Deusto Course Guide

OmpSs@FPGA Team

December 12, 2018

Index

1

Course Environment

1.1 Hard Disk: Ubuntu system and code

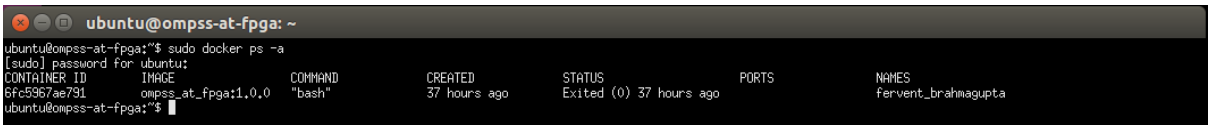
Ubuntu 16.04 running in the bootable Hard disk is prepared with a docker container that will help you to cross-compile to the Zynq 7000 (zybo). Next section explains how to access to this docker container and copy from/to files when you compile. Also, this ubuntu has the `minicom` application intalled to connect via serie with the zybo (explained later). Once you have compiled you can also copy data to the zybo using ssh (explained later).

1. Push `F9/F12` to make BIOS appear.
2. Boot from the external hard disk ubuntu, password: ubuntu.
3. Work in the `tutorial` directory
4. Original source code should be at this directory (or provided to you by USB if you are running your own system).

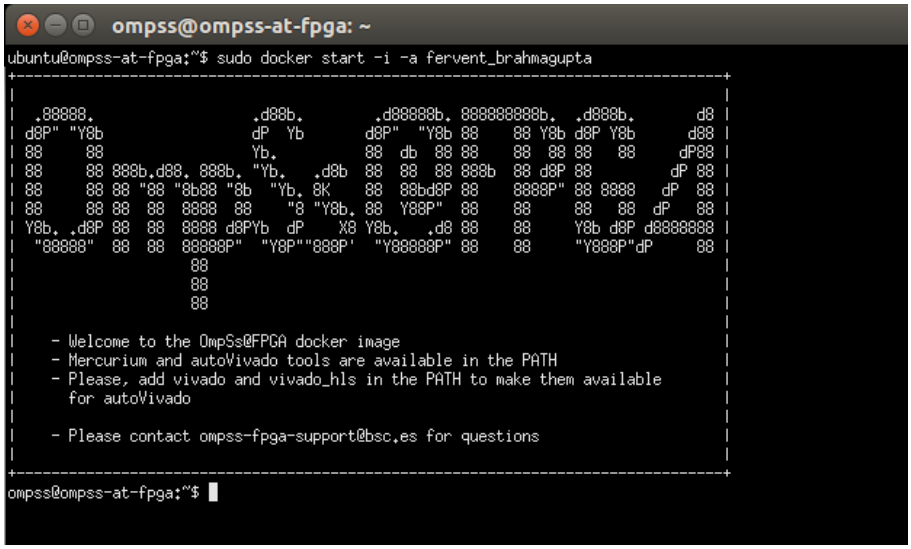
1.2 Docker: Cross-compilation environment

1.2.1 Login in the container

1. Docker container has been already created.
2. Run: `sudo docker ps -a`



3. Run: `sudo docker start -i -a name_container`



4. You can exit the container running exit. To `login in` again can repeat start and attach.

1.2.2 Sharing data between host Ubuntu system and docker cotainer

1. At `/home/ubuntu` docker container directory you have access to the Ubuntu 16.04 host file system (or the directory you have chosen if you run the docker image to create a new container).

2. You can `cp` from/to host directories to/from contanier directories, or use directly a host directory.

1.2.3 Cross-compiling

1. There is an example inside the container directory. You can look at it and based on the presentation we have provided you, you should be able to do a naive version. Indeed, the makefile with the course code is also prepared to compile and generate a OmpSs binary and the bitstream of the accelerators.

2. Open/Edit the Makefile of the hackathon code and figure out if you have to add any directive or compilation flag.

   - In order to estimate the performance of your accelerators before starting the time consuming process of generating the hardware, you can target VERSION_CODE=NAME make HLS-zybo .
   - The important flag in this target is the OmpSs@FPGA compilation flag `--to_step=HLS`, which will make the compilation to stop after the Vivado HLS project is generated. Later we will explain how to analyze the estimated performance.
   - `VERSION_CODE` is an environment variable that can be defined at command line, and you can use `#ifdef` in the source code to activate or deactivate parts of the source code.

3. Open/edit the program of the course code and follow the instructions explained below to start the Hands-on.

4. In order to partially or completely compile to generate the bitstream, the computer should be connected to the network (wireless or not).

5. Compilation take a while. Maybe you can use `nohup` to run the compilation so that you an continue working in the following challenge meanwhile it is compiling.

