MEEP

MareNostrum Experimental Exascale Platform

D6.1 Emulation platform specification

Version 1.1

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), Daniel Jiménez (BSC), Alexander Fell (BSC) and Teresa Cervero (BSC)
Contributors	Antonio Filgueras (BSC)
Reviewers	Xavier Martorell (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), Daniel Jiménez (BSC), Alexander Fell (BSC) and Teresa Cervero (BSC)	Initial draft
V 1.1	Xavier Martorell (BSC)	Review

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.

Table of Contents

1. Executive Summary

This document is part of a collection of documents that describe the MEEP project. Work Package (WP) 4 describes the target Emulated Accelerator (EA) architecture and the resulting RTL. WP5 describes the software stack that runs on top of the EA provided by WP4. This document reports the activities planned in MareNostrum Experimental Exascale Platform (MEEP) for WP6, under task T6.1 on the FPGA emulation platform. WP6 combines the WP4 and WP5 components to run on the FPGA emulation platform described in this document and by the following task description:

> **Task 6.1 Platform definition document and acquisition (M1-M6)** In this task, a platform definition document will be prepared, which motivates the selected FPGA hardware platform to be used for emulation, together with the HW specs for the FPGA implementation of the RTL, the FPGA shell and other required IP, including the necessary IPs, with a specific plan to the acquisition of each. A few options are High Bandwidth Memory (HBM) enabled FPGA cards, or Open Compute Accelerator Module (OAM) cards. The available IPs will be evaluated from the open source community, as well as from vendors.

This report details the contributions from the MEEP WP6 partner Barcelona Supercomputing Center (BSC), with input from UNIZG and TUBITAK for FPGA Emulation Platform design and development based on the MEEP Exascale Accelerator Architecture and RTL to be mapped on to this FPGA emulator, to be used as Software Development Vehicle (SDV) and pre-silicon validation platform. The primary purpose of this document is to describe the FPGA internal RTL used to support the emulator and the related software components on the host that enable communication.

The document is structured as follows:

In Section 2, we briefly describe the MEEP platform 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 of the project. This is a software/hardware co-design project where the result of the co-design is implemented on the FPGA. This enables 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 the MEEP system with a focus on the software stack that exists in the Host server and FPGA to enable emulation. This covers the host interface, FPGA shell or core logic that provides basic emulator services, and the accelerator implementation.

In Section 5, we discuss the use of the MEEP infrastructure in other projects, ranging from other accelerators and processors to accelerators targeting FPGA implementations.

In Section 6, we provide the timeline and deliverables for the MEEP project of the FPGA emulator. The first phase focuses on a single FPGA demonstration. The second phase demonstrates the full system (up to 10 nodes, host servers plus eight FPGAs, running applications.

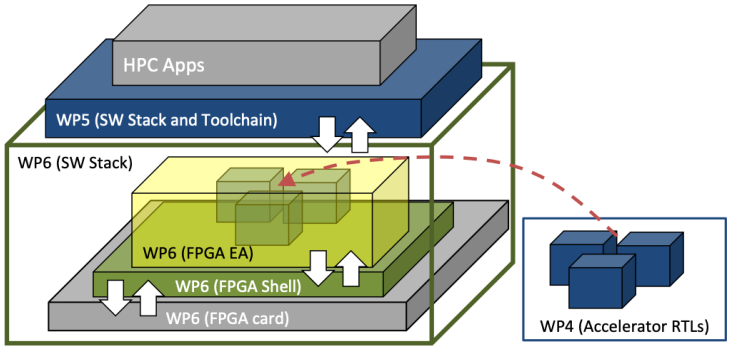

Figure 1. The relationship of the different work packages, software (WP5), RTL and Architecture (WP4), and FPGA emulator (WP6).

2. MEEP Platform 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 projects like 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. From a high-level perspective, MEEP is defined by the layered structure depicted in Figure 2, where the colored layers (Full Stack Exascale Design and FPGA Emulated HW) are the areas of interest for this deliverable. In this sense, 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.

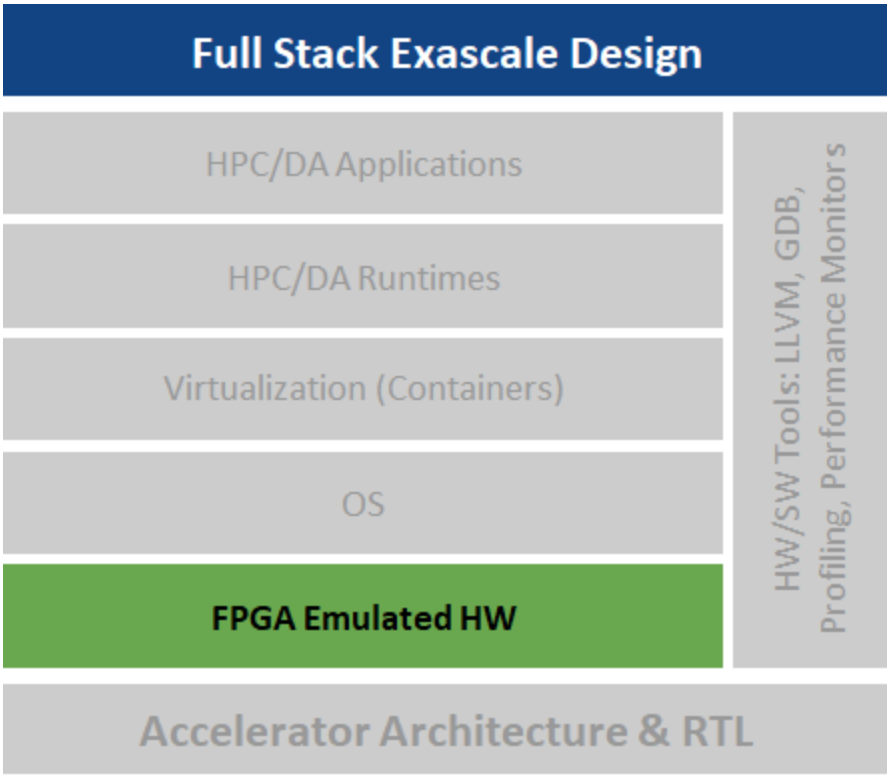

Figure 2. MEEP Full Stack Exascale in relationship to the FPGA Emulator. The focus of this document.

Figure 2 not only depicts the layered structure of MEEP, but also the relationship among these layers. In the end, HPC applications have to be run on the FPGA Emulated HW, which is implemented on a physical infrastructure. In this sense, FPGA Emulated HW might be seen as a logical layer that allows a seamless adaptation between real characteristics of the subjacent physical infrastructure, and the target accelerator under study. At the same time, this co-design approach is supported by some kind of hardware and software tools. In order to deal with this complexity, MEEP groups these layers in three main operative/functional work packages (WPs).

