D6.5 -First AIT release

Version 1.1

Document Information

Contract Number	946002
Project Website	https://meep-project.eu
Contractual Deadline	30/06/2023
Dissemination Level	Public (PU)
Nature	Others
Author	Elias Perdomo (BSC), Miquel Vidal (BSC)
Contributors	José Oliver (BSC), Teresa Cervero (BSC)
Reviewers	Carlos Álvarez (BSC), Behzad Salami (BSC), John Davis (BSC)

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.

Change Log

Version	Description of Change
v0.1	Initial structure
v0.2	Complete version for internal review
v0.3	Adding Appendix I, II and III
v0.4	Internal Review
v1.0	Final updates
v1.1	Second internal 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 (TÜBITAK BILGEM).

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

Contents

Executive Summary

This document is part of a collection that describes the MareNostrum Experimental Exascale Platform (MEEP) Project. This project is organized into six work packages (WP1 to WP6). Of these, WP4, 5, and 6 cover technical content. Figure 1 shows the relationship between these technical WPs.

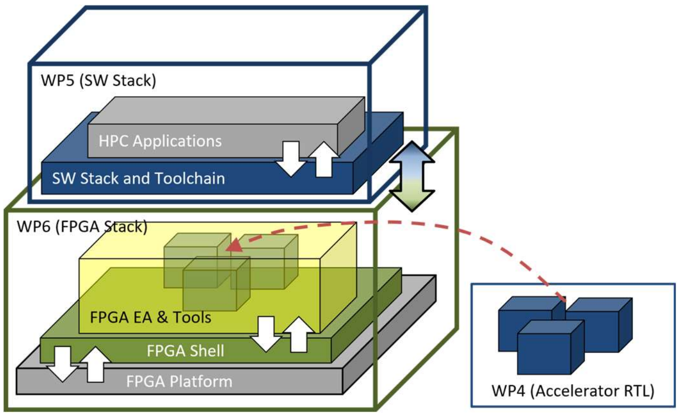

Figure 1. Layered structure of the MEEP project based on its work packages (WP4, WP5 and WP6)

- WP4 describes the Accelerated Compute and Memory Engine (ACME) accelerator architecture and its related RTL, verification, and performance modeling simulations.
- WP5 covers software, affecting all the layers of the software stack.
- WP6 is where the other WPs come together, by running SW applications in a miniaturized version of the ACME architecture on the emulation platform.

All these WPs together are focused on the development of a set of tools that enable the exploitation of the MEEP FPGA-based emulation platform as a hardware/software co-design laboratory.

This *D6.5 First AIT release (M42)* complements the previous and current deliverables:

- *D6.1 Emulation Platform specification (M6).*
- *D6.2 Emulated accelerator initial release (M18).*
- *D6.3 Emulated accelerator second release with full capacity of inter-accelerator communication (M36).*
- *D6.4 Full emulation prototype release (M42).*

D6.5 First AIT release (M42) presents the first release of the Accelerator Integration Tool (AIT), which provides a basic framework for designing and generating a self-contained design, fully compatible with the MEEP FPGA Shell. The scope of this deliverable is constrained to the WP6 Task 6.4 *Automation Integration Tool* (M18-M42) results. The purpose of this task is to allow the automatic generation of bitstreams for any accelerator design in an easy-to-use way, without requiring deep knowledge on the specific hardware from a developer. AIT is a tool used with the purpose of executing a full application but delegating the acceleration of some parts of the code on hardware accelerators. The generation of these accelerators takes advantage of Xilinx tools in the background. The contribution of MEEP to this tool is the integration of it with the MEEP FPGA Shell, which guarantees seamless communication between the accelerator and the host and opens the possibility of exploiting all the MEEP FPGA Shell features.

As part of the FPGA-oriented development flow of AIT, the tool provides a configurable wrapper around the functional design for adding the capability to the whole design of being self-hosted and communicating with the rest of the system (i.e., the host and other FPGAs). As shown in Figure 2, one of the main advantages of the MEEP FPGA Shell is the ability to provide a flexible, scalable, and customizable wrapper for various kinds of accelerators targeting the FPGAs. The main elements of the FPGA Shell, mainly HBM memory and communication mechanisms, are described in deliverables D6.2, D6.3 and D6.4.

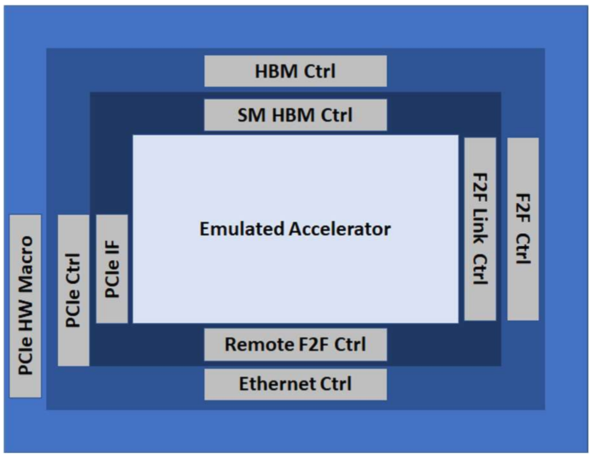

Figure 2. Layered structure of the MEEP FPGA Shell used as a communication wrapper for any targeted emulated accelerator

This D6.5 document describes the feature of synthesizing OmpSs@FPGA[1]-based accelerated kernels by using AIT. This produces an Emulated Accelerator component with the novelty of introducing the MEEP FPGA Shell as a communication wrapper for the generated design. To explain all the modifications applied on the original AIT, an *NBody* application has been used as a proof of concept (PoC).

[1] OmpSs@FPGA: https://github.com/bsc-pm-ompss-at-fpga

Introduction

In this D6.5 we present how we are expanding the MEEP FPGA Shell support not only to HW-oriented emulate accelerators but also for easily offloading SW application tasks to FPGA devices (Figure 3).

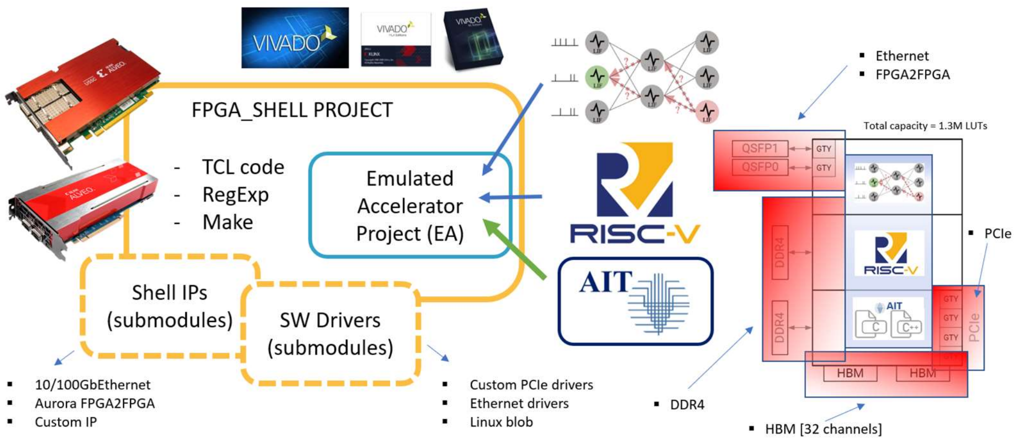

Figure 3. The MEEP-FPGA Shell project including the AIT feature

This document presents an overview of the MEEP FPGA Shell, and the modifications developed on it to be integrated with AIT. In the end, the AIT flow is focused on project generation and its implementation in a reliable, reproducible, extensible, and automated way targeting different FPGA Alveo Boards. More details about the MEEP FPGA Shell are provided in D6.4.

What makes the MEEP FPGA Shell different from other projects such as Amazon F1 and the Xilinx Vitis Platform is the holistic HW/SW support provided for RISC-V-based emulated accelerators. Nevertheless, any IP, for example, Neural Network emulated accelerators that comply with the innermost layer constraints can be used effortlessly.

The rest of this document is structured in the following sections:

- Section 1: Accelerator Integration Tool. This section is focused on providing insights about the tool, including its relationship with the OmpSs@FPGA version of the OmpSs[2] programming model. This section presents why OmpSs@FPGA, what for and its current status.

  In addition, this section presents the new AIT extensions, implemented within the scope of the MEEP project, to its integration flow to make it compatible with the MEEP FPGA Shell flow.

- Section 2: AIT + MEEP FPGA Shell. This section points out the D6.4 *Full emulation prototype release* deliverable for the MEEP FPGA Shell details. Nevertheless, the main scope is to present the modifications to the MEEP FPGA Shell flow that were implemented to make it compatible with the AIT integration flow.

- Section 3: AIT at MEEP release. This section describes the status of the resultant integration process between the AIT integration flow and the MEEP FPGA Shell. Results based on

[2] OmpSs: https://pm.bsc.es/ompss

different benchmarks available are also described and we performed a summary of the next steps.

- Section 4: AIT at MEEP contributions to other projects from the FPGA perspective. Finally, this section describes how the results of the FPGA efforts, in the context of the AIT at MEEP project, have been shared with other projects. With this, we demonstrate the long-term vision of the MEEP infrastructure, in terms of FPGA-based emulation platform, and its associated tool kit.
- Section 5: This section summarizes the main outcomes of this first AIT releasee.

1. Accelerator Integration Tool (AIT)

The Accelerator Integration Tool (AIT) is a fundamental part of the OmpSs@FPGA toolchain, acting as a bridge between Mercurium[3] compiler and the FPGA vendor tools to generate OmpSs@FPGA compatible bitstreams.

1.1 OmpSs@FPGA

OmpSs is a task-based programming model developed at BSC, which aims to minimize code modifications when the original code of an application is ported and parallelized. Particularly, annotations of pragma directives are used with the purpose of getting productive parallel programming, even for heterogeneous architectures, such as GPUs and FPGA.

Programming model directives allow specifying the target device of any given task as well as the input data that it requires and the output data it produces. This input and output data specification allows the runtime to detect dependencies among tasks to ensure a correct task execution scheduling.

Figure 4 shows the compilation flow of the OmpSs@FPGA extension [OMPSS-FPGA]. Input source code is fed to the Mercurium compiler which splits it into host and FPGA code. Host code is then compiled by the backend compiler (i.e., gcc, icc), while FPGA code is transformed and encased in a High-Level Synthesis (HLS) wrapper.

This HLS code, along with a *json* file describing the transformed accelerators, is sent to the Accelerator Integration Tool (AIT) which, using the specific vendor tools, converts HLS accelerators to IP cores. Then these IP cores are coupled with a hardware runtime and additional glue logic and finally integrated into a pre-assembled board-specific FPGA design template containing FPGA-Host interconnection (e.g., PCIe) and memory interfaces.

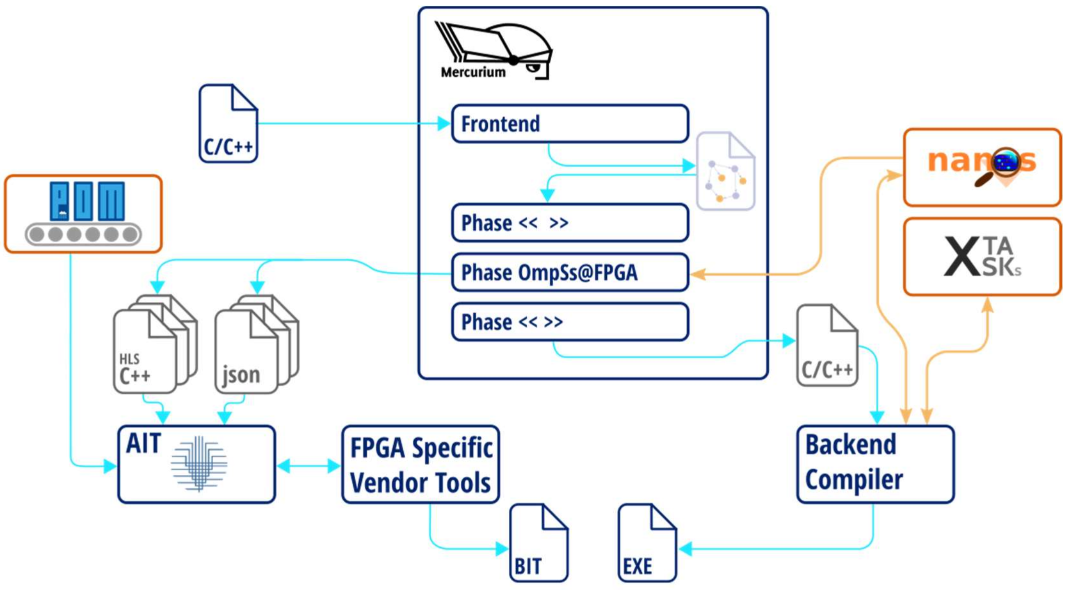

Figure 4. OmpSs@FPGA compilation flow

[3] Mercurium: https://www.bsc.es/research-and-development/software-and-apps/software-list/mercurium-ccfortran-source-source-compiler

The AIT integration flow is comprised of several steps, each of which is responsible for a specific task:

- **HLS**: accelerated kernels HLS code files are converted to IP cores using the FPGA vendor HLS tool.
- **design**: accelerator IP cores are instantiated inside a pre-assembled board template design and interconnected with both the OmpSs@FPGA hardware runtime and the host.
- **synthesis**: design generated in the previous step is synthesized using the FPGA vendor tools.
- **implementation**: design synthesized in the previous step is placed and routed for the targeted FPGA using the vendor tools.
- **bitstream**: placed-and-routed design is written into a bitstream for the target FPGA.

A known limitation of the AIT tool is that the board-specific design templates are pre-assembled and cannot be modified or configured at compile time, meaning that some board components might be present in the final bitstream even though they are not used; and, analogously, the programmer may require some component not existing in the pre-assembled template. It is in this case where we can take advantage of the flexibility of the MEEP FPGA Shell to generate fully customizable board shells to integrate OmpSs@FPGA designs into (Figure 5).

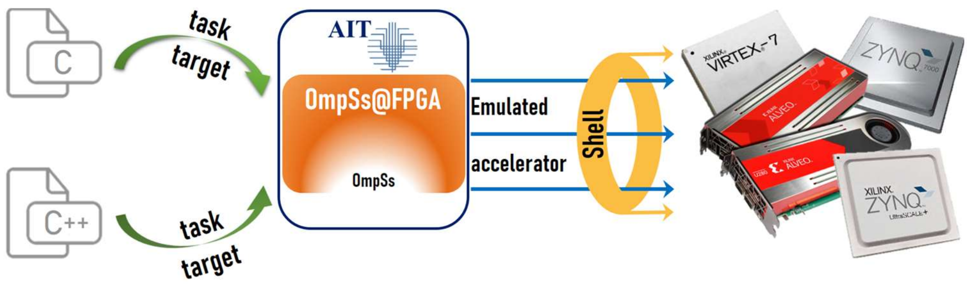

Figure 5. AIT + FPGA Shell features

1.2 Extensions to AIT

Following the path of previous European projects such as EuroEXA [EUROEXA], where support for Alveo U200 was added, and Textarossa [TEXTAROSSA], where placement and memory interconnection improvements were developed, in this project continuous development has been done over AIT to further support new boards and features.