6. Remember:

   - For partial compilation:

```
VERSION_CODE=_V9_ nohup make  HLS-zybo >& output_V9.txt &
```

- For complete compilation:

```
nohup make  bitstream-zybo >& output.txt &
```

1.2.4 Vivado HLS analysis

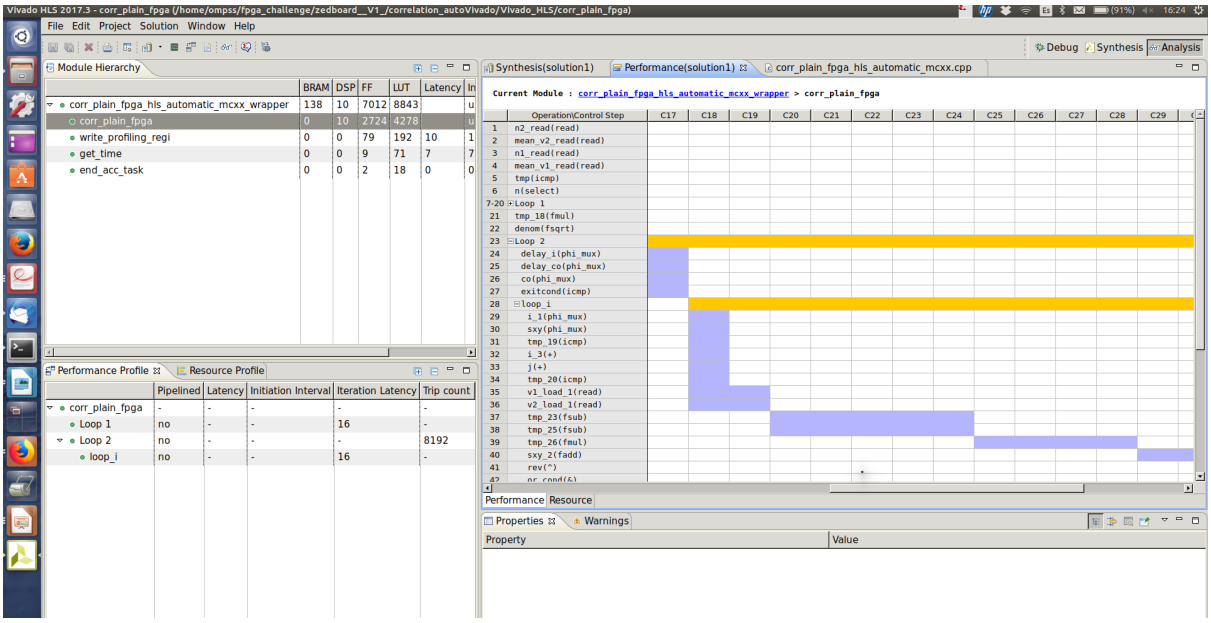
1. After partial compilation upto step HLS, you can analyze the vivado hls project.

```
VERSION_CODE=_VX_ nohup make  HLS-zybo >& output_VX.txt &
```

2. Run, for instance:

```
vivado_hls -p zybo__VX_/correlation_autoVivado/Vivado_HLS/accelerator_name
```

3. Then, click Analysis. Look for the function accelerated. You should see something similar to:

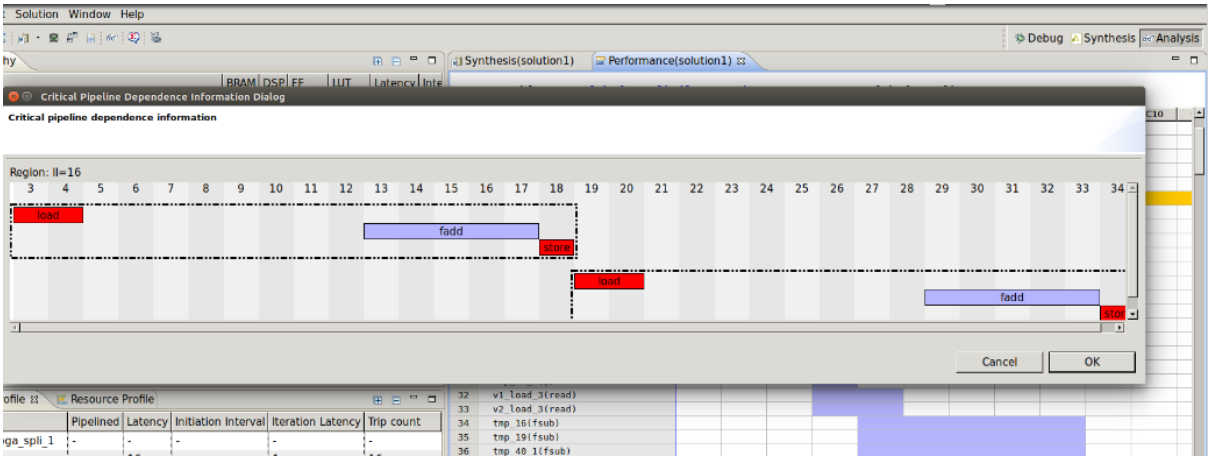


4. The important thing here is the `II` one can achieve per iteration. At bottom left, you can see the names of the different loops in the code. If you add labels to your code, those labels will appear here.

