some of the BLAS levels offered by BLIS do not leverage the multi-thread capabilities present in the library. For instance, level 1 BLAS routines (vector addition, axpy among others) lack this capability. On the other hand, level 3 BLAS routines, such as matrix multiplication, take advantage of this feature and performance improvements can be seen.

For OpenMP SIMD directives, BLIS also renders this capability, provided that, during the configuration phase, BLIS is able to detect that the compiler supports this directive. Unfortunately, BLIS is not always able to detect this feature, and for this reason we are modifying the implementation and explicitly add simd directives.

Modifications

As previously mentioned, instead of relying on this BLIS verification mechanism, we modified the source code to explicitly try to vectorize certain parts of the code if the compiler is able to. To highlight the nature of these modifications, check the next example.

Original source code:

```
// ...
PRAGMA_SIMD \
for ( dim_t i = 0; i < n; ++i ) \
{ \
  PASTEMAC(ch,addjs)( chi1[i], psi1[i] ); \
} \
// ...
```

Modified source code:

```
// ...
_Pragma("omp␣simd") \
for ( dim_t i = 0; i < n; ++i ) \
{ \
  PASTEMAC(ch,addjs)( chi1[i], psi1[i] ); \
} \
// ...
```

Library version and configurations

The version of this library used in MEEP is based on BLIS version 0.9.0, commit 4603324e [21]. Interestingly, the BLIS library provides a set of configurations that implement optimizations for a specific set of platforms. However, we cannot take advantage of these optimizations because there is no configuration for RISC-V platforms. For this reason, we have to rely on the generic configuration that uses the set of generic kernels and do not have optimizations in place that we can take advantage of.

In MEEP we explore and test this library in a large set of computing platforms with different features and behaviours. For this reason, we provide and maintain a BLIS version per platform because each one might need a different type of configuration. To this end, in the MEEP repository for BLIS, we have a branch for each of the necessary configurations and platforms. This allows to rapidly modify, adapt and deploy a particular version if we find a problem or improvement. On the down side, we pay the price of having a large set of versions (branches) that need to be maintained.

Last but not least, BLIS is available to users in two different flavours:

- BLIS source code per platform;
- BLIS RPM package: RPM packages for ACME-EA releases.

All of these releases are available at MEEP Toolchain webpage.

5.7. Numpy

Numpy is a Python package that has support for scientific computing. It provides efficient support for different multidimensional arrays, and mathematical functions. In order to get benefit of Numpy in the MEEP prototype we have to enable this python package to work with the custom BLIS library. Numpy uses the BLAS and LAPACK interfaces to access to the efficient implementation of multidimensional array sand linear algebra functions. To enable Numpy to run with the MEEP BLIS library, it has been installed from sources which can be found in the Numpy Github repository.

Before compiling Numpy, we have to enable LAPACK to use the MEEP BLIS library. We can compile one of the LAPACK implementations from source code linking it with the MEEP BLIS library. In our case, we have used the LAPACK reference implementation which can be found in the LAPACK github repository. To indicate the location of the BLAS library used in LAPACK you have to modify the *make.inc* setting the MEEP BLIS library path in the variable BLASLIB (eg. BLASLIB = /apps/riscv/ubuntu/blis/lib/libblis.so). Then you just need to follow the normal cmake installation.

Once we have LAPACK compiled with BLIS, we need to indicate the location of the LAPACK libraries and the BLIS library to the Numpy installation configuration. First, we had to edit the *cite.cfg* and set the BLIS library path at the *[blis]* tag. Once the path has been set, we have to compile Numpy specifying the location of the LAPACK in the *LAPACK* variable before executing the installation command. An installation command example can be found below.

```
LAPACK=/home/user/.local/lapack/liblapack.so ATLAS=None CFLAGS='-O3'\
   python3 setup.py install --user.
```

6. Container support

In this section, we present the work performed to allow the execution of containers in the ACME-EA platforms. We have selected three container engines to validate our work: Moby, Podman and Singularity. Moby is the open source version of the Docker stack which is the most popular container engine. Podman is also a very popular engine because has a compatible interface with Docker but with a simplified execution in rootless mode. Finally, Singularity is the most popular container engine in the HPC world because it easily works with traditional resource managers and devices

6.1. Enabling container support on ACME-EA

The work for enabling the container support for the ACME-EA platform is organized in two main tasks: one for testing and enabling the container engine software; and another to test and enable the required kernel modules are available in the system and properly configured.

Regarding the first task, we have check if the container engines or required dependencies were available in the Linux reference distribution for the project (Fedora-33) for the RISC-V 64-bits architecture. All three engines are implemented with Go, so it is required in the three cases. We did not find a working version of the container packages in the distribution and the version of the Go packages provided was not fulfilling the engines requirements. To fix it, we generate new RPM packages for the dependencies Go and runc as well as for Moby, Podman and Singularity which can be found in the MEEP RPM repository. The modified specs for generating this RPMS can be found in the MEEP OS RPM specs gitlab repository.

For enabling the execution of containers, the kernel must contain certain modules and the system must be configured in proper way to allow container engines to successfully create and run containers. To facilitate this task to system administrator, we have implemented an script which checks if the system is configured in a proper way (available at this repository). It is testing if mandatory and recommended modules such as cgroups, user namespaces, selinux, apparmor are available and properly configured, or some virtual networking capabilities are available or if resources limits are properly set. A part from that, it also test if one of the container engine is available and running, and finally it tries to run a ”hello world” testing container.

6.2. Working with distributed applications

The main difference between container engines is the networking management and it could affect the execution of distributed computing frameworks like MPI, COMPSs or Spark multi-node applications. In the case of Moby (Docker),it creates a virtual IP networks per host were containers are deployed. If you want to communicate to containers in different host, you have to create an overlay network to bridge the networks between nodes and deploy the containers in this nodes. It introduces an overhead for the overlay management and it is very difficult to use the MPI network fabrics (infiniband,...). In contrast, it facilitates the configuration of the framework, for instance you just need to set the MPI hosfile with the IPs of the containers. Another option for this container engine is to expose the ports used by the remote process managers and communication services (e.g 22/ssh, 443/https,...) to some port in the host and use the host IPs. This will reduce the overlay management overhead, but it will require a more complex framework configuration and the access to specific network fabric is not possible because the container is still using a virtual network.

In the case of Singularity, they use the host networking services by default. It has the disadvantage that the user has to be aware of the ports used by the hosts or other containers running in the same host. It does not allow two containers to be deployed using the same port. In contrast, it allows the usage of specialized networking fabrics as in the case of MPI applications. Network devices, drivers and libraries of the host can be bound and used from the container. More details about how to use MPI with Singularity containers can be found in this link.

6.3. Container releases

A part form enabling the use of containers, we have created several containers images compatible with the RISC-V 64bits architecture including some of the software stack elements. You can find them in the MEEP Container Image repository. In this repository we can find the following images:

- riscv64/fedora: A container image for RISC-V 64bit architecture with the a basic Fedora installation. It is used as the base image for the rest of images.
- riscv64/compss: Inherited from riscv64/fedora, it contains the COMPSs and PyCOMPSs programming model and runtime. It can be used to run the different COMPSs workflows described in Section 8.4.
- riscv64/tflite: Inherited from riscv64/fedora, it contains the TensorFlow Lite framework. It can be used to run the TensorFlow models described in Section 8.3.1.
- riscv64/spark: Inherited from riscv64/fedora, it contains the Spark framework. It can be used to run the Epistasis use case described in Section 8.3.2.

7. Performance Analysis Methodology

In this section we will describe the *profiling support* required to be installed in the MEEP software stack; as well as the POP [5] methodology, used as a driving model to carry on our performance analysis. POP methodology provides a quantitative way of measuring relative impact in performance of the different factors inherent in parallelisation. The section is completed by extending the POP methodology with vector analysis.

7.1. Profiling support

The following sections will describe the software components we have included in the software stack in order to acquire basic information about the execution of benchmarks. Extrae [15] will allow to generate Paraver [4] traces (events spread among timelines) that can be analyzed post-mortem.

PAPI [37] and Libunwind [3] enables the access to hardware counters and the execution callstack respectively. Such information will be requested by Extrae and injected in the Paraver trace in order to complete the view.

7.1.1. Extrae

Extrae [15] is the package devoted to generate Paraver [4] trace-files for a post-mortem analysis. Extrae is a tool that uses different interposition mechanisms on inject probes into the target application so as to gather information regarding the application performance.

In order to facilitate the configuration, Extrae can be configured through an XML file. The distributed package contains several examples.

1. **Interposition mechanisms**

   Extrae takes advantage of multiple interposition mechanisms to add monitors into the application. No matter which mechanism is being used, the target is the same, to collect performance metrics at known applications points to finally provide the performance analyst a correlation between performance and the application execution. Extrae currently uses the following interposition mechanisms:

   (a) **Linker preload (LD_PRELOAD)**

   Most of the current operating systems allow injecting a shared library into an application before the application gets actually loaded. If the library that is being preloaded provides the same symbols as those contained in shared libraries of the application, such symbols can be wrapped in order to inject code in these calls. In Linux systems this technique is commonly known by using the LD_PRELOAD environment variable. Extrae contains substitution symbols for many parallel runtimes, as OpenMP (either Intel, GNU or IBM runtimes), pthread, CUDA accelerate applications, and MPI applications.

This interposition mechanism has been the one most widely used in the context of MEEP throughout all the performance analysis that will be reported in D5.4.

(b) **DynInst**

Dyninst is an instrumentation library that allows modifying the application by injecting code at specific code locations. Although it originally allowed modifying the application code when the application was run, now it supports rewriting the binary of the application so the code injection is required only once. Extrae uses Dyninst to instrument different parallel programming runtimes as OpenMP (either for Intel, GNU or IBM runtimes), CUDA accelerated applications, and MPI applications. Dyninst also offers Extrae the possibility to easily instrument user functions by simply listing them in a file.

In the context of MEEP, we do not use this mechanism so the distributed Extrae in the MEEP Software Stack comes without *DynInst* support.

(c) **Additional instrumentation mechanisms**

Extrae also takes the advantage of some parallel programming runtimes that have their own instrumentation (or profile) mechanisms available for performance tools. These include the widely-known Message Passing Interface (MPI) which provides the Profile-MPI (PMPI) layer. There are some compilers that allow instrumenting application routines by using special compilation flags during compilation and link phases.

(d) **Extrae API**

Finally, Extrae gives the user the possibility to manually instrument the application and emit its own events it the previous mechanisms do not fulfill the user's needs. The Extrae API is detailed in the Extrae user-guide documentation that accompanies the package.

2. **Sampling mechanisms**

Extrae does not only offer the possibility to manually instrument the application code, but also offers to use sampling mechanisms to gather performance data. While adding monitors into specific location of the application produces insight which can be easily correlated with source code, the resolution of such data is directly related with the application control flow. Adding sampling capabilities into Extrae allows providing performance information of regions of code which has not been instrumented.

Currently, Extrae sports two different sampling mechanisms. The first mechanism is the old-known signal timers, which fires the sampling handler at a specific time interval. The second sampling mechanism uses the processor performance counters to fire the sampling handler at a specified interval of events interval. While the first mechanism can provide totally uncorrelated samples with the application code, the second mechanism, using the appropriate performance counters, can provide insight of the application but still presenting some correlation with the application code/performance.

The monitors added by Extrae gather different types of information. Depending on the monitor placement, each monitor can be taught to gather specific information. The most common information gathered is:

(a) **Timestamp**

When analyzing the behavior of an application, it is important to have a fine-grained timestamping mechanism (up to nanoseconds). Extrae provides a set of clock functions that are specifically implemented for different target machines in order to provide the most accurate possible timing. On systems that have daemons that inhibit the usage of these timers or that do not have a specific timer implementation, Extrae still uses advanced POSIX clocks to provide nanosecond resolution timestamps with low cost.

In the context of MEEP project we have used this last option by enabling it at the `configure` command of the building/installation Extrae process (`--enable-posix -clock`).

(b) **Performance and other counter metrics**

Extrae uses the PAPI and the PMAPI interfaces to collect information regarding the microprocessor performance. With the advent of the components in the PAPI software, Extrae is not only able to collect information regarding the microprocessor, but also allows studying multiple components of the system (disk, network, operating system, among others) and also extend the study over the microprocessor (power consumption and thermal information). Extrae mainly collects these counter metrics at the parallel programming calls and at samples. It also allows capturing such information at the entry and exit points of the instrumented user routines.

(c) **Reference to the source code**

Analyzing the performance of an application requires relating the code that is responsible for such performance. This way the analyst can locate the performance bottlenecks and suggest improvements on the application code. Extrae provides information regarding the source code that was being executed (in terms of name of function, file name and line number) at specific location points like programming model API calls or sampling points.

7.1.2. libunwind

The primary goal of this library is to define a portable and efficient C programming interface (API) to determine the call-chain of a program [3]. The API additionally provides the means to manipulate the preserved (callee-saved) state of each call-frame and to resume execution at any point in the call-chain (non-local goto). The API supports both local (same-process) and remote (across-process) operation. As such, the API is useful in a number of applications. Some examples include:

- **exception handling**

  The libunwind API makes it trivial to implement the stack-manipulation aspects of exception handling.

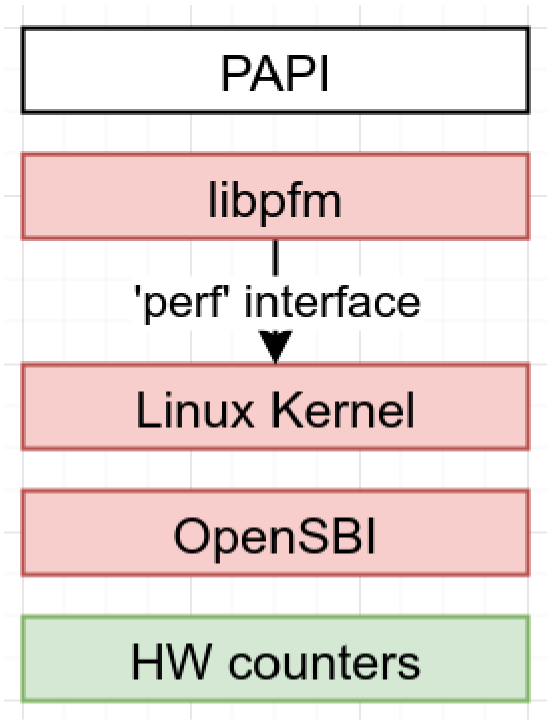


Figure 3: Software layers needed to access HW counters.

- **debuggers**

  The libunwind API makes it trivial for debuggers to generate the call-chain (backtrace) of the threads in a running program.

- **introspection**

  It is often useful for a running thread to determine its call-chain. For example, this is useful to display error messages (to show how the error came about) and for performance monitoring/analysis.

- **efficient setjmp()**

  With libunwind, it is possible to implement an extremely efficient version of setjmp(). Effectively, the only context that needs to be saved consists of the stack-pointer(s).

In the context of MEEP, we use Extrae (Section 7.1.1) to do the tracing and profiling and Extrae relies on libunwind for its *sampling* feature (needed by the proposed Vector Analysis Methodology). So we provide libunwind in the MEEP Software Stack through an RPM package.

7.1.3. Enabling hardware counters

Extrae (Section 7.1.1) leverages PAPI to read HW performance counters. Unfortunately, PAPI does not currently provide support for RISC-V architectures. This is mainly because PAPI relies, in turn, on lower software layers that lack (or have very preliminary RISC-V support) as are the red boxes depicted in Figure 3.

The option taken to overcome this limitation in the MEEP project was to use a **PAPI-like** interface that provides to Extrae the minimal API/functionality that it needs while reading the HW counters by accessing directly the RISC-V CSR event registers, thus avoiding the use of `libpfm`.

To make Extrae work with this PAPI-like interface, the following options were needed at the `configure` command of the building process: `--enable-riscv64 --with-papi=<path-papi-li`
`ke> --with-papi-headers=<path-regular-papi>/include`.

