OmpSs-2@FPGA User Guide

Release git

BSC Programming Models

29/05/26

CONTENTS

1 Install OmpSs-2@FPGA toolchain **3**
- 1.1 Prerequisites . . . 3
  - 1.1.1 Git Large File Storage . . . 3
  - 1.1.2 Vendor backends - Xilinx Vivado . . . 3
  - 1.1.3 QDMA PCIe linux drivers . . . 3
  - 1.1.4 Zynq/ZynqMP linux drivers . . . 4
- 1.2 Stable release . . . 4
  - 1.2.1 Native build . . . 4
  - 1.2.2 Cross-compiling toolchain . . . 4
- 1.3 Individual git repositories . . . 5
  - 1.3.1 Accelerator Integration Tool (AIT) . . . 5
  - 1.3.2 Kernel module . . . 5
  - 1.3.3 XDMA . . . 6
  - 1.3.4 xTasks . . . 6
  - 1.3.5 ovni . . . 6
  - 1.3.6 Nanos6-fpga . . . 6
  - 1.3.7 LLVM/Clang . . . 7
- 1.4 Alveo hardware setup . . . 7
- 1.5 Zynq and ZynqMP hardware setup . . . 9
  - 1.5.1 Prepare the SD card . . . 9

2 Develop OmpSs-2@FPGA programs **11**
- 2.1 Limitations . . . 11
- 2.2 Specific differences in clauses and directives in OmpSs-2@FPGA VS OmpSs-2 . . . 11
- 2.3 FPGA specific clauses . . . 12
  - 2.3.1 num_instances . . . 12
  - 2.3.2 affinity . . . 12
  - 2.3.3 copy_in/out . . . 12
  - 2.3.4 copy_deps . . . 13
- 2.4 Implicit Message Passing (IMP) clauses . . . 13
  - 2.4.1 data_dist . . . 13
  - 2.4.2 owner . . . 14
  - 2.4.3 Task ownership in dependencies . . . 14
- 2.5 Calls to Nanos6 API . . . 15
  - 2.5.1 Nanos6 FPGA Architecture API . . . 15

3 Compile OmpSs-2@FPGA programs **19**
- 3.1 LLVM/Clang FPGA Phase options . . . 19
  - 3.1.1 fompss-fpga-wrapper-code . . . 19
  - 3.1.2 fompss-fpga-ait-flags . . . 19

3.1.3 fompss-fpga-memory-port-width . . . 20
3.1.4 fompss-fpga-check-limits-memory-port . . . 20
3.1.5 fompss-fpga-instrumentation . . . 20
3.2 AIT options . . . 20
3.2.1 AIT options . . . 21
3.2.2 AIT user configuration file . . . 24
3.2.3 Design optimization parameters . . . 27
3.2.4 Accelerator placement options . . . 28
3.3 Binaries . . . 30
3.4 Bitstream . . . 30
3.4.1 HW Instrumentation . . . 30
3.4.2 Shared memory port . . . 30
3.5 Boot Files . . . 31

4 Running OmpSs-2@FPGA Programs **33**
4.1 Nanos6 FPGA Architecture configuration . . . 33
4.2 Running single node applications . . . 34
4.2.1 Running on QDMA-based FPGAs . . . 34
4.2.2 Running on System-on-Chips . . . 35
4.3 Running OMPIF applications . . . 35
4.3.1 Load cluster configuration scripts . . . 35
4.3.2 Application execution . . . 35
4.4 POM AXI-Lite interface memory map . . . 37
4.4.1 How to enable the AXI-Lite interface . . . 37
4.4.2 How to read the registers with QDMA . . . 37
4.5 Ovni FPGA instrumentation . . . 38
4.5.1 Prerequisites . . . 38
4.5.2 Running the application . . . 38
4.5.3 Processing traces . . . 39

5 Generate boot files for Xilinx SoC boards **41**
5.1 Prerequisites . . . 41
5.1.1 PetaLinux installation . . . 41
5.2 PetaLinux project setup . . . 41
5.2.1 Unpack the bsp . . . 42
5.2.2 Configure PetaLinux . . . 42
5.2.3 Configure linux kernel . . . 42
5.3 Generate boot files manually . . . 43
5.3.1 Add OmpSs@FPGA node to the device tree . . . 43
5.3.2 Build the Linux system . . . 43
5.3.3 Create BOOT.BIN file . . . 43
5.4 Use AIT to generate boot files . . . 43
5.5 Copy the files to the SD boot partition . . . 44

6 Server user guides **45**
6.1 quar user guide . . . 45
6.1.1 General remarks . . . 45
6.1.2 Node specifications . . . 45
6.1.3 Logging into the system . . . 46
6.1.4 Module structure . . . 46
6.1.5 Build applications . . . 46
6.1.6 Running applications . . . 46
6.2 crdbmaster user guide . . . 51
6.2.1 General remarks . . . 51

6.2.2 Node specifications 51
6.2.3 Logging into the system 52
6.2.4 Module structure 52
6.2.5 Build applications 52
6.2.6 Running applications 52
6.3 xaloc user guide 54
6.3.1 General remarks 55
6.3.2 Node specifications 55
6.3.3 Logging into the system 55
6.3.4 Module structure 55
6.3.5 Build applications 56
6.3.6 Running applications 56
6.4 llebeig user guide 57
6.4.1 General remarks 58
6.4.2 Logging into the system 58
6.4.3 Module structure 58
6.4.4 Build applications 58
6.5 MEEP-FPGA user guide 58
6.5.1 General remarks 59
6.5.2 Node specifications 59
6.5.3 Logging into the system 59
6.5.4 Module structure 59
6.5.5 Build applications 60
6.5.6 Running applications 60
6.6 bora user guide 64
6.6.1 General remarks 64
6.6.2 Logging into the system 64
6.6.3 Module structure 65
6.6.4 Build applications 65
6.7 ikergune user guide 65
6.7.1 General remarks 65
6.7.2 Logging into the system 65
6.7.3 Module structure 66
6.7.4 Build applications 66

Index **67**

The information included in this document is provided "as is", with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. The document is not guaranteed to be complete and/or error-free at this stage and it is subject to changes without furthernotice. Barcelona Supercomputing Center will not assume any responsibility for errors or omissions in this document. Please send comments, corrections and/or suggestions to ompss-fpga-support at bsc.es. This document is provided for informational purposes only.

Note: There is a PDF version of this document at http://pm.bsc.es/ftp/ompss-2-at-fpga/doc/user-guide-git/OmpSs2FPGAUserGuide.pdf

CHAPTER

ONE

INSTALL OMPSS-2@FPGA TOOLCHAIN

This page should help you install the OmpSs-2@FPGA toolchain. However, it is preferable using the pre-build Docker image with the latest stable toolchain. They are available at DockerHUB. Moreover, we distribute pre-built SD images for some SoC. Do not hesitate to contact us at ompss-fpga-support@bsc.es if you need help.

First, it describes the prerequisites to do the toolchain installation. After that, the following sections explain different approaches to do the installation.

1.1 Prerequisites

- Git Large File Storage (https://git-lfs.github.com/)
- Python 3.7 or later (https://www.python.org/)
- Vendor backends: - Xilinx Vivado 2021.1 or later (https://www.xilinx.com/products/design-tools/vivado.html)

1.1.1 Git Large File Storage

AIT repository uses Git Large File Storage to handle relatively-large files that are frequently updated (i.e. hardware runtime IP files) to avoid increasing the history size unnecessarily. You must install it so Git is able to download these files.

Follow instructions on their website to install it.

1.1.2 Vendor backends - Xilinx Vivado

Follow the installation instructions from Xilinx Vitis HLS and Vivado. You will need to enable support for the devices you're working on, as well as install the board files for the given devices.

1.1.3 QDMA PCIe linux drivers

QDMA drivers are needed for alveo PCIe-attached devices. These drivers can be downloaded from the AMD PCIe driver github The kernel module as well as the userspace utilities (`dma-ctl`) are required.

1.1.4 Zynq/ZynqMP linux drivers

See *Kernel module*

1.2 Stable release

There is a meta-repository that points to latest stable version of all tools: https://github.com/bsc-pm-ompss-at-fpga/ompss-2-at-fpga-releases. It contains a Makefile which, based on some environment variables, will compile and install the toolchain. The environment variables are:

- `TARGET` [Def: `native`] Linux architecture that toolchain will target
- `PREFIX` [Def: `/`] Installation prefix for the tools
- `PREFIX_HOST` [Def: `PREFIX`] Installation prefix for the host tools (i.e. llvm, ait)
- `PREFIX_TARGET` [Def: `PREFIX`] Installation prefix for the target tools (i.e. nanos6, libxdma, libxtasks, ovni)
- `PLATFORM` [Def: `qdma`] Board platform for execution. Supported platforms: zynq, qdma.
- `XTASKS_PLATFORM` [Def: `PLATFORM`] Board platform that xtasks backend will target. Supported backends: zynq, qdma.
- `XDMA_PLATFORM` [Def: `PLATFORM`] Board platform that xdma backend will target. Supported backends: zynq, qdma.
- `BUILDCPUS` [Def: `nproc`] Number of processes used for building

`make help` provides an exhaustive list of supported configuration variables and targets.

1.2.1 Native build

The following example will build the toolchain for the current native architecture and an Alveo support and install it in `/opt/bsc/ompss-2`.

```
git clone --recursive https://github.com/bsc-pm-ompss-at-fpga/ompss-2-at-fpga-releases.
↪git
cd ompss-2-at-fpga-releases
export PREFIX=/opt/bsc/ompss-2
export PLATFORM=qdma
make
```

1.2.2 Cross-compiling toolchain

The following example will cross-build the toolchain to generate binaries that can run in the *aarch64-linux-gnu* architecture and install it in `/opt/bsc/host-arm64/ompss-2` and `/opt/bsc/arm64/ompss-2`:

```
git clone --recursive https://github.com/bsc-pm-ompss-at-fpga/ompss-2-at-fpga-releases.
↪git
cd ompss-2-at-fpga-releases
export TARGET=aarch64-linux-gnu
export PREFIX_HOST=/opt/bsc/host-arm64/ompss-2
export PREFIX_TARGET=/opt/bsc/arm64/ompss-2
export PLATFORM=zynq
make
```

1.3 Individual git repositories

The master branches of all tools should generate a compatible toolchain. Each package should contain information about how to compile/install itself, look for the README files. The following points briefly describe each tool and provide a possible build configuration/setup for each one. We assume that all packages will be installed in a Linux OS in the `/opt/bsc/arm64/ompss-2` folder. Moreover, we assume that the packages will be cross-compiled from an Intel machine to be run on an ARM64 embedded board.

List of tools to install:

- AIT
- Kernel module
- xdma
- xtasks
- ovni
- Nanos6-fpga
- LLVM

1.3.1 Accelerator Integration Tool (AIT)

You can install the AIT package through the pip repository `python3 -m pip install ait-bsc` or cloning the git repository:

```
git clone https://github.com/bsc-pm-ompss-at-fpga/ait
cd ait
git lfs install
git lfs pull
export AIT_HOME="/path/to/install/ait"
export DEB_PYTHON_INSTALL_LAYOUT=deb_system
python3 -m pip install . -t $AIT_HOME

export PATH=PREFIX/ait/:$PATH
export PYTHONPATH=$AIT_HOME:$PYTHONPATH
```

1.3.2 Kernel module

The driver is only needed to execute the applications. To compile them, the library must be installed on the host but the kernel module may not be loaded. Example to cross-compile the driver:

```
git clone https://github.com/bsc-pm-ompss-at-fpga/ompss-at-fpga-kernel-module.git
cd ompss-at-fpga-kernel-module
export CROSS_COMPILE=aarch64-linux-gnu-
export KDIR=/home/my_user/kernel-headers
export ARCH=arm64
make
```

1.3.3 XDMA

Example to cross-compile the library and install it in the `/opt/bsc/arm64/ompss-2/libxdma` folder:

```
git clone https://github.com/bsc-pm-ompss-at-fpga/xdma.git
cd xdma/src/zynq
export CROSS_COMPILE=aarch64-linux-gnu-
export KERNEL_MODULE_DIR=/path/to/ompss-at-fpga/kernel/module/src
make
make PREFIX=/opt/bsc/arm64/ompss-2/libxdma install
```

1.3.4 xTasks

Example to cross-compile the library and install it in the `/opt/bsc/arm64/ompss-2/libxtasks` folder:

```
git clone https://github.com/bsc-pm-ompss-at-fpga/xtasks.git
cd xtasks/src/zynq
export CROSS_COMPILE=aarch64-linux-gnu-
export LIBXDMA_DIR=/opt/bsc/arm64/ompss-2/libxdma
make
make PREFIX=/opt/bsc/arm64/ompss-2/libxtasks install
```

1.3.5 ovni

Example to cross-compile the library and install it in the `/opt/bsc/arm64/ompss-2/libovni` folder:

```
git clone https://github.com/bsc-pm-ompss-at-fpga/ovni.git
mkdir ovni-build
cd ovni-build
cmake \
  -DCMAKE_INSTALL_PREFIX=$(PREFIX_TARGET)/libovni \
  -DUSE_MPI=OFF \
  -DCMAKE_C_COMPILER=aarch64-linux-gnu-gcc
../ovni
make
make install
```

1.3.6 Nanos6-fpga

Example to cross-compile the runtime library and install it in the `/opt/bsc/arm64/ompss-2/nanos6-fpga` folder:

```
git clone https://github.com/bsc-pm-ompss-at-fpga/nanos6-fpga.git
cd nanos6-fpga
./autogen.sh
mkdir build-fpga-arm64
cd build-fpga-arm64
../configure --prefix=/opt/bsc/arm64/ompss-2/nanos6-fpga --host=aarch64-linux-gnu \
  --enable-fpga \
  --with-xtasks=/opt/bsc/arm64/ompss-2/libxtasks \
  --with-ovni=/opt/bsc/arm64/ompss-2/ovni \
```

(continues on next page)

(continued from previous page)

```
  --disable-discrete-deps \
  --disable-all-instrumentations \
  --enable-stats-instrumentation \
  --enable-verbose-instrumentation \
  --enable-ovni-instrumentation
make
make install
```

1.3.7 LLVM/Clang

Example to build a LLVM/Clang cross-compiler that runs on the host and creates binaries for another platform (ARM64 in the example):

```
git clone https://github.com/bsc-pm-ompss-at-fpga/llvm.git
mkdir build-fpga
cd build-fpga
cmake -G Ninja \
  -DCMAKE_INSTALL_PREFIX=/opt/bsc/host-arm64/ompss-2/llvm \
  -DLLVM_TARGETS_TO_BUILD="AArch64" \
  -DCMAKE_BUILD_TYPE=Release \
  -DCLANG_DEFAULT_NANOS6_HOME=/opt/bsc/arm64/ompss-2/nanos6-fpga \
  -DLLVM_USE_SPLIT_DWARF=ON \
  -DLLVM_ENABLE_PROJECTS="clang" \
  -DLLVM_INSTALL_TOOLCHAIN_ONLY=ON \
  -DCMAKE_C_COMPILER=clang \
  -DCMAKE_CXX_COMPILER=clang++ \
  -DLLVM_USE_LINKER=lld \
../llvm/llvm
nina
ninja install
```

1.4 Alveo hardware setup

OmpSs-2@FPGA loads bitstreams using jtag. Therefore, a USB port needs to be connected from the host to the FPGA.

