OmpSs@FPGA User Guide

Release 3.3.1

BSC Programming Models

May 27, 2022

CONTENTS

Note: There is a PDF version of this document at http://pm.bsc.es/ftp/ompss-at-fpga/doc/user-guide-3.3.1/OmpSsFPGAUserGuide.pdf

CHAPTER

ONE

INSTALL OMPSS@FPGA TOOLCHAIN

This page should help you install the OmpSs@FPGA toolchain. However, it is preferred using the pre-build Docker image with the latest stable toolchain. They are available at DockerHUB (https://hub.docker.com/r/bscpm/ompss_at_fpga). 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 toolcahin 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 3.5 or later (https://www.python.org/)
- Vendor backends: - Xilinx Vivado 2018.3 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 installation instructions from Xilinx Vivado, VivadoHLS and SDK, as well as the device support for the devices you're working, should be enabled during setup. However, components can be added or removed afterwards.

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-at-fpga-releases. It contains a Makefile that based on some environment variables will compile and install the toolchain. The environment variables are:

- `TARGET` [Def: `aarch64-linux-gnu`] Linux architecture that toolchain will target
- `PREFIX_HOST` [Def: `/`] Installation prefix for the host tools (e.g. mcxx, ait)

- `PREFIX_TARGET` [Def: `/`] Installation prefix for the target tools (e.g. nanox, libxdma)
- `EXTRAE_HOME` [Def:] (Optional) Extrae installation path
- `BUILDCPUS` [Def: `nproc`] Number of processes used for building

The following example will cross-build the toolchain for the *aarch64-linux-gnu* architecture and install it in `/opt/bsc/host-arm64/ompss` and `/opt/bsc/arm64/ompss`:

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

1.3 Individual git repos

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` 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](https://github.com/bsc-pm-ompss-at-fpga/ait)
- [Kernel module](https://github.com/bsc-pm-ompss-at-fpga/ompss-at-fpga-kernel-module)
- [xdma](https://github.com/bsc-pm-ompss-at-fpga/xdma)
- [xtasks](https://github.com/bsc-pm-ompss-at-fpga/xtasks)
- [Nanos++](https://github.com/bsc-pm-ompss-at-fpga/nanox)
- [Mercurium](https://github.com/bsc-pm-ompss-at-fpga/mcxx)

1.3.1 Accelerator Integration Tool (AIT)

To install AIT just clone the repository, run the `<ait>/install.sh` script and add `PREFIX/ait/` to PATH.

```
git clone https://github.com/bsc-pm-ompss-at-fpga/ait
cd ait
git lfs install
git lfs pull
./install.sh PREFIX/ait/ all
export PATH=PREFIX/ait/:$PATH
```

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
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/libxdma` folder:

```
git clone https://github.com/bsc-pm-ompss-at-fpga/xdma
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/libxdma install
```

1.3.4 xTasks

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

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

1.3.5 Nanos++

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

```
git clone https://github.com/bsc-pm-ompss-at-fpga/nanox
cd nanox
autoreconf -ifv
mkdir build-fpga-arm64
cd build-fpga-arm64
../configure --prefix=/opt/bsc/arm64/ompss/nanox --host=aarch64-linux-gnu host_
↪alias=aarch64-linux-gnu --with-xtasks=/opt/bsc/arm64/ompss/libxtasks [--with-
↪extrae=/opt/bsc/arm64/ompss/extrae]
make
make install
```

1.3.6 Nanos++ with cluster support

Example to cross-compile the runtime library, with cluster architecture enabled, and install it in the `/opt/bsc/arm64/ompss/nanox` folder. The example assumes that mpich and GASNet are already compiled and installed.

```
git clone https://github.com/bsc-pm-ompss-at-fpga/nanox
cd nanox
autoreconf -ifv
mkdir build-fpga-cluster-arm64
cd build-fpga-cluster-arm64
export PATH=$PATH:/opt/bsc/arm64/ompss/mpich/bin
../configure --prefix=/opt/bsc/arm64/ompss/nanox --host=aarch64-linux-gnu host_
↪alias=aarch64-linux-gnu --with-xtasks=/opt/bsc/arm64/ompss/libxtasks --with-gasnet=/
↪opt/bsc/arm64/ompss/gasnet --with-mpi=/opt/bsc/arm64/ompss/mpich LDFLAGS="-Wl,--
↪allow-shlib-undefined" MPICC=mpicc MPICXX=mpicxx CC=aarch64-linux-gnu-gcc␣
↪CXX=aarch64-linux-gnu-g++
make
make install
```

1.3.7 Mercurium

Example to build a Mercurium 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/mcxx
cd mcxx
autoreconf -ifv
mkdir build-fpga
cd build-fpga
../configure --prefix=/opt/bsc/host-arm64/ompss/mcxx-arm64 --with-nanox=/opt/bsc/
↪arm64/ompss/nanox --enable-ompss --enable-tl-openmp-nanox --target=aarch64-linux-
↪gnu target_alias=aarch64-linux-gnu
make
make install
```

CHAPTER

TWO

DEVELOP OMPSS@FPGA PROGRAMS

Most of the required information to develop an OmpSs@FPGA application should be in the general OmpSs documentation (https://pm.bsc.es/ompss-docs/book/index.html). Note that, there may be some unsupported/not-working OmpSs features and/or syntax when using FPGA tasks. If you have some problem or realize any bug, do not hesitate to contact us or open an issue.

To create an FPGA task you need to add the `target` directive before the `task` directive. For example:

```
const unsigned int LEN = 8;

#pragma omp target device(fpga)
#pragma omp task out([LEN]dst, const char 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@FPGA application:

- Only C/C++ are supported, not Fortran.
- Only function declarations can be annotated as FPGA tasks.
- Avoid using global variables which are not listed in the dependences/copies. They can be used through function arguments.
- The macros cannot be used within #pragmas as explained in OmpSs user guide (https://pm.bsc.es/ftp/ompss/doc/user-guide/faq-macros.html).
- The HLS source code generated by Mercurim for each FPGA task will not contain the includes in the original source file but the ones finished in .fpga.h or .fpga .
- The FPGA task code cannot perform general system calls, and only some Nanos++ 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 Clauses of target directive

The following sections list the clauses that can be used in the `target` directive.

2.2.1 num_instances

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

```
const unsigned int LEN = 8;

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

2.2.2 onto

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 `onto(0)` it can only run in accelerators that are of *type 0*. The value provided in this clause will overwrite the value automatically generated by Merciurim (a hash based on the source file and function name) to match the tasks. Usage example:

```
const unsigned int LEN = 8;

#pragma omp target device(fpga) onto(100)
#pragma omp task out([LEN]dst)
void memset_char(char * dst, const char val) {
  for (unsigned int i=0; i<LEN; ++i) {
    dst[i] = val;
  }
}

#pragma omp target device(fpga) onto(101)
#pragma omp task out([LEN]dst)
void memset_float(float * dst, const float val) {
  for (unsigned int i=0; i<LEN; ++i) {
    dst[i] = val;
  }
}
```

2.2.3 localmem

Defines the memory regions that the FPGA task wrapper must catch in BRAMs. 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. If the parameter is not labeled with the `const` modifier, the wrapper includes support for writing back the local copy into memory after the task execution. 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 omp target device(fpga) localmem([LEN]dst)
#pragma omp task out([LEN]dst)
void memset(char * dst, const char val) {
  for (unsigned int i=0; i<LEN; ++i) {
```

(continues on next page)

(continued from previous page)

```
    dst[i] = val;
  }
}
```

2.2.4 localmem_copies

Promote the task copies like they were annotated into the `localmem` clause. This clause is enabled by default, unless the `localmem` clause is present.

2.2.5 no_localmem_copies

Do not promote the task copies into the `localmem` clause.

2.2.6 period

Defines the task period in microseconds. The usage of this clause makes the task a recurrent task that is executed (at most) every period microseconds. Usage example where a task is executed every second:

```
const unsigned int LEN = 8;

#pragma omp target device(fpga) period(1000000)
#pragma omp task
void memset(char * dst, const char val) {
  for (unsigned int i=0; i<LEN; ++i) {
    dst[i] = val;
  }
}
```

2.2.7 num_repetitions

Defines the number of repetitions that a recurrent task has to be executed before it becomes finished. The usage of this clause makes the task a recurrent task that is executed N times. Usage example where the task body is executed 100 times:

```
const unsigned int LEN = 8;

#pragma omp target device(fpga) num_repetitions(100)
#pragma omp task
void memset(char * dst, const char val) {
  for (unsigned int i=0; i<LEN; ++i) {
    dst[i] = val;
  }
}
```

2.3 Calls to Nanos++ API

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

2.3.1 Nanos++ Instrumentation API

The following sections list and summarize the Nanos++ Instrumentation API relevant for the FPGA tasks.

nanos_instrument_register_key_with_key

Register an event key.

Arguments:

- **event_key**: Integer event identifier. It should be the same passed to calls inside the FPGA task. The event key can be any positive integer value greater then or equal to 1000. Events in the 0-999 range are reserved.
- **key**: An string identifying the event or value, it is used by the nanos instrumentation API to reference an event or a value.
- **description**: The description string that will be visualized in a trace.
- **abort_when_registered**: Indicates if the runtime should abort when registering an event or value with a key that has been already registered,

Return value:

- NANOS_OK on success, NANOS_ERROR on error.

```
nanos_err_t nanos_instrument_register_key_with_key(
    nanos_event_key_t event_key,
    const char* key,
    const char* description,
    bool abort_when registered
);
```

nanos_instrument_register_value_with_val

Register a value Registering a value is optional. Usually it's only useful if the event has an enumerated value, such a set of states.