One last thing needed to read HW counters with this mechanism was to modify the OpenSBI to set the permissions to allow this access at startup. This is normally done on-the-fly through the `perf` kernel interface but, as we we are *shortcutting* its use, we need to *hardcode* the needed allowing permissions. Listing 4 shows the implemented modifications at lines 5 and 18-19.

```
 1 /* Disable user mode usage of all perf */
 2 /*counters except default ones (CY, TM, IR) */
 3 if (misa_extension('S') && sbi_hart_priv_version(scratch) \
 4   >= SBI_HART_PRIV_VER_1_10)
 5   csr_write(CSR_SCOUNTEREN, 7); -->csr_write(CSR_SCOUNTEREN, -1);
 6
 7 /**
 8 * OpenSBI doesn't use any PMU counters in M-mode.
 9 * Supervisor mode usage for all counters are enabled by default
10 * But counters will not run until mcountinhibit is set.
11 */
12 if (sbi_hart_priv_version(scratch) >= SBI_HART_PRIV_VER_1_10)
13   csr_write(CSR_MCOUNTEREN, -1);
14
15 /* All programmable counters will start running */
16 /*at runtime after S-mode request */
17 if (sbi_hart_priv_version(scratch) >= SBI_HART_PRIV_VER_1_11)
18   csr_write(CSR_MCOUNTINHIBIT, 0xFFFFFFF8);\
19   -->csr_write(CSR_MCOUNTINHIBIT, 0x00000000);
```

Listing 4: OpenSBI modifications in `mstatus_init` function (in `lib/sbi/sbi_hart.c`).

It is worth mention that this version of OpenSBI together with the PAPI-like interface are both included in the MEEP Software Stack.

7.2. POP Methodology

Attempting to optimise performance of a parallel code can be a daunting task, and often it is difficult to know where to start. For example, we might ask if the way computational work is divided is a problem? Or perhaps the chosen communication scheme is inefficient? Or does something else impact performance? To help address this issue, POP ([5]) has defined a methodology for analysis of parallel codes to provide a quantitative way of measuring relative impact of the different factors inherent in parallelisation. This subsection introduces these metrics, explains their meaning, and provides insight into the thinking behind them.

A feature of the methodology is that it uses a hierarchy of metrics (Figure 4), each metric reflecting a common cause of inefficiency in parallel programs. These metrics then allow comparison of parallel performance (e.g. over a range of thread/process counts, across different machines, or at different stages of optimisation and tuning) to identify which characteristics of the code

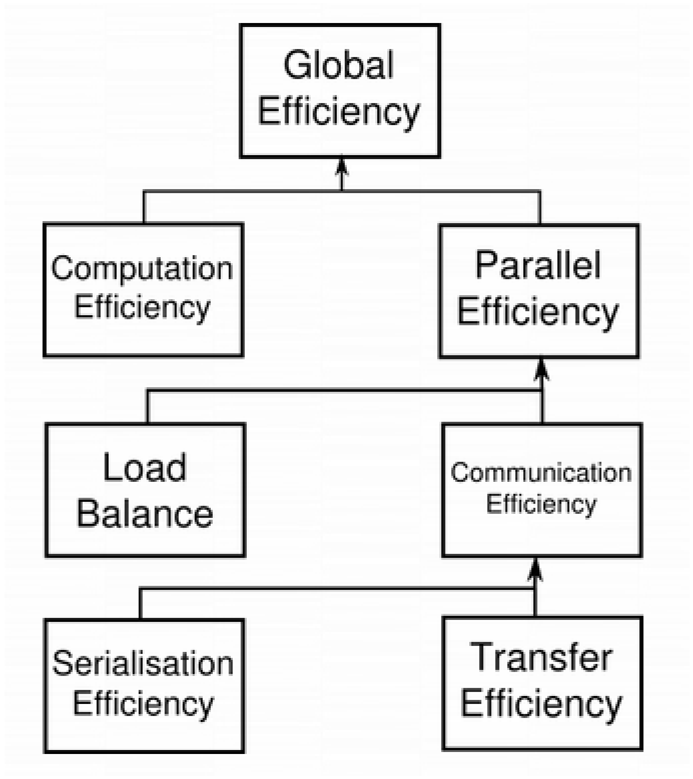


Figure 4: POP metrics.

contribute to inefficiency.

The first step for calculating these metrics is to use a suitable tool (e.g. Extrae ([15])) to generate trace data whilst the code is executed. The traces contain information about the state of the code at a particular time (e.g. it is in a communication routine or doing useful computation) and also contains values from processor hardware counters (e.g. number of instructions executed, number of cycles).

The metrics are then calculated as efficiencies between 0 and 1, with higher numbers being better. In general, we regard efficiencies above 0.8 as acceptable, whereas lower values indicate performance issues that need to be explored in detail. The ultimate goal then for the POP methodology is rectifying these underlying issues.

The approach outlined here is applicable to various parallelism paradigms, however for simplicity the POP metrics presented here are couched in terms of a distributed-memory message-passing environment (e.g. MPI). For this the following values are calculated for each process from the trace data: time doing useful computation, time in communication, number of instructions & cycles during useful computation. Useful computation excludes time within the overheads of parallelism.

At the top of the hierarchy is ***Global Efficiency (GE)***, which is used to judge overall quality of parallelisation. Typically, inefficiencies in parallel code have two main sources:

- Overheads imposed by the parallel nature of a code

- Poor scaling of computation with increasing numbers of processes

and to reflect this we define two sub-metrics to measure these two inefficiencies. These are ***Parallel Efficiency*** and ***Computation Efficiency***, and our top-level GE metric is the product of these two sub-metrics:

GE = Parallel Efficiency * Computation Efficiency

Parallel Efficiency (PE) reveals the inefficiency in splitting computation over processes and then communicating data between processes. As with GE, PE is a compound metric whose components reflect two important factors in achieving good parallel performance in code:

- Ensuring even distribution of computational work across processes

- Minimising time communicating data between processes

These are measured with ***Load Balance Efficiency*** and ***Communication Efficiency***, and PE is defined as the product of these two sub-metrics:

PE = Load Balance Efficiency * Communication Efficiency

Load Balance (LB) is computed as the ratio between average useful computation time (across all processes) and maximum useful computation time (also across all processes):

LB = average computation time / maximum computation time

Communication Efficiency (CommE) is the maximum across all processes of the ratio between useful computation time and total runtime:

CommE = maximum computation time / total runtime

CommE identifies when code is inefficient because it spends a large amount of time communicating rather than performing useful computations. CommE is composed of two additional metrics that reflect two causes of excessive time within communication:

- Processes waiting at communication points for other processes to arrive (i.e. serialisation)

- Processes transferring large amount of data relative to the network capacity

These are measured using ***Serialisation Efficiency*** and ***Transfer Efficiency***. For a detailed description of these two submetrics, please refer to [5].

The final metric in the hierarchy is ***Computation Efficiency (CompE)***, which are ratios of total time in useful computation summed over all processes. For strong scaling (i.e. problem size is constant) it is the ratio of total time in useful computation for a reference case (e.g. on 1 processor or 1 compute node) to the total time as the number of processes (or nodes) is increased. For CompE to have a value of 1 this time must remain constant regardless of the number of processes.

Insight into possible causes of poor computation scaling can be investigated using metrics devised from processor hardware counter data. Two causes of poor computational scaling are:

- Dividing work over additional processes increases the total computation required

- Using additional processes leads to contention for shared resources

these can be investigated using ***Instruction Scaling*** and ***Instructions Per Cycle (IPC) Scaling***.

Instruction Scaling is the ratio of total number of useful instructions for a reference case (e.g. 1 processor) compared to values when increasing the numbers of processes. A decrease in Instruction Scaling corresponds to an increase in the total number of instructions required to solve a computational problem.

IPC Scaling compares IPC to the reference, where lower values indicate that rate of computation has slowed. Typical causes for this include decreasing cache hit rate and exhaustion of memory bandwidth, these can leave processes stalled and waiting for data.

7.3. Vector methodology

The main goal of this task is to create a vector analysis methodology that will allow to compare application performance with respect to the vector arithmetic behavior. The vector analysis methodology is based on two main ideas. First, vector coverage, representing the portion of code that has been actually vectorized. Second, vector efficiency, representing the actual length of vector instructions with respect to the maximum allowed by the architecture. We defined different metrics that may capture both coverage and efficiency of the vectorial behavior of applications.

Dealing with vector coverage, we propose the following metrics:

- **Arithmetic Computational Density (ACD)**, measures the number of arithmetic instructions with respect to the total number of instructions.

- **Arithmetic Vector Density (AVD)**, measures the number of vector arithmetic instructions with respect to the total number of arithmetic instructions.

Dealing with vector efficiency, we propose the following metric:

- **Average Vector Length (AVL)**, measures the average vector length for all vector arithmetic instructions.

In addition to these metrics, we also recommend to substitute the **Instructions Per Cycle (IPC)** measurement for **Operations Per Cycle (OPC)**; due in applications sensitive to use vector instructions the IPC is not as important as OPC, so the latest will be the target to maximize.

One of the main goals of these metrics is to be generic and they can be potentially applied in any HW architecture. In order to calculate them, the following set of HW counters is required:

- Set of counters to measure actual number of operations:

  - **BYTE_OPS**: To count the number of arithmetic byte type operations.

  - **HALF_OPS**: To count the number of arithmetic half-word type operations.

  - **WORD_OPS**: To count the number of arithmetic word operations.

- Set of counters to measure actual number of instructions:
  - **S_BYTE_INS**: To count the number of arithmetic scalar byte type instructions.
  - **S_HALF_INS**: To count the number of arithmetic scalar half-word type instructions.
  - **S_WORD_INS**: To count the number of arithmetic scalar word type instructions.
  - **V_BYTE_INS**: To count the number of arithmetic vector byte type instructions.
  - **V_HALF_INS**: To count the number of arithmetic vector half-word type instructions.
  - **V_WORD_INS**: To count the number of arithmetic vector word type instructions.

Provided that previous counters are available together with other well-known counters such as INS (number of instructions) and CYC (number of cycles), the proposed vector analysis metrics can be computed as follows:

Computational Density:

$$CD = \frac{S\_BYTE\_INS+S\_HALF\_INS+S\_WORD\_INS+V\_BYTE\_INS+V\_HALF\_INS+V\_WORD\_INS}{INS}$$

Arithmetic Vector Density:

$$AVD = \frac{V\_BYTE\_INS+V\_HALF\_INS+V\_WORD\_INS}{S\_BYTE\_INS+S\_HALF\_INS+S\_WORD\_INS+V\_BYTE\_INS+V\_HALF\_INS+V\_WORD\_INS}$$

We can easily compute a new derived metric called **Vector Computational Density (VCP)** as the product of *Computational Density* and *Arithmetic Vector Density* (VCD=AVD*CD).

The **Average Vector Length (AVL)** can be computed per data type:

- **AVL_b** $= \frac{BYTE\_OPS}{V\_BYTE\_INS+S\_BYTE\_INS}$
- **AVL_h** $= \frac{HALF\_OPS}{V\_HALF\_INS+S\_HALF\_INS}$
- **AVL_w** $= \frac{WORD\_OPS}{V\_WORD\_INS+S\_WORD\_INS}$

Or we can compute an aggregated value for all the types as:

$$AVL = \frac{BYTE\_OPS+HALF\_OPS+WORD\_OPS}{S\_BYTE\_INS+S\_HALF\_INS+S\_WORD\_INS+V\_BYTE\_INS+V\_HALF\_INS+V\_WORD\_INS}$$

The **Operations Per Cycle (OPC)** metric can be computed as:

$$OPC = \frac{BYTE\_OPS+HALF\_OPS+WORD\_OPS}{CYC}$$

If we want to consider also memory instructions, an extended set of HW counters is needed by adding the following ones:

- **L_BYTE_ST1**: To count the number of load instructions (byte type, stride 1)
- **L_BYTE_STN**: To count the number of load instructions (byte type, stride n)
- **L_BYTE_IND**: To count the number of load instructions (byte type, indexed)

- **L_HALF_ST1**: To count the number of load instructions (half-word type, stride 1)
- **L_HALF_STN**: To count the number of load instructions (half-word type, stride n)
- **L_HALF_IND**: To count the number of load instructions (half-word type, indexed)
- **L_WORD_ST1**: To count the number of load instructions (word type, stride 1)
- **L_WORD_STN**: To count the number of load instructions (word type, stride n)
- **L_WORD_IND**: To count the number of load instructions (word type, indexed)

And the equivalent **store** versions: S_BYTE_ST1, S_BYTE_STN, S_BYTE_IND, S_HALF_ST1, S_-HALF_STN, S_HALF_IND, S_WORD_ST1, S_WORD_STN, S_WORD_IND.

Our final goal is to use this set of metrics in the MEEP environments (RISC-V and ACME) but in the meantime we have started working in x86 architectures due to: 1) some extra complexities arising from using PAPI in RISC-V architectures; and 2) the activity is a collaboration with the POP2 Centre of Excellence, that usually apply performance analysis methodologies to commodity clusters.

We defined the metrics we described previously, together with others that help us in the analysis, in terms of the x86 PAPI counters available on MN4 for double precision instructions/operations:

- AVL= $\frac{PAPI\_DP\_OPS}{PAPI\_VEC\_DP}$
- OPC= $\frac{PAPI\_DP\_OPS}{PAPI\_TOT\_CYC}$
- IPC= $\frac{PAPI\_TOT\_INS}{PAPI\_TOT\_CYC}$
- ACD= $\frac{PAPI\_VEC\_DP}{PAPI\_TOT\_INS}$
- AVD=

$= \frac{FP\_ARITH{:}128B\_PACKED\_DOUBLE + FP\_ARITH{:}256B\_PACKED\_DOUBLE + FP\_ARITH{:}512B\_PACKED\_DOUBLE}{PAPI\_VEC\_DP}$

When compiling we found the different compiler flags needed to enable/disable vectorizations and setting the vector length used by the hardware on an x86 architecture. In this regard, four different set-ups have been tested for the different applications/benchmarks:

- AVX-512: flags to enable AVX-512 vectorization are used.
- AVX-2: flags to enable AVX-2 vectorization are used.
- NO FLAG: no specific flag related to vectorization is passed to the compiler.
- NO VEC: vectorization is explicitly disabled.

7.3.1. Validation with sample codes

The following codes have been analysed following the methodology:

- vAdd (implemented synthetic kernel, see Listing 5, similar to the *Add* kernel part of the Stream benchmark, see Section 8.1.1).
- DAXPY (implemented synthetic kernel, see Listing 6, same as the *Axpy* kernel part of the RISC-V benchmarks presented in Section 8.2.1).
- FFTXlib (see Section 8.2.4 for a detailed description).
- HPCG (see Section 8.2.3 for a detailed description).
- CloudMicrophysics (see Section 8.2.5 for a detailed description).

Two different modes of analysis have been envisioned: when the code is just one kernel (like vAdd or DAXPY) we just capture the values of the HW counters for the whole execution and calculate the corresponding metrics.

Otherwise, it is, when we are dealing with more complex codes including multiple functions, we use sampling. Extrae offers to use sampling mechanisms to gather performance data [15]. This technique allows us to perform a differentiated study *per function*, as one would expect different vectorial behavior on each one.

We found out, however, that in most cases the sampling rate provided by Extrae was not enough to capture with accuracy the real behavior. In these cases, we included two more techniques, namely, clustering [14] and folding [16].

Cluster analysis is applied to detect different trends in the application computation regions with minimum user intervention. This detection provides an unique insight of the application behavior that serves as a starting point to perform different types of analyses around the applications' computation structure.

The folding provides very detailed performance information of these code regions on iterative and regular applications. The folding combines the instrumentation with the sampling information to unveil the performance evolution and to augment the details offered by simply using instrumentation or sampling. The folding consists in *collapsing* all samples obtained in the different identified clusters in the clustering phase into one synthetic representative instance of each cluster.