5. In the case of the `plain_fpga` there is not pipeline at all (we didn't specify anything), and the latency we pay per iteration is 16 cycles per interation!!!

6. At top right, you can also observe the cycles per iterations that each loop has, and the high level pnemotecnic of the operations.

   - Yellow: indicates overall latency per iteration
   - Red: indicates conflicts. You can click the red cycles to see where the conflits are.



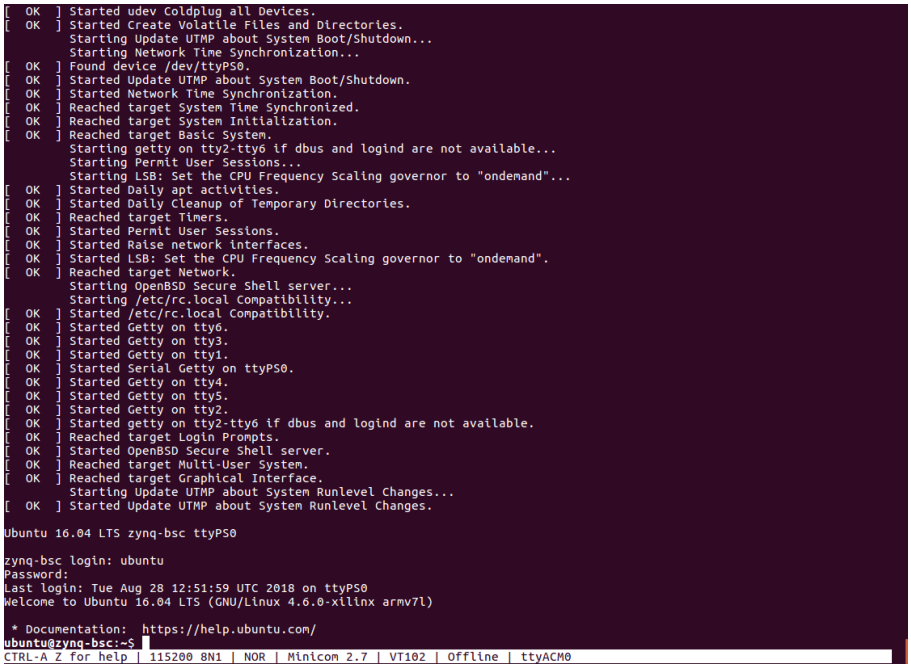
1.3 Zynq Board System

1.3.1 Boot from SD

1. **BE SURE your board is turned off**
2. Insert the microSD card in the board
3. Connect the board to the power supplier
4. Connect the microUSB side of the microUSB-USB cable to UART connection.
5. Connect the USB side of the microUSB-USB cable to the computer.
6. Turn on the board
7. Run `dmesg` in your computer and look for something like: "`....  ttyACM0:  USB ACM device...` " or "`....  ttyUSB0  ....  `" . This is what you have to use to setup the `minicom` port.

1.3.2 Minicom Connection

1. Assuming that you have seen `ttyACM0` in the `dmesg` log, run "`sudo minicom -D /dev/ttyACM0 -b 115200`"
2. At `login` use ubuntu:ubuntu .



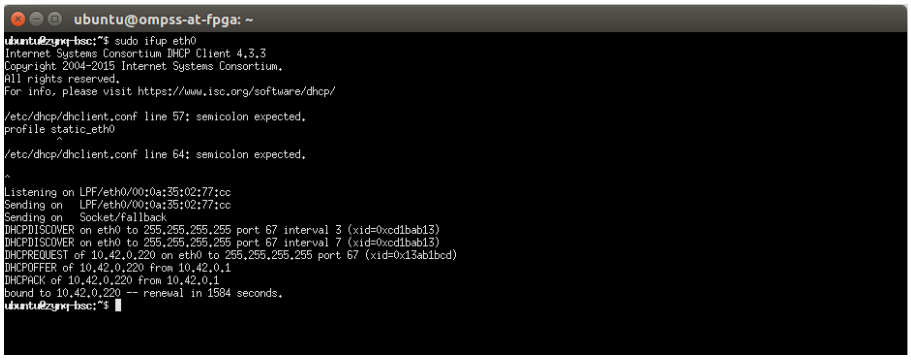
3. The Zynq System is also running an Ubuntu 16.04.

4. Look at the date. If this is not the current one, you can run `sudo date -s "DD MMM YYYY HH:MM:SS"` so that you can set the current day and hour. DD stands for Day, MMM stands for the first three letter of the month, YYYY stands for the year, and HH:MM:SS stands for hour, minutes, seconds.

1.3.3 Ethernet connection

You can access through ethernet connection using ssh to connect to the board or scp to copy data to/from the host. This will be useful to transfer files between host Ubuntu system and Zynq system.