1.2.1 Support for new boards

Two new boards have been added to the list of AIT supported devices: the Alveo U280, both with DDR or HBM memory; and the Alveo U55C.

Alveo U280

As described in deliverable D6.1, Phase 1 of the MEEP development process implemented a small cluster of 4 workstations equipped with a host PC and one Alveo U280, and 2 more with an Alveo U55C each, to demonstrate the initial functionality and some key features.

Support for the Alveo U280 was developed in order to generate OmpSs@FPGA bitstreams for such cluster.

Two versions of this board were implemented: one using DDR4 memory and another with HBM, giving the programmer the ability to choose which type of memory to use at compile time.

Alveo U55C

During Phase 2 of the development process described in deliverable D6.1 the MEEP FPGA cluster, a large-scale cluster with multiple nodes and multiple FPGAs per node, featuring FPGA-to-FPGA and network-based communication, was put into place.

The MEEP FPGA cluster consists of 2 racks, each containing 4 admin nodes, 12 compute nodes and 2 100GbE switches. Each of the compute nodes contains 2 server-grade Intel Xeon processors and 8 Xilinx Alveo U55C FPGAs, for a total of 96 interconnected FPGAs. In comparison with the Alveo U280, the Alveo U55C board doubles per-FPGA HBM capacity from 8 GB to 16 GB but does not feature DDR4 memory.

Consequently, additional changes regarding board memory type were required to include support for the Alveo U55C in AIT.

1.2.2 Support for HBM memories

One of the main goals of AIT and OmpSs@FPGA is to abstract all FPGA-related complexity and heterogeneity by providing a set of high-level tools allowing programmers to guide the implementation process. This includes providing an efficient exploitation of available memory resources, without requiring an in-depth knowledge of how it is arranged, or which type of technology is used.

Prior to the MEEP project, no HBM-based boards were supported by AIT. All available boards were either DDR-based discrete boards or SoC systems with the memory subsystem shared directly with the CPU.

On DDR-based systems, each memory module is accessed through a single slave AXI interface. In order to provide access to the whole capacity of the memory, a simple solution is to aggregate all banks into a single interface and present the programmer a single contiguous address space combining the capacity of all memory modules available in the system. This can be achieved with an AXI Interconnect IP which internally contains a crossbar.

This solution is shown in Figure 6. A limitation of this implementation is that it only allows a single access to each memory module at any given time. Therefore, given that large contiguous chunks of memory space are assigned to each module, large data structures (such as matrices) are usually allocated to a single memory module, preventing accelerators from accessing data structures in parallel.

To overcome this limitation AIT offers several memory optimization features, such as access interleaving and priorities, in the style of compiler flags.

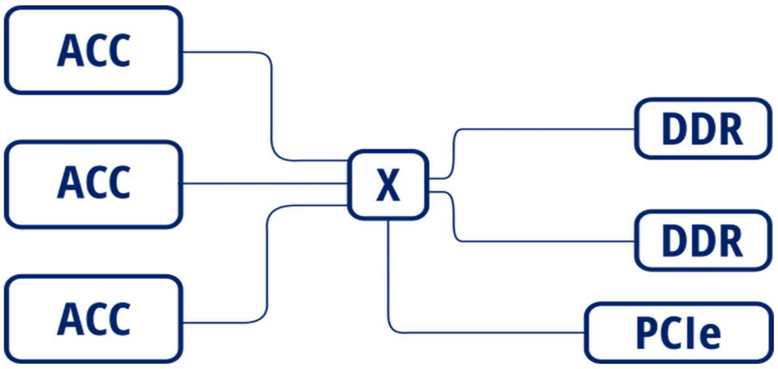

Figure 6. Memory interconnection diagram of a DDR-based discrete board

On HBM-based FPGAs, in contrast, there is a single memory module providing access to the entire HBM memory through a series of channels acting like standard DDR interfaces. Each channel is serviced by a single memory controller and can be accessed through two different AXI interfaces. The HBM memory module consists of two stacks of 8 memory channels for a total of 32 pseudo-independent AXI interfaces.

In this case, access to the full address space is enabled through the Global Addressing configuration that enables HBM internal micro-switches, meaning that no external crossbar is needed. Nevertheless, a simple 1-to-1 AXI Interconnect IP is used on each AXI interface for data width and protocol conversion, as can be seen in Figure 7.

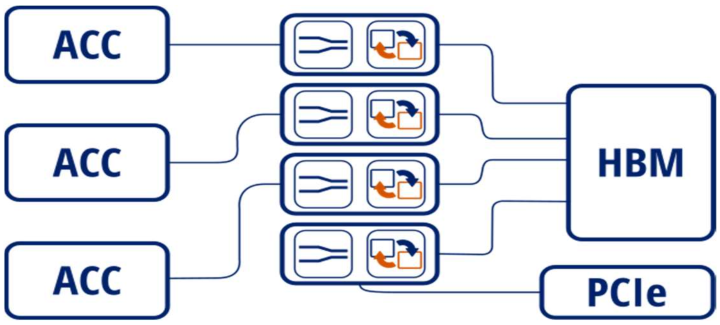

Figure 7. Memory interconnection diagram of an HBM-based discrete board

All previous datapath optimizations applicable to DDR memories such as access interleaving and priorities, as well as others regarding placement, can be also used on HBM memories out of the box.

1.2.3 Memory AXI bonding

Thanks to AIT's abstraction of the underlying FPGA, porting an application from one board to another can be achieved without any code modification, neither in the algorithm nor in the OmpSs@FPGA code. Sometimes there are, however, non-negligible differences in hardware architectures between boards that may require some attention from a platform perspective.

Porting from a DDR-based board to an HBM-based one can result in a loss in memory bandwidth. DDR chips offer a native interface of 512 bits, while HBM channels provide a native interface of 256 bits. To maintain DDR bandwidth of 512 bits per cycle, HBM ports have to double the clock frequency compared to the user logic one. However, this approach puts more pressure on the place and route algorithm, making harder the fact of closing timing.

An alternative solution is to bond together every two 256 bit HBM channels into a single virtual 512 bit channel, while maintaining the same user clock frequency for all logic. With this feature, AIT is able to offer 16 AXI channels running at the same frequency but with double the bandwidth.

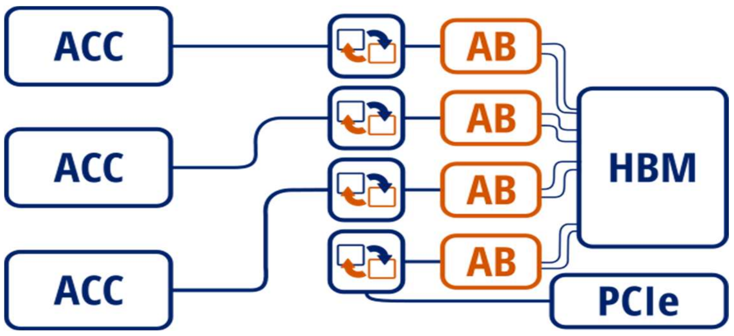

Figure 8. Memory interconnection diagram for an HBM-based discrete board with AXI bonding

As can be seen in Figure 8, from the point of view of the PCIe and the accelerators, memory interconnection is exactly the same as it exposes a single AXI interface for each virtual 512b channel. Additionally, as the exposed AXI interfaces are double the width, there is no need for data width conversion and only an AXI4-to-AXI3 protocol converter is inserted.

This specific optimization has been successfully exploited in the b8c sparse-matrix dense-vector multiplication (SpMV) [b8c] use case described in deliverable D6.4.

1.2.4 Integration with MEEP FPGA Shell

To guarantee AIT compatibility with the MEEP FPGA Shell flow, minimal changes were required on both sides, AIT and MEEP FPGA Shell. Thanks to the fact that AIT is almost fully board agnostic these modifications were relatively simple.

The main objective is for AIT to generate the Emulated Accelerator design that the MEEP FPGA Shell uses to plug it in the dynamic region of the shell and surround it with the user-defined FPGA components and I/O connectivity. The FPGA Shell then takes care of synthesizing, placing and routing the full design. To do that, the design step of the AIT flow has been adapted to support the generation of the OmpSs@FPGA core design, which only contains the accelerated kernels and hardware runtime IPs, with no external components integrated. For this purpose, a new virtual board (*fpga_virtual board*), representing a MEEP-compatible Emulated Accelerator, has been added to the AIT Xilinx backend.

Each supported board must provide, at least, three basic elements: a json containing board metadata (Appendix I), a pre-assembled tcl base design, and a tcl script implementing some basic procedures regarding address mapping, clock, reset and AXI interconnection, as well as HDL wrapper generation (Appendix II).

The *fpga_shell virtual* board appropriately exposes clock and reset signals as external interfaces, as well as both AXI and AXI-Lite interfaces while dynamically generating the *accelerator_def.csv* file that specifies the MEEP Shell configuration.

Figure 9 shows an example of a completed AIT design targeting the MEEP FPGA Shell. The OmpSs@FPGA hardware runtime (*Hardware_Runtime*) and three accelerator blocks (*calculate_forces, update_particles, solve_nbody*) are observed along with their interconnections, as well as the aforementioned exposed AXI interfaces (*S_AXI*), clock (*aclk*) and reset (*ext_reset*).

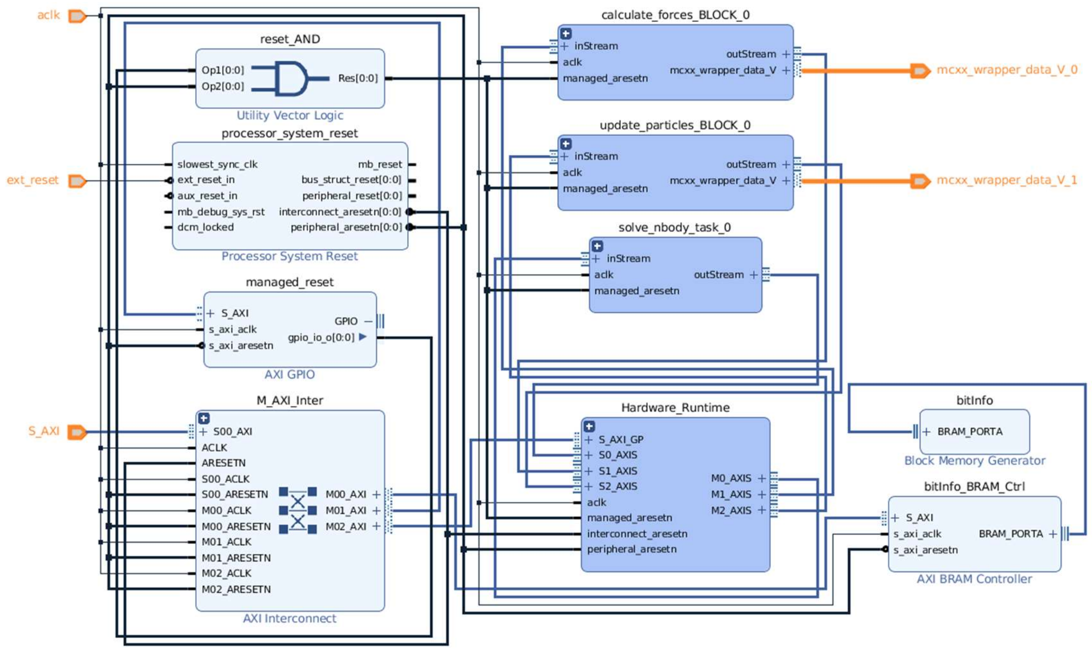

Figure 9. Block Design of an accelerated Nbody application generated by AIT

Finally, AIT generates the accelerator_mod.sv System Verilog with the Emulated Accelerator top level port definition. The MEEP FPGA Shel will parse this file to extract the EA interfaces and the necessary wires to be connected to the shell.

2. AIT + MEEP FPGA Shell

The MEEP FPGA Shell plays a pivotal role in providing seamless and efficient access to MEEP Infrastructure. This Linux-oriented tool focuses on FPGA usability and offers a comprehensive set of interfaces that can be readily utilized by users, regardless of their experience with Vivado[4]. With the MEEP FPGA Shell, users can effortlessly create, test, and validate their IPs, while enabling communication between the user IP and other elements such as the host or another FPGA. The MEEP FPGA Shell promotes homogeneity by providing standardized interfaces out of the box, ensuring a consistent user experience. Moreover, being an open-source tool, available through the corresponding GIT repository, users might broadly collaborate by proposing new features, extensions, and bug fixing, which benefits the community.

2.1 MEEP FPGA Shell

The MEEP FPGA Shell has been conceived as a wrapper for any design that wants to be emulated on an FPGA, initially targeting Alveo U280 and U55C but not limited to them. The MEEP FPGA Shell becomes a top module project which is composed of two main submodules:

1) The Shell IPs, which are all the communication IPs (i.e., Ethernet, Aurora, PCIe, etc.).

2) The Emulated Accelerator (EA), which includes the custom accelerator design to be implemented in the FPGA.

In addition, from the software perspective, the final design project also includes software drivers as a layer in between the hardware and the operating system.

The MEEP FPGA Shell sets itself apart from projects like Amazon F1 and the Xilinx Vitis Platform through its distinctive features. While one notable aspect is its configurable hardware, Shell's true differentiating factor lies in its extensive support for both hardware and software components of RISC-V-based accelerators. This comprehensive support enables seamless integration and utilization of such accelerators within the MEEP FPGA Shell.

Moreover, the versatility of the MEEP FPGA Shell extends beyond RISC-V-based designs. It can seamlessly accommodate other types of IP as well, including Neural Network emulated accelerators. The only requirement for this is to ensure the design adheres to the MEEP FPGA Shell interfaces. This flexibility allows developers to leverage any kind of accelerators within the MEEP FPGA Shell environment.

A detailed explanation of the MEEP FPGA Shell can be found in the information presented in previous deliverables, *D6.1 Platform definition and acquisition (M6), D6.2 Emulated accelerator initial release (M18), D6.3 Emulated accelerator second release* with full capability of inter-accelerator communication (M30) and in the current and final *D6.4 Full emulation prototype release (M42)*.

2.2 Envisioned programming model with AIT + MEEP FPGA Shell

AIT takes MEEP FPGA Shell approach a step further, by expanding its support, encompassing not only hardware-focused emulated accelerators but also incorporating the ability to

[4] Vivado: https://www.xilinx.com/developer/products/vivado.html

effortlessly offload software tasks to FPGA devices. For this purpose an additional Emulated Accelerator is created in the MEEP FPGA Shell to support AIT accelerators.

