OmpSs-2@FPGA User Guide

Release git

BSC Programming Models

Oct 04, 2024

CONTENTS

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 (https://hub.docker.com/r/bscpm/ompss_2_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 3.7 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.

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

- `PREFIX_TARGET` [Def: `/`] Installation prefix for the target tools (e.g. nanos6, libxdma, libxtasks)
- `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-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
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](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)
- [Nanos6-fpga](https://github.com/bsc-pm-ompss-at-fpga/nanos6-fpga)
- [LLVM](https://github.com/bsc-pm-ompss-at-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 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 --disable-discrete-deps -
↪-disable-all-instrumentations --enable-stats-instrumentation --enable-verbose-
↪instrumentation
make
make install
```

1.3.6 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
cd llvm
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
```

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 (https://pm.bsc.es/ompss-docs/book/index.html). 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 realize 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@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@FPGA VS OmpSs

Despite OmpSs@FPGA mostly follows the OmpSs behaviour, there are specific clauses or directives that are not yet implemente

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

2.3 Clauses of task directive

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

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) {
```

(continues on next page)

(continued from previous page)

```
  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 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.

2.4.1 Nanos6 FPGA Architecture API

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

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);
```

nanos_fpga_free

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

nanos_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 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 *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.

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 \
  src/dotproduct.c -o dotproduct \
  -fompss-fpga-ait-flags "--board=alveo_u200 --name=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 \
  src/dotproduct.c -o dotproduct \
  -fompss-fpga-ait-flags "--board=alveo_u200 --name=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-memory-port-width 512 \
  src/dotproduct.c -o dotproduct \
  -fompss-fpga-ait-flags "--board=alveo_u200 --name=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-check-limits-memory-port \
  src/dotproduct.c -o dotproduct \
  -fompss-fpga-ait-flags "--board=alveo_u200 --name=dotproduct"
```

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, alveo_u280, alveo_u280_hbm,␣
↪alveo_u55c, com_express, kv260, simulation, 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:
```

(continues on next page)

(continued from previous page)

```
                        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')
  --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
  --bitinfo_note BITINFO_NOTE
                        custom note to add to the bitInfo

Data path:
  --datainterfaces_map DATAINTERFACES_MAP
                        path of mappings file for the data interfaces
  --memory_interleaving_stride MEM_INTERLEAVING_STRIDE
                        size in bytes of the stride of the memory interleaving. By␣
↪default there is no interleaving
  --disable_creator_ports
                        Disable memory access ports in the task-creation accelerators

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)
  --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')
```

(continues on next page)

(continued from previous page)

```
  --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 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')

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

Miscellaneous:
```

(continues on next page)

(continued from previous page)

```
  -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 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:
  --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
  --placement_file PLACEMENT_FILE
                        json file specifying accelerator placement
  --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
  --regslice_pipeline_stages REGSLICE_PIPELINE_STAGES
                        number of register slice pipeline stages per SLR
                        '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_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
  --interconnect_opt OPT_STRATEGY
                        AXI interconnect optimization strategy: Minimize 'area' or␣
↪maximize 'performance'
                        (def: 'area')
  --interconnect_priorities
                        enable priorities in the memory interconnect
  --simplify_interconnection
                        simplify interconnection between accelerators and memory.␣
↪Might negatively impact timing
  --power_monitor       enable power monitoring infrastructure
  --thermal_monitor     enable thermal monitoring infrastructure
  --debug_intfs INTF_TYPE
                        choose which interfaces mark for debug and instantiate the␣
↪correspondent ILA cores
```

(continues on next page)

(continued from previous page)

