DOI: 10.14529/jsfi210105

The MareNostrum Experimental Exascale Platform (MEEP)

Alexander Fell[1]***, Daniel J. Mazure***[1]***, Teresa C. Garcia***[1]***, Borja Perez***[1]***, Xavier Teruel***[1]***, Pete Wilson***[1]***, John D. Davis***[1]

Nascent Open Source Instruction Set Architectures such as OpenPOWER or RISC-V, allow software/hardware co-designers to fully utilize the underlying hardware, modify it or extend it based on their needs. In this paper, we introduce the vision of the MareNostrum Experimental Exascale Platform (MEEP), an Open Source platform enabling software and hardware stack experimentation targeting the High-Performance Computing (HPC) ecosystem. MEEP is built with state-of-the-art FPGAs that support PCIe and High Bandwidth Memory (HBM), making it ideal to emulate chiplet-based HPC accelerators such as ACME, at the chip, package, and/or system level. MEEP provides an FPGA Shell containing standardized interfaces (I/O and memory), enabling an emulated accelerator to communicate with the hardware of the FPGA and ensures quick integration. The first demonstration of MEEP is mapping a new accelerator, the Accelerated Compute and Memory Engine (ACME), on to this digital laboratory. This enables exploration of this novel disaggregated architecture, which separates the computation from the memory operations, optimizing the accelerator for both dense (compute-bound) and sparse (memory-bandwidth bound) workloads. Dense workloads focus on the computational capabilities of the engine, while dedicated processors for memory accesses optimize non-unit stride and/or random memory accesses required by sparse workloads. MEEP is an open source digital laboratory that can provide a future environment for full-stack co-design and pre-silicon exploration. MEEP invites software developers and hardware engineers to build the application, compiler, libraries and the hardware to solve future challenges in the HPC, AI, ML, and DL domains.

Keywords: high performance computing (HPC), accelerator, software stack, open source hardware.

Introduction

Today, Linux is the omnipresent Operating System (OS) in the Internet-of-Things (IoT) space running on small embedded systems, in the mobile space in the form of Android and it is ubiquitous in High-Performance Computing (HPC) and cloud-based systems with various Open Source Software (OSS) components built on top. OSS provides the foundational building blocks for the OS, toolchain, runtimes, frameworks, libraries, and all the way up to the application layer enabling rapid development and extension of any layer in the software stack.

However, when examining hardware, current commercial off the shelf solutions (COTS) are closed hardware ecosystems that only enable integration at the peripheral level through a defined interface such as PCIe and proprietary drivers. This stifles innovation by limiting optimizations to the software stack, preventing true software/hardware co-design. This problem has been intensified by the end of Dennard scaling and the dramatic slowdown in Moore's Law, requiring specialized hardware, in form of accelerators, to meet the system power and performance requirements. At the same time, the software stack is evolving, becoming more abstract. This enables higher programmer productivity, but sacrificing hardware efficiency. Thus, application owners will need to co-design the full stack, all layers of hardware and software, in order to meet their performance and power (FLOPs/W) targets. This level of tight integration is not possible in a closed or even partially open ecosystem.

[1]Barcelona Supercomputing Center (BSC), Barcelona, Spain

There have been Open Source Hardware (OSH) platforms in the past, but Moore's Law and many other reasons inhibited their adoption, e.g. continuous general purpose processor performance improvements, time to market, cost, software development, etc. Furthermore, unlike Linux, previous OSH was entangled in the companies that created them and/or generally poor quality. Mirroring the same model as Linux, RISC-V has followed a similar development path and has enjoyed significant industrial and academic adoption. The RISC-V ecosystem is in the nascent period where it can become the de facto open hardware platform of the future, having the same opportunity in hardware that Linux created as a foundation for OSS. This enables the co-design of the RISC-V hardware and the entire software stack, creating a better overall solution than the closed hardware approach that is done today.

In this paper, we introduce a new digital laboratory for software and hardware development, called the MareNostrum Experimental Exascale Platform (MEEP). MEEP is a high-level hardware emulation platform that also can be used as a Software Development Vehicle (SDV) and is described in Section 1. Its name (meep meep) pays homage to the RoadRunner supercomputer, the first PetaFLOPS system [1]. MEEP leverages OSS and extends various software layers in the stack to run on a RISC-V based accelerator and consists of a set of defined hardware interfaces, enabling a wide range of accelerators and processors to be emulated. This results in a platform that permits rapid development and testing of new HPC and High Performance Data Analytics (HPDA) hardware accelerators and the associated software ecosystem. The initial targeted software stack is presented in Section 2, comprising of existing HPC and emerging HPDA applications in fields such as machine learning, computer vision, and deep learning, co-designed with those accelerators to meet the desired power and performance expectations.

We demonstrate MEEP's SDV and hardware emulation capabilities by mapping an accelerator called the Accelerated Compute and Memory Engine (ACME) into MEEP. ACME separates computation and memory operations into compute and memory tiles similar to [24]. All tiles are interconnected by a Network-on-Chip (NoC) to ensure scalability. The compute tiles feature RISC-V scalar cores supporting a variety of coprocessor accelerators with a common set of interfaces. Those coprocessor accelerators support vector operations executed on a Vector Processing Unit (VPU) or alternatively, Systolic Arrays (SA) for image processing and neural networks, all connected to the scalar core through the Open Vector Interface (OVI) [22]. Furthermore, the memory tile is responsible for satisfying all memory operations.

Section 3 describes the ACME architecture in greater detail. Section 4 details the FPGA infrastructure and mapping ACME into the MEEP platform. We provide a synopsis of related work in Section 5 followed by a conclusion with a description of future work of this project.

1. MareNostrum Experimental Exascale Platform (MEEP)

MEEP is a flexible FPGA-based emulation platform based on European IP blocks, that explores hardware/software co-designs for Exascale Supercomputers and other hardware targets. As a platform, MEEP provides 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 (refer to Section 4.1). In addition to RISC-V architecture and hardware ecosystem improvements, MEEP also advances the RISC-V software ecosystem with an enhanced and extended software toolchain and software stack, including a suite of HPC and HPDA applications.

MEEP ambitions to play two important roles within the Exascale computation paradigm:

- SDV platform: Enables software development and experimentation, accelerating software maturity compared to the software simulation limitations by enabling software readiness for new hardware. In order to do this, MEEP executes the whole software stack, enabling a set of tools and mechanisms to integrate new functionalities for future challenges in the HPC and HPDA domains. MEEP is designed to run traditional HPC workloads and ecosystems. Furthermore, we see a broader set of applications with high compute requirements in new HPDA AI/ML/DL frameworks that will also be supported, as well as more complex workflows based on experimental programming models (e.g., COMPSs). More details are provided in Section 2.
- Pre-silicon validation platform: Allows testing and validating of new IPs blocks and/or systems before being committed to silicon. Thus, MEEP can leverage the FPGA built-in components and hardware macros, and efficiently map other components (IPs or custom designs) to the available structures of the FPGA. In addition, MEEP provides a foundation for building 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 MEEP FPGA Shell (PCIe, HBM, DDR, Ethernet, and other interfaces), detailed in Section 4.

In Fig. 1, the whole MEEP software and hardware stack is shown. The underlying hardware, emulated by an FPGA, is able to run the software stack including the Linux OS as described in Section 2. The communication between the emulated hardware and software is enabled by the MEEP FPGA Shell, which provides a set of standard interfaces to the I/O and memory components available in the FPGA, such as memory controllers to the memory(s), Ethernet, or PCI Express connectivity. Within the MEEP FPGA Shell, there is an emulation payload that is user defined ranging from High-Level Synthesis (HLS) accelerators to RTL accelerators and/or processors. In this paper, we present an emulated accelerator based on ASIC RTL containing a self-hosted accelerator called ACME (refer to Section 3). Finally, MEEP provides a software and hardware toolchain for debugging, profiling and performance monitoring.