AIT flow, as stated before, is capable of generating OmpSs@FPGA core designs based on different applications. What the MEEP FPGA Shell will do is to surround this AIT core design with the user-defined FPGA components and I/O connectivity. After that, the MEEP FPGA Shell is responsible for synthesizing, placing and routing the full design. The resulting flow is the following:

1. AIT produces a core design, as a composition of hardware accelerators from the application benchmarks used as input (i.e., *Nbody, Matmul, Cholesky*). As a first step, the hardware runtime IPs divides the application into kernels. Then, based on that information, AIT generates and connects the different accelerator kernels, while also exposes the interfaces that these IPs need to use for I/O communication (memory, PCIe, or any other interface).
2. Next, the MEEP FPGA Shell takes the core design produced by AIT and connects its interfaces, according to the selected board and the accelerator configuration file, to the MEEP FPGA Shell.
3. As a result, a project design is generated, which includes the AIT design and the MEEP FPGA Shell together.

This process, including the call to AIT, is currently handled by the MEEP FPGA Shell. More details are explained in the following section.

2.3 AIT as a new emulated accelerator

AIT, like any other emulated accelerator seeking integration within our Shell, becomes almost fully board agnostic. AIT's responsibility is primarily focused on generating an Emulated Accelerator (EA) from the code, while the MEEP FPGA Shell utilizes it to seamlessly integrate it into the dynamic region of the FPGA. Additionally, the MEEP FPGA Shell envelops the EA with user-defined FPGA components and facilitates I/O connectivity (Figure 10). MEEP FPGA Shell subsequently handles the synthesis, placement, and routing of the complete design.

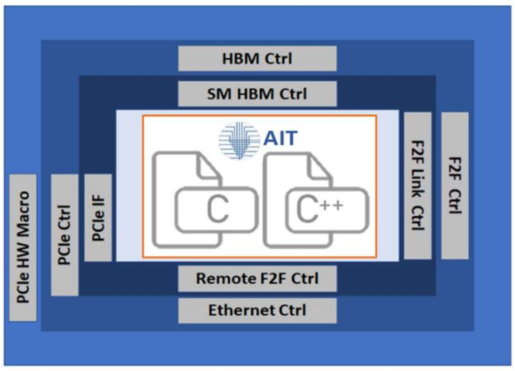

Figure 10. MEEP FPGA Shell enveloping AIT with the provided interfaces

For the connection with AIT the MEEP FPGA Shell makes available for the user all its features: the PCIe macro, two QSFP ports for FPGA to FPGA communication, such as Ethernet and Aurora, two stacks of HBM memory with 8 channels per stack, and only for the U280, two blocks of DDR4 memory. Nevertheless, AIT requires from the MEEP FPGA Shell: 1) the exposition of both AXI and AXI-Lite interfaces and 2) their associated clock and reset signals as external interfaces.

These are necessary for AIT to interact with memory and the host. For this regard a new feature to add AXI Slave interfaces was added to the MEEP FPGA Shell.

Any accelerator that seeks to be integrated with the MEEP FPGA Shell needs to fulfill the following requirements:

- Create a "***meep_shell***" folder as part of the accelerator repository.
- Include the project ***filelist*** for the accelerator.
- In case the accelerator needs to prepare any conditions before being implemented it can be done in the "***accelerator_init.sh***" file (Code snippet 1). However, this is optional.
- In case the accelerator needs to be built first, it can be done in the "*accelerator_build.sh*" file (Code snippet 2). However, this is optional.
- Define the interfaces needed from the Shell in the "***accelerator_def.csv***" file (Code snippet 3). This file is the important one.

```
git submodule update --init --recursive

pushd ait-bsc
  git lfs install
  git lfs pull
popd

AIT_Installation=`pwd`/AIT_Installation
export PATH=$AIT_Installation/bin/:$PATH
export PYTHONPATH=$AIT_Installation

export DEB_PYTHON_INSTALL_LAYOUT=deb_system
python3 -m pip install ./ait-bsc -t $AIT_Installation --upgrade
```

Code snippet 1. File "*accelerator_init.sh*" of the Emulated Accelerator AIT (ea_ait)

```
echo "Starting AIT@MEEP integration flow"

# Exporting AIT directories
AIT_Installation=`pwd`/AIT_Installation
export PATH=$AIT_Installation/bin/:$PATH
export PYTHONPATH=$AIT_Installation

EA_PARAM=${1-nbody}
EA_ROOT_DIR=`pwd`

if [ -d $EA_ROOT_DIR/benchmarks/$EA_PARAM ] ; then
  pushd $EA_ROOT_DIR/benchmarks/$EA_PARAM
    if ./build.sh; then
      mkdir -p $EA_ROOT_DIR/src
      mv ${EA_PARAM}_ait/ea_ait.v $EA_ROOT_DIR/src/
      mv ${EA_PARAM}_ait/accelerator_mod.sv $EA_ROOT_DIR/meep_shell/
      mv ${EA_PARAM}_ait/accelerator_def.csv $EA_ROOT_DIR/meep_shell/
    else
      echo "Integration of $EA_PARAM benchmark failed"
      exit 1
    fi
  popd

  sed -i "s/\(set EA_PARAM \)\(.*\)/\1${EA_PARAM}/g" \
    $EA_ROOT_DIR/meep_shell/tcl/project_options.tcl
else
  echo "Benchmark $EA_PARAM does not exist"
  exit 1
fi
```

Code snippet 2. File "*accelerator_build.sh*" of the Emulated Accelerator AIT (ea_ait)

Code snippet 1 describes the general methodology for ensuring that AIT submodule is properly cloned and installed and that the required environmental variables are correctly set before calling AIT to build our accelerator.

The above Code snippet 2 block builds prepares the condition to make a call to the `build.sh` in the standard conditions for AIT to build our accelerator by defining, among other options:

- The path to the AIT installation to be used.
- The application to be implemented among the available ones (Nbody in this case, Cholesky and MatMul).
- Moves the files top module of our accelerator, the *accelerator_mod.sv* and *accelerator_def.csv* to the *meep_shell* directory where the MEEP FPGA Shell requests them.

```
EANAME=ea_ait
DDR4,no
AURORA,no
UART,no
ETHERNET,no
BROM,no
BRAM,no
PCIE,yes,pcie_axi,,PCIE_CLK,pcie_clk,pcie_rstn,dma,0,,00,add_alite
SLV_ALITE,yes,S_AXI,1,CLK0,0x0,256K
```

Code snippet 3. File "*accelerator_def.csv*" of the Emulated Accelerator AIT (ea_ait)

The above Code snippet 3 block defines the base definition interfaces for the *accelerator_def.csv*:

- PCIe interface features: This defines the use of DMA and connects it to the HBM `S_AXI_2`.
- Slave AXI-Lite interfaces: This defines that there is going to be just one AXI-Lite Slave interface (`S_AXI`), connected to the main clock CLK0, which addresses 256KB starting from address `0x0`.
- The lines related to DDR4, Aurora, UART, Ethernet, BROM and BRAM are optional and not needed in the file *accelerator_def.csv* of the AIT accelerator because they are explicitly set to "NO" for not to create these IPs.

Additionally, in compile time AIT will include the following definitions:

- Clock and Reset: Defines the main clock, its frequency and its associated reset.
- Memory connection: connects the accelerators' memory interfaces to the HBM, specifying which HBM pseudo-channel to use, interface data width and protocol, and their associated clock and reset.

For our EA from AIT, MEEP FPGA Shell had to add support for two new features:

- PCIe AXI-Lite interface: The field *add_alite* has been added to the PCIE interface definition to enable an AXI-Lite interface on the PCIe communication IP.
- Complementary support for AXI-Lite Slave interfaces have been added to the MEEP FPGA Shell. The AXI-Lite interfaces, defined in *accelerator_def.csv* with the tag SLV_ALITE, will be connected by the MEEP FPGA Shell to the AXI-Lite master interface of the PCIe communication IP mentioned above.

Figure 11 shows an example of a completed MEEP Shell implementation design generated for an Nbody application generated by AIT. The HBM, the QDMA and the Shell InfoROM blocks can be observed along with their interconnection, as well as the AXI interfaces, clock and reset in orange that will be connected to the AIT design.

Once the bitstream is obtained, the standardized scripts to download MEEP Shell bitstreams into the FPGAs can be used (load_bitstream.sh) which have been explained in previous deliverables). With the bitstream loaded in the FPGA we can then use OmpSs@FPGA functions and scripts to get information of the AIT implementation or execute the application from the host.

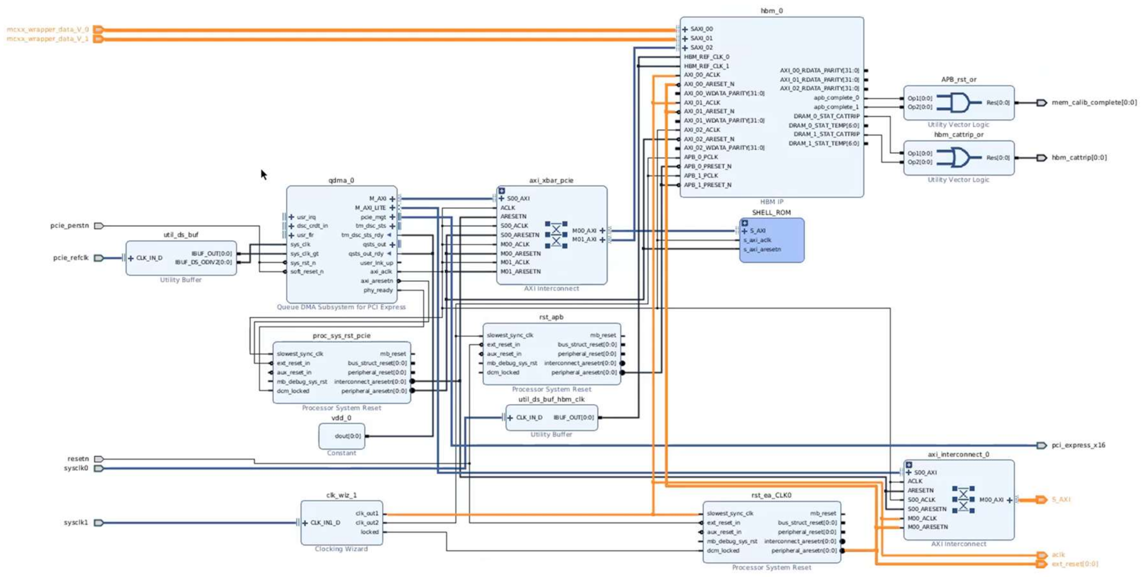

Figure 11. Block Design of MEEP Shell implementation generated for an Nbody application generated by AIT

3. AIT at MEEP release

This first AIT release extends the original MEEP FPGA Shell support, for hardware oriented emulated accelerators, to SW oriented emulated accelerators. Currently we offer support for Nbody, Cholesky and MatMul but this is planned to be extended to more benchmarks that are included in the AIT benchmark database.

3.1 Status

The current integration of AIT as an Emulated Accelerator of the MEEP FPGA Shell consists of a repository acting as a bridge between the two standalone tool repositories. Full repository structure can be seen in Figure 12.

Inside the MEEP FPGA Shell repository there is a file called *ea_url.txt* within the directory *support/ea_ait* which contains the URL and commit hash of the target AIT@MEEP bridge repository. This specific commit will be cloned when initializing the FPGA Shell project environment.

At the same time, in the AIT@MEEP repository there is the aforementioned directory meep_shell required by the FPGA Shell flow, which contains both accelerator_init.sh and *accelerator_build.sh* scripts and *project_options.tcl* defining some FPGA Shell tcl variables.

This bridge repository also contains a git submodule pointing to the AIT main repository [AIT], so all new features and developments can be automatically exploited by the FPGA Shell without the need of further integration efforts.

Finally, the bridge repository contains the available applications HLS code and building scripts inside the benchmarks repository.

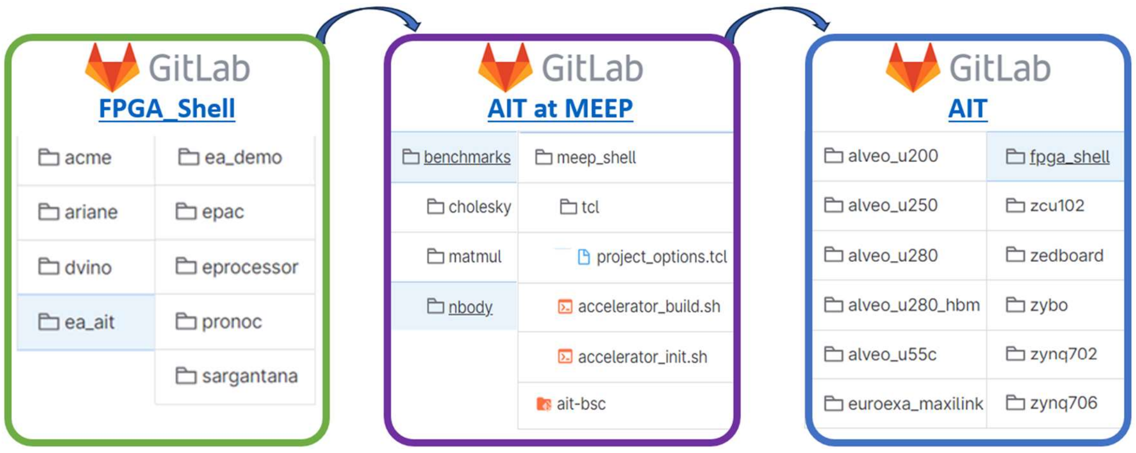

Figure 12. Repository structure of the AIT@MEEP integration

Code snippet 4 contains an example *build.sh* script for the Nbody application, showing the variables defined as well as the AIT call. Additionally, it moves the output products to the application root directory.

```
BENCHMARK=nbody
BOARD=fpga_shell
FPGA_CLOCK=300
WRAPPER_VERSION=13

if [ -d ${BENCHMARK}_ait ]; then
  rm -r ${BENCHMARK}_ait
fi

ait --name=$BENCHMARK \
    --board=$BOARD \
    -c=$FPGA_CLOCK \
    --interconnect_opt=performance \
    --max_deps_per_task=3 \
    --max_args_per_task=11 \
    --max_copies_per_task=11 \
    --picos_tm_size=32 \
    --picos_dm_size=102 \
    --picos_vm_size=102 \
    --to_step=design \
    --wrapper_version=$WRAPPER_VERSION

mv ${BENCHMARK}_ait/xilinx/accelerator_mod.sv ${BENCHMARK}_ait/
mv ${BENCHMARK}_ait/xilinx/board/fpga_shell/accelerator_def.csv ${BENCHMARK}_ait/
mv ${BENCHMARK}_ait/xilinx/ea_ait.v ${BENCHMARK}_ait/
```

Code snippet 4. Building script "*build.sh*" of the Nbody application for the Emulated Accelerator AIT (ea_ait)

Each application building script must define, among other OmpSs@FPGA options:

- The AIT target board, in this case *fpga_shell*.
- The target FPGA clock frequency.
- The version of the OmpSs@FPGA HLS wrapper, which depends on the compiler used to generate the wrapper.
- Application-specific optimization flags, such as memory interconnection tuning, hardware runtime configuration or placement and floorplanning constraints.
- The AIT call, running the tool up to the *design* step.