1. First, from the minicom connection run `sudo ifup eth0`. Usually, it is `10.42.0.220`.



- Login: The `IP` obtained in previous steps is the one you can use to ssh or scp.
- Assuming `IP: 10.42.0.220` run `ssh -X ubuntu@10.42.0.220` if you want to remotely login
- Copy host to zybo: `scp files ubuntu@10.42.0.220:path`
- Copy zedobard to host: `scp ubuntu@10.42.0.220:path/file`

1.3.4 Application Execution

1. First, do a complete compilation in the Docker container:

```
nohup make  bitstream-zybo>& output.txt &
```

2. Then, transfer the following files from the Host system to the Zynq system:
   - ARM binary file, at the current compilation directory.
   - Bitstream (.bin) : this file should be under the `zybo/correlation_autoVivado/` and it is the bitstream that we have to write in the FPGA device.
   - name.xtasks.config : this file should be under the `zybo/correlation_autoVivado/` and it is the configuration runtime file that is necessary when running the program.
3. Login into the Zynq System
4. Do the following steps:
   - Run: `cat bitstream.bin > /dev/xdevcfg`
   - Run: `NX_ARGS="--summary" ./program arguments`

1.4 Shutdown

1. Correctly shutdown the machine with `sudo halt` or `sudo shutdown now`.

2

First Day

2.1 Cross-Correlation

Tides are sea level changes caused by the gravitational influence of the Sun and the Moon.

Locations with similar tide waveforms will attain higher levels of data correlation. Correlation will be close to 1.0 if the signals are in synchrony, while correlation will be close to -1.0 if signals are in opposition. Locations with unrelated tide waveforms will get lower levels of data correlation (lower values, positive or negative, but closer to 0.0). Therefore, using a cross-correlation we want to check whether tide wafeforms are equivalent across the planet Earth. The UNESCO Intergovernmental Oceanographic Commission (IOC) publishes online nearly-live data from various providers. We provide you with some input data that you can use to check the correctness of your code.

Are the tide waveforms equivalent in the various places on Earth? We have the `correlation.c` program that computes the correlation between the data stored in two TXT files. Compile and run this application. You will see that it is provided with two TXT files as arguments. Use, for example:

```
./correlation datafiles/ibiz-30.txt datafiles/tarr-30.txt
    ...
    correlation is 0.90
    ...
./correlation datafiles/ibiz-30.txt datafiles/barc-30.txt
    ...
    correlation is 0.86
    ...
```

Let's see how we can implement such a function... in the FPGA. Look at the source code of the application (`correlation.c`) and try to understand how it works. Now, you should be ready to proceed with the first Exercise.

2.1.1 OmpSs@FPGA porting

Implement the correlation function for the FPGA inside the correlation.c file. In order to do so, let's see the interface of the function:

```
void corr_fpga(float * v1, float mean_v1, int n1,
               float * v2, float mean_v2, int n2,
               float * co, int * delay_co);
```

The `corr_fpga` function receives two arrays of floating point values, the mean values precomputed from them, and the numbers of elements contained on each of them. As a result, the function returns the correlation between the values of the two arrays, and the delay computed.

The function should compute the correlation between these two arrays of data. Each array has been loaded from one of the TXT files provided as parameters, so the final result will be the correlation between the waveforms in both locations.

When compiling the source code to the bitstream, the compilation process will take 30 minutes, also depending on the host computer.

But... although the Makefile is already prepared to generate the application binary and the bitstream for the FPGA, it is not prepared yet to just reach the HLS step. In order to avoid the long compilation time now, we ask you to just analyze the performance estimation of the accelerator, before generating the full hardware, by compiling upto the HLS synthesis.

Then, compile the application until the HLS synthesis step (look at the information above about the environment, compilation, analysis and execution). You should use:

```
VERSION_CODE=V0 make HLS-zybo # it will fail, please keep reading
```

Open `vivado_hls` project using:

```
vivado_hls -p zybo_V0/correlation_autoVivado/Vivado_HLS/accelerator_name
```

And answer the following questions:

- What is the iteration latency of the loop-i loops?
- What is the pipeline achieved in each loop?
- What is the Vivado directive that you should use in order to force any `pipeline` in the loops?

2.1.2 Overlap iterations

Now you will need to annotate the function with the OmpSs target and task directives (see the annotation of the function `corr_plain` as an example), and add an HLS directive (`#pragma HLS PIPELINE II=1`) in the innermost loop i (the nested one under delay loop).

Then, compile the application until the HLS synthesis step (look at the information above about the environment, compilation, analysis and execution). You should use:

```
VERSION_CODE=V1 make HLS-zybo # it will fail, please keep reading
```

Open `vivado_hls` project using:

```
vivado_hls -p zybo_V1/correlation_autoVivado/Vivado_HLS/accelerator_name
```

And answer the following questions:

- What is the iteration latency of the loop-i loops?
- What is the pipeline achieved in the innermost loop i (nested loop within delay loop)?
- If there are n iterations in the first loop-i and the innermost loop-i (under dealy loop), what is the overall expected latency for each loop-i?