2. Software Stack

The MEEP software stack includes all the levels at which the software can operate: from the application level to the low-level operating system services (e.g., communication). Fig. 1 provides a detailed view of this software stack running on top of the *FPGA Emulated Hardware Accelerator* layer. We use a top-down approach to describe the software layers in the stack.

We have identified a set of target HPC and HPDA applications to demonstrate the SDV capabilities of MEEP. These workloads include traditional HPC applications (e.g. Alya, Quantum Expresso, OpenIFS, and NEMO), HPDA applications based on TensorFlow [5] and Apache Spark [34] (e.g. MLperf, AIS-Predict and SMUFIN), and work-flows based on the COMPSs [16] programming model (e.g. Guidance, MLMC, BioBB, NMMB-Monarch, and Dislib). This wide set of applications provides a representative suite of highly-relevant HPC and HPDA workloads that can be enabled with MEEP as an SDV, executing actual code on the emulated ACME accelerator.

We take a bottom-up approach to the software-hardware co-design methodology, in terms of the SDV support. From the applications shown in Fig. 1, we extract simplified computation kernels that represent a majority of the execution time of the application. These simplified, yet high-relevance, software microkernels are considered as the starting point of the analysis and verification process, and include kernels such as GEMM, SpMV, FFT, Somier, and AXPY. The latter is used as a guiding example in Section 3.4. In addition to the mentioned workloads, we

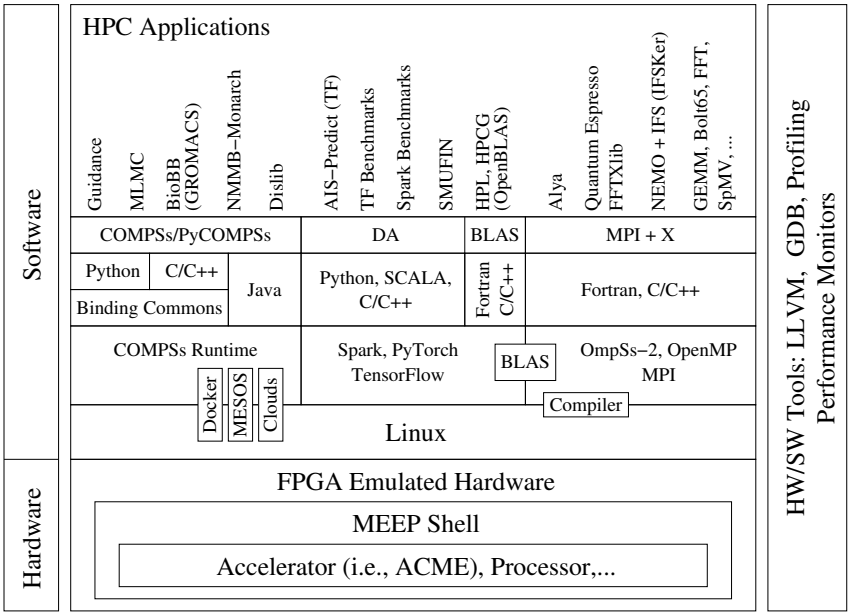

Figure 1. Full Stack of the MEEP design

also include a set of benchmarks and kernels. Benchmarks such as the High-Performance Linpack, Conjugate Gradient, or other mock-up applications like the FFTXlib code reproduce the behavior of the Quantum Expresso, enabling the testing of the expected system performance using a simpler code rather than the entire application. In the HPDA context, the TensorFlow and Apache Spark benchmarks are used to test data analytic service behavior. Finally, once all of these smaller codes run on the SDV, we can execute the complete applications with confidence. Note, this set of applications and benchmarks also includes applications requiring specific hardware characteristics such as the processing of images or natural languages. The main objective of this set of programs is to evaluate the performance of special-purpose hardware accelerators, i.e., SAs and data streaming.

Below the application level we can find the compiler support and the use of different programming languages. The MEEP project bases its optimization support on LLVM and Clang [14]. In this context, we are developing compiler extensions that exploit the performance of the ACME architecture features. The set of selected applications imposes a great variety of programming languages (e.g. C/C++, Fortran, Java, or Python), but the lack of sufficient software ecosystem maturity currently limits the direct use of all these application and certain optimizations. When this is the case, we rely on the generic compilation mechanism supported by the system and the use of third-party optimized libraries to exploit features or limited capabilities of the RISC-V software ecosystem and toolchain.

In addition to the compilation support, the applications have additional libraries and services for commonly optimized algorithms for the platform (e.g. BLAS-like library services). Additional examples include support for run-time libraries for the traditional HPC applications and new work flows (i.e., MPI [17], OpenMP [18], and COMPSs [16]). Porting and optimizing these libraries and services for the RISC-V based ACME architecture is a huge step in the direction of abstraction between the application level and the platform's OS, as well as generally improving the RISC-V software ecosystem maturity.

Further, the ACME architecture also supports the use of containers as the fundamental mechanism to pack, deploy and offload the execution of kernels and applications. The implementation process for containerization is initially based on the COMPSs programming model, as it is consid-

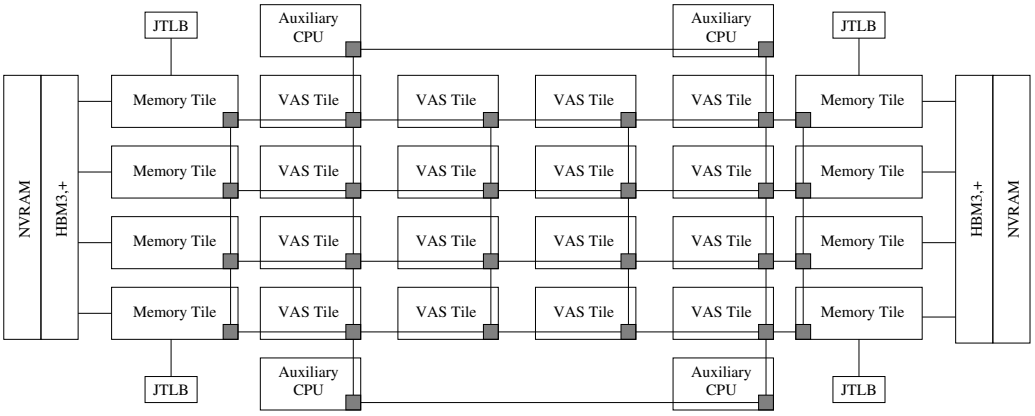

Figure 2. An overview of the Accelerated Compute and Memory Engine (ACME)

ered the most complex use case. There is the more common approach that packages application tasks, phases or stages in containers. However, if this cannot be supported by the application, we can use a different approach where the container is considered as a single-phase work-flow execution, i.e. a COMPSs application is defined a single task or container.

The OS is the lowest software layer in the software stack and MEEP relies on the Linux OS as the engine for this hardware abstraction. A modified, lightweight version of Linux takes care of the low-level duties such as discovering, enumerating, and initializing all physical components of the system. In order to reduce data movement in the system, the ACME architecture executes most of the application code in the accelerator (a self-hosted accelerator), requiring a minimal set of OS features for this execution mode. The traditional Host CPU provides services, like global resource management and name services. However, for pragmatic implementation reasons, ACME also supports a traditional offload execution mode, similar to what is used for General Purpose GPU computation today. In both execution modes, a minimum OS support is required in ACME to configure the hardware and provide the fundamental services: application execution, access to memory, or offload and execute a kernel.

