MEEP

MareNostrum Experimental Exascale Platform

D5.1 Benchmark suite of HPC applications

Version 1.4

Document Information

Contract Number	946002
Project Website	https://meep-project.eu
Contractual Deadline	30/06/2020
Dissemination Level	Public (PU)
Nature	Report (R)
Author	John Davis (BSC)
Contributors	Rosa M. Badia (BSC), Vicenç Beltran (BSC), David Carrera (BSC), Jorge Ejarque (BSC), Roger Ferrer (BSC), Marta Garcia (BSC), Jorda Polo (BSC), Aaron Call (BSC), Alp Sarkisla (TUBITAK), Leon Dragić (UNIZG), and Xavier Teruel (BSC)
Reviewers	Eduard Ayguadé (BSC), Elisenda Rasero (BSC), Sergi Madonar (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
V 1.0	John Davis (BSC)	Initial draft.
V 1.1	Eduard Ayguadé (BSC), Sergi Madonar (BSC), Elisenda Rasero (BSC)	Review.
V1.2	John Davis (BSC)	Add copyright page and contributors list
V1.3	Xavier Teruel (BSC), Jorge Ejarque (BSC), Aaron Call (BSC), Alp Sarkisla (TUBITAK), and Leon Dragić (UNIZG).	Adding application's selection criteria, application description - kernel decomposition, and individual kernel's description tables in the document. All changes located in Annex A.
V1.4	John Davis (BSC), Xavier Teruel (BSC) and Sergi Madonar (BSC)	Review, minor updates and edits.

COPYRIGHT

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.

DISCLAIMER

Table of Contents

1. Executive Summary

This document outlines the design of the Software Stack for the MareNostrum Experimental Exascale Platform (MEEP). The stack should support the applications selected for the co-design approach. This document is the result of the activities in Work Package 5 (WP5), under task T5.1:

> **Task 5.1: Application identification (M2-M6)** In this task, the focus is to identify HPC, AI, and data analytic workloads that map well to the self-hosted accelerator. The first goal will be to identify traditional HPC applications. The extended goal will be to identify candidate applications from TensorFlow, Apache Spark, or similar frameworks. These applications will be used to drive the hardware/software co-design.

The work done in this task has been influenced by the architectural efforts of WP4 and the possibilities to map the software stack on to the emulator created in WP6. It details the contributions from the MEEP WP5 partner Barcelona Supercomputing Center (BSC), with inputs from UNIZG and TUBITAK for applications driving the design and development of the MEEP Exascale Accelerator Architecture and RTL to be mapped on to the FPGA emulator or Software Development vehicle.

The document is structured as follows. In Section 2, we briefly summarize the MEEP project objectives for background. The goal is to provide broader context for the project and enable this document to stand on its own.

In Section 3, we provide a description of the Software Stack in MEEP and the goals in the different layers in the stack. This is a software/hardware co-design project where the result of the co-design is implemented on a FPGA-based emulation platform, enabling more complete software development and pre-silicon validation. The primary relationship with the software stack is the ability for MEEP to be a Software Development Vehicle (SDV) at the system level and not just a single accelerator.

In Section 4, we discuss each one of the software layers for the HPC/HPDA application stack for MEEP. This is a top-down approach starting with the key HPC and emerging HPDA applications and associated kernels that drive the software/hardware co-design for MEEP. MEEP provides an opportunity to alter all layers in the software stack, including the toolchain.

In Section 5, we discuss the target HW and architecture. MEEP is defining a self-hosted accelerator architecture for both dense and sparse workloads based on chiplets combined in a large package or module combined with HBM and NVRAM. HBM solutions exist today and we can demonstrate them in the pilot. We will be unable to demonstrate tightly-coupled NVRAM.

In Section 6 we provide the timeline and deliverables for the MEEP project on the software stack side. These deliverables are demonstrated on the FPGA emulator starting with a single FPGA, using an offload computation model and progressing to a multi-FPGA solution with self-hosting capabilities.

Finally, in Annex A, we provide more details on the applications used in the MEEP project that guide the co-design of the Accelerated Memory and Compute Engine (ACME) described in D4.1, MEEP System architecture and emulation platform specification. This Annex provides the selection criteria and the high-level characteristics that determine the ACME architecture decisions.

2. Introduction

MEEP is a flexible FPGA-based emulation platform that will explore hardware/software co-designs for Exascale Supercomputers and other hardware targets, based on European-developed IP. MEEP provides two very important functions: 1) an evaluation platform of pre-silicon IP and ideas, at speed and scale; and 2) a software development and experimentation platform to enable software readiness for new hardware. MEEP enables software development, accelerating software maturity, compared to the limitations of software simulation. IP can be tested and validated before moving to silicon, saving time and money.

The objectives of MEEP are to leverage and extend results from previous projects, including EPI and the POP CoE, in the following ways:

- Define, develop, and deploy an FPGA-based emulation platform targeting European-based Exascale Supercomputer RISC-V-based IP development, especially hardware/software co-design.
- Develop a base FPGA shell that provides memory and I/O connectivity to the host CPU and other FPGAs.
- Build FPGA tools and support to map enhanced EPI (scalar core, vector core, cache, and NoC) and MEEP IP into the FPGA core, validating and demonstrating European IP.
- Develop the software toolchain (LLVM compiler, debugger, profiler, OS, and drivers) for RISC-V based accelerators to enable application development and porting.

MEEP will deliver a series of Open-Source IPs, when possible, that can be used for academic purposes and integrated into a functional accelerator or cores for traditional and emerging HPC applications. This is an exciting target for IPs generated from projects like EPI, and an IP source for follow-on projects as well. MEEP will provide a foundation for building European-based chips and infrastructure to enable rapid prototyping using a library of IPs and a standard set of interfaces to the Host CPU and other FPGAs in the system using the FPGA shell. In addition to RISC-V architecture and hardware ecosystem improvements, MEEP will also improve the RISC-V software ecosystem with an improved and extended software tool chain and suite of HPC and HPDA (High Performance Data Analytics) applications.

3. Overview

The priority of MEEP is to run traditional HPC workloads and ecosystem[1]. Furthermore, we see a broader set of applications with high compute requirements in new High Performance Data Analytics (HPDA)

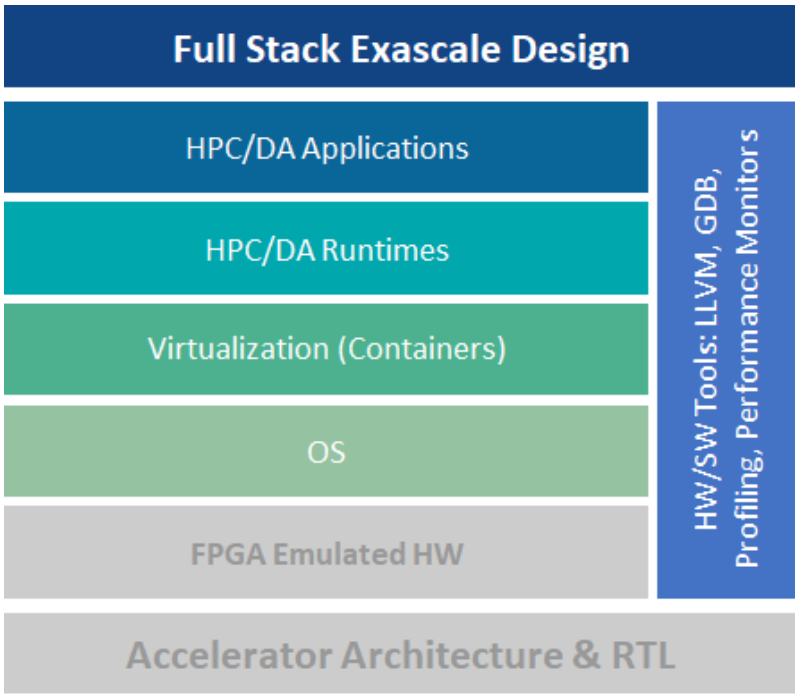

Figure 1 MEEP Full Stack Exascale in relation to the software stack. The focus of this document.

AI/ML/DL frameworks. MEEP will identify workloads from traditional HPC as well as HPDA frameworks to broaden the scope of applications, and if possible select applications that demonstrate shared computational kernels. MEEP is pushing the accelerator model beyond the traditional offload model and in this scenario, the application execution model requires a full featured OS running in the accelerator, in order to be classified as self-hosted. For this, we will devise a stripped-down version of Linux to run on the accelerator, with container support, and host-processor interface. We are targeting classic HPC synthetic benchmarks like Linpack and HPCG, as well as complete applications like GROMACS and Quantum Espresso. MEEP will add other workloads from AI/ML/DL benchmarks, running in frameworks like COMPSs, TensorFlow and Apache Spark. Figure 1, below, provides the main components of the MEEP software stack layers that will be modified or extended to support a RISC-V-based accelerator for exascale computing.

Moving below the frameworks and runtimes, the goal is to bridge the gap between HPC and distributed applications through the use of containerization. As a result, HPC would be accessible by third parties for executing not only HPC-based software, but also distributed applications. In this way, the dense compute capabilities of HPC systems will become available to a wider set of applications. The goal is to encapsulate the software stack into container formats that offer seamless communication between the interdependent components, an encapsulated, defined, and portable environment. MEEP will also

[1] As part of the revision V1.3 it has been included a new section (see Annex A) where the selection criteria have been clarified. This section also provides further information about the main usage of these applications within the MEEP project.

support traditional HPC applications. OpenMP/MPI libraries will be developed for big data container communication, where MPI will be used as a coordination language, which handles the communication between processes.