- Software ecosystem (Layers 2 to 5, moving from the top-down; and vertical one) is grouped under WP5. For more details, we recommend reading *D5.1 Application Suite and Software Stack* document.
- Targeted accelerator details (bottom layer) are set under WP4. More information might be read in *D4.1 MEEP Accelerator Architecture*.
- Physical infrastructure and the FPGA emulated hardware (layer 6) are collected as part of WP6. On one hand, physical hardware infrastructure detailed at the end of this document in section 4.4. The majority of this document outlines the design of the FPGA Software Stack for MEEP that supports the applications used for the co-design approach. This document is influenced by the architectural efforts of WP4 and the ability to map the HPC/HPDA software stacks on to the emulator created in WP5.

3. Software Stack Overview

The architecture work package, WP4, is defining an exascale accelerator architecture for HPC and HPDA applications. The main goal is to demonstrate a path to high performance and energy efficient acceleration. One of the big challenges to overcome is to reduce the data movement that exists in current accelerated compute infrastructure. Data is stored on the host server and marshalled to/from accelerators through the host. Significant energy is consumed because of the data movement. MEEP allows us to re-examine where data resides in the system and how to reduce data movement and the associated energy overhead. The FPGA software provides the mechanisms to build an emulated system.

The priority of MEEP is to run traditional HPC workloads and ecosystems. Furthermore, we see a broader set of applications with high compute requirements in new High Performance Data Analytics (HPDA) AI/ML/DL frameworks. We 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 class HPC synthetic benchmarks like Linpack and HPCG, as well as applications like GROMACS and QuantumEspresso. MEEP will add other workloads from AI/ML/DL benchmarks, running in frameworks like COMPS, TensorFlow and Apache Spark. Figure 2, above, shows how the FPGA emulator is the target for the Accelerator Architecture and RTL, which runs the software stack on top. Over the course of the project, we have the flexibility to map the logical design onto the physical hardware, both FPGA and host CPUs. The focus of this document is the software on the FPGA and in the system that enables the mapping of the MEEP accelerator, enabling the software development vehicle and pre-silicon validation work to be done.

The emulator software stack is composed of a few different components: the host or CPU side code and the base FPGA RTL required as infrastructure for any system. On the host side, we need to define the communication mechanisms and memory management for the system. We also have the flexibility to leverage the host cores for work in the emulated accelerator as well, if necessary. We will also detail the FPGA RTL and components required to abstract away the FPGA features into high level features that MEEP or any other project can use for their implementation.

4. MEEP System

The development of the MEEP system described in this document focuses on the software components and stack and the hardware platform for deployment of the emulation platform. This project is divided into two major phases. The first phase implements MEEP features in the context of a single FPGA. This demonstrates the initial functionality and some of the key features available in MEEP. The second phase, developed concurrently, implements the large scale emulator that enables MEEP to run applications at a system, multiple nodes, level. The final MEEP system will be based on a collection of the nodes shown in Figure 3, below.

Each node is composed of a host server and eight FPGAs in OAM (Open Compute Accelerator Module) [4] packages with a fully connected topology on a Universal Base Board (UBB) PCB, as shown in Figure 3, below. The host server is a standard dual socket x86 system, yet to be defined, but most likely AMD based. From a practical perspective, the host will either be an Intel or an AMD CPU-based server, as those are the CPUs that are supported with software drivers by the FPGA manufacturers. A single node in MEEP is composed of a dual socket, x86 host server and 8 OAM FPGAs on a UBB connected via PCIe. The FPGA shown on the right of Figure 3, is an FPGA with HBM memory with a green outer FPGA shell and an inner region devoted to the Emulated Accelerator (EA) or other functionality.The target for MEEP is to build a system with ten nodes to demonstrate the applications at sufficient scale.

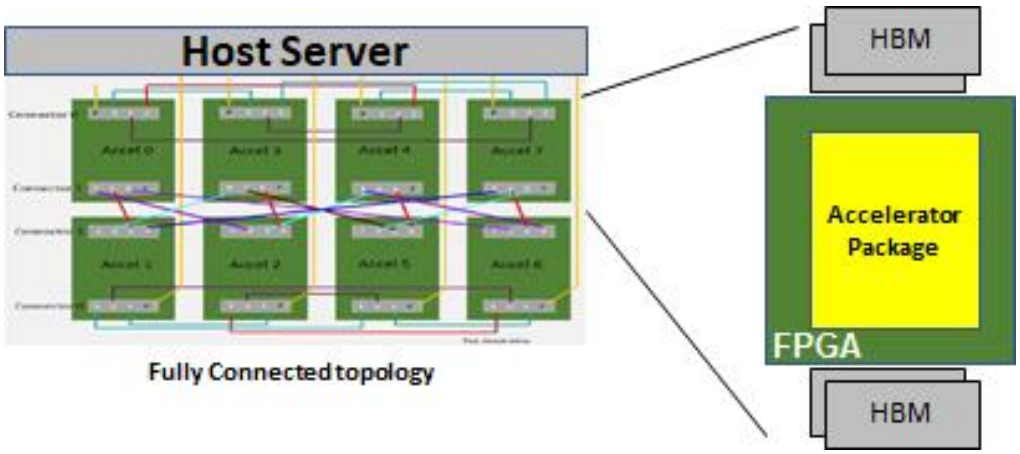

Figure 3. A single node in MEEP is composed of a dual socket, x86 host server and 8 OAM FPGAs on a UBB connected via PCIe. Higher level detail of the FPGA shows an FPGA with HBM memory (on interposer) with a green outer FPGA shell and an inner region devoted to the emulated accelerator or other functionality.

MEEP has a lot of connectivity potential that needs to be exposed. The host server is connected by PCIe to each of the FPGAs (host interface) and all FPGAs are fully connected with dedicated SerDes (Serializer/Deserializer) links, enabling all-to-all communication between the FPGAs or emulated accelerators. Furthermore, each FPGA has a dedicated auxiliary port that may be used for 100 Gb Ethernet (QSFP ports), and if available (depends on the UBB implementation), it would be used for communication between FPGAs on different UBBs. This communication infrastructure can be exposed to the software layers in a variety of ways, like OpenMP/MPI, etc. Thus, we have software APIs to enable as well as the RTL and IP blocks inside the FPGA to move the bits around. All of these communication options enable a wide variety of EA mappings and system compositions utilizing one or more FPGAs to define a node.

As shown on the right side of Figure 3, each FPGA in the system will use a standard FPGA shell defined in MEEP, the green (dark) perimeter. This FPGA shell is composed of IP blocks common to most FPGA-based systems for communication and access to memory. This includes the FPGA side of the PCIe controller, memory controller and interface for the HBM, and communication blocks and interfaces for nearest-neighbor and remote FPGA communication. One of MEEP's goals is to define a general, high performance interface at the boundary of the FPGA shell and accelerator package that can be reused across many projects.