3. Accelerated Compute and Memory Engine (ACME)

The ACME accelerator, as a demonstrator for MEEP, represents the desire to improve the performance for dense as well as sparse workloads. Dense workloads are compute bound, since the performance depends on the number of fused multiply-add (FMA) or fused multiply-accumulate (FMAC) units available in the system, coupled with a reasonable memory hierarchy that can efficiently exploit caching and data reuse. Unfortunately, sparse workloads are limited by the memory bandwidth and hence the focus shifts from computational units to maximizing memory utilization. The purpose of ACME is to find a reasonable balance between the memory hierarchy design for sparse workloads and the FMA architecture for dense workloads.

ACME relies on multiple specialized cores to manage the internal and external resources to improve energy efficiency and performance. It decouples the arithmetic and memory operations by disaggregating the memory access from the execution [24].

3.1. Overview

Moving from a high level view of ACME (Fig. 2) down to a more detailed view (Fig. 3), the core of the accelerator is the Vector And Systolic (VAS) Accelerator Tile. It consists of a cluster of

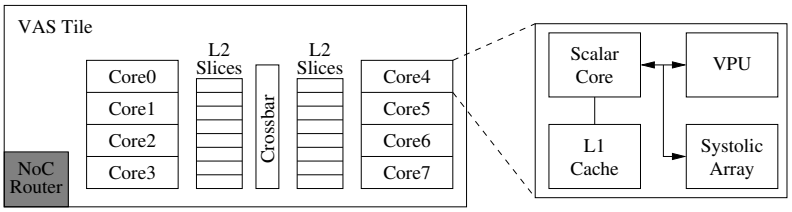

Figure 3. The VAS Tile of ACME

eight scalar RISC-V cores with each core having the capability of supporting several coprocessors: a Vector Processing Unit (VPU) with 16 vector lanes, and based on the targeted application, different Systolic Arrays (SA) designs.

Each VAS Tile includes a distributed L2 data cache of 4MB in size with 16-ways and 16 banks. This L2 can be reconfigured partially to function as a scratchpad depending on the type of workload. Through a NoC, the VAS Tiles are able to communicate with the Memory Tiles to facilitate memory operations decoupling access and execute operations.

Each Memory Tile consists of a Memory CPU (MCPU), which organizes and dispatches memory requests originating in the VAS Tiles to the associated Memory Controller (MC) and its attached slice of the overall High Bandwidth Memory (HBM). The memory address space is mapped across 16 MCs. Therefore, requests that are out of the address range of the current MC, need to be forwarded to the Memory Tiles and MCs that own that address range. The current implementation of ACME integrates HBM due to the limitations of the MEEP FPGA platform. However, non-volatile memories (NVRAM) technologies will be considered in the future [33], when available.

ACME is equipped with four auxiliary CPUs, which are similar to the MCPUs. Those CPUs establish the connectivity to the external components and also run the OS with its daemons and services which are usually provided by the host CPU. Therefore, ACME becomes self-sufficient, orchestrating its own internal resources and hence it is a self-hosted accelerator. By enabling this capability in ACME, the application data can be stored in a single location and does not have to be moved between the host CPU and accelerator, saving significant cost and energy. In this scenario, we envision the host CPU only having reduced capabilities to provide storage and naming services to supply data and organize communication.

3.2. Memory Tile

Each Memory Tile is equipped with a Memory Controller (MC), a slice of HBM, a Jumbo Translation Lookaside Buffer (JTLB) for address translation and the MCPU as the central element of the Tile. In order to minimize hardware overhead and benefit from shared structures and resources, the MCPU is a fine-grain multithreaded core, a standard RISC-V RV64IMAC core without support for floating-point operations, that is biased to manage memory requests and related operations. There is a one-to-one mapping of hardware threads in the MCPU and hardware threads in the VAS tile. The bonded threads form the decoupled access/execute architecture in ACME. Additional MCPU architecture details will be determined by simulation [19].

3.2.1. MCPU memory access orchestration

The MCPU maintains a queue of outstanding memory requests for its associated memory controller. It is able to reorder those outstanding requests based on the DRAM page to be opened

and the age of the request, avoiding starvation which may produce exceptionally long latencies. These requests are enriched with meta-data indicating the destination of the data in the VAS Tile such as the cache, scratchpad or directly into the register file. Based on the access patterns from other threads and cores, there may be multiple recipients for a single memory request that the MCPU must track. In some cases, based on the spatial and temporal nature and number of recipients for a particular set of addresses, it may make sense to store the entire DRAM page in an L3 Row Buffer (RB in Fig. 4). This effectively creates a virtual DRAM open page, increasing the DRAM interleaving and memory bandwidth, while reducing the latency.

Non-unit stride memory operations to scatter or gather single vector elements from specified indices result in non-uniform access patterns which are typically performance limiting. In order to optimize this case, the MCPU implements vector index registers to make the scatter and gather operations more efficient and high performance. The goal is to provide unit stride data structures to the VAS Tiles even for sparse data structures, dramatically improving the bit efficiency by removing the parasitic bits that exist in traditional cache structures. Every byte in the vector will be consumed by the VAS Tile, whereas in a traditional cache, the worst and sometimes very common case is that only one word per cache line is consumed by the computation, although other words are moved and stored throughout the on-chip memory hierarchy vs being pruned at the memory controller. The MCPU supports this capability to only move around useful bits, saving energy and improving performance.

The MCPU enables more advanced memory request reordering, caching DRAM pages to improve access timings for multiple accesses, and coalescing non-unit stride data accesses. Moreover, further optimizations are possible based on the ability to specify and retain application data structure information like the overall vector length. Thus, the MCPU can more deeply understand how to manage the memory requests because a variety of program information is known without requiring prediction. For example, the ACME architecture can take advantage of large memory windows by providing vector memory access patterns before the application requires the data. With this knowledge, ACME can provide better memory orchestration, increasing effective memory bandwidth, and reducing unnecessary data movement on chip, especially for non-unit stride access patterns. Thus, ACME shifts the energy usage in the system based on the type of application. For dense applications that are compute bound, the MCPU utilization is very low, enabling most of the system energy to be directed to the VAS Tiles and associated accelerators. However, for sparse workloads, the MCPU utilization is high as it optimizes the memory hierarchy, shifting energy to the MCPUs and away from the VAS Tiles.

3.3. Vector and Systolic (VAS) Accelerator Tile

ACME is targeting both dense and sparse workloads which stress different aspects of the architecture. As a result, ACME adapts the number of active vector lanes per core depending on the arithmetic intensity. The VAS tile is composed of several scalar cores. Each scalar core decodes and dispatches instructions to the tightly coupled functional units and loosely-coupled coprocessor accelerators. Thus, the scalar core issues and commits all instructions as a single instruction stream, but the actual instruction execution depends on its type. Scalar instructions are executed on the scalar core, vector instructions are executed on the vector coprocessor or VPU, systolic array instructions are executed in the SA and memory operations are handled by the Memory Tile.

The VPU is connected through an Open Vector Interface (OVI) [22] to the scalar core. Instructions are forwarded, along with cache lines, over this interface to the VPU. This interface is sufficient for applications with unit stride access patterns. For applications with data that is not laid out in memory in unit stride, this interface suffers from significant performance and energy degradation, transporting useless data from the DRAM, through the cache hierarchy to the VPU and is then discarded. We call these bytes, *parasitic bytes*, consuming both energy and performance. Thus, we extend the VPU to also have its own memory interface to a shared L2 scratchpad, where data is stored in a dense representation as unit stride vectors. The hardware-managed scratchpad acts as a second level register file, enabling dynamic or virtual vector lengths with streaming support to the accelerators. By using vector length agnostic programming techniques, the application can convey application data structure details to the architecture. Not only is all data in the scratchpad useful, ACME also exterminates the *parasitic bytes*.