Observe that loop i's are taking initiation intervals (II) larger than one. This is mostly due to resource conflicts caused by the conditional sentence in the body of the for-loop, and/or the accumulation of values to the same variables (sx, sy, sxy), between iterations, due to the latency they have. This fact will hamper the possibilities that the HLS compiler uses pipelining to optimize the loop. Note that the ideal situation is when the code achieves an II=1 or equivalent in the loop i.

On the top-right window of the `vivado_hls` view you can see red conflicts. Double click on them to see how they affects to the II.

Compile the application to generate the bitstream for this version of the code using:

```
VERSION_CODE=V1  nohup  make bitstream-zybo-i >& output_V1_instrumentation.out &
# Compiled with instrumentation to use later
# Remind: it may take more than 20 minutes
```

Meanwhile it is compiling... we can pass to the next point.

2.1.3 Overlap iterations and Loop Latency

Change the source code to increase the distance between two accumulations for those two loop i's. This can be usually done using several accumulators (or a vector of accumulators, NOT THE FULL SIZE OF THE INPUT VECTOR), and splitting (strip-mining) each loop i. If you have to add new loops, it is good to add labels to them, to identify them in the Vivado reports.

Recompile upto HLS synthesis again, and analyze with Vivado HLS the performance of loop i's.

```
VERSION_CODE=V2 make HLS-zybo
```

Open `vivado_hls` project using:

```
vivado_hls -p zybo_V2/correlation_autoVivado/Vivado_HLS/accelerator_name
```

And answer the following questions:

- What is the iteration latency of each loop-i loops?
- Analyze the resource conflicts to know the reason of those values of achieved II. You can also analyze the output of OmpSs@FPGA compilation (verbose) so that you can see the Vivado HLS messages.
- Have you reduced the overall loop latency? Can you reduce reduce the latency improving or unrolling other loops?

Once previous full compilation has finished, compile the application to generate the bitstream for this version of the code using:

```
VERSION_CODE=V2 nohup make bitstream-zybo >& output_V2.txt
# Remind: it may take more than 20 minutes
```

Meanwhile it is compiling... we can pass to the next point.

2.1.4 Avoid Memory Conflicts and General Optimizations

Based on the previous analysis, try to reduce the latency of the loops with large latency by using `#pragma HLS unroll`, and figure out if after doing that you should use

`#pragma HLS ARRAY_PARTITION variable=name type factor=XX`

due to memory accesses conflicts. The `XX` factor should be equal or larger than latency of the conflict, and usually less or equal to the number of accumulators used to make independent operations.

Recompile upto HLS synthesis again, and analyze with Vivado HLS the performance of the application.

```
VERSION_CODE=V3 make HLS-zybo
```

Open `vivado_hls` project using:

```
vivado_hls -p zybo_V3/correlation_autoVivado/Vivado_HLS/accelerator_name
```

And answer the following questions:

- What is the iteration latency of each loop-i loops?
- Have you realized that the `select` operation helps us to improve the II? It could be used in the first loop-i if enough resources were avaliable.

Once previous full compilation has finished, compile the application to generate the bitstream for this version of the code using:

```
VERSION_CODE=V3  nohup  make bitstream-zybo >& output_V3.out &
# Remind: it may take more than 20 minutes
```

2.1.5 Cross-correlation performance

Compute the speedup you have obtained, comparing the performance to the sequential version and previous versions. In order to run the correlation versions you have obtained you will have to:

1. Load the bitstream:

```
cat correlation.bin > /dev/xdevcfg
```

2. Run the program:

```
./correlation datafiles/ibiz-30.txt datafiles/tarr-30.txt
```

3. Repeat for each version

2.1.6 Software Optimizations and other tuning

Based on the previous analysis, we can try to add `if` statement to reduce the II achieved in the first loop-i. Also, we could reduce the number of innermost loop iterations of the loop-i (under dealy loop). Do the modifications that you consider to potentially improve the overall performance.

Recompile upto HLS synthesis again, and analyze with Vivado HLS the performance of loop i's. And answer the following questions:

- Have you reduced the II value of each loop-i? Why? Do you have enough hardware resources?
- Recompile it upto HLS synthesis again for zedboard. For a zedboard, will you obtain a benefit from introducing a `if` statement and reducing the innermost loop-i iterations?

2.2 Hardware Instrumentation

OmpSs@FPGA allows to do a by default basic instrumentation of the copies and execution of the accelerators. The execution of this instrumented code, in case of activating the instrumentation at execution time, will generate a Paraver trace that you can visualize with the Paraver tool. We have prepared the Makefile to compile the code to have this instrumentation. Also, there is a very recent instrumentation feature that allows the programmer to introduce user events inside the accelerator to show up possible problems in the code.