4.1 Host Interface

Each FPGA is connected to the Host Server through a PCIe interface. This is a general purpose communication infrastructure that allows the host to interact with the FPGA. The initial interactions between the host server and FPGAs will be similar to an accelerator offload model, similar to what is done with GPUs today. In this case, the host interface will be used to move data between the host and the Emulated Accelerators (EA) of FPGAs, move code or acceleration kernels from the host to the EA, and handle the control flow of the application including kernel initiation. Thus, the host interface must handle bulk data transfers as well as register reading and writing or similar mailbox-like functionality in phase 1, which focuses on a single EA implementation. As more functionality moves into the EA, making it self-hosted, more of the application can execute in the EA and the host provides services, like name services, access to storage, resource scheduling, etc. In this scenario, data is resident in the EA and must be shared across the accelerators. As a consequence, the host interface is still essential, but ideally not for bulk memory transfers

4.2 FPGA Shell

The FPGA design is going to be composed of two main logic areas, the shell and, inside of it, the Emulated Accelerator (EA) package. The shell is meant to be a static perimeter architecture which guarantees that the inside accelerator package can be interchangeable for any other package when meeting a defined I/O interface between the shell and the accelerator package, as shown below in Figure 4.

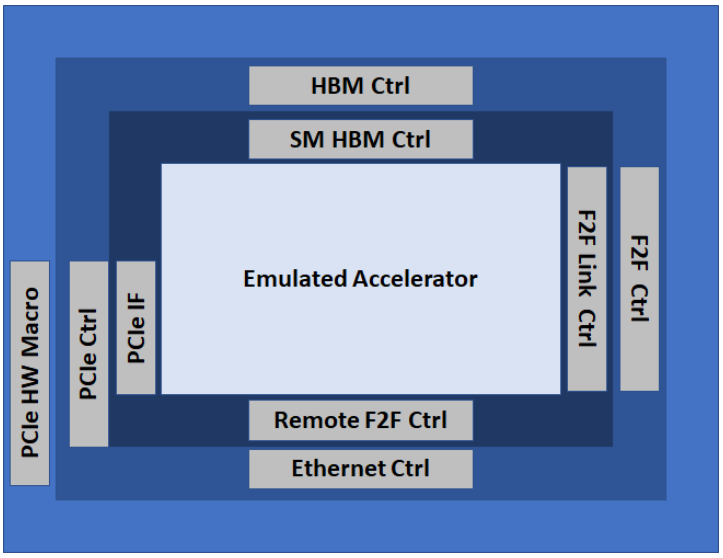

Figure 4. The MEEP FPGA shell (the three blue outer concentric rectangles).

MEEP presents the FPGA shell as three distinct layers. The outermost ring or layer (PCIe HW Macro) corresponds to the host communication interface or PCIe connection. In order for the system to have I/O connectivity at system boot, the PCIe controller (PCIe Ctrl) in the FPGA must be programmed at power on and ready for enumeration by the BIOS. This can be done later, but in order to mimic a real system, the PCIe controller must be operational at system power-on or reboot. The next concentric ring includes FPGA vendor specific IP. This includes the HBM controller (HBM Ctrl), FPGA-to-FPGA links in the UBB using the FPGA SerDes (F2F Ctrl) and Aurora interfaces and

the Ethernet MACs used for 100Gb Ethernet (Ethernet Ctrl and QSFP ports, shown in Figure 5). The innermost concentric ring is the MEEP shell interface to the internal FPGA logic of the emulated accelerator or other FPGA functionality. This ring is MEEP specific RTL that provides a common interface to the remaining FPGA logic. The goal is to keep it as minimal and flexible as possible. This provides the biggest payload or resources for the MEEP emulator. Figure 5, below, translates MEEP shell RTL and related IP components to physical hard macros mapped in the XIlinx VU37P FPGA, highlighted by the yellow boxes.

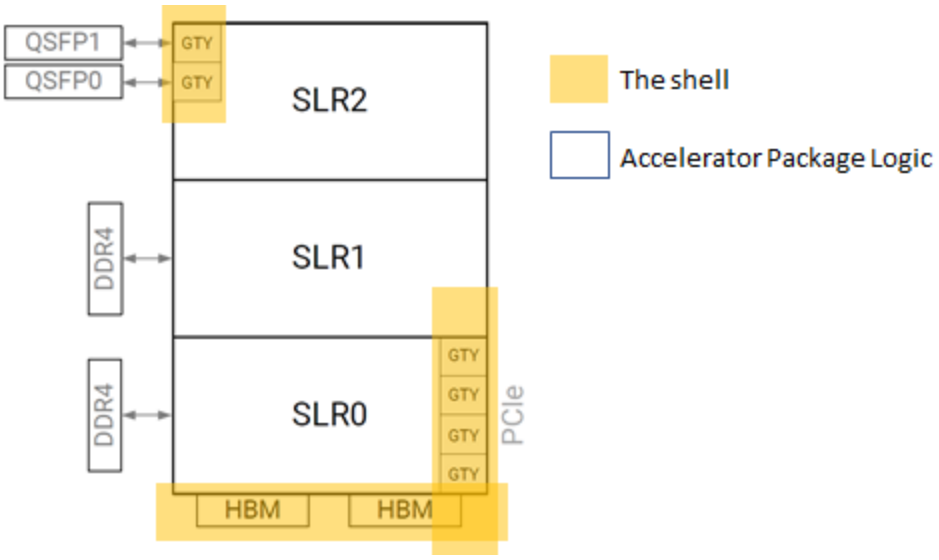

Figure 5. The MEEP Shell mapped to the SLRs in the VU37P FPGA, shown in yellow.

Super Logic Regions (SLRs) are physically independent blocks that can be thought as individual FPGAs which connect each other through extremely efficient connections as they all are built in the same silicon interposer (see Figure 10).

We have the option of implementing the MEEP shell using partial reconfiguration or as a static FPGA configuration. The cost-benefit analysis of partial reconfiguration is left to the design and evaluation of this project, completed later. If we pursue a system with partial reconfiguration, the yellow area will represent the static region of the FPGA, programmed at power on, where glue logic will be added to the Xilinx IPs to define the MEEP shell. The glue logic will be at least the soft IPs to use the interfaces, FIFOs and addressing schemes, and the DMA to create the bridge between PCIe interface and HBM. As shown in Figure 4, PCIe and HBM macros and related hardware are located in SLR0. This leads to a Emulation Platform design which uses SLR0 to implement the main components of the Shell and efficiently communicates with other SLRs to balance how the accelerator logic (as opposed to the shell logic) is spread over the entire device. In addition, as it will be explained later, the top left corner of SLR2 will be reserved to the F2F connectivity through the QSFP.

There are two main phases in the MEEP development: 1) The single FPGA development in phase 1 and 2) the Multi-FPGA development in phase 2. The phase 1 development will use the Xilinx Alveo U280 data center accelerator cards. We have multiple cards to facilitate early FPGA-to-FPGA communication development as well, but the idea is to scale the system beyond what can be done with these cards in a denser form factor. Phase 2 will use the OAM platform to place up to 8 OAM FPGA devices in a UBB chassis. This will be a significant scale to emulate a larger system.

4.2.1 PCIe Interface

A common and well-defined interface establishes communication between the FPGAs and the host server via PCIe. The advantage of PCIe is not only the availability of OS drivers, but also the integrated hard IPs in the FPGA.