The following images show the location of the USB port on the PCIe back plate of Alveo cards.

Once connected, you should be able to see the USB-UART device using `lsusb`:

```
> lsusb -d 0403:6011
Bus 001 Device 003: ID 0403:6011 Future Technology Devices International, Ltd FT4232H␣
↪Quad HS USB-UART/FIFO IC
```

Then appropriately set `RW` permissions for the USB devices. The previous `lsusb` also shows the device bus and device, therefore, the device file for the previous `lsusb` output is:

```
/dev/bus/usb/001/003
```

Then vivado should be able to access JTAG targets. The following Vivado tcl script connects lists available hardware targets and jtag devices:

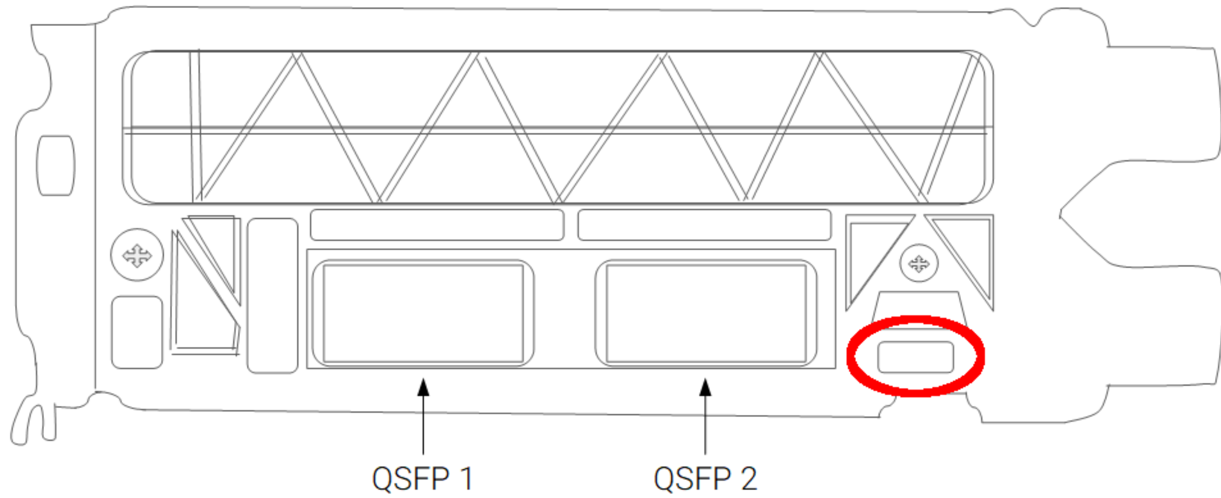

Fig. 1: Location of the USB port on an Alveo U200/U250

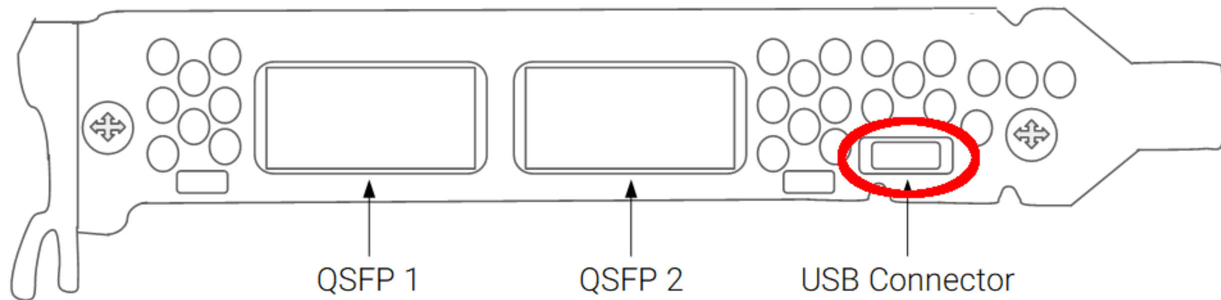

Fig. 2: Location of the USB port on an Alveo U55c

```
open_hw_manager
connect_hw_server
get_hw_targets
foreach target [get_hw_targets] {
    open_hw_target $target
    puts $target
    puts [get_hw_devices]
}
```

It can be run in batch mode, or pasted into the command shell in the gui (or in tcl mode).

```
vivado -nolog -nojournal -batch -source test_hw_server.tcl
```

It should print, among other output, something like this:

```
...
...
INFO: [Labtoolstcl 44-466] Opening hw_target localhost:13330/xilinx_tcf/Xilinx/
→21290594G00LA
localhost:13330/xilinx_tcf/Xilinx/21290594G00LA
xcu200_0
INFO: [Common 17-206] Exiting Vivado at Tue Feb 17 17:56:06 2026...
```

1.5 Zynq and ZynqMP hardware setup

Current OmpSs-2@FPGA toolchain supports both Zynq (zynq702, zynq706, zybo and zedboard) and ZynqMP (zcu102 and Kria kv260) boards.

1.5.1 Prepare the SD card

SD images based on Ubuntu cloud with the preinstalled toolchain are distributed. They can be downloaded from the OmpSs-2@FPGA SD download page

Once downloaded, dump the image into the SD card. A 4GB SD card or bigger is required.

```
gunzip < image.img.gz | dd of=/dev/mmcblk0 bs=16M status=progress
```

This uncompresses the downloaded image file using `gunzip` and sends it do `dd` to write it on the device. Two partitions are created: boot and rootfs.

Setting up the boot partition

The boot partition (`mmcblk0p1`) contains the files required to boot the board. These files consists on:

- First Stage Bootloader (FSBL)
- Linux kernel
- Device-tree
- Bitstream

Inside the OmpSs-2@FPGA SD image boot partition there will be a folder for each of the supported boards containing all the required boot files.

In order to boot a specific board, you must copy the files from inside the corresponding folder to the root of the boot partition. For example, to boot a ZCU102 board, you must run:

```
BOOT_DIR=$(udisksctl mount --block-device /dev/mmcblk0p1)
cp ${BOOT_DIR}/boot-zcu102/* ${BOOT_DIR}/
udisksctl unmount --block-device /dev/mmcblk0p1
```

Setting up the rootfs

The rootfs partition (`mmcblk0p2`) contains a Ubuntu 24.04 root filesystem.

There is a systemd service configured to initialize the board with an example bitstream which implements a dotproduct computation and to load the OmpSs-2@FPGA kernel module for the corresponding kernel version.

To set up Ubuntu for a specific board, you must create a soft-link to the bitstream and ko files that the systemd service has to load. Following the example of the ZCU102, you must do:

```
ROOTFS_DIR=$(udisksctl mount --block-device /dev/mmcblk0p2)
pushd ${ROOTFS_DIR}/opt/init_fpga/
ln -s bitstream-zcu102/bitstream.bin bitstream.bin
ln -s ompss-fpga-6.1.30-xilinx-v2023.2.ko ompss-fpga.ko
popd
udisksctl unmount --block-device /dev/mmcblk0p2
```

CHAPTER

TWO

DEVELOP OMPSS-2@FPGA PROGRAMS

Most of the required information to develop an OmpSs-2@FPGA application should be in the general OmpSs-2 documentation. Note that, there may be some unsupported/not-working OmpSs-2 features and/or syntax when using FPGA tasks. If you have some problem or encounter any bug, do not hesitate to contact us or open an issue.

To create an FPGA task you need to add the `device` clause in the `task` directive. For example:

```
const unsigned int LEN = 8;

#pragma oss task device(fpga) out([LEN]dst, val)
void memset(char * dst, const char val) {
  for (unsigned int i=0; i<LEN; ++i) {
    dst[i] = val;
  }
}
```

2.1 Limitations

There are some limitations when developing an OmpSs-2@FPGA application:

- Only C/C++ are supported, not Fortran.
- Only function declarations can be annotated as FPGA tasks.
- The HLS source code generated by Clang for each FPGA task will not contain the includes in the original source file but the ones finished in ".fpga.h".
- The FPGA task code cannot perform general system calls, and only some Nanos6 APIs are supported.
- The usage of `size_t`, `signed long int` or `unsigned long int` is not recommended inside the FPGA accelerator code. They may have different widths in the host and in the FPGA.

2.2 Specific differences in clauses and directives in OmpSs-2@FPGA VS OmpSs-2

Despite OmpSs-2@FPGA mostly follows the OmpSs-2 behaviour, there are specific clauses or directives that are not yet implemented or whose implementation slightly differs from the OmpSs-2 specification:

- `taskyield` and `atomic` directives are **not supported**.
- `critical` directive is supported as OmpSs-2 specifies. Specifically: it implements a **global** (all accelerators) mutual exclusion section.

2.3 FPGA specific clauses

The following clauses can be used in the `task` directive of FPGA tasks.

2.3.1 num_instances

Defines the number of instances to place in the FPGA bitstream of a task. Usage example:

```
const unsigned int LEN = 8;

#pragma oss task device(fpga) out([LEN]dst) num_instances(3)
void memset(char * dst, const char val) {
  for (unsigned int i=0; i<LEN; ++i) {
    dst[i] = val;
  }
}
```

2.3.2 affinity

The information in this clause is used at runtime to send the tasks to the corresponding FPGA accelerator. This means that a FPGA task has the `affinity(0)` it will run in accelerator 0 of that type. This clause is useful to manage task scheduling in the user code when there is more than one accelerator of the same type (num_instances > 1).

```
const unsigned int LEN = 8;

#pragma oss task device(fpga) out([LEN]dst) num_instances(4) affinity(af)
void memset_char(char * dst, const char val, int af) {
  for (unsigned int i=0; i<LEN; ++i) {
    dst[i] = val;
  }
}

#pragma oss task device(fpga) out([size]dst)
void memset_task_creator(float * dst, int size, const float val) {
  for (unsigned int i=0; i<size/LEN; ++i) {
    memset_char(dst + i*LEN, val, i%4);
  }
}
```

2.3.3 copy_in/out

Defines the memory regions that the FPGA task wrapper must catch in BRAMs/URAMs. This creates a local copy of the parameter in the FPGA task accelerator which can be accessed faster than dispatching memory accesses. The data is copied from the FPGA addressable memory into the FPGA task accelerator before launching the task execution. Depending on the type of clause (`copy_in`, `copy_out`, `copy_inout`), the wrapper includes support for reading/writing the local copy from/into memory. Both input and output data movements, may be dynamically disabled by the runtime based on its knowledge about task copies and predecessor/successor tasks. Usage example:

```
const unsigned int LEN = 8;

#pragma oss task device(fpga) out([LEN]dst) copy_out([LEN]dst)
void memset(char * dst, const char val) {
  for (unsigned int i=0; i<LEN; ++i) {
    dst[i] = val;
  }
}
```

2.3.4 copy_deps

Promote the task dependencies like they were annotated into the `copy` clause.

2.4 Implicit Message Passing (IMP) clauses

The following clauses are part of the Implicit Message Passing model. Furrently these are only supported in FPGA tasks.

2.4.1 data_dist

The `data_dist` clause allows to specify a data distribution among different nodes in the cluster. The syntax is defined as follows.

```
data_dist(distribution_type | arithmetic_expression, base_pointer, size_expression)
```

- `distribution_type`: A string specifying the distribution type to use, currently supported types are `all`, `block`, `cyclic`. Alternatively, an arithmetic expression can be used to specify a `block_cyclic` distribution.
- `arithmetic_expression`: Specifies the block size for a `block_cyclic` distribution, alternatively, a string can be used for other data distribution types.
- `base_pointer`: Base pointer of the array that will be distributed.
- `size_expression`: Size of the array to be distributed.

Data distribution will apply to fpga nested tasks. In the following example, `bar` tasks will assume that data is distributed following the `block` distribution. In a given node, tasks will run (or not) based on the ownership defined by the data distribution.

```
#pragma oss task device(fpga) inout([data]data) data_dist("block", data, size)
void foo(int * data, int size) {
    for (int i = 0; i<size; i+=BS) {
        bar(&data[i]);
    }
}
```

all

Data is replicated and owned by all nodes in the cluster. If if this data is modified, updates need to be propagated to all nodes in the cluster.

block

Data is distributed in even blocks among all nodes in the cluster. Each node is assigned `size_expression/NUM_NODES` consecutive elements of the array specified from `base_pointer`.

cyclic

Data elements are assigned round-robin to different nodes. Each *single* data element will be assigned to a different node, for instance, given an array of length 6, and a cluster with 3 nodes, data will be assigned to nodes 0, 1, 2, 0, 1, 2.

block-cyclic

If an arithmetic expression is used, block-cyclic distribution is used. This arithmetic expression defines the size of the blocks that will be distributed in a cyclic fashion.

2.4.2 owner

This clause specifies the ownership of the task. It defines which node in the cluster will execute this task. If no owner is specified, the task ownership is determined by the owner of the first `inout` dependency in the task. If no `inout` dependency is defined, task ownership is defined by `out` and then `in` data ownership.

2.4.3 Task ownership in dependencies

Data ownership for a particular data dependency can be specified by adding the data owner to the data dependency:

```
in([shape]array;owner)
```

Owner needs to be a task argument, for example:

```
#pragma oss task device(fpga) inout([size]data;dataOwner)
void foo(int *data, size, dataOwner) {
    ...
}

#pragma oss task device(fpga) inout(...)
void bar() {
    ...
    int owner = x % cluster_size;
    foo(data, size, owner);
    ...
}
```

2.5 Calls to Nanos6 API

The list of Nanos6 APIs and their details can be found in the following section. Note that not all Nanos6 APIs can be called within FPGA tasks and others only are supported within them.