Once executed, this script will generate the design shown in Figure 9, the interface definition file shown in Code snippet 5 and the Emulated Accelerator top wrapper file *accelerator_mod.sv* (Appendix III).

Interface definition file *accelerator_def.csv* is populated with application-specific clock and interfaces during AIT execution:

- Accelerator clock interface `aclk` is set as `CLK0` of the FPGA Shell, configured to 300MHz frequency, and associated with `ext_reset`, which is a low active reset.
- Accelerator data ports `mcxx_wrapper_data_V_0` and `mcxx_wrapper_data_V_1`, defined as `CLK0-synchronous` `256b` `AXI3` interfaces, will be connected to the HBM `S_AXI_1` and `S_AXI_2` pseudo-channels respectively.

Continuing with the Nbody EA generation using AIT, the MEEP FPGA Shell might produce the final design depicted in Figure 11, which are the designs shown in Figure 9 and Figure 11 connected in a *system_top.v.* In this case, the user will need to generate the bitstream manually later. The MEEP FPGA Shell might also deliver the bitstream. This happens when it is set to complete all the steps of the FPGA flow.

```
EANAME=ea_ait
DDR4,no
AURORA,no
UART,no
ETHERNET,no
BROM,no
BRAM,no
PCIE,yes,pcie_axi,,PCIE_CLK,pcie_clk,pcie_rstn,dma,0,,00,add_alite
SLV_ALITE,yes,S_AXI,1,CLK0,0x0,256K
CLK0,300000000,aclk,ext_reset,LOW
HBM,yes,mcxx_wrapper_data_V_0,AXI3-256,CLK0,0x0,mem_calib_complete,01
HBM,yes,mcxx_wrapper_data_V_1,AXI3-256,CLK0,0x0,mem_calib_complete,02
```

Code snippet 5. Interface definition file "*accelerator_def.csv*" of the Nbody application for the Emulated Accelerator AIT (ea_ait)

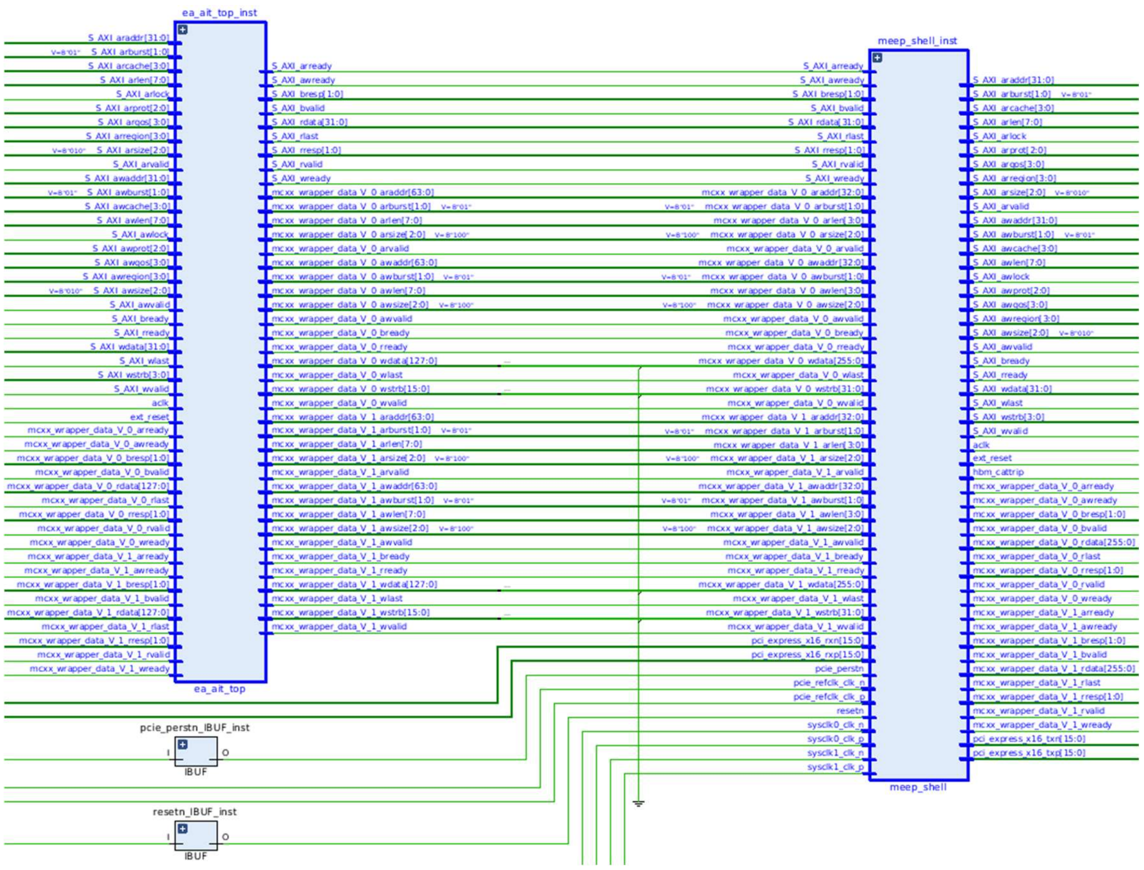

Figure 13. Schematic Design of AIT + MEEP FPGA Shell implementation

3.2 Results

As a proof of correctness and functionality of the AIT+MEEP FPGA Shell integration, this section presents the results of the Nbody application. More details about the Nbody code are included in Appendix IV. There, a compilation example of Nbody code can be found, including Nbody functions, and the corresponding pragmas to allow OmpSs@FPGA converting those into hardware accelerators in the FPGA.

Once all the process of calling AIT, building the Shell around the design and the bitstream is obtained, the standardized scripts to download MEEP FPGA Shell bitstreams into the FPGAs can be used (load_bitstream.sh) which have been explained in previous deliverables. With the bitstream loaded in the FPGA, OmpSs@FPGA functions and scripts (*read_bitinfo*) can be used to get the information contained into the AIT *bitinfo ROM* (Figure 14) through our MEEP FPGA Shell QDMA interface as a proof of concept.

```
user@picu: read_bitinfo
Bitstream info version: 10
Number of acc: 3
AIT version: 6.7.12
Wrapper version 13
Board base frequency (Hz) 300
Interleaving stride 0
Features:
[ ] Instrumentation
[ ] Hardware counter
[x] Performance interconnect
[ ] Simplified interconnection
[ ] POM AXI-Lite
[x] POM task creation
[x] POM dependencies
[x] POM lock
[x] POM spawn queues
[ ] CMS enabled
Cmd In addr 0x6000 len 256
Cmd Out addr 0x8000 len 256
Spawn In addr 0x2000 len 1024
Spawn Out addr 0x4000 len 1024
Managed rstn addr 0xA000
Hardware counter addr 0x0
POM AXI-Lite addr 0x0
CMS addr 0x0

xtasks accelerator config:
type        count  freq(KHz) description
7705865291  1      300000    solve_nbody_task
6057883607  1      300000    calculate_forces_BLOCK
5319290900  1      300000    update_particles_BLOCK

ait command line:
ait --name=nbody --board=fpga_shell -c=300 --interconnect_opt=performance --
max_deps_per_task=3 --max_args_per_task=11 --max_copies_per_task=11 --
picos_tm_size=32 --picos_dm_size=102 --picos_vm_size=102 --to_step=design --
wrapper_version=13

Hardware runtime VLNV:
bsc:ompss:picosompssmanager:7.1
bitinfo note: ‘ ‘
```

Figure 14. Output of the OmpSs@FPGA *read_bitinfo* tool showing bitinfo contents

These first results show that the access from the host to the MEEP FPGA Shell and from it to the AIT bitinfo ROM over PCIe is correct. This info also shows the features set the Nbody benchmark to have such as:

- The total number and frequency of the Nbody accelerators.
- The Hardware runtime parameters.
- The AIT, bitstream and wrapper versions.

```
=== Check 8192_2048_1_debug ===
> Result validation: OK
==================== RESULTS =====================
Benchmark: N-Body (OmpSs)
Total particles: 8192
Timesteps: 1
```

```
Verification: successful
Warm up time (secs): 0.000000
Execution time (secs): 0.030019
Flush time (secs): 0.000535
Throughput                            (gpairs/s):                              2.235512
       ==================================================
=== Check 8192_2048_50_debug ===
> Result validation: OK
==================== RESULTS =====================
Benchmark: N-Body (OmpSs)
Total particles: 8192
Timesteps: 50
Verification:
successful Warm up time (secs): 0.000000
Execution time (secs): 1.418851
Flush time (secs): 0.000491
Throughput                            (gpairs/s):                              2.364902
       ==================================================
=== Check 32768_2048_32_perf ===
==================== RESULTS =====================
Benchmark: N-Body (OmpSs)
Total particles: 32768
Timesteps: 32
Verification: n/a
Warm up time (secs): 0.000000
Execution time (secs): 14.533341
Flush time (secs): 0.000615
Throughput (gpairs/s): 2.364201
==================================================
```

Figure 15. Output of the nbody execution

By executing the application using the conventional OmpSs standard flow and scripts, the obtained results in terms of performance and execution times, are the same of those previously achieved with AIT without utilizing the MEEP FPGA Shell for the same number of accelerators. This outcome serves as compelling evidence that the integration of AIT with the MEEP FPGA Shell has been successful and functions seamlessly.

The proper functioning of the hardware runtime, accurate memory access, and successful operation of the PCIe interface further contribute to the overall efficiency and reliability of the integrated system. These factors indicate that the underlying hardware components are performing their intended tasks correctly, facilitating the smooth execution of the application.

The holistic nature of this proof of concept is crucial as it validates the comprehensive integration between AIT and the MEEP FPGA Shell. It showcases that the combined solution effectively leverages the capabilities of both components to deliver the desired outcomes. This achievement not only demonstrates the technical feasibility of the integration but also instills confidence in the future development and deployment of the integrated system.

In summary, the successful execution of the application using the OmpSs standard flow and scripts, coupled with consistent performance results and proper functioning of hardware components, serves as a robust and comprehensive validation of the integration between AIT and the Shell. This achievement establishes a strong foundation for further advancements and wider adoption of the integrated solution.

4. MEEP contributions to other projects from the FPGA perspective

The implementation efforts explained in this document will be exploited by other projects. Such as the OmpSs@FPGA toolchain and its related projects.

OmpSs@FPGA profits greatly from the development done in AIT in the context of the MEEP project by adding support for two additional boards (Alveo U280 and Alveo U55C), the integration of HBM-based designs, and the ability to generate self-contained designs in order to perform simulations and tests over HDL code without the need to generate bitstreams.

In the same direction, **Textarossa** [TEXTAROSSA] project will also benefit from the integration efforts done for the Alveo U280 board.

MEEP will profit from the extension done with AIT and the MEEP FPGA Shell to provide support not only for hardware oriented emulated accelerators but also for software oriented emulated accelerators. For the MEEP FPGA Shell as a Tool there have been improvements because of the need of AIT to support QDMA Lite and Slave AXI Lite interfaces. In addition, we have done fists steps on the automatization in the creation of the *accelerator_def.csv* file to create its configuration lines associated with the actual interfaces of the emulated accelerator that is going to be implemented.

Colleagues from the University of Paderborn have shown interest in using the MEEP FPGA Shell, and they might find this AIT feature appealing. Regular meetings have been scheduled to give them support on the use of the MEEP FPGA Shell. This effort might open the possibility of future collaborations and contributions to the extension of these tools.

4.1 Next steps

4.1.1 Fast-prototyping, power consumption, area and frequency estimation

Traditionally, FPGA development follows a sequential flow, starting from an algorithmic description of the application, followed by architectural synthesis to obtain an RTL-level description. This is then followed by the logic synthesis and place & route steps to generate a final circuit description with precise area and performance values. This process is time-consuming process. Fast Prototyping is an emerging design methodology that aims to achieve rapid and reliable integration solutions capable of meeting various design constraints.

Fast Prototyping introduces a higher level of abstraction, encompassing specification to system-level design. It includes control structures, multi-dimensional data, and hierarchy to address the complexities of modern applications. It also takes into account all units of the architecture, such as data paths, control units, and memory units, to provide a realistic cost characterization. The methodology explores application parallelism, considering multiple architectural solutions for a given specification across various FPGA families.

The benefits of Fast Prototyping extend beyond delivering feasible solutions. It also provides valuable information for architectural design exploration of the targeted design. Furthermore, its low complexity enables a larger design space exploration. This methodology can be seen as a global exploration estimation technique, drawing on existing works in estimation and high-level synthesis (HLS), including memory size estimation, scheduling techniques, and data flow modeling. However, the key difference lies in its emphasis on defining efficient architectures.

By employing Fast Prototyping, users can obtain immediate yet approximate results regarding power consumption, design area estimation, and maximum achievable frequency. While these results may not be highly accurate, they offer valuable insights to guide the design process effectively.

5. Conclusion

The development efforts described in this deliverable have yielded two-fold improvements both in AIT and the MEEP FPGA Shell.

Firstly, AIT benefits from the integration by incorporating a configurable FPGA shell. This integration allows programmers to fully customize the shell of the target board, enabling them to choose the specific communication IPs, memory modules, and network interfaces they desire in the final bitstream. This level of customization enhances the flexibility and adaptability of AIT, empowering programmers to tailor the FPGA shell to their specific requirements.

Secondly, the MEEP FPGA Shell expands its support beyond its initial focus on RISC-V based hardware emulated accelerators. This expansion firstly evolved to encompass a wide range of hardware accelerators that adhere to the Shell requirements. Moreover, with the integration of AIT, the MEEP FPGA Shells have taken a significant stride towards supporting not only hardware (HW) but also software (SW) based emulated accelerators. This expanded support opens up new possibilities for utilizing the MEEP FPGA Shell across a wider range of hardware and software acceleration scenarios.

Furthermore, the MEEP FPGA Shell has incorporated certain features that were not initially part of its design but are essential for AIT's requirements. These added features, such as QDMA Lite or Slave AXI Lite ports, are now available for any accelerator that may require them. Overall, these advancements in the MEEP FPGA Shells' support and feature set demonstrate their increased versatility and capability to accommodate various types of accelerators.

In summary, the development efforts have brought about dual improvements in both AIT and the MEEP FPGA Shell. AIT benefits from a customizable FPGA shell, enabling programmers to tailor the target board's shell to their specific needs. Meanwhile, the MEEP FPGA Shell expands its support to a wider range of hardware and software accelerators, while incorporating additional features required by AIT. This integration represents a significant advancement in the capabilities and versatility of both AIT and the MEEP FPGA Shell.