In addition, the SAs use the same OVI interface to the scalar core and memory interface to the scratchpad, enabling a wide variety of SA implementations, from image and video processing to neural networks. Historically, SAs are set up as memory mapped accelerators with on-chip memory regions that stream the data through the arrays with mailboxes to coordinate execution. OSH enables new instructions to be defined, leveraging the software toolchain to manage resources more effectively. Thus, we can use traditional memory and compute operations defined as custom RISC-V instructions to control and orchestrate these accelerators. Finally, the SAs share a common infrastructure even though they are computationally different. We implement a common SA shell that provides the coprocessor interface, as well as a memory interface to the L2 scratchpad to support multiple SA variants.

Each core in the VAS Tile supports up to 8 threads to hide memory latency for sparse workloads. A single-threaded mode is also supported to target dense computations using all the computational resources. This applies to kernels like GEMM or the computations performed by the SAs. The multithreaded mode targets sparse workloads that are memory-bandwidth bound to balance the memory bandwidth with the computational intensity (number of vector lanes). More threads each bound to a fusion of vector lanes, are supported to hide the memory latency. The smallest unit of vector compute is two vector lanes or a vector-lane pair. Within the VPU, multiple vector-lane pairs can be fused together, with up to 16 lanes (most useful for single-threaded dense applications). We believe this combination of multithreading and long virtual vectors enables a very large memory window to maximize HBM utilization and efficiency. In this case, we have selected a coarse-grain multithreading approach that enables the scalar cores to coordinate all the work required to support the various coprocessors. This configuration provides a lot more flexibility with regard to memory scheduling and opportunities for spatial and temporal locality versus a very long Virtual Vector Length (VVL) alone. Ideally, this also leads to a better dynamic load balance in the system.

3.4. Example SAXPY

In this section, we describe how the various components of ACME interact. For clarity, we focus on only one scalar RISC-V core in the VAS Tile and one Memory Tile as shown in Fig. 4.

For this example, we use a dense vector length agnostic SAXPY kernel example to explain the execution model. Vector instructions are forwarded to the VPU from the scalar core using the OVI, while vector memory operations are directed to the Memory Tile by the scalar core. A VAS Tile always forwards the memory operation to the same Memory Tile, to the bonded MCPU

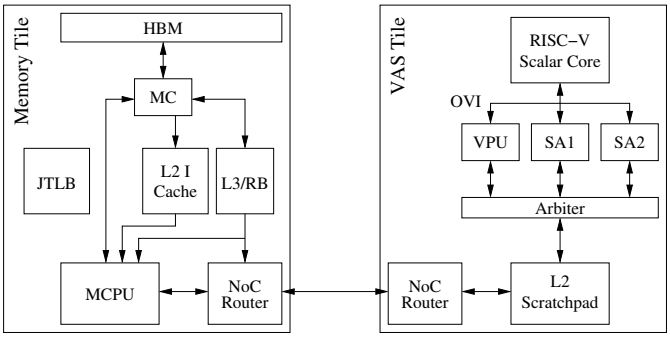

Figure 4. A simplified ACME used in the example

hardware thread. Hence, the MCPU in that tile is aware of the state of the program executed on the scalar core.

The application to be executed is $Z = aX + Y$, a function from the standard Basic Linear Algebra Subprograms (BLAS) library [3]. There are two input vectors in listing 5, X and Y, with n single precision (32 bit) floating point values each and a scalar value a. In the listing, the application vector length (AVL), n, is stored in register `a0`, the value of a is stored in `fa0`, while the elements for X and Y are located at the addresses stored in registers `a1` and `a2`, respectively.

First, a `vsetvli` instruction is executed by the scalar core and forwarded to the MCPU as a transaction. It contains the AVL from `a0`, the type of the elements (`e32`), the vector register grouping (`m8`) and the desired Virtual Vector Length (VVL). The MCPU processes the request and sets the VVL for all subsequent memory operations (until the next `vsetvli`) and acknowledges the transaction. Note, the novelty of ACME is that the VVL can be larger than the Physical Vector Length (PVL) that the VPU's 1st level Vector Register File (VRF) supports. ACME implements a dual purpose L2 memory structure that can operate as a cache and/or scratchpad. To accommodate the entire VVL of the vector load instruction (`vle`) in line 3, dedicated scratchpad memory space needs to be reserved and associated with `v0`.

As an example, consider the AVL to have 10^6 elements, while for single precision values PVL = 4. In this scenario let VVL = 16. Therefore, the vector load returns a vector containing 16 elements to be stored in the scratchpad and the opportunity to preload the first 4 elements into the VRF as PVL.

The next instruction is a vector load instruction (line 3). It instructs the scalar core, to start filling vector register `v0` with elements starting from address in `a1`. A transaction containing this instruction is forwarded to the MCPU, which starts collecting the elements from the indicated address. In case of an ACME having multiple Memory Tiles, and if the address is not in the range of the attached physical memory, transactions are created with the addresses of the missing elements and forwarded to the Memory Tiles that serve the requested addresses.

Based on the vector lengths computed earlier, the MCPU starts returning VVL elements. A scoreboard mechanism inside the VAS Tile keeps track of the data arrival and stores it into the vector registers and allocated space in the scratchpad.

Since this program is a dense workload, this load instruction represents a loading pattern with unit strides. Sparse workloads are indicated by non-unit stride or gather/scatter memory operations. In case of a load instruction, the MCPU collects the required elements and coalesces them into a dense vector. The opposite is done with stores, sending data from the scratchpad to memory. Therefore, NoC bandwidth and memory storage is preserved by avoiding handling elements, which are discarded eventually.

```
1  saxpy:
2      vsetvli a4, a0, e32, m8      ; determine VVL from AVL
3      vle32.v v0, (a1)             ; load vector
4      sub a0, a0, a4
5      slli a4, a4, 2
6      add a1, a1, a4
7      vle32.v v8, (a2)
8      vfmacc.vf v8, fa0, v0        ; Z = aX + Y
9      vse32.v v8, (a2)             ; store vector
10     add a2, a2, a4
11     bnez a0, saxpy               ; jump to next loop iteration
12     ret
```

Figure 5. The assembly source code for a RISC-V with vector instructions from the vector extension in bold

The next three instructions in Fig. 5 in lines 4–6 are scalar instructions executed on the scalar core itself, while the following vector load in line 7 is similar to the one in line 3.

The FMAC vector instruction (line 8) is forwarded to the VPU together with the scalar value in register `fa0`. The result of that operation is stored back into `v8`. This requires a coordinated streaming of input and output data for the operands in the scratchpad, and in this case a shared source and destination register.

Once the entire VVL has been computed, the `vfmacc` instruction is retired. Since a store instruction follows, a transaction with the data in the scratchpad region associated with `v8` is issued to the Memory Tile, which then moves the data back into the HBM.

This concludes one iteration of the loop, in which a part of the entire application vector has been computed. In case of AVL > VVL multiple iterations of the loop are executed. Since the MCPU is aware of AVL as well as VVL, due to the `vsetvli` instruction initially sent to the MCPU, multiple repeating memory operations with different offsets are expected. Therefore, while the VPU computes the results for the current iteration, the MCPU is able to preload the next vector with VVL elements and temporarily store it in its L3 RB to have it ready as soon as the first load instruction of the next iteration arrives at the Memory Tile. Hence computation and memory access are parallelized.

ACME offers many opportunities to further optimize the architecture. For instance, a scaled down version of the program executing on the scalar core may be executed on the MCPU similar to [11] or [25]. However, this is part of the future work and exploration.