2.5.1 Nanos6 FPGA Architecture API

The following sections list and summarize the Nanos6 FPGA architecture API.

Memory Management

nanos6_fpga_malloc

Allocates memory in the FPGA address space and returns a pointer valid for the FPGA tasks. The returned pointer cannot be dereferenced in the host code.

Arguments:

- **size**: Size in bytes to allocate.
- **fpga_addr**: Pointer to the FPGA address space as a 64-bit integer.

Return value:

- NANOS6_FPGA_SUCCESS on success, NANOS6_FPGA_ERROR on error.

```
typedef enum {
    NANOS6_FPGA_SUCCESS,
    NANOS6_FPGA_ERROR
} nanos6_fpga_stat_t;

nanos6_fpga_stat_t nanos6_fpga_malloc(uint64_t size, uint64_t* fpga_addr);
```

nanos6_fpga_free

```
nanos6_fpga_stat_t nanos6_fpga_free(uint64_t fpga_addr);
```

nanos6_fpga_memcpy

```
typedef enum {
    NANOS6_FPGA_DEV_TO_HOST,
    NANOS6_FPGA_HOST_TO_DEV
} nanos6_fpga_copy_t;

nanos6_fpga_stat_t nanos6_fpga_memcpy(
    void* usr_ptr,
    uint64_t fpga_addr,
    uint64_t size,
    nanos6_fpga_copy_t copy_type);
```

Data copies

These Nanos6 API can only be called inside an FPGA task. They allow copies to be performed through a single port that can be wider than the data type being copied.

If any of the data copy API calls are used, the *fompss-fpga-memory-port-width* option is mandatory.

Data accessed through this functions has to be **aligned to the port width**, otherwise this will result in undefined behaviour.

Also, data should to be **multiple of the port width**. If this cannot be guaranteed, *fompss-fpga-check-limits-memory-port* option is needed so that no out of bounds data is accessed, otherwise this will result in undefined behaviour.

nanos6_fpga_memcpy_wideport_in

```
nanos6_fpga_stat_t nanos6_fpga_memcpy_wideport_in(void* dst, const unsigned long long␣
↪int addr, const unsigned int num_elems);
```

Arguments:

- `dst`: Pointer to the destination (local) data. It can be any data type.
- `addr`: FPGA memory address space where the data is stored.
- `num_elems`: Number of elements of the array type to be copied.

nanos6_fpga_memcpy_wideport_out

```
nanos6_fpga_stat_t nanos6_fpga_memcpy_wideport_out(void* dst, const unsigned long long␣
↪int addr, const unsigned int num_elems);
```

Arguments:

- `dst`: Pointer to the source (local) data. It can be any data type.
- `addr`: FPGA memory address space where the data is written.
- `num_elems`: Number of elements of the array type to be copied.

OMPIF cluster API

OMPIF is an API that allows direct FPGA-to-FPGA communication.

OMPIF API resembles MPI API with few assumptions and simplifications.

- Data types are not used, raw data and its size in bytes is used instead.
- A single implicit communicator that includes all FPGAs in the cluster is assumed in collectives.
- For send/receive, dependencies can be added for task synchronization.
- The size of a message is limited by the 16-bit sequence number of the OMPIF header and the OMPIF packet size. Currently, it is set to 8960 bytes/packet: 65536*8960=587202560 (560MB).

API calls are defined as follows:

```
void OMPIF_Send(const void *data, unsigned int size, int destination, unsigned char tag,␣
↪unsigned char numDeps, const unsigned long long int deps[]);
```

Arguments:

- `data`: Pointer to the local data. It must be multiple of 64 bytes.
- `size`: Size in bytes. It must be less than 560MB.
- `destination`: Rank of the desination.
- `tag`: Message tag. It must be less than 256.
- `numDeps`: Size of the `deps` array.
- `deps`: Dependence array.

```
void OMPIF_Recv(void *data, unsigned int size, int source, unsigned char tag, unsigned␣
↪char numDeps, const unsigned long long int deps[]);
```

It has the same arguments as OMPIF_Send. `data` is a pointer where to put the received message, it also must be multiple of 64 bytes. `source` is the rank of the source.

```
void OMPIF_Allgather(void* data, unsigned int size);
```

Arguments:

- `data`: Pointer with the local data to send and where the received data is stored.
- `size`: Size in bytes to send and to receive from each rank. It must be less than 560MB.

```
void OMPIF_Bcast(void* data, unsigned int size, int root);
```

Arguments:

- `data`: For the root, pointer to the local data. For the rest, pointer where the data is stored. It must be multiple of 64 bytes.
- `size`: Size in bytes to send for the root, and to receive for the rest. Must be less than 560MB.
- `root`: Rank of the root.

```
unsigned char OMPIF_Comm_rank();
```

Return value: The rank of the calling FPGA.

```
unsigned char OMPIF_Comm_size();
```

Return value: The number of FPGAs in the cluster.

CHAPTER

THREE

COMPILE OMPSS-2@FPGA PROGRAMS

To compile an OmpSs-2@FPGA program you should follow the general OmpSs-2 compilation procedure using the LLVM/Clang compiler. More information is provided in the OmpSs-2 User Guide (https://pm.bsc.es/ftp/ompss-2/doc/user-guide/llvm/index.html). The following sections detail the specific options of LLVM/Clang to generate the binaries, bitstream and boot files.

The entire list of LLVM/Clang options (for the FPGA phase) and AIT arguments are available here:

3.1 LLVM/Clang FPGA Phase options

The following sections list and summarize the LLVM/Clang options from the FPGA Phase.

Note: Do not forget the flag `-fompss-2` in both compilation and linking stages of your application. Otherwise, your application will not be compiled with parallel support or not linked to the tasking runtime library.

3.1.1 fompss-fpga-wrapper-code

`[Available in release 2.0.0]`

Enables FPGA task extraction into independent HLS wrappers.

This option is mandatory when generating a bitstream.

```
clang -fompss-2 -fompss-fpga-wrapper-code \
  -fompss-fpga-ait-flags "--board=alveo_u200 --name=dotproduct" \
  src/dotproduct.c -o dotproduct
```

3.1.2 fompss-fpga-ait-flags

`[Available in release 2.0.0]`

String of whitespace-separated list of AIT flags that will be passed to the tool. Also enables AIT on the linking stage.

This option is mandatory when generating a bitstream.

```
clang -fompss-2 -fompss-fpga-wrapper-code \
  -fompss-fpga-ait-flags "--board=alveo_u200 --name=dotproduct" \
  src/dotproduct.c -o dotproduct
```

3.1.3 fompss-fpga-memory-port-width

`[Available in release 2.0.0]`

Enables wide-port feature of OmpSs@FPGA.

Accelerator memory interfaces will be merged into a single wide-port of arbitrary power-of-2 size. Code inside the wrapper will be generated in order to pack and unpack local variables when reading and writing from memory.

```
clang -fompss-2 -fompss-fpga-wrapper-code \
  -fompss-fpga-ait-flags "--board=alveo_u200 --name=dotproduct" \
  -fompss-fpga-memory-port-width 512 \
  src/dotproduct.c -o dotproduct
```

3.1.4 fompss-fpga-check-limits-memory-port

`[Available in release 2.0.0]`

By default the compiler assumes that all the data that has to be copied to and from memory is multiple of the wide-port size.

This option adds checks inside the pack/unpacking code to manage copies smaller than the wide-port size.

```
clang -fompss-2 -fompss-fpga-wrapper-code \
  -fompss-fpga-ait-flags "--board=alveo_u200 --name=dotproduct" \
  -fompss-fpga-check-limits-memory-port \
  src/dotproduct.c -o dotproduct
```

3.1.5 fompss-fpga-instrumentation

`[Available in release 3.1.0]`

Enables HW instrumentation.

This option will add nanos6 FPGA instrumentation API function needed to trace dependency copies and kernel execution.

```
clang -fompss-2 -fompss-fpga-wrapper-code \
  -fompss-fpga-ait-flags "--board=alveo_u200 --name=dotproduct" \
  -fompss-fpga-instrumentation \
  src/dotproduct.c -o dotproduct
```

3.2 AIT options

The following sections list and summarize the Accelerator Integration Tool (AIT) parameters and configurations.

For the complete `--help` output, see:

3.2.1 AIT options

The AIT behavior can be modified with the available options. They are summarized and briefly described in the AIT help, which is:

```
usage: ait -b BOARD -n NAME
The Accelerator Integration Tool (AIT) automatically integrates OmpSs@FPGA accelerators␣
↪into FPGA designs using different vendor backends.

Required:
  -b, --board BOARD     board model. Supported boards by vendor:
                        xilinx: alveo_u200, alveo_u250, alveo_u280, alveo_u280_hbm,␣
↪alveo_u55c, com_express, kv260, zcu102, zedboard, zybo, zynq702, zynq706
  -n, --name NAME       project name

Generation flow:
  -d, --dir DIR         path where the project directory tree will be created
                        (def: .)
  --disable_IP_caching  disable IP caching
                        Significantly increases generation time
  --disable_utilization_check
                        disable resources utilization check during HLS generation
  --disable_board_support_check
                        disable board support check
  --from_step FROM_STEP
                        initial generation step
                        Generation steps by vendor:
                        xilinx: HLS, design, synthesis, implementation, bitstream, boot
                        (def: 'HLS')
  --IP_cache_location IP_CACHE_LOCATION
                        path where the IP cache will be located
                        (def: /var/tmp/ait/<vendor>/IP_cache/)
  --to_step TO_STEP     final generation step
                        Generation steps by vendor:
                        xilinx: HLS, design, synthesis, implementation, bitstream, boot
                        (def: 'bitstream')

Bitstream configuration:
  -c, --clock CLOCK     FPGA clock frequency in MHz
                        (def: 100)
  --hwcounter           add a hardware counter to the bitstream
  --bitinfo_note BITINFO_NOTE
                        custom note to add to the bitinfo
  --disable_static_constraints
                        disable static constraints
                        May impact negatively on timing

Data path:
  --memory_interleaving_stride MEM_INTERLEAVING_STRIDE
                        size in bytes of the stride of the memory interleaving
                        Must be power of 2
                        If set to 0 bytes, interleaving will not be enabled
                        (def: 0)
```

(continues on next page)

(continued from previous page)

```
Hardware Runtime:
  --cmdin_queue_len CMDIN_QUEUE_LEN
                        maximum length (64-bit words) of the queue for the hwruntime␣
↪command in
                        This argument is mutually exclusive with --cmdin_subqueue_len
  --cmdin_subqueue_len CMDIN_SUBQUEUE_LEN
                        length (64-bit words) of each accelerator subqueue for the␣
↪hwruntime command in
                        This argument is mutually exclusive with --cmdin_queue_len
                        Must be power of 2
                        (def: max(64, 1024/num_instances))
  --cmdout_queue_len CMDOUT_QUEUE_LEN
                        maximum length (64-bit words) of the queue for the hwruntime␣
↪command out
                        This argument is mutually exclusive with --cmdout_subqueue_len
  --cmdout_subqueue_len CMDOUT_SUBQUEUE_LEN
                        length (64-bit words) of each accelerator subqueue for the␣
↪hwruntime command out
                        This argument is mutually exclusive with --cmdout_queue_len
                        Must be power of 2
                        (def: max(64, 1024/num_instances))
  --disable_spawn_queues
                        disable the hwruntime spawn in/out queues
  --spawnin_queue_len SPAWNIN_QUEUE_LEN
                        length (64-bit words) of the hwruntime spawn in queue
                        Must be power of 2
                        (def: 1024)
  --spawnout_queue_len SPAWNOUT_QUEUE_LEN
                        length (64-bit words) of the hwruntime spawn out queue
                        Must be power of 2
                        (def: 1024)
  --hwruntime_interconnect HWR_INTERCONNECT
                        type of hardware runtime interconnection with accelerators
                        centralized
                        distributed
                        (def: centralized)
  --max_args_per_task MAX_ARGS_PER_TASK
                        maximum number of arguments for any task in the bitstream
                        (def: 15)
  --max_deps_per_task MAX_DEPS_PER_TASK
                        maximum number of dependencies for any task in the bitstream
                        (def: 8)
  --max_copies_per_task MAX_COPIES_PER_TASK
                        maximum number of copies for any task in the bitstream
                        (def: 15)
  --enable_pom_axilite  enable the POM axilite interface with debug counters

Picos:
  --picos_num_dcts NUM_DCTS
                        number of DCTs to instantiate
                        (def: 1)
```

(continues on next page)

(continued from previous page)

```
  --picos_tm_size PICOS_TM_SIZE
                        size of the TM memory
                        (def: 128)
  --picos_dm_size PICOS_DM_SIZE
                        size of the DM memory
                        (def: 512)
  --picos_vm_size PICOS_VM_SIZE
                        size of the VM memory
                        (def: 512)
  --picos_dm_ds DATA_STRUCT
                        data structure of the DM memory
                        BINTREE: Binary search tree (not autobalanced)
                        LINKEDLIST: Linked list
                        (def: BINTREE)
  --picos_dm_hash HASH_FUN
                        hashing function applied to dependence addresses
                        P_PEARSON: Parallel Pearson function
                        XOR
                        (def: P_PEARSON)
  --picos_hash_t_size PICOS_HASH_T_SIZE
                        DCT hash table size
                        (def: 64)

User-defined files:
  --user_config USER_CONFIG
                        path to the JSON file containing user configuration
  --user_constraints USER_CONSTRAINTS
                        path to the user defined constraints file
  --user_pre_design USER_PRE_DESIGN
                        path to the user TCL script to be executed before the design␣
↪step (not after the board base design)
  --user_post_design USER_POST_DESIGN
                        path to the user TCL script to be executed after the design step

Miscellaneous:
  -h, --help            show this help message and exit
  -i, --verbose_info    print extra information messages
  --dump_board_info     dump board info json for the specified board
  -j, --jobs JOBS       specify the number of jobs to run simultaneously
                        By default it will use as many jobs as cores with at least 5GB␣
↪of dedicated free memory, or the value returned by `nproc`, whichever is less.
  --mem_per_job MEM_PER_JOB
                        specify the memory per core used to estimate the number of jobs␣
↪to launch
                        (def: 5G)
  -k, --keep_files      keep files on error
  -v, --verbose         print vendor backend messages
  --version             print AIT version and exits

Xilinx-specific arguments:
  --regslice_pipeline_stages REGSLICE_PIPELINE_STAGES
                        number of register slice pipeline stages per SLR
```

(continues on next page)

(continued from previous page)

```
                        'x:y:z': add between 1 and 5 stages in master:middle:slave SLRs
                        auto: let Vivado choose the number of stages
                        (def: auto)
  --interconnect_regslices
                        enable register slices on AXI interconnects
  --interconnect_opt OPT_STRATEGY
                        AXI interconnect optimization strategy: Maximize 'performance'␣
↪or minimize 'area'
                        (def: performance)
  --interconnect_priorities
                        enable priorities in the memory interconnect
  --power_monitor       enable power monitoring infrastructure
  --thermal_monitor     enable thermal monitoring infrastructure
  --ignore_eng_sample   ignore engineering sample status from chip part number
  --target_language TARGET_LANG
                        choose target language to synthesize files to: vhdl or verilog
                        (def: verilog)

  environment variables:
    PETALINUX_BUILD   path where the Petalinux project is located
```

For a detailed explanation of the user configuration file, see:

3.2.2 AIT user configuration file

AIT behaviour can be fine-tuned by overridding some configuration values using the `--user_config` and passing a user-provided JSON file. The values specified in the file overrides any configuration done via AIT parameters.