6. References

[AIT] Programming Models Group (BSC-CNS), Accelerator Integration Tool (AIT). https://github.com/bsc-pm-ompss-at-fpga/ait

[OMPSS-FPGA] J. M. de Haro *et al.*, "OmpSs@FPGA Framework for High Performance FPGA Computing," IEEE Transactions on Computers, vol. 70, no. 12, pp. 2029-2042, 2021, doi: 10.1109/TC.2021.3086106

[b8c] J. Oliver *et al.*, "b8c: SpMV accelerator implementation leveraging high memory bandwidth," *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023, doi: 10.1109/FCCM57271.2023.00044

[EUROEXA] EuroEXA project. https://euroexa.eu

[TEXTAROSSA] Textarossa project. https://textarossa.eu

Appendix I

basic_info.json describing the *fpga_shell* AIT board.

```json
{
  "name": "fpga_shell",
  "chip_part": "xcu280-fsvh2892-2L-e",
  "es": "",
  "board_part": "",

  "arch": {
    "device": "shell"
  },

  "frequency": {
    "min": 5,
    "max": 933
  },

  "mem": {
    "type": "hbm",
    "num_banks": 32,
    "bank_size": "256M"
  },

  "address_map": {
    "mem_base_addr": "0x0",
    "ompss_base_addr": "0x0"
  }
}
```

Appendix II

procs.tcl script file containing some AIT board-specific procedures definitions.

```
namespace eval AIT {
  namespace eval board {
    # HBM port 00 used by QDMA
    variable external_axi_num 1

    # Return an arbitrary maximum
    proc get_available_axi_intfs {} {
      return 99
    }

    rename connect_to_axi_intf generic_connect_to_axi_intf

    # If connecting an slave interface, generate external port
    # and add the interface to accelerator_def.csv file
    proc connect_to_axi_intf {src mode {num ""}} {
      if {$mode eq "M"} {
        generic_connect_to_axi_intf $src $mode $num
      } else {
        variable external_axi_num

        make_bd_intf_pins_external $src

        set ext_intf_port [string trim [get_bd_intf_ports -of_objects \
                                        [get_bd_intf_nets -of_objects \
                                         [get_bd_intf_pins $src]]] {/}]

        set acc_def_file [open board/${::AIT::board}/accelerator_def.csv "a"]
        set axi_def "HBM,yes,$ext_intf_port,AXI3-256,CLK0,0x0, \
                    mem_calib_complete,[format %02u $external_axi_num]"
        puts $acc_def_file $axi_def
        close $acc_def_file

        incr external_axi_num
      }
    }

    proc connect_clock {src_clk} {
      connect_bd_net $src_clk [get_bd_ports aclk]
    }

    # Generate CLK0 interface in accelerator_def.csv file
    proc set_and_get_freq {targetFreq} {
      set acc_def_file [open board/${::AIT::board}/accelerator_def.csv "a"]
      set clk_def "CLK0,[expr $targetFreq*1000000],aclk,ext_reset,LOW"
      puts $acc_def_file $clk_def
      close $acc_def_file

      return $targetFreq
    }

    proc get_base_freq {} {
      return ${::AIT::clockFreq}
    }

    proc configure_address_map {} {
      foreach data_intf [get_bd_addr_segs *mcxx_*] {
        assign_bd_address $data_intf -offset 0x0 -range 8G
      }
    }

    # Generate ea_ait.v and accelerator_top.v
    proc generate_wrapper {} {
      set_property target_language verilog [current_project]

      make_wrapper -files [get_files [current_bd_design].bd] -top -import -force

      exec cp [get_files *_wrapper.v] ea_ait.v
      exec echo "module ea_ait_top (" > accelerator_mod.sv
      exec sed "/input/,\$!d" ea_ait.v >> accelerator_mod.sv
      exec sed -i "/wire/,\$d" accelerator_mod.sv
      exec sed -i -e "s/;/\,/g" accelerator_mod.sv
      exec sed -i -e "s/]/] /g" accelerator_mod.sv
      exec echo ");" >> accelerator_mod.sv
      exec cat accelerator_mod.sv > accelerator_top.v
      exec echo "\nendmodule" >> accelerator_mod.sv
      exec sed -i -z -e "s/\,\\n\\n\);/\\n\);/g" accelerator_mod.sv
      exec sed -i "/${::AIT::name_Design} ${::AIT::name_Design}_i/,\$!d" ea_ait.v
      exec cat ea_ait.v >> accelerator_top.v
      exec sed -i -e "s/,/ ,/g" accelerator_top.v
      exec mv accelerator_top.v ea_ait.v
      exec sed -i -z -e "s/\,\\n\\n\);/\\n\);/g" ea_ait.v
    }
  }
}
```

Appendix III

Emulated Accelerator top wrapper file accelerator_mod.sv generated by AIT for an nbody application.

```
module ea_ait_top (
  input [31:0] S_AXI_araddr,
  input [1:0] S_AXI_arburst,
  input [3:0] S_AXI_arcache,
  input [7:0] S_AXI_arlen,
  input [0:0] S_AXI_arlock,
  input [2:0] S_AXI_arprot,
  input [3:0] S_AXI_arqos,
  output S_AXI_arready,
  input [3:0] S_AXI_arregion,
  input [2:0] S_AXI_arsize,
  input S_AXI_arvalid,
  input [31:0] S_AXI_awaddr,
  input [1:0] S_AXI_awburst,
  input [3:0] S_AXI_awcache,
  input [7:0] S_AXI_awlen,
  input [0:0] S_AXI_awlock,
  input [2:0] S_AXI_awprot,
  input [3:0] S_AXI_awqos,
  output S_AXI_awready,
  input [3:0] S_AXI_awregion,
  input [2:0] S_AXI_awsize,
  input S_AXI_awvalid,
  input S_AXI_bready,
  output [1:0] S_AXI_bresp,
  output S_AXI_bvalid,
  output [31:0] S_AXI_rdata,
  output S_AXI_rlast,
  input S_AXI_rready,
  output [1:0] S_AXI_rresp,
  output S_AXI_rvalid,
  input [31:0] S_AXI_wdata,
  input S_AXI_wlast,
  output S_AXI_wready,
  input [3:0] S_AXI_wstrb,
  input S_AXI_wvalid,
  input aclk,
  input ext_reset,
  output [63:0] mcxx_wrapper_data_V_0_araddr,
  output [1:0] mcxx_wrapper_data_V_0_arburst,
  output [3:0] mcxx_wrapper_data_V_0_arcache,
  output [7:0] mcxx_wrapper_data_V_0_arlen,
  output [1:0] mcxx_wrapper_data_V_0_arlock,
  output [2:0] mcxx_wrapper_data_V_0_arprot,
  output [3:0] mcxx_wrapper_data_V_0_arqos,
  input mcxx_wrapper_data_V_0_arready,
  output [3:0] mcxx_wrapper_data_V_0_arregion,
  output [2:0] mcxx_wrapper_data_V_0_arsize,
  output mcxx_wrapper_data_V_0_arvalid,
  output [63:0] mcxx_wrapper_data_V_0_awaddr,
  output [1:0] mcxx_wrapper_data_V_0_awburst,
  output [3:0] mcxx_wrapper_data_V_0_awcache,
  output [7:0] mcxx_wrapper_data_V_0_awlen,
  output [1:0] mcxx_wrapper_data_V_0_awlock,
  output [2:0] mcxx_wrapper_data_V_0_awprot,
  output [3:0] mcxx_wrapper_data_V_0_awqos,
  input mcxx_wrapper_data_V_0_awready,
  output [3:0] mcxx_wrapper_data_V_0_awregion,
  output [2:0] mcxx_wrapper_data_V_0_awsize,
  output mcxx_wrapper_data_V_0_awvalid,
  output mcxx_wrapper_data_V_0_bready,
  input [1:0] mcxx_wrapper_data_V_0_bresp,
  input mcxx_wrapper_data_V_0_bvalid,
  input [127:0] mcxx_wrapper_data_V_0_rdata,
  input mcxx_wrapper_data_V_0_rlast,
  output mcxx_wrapper_data_V_0_rready,
  input [1:0] mcxx_wrapper_data_V_0_rresp,
  input mcxx_wrapper_data_V_0_rvalid,
  output [127:0] mcxx_wrapper_data_V_0_wdata,
  output mcxx_wrapper_data_V_0_wlast,
  input mcxx_wrapper_data_V_0_wready,
  output [15:0] mcxx_wrapper_data_V_0_wstrb,
  output mcxx_wrapper_data_V_0_wvalid,
  output [63:0] mcxx_wrapper_data_V_1_araddr,
  output [1:0] mcxx_wrapper_data_V_1_arburst,
  output [3:0] mcxx_wrapper_data_V_1_arcache,
  output [7:0] mcxx_wrapper_data_V_1_arlen,
  output [1:0] mcxx_wrapper_data_V_1_arlock,
  output [2:0] mcxx_wrapper_data_V_1_arprot,
  output [3:0] mcxx_wrapper_data_V_1_arqos,
  input mcxx_wrapper_data_V_1_arready,
  output [3:0] mcxx_wrapper_data_V_1_arregion,
  output [2:0] mcxx_wrapper_data_V_1_arsize,
  output mcxx_wrapper_data_V_1_arvalid,
  output [63:0] mcxx_wrapper_data_V_1_awaddr,
  output [1:0] mcxx_wrapper_data_V_1_awburst,
  output [3:0] mcxx_wrapper_data_V_1_awcache,
  output [7:0] mcxx_wrapper_data_V_1_awlen,
  output [1:0] mcxx_wrapper_data_V_1_awlock,
  output [2:0] mcxx_wrapper_data_V_1_awprot,
  output [3:0] mcxx_wrapper_data_V_1_awqos,
  input mcxx_wrapper_data_V_1_awready,
  output [3:0] mcxx_wrapper_data_V_1_awregion,
  output [2:0] mcxx_wrapper_data_V_1_awsize,
  output mcxx_wrapper_data_V_1_awvalid,
  output mcxx_wrapper_data_V_1_bready,
  input [1:0] mcxx_wrapper_data_V_1_bresp,
  input mcxx_wrapper_data_V_1_bvalid,
  input [127:0] mcxx_wrapper_data_V_1_rdata,
  input mcxx_wrapper_data_V_1_rlast,
  output mcxx_wrapper_data_V_1_rready,
  input [1:0] mcxx_wrapper_data_V_1_rresp,
  input mcxx_wrapper_data_V_1_rvalid,
  output [127:0] mcxx_wrapper_data_V_1_wdata,
  output mcxx_wrapper_data_V_1_wlast,
  input mcxx_wrapper_data_V_1_wready,
  output [15:0] mcxx_wrapper_data_V_1_wstrb,
  output mcxx_wrapper_data_V_1_wvalid
);

endmodule
```

Appendix IV

Emulated Accelerator top wrapper file *accelerator_mod.sv* generated by AIT for an nbody application.