Arguments:

- **val**: Integer value. It should be the same passed to calls inside the FPGA task.
- **key**: String identifier for the event used in `nanos_instrument_register_key_with_key` call.
- **value**: An string identifying the event value to be registered.
- **description**: The description string that will be visualized in a trace.
- **abort_when_registered**: Indicates if the runtime should abort when registering an event or value with a key that has been already registered,

Return value:

- NANOS_OK on success, NANOS_ERROR on error.

```
nanos_err_t nanos_instrument_register_value_with_val(
    nanos_event_value_t val,
    const char* key,
    const char *value,
    const char* description,
    bool abort_when registered
);
```

nanos_instrument_burst_begin

This Nanos++ API can be called inside an FPGA task.

```
nanos_err_t nanos_instrument_burst_begin(
    nanos_event_key_t event,
    nanos_event_value_t value
);
```

nanos_instrument_burst_end

This Nanos++ API can be called inside an FPGA task.

```
nanos_err_t nanos_instrument_burst_end(
    nanos_event_key_t event,
    nanos_event_value_t value
);
```

nanos_instrument_point_event

This Nanos++ API can be called inside an FPGA task.

```
nanos_err_t nanos_instrument_point_event(
    nanos_event_key_t event,
    nanos_event_value_t value
);
```

Full example

```
const unsigned int BSIZE = 256;
const nanos_event_key_t DOT_COMPUTATION = 1000;
const nanos_event_key_t DOT_ITERATION = 1001;

#pragma omp target device(fpga)
#pragma omp task in([BSIZE]v1, [BSIZE]v2) inout([1]result)
void dotProduct(float *v1, float *v2, float *result) {
  nanos_instrument_burst_begin(DOT_COMPUTATION, 1);
  int resultLocal = result[0];
  for (unsigned int i = 0; i < BSIZE; ++i) {
    nanos_instrument_point_event(DOT_ITERATION, i);
    resultLocal += v1[i]*v2[i];
  }
  result[0] = resultLocal;
  nanos_instrument_burst_end(DOT_COMPUTATION, 1);
}

int main() {
  ...
  //register fpga events
  nanos_instrument_register_key_with_key(DOT_COMPUTATION, "dotProduct_computation",
↪"dot product computation", true);
  nanos_instrument_register_key_with_key(DOT_ITERATION, "dotProduct_iteration", "dot␣
↪product main loop iteration", true);
```

(continues on next page)

(continued from previous page)

```
  for (unsigned int i = 0; i < vecSize; i += BSIZE) {
    dotProduct(v1+i, v2+i, &result);
  }
  #pragma omp taskwait
  ...
}
```

2.3.2 Nanos++ FPGA Architecture API

The following sections list and summarize the Nanos++ FPGA Architecture API. The documentation is for the version 10.

Memory Management

nanos_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:

- **len**: Length in bytes to allocate.

Return value:

- Pointer to the allocated region in the FPGA address space.

```
void * nanos_fpga_malloc(
    size_t len
);
```

nanos_fpga_free

```
void nanos_fpga_free(
    void * fpgaPtr
);
```

nanos_fpga_memcpy

```
typedef enum {
    NANOS_COPY_HOST_TO_FPGA,
    NANOS_COPY_FPGA_TO_HOST
} nanos_fpga_memcpy_kind_t;

void nanos_fpga_memcpy(
    void * fpgaPtr,
    void * hostPtr,
    size_t len,
    nanos_fpga_memcpy_kind_t kind
);
```

Periodic tasks

nanos_get_periodic_task_repetition_num

This Nanos++ API can be called inside an FPGA task.

Returns the current repetition number inside a periodic task. First execution in the repetition 1. It will return a 0 if called outside a periodic task.

```
unsigned int nanos_get_periodic_task_repetition_num();
```

nanos_cancel_periodic_task

This Nanos++ API can be called inside an FPGA task.

Aborts the remaining repetitions of a periodic task and finishes it at the end of task code.

```
void nanos_cancel_periodic_task();
```

Time information

nanos_fpga_get_time_cycle

This Nanos++ API only can be called inside an FPGA task.

Returns the current timestamp since last reset in FPGA task accelerator cycles.

```
unsigned long long int nanos_fpga_get_time_cycle();
```

nanos_fpga_get_time_us

This Nanos++ API only can be called inside an FPGA task.

Returns the current timestamp since last reset in microseconds.

```
unsigned long long int nanos_fpga_get_time_us();
```

Data copies

These Nanos++ API only can be called inside an FPGA task. They alloy 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 *fpga_memory_port_width* option is mandatory.

Data accessed through this functions, has to be **aligned to the port width**. If alignment cannot be guaranteed *fpga_unaligned_memory_port* option is needed. Otherwise behaviour is undefined.

Also, data should to be **multiple of the port width**. If this cannot be guaranteed, *fpga_check_limits_memory_port* option is needed so that no out of bounds data is accessed. Otherwise this will result in undefined behaviour.

nanos_fpga_memcpy_wideport_in

```
void nanos_fpga_memcpy_wideport_in(T* dest, const unsigned long long int src, const␣
↪unsigned int elems)
```

Arguments:

- `dest`: Pointer to the destination (local) data. It can be any data type.
- `src`: FPGA memory address space where the data is stored.
- `elems`: Number of elements of type `T` to be copied.

nanos_fpga_memcpy_wideport_out

```
void nanos_fpga_memcpy_wideport_out(const unsigned long long int dest, T src, const␣
↪unsigned int elems)
```

Arguments:

- `dest`: FPGA memory address where the data is going to be copied to.
- `src`: Pointer to the *source* (local) data. It can be any data type.
- `elems`: Number of elements of type `T` to be copied.

CHAPTER

THREE

COMPILE OMPSS@FPGA PROGRAMS

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

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

3.1 Mercurium FPGA Phase options

The following sections list and summarize the Mercurium options from the FPGA Phase.

3.1.1 force_fpga_periodic_support

Force enabling the periodic tasks support in all FPGA task accelerators. This feature is only enabled in the FPGA task accelerators that require it (the target directive has the period or num_repetitions caluses). Usage example:

```
fpgacc --ompss --bitstream-generation src/dotproduct.c -o dotproduct \
  --Wf,--board=zedboard --variable=force_fpga_periodic_support:1
```

3.1.2 fpga_check_limits_memory_port

[Available in release 2.5.0] Controls whether the limits of local variables should be enforced when fulfilled using the shared memory port. This means, checking that data is not wrote/read behind the local variable limits. This happens when the variable width is not multiple of shared memory port width. By default the check is enabled. Usage example to disable the check:

```
fpgacc --ompss --bitstream-generation src/dotproduct.c -o dotproduct --Wf,--
↪board=zedboard \
  --variable=fpga_memory_port_width:128 --variable=fpga_check_limits_memory_port:0
```

Warning: This option can reduce the resources usage, but the application must ensure that the localmem variables have widths multiple of the shared memory port width.

3.1.3 fpga_directive_data_pack

[Available in release 3.0.0] Controls whether the DATA_PACK directive is placed in the HLS wrapper for struct/class parameters. It is enabled by default (the HLS generation remains equal to previous releases). Usage example to disable the support:

```
fpgacc --ompss --bitstream-generation src/dotproduct.c -o dotproduct \
  --Wf,--board=zedboard --variable=fpga_directive_data_pack:0
```

3.1.4 fpga_ignore_deps_task_spawn

Ignore the data dependences when spawning a task inside a FPGA task accelerator. This creates the task like it was annotated without data dependences. Note that this only affects the task spawn inside a FPGA task accelerator, not other devices which may spawn the task task. Usage example:

```
fpgacc --ompss --bitstream-generation src/dotproduct.c -o dotproduct \
  --Wf,--board=zedboard --variable=fpga_ignore_deps_task_spawn:1
```

3.1.5 fpga_memory_port_width

Defines the width (in bits) of memory ports (only for wrapper localmem data) for FPGA task accelerators. Usage example:

```
fpgacc --ompss --bitstream-generation src/dotproduct.c -o dotproduct \
  --Wf,--board=zedboard --variable=fpga_memory_port_width:128
```

It assumes that all task arguments read through the shared wide port are aligned to the port width. Then, the first element of all arguments with a localmem is aligned to the shared memory port width. For example, a shared memory port with a width of 512 bits will require an alignment of 64 bytes. This alignment has to be enforced in all FPGA task levels. The alignment of data copies managed by Nanos++ can be modified with the `fpga-alloc-align` option (see nanox-fpga-help for more information). Alternatively, the Mercurium option `fpga_unaligned_memory_port` adds support for unaligned arguments but with a higher FPGA resource consumption.

3.1.6 fpga_memory_ports_mode

[Available in release 2.5.0] Changes the creation strategy of memory ports between `dedicated` (default) and `type`. The `dedicated` mode creates a port for each task parameter. The `type` mode creates a port for each parameter data type. Usage example:

```
fpgacc --ompss --bitstream-generation src/dotproduct.c -o dotproduct \
  --Wf,--board=zedboard  --variable=fpga_memory_ports_mode:type
```

3.1.7 fpga_unaligned_memory_port

[Available in release 2.3.0] Enables the logic to support unaligned memory regions handled by the shared memory port. This option only has effect when the fpga_memory_port_width option is also present. Usage example:

```
fpgacc --ompss --bitstream-generation src/dotproduct.c -o dotproduct \
  --Wf,--board=zedboard  --variable=fpga_memory_port_width:128 \
  --variable=fpga_unaligned_memory_port:1
```

Warning: This option will increase the resources consumption of FPGA wrappers. However, it may be mandatory depending on application data partition.