Current supported configuration:

- Defining placement per accelerator instance
- Defining memory interface connection per data interface
- Disabling data interfaces
- Marking data interfaces for debug
- Defining register slices pipeline stages per accelerator instance

Note: It is not mandatory to define every configuration possible for each accelerator, instance or interface in the design. If a configuration is not defined by the user configuration file, it will either take the value passed by an AIT parameter or the default one.

placement

In order to define placement for an accelerator instance, the key `placement` has to be placed inside the instance object of the accelerator that wants to be placed.

```
{
    "accs": {
        "dotproduct": {
            "instances": {
                "0": {
                    "placement": "0"
                },
                "1": {
                    "placement": "1"
                }
            }
        }
    }
}
```

This will constrain instance 0 of `dotproduct` accelerator to the SLR0 and instance 1 to the SLR1.

Accelerator instances without placement defined will be automatically managed by Vivado.

dst

The memory interface where an accelerator instance data interface will be connected can be specified with the key `dst` inside the interfaces object.

```
{
    "accs": {
        "dotproduct": {
            "instances": {
                "0": {
                    "interfaces": {
                        "v1": {
                            "dst": "2"
                        }
                    }
                },
                "1": {
                    "interfaces": {
                        "v1": {
                            "dst": "None"
                        },
                        "result": {
                            "dst": "0"
                        }
                    }
                }
            }
        }
    }
}
```

This will connect data interface v1 of instance 0 of dotproduct accelerator to the memory interface 2, data interface result of instance 1 to memory interface 0, while data interface v1 of instance 1 will be left dangling. Data interfaces without dst defined will be connected automatically to a memory interface following a least-occupied policy.

debug

Data interfaces can be marked for debug and connected to an ILA automatically with the key debug inside the interfaces object.

```
{
    "accs": {
        "dotproduct": {
            "instances": {
                "0": {
                    "interfaces": {
                        "v1": {
                            "debug": "True"
                        }
                    }
                },
                "1": {
                    "interfaces": {
                        "v1": {
                            "debug": "True"
                        },
                        "result": {
                            "dst": "False"
                        }
                    }
                }
            }
        }
    }
}
```

This will mark for debug data interfaces v1 of instances 0 and 1 of dotproduct and connect them to an ILA. No action will be taken for data interfaces without debug defined or defined as False.

regslice_pipeline_stages

The number of pipeline stages created for each section of an SLR-crossing register slice can be configured with the key regslice_pipeline_stages for each accelerator instance.

```
{
    "accs": {
        "dotproduct": {
            "instances": {
                "0": {
                    "regslice_pipeline_stages": "2:3:4"
                },
                "1": {
                    "regslice_pipeline_stages": ":auto:1"
```

(continues on next page)

(continued from previous page)

```
                }
            }
        }
    }
}
```

This will configure the register slices affecting instance `0` of `dotproduct` as having 2 pipeline stages in the source section, 3 in the middle and 4 in the destination one. Register slices of instance 1 of the `dotproduct` accelerator the following way: default (or the value passed to AIT by the `--regslice_pipeline_stages` parameter) for the source section, `auto` for the middle one (meaning that Vivado will automatically decide the number of stages) and 1 for the destination section.

Finally, the following sections describe more advanced configurations regarding design optimization and accelerator placement:

3.2.3 Design optimization parameters

Memory access interleaving

By default, FPGA memory is allocated sequentially. By setting the `--memory_interleaving_stride=<stride>` option will result in allocations being placed in different modules each *stride* bytes. Therefore accelerator memory accesses will be scattered across the different memory interfaces.

For instance, setting `--memory_interleaving_stride=4096`. Will result in the first 4k being allocated to bank 0, next 4k are allocated info bank 1, and so on.

This may improve accelerator memory access bandwidth when combined with the *Interconnect optimization strategy* option:

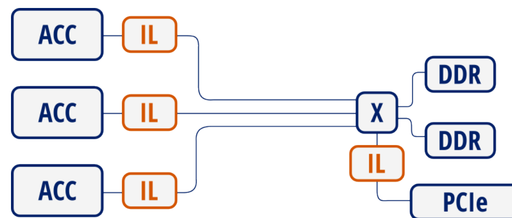

Fig. 1: Memory interconnection with memory interleaving enabled

Interconnect optimization strategy

Option `--interconnect_opt=<optimization strategy>` defines the optimization strategy for AXI interconnects.

This option only accepts `area` or `performance` strategies. While *area* results in lower resource usage, performance is lower than the *performance* setting.

In particular, using *area* prevents access from different slaves into different masters do be performed in parallel.

See also Xilinx PG059 for more details on the different strategies.

Interconnect register slices

By specifying `--interconnect_regslices` option, it enables *outer and auto* register slice mode on interconnect cores.

This mode places an extra *outer* register between the inner interconnect logic (crossbar, width converter, etc.) and the outer core slave interfaces. It also places an *auto* register slice if the slave interface is in the same clock domain. See Xilinx PG059 for details on these modes.

3.2.4 Accelerator placement options

This section documents how to constrain accelerators to a particular Super Logic Region (SLR) in a device.

Accelerator placement is controlled through the user configuration file. See *placement*.

On an Alveo U200, which has 3 SLRs, external interfaces are placed as follows:

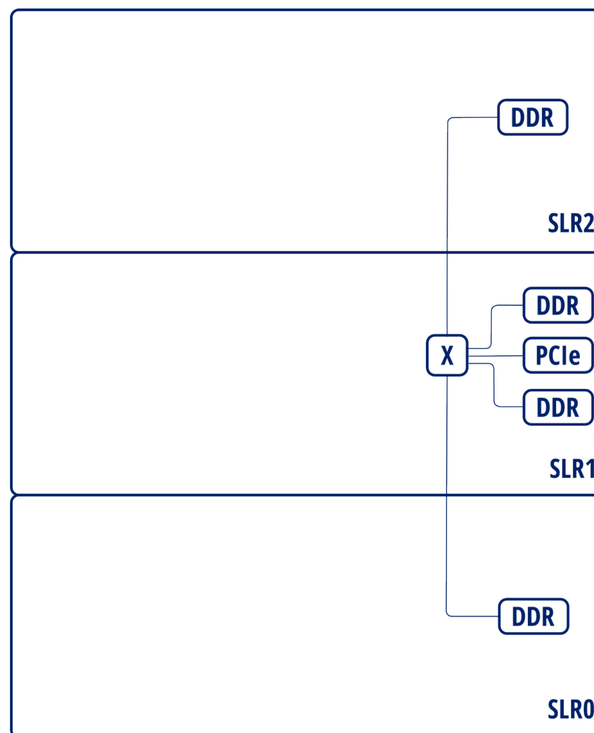

Fig. 2: Interface layout for Alveo U200

By default, all user accelerators are placed as Vivado considers. Sometimes it places a kernel accelerator between 2 SLRs, usually negatively impacting timing. Users can enforce accelerators to be constrained to an SLR region in order to prevent it from being scattered across multiple SLRs. For instance, a user can specify something as follows:

Additionally, AIT adds register slices in the AXI and AXI-Stream interfaces of each accelerator and automatically configures those that cross multiple SLRs in order to insert pipeline stages to mitigate critical paths. For example, register slices added for the previous design will result in the following layout:

Users can configure the number of pipeline stages to be inserted for each section of the SLR-crossing register slice (source, middle and destination) using either the global option `--regslice_pipeline_stages` or specify a per-instance configuration through the user configuration file (*placement*).

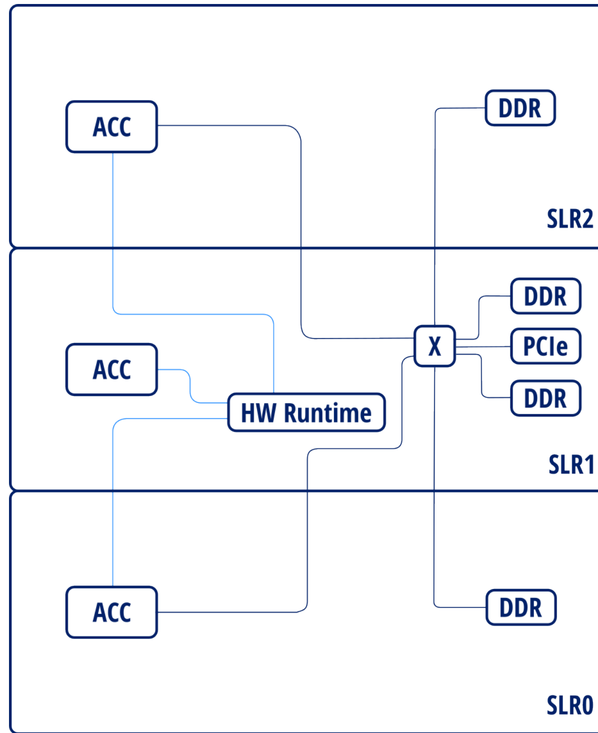

Fig. 3: Placed instance diagram

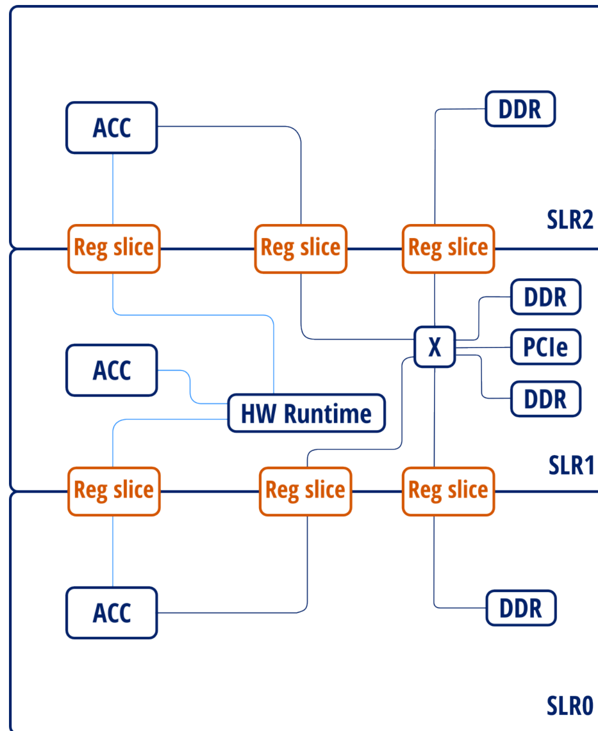

Fig. 4: Placed instance diagram with register slices

3.3 Binaries

To compile applications with LLVM/Clang you must add the flag `-fompss-2` when using either:

- `clang++` for C++ applications.
- `clang` for C applications.

3.4 Bitstream

Note: LLVM/Clang expects the Accelerator Integration Tool (AIT) to be available on the PATH, if not the linker will fail. Moreover, AIT expects Vitis HLS and Vivado to be available in the PATH.

To generate the bitstream, you should enable the bitstream generation in the LLVM/Clang compiler (using the `-fompss-fpga-wrapper-code` flag) and provide the FPGA linker (aka AIT) flags with `-fompss-fpga-ait-flags` option. If the FPGA linker flags does not contain the `-b` (or `--board`) and `-n` (or `--name`) options, the linker phase will fail.

For example, to compile the dotproduct application, in debug mode, for the Alveo U200, with a target frequency of 300Mhz, you can use the following command:

```
clang -fompss-2 -fompss-fpga-wrapper-code \
  src/dotproduct.c -o dotproduct-d \
  -fompss-fpga-ait-flags "--board=alveo_u200 --clock=300 --name=dotproduct"
```

3.4.1 HW Instrumentation

You must use the `-fompss-fpga-instrumentation` option of LLVM/Clang to enable the HW instrumentation generation. Keep in mind that a bitstream generated with instrumentation support will hang if instrumentation is not enabled at the runtime level.

For example, the previous compilation command with the instrumentation available will be:

```
clang -fompss-2 -fompss-fpga-wrapper-code \
  src/dotproduct.c -o dotproduct-d \
  -fompss-fpga-instrumentation \
  -fompss-fpga-ait-flags "--board=alveo_u200 --clock=300 --name=dotproduct"
```

3.4.2 Shared memory port

By default, LLVM/Clang generates an independent port to access the main memory for each task argument. Moreover, the bit-width of those ports equals to the argument data type width. This can result in a huge interconnection network when there are several task accelerators or they have several non-scalar arguments.

This behavior can be modified to generate unique shared port to access the main memory between all task arguments. This is achieved with the `-fompss-fpga-memory-port-width` option of LLVM/Clang which defines the desired bit-width of the shared port. The value must be a common multiple of the bit-widths for all task arguments.

The usage of the LLVM/Clang variable to generate a 128 bit port in the previous dotproduct command will be like:

```
clang -fompss-2 -fompss-fpga-wrapper-code \
  src/dotproduct.c -o dotproduct-d \
  -fompss-fpga-memory-port-width 128 \
  -fompss-fpga-ait-flags "--board=alveo_u200 --clock=300 --name=dotproduct"
```

3.5 Boot Files

Some boards do not support loading the bitstream into the FPGA after the boot, therefore the boot files should be updated and the board rebooted. AIT supports the generation of boot files for some boards but the step is disabled by default and should be enabled by hand.

First, you need to set the following environment variables:

- `PETALINUX_BUILD`. Petalinux project directory. See *Generate boot files for Xilinx SoC boards* to have more information about how to setup a petalinux project build.