Despite the fact that Xilinx supports two different types of PCIe blocks: XDMA and QDMA, MEEP focuses its efforts on the more flexible QDMA implementation. Figure 6 represents how the PCIEe interface would look like from the MEEP architectural perspective. 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 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 provides support for 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.

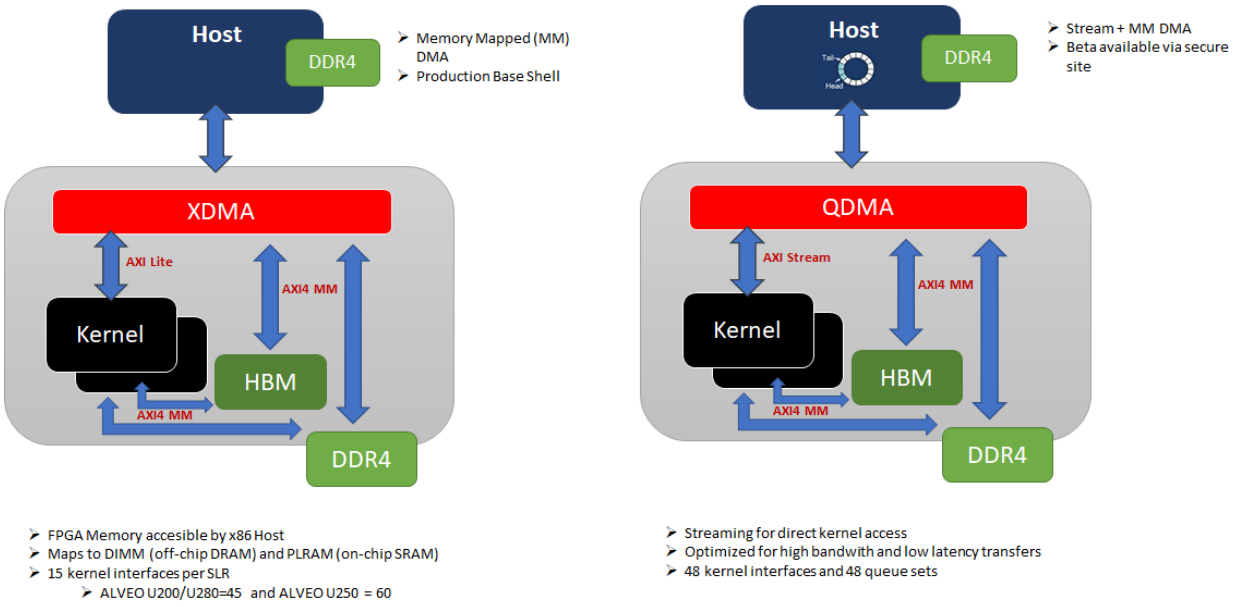

Figure 6. PCIe interface design using XDMA vs QDMA Shells [Source: [9]]

A more detailed comparison between these two blocks is shown in Figure 7. While the XDMA features are shown on the left, the QDMA features shown on the right highlight some important differences between the two blocks. QDMA offers more functionality with slightly higher resource costs. Furthermore, it is interesting to highlight two aspects of QDMA. First, it implements a high performance configurable Scatter Gather DMA for the PCIe integrated block; and second, the IP provides an AXI4-Stream user interface instead of AXI-Lite.

	XDMA Shell	QDMA Shell
Host Interface	PCIe® Gen3x 16	PCIe Gen3x 16
Datapath	512-bit wide AXI Memory Mapped	512-bit wide AXI Streaming and Memory Mapped
DMA Transactions	Memory Mapped transfers into DDR4/PLDAM	1. Direct streaming transfers into Kernel 2. Memory Mapped transfers into DDR4/PLRAM
Kernel Interfaces	U200: Up to 45 AXI interfaces U250: Up to 60 AXI interfaces	Up to 48 AXI interfaces
Maximum Transfer Size	256MB	4MB with 4KB transfer size
DDR4 Channels	U200: Up to 4x DDR4-2400 16GB (64GB max) U250: Up to 4x DDR4-2400 16GB (64GB max)	U200: Up to 3x DDR4-2400 16GB (48GB max) U250: Up to 4x DDR4-2400 16GB (64GB max)
On-Chip Memory (PLRAM)	1 per SLR up to 4MB	1 per SLR up to 4MB
Shell Size	1X	1.25X

Figure 7. XDMA vs QDMA Shells comparison [Source: [9]]

From a general point of view, in MEEP, PCIe has four aspects to work on:

1. Hard macros. Taking advantage of these structures we reduce design efforts and maximize reutilization.
2. QDMA IP configuration. Customize QDMA IP for MEEP. .
3. Xilinx PCIe QDMA Drivers [10]. Leveraging the existing driver and extending the functionality to support MEEP.
4. Xilinx APIs for FPGA - Host communication. This is the API that WP5 will use for I/O between the host and the EA (FPGA).

The first two groups are out of the scope of this document and can be found in various Xilinx provided documentation [12]. The FPGA platform is described in high-level detail in Section 4.4.

4.2.1.1 Xilinx PCIe DMA Drivers: QDMA

The Xilinx PCIe Multi Queue DMA (QDMA) IP provides high-performance direct memory access (DMA) via PCI Express, and it can be implemented in UltraScale+ devices.

In order to establish and control the communication with PCIe peripheral, QDMA requires following driver:

- Linux kernel driver (version release 2019.2) [11]: This one allows the system to recognize and interact with the peripheral.

Both the linux kernel driver and the DPDK driver can be run on a PCI Express root port host PC to interact with the QDMA endpoint IP via PCI Express. For the QDMA driver, the kernel code is GPL (to take advantage of all kernel features), and the user land code is licensed under BSD enabling full flexibility [13]. WP5 determines the overall requirements for the PCIe driver.

On the software side, the QDMA IP is seen as a device node in the /sys filesystem. At this point, several queues can be created and activated using the "dmactl" utility provided by Xilinx. The queues are then ready for DMA operations through a linux device that is created in /dev. For example, if the PCIe device has been assigned to bus 02, the device node "/dev/qdma0200x-MM-y" will represent the queue associated to the Virtual Function "x", and lying in slot y, and the memory

ranges that can be accessed. From the user level, DMA operations are initiated by reading or writing to this node.

The memory regions accessible through the QDMA files in "/dev/" are defined in the Vivado project containing the QDMA IP. The DDR4 and HBM memory of the Alveo U-280 FPGA board in Phase 1 or just HBM memory in the OAM FPGA in Phase 2 and the BlockRAM of the FPGA chip can be made accessible in this way to the user applications.

The QDMA driver (in file qdma_mod.c [6]) allows for its configuration, and retrieval of parameters, connected to the /sys qdma node:

- Maximum number of queues per physical function
- Various internal buffers sizes
- Number of kernel threads to create for queue processing and queue completion handling
- Creating, starting, stopping and deleting queues
- Map and unmap BAR data to user processes address space
- Read and write user registers

This is done using the APIs and structure shown, below, in Figure 8. The QDMA driver provides both user space APIs and the associated kernel space functionality to manipulate the device and present a variety of queues and other capabilities to specialize the device driver and interface to the FPGA, presenting a similar API that would exist for the EA, making the FPGA transparent to the SW and application developer.