So, depending on the kind of code under study, the methodology is established as follows:

- Simple kernels: Extrae tracing + Paraver analysis.
- Benchmarks: Extrae tracing with Sampling to map HW counters readings with code functions + Clustering + Folding to increase the number of samples per cluster + Paraver analysis.

The defined metrics have been gathered, first, for the two proposed synthetic kernels. The source code for both synthetic kernels can be seen in Listing 5 (vAdd) and Listing 6 (DAXPY) respectively.

```
#define LENGTH 80000000
void main(void) {
   double y[LENGTH], x[LENGTH];
   for(int i = 0; i < LENGTH; i++)
      y[i] = x[i] + y[i];
}
```

Listing 5: vAdd source code.

Version	AVL	OPC	IPC	ACD	AVD
AVX-512	8.00	0.20	0.18	0.14	1.00
AVX-2	4.00	0.19	0.32	0.15	1.00
NO FLAG	2.00	0.16	0.52	0.16	1.00
NO VEC	1.00	0.12	0.72	0.16	0.00

Table 2: vAdd results.

Version	AVL	OPC	IPC	ACD	AVD
AVX-512	8.00	0.41	0.18	0.28	1.00
AVX-2	4.00	0.40	0.34	0.30	1.00
NO FLAG	2.00	0.25	0.46	0.27	1.00
NO VEC	1.00	0.16	0.57	0.28	0.00

Table 3: DAXPY results.

```
#define LENGTH 80000000
void main(void) {
   double y[LENGTH], x[LENGTH];
   double a = 3.0;
   for(int i = 0; i < LENGTH; i++)
      y[i] = a*x[i] + y[i];
}
```

Listing 6: DAXPY source code.

These metrics were obtained on BSC's MareNostrum4, using Intel compiler version 17.0.4, and for each version we highlight the following compilation flags:

- AVX-512: `-qopenmp -O3 -xCOMMON-AVX512`
- AVX-2: `-qopenmp -O3 -xCORE-AVX2`
- NO FLAG: `-qopenmp -O3`
- NO VEC: `-qopenmp -O3 -no-vec`

Results for vAdd synthetic kernel are presented in Table 2.

Regarding AVL, the observed behavior perfectly matches the theoretically expected. It looks that AVX-512 instructions are really costly (this can be inferred by comparing with AVX-2, where OPC is almost the same while keeping a relatively higher IPC). Results in the IPC column are also reasonable as vector instructions are costly. Both AVL and AVD columns allow for a sanity check to confirm that the used compilation flags are working as expected.

Next, Table 3 presents the gathered metrics for DAXPY synthetic kernel.

Basically the same conclusions as for vAdd apply. In this case the values in OPC and ACD columns double the ones shown for vAdd: this is also quite expected as in this case we are performing two floating-point operations in each iteration of the loop (instead of just one).

Version	AVL	OPC	IPC	ACD	AVD
AVX-512	1.08	1.28	2.27	0.52	0.01
AVX-2	1.07	1.26	2.31	0.51	0.02
NO FLAG	1.05	1.10	2.67	0.39	0.04
NO VEC	1.05	1.10	2.68	0.39	0.04

Table 4: FFTXlib results.

Version	AVL	OPC	IPC	ACD	AVD
AVX-512	4.99	0.93	1.02	0.18	0.61
AVX-2	1.73	0.67	1.47	0.26	0.26
NO FLAG	1.73	0.69	1.65	0.24	0.73
NO VEC	1.00	0.64	1.75	0.36	0.00

Table 5: HPCG results.

To highlight the use of this vector analysis methodology, Table 4, Table 5, and Table 6 report the average results obtained for the FFTXlib, HPCG, and the CloudMicrophysics kernel benchmarks respectively.

Overall, it can be seen that FFTXlib does not benefit from vectorization. Only a small increase in both AVL and OPC can be observed when enabling longer vector lengths by compilation flags (avx512 and avx2). This may be explained by the fact that this benchmark relies on scalar instructions to perform all the required computations. This fact is also highlighted by the AVD metric, which is the ratio between arithmetic vector instructions and overall arithmetic instructions (as we can see, this value is almost zero). This behavior, however, is not uniform across all sampled functions as it can be seen in Table 7.

The HPCG benhmark presents better numbers when compared to FFTXlib in terms of vectorization. It can be seen that, for instance, AVL metric is more than half of the theoretical value for the AVX-512 version. This is also underlined by the AVD value, which tell us that more than half of the arithmetic instructions executed are vector instructions. Table 8 presents the detailed results splitted by functions for this benchmark.

Detailed (per function) results for the CloudMicrophysics kernel are not provided as only one function is sampled (cloudsc_c) so the results are exactly the same than the average already presented in Table 6.

Version	AVL	OPC	IPC	ACD	AVD
AVX-512	1.31	0.47	1.73	0.21	0.04
AVX-2	1.27	0.49	1.83	0.21	0.09
NO FLAG	1.17	0.47	1.95	0.20	0.17
NO VEC	1.00	0.44	2.04	0.22	0.00

Table 6: cloudsc results.

Function	Coverage	AVL	OPC	IPC	ACD	AVD
fftw_no_twiddle_32	11.48%	1.16	1.03	2.10	0.43	0.02
fftwi_twiddle_9	10.39%	1.01	1.81	2.98	0.60	0.00
fftw_twiddle_9	10.30%	1.01	1.79	2.97	0.60	0.00
fftwi_no_twiddle_32	9.01%	1.00	1.04	2.37	0.44	0.00
fftwi_no_twiddle_9	7.79%	1.01	1.81	2.98	0.60	0.00
fftw_no_twiddle_9	7.48%	1.00	1.80	2.99	0.60	0.00
prepare_psi	7.07%	2.00	0.28	0.63	0.22	1.00
test(MAIN_)	4.93%	3.95	1.46	0.88	0.42	0.42
fft_y_stick_	0.04%	0.99	2.46	2.46	0.41	0.00

Table 7: FFTXlib results for AVX-512 case detailed by function. The coverage column represents the percentage of the total execution **time** spent in each function.

Function	Coverage	AVL	OPC	IPC	ACD	AVD
ComputeSYMGS_ref	71.45%	4.72	0.89	0.98	0.19	0.56
ComputeSPMV_ref	27.41%	5.95	1.05	1.12	0.16	0.75

Table 8: HPCG results for AVX-512 case detailed by function. The coverage column represents the percentage of the total execution **time** spent in each function.

8. Benchmarks description

This section describes the set of benchmarks used to explore the performance of MEEP environments. This set of benchmarks is divided into four categories: System, HPC, Data Analytics and Workflows. The performance analysis of all these benchmarks will be reported in deliverable *D5.4 - Final Release of the Software Stack*.

8.1. System benchmarks

This section describes the benchmarks used to explore the intrinsic performance of a system in terms of memory system bandwidth (Stream) and overheads of common HPC programming models such as OpenMP and MPI (EPCC OpenMP and EPCC OpenMP/MPI).

To test the benchmarks on all MEEP environments which have specific features, we have created configuration files per environment and a set of make and run scripts to deal with all of this diversity and to have an automatic methodology to build and run them. Therefore, from an user perspective the steps needed to test any of the benchmarks on a specific platform are:

1. Compile the benchmark for the desired platform:

   ```
   ./make-meep-bench.sh <platform-name>
   ```

2. Configure the SLURM parameters for the desired run:

   ```
   ./configure-slurm <slurm-config>
   ```

3. Execute the benchmark for the desired platform:

   ```
   ./run-meep-bench.sh <platform-name>
   ```

In the end, the user will have all the results under the **output** folder, that will also include all the compilation information.

8.1.1. Stream

Description

The STREAM benchmark is a synthetic benchmark built with the intent of measuring the memory bandwidth of accessing the main memory of a system (in MB/s), by executing simple vector kernels (copy, scale, add and triad) [28]:

- Copy: `c[i] = a[i]`
- Scale: `b[i] = s * c[i]`
- Add: `c[i] = a[i] + b[i]`
- Triad: `a[i] = b[i] + s * c[i]`

Objectives

As stated in the description, Stream is used to measure the performance of accessing the main memory system of a computing platform. In this project, we use Stream to evaluate the performance of the entire memory system (from cache to main memory). Furthermore, Stream is also used as a co-design tool in the development process of the various iterations of the ACME computing platforms, with special focus on the memory architecture. Specifically, this benchmark is used to test memory architecture features that are being deployed. By analyzing the resultant performance, conclusions are drawn and feedback is provided to the hardware developers to aid in the development of these computing platforms.

Given that we are testing the memory bandwidth of various MEEP environments, reading memory bandwidth in ***MB/s*** does not provide a clear picture when comparing them. Therefore, we modified Stream to report memory bandwidth in ***Bytes per Cycle (bytes/cycle)***:

$bytes/cycle = \frac{M}{T} * \frac{1}{F},$

M is the total amount of memory involved in the test (in bytes), T is the execution time of the test (in seconds) and F is the clock frequency of the processor (in Hz).

Modifications

In the context of the MEEP project, we extended the capabilities of the Stream benchmark to not only give us the bandwidth of accessing main memory but also to inspect the bandwidth of the different cache levels of a system. Measuring the performance of the different cache levels may present some challenges regarding the default timing model of Stream. Originally, Stream iterates over ***NTIMES*** each specific kernel (copy, scale, add and triad):

```
for (i=0;i<NTIMES;i++){
  time(start);
  kernel();
  time(end);
}
```

Which means that, depending on the size and complexity of the test and the granularity of the timing model, we may get an invalid elapsed time. To circumvent this issue we moved the timing operations outside of the for loop, and increased substantially the ***NTIMES*** variable to make sure that in the limit we get closer to a valid elapsed time and also to eliminate the loop overhead.

```
time(start);
for (i=0;i<NTIMES;i++){
  test();
}
time(end);
```

Additionally, we also modified the time function from ***mysecond()*** that uses ***gettimeofday()*** to ***clock_gettime()***. This allow us to have nanosecond resolution which may be required to time the tests that have small array sizes.

How are we able to measure the different cache levels bandwidth?

Even though we may not be able to accurately map each test to the desired cache level, one can attempt to do so by varying the array sizes of each test. To exemplify this idea, let's assume we have a computer system with only one cache level with a size of S_c, a main memory of size S_m and let's us also define the test as a simple array copy, such that

```
for(int i=0;i<N;i++){
  y[i]=x[i]
}
```

In this scenario, we will be working over two arrays of sizes S_x and S_y and to make sure that we may be working on the cache we have to make:

$S_x + S_y < S_c$.

At the same time, to make sure we are accessing main memory we have to ascertain that:

$S_c < S_x + S_y < S_m$.

Software release

We provide the source code and workload configurations used to test each MEEP environment. This is available at MEEP Benchmarks webpage (Stream table entry).

8.1.2. EPCC-OpenMP

Description

The EPCC OpenMP benchmark measures the computational overhead in micro-seconds of multiple OpenMP directives [18, 20, 27].

Objectives

From the complete set of OpenMP directives that this benchmark provides, we selected the following subset:

- Synchronisation directives
- Loop scheduling clauses
- Tasking constructs

Starting with the selected **synchronization directives**, it is highlighted the important characteristics and implications of each them:

- ***Parallel construct*** (Listing 7): Defines a parallel region with a specific number of threads. All threads execute the code within this parallel region and at the end there is an implicit barrier (synchronisation point for all threads).

```
for (j = 0; j < innerreps; j++) {
  #pragma omp parallel
  {
    delay(delaylength);
  }
}
```

Listing 7: Benchmark source code that uses the parallel construct: All threads will execute the

delay function.

- ***For loop construct*** (Listing 8): Used to parallelise the execution of all iterations in a for loop.

```
#pragma omp parallel private(j){
  for (j = 0; j < innerreps; j++) {
    #pragma omp for
    for (i = 0; i < nthreads; i++) {
      delay(delaylength);
    }
  }
}
```

Listing 8: Benchmark source code that uses the parallel construct followed by a omp for directive.

- ***Single construct*** (Listing 9): Define a region of code within a parallel region that is executed by only one thread. The internal mechanism of control is by using a flag to define if a thread should execute this region (when the flag is set other threads ignore this code region). There is an implicit barrier at the end of this region.

```
#pragma omp parallel private(j)
{
  for (j = 0; j < innerreps; j++) {
    #pragma omp single
    delay(delaylength);
  }
}
```

Listing 9: Benchmark source code that uses the single construct: Only one thread will execute the ***delay*** function.

- ***Critical construct*** (Listing 10): Defines a section of code that can only be executed by one thread at a time.

```
#pragma omp parallel private(j)
{
  for (j = 0; j < innerreps / nthreads; j++) {
    #pragma omp critical
    {
      delay(delaylength);
    }
  }
}
```

Listing 10: Benchmark source code that uses the critical construct: One thread at a time will execute the ***delay*** function.

- ***Atomic construct*** (Listing 11): Defines that a single statement that modifies the value of a variable, in a parallel region, can only be executed by one thread at a time.

```
#pragma omp parallel private(j) firstprivate(b)
{
  for (j = 0; j < innerreps / nthreads; j++) {
    #pragma omp atomic
    aaaa += b;
    b *= c;
  }
}
```
Listing 11: Benchmark source code that uses the critical construct: One thread at a time will execute the ***aaaa +=b*** instruction.

- ***Lock/unlock runtime routine*** (Listings 12):

```
omp_lock_t lock;
#pragma omp parallel private(j)
{
  for (j = 0; j < innerreps / nthreads; j++) {
    omp_set_lock(&lock);
    delay(delaylength);
    omp_unset_lock(&lock);
  }
}
```
Listing 12: Benchmark source code that uses the lock/unlock routines: All threads will execute the ***delay*** function one at a time.

Following with the **loop scheduling clauses**:

- ***Static*** (Listings 13): As the name states, this schedule mechanism assigns a fixed number of iteration chunks to each thread (usually in a round-robin fashion). Importantly, the major difference between ***static*** and ***guided*** is the fact that the assignment of iterations to threads is done before computations in the loop start, rendering this clause overhead smaller compared with ***static*** and ***guided***.

```
#pragma omp parallel private(j)
{
  for (j = 0; j < innerreps; j++) {
    #pragma omp for schedule(static,cksz)
    for (i = 0; i < itersperthr * nthreads; i++) {
      delay(delaylength);
    }
  }
}
```
Listing 13: Benchmark source code that uses the static schedule clause.

- ***Dynamic*** (Listings 14): In this schedule mechanism, each thread executes a chunk of iterations from the loop and then requests another, until no more iterations are left to be executed.

```
#pragma omp parallel private(j)
{
  for (j = 0; j < innerreps; j++) {
    #pragma omp for schedule(dynamic,cksz)
    for (i = 0; i < itersperthr * nthreads; i++) {
      delay(delaylength);
    }
  }
}
```
Listing 14: Benchmark source code that uses the dynamic schedule clause.

- ***Guided*** (Listings 15): Similar to ***dynamic***, in the sense that each thread executes a chunk and then requests another. However, the chunk size is computed differently, so that the chunk size is progressively reduced as we reach the end of the iteration space.

```
#pragma omp parallel private(j)
{
  for (j = 0; j < innerreps; j++) {
    #pragma omp for schedule(guided,cksz)
    for (i = 0; i < itersperthr * nthreads; i++) {
      delay(delaylength);
    }
  }
}
```

Listing 15: Benchmark source code that uses the guided schedule clause.

Ending with the **tasking constructs**:

- ***Parallel task generation*** (Listings 16): Each thread in the team will iterate through its own for loop and create a task that will execute the delay function.

```
#pragma omp parallel private( j )
{
  for ( j = 0; j < innerreps; j ++ ) {
    #pragma omp task {
      delay( delaylength );
    }
  }
}
```

Listing 16: Benchmark source code that creates tasks by all threads. Each task is created to execute the delay function.