```
                        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
  --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
```

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`

On an Alveo U200, which has 3 Super logic regions, external interfaces are placed as follows:

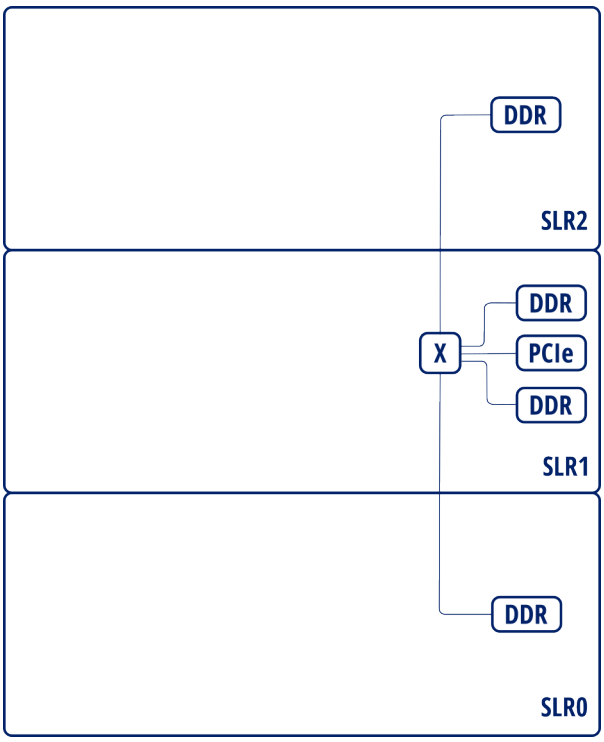

Fig. 1: Interface layout for Alveo U200

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:

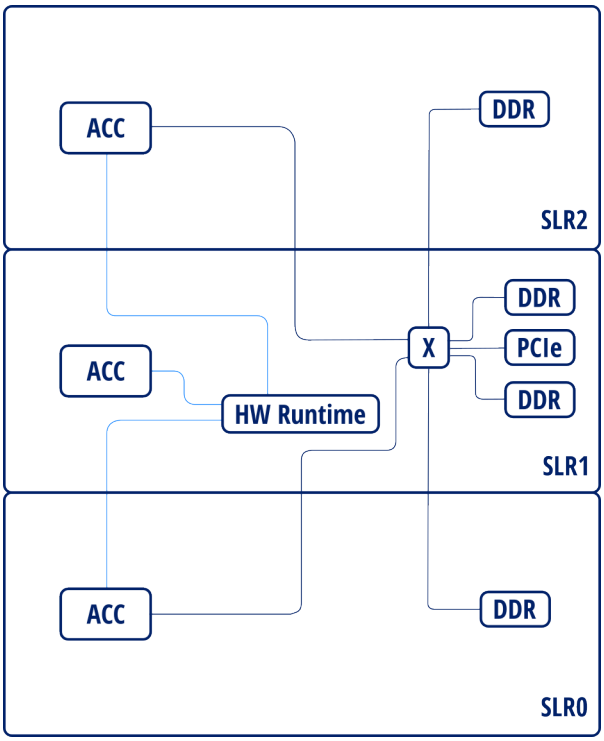

Fig. 2: Placed instance diagram

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. For example, activating register slices for the previous design will result in the following layout:

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`

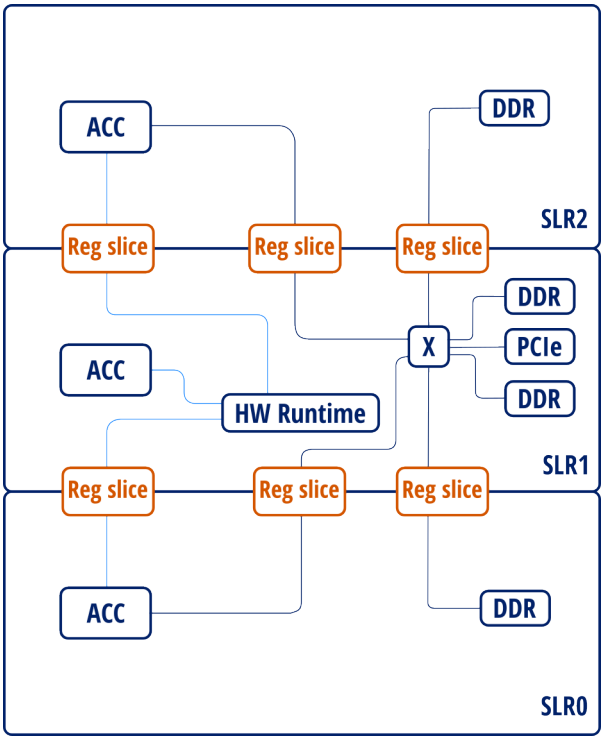

Fig. 3: Placed instance diagram with register slices

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.

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:

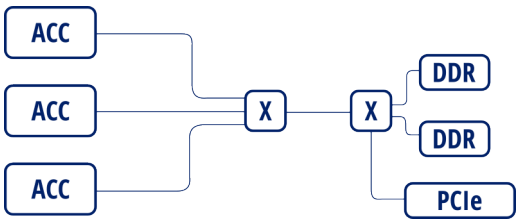

Fig. 4: 2 stage interconnection

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:

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.

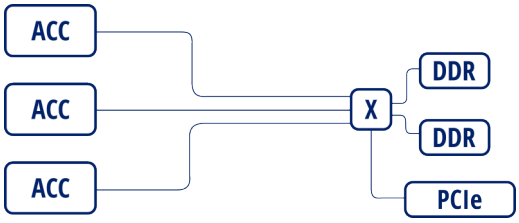

Fig. 5: Simplified interconnection

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:

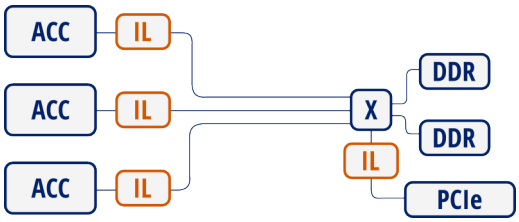

Fig. 6: Simplified interconnection with memory interleaving

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

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.

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 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 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. 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 `-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 (https://pm.bsc.es/ftp/ompss-2/doc/user-guide/nanos6/index.html).

4.1 Nanos6 FPGA Architecture options

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

4.1.1 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, 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 = 16
                # 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
                # 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.␣
↪Default is true
```