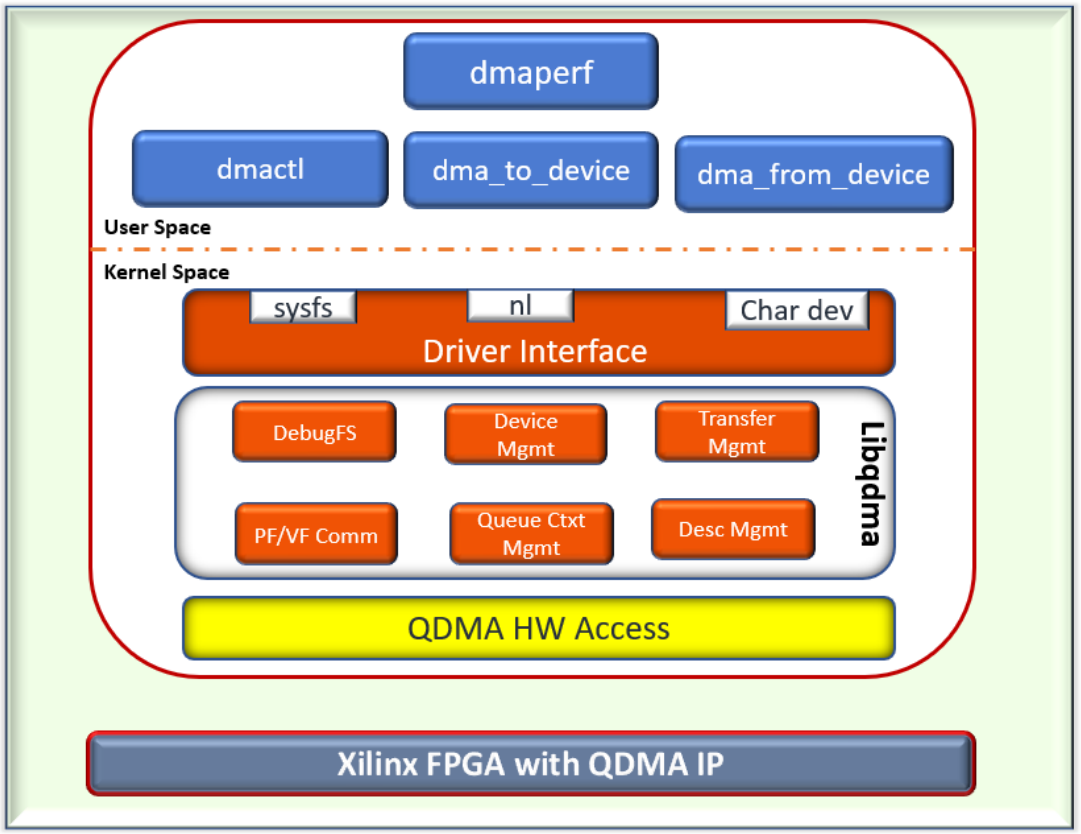

Figure 8. Xilinx PCIe QDMA Driver

User applications open the queue devices (e.g., /dev/qdma02000-MM-1), and use the read/write and lseek system calls to program the DMAs in order to transfer data from the PCIe address space (to/from the host side), to the card memories, being those the DDR, HBM, or FPGA BRAMs.

In summary, Xilinx's DMA module, once inserted into the kernel of the host server, establishes the software interface between the host server and the FPGAs by creating new device nodes. Any data sent or received by those nodes is automatically transmitted across to or from the respective memories in the FPGA.

4.2.1.2 Xilinx APIs for FPGA-Host communication

On top of the driver functionality, we will leverage the xtasks interface that we currently use for OmpSs@FPGA support. The following table (Figure 9) lists the functions, and briefly describes them:

Function	Description
xtasksInit()	Initializes xtasks internal structures based on the FPGA configuration
xtasksFini()	Finalizes xtasks
xtasks_stat xtasksGetNumAccs(size_t *count)	Get the number of available accelerators in the system
xtasksGetAccs(size_t const maxCount, xtasks_acc_handle *array, size_t *count)	Retrieves the accelerators available in the FPGA, cores, vector units...
xtasks_stat xtasksGetAccInfo(xtasks_acc_handle const handle, xtasks_acc_info *info)	Gets the properties of a specific accelerator
xtasks_stat xtasksConfigureAcc(xtasks_acc_handle const handle, xtasks_acc_config * config)	Writes new configuration parameters to the specific accelerator. It can be used to implement the initial reset
xtasks_stat xtasksMalloc(size_t len, xtasks_mem_handle *handle)	Allocates memory in the FPGA address space
xtasks_stat xtasksFree(xtasks_mem_handle handle)	Releases memory in the FPGA address space
xtasks_stat xtasksMemcpy(xtasks_mem_handle const handle, size_t offset, size_t len, void *usr, xtasks_memcpy_kind const kind)	Synchronously copy data to/from an allocation. The kind parameter indicates Host-to-Acc or Acc-to-Host.
xtasks_stat xtasksMemcpyAsync(xtasks_mem_handle const handle, size_t offset, size_t len, void *usr, xtasks_memcpy_kind const kind, xtasks_memcpy_handle *cpyHandle)	Asynchronously copy data to/from an allocation

xtasks_stat xtasksTestCopy(xtasks_memcpy_handle *handle)	Test the status of a copy operation
xtasks_stat xtasksSyncCopy(xtasks_memcpy_handle *handle)	Synchronously wait for a copy operation

Figure 9. Xtasks interface for OmpSs

4.2.2 HBM Interface

High Bandwidth Memory (HBM) is the high performance DRAM interface of the near future that massively increases system memory bandwidth [2] using system in package technology (SiP), as Figure 10 shows. This high-performance RAM interface also changes the form factor by using a 3D-stacked SDRAM and is produced by Samsung, AMD and SK Hynix. This interface presents two interesting options to explore with the RTL: 1) Multiple (8 per stack) independent memory controllers or 2) One large monolithic memory controller per stack.

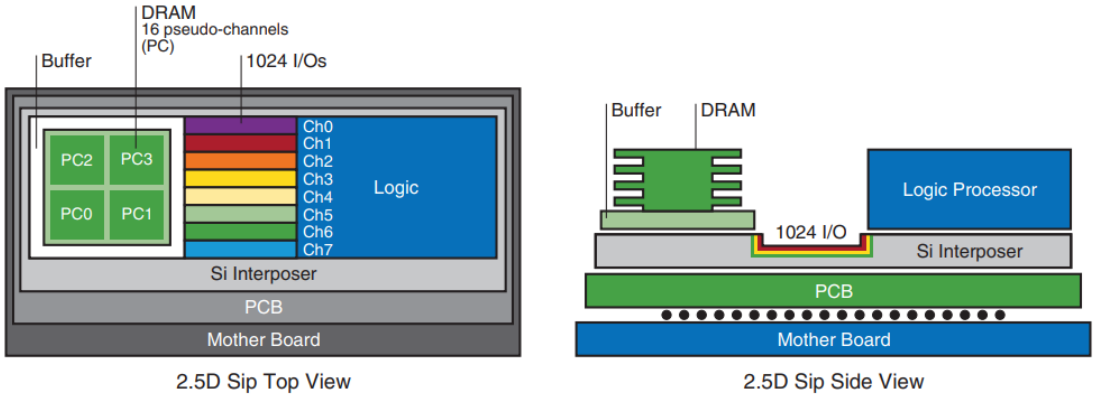

Figure 10 HBM Structure [2].