3.1.8 fpga_unordered_args

[Available in release 2.5.0] Controls whether the support for handling the task argument out of order must be implemented or not. It is enabled by default up to release 2.5.2 (the HLS generation remains equal to previous releases). It is disabled by default since release 3.0.0. Usage example to disable the support:

```
fpgacc --ompss --bitstream-generation src/dotproduct.c -o dotproduct \
  --Wf,--board=zedboard --variable=fpga_unordered_args:0
```

3.2 AIT options

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 board model. Supported boards by vendor: xilinx: alveo_u200, alveo_u250, axiom, com_express, euroexa_maxilink, euroexa_maxilink_quad, zcu102, zedboard, zybo, zynq702, zynq706

-n NAME, --name NAME project name

Generation flow:

-d DIR, --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 CLOCK FPGA clock frequency in MHz (def: ‘100’)

--hwruntime HWRUNTIME add a hardware runtime. Available hardware runtimes by vendor: xilinx: pom, som (def: som)

--hwcounter add a hardware counter to the bitstream

--wrapper_version WRAPPER_VERSION version of accelerator wrapper shell. This information will be placed in the bitstream information

--datainterfaces_map DATAINTERFACES_MAP path of mappings file for the data interfaces

--placement_file PLACEMENT_FILE json file specifying accelerator placement

--floorplanning_constr FLOORPLANNING_CONSTR built-in floorplanning constraints for accelerators and static logic acc: accelerator kernels are constrained to a SLR region static: each static logic IP is constrained to its relevant SLR all: enables both 'acc' and 'static' options By default no floorplanning constraints are used

--slr_slices SLR_SLICES enable SLR crossing register slices acc: create register slices for SLR crossing on accelerator-related interfaces static: create register slices for static logic IPs all: enable both 'acc' and 'static' options By default they are disabled

--memory_interleaving_stride MEM_INTERLEAVING_STRIDE size in bytes of the stride of the memory interleaving. By default there is no interleaving

--bitinfo_note BITINFO_NOTE custom note to add to the bitInfo

User-defined files:

--user_constraints USER_CONSTRAINTS path of user defined constraints file

--user_pre_design USER_PRE_DESIGN path of user TCL script to be executed before the design step (not after the board base design)

--user_post_design USER_POST_DESIGN path of user TCL script to be executed after the design step

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_accs)

--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_accs)

--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')

--disable_spawn_queues disable the hwruntime spawn in/out queues

Picos:

--picos_max_args_per_task PICOS_MAX_ARGS_PER_TASK maximum number of arguments for any task in the bitstream (def: '15')

--picos_max_deps_per_task PICOS_MAX_DEPS_PER_TASK maximum number of dependencies for any task in the bitstream (def: '8')

--picos_max_copies_per_task PICOS_MAX_COPIES_PER_TASK maximum number of copies for any task in the bitstream (def: '15')

--picos_num_dcts NUM_DCTS number of DCTs instantiated (def: '1')

--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')

Miscellaneous:

-h, --help show this help message and exit

-i, --verbose_info print extra information messages

-k, --keep_files keep files on error

-v, --verbose print vendor backend messages

--version print AIT version and exits

Xilinx-specific arguments:

--debug_intfs INTF_TYPE choose which interfaces mark for debug and instantiate the correspondent ILA cores AXI: debug accelerator's AXI interfaces stream: debug accelerator's AXI-Stream interfaces both: debug both accelerator's AXI and AXI-Stream interfaces custom: debug user-defined interfaces none: do not mark for debug any interface (def: 'none')

--debug_intfs_list DEBUG_INTFS_LIST path of file with the list of interfaces to debug

--ignore_eng_sample ignore engineering sample status from chip part number

--interconnect_opt OPT_STRATEGY AXI interconnect optimization strategy: Minimize 'area' or maximize 'performance' (def: 'area')

–interconnect_regslice INTER_REGSLICE_LIST [INTER_REGSLICE_LIST . . .] enable register slices on AXI interconnects all: enables them on all interconnects mem: enables them on interconnects in memory datapath hwruntime: enables them on the AXI-stream interconnects between the hwruntime and the accelerators

-j JOBS, --jobs JOBS specify the number of Vivado jobs to run simultaneously By default it will use as many jobs as cores with at least 3GB of dedicated free memory, or the value returned by *nproc*, whichever is less.

--target_language TARGET_LANG choose target language to synthesize files to: VHDL or Verilog (def: 'VHDL')

--simplify_interconnection simplify interconnection between accelerators and memory. Might negatively impact timing

environment variables: PETALINUX_INSTALL path where Petalinux is installed PETALINUX_BUILD path where the Petalinux project is located

3.2.2 Accelerator placement options

This section documents how to constrain accelerators to a particular SLR region in a device.

There are three flags that control accelerator placement:

- Constraints: `--floorplanning_constr`
- Slices: `--slr_slices`
- Configuration file `--placement_file`

VU9 or Alveo U200 have 3 Super logic regions, external interfaces are placed as follows:

By default, all user accelerators are placed as vivado considers. Sometimes it places a kernel accelerator between 2 SLR, 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 SLR. For instance, a user can specify something as follows:

Additionally, users can apply register slices between the SLR crossings to further help timing at the cost of using additional fpga resources. Users can control this by setting different settings for constraints and register slices.

User flags

Constraints

Constraints affecting different sets of IPs can be individually enabled. This is done by setting the `--floorplanning_constr=<constraint level>` flag. This can take four different values: *[none]*, *acc*, *static*, *all*.

These are specified as follows:

[none]

Nothing is constrained to a particular region. This is the default behavior.

This is done by not specifying the `--floorplanning_constr`

acc

Accelerator kernels are constrained to be in a slr region.

static

Static logic is constrained to a particular region. Each of the static logic IP is constrained to its relevant region. For instance PCI IP is going to be constrained to the slr that contains it IO pins, which is SLR 1 in the case of the U200.