calculate_forces_BLOCK_ompss.cpp

```
///////////////////
// Automatic IP Generated by OmpSs@FPGA compiler
///////////////////
// The below code is composed by:
//  1) User source code, which may be under any license (see in original source
code)
//  2) OmpSs@FPGA toolchain code which is licensed under LGPLv3 terms and
conditions
///////////////////
// Top IP Function: calculate_forces_BLOCK_wrapper
// Accel. type hash: 6057883607
// Num. instances: 1
// Wrapper version: 13
///////////////////
#define __HLS_AUTOMATIC_MCXX__ 1

#include <cstring>
#include <hls_stream.h>
#include <ap_int.h>

#include "src/kernel.fpga.h"
unsigned long long int __mcxx_raw_params[11];
static unsigned long long int __mcxx_taskId;
static unsigned long long int __mcxx_parent_taskId;
extern hls::stream<ap_uint<64> > mcxx_inPort;
extern ap_uint<68> mcxx_outPort;
extern ap_uint<128> * mcxx_wrapper_data;
template<typename T> void __mcxx_memcpy_port_in(T * dst, const unsigned long long
int src, const size_t len);
template<typename T> void __mcxx_memcpy_port_out(const unsigned long long int dst,
const T * src, const size_t len);
void __mcxx_send_finished_task_cmd(const unsigned char destId);

static const unsigned int NCALCFORCES = 8;
const unsigned int FPGA_PWIDTH = 128;
static float __mcxx_sqrtf(float x);
static void calculate_forces_part(const float pos_x1, const float pos_y1, const
float pos_z1, const float mass1, const float pos_x2, const float pos_y2, const
float pos_z2, const float weight2, float *fx, float *fy, float *fz)
{
#pragma HLS inline
  const float local_x = *fx;
  const float local_y = *fy;
  const float local_z = *fz;
  const float diff_x = pos_x2 - pos_x1;
  const float diff_y = pos_y2 - pos_y1;
  const float diff_z = pos_z2 - pos_z1;
  const float distance_squared = diff_x * diff_x + diff_y * diff_y + diff_z *
diff_z;
  const float distance = __mcxx_sqrtf(distance_squared);
  const float force = mass1 / (distance_squared * distance) * weight2;
  const float force_corrected = distance_squared == 0.00000000000000000000000e+00f
? 0.00000000000000000000000e+00f : force;
  *fx = local_x + force_corrected * diff_x;
  *fy = local_y + force_corrected * diff_y;
  *fz = local_z + force_corrected * diff_z;
```

```
}
static const unsigned int BLOCK_SIZE = 2048;
static void calculate_forces_BLOCK_moved(float *x, float *y, float *z, const float
*pos_x1, const float *pos_y1, const float *pos_z1, const float *mass1, const float
*pos_x2, const float *pos_y2, const float *pos_z2, const float *weight2)
{
  int j;
  int i;
#pragma HLS inline
#pragma HLS array_partition variable=x cyclic factor=NCALCFORCES
#pragma HLS array_partition variable=y cyclic factor=NCALCFORCES
#pragma HLS array_partition variable=z cyclic factor=NCALCFORCES
#pragma HLS array_partition variable=pos_x1 cyclic factor=NCALCFORCES/2
#pragma HLS array_partition variable=pos_y1 cyclic factor=NCALCFORCES/2
#pragma HLS array_partition variable=pos_z1 cyclic factor=NCALCFORCES/2
#pragma HLS array_partition variable=mass1 cyclic factor=NCALCFORCES/2
#pragma HLS array_partition variable=pos_x2 cyclic factor=FPGA_PWIDTH/64
#pragma HLS array_partition variable=pos_y2 cyclic factor=FPGA_PWIDTH/64
#pragma HLS array_partition variable=pos_z2 cyclic factor=FPGA_PWIDTH/64
#pragma HLS array_partition variable=weight2 cyclic factor=FPGA_PWIDTH/64
  for (j = 0; j < BLOCK_SIZE; j++)
    {
      for (i = 0; i < BLOCK_SIZE; i++)
        {
#pragma HLS pipeline II=1
#pragma HLS unroll factor=NCALCFORCES
          calculate_forces_part(pos_x1[i], pos_y1[i], pos_z1[i], mass1[i],
pos_x2[j], pos_y2[j], pos_z2[j], weight2[j], &x[i], &y[i], &z[i]);
        }
    }
}
void calculate_forces_BLOCK_wrapper() {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_hs port=mcxx_inPort
#pragma HLS INTERFACE ap_hs port=mcxx_outPort
#pragma HLS INTERFACE m_axi port=mcxx_wrapper_data

  unsigned long long int __commandArgs, __bufferData;
  unsigned char __commandCode;
  static ap_uint<1> __reset = 0;
  #pragma HLS RESET variable=__reset
  static float x[2048L];
  static float y[2048L];
  static float z[2048L];
  static float pos_x1[2048L];
  static float pos_y1[2048L];
  static float pos_z1[2048L];
  static float mass1[2048L];
  static float pos_x2[2048L];
  static float pos_y2[2048L];
  static float pos_z2[2048L];
  static float weight2[2048L];

  if (__reset == 0) {
    __reset = 1;
  }
  __bufferData = mcxx_inPort.read();
  __commandCode = __bufferData;
  __commandArgs = __bufferData>>8;
  if (__commandCode == 1) {
    unsigned char __i;
    ap_uint<8> __paramInfo[11];
    unsigned char __comp_needed, __destID;
    __mcxx_taskId = mcxx_inPort.read();
    __mcxx_parent_taskId = mcxx_inPort.read();
    __comp_needed = __commandArgs>>24;
    __destID = __commandArgs>>32;
    for (__i = 0;
    __i < 11;
    __i++) {
      __paramInfo[__i] = mcxx_inPort.read();
      __mcxx_raw_params[__i] = mcxx_inPort.read();
    }
    in_copies_region: {
      if (__paramInfo[0][4]) {
        __mcxx_memcpy_port_in<float>(x,__mcxx_raw_params[0],(2048L));
      }
      if (__paramInfo[1][4]) {
        __mcxx_memcpy_port_in<float>(y,__mcxx_raw_params[1],(2048L));
      }
      if (__paramInfo[2][4]) {
        __mcxx_memcpy_port_in<float>(z,__mcxx_raw_params[2],(2048L));
      }
      if (__paramInfo[3][4]) {
        __mcxx_memcpy_port_in<float>(pos_x1,__mcxx_raw_params[3],(2048L));
      }
      if (__paramInfo[4][4]) {
        __mcxx_memcpy_port_in<float>(pos_y1,__mcxx_raw_params[4],(2048L));
      }
      if (__paramInfo[5][4]) {
        __mcxx_memcpy_port_in<float>(pos_z1,__mcxx_raw_params[5],(2048L));
      }
      if (__paramInfo[6][4]) {
        __mcxx_memcpy_port_in<float>(mass1,__mcxx_raw_params[6],(2048L));
      }
      if (__paramInfo[7][4]) {
        __mcxx_memcpy_port_in<float>(pos_x2,__mcxx_raw_params[7],(2048L));
      }
      if (__paramInfo[8][4]) {
        __mcxx_memcpy_port_in<float>(pos_y2,__mcxx_raw_params[8],(2048L));
      }
      if (__paramInfo[9][4]) {
        __mcxx_memcpy_port_in<float>(pos_z2,__mcxx_raw_params[9],(2048L));
      }
      if (__paramInfo[10][4]) {
        __mcxx_memcpy_port_in<float>(weight2,__mcxx_raw_params[10],(2048L));
      }
    }
    if (__comp_needed) {
      calculate_forces_BLOCK_moved(x, y, z, pos_x1, pos_y1, pos_z1, mass1, pos_x2,
pos_y2, pos_z2, weight2);
    }
    out_copies_region: {
      if (__paramInfo[0][5]) {
        __mcxx_memcpy_port_out<float>(__mcxx_raw_params[0],x,(2048L));
      }
      if (__paramInfo[1][5]) {
        __mcxx_memcpy_port_out<float>(__mcxx_raw_params[1],y,(2048L));
      }
      if (__paramInfo[2][5]) {
        __mcxx_memcpy_port_out<float>(__mcxx_raw_params[2],z,(2048L));
      }
    }
    send_finished_task_cmd: {
      #pragma HLS PROTOCOL fixed
      __mcxx_send_finished_task_cmd(__destID);
    }
  }
}
void __mcxx_write_out_port(const unsigned long long int data, const unsigned short
dest, const unsigned char last){
#pragma HLS INTERFACE ap_hs port=mcxx_outPort register
  ap_uint<68> tmp = data;
  tmp = (tmp << 4) | ((dest & 0x7) << 1) | (last & 0x1);
  mcxx_outPort = tmp;
}

void __mcxx_send_finished_task_cmd(const unsigned char destId){
  unsigned long long int header = 0x03;
  __mcxx_write_out_port(header, destId, 0);
  __mcxx_write_out_port(__mcxx_taskId, destId, 0);
  __mcxx_write_out_port(__mcxx_parent_taskId, destId, 1);
}

float __mcxx_sqrtf(float x){
#pragma HLS INLINE
  return sqrtf(x);
}

template<typename T> void __mcxx_memcpy_port_in(T * local, const unsigned long long
int addr, const size_t len){
#pragma HLS INTERFACE m_axi port=mcxx_wrapper_data
#pragma HLS inline
  for (unsigned int __j=0;
   __j<(len == 0 ? 0 : (((len)*sizeof(T) - 1)/sizeof(ap_uint<128>)+1));
   __j++) {
    ap_uint<128> __tmpBuffer;
    __tmpBuffer = *(mcxx_wrapper_data + addr/sizeof(ap_uint<128>) + __j);
    #pragma HLS PIPELINE
    #pragma HLS UNROLL region
    for (unsigned int __k=0;
     __k<((sizeof(ap_uint<128>)/sizeof(T)));
     __k++) {
    if (((__j*(sizeof(ap_uint<128>)/sizeof(T))+__k) >= (len))) continue;
      union {
        unsigned long long int raw;
        T typed;
      }
 cast_tmp;
      cast_tmp.raw = __tmpBuffer.range((__k+1)*sizeof(T)*8-1,__k*sizeof(T)*8);
      #pragma HLS DEPENDENCE variable=local inter false
      local[__j*(sizeof(ap_uint<128>)/sizeof(T))+__k] = cast_tmp.typed;
    }
  }
}

template<typename T> void __mcxx_memcpy_port_out(const unsigned long long int addr,
const T * local, const size_t len){
#pragma HLS INTERFACE m_axi port=mcxx_wrapper_data
#pragma HLS inline
  for (unsigned int __j=0;
   __j<(len == 0 ? 0 : (((len)*sizeof(T)-1)/sizeof(ap_uint<128>)+1));
   __j++) {
    ap_uint<128> __tmpBuffer;
    #pragma HLS PIPELINE
    #pragma HLS UNROLL region
    for (unsigned int __k=0;
     __k<((sizeof(ap_uint<128>)/sizeof(T)));
     __k++) {
    if (((__j*(sizeof(ap_uint<128>)/sizeof(T))+__k) >= (len))) continue;
      union {
        unsigned long long int raw;
        T typed;
      }
 cast_tmp;
```

```
      cast_tmp.typed = local[__j*(sizeof(ap_uint<128>)/sizeof(T))+__k];
      __tmpBuffer.range((__k+1)*sizeof(T)*8-1,__k*sizeof(T)*8) = cast_tmp.raw;
    }
    const int rem = len-(__j*(sizeof(ap_uint<128>)/sizeof(T)));
    const unsigned int bit_f = 0;
    const unsigned int bit_l = rem >= (sizeof(ap_uint<128>)/sizeof(T)) ?
(sizeof(ap_uint<128>)*8-1) : (rem*sizeof(T)*8-1);
   mcxx_wrapper_data[addr/sizeof(ap_uint<128>) + __j].range(bit_l, bit_f) =
__tmpBuffer.range(sizeof(ap_uint<128>)*8-1, bit_f);
  }
}

#undef __HLS_AUTOMATIC_MCXX__
```