- ***Serial task generation*** (Listings 17): In this example only one thread iterates though a loop, creating one task per iteration. Remaining threads wait at an implicit barrier.

```
#pragma omp parallel private(j)
{
  #pragma omp master
  {
    for (j = 0; j < innerreps * nthreads; j++) {
      #pragma omp task
      {
        delay(delaylength);
      }
    }
  }
}
```

Listing 17: Benchmark source code that creates tasks by one thread. Each task will execute the delay function.

- ***Task tree generation*** (Listings 18): Generation of tasks in parallel via recursive binary tree function.

```
#pragma omp parallel private(j)
{
  for (j = 0; j < (innerreps >> DEPTH); j++) {
    #pragma omp task
    {
      branchTaskTree(DEPTH);
      delay(delaylength);
    }
  }
}

void branchTaskTree(int tree_level) {
  if ( tree_level > 0 ) {
    #pragma omp task
    {
      branchTaskTree(tree_level - 1);
      branchTaskTree(tree_level - 1);
      delay(delaylength);
    }
  }
}
```

Listing 18: Benchmark source code that creates tasks by one thread. Each task will execute the delay function.

Regarding the timing model, and simply put, is defined as follows:

1. Get the reference time of the code region: t_{ref}
2. Get the time of the code region using the OpenMP directive: t_{omp}
3. Compute OpenMP overhead: $t_{overhead} = t_{omp} - t_{ref}$

The EPCC-OpenMP benchmark is used to measure the overhead of the selected set of OpenMP directives in **clock cycles**.

Software release

We provide the source code and workload configurations used to test each MEEP environment. This is available at MEEP Benchmarks webpage (EPCC-OpenMP table entry).

8.1.3. EPCC-OpenMP/MPI

Description

The EPCC-OpenMP/MPI benchmark measures the overhead for mixed-mode OpenMP/MPI programming [19]. Specifically, this benchmark provides a set of micro benchmarks for both point-to-point (for example, ping-pong, halo exchange among others) and collective communications (for example, gather, scatter among others).

Objectives

In MEEP we will focus on the following collective communications:

- ***MPI Gather*** & ***Scatter***: The MPI scatter mechanism can be viewed as the process of sending data from the root process to all processes in a set. MPI gather is quite the opposite, i.e., all processes in a set send data to one single process.
- ***MPI AlltoAll*** & ***AllReduce***: Involves the computation of data from all processes and instead of centralize the result in one process the results will be accessed to all processes.

- ***MPI Barrier***: This is a synchronization mechanism, which means that all processes must wait in a specific point until every process in the set reaches that point.

And for point-to-point communications, we report the following:

- ***Master-only, point-to-point communications***: MPI communication takes place in the master thread, outside of parallel regions.
- ***Master-only, halo exchange***: All MPI processes participate and the processes are arranged in a ring, where each process exchanges messages with its two neighbouring processes.

The metric used to measure the overhead of MPI communications is **clock cycles**.

Software release

We provide the source code and workload configurations used to test each MEEP environment. This is available at MEEP Benchmarks webpage (EPCC-OpenMP/MPI table entry).

8.2. HPC benchmarks

This section lists and describes the set of HPC benchmarks selected to analyze the performance of all available MEEP environments:

- RISC-V Benchmarks
- HPL - High-Performance Linpack
- HPCG - High Performance Conjugate Gradients
- FFTXlib
- CloudMicrophysics
- Advection-MPDATA

Similar to what was described in the System benchmarks section, to test these benchmarks on all MEEP environments which have specific features, we have created configuration files per environment and a set of make and run scripts to deal with all of this diversity and to have an automatic methodology to build and run them. Therefore, from an user perspective the steps needed to test any of the benchmarks on a specific platform are:

1. Compile the benchmark for the desired platform:
```
./make-meep-bench.sh <platform-name>
```
2. Configure the SLURM parameters for the desired run:
```
./configure-slurm <slurm-config>
```
3. Execute the benchmark for the desired platform:
```
./run-meep-bench.sh <platform-name>
```

In the end, the user will have all the results under the **output** folder, that will also include all the compilation information.

8.2.1. RISC-V Benchmarks

Description

The RISCV-V Benchmarks [17] provide a large set of kernels that can be used to test simple (also more complex) traditional HPC workloads. Moreover, this benchmark is targeted to run on RISC-V platforms, although it can also be run in other platforms, provided some additional configuration.

Objectives

In the context of the MEEP project, we are focused on the following set of kernels:

- **Axpy** - Performs a multiply and add operation of arrays. $y \leftarrow \alpha x + y$. A basic implementation of this multiply-add operation can be:

```
for (i=0; i<n; i++) {
        dy[i] += a*dx[i];
}
```

- **Gemm** - General matrix multiplication. $C \leftarrow \alpha AB + \beta C$. A basic implementation of the matrix multiplication can be:

```
for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++) {
                for (int k = 0; k < K; k++) {
                        c[i][j] += a[i][k] * b[k][j];
                }
        }
}
```

  This kernel generally represents a compute bound problem.

- **SpMv** - Sparse matrix-vector multiplication operation. $y \leftarrow Ax$:

```
for (row=0; row<nrows; row++) {
        elem_t sum = 0.0;
        for (idx=ia[row]; idx<ia[row+1]; idx++) {
                sum += a[idx] * x[ja[idx]];
        }
        y[row] = sum;
}
```

- **Somier** - Is a kernel inspired in the old bed base model composed of a mesh of springs. For each point of the mesh this kernel computes the position, acceleration and velocity of this 3D structure. Exemplification of the nature of the computations:

```
for(i = 0; i<n; i++) {
        for(j = 0; j<n; j++) {
                for(k = 0; k<n; k++) {
                        V[0][i][j][k] += A[0][i][j][k]*dt;
                        V[1][i][j][k] += A[1][i][j][k]*dt;
```

```
                              V[2][i][j][k] += A[2][i][j][k]*dt;
                    }
          }
}
```

- **FFT** - This kernel uses the FFTW, a C subroutine library for computing the discrete Fourier transform (DFT) [22]. This is the most complex kernel as it integrates the FFTW library to compute the Fourier Transform.

The set of selected kernels serve the purpose of analysing the behaviour of simple memory- and compute-bound kernels running on all the available MEEP environments. Specifically, we are focusing on multi-thread and vector instruction performance. To this end we are primarily targeting the **OPC** metric in the context of the vector performance methodology.

Contributions

The RISC-V Benchmarks were developed in the context of the EPI project. The MEEP project also contributed to the development of this repository of kernels, specifically:

- Establish a standard and common infrastructure to develop, build and run a kernel;
- Provide support for BLAS libraries for a small subset of kernels: Axpy and GEMM;
- Provide a new version for a subset of kernels that uses vector instructions based on the OpenMP simd construct: Axpy, Gemm, Somier, SpMv;
- Additionally, we also put forward a ”Baremetal” versions of a subset of kernels.

Software release

The source code and workload configurations used to test each MEEP environment are available at MEEP Benchmarks webpage (RISC-V Benchmarks table entry).

8.2.2. HPL

Description

HPL (High-Performance Linpack) is a portable implementation of the High-Performance Linpack benchmark and it solves a linear system of order N: $Ax = b$ by first computing the LU factorization [8]. It is written in C and requires an MPI and BLAS implementation.

Objectives

This benchmark is utilized to analyse the behaviour of more complex application patterns in the available MEEP environments. Specifically, we are focusing on multi-thread (supported by OpenMP) and also multi-node execution (supported on MPI) for two different types of compilation setups:

- scalar: where vectorization is disabled and therefore only scalar instructions are executed.
- vector: where auto-vectorization is enabled and therefore scalar and also vector instructions

are executed.

Regarding the performance analysis, we will evaluate the behavior of this benchmark using the **OPC** metric in the context of the vector analysis methodology.

Software release

The source code and workload configurations used to test each MEEP environment are available at MEEP Benchmarks webpage (HPL table entry).

8.2.3. HPCG

Description

HPCG [26] is a software package that performs a fixed number of multigrid preconditioned (using a symmetric Gauss-Seidel smoother) conjugate gradient (PCG) iterations using double precision (64 bit) floating point values. HPCG is intended as a complement to the High Performance LINPACK (HPL) benchmark (Section 8.2.2), currently used to rank the TOP500 computing systems. The computational and data access patterns of HPL are still representative of some important scalable applications, but not all. HPCG is designed to exercise computational and data access patterns that more closely match a different and broad set of important applications, and to give incentive to computer system designers to invest in capabilities that will have impact on the collective performance of these applications.

HPCG is a complete, stand-alone code that measures the performance of basic operations in a unified code:

- Sparse matrix-vector multiplication.
- Vector updates.
- Global dot products.
- Local symmetric Gauss-Seidel smoother.
- Sparse triangular solve (as part of the Gauss-Seidel smoother).
- Driven by multigrid preconditioned conjugate gradient algorithm that exercises the key kernels on a nested set of coarse grids.
- Reference implementation is written in C++ with MPI and OpenMP support.

Objectives

In the context of MEEP, we will evaluate the performance of the reference HPCG implementation in terms of ***OPC*** for the different kernels of the code (as provided by the output of the software itself), in different platforms. We will consider both OpenMP (for single-node executions) and MPI (for multi-node executions) versions of the code and will evaluate its scalability for two different compilation setups:

- scalar: where vectorization is disabled.

- vector: where auto-vectorization is enabled.

Software release

The source code and workload configurations used to test each MEEP environment are available at MEEP Benchmarks webpage (HPCG table entry).

8.2.4. FFTXlib

Description

FFTXlib is mainly a rewrite and optimization of earlier versions of FFT related routines inside Quantum ESPRESSO (QE) pre-v6; and finally their replacement. Despite many similarities, current version of FFTXlib dramatically changes the FFT strategy in the parallel execution, from 1D+2D FFT performed in QE pre v6 to a 1D+1D+1D one; to allow for greater flexibility in parallelization.

FFTXlib module is a collection of driver routines that allows the user to perform complex 3D Fast Fourier Transform (FFT) in the context of plane wave based electronic structure software. It contains routines to initialize the array structures, to calculate the desired grid shapes. It imposes underlying size assumptions and provides correspondence maps for indices between the two transform domains.

Once this data structure is constructed, forward or inverse in-place FFT can be performed. For this purpose FFTXlib can either use a local copy of an earlier version of FFTW (a commonly used open source FFT library), or it can also serve as a wrapper to external FFT libraries via conditional compilation using pre-processor directives. It supports both MPI and OpenMP parallelisation technologies.

FFTXlib is currently employed within Quantum Espresso package [6], a widely used suite of codes for electronic structure calculations and materials modeling in the nanoscale, based on planewave and pseudopotentials.

Objectives

In the context of MEEP, we will evaluate the performance of this benchmark in terms of execution time for the different kernels of the code (as provided by the output of the software itself), in different platforms. We will consider the OpenMP version of the code and will evaluate its scalability (when possible) for two different compilation setups:

- scalar: where vectorization is disabled.
- vector: where auto-vectorization is enabled.

Software release

The source code and workload configurations used to test each MEEP environment are available at MEEP Benchmarks webpage (FFTXlib table entry).

8.2.5. CloudMicrophysics

Description

CloudMicrophysics refers to an application developed in the context of the ECMWF Escape project called Cloud microphysics scheme (for more information we refer the reader to document D1.1 Batch 1: Definition of several Weather & Climate Dwarfs [23]). Simply put, CloudMicrophysics computes the cloud and precipitation processes that are present in the the IFS model.

Objectives

The original source code provides a set of versions, however in MEEP we focus on the Standalone C version to test the performance of the available platforms. Furthermore, this application is also used as a co-design tool to test the autovectorization capabilities of the compiler used in MEEP (LLVM-EPI).

In terms of performance analysis, CloudMicrophysics is used to explore the vector performance of a more mature application on MEEP environments that contain hardware supporting this feature (execution of vector instructions). To this end, we will mostly focus on the **OPC** metric provided by the vector analysis methodology.

Modifications

Given that CloudMicrophysics provides an entire infrastructure to build and run different version, and aiming to reduce the complexity of this miniApp, we have removed all of the parts that were not related with the C version of this code. Moreover, we removed the dependencies on the build tools (Ecbuild and CMake) and have written a Makefile to have a more flexible and easy way to build and run different configurations of this application. This also allowed us to build and deploy CloudMicrophysics on SDV computing platforms that are characterized by the lack of the common infrastructure that we have in production platforms. Focusing on the relevant kernels, we have modified the source code to explicitly vectorize most parts of the code. These modifications entail the addition at the beginning of for loops of vectorization directives such as ***pragma pragma omp simd*** or ***pragma clang loop vectorize(enable)***.

Contributions

We have detected a bug in the C version that was reported to the maintainers with the suggestion for a correction (bug report [31]).

8.2.6. Advection-MPDATA

Description

Advection-MPDATA also refers to another application developed in the context of the ECMWF Escape project called MPDATA (multidimensional positive definite advection transport algorithm) for unstructured meshes (for more information we refer the reader to document D1.1 Batch 1: Definition of several Weather & Climate Dwarfs [23]). Its purpose is to solve the PDEs modelling the advection on a sphere using an unstructured mesh with the MPDATA algorithm.

Objectives

Advection-MPDATA is used to understand the performance of running a more complex application in the available MEEP environments. Given that this application uses MPI and OpenMP, we will take advantage of Advection-MPDATA to inspect the multi-thread and multi-node performance of this application. Additionally, we will also explore the vector performance of some parts of the code that have been translated to C and therefore take advantage of compiler autovectorization infrastructure.

Modifications

This application is written in Fortran and compared with CloudMicrophysics requires a more complex infrastructure to build and run this application. However, we did not reduced this complexity and instead ported some of the most time consuming functions to C. Specifically, we focused on two distinct functions: ***compute_fluxzdiv*** which is the most time consuming and ***limit_scalar_flux*** that overall represents a bigger computational loop. These modifications will allow us to take advantage of the autovectorization capabilities of the LLVM-EPI compiler as well as the VPU hardware of MEEP.

8.3. Data Analytics benchmarks

This section describes the benchmarks used to explore common data analytics performance on MEEP environments. We use two data analytics runtimes:

- TensorFlow is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks. Due to the build system did not support for RISC-V on MEEP we employ TensorFlow Lite framework. TensorFlow Lite only provides inference and it is designed focusing on edge environments. As earlier explained we have enabled the runtime on our RISC-V platforms. TensorFlow has become commodity for training models. In the context of MEEP project we will use the three main neural networks from which most works derive: MobileNet, ResNet50 and VGG-19. Additionally we use MNIST as a functionality checker, as it has become the ”hello world” for deep learning.

- Apache Spark is a multi-language engine for executing data engineering, data science and machine learning on single-node machines or clusters. It allows for either batch or streaming data with languages such Python, SQL, Scala, R or Java. Its main feature is that it brings data closer to the place where it is computed: i.e, moves data from disk to memory for faster processing. Thus increasing the performance over traditional big data. It has become a commodity technology for data mining and machine learning.

8.3.1. TensorFlow Lite models

Description

Given that TensorFlow lite does only inference, we use pre-trained models. Over the trained model we use a synthetic benchmark to assess inference timings over the model.

The models are a set of neural networks representative of current data analytics architectures. The selected networks are:

- MNIST [29]: its input is a set of 10 hand-written numbers from 0 to 9. It identifies the corresponding hand-written number. It is widely used as a hello world for deep learning.
- VGG-19 [36]: VGG19 is a variant of VGG model, which in short consists of 19 layers (16 convolution layers, 3 Fully connected layers, 5 MaxPool layers, and 1 SoftMax layer). There are other variants of VGG like VGG11, VGG16, and others. VGG19 has 19.6×10^9 FLOPs.
- ResNet50: ResNet50 is a variant of the ResNet [24] model, which has 48 Convolution layers along with 1 MaxPool and 1 Average Pool layer. It has 3.8×10^9 FLOPS. It is a widely used ResNet model.
- MobileNet [25]: the MobileNet model is based on depthwise separable convolutions, which is a form of factorized convolutions that factorize a standard convolution into a depthwise convolution and a 1×1 convolution called a pointwise convolution.