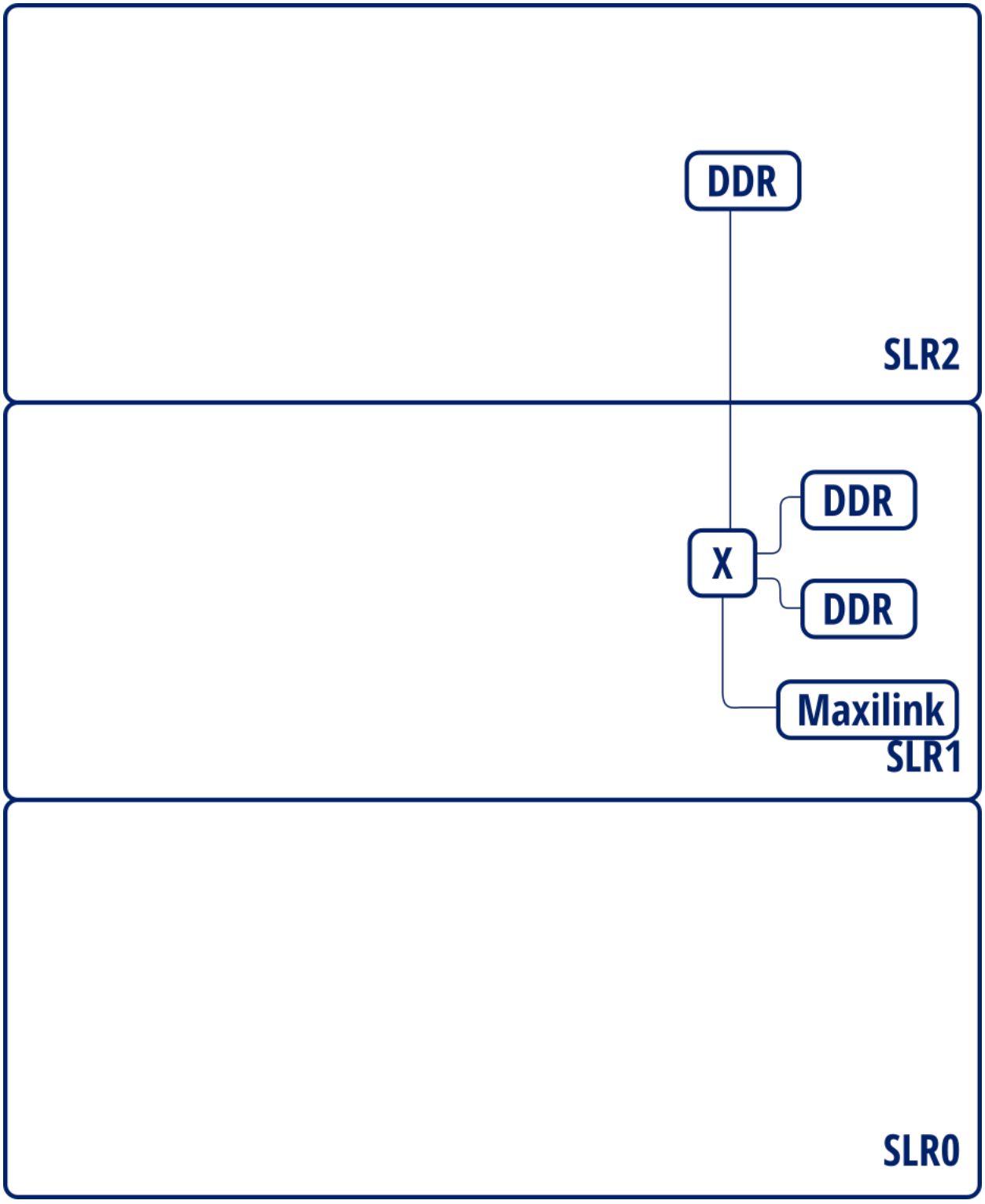

Fig. 1: Interface layout for Alveo U200 / VU9P

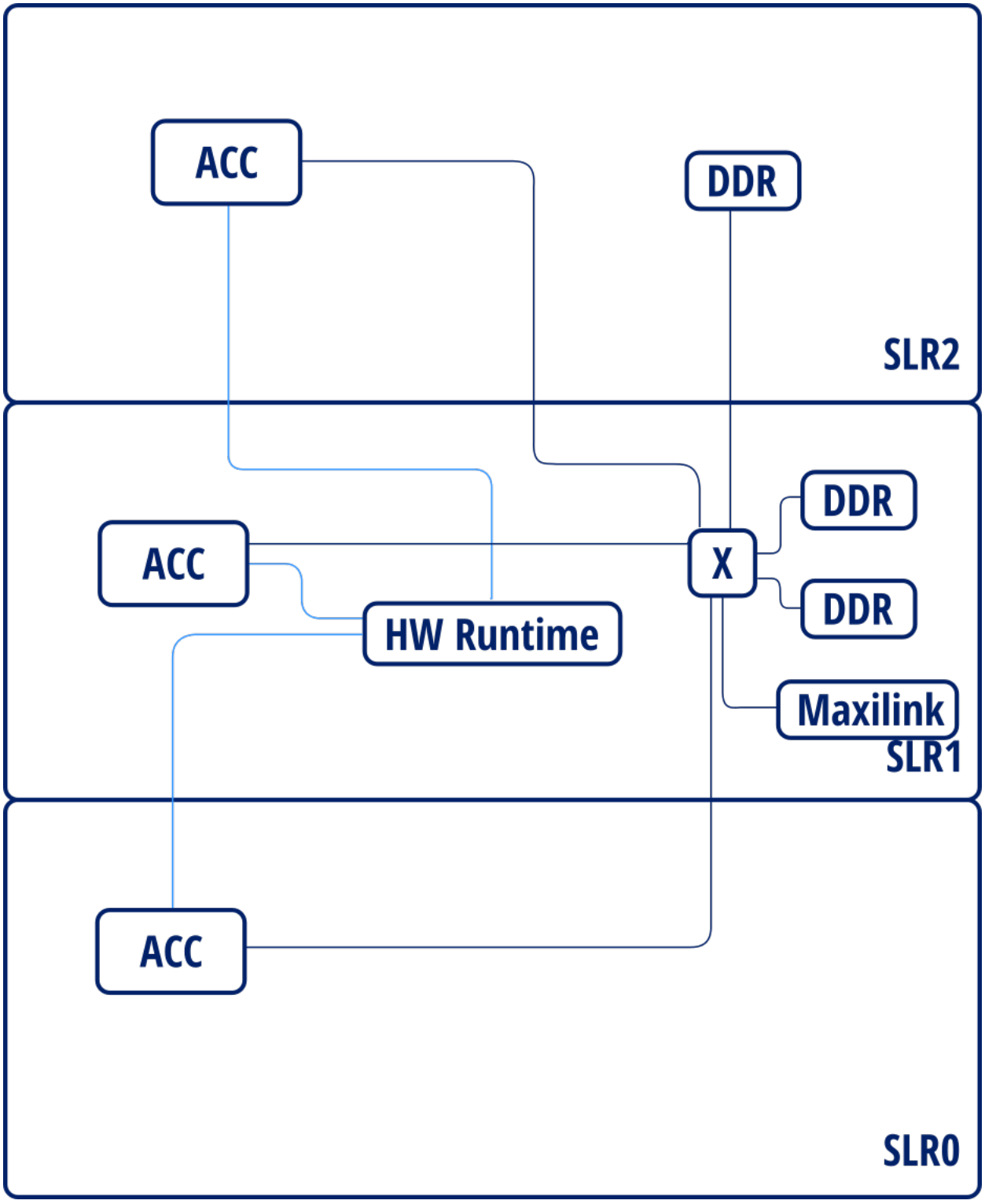

Fig. 2: Placed instance diagram

all

Enables *acc* and *static*

Slices

Slices can be automatically placed in SLR crossings to improve timing. `--slr_slices` flag controls the settings. It can take four different values: *[none]*, *acc*, *static*, *all*.

[none]

No register slices are created for slr crossing, this is the default behaviour.

This is achieved by omitting `--slr_slices` flag.

acc

Register slices for SLR crossing are created for accelerator related interfaces: - Accelerator - hw runtime - Accelerator - DDR interconnect

static

Register slices are created for static logic (DDR MIGs, PCI, communication infrastructure, etc.).

all

Enables both *acc* and *static*.

Configuration file

Configuration file is a json file that determines the placement of each accelerator instance. It's specified using the `--placement_file` option. It should contain a dictionary of accelerator types.r Each accelerator type must contain a list of SLR numbers, one for each instance, indicating where the accelerator is going to be placed. For instance:

```
{
    "calculate_forces_BLOCK" : [0, 0, 1, 2, 2],
    "solve_nbody_task": [1],
    "update_particles_BLOCK": [1]
}
```

This constrains 2 of the 4 `calculate_forces_BLOCK` accelerators to be in SLR0, one of them in SLR1 and the remaining 2 in SLR2. Also, `solve_nbody_task` and `update_particles_BLOCK` will be placed in SLR1.

3.2.3 Accelerator interconnect options

Simplified interconnect

By default, memory interconnection is implemented as 2 interconnection stages:

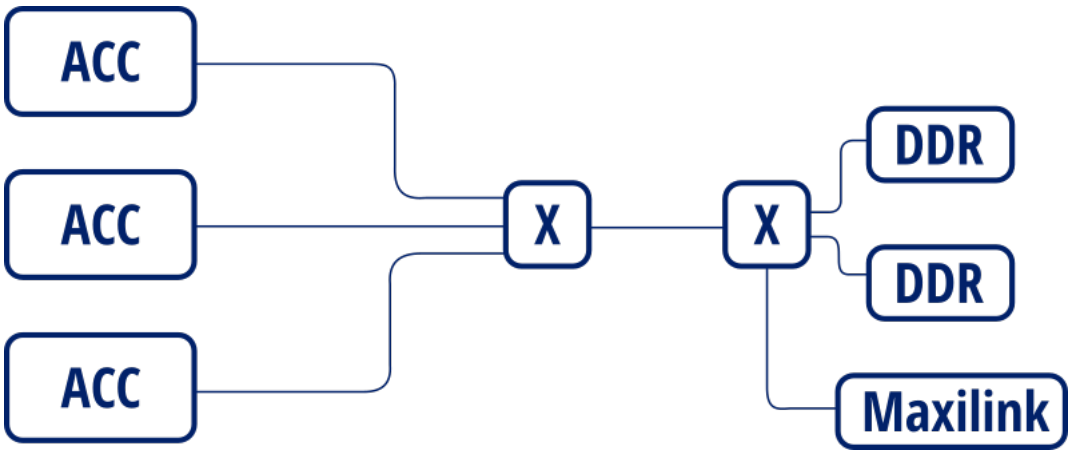

Fig. 3: 2 stage insterconnection

This is done in order to save resources in the case that there's data access ports. However, this serializes data accesses. This prevents accelerators from accessing different memory banks in parallel.

By setting the –simplify_interconnection will result in the following:

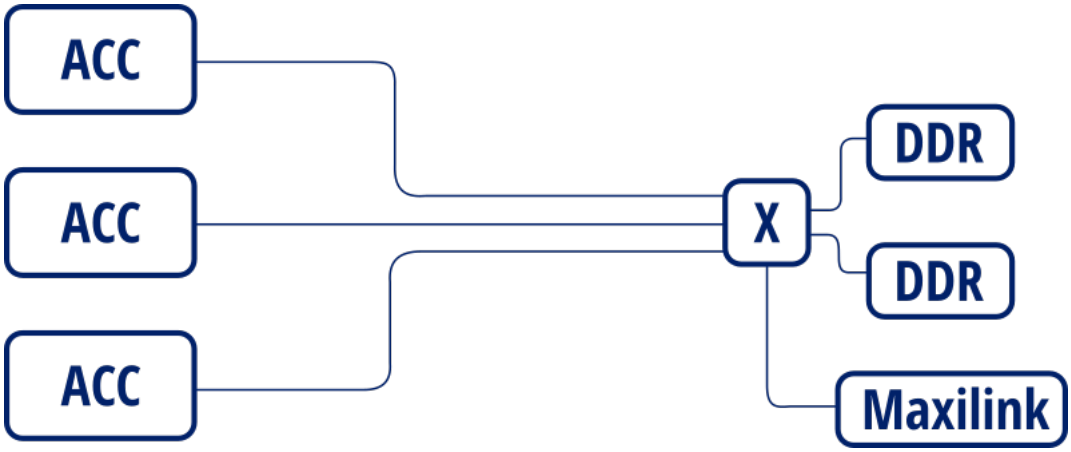

Fig. 4: Simplified interconnection

When also setting `--interconnect_opt=performance` can allow accelerators to concurrently access different banks, effectively increasing overall available bandwidth. Otherwise, accesses will not be performed in parallel as the interconnect is configured in "area" mode.