3.5. Simulating ACME

The development of new architectures involves high level decisions that determine the general flavor of the design. These decisions, which include details such as core counts or the overall cache hierarchy, should be made early, via simulation, so later FPGA-based analysis can focus on what it should: lower level details. In the case of ACME, more aggressive design points require early investigation before implementation, like the proposed novel data handling schemes of the MCPU and its automatically managed scratchpad. However, simulation of HPC-capable architectures is still time-consuming, due to the size of the workloads and number of cores that need to be simulated in order to fully exercise the modelled system. This is at odds with the very purpose of

early simulation, which is performing large amounts of executions to identify certain parameters of the system and evaluate new ideas.

For the reasons above, MEEP proposes Coyote [19], a new execution-driven simulator for the RISC-V ISA. Coyote strikes a balance between simulation throughput and fidelity by focusing on the modelling of the data movement throughout the memory hierarchy. This captures the right amount of detail to enable the comparison of different designs while preventing simulation times from growing beyond the limits of reasonable design space exploration. To leverage previous community efforts, Coyote is based on two pre-existing simulators: Spike [21] and Sparta [23]. Spike, as the open-source golden-standard for RISC-V, provides the capabilities for functional simulation, including the vector extension and also the modelling of L1 caches. Additionally, it has been extended to support other features such as RAW dependency tracking and different instruction latencies. Sparta, a framework to build event-driven simulators developed by SiFive, sits on top of Spike to provide the modelling capabilities for the memory hierarchy of the system, which is exercised by events generated by the execution of simulated applications in Spike.

In order to reduce the overhead of the interaction between the two pieces that build Coyote, they have been integrated as slaves into a *simulation orchestrator*. It is in charge of managing the communication of events between the functional and the event-driven side of Coyote and keeping the clocks synchronized. In every cycle, an instruction is simulated in each of the simulated active cores. The simulation of an instruction might have different effects:

- A RAW dependency might be detected in the registers that are read. In this case, the core is marked as inactive. It will not be able to execute again until after the dependency is satisfied.
- An event that needs to be communicated to the Sparta event-driven engine might be generated. This includes requests to the L2, interactions with the MCPU and more.

After the simulation of instructions, events are submitted to Sparta. Then, the Sparta clock is advanced and events are handled. Submitting an event to Sparta will internally generate a chain of events that will correctly exercise the modelled memory hierarchy and interact with the MCPUs. The end result of this chain of events is usually an interaction back to the simulation orchestrator, that will update certain values in Spike. An example of this is marking a RAW dependency as satisfied when a memory access submitted to the L2 is satisfied. The handling of the different types of events has been implemented following the visitor design pattern, for easy maintainability and extensibility.

Coyote models tiled architectures similar to ACME. This includes core-private L1s and a slice of banked L2 per tile. The L2 can be configured as private to the cores that belong to a tile or fully shared across the system, forming a NUCA. A basic memory controller, which implements a subset of memory commands and different address mapping policies. The NoC connecting the tiles and memory controllers is modelled implementing three different levels of accuracy, ranging from a very simple model based on the average number of cycles to travel through the network to a detailed one that is based on the well-known Booksim simulator [13] (integrated also as a slave to the simulation orchestrator). Many of the parameters regarding the sizing and timing of each of the components can be configured through parameters in order to evaluate the behavior of different configurations.

With respect to applications, Coyote executes baremetal applications compiled using either the standard GNU compiler for RISC-V or the EPI compiler, if vector intrinsics are used. As a result, it produces general statistics related the use of the memory hierarchy, such as the miss rate

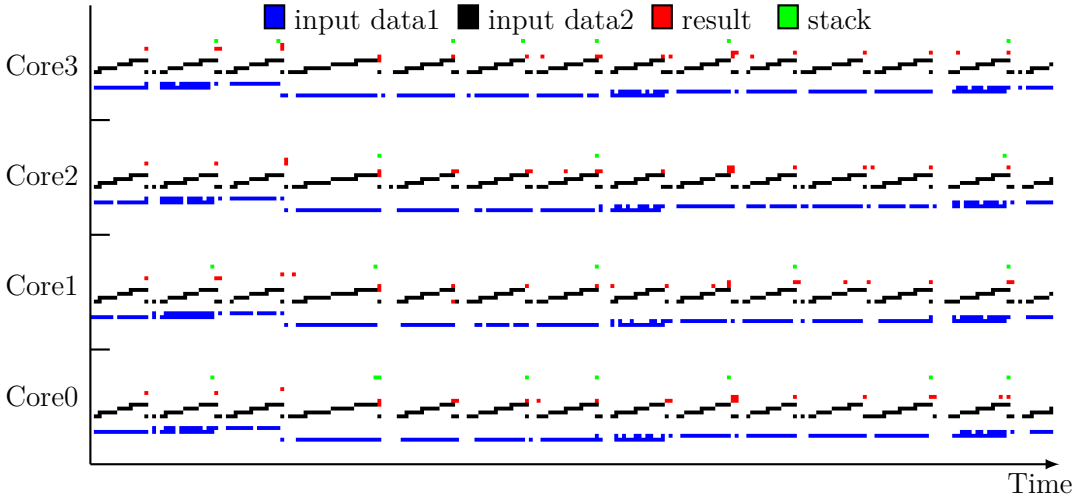

Figure 6. Paraver L2 miss trace for a 4 core matrix multiplication. Different accessed data structures have different colors

M3	V3	V7	V11	V15	M7
M2	V2	V6	V10	V14	M6
M1	V1	V5	V9	V13	M5
M0	V0	V4	V8	V12	M4

(a) Fully shared cache forming a NUCA

M3	V3	V7	V11	V15	M7
M2	V2	V6	V10	V14	M6
M1	V1	V5	V9	V13	M5
M0	V0	V4	V8	V12	M4

(b) Tile with private L2 caches

Figure 7. Heatmap (the lighter the shade, the sparser the traffic) for the destination of packets (M – Memory Tile, V – VAS Tile) in a matrix multiplication

per L2 slice, the number of stalls in the L2 due to MSHR exhaustion or the amount of requests serviced by each memory bank. Coyote can also produce a trace that can be visualized using paraver [20] to identify patterns and analyze the behavior of applications in detail. As an example, Fig. 6 shows a portion of an L2 miss trace for a vector matrix multiplication using 4 simulated cores. Misses for different data structures have different colors, which eases the behavioral analysis and and helps to identify patterns. Regarding the NoC, Coyote also produces a heat map to easily identify hotspots, which are potentially related to inefficient data management. Figure 7 compares two heatmaps for a vector matrix multiplication simulated on a 16 VAS Tile and 8 Memory Tile configuration. Memory Tiles are placed in the first and last columns, for a scaled down version of the proposed ACME layout shown in Fig. 2. Both heatmaps are for the execution of the exact same application and only differ regarding how the L2 is shared, either forming a NUCA across all the tiles (Fig. 7a) or private to each tile (Fig. 7b). This figure shows that the NUCA generates a lot more traffic indicated by the darker shades, among the VAS tiles, while relieving the memory tiles (lighter shades).

As a result of its modelling capabilities and its overall design, which strikes a balance between fidelity and simulation throughput, Coyote can provide insight into how high level design decisions

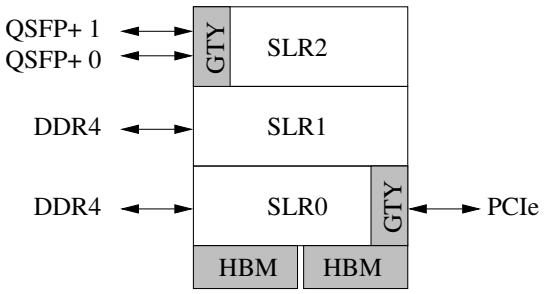

Figure 8. The Xilinx Virtex Ultrascale+ VU37P with HBM combines three Super Logic Regions (SLR) and HBM to a System-in-Package (SiP). The shaded blocks are part of the MEEP Shell

impact the performance of HPC architectures. This enables the evaluation of different designs prior to more costly FPGA implementation and emulation.

