SimplePL: an OpenCL-like runtime system for HLS OpenCL Kernels

Grammatopoulos Athanasios Vasileios
Technical University of Crete

November 9, 2017

Abstract

SimplePL is a runtime system that features an OpenCL-like application programming interface to interact with high level synthesis OpenCL kernels on your programmable logic. The system consists of a C Library that interacts with Linux kernel modules. SimplePL simplifies the development of Linux applications that use HLS OpenCL kernels. Applications can program the PL with pre-compiled bitfiles on demand and call it's HLS kernels IP cores through the library in a similar way an OpenCL application calls OpenCL kernels. SimplePL can schedule the workload on the available IP cores, manage the physical and the virtual addresses of the PL's units, create and manage memory buffers, handle the low level communication between the application and the PL and all these by hiding all the complexity of the programmer.

Keywords: SimplePL, FPGA, OpenCL, HLS, Vivado

Abbreviations

API	*Application Programming Interface*
ASIC	*Application Specific Integrated Circuit*
CPU	*Central Processing Unit*
CUDA	*Compute Unified Device Architecture*
FPGA	*Field-Programmable Gate Array*
GPGPU	*General-Purpose computing on Graphics Processing Units*
GPU	*Graphics Processing Unit*
HDL	*Hardware Description Language*
HLL	*High-Level Language*
HPC	*High Performance Computing*
ICD	*Installable Client Driver*
IP core	*Intellectual Property core*
JSON	*JavaScript Object Notation*
LLVM	*Low Level Virtual Machine*
OS	*Operating System*
RTL	*Register Transfer Logic*
SDK	*Software Development Kit*
SIMD	*Single Instruction Multiple Data*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very High Speed Integrated Circuit*

Acknowledgment

I would like to thank my supervisor Dr. Ioannis Papaefstathiou for his mentoring and guiding on my diploma thesis. Furthermore I would like to thank Ph.D Paulos Malakonakis assistant of the Microprocessor and Hardware laboratory for pointing me on the right direction to search and for the support he and the lab gave me. Last but not least, I would like to thank my family and my friends, who supported me all these year through my university studies and helped me became the man I am today.

Contents

1 Introduction

This work consists of two parts. The first part, our main work, describes our SimplePL system and its ecosystem. On a small second part, we provide our results on a number of benchmarks we run on OpenCL devices.

1.1 SimplePL

1.1.1 Performance

We always want the most from our computers, we want the maximum performance. We want to speed up the performance of our system, not always in the traditional way. We can speedup our programs by running them on a faster system (ex. on a faster hardware), we can get speedup by running them on parallel if that's possible (ex. using multi-threading), or we can run them with no time speedup at all, but with less power cost, that's what we call performance per watt.

In general, we seek to get a better performance from our systems. To do so, we combine different architectures and technologies in one system. By doing this, we can execute one application on different parts of our system to exploit each part's advantages and avoid another's disadvantages.

On a modern desktop computer system, to solve a heavy computing loads, we have a CPU working with the help of a GPGPU and we call this heterogeneous computing. We program such applications using C/C++ (mostly) and 2 popular C-like languages / application programming interfaces / frameworks, the CUDA by Nvidia and the OpenCL by Khronos Group.

On the other hand, to increase the performance of a system, we use to develop applications on FPGA boards or create ASIC boards, if it is possible. An FPGA or an ASIC board, many times, can increase the performance of an application and do so with lower power usage too. But such implementation costs a lot to be develop (because of the complexity of such implementations), thus the industry most times avoids them due to the development costs and the special development requirements.

1.1.2 FPGA tools for HLL

What changed the last years are the tools that can produce HDL code from high-level language code became better and also took advantage of the OpenCL standard. By using programs like Xilinx's Vivado HLS we can write C/C++/OpenCL code and synthesize it for use on our FPGA. In addition to that, frameworks like Xilinx's SDAccel let us use FPGA boards as an OpenCL device, but only some special boards are supported (no embedded FPGA board support).

The idea of using an FPGA as an OpenCL device is really good. OpenCL has a good standard and many programmers are already familiar with the language. Many programs use OpenCL to increase their performance and many popular libraries (like BLAS) was already been ported on OpenCL. If there was a way to easily port such libraries on our embedded FPGA boards, many applications would be able to be run on low-power embedded FPGA boards. So the logical next step is to give the developers tools to develop OpenCL applications on an FPGA, without limiting them on a few boards.

Since we were using Vivado HLS to port our OpenCL kernels on our FPGAs boards, we wanted to start porting hole OpenCL libraries, what we were interested in, on our embedded FPGAs boards. Such libraries are quite popular and many applications are using the to accelerate their computations. By porting them we could provide the opportunity to the applications that use them to run on an FPGA and hopefully achieve better performance per watt.

1.1.3 The need of a library

While we were trying to do so, we came across many problems that could be solved with the use of a library that could handle all the needed software to hardware connections, that is our libraries interactions with the IP cores inside the FPGA. Those problem could be solved by writing specific coded, inside each library's code, but since most of the libraries were complex, this process could require to re-write almost all the library's code. Thus, since we had to solve all these connection problems, we decided to focus on a library that handles them. By doing so, we would be able to port more libraries easier and faster.

The development of such a library is not trivial. The library had to solve those connection problems with the hardware in the back-end and on the same time provide an easy API front-end to the developers. Additionally, the usage and the development should be easy to use, as this was the main concept of developing in HLL for FPGAs. It was clear to us, that we could use an OpenCL-like model to describe our FPGAs as OpenCL-like devices and to provide an OpenCL-like API for the developers.

These are the reasons why we created SimplePL. Our OpenCL-like runtime system for HLS OpenCL kernels. To provide an easy and simple to use system for applications that want to use HLS OpenCL kernels.

1.2 OpenCL Benchmarks

1.2.1 OpenCL implementations

The OpenCL run on OpenCL supported devices with the help of an OpenCL ICD. Each OpenCL implementation has its own ICD and may execute the OpenCL kernels in any way it wants, as long as, it supports the OpenCL standard. Those ICD of course may or may not have support for Windows, Linux or Macintosh systems and usually support specific