However, the downside is that this can affect timing and resource usage when this interconnection mode is enabled.

Memory access interleave

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 *Simplified interconnect* and *Interconnect optimization strategy* options.

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. This is specially relevant when using *Simplified interconnect*.

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

Interconnect register slices

By specifying `--interconnect_regslice=<interconnect group>` option, it enables *outer and auto* register slice mode on selected 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.

Interconnect groups are defined as follows:

- `all`: enables them on all interconnects.
- `mem`: enables them on interconnects in memory data path (accelerator - DDR)
- `hwruntime`: enables them on the AXI-stream interconnects between the hwruntime and the accelerators (accelerator control)

Interface debug

Interfaces can be set up for debugging through ILA cores. By setting debugging options, different buses will be set up for debugging and the corresponding ILA cores are generated as needed.

There are two modes to set up debugging, By enabling debug in interface group through `--debug_intf=<interface group>` or selecting individual interfaces using `--debug_intf_list=<interface list files>`.

Interface group selection

Interfaces can be marked for debug in different groups specified in the `--debug_intf=<interface group>`:

- `AXI`: Debug accelerator's AXI 4 memory mapped data interfaces interfaces
- `stream`: Debug accelerator's AXI-Stream control interfaces
- `both`: Debug both `AXI` and `stream` interface groups.
- `custom`: Debug user-defined interfaces
- `none`: Do not mark for debug any interface (this is the default behaviour)

Interface list

Al list of interfaces can be specified in order to enable individual interfaces through the `--debug_intf_list=<interface list files>` option.

Interface list contains a list of interface paths, one for each line. Interface paths are block design connection paths. Ait creates an interface list with all accelerator data interfaces named `<project name>.datainterfaces.txt`. First column is the slave end (origin) of the connection and second column specifies the master (destination) end.

Accelerator data interfaces are specified as

```
<accelerator>_<0>/<accelerator>_ompss/<interface name>
```

For instance to debug interface `x` and `y` from accelerator `foo` interface list should look as follows:

```
/foo_0/foo_ompss/m_axi_mcxx_x
/foo_0/foo_ompss/m_axi_mcxx_y
```

3.3 Binaries

There are two specific Mercurim front-ends for the FPGA devices:

- `fpgacxx` for C++ applications.
- `fpgacc` for C applications.

3.4 Bitstream

Note: Mercurium expects the Accelerator Integration Tool (AIT, formerly autoVivado) to be available on the PATH, if not the linker will fail. Moreover, AIT expects VivadoHLS and Vivado to be available in the PATH.

Warning: Sourcing the Vivado `settings.sh` file may break the cross-compilation toolchain. Instead, just add the directory of vivado binaries in the PATH.

To generate the bitstream, you should enable the bitstream generation in the Mercurium compiler (using the –bitstream-generation flag) and provide it the FPGA linker (aka AIT) flags with `--Wf` option. If the FPGA linker flags does not contain the `-b` (or `--board`) and `-n` (or `--name`) options, Mercurim will not launch AIT.

For example, to compile the dotproduct application, in debug mode, for the Zedboard, with a target frequency of 100Mhz, you can use the following command:

```
arm-linux-gnueabihf-fpgacc --debug --ompss --bitstream-generation \
  src/dotproduct.c -o dotproduct-d \
  --Wf,"--board=zedboard,--clock=100,--name=dotproduct,--hwruntime=som"
```

3.4.1 HW Instrumentation

You can use the `--instrument` (or `--instrumentation`) option of Merciurim to enable the HW instrumentation generation. The instrumentation can be generated and not used when running the application, but if you generate

the bitstream without instrumentation support you will not be able to instrument the executions in the FPGA accelerators. Note that the application binary also has to be compiled with the `--instrument` (or `--instrumentation`) option.

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

```
arm-linux-gnueabihf-fpgacc --instrument --ompss --bitstream-generation \
  src/dotproduct.c -o dotproduct-d \
  --Wf,"--board=zedboard,--clock=100,--name=dotproduct,--hwruntime=som"
```

3.4.2 Shared memory port

By default, Mercurium 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 `fpga_memory_port_width` option of Merciurim 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. Also, values above 128 may require properly setting the alignment in FPGA allocator (see option `fpga-alloc-align` in nanox-fpga-help).

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

```
arm-linux-gnueabihf-fpgacc --ompss --bitstream-generation \
  src/dotproduct.c -o dotproduct-d \
  --variable=fpga_memory_port_width:128 \
  --Wf,"--board=zedboard,--clock=100,--name=dotproduct,--hwruntime=som"
```

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. This step is not needed for the z7000 family of devices as the bitstream can be loaded after boot. 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_INSTALL`. Petalinux installation directory.
- `PETALINUX_BUILD`. Petalinux project directory. See *Create boot files for ultrascale* to have more information about how to setup a petalinux project build.

Then you can invoke AIT with the same options provided in `--Wf` and the following new options: `--from_step=boot --to_step=boot`. Also, you may directly add the `--to_step=boot` option in `--Wf` during the Mercurium launch.

CHAPTER

FOUR

RUNNING OMPSS@FPGA PROGRAMS

To run an OmpSs@FPGA program you should follow the general OmpSs run procedure. More information is provided in the OmpSs User Guide (https://pm.bsc.es/ftp/ompss/doc/user-guide/run-programs.html).

4.1 Nanos++ FPGA Architecture options

The Nanos++ behavior can be tuned with different environment options. They are summarized and briefly described in the Nanos++ help (`nanox --help`). The FPGA architecture options are also available at:

4.1.1 Nanos++ FPGA Architecture options

The Nanos++ behavior can be tuned with different environment options. They are summarized and briefly described in the Nanos++ help, the FPGA architecture section is shown below:

```
FPGA specific options
NX_ARGS options
   --fpga-alloc-align [=]<integer + suffix>
     FPGA allocation alignment (def: 16)
   --fpga-alloc-pool-size [=]<integer + suffix>
     FPGA device memory pool size (def: 512MB)
   --fpga-create-callback --no-fpga-create-callback
     Register the task creation callback during the plugin initialization (def: false␣
↪- automatically enabled when needed)
   --fpga-create-callback-disable --no-fpga-create-callback-disable
     Disable the registration of the task creation callback to handle task creation␣
↪from the FPGA (def: false)
   --fpga-disable --no-fpga-disable
     Disable the support for FPGA accelerators and allocator
   --fpga-enable --no-fpga-enable
     Enable the support for FPGA accelerators and allocator
   --fpga-finish-task-burst [=]<integer>
     Max number of tasks to be finalized in a burst when limit is reached (def: 8)
   --fpga-helper-threads [=]<integer>
     Defines de number of helper threads managing fpga accelerators (def: 1)
   --fpga-hybrid-worker --no-fpga-hybrid-worker
     Allow FPGA helper thread to run smp tasks (def: enabled)
   --fpga-idle-callback --no-fpga-idle-callback
     Perform fpga operations using the IDLE event callback of Event Dispatcher (def:␣
↪enabled)
   --fpga-instrumentation-buffer-size [=]<integer + suffix>
     Maximum number of events to be saved from a FPGA task (def: 170)
```

(continues on next page)

(continued from previous page)

```
   --fpga-instrumentation-callback --no-fpga-instrumentation-callback
     Handle the FPGA instrumentation using the IDLE event callback of Event␣
↪Dispatcher (def: enabled)
   --fpga-max-pending-tasks [=]<integer>
     Number of tasks allowed to be pending finalization for an fpga accelerator (def:␣
↪4)
   --fpga-max-threads-callback [=]<integer>
     Max. number of threads concurrently working in the FPGA IDLE callback (def: 1)
   --fpga-num [=]<integer>
     Defines de number of FPGA acceleratos to use (def: #accels from libxtasks)
Environment variables
   NX_FPGA_ALLOC_ALIGN = <integer + suffix>
     FPGA allocation alignment (def: 16)
   NX_FPGA_ALLOC_POOL_SIZE = <integer + suffix>
     FPGA device memory pool size (def: 512MB)
   NX_FPGA_DISABLE = yes/no
     Disable the support for FPGA accelerators and allocator
   NX_FPGA_ENABLE = yes/no
     Enable the support for FPGA accelerators and allocator
   NX_FPGA_FINISH_TASK_BURST = <integer>
     Max number of tasks to be finalized in a burst when limit is reached (def: 8)
   NX_FPGA_HELPER_THREADS = <integer>
     Defines de number of helper threads managing fpga accelerators (def: 1)
   NX_FPGA_HYBRID_WORKER = yes/no
     Allow FPGA helper thread to run smp tasks (def: enabled)
   NX_FPGA_INSTRUMENTATION_BUFFER_SIZE = <integer + suffix>
     Maximum number of events to be saved from a FPGA task (def: 170)
   NX_FPGA_MAX_PENDING_TASKS = <integer>
     Number of tasks allowed to be pending finalization for an fpga accelerator (def:␣
↪4)
   NX_FPGA_MAX_THREADS_CALLBACK = <integer>
     Max. number of threads concurrently working in the FPGA IDLE callback (def: 1)
   NX_FPGA_NUM = <integer>
     Defines de number of FPGA acceleratos to use (def: #accels from libxtasks)
```

CHAPTER

FIVE

CREATE BOOT FILES FOR ULTRASCALE

The newer versions of the Accelerator Integration Tool (AIT, formerly autoVivado) support the automatic generation of boot files for some boards. This includes the steps in *Petalinux (2016.3) build for a custom hdf* or *Petalinux (2018.3) build for a custom hdf*, which are the ones repeated for every BOOT.BIN generation. The steps in *Petalinux project setup* are needed to setup the petalinux project build environment. Assuming that you have a valid petalinux build, you can use the ait functionality with the following points:

- Add the option `--to_step=boot` when calling ait.
- Provide the Petalinux installation and project directories using the following environment variables:
  - `PETALINUX_INSTALL` Petalinux installation directory.
  - `PETALINUX_BUILD` Petalinux project directory.