2.2.1 Default Instrumentation

Setup the instrumentation environment:

```
export EXTRAE_CONFIG_FILE="extrae.xml"
```

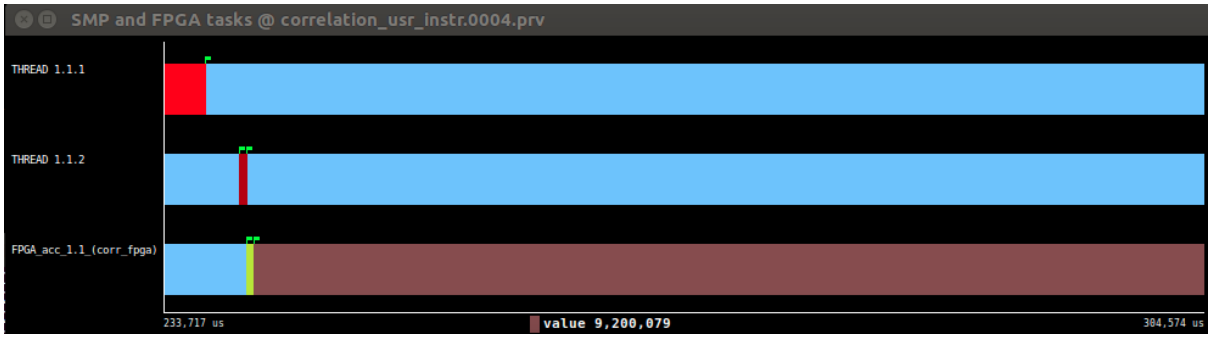
Run the first version of the correlation program running on the FPGA:

```
NX_ARGS="--instrumentation=extrae" ./correlation datafiles/ibiz-30.txt datafiles/tarr-30.txt
```

This will generate a Paraver trace `correlation.prv,pcf,raw`. Copy those files to the docker container and run Paraver:

```
wxparaver correlation.prv
```

Use the configuration file `smp_and_fpga_tasks.cfg` to see the tasks executed in the SMP and in the FPGA. In fact, just one correlation task is executed in each device.



You can zoom-in in the execution of the accelerator to see the copy in and out of the data at the beginning and the end of the accelerator execution.

2.2.2 User Instrumentation

OmpSs@FPGA allows the programmer to insert user instrumentation to the hardware accelerators. Although there are some limitation on the number of events to be evicted by the accelerator, it provides to the programmer with a very useful tool. In order to insert user events we suggest to register the key event in the main program using, for instance:

```
nanos_instrument_register_key_with_key(LOOP_I, "delay loop", "Delay loop iteration", true);
```

This call register event key "dealy loop" with label "Delay loop iteration" (what will appear in the Paraver trace), and key value `LOOP_I`. The key value ca be defined in the ".fpga.h" header application file. For instance:

```
#define LOOP_I 10000
```

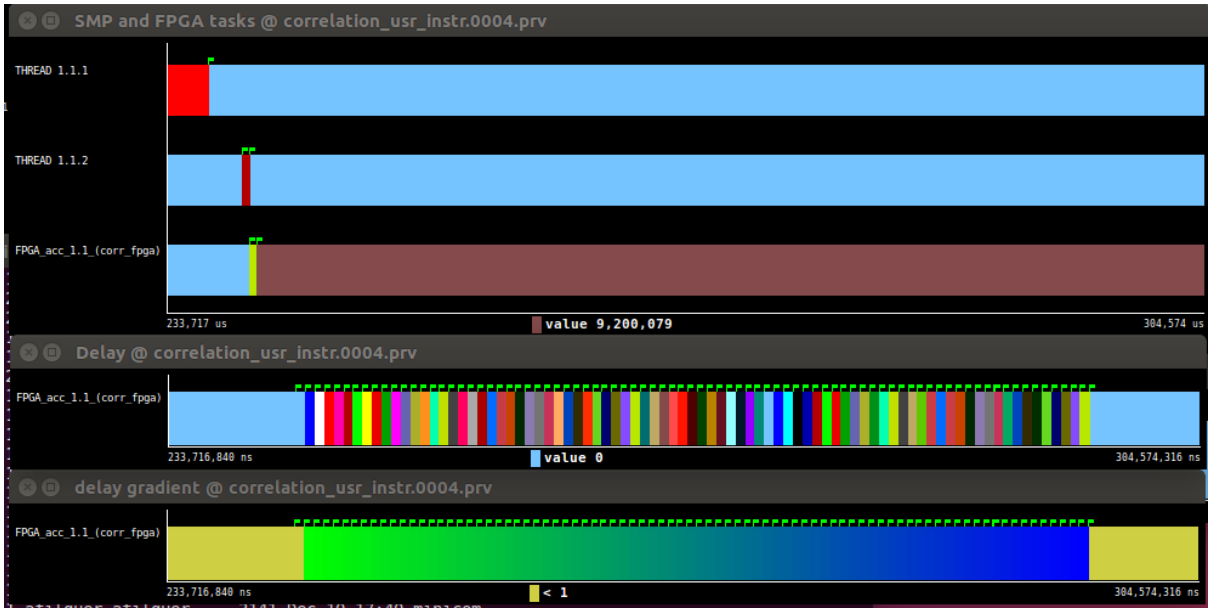
Then, in order to mark a burst (with begin and end) in the accelerator state, we can use the following calls. In the example, we can use burst begin and end to determine the beginning and the end of the delay iteration:

```
nanos_instrument_burst_begin(LOOP_I, delay+maxdelay);
#For the beginning of the burst

nanos_instrument_burst_end(LOOP_I, delay+maxdelay);
#For the beginning of the burst
```