Regarding the Operating System (OS), two complementary approaches will be considered to expose the MEEP hardware accelerators to the rest of the software stack: 1) offloading of accelerated kernels from a host device to an accelerator device, similar in spirit to how CUDA and OpenCL work; and 2) natively execute a fully-fledged Linux distribution running on the accelerator that will allow the native execution in the accelerator of most common software components as well as container technologies, pushing the accelerator model beyond the traditional offload model explored in the first option.

When it comes to the compiler toolchain, we will look into extending the RISC-V Vector support in two ways. A first approach in which we exploit vector-length agnostic vectorization to provide the CPU, via instruction hints or a similar mechanism, information about the vectorized code. This way the CPU can adjust the vector length to the most suitable length (e.g. to minimize cache misses) for a given code. A second approach in which we study how vector-length agnostic code can be turned into non-agnostic code via a JIT process in runtime. This way constants that were unknown at compile time (such as the actual vector register size) can be leveraged to generate the best code possible without compromising the ISA portability.

All the necessary mechanisms to conduct software profiling, including the application itself and the supporting runtimes, will be considered in the lifetime of MEEP to allow us to better understand their behavior. The availability of hardware counters is considered necessary to apply the methodologies developed in the Performance Optimization and Productivity (POP) European Center for Excellence to be able to detect symptoms that will finally provide hints about software and/or architecture opportunities for performance improvement.

4. MEEP Software Stack

The MEEP software stack is changing the way we think about accelerators and which applications can execute efficiently on this hardware. This is also an opportunity to extend these applications into the RISC-V ecosystem, creating a completely open ecosystem from hardware to application software. Figure 2 below provides a more detailed view of the MEEP software stack.

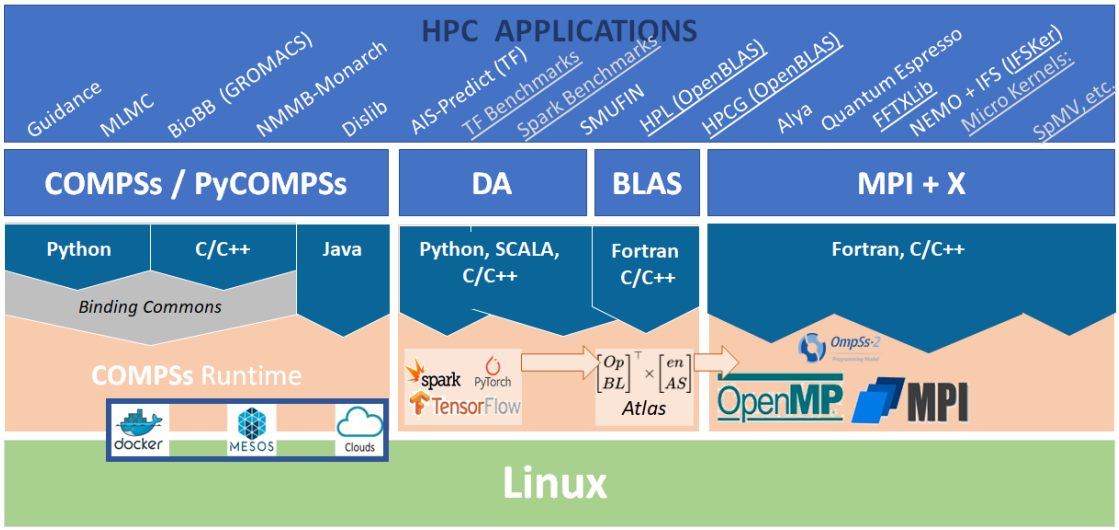

Figure 2. Detailed MEEP software stack.

We have identified a suite of potential workloads to accelerate on the MEEP platform. This includes traditional HPC workloads, like HPL (High-Performance LINPACK), HPCG, GROMACS, ALYA, NEMO and QuantumEspresso. There are also some emerging HPC bioinformatic workloads and workflows like SMUFIN, BioBB and Guidance. In the emerging HPDA application space, we are exploring canonical TensorFlow and Apache Spark benchmarks. Because of the nascent RISC-V ecosystem, we are limited in the types of applications we can support because of lack of or very limited programming language support for FORTRAN, Java, and Python, in the RISC-V ecosystem. We strive to include as many applications shown in FIgure 2 above, but may be limited by ecosystem immaturity.

Moving below the frameworks and runtimes, the goal is to bridge the gap between HPC and distributed applications through the use of containerization. As a result, HPC would be accessible by third parties for executing not only HPC-based software but also distributed applications. In this way, the dense compute capabilities of HPC systems will become available to a wider set of applications. This will be achieved by fulfilling the following objectives: enable easy access and use of HPC resources and facilities as a service, develop and integrate portable containers for HPC, make HPC software transparently available to distributed applications through abstractions for MPI, OpenMP and other HPC based libraries, enable access to HPC resources by integrating light-weight and scalable task-based frameworks, and extend cloud-based system (for HPDA) and deployment management tools for HPC. Finally, if possible, we will enable self-management, monitoring, migration and on-demand scale-up of computation and memory.

In the context of HPDA applications, the goal is to encapsulate the software stack into container formats that offer seamless communication between the interdependent components, an encapsulated, defined, and portable environment for the accelerator. We will leverage the recent advancements in the field of

containerisation for deployment in the HPC infrastructure and address the challenge of maintaining lightweight containers with limited overhead on the computational libraries. More specifically each container will be self-contained as a micro-service and thus will contribute towards resiliency and scalability.

Furthermore, to support traditional HPC applications, OpenMP/MPI libraries will be developed for big data container communication, where MPI will be used as a coordination language, which handles the communication between processes or containers. Porting OpenMP/MPI to containers is a step in the direction of container and big data frameworks support and will be the initial starting point for our extended work.

One of the main challenges for energy efficient Exascale computing is data management and movement. Traditional general purpose processors manage the data movement or marshalling to and from the accelerators, expending significant energy. In MEEP, we would like to demonstrate data resident in the accelerators, thus obviating the need to move data from the host general purpose process. This also allows for sequential code and code with a low degree of parallelism to run in the accelerator. This requires OS support in the accelerator to enable these self-hosting capabilities.

The toolchain is an important part of any system. As a nascent ecosystem, this is especially true for RISC-V and the MEEP accelerator. We will be developing extensions to the LLVM compiler and other essential toolchain components. This will contribute to a more robust environment.

Finally, application and system performance visibility is an essential ingredient in understanding the platform and interactions of the software and hardware. MEEP will leverage and extend existing tools from the POP.

This section provides the details for the different layers in the MEEP software stack shown in Figure 1. Regarding the applications that will allow software/hardware codesign, we consider both traditional High-performance Computing (HPC) applications, emerging high-performance data analytic (HPDA) applications as well as their combined use (HPC/DA) workflows. Then we consider the HPC/DA runtimes (mainly OpenMP/MPI, TensorFlow, Apache Spark and COMPSs) that are needed to support the execution of these applications on top of the target FPGA-based emulation platform. The rest of this section dives into the details of the MEEP software stack and the integration and cooperation required between the software layers, as well as hardware components., relying on the use of containers as the mechanism to package, deploy and offload the execution of HPC/DA applications. Regarding the Operating System (OS), two complementary approaches will be considered to expose the MEEP hardware accelerators to the rest of the software stack: 1) offloading of accelerated kernels from a host device to an accelerator device, similar in spirit to how CUDA and OpenCL works; and 2) natively execute a fully-fledged Linux distribution running on the accelerator that will allow the native execution in the accelerator of most common software components as well as container technologies, pushing the accelerator model beyond the traditional offload model explored in the first option. Finally, the rest of components in the toolchain, including an LLVM-based compiler and performance monitoring tools to replicate the methodology that is used in the Performance Optimization and Productivity (POP) European Center for Excellence to understand.

4.1 HPC/DA Benchmark Suite

The MEEP software stack has assembled a team with broad expertise in HPC and emerging HPDA systems. This not only includes the applications, but the software libraries and frameworks that we use to run the applications. As shown above in Figure 2, there is a rich set of applications and we will focus on the applications that simultaneously have the most RISC-V ecosystem support and the largest HPC impact. This includes applications, libraries and runtimes. In the early phase of the project, MEEP will use microkernels (such as DGEMM, SpMV, FFT, and others), representative code hot spots (functions, loops, etc.) to drive the software/hardware co-design aspects of the MEEP accelerator.

4.1.1 High-performance Computing (HPC) Applications

The HPC workloads selected to evaluate the MEEP platform will be incremental, starting from well-known HPC benchmarks such as HPL (High Performance Linpack) and HPCG (High Performance Conjugate Gradient) and continuing with four widely used HPC applications: Quantum Espresso, Alya, NEMO, and IFS.

The High Performance Computing Linpack (HPL) [19] is a benchmark solving a dense linear equation system on distributed-memory computers. This benchmark is widely used in the HPC community to provide performance results for the TOP500 list. It is implemented on top of the Message Passing Interface MPI, and it requires an implementation of the Basic Linear Algebra Subprograms (BLAS) library.

The High Performance Conjugate Gradients (HPCG) [20,21] is an alternative benchmark also used to rank HPC systems, complementing the metrics provided by the HPL benchmark. The computational and data access pattern of this benchmark pursue to closely match a broad range of actual memory-bounded HPC applications. The reference implementation is written in C++ with MPI and OpenMP support.

Quantum Espresso [22, 23] is a suite for electronic-structure calculations and materials modeling at the nanoscale. It is based on density-functional theory, plane waves, and pseudo-potentials. One of its main components is the Pwscf package, which solves the self-consistent Kohn and Sham equations, but among other packages we can also found: atomic, Phonon, and NEB. The parallelization of Quantum Espresso is achieved using both MPI and OpenMP, and it also leverages the BLAS and LAPACK interfaces. Although Quantum Espresso contains an internal copy of these two linear algebra components, the system can also use any other architecture-optimized replacement for those components (e.g., MKL or ATLAS).