solve_nbody_task_ompss.cpp

```
 ///////////////////
// Automatic IP Generated by OmpSs@FPGA compiler
///////////////////
// The below code is composed by:
//  1) User source code, which may be under any license (see in original source
code)
//  2) OmpSs@FPGA toolchain code which is licensed under LGPLv3 terms and
conditions
///////////////////
// Top IP Function: solve_nbody_task_wrapper
// Accel. type hash: 7705865291
// Num. instances: 1
// Wrapper version: 13
///////////////////
#define __HLS_AUTOMATIC_MCXX__ 1

#include <systemc.h>
#include <cstring>
#include <hls_stream.h>
#include <ap_int.h>

#include "src/kernel.fpga.h"
unsigned long long int __mcxx_raw_params[5];
template <typename T>struct mcxx_ptr_t;
template <typename T>struct mcxx_ref_t;
static unsigned long long int __mcxx_taskId;
static unsigned long long int __mcxx_parent_taskId;
extern hls::stream<ap_uint<64> > mcxx_inPort;
extern ap_uint<68> mcxx_outPort;
extern hls::stream<ap_uint<8> > mcxx_spawnInPort;
extern ap_uint<128> * mcxx_wrapper_data;
template<typename T> void __mcxx_memcpy_port_in(T * dst, const unsigned long long
int src, const size_t len);
template<typename T> void __mcxx_memcpy_port_out(const unsigned long long int dst,
const T * src, const size_t len);
void __mcxx_send_finished_task_cmd(const unsigned char destId);
template <typename T>struct mcxx_ref_t{
  unsigned long long int offset;
  ap_uint<128> buffer;
  mcxx_ref_t(const unsigned long long int offset)  {
#pragma HLS INTERFACE m_axi port=mcxx_wrapper_data
    this->buffer = *(mcxx_wrapper_data + offset/sizeof(ap_uint<128>));
    this->offset = offset;
  }
  operator T() const  {
    union {
 unsigned long long int raw;
 const T typed;
 }
 cast_tmp;
    const size_t off = this->offset%sizeof(ap_uint<128>);
    cast_tmp.raw = this->buffer.range((off+sizeof(const T))*8-1,off*8);
    return cast_tmp.typed;
  }
  mcxx_ref_t<T>& operator=(const T value)  {
#pragma HLS INTERFACE m_axi port=mcxx_wrapper_data
    union {
 unsigned long long int raw;
 T typed;
 }
 cast_tmp;
    cast_tmp.typed = value;
    const size_t off = this->offset%sizeof(ap_uint<128>);
    this->buffer.range((off+sizeof(T))*8-1,off*8) = cast_tmp.raw;
    *(mcxx_wrapper_data + this->offset/sizeof(ap_uint<128>)) = this->buffer;
    return *this;
  }
  mcxx_ptr_t<T> operator&()  {
    return mcxx_ptr_t<T>(this->offset);
  }
}
;
template <typename T>struct mcxx_ptr_t{
  unsigned long long int val;
  mcxx_ptr_t() : val(0) {
}
  mcxx_ptr_t(unsigned long long int val) {
 this->val = val;
 }
  mcxx_ptr_t(T* ptr) {
 this->val = (unsigned long long int)ptr;
 }
  template <typename V>  mcxx_ptr_t(mcxx_ptr_t<V> const &ref) {
 this->val = ref.val;
 }
  operator T*() const {
 return (T *)this->val;
 }
  operator unsigned long long int() const {
 return this->val;
 }
  operator mcxx_ptr_t<const T>() const  {
    mcxx_ptr_t<const T> ret;
    ret.val = this->val;
    return ret;
  }
  template <typename V>  mcxx_ptr_t<T> operator + (V const val) const  {
    mcxx_ptr_t<T> ret;
    ret.val = this->val + val*sizeof(T);
    return ret;
  }
  template <typename V>  mcxx_ptr_t<T> operator - (V const val) const  {
    mcxx_ptr_t<T> ret;
    ret.val = this->val - val*sizeof(T);
    return ret;
  }
  mcxx_ref_t<T> operator[](size_t idx)  {
    return mcxx_ref_t<T>(this->val + idx);
  }
  mcxx_ref_t<T> operator*()  {
    return mcxx_ref_t<T>(this->val);
  }
  operator ap_uint<128> *() const  {
    return (ap_uint<128> *)(mcxx_wrapper_data + this->val/sizeof(ap_uint<128>));
  }
}
;

static const unsigned int NCALCFORCES = 8;
const unsigned int FPGA_PWIDTH = 128;
static const unsigned int FORCE_FPGABLOCK_SIZE = 3 * 2048;
static const unsigned int PARTICLES_FPGABLOCK_SIZE = 8 * 2048;
static const unsigned int FORCE_FPGABLOCK_X_OFFSET = 0 * 2048;
static const unsigned int FORCE_FPGABLOCK_Y_OFFSET = 1 * 2048;
static const unsigned int FORCE_FPGABLOCK_Z_OFFSET = 2 * 2048;
static const unsigned int PARTICLES_FPGABLOCK_POS_X_OFFSET = 0 * 2048;
static const unsigned int PARTICLES_FPGABLOCK_POS_Y_OFFSET = 1 * 2048;
static const unsigned int PARTICLES_FPGABLOCK_POS_Z_OFFSET = 2 * 2048;
static const unsigned int PARTICLES_FPGABLOCK_MASS_OFFSET = 6 * 2048;
static const unsigned int PARTICLES_FPGABLOCK_WEIGHT_OFFSET = 7 * 2048;
enum mcc_enum_anon_20
{
  NANOS_ARGFLAG_DEP_OUT = 8,
  NANOS_ARGFLAG_DEP_IN = 4,
  NANOS_ARGFLAG_COPY_OUT = 2,
  NANOS_ARGFLAG_COPY_IN = 1,
  NANOS_ARGFLAG_NONE = 0
};
struct __attribute__((__packed__))  mcc_struct_anon_76
{
  unsigned long long int address;
  unsigned char flags;
  unsigned char arg_idx;
  unsigned int size;
};
typedef struct mcc_struct_anon_76 nanos_fpga_copyinfo_t;
extern void nanos_fpga_create_wd_async(const unsigned long long int type, const
unsigned char instanceNum, const unsigned char numArgs, const unsigned long long
int *args, const unsigned char numDeps, const unsigned long long int *deps, const
unsigned char *depsFlags, const unsigned char numCopies, const
nanos_fpga_copyinfo_t *copies);
static void calculate_forces(const int n_blocks, mcxx_ptr_t< float > forces,
mcxx_ptr_t< const float > particles)
{
  int j;
  int i;
#pragma HLS inline
  for (j = 0; j < n_blocks; j++)
    {
      for (i = 0; i < n_blocks; i++)
        {
          mcxx_ptr_t< float > forcesTarget = forces + i * FORCE_FPGABLOCK_SIZE;
          mcxx_ptr_t< const float > block1 = particles + i *
PARTICLES_FPGABLOCK_SIZE;
          mcxx_ptr_t< const float > block2 = particles + j *
PARTICLES_FPGABLOCK_SIZE;
          {
            mcxx_ptr_t< float > mcc_arg_0 = forcesTarget +
FORCE_FPGABLOCK_X_OFFSET;
            mcxx_ptr_t< float > mcc_arg_1 = forcesTarget +
FORCE_FPGABLOCK_Y_OFFSET;
            mcxx_ptr_t< float > mcc_arg_2 = forcesTarget +
FORCE_FPGABLOCK_Z_OFFSET;
            mcxx_ptr_t< const float > mcc_arg_3 = block1 +
PARTICLES_FPGABLOCK_POS_X_OFFSET;
            mcxx_ptr_t< const float > mcc_arg_4 = block1 +
PARTICLES_FPGABLOCK_POS_Y_OFFSET;
            mcxx_ptr_t< const float > mcc_arg_5 = block1 +
PARTICLES_FPGABLOCK_POS_Z_OFFSET;
            mcxx_ptr_t< const float > mcc_arg_6 = block1 +
PARTICLES_FPGABLOCK_MASS_OFFSET;
            mcxx_ptr_t< const float > mcc_arg_7 = block2 +
PARTICLES_FPGABLOCK_POS_X_OFFSET;
            mcxx_ptr_t< const float > mcc_arg_8 = block2 +
PARTICLES_FPGABLOCK_POS_Y_OFFSET;
            mcxx_ptr_t< const float > mcc_arg_9 = block2 +
PARTICLES_FPGABLOCK_POS_Z_OFFSET;
            mcxx_ptr_t< const float > mcc_arg_10 = block2 +
PARTICLES_FPGABLOCK_WEIGHT_OFFSET;
            unsigned long long int mcxx_args[11L] = {[0] = (unsigned long long
int)mcc_arg_0, [1] = (unsigned long long int)mcc_arg_1, [2] = (unsigned long long
int)mcc_arg_2, [3] = (unsigned long long int)mcc_arg_3, [4] = (unsigned long long
int)mcc_arg_4, [5] = (unsigned long long int)mcc_arg_5, [6] = (unsigned long long
int)mcc_arg_6, [7] = (unsigned long long int)mcc_arg_7, [8] = (unsigned long long
int)mcc_arg_8, [9] = (unsigned long long int)mcc_arg_9, [10] = (unsigned long long
int)mcc_arg_10};
            unsigned long long int mcxx_deps[3L] = {[0] = (unsigned long long
int)mcc_arg_0 + 0L, [1] = (unsigned long long int)mcc_arg_3 + 4L * (((0) - (0L))),
[2] = (unsigned long long int)mcc_arg_8 + 4L * (((0) - (0L)))};
            unsigned char mcxx_deps_flags[3L] = {[0] = NANOS_ARGFLAG_DEP_IN |
NANOS_ARGFLAG_DEP_OUT, [1] = NANOS_ARGFLAG_DEP_IN, [2] = NANOS_ARGFLAG_DEP_IN};
            nanos_fpga_copyinfo_t mcxx_copies[11L] = {[0] = {.address = (unsigned
long long int)mcc_arg_0, .flags = NANOS_ARGFLAG_COPY_IN | NANOS_ARGFLAG_COPY_OUT,
.arg_idx = 0, .size = (unsigned int)2048L * (unsigned int)sizeof(float)}, [1] =
{.address = (unsigned long long int)mcc_arg_1, .flags = NANOS_ARGFLAG_COPY_IN |
NANOS_ARGFLAG_COPY_OUT, .arg_idx = 1, .size = (unsigned int)2048L * (unsigned
int)sizeof(float)}, [2] = {.address = (unsigned long long int)mcc_arg_2, .flags =
NANOS_ARGFLAG_COPY_IN | NANOS_ARGFLAG_COPY_OUT, .arg_idx = 2, .size = (unsigned
int)2048L * (unsigned int)sizeof(float)}, [3] = {.address = (unsigned long long
int)mcc_arg_3, .flags = NANOS_ARGFLAG_COPY_IN, .arg_idx = 3, .size = (unsigned
int)2048L * (unsigned int)sizeof(const float)}, [4] = {.address = (unsigned long
long int)mcc_arg_4, .flags = NANOS_ARGFLAG_COPY_IN, .arg_idx = 4, .size = (unsigned
int)2048L * (unsigned int)sizeof(const float)}, [5] = {.address = (unsigned long
long int)mcc_arg_5, .flags = NANOS_ARGFLAG_COPY_IN, .arg_idx = 5, .size = (unsigned
int)2048L * (unsigned int)sizeof(const float)}, [6] = {.address = (unsigned long
long int)mcc_arg_6, .flags = NANOS_ARGFLAG_COPY_IN, .arg_idx = 6, .size = (unsigned
int)2048L * (unsigned int)sizeof(const float)}, [7] = {.address = (unsigned long
long int)mcc_arg_7, .flags = NANOS_ARGFLAG_COPY_IN, .arg_idx = 7, .size = (unsigned
int)2048L * (unsigned int)sizeof(const float)}, [8] = {.address = (unsigned long
long int)mcc_arg_8, .flags = NANOS_ARGFLAG_COPY_IN, .arg_idx = 8, .size = (unsigned
int)2048L * (unsigned int)sizeof(const float)}, [9] = {.address = (unsigned long
long int)mcc_arg_9, .flags = NANOS_ARGFLAG_COPY_IN, .arg_idx = 9, .size = (unsigned
int)2048L * (unsigned int)sizeof(const float)}, [10] = {.address = (unsigned long
long int)mcc_arg_10, .flags = NANOS_ARGFLAG_COPY_IN, .arg_idx = 10, .size =
(unsigned int)2048L * (unsigned int)sizeof(const float)}};
            nanos_fpga_create_wd_async(6057883607LU, 255, 11, mcxx_args, 3,
mcxx_deps, mcxx_deps_flags, 11, mcxx_copies);
          }
        }
    }
}
static void update_particles(const int n_blocks, mcxx_ptr_t< float > particles,
mcxx_ptr_t< float > forces, const float time_interval)
```

```
{
  int i;
#pragma HLS inline
  for (i = 0; i < n_blocks; i++)
    {
      {
        union  fpga_cast_union_0_t
        {
          unsigned long long int raw;
          float typed;
        };
        union fpga_cast_union_0_t fpga_spawn_cast_mcc_arg_15;
        mcxx_ptr_t< float > mcc_arg_13 = particles + i * PARTICLES_FPGABLOCK_SIZE;
        mcxx_ptr_t< float > mcc_arg_14 = forces + i * FORCE_FPGABLOCK_SIZE;
        const float mcc_arg_15 = time_interval;
        fpga_spawn_cast_mcc_arg_15.typed = mcc_arg_15;
        unsigned long long int mcxx_args[3L] = {[0] = (unsigned long long
int)mcc_arg_13, [1] = (unsigned long long int)mcc_arg_14, [2] =
fpga_spawn_cast_mcc_arg_15.raw};
        unsigned long long int mcxx_deps[2L] = {[0] = (unsigned long long
int)mcc_arg_13 + 4L * (((PARTICLES_FPGABLOCK_POS_X_OFFSET) - (0L))), [1] =
(unsigned long long int)mcc_arg_13 + 4L * (((PARTICLES_FPGABLOCK_POS_Y_OFFSET) -
(0L)))};
        unsigned char mcxx_deps_flags[2L] = {[0] = NANOS_ARGFLAG_DEP_IN |
NANOS_ARGFLAG_DEP_OUT, [1] = NANOS_ARGFLAG_DEP_IN | NANOS_ARGFLAG_DEP_OUT};
        nanos_fpga_copyinfo_t mcxx_copies[2L] = {[0] = {.address = (unsigned long
long int)mcc_arg_13, .flags = NANOS_ARGFLAG_COPY_IN | NANOS_ARGFLAG_COPY_OUT,
.arg_idx = 0, .size = (unsigned int)16384L * (unsigned int)sizeof(float)}, [1] =
{.address = (unsigned long long int)mcc_arg_14, .flags = NANOS_ARGFLAG_COPY_IN |
NANOS_ARGFLAG_COPY_OUT, .arg_idx = 1, .size = (unsigned int)6144L * (unsigned
int)sizeof(float)}};
        nanos_fpga_create_wd_async(5319290900LU, 255, 3, mcxx_args, 2, mcxx_deps,
mcxx_deps_flags, 2, mcxx_copies);
      }
    }
}
static void solve_nbody(mcxx_ptr_t< float > particles, mcxx_ptr_t< float > forces,
const int n_blocks, const int timesteps, const float time_interval)
{
  int t;
#pragma HLS inline
  for (t = 0; t < timesteps; t++)
    {
      calculate_forces(n_blocks, forces, particles);
      update_particles(n_blocks, particles, forces, time_interval);
    }
}
static unsigned long long int nanos_fpga_current_wd(void);
typedef void *nanos_wd_t;
enum mcc_enum_anon_18
{
  NANOS_OK = 0,
  NANOS_UNKNOWN_ERR = 1,
  NANOS_UNIMPLEMENTED = 2,
  NANOS_ENOMEM = 3,
  NANOS_INVALID_PARAM = 4,
  NANOS_INVALID_REQUEST = 5
};
typedef enum mcc_enum_anon_18 nanos_err_t;
static nanos_err_t nanos_fpga_wg_wait_completion(unsigned long long int uwg,
unsigned char avoid_flush);
extern void nanos_handle_error(nanos_err_t err);
static void solve_nbody_task_moved(mcxx_ptr_t< float > particles, mcxx_ptr_t< float
> forces, const int n_blocks, const int timesteps, const float time_interval)
{
  solve_nbody(particles, forces, n_blocks, timesteps, time_interval);
  {
    nanos_err_t nanos_err;
    unsigned long long int nanos_wd_ = nanos_fpga_current_wd();
    nanos_err = nanos_fpga_wg_wait_completion(nanos_wd_, 0);
    if (nanos_err != NANOS_OK)
      {
        nanos_handle_error(nanos_err);
      }
  }
}
void solve_nbody_task_wrapper() {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_hs port=mcxx_inPort
#pragma HLS INTERFACE ap_hs port=mcxx_outPort
#pragma HLS INTERFACE ap_hs port=mcxx_spawnInPort
#pragma HLS INTERFACE m_axi port=mcxx_wrapper_data

  unsigned long long int __commandArgs, __bufferData;
  unsigned char __commandCode;
  static ap_uint<1> __reset = 0;
  #pragma HLS RESET variable=__reset
  mcxx_ptr_t< float > particles;
  mcxx_ptr_t< float > forces;
  int n_blocks;
  int timesteps;
  float time_interval;

  if (__reset == 0) {
    __reset = 1;
  }
  __bufferData = mcxx_inPort.read();
  __commandCode = __bufferData;
  __commandArgs = __bufferData>>8;
  if (__commandCode == 1) {
    unsigned char __i;
    ap_uint<8> __paramInfo[5];
    unsigned char __comp_needed, __destID;
    __mcxx_taskId = mcxx_inPort.read();
    __mcxx_parent_taskId = mcxx_inPort.read();
    __comp_needed = __commandArgs>>24;
    __destID = __commandArgs>>32;
    for (__i = 0;
    __i < 5;
    __i++) {
      __paramInfo[__i] = mcxx_inPort.read();
      __mcxx_raw_params[__i] = mcxx_inPort.read();
    }
    in_copies_region: {
    {
      particles = __mcxx_raw_params[0];
    }
    {
      forces = __mcxx_raw_params[1];
    }
    {
      union {
        int n_blocks;
        unsigned long long int n_blocks_task_arg;
      }
      mcc_arg_2;
      mcc_arg_2.n_blocks_task_arg = __mcxx_raw_params[2];
      n_blocks = mcc_arg_2.n_blocks;
    }
    {
```

```
      union {
        int timesteps;
        unsigned long long int timesteps_task_arg;
      }
      mcc_arg_3;
      mcc_arg_3.timesteps_task_arg = __mcxx_raw_params[3];
      timesteps = mcc_arg_3.timesteps;
    }
    {
      union {
        float time_interval;
        unsigned long long int time_interval_task_arg;
      }
      mcc_arg_4;
      mcc_arg_4.time_interval_task_arg = __mcxx_raw_params[4];
      time_interval = mcc_arg_4.time_interval;
    }
    }
    if (__comp_needed) {
      solve_nbody_task_moved(particles, forces, n_blocks, timesteps,
time_interval);
    }
    send_finished_task_cmd: {
      #pragma HLS PROTOCOL fixed
      __mcxx_send_finished_task_cmd(__destID);
    }
  }
}
void __mcxx_write_out_port(const unsigned long long int data, const unsigned short
dest, const unsigned char last){
#pragma HLS INTERFACE ap_hs port=mcxx_outPort register
  ap_uint<68> tmp = data;
  tmp = (tmp << 4) | ((dest & 0x7) << 1) | (last & 0x1);
  mcxx_outPort = tmp;
}

void __mcxx_send_finished_task_cmd(const unsigned char destId){
  unsigned long long int header = 0x03;
  __mcxx_write_out_port(header, destId, 0);
  __mcxx_write_out_port(__mcxx_taskId, destId, 0);
  __mcxx_write_out_port(__mcxx_parent_taskId, destId, 1);
}

void nanos_handle_error(nanos_err_t err){
}
unsigned long long int nanos_fpga_current_wd(){
 return __mcxx_taskId;
}

nanos_err_t nanos_fpga_wg_wait_completion(unsigned long long int uwg, unsigned char
avoid_flush){
  unsigned long long int tmp;
  __mcxx_write_out_port(__mcxx_taskId /*data*/, /*taskwait*/3, 1 /*last*/);
    {

#pragma HLS PROTOCOL fixed
      wait();
      tmp = mcxx_spawnInPort.read();
      wait();
    }

  return NANOS_OK;
}

void nanos_fpga_create_wd_async(const unsigned long long int type, const unsigned
char instanceNum, const unsigned char numArgs, const unsigned long long int * args,
const unsigned char numDeps, const unsigned long long int * deps, const unsigned
char * depsFlags, const unsigned char numCopies, const nanos_fpga_copyinfo_t *
copies){
#pragma HLS inline
  const unsigned short destId = 2;
  unsigned long long int tmp = 0;
  tmp = (tmp << 8) | numCopies;
  tmp = (tmp << 8) | numDeps;
  tmp = (tmp << 8) | numArgs;
  tmp = tmp << 8;
  __mcxx_write_out_port(tmp, destId, 0);
  __mcxx_write_out_port(__mcxx_taskId, destId, 0);
  tmp = instanceNum;
  tmp = (tmp << 40) | type;
  __mcxx_write_out_port(tmp, destId, 0);
  for (unsigned char idx = 0;
 idx < numDeps;
 ++idx) {
    tmp = depsFlags[idx];
    tmp = (tmp << 56) | deps[idx];
    __mcxx_write_out_port(tmp, destId, (idx == (numDeps - 1))&&(numArgs ==
0)&&(numCopies == 0));
  }
  for (unsigned char idx = 0;
 idx < numCopies;
 ++idx) {
    tmp = copies[idx].address;
    __mcxx_write_out_port(tmp, destId, 0);
    tmp = copies[idx].size;
    tmp = (tmp << 24) | copies[idx].arg_idx;
    tmp = (tmp << 8) | copies[idx].flags;
    __mcxx_write_out_port(tmp, destId, (idx == (numCopies - 1))&&(numArgs == 0));
  }
  for (unsigned char idx = 0;
 idx < numArgs;
 ++idx) {
    __mcxx_write_out_port(args[idx], destId, (idx == (numArgs - 1)));
  }
}

template<typename T> void __mcxx_memcpy_port_in(T * local, const unsigned long long
int addr, const size_t len){
#pragma HLS INTERFACE m_axi port=mcxx_wrapper_data
#pragma HLS inline
  for (unsigned int __j=0;
   __j<(len == 0 ? 0 : (((len)*sizeof(T) - 1)/sizeof(ap_uint<128>)+1));
   __j++) {
    ap_uint<128> __tmpBuffer;
    __tmpBuffer = *(mcxx_wrapper_data + addr/sizeof(ap_uint<128>) + __j);
    #pragma HLS PIPELINE
    #pragma HLS UNROLL region
    for (unsigned int __k=0;
     __k<((sizeof(ap_uint<128>)/sizeof(T)));
     __k++) {
    if (((__j*(sizeof(ap_uint<128>)/sizeof(T))+__k) >= (len))) continue;
      union {
        unsigned long long int raw;
        T typed;
      }
 cast_tmp;
      cast_tmp.raw = __tmpBuffer.range((__k+1)*sizeof(T)*8-1,__k*sizeof(T)*8);
      #pragma HLS DEPENDENCE variable=local inter false
      local[__j*(sizeof(ap_uint<128>)/sizeof(T))+__k] = cast_tmp.typed;
    }
  }
}

template<typename T> void __mcxx_memcpy_port_out(const unsigned long long int addr,
const T * local, const size_t len){
#pragma HLS INTERFACE m_axi port=mcxx_wrapper_data
#pragma HLS inline
  for (unsigned int __j=0;
   __j<(len == 0 ? 0 : (((len)*sizeof(T)-1)/sizeof(ap_uint<128>)+1));
   __j++) {
    ap_uint<128> __tmpBuffer;
    #pragma HLS PIPELINE
    #pragma HLS UNROLL region
    for (unsigned int __k=0;
     __k<((sizeof(ap_uint<128>)/sizeof(T)));
     __k++) {
    if (((__j*(sizeof(ap_uint<128>)/sizeof(T))+__k) >= (len))) continue;
      union {
        unsigned long long int raw;
        T typed;
      }
 cast_tmp;
      cast_tmp.typed = local[__j*(sizeof(ap_uint<128>)/sizeof(T))+__k];
      __tmpBuffer.range((__k+1)*sizeof(T)*8-1,__k*sizeof(T)*8) = cast_tmp.raw;
    }
    const int rem = len-(__j*(sizeof(ap_uint<128>)/sizeof(T)));
    const unsigned int bit_f = 0;
    const unsigned int bit_l = rem >= (sizeof(ap_uint<128>)/sizeof(T)) ?
(sizeof(ap_uint<128>)*8-1) : (rem*sizeof(T)*8-1);
   mcxx_wrapper_data[addr/sizeof(ap_uint<128>) + __j].range(bit_l, bit_f) =
__tmpBuffer.range(sizeof(ap_uint<128>)*8-1, bit_f);
  }
}


#undef __HLS_AUTOMATIC_MCXX__
```