The following sections explain how to build a petalinux project and how to generate a BOOT.BIN using this project.

5.1 Prerequisites

- Petalinux installer (https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html).
- Vivado handoff file (hdf) from a synthesized Vivado project.

5.1.1 Petalinux installation

Petalinux is installed running its auto-installer package:

```
./petalinux-v2016.3-final-installer.run
```

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. Note that the petalinux settings may change the ARM cross compilers breaking the OmpSs@FPGA tool-chain.

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

5.2 Petalinux project setup

The following steps should be executed once. After them, you will be able to build different boot files just using the AIT option or executing the steps in any of the following sections: *Petalinux (2016.3) build for a custom hdf* (for Petalinux 2016.3) or *Petalinux (2018.3) build for a custom hdf* (for Petalinux 2018.3).

5.2.1 Unpack the bsp

Unpack the bsp to create the petalinux project.

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

5.2.2 [Optional] Fix known problems in AXIOM-ZU9EG-2016.3 project

Here are some patches for known problems:

Problem downloading Root FS

There is a problem during the BSP build when the scripts try to download the rootfs image from the Axiom webservers. The problem is that the download script checks that the server is alive with a `ping` and the AXIOM server is not responding to such type of traffic.

`Patch file (axiom_bsp_patch_00.diff)`

5.2.3 [Optional] Modify the FSBL to have the Fallback system

We developed a modification of Xilinx First Stage Boot Loader (FSBL) to support a fallback boot to a valid known BOOT.BIN file. More information in:

FSBL Fallback Mechanism

We developed a modification of Xilinx First Stage Boot Loader (FSBL) to support a fallback boot to a valid known BOOT.BIN file. The idea is to have a mechanism to allow the boards boot using a known BOOT.BIN file after a failed boot due to the usage of a wrong/corrupted BOOT.BIN. Fallback mechanism operational flow is the following:

- Does the `FLBK.TXT` file exist in the root of the BOOT partition?
- Yes. Enter in fallback mode and boot using the `FLBK.BIN` file
- No. Create the `FLBK.TXT` file and follow with a regular boot (usually using the `BOOT.BIN` file).
- The OS should mount the boot partition and remove the `FLBK.TXT` file after each boot (aka a successful boot).

FSBL Patch

`FSBL Patch file (fsbl_patch_v010.diff)`

In addition to the patch, the read-only filesystem protection must be disabled before the BOOT.BIN generation to allow the fallback mechanism work. This can be done by editing the `xparameters.h` file (`components/bootloader/zynqmp_fsbl/zynqmp_fsbl_bsp/psu_cortexa53_0/include/xparameters.h`) and removing any definition of `FILE_SYSTEM_READ_ONLY` pre-processor variable. Note that this header is re-generated/updated by petalinux tools in some steps. We need to ensure that it is properly edited when the bootloader is compiled, usually during the `petalinux-build` step.

System cleanup service

`Systemctl service (remove-fsbl-flbk.service)`

This service mounts the boot partition and removes the FSBL.TXT file created by the BSC FSBL. If the file is not removed, the next boot will use the `FLBK.BIN` file instead of `BOOT.BIN`. To install the service, copy the service file in the /etc/systemd/system/ folder and enable it with the following commands (they may require root privileges):

```
systemctl daemon-reload
systemctl enable remove-fsbl-flbk.service
systemctl start remove-fsbl-flbk.service
```

5.2.4 Configure petalinux

Run petalinux configuration. No changes need to be made to petalinux configuration, but this step has to be run.

```
export GIT_SSL_NO_VERIFY=1 #Ignore broken certificates
petalinux-config
```

After configuration this step, petalinux will download any needed files from external repositories.

5.2.5 Configure linux kernel

To enter the kernel configuration utility, run:

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

[Optional] Enable Xilinx DMA driver

Note: This step is only needed when the the use of DMA engines is desired.

Xilinx driver support has to be enabled in order to support Xilinx DMA engine devices. Usually, this is not needed as OmpSs@FPGA does not make use them to send tasks, neither information, between the host and the FPGA device. It can be enabled in: `Device drivers → DMA Engines Support → Xilinx axi DMAS`

Fix old kernels

In petalinux <2017, there is a known problem in the Xilinx DMA implementation. To fix it, download `xilinx_dma.c` and replace it in `<project dir>/build/linux/kernel/download/linux-4.6.0-AXIOM-v2016/drivers/dma/xilinx/xilinx_dma.c`, when using a remote kernel, otherwise in `<petalinux install dir>/components/linux-kernel/xlnx-4.6/drivers/dma/xilinx/xilinx\_dma.c`.

[Optional] Increase the CMA (Contiguous Memory Area)

You may want to increase the CMA size. It is used by Nanos++ as memory for the FPGA device copies. Its size can be set in: `Device drivers → Generic Driver Options → DMA Contiguous Memory Allocator`

5.3 Petalinux (2016.3) build for a custom hdf

Once petalinux project is setup, you can update it to contain a custom bitstream with your hardware. This steps can be repeated several times without executing again the steps in the *Petalinux project setup* section. Moreover, AIT supports the automatic execution of the following steps as explained in the beginning of this page.

First, you need to import the hardware description file (hdf) in the petalinux project. This is done executing the following command in the root directory of the petalinux project build.

```
petalinux-config --get-hw-description <path to application hdf file>
```

5.3.1 Add missing nodes to device tree

Some nodes should be added to the device tree before compiling it.

misc_clk_0

Edit the file `./subsystems/linux/configs/device-tree/pl.dtsi` to add or edit the node `misc_clk_0`. It should have the following contents (ensure that clock-frequency is correctly set):

```
misc_clk_0: misc_clk_0 {
  compatible = "fixed-clock";
  #clock-cells = <0>;
  clock-frequency = <200>;
};
```

pl_bsc.dtsi

AIT will generate a `pl_bsc.dtsi` file in the main Vivado project folder. This file contains the missing nodes in the `amba_pl` based on your application build. This file must be copied in `./subsystems/linux/configs/device-tree/` folder and included in `./subsystems/linux/configs/device-tree/system-conf.dtsi` file.

For example, it will be located in `test_ait/Vivado/test/` folder if the project name is `test`.

5.3.2 Build the Linux system

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

```
petalinux-build
```

Error in fsbl compilation

In some cases, fsbl compilation triggered during the petalinux build can fail. This is due to a bad cleanup from previous compilation. In this cases, a complete fsbl cleanup and a new build must be performed. Note, that this extra cleanup may collision with the steps described in *[Optional] Modify the FSBL to have the Fallback system*.

```
petalinux-build -c bootloader -x mrproper
petalinux-build
```

5.3.3 [Optional] Build PMU Firmware

Run hsi (included in petalinux and Xilinx SDK).

```
hsi
```

Inside hsi run

```
set hwdsgn [open_hw_design <hardware.hdf>]
generate_app -hw $hwdsgn -os standalone -proc psu_pmu_0 -app zynqmp_pmufw -compile -
↪sw pmufw -dir <dir_for_new_app>
```

Warning: As of vivado 2016.3 pmu firmware breaks Trenz's TEBF0808 boot

5.3.4 Create BOOT.BIN file

```
petalinux-package --force --boot --fsbl images/linux/zynqmp_fsbl.elf --fpga <path to␣
↪application bit file> --u-boot images/linux/u-boot.elf
cp BOOT.BIN images/linux/image.ub <path to boot partition>
```

When using PMU firmware, pmu binary has to be included in boot.bin file. To do so, add the `--pmufw <pmufw.elf>` argument to the `petalinux-package` command.

5.4 Petalinux (2018.3) build for a custom hdf

Once petalinux 2018.3 project is setup, you can update it to contain a custom bitstream with your hardware. This steps can be repeated several times without executing again the steps in the *Petalinux project setup* section. Moreover, AIT supports the automatic execution of the following steps as explained in the beginning of this page.

First, you need to import the hardware description file (hdf) in the petalinux project. This is done executing the following command in the root directory of the petalinux project build.

```
petalinux-config --get-hw-description <path to application hdf file>
```

5.4.1 Add missing nodes to device tree

Some nodes should be added to the device tree before compiling it.

pl_bsc.dtsi

AIT will generate a `pl_bsc.dtsi` file in the main Vivado project folder. This file contains the missing nodes in the `amba_pl` based on your application build. For example, it will be located in `test_ait/Vivado/test/` folder if the project name is `test`. The contents of such file must be placed at the end of `./project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi` file. Note that any remaining contents from a previous build must be removed before. The following command will append the `pl_bsc.dtsi` content at the end of `system-user.dtsi` file:

```
cat <path to vivado project>/pl_bsc.dtsi >>project-spec/meta-user/recipes-bsp/device-
↪tree/files/system-user.dtsi
```

5.4.2 Build the Linux system

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

```
petalinux-build
```

5.4.3 Create BOOT.BIN file

```
petalinux-package --force --boot --fsbl images/linux/zynqmp_fsbl.elf --fpga <path to␣
↪application bit file> --u-boot images/linux/u-boot.elf
cp BOOT.BIN images/linux/image.ub <path to boot partition>
```

CHAPTER

SIX

CLUSTER INSTALLATIONS

6.1 Ikergune cluster installation

The OmpSs@FPGA releases are automatically installed in the Ikergune 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 the Ikergune cluster should be the same as in the Docker images.