(continues on next page)

(continued from previous page)

```
                        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
```

CHAPTER

FIVE

CREATE BOOT FILES FOR ULTRASCALE

The newer versions of the Accelerator Integration Tool (AIT) 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:

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:

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-2@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/bsc/ARCH/ompss-2/` directory.
- During the updates, the installation will not be available for the users' usage.
- After the installation, an informative email will be sent.

6.1.2 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
```

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

```
module load petalinux
```

6.1.3 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*.

6.1.4 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/bsc/[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/bsc/arm32/ompss/git/kernel-module/ompss_fpga.ko
```

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

Loading bitstreams

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-2@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

- The OmpSs@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.
- During the updates, the installation will not be available for the users' usage.
- Usually, the installation takes about 30 minutes.
- After the installation, an informative email will be sent.

6.2.2 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.xilinx.com/products/boards-and-kits/alveo/u200.html

6.2.3 Logging into xaloc

Xaloc is accessible from HCA `ssh.hca.bsc.es` Alternatively, it can be accessed through the `4810` port in HCA and ssh connection will be redirected to the actual host:

```
ssh -p 4810 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
```

6.2.4 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:

```
module load vivado/2023.2 ompss-2/x86_fpga/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

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`: a Versal VCK5000 board

The easiest way to allocate an FPGA is to run bash through `srun` with the `--gres` option:

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

Where `BOARD` is the FPGA to allocate, in this case `versal`, and `N` the number of FPGAs to allocate, that is 1.

For instance, the command:

```
srun --gres=fpga:versal:1 --pty bash
```

Will allocate the FPGA and run an interactive bash 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 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 [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 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           pcie          usb
0:      XXXXXXXXXXXXX  0000:02:00.0  002:002
```

Set up qdma queues

Note: This step is performed by `load_bitstream` script, which creates a single bidirectional memory mapped queue. This is only needed if other configuration is needed.

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 `dmactl` tool is used. For instance: In order to create and start a memory mapped qdma queue with index 1 run:

```
dmactl qdma02000 q add idx 1 mode mm dir bi
dmactl qdma02000 q start idx 1 mode mm dir bi
```

OmpSs 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:

```
dmactl qdma02000 q stop idx 1 mode mm dir bi
dmactl qdma02000 q del idx 1 mode mm dir bi
```

For more information, see

```
dmactl --help
```

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:

```
Reading bitinfo of FPGA 0000:b3:00.0
Bitstream info version: 11
Number of acc:  8
AIT version:    7.1.0
Wrapper version 13
Board base frequency (Hz)   156250000
Interleaving stride 32768
Features:
[ ] Instrumentation
[ ] Hardware counter
[x] Performance interconnect
[ ] Simplified interconnection
[ ] POM AXI-Lite
[x] POM task creation
[x] POM dependencies
[ ] POM lock
[x] POM spawn queues
[ ] Power monitor (CMS) enabled
[ ] Thermal monitor (sysmon) enabled
Cmd In addr 0x2000 len 128
Cmd Out addr 0x4000 len 128
Spawn In addr 0x6000 len 1024
Spawn Out addr 0x8000 len 1024
Managed rstn addr 0xA000
Hardware counter addr 0x0
POM AXI-Lite addr 0x0
Power monitor (CMS) addr 0x0
Thermal monitor (sysmon) addr 0x0

xtasks accelerator config:
type        count   freq(KHz)   description
5839957875  1   300000      matmulFPGA
7602000973  7   300000      matmulBlock

ait command line:
ait --name=matmul --board=alveo_u200 -c=300 --memory_interleaving_stride=32K --
↪simplify_interconnection --interconnect_opt=performance --interconnect_regslice=all␣
↪--floorplanning_constr=all --slr_slices=all --placement_file=u200_placement_7x256.
↪json --wrapper_version 13

Hardware runtime VLNV:
bsc:ompss:picosompssmanager:7.3

bitinfo note:
''
```