Alya [16] is a high-performance code that simulates computational mechanics; it can solve a wide variety of physics including but not limited to incompressible/compressible flows, non-linear solid mechanics, chemistry, particle transport, heat transfer and turbulence modeling. It is a multiphysics coupled code, meaning that it can solve different physics within the same simulation. These features make Alya ready to be used to solve real-world problems proposed by the research community, but also in the engineering and industrial domains. Alya is developed in Fortran and uses MPI for distributed-memory parallelization. A partial porting to OpenMP is available, meaning that OpenMP can be enabled for some modules. Alya has few external dependencies; this makes it a highly portable code and a good target for MEEP, limiting the number of libraries necessary to run it.

NEMO [17] (Nucleus for European Modelling of the Ocean) is a framework to model the ocean. It has a wide variety of applications whose prime objectives are oceanographic and climate research, operational ocean forecasts and seasonal weather forecasts. It covers both research and operational activities needs. The system consists of three principal engines that allow to model the ocean dynamics, sea-ice

thermodynamics and dynamics, and marine biogeochemistry. The source code of NEMO is written in Fortran; with an MPI implementation that allows it to run in large parallel systems. NEMO includes built-in interfaces for XIOS and OASIS. XIOS is an I/O library. It can be used in library mode or server mode, meaning that it will run in additional processes. OASIS is a coupler that allows different climate codes to be executed in parallel and exchange data.

Quantum Espresso, Alya and NEMO are part of the Unified European Applications Benchmark Suite (UEABS). The UEABS is a set of 13 relevant, scalable and maintained application codes, with publicly available application codes and datasets, of a size which can realistically be run on large systems.

OpenIFS (Open Integrated Forecast System) is a comprehensive Earth system model based on numerical weather prediction. It is parallelized using MPI and OpenMPI. OpenIFS is an easy to use version of IFS for research institutions. The official IFS version is used for the operational weather forecast. It produces from Medium-range forecasts (two weeks ahead) to long-range forecasts (up to one year ahead). OpenIFS can run coupled with NEMO, where OpenIFS provides the atmospheric model and NEMO the ocean surface model. It can also be coupled with other software for the land surface model and sea ice model.

Bolt65 is a performance-optimized HEVC hardware/software suite for Just-in-Time video processing developed as a part of the research activities of HPC Architecture and Application Research Group at Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia. Bolt65 is a "clean-room" suite that consists of an encoder, decoder, and transcoder based on HEVC standard. Special focus in the development of the Bolt65 is set on the performance-efficiency achieved by low-level optimizations and hardware-software co-design adapted for the efficient exploitation of heterogeneous accelerator-based architectures. Another important focus of Bolt65 is the just-in-time processing requirement which sets constraints on processing time making Bolt65 suitable for encoding/transcoding on demand.

4.1.2 High-Performance Data Analytics (HPDA) Applications

MEEP will also consider in its applications portfolio some of applications and application workflows from different application areas that are developed on top of PyCOMPSs/COMPSs and BSC's SMUFIN (Somatic MUtation FINder [9]), a reference-free method designed to identify somatic variation on tumor genomes from the direct comparison with the corresponding normal genome of the same patient. For Spark and TensorFlow, the idea is to mainly use benchmarks, and possibly internal workloads like AIS-Predict, a TensorFlow application for vessel trajectory prediction.

Guidance [3] is a tool for Genome-Wide Association Studies (GWAS) developed by the BSC Life Sciences department with other collaborators. This application, written in Java on top of COMPSs, composes multiple external binaries and scripts. A characteristic of this application is that the different external binaries, although sequential, may require a large amount of memory. COMPSs runtime can take into account these memory requirements and exploit the memory available at each node. As a reference, a production run on hundreds of nodes of MareNostrum 4 involves hundreds of thousands of tasks and half a million files. The challenge for MEEP is to support the application environment and scale down the workload to run on the emulated accelerator.

In the framework of the project ExaQUte, BSC is collaborating in the development of Multi-Level Monte Carlo (MLMC) codes that aim at Uncertainty Quantification (UQ) and Optimization Under Uncertainties (OUU) for Multiphysics and multiscale problems on geometrically complex domains. The MLMC codes

involve the simulations with Kratos multiphysics, the MMG mesher and the XMC Monte Carlo library, everything orchestrated with PyCOMPSs. As a reference, the codes are currently executed on hundreds of nodes of MareNostrum 4 showing good scalability, aiming at the simulation of the CAARC Wind Tunnel test.

In the framework of the BioExcel CoE, BSC and IRB have been developing BioBB [4], a library of building blocks for the composition of molecular dynamic workflows. The library is built on top of PyCOMPSs for large-scale execution. The workflows typically combine invocations to GROMACS to do data processing and analytics methods. As a reference, the workflows can scale to a large number of cores (e.g., a sample execution using 38,400 cores in MareNostrum 4) and are currently used for the research on COVID-19 and are an excellent target for acceleration.

The application Multiscale Online Nonhydrostatic Atmosphere Chemistry model (NMMB-Monarch) aims at providing short to medium range weather and gas-phase chemistry forecasts from regional to global scales that performs weather and dust forecast [5]. The application combines multiple sequential scripts and MPI simulations. PyCOMPSs enables the smooth orchestration of all them as a single workflow. NMMB-Monarch is executed in production twice a day on the MareNostrum 4 and results are offered as services.

Dislib [6] is a distributed machine learning library parallelized with PyCOMPSs that enables large-scale data analytics on HPC infrastructures. Inspired by scikit-learn, dislib provides an estimator-based interface that improves productivity by making algorithms easy to use and interchangeable. This similar interface makes programming with dislib very easy for scientists already familiar with scikit-learn. Dislib also provides a distributed data structure that can be operated as a regular Python object. Dislib has been compared with Dask-ML and MLlib showing better results in terms of performance, but also the ability to handle larger datasets that its competitors.

Given the growing importance of genomics and related research, MEEP is investigating the acceleration of SMUFIN, a reference-free approach based on a direct comparison between normal and tumoral genomic samples from the same patient, implemented in C++. The basic idea behind SMUFIN can be summarized in the following steps: (i) input two sets of nucleic acid reads, normal and tumoral; (ii) build frequency counters of substrings in the input reads; and (iii) compare branches to find imbalances, which are then extracted as candidate positions for variation. Internally, SMUFIN consists of a set of checkpointable stages that are combined to build fully fledged workloads. These stages can be shaped on computing platforms depending on different criteria, such as availability or cost-effectiveness, allowing executions to be adapted to its environment. Data can be split into one or more partitions, and each one of these partitions can then be placed and distributed as needed: sequentially in a single machine, in parallel in multiple nodes, or even in different hardware depending on the characteristics of the stage.

One of the main characteristics of the current version of SMUFIN is its ability to use NVM as a memory extension. If available in the MEEP architecture, two different ways of using this extension could be explored. First, using an NVM optimized Key-Value Store such as RocksDB, and second, using a custom optimized swapping mechanism to flush memory directly to the device. When such memory extensions are available, a maximum size for the data structures is set; once such size is reached, data is flushed to the memory extension while a new empty structure becomes available. Generally speaking, bigger sizes are recommended: they help avoid duplicate data, and also lead to higher performance, as writing big chunks to a Non-Volatile Memory allows the device(s) to exploit internal parallelism typical in flash

drives. If NVM is not available, HBM could be explored as an alternative for SMUFIN since its intermediate data is not required to be persisted.

TensorFlow's Official Models are a collection of well-maintained and optimized models that are used to evaluate the performance of TensorFlow for computer vision, natural language processing, and recommendation systems. In addition to the official models, TensorFlow performance will also be evaluated with AIS-Predict, an application used to forecast the pollution (NOx, SOx and CO_2) produced by ships on the port of Barcelona. The data available are rasters (images) of the pollution with a granularity of 10 minutes. With these images, what the project does is predicting the next images of pollution like if it was a video. The technique used is a Deep Learning Network, in particular Convolutional-LSTM, which mixes the Convolutional Neural Network for images approach with the Long-Short Term Memory networks for time series.

Spark-Bench is a flexible system for benchmarking and simulating Spark jobs, and can be used to evaluate standalone machine learning pipelines running with Spark's internal machine learning library, MLlib. MLlib in turn can be configured to use optimized libraries for certain operations, such as OpenBLAS or MKL. Moreover, Spark can also be used with external ML pipelines. When running with external pipelines, Spark only takes care of the ingestion and distribution of data, while external libraries like XGBoost or TensorFlow perform the training. In addition to machine learning workloads, a subset of TPC-DS can be used to evaluate ETL (Extraction, Transformation, and Loading of data), which represents a significant part of any Spark pipeline.

4.2 HPC/DA Runtimes

At the core of these applications are the runtimes that enable deployment, scaling, and resiliency. They all provide some high-level semantics to define the parallel regions of code and management of resources. Whether it is a sequential or parallel programming model, the runtimes are key to enabling the scale out aspects of high performance computing.

4.2.1 OpenMP/MPI

OpenMP is an industrial standard that defines a parallel programming model based on compiler directives and runtime APIs. OpenMP supports three main models: the worksharing model, the tasking model and the device model. In the worksharing model work is distributed among threads, usually in the form of parallel loops. The tasking model is based around the concept of task and is useful for irregular parallelism that is difficult to exploit using parallel loops of the worksharing model. The device model allows OpenMP applications to offload part of their work to accelerators such as GPUs or other specialized hardware. In addition to these three parallel models, OpenMP provides support for semi-automatic vectorization by means of SIMD compiler directives. The MEEP software stack will use the LLVM OpenMP runtime.