We train each of the selected models, and we get a pre-trained graph for each of the models. From there, the benchmark consists of taking an input graph and an input image. Then, it runs inference over 50 iterations (parametrizable) and outputs the average inference time and standard deviation, as well as the fastest and longest inference timings.

Objectives

The benchmark is highly parametrizable in all its components. The parameters we will explore for the benchmark are:

- Input images: all images that will be inferred on each iteration of the benchmark.
- Use xnnpack (boolean): whether or not to enable XNNPACK [7] algebraic optimizations. Highly benefits performance if vectorial instructions are enabled.
- Num threads: number of threads to be used for TFLite
- Allow fp16 (boolean): whether or not to enable FP16 operations. It will be used depending on what architectures provide.
- Num runs: the number of iterations to be done for each input. Default: 50.
- Graph: the model to be used for inference.
- Dry run: if true, does an execution loading the model and allocating tensors but without any computation (i.e., inference) performed.
- Warmup parameters: parameters to define how many inference iterations (or amount of seconds) to do before running the actual benchmark.
- Use caching (boolean): to enable or not the use of cache.
- Run frequency: to execute at a given frequency rather than a given delay. By default, the benchmark waits a pre-set time between inferences. However, we can define target frames

per second. If not possible, the benchmark will initiate the next iteration without waiting for the completion of the previous one, doing its best to catch up.

- Run delay: delay seconds between inference iterations.
- Max seconds: maximum seconds for the benchmark to complete. If exceeding mid-iteration, the benchmark will complete the iteration but will stop afterward. Default: 150s.
- Min seconds: minimum seconds to re-iterate for. Possible to make the number of inference iterations done higher than the set.

The general metric to be used will be frames per cycle. Generally speaking, frames per second are the most used measure of performance. However in the MEEP project we have several platforms with very different clock frequencies and characteristics. Consequently a better measure is to translate seconds into cycles.

Software release

The pre-trained models are offered as an additional RPM package which can be found at MEEP Data Analytics Benchmarks webpage.

8.3.2. Spark Epistasis use case

Description

Epistasis is the interaction between genes that influences a phenotype. Genes can either mask each other so that one is considered "dominant," or they can combine to produce a new trait. It is the conditional relationship between two genes that can determine a single phenotype of some traits.

An HPC application has been developed to find all these interactions. The application uses Apache Spark to move the data from disk to memory. Since genome data is massive, the genome is split into smaller partitions. Once each of the partitions is moved in memory by Spark, the application leverages numpy to make the computational part.

Objectives

The objective of the workload is to compute as many variations as possible in the lesser time possible. Consequently our base metric will be cycles. Instead of using the traditional seconds, as we have said earlier, it is a better measure when comparing with very different clock frequencies among the different MEEP environments.

The parameters we can tune by the workload are:

- **Vectorial vs non-vectorial**: performance x86 with vectorial instructions vs without vectorial instructions (useful to compare with RISC-V platforms).
- **Number of nodes**: limited number on arriesgado due to availability. We can't do multi-node testing on SDV or ACME-EA v0 (unless more nodes available with Ethernet communication)
- **Number of Patients**: use different cohorts with different amounts of patients.

- **Partition sizes**: dataset partition sizes. The number of samples that are processed at the same time.
- **Cross Validation sets**: use a 5-fold CV or a 10-fold CV.
- **Network usage**: relevant in the case of multi-node runs.

Software release

The Epistasis use-case RPM can be found at MEEP Data Analytics Benchmarks webpage.

8.4. Workflows benchmarks

One part of the MEEP Software Stack is devoted to the development and orchestration of parallel and distributed workflows with COMPSs. In this section, we present a set of Workflows implemented with PyCOMPSs (the Python binding of COMPSs) which could take benefit of MEEP capabilities. In the first part of the section, we present a set of dislib algorithms which implement distributed workflows for ML. In the second part of the section, we present another workflow use case which is focused on Hyper-Dimentional Computing.

The Software release of these workflows can be found in the MEEP Workflows Benchmarks webpage

8.4.1. Dislib Algorithms

Description

The Distributed Computing Library is a machine learning library that is built on top of PyCOMPSs, thus provides machine learning algorithms that are distributed and parallel. The library focuses on the execution of data analysis algorithms on distributed platforms such as supercomputers.

QR Decomposition QR decomposition is the decomposition of a matrix into a QR product. A more formal description would be the following: *Let A be a $m \times n$ matrix where $m > n$, this can be decomposed in a product of an orthogonal matrix Q (a real square matrix that all of its rows and columns are orthonormal vectors) and an upper triangular matrix R.*

There are several algorithms that perform such decomposition, which is usually used for calculating the linear least squares. The most common algorithms are the Gram-Schmidt process, Householder-Transformations, and several modifications of the past two. In dislib, the algorithm that is used is the Householder-Transformations with a block factorization.

Matrix Multiplication (MATMUL) Matrix multiplication is a classic algorithm that consists of multiplying two matrices. Even though this may seem an easy problem it needs a high computational power when the sizes of the matrices increase, the asymptotic cost is $O(n^3)$.

To parallelize the application, dislib approaches the problem by dividing the multiplications matrices into smaller ones, this way parallelizing the computations.

Cascade Support Vector Machine (CSVM) This is an algorithm that is used for classifying data in a supervised environment. As opposed to linear regression, the SVM can perform an efficient classification of non-linear data, using a kernel trick.

In the case of parallel environments, the SVM algorithm can be adapted to a CSVM. This is the case of dislib's CSVM, which is based on Graf et al. implementation. This implementation creates a cascade-like structure that will allow the algorithm to be parallelized. Therefore the algorithm will break the datasets into N sets that will be trained separately and then will be merged in pairs to compute the support vectors, this is going to be done iteratively until a single set emerges.

KMeans This algorithm belongs to the clustering algorithms family. When using this algorithm the user has to have an idea of how many clusters will the dataset have, since one of the algorithm parameters is the number of clusters (K). This algorithm usually starts with K (the number of defined clusters by the user) randomly set centers in the space. Then it assigns to every data point a single center based on the closest distance to it. After assigning the points each center is recalculated to be set in the new center for all the points assigned. Then this process is done iteratively until the centers converge (are not updated from their position).

The parallel version is performed by creating one task for every row in a block of the ds-array. Then reduction is performed, which adds all the data points that belong to a center.

Gaussian Mixture Model (GMM) This algorithm is used to represent the distribution of a series of data as the sum of several Gaussian components. Those are assumed to be generated from a mixture of Gaussian distributions. The goal of this algorithm is to maximize the likelihood of the model that is generated to describe the data. Similar to KMeans, this is a clustering algorithm. However, in this case, there is no need to define the number of clusters, so it can be considered unsupervised learning.

Random Forest Classifier (RF) This algorithm is used to classify data in different classes. It creates a series of decision trees that will use to aggregate their predictions. The use of the random forest instead of a single decision tree classifier is helpful since most applications will have a higher prediction when aggregating the predictions compared to when a single one is used.

Objectives

The objective has been to check the distribution. Moreover, we wanted to evaluate in MEEP the kernel acceleration using vectorized mathematical libraries with single and multicore and comparing to executing with scalar instructions. Furthermore, another main objective has been to execute using multiple worker nodes and see the performance of MEEP when using multiple nodes to execute the application.

The performance analysis has been done by comparing the execution times of the applications using different mathematical acceleration libraries (different versions of BLIS). Additionally, using

Extrae we have also evaluated the parallelism and execution times of different parts in the application functions.

Modifications

To run in the ACME EA platform, no code modifications are required. However to get benefit of the acceleration provided by the platform, users have to use a Numpy version which is able to use the accelerated BLIS. To do it, the user has to compile Numpy specifying the BLIS library location as explained in Section 5.7 and add the Numpy installation location to the PYTHONPATH environment variable.

8.4.2. Hyper-Dimensional Computing (HDC)

Description

Hyper-Dimensional Computing also known as Vector Symbolic Architecture, is a computing framework that tries to emulate the animal nervous system. It does so by representing the space using the properties of high-dimensional random vectors. The higher level idea is to represent information $x \in \mathcal{X}$ by protecting it to the $\mathcal{X}$ hyperspace with d dimensions, usually those being 10.000. The space is usually represented as binary $\mathcal{H} = \{0, 1\}^d$ or bipolar $\mathcal{H} = \{-1, 1\}^d$. One essential part of HDC is encoding the data, therefore it is needed to create a mapping from the data space, to the hyperspace $\phi : \mathcal{X} \to \mathcal{H}$. The encoding has the property that vectors are holographic. In itself this means that the dimensions of the hypervectors are independent and identically distributed, this allows the hypervectors to be robust and each dimension carries the same amount of information.

Beijing $PM_{2.5}$ pollution This application tries to assess the pollution in Bejing by learning from different features from the dataset [1]. From these features, metrics such as pressure, wind direction, wind speed, accumulated snow, accumulated rain, dew point, $PM_{2.5}$, can be extracted, additionally, there is time data, which is an hour, day, and month. For this learning task, the aim is to be able to predict the temperature.

This application is used to show the different prediction accuracy depending on the basis hypervector used for encoding the data. The aim is to show that using circular hypervectors will have a better accuracy result since there is the time data that can be represented using circular data.

Objectives

The objective has been to check the distribution. Moreover, we wanted to evaluate in MEEP the kernel acceleration using vectorized mathematical libraries with single and multicore and comparing to executing with scalar instructions. Furthermore, another main objective has been to execute using multiple worker nodes and see the performance of MEEP when using multiple nodes to execute the application.

The performance analysis has been done by comparing the execution times of the applications using different mathematical acceleration libraries (different versions of BLIS). Additionally, using

[1] `https://archive.ics.uci.edu/ml/datasets/Beijing+PM2.5+Data`

Extrae we have also evaluated the parallelism and execution times of different parts in the application functions.

Contributions

This HDC application has been ported to the COMPSs framework, therefore parallelizing it. Apart from that, to get benefit of the acceleration provided by MEEP, the user has to use a Numpy version which is able to use the accelerated BLIS. To do it, the user has to first install Numpy specifying the BLIS library location as explained in Section 5.7 and add the installation location to the PYTHONPATH environment variable.

8.5. Systolic Array benchmarks

In *D5.1 Benchmark suite of HPC applications* we also included the MLPerf and the Bolt66-App as part of the benchmarks suite. These applications aim to functionally validate the implementation of the two MEEP Systolic Arrays, as well as to evaluate their performance.

We will present the application porting as well as its evaluation in the upcoming deliverable *D5.4 Final release of the software stacked* (M42). The application porting will leverage the custom instructions presented in Section 4.3 and detailed in Appendix A.

9. The MEEP Offload Mode

In this section we present the Offload Mode of the MEEP accelerator, and the support required for single and multiple devices.

9.1. Single-device support

We have implemented a prototype infrastructure supporting OpenMP offload between the Intel Host, acting as the application runner, and the RISC-V on the FPGA, acting as the device accelerator.

In this section we present the compiler and runtime infrastructure that we have implemented for this purpose. It is based on a previous implementation done by FORTH (Crete, Greece), in the context of the EPI project, that we have adapted to the MEEP environment.

9.1.1. Compiler support for MEEP offload

The support for the MEEP offload uses the code generated from the LLVM compiler. For the MEEP offload, the compiler is invoked to compile for an Intel Host and a RISC-V target device. It has been configured to generate x86_64 code for the Host and RISC-V rv64imafdc code for the RISC-V as an accelerator.

When configured, the compiler generates code for the OpenMP directives in such a way that the regular code runs in the Intel architecture, and the code annotated with the *target* directive is spawned onto the RISC-V accelerator. LLVM is able to encapsulate the RISC-V binary as a *target* section in the host binary. The management of the RISC-V code is left to the *libomptarget* LLVM-OpenMP support library.

9.1.2. Runtime support for MEEP offload

The support for OpenMP *target* on the Host side is implemented as a plugin to the *libomptarget* library. In our case, we have adapted the plugin developed by FORTH in the EPI project to work with the RISC-V accelerator on the FPGA.

Currently, the support that has been implemented on the prototype covers these services:

- Support the Host side communications with the target device.
- Check and transfer the RISC-V binary to the RISC-V environment.
- Determine the number of devices available. We currently support a single one.
- Initialization and finalization of the application on the device side.
- Allocating and deallocating data areas for the device.

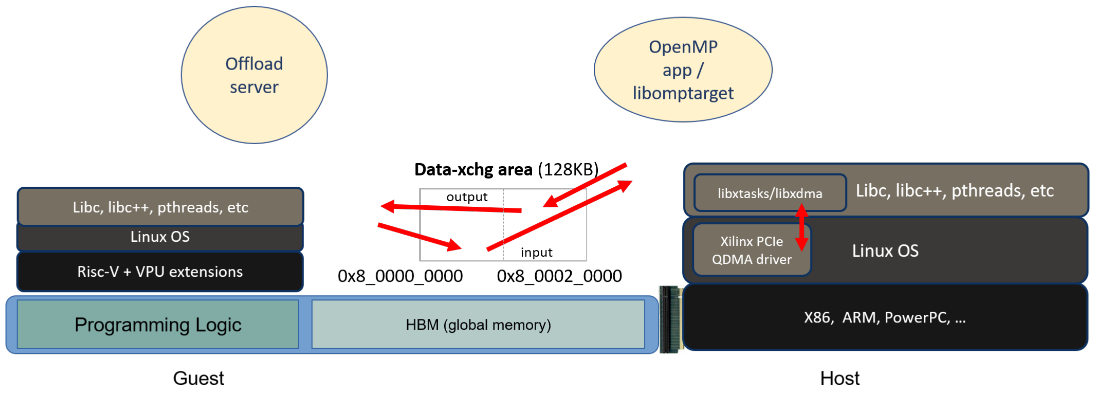


Figure 5: Diagram of the MEEP Offload infrastructure

- Transfer data into the device, and out of the device.
- Manage the *target* regions of code to run code onto the device.

The current implementation is missing the following functionalities:

- Manage the *target teams* regions.
- Asynchronous management of data transfers.
- Asynchronous management of code regions.

Figure 5 shows the way communications are implemented between the Host and the RISC-V.

The communications are implemented through a specific shared memory area laid out between the Host and the RISC-V (Data-xchg). This shared memory region is implemented either as a BlockRAM in the FPGA chip, or as part of the DRAM/HBM memory on the FPGA board. This shared memory area is accessed using the QDMA driver services for DMA transfers. In the current implementation the DMA transfers issued through the QDMA driver are not interacting with the Openpiton memory hierarchy on the RISC-V side. Any data that is actually on the processors cache memory is not invalidated when a DMA transfer happens from the Host side. For this reason, the prototype cannot use the full range of HBM memory to support large data structures. The shared memory area is implemented in the I/O space, and thus is not cached by Openpiton in the cache hierarchy. We have implemented this infrastructure as a demonstrator prototype, not full-fledged, so no additional copies are done between the reduced-size memory in the I/O space, and the Offload server memory. This reduces the size of the application data to only a few tens of Kbytes.

The shared memory area is structured as a small descriptor containing the identifier of each service requested, and a specific *slot* area that is structured according to each service requested. The description of the services is presented in section 9.1.3.

9.1.3. RISC-V side offload support

On the RISC-V side we run a server that waits for requests for *target* offload commands from the host.

When transferring a binary file, it is temporarily stored in the *slot* area, saved in the local file system on the RISC-V side, and then loaded into the server application space.

The *target* binary has a specific *entries* section, that is used to find the symbols that can be invoked as OpenMP *target* regions. This section is used every time that a new *target* region is invoked from the host, through the *offload target region service*.

Before invoking a *target* region, the Host code allocates the data regions needed, and copies the input data for the variables marked as *to*.

