# OmpSs@FPGA User Guide

***Release 2.5.1***

**BSC Programming Models**

**Apr 25, 2022**

# CONTENTS

**Note:** There is a PDF version of this document at http://pm.bsc.es/ftp/ompss-at-fpga/doc/user-guide-2.5.1/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 macros cannot be used within #pragmas as explained in OmpSs user guide (https://pm.bsc.es/ftp/ompss/doc/user-guide/faq-macros.html).
- The HLS source code generated by Mercurim for each FPGA task will not contain the includes in the original source file but the ones finished in .fpga.h or .fpga .
- The FPGA task code cannot perform general system calls, and only some Nanos++ APIs are supported.
- The usage of `size_t`, `signed long int` or `unsigned long int` is not recommended inside the FPGA accelerator code. They may have different widths in the host and in the FPGA.

## 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;
  }
}
```

### 1.2.3 localmem_copies

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

### 1.2.4 no_localmem_copies

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

### 1.2.5 localmem

Defines the memory regions that the FPGA task wrapper must catch in BRAMs. This creates a local copy of the parameter in the FPGA task accelerator which can be accessed faster than dispatching memory accesses. The data is

copied from the FPGA addressable memory into the FPGA task accelerator before launching the task execution. If the parameter is not labeled with the `const` modifier, the wrapper includes support for writing back the local copy into memory after the task execution. Both input and output data movements, may be dynamically disabled by the runtime based on its knowledge about task copies and predecessor/successor tasks. Usage example:

```
const unsigned int LEN = 8;

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

### 1.2.6 period

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

```
const unsigned int LEN = 8;

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

### 1.2.7 num_repetitions

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

```
const unsigned int LEN = 8;

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

## 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 )`
- `unsigned int nanos_get_periodic_task_repetition_num()`
- `void nanos_cancel_periodic_task()`

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];
  }
  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 force_fpga_periodic_support

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

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

### 2.1.2 fpga_check_limits_memory_port

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

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

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

### 2.1.3 fpga_ignore_deps_task_spawn

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

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

### 2.1.4 fpga_memory_port_width

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

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

### 2.1.5 fpga_memory_ports_mode

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

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

### 2.1.6 fpga_unaligned_memory_port

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

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

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

### 2.1.7 fpga_unordered_args

[Available in release 2.5.0] Controls whether the support for handling the task argument out of order must be implemented or not. By default it is enabled, the HLS generation remains equal to previous releases. Usage example to disable the support:

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

## 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-instrumentation-buffer-size [=]<integer + suffix>
     Maximum number of events to be saved from a FPGA task (def: 170)
```

(continues on next page)

(continued from previous page)

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

CHAPTER

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

# CLUSTER INSTALLATIONS

## 5.1 Ikergune cluster installation

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

### 5.1.1 General remarks

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

### 5.1.2 Module structure

The ompss modules are:

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

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

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

And for ARM 32bits:

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

To list all available modules in the system run:

```
module avail
```

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

### 5.1.3 Build applications

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

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

### 5.1.4 Running applications

#### Log into a worker node (interactive jobs)

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

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

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

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

For instance:

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

Then get the node that has been allocated for you:

```
squeue
```

The output should look similar to this:

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

Then, you can log into your node:

```
ssh ethNode-3
```

#### Load ompss kernel module

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

Kernel module binaries are provided in

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

Where `arch` is one of:

- `arm32`
- `arm64`

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

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

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

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

#### Loading bistreams

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

```
load_bitstream bitstream.bin
```

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

## 5.2 Sert installation

### 5.2.1 Access to the nodes

Sert nodes are behind a gateway. From the DAC gateway, connect via ssh to the sert cluster and then select `sert` remote host:

```
ssh gw.ac.upc.edu
```

Or if you are already inside DAC's internal network, you can just ssh into the login nodes:

```
ssh sert
```

You now should be logged into an entry node. .. note:

```
Entry nodes lack the resources needed to build or run applications.
You must log into a compute node in order to build an application.
See :ref:`sert-connect-compute` or :ref:`sert-connect-fpga`
```

There, you can access the scratch file system. This file system is shared accross all the node in the cluster.

It is mounted on

```
/scratch/nas/[N]/$USER
```

Where `N` is either 3 or 4

### 5.2.2 Building applications

#### Connecting to a compute node

In order to build applications, you must connect to a compute node. Since the installation is the scratch file system, it is accessible from all cluster nodes.

You can launch in interactive session using:

```
srun --cpus-per-task [N] --mem [M] --pty /bin/bash
```

Or if you need graphic applications:

```
srun.x11 --cpus-per-task [N] --mem [M]
```

Where `N` Os the number of CPUs to use and `M` is the requested memory in MB.

A failry large ammount of memory is needed to build a bitstream. Around 24GB should be enough for the installed alpha data devices. The CPU number is not critical as some phases in the bitstream generation process need to be run sequentially.

Using between 8 and 12 cores should be enough in most cases.

#### Setting up the environment

A installation from all components' git master branch is maintained by the CI system.

Users can use this installation by sourcing the initialization script:

```
source /scratch/nas/4/pmtest/opt/modules/init/bash
```

Then load the needed modules to set up the environment:

```
module load vivado/2020.1 ompss/x86_fpga/git
```

This will set up the environment in order to use the toolchain.

From this point, all tools needed to build and run applications should be available.

### 5.2.3 Running applications

#### Connecting to the FPGA nodes

There are only 2 FPGA node, which are `sert-1002` and `sert-1002` These nodes are in the *fpga* slurm partition. Use `srun` in order to open an interactive session into these nodes:

```
srun -A fpga -p fpga --nodelist sert-[1001,1002] --cpus-per-task [1-24] --mem-per-cpu␣
↪900 --pty /bin/bash
```

The `--mem-per-cpu` option is needed when asking many cpus, since slurm guarantees 2GB of RAM per asked CPU by default, which may be too much for fpga nodes as they have 32GB of memory each one.

To run an interactive session with X11 forwarding, execute the following command (remember to enable it previously when accessing through ssh in the gateway and the sert cluster with the -X option)

```
srun.x11 -A fpga -p fpga --nodelist sert-[1001,1002] --cpus-per-task [1-24] --mem-per-
↪cpu 900
```

**Note:** Do not use FPGA nodes to do general development. As FPGA access needs to be exclussive, it prevents other users from accessing the resources. FPGA nodes are only intended for running and debugging fpga applications.

#### Load bitstream

Once the bitstream is generated, you need the flash tool from AlphaData.

```
/scratch/nas/4/pmtest/opt/admpcie7v3_sdk-1.0.0/host/util-v1_8_0b2/proj/linux/flash/
↪flash program 0 path/to/bitstream
```

The bitstream files to load are the ones ending in .bit

#### Reboot sert-[1001,1002]

The flash application loads the bitstream in a flash memory, but it is not loaded on the FPGA until it reboots. To do so, you need access to the ipmi for the node you want to use. This script will do the work `reboot-sert.sh`

```
./reboot-sert.sh # PASSWORD [-force] where '#' is 1 or 2 for node 1001 or 1002␣
↪respectivelly
```

### 5.2.4 Considerations when developing applications

All communication is done through the PCI. The current implementation of xtasks uses a lock to give unique access to the PCI to only one thread. This means that only one thread can send or receive tasks to/from the FPGA. Therefore, using all threads of the node (24) will usually slow the application due to the abuse of locks. Use the `--smp-workers` and `--fpga-helper-threads` flags (`NX_ARGS`) to control the number of threads.

CHAPTER

SIX

# FAQ: FREQUENTLY ASKED QUESTIONS

## 6.1 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

## 6.2 How to keep HLS intermediate files generated by Mercurium?

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

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

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

#### Edit the HLS intermediate files and call AIT

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

the AIT command to invoke will be:

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

## 6.3 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.

- genindex

# INDEX