6.1.1 General remarks

- All software is installed in a version folder under the `/apps` directory.
- During the updates, the installation will not be available for the users' usage.
- Usually, the installation just takes 20 minutes.
- After the installation, an informative email will be sent.

6.1.2 Module structure

The ompss modules are:

- ompss/arm64_fpga/*[release version]*
- ompss/arm32_fpga/*[release version]*

Both require having some vivado loaded and a cross-gcc. For example, to load the toolchain for aarch64 git version we need to execute:

```
module load vivado gcc-arm/6.2.0_aarch64 ompss/arm64_fpga/git
```

And for ARM 32bits:

```
module load vivado gcc-arm/6.2.0_gnueabihf ompss/arm32_fpga/git
```

To list all available modules in the system run:

```
module avail
```

Other modules may be required to generate the boot files for some boards, for example: - petalinux

6.1.3 Build applications

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

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

6.1.4 Running applications

Log into a worker node (interactive jobs)

Ikergune cluster uses SLURM in order to manage access to computation resources. Therefore, to log into a worker node, an allocation in one of the partitions have to be made.

There are 2 partitions in the cluster: * `ikergune-eth`: arm32 zynq7000 (7020) nodes * `ZU102`: Xilinx zcu102 board

In order to make an allocation, you must run `salloc`:

```
salloc -p [partition]
```

For instance:

```
salloc -p ikergune-eth
```

Then get the node that has been allocated for you:

```
squeue
```

The output should look similar to this:

```
JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
 8547 ikergune-     bash afilguer  R      16:57      1 Node-3
```

Then, you can log into your node:

```
ssh ethNode-3
```

Load ompss kernel module

The ompss-fpga kernel module has to be loaded before any application using fpga accelerators can be run.

Kernel module binaries are provided in

```
/apps/[arch]/ompss/[release]/kernel-module/ompss_fpga.ko
```

Where `arch` is one of:

- `arm32`
- `arm64`

`release` is one of the ompss@fpga releases currently installed.

For instance, to load the 32bit kernel module for the `git` release, run:

```
sudo insmod /apps/arm32/ompss/git/kernel-module/ompss_fpga.ko
```

You can also run `module avail ompss` for a list of the currently installed releases.

Loading bistreams

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

```
load_bitstream bitstream.bin
```

Note that the `.bin` file is being loaded. Trying to load the `.bit` file will result in an error.

6.2 Xaloc cluster installation

The OmpSs@FPGA releases are automatically installed in the Xaloc 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 the Xaloc cluster should be the same as in the Docker images.

6.2.1 General remarks

- All software is installed in a version folder under the `/opt/bsc` directory.
- During the updates, the installation will not be available for the users' usage.
- Usually, the installation just takes 20 minutes.
- After the installation, an informative email will be sent.

6.2.2 Module structure

The ompss modules are:

- ompss/x86_fpga/*[release version]*

It requires having some vivado loaded:

```
module load vivado ompss/x86_fpga/git
```

To list all available modules in the system run:

```
module avail
```

6.2.3 Build applications

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

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

6.2.4 Running applications

Get access to an installed fpga

Xaloc cluster 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.

There is 1 partition in the cluster: * `fpga`: Alveo U200 board

In order to make an allocation, you must run `salloc`:

```
salloc -p [partition]
```

For instance:

```
salloc -p fpga
```

Then get the node that has been allocated for you:

```
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 xaloc
```

Loading bistreams

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

```
load_bitstream bitstream.bit
```

6.3 Quar cluster installation

The OmpSs@FPGA releases are automatically installed in the Quar 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 the Quar cluster should be the same as in the Docker images.

6.3.1 General remarks

- All software is installed in a version folder under the `/opt/bsc` directory.
- During the updates, the installation will not be available for the users' usage.
- Usually, the installation just takes 20 minutes.
- After the installation, an informative email will be sent.

6.3.2 Module structure

The ompss modules are:

- ompss/x86_fpga/*[release version]*

It requires having some vivado loaded:

```
module load vivado ompss/x86_fpga/git
```

To list all available modules in the system run:

```
module avail
```

6.3.3 Build applications

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

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

6.3.4 Running applications

Get access to an installed fpga

Quar cluster 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.

There is 1 partition in the cluster: * `fpga`: Alveo U200 board

In order to make an allocation, you must run `salloc`:

```
salloc -p [partition]
```

For instance:

```
salloc -p fpga
```

Then get the node that has been allocated for you:

```
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 the FPGA configuration.

```
load_bitstream bitstream.bit
```

6.4 crdbmaster cluster installation

The OmpSs@FPGA releases are automatically installed in the crdbmaster 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 the crdbmaster cluster should be the same as in the Docker images.

6.4.1 General remarks

- All software is installed in a version folder under the `/opt/bsc` directory.
- During the updates, the installation will not be available for the users' usage.
- Usually, the installation just takes 20 minutes.
- After the installation, an informative email will be sent.

6.4.2 System overview

Current setup consists of an x86 login node and a CRDB directly connected to it. Serial lines and jtag are connected to the login node, allowing node management as well as debug and programming.

CRDB

CRDB is a FPGA development board developed within the euroexa project.

It has two discrete devices, a Zynq Ultrascale XCZU9EG and a Virtex Ultrascale+ XCVU9P. Both devices are directly connected.

CRDB itself has 16GB system memory and 16x3 GB FPGA memory.

6.4.3 Logging into the system

Crdbmaster login node is accessible via ssh at `crbmaster.bsc.es`

6.4.4 Module structure

The ompss modules are:

- ompss/arm64_fpga/*[release version]*

It requires having some vivado loaded:

```
module load vivado ompss/arm64_fpga/git
```

To list all available modules in the system run:

```
module avail
```

6.4.5 Build applications

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

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

6.4.6 Running applications

Get access to an installed fpga

crdbmaster cluster 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.

There is 1 partition in the cluster: * `fpga`: EuroEXA CRDB board

In order to make an allocation, you must run `salloc`:

```
salloc -p [partition]
```

For instance:

```
salloc -p fpga
```

Then get the node that has been allocated for you:

```
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 crdb
```

Then, you can log into your node:

```
ssh crdb
```

OmpSs Initialization

Before the application can be executed, the ZU9 environment has to be set up in order to run ompss applications, and the VU9 design must be loaded.

The following command loads the needed kernel modules and resets device to device communications. Note that the ZU9 design is already loaded upon Linux boot.

```
init_ompss-[ompss-fpga-release]-vu9-programming [vu9-bitstream.bit]
```

For instance, to initialize environment for 3.2.0 release, run:

```
init_ompss-3.2.0-vu9-programming mybitstream.bit
```

Loading bistreams

Since system has two FPGA devices, Devices can be independently. This is useful when reloading the bitstream after ompss initialization.

VU9 can be loaded using:

```
load_vu9 myNewBitstream.bit
```

Also, Zynq bitstream can be reloaded on runtime. However **this is not needed during normal operation**

```
load_zu9 myZynqBitstream.bit
```

CRDB cold reboot

Attach to serial ports

Board serial ports and debug port are only available to the user that has the active running job on the board. Also, attached minicom instances will be killed upon job end. Devices are `/dev/ttyUSB0` for the management controller and `/dev/ttyUSB2` for the processing system serial line.

```
minicom -D /dev/ttyUSB0     # management
minicom -D /dev/ttyUSB2     # serial line
```

Serial port settings

Minicom serial port settings in order to connect to the CRDBs Serial ports

```
A -    Serial Device      : /dev/ttyUSB0
B - Lockfile Location     : /var/lock
C -   Callin Program      :
D -  Callout Program      :
E -    Bps/Par/Bits       : 115200 8N1
F - Hardware Flow Control : No
G - Software Flow Control : No
```

In order to use these settings, this has to be saved into minicom's config file.

Set ~/.minirc.dfl contents as follows:

```
pu port /dev/ttyUSB0
pu baudrate 115200
pu rtscts No
pu xonxoff No
```

Hard reboot

Once connected to the management serial line, you should see a menu like this: (You may need to press enter for it to refresh)

```
MainMenu
0) Interrupt config mcu
1) I2C Scan menu
2) I2C CMD menu (1 byte register)
3) I2C IO expander 1 menu
4) I2C IO expander 2 menu
5) I2C CMD menu (2 byte register)
6) Execute startup sequence eMMC
7) Execute startup sequence SD
```

Then select 3