The OpenMP device model can be used to offload the computation from the host nodes to the accelerator nodes (by leveraging an OpenCL runtime as explained in the Operating System section). However, this brings up questions on how the different accelerators can share data. The current OpenMP model is very host-centric which leads to bottlenecks in the presence of lots of accelerators. Being able to communicate between accelerators without involving a host is a problem that needs to be solved and

a good topic to explore in the MEEP project. New OpenMP constructs or new runtime APIs may be needed here. Some of these developments also impact the LLVM compiler.

MPI (Message Passing Interface) is a runtime library for parallelization of applications in distributed memory systems, such as clusters, that implements a SPMD (Single Program Multiple Data) paradigm. MPI is typically combined with OpenMP in which the outer level of the computation across different nodes, not sharing memory, is distributed using MPI. The combination of MPI and OpenMP can be considered the classic building blocks for HPC applications. In MEEP, the OpenMPI implementation will be considered as the library providing MPI.

Finally, there are questions regarding the distribution of computation among the different nodes: DLB (Dynamic Load Balancing) [13] seems suitable to address potential unbalances exposed by applications running on the accelerators. DLB seems like a good candidate for integration both at the OpenMP level of applications. That would be implemented in the LLVM OpenMP runtime by providing more fine-grained balance controlling mechanisms to DLB. To do that we aim at implementing the concept of "unshackled threads" for the task model. This way, the runtime will be able to use an extra number of threads (not strictly tied to the OpenMP parallel region) to execute OpenMP tasks. Then DLB will set the precise number for different regions of the code accordingly. This will require defining some minimal interface between OpenMP and DLB to change the number of unshackled threads.

4.2.2 TensorFlow

TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks. Its flexible architecture allows for the easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. TensorFlow computations are expressed as stateful dataflow graphs. The name TensorFlow derives from the operations that such neural networks perform on multidimensional data arrays, which are referred to as tensors.

TensorFlow provides stable Python (for version 3.7 across all platforms) and C APIs; and without API backwards compatibility guarantee: C++, Go, Java, JavaScript and Swift. While TensorFlow itself will only run in the host node, the integration with the MEEP software stack could be accomplished by porting and offloading some of the operations performed by TensorFlow's libraries to the accelerator, similar to what is done with the TPUs (Tensor Processing Unit) and GPUs today.

In addition to TensorFlow, TensorFlow Lite is a set of tools to help developers run TensorFlow models on mobile, embedded, and IoT devices. An initial version of TensorFlow Lite has been ported to RISC-V. We plan to use this as a building block and extend it for the MEEP accelerator. It enables on-device machine learning inference with low latency and a small binary size. TensorFlow Lite consists of two main components:

- The TensorFlow Interpreter, which runs specially optimized models on many different hardware types, including mobile phones, embedded Linux devices, and microcontrollers.
- The TensorFlow converter, which converts TensorFlow models into an efficient form for use by the interpreter, and can introduce optimizations to improve binary size and performance.

TensorFlow Lite consists essentially of inference tools for edge computing. TensorFlow Lite plans to provide high performance on-device inference for any TensorFlow model. However, the TensorFlow Lite interpreter currently supports a limited subset of TensorFlow operators that have been optimized for on-device use. This means that some models require additional steps to work with TensorFlow Lite. Overall, MEEP will evaluate how to extend TensorFlow Lite to include more features, moving it closer to TensorFlow.

4.2.3 Apache Spark

Apache Spark (Spark for short) has as its architectural foundation the Resilient Distributed Dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant way. Spark facilitates the implementation of both iterative algorithms, which visit their data set multiple times in a loop, and interactive/exploratory data analysis, i.e., the repeated database-style querying of data. The latency of such applications may be reduced by several orders of magnitude compared to Apache Hadoop MapReduce implementation. Among the class of iterative algorithms are the training algorithms for machine learning systems, which formed the initial impetus for developing Apache Spark.

Apache Spark requires a cluster manager and a distributed storage system. For cluster management, Spark supports standalone (native Spark cluster, where you can launch a cluster either manually or use the launch scripts provided by the install package. It is also possible to run these daemons on a single machine for testing), Hadoop YARN, Apache Mesos or Kubernetes. For distributed storage, Spark can interface with a wide variety, including Alluxio, Hadoop Distributed File System (HDFS), Cassandra, OpenStack Swift, Amazon S3, Kudu, Lustre file system, or a custom solution can be implemented.

Apache Spark and its ecosystem rely on the JVM, which doesn't have an interpreter for RISC-V, so Spark itself will only run in the host node. The integration with the MEEP software stack will be accomplished by porting and offloading some of the operations performed by Spark's internal and/or external machine learning libraries to the accelerator.

4.2.4 COMPSs

PyCOMPSs/COMPSs is a parallel task-based programming model for distributed computing platforms. It supports Python, Java, and C/C++ applications. Based on a sequential interface, at execution time the COMPSs runtime is able to exploit the inherent parallelism of applications at task level.

A PyCOMPSs/COMPSs application is composed of tasks, methods annotated. The annotation identifies the tasks, and includes the directionality of their parameters (if a parameter is read, written or both). The runtime builds at execution time a task-graph (or workflow) of the application taking into account the task data-dependencies, and schedules and executes the tasks in the distributed infrastructure, also taking care of the required data transfers. The PyCOMPSs/COMPSs interface supports multiple extensions, like the inclusion of hardware/software constraints to the task execution (i.e. a given number of CPU cores or a minimum amount of memory to be allocated in the node), and has extensions to support multicore and multinode tasks (i.e., a task can be an invocation to an MPI application whose execution involves multiple nodes).

PyCOMPSs/COMPSs runtime is deployed in a distributed mode following the master-worker paradigm, with the computational infrastructure being described in an XML file. The application starts in the master

node and tasks are offloaded to worker nodes. All data scheduling decisions and data transfers are performed by the runtime. The path for the integration of the PyCOMPSs/COMPSs runtime with RISC-V MEEP platform is described in section 4.3 and through the offloading mechanisms described in section 4.4.

With regard to the execution environment, COMPSs runtime supports the execution of applications in large clusters, clouds and federated clouds, and container managed infrastructures (Docker and Singularity). The runtime supports different features, such as elasticity (both in clouds and slurm-managed clusters), failure management at task-level and tasks' exception management.

In the current version of COMPSs, the interaction with container engines is done by encapsulating the whole application in a container image. (In contrast to Apache Spark and TensorFlow, which use containers to define a stage of computation in the dataflow graph for a set of tasks.) On execution, a COMPSs script interacts with the container script to start the containers and deploy the application in the containers.

As a work in progress, COMPSs is being extended to support individual tasks deployed in container images (similar to Spark and TensorFlow). In this case, the container is created at task invocation. While Kubernetes is not used in the current developments, extensions to leverage Kubernetes are being considered in the scope of the MEEP project.

4.3 Virtualization/Containers

The MEEP project envisions the usage of containers as the mechanism to package, deploy and offload the execution of applications. However, since the project is based on a new architecture, not all needed software components will be available from the beginning. For this reason, we have defined an incremental plan for this objective, composed of 3 steps. In all cases, we are considering that the application is started in the host, and parts of the application (or the whole application in some cases), are offloaded to the accelerator for their execution. For simplicity, we will call "tasks" to the application parts offloaded to the accelerator. The deployment process will be based on COMPSs as the base software, both for COMPSs and non-COMPSs applications. In case of non-COMPSs applications, the application will be deployed as a single task into the accelerator, i.e., will be considered as a COMPSs application with a single task.

- Step 1: The first scenario we are considering is with limited support of the container engine. While the container technology will be available from the beginning for the host, it will not be available for the accelerator. In this case, the application starts in the host and tasks are offloaded to the accelerator using some low level API.
- Step 2: This second scenario will consider the use of the container engine in the accelerator, but the full software stack cannot be deployed in container images for the accelerator. For instance, the accelerator can run compiled binaries but it is not possible to run Java or Python programs due to the lack of JVM or Python interpreter support in the current RISC-V ecosystem. In this case, COMPSs runtime will run in the host and it will deploy the tasks previously encapsulated in container images in the accelerator using the container engine API or command-line interface
- Step 3: In this third scenario, accelerators have full stack support, so the whole COMPSs application, including the runtime, will be able to run in the accelerators in a set of containers. Docker/Kubernetes will be used to manage a cluster of accelerators.

The MEEP project can build on nascent container support. With regard to the container software to be used, [7] describes the installation process for Docker containers version in RISC-V. The article presents this process based on a RISC-V virtual machine. The process includes the installation of Golang in RISC-V and the Docker installation. In [8] it is described how to build Docker images for RISC-V, either with QEMU emulation, using (multiple) remote native nodes or cross-compilation in multi-stage builds.

As a summary, there are currently small prototypes trying to port container technologies to RISC-V, some of them appear to succeed in porting the container engines. However, the images deployed in these prototypes are simple software stacks mainly based on C++ and Go. Supporting complex stacks to achieve Step 3 will also depend on the maturity of the RISC-V support for other languages such as Java or Python.