4. FPGA

FPGAs enable MEEP to provide silicon validation capabilities. MEEP consists of 12 nodes comprising 8 Xilinx Virtex Ultrascale+ FPGAs [31], each able to establish a dense interconnect among the FPGAs. Those FPGAs exhibit up to 500 Mb of total on-chip integrated memory, in addition to up to 16 GB of HBM. With these characteristics, MEEP is a foundation for hardware emulation and SDV for other projects as eProcessor [8], and can be exploited in projects such as DRAC [15] or OpenPiton [7].

Xilinx Virtex Ultrascale+ FPGAs are built on top of a silicon interposer, connecting three Super Logic Regions (SLRs) together in the same package (System in Package, SiP). The connections among the SLRs are silicon based vias labeled as Super Logic Lines. This technology allows the FPGA part to exceed traditional sizes. They may be thought as three smaller FPGAs fused together in the same silicon forming a new larger device (refer to Fig. 8).

These FPGAs contain up to 1.3M LUTs, 260k Flip-Flops (FF), 2k BRAMs (36 Kbits) and 9k DSPs, among other common elements. In addition, the FPGAs contain integrated hard IPs such as PCIE4C (PCIe Gen4), CMAC (100 Gbit Ethernet), and memory controllers for the HBM and DDR RAM. A common collection of I/O and memory interfaces are implemented for these building blocks, which facilitates the use of MEEP as a silicon validation platform. These interfaces define the MEEP FPGA Shell, which is the foundation for an engine to enable the SDV and pre-silicon validation cycle.

4.1. FPGA Shell

The FPGA implementation is composed of two main components: the MEEP FPGA Shell and the emulation region for accelerators and/or CPUs. The MEEP FPGA Shell is a static, customizable perimeter architecture which guarantees that the emulation region remains portal/interchangeable across any other FPGA package or device that meets the defined I/O and memory interface specifications. The MEEP FPGA Shell goes beyond Xilinx' approach, which consists of multiple proprietary platforms, which may not meet the demands of the accelerator. Instead the MEEP FPGA Shell is fully customizable by the user to match a given FPGA architecture. On top of that, the MEEP Shell provides QDMA, FPGA-to-FPGA and Ethernet solutions, which Xilinx platforms lack. One example for an accelerator is ACME (refer to Section 3), but others can be considered as well as shown in Section 4.2.

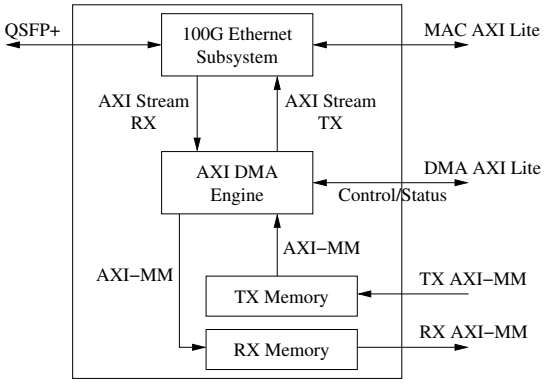

Figure 9. The MEEP FPGA Shell Ethernet IP. It can be used as long as the accelerator is compliant with the specified interfaces

The basic components of the MEEP FPGA Shell are PCIe, HBM, 100 Gb Ethernet, and Aurora for raw FPGA-to-FPGA communication. A DDR4 controller can also be included, which is used to simulate an envisioned Non-Volatile Memory (NVRAM), which is part of a future system beyond the scope of this project. The Shell is generated through a TCL-based script which lets the user create a custom MEEP FPGA Shell configuration, followed by the actual building process, parsing the emulated accelerator and checking that all the desired connections between the MEEP FPGA Shell and the accelerator engine are instantiated.

For the experiments, the Alveo U280 featuring the Xilinx Ultrascale+ VU37P FPGA [30] will be used to implement the MEEP FPGA Shell as well as emulate the ACME accelerator.

4.1.1. PCIe interface

As part of the MEEP FPGA Shell, PCIe implements the communication interface between the host and the emulated hardware. PCIe is already included as a hard IP in the VU37P FPGA, resulting in a reduction of resource requirements when compared to pure soft-logic PCIe implementations. Xilinx supports two different types of PCIe blocks, XDMA and QDMA. The Xilinx platforms or Amazon F1 utilize the XDMA block. However, MEEP FPGA Shell focuses its efforts on the QDMA implementation for flexibility and streaming capabilities.

The main difference between QDMA and other DMA offerings is the concept of queues, derived from the *queue set* concepts of Remote Direct Memory Access (RDMA) from high-performance computing (HPC) interconnects. These queues can be individually configured by the interface type. Based on how the DMA descriptors are loaded for a single queue, each queue provides a very low overhead option for continuous update functionality. The QDMA solution supports multiple Physical/Virtual Functions with scalable queues, and is ideal for applications that require small packet performance at low latency, as well as streaming data. QDMA offers more functionality at slightly higher resource costs with two advantages: First, it implements a high performance configurable scatter/gather DMA for the PCIe IP. Second, the IP provides an AXI4-Stream user interface in addition to AXI Memory Mapped and AXI-Lite. QDMA can be set to PCIe Gen3 or Gen4. In both cases, the peak bandwidth is 16 GB/s in each direction.

4.1.2. HBM interface

The HBM is the high performance DRAM interface that massively increases system memory bandwidth using SiP technology [27]. HBM enables maximum bandwidth, efficient routing and

Table 1. The clock frequencies and LUT utilization for various MEEP Shell IPs

Shell IP	Frequency [MHz]	User Clock	Resources [LUT]
PCIe (QDMA)	250	Fixed	70376
HBM	≤ 450	Maximum	1539
Ethernet	322.26	Fixed	7444
Aurora	≤ 402.23	Maximum	1500
Aurora (DMA Mode)			4849
DDR4	300	Fixed	18823

logic utilization, and optimizes power efficiency for workloads that process large datasets. The MEEP FPGA Shell provides a standard interface to HBM. An HBM stack of four DRAM dies has two 128 bit channels per die for a total of 8 channels and a width of 1024 bits [29]. In comparison, the bus width of GDDR memories is 32 bits, with 16 channels and a 512 bit wide memory interface. Memory bandwidth for Xilinx Virtex Ultrascale+ FPGAs with HBM is up to 460 GB/s delivered by two stacks of Samsung HBM2 with 8GB each. There is the flexibility to access pairs of pseudo channels or aggregate all the pseudo channels across two stacks of HBM together as a large memory access engine with much larger memory bandwidth and larger granularity. In the context of the accelerator, HBM related logic can be clocked at up to 450 MHz.

4.1.3. Aurora interface

The MEEP FPGA Shell can be configured to use of GTY transceivers [28] for point-to-point FPGA communication. Xilinx' Aurora IP wraps up the transceivers and encodes a 64b/66b protocol to provide enough state changes to allow a reasonable clock recovery and alignment of the data stream at the receiver. This allows easy adoption of custom protocols inside the accelerator which only needs to encode its own protocol in a 64 bit wide bus and assert a valid signal once the data is ready. The accelerator can also rely on a DMA based solution. In this scenario, the accelerator needs to configure the DMA to program transactions from the memory. This communication interface is full-duplex, so the same principles apply for the receiver (RX) and transmitter (TX). On the RX side, the implementation must be designed carefully though, as there is no back-pressure capability and it must be ready to consume any incoming data. In the context of MEEP, the user logic associated to the Aurora interface has been validated working at frequencies of up to 402.832 MHz.