Then you can invoke AIT with the same options provided in `-fompss-fpga-ait-flags` and the following new options: `--from_step=boot --to_step=boot`. Also, you may directly add the `--to_step=boot` option in `-fompss-fpga-ait-flags` during the LLVM/Clang launch.

CHAPTER

FOUR

RUNNING OMPSS-2@FPGA PROGRAMS

To run an OmpSs-2@FPGA program you should follow the general OmpSs-2 run procedure. More information is provided in the OmpSs-2 User Guide.

4.1 Nanos6 FPGA Architecture configuration

The Nanos6 behavior can be tuned with different configuration options. They are summarized and briefly described in the Nanos6 default configuration file as well as in the OmpSs-2 user guide, the FPGA architecture section is shown below:

```
[devices]
        directory = true
        [devices.fpga]
                # Enable/disable the reverse offload service
                reverse_offload = false
                # Byte alignment of the fpga memory allocations
                alignment = 128
                # If xtasks supports async copies, it can be "async", if not, the␣
↪runtime can use the default xtasks memcpy and
                # simulate an asynchronous copy spawning a new thread with "forced async
↪". Copies can also be synchronous with "sync".
                mem_sync_type = "sync"
                page_size = 0x8000
                requested_fpga_memory = 0x40000000
                # Enable FPGA device service threads. It is useful to disable them when␣
↪using the broadcaster, because in that case the
                # FPGAs are handled by the broadcaster device service and the FPGA␣
↪services are not used.
                enable_services = true
                # Number of events per accelerator in the instrumentation buffer. Must␣
↪be >= 128 and < 2**32.
                num_instr_events = 128
                # Maximum number of FPGA tasks running at the same time
                streams = 16
                [devices.fpga.polling]
                        # Indicate whether the FPGA services should constantly run while␣
↪there are FPGA tasks
                        # running on their FPGA. Enabling this option may reduce the␣
↪latency of processing FPGA
                        # tasks at the expenses of occupying a CPU from the system.␣
```

(continues on next page)

(continued from previous page)

```
↪Default is true
                        pinned = true
                        # The time period in microseconds between FPGA service runs.␣
↪During that time, the CPUs
                        # occupied by the services are available to execute ready tasks.␣
↪Setting this option to 0
                        # makes the services to constantly run. Default is 1000
                        period_us = 1000
```

4.2 Running single node applications

4.2.1 Running on QDMA-based FPGAs

See *Alveo hardware setup* to setup the hardware requisites for Alveo devices.

Loading the bitstream

Bitstreams can be loaded via JTAG through a USB connection with the FPGA board.

Run the following commands in Vivado interactive mode:

```
vivado -nolog -nojournal -mode tcl
```

```
open_hw_manager
connect_hw_server
open_hw_target
foreach {dev} [get_hw_devices] {
  current_hw_device ${dev}
  set_property PROGRAM.FILE <BITSTREAM_PATH> ${dev}
  program_hw_devices ${dev}
}
exit
```

Set up qdma queues

For DMA transfers to be performed between system main memory and the FPGA memory, qdma queues has to be set up by the user *prior to any execution*.

In this case `dma-ctl` tool is used. For instance, in order to create and start a memory mapped qdma queue with index 1 run:

```
dma-ctl qdmab3000 q add idx 1 mode mm dir bi
dma-ctl qdmab3000 q start idx 1 mode mm dir bi
```

OmpSs-2@FPGA runtime system expects an mm queue at index 1, which can be created with the commands listed above.

In the same fashion, these queues can also be removed:

```
dma-ctl qdmab3000 q stop idx 1 mode mm dir bi
dma-ctl qdmab3000 q del idx 1 mode mm dir bi
```

For more information, see

```
dma-ctl --help
```

4.2.2 Running on System-on-Chips

Loading bistreams

The FPGA bitstream needs to be loaded before the application can run. Xilinx provides the `fpgautil` utility in order to simplify bitstream loading.

```
fpgautil -b bitstream.bin
```

4.3 Running OMPIF applications

Multi-node multi-fpga applications developed using the *OMPIF cluster API* need special setup that is not needed in regular OmpSs@FPGA applications.

4.3.1 Load cluster configuration scripts

Cluster configuration is done using the `ompif.py` cluster configuration script, which is available in the ompss-2 module (`ompss-2/x86_64/git` for instance). For more information about this tool, you can use the `--help` flag to get general information about the tool or each individual command:

```
ompif.py --help
ompif.py create_cluster_file --help
```

4.3.2 Application execution

This section covers cluster environment setup and application execution. It is assumed that `vivado` and `dma-ctl` are available in the path. Those are available in the ompss-2 environment module.

Creating cluster description file

A json file describing the cluster needs to be created for the cluster to be automatically configured. It contains, for each FPGA, its index inside the node, the node name and the bitstream absolute path:

```
[
    { "fpga": FPGA_INDEX, "node": NODE_NAME, "bitstream": BITSTREAM_PATH }
]
```

The following example configures a cluster using 4 FPGAs in 2 different nodes (2 FPGAs each node):

```
[
    { "node": "fpgan01", "fpga": 0, "bitstream" : "/absolute/path/to/bitstream.bit" },
    { "node": "fpgan01", "fpga": 1, "bitstream" : "/absolute/path/to/bitstream.bit" },
    { "node": "fpgan02", "fpga": 0, "bitstream" : "/absolute/path/to/bitstream.bit" },
    { "node": "fpgan02", "fpga": 1, "bitstream" : "/absolute/path/to/bitstream.bit" }
]
```

Information regarding FPGAs, can be found in `/etc/motd` in each of the FPGA nodes.

The `ompif.py` management script includes a tool to generate the cluster file based on the allocated nodes in the current job and the number of FPGAs requested:

```
ompif.py create_cluster_file number_of_fpgas bitstream_file
```

For instance to configure a cluster with FPGAs with `bitstream.bit` file:

```
ompif.py create_cluster_file 6 bitstream.bit
```

This reads the list of nodes from the `$SLURM_JOB_NODELIST` environment variable set by slurm. A node list can be explicitly specified using the `--nodelist` flag.

Configuring the FPGA cluster

Once the cluster file is created, the cluster can be configured using the `create_cluster` command from the `ompif.py` tool.

```
ompif.py create_cluster cluster.json
```

The script will automatically launch servers and connect to them. For each node, a log file `${NODENAME}_servitor.log` is created in the current working directory. You can change the path with the `--log_prefix` flag. Also, for each node the script creates the `xtasks_devs_$(hostname).sh` file, with the `XTASKS_PCI_DEV` and `XDMA_QDMA_DEV` environment variables. By default it is created in the current working directory, but it can be changed with the `--xtasks_cluster_prefix` flag. The use of this file is explained in *Start xtasks servers*. The script also creates the `xtasks.cluster` file needed by your application in the current working directory. This file must be in the same directory where you launch the application, or also you can set the path in the `XTASKS_CLUSTER_FILE` environment variable. If not, the application will assume you are executing in single-node mode, and will not connect to the remote servers.

Start xtasks servers

A remote server that listens for FPGA tasks needs to be started in each of the remote nodes:

```
ompif.py launch_xtasks_server cluster.json
```

Log files are created in the current directory named `${NODENAME}_xtasks.log`. You can change the path with the `--log_prefix` flag. The script launches the server in the current working directory, so you must have the `xtasks.cluster` file in the same directory. For the moment, it can't be set with an environment variable. Also, by default all `xtasks_devs_$(hostname).sh` files must be in the current directory as well. However, you can set another path with the `--xtasks_devs_prefix` flag. Then, the cluster application can be run as usual.

Debugging

There are many debug registers that can be read with QDMA, including the number of received messages, number of corrupted messages, number of send/receive tasks, etc. More details in *POM AXI-Lite interface memory map*.

4.4 POM AXI-Lite interface memory map

The POM hardware runtime includes an optional AXI-Lite interface to access internal debug registers (read-only). The mapped address space is 16KB (14-bit addresses).

The available registers and their respective addresses are (size in bytes):

Register name	Address	Size	Description
COPY_OPT_IN	0x0	4	Number of copy in optimizations in the command in queue (both internal and external)
COPY_OPT_OUT	0x4	4	Number of copy in optimizations in the command in queue (both internal and external)
ACC_AVAIL	0x8	8	One bit per accelerator, indicating the availability status
QUEUE_NEMPTY	0x10	8	One bit per accelerator, indicating if the internal queue is not empty
CMD_IN_N_CMDS	0x800	4 per acc	For each accelerator, the number of commands it has received
CMD_OUT_N_CMI	0x900	4 per acc	For each accelerator, the number of commands it has issued
ACC_AVAIL_COUI	0xA00	8 per acc	For each accelerator, the total number of execution cycles

4.4.1 How to enable the AXI-Lite interface

You can enable it with the `--enable_pom_axilite` flag in AIT. For a full list of AIT options, see *AIT options*

4.4.2 How to read the registers with QDMA

You can use the script located in the https://gitlab.pm.bsc.es/ompss-at-fpga/rtl/meep-ompss-fpga-cluster repository, under `scripts/axilite_cntrl.py`.

```
python3 $MEEP-OMPSS-FPGA-CLUSTER/scripts/axilite_cntrl.py --read_pom
```

The script needs the `XTASKS_PCI_DEV` variable.

If you are using OMPIF, you can read all registers, including the message sender and receiver, with the `cluster.json` file.

```
python3 $MEEP-OMPSS-FPGA-CLUSTER/scripts/axilite_cntrl.py --cluster cluster.json
```

If some FPGAs are on remote nodes, the script automatically launches servers on each remote node. However, you need to have the `xtasks_devs_$(hostname).sh` files in the current working directory.

4.5 Ovni FPGA instrumentation

FPGA accelerator instrumentation is provided via ovni. By default, dependency copies and kernel execution are traced when instrumentation is enabled.

4.5.1 Prerequisites

The bitstream needs to be compiled with instrumentation support to create trace accelerator events. Instrumentation can be enabled by adding the `-fompss-fpga-instrumentation` flag when building the bitstream. More details on building OmpSs@FPGA applications are available in the *Compile OmpSs-2@FPGA programs* section.

Also, ovni and paraver need to be installed to create and visualize traces. Ovni is automatically installed and updated in all the nodes.

4.5.2 Running the application

Enabling instrumentation

Instrumentation needs to be enabled in nanos6 configuration toml file. The default file is located in the nanos6 installation directory.

```
$OMPSS_FPGA_HOME/bsc/x86_64/ompss-2/<release>/nanos6/share/nanos6.toml
```

`$OMPSS_FPGA_HOME` is defined by the OmpSs@FPGA environment module. The `release` is the specific release that is being used, `3.2.1` or `git`, for instance.

This file can be copied to the working directory to edit it and override default nanos6 settings. For instance, to copy the configuration file from the git release, run:

```
cp $OMPSS_FPGA_HOME/bsc/x86_64/ompss-2/git/nanos6/share/nanos6.toml .
```

Then, set the instrument entry to `ovni`:

```
[version]
    instrument = "ovni"
```

Then run the application as usual. See *Running OmpSs-2@FPGA Programs* for more details.

Note: Using an instrumentation-enabled bitstream without enabling instrumentation at the runtime level will result in the application hanging.

After the program finishes, an `ovni` directory containing ovni traces should be created.

4.5.3 Processing traces

Ovni traces need to be converted to paraver traces to be visualized using paraver. Paraver traces need to be generated from ovni traces for visualization. This is done using `ovniemu` tool:

```
ovniemu -x myapp.xtasks.config ovni/
```

In this example, `myapp.xtasks.config` is passed to `ovniemu` (using `-x` flag) to the tool is able to read accelerator names for properly displaying them.

The output from the emulation process should look like this:

```
ovniemu: INFO: loaded 16 streams
ovniemu: INFO: sorting looms by name
ovniemu: INFO: loaded 1 looms, 1 processes, 16 threads and 8 cpus
ovniemu: INFO: generated with libovni version 1.10.0 commit unknown
ovniemu: INFO: the following 3 models are enabled:
ovniemu: INFO:    nanos6 1.1.0 '6' (67 events)
ovniemu: INFO:      ovni 1.1.0 'O' (18 events)
ovniemu: INFO:    xtasks 1.0.0 'X' (1 events)
ovniemu: INFO: emulation starts
ovniemu: INFO: loom.fpgan10.770230 burst stats: median/avg/max =  98/102/361 ns
ovniemu: INFO: 100.0% done at avg 1240 kev/s
ovniemu: INFO: processed 1446408 input events in 1.17 s
ovniemu: INFO: writing traces to disk, please wait
ovniemu: INFO: emulation finished ok
```

In the `ovni/` directory, two paraver traces should have been created as well as some paraver config files:

```
cfg/                     Paraver config files
cpu.pcf                  CPU paraver trace files
cpu.prv
cpu.row
loom.fpgan10.770230/     Ovni trace directory
thread.pcf               Thread trace files
thread.prv
thread.row
```

FPGA events are emitted to the `thread` trace. See also Paraver web page for further info on the visualization tool and download links.

CHAPTER

FIVE

GENERATE BOOT FILES FOR XILINX SOC BOARDS

To generate the required files to boot the SoC boards (Zynq and Zynq Ultrascale+ families), Xilinx offers the PetaLinux set of tools. Additionally, OmpSs@FPGA toolchain supports the automatic generation of boot files using these tools.

The following sections describe how to generate the boot files both manually and automatically.

5.1 Prerequisites

- PetaLinux installer (2021.2 or newer)
- Xilinx support archive (XSA) file from a synthesized Vivado project
- Board Support Package (BSP) file for the target board

5.1.1 PetaLinux installation

PetaLinux is installed running its auto-installer package:

```
./petalinux-v2023.2-10121855-installer.run --dir <installation directory>
```

After installation, you should source the PetaLinux environment file. Usually, this needs to be done every time you want to run from a new terminal.

> **Caution:** Sourcing PetaLinux settings file may change the ARM cross-compilers.

```
source <petalinux install dir>/settings.sh
```

5.2 PetaLinux project setup

The following steps should be executed once. After that, you will be able to generate boot files for the target board just using the `ait` feature or executing the steps in any of the following sections.