Note that we add `+maxdelay` to delay to avoid negative values in the events.

Use the configuration file `xcorr_delay_loop.cfg` to see the tasks executed in the SMP and in the FPGA. In fact, just one correlation task is executed in each device.


SMP and FPGA tasks @ correlation_usr_instr.0004.prv
THREAD 1.1.1
THREAD 1.1.2
FPGA_acc_1.1_(corr_fpga)
233,717 us
value 9,200,079
304,574 us
Delay @ correlation_usr_instr.0004.prv
FPGA_acc_1.1_(corr_fpga)
233,716,840 ns
value 0
304,574,316 ns
delay gradient @ correlation_usr_instr.0004.prv
FPGA_acc_1.1_(corr_fpga)
233,716,840 ns
< 1
304,574,316 ns

3

Second Day

3.1 Matrix Multiply

Matrix Multiply is a common kernel that is used in several applications like the `kmeans`. The typical code of the matrix multiply of two matrices `A` and `B` follows:

```
void matmul(elem_t *a, elem_t *b, elem_t *c) {
   unsigned int i, j, k;

   for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
         elem_t sum = c[i*N+ j];
         for (k = 0; k < N; k++) {
            sum += a[i*N+ k] * b[k*N+ j];
         }
         c[i*N+ j] = sum;
      }
   }
}
```

We provide you the naive source code of a Tile Matrix Multiply that you can use as baseline to compute the speedup.

3.1.1 Overlap iterations and Avoiding Memory Conflicts

In the first step we ask you to port this kernel `matmulBlock` to OmpSs@FPGA so that you create one accelerator of this kernel function. Based on the previous exercises with cross-correlation, insert the necessary `Vivado HLS` pragmas to achieve a good II (II=4) in loop j. Remember that you may want to label the loops, and also do partitioning of the matrices.

Recompile upto HLS synthesis again, and analyze with Vivado HLS the performance of loop i's.

```
VERSION_CODE=MXM make HLS-zybo
```

Open `vivado_hls` project using:

```
vivado_hls -p zybo_MXM/matmul_autoVivado/Vivado_HLS/accelerator_name
```

To be sure that you are achieving a good II. Once you have obtain a good expected performance, compile the application to generate the bitstream for this version of the code using:

```
VERSION_CODE=MxM  nohup  make bitstream-zybo >& output_MXM.out &
# Remind: it may take more than 20 minutes
```

3.1.2 Overlaping Communications and Computations: Dataflow

Following the same idea of the presentation slides, perform a overall modification of the kernel `matmul` so that it can do a dataflow implementation of a `32x32` blocking matmul. Clauses `copy_deps no_localmem_copies` allows the programmer to specify that wants to allocate memory in kernel memory space, that the runtime performs the copy from/to user to/from this kernel memory space for input and output respectively. Unlike previous versions, the automatically generated wrapper of the function will not declare local variables neither generate copies from/to host memory to/from local memories. In this case, it is reponsibility of the programmer to read and write the input and output data respectively. That helps the programmer to use the `DATAFLOW Vivado HLS` pragma to try to overlap computation and communications.

Recompile upto HLS synthesis again, and analyze with Vivado HLS the performance of loop i's.

```
VERSION_CODE=MXM_DATAFLOW make HLS-zybo
```

Open `vivado_hls` project using:

```
vivado_hls -p zybo_MXM_DATAFLOW/matmul_autoVivado/Vivado_HLS/accelerator_name
```

To be sure that you are achieving a good II.

Once you have obtain a good expected performance, compile the application to generate the bitstream for this version of the code using:

```
VERSION_CODE=MxM_DATAFLOW  nohup  make bitstream-zybo >& output_MXM.out &
# Remind: it may take more than 20 minutes
```

3.1.3 MxM performance

Compute the speedup you have obtained, comparing the performance to the sequential version and previous versions for a problem size of `1024x1024`. In order to run the matmul versions you have obtained you will have to:

1. Load the bitstream:

   ```
   cat matmul.bin > /dev/xdevcfg
   ```

2. Run the program:

   ```
   ./matmul 1024 1
   ```

3. Repeat it for each version

3.1.4 FPGA Blocking

Zybo board has a SMP with two Cortex-A9, at 666MHz. For cases where the accelerator is fast enough, there may happen that the cores are not fast enough to feed the accelerator/s. The reason is that there is an overhead in the creation of the tasks and copy of the input/output data that could be hiden/reduced if the accelerator could perform the full matrix multiply from insde, without SMP neither runtime intervention. However, there may be the need of having the input and output dependences, and that the runtime takes care of preparing the input/output memory data to allow the accelerator to copy in/out the data from inside the FPGA.