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 Xaloc, when allocating an FPGA with slurm, a Vivado HW server is automatically launched for each FPGA:
    - FPGA 0 uses port 3120
2. On the local machine, assuming that Xaloc's HW Server runs on port 3121, let all connections to port 3121 be

forwarded to Xaloc by doing `ssh -L 3121:xaloc:3121 [USER]@ssh.hca.bsc.es -p 8410`.

3. Finally, from the local machine, connect to Xaloc'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.

6.3 Quar cluster installation

La Quar is a small town and municipality located in the comarca of Berguedà, in Catalonia.

It's also an intel machine containing two Xilinx Alveo U200 accelerator cards.

The OmpSs-2@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

- The OmpSs@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.
- During the updates, the installation will not be available for the users' usage.
- Usually, the installation takes about 30 minutes.
- After the installation, an informative email will be sent.

6.3.2 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
- FPGA: Xilinx Alveo U200
    - https://www.xilinx.com/products/boards-and-kits/alveo/u200.html

6.3.3 Logging into quar

Quar is accessible from HCA `ssh.hca.bsc.es` Alternatively, it can be accessed through the `4819` port in HCA and ssh connection will be redirected to the actual host:

```
ssh -p 4819 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
```

6.3.4 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:

```
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

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`: two Alveo U200 boards

The easiest way to allocate an FPGA is to run bash through `srun` with the `--gres` option:

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

Where `BOARD` is the FPGA to allocate, in this case `alveo_u200`, and `N` the number of FPGAs to allocate, either 1 or 2.

For instance, the command:

```
srun --gres=fpga:alveo_u200:2 --pty bash
```

Will allocate both FPGAs and run an interactive bash 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 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 [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 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           pcie          usb
0:      21290594G00LA  0000:b3:00.0  001:002
1:      21290594G00EA  0000:65:00.0  001:003
```

Set up qdma queues

Note: This step is performed by `load_bitstream` script, which creates a single bidirectional memory mapped queue. This is only needed if other configuration is needed.

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 `dmactl` tool is used. For instance: In order to create and start a memory mapped qdma queue with index 1 run:

```
dmactl qdmab3000 q add idx 1 mode mm dir bi
dmactl qdmab3000 q start idx 1 mode mm dir bi
```

OmpSs 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:

```
dmactl qdmab3000 q stop idx 1 mode mm dir bi
dmactl qdmab3000 q del idx 1 mode mm dir bi
```

For more information, see

```
dmactl --help
```

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:

```
Reading bitinfo of FPGA 0000:b3:00.0
Bitstream info version: 11
Number of acc:  8
AIT version:    7.1.0
Wrapper version 13
Board base frequency (Hz)   156250000
Interleaving stride 32768
Features:
[ ] Instrumentation
[ ] Hardware counter
[x] Performance interconnect
[ ] Simplified interconnection
[ ] POM AXI-Lite
[x] POM task creation
[x] POM dependencies
[ ] POM lock
[x] POM spawn queues
[ ] Power monitor (CMS) enabled
[ ] Thermal monitor (sysmon) enabled
Cmd In addr 0x2000 len 128
Cmd Out addr 0x4000 len 128
Spawn In addr 0x6000 len 1024
Spawn Out addr 0x8000 len 1024
Managed rstn addr 0xA000
Hardware counter addr 0x0
POM AXI-Lite addr 0x0
Power monitor (CMS) addr 0x0
Thermal monitor (sysmon) addr 0x0

xtasks accelerator config:
type        count   freq(KHz)   description
5839957875  1   300000      matmulFPGA
7602000973  7   300000      matmulBlock

ait command line:
ait --name=matmul --board=alveo_u200 -c=300 --memory_interleaving_stride=32K --
↪simplify_interconnection --interconnect_opt=performance --interconnect_regslice=all␣
↪--floorplanning_constr=all --slr_slices=all --placement_file=u200_placement_7x256.
↪json --wrapper_version 13

Hardware runtime VLNV:
bsc:ompss:picosompssmanager:7.3

bitinfo note:
''
```

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
```

(continues on next page)

(continued from previous page)

```
  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
  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
```

6.4 crdbmaster cluster installation

The OmpSs-2@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

- The OmpSs@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.
- During the updates, the installation will not be available for the users' usage.
- Usually, the installation takes about 30 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-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:

```
module load vivado/2023.2 ompss-2/arm64/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-2@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)
```

(continues on next page)

(continued from previous page)

```
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-2@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

- The OmpSs@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.
- During the updates, the installation will not be available for the users' usage.
- Usually, the installation takes about 30 minutes.
- After the installation, an informative email will be sent.