Xilinx transceivers are composed of four lanes in a configuration named *Quad* sharing a common reference clock. With this configuration, each of the four lanes can be used to create different topologies such as torus, hypercube, etc.

4.1.4. 100 Gb Ethernet interface

The MEEP FPGA Shell also includes a 100 Gb Non-Return-to-Zero (NRZ) Ethernet solution shown in Fig. 9. It is built around the hard IP CMAC [32] and is present in the FPGA. This solution creates a ready-made interface to grant Ethernet functionality between the accelerator and any other device in the network, including an Ethernet switch. The accelerator only needs to

be compliant with several AXI interfaces in order to set-up, control and make use of the Ethernet IP. The associated Ethernet user logic is clocked at 322.266 MHz.

4.1.5. DDR4 interface

DDR4 is offered in the MEEP FPGA Shell either as main memory or it can be used to emulate NVRAM or other hybrid memory architectures with an order of magnitude higher effective capacity compared to HBM [12]. Current technology couples HBM with NVRAM as demonstrated by Intel [6, 33]. The accelerator can leverage this MEEP FPGA Shell feature to emulate such systems.

4.1.6. MEEP FPGA Shell details

In Tab. 1, a summary of the clocks associated with each of the MEEP FPGA Shell IPs is shown. Depending on the design, the maximum frequencies allowed for the HBM and Aurora may not be achieved by the other IPs, forcing the HBM and Aurora to be clocked down. The frequencies for the PCIe, Ethernet, and DDR4 are fixed by the specifications and cannot be configured by the user. As shown in the table, because the MEEP FPGA Shell leverages hard IPs, it only requires approximately 100k LUTs in total, which amounts to 8% of the overall resources available in the VU37P FPGA. The remaining resources can be used for the accelerator implementation.

4.2. Other Accelerator Examples Mapped to MEEP

The path to a fully functional ACME accelerator emulated in MEEP is in the near future. However, there are several intermediate milestones that demonstrate the utility and capabilities of MEEP. We present two different RISC-V IPs embedded in a user configurable MEEP FPGA Shell. Due to the standardized interfaces, the integration reduces time for implementation and validation, while the use of hard IP minimize the resource requirements of the MEEP FPGA Shell. Thus, the MEEP FPGA Shell can be used as a foundation to test multiple accelerators, processors or other IPs with varying configurations. The projects such as OpenPiton [7] and DRAC [15] are both aligned with the concept of the accelerator. OpenPiton is scalable in terms of cores. On the other hand, DRAC is a design which implements the scalar core of Lagarto and a VPU with two lanes. Both have been packaged as two different accelerators using the MEEP FPGA Shell as a demonstrator for the combination of the Shell and the accelerator. These projects serve two purposes: enabling a general MEEP FPGA Shell that is reusable across multiple projects and exploring a baseline IP to extend for ACME.

Embedding the accelerator into the MEEP FPGA Shell is straightforward, since only the interfaces the Shell offers have to match (refer to Fig. 9), avoiding any other configuration overhead.

In both cases, the MEEP FPGA Shell has been configured by the user to contain a PCIe QDMA and HBM without Aurora or Ethernet connectivity. The resource utilization in this configuration is 5% of the total FPGA resources (compared to Tab. 1). Likewise, OpenPiton has been mapped into the MEEP FPGA Shell as a many-core system consisting of four Ariane RISC-V cores (RV64GC). Fig. 10 reveals that this OpenPiton configuration requires 28% of the overall LUTs available on the FPGA. In comparison, DRAC with its Lagarto RISC-V core, including its 2-lane VPU requires 18% of the FPGA.

The use of the Shell significantly reduces implementation time during the accelerator development, since all Shell features are supported out of the box, with a proven validated set of

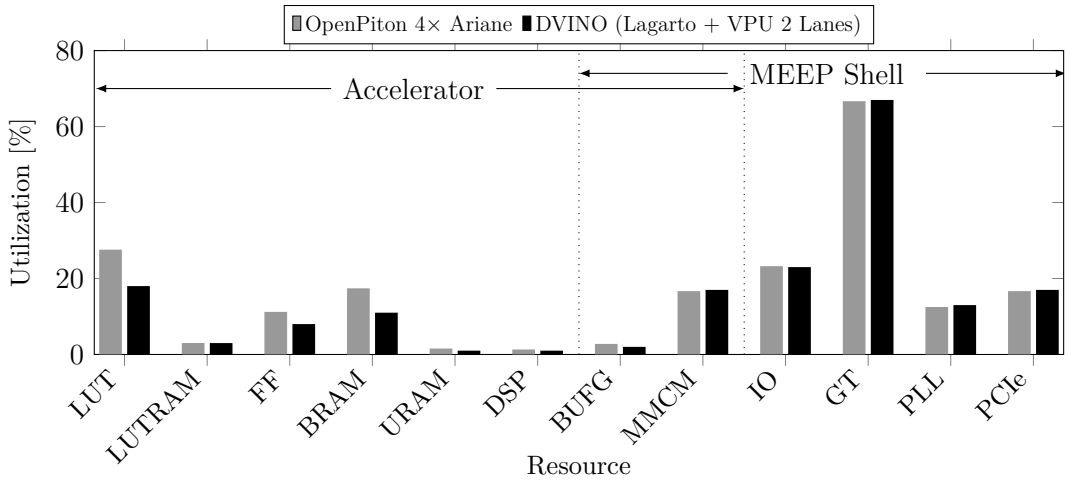

Figure 10. Utilization report for OpenPiton in a many-core (4 RV64GC Ariane cores) and DRAC (Lagarto and a 2-lanes VPU coprocessor)

interfaces. Connecting OpenPiton and DRAC to the Shell worked as expected the first time they have been connected, demonstrating one of the main goals of MEEP, which is to provide a solid hardware environment for software development and silicon validation.

As it is evident from Fig. 10, the selected accelerator does not have any impact on the I/O, GT, PLL, nor PCIe resource requirements, while BUFG and MMCM are affected marginally. The resources depicted on the right side in the figure belong to the MEEP FPGA Shell, which is the same for both accelerators. Therefore, the limiting factor in embedding a different accelerator is the availability of FPGA resources mentioned on the left side in the figure, mainly the LUTs and BRAM.

5. Related Work

Several platforms targeting simulation accelerators and small-scale SDVs precede MEEP. However, they are either closed source or do not support OSH and the related ecosystem. The PROTOFLEX [9] project is an FPGA-based architecture to accelerate full-system multiprocessor simulation and to facilitate high-performance instrumentation. It is not a specific simulator itself, but a template that offers a set of practical approaches for developing FPGA-accelerated simulators. It virtualizes the execution of many logical processors onto a consolidated number of multiple-context execution engines in the FPGA, which can be scaled, as needed, to deliver the necessary simulation performance at large savings in complexity.

Another example is DIABLO [26], a cost-efficient FPGA-based emulation methodology, which treats FPGAs as whole computers with tightly integrated hardware and software. DIABLO is not based on FPGA prototyping, but uses FPGAs to accelerate parameterized abstract performance models of the system instead. DIABLO is fully parameterizable and fully instrumented, and supports repeatable deterministic experiments. Most of the efforts on its model are focused on networking analysis and optimizations.

The MANGO [10] project is an FPGA-based many-core emulation platform. It facilitates the exploration of heterogeneous accelerators for being used in HPC systems running multiple applications with different Quality of Service (QoS) levels. To make this possible, MANGO has explored different but interrelated mechanisms across the architecture and system software. The

platform provides a large-scale cluster of multiple FPGA boards intended for experimenting with customized many-core systems, at the level of both processor and interconnect/system architecture, along with the supporting software stack.