4.4 Operating System

The Operating System (OS) is the software layer between the bare metal and the rest of the software stack. The OS is in charge of low-level tasks such as discover, enumerate and initialize all hardware components in a system. These hardware components are exported to other software layers through a set of well-defined interfaces that abstract hardware specific details. Linux is the most used OS in cloud, edge and HPC computing, so it is the natural choice for this project. The Linux version 4.15 was the first to include support for the RISC-V ISA. Since then, the Linux kernel has been extended to support several developer boards based on different RISC-V designs. Moreover, some Linux distributions such as Debian (https://wiki.debian.org/RISC-V#Progress) or

Fedora (https://fedoraproject.org/wiki/Architectures/RISC-V/Installing) are being ported to the RISC-V platform.

MEEP plans to follow two complementary approaches to expose the MEEP hardware accelerators to the rest of the software stack: the offloading and the native modes:

The *offloading mode* will be based on an offloading API to offload *accelerated kernels* from a host device to an accelerator device, similar in spirit to how CUDA and OpenCL works. The offloading API will support the discovery, enumeration and initialization of the accelerators present in a system. It will also include support to move data from host to device and vice-versa, as well as, methods to trigger and synchronize the execution of kernels in the accelerator. We will try to leverage standard APIs such as OpenCL, as much as possible, but with the ability to extend these APIs if necessary.

The other approach used to expose the MEEP hardware accelerator is the *native mode*. The native *mode* will be based on a fully-fledged Linux distribution running on the accelerator that will allow the native execution in the accelerator of most common software components (MPI, OpenMP, etc), as well as container technologies such as LXC.

Even the *offloading mode* will require a minimal OS to setup the hardware and provide the most basic services. Hence, the first goal of this task is to setup and boot a stripped-down Linux kernel that supports the RISC-V FPGA development board used in WP6. This initial OS will only contain the minimal set of features required to support the *offload model*. The second goal will be to deploy a full-fledged Linux distribution that can be used to install and run the most common software packages in the RISC-V FPGA

boards, so most of the programming models and frameworks defined in Section 4.2, as well as, the container technologies mentioned in Section 4.3 can run natively on the MEEP hardware.

4.5 LLVM Compiler

LLVM is an umbrella project hosted by the LLVM foundation for the development of compilers and related tooling. LLVM has good support for C and C++, has increasingly attracted interest in the HPC community (like in the Exascale Compute Project [14]). Support for Fortran is work in progress in LLVM and is thus not supported in RISC-V yet.

The V-extension adds vectorization capabilities to the RISC-V architecture. Vector instructions, including those of the V-extension, can be exploited using OpenMP SIMD directives and this is the initial work that the EPI project is carrying out with the LLVM compilation infrastructure. One distinctive feature of the V-extension is the reinstatement of concepts from vector computing of the 1970s. Current compilation infrastructures (including vectorizers) are well suited for SIMD architectures (such as Intel's AVX-512 or Arm's SVE) but need to be revisited in the light of a new reality of vector-length agnostic ISAs (such as SVE or the RISC-V V-extension itself).

Vector-length agnostic compilation brings up new challenges, some of which can compromise the achievable performance (it has been reported [10] that at the moment the penalty is about 10%). It is worth exploring practical techniques to recover the lost performance: one of them is applying *revectorization* techniques either in compile time [11] or at runtime [12].

As another approach to further vector-length agnostic compilation we propose defining an interplay mechanism between the CPU and the software, embodied by hint instructions. The compiler would emit such instructions after analyzing the memory access operations in vectorized loops. This mechanism will convey information to the CPU regarding the memory. The CPU should be able to use this information to cap the loop vector length if needed. Some algorithms may exacerbate their working sets when vectorized putting additional pressure on the memory system. The CPU may choose to reduce the vector length in these cases based on the information provided by the hints. The decision of the CPU could in some circumstances be reused in a JIT setting to create a feedback in which the vector-length agnostic code is recompiled to fixed-length vector code.

Finally, a few core assumptions of vectorizers may need to be revisited: vector registers may be large and modeling them as usual (scalar) registers incurs overheads such as copies. A technique here to explore is increasing the aggressiveness of rematerialization during register allocation of the vector registers.

4.6 Software Profiling Tools

Software profiling allows us to better understand the application behaviour under certain circumstances. With such information, we will be able to detect symptoms that will finally provide hints about the software or architecture boundaries. The application analysis can be carried out according to different criteria. First, with respect to the level of detail we want to obtain. Second, and depending on the nature of the metrics, it will allow the analysis of a single application run, or a comparative study of multiple executions when increasing or decreasing the amount of used resources.

In an initial phase of the study, we can work with high-level metrics that define the main characteristics of the application: overall execution time and how it evolves as we add more system resources (processors/accelerators, memory, etc.), allowing us to compute the speed-up. Deeper in our methodology, we will study the different phases into which the application execution is divided, that is, the program structure. This level will help to understand where it will be more beneficial to apply our optimization efforts, detecting critical phases, and being able to establish what will be our area of interest. Once the critical phase(s) has/have been detected (i.e., we have set the focus of the analysis), we will obtain more detailed metrics that define which aspects of execution may be candidates for improvement. The metrics used during this phase of the study will consist of efficiency ratios, indicating how a certain aspect of the application improves or degrades when scaling the system. Among the aspects subject to study we find: computational behavior, serialization of communications, data transfers and load imbalance.

The methodology described in this section, that will be used as a reference in the MEEP project, is based on the methodology used in the Performance Optimization and Productivity (POP [14]) European Center for Excellence. Among the activities planned in the scope of this work package, we do not withdraw a potential collaboration with that project (e.g., requesting a Performance Assessment for any of the candidate applications we are considering in this deliverable). It is also important to mention this methodology focuses on the study of pure MPI applications, although the study of performance metrics for hybrid applications using MPI and OpenMP is being developed in its current phase. With this hybrid approach, we will be able to cover a large part of the applications described in the previous sections. For other applications based on the COMPSs programming model, we believe the methodology could be perfectly adapted; while for Data Analytics oriented applications, we should explore the opportunities it offers. The great challenge in terms of application/runtime profiling, and it applies to all previously mentioned cases, appears when we add the accelerator devices in the game.

The majority of metrics described in the previous paragraphs will be obtained using the Extrae [15] instrumentation library. Therefore, it is essential that this tool can execute properly on the architecture we want to analyze. In turn, Extrae also depends on third party libraries providing essential information for the methodology deployment (libunwind, to explore the call-frame; or PAPI, in order to obtain hardware counters, just to name some examples). The quality of the software profiling will depend on the correct operation of these tools in the target architecture. The MEEP project will explore both facets, software library support and instrumentation library support for RISC-V. Feedback to architecture designers will be provided to include the necessary hardware counters that are really necessary to apply the methodology described above.

5. Target Hardware

The MEEP project is emulating an accelerator that is 5 years in the future with FPGAs. The goal is to map as much of a single accelerator to a single FPGA and compose a system of multiple FPGAs or emulated accelerators. By taking this approach, we can focus on emulation speed vs overall emulated accelerator performance. We can put all the main components of the accelerator in the FPGA, but scaled down to fit in the resources that FPGA provides. We can leverage the FPGA built-in components and hardware macros and efficiently map components to FPGA structures to each reasonable performance of multiple accelerator cores operating in parallel.

The self-hosted accelerator conceived in MEEP is envisioned as a collection of chiplets that are composed together in a module, similar to modern and near-future CPUs and GPUs. The goal of MEEP is to capture the essential components of one of the chiplets in the FPGA and replicate that instance multiple times, a one-to-one mapping of an emulated accelerator to an FPGA. The components include the memory hierarchy from HBM and the intelligent memory controllers down to the scratchpads, and L1 caches, the scalar cores, Vector Processing Unit and Systolic Array. Figure 3, below shows the high-level block diagram of the self-hosted accelerator.

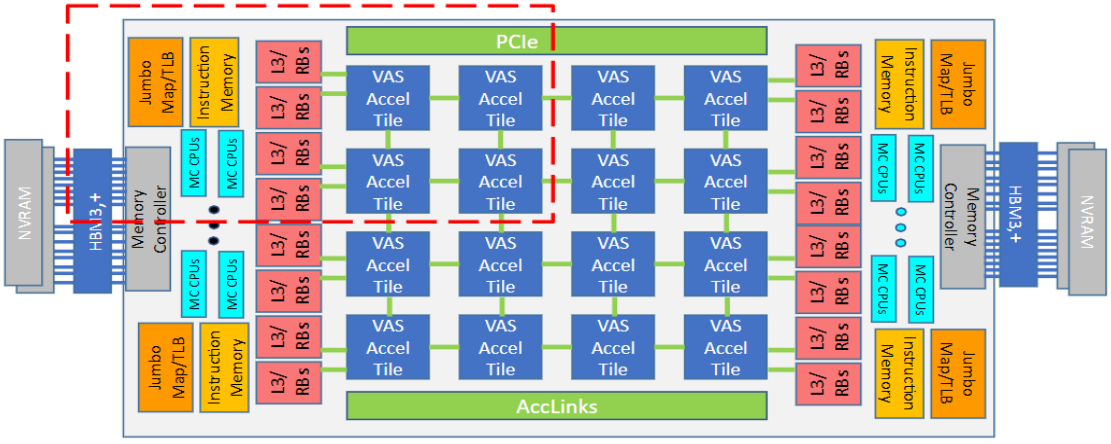

Figure 3. Target self-hosted accelerator chiplet architecture.