update_particles_BLOCK_ompss.cpp

```
///////////////////
// Automatic IP Generated by OmpSs@FPGA compiler
///////////////////
// The below code is composed by:
//  1) User source code, which may be under any license (see in original source
code)
//  2) OmpSs@FPGA toolchain code which is licensed under LGPLv3 terms and
conditions
///////////////////
// Top IP Function: update_particles_BLOCK_wrapper
// Accel. type hash: 5319290900
// Num. instances: 1
// Wrapper version: 13
///////////////////
#define __HLS_AUTOMATIC_MCXX__ 1

#include <cstring>
#include <hls_stream.h>
#include <ap_int.h>

#include "src/kernel.fpga.h"
unsigned long long int __mcxx_raw_params[3];
static unsigned long long int __mcxx_taskId;
static unsigned long long int __mcxx_parent_taskId;
extern hls::stream<ap_uint<64> > mcxx_inPort;
extern ap_uint<68> mcxx_outPort;
extern ap_uint<128> * mcxx_wrapper_data;
template<typename T> void __mcxx_memcpy_port_in(T * dst, const unsigned long long
int src, const size_t len);
template<typename T> void __mcxx_memcpy_port_out(const unsigned long long int dst,
const T * src, const size_t len);
void __mcxx_send_finished_task_cmd(const unsigned char destId);

static const unsigned int NCALCFORCES = 8;
const unsigned int FPGA_PWIDTH = 128;
static const unsigned int BLOCK_SIZE = 2048;
static const unsigned int PARTICLES_FPGABLOCK_MASS_OFFSET = 6 * 2048;
static const unsigned int PARTICLES_FPGABLOCK_VEL_X_OFFSET = 3 * 2048;
static const unsigned int PARTICLES_FPGABLOCK_VEL_Y_OFFSET = 4 * 2048;
static const unsigned int PARTICLES_FPGABLOCK_VEL_Z_OFFSET = 5 * 2048;
static const unsigned int PARTICLES_FPGABLOCK_POS_X_OFFSET = 0 * 2048;
static const unsigned int PARTICLES_FPGABLOCK_POS_Y_OFFSET = 1 * 2048;
static const unsigned int PARTICLES_FPGABLOCK_POS_Z_OFFSET = 2 * 2048;
static const unsigned int FORCE_FPGABLOCK_X_OFFSET = 0 * 2048;
static const unsigned int FORCE_FPGABLOCK_Y_OFFSET = 1 * 2048;
static const unsigned int FORCE_FPGABLOCK_Z_OFFSET = 2 * 2048;
static void update_particles_BLOCK_moved(float *particles, float *forces, const
float time_interval)
{
  int e;
#pragma HLS inline
#pragma HLS array_partition variable=forces cyclic factor=FPGA_PWIDTH/64
#pragma HLS array_partition variable=particles cyclic factor=FPGA_PWIDTH/64
  for (e = 0; e < BLOCK_SIZE; e++)
    {
#pragma HLS pipeline II=7
#pragma HLS dependence variable=particles inter false
#pragma HLS dependence variable=forces inter false
      const float mass = particles[PARTICLES_FPGABLOCK_MASS_OFFSET + e];
      const float velocity_x = particles[PARTICLES_FPGABLOCK_VEL_X_OFFSET + e];
      const float velocity_y = particles[PARTICLES_FPGABLOCK_VEL_Y_OFFSET + e];
      const float velocity_z = particles[PARTICLES_FPGABLOCK_VEL_Z_OFFSET + e];
      const float position_x = particles[PARTICLES_FPGABLOCK_POS_X_OFFSET + e];
      const float position_y = particles[PARTICLES_FPGABLOCK_POS_Y_OFFSET + e];
      const float position_z = particles[PARTICLES_FPGABLOCK_POS_Z_OFFSET + e];
      const float time_by_mass = time_interval / mass;
      const float half_time_interval = 5.000000000000000000000000e-01f *
time_interval;
      const float velocity_change_x = forces[FORCE_FPGABLOCK_X_OFFSET + e] *
time_by_mass;
      const float velocity_change_y = forces[FORCE_FPGABLOCK_Y_OFFSET + e] *
time_by_mass;
      const float velocity_change_z = forces[FORCE_FPGABLOCK_Z_OFFSET + e] *
time_by_mass;
      const float position_change_x = velocity_x + velocity_change_x *
half_time_interval;
      const float position_change_y = velocity_y + velocity_change_y *
half_time_interval;
      const float position_change_z = velocity_z + velocity_change_z *
half_time_interval;
      particles[PARTICLES_FPGABLOCK_VEL_X_OFFSET + e] = velocity_x +
velocity_change_x;
      particles[PARTICLES_FPGABLOCK_VEL_Y_OFFSET + e] = velocity_y +
velocity_change_y;
      particles[PARTICLES_FPGABLOCK_VEL_Z_OFFSET + e] = velocity_z +
velocity_change_z;
      particles[PARTICLES_FPGABLOCK_POS_X_OFFSET + e] = position_x +
position_change_x;
      particles[PARTICLES_FPGABLOCK_POS_Y_OFFSET + e] = position_y +
position_change_y;
      particles[PARTICLES_FPGABLOCK_POS_Z_OFFSET + e] = position_z +
position_change_z;
      forces[FORCE_FPGABLOCK_X_OFFSET + e] = 0.000000000000000000000000e+00f;
      forces[FORCE_FPGABLOCK_Y_OFFSET + e] = 0.000000000000000000000000e+00f;
      forces[FORCE_FPGABLOCK_Z_OFFSET + e] = 0.000000000000000000000000e+00f;
    }
}
void update_particles_BLOCK_wrapper() {
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_hs port=mcxx_inPort
#pragma HLS INTERFACE ap_hs port=mcxx_outPort
#pragma HLS INTERFACE m_axi port=mcxx_wrapper_data

  unsigned long long int __commandArgs, __bufferData;
  unsigned char __commandCode;
  static ap_uint<1> __reset = 0;
  #pragma HLS RESET variable=__reset
  static float particles[16384L];
  static float forces[6144L];
  float time_interval;

  if (__reset == 0) {
    __reset = 1;
  }
  __bufferData = mcxx_inPort.read();
  __commandCode = __bufferData;
  __commandArgs = __bufferData>>8;
  if (__commandCode == 1) {
    unsigned char __i;
    ap_uint<8> __paramInfo[3];
    unsigned char __comp_needed, __destID;
    __mcxx_taskId = mcxx_inPort.read();
    __mcxx_parent_taskId = mcxx_inPort.read();
    __comp_needed = __commandArgs>>24;
    __destID = __commandArgs>>32;
    for (__i = 0;
    __i < 3;
    __i++) {
      __paramInfo[__i] = mcxx_inPort.read();
      __mcxx_raw_params[__i] = mcxx_inPort.read();
    }
    in_copies_region: {
      if (__paramInfo[0][4]) {
        __mcxx_memcpy_port_in<float>(particles,__mcxx_raw_params[0],(16384L));
      }
      if (__paramInfo[1][4]) {
        __mcxx_memcpy_port_in<float>(forces,__mcxx_raw_params[1],(6144L));
      }
    {
      union {
        float time_interval;
        unsigned long long int time_interval_task_arg;
      }
      mcc_arg_2;
      mcc_arg_2.time_interval_task_arg = __mcxx_raw_params[2];
      time_interval = mcc_arg_2.time_interval;
    }
    }
    if (__comp_needed) {
      update_particles_BLOCK_moved(particles, forces, time_interval);
    }
    out_copies_region: {
      if (__paramInfo[0][5]) {
        __mcxx_memcpy_port_out<float>(__mcxx_raw_params[0],particles,(16384L));
      }
      if (__paramInfo[1][5]) {
        __mcxx_memcpy_port_out<float>(__mcxx_raw_params[1],forces,(6144L));
      }
    }
    send_finished_task_cmd: {
      #pragma HLS PROTOCOL fixed
      __mcxx_send_finished_task_cmd(__destID);
    }
  }
}
void __mcxx_write_out_port(const unsigned long long int data, const unsigned short
dest, const unsigned char last){
#pragma HLS INTERFACE ap_hs port=mcxx_outPort register
  ap_uint<68> tmp = data;
  tmp = (tmp << 4) | ((dest & 0x7) << 1) | (last & 0x1);
  mcxx_outPort = tmp;
}

void __mcxx_send_finished_task_cmd(const unsigned char destId){
  unsigned long long int header = 0x03;
  __mcxx_write_out_port(header, destId, 0);
  __mcxx_write_out_port(__mcxx_taskId, destId, 0);
  __mcxx_write_out_port(__mcxx_parent_taskId, destId, 1);
}

template<typename T> void __mcxx_memcpy_port_in(T * local, const unsigned long long
int addr, const size_t len){
#pragma HLS INTERFACE m_axi port=mcxx_wrapper_data
#pragma HLS inline
  for (unsigned int __j=0;
   __j<(len == 0 ? 0 : (((len)*sizeof(T) - 1)/sizeof(ap_uint<128>)+1));
   __j++) {
    ap_uint<128> __tmpBuffer;
    __tmpBuffer = *(mcxx_wrapper_data + addr/sizeof(ap_uint<128>) + __j);
    #pragma HLS PIPELINE
    #pragma HLS UNROLL region
    for (unsigned int __k=0;
     __k<((sizeof(ap_uint<128>)/sizeof(T)));
     __k++) {
    if (((__j*(sizeof(ap_uint<128>)/sizeof(T))+__k) >= (len))) continue;
      union {
        unsigned long long int raw;
        T typed;
      }
 cast_tmp;
      cast_tmp.raw = __tmpBuffer.range((__k+1)*sizeof(T)*8-1,__k*sizeof(T)*8);
      #pragma HLS DEPENDENCE variable=local inter false
      local[__j*(sizeof(ap_uint<128>)/sizeof(T))+__k] = cast_tmp.typed;
    }
  }
}

template<typename T> void __mcxx_memcpy_port_out(const unsigned long long int addr,
const T * local, const size_t len){
#pragma HLS INTERFACE m_axi port=mcxx_wrapper_data
#pragma HLS inline
  for (unsigned int __j=0;
   __j<(len == 0 ? 0 : (((len)*sizeof(T)-1)/sizeof(ap_uint<128>)+1));
   __j++) {
    ap_uint<128> __tmpBuffer;
    #pragma HLS PIPELINE
    #pragma HLS UNROLL region
    for (unsigned int __k=0;
     __k<((sizeof(ap_uint<128>)/sizeof(T)));
     __k++) {
    if (((__j*(sizeof(ap_uint<128>)/sizeof(T))+__k) >= (len))) continue;
      union {
        unsigned long long int raw;
        T typed;
      }
 cast_tmp;
      cast_tmp.typed = local[__j*(sizeof(ap_uint<128>)/sizeof(T))+__k];
      __tmpBuffer.range((__k+1)*sizeof(T)*8-1,__k*sizeof(T)*8) = cast_tmp.raw;
    }
    const int rem = len-(__j*(sizeof(ap_uint<128>)/sizeof(T)));
    const unsigned int bit_f = 0;
    const unsigned int bit_l = rem >= (sizeof(ap_uint<128>)/sizeof(T)) ?
(sizeof(ap_uint<128>)*8-1) : (rem*sizeof(T)*8-1);
   mcxx_wrapper_data[addr/sizeof(ap_uint<128>) + __j].range(bit_l, bit_f) =
__tmpBuffer.range(sizeof(ap_uint<128>)*8-1, bit_f);
  }
}


#undef __HLS_AUTOMATIC_MCXX__
```