Mont-Blanc [2] was a predecessor of the European Processing Initiative (EPI) [4] relying on the results of three consecutive projects (i.e., Mont-Blanc 1, 2, and 3), which explored the viability of using highly energy efficient ARM-based processors for HPC. The main objective of Mont-Blanc 2020 was to start developing building blocks (IPs) for an HPC processor.

Conclusion and Future Work

A large community is required to develop the OSS and OSH ecosystem. We must build tools and systems to enable independent development and exploration of software and hardware; one cannot exist without the other. MEEP is a digital laboratory that enables RISC-V ecosystem development for the HPC and HPDA domains based on a large-scale platform based on composable and reuseable IP blocks. In order to rapidly move forward in the RISC-V ecosystem, we need tools in the community to support software and hardware development. MEEP provides the former as a large-scale SDV and the latter as a pre-silicon hardware emulation platform. Combined, MEEP can realize a future vision HPC and HPDA accelerators in the form of the ACME architecture and demonstrate this full stack.

We have implemented the first phase of the MEEP platform using four Xilinx Alveo U280 [30] cards and have focused on the base infrastructure for the MEEP FPGA Shell, and a variety of initial OSH cores, and multiprocessor SoCs. We see a bright future for the RISC-V ecosystem in HPC and HPDA and plan to continue to develop and contribute using MEEP.

Acknowledgments

This work has been supported by the EU H2020 project MareNostrum Experimental Exascale Platform (MEEP), and funded by the European Commission under the grant agreement No. 946002.

References

1. Fact Sheet & Background: Roadrunner Smashes the Petaflop Barrier (2008), `http://www-03.ibm.com/press/us/en/pressrelease/24405.wss`

2. The Mont-Blanc Project (2020), `https://www.montblanc-project.eu/`

3. Basic Linear Algebra Subprograms (2021), `http://www.netlib.org/blas/`

4. The European Processor Initiative (2021), `https://www.european-processor-initiative.eu/`

5. Abadi, M., Agarwal, A., Barham, P., et al.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), `https://www.tensorflow.org/`

6. Altman, A., Arafa, M., Balasubramanian, K., et al.: Intel Optane Data Center Persistent Memory. In: 2019 IEEE Hot Chips 31 Symposium (HCS), 18-20 Aug. 2019, Cupertino, CA, USA. pp. i–xxv. IEEE (2019), DOI: 10.1109/HOTCHIPS.2019.8875668

7. Balkind, J., McKeown, M., Fu, Y., et al.: OpenPiton: an open source hardware platform for your research. Commun. ACM 62(12), 79–87 (2019), DOI: 10.1145/3366343

8. BSC: eProcessor: European, extendable, energy-efficient, energetic, embedded, extensible, Processor Ecosystem (2021), https://www.bsc.es/research-and-development/projects/eprocessor-european-extendable-energy-efficient-energetic-embedded

9. Chung, E.S., Nurvitadhi, E., Hoe, J.C., Falsafi, B., Mai, K.: PROToFLEX: FPGA-accelerated Hybrid Functional Simulator. In: 2007 IEEE International Parallel and Distributed Processing Symposium, 26-30 March 2007, Long Beach, CA, USA. pp. 1–6. IEEE (2007), DOI: 10.1109/IPDPS.2007.370516

10. Flich, J.: MANGO: Exploring Manycore Architectures for Next-GeneratiOn HPC Systems. In: 2017 Euromicro Conference on Digital System Design (DSD), 30 Aug.-1 Sept. 2017, Vienna, Austria. pp. 478–485. IEEE (2017), DOI: 10.1109/DSD.2017.51

11. Hashemi, M., Mutlu, O., Patt, Y.N.: Continuous runahead: Transparent hardware acceleration for memory intensive workloads. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 15-19 Oct. 2016, Taipei, Taiwan. pp. 61:1–61:12. IEEE Computer Society (2016), DOI: 10.1109/MICRO.2016.7783764

12. Izraelevitz, J., Yang, J., Zhang, L., et al.: Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. CoRR abs/1903.05714 (2019), http://arxiv.org/abs/1903.05714

13. Jiang, N., Becker, D.U., Michelogiannakis, G., et al.: A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator. In: 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 21-23 April 2013, Austin, TX, USA. pp. 86–96. IEEE (2013), DOI: 10.1109/ISPASS.2013.6557149

14. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization, 20-24 March 2004, San Jose, CA, USA. pp. 75–86. IEEE (2004), DOI: 10.1109/CGO.2004.1281665

15. Leyva-Santes, N.I., Perez, I., Hernández-Calderón, C.A., et al.: Lagarto I RISC-V Multi-core: Research Challenges to Build and Integrate a Network-on-Chip. In: Supercomputing, 25-29 March 2019, Monterrey, Mexico,. pp. 237–248. Springer, Cham (2019), DOI: 10.1007/978-3-030-38043-4_20

16. Lordan, F., Tejedor, E., Ejarque, J., et al.: ServiceSs: An Interoperable Programming Framework for the Cloud. Journal of Grid Computing 12, 1–25 (2013), DOI: 10.1007/s10723-013-9272-5

17. Message Passing Interface Forum: MPI: A message-passing interface standard. Version 3.1 (2015), https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, accessed: 2021-04-23

18. OpenMP Architecture Review Board: OpenMP Application Programming Interface 5.1 (2020), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf, accessed: 2021-04-23

19. Perez, B., Fell, A., Davis, J.D.: Coyote: An Open Source Simulation Tool to Enable RISC-V in HPC. In: Design, Automation, and Test in Europe, DATE (2021)

20. Pillet, V., Labarta, J., Cortes, T., Girona, S.: PARAVER: A Tool to visualize and analyze parallel Code. WoTUG-18 44 (1995)

21. RISCV: The Spike RISC-V ISA Simulator (2021), https://github.com/riscv/riscv-isa-sim

22. Semidynamics: Open Vector Interface (2021), https://github.com/semidynamics/OpenVectorInterface

23. Si-Five: The Sparta Framework (2021), https://github.com/sparcians/map/tree/master/sparta

24. Smith, J.E.: Decoupled Access/Execute Computer Architectures. ACM Trans. of Computer Sys. 2(4), 289 (1984)

25. Srinivasan, V., Chowdhury, R.B.R., Rotenberg, E.: Slipstream Processors Revisited: Exploiting Branch Sets. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 30 May-3 June 2020, Valencia, Spain. pp. 105–117. IEEE (2020), DOI: 10.1109/ISCA45697.2020.00020

26. Tan, Z., Qian, Z., Chen, X., et al.: DIABLO: A Warehouse-Scale Computer Network Simulator Using FPGAs. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, 14-18 March 2015, Istanbul, Turkey. pp. 207–221. ACM, New York, NY, USA (2015), DOI: 10.1145/2694344.2694362

27. Wissolik, M., Zacher, D., Torza, A., Day, B.: Virtex UltraScale+ HBM FPGA: A Revolutionary Increase in Memory Performance (2019)

28. Xilinx: UltraScale Architecture GTY Transceivers (2017)

29. Xilinx: AXI High Bandwidth Memory Controller v1.0 LogiCORE IP Product Guide (2019)

30. Xilinx: Alveo U280 Data Center Accelerator Card (2020), https://www.xilinx.com/products/boards-and-kits/alveo/u280.html

31. Xilinx: Virtex UltraScale+ (2020), https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html

32. Xilinx: UltraScale Devices Integrated 100G Ethernet Subsystem v2.6 (2021)

33. Yang, J., Kim, J., Hoseinzadeh, M., et al.: An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. CoRR abs/1908.03583 (2019), http://arxiv.org/abs/1908.03583

34. Zaharia, M., Xin, R.S., Wendell, P., et al.: Apache Spark: A unified engine for big data processing. Communications of the ACM 59(11), 56–65 (2016), DOI: 10.1145/2934664