The MEEP FPGA shell provides a standard interface to HBM memory. An HBM stack of four DRAM dies (4-High) has two 128-bit channels per die for a total of 8 channels and a width of 1024 bits in total [2]. In comparison, the bus width of GDDR memories is 32 bits, with 16 channels for a graphics card with a 512-bit memory interface. Memory bandwidth for Xilinx Virtex Ultrascale+ FPGAs with HBM is up to 460GB/s delivered by two stacks of Samsung HBM2 memory. There is the flexibility to access pairs of pseudo channels or aggregate all the pseudo channels across to stacks of HBM together as a large memory access engine with much larger memory bandwidth and larger granularity. In the context of the EAs, the actual operating frequency of the EA (100 - 200 MHz) is much slower than traditional accelerators implemented in an ASIC (1-2 GHz) meaning the effective bandwidth of the HBM as seen from the EA is much higher compared to the effective bandwidth of the HBM to an ASIC. Thus, MEEP can use time dilation to adjust the bandwidth and latency to match the emulated platform.

The FPGA hardware enables the MEEP platform to embed additional resources around the memory controllers. We present the first emulated accelerator architecture in Section 4.3 (Figure 13). This is a disaggregated architecture that allows us to embed more intelligence near the memory controller and in doing so, select how to use the memory interface, either as a large monolithic interface or many small interfaces, depending on application needs and characteristics, a nice software-hardware co-design opportunity. The key challenge is to combine the knowledge of the larger data

structures (vectors), with the application program flow to optimize the use of the on-chip memory hierarchy.

4.2.3 FPGA-to-FPGA Interface

Modern CPU and GPU architectures are composed of multiple die that have some type of interconnect. The MEEP platform also has interconnect between the FPGAs for similar purposes. In both phases of the project, we will develop the FPGA-to-FPGA (F2F) communication infrastructure to provide this capability. In all cases, we must provide the RTL to address local and remote FPGAs, send and receive data, provide control flow, and define minimum and maximum packet or data payload length. The local FPGA communication is defined by direct point-to-point links between FPGAs. In Phase 1, we will construct a 1D ring using the two QSFP links, as shown in Figure 11. In order to test the QSFP links and PCIe peer-to-peer communication, a three server configuration will be enabled in Phase 1. Servers will be configured as follows: 2 servers containing 1 Alveo U-280 each, and 1 server containing 2 Alveo U-280 cards (referred to as X1 and X2 in Figure 11, respectively).

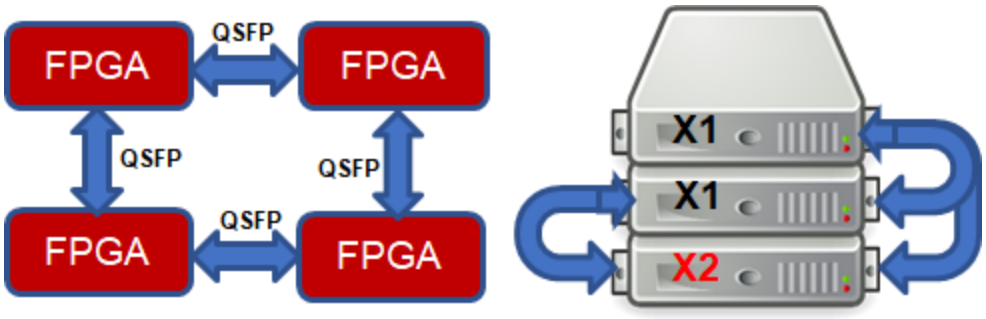

Figure 11. Phase 1: FPGA-to-FPGA interface using a 1D ring using QSFP links.

This configuration also enables investigation of peer-to-peer PCIe communication between the two Alveo U-280 cards in the same server. FPGAs that are not adjacent, diagonals in Figure 11, are remote FPGAs in terms of the addressing scheme. In Phase 1, intermediate FPGAs must act as forwarding agents for these data packets. In Phase 2, the local FPGAs are those that are connected on the UBB with direct peer-to-peer SerDes links. If the UBBs provide auxiliary QSFP links for each OAM, we can use those to address remote FPGAs via a 100 Gb Ethernet switch. For the local F2F communication in Phase 2, we will use the High-speed SerDes (GTY transceivers) and the Xilinx Aurora IP to facilitate local F2F communication. This physical connectivity is shown in Figure 12 and provided by the UBB. Note, that the UBB implementation determines the physical connectivity between the OAM (FPGA) modules.

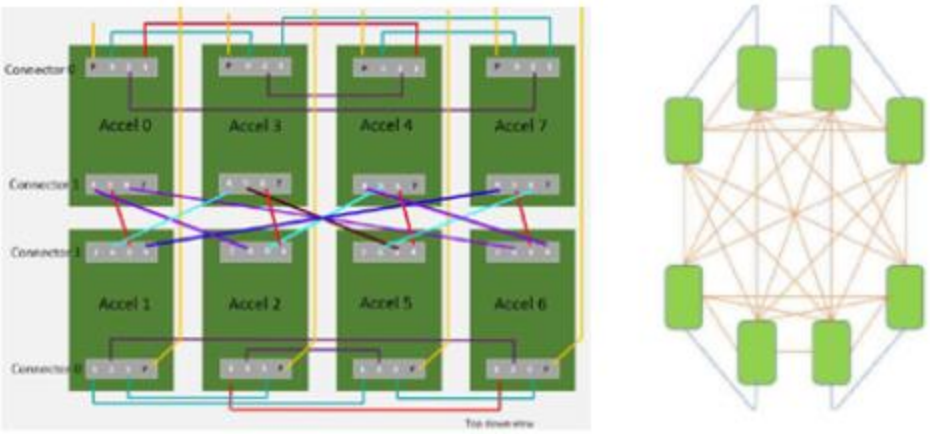

Figure 12. OAM FPGA fully connected topology in the UBB. On the right, links between FPGAs are in yellow and links between the FPGA and the host are shown in blue.

Xilinx provides the RTL Aurora module to enable the interface between the FPGAs. Aurora infers inner Xilinx Gigabit transceivers which are the physical layer to this high-speed communication. On top of that, Aurora encodes the data in a 64b/66b protocol to provide enough state changes to allow reasonable clock recovery and alignment of the data stream at the receiver [1].