The *offload target region service* receives a function identifier and its arguments. The server uses the services of the foreign-function interface (*libffi*) in order to invoke the proper function implementing the requested region, with its associated arguments.

After the execution of a region, the Host code transfers the data out for those variables marked as *from*.

9.1.4. Testing

We have tested the prototype infrastructure with a pair of examples:

- A test sample showing the ability to initialize and copy data.
- A matrix multiplication example.

With these tests we demonstrated the feasibility to implement the OpenMP offload services on the MEEP environment.

9.1.5. BLIS single-device approach

The offload execution mode of BLIS differs from the self-hosted mode in the sense that when the host launches the application and we have a call to a BLIS service, the respective workload is offloaded to the available accelerators in the computing platform. Specifically, we offload and distribute the workload in two different forms: 1) The work performed from BLIS is offloaded to only one accelerator (single-device) or 2) the workload is distributed among a set of devices (multi-device approach).

The BLIS single-device approach exploration was developed using a computing platform that is characterized by a host (CPU) and a set of connected accelerators (GPUs). Moreover, to offload computations from the host to the accelerator we relied on the OpenMP programming model, specifically the **omp target** construct.

Next, we highlight the nature of adaptations that are needed for an application to take advantage of the BLIS offload version. Thus, we have to make two clear distinctions: first, the modification in the application side that the end user has to perform and second, the modifications on the BLIS library. From the application side, the user should be responsible for creating a data shared environment using the **#pragma omp target data map** directive (see Listing 19). This will allow the re-utilization of data on the accelerator and no extra communications between host and accelerator are needed, because we are working with data that lives in the device (with the exception of synchronization that might need to occur if the host updates variable that are in the device).

```
int main(){
  // Initalization phase ...
  init(x,y);
  // 1st set of computations ...
  // Offload BLAS computations to accelerator
  #pragma omp target data map(to:x[0:n]) map(tofrom:y[0:n])
  {
    // 1st call to a BLIS routine
    cblas_daxpy(n, scalar, x, 1, y, 1);
    // 2nd call of a BLIS routine
    double dot_product = cblas_ddot(n,x,1,y,1);
    // More BLIS routine calls ...
  }
  // More computations ...
  // Final ...
}
```

Listing 19: Example of an application that defines a data shared environment where a set of BLIS routines are called.

The modifications needed in the BLIS library share similarities with the application side, specifically, here we also have to create a data shared environment. The additional step is the inclusion of the ***#pragma omp target teams distribute parallel for*** (see Listing 20) directive before the for loops so that the workload is actually divided and computed by the accelerator.

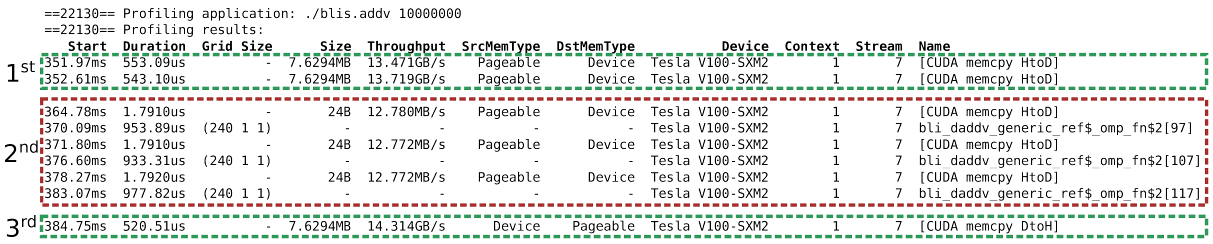
```
void PASTEMAC3(ch,opname,arch,suf)(conj_t conjx,dim_t n,
                ctype* restrict x,inc_t incx,ctype* restrict y,
                inc_t incy,cntx_t* restrict cntx)
{
  // ...
  _Pragma("omp target data map(to:x[0:n]) map(tofrom:y[0:n])")
  {
    ctype* restrict chi1 = x;
    ctype* restrict psi1 = y;

    if ( bli_is_conj( conjx ) ) {
      if ( incx == 1 && incy == 1 ) {
        _Pragma("omp target teams distribute parallel for")
        for ( dim_t i = 0; i < n; ++i ) {
         PASTEMAC(ch,addjs)( chi1[i], psi1[i] );
        }
    }
    else {
      // ...
    }
  }
}
```

Listing 20: Example of the nature of modifications done in the BLIS kernels.

Figure 6 shows a timeline of an application that calls three BLAS kernels (in this example the vector addition). We can decompose this figure in three phases:

- In the first phase we have the transfer of data from host to device that corresponds to the two arrays (x and y) that we are updating (shown in the figure as the top dashed-green box).
- The second phase (middle dashed-red box) demonstrates the call of the three BLAS level 1 kernels.
- The third phase (bottom dashed-green box), shows the retrieving of the data after all computations have been finished in the accelerator.



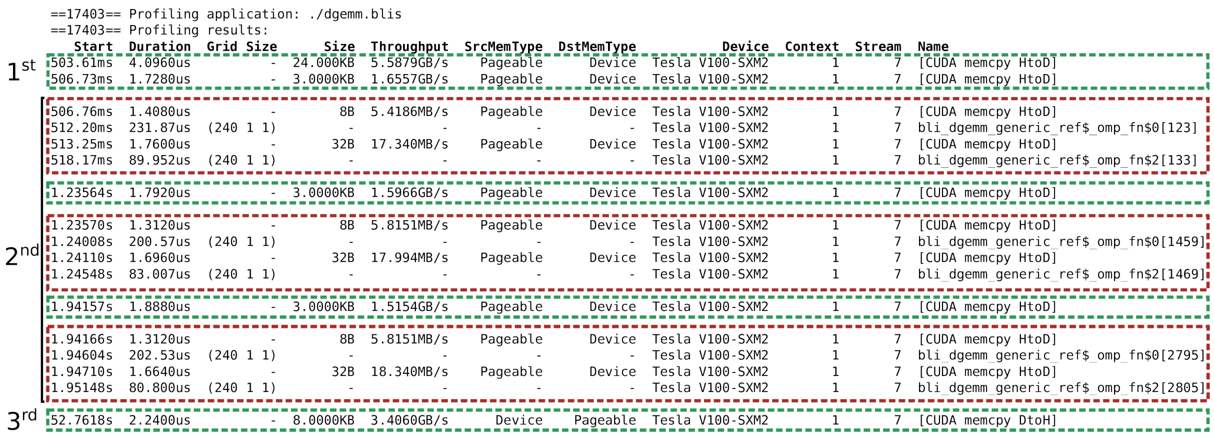
```
==22130== Profiling application: ./blis.addv 10000000
==22130== Profiling results:
   Start  Duration  Grid Size       Size  Throughput  SrcMemType  DstMemType           Device  Context  Stream  Name
351.97ms  553.09us          -  7.6294MB  13.471GB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
352.61ms  543.10us          -  7.6294MB  13.719GB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
364.78ms  1.7910us          -       24B  12.780MB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
370.09ms  953.89us  (240 1 1)         -           -           -           -  Tesla V100-SXM2        1       7  bli_daddv_generic_ref$_omp_fn$2[97]
371.80ms  1.7910us          -       24B  12.772MB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
376.60ms  933.31us  (240 1 1)         -           -           -           -  Tesla V100-SXM2        1       7  bli_daddv_generic_ref$_omp_fn$2[107]
378.27ms  1.7920us          -       24B  12.772MB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
383.07ms  977.82us  (240 1 1)         -           -           -           -  Tesla V100-SXM2        1       7  bli_daddv_generic_ref$_omp_fn$2[117]
384.75ms  520.51us          -  7.6294MB  14.314GB/s      Device    Pageable  Tesla V100-SXM2        1       7  [CUDA memcpy DtoH]
```

Figure 6: Time profile of an application that calls three times the BLAS vector addition kernel, using Nvidia's profiler tool. Time dimension is read in each row, while the type of computations can be seen in the last column (column called ***Name*** with attributes such as ***CUDA memcpy HtoD***, ***bli_daddv_generic_ref***, among others).

For BLAS levels 1 and 2, this analysis demonstrates that we have a successful offload and re-utilization of data, as we do not see any memory operations in between the BLAS kernel executions. However, for BLAS level 3 kernels, we do not achieve the same behaviour, therefore, we are not able to apply the same methodology. Yes, it is true that we have a successful offload of the BLIS routines to the accelerator, but we are not able to reuse data. Hence, we end up with extra communications in between host and device, when calling consecutive BLAS level 3 kernels. This behaviour is shown in Figure 7, where we can report the following:

- First phase with the transfer of data from host to the accelerator (top dashed-green box).
- In the next phase we can group the three calls to the matrix multiplication BLAS kernel (red-dashed boxes). Here, we can observe that in between BLIS kernel calls there are two sets of data transfers from the host to the accelerator (dashed-green boxes). It is true that during the entire execution of each BLIS kernel we have data re-use, altough in between kernel calls we observe data transfers and if data was being reused, these data transfers between the kernels calls should not be present. This should be true because we already transferred this data in the beginning of the computations and all kernels work over the same data. This behaviour might be explained by the fact that the OpenMP runtime is not able to detect that new allocations for this data are done by the BLIS infrastructure.
- The last phase is characterized by the retrieving of the resultant matrix to the host side (bottom dashed-green box).

In summary, we achieve the desired behaviour for BLAS 1 and 2 levels, but there is no re-utilization of data for the third BLAS level and thus a different approach should be devised to achieve a re-utilization of data in the accelerators.



```
==17403== Profiling application: ./dgemm.blis
==17403== Profiling results:
   Start  Duration  Grid Size      Size  Throughput  SrcMemType  DstMemType          Device  Context  Stream  Name
503.61ms  4.0960us          -  24.000KB  5.5879GB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
506.73ms  1.7280us          -  3.0000KB  1.6557GB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
506.76ms  1.4080us          -        8B  5.4186MB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
512.20ms  231.87us  (240 1 1)         -           -           -           -  Tesla V100-SXM2        1       7  bli_dgemm_generic_ref$_omp_fn$0[123]
513.25ms  1.7600us          -       32B  17.340MB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
518.17ms  89.952us  (240 1 1)         -           -           -           -  Tesla V100-SXM2        1       7  bli_dgemm_generic_ref$_omp_fn$2[133]
1.23564s  1.7920us          -  3.0000KB  1.5966GB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
1.23570s  1.3120us          -        8B  5.8151MB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
1.24008s  200.57us  (240 1 1)         -           -           -           -  Tesla V100-SXM2        1       7  bli_dgemm_generic_ref$_omp_fn$0[1459]
1.24110s  1.6960us          -       32B  17.994MB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
1.24548s  83.007us  (240 1 1)         -           -           -           -  Tesla V100-SXM2        1       7  bli_dgemm_generic_ref$_omp_fn$2[1469]
1.94157s  1.8880us          -  3.0000KB  1.5154GB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
1.94166s  1.3120us          -        8B  5.8151MB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
1.94604s  202.53us  (240 1 1)         -           -           -           -  Tesla V100-SXM2        1       7  bli_dgemm_generic_ref$_omp_fn$0[2795]
1.94710s  1.6640us          -       32B  18.340MB/s    Pageable      Device  Tesla V100-SXM2        1       7  [CUDA memcpy HtoD]
1.95148s  80.800us  (240 1 1)         -           -           -           -  Tesla V100-SXM2        1       7  bli_dgemm_generic_ref$_omp_fn$2[2805]
52.7618s  2.2400us          -  8.0000KB  3.4060GB/s      Device    Pageable  Tesla V100-SXM2        1       7  [CUDA memcpy DtoH]
```

Figure 7: Time profile of an application that calls three times the BLAS vector addition kernel, using Nvidia's profiler tool. Time dimension is read in each row, while the type of computations can be seen in the last column (column called ***Name*** with attributes such as ***CUDA memcpy HtoD***, ***bli_dgemm_generic_ref***, among others).

9.2. Multi-device support

One of the possible scenarios considered earlier in the MEEP project was that a single node could offer many accelerators where work could be offload to. This led us to identify a gap in OpenMP support for offloading.

OpenMP has offloading support since version 4.0. However, the interface offered by OpenMP only allows offloading to a single device at a time. Under a context of a host node with many, regular, accelerators, OpenMP does not offer convenient syntax for this use case.

9.2.1. OpenMP extensions

We proposed an extension to OpenMP in which we introduce a new OpenMP construct called `target spread`. Instead of receiving a single `device` clause, the spread construct has a `devices` clause which represents the set of devices that will execute the offloaded region. Rather than choosing a design similar to that of the OpenMP parallel construct, where execution would be replicated among devices, we chose to constraint `target spread` to OpenMP loops.

Constraining ourselves to loops allows us to introduce two special values, `omp_spread_start` and `omp_spread_size` which represent the set of iterations that a device executes. Our initial implementation focused on a static scheduling approach: the iterations are divided among devices using a chunk size that can be specified by the user. Listing 21 shows an example of the `target spread` construct applied to a SAXPY kernel.

```
void saxpy_multi_dev(int n, float a, float *x, float *y) {
  #pragma omp target spread \
    devices(0, 1, 2, 3) \
    spread_schedule(static, 1024) \
    map(to: a[omp_spread_start:omp_spread_size]) \
    map(tofrom: y[omp_spread_start:omp_spread_size])
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}
```
Listing 21: Example of the `target spread` construct applied to a simple SAXPY kernel.

One limitation of centering the extension around loops is that data transfers in OpenMP can be defined at arbitrary parts of the code. We generalised the data transfer constructs into spread versions. The multiple devices context was defined in a new clause called `range`. Listing 22 shows the spread version of data transfer constructs OpenMP. Listing 23 shows that the same syntax can be used in the scoped `target data` construct.

```
#pragma omp target enter data spread \
  devices(2,0,1) \
  range(1:N-2) \
  chunk_size(4) \
  nowait \
  map(to:A[omp_spread_start-1:omp_spread_size+2], \
  B[omp_spread_start :omp_spread_size ])

#pragma omp target exit data spread \
  devices(2,0,1) \
  range(1:N-2) \
  chunk_size(4) \
  nowait \
  map(from:A[omp_spread_start:omp_spread_size], \
  B[omp_spread_start:omp_spread_size])

#pragma omp target update spread \
  devices(2,0,1) \
  range(1:N-2) \
  chunk_size(4) \
  nowait \
  to( A[omp_spread_start-1:omp_spread_size+2]) \
  from(B[omp_spread_start :omp_spread_size ])
```
Listing 22: Example of the `target {enter|exit|update} data spread` constructs

```
#pragma omp target data spread \
  devices(2,0,1) \
  range(1:N-2) \
  chunk_size(4) \
  map(tofrom:A[omp_spread_start-1:omp_spread_size+2], \
  B[omp_spread_start :omp_spread_size ])
{
  ...
}
```
Listing 23: Example of the target data spread construct

9.2.2. Compiler support for multi-devices

The proposal of the previous section was implemented in the C/C++ frontend of LLVM. Very minimal changes were done to the runtime, only to support a new kind of schedule for the loops that we marked as `target spread`.

Clang lowers its C/C++ input, represented in the frontend using the Clang AST, directly into LLVM IR which already contains calls to the runtime. Our proposal can be seen as another kind of *desugaring* from OpenMP to the OpenMP runtime functions. In our case it was possible to express our transformation in the form of the existing `target` construct.

The implementation has been tested on a system equipped with four NVIDIA A100 GPUs.

9.2.3. Middle-ware extensions

In a multi-device environment, each accelerator will have a range of DMA-capable memory, to be used for the Host – Accelerator communications.

Supporting multiple devices for the MEEP offload mode will require to update the *libomptarget* LLVM-OpenMP support library to access these separate memory areas for the implementation of the DMA transfers to the different accelerators.

The OpenMP infrastructure already supports Accelerator-IDs, in order to operate with different accelerators, and is able to distribute the work from a target loop onto them.

