OmpSs@FPGA User Guide

Release 2.2.0

BSC Programming Models

Apr 25, 2022

CONTENTS

1 Develop OmpSs@FPGA programs **3**
- 1.1 Limitations . . . 3
- 1.2 Clauses of target directive . . . 3
  - 1.2.1 num_instances . . . 4
  - 1.2.2 onto . . . 4
- 1.3 Calls to Nanos++ API . . . 4
  - 1.3.1 Nanos++ Instrumentation API . . . 5
  - 1.3.2 Nanos++ FPGA Architecture API . . . 7

2 Compile OmpSs@FPGA programs **9**
- 2.1 Mercurium FPGA Phase options . . . 9
  - 2.1.1 fpga_memory_port_width . . . 9
  - 2.1.2 force_fpga_periodic_support . . . 9
  - 2.1.3 disable_unaligned_fpga_memory_port_width . . . 9
- 2.2 Binaries . . . 10
- 2.3 Bitstream . . . 10
  - 2.3.1 HW Instrumentation . . . 10
  - 2.3.2 Shared memory port . . . 11
- 2.4 Boot Files . . . 11

3 Running OmpSs@FPGA Programs **13**
- 3.1 Nanos++ FPGA Architecture options . . . 13
  - 3.1.1 Nanos++ FPGA Architecture options . . . 13

4 Create boot files for ultrascale **15**
- 4.1 Prerequisites . . . 15
  - 4.1.1 Petalinux installation . . . 15
- 4.2 Petalinux project setup . . . 15
  - 4.2.1 Unpack the bsp . . . 16
  - 4.2.2 [Optional] Fix known problems in AXIOM-ZU9EG-2016.3 project . . . 16
  - 4.2.3 [Optional] Modify the FSBL to have the Fallback system . . . 16
  - 4.2.4 Configure petalinux . . . 17
  - 4.2.5 Configure linux kernel . . . 17
- 4.3 Petalinux (2016.3) build for a custom hdf . . . 18
  - 4.3.1 Add missing nodes to device tree . . . 18
  - 4.3.2 Build the Linux system . . . 18
  - 4.3.3 [Optional] Build PMU Firmware . . . 19
  - 4.3.4 Create BOOT.BIN file . . . 19
- 4.4 Petalinux (2018.3) build for a custom hdf . . . 19
  - 4.4.1 Add missing nodes to device tree . . . 19

4.4.2 Build the Linux system 20
4.4.3 Create BOOT.BIN file 20

5 FAQ: Frequently Asked Questions **21**
5.1 What is OmpSs? 21

Index **23**

Note: There is a PDF version of this document at http://pm.bsc.es/ftp/ompss-at-fpga/doc/user-guide-2.2.0/OmpSsFPGAUserGuide.pdf

CHAPTER

ONE

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;
  }
}
```

1.1 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 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.hpp 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.

1.2 Clauses of target directive

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

1.2.1 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;
  }
}
```

1.2.2 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;
  }
}
```

1.3 Calls to Nanos++ API

The list of APIs that can be called within a FPGA task is:

- `nanos_err_t nanos_instrument_burst_begin(nanos_event_key_t event, nanos_event_value_t value)`
- `nanos_err_t nanos_instrument_burst_end(nanos_event_key_t event, nanos_event_value_t value)`
- `nanos_err_t nanos_instrument_point_event(nanos_event_key_t event, nanos_event_value_t value)`
- `nanos_err_t unsigned long long int nanos_fpga_current_wd()`
- `nanos_err_t nanos_fpga_wg_wait_completion( unsigned long long int uwg, unsigned char avoid_flush )`

- `void nanos_fpga_create_wd_async(const unsigned long long int type, const unsigned char numArgs, const unsigned long long int * args, const unsigned char numDeps, const unsigned long long int * deps, const unsigned char * depsFlags, const unsigned char numCopies, const nanos_fpga_copyinfo_t * copies)`

The list of Nanos++ APIs and their details can be found here:

1.3.1 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];
```

(continues on next page)

(continued from previous page)

```
  }
  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);

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

1.3.2 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();
```

CHAPTER

TWO

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 is available here:

2.1 Mercurium FPGA Phase options

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

2.1.1 fpga_memory_port_width

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

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

2.1.2 force_fpga_periodic_support

Force enabling the periodic tasks support in all FPGA accelerators. This feature is only enabled in the FPGA 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
```

2.1.3 disable_unaligned_fpga_memory_port_width

[Experimental in pre-production] Disables 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=disable_unaligned_fpga_memory_port_width:1
```

Warning: This option will save resources in the logic of FPGA wrappers. However, that may be unsafe and result in bad application results due to data overwrite and/or data shift.

2.2 Binaries

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

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

2.3 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"
```

2.3.1 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"
```

2.3.2 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.

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"
```

2.4 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

THREE

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).

3.1 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:

3.1.1 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 spefific 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-max-pending-tasks [=]<integer>
     Number of tasks allowed to be pending finalization for an fpga accelerator (def:␣
↪4)
```

(continues on next page)

(continued from previous page)

```
   --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_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

FOUR

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.

4.1 Prerequisites

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

4.1.1 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
```

4.2 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).

4.2.1 Unpack the bsp

Unpack the bsp to create the petalinux project.

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

4.2.2 [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)`

4.2.3 [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
```

4.2.4 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.

4.2.5 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`

4.3 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>
```

4.3.1 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`.

4.3.2 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
```

4.3.3 [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

4.3.4 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.

4.4 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>
```

4.4.1 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 follwoing commant 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
```

4.4.2 Build the Linux system

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

```
petalinux-build
```

4.4.3 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

FIVE

FAQ: FREQUENTLY ASKED QUESTIONS

5.1 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

- genindex

INDEX

B

boot
    ultrascale, 14

C

compile
    OmpSs@FPGA, 8

D

develop
    OmpSs@FPGA, 1

F

FAQ, 20
    About OmpSs, 21
FSBL Fallback Mechanism, 16

M

Mercurium FPGA Phase options, 9

N

Nanos++ API, 5
Nanos++ FPGA Architecture options, 13

O

OmpSs@FPGA
    compile, 8
    develop, 1
    running, 11

P

Problem downloading Root FS, 16

R

running
    OmpSs@FPGA, 11

U

ultrascale
    boot, 14