Raw data communication can be implemented using glue logic to directly write into the Aurora module using the AXI Stream Protocol. On the other end, the FPGA receives the data and processes it, then sends it to the memory if required. Note, the QSFP connectors are not supported in the Vitis version of the deployment shell and may require a different implementation for the same functionality. For the MEEP platform, all implementation will focus on the (re-)use case for the Phase 2 deployment. We may just focus on the remote communication for the FPGAs with the QSFP interface.

Finally, we can map these physical F2F links to higher level software communication primitives like MPI_send and MPI_receive, OpenMP, or other framework communication APIs. This provides accelerated communication primitives and co-design acceleration opportunities.

4.2.4 Optional Network Interface

The Xilinx VU37P FPGA has two built-in 100 Gb Ethernet capable MACs. If one or more 100 Gb Ethernet MACs are available in the UBB, we can connect each FPGA in the OAM package (8 FPGAs in OAM modules per UBB) to an Ethernet switch. Thus, in Phase 2, there would be local F2F connections on the UBB and the capability to directly connect via Ethernet to other FPGAs connected to the Ethernet switch. This remote FPGA communication on the EA dedicated network could be mapped to communication primitives and bypass the host CPU complex. The MEEP platform would extend the MEEP shell to include this remote communication capability. This will be determined by the UBB implementation. If available, the Phase 1 platform will prototype this capability. Overall, the architecture and design of the F2F communication will determine the capabilities at the shell. The addressing and routing of the F2F messages, either local or route, can be unified into a single interface or explicitly separated and handled by the EA system.

4.3 FPGA Accelerator Emulation Target

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.

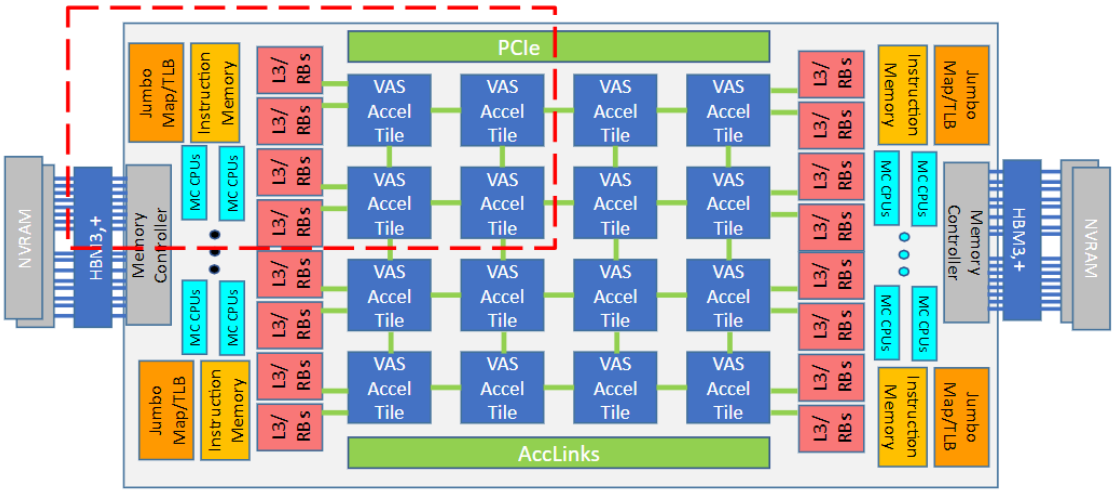

Figure 13. Target self-hosted accelerator chiplet architecture.

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 13, above, shows the high-level block diagram of the self-hosted accelerator.

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, giving a 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 the 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, as part of the VAS Accelerator Tile, 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 to have enough computing capabilities left 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 leveraging Partitioned Global Address Space (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 in the host. As a consequence, the host might be seen as a service provider.

It is clear that all of the envisioned chiplet cannot be mapped into a single FPGA. As shown above in Figure 13. 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 13 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 the Work Package 4 Architecture deliverables.

4.4 FPGA Emulator Hardware

Over the course of the project, MEEP will have two different hardware platforms that align with the two different phases of the project: Phase 1, single FPGA implementation and Phase 2, multi-FPGA implementation. In both phases, we will be using the same FPGA, the Ultrascale+, VU37P. There is overlap between the hardware in the two phases so we can also align with the availability of the

hardware. Phase 1 uses FPGA hardware that is available today and Phase 2 will target high-density FPGA modules. The VU37P has the following features, shown in Figure 14:

	Device Name	VU37P
Logic	System Logic Cells (K)	2,860
	CLB Flip-Flops (K)	2,615
	CLB LUTs (K)	1,308
Memory	Max. Distributed RAM (Mb)	36.7
	Total Block RAM (Mb)	70.9
	UltraRAM (Mb)	270
	HBM DRAM (Gb)	64
	HBM AXI Ports	32
Clocking	Clock Management Tiles (CMTs)	12
Integrated IP	DSP Slices	9,024
	PCIe® Gen3 x16 / Gen4 x8	6
	CCIX Ports(2)	4
	150G Interlaken	4
	100G Ethernet w/ RS-FEC	8
I/O	Max. Single-Ended HP I/Os	624
	GTY 32.75Gb/s Transceivers	96
Speed Grades	Extended(1)	-1, -2L, -3

Figure 14. VU37P device features [8].

MEEP will mate the Xilinx FPGAs with AMD-based systems.

4.4.1 Phase 1 FPGA Platform

MEEP follows the currently available hardware to use the latest technology for the FPGAs. The current hardware platform uses the Alveo U280 actively cooled FPGA card, as shown in Figure 15, below [8]. This uses the VU37P FPGA. This FPGA has various hard IP blocks that can be used in the system, including: HBM and related memory controller, PCIe controller and drivers, QSFP and ethernet functionality. These hard macros align well with some of the base functionality of the EA. MEEP will start out with 4 cards as shown in Figure 11, above. This enables preliminary work on the multi-FPGA platform. These cards also have DDR memory, which will not be relevant for this project, but could be useful in other future projects. The main difference between the FPGAs in this platform and the Phase 2 platform is the lack of SerDes connections between FPGAs. The all-to-all communication provides multiple mapping opportunities for the EA onto the FPGAs.

Below, in Figure 15, are the general characteristics of the VU37P FPGA in the Alveo U280:

Card Specifications	U280 Production
Thermal Cooling	Active
Compute	
INT8 TOPs (peak)	24.5
Dimensions	
Width	Dual Slot
Form Factor	Full Height, Full Length
DRAM Memory	
HBM2 Total Capacity	8GB
HBM2 Total Bandwidth	460GB/s
DDR Format	2x 16GB 72b DIMM DDR4
DDR Memory Capacity	32GB
DDR Total Bandwidth	38GB/s
SRAM Memory	
Internal SRAM Capacity	41MB
Internal SRAM Total Bandwidth	30TB/s
Interfaces	
PCI Express	Gen4x8 with CCIX
Network Interfaces	2x QSFP28 (100GbE)
Logic Resources	
Look-up Tables (LUTs)	1,079,000
Power	
Maximum Total Power	225W

Figure 15. Xilinx Alveo U280 Data Center

Figure 16. Xilinx Alveo U280 Data Center Accelerator Card for Phase 1.