We ask you to create a function call `matmulFull` with the part of the code of the main program that perform the tiled matrix multiply:

```
void matmulFull(int msize, elem_t *a, elem_t *b, elem_t *c) {
   unsigned int const m2size = msize*msize;
   for (unsigned int i = 0; i < msize/BSIZE; i++) {
      for (unsigned int j = 0; j < msize/BSIZE; j++) {
         unsigned int const ci = j*b2size + i*BSIZE*msize;
```

```
            for (unsigned int k = 0; k < msize/BSIZE; k++) {
               unsigned int const ai = k*b2size + i*BSIZE*msize;
               unsigned int const bi = j*b2size + k*BSIZE*msize;
               matmulBlock(&a[ai], &b[bi], &c[ci]);
            }
         }
      }
   }
```

Then, add the following directives to the function:

```
#pragma omp target device(fpga) copy_deps no_localmem_copies num_instances(1)
#pragma omp task in([msize*msize]a, [msize*msize]b) inout([msize*msize]c)
```

Note that we specify the overall size of the input matrices `a,b` and input and output matrix `c`, since the runtime will take care of allocating and copying data.

On the other hand, you should remove the task and target device pragmas from the `matmulBlock`.

Once you have obtain a good expected performance, compile the application to generate the bitstream for this version of the code using:

```
VERSION_CODE=MxM_DATAFLOW_BLOCKING  nohup  make bitstream-zybo >& \
            output_MXM_DATAFLOW_BLOCKING.out &
# Remind: it may take more than 20 minutes
```

Compute the speedup you have obtained, comparing the performance to the sequential version and previous versions for a problem size of `1024x1024`. In order to run the matmul versions you have obtained you will have to:

1. Load the bitstream:

   ```
   cat matmul.bin > /dev/xdevcfg
   ```

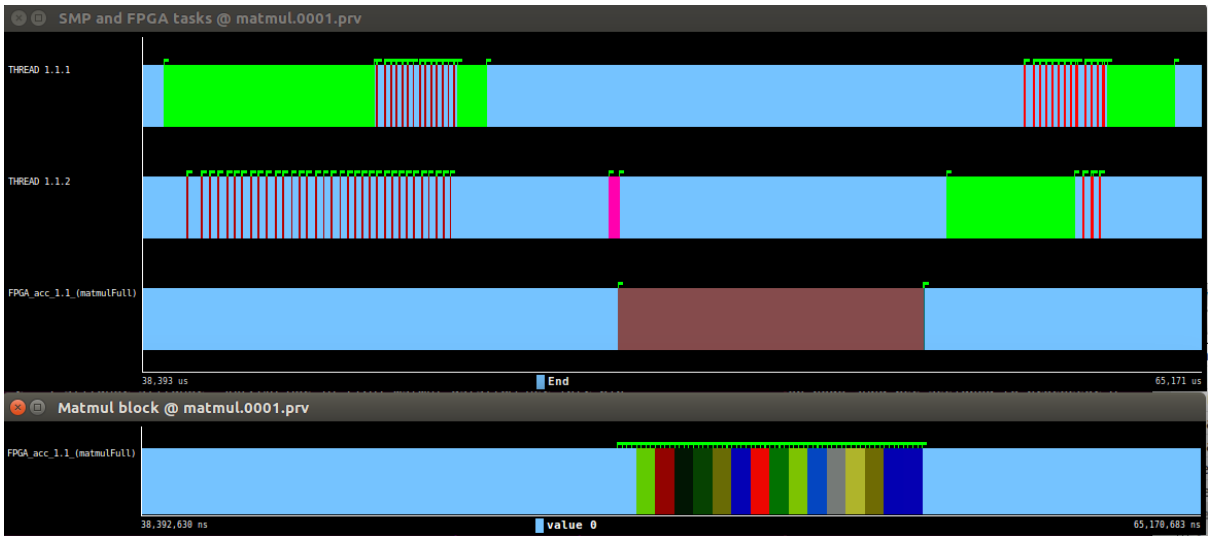
2. Run the program:

   ```
   ./matmul 1024 1
   ```

3. Repeat it for each version

3.1.5 User Instrumentation

Introduce user events to the blocking dataflow version of the matrix multiply so that key value is `ci` and the beginning and end of each burst is just before and after the matmulBlock call. Then, generate a trace and try to understand what you are seeing there.



Maybe you can try to change loops i, j and k.