```
i2cIOExpander1Menu
Polling: DISABLED
Pin: 00,    output enabled: 1,                   description: EN_5V
Pin: 01,    output enabled: 1,                   description: EN_3V3
Pin: 02,    output enabled: 1,                   description: EN_12V
Pin: 03,    output enabled: 0,                   description: nSYS_RESET
Pin: 04,    output enabled: 0,                   description: nCB_RESET
Pin: 05, interrupt enabled: 0, actual value: 0, description: -
Pin: 06, interrupt enabled: 0, actual value: 1, description: nTHRM
Pin: 07, interrupt enabled: 0, actual value: 1, description: nSMB_ALERT
Pin: 10, interrupt enabled: 0, actual value: 0, description: PERST
Pin: 11, interrupt enabled: 0, actual value: 1, description: PEWAKE
Pin: 12, interrupt enabled: 0, actual value: 1, description: nTHRM_TRIP
Pin: 13, interrupt enabled: 0, actual value: 1, description: PWR_OK
Pin: 14,    output enabled: 1,                    description: nPWRBTN
Pin: 15, interrupt enabled: 0, actual value: 0, description: nTYPE(0)
Pin: 16, interrupt enabled: 0, actual value: 1, description: nTYPE(1)
Pin: 17, interrupt enabled: 0, actual value: 0, description: nTYPE(2)
Enter [pin] to toggle the interrupt enabled/output state
P to toggle polling enable status
C to change input/output configuration
I to initialize
R to refresh
```

Then enter `02` twice to switch off and on the 12V rail

FPGA jtag debug

A vivado 2020.1 hw_server is running on the controller node so users can remotely access jtag interface.

This is done by specifying a remote target in vivado hardware manager: `crdbmaster.bsc.es:3121`

Users also can run a local vivado instance by running vivado in the master node and forwarding X graphics. However, this is not recommended as waveform visualization is graphics intensive and latency is not negligible.

6.5 Llebeig cluster installation

The OmpSs@FPGA releases are automatically installed in the Llebeig 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 the Llebeig cluster should be the same as in the Docker images.

6.5.1 General remarks

- All software is installed in a version folder under the `/opt/bsc` directory.
- During the updates, the installation will not be available for the users' usage.
- Usually, the installation just takes 20 minutes.
- After the installation, an informative email will be sent.

6.5.2 Module structure

The ompss modules are:

- ompss/x86_fpga/*[release version]*

It requires having some vivado loaded:

```
module load vivado ompss/x86_fpga/git
```

To list all available modules in the system run:

```
module avail
```

6.5.3 Build applications

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

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

CHAPTER

SEVEN

FAQ: FREQUENTLY ASKED QUESTIONS

7.1 What is OmpSs?

OmpSs is an effort to integrate features from the StarSs programming model developed at BSC into a single programming model. In particular, our objective is to extend OpenMP with new directives to support asynchronous parallelism and heterogeneity (devices like GPUs, FPGAs). Then, OmpSs@FPGA is the extension of OmpSs tools to fully support FPGA devices.

Note: For more information about OmpSs programming model refer to https://pm.bsc.es/ompss

7.2 How to keep HLS intermediate files generated by Mercurium?

Mercurium generates an intermediate C++ HLS source file for each FPGA task defined in the source code. Those files are used by AIT to generate the FPGA bitstream, and removed after the invocation. To keep the Mercurium intermediate files, there is the `-k` option. It will keep the C/C++ intermediate files for the native compiler and the C++ HLS files for AIT. Usage example:

```
fpgacc --ompss --bitstream-generation -k src/dotproduct.c -o dotproduct \
  --Wf,--board=zedboard
```

Note: HLS source files are not generated if `--bitstream-generation` option is not used in Mercurium call

Edit the HLS intermediate files and call AIT

It is not recommended, but under some circumstances one would need to edit the HLS intermediate files before the AIT call. The following steps shows how to do so.

1. We call Mercurium with the desired options and two extra flags: - `-k` which keeps the intermediate files generated by the compiler. - `--verbose` which enables the verbose mode of the compiler. This is needed to gather the AIT command call that Mercurium executes.

   We could add an unsupported argument in the AIT flags to make it abort, as we do not want the bitstream before the modifications. For example, we could add the `--abort` flag to `--Wf` option of Mercurium. Example of Mercurium call and generated output:

```
[user@machine] $ fpgacc --ompss -k --verbose --bitstream-generation --Wf,-
↪b=zcu102,--hwruntime=som,--abort foo.c -o foo
Loading compiler phases for profile 'fpgacc'
```

(continues on next page)

(continued from previous page)

```
Compiler phases for profile 'fpgacc' loaded in 0.02 seconds
Compiling file 'foo.c'
gcc -E -D_OPENMP=200805 -D_OMPSS=1 -I/home/user/opt/ompss/git/nanox/include/nanox␣
↪-include nanos.h -include nanos_omp.h -include nanos-fpga.h -std=gnu99 -D_MCC -
↪D_MERCURIUM -o /tmp/fpgacc_zHMisA foo.c
File 'foo.c' preprocessed in 0.01 seconds
Parsing file 'foo.c' ('/tmp/fpgacc_zHMisA')
File 'foo.c' ('/tmp/fpgacc_zHMisA') parsed in 0.02 seconds
Nanos++ prerun
Early compiler phases pipeline executed in 0.00 seconds
File 'foo.c' ('/tmp/fpgacc_zHMisA') semantically analyzed in 0.01 seconds
Checking parse tree consistency
Parse tree consistency verified in 0.00 seconds
Freeing parse tree
Parse tree freed in 0.00 seconds
Checking integrity of nodecl tree
Nodecl integrity verified in 0.00 seconds
foo.c:1:13: info: unless 'no_copy_deps' is specified, the default in OmpSs is now
↪'copy_deps'
foo.c:1:13: info: this diagnostic is only shown for the first task found
foo.c:2:13: info: adding task function 'foo' for device 'fpga'
Nanos++ phase
foo.c:10:3: info: call to task function 'foo'
foo.c:2:13: info: task function declared here
FPGA bitstream generation phase analysis - ON
Compiler phases pipeline executed in 0.01 seconds
Prettyprinted into file 'fpgacc_foo.c' in 0.00 seconds
Performing native compilation of 'fpgacc_foo.c' into 'foo.o'
gcc -std=gnu99 -c -o foo.o fpgacc_foo.c
File 'foo.c' ('fpgacc_foo.c') natively compiled in 0.02 seconds
objdump -w -h foo.o 1> /tmp/fpgacc_d9sKiY
gcc -o foo -std=gnu99 foo.o -Xlinker --enable-new-dtags -L/home/user/opt/ompss/
↪git/nanox/lib/performance -Xlinker -rpath -Xlinker /home/user/opt/ompss/git/
↪nanox/lib/performance -Xlinker --no-as-needed -lnanox-ompss -lnanox-c -lnanox -
↪lpthread -lrt -lnanox-fpga-api
Link performed in 0.04 seconds
ait -b=zcu102 --hwruntime=som --abort --wrapper_version=7 -n=foo
usage: ait -b BOARD -n NAME
ait error: unrecognized arguments: --abort. Try 'ait -h' for more information.
Link fpga failed
Removing temporary filename '/tmp/fpgacc_d9sKiY'
Removing temporary filename 'foo.o'
Removing temporary filename '/tmp/fpgacc_zHMisA'
```

2. Now the intermediate files are available and we could edit them as desired. In the example, `7288177970:1:foo_hls_automatic_mcxx.cpp` is available:

```
[user@machine] $ ls -l
total 44
-rw-r--r-- 1 user user  3735 Jun  5 10:27 7288177970:1:foo_hls_automatic_mcxx.cpp
-rwxr-xr-x 1 user user 17568 Jun  5 10:27 foo
-rw-r--r-- 1 user user   235 Jun  4 15:17 foo.c
-rw-r--r-- 1 user user 13460 Jun  5 10:27 fpgacc_foo.c
```

3. After modifying the HLS intermediate files, the AIT call that Mercurium usually performs has to be executed. In the verbose output of the first step, there is one line with the invoked call. This call has to be repeated, removing the extra options added to make AIT abort (if any). In the example, the `--abort` option has to be removed and

the AIT command to invoke will be:

```
[user@machine] $ ait -b=zcu102 --hwruntime=som --wrapper_version=7 -n=foo
Using xilinx backend
Checking vendor support for selected board
...
```

7.3 Problems with structure/symbol redefinition in FPGA tasks

In C sources (not in C++), Mercurium imports the symbols used by FPGA tasks into the HLS intermediate source codes. Therefore, the usage of `.fpga` headers is no longer needed unless some specific cases. If they are used, the result may be duplicated definition of some symbols (in the HLS source and in the included `.fpga` header). The solution may be as simple as rename the `.fpga` headers into regular `.h` files.

In C++, the management of the symbols is more complex and the symbols are not automatically imported to HLS source files. Basically, only the functions in the same source file of the FPGA task are imported. So, the usage of `.fpga.hpp` headers may be needed.

7.4 Hide/change FPGA task code during Mercurium binary compilation

At some undesirable point, one could need to hide some application code to Mercurium but not to HLS tools from FPGA vendor. To this end, Mercurium defines the `__HLS_AUTOMATIC_MCXX__` compiler preprocessor variable. Note that does not make sense to directly use this in the task source code. Because it will be handled by Mercurium (during the HLS intermediate files generation) and the protected code will be removed. Instead, it could be used in a `.fpga.h` or `.fpga.hpp` header file, which are still included in the HLS intermediate files. However, the functions must be self-contained to avoid further dependencies and circular dependencies.

task.cpp example:

```
#include "task.fpga.hpp"

#pragma omp task device(fpga)
#pragma omp task in([LEN]array) out([1]reduction)
void array_reduction(const long long int *array, long long int *reduction) {
    my_longer_type_t acum = 0;
    for (int i=0; i<LEN; i++) acum += array[i];
    *reduction = acum/LEN;
}
```

task.fpga.hpp example:

```
#ifdef __HLS_AUTOMATIC_MCXX__
    #include <ap_int.h>
    typedef apint<100> my_longer_type_t;
#else
    typedef long long int my_longer_type_t;
#endif
```

- genindex

INDEX