5.2.1 Unpack the bsp

Unpack the bsp to create the PetaLinux project:

```
petalinux-create -t project -s <path to board bsp> -n <project name>
```

5.2.2 Configure PetaLinux

Run PetaLinux configuration and change the root filesystem type to `ext4`:

```
petalinux-config
```

```
Image Packaging Configuration →
    Root filesystem type →
        EXT4 (SD/eMMC/SATA/USB)
```

You might also want to disable automatic copy to tftpboot to avoid a warning message at every build:

```
Image Packaging Configuration →
    Copy final images to tftpboot
```

Note: Some boards may fail to build the First Stage Boot Loader (FSBL) due to its size.

In order to shrink the FSBL provide the following flags to the compiler (this will disable support for NAND and QSPI boot modes):

```
FSBL Configuration →
    FSBL compiler flags →
        -DFSBL_NAND_EXCLUDE, -DFSBL_QSPI_EXCLUDE
```

5.2.3 Configure linux kernel

To enter the kernel configuration utility, run:

```
petalinux-config -c kernel
```

Note: You may want to increase the CMA size. It is used by Nanos6 as memory for the FPGA device copies. Its size can be set in:

```
Device drivers →
    Generic Driver Options →
        DMA Contiguous Memory Allocator
```

5.3 Generate boot files manually

Once PetaLinux project is setup, you can update it to contain a custom bitstream with your hardware. These steps can be repeated several times without executing again the steps in the *PetaLinux project setup* section.

First, you need to import the Xilinx support archive file (xsa) in the PetaLinux project. This is done by executing the following command in the root directory of the PetaLinux project build.

```
petalinux-config --silent-config --get-hw-description <path to project xsa file>
```

5.3.1 Add OmpSs@FPGA node to the device tree

The OmpSs@FPGA node must be added to the device-tree before compiling it.

Copy the file `pl_ompss_at_fpga.dtsi` generated by AIT to `project-spec/meta-user/recipes-bsp/device-tree/files/`.

Edit files `project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi` and `project-spec/meta-user/recipes-bsp/device-tree/device-tree.bbappend` and add the following lines:

```
echo '/include/ "pl_ompss_at_fpga.dtsi"' > project-spec/meta-user/recipes-bsp/device-
↪tree/files/system-user.dtsi
echo 'SRC_URI:append = " file://pl_ompss_at_fpga.dtsi"' > project-spec/meta-user/recipes-
↪bsp/device-tree/device-tree.bbappend
```

5.3.2 Build the Linux system

When the project is correctly updated, you can build it with the following command:

```
petalinux-build
```

5.3.3 Create BOOT.BIN file

Finally, generate the boot files by running the following command:

```
petalinux-package --force --boot --fsbl images/linux/zynq*_fsbl.elf --fpga <path to␣
↪bitstream file> --u-boot images/linux/u-boot.elf
```

5.4 Use AIT to generate boot files

The Accelerator Integration Tool (AIT) can automatically and transparently perform the steps described in *Generate boot files manually*. To do so, you must:

- Add the option `--to_step=boot` on the AIT call to enable the `boot` step
- Set the environment variable `PETALINUX_BUILD` with the path to the pre-configured PetaLinux build project of the target board

Once the boot files have been correctly generated, AIT will copy them into the project directory at `<AIT project path>/boot`.

5.5 Copy the files to the SD boot partition

Finally, mount the boot partition of the board SD into your system and copy the required files (device name and paths might not be the same):

```
udisksctl mount --block-device /dev/mmcblk0p1
cp <path to petalinux project>/images/linux/BOOT.BIN /media/<user>/boot/
cp <path to petalinux project>/images/linux/image.ub /media/<user>/boot/
cp <path to petalinux project>/images/linux/boot.scr /media/<user>/boot/
udisksctl unmount --block-device /dev/mmcblk0p1
```

CHAPTER SIX

SERVER USER GUIDES

6.1 quar user guide

quar takes its name from La Quar, which is a small town and municipality located in the comarca of Berguedà, in Catalonia.

The OmpSs-2@FPGA releases are automatically installed in the server. They are available through a module file for each target architecture. This document describes how to load and use the modules to compile an example application. Once the modules are loaded, the workflow in the server should be the same as in the Docker images.

6.1.1 General remarks

- The OmpSs-2@FPGA toolchain is installed in a version folder under the `/opt/bsc/` directory.
- Third-party libraries required to run some programs are installed in the corresponding folder under the `/opt/lib/` directory.
- The rest of the software (Xilinx toolchain, slurm, modules, etc.) is installed under the `/tools/` directory.

6.1.2 Node specifications

- CPU: Intel Xeon Silver 4208 CPU
  - https://ark.intel.com/content/www/us/en/ark/products/193390/intel-xeon-silver-4208-processor-11m-cache-2-10-ghz.html.
- Main memory: 64GB DDR4-3200
- FPGAs:
  - 2x Xilinx Alveo U200
    - https://www.amd.com/en/products/accelerators/alveo/u200/a-u200-a64g-pq-g.html

6.1.3 Logging into the system

quar is accessible from HCA `ssh.hca.bsc.es` Alternatively, it can be accessed through the `8419` port in HCA and ssh connection will be redirected to the actual host:

```
ssh -p 8419 ssh.hca.bsc.es
```

Also, this can be automated by adding a `quar` host into ssh config:

```
Host quar
    HostName ssh.hca.bsc.es
    Port 8419
```

6.1.4 Module structure

The ompss-2 modules are:

- `ompss-2/x86_64/*[release version]*`

This will automatically load the default Vivado version, although an arbitrary version can be loaded before ompss-2:

```
module load vivado/2023.2 ompss-2/x86_64/git
```

To list all available modules in the system run:

```
module avail
```

6.1.5 Build applications

To generate an application binary and bitstream, you could refer to *Compile OmpSs-2@FPGA programs* as the steps are general enough.

Note that the appropriate modules need to be loaded. See *Module structure*.

6.1.6 Running applications

Get access to an installed fpga

The server uses Slurm in order to manage access to computation resources. Therefore, to be able to use the resources of an FPGA, an allocation in one of the partitions has to be made.

You can check the number and name of partitions and nodes by running:

```
sinfo -Nel
```

There is 1 partition in the node:

- `fpga`: 2x alveo_u200

In order to make an allocation of computing resources, you must run `salloc` with the `--gres` option.

For instance:

```
salloc -p fpga --gres=fpga:BOARD:N
```

Where BOARD is the FPGA to allocate and N the number of FPGAs to allocate.

This command will allocate the number of specified FPGAs with the required tools and file permissions already set by slurm and prevent other users from using those resources.

Once inside an allocation you can run a script or an interactive job with a subset of the allocated resources with srun:

For an interactive job, run:

```
srun --gres=fpga:BOARD:N --pty bash
```

To execute a script, run:

```
srun --gres=fpga:BOARD:N script.sh
```

Note: You can also allocate and run a job in a single command with srun. There is no need to pre-allocate resources with salloc.

Warning: Just running an salloc will not set the OmpSs-2@FPGA environment variables. In order to do so, you must run your job through srun.

Alternatively you can also run your jobs asynchronously through an sbatch command, passing a slurm job script as argument:

```
sbatch --gres=fpga:BOARD:N job_script.sh
```

Being an example job_script.sh:

```
#!/bin/bash
#
#SBATCH --job-name=ompss-2_fpga_test
#SBATCH --output=out.txt
#SBATCH --time=05:00
#SBATCH --gres=fpga:BOARD:N
#SBATCH -p fpga

module load ompss-2/x86_64/git

cd test
make binary

srun --gres=fpga:BOARD:N exec_test.sh
```

To get information about the active slurm jobs, run:

```
squeue
```

The output should look similar to this:

```
JOBID   PARTITION   NAME   USER       ST  TIME    NODES   NODELIST(REASON)
1312    fpga        bash   afilguer   R   17:14   1       quar
```

Loading bistreams

The FPGA bitstream needs to be loaded before the application can run. The `load_bitstream` utility is provided in order to simplify programming the FPGA.

```
load_bitstream bitstream.bit [index]
```

The utility receives a second optional parameter to indicate which of the allocated FPGAs to program. The default behavior is to program all the allocated FPGAs with the bitstream.

To know which FPGAs have been allocated, you can run the `report_slurm_node` tool. The output should be similar to this:

```
LOCAL_ID  PCI_DEV       USB_DEV  QDMA_DEV  HWSERVER_PORT  GLOBAL_ID
0         0000:b3:00.0  001:002  b3000     13330          0
1         0000:65:00.0  001:003  65000     13331          1
```

You can also run `load_bitstream` with the `-h` option to see which FPGAs are available to program:

```
Usage load_bitstream bitstream.bit [index]
Available devices:
index:  jtag           pcie          usb
0:      21290594G00LA  0000:b3:00.0  001:002
1:      21290594G00EA  0000:65:00.0  001:003
```

Get current bitstream info

In order to get information about the bitstream currently loaded into the FPGA, the tool `read_bitinfo` is installed in the system.

```
read_bitinfo
```

Note that an active slurm reservation is needed in order to query the FPGA.

This call should return something similar to the sample output for a cholesky decomposition application:

```
Bitinfo of FPGA 0000:b3:00.0:
Bitinfo version: 16
Bitstream user-id: 0xA41B665D
AIT version: 8.2.0
Wrapper version: 13
Number of accelerators: 9
Board base frequency: 156.25 MHz
Dedicated FPGA memory: 64 GB
Memory interleaving: 32768

Features:
[ ] Instrumentation
[ ] Hardware counter
Interconnect optimization
    [ ] Area
    [x] Performance
Picos OmpSs Manager
    [ ] AXI-Lite
```

(continues on next page)

(continued from previous page)

```
    [x] Task creation
    [x] Dependencies
    [ ] Lock
    [x] Spawn queues
[ ] Power monitor (CMS)
[ ] Thermal monitor (sysmon)
[ ] OMPIF
[ ] IMP

Address map:
Managed rstn - address 0x1000
CmdIn - address 0x2000 length 64
CmdOut - address 0x4000 length 64
SpawnIn - address 0x6000 length 1024
SpawnOut - address 0x8000 length 1024
Hardware counter - not enabled
POM AXI-Lite - not enabled
Power monitor (CMS) - not enabled
Thermal monitor (sysmon) - not enabled

xtasks accelerator config:
type        count   freq(KHz)   description
5862896218  1   300000      cholesky_blocked
5459683074  1   300000      omp_trsm
5459653839  1   300000      omp_syrk
5459186490  6   300000      omp_gemm

ait command line:
ait --name=cholesky --board=alveo_u200 -c=300 --user_config=config_files/alveo_u200_
↪performance.json --memory_interleaving_stride=32K --interconnect_priorities --
↪interconnect_regslices --max_deps_per_task=3 --max_args_per_task=3 --max_copies_per_
↪task=3 --picos_tm_size=256 --picos_dm_size=645 --picos_vm_size=775 --wrapper_version 13

Hardware runtime VLNV:
bsc:ompss:picos_ompss_manager:7.5
```

Remote debugging

Although it is possible to interact with Vivado's Hardware Manager through ssh-based X forwarding, Vivado's GUI might not be very responsive over remote connections. To avoid this limitation, one might connect a local Hardware Manager instance to targets hosted on quar, completely avoiding X forwarding, as follows.

1. On quar, when allocating an FPGA with Slurm, a Vivado HW server is automatically launched for each FPGA:
   - FPGA 0 uses port 3120
   - FPGA 1 uses port 3121
2. On the local machine, assuming that quar's HW Server runs on port 3120, let all connections to port 3120 be forwarded to quar by doing `ssh -L 3120:quar:3120 [USER]@ssh.hca.bsc.es -p 8410`.
3. Finally, from the local machine, connect to quar's hardware server:
   - Open Vivado's Hardware Manager.

- Launch the “Open target” wizard.
- Establish a connection to the local HW server, which will be just a bridge to the remote instance.

Enabling OpenCL / XRT mode

FPGA in quar can be used in OpenCL / XRT mode. Currently, XRT 2022.2 is installed. To enable XRT the shell has to be configured into the FPGA and the PCIe devices re-enumerated after configuration has finished.

This is done by running

```
load_xrt_shell
```

Note that this has to be done while a slurm job is allocated. After this process has completed, output from `lspci -vd 10ee:` should look similar to this:

```
b3:00.0 Processing accelerators: Xilinx Corporation Device 5000
        Subsystem: Xilinx Corporation Device 000e
        Flags: bus master, fast devsel, latency 0, NUMA node 0
        Memory at 383ff0000000 (64-bit, prefetchable) [size=32M]
        Memory at 383ff4000000 (64-bit, prefetchable) [size=256K]
        Capabilities: <access denied>
        Kernel driver in use: xclmgmt
        Kernel modules: xclmgmt

b3:00.1 Processing accelerators: Xilinx Corporation Device 5001
        Subsystem: Xilinx Corporation Device 000e
        Flags: bus master, fast devsel, latency 0, IRQ 105, NUMA node 0
        Memory at 383ff2000000 (64-bit, prefetchable) [size=32M]
        Memory at 383ff4040000 (64-bit, prefetchable) [size=256K]
        Memory at 383fe0000000 (64-bit, prefetchable) [size=256M]
        Capabilities: <access denied>
        Kernel driver in use: xocl
        Kernel modules: xocl
```

Also XRT devices should show up as ready when running `xbutil examine`. Note that the xrt/2022.2 has to be loaded.

```
module load xrt/2022.2
xbutil examine
```

And it should show this output:

```
System Configuration
  OS Name              : Linux
  Release              : 5.4.0-97-generic
  Version              : #110-Ubuntu SMP Thu Jan 13 18:22:13 UTC 2022
  Machine              : x86_64
  CPU Cores            : 16
  Memory               : 63812 MB
  Distribution         : Ubuntu 18.04.2 LTS
  GLIBC                : 2.31
  Model                : PowerEdge T640

XRT
```

(continues on next page)

(continued from previous page)

```
  Version               : 2.14.354
  Branch                : 2022.2
  Hash                  : 43926231f7183688add2dccfd391b36a1f000bea
  Hash Date             : 2022-10-08 09:49:58
  XOCL                  : 2.14.354, 43926231f7183688add2dccfd391b36a1f000bea
  XCLMGMT               : 2.14.354, 43926231f7183688add2dccfd391b36a1f000bea

Devices present
BDF                :  Shell                            Platform UUID                 ␣
↪ Device ID        Device Ready*
-----------------------------------------------------------------------------------------
↪---------------------------------
[0000:b3:00.1]  :  xilinx_u200_gen3x16_xdma_base_2  0B095B81-FA2B-E6BD-4524-72B1C1474F18␣
↪ user(inst=128)  Yes


* Devices that are not ready will have reduced functionality when using XRT tools
```

6.2 crdbmaster user guide

crdbmaster takes its name from *CRDB*, a compute node developed within the EuroEXA project.