9.2.4. BLIS multi-device approach

The exploration done in this context is purely a conceptual formulation of how this approach should behave. Thus, the BLIS multi-device approach should follow a similar pattern as shown in the *BLIS single-device approach* section. This means that in the application side a data parallel environment is created (Listings 24) using the ***target data spread*** construct. Here, we should define the variables that are going to be offloaded to the multiple devices. The set of multiple devices can be defined as a variable (***DEVICESET***) that is set manually or inquired by the runtime. By default each offloaded variable will be divided into chunks as a function of the number of devices.

```
int main(){
  // Initalization phase ...
  init(x,y);
  // 1st set of computations ...
  // Offload BLAS computations to accelerator
  #pragma omp target data spread \
  map(spread(devices(DEVICESET),range(0:n)), \
  to: x[omp_chunk_start:omp_chunk_size]) \
  map(spread(devices(DEVICESET),range(0:n)), \
  tofrom: y[omp_chunk_start:omp_chunk_size])
  {
    // 1st call to a BLIS routine
    cblas_daxpy(n, scalar, x, 1, y, 1);
    // 2nd call of a BLIS routine
    double dot_product = cblas_ddot(n,x,1,y,1);
    // More BLIS routine calls ...
  }
  // More computations ...
  // Final ...
}
```

Listing 24: Example of an application that defines a data shared environment where a set of BLIS routines are called.

The modifications in the BLAS library rely on the newly proposed ***target spread devices*** construct (Listings 25). The first step should be to reproduce again the data shared environment using the aforementioned ***target data spread*** construct. To offload and distribute work among the set of devices we should add the ***target spread devices*** construct, when encountering a for loop. To make sure that work is computed in parallel by all the devices, we have to setup a set threads (minimum one per device) and therefore, we have to add the ***parallel*** and ***single*** constructs (the last one just to have one thread creating the required tasks to complete the assigned computations).

```
void PASTEMAC3(ch,opname,arch,suf)(conj_t conjx,dim_t n,
                ctype* restrict x,inc_t incx,ctype* restrict y,
                inc_t incy,cntx_t* restrict cntx)
{
  // ...
  _Pragma("omp target data spread \
  map(spread(devices(DEVICESET), range(0:n)), \
  to: x[omp_chunk_start:omp_chunk_size]) \
  map(spread(devices(DEVICESET), range(0:n)), \
  tofrom: y[omp_chunk_start:omp_chunk_size])")
  {
    if ( bli_is_conj( conjx ) ) {
      if ( incx == 1 && incy == 1 ) {
         _Pragma("omp parallel")
         _pragma("omp single")
         _Pragma("omp target spread devices(DEVICESET) nowait")
         for ( dim_t i = 0; i < n; ++i ) {
          PASTEMAC(ch,addjs)( x[i], y[i] );
         }
       }
       else {
         // ...
       }
    }
  }
}
```

Listing 25: Example of the nature of modifications done in the BLIS kernels.

9.2.5. Impact on OpenMP

While flexible, CPUs may not be able to fulfill the performance requirement of specific workloads. In that sense, we believe that accelerators will become more prevalent. A plausible scenario will be systems equipped with several accelerators, with similar performance characteristics, as it may happen in multi-GPU systems.

In that sense, OpenMP should provide an answer in the form of a convenient mechanism to exploit those multi-device systems. Our proposal is a first step towards that goal. It has been presented in meetings at the OpenMP committee, and while our proposal may not be the one eventually chosen, we believe it has sparked conversations about the forthcoming multi-device reality of systems.

10. Conclusions

In this deliverable we have presented the current status of the full MEEP Software Stack: from the Operating System low-level support, to the higher levels of the application layer. The integration of all these components aim to allow ACME EA programmers to exploit all the system capabilities, being able to deploy their own use cases, and to obtain useful information to infer the performance behaviour. As described in the Description of Action (DoA): *”all the application that have been identifed are ported to run on top of the emulation platform”*; and for each application's entry we have also described the metric of interest and its evaluation methodology (from DoA: *”The final phase will focus on application performance evaluation and debugging”*). In short, we are fulfilling the deliverable's requirements and, in addition, reporting the current state of the whole Software Stack.

On the Operating System, we have included the support for communications on the ACME EA infrastructure. On the one hand, we have implemented a Linux driver on both the host and RISC-V sides supporting Ethernet communications over the FPGA PCIe connector. On the host side, we have integrated this support on the QDMA driver by using another driver provided by Xilinx: the Open NIC driver. On the RISC-V side, we have adapted the same Open NIC driver to work with the shared memory that the platform implements.

On the other hand, we have adapted the Ethernet driver developed on the EPI project to the MEEP infrastructure to work with the FPGA QSFP connectors at 10/100 Gbit, allowing point to point communications with other FPGA boards, or connectivity through a switch.

The compiler infrastructure available on MEEP, based on LLVM, effectively supports the two main accelerators of MEEP: the RISC-V Vector Extension and the two Systolic Arrays of MEEP.

The vector support leverages previous work done on LLVM in other projects. Thanks to the vector length agnostic nature of the RISC-V Vector Extension allows for exploring scenarios where the software can communicate facts to the hardware. The hardware can choose to change some of its characteritics, such as the vector length, as an answer to this information. We have explored prefetch instructions to convey memory access information to the CPU with mixed results.

The Systolic Array support is a new development that enables interfacing the MEEP Systolic Arrays via an ISA interface. This ISA interface is built on top of an extension of RISC-V.

The workflow management system provided in the MEEP Software stack is COMPSs, it provides a programming model and runtime to create parallel and distributed workflows as simple Java programs and Python Scripts(PyCOMPSs). We have ported the COMPSs runtime to run in RISC-V 64-bit architecture, and this modifications have been incorporated in the main development branch and released in the latest COMPSs version 3.0 and 3.1. We have also created RPM packages and container images to facilitate its installation and usage.

BLIS is the BLAS library used to provide applications the linear algebra functionalities that they required. This library has been adapted to be used on each MEEP environment: first, we have added support for execution of vector instructions based on the OpenMP SIMD directives and second we provide a set of configurations for each of the MEEP computing platforms. Moreover, we explored the use of this library with a mechanisms, based on OpenMP, to offload all BLIS computations into the available platform accelerators. Specifically, we base this approach on the ***target teams distribute*** for platforms characterized by a single accelerator and on the newly proposed ***target spread*** construct that can be used to distribute work among a set of accelerators.

We have enabled the runtimes of TensorFlow Lite and Apache Spark for RISC-V architectures. Moreover, Epistasia use-case for Spark can be run inside a singularity container. We have included both runtimes as RPM packages. We have also enabled benchmarks for four deep neural networks comprising 99% of the use-cases for deep learning. Finally Epistasia use-case can be offloaded through numpy and BLIS, as most of the computational part is done through numpy.

Regarding container support, we have enabled the usage of most used container engines for the RISC-V 64bits architectures and included in the MEEP OS distribution. We have created RPM packages for Moby, the open source version of Docker, which is the most used container engine, Podman a trending alternative for Docker and Singularity as the most used container engine in HPC environments.

In Section 7.1 we have described the software components included in the MEEP software stack that will allow to apply the proposed *Performance Analysis Methodology* to the different benchmarks. This list of components includes Extrae, PAPI, and Libunwind. We have created RPM packages for all of them, as it can be seen in Tables 9 and 10.

The system benchmarks are intended to understand the behaviour of all MEEP environments. One of the benchmarks is called Stream and is used to benchmark the performance of the memory architecture. The remaining system benchmarks, EPCC-OpenMP and EPCC-OpenMP/MPI, are applied to understand the overheads of common HPC runtimes such as OpenMP and MPI. The set of HPC benchmarks range from very simple and common HPC operations (RISC-V benchmarks) and evolve to more complex and representative HPC workloads (HPL, HPGC, FFTXLib, CloudMicrophysics and Advection-MPDATA). The goal is to understand the performance of the applications on all MEEP environments by looking at different characteristics such as vector instruction performance, multi-thread and multi-node executions.

Actually this goal could be extended to Workflow and Data Analytic benchmarks, although in these cases we plan to check the performance in a higher level of abstraction, looking for scalability tests and general characterization of these kind of workloads.

The MEEP project also envisioned a system with many accelerators, exposing a gap in the OpenMP support for more than one device in the context of offloading. We proposed a new extension to the OpenMP target model with the goal to reduce this gap. This proposal was shared with the OpenMP committee.

10.1. Summary of releases

In this section we gather all the information about releases spreaded all along this document. In addition we also include the specific type of release for each software item.

All the releases have been centralized into a unique web site: . Visitors may navigate among its different sections and found the desired software component. There are three sections specifically devoted to software:

- OS Layer: have the description to install the Operating System. It points to the different

Software Component	**IMG**	**RPM**	**SRC**	**Docker**
LLVM Compiler (Vector/SA)	Yes	–	Yes	riscv64/fedora
LLVM Compiler Multi-device	–	–	Yes	
Ethernet driver	Yes	–	–	
Moby	Yes	Yes	–	
Podman	–	Yes	–	
Singularity	–	Yes	–	
Java Zero VM 11.x	Yes	–	–	riscv64/fedora
Java Server JIT	Yes	Yes	–	riscv64/fedora
Python 3.x	Yes	–	–	riscv64/fedora
Libunwind	Yes	Yes	–	

Table 9: List of fundamental packages

files needed for that process.

- Toolchain: have the list of software components that could be installed in the system: compiler, runtimes, and libraries.
- Benchmarks: have the list of workloads tested in the project and provided for reproducibility purposes.

For each entry in the release website, the content will direct the visitor to the corresponding releases. As described in the Section 2.1 they could be any of the following options:

- included in the OS image,
- a source code repository,
- an installabe RPM package, or
- included in a docker image.

In certain cases, there will be multiple of these options available. For instance, the Extrae package could be already included in the Operating System default image, but also available as an independent RPM package, so any update happening in this package could be updated by running the `yum` command better that downloading again the full Operating System image. Tables 9, 10, and 10.1 summarizes the types of releases available for each software component.

Software Component	IMG	RPM	SRC	Docker
COMPSs	Yes	Yes	Yes	riscv64/compss
TF Lite	Yes	Yes	Yes	riscv64/tflite
Apache Spark	–	Yes	Yes	riscv64/spark
MPICH	Yes	–	–	riscv64/fedora
BLIS (self-hosted)	Yes	Yes	Yes	
BLIS (spread)	–	–	Yes	
Extrae	Yes	Yes	–	
PAPI	Yes	Yes	–	
PAPI LW	Yes	Yes	–	
PAPI LW (vhwc)	Yes	Yes	–	

Table 10: List of runtimes and libraries

Software Component	IMG	RPM	SRC	Docker
Dislib	–	pip	–	riscv64/compss
Epistasia (Spark)	–	Yes	–	riscv64/spark
TFLite Benchmakrs	–	Yes	–	riscv64/tflite
RISC-V Benchmarks	–	–	Yes	
HPCG	–	–	Yes	
HPL	–	–	Yes	
FFTXlib	–	–	Yes	
EPCC Benchmarks	–	–	Yes	
MPI Benchmarks	–	–	Yes	
Stream	–	–	Yes	

Table 11: List of benchmarks

11. List of Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
BLAS	Basic Linear Algebra Services
BSC	Barcelona Supercomputing Center
CGMT	Coarse-Grain Multithreading
CoE	Center of Excellence
CPU	Central Processing Unit
DA	Data Analytics
DL	Deep Learning
DoA	Description of Action (Annex 1 of the Grant Agreement)
DMA	Direct Memory Access
DTB	Device Tree Blob
DTS	Device Tree Source
Dx.y	(MEEP) Deliverable, where *x* is the WP, and *y* is the document id within the WP
EA	Emulated Accelerator
EPI	European Processor Initiative
FGMT	Fine-Grain Multi-Threading
FPGA	Field Programmable Gate Array
GB	Gigabyte, 10^9 bytes
GPU	Graphics Processing Unit
GiB	Gibibit, 2^{30} bits
HBM	High Bandwidth Memory
HPC	High Performance Computing
HPCG	High Performance Conjugate Gradient
HPDA	High Performance Data Analytics
HPL	High Performance Linpack

HSS Hart software Services

ISA Instruction Set Architecture

JIT Just in Time (compilers)

MC Memory Controller

MEEP MareNostrum Experimental Exascale Platform

Mnn (MEEP) Project Month, where *nn* is a numerical value

MSn (MEEP) Project Milestone, where *n* is a numerical value

ML Machine Learning

NIC Network Interface Card

NVRAM Non-volatile Random Access Memory (e.g., 3D XPoint)

OAI Open Accelerator Infrastructure

OAI-OAM Open Accelerator Infrastructure OCP Accelerator Module

OAM Open Compute Accelerator Module

OCP Open Compute Project

ONIC Open NIC

OOO Out of Order (CPU)

OS Operating System

PGAS Partitioned Global Address Space

POP2 Performance Optimisation and Productivity

QDMA Queue Direct Memory Access (Xilinx)

ROM Read-Only Memory

RTL Register Transfer Level (Hardware Description Language)

SA Systolic Array

SBI Supervisor Binary Interface

SCIF Symmetric Communication Interface

SD Secure Digital (card)

SDK Software Development Kit

SIMD	Single Instruction Multiple Data
SoC	System on Chip
TPU	Tensor Processing Unit
TCG	Tiny Core Generator
UBB	Universal Base Board
VOP	Virtio Over PCIe
VPU	Vector Processing Unit
WP	Project Work Package

12. References

[1] Dimemas. https://tools.bsc.es/dimemas. Accessed: 2022-11-07.

[2] . European processor initiative. https://www.european-processor-initiative.eu/. Accessed: 2021-07-08.

[3] libunwind. https://www.nongnu.org/libunwind/. Accessed: 2022-11-08.

[4] Paraver. https://tools.bsc.es/paraver. Accessed: 2022-11-08.

[5] Pop. https://pop-coe.eu/. Accessed: 2022-11-07.

[6] Quantum espresso. https://www.quantum-espresso.org/. Accessed: 2022-09-20.

[7] Xnnpack. https://github.com/google/XNNPACK. Accessed: 2022-11-11.

[8] A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary. Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. https://www.netlib.org/benchmark/hpl/. Accessed: 2021-07-08.

[9] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[10] Adams, D. The hitchhiker's guide to the galaxy. Greatest Book Ever Written (GBEW), Oct. 1979.

[11] Apache Software Foundation. Apache spark: Unified engine for large-scale data analytics. https://spark.apache.org. Accessed: 2022-12-07.

[12] Apache Software Foundation. Tensorflow for mobile and edge. https://www.tensorflow.org/lite. Accessed: 2022-12-07.

[13] Ayers, G., Litz, H., Kozyrakis, C., and Ranganathan, P. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, p. 513–526.

[14] Barcelona Supercomputing Center. Cluster analysis. https://tools.bsc.es/cluster-analysis. Accessed: 2021-10-25.

[15] Barcelona Supercomputing Center. Extrae. https://tools.bsc.es/extrae. Accessed: 2022-11-08.

[16] Barcelona Supercomputing Center. Folding: Detail performance evolution. https://tools.bsc.es/folding. Accessed: 2021-10-25.

[17] Barcelona Supercomputing Center-Centro Nacional de Supercomputación. Risc-v benchmarks. https://gitlab.bsc.es/benchmarks/risc-v-benchmarks.git/. Accessed: 2022-11-10.

[18] Bull, J. M. Measuring synchronisation and scheduling overheads in openmp. In *Proceedings of First European Workshop on OpenMP* (1999), pp. 99–105.

[19] Bull, J. M., Enright, J. P., and Ameer, N. A microbenchmark suite for mixed-mode openmp/mpi. In *Evolving OpenMP in an Age of Extreme Parallelism* (Berlin, Heidelberg, 2009), M. S. Müller, B. R. de Supinski, and B. M. Chapman, Eds., Springer Berlin Heidelberg, pp. 118–131.