At the core of the self-hosted accelerator is the VAS (Vector And Systolic) Accelerator Tile. This is a cluster of 8 cores, with each core supporting 16 Vector Lanes in the VPU. These vector lanes can be subdivided down to 2 lanes and assigned to a single thread. Multiple power of two lanes (2, 4, 8, or 16) can be fused together and supported by a single thread as well. The associated scalar core can support up to 8-way coarse-grain multi-threading (CGMT) to support the lane configuration. At a high-level, dense HPC computations will be supported by a fully fused VPU whereas sparse computations rely on multiple outstanding misses which the 8 threads can issue and be supported by 2-lane VPUs given VPU throughput is less of an issue compared to memory bandwidth limitations. Thus, the tile can support up to 64 contexts depending on the vector personality, a spectrum describing characteristics of the application from dense workloads like DGEMM with a lot of data reuse (16 lane VPU), to sparse workloads like SpMV with little or no reuse (8 x 2 Lane VPUs). MEEP will also explore that opportunity to implement the capabilities of a Systolic Array into the VPU or if that functionality must be separate.

Although not shown in the diagram, the 8 cores share an L2 data cache that is 4 MB in size with 16-ways and 16 banks. We enable this shared resource to act as a scratchpad by disabling ways in the L2. This is transparent to the programmer, but visible to the compiler in the context of virtual register file details like the number of registers and the register file length. This architecture preserves the same programming model as the EPI EPAC vector processor with added capabilities for more accelerators. This architecture also supports an L3 cache that can double as a DRAM row buffer. If there is significant reuse of data in the row buffer, it can be stored on chip and act as a DRAM row buffer creating a higher dimension virtually interleaved DRAM. The row buffer (re)placement policies as well as many other HBM and on-chip memory policies are controlled and informed by the Memory Controller (MC) CPU. These CPUs support fine grain multi-threading (FGMT) or out-of order (OOO) cores to manage the memory requests, both scalar and vector, of the VAS cores. The type of core is still to be determined based on investigations within MEEP. In this case, the MC CPU is another coprocessor that happens to reside near the memory controllers of the HBM memory. As a memory coprocessor, these cores can also operate on vector index registers for scatter/gather operations, memory operations like atomics and simple arithmetic operations. There is also a shared, multi-bank L2 instruction cache as well as a large TLB for 4K, 2 MB, and 1 GB pages, the latter page sizes associated with accelerator workloads and the former with OS pages. MC CPUs are overprovisioned in the accelerator. The purpose is to use some of the extra MC CPUs to run the local OS, daemons and other resource scheduling and accounting software. This puts the self in self-hosting.

There are several I/O options for the chiplet. PCIe provides a universal interconnect to the chip and system. Likewise, an accelerator Link is useful for inter- and intra-chiplet data movement. In general, to minimize complexity, coherence beyond the chiplet boundary is still under investigation. This enables pooling the HBM memory into larger instances by leverage PGAS and other techniques to cut down on unnecessary data movement between the host and accelerator. Essentially, all application data can reside on the accelerator because most of the code runs in the accelerator and not the host. The host provides services.

It is clear that all of the envisioned chiplet cannot be mapped into a single FPGA. As shown above in FIgure 3. MEEP plans to map part of the design (shown in the red dotted box for illustration) into an FPGA for accurate analysis and future scaling. We can leverage the HBM and PCIe macros in the FPGA and build additional functionality, like the MC CPUs. Likewise, we can implement fewer VAS Accel tiles because the focus is on the higher level system operation at scale vs. demonstrating absolute chiplet performance. Figure 3 is a good description of the high-level capabilities of the accelerator. How much we can fit will be determined by the FPGA and efficient resource utilization. More details about the architecture can be found in Work the Package 4 Architecture deliverable and in the Work Package 6 FPGA deliverables.

6. Timeline and Deliverables

This work package has seven main tasks that align with various software layers outlined in Section 4 and listed below. Figure 4, below, provides the monthly timeline for the project, milestones, and the milestone delivery dates.

- Task 5.1: Application identification
- Task 5.2: Compiler support for the self-hosted accelerator
- Task 5.3: Stripped down RISC-V Linux to run on the accelerator, host-processor interface design and container support
- Task 5.4: HPC Runtime environment
- Task 5.5: Data analytics Runtime environment
- Task 5.6: Performance profiling and tuning
- Task 5.7: Benchmark selection and preparation

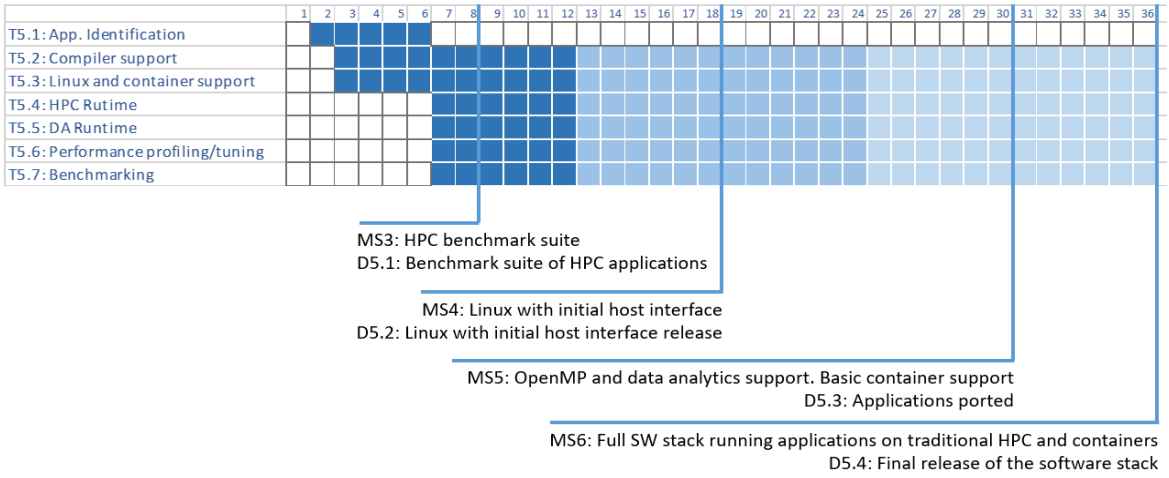

Figure 4. Monthly timeline for the MEEP project, including milestones and deliverables with their delivery dates.

This document serves as the first deliverable D5.1 and satisfies the requirements in MS3. The overall project has experienced some delays due to COVID-19, but we are confident that we can adjust our effort to compensate for these delays and maintain the schedule provided above.

7. References

[1] Sonia Ben Mokhtar, Antoine Boutet, Pascal Felber, Marcelo Pasin, Rafael Pires, and Valerio Schiavoni. X-search: revisiting private web search using intel SGX. In K. R. Jayaram, Anshul Gandhi, Bettina Kemme, and Peter R. Pietzuch, editors, Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017, pages 198–208. ACM, 2017.

[2] Rafael Pires, Daniel Gavril, Pascal Felber, Emanuel Onica, and Marcelo Pasin. A lightweight mapreduce framework for secure processing with SGX. In Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017, Madrid, Spain, May 14-17, 2017, pages 1100–1107. IEEE Computer Society / ACM, 2017.

[3] Guindo-Martínez, Marta, Ramon Amela, Sílvia Bonàs-Guarch, Montserrat Puiggròs, Cecilia Salvoro, Irene Miguel-Escalada, Caitlin E. Carey et al. "The impact of non-additive genetic associations on age-related complex diseases." bioRxiv (2020).

[4] Andrio, Pau, Adam Hospital, Javier Conejero, Luis Jordá, Marc Del Pino, Laia Codo, Stian Soiland-Reyes et al. "BioExcel Building Blocks, a software library for interoperable biomolecular simulation workflows." Scientific data 6, no. 1 (2019): 1-8.

[5] Conejero, Javier, Cristian Ramon-Cortes, Kim Serradell, and Rosa M. Badia. "Boosting atmospheric dust forecast with pycompss." In 2018 IEEE 14th International Conference on e-Science (e-Science), pp. 464-474. IEEE, 2018.

[6] Cid-Fuentes, Javier Álvarez, Salvi Solà, Pol Álvarez, Alfred Castro-Ginard, and Rosa M. Badia. "dislib: Large Scale High Performance Machine Learning in Python." In 2019 15th International Conference on eScience (eScience), pp. 96-105. IEEE, 2019.

[7] Docker Containers on RISC-V Architecture, https://medium.com/@carlosedp/docker-containers-on-risc-v-architecture-5bc45725624b

[8] Early look at Docker containers on RISC-V, https://medium.com/@tonistiigi/early-look-at-docker-containers-on-risc-v-40ed43b16b09

[9] Moncunill, V., Gonzalez, S., Be`a, S., Andrieux, L.O., Salaverria, I., Royo, C., Martinez, L., Puiggr`os, M., Segura-Wang, M., St¨utz, A.M., et al.: Comprehensive characterization of complex structural variations in cancer by directly comparing genome sequence reads. Nature biotechnology 32(11), 1106–1112 (2014)

[10] A Performance Analysis of Vector Length Agnostic Code. Angel Pohl, Mirko Greese, Biagio Cosenza, Ben Juurlink. International Conference on High Performance Computing & Simulation (HPCS), Workshop APPMM, 2019

[11] Charith Mendis, Ajay Jain, Paras Jain, and Saman Amarasinghe. 2019. Revec: program rejuvenation through revectorization. In *Proceedings of the 28th International Conference on Compiler Construction* (*CC 2019*). Association for Computing Machinery, New York, NY, USA, 29–41. DOI:https://doi.org/10.1145/3302516.3307357

[12] N. Hallou, E. Rohou, P. Clauss and A. Ketterlin, "Dynamic re-vectorization of binary code," *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, Samos, 2015, pp. 228-237, doi: 10.1109/SAMOS.2015.7363680.