The OmpSs-2@FPGA releases are automatically installed in the server. They are available through a module file for each target architecture. This document describes how to load and use the modules to compile an example application. Once the modules are loaded, the workflow in the server should be the same as in the Docker images.

6.2.1 General remarks

- The OmpSs-2@FPGA toolchain is installed in a version folder under the `/opt/bsc/` directory.
- Third-party libraries required to run some programs are installed in the corresponding folder under the `/opt/lib/` directory.
- The rest of the software (Xilinx toolchain, slurm, modules, etc.) is installed under the `/tools/` directory.

6.2.2 Node specifications

- CPU: Intel Xeon E3-1220 CPU
  - https://www.intel.com/content/www/us/en/products/sku/52269/intel-xeon-processor-e31220-8m-cache-3-10-ghz/specifications.html
- Main memory: 32GB DDR3-1600
- FPGAs:
  - Xilinx Kria KV260
    - https://www.amd.com/en/products/system-on-modules/kria/k26/kv260-vision-starter-kit.html
  - Xilinx Zynq Ultrascale+ ZCU102
    - https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/ek-u1-zcu102-g.html

  - Xilinx Zynq 7000 ZC702
    - https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/ek-z7-zc702-g.html
  - Avnet/Digilent Zedboard (XC7Z7020)
    - https://www.avnet.com/americas/products/avnet-boards/avnet-board-families/zedboard/

6.2.3 Logging into the system

crdbmaster login node is accessible via ssh at `crdbmaster.bsc.es`.

```
ssh crdbmaster.bsc.es
```

6.2.4 Module structure

The ompss-2 modules are:

- `ompss-2/arm64/*[release version]*`

This will automatically load the default Vivado version, although an arbitrary version can be loaded before ompss-2:

```
module load vivado/2023.2 ompss-2/arm64/git
```

To list all available modules in the system run:

```
module avail
```

6.2.5 Build applications

To generate an application binary and bitstream, you could refer to *Compile OmpSs-2@FPGA programs* as the steps are general enough.

Note that the appropriate modules need to be loaded. See *Module structure*.

6.2.6 Running applications

Get access to an installed fpga

The server uses Slurm in order to manage access to computation resources. Therefore, to be able to use the resources of an FPGA, an allocation in one of the partitions has to be made.

You can check the number and name of partitions and nodes by running:

```
sinfo -Nel
```

There are 2 partitions in the node:

- `arm64`: kv260, zcu102
- `arm32`: zedboard, zynq702

In order to make an allocation of computing resources, you must run `srun`.

For instance:

```
srun -p arm32 --pty bash
```

Or allocate a specific board with:

```
srun -p arm64 --nodelist=zcu102 --pty bash
```

These commands will allocate an FPGA and run an interactive bash inside the FPGA node with the required tools and file permissions already set by slurm.

To get information about the active slurm jobs, run:

```
squeue
```

The output should look similar to this:

```
JOBID   PARTITION   NAME    USER        ST  TIME    NODES   NODELIST(REASON)
1312    arm32       bash    afilguer    R   17:14   1       zynq702
```

Loading bistreams

The FPGA bitstream needs to be loaded before the application can run. Xilinx provides the `fpgautil` utility in order to simplify bitstream loading.

```
fpgautil -b bitstream.bin
```

Get current bitstream info

In order to get information about the bitstream currently loaded into the FPGA, the tool `read_bitinfo` is installed in the system.

```
read_bitinfo
```

Note that an active slurm reservation is needed in order to query the FPGA.

This call should return something similar to the sample output for a matrix multiplication application:

```
Bitinfo version: 16
Bitstream user-id: 0xDD3ABA12
AIT version: 8.2.0
Wrapper version: 13
Number of accelerators: 3
Board base frequency: 125.00 MHz
Dedicated FPGA memory: not available
Memory interleaving: not enabled

Features:
[ ] Instrumentation
[ ] Hardware counter
Interconnect optimization
    [ ] Area
    [x] Performance
Picos OmpSs Manager
```

(continues on next page)

(continued from previous page)

```
    [ ] AXI-Lite
    [x] Task creation
    [x] Dependencies
    [ ] Lock
    [x] Spawn queues
[ ] Power monitor (CMS)
[ ] Thermal monitor (sysmon)
[ ] OMPIF
[ ] IMP

Address map:
Managed rstn - address 0x80001000
CmdIn - address 0x80002000 length 256
CmdOut - address 0x80004000 length 256
SpawnIn - address 0x80006000 length 1024
SpawnOut - address 0x80008000 length 1024
Hardware counter - not enabled
POM AXI-Lite - not enabled
Power monitor (CMS) - not enabled
Thermal monitor (sysmon) - not enabled

xtasks accelerator config:
type        count   freq(KHz)   description
5839957875  1   100000      matmulFPGA
7602000973  2   100000      matmulBlock

ait command line:
ait --name=matmul --board=zynq702 -c=100 --interconnect_regslices --wrapper_version 13

Hardware runtime VLNV:
bsc:ompss:picos_ompss_manager:7.5
```

6.3 xaloc user guide

xaloc takes its name from the Catalan form of Sirocco, which is the name of the Mediterranean wind that comes from the southeast.

The OmpSs-2@FPGA releases are automatically installed in the server. They are available through a module file for each target architecture. This document describes how to load and use the modules to compile an example application. Once the modules are loaded, the workflow in the server should be the same as in the Docker images.

6.3.1 General remarks

- The OmpSs-2@FPGA toolchain is installed in a version folder under the `/opt/bsc/` directory.
- Third-party libraries required to run some programs are installed in the corresponding folder under the `/opt/lib/` directory.
- The rest of the software (Xilinx toolchain, slurm, modules, etc.) is installed under the `/tools/` directory.

6.3.2 Node specifications

- CPU: Dual Intel Xeon X5680
  - https://ark.intel.com/content/www/us/en/ark/products/47916/intel-xeon-processor-x5680-12m-cache-3-33-ghz-6-40-gts-in html
- Main memory: 72GB DDR3-1333
- FPGA:
  - Xilinx Versal VCK5000
    - https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/vck5000.html

6.3.3 Logging into the system

xaloc is accessible from HCA `ssh.hca.bsc.es` Alternatively, it can be accessed through the `8410` port in HCA and ssh connection will be redirected to the actual host:

```
ssh -p 8410 ssh.hca.bsc.es
```

Also, this can be automated by adding a `xaloc` host into ssh config:

```
Host xaloc
    HostName ssh.hca.bsc.es
    Port 8410
```

6.3.4 Module structure

The ompss-2 modules are:

- `ompss-2/x86_64/*[release version]*`

This will automatically load the default Vivado version, although an arbitrary version can be loaded before ompss-2:

```
module load vivado/2023.2 ompss-2/x86_64/git
```

To list all available modules in the system run:

```
module avail
```

6.3.5 Build applications

To generate an application binary and bitstream, you could refer to *Compile OmpSs-2@FPGA programs* as the steps are general enough.

Note that the appropriate modules need to be loaded. See *Module structure*.

6.3.6 Running applications

Warning: Although the Versal board is installed and can be allocated via Slurm there is no toolchain support yet.

Get access to an installed fpga

The server uses Slurm in order to manage access to computation resources. Therefore, to be able to use the resources of an FPGA, an allocation in one of the partitions has to be made.

You can check the number and name of partitions and nodes by running:

```
sinfo -Nel
```

There is 1 partition in the node:

- `fpga`: versal

In order to make an allocation of computing resources, you must run `salloc` with the `--gres` option.

For instance:

```
salloc -p fpga --gres=fpga:BOARD:N
```

Where `BOARD` is the FPGA to allocate and `N` the number of FPGAs to allocate.

This command will allocate the number of specified FPGAs with the required tools and file permissions already set by slurm and prevent other users from using those resources.

Once inside an allocation you can run a script or an interactive job with a subset of the allocated resources with `srun`:

For an interactive job, run:

```
srun --gres=fpga:BOARD:N --pty bash
```

To execute a script, run:

```
srun --gres=fpga:BOARD:N script.sh
```

Note: You can also allocate and run a job in a single command with `srun`. There is no need to pre-allocate resources with `salloc`.

Warning: Just running an `salloc` will not set the OmpSs-2@FPGA environment variables. In order to do so, you must run your job through `srun`.

Alternatively you can also run your jobs asynchronously through an `sbatch` command, passing a slurm job script as argument:

```
sbatch --gres=fpga:BOARD:N job_script.sh
```

Being an example `job_script.sh`:

```
#!/bin/bash
#
#SBATCH --job-name=ompss-2_fpga_test
#SBATCH --output=out.txt
#SBATCH --time=05:00
#SBATCH --gres=fpga:BOARD:N
#SBATCH -p fpga

module load ompss-2/x86_64/git

cd test
make binary

srun --gres=fpga:BOARD:N exec_test.sh
```

To get information about the active slurm jobs, run:

```
squeue
```

The output should look similar to this:

```
JOBID   PARTITION   NAME    USER        ST  TIME    NODES   NODELIST(REASON)
1312    fpga        bash    afilguer    R   17:14   1       quar
```

To know which FPGAs have been allocated, you can run the `report_slurm_node` tool. The output should be similar to this:

```
LOCAL_ID  PCI_DEV       USB_DEV  QDMA_DEV  HWSERVER_PORT  GLOBAL_ID
0         0000:02:00.0  002:002  02000     13330          0
```

6.4 llebeig user guide

llebeig takes its name from the Catalan form of Libeccio, which is the name of the Mediterranean wind that comes from the southwest.

The OmpSs-2@FPGA releases are automatically installed in the server. They are available through a module file for each target architecture. This document describes how to load and use the modules to compile an example application. Once the modules are loaded, the workflow in the server should be the same as in the Docker images.

6.4.1 General remarks

- The OmpSs-2@FPGA toolchain is installed in a version folder under the `/opt/bsc/` directory.
- Third-party libraries required to run some programs are installed in the corresponding folder under the `/opt/lib/` directory.
- The rest of the software (Xilinx toolchain, modules, etc.) is installed under the `/tools/` directory.

6.4.2 Logging into the system

llebeig is accessible from HCA `ssh.hca.bsc.es` Alternatively, it can be accessed through the `8412` port in HCA and ssh connection will be redirected to the actual host:

```
ssh -p 8412 ssh.hca.bsc.es
```

Also, this can be automated by adding a `llebeig` host into ssh config:

```
Host llebeig
    HostName ssh.hca.bsc.es
    Port 8412
```

6.4.3 Module structure

The ompss-2 modules are:

- `ompss-2/x86_64/*[release version]*`

This will automatically load the default Vivado version, although an arbitrary version can be loaded before ompss:

```
module load vivado/2023.2 ompss-2/x86_64/git
```

To list all available modules in the system run:

```
module avail
```

6.4.4 Build applications

To generate an application binary and bitstream, you could refer to *Compile OmpSs-2@FPGA programs* as the steps are general enough.

Note that the appropriate modules need to be loaded. See *Module structure*.

6.5 MEEP-FPGA user guide

The MEEP-FPGA cluster, also known as *makinote*, is an FPGA cluster composed of 12 nodes with 8 FPGAs each for a total of 96 FPGAs. It also contains 4 nodes without FPGAs used for compilation and synthesis.

The OmpSs-2@FPGA releases are automatically installed in the MEEP-FPGA cluster. They are available through a module file for each target architecture. This document describes how to load and use the modules to compile an example application. Once the modules are loaded, the workflow in MEEP-FPGA should be the same as in the Docker images.

6.5.1 General remarks

- The OmpSs-2@FPGA toolchain is installed in a version folder under the `/home/genu/pmtest/opt/bsc/` directory.
- OmpSs-2@FPGA modules need to manually enabled.
- This cluster uses BSC HPC accounts. Users look like bsc0xxxxx.

6.5.2 Node specifications

Full node specifications are available at the support knowledge center: https://www.bsc.es/supportkc/docs/MEEP/overview

- CPU: Intel Xeon Gold 6330 with 28 cores @ 2.0GHz
  - https://ark.intel.com/content/www/us/en/ark/products/212458/intel-xeon-gold-6330-processor-42m-cache-2-00-ghz.html
- Main memory: 256 GB (16 RDIMM x 16GB DDR4 @ 3200 MHz)
- **FPGAs:**
  - 8x Xilinx Alveo UC55c

There are 12 FPGA nodes, 4 synthesis nodes and a login node. Synthesis and login nodes do not have FPGAs.

6.5.3 Logging into the system

Login node is accessible from the BSC internal network. To access from an external network, the VPN must be used. The login node is accessible from `fpgalogin1.bsc.es`

```
ssh bscxxxxx@fpgalogin1.bsc.es
```

6.5.4 Module structure

The default environment does not have the available modules for building OmpSs@FPGA applications. A suitable environment can be set up:

```
source ~pmtest/tools/ompss_fpga_init.sh
```

This will enable OmpSs-2@FPGA modules. Additionally, reasonably recent versions of python, cmake or clang are enabled.

Note: The loaded python 3.11, while it's needed by ait, will break gdb and maybe other system applications

The OmpSs-2@FPGA modules are:

- `ompss-2/x86_64/*[release version]*`

This will automatically load the default Vivado version, although an arbitrary version can be loaded before ompss:

```
module load vivado/2023.2 ompss-2/x86_64/git
```

To list all available modules in the system run:

```
module avail
```

6.5.5 Build applications

To generate an application binary and bitstream, you could refer to *Compile OmpSs-2@FPGA programs* as the steps are general enough.

Note that the appropriate modules need to be loaded. See *Module structure*.

To allocate a job in the synthesis nodes, the `gpp` partition needs to be used. To enable remote x11 graphics, the `--x11` needs to be specified.

For instance, to start an interactive session with graphics:

```
salloc -c 10 --mem=64G -t 4:00:00 -p gpp --x11
```

The job will be using 10 cores and 64GB of memory.

For a batch job with no graphics:

```
sbatch -c 10 --mem=64G -t 4:00:00 -p gpp build_script.sh
```

6.5.6 Running applications

This section describes how to allocate resources and set up the environment to run an OmpSs-2@FPGA application. To execute the application itself, refer to *Running OmpSs-2@FPGA Programs*.

Get access to an installed fpga

To run OmpSs-2@FPGA applications, a job needs to be allocated in the FPGA nodes. These nodes are the *main* partition. Therefore, no partition needs to be specified.