[20] Bull, J. M., and O'Neill, D. A microbenchmark suite for openmp 2.0. *SIGARCH Comput. Archit. News 29*, 5 (dec 2001), 41–48.

[21] Field G. Van Zee and Robert A. van de Geijn. Blis repository. https://github.com/flame/blis. Accessed: 2022-11-07.

[22] Frigo, M., and Johnson, S. G. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2 (2005), 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[23] Gianmarco Mengaldo. Batch 1: Definition of several weather & climate dwarfs. http://www.hpc-escape.eu/media-hub/escape-pub/escape-deliverables. Accessed: 2022-08-01.

[24] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition, 2015.

[25] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[26] J. Dongarra, P. Luszczek, M. Heroux, K. Ye. Hpcg benchmark. https://hpcg-benchmark.org. Accessed: 2022-09-20.

[27] J. M. Bull, F. R., and McDonnell, N. A microbenchmark suite for openmp tasks. In *Proceedings of the 8th international conference on OpenMP in a Heterogeneous World (IWOMP '12)* (2012), pp. 271–274.

[28] John D. McCalpin, Ph.D. Stream: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream/. Accessed: 2022-06-09.

[29] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[30] Lordan, F., Tejedor, E., Ejarque, J., Rafanell, R., Alvarez, J., Marozzo, F., Lezzi, D., Sirvent, R., Talia, D., and Badia, R. M. ServiceSs: an interoperable programming framework for the Cloud. *Journal of Grid Computing 12*, 1 (2014), 67–91.

[31] MEEP. Cloudmicrophysics bug report. https://gitlab.bsc.es/meep/common-dashboard/wp5-software-stack/uploads/91ed32d1f4a429be23b6f120d5e59e62/cloudmicrophysics-bug-report.pdf. Accessed: 2022-12-27 (confidential).

[32] MEEP Consortium. Deliverable d5.1: Benchmark suite.

[33] MEEP Consortium. Deliverable d5.2: Linux with initial host interface release, based on the requirements document.

[34] MEEP Consortium. Deliverable d6.3: Emulated accelerator second release with full capability of inter-accelerator communication.

[35] NumPy. Numpy. `https://numpy.org`. Accessed: 2022-12-07.

[36] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition, 2014.

[37] Terpstra, D., Jagode, H., You, H., and Dongarra, J. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009* (Berlin, Heidelberg, 2010), M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds., Springer Berlin Heidelberg, pp. 157–173.

[38] Van Zee, F. G., and van de Geijn, R. A. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software 41*, 3 (June 2015), 14:1–14:33.

A. Systolic Array Specification

This appending includes the current version of the MEEP Systolic Array specification.

MEEP Systolic Array Extension

Version 0.6.2

Table of Contents

1. Introduction

This is the MEEP Systolic Array Extension (MEEP SA). This is an extension of the RISC-V ISA intended to offer an instruction level interface to systolic array operation as envisioned in the context of the MEEP project.

This ISA extension depends on the Vector Extension (V) ISA being available.

This extension models the access to the systolic array functionality similarly to that of a coprocessor.

Note | The MEEP SA requires at the very least a base RISC-V implementation of RV64IMV.

1.1. ISA name

This extension is named `meepsa`. Given that it is a nonstandard extension, the ISA specification is `Xmeepsa`. For instance a Linux capable 64-bit CPU that implements the V extension 0.7.1 and the MEEP Systolic Array could be named `RV64GC_V0p7_Xmeepsa`.

2. MEEP SA Programmer Model

The MEEP SA extension provides an instruction-based interface for the 2 systolic arrays considered in the MEEP project. Each systolic array is assigned an identifier 0 or 1 which is used to identify the architectural state of each systolic array. This identifier is generically noted in this specification using <n> notation.

This extension adds 31 systolic registers per systolic array to the base scalar RISC-V ISA of size `SALEN.<n>`. The extension also adds three unprivileged CSRs `sastatus.<n>`, `saoplen.<n>.0`, `saoplen.<n>.1` of size XLEN bits.

Table 1. New systolic CSRs

Address	Privilege	Name	Description
`0x8D0+n`	URW	`sasatus.<n>`	Systolic array status
`0x8D2+n`	URW	`saoplen.<n>.0`	Systolic array operational length 0.
`0x8D4+n`	URW	`saoplen.<n>.1`	Systolic array operational length 1.

The value of `saoplen.<n>.<m>` is always zero or positive magnitude smaller or equal to `SALEN.<n>` / 8.

Note | Two operational lengths are specified for systolic arrays that operate with bidimensional data.

2.1. Systolic Array Identification

All the instructions in this extension are executed under the context of a specific systolic array. There is a systolic array identifier of 1 bit called `said` which is encoded in the instructions.

Note | This extension supports up to 2 systolic arrays at the same time in the same system. A systolic array may not have a use for `saoplen.<n>.1` in which case it is assumed to be hardcoded to value 1. Also in this case `sastatus.<n>.illoplen.1` may be set to 1 if the value configured in `saoplen.<n>.1` is not 1.

2.2. Systolic Registers

The MEEP SA extension adds 31 architectural systolic registers `sa.<n>.0-sa.<n>.30` to the base scalar RISC-V ISA. Their size is `SALEN.<n>` bits.

Systolic registers are logically divided in elements of 8-bit size, numbered from 0 to `SALEN.<n>` / 8.

Note | The assembly syntax does not use the systolic identifier in the systolic register names because the instruction already establishes the context. Their names in the assembly syntax are `sa0` to `sa31`.

Note | The registers are only divided in elements of size byte for semantic purposes. An implementation may group the elements and require `saoplen.<n>.<m>` be a multiple of that group size.

2.3. Systolic Specific Registers

A systolic array may define few systolic-specific registers (`ssr`). Those registers have XLEN size and and are not exposed to the rest of the architecture.

2.4. Systolic Array Status, `sastatus.<n>`

This CSR contains the operational state of a specific systolic array. This is a read-only register.

This specification defines the following bits in this CSR.

Table 2. `sastatus.<n>` register layout

Bits	Name	Description
63:32	implementation	Implementation-defined status of the SA
31:4	reserved	Reserved for the MEEP-SA spec
4	`sastatus.<n>.illoplen.1`	The SA cannot operate under the given operational length 1
3	`sastatus.<n>.illoplen.0`	The SA cannot operate under the given operational length 0
2	`sastatus.<n>.busy`	The SA is operating
1	`sastatus.<n>.ready`	The SA is ready to accept an operation
0	`sastatus.<n>.enabled`	The SA is enabled and can execute operations

Note | Some of those fields may not be needed and will be removed.

`sastatus.<n>.enabled` establishes that operations for the systolic array <n> can be executed by the instruction. When this bit is set to zero a systolic array instruction directed to the systolic array <n> will cause an illegal instruction fault.

`sastatus.<n>.ready` establishes that a systolic array can accept new operation requests. When this bit clear the execution of a systolic array must behave like a no-operation. A systolic array implementation may choose to always present this bit as set and stall the execution of an instruction until it can accept it.

`sastatus.<n>.busy` establishes that a systolic array is operating. This status is purely informational and does not have functional consequences for the software.

`sastatus.<n>.illoplen.<m>` establishes that a systolic array was requested an operational length that is not valid. These bit are modified by the instructions `sa.setopleni` and `sa.setoplen`.

Bits 31:4 are reserved for further extensions of MEEP-SA. These bits should be left cleared.

Bits 63:32 are reserved for the implementation. Their allowable values are implementation-defined but must include an all-zeros valid setting.

3. Systolic Array Instruction Formats

This specification defines the following instruction formats.

Note | This extension uses the major opcode `custom-1` as defined in the RISC-V Instruction Set Manual. This means that `inst[6:0]=0101011`.

3.1. SACFG formats

3.1.1. SACFG.i

31 27	26	25	24 20	19 15	14 12	11 7	6 0
0 0 0 0 0	said	1	sacsr	zimm[4:0]	1 1 1	rd	0 1 0 1 0 1 1
		i			sacfg		custom-1

3.1.2. SACFG.r

31 27	26	25	24 20	19 15	14 12	11 7	6 0
0 0 0 0 0	said	0	sacsr	rs1	1 1 1	rd	0 1 0 1 0 1 1
		r			sacfg		custom-1

3.2. SAOP format

This is the format for operations that are going to be carried out by the Systolic Array.

31 27	26	25	24 20	19 15	14 13	12	11 7	6 0
sasrc3/sadst2	said	nd	sasrc2	sasrc1	op[0:1]	0	sadst1	0 1 0 1 0 1 1
						saop		custom-1

3.3. SAMEM formats

3.3.1. SAMEM.L format

31 29	28 27	26	25 22	21 20	19 15	14 12	11 7	6 0
0 0 0	opdim	said	0 0 0 0	esize	rs1	0 0 1	sadest	0 1 0 1 0 1 1
				element size	address	samem.load	destination of load	custom-1

3.3.2. SAMEM.S format

31 29	28 27	26	25 22	21 20	19 15	14 12	11 7	6 0
0 0 0	opdim	said	0 0 0 0	esize	rs1	0 1 1	sasrc	0 1 0 1 0 1 1
				element size	address	samem.store	source of store	custom-1

4. Common instructions

The instruction interface defines a set of common instuctions that are available for all the systolic arrays.

A systolic array register operand `sa.<n>.<m>` is encoded in a 5-bit field using the binary encoding of `<m>`.

Note | `0b11111` is not a valid encoding for a systolic array register.

4.1. Set operational length

Instruction `sa.setopleni.<n>.<m> rdest, zimm5` is used to set the operational length `m` of a systolic array.

There is a register form of this instruction `sa.setoplen.<n>.<m> rdest, rs1`. The value of the operational length `<m>` is set from the value in the the register `rsrc1`.

Both instructions are encoded with the SACFG format. `sa.setopleni.<n>.<m>` is encoded using the `SACFG.i` format. `sa.setoplen.<n>.<m>` is encoded using the `SACFG.r` format.

Table 3. `sacsr` field encoding

Register	`sacsr`	Notes
`0b00000`	`sastatus.<n>.0`	Not a valid operand of `sa.setoplen` / `sa.setopleni`.
`0b00001`	`saoplen.<n>.0`	
`0b00010`	`saoplen.<n>.1`	
`0b0xxxx`	Reserved encoding.	
`0b1nnnn`	Designates `ssr` identified by nnnn.	Not mandatory to be a valid operand for `sa.setoplen` / `sa.setopleni`.

4.1.1. Assembly syntax

```
sa.setopleni.<n>.<m> rd, zimm5
sa.setoplen.<n>.<m> rd, rs1
```

4.1.2. Semantics

If the systolic array does not support the operational length of the `zimm5` operand (or the value in register `rs1`), then `sastatus.<n>.illoplen.<m>` is set to 1 and `saoplen.<n>.<m>` is set to zero.

Otherwise `sastatus.<N>.illoplen.<m>` is set to 0 and `saoplen.<n>.<m>` is set to the XLEN zero-extended value of `zimm5` operand.

The determined value of `saoplen.<n>.<m>` is returned in register `rd`.

Note | Decide if we want to provide a mechanism in which the SA allows the software to obtain a valid operational length.

4.2. Set Systolic Specific Registers

This is encoded using the SACFG format where `rdest` is `x0` and `sacsr` is a value ranging `0b10000` to `0b11111`. Both `SACFG.i` and `SACFG.r` formats can be used. `SACFG.i` zero extends to XLEN its immediate operand.

4.2.1. Assembly syntax

```
sa.setssr.<n> <ssr-id>, zimm5
sa.setssr.<n> <ssr-id>, rs1
```

`ssr-id` is an immediate ranging from `0` to `15` that is encoded in the field `sacsr` as `0b10000 + ssr-id`.

4.2.2. Semantics

If `ssr-id` is not a valid systolic-specific register for the systolic array <n> or `rdest` is not `0b00000` this instruction causes an illegal instruction.

Otherwise the value designated by the operand in `rs1` or the zero extended value to XLEN of `zimm5` is set to the systolic-specific register of <n> designated by `ssr-id`.

4.3. Memory accesses

Instruction `sa.load<dim>.<n>` is used to load data in memory to the systolic registers. This instruction is encoded using the `SAMEM.L` format.

Instruction `sa.store<dim>.<n>` is used to store data in systolic arrays to memory. This instruction is encoded using the `SAMEM.S` format.

4.3.1. Assembly syntax

```
sa.load1d0.<n>.<esize> sadest, (rs1)
sa.load1d1.<n>.<esize> sadest, (rs1)
sa.load2d0x1.<n>.<esize> sadest, (rs1)
sa.store1d0.<n>.<esize> sasrc, (rs1)
sa.store1d1.<n>.<esize> sasrc, (rs1)
sa.store2d0x1.<n>.<esize> sasrc, (rs1)
```

4.3.2. Semantics

The amount of data transferred from/to memory is specified by the `<opdim>` operand.

Table 4. `opdim` field encoding

Assembly	`opdim[1:0]`	Data transferred
`1d0`	`0b00`	`saoplen.<n>.0` elements
`1d1`	`0b01`	`saoplen.<n>.1` elements
`2d0x1`	`0b10`	`saoplen.<n>.0` times `saoplen.<n>.1`
	`0b11`	Reserved encoding. Unused.

`sa.load<opdim>.<n>.<esize>` transfers `opdim` times `esize` consecutive bytes starting from the address at `rs1` into consecutive elements (starting from element numbered 0) of register `sadest`.

`sa.store<opdim>.<n>.<esize>` transfers `opdim` consecutive number of elements of size `esize` bytes (starting from element numbered 0) from register `sasrc` to consecutive memory addresses starting from address at `rs1`.

Table 5. `esize` field encoding

`esize[1:0]`	Assembly	Value (bytes)	Description
`0b00`	`e8`	1	Elements of 8-bit
`0b01`	`e16`	2	Elements of 16-bit
`0b10`	`e32`	4	Elements of 32-bit
`0b11`	`e64`	8	Elements of 64-bit

Note | A systolic array may require `rs1` be an aligned memory address depending on the value of `esize`.

Note | Not all the values of `esize` must be supported by a systolic array.

`esize` value must allow `opdim` elements be representable in a systolic register otherwise the instruction yields illegal instruction.

4.4. Generic operation

Instruction `sa.op.<n>` is used to trigger an operation of the systolic array.

4.4.1. Assembly syntax

```
sa.op11.<n>.<op> sadst1, sasrc1
sa.op12.<n>.<op> sadst1, sasrc1, sasrc2
sa.op13.<n>.<op> sadst1, sasrc1, sasrc2, sasrc3
sa.op22.<n>.<op> sadst1, sadst2, sasrc1, sasrc2
```

The two forms exists to accomodate two inputs and two outputs and three inputs and one output operations.

When a systolic array register operand is not present in the instruction, its encoding is `0b11111`.

Field `nd` encodes the numer of destination registers. `0b0` is one destination register and `0b1` encodes two destination registers.

4.4.2. Semantics

The meaning of the operation `<op>` is implementation-defined by the systolic array `<n>`. The systolic array expresses the operation in terms of the values of the different source operand registers and the values of `saoplen.<n>.<m>`.

Note | Systolic-specific registers can participate as input operands of the operation.

5. HEVC Imaging Accelerator

5.1. Identifier

The systolic array identifier for the HEVC Imaging Accelerator is 0.

5.2. Operations

op[1:0]	Description
`0b00`	Computes something
`0b01`	Computes something else

6. Neural Network Inference Accelerator

6.1. Identifier

The systolic array identifier for the Neural Network Inference Accelerator is 1.

6.2. Specific operations

op[1:0]	Assembly	Description
`0b00`	`sa.op.1.noact sadst1, sasrc1, sarc2, sasrc3`	No activation function.
`0b01`	`sa.op.1.crelu sadst1, sasrc1, sarc2, sasrc3`	Activation function is ReLU.
`0b10`	`sa.op.1.htanh sadst1, sasrc1, sarc2, sasrc3`	Activation function is hyperbolic tangent.