6.5.2 Logging into llebeig

Llebeig is accessible from HCA `ssh.hca.bsc.es` Alternatively, it can be accessed through the `4812` port in HCA and ssh connection will be redirected to the actual host:

```
ssh -p 4812 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
```

6.5.3 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
```

6.5.4 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*.

CHAPTER

SEVEN

FAQ: FREQUENTLY ASKED QUESTIONS

7.1 What is OmpSs-2?

OmpSs-2 is a programming model composed of a set of directives and library routines that can be used in conjunction with a high-level programming language in order to develop concurrent applications. It is, by the way, the second generation of the OmpSs programming model. The name originally comes from two other programming models: OpenMP and StarSs. The design principles of these two programming models constitute the fundamental ideas used to conceive the OmpSs philosophy. OmpSs-2@FPGA is the extension of OmpSs-2 tools to fully support FPGA devices.

Note: For more information about OmpSs-2 programming model refer to https://pm.bsc.es/ompss-2

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'
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
```

7.5 Statically scheduling tasks to different task instances

Warning: The syntax exposed in this page is under development and may change in the future.

The onto clause can take 2 expressions to define the type of FPGA accelerator and the instance of such type that will handle the spawned task. The first expression must be a constant integer uniquely identifying the task type in the application. The lower 32 bits define the task type and can take any arbitrary value, Mercurium generates a hash of the function name when the onto clause is not specified. The following 8 bits (39 down to 32) specify the architecture where task has to be executed. Currently, the architecture bits are 0x1 for FPGA and 0x2 for SMP. The second clause expression will be evaluated during the task creation and defines the instance that will execute the task. It must evaluate to an integer between 0 and the number of instances specified for the child task -1. Otherwise, the behavior is undefined.

The following C example shows how to statically schedule the matmulBlock tasks in the different instances.

```
unsigned char instance_num_glob;

#pragma omp target device(fpga) onto(0x100000010, instance_num_glob) num_instances(5)
#pragma omp task in([BSIZE*BSIZE]a, [BSIZE*BSIZE]b) inout([BSIZE*BSIZE]c)
void matmulBlock(const elem_t *a, const elem_t *b, elem_t *c, const unsigned char␣
↪instance_num);

#pragma omp target device(fpga) copy_in([msize*msize]a, [msize*msize]b) copy_
↪inout([msize*msize]c)
#pragma omp task
void matmulFPGA(const elem_t *a, const elem_t *b, elem_t *c, const unsigned int␣
↪msize) {
  const unsigned int b2size = BSIZE*BSIZE;
  instance_num_glob = 0;
  for (unsigned int i = 0; i < msize/BSIZE; i++) {
    for (unsigned int k = 0; k < msize/BSIZE; k++) {
      unsigned int const ai = k*b2size + i*BSIZE*msize;
      for (unsigned int j = 0; j < msize/BSIZE; j++) {
        unsigned int const bi = j*b2size + k*BSIZE*msize;
        unsigned int const ci = j*b2size + i*BSIZE*msize;
        matmulBlock(a + ai, b + bi, c + ci);
        instance_num_glob = (instance_num_glob + 1) > 5 ? 0 : (instance_num_glob + 1);
      }
    }
  }
  #pragma omp taskwait
}
```

7.6 Remarks/Limitations

The current limitations are:

- The second onto expression (which defines the instance) is only evaluated when spawning tasks within another FPGA task.
- When the second onto expression references a variable it must be either a task argument of a global variable.
- In C sources, global variables are automatically privatized for each FPGA task accelerator.

- In C++ sources, global variables must be moved to intermediate HLS C++ code with the *#pragma omp target device(fpga,smp)* directive on top of variable declarations. See the example below. The pragma will only move the variable declaration to the FPGA intermediate code, effectively privatizing the variable for each FPGA task accelerator.

```
#pragma omp target device(fpga,smp)
unsigned char instance_num_glob;
```

- genindex

INDEX