For instance, an interactive job looks like this:

```
salloc - c 56 --mem=128G -t 4:00:00 --constraint=dmadma
```

This will allocate a full node in *qdma* mode, which is needed for running OmpSs-2@FPGA applications. The fill node will be allocated and the 8 FPGAs are available to the user.

Information about the FPGAs is stored in `/etc/motd` file. This file specifies board serial number, USB port, PCIe slot, and the network ports used by each FPGA.

For instance:

```
+-----------------------------------------------------------------------------------------
↪-------------------------------------------+
|    +-------+        +-------+        +-------+        +-------+        +-------+       ␣
↪+-------+        +-------+        +-------+    |
|    |swp26 |        |swp25 |        |swp24 |        |swp23 |        |swp22 |       ␣
↪|swp21 |        |swp20 |        |swp19 |    |
|    +-------+        +-------+        +-------+        +-------+        +-------+       ␣
↪+-------+        +-------+        +-------+    |
+--------^---------------^----------------^----------------^----------------^------------
```

(continues on next page)

(continued from previous page)

```
↪--^----------------^---------------^---------+
        |               |                |                |                |        ␣
↪  |                |               |
+--------|--------+-------|--------+-------|--------+-------|--------+-------|--------+----
↪--|--------+-------|-------+-------|--------+
|       v        |       v        |       v        |       v        |       v        |  ␣
↪  v        |       v       |       v        |
|    +-------+    |   +-------+    |    +-------+    |    +-------+    |    +-------+    | ␣
↪+-------+    |    +-------+   |    +-------+    |
|    |       |    |   |       |    |    |       |    |    |       |    |    |       |    | ␣
↪|       |    |    |       |   |    |       |    |
|    |QSFP0 |    |   |QSFP0 |    |    |QSFP0 |    |    |QSFP0 |    |    |QSFP0 |    | ␣
↪|QSFP0 |    |    |QSFP0 |   |    |QSFP0 |    |
|    |       |    |   |       |    |    |       |    |    |       |    |    |       |    | ␣
↪|       |    |    |       |   |    |       |    |
|    +-------+    |   +-------+    |    +-------+    |    +-------+    |    +-------+    | ␣
↪+-------+    |    +-------+   |    +-------+    |
|                |                |                |                |                |  ␣
↪           |               |                |
|    +-------+    |   +-------+    |    +-------+    |    +-------+    |    +-------+    | ␣
↪+-------+    |    +-------+   |    +-------+    |
|    |       |    |   |       |    |    |       |    |    |       |    |    |       |    | ␣
↪|       |    |    |       |   |    |       |    |
|    |QSFP1 | <----> |QSFP1 |    |    |QSFP1 |  <----> |QSFP1 |    |    |QSFP1 | <---->␣
↪|QSFP1 |    |    |QSFP1 | <----> |QSFP1 |    |
|    |       |    |   |       |    |    |       |    |    |       |    |    |       |    | ␣
↪|       |    |    |       |   |    |       |    |
|    +-------+    |   +-------+    |    +-------+    |    +-------+    |    +-------+    | ␣
↪+-------+    |    +-------+   |    +-------+    |
|   onic180s0f0  |  onic179s0f0  |   onic204s0f0  |   onic205s0f0  |   onic26s0f0   | ␣
↪onic25s0f0   |   onic51s0f0  |   onic52s0f0   |
|                |                |                |                |                |  ␣
↪           |               |                |
|      +-+       |      +-+       |      +-+       |      +-+       |      +-+       |  ␣
↪  +-+        |      +-+       |      +-+        |
|      | |       |      | |       |      | |       |      | |       |      | |       |  ␣
↪  | |        |      | |       |      | |        |
|      +-+       |      +-+       |      +-+       |      +-+       |      +-+       |  ␣
↪  +-+        |      +-+       |      +-+        |
|   USB-UART-    |  USB-UART-    |   USB-UART-    |   USB-UART-    |  USB-UART-     | ␣
↪USB-UART-    |   USB-UART-   |   USB-UART-    |
| XFL1ND323BSU   |XFL1Y1BX0JYT   | XFL1E3102VRH   | XFL12GU0UBJA   | XFL1UZW5U0MR  ␣
↪|XFL1G5IYME1R   | XFL12IUWGVDB  |  XFL1D2QP00YZ  |
+----------------+---------------+----------------+----------------+----------------+----
↪-----------+---------------+----------------+

+------------+---------+--------------+----------+---------+-----------------------+-----
↪---+------------+-------------+-------------+
| FPGA Card  | Chassis | FPGA Serial  | PCIe Bus | USBPort | ttyUSBx               |␣
↪QSFP0  | QSFP1      | QDMA onic   | onic IP     |
+------------+---------+--------------+----------+---------+-----------------------+-----
↪---+------------+-------------+-------------+
```

(continues on next page)

(continued from previous page)

```
| fpgan08f01 | 3        | XFL1D2QP00YZ | 34:00.0   | 1        | USB-UART-XFL1D2QP00YZ |␣
↪Switch | fpgan08f02 | onic52s0f0   | 10.0.1.1/24 |
+------------+---------+-------------+---------+---------+----------------------+-----
↪---+------------+-------------+-------------+
| fpgan08f02 | 4        | XFL12IUWGVDB | 33:00.0   | 2        | USB-UART-XFL12IUWGVDB |␣
↪Switch | fpgan08f01 | onic51s0f0   | 10.0.2.1/24 |
+------------+---------+-------------+---------+---------+----------------------+-----
↪---+------------+-------------+-------------+
| fpgan08f03 | 5        | XFL1G5IYME1R | 19:00.0   | 3        | USB-UART-XFL1G5IYME1R |␣
↪Switch | fpgan08f04 | onic25s0f0   | 10.0.3.1/24 |
+------------+---------+-------------+---------+---------+----------------------+-----
↪---+------------+-------------+-------------+
| fpgan08f04 | 6        | XFL1UZW5U0MR | 1a:00.0   | 4        | USB-UART-XFL1UZW5U0MR |␣
↪Switch | fpgan08f03 | onic26s0f0   | 10.0.4.1/24 |
+------------+---------+-------------+---------+---------+----------------------+-----
↪---+------------+-------------+-------------+
| fpgan08f05 | 7        | XFL12GU0UBJA | cd:00.0   | 5        | USB-UART-XFL12GU0UBJA |␣
↪Switch | fpgan08f06 | onic205s0f0 | 10.0.5.1/24 |
+------------+---------+-------------+---------+---------+----------------------+-----
↪---+------------+-------------+-------------+
| fpgan08f06 | 8        | XFL1E3102VRH | cc:00.0   | 6        | USB-UART-XFL1E3102VRH |␣
↪Switch | fpgan08f05 | onic204s0f0 | 10.0.6.1/24 |
+------------+---------+-------------+---------+---------+----------------------+-----
↪---+------------+-------------+-------------+
| fpgan08f07 | 9        | XFL1Y1BX0JYT | b3:00.0   | 7        | USB-UART-XFL1Y1BX0JYT |␣
↪Switch | fpgan08f08 | onic179s0f0 | 10.0.7.1/24 |
+------------+---------+-------------+---------+---------+----------------------+-----
↪---+------------+-------------+-------------+
| fpgan08f08 | 10       | XFL1ND323BSU | b4:00.0   | 8        | USB-UART-XFL1ND323BSU |␣
↪Switch | fpgan08f07 | onic180s0f0 | 10.0.8.1/24 |
+------------+---------+-------------+---------+---------+----------------------+-----
↪---+------------+-------------+-------------+
```

Loading bistreams

The FPGA bitstream needs to be loaded before the application can run. The `load_bitstream` utility is provided in order to simplify programming the FPGA.

```
load_bitstream bitstream.bit [index] ...
```

The utility receives a second optional parameter to indicate which of the allocated FPGAs to program. More than one index can be specified. In such case, all the specified FPGAs will be programmed using the given bitstream.

To know which FPGAs indices have been allocated, run `load_bitstream` with the help (`-h`) option. The output should be similar to this:

```
Usage load_bitstream bitstream.bit [index]
Available devices:
index:  jtag serial     pcie
0:      XFL1D2QP00YZ    34:00.0
1:      XFL12IUWGVDB    33:00.0
2:      XFL1G5IYME1R    19:00.0
```

(continues on next page)

(continued from previous page)

```
3:      XFL1UZW5U0MR    1a:00.0
4:      XFL12GU0UBJA    cd:00.0
5:      XFL1E3102VRH    cc:00.0
6:      XFL1Y1BX0JYT    b3:00.0
7:      XFL1ND323BSU    b4:00.0
```

Get current bitstream info

In order to get information about the bitstream currently loaded into the FPGA, the tool `read_bitinfo` is installed in the system. This tool is available when the `ompss-2` environment module is loaded.

```
read_bitinfo
```

Note that an active slurm reservation is needed in order to query the FPGA.

This call should return something similar to the sample output for a OMPIF test application:

```
Bitinfo of FPGA 0000:cc:00.0:
Bitinfo version:    13
Bitstream user-id:  0x479B8510
AIT version:    7.7.2
Wrapper version 13
Number of acc:  5
Board base frequency (MHz)  100.000000
Interleaving not enabled

Features:
[ ] Instrumentation
[ ] Hardware counter
[x] Performance interconnect
[ ] Simplified interconnection
[x] POM AXI-Lite
[x] POM task creation
[ ] POM dependencies
[ ] POM lock
[x] POM spawn queues
[ ] Power monitor (CMS)
[ ] Thermal monitor (sysmon)
[x] OMPIF

Managed rstn addr 0x10000
Cmd In addr 0xC000 len 128
Cmd Out addr 0xE000 len 128
Spawn In addr 0x8000 len 1024
Spawn Out addr 0xA000 len 1024
Hardware counter not enabled
POM AXI-Lite addr 0x4000
Power monitor (CMS) not enabled
Thermal monitor (sysmon) not enabled

xtasks accelerator config:
type        count   freq(KHz)   description
```

(continues on next page)

(continued from previous page)

```
8381065717  1       100000      send_receive_test
8454279320  1       100000      allgather_test_task
7899490654  1       100000      broadcast_test_task
4294967299  1       100000      ompif_message_sender
4294967300  1       100000      ompif_message_receiver

ait command line:
ait --name=ompif_test --board=alveo_u55c -c=100 --enable_pom_axilite --interconnect_
↪opt=performance --wrapper_version 13

Hardware runtime VLNV:
bsc:ompss:picos_ompss_manager:7.3
```

Running cluster applications

See *Running OMPIF applications*.

6.6 bora user guide

bora takes its name from Borá, which is a small town and municipality located in the state of São Paulo in Brazil.

The OmpSs-2@FPGA releases are automatically installed in the server. They are available through a module file for each target architecture. This document describes how to load and use the modules to compile an example application. Once the modules are loaded, the workflow in the server should be the same as in the Docker images.

6.6.1 General remarks

- The OmpSs-2@FPGA toolchain is installed in a version folder under the `/opt/bsc/` directory.
- Third-party libraries required to run some programs are installed in the corresponding folder under the `/opt/lib/` directory.
- The rest of the software (Xilinx toolchain, modules, etc.) is installed under the `/tools/` directory.

6.6.2 Logging into the system

bora is accessible at IP `84.88.51.137`:

```
ssh 84.88.51.137
```

Also, this can be automated by adding a `bora` host into ssh config:

```
Host bora
    HostName 84.88.51.137
```

6.6.3 Module structure

The ompss-2 modules are:

- `ompss-2/x86_64/*[release version]*`

This will automatically load the default Vivado version, although an arbitrary version can be loaded before ompss:

```
module load vivado/2024.2 ompss-2/x86_64/git
```

To list all available modules in the system run:

```
module avail
```

6.6.4 Build applications

To generate an application binary and bitstream, you could refer to *Compile OmpSs-2@FPGA programs* as the steps are general enough.

Note that the appropriate modules need to be loaded. See *Module structure*.

6.7 ikergune user guide

ikergune takes its name from a technology transfer project with a Basque industrial company.

The OmpSs-2@FPGA releases are automatically installed in the server. They are available through a module file for each target architecture. This document describes how to load and use the modules to compile an example application. Once the modules are loaded, the workflow in the server should be the same as in the Docker images.

6.7.1 General remarks

- The OmpSs-2@FPGA toolchain is installed in a version folder under the `/opt/bsc/` directory.
- Third-party libraries required to run some programs are installed in the corresponding folder under the `/opt/lib/` directory.
- The rest of the software (Xilinx toolchain, modules, etc.) is installed under the `/tools/` directory.

6.7.2 Logging into the system

ikergune is accessible from crdbmaster `crdbmaster.bsc.es` Alternatively, it can be accessed through the `8422` port in crdbmaster and ssh connection will be redirected to the actual host:

```
ssh -p 8422 crdbmaster.bsc.es
```

Also, this can be automated by adding a `ikergune` host into ssh config:

```
Host ikergune
    HostName crdbmaster.bsc.es
    Port 8422
```

6.7.3 Module structure

The ompss-2 modules are:

- `ompss-2/x86_64/*[release version]*`

This will automatically load the default Vivado version, although an arbitrary version can be loaded before ompss:

```
module load vivado/2024.2 ompss-2/x86_64/git
```

To list all available modules in the system run:

```
module avail
```

6.7.4 Build applications

To generate an application binary and bitstream, you could refer to *Compile OmpSs-2@FPGA programs* as the steps are general enough.

Note that the appropriate modules need to be loaded. See *Module structure*.

- genindex

INDEX

A

AIT design optimization, 27
AIT options, 20
AIT placement, 28
AIT user config file, 24
ait_options, 20

B

boot
    xilinx, 39
bora, 64

C

compile
    OmpSs-2@FPGA, 17
crdbmaster, 51

D

develop
    OmpSs-2@FPGA, 10

I

ikergune, 65
install
    toolchain; OmpSs-2@FPGA, 1
installation, 44

L

llebeig, 57
LLVM/Clang FPGA Phase options, 19

M

meep, 58

N

Nanos6 API, 15
Nanos6 FPGA Architecture configuration, 33

O

OmpSs-2@FPGA
    compile, 17
    develop, 10
    running, 31
Ovni FPGA instrumentation, 37

P

POM AXI lite, 37

Q

quar, 45

R

Run multi-node OMPIF/IMP applications, 35
Run single-node applications, 34
running
    OmpSs-2@FPGA, 31

T

toolchain; OmpSs-2@FPGA
    install, 1

X

xaloc, 54
xilinx
    boot, 39