[13] Marta Garcia, Jesus Labarta, Julita Corbalan, Hints to improve automatic load balancing with LeWI for hybrid applications, Journal of Parallel and Distributed Computing, Volume 74, Issue 9, 2014, Pages 2781-2794, ISSN 0743-7315,

[14] Exascale Computing Project (ECP) Software Technology Update Report (November 2019) https://www.exascaleproject.org/the-ecp-software-technology-update-report-is-available/

[15] Michael Wagner, Stephan Mohr, Judit Giménez, and Jesús Labarta. "A structured approach′ to performance analysis." In International Workshop on Parallel Tools for High Performance Computing, pp. 1-15. Springer, Cham, 2017.

[16] Harald Servat, Germán Llort, Kevin Huck, Judit Giménez, and Jesús Labarta. "Framework for a productive performance optimization." Parallel Computing 39, no. 8 (2013): 336-353.

[17] Mariano Vázquez, Guillaume Houzeaux, Seid Koric, Antoni Artigues, Jazmin Aguado-Sierra, Ruth Arís, Daniel Mira et al. "Alya: Multiphysics engineering simulation toward exascale." Journal of computational science 14 (2016): 15-27.

[18] Gurvan Madec, Romain Bourdallé-Badie, Pierre-Antoine Bouttier, Clement Bricaud, Diego Bruciaferri, Daley Calvert, Jérôme Chanut et al. "NEMO ocean engine." (2017).

[19] Jack Dongarra. Performance of Various Computers Using Standard Linear Equations Software, Technical Report CS-89-85, University of Tennessee, 1989.

[20] Michael A. Heroux, Jack Dongarra, Piotr Luszczek "HPCG Technical Specification," Sandia National Laboratories Technical Report, SAND2013-8752, October, 2013.

[21] Jack Dongarra, Michael A. Heroux "Toward a New Metric for Ranking High Performance Computing Systems," Sandia National Laboratories Technical Report, SAND2013-4744, June, 2013.

[22] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, A. Dal Corso, S. Fabris, G. Fratesi, S. de Gironcoli, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri, L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero, A. P. Seitsonen, A. Smogunov, P. Umari, R. M. Wentzcovitch, J.Phys.: Condens.Matter 21, 395502 (2009)

[23] P. Giannozzi, O. Andreussi, T. Brumme, O. Bunau, M. Buongiorno Nardelli, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, M. Cococcioni, N. Colonna, I. Carnimeo, A. Dal Corso, S. de Gironcoli, P. Delugas, R. A. DiStasio Jr, A. Ferretti, A. Floris, G. Fratesi, G. Fugallo, R. Gebauer, U. Gerstmann, F. Giustino, T. Gorni, JJia, M. Kawamura, H.-Y. Ko, A. Kokalj, E. K¨u¸c¨ukbenli, M .Lazzeri, M. Marsili, N. Marzari, F. Mauri, N. L. Nguyen, H.-V. Nguyen, A. Otero-de-la-Roza, L. Paulatto, S. Poncé, D. Rocca, R. Sabatini, B. Santra, M. Schlipf, A. P. Seitsonen, A. Smogunov, I. Timrov, T. Thonhauser, P. Umari, N. Vast, X. Wu, S. Baroni, J.Phys.: Condens.Matter 29, 465901 (2017)

8. List of acronyms

AI Artificial Intelligence

API Application Programming Interface

CGMT Coarse-Grain Multi-Threading

CoE Center of Excellence

DL Deep Learning

EA Emulated Accelerator

EPI European Processor Initiative

FGMT Fine-Grain Multi-Threading

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

HBM High Bandwidth Memory

HPC High Performance Computing

HPCG High Performance Conjugate Gradient

HPDA High Performance Data Analytics

HPL High Performance Linpack

ISA Instruction Set Architecture

MEEP MareNostrum Experimental Exascale Platform

ML Machine Learning

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

OOO Out of Order (CPU)

PGAS Partitioned Global Address Space

POP2 Performance Optimisation and Productivity

RTL Register Transfer Level (Hardware Description Language)

TPU Tensor Processing Unit

Annex A Application selection summary

The selection criteria in the MEEP project has been established according to one prerequisite, ecosystem maturity and four driving principles. The prerequisite is the main objective, select codes that are highly relevant (and used) in current HPC systems. Next, we are targeting the RISC-V ecosystem which lacks software maturity, and thus limits the applications. This maturity is a moving target, which we hope to help move and expand the RISC-V software ecosystem. The driving principles deals with the targeted markets, code complexity, programming models and kernel characteristics.

We have honored that requisite basing our first exploration of applications and workloads on several interviews with the operations department at BSC; as well as relying on the expertise on HPC from other BSC departments: CASE, Earth science, and Life science.

Table 1 summarizes the set of the driving principles used to influence the selection criteria for the target applications in the MEEP project for the Accelerated Compute and Memory Engine (ACME) architecture. These applications and related characteristics are inputs to the co-design process and justifying the ACME architecture design decisions. The main goal is to cover each one of the targets described in that table.

Driving principle	Target	Justification
Relevant markets	Traditional HPC, HPC Data Analytics and Work-flows.	Considering three main areas of interest for the HPC community.
Range of complexity	Benchmarks, applications, and work-flows; including for each application relevant kernels.	Allowing to approach each problem from simple kernels to complex work-flow applications.
Programming models	MPI, OpenMP and COMPSs.	Well established programming models in the HPC community; COMPSs programming model will enable work-flows and heterogeneous execution (host + devices).
Kernel characteristics	Access pattern, data reuse and application boundary.	Data access pattern and reusability will determine the usage of the memory hierarchy, potential vector instructions or gather/scatter operations. The application boundaries will allow to determine where the pressure is located.

Table 1:MEEP driving principles for application selection.

Table 2, in the other hand, crosses the list of applications described in Section **0** with the *relevant markets* they belong to (first column, at the left), and the *range of complexity* and *programming models* (last columns, in the right) they are characterized by; keeping the *kernel characteristics* criteria to be described later on in this annex. In addition of the three relevant markets, we have also add a fourth section including codes specially devoted to test certain hardware components explored in the ACME architecture (i.e., *Systolic Arrays*). We have named this section "Hardware specific codes". In this table we have included the aforementioned programming models, but we have added two additional columns

to gather those applications using a different programming model. We have defined two different categories: one for *task-based* programming models, another to *thread-based* programming models. We marked both of them with an asterisk to indicate that further information will be provided in the text with respect to the specific programming models these applications are leveraging. In general, an asterisk annotating any column or row in the following tables, will inform the reader that he can obtain additional details in the text.

Market	Program	Complexity			Programming Model				
		Benchmark	Application	Work-flow	MPI	OpenMP	COMPSs	Thread-based*	Task-based*
High-Performance Computing	Alya		*Yes*		*Yes*	*Yes*			
	Quantum Espresso		*Yes*		*Yes*	*Yes*			
	OpenIFS		*Yes*		*Yes*	*Yes*			
	NEMO		*Yes*		*Yes*	*Yes*			
	HPL	*Yes*			*Yes*	*Yes*			
	HPCG	*Yes*			*Yes*	*Yes*			
	FFTXlib	*Yes*			*Yes*	*Yes*			
High-Performance Computing for Data Analytics	AIS-Predict		*Yes*			*Yes*			
	SMUFIN		*Yes*					*Yes*	
	TF Benchmarks	*Yes*							*Yes*
	Spark Benchmarks	*Yes*							*Yes*
	Dislib		*Yes*			*Yes*	*Yes*		
High-Performance Computing for Work-flows	Guidance			*Yes*			*Yes*		
	MLMC			*Yes*	*Yes*	*Yes*	*Yes*		
	BioBB (Gromacs)			*Yes*	*Yes*	*Yes*	*Yes*		
	NMMB-Monarch			*Yes*	*Yes*		*Yes*		
Hardware Specific Codes	MLPerf (Inference)	*Yes*							*Yes*
	Bolt-65		*Yes*					*Yes*	

Table 2: List of MEEP applications according to market, complexity level, and programming models

We should also consider this table as an initial list of available applications that can be used for testing the ACME architecture. Each element of the table can be removed (if not providing any additional value with respect to another) or substituted (if there is a technical issue preventing their use, e.g., language support or complex software requirements not foreseen in this selection phase). The list may also grow on-demand if new features are found in the process of defining the ACME architecture. Finally, we also expect to use synthetic benchmarks to test particular aspects of the implementation (e.g., stream, to test memory bandwidth; or the MPI benchmarks to test network latency, bandwidth or parallel I/O operations). Although these benchmarks do not characterize any particular application, they provide interesting performance metrics to analyze and compare different RTL implementations.

Among the list of applications there are still some particularities we also want to highlight. For instance, the FFTXlib benchmark (it can be considered a miniApp or a mock-up application), is the characterization of the Quantum Espresso software. It will allow to work at different levels of complexity: starting from

the characteristic kernel (i.e., FFT), raising the bar up to benchmark, and ending the study at the application level.

In this table we have not included, but they can also be considered, the *BENCH*[2] configurations of NEMO. It consists of a set of simplified versions of the application that can provide information about the MPI communications, their costs and structure. Similarly, we have not included, but they can also be considered the *weather and climate dwarfs*[3] of the OpenIFS application.

The SMUFIN code does not declare the use any of the programming models listed in the table, but it is parallelized using POSIX threads. Among nodes, it relies on the access to the whole data, dividing the work by means of process identification. No more communication exists among processes.

Tensor Flow and Spark benchmarks do not declare the use of any of the programming models listed in the table, but they rely on the optimization of third party libraries (e.g., BLAS-like libraries).

MLPerf Inference benchmark is the compilation of multiple benchmarks that uses Tensorflow, Pytorch/Onnx frameworks. These benchmarks include common tasks such as image or language processing. MLPerf has different query scenarios to test different cases.

Bolt65 is an application that can encode (from raw video to hevc/h.265 bitstream), decode (from HEVC/h.265 bitstream to raw video), or transcode video (from hevc/h.265 bitstream encoded with parameters A, to hevc/h.265 bitstream encoded with parameters B). The application is written in native C++. The optimizations used are multithreading (based on the native support in C++) and vector extensions (i.e., AVX, AVX2, NEON, SVE2, and EPI vector extension).

Table 3 gathers the computational kernels characterizing the set of MEEP applications. It is important to notice that computational kernels do not reflect any other feature than regions of code which the main component defining the behavior is the processor. In other words, we exclude the communication pattern or file I/O operations from this equation.

The Alya application could be characterized by two different phases: the solver (using the SpMV kernel) and the assembly of matrices phase (using the Alya graph traversal *ad-hoc* kernel). The latter is embedded in the application and should be isolated in order to further study its behavior.

The AIS-Predict and SMUFIN applications are characterized by other kernels requiring further performance study. In the table we have just included those which may become a bottleneck in certain use cases. Exactly the same applies to Tensor Flow and Spark benchmarks. They contain several benchmarks that could be included in the performance study in the future.

The Guidance code performs a Genome-Wide Association Study (GWAS) for uncovering genetic variants related to complex diseases. It has been extensively used to test the COMPSs work-flow characteristics, being one of the main reasons to include that application in our initial list. Unfortunately, the code uses some binaries that are not distributed as open source and they cannot be compiled for the RISC-V

[2] E. Maisonnave, S. Masson. *NEMO 4.0 performance: how to identify and reduce unnecessary communications.* Sorbonne Universités. TR/CMGC/19/19. [on-line: accessed March 29th 2021]. At: https://cerfacs.fr/wp-content/uploads/2019/01/GLOBC-TR_Maisonnave-Nemo-2019.pdf

[3] Andreas Müller, et al. The ESCAPE project: Energy-efficient Scalable Algorithms for Weather Prediction at Exascale. The SCAPE project. [on-line: accessed March 29th 2021]. At: https://doi.org/10.5194/gmd-12-4425-2019

architecture. Also it contains some dependencies that will add extra burden to the execution of this program. These reasons have been considered for this early discarded application.

For the MLPerf Inference benchmark we have chosen a set of representative benchmarks they can be easily isolated from the complete suite, such as Image classification, Object detection, Language processing, and Segmentation. Al these benchmarks rely on the common matrix multiplication (and accumulation) algorithm, so we are considering DGEMM as the most representative kernel for the suite.

Program(s)	*Kernel(s)*													
	DGEMM	DGESVD	FFT	Stencil	NBody	SpMV	SpMM	Pairwise_distance*	Alya Graph-Tr*	Hadamard product*	Bloom filter*	Sparse hash*	[Inv] Transformation*	[De] Quantization*
Alya						•			•					
Quantum Espresso	•		•											
OpenIFS			•											
NEMO				•										
HPL	•													
HPCG						•								
FFTXlib			•											
AIS-Predict										•				
SMUFIN (prune)											•			
SMUFIN (count)												•		
TF Benchmarks	•													
Spark Benchmarks	•													
Dislib (kmeans, pca)	•	•						•						
Guidance (*disc.*)*														
MLMC						•	•							
BioBB (Gromacs)	•		•		•									
NMMB-Monarch			•											
MLPerf	•													
Bolt65-App													•	•

Table 3: Relationship between MEEP applications and computational kernels.

HEVC/h.265 is a multistage process and consists of big number of kernels. However, use case that we are targeting in MEEP is a processing block consisting of 4 kernels - Transformation, Quantization, Dequantization, Inverse Transformation). Within Bolt65 execution, Transformation and Inverse Transformation (DCT, IDCT) are most worthy since they consume majority of compute time.

Finally, Table 4 shows the summary of computational kernel characteristics. Such characteristics have been classified with respect to:

- **Access pattern:** *Dense*, most of the access are consecutive (unit-stride) in memory; and *Sparse*, most of the data accesses are non-consecutive, i.e., big stride accesses or random.

- **Data reuse:** *High*, the data is usually used multiple times (spatial locality) and in a short time period (temporal locality). *Low*, data can be used more than once (but not often), or in a spaced time slot; *Stream*, data is used only once, in a stream-like mode.
- **Performance limitations:** The applications can be ***memory-bandwidth-***, or ***compute***- bound. According to the core-to-memory topology, and taking into account multi-threaded applications, memory bounded characteristic may contain three different values: empty, meaning the application is not memory bounded; 1-Th, meaning the application is memory-bounded for a single thread; or n-Th, meaning the application is memory-bounded in the presence of multiple threads (stressing the memory bandwidth to a given socket). The relationship between computational instructions and memory operations is taking a commodity processor as the reference to determine compute- or memory- limits.

Kernel(s)	*Characteristics*						
	Data Access		Data Reuse			Perf. Limitations	
	Dense	Sparse	High	Low	Stream	Compute	Memory
DGEMM	*Yes*		*Yes*			*Yes*	
DGESVD	*Yes*		*Yes*			*Yes*	
FFT	*Yes*	*	*Yes*	*		*Yes*	*n-Th*
Stencil	*Yes*			*Yes*			*1-Th*
NBody		*Yes*		*Yes*		*Yes*	*n-Th*
SpMV		*Yes*		*Yes*			*1-Th*
SpMM		*Yes*		*Yes*			*1-Th*
Pairwise_distance*	*Yes*	*	*Yes*			*Yes*	*n-Th**
Alya Graph-Tr*	*Yes*			*Yes*		*Yes*	*n-Th*
Hadamard product*	*Yes*			*Yes*	*	*Yes*	*n-Th*
Bloom filter*		*Yes*		*Yes*			*1-Th*
Sparse hash*		*Yes*		*Yes*			*1-Th*
[Inv-]Transformation*	*Yes*				*Yes*	*Yes*	
[De-]Quantization*	*Yes*				*Yes*	*Yes*	

Table 4: MEEP computational kernels characteristics (based on data access, data reusability, and code boundaries).

First block of kernels are well known services in the HPC community as they are widely used in traditional HPC codes. In some cases, it will depend on the specific implementation we use, but we assume most common approaches for all of them. For instance, *DGEMM* and *DGESVD* will be implemented using blocking techniques in order to leverage temporal and spatial locality.

For all the cases, we also consider a trade-off across the different application phases. For instance, *FFT* mostly uses dense operations accessing consecutive storage locations, but the transpose phases is characterized as sparse. In that case data reuse could be classified as medium (i.e., between low and high reusability ratios).

The *NBody* code simulates a dynamical system of N particles (i.e., bodies) under the influence of physical forces among them. In practice, the algorithm must solve the system of equations of motion for each particle in each of the simulated steps.

Iterative *stencil* loops are used to solve the underlying physics of the problem. Specifically, this operation computes a given element of the matrix by operating with its neighbors (defined by the stencil pattern, e.g., 4-point stencil). This operation is widely used to solve partial differential equations.

SpMV and *SpMM* are initially characterized as sparse, but accesses could also be consecutive in memory. Data reuse for the *SpMV* could be characterized as pseudo-stream, due to in one single step of the algorithm, it only uses once the data.

Pairwise_distance is divided in three phases: 1) compute power of 2 of each element of the matrix; 2) matrix multiplication; and 3) adding elements and compute the square root in order to compute the distance. Each of the phases could be characterized with different boundaries. While the matrix multiplication is clearly compute bound, the compute of distances is memory bound. In addition, between phase 1 and 2, the algorithm computes the transpose of one of the matrices in order to guarantee more consecutive accesses for the matrix multiplication. The transpose phase, by itself, will increase the sparseness of data access, but will benefit the following phase (which takes more time to execute, compared with this transposition, due to the high computational cost) of highly dense memory access pattern.

Hadamard product computes the element-wise multiplication of two matrices, taking two matrices of the same dimension, and producing another matrix (also of the same dimension). The characteristic of this kernel will depend on the topology of the input- and output- matrices, but for the purpose of this project, we consider the most common use-case, which operates on dense matrices.

The *bloom filter* code consists of the construction of a Bloom filter. Every step requires computing N hash functions and accesses N bits in memory. Due to the random distribution of these memory accesses, we describe the kernel as memory-bound. Moreover, many of these may be built by different threads, further stressing the random memory accesses.

The *sparse hash* code consists of the construction of a sparse hash table. Every step requires looking up and setting a bitmap, and allocating a few sequential bytes of memory. There is almost no reuse of memory (i.e., low) between subsequent steps, and the computation is pretty small, so it's described as a memory-bound problem *per se*. Like the *bloom filter* code, many of these may be built by different threads, further stressing the random memory accesses, but also compared to it, this one is more memory bound since computation is smaller.

Regarding Bolt65 kernels, all of them have dense memory access (i.e., the accesses are consecutive in memory since the blocks are already prepared in non-stride fashion). Even though data is used in a streaming model, there is significant reuse and as a result, these kernels could become compute-bound.

- From the simplistic point of view, HEVC Transformation is split in two one-dimensional transforms separated by matrix transpose. A one-dimensional transform consists of NxN matrix multiplication followed by scaling.
- The Quantization kernel is additional scaling process of each matrix element. It is executed in parallel where number of Quantization lanes is defined by width of the data bus.

The Dequantization and Inverse transformation are the inverse processes of these two kernels.

Tables 1-4 drill down from the high-level applications spanning traditional HPC applications to emerging HPDA applications, to the details of the hotspot analysis of the applications and the related characteristics. The kernel classification in Table 4 provides the co-design optimization goals for the ACME Architecture of high performance with high energy efficiency for both dense and sparse HPC and HPDA workloads.