4.4.2 Phase 2 FPGA Platform

The Phase 2 platform uses a different package, the OAM [4], and a chassis that provides up to 8 FPGAs connected together on a PCB. The Open Compute Project [4] describes a standard accelerator platform that interconnects host CPUs to accelerators as well as inter-OAM connectivity as well. The OAMs are connected on a PCB called the Universal Base Board (UBB). There are multiple providers of the UBB with different connectivity [7]. The preferred UBB connectivity for Phase 2 is all-to-all communication between the OAM modules as well as ethernet connectivity for each OAM. As shown in Figure 12, the OAM modules are also connected to the host servers via PCIe. While the base FPGA is the same, the system connectivity allows for FPGA-to-FPGA communication and the possibility for network-attached FPGAs as well. Using the UBB platform, Phase 2 enables the large scale deployment of the EA as well as a substantial software development platform. There are at least three vendors providing the UBB platform: Inspur, Hyve and ZT Systems. As the UBB platforms become available, we will select the best system for MEEP. The UBB will be connected to AMD EPYC server CPUs. This CPU platform provides many PCIe lanes. As a fallback, if the Phase 2 OAM modules are not available, we can use the plentiful PCIe lanes in these servers connected to up to 8 Alveo U280 FPGA cards. This sacrifices the inter-FPGA connectivity, but still provides the scale and scope of the MEEP platform.

5. Common Emulator & Accelerator

The MEEP system is very flexible and can be used to emulate other accelerators and CPUs. As an emulator platform, the hardware macros (PCIe, HBM, DDR, and Ethernet) reduce the effort to emulate these new architectures for performance analysis and software development. There is also the option to build architectures and systems for FPGAs, instead of emulating a system. This provides capabilities to map RTL or HLS kernels and reuse the shell.

We have identified several future projects that can leverage the platform. These include out of order RISC-V processor designs like eProcessor to enable hardware module support and software

development support. MEEP can also support extensions to the EPAC accelerator to support other application domains. MEEP can integrate and investigate the evolution of technologies like HBM and the related memory hierarchy. Likewise, MEEP can be a new target for OmpSs on FPGAs. Finally, evolving the MEEP platform with new FPGAs will extend the platform's capabilities, both in terms of new technology integration like non-volatile memory and higher logic capacities and internal hardware macros. This will extend the later version of MEEP for new projects and larger systems.

6. Timeline and Deliverables

In a nutshell, WP6 aims to join together the efforts of WP4 (HW) and WP5 (SW) on top of a target FPGA accelerator emulation platform. The deliverables are well-aligned with the proposed project reviews (at month 18 and month 36). This document serves as the first deliverable describing the emulator software components. Although the FPGA platform is substantial, the first milestone of development will focus on the bringup of a single FPGA-based accelerator on a single node at M18. The next milestone will be to develop the mechanisms to enable communication between neighboring accelerators on a single PCB, provided in the milestone deliverables by M30. Finally, full system (multi-node) applications can be demonstrated on this platform between M30 and M36, as shown below in Figure 17. Please note that the combination of FPGA shell plus the supporting communication and software and tools ecosystem will be designed to support many other environments and projects, as described in Section 5.

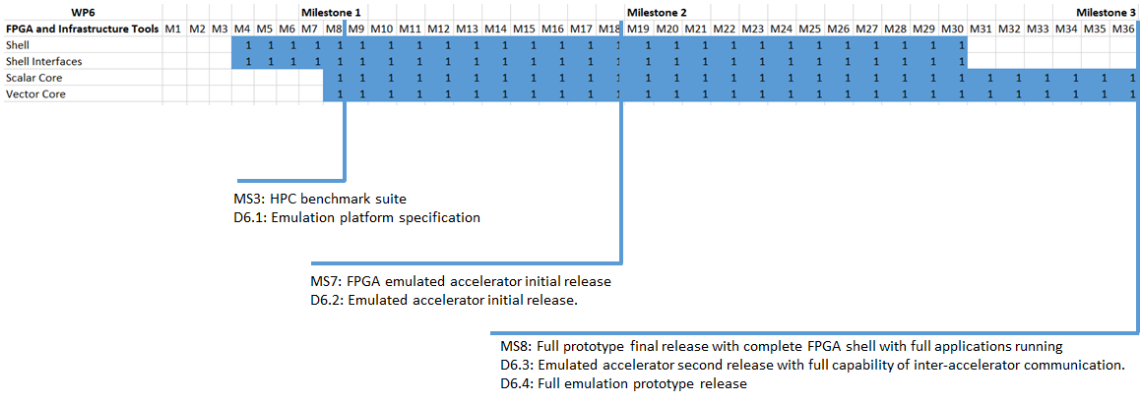

Figure 17. WP6 timeline with milestones and related deliverables.

7. References

[1] https://en.wikipedia.org/wiki/64b/66b_encoding

[2] https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf

[3]https://www.xilinx.com/support/documentation/white_papers/wp508-hbm2.pdf

[4] https://www.opencompute.org/wiki/Server/OAI

[5] https://www.opencompute.org/

[6] https://github.com/Xilinx/dma_ip_drivers/blob/master/QDMA/linux-kernel/driver/src/qdma_mod.c

[7] http://files.opencompute.org/oc/public.php?service=files&t=f924e6fefb20c9be651c61c0f4b1a5cc

[8] https://www.xilinx.com/support/documentation/data_sheets/ds963-u280.pdf

[9] https://forums.xilinx.com/t5/Adaptable-Advantage-Blog/Improve-Your-Data-Center-Performance-With-The-New-QDMA-Shell/ba-p/990371

[10] https://github.com/Xilinx/dma_ip_drivers

[11] https://xilinx.github.io/dma_ip_drivers/2019.2/linux-kernel/html/index.html

[12] https://www.xilinx.com/support/documentation/ip_documentation/qdma/v4_0/pg302-qdma.pdf

[13] https://forums.xilinx.com/t5/PCIe-and-CPM/PCIe-Drivers-from-Xilinx-vs-3rd-party/td-p/973816

8. List of acronyms

AI Artificial Intelligence

API Application Programming Interface

BSD Berkeley Software Distribution (Licence)

CGMT Coarse-Grain Multi-Threading

CoE Center of Excellence

CPU Computation Processing Unit

DL Deep Learning

EA Emulated Accelerator

EPI European Processor Initiative

FGMT Fine-Grain Multi-Threading

FIFO First In First Out

FPGA Field Programmable Gate Array

GPL General Public Licence

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

IP Intellectual Property

ISA Instruction Set Architecture

MEEP MareNostrum Experimental Exascale Platform

MPI Message Passing Interface

ML Machine Learning

NoC Network on Chip

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 Projects

OmpSs OpenMP and Stars combined programming model

OOO Out of Order (CPU)

OS Operating System

PGAS Partitioned Global Address Space

POP Performance Optimisation and Productivity

RISC Reduced Instruction Set Computer

RTL Register Transfer Level (Hardware Description Language)

SDV Software Development Vehicle

SLR Super Logic Region

TPU Tensor Processing Unit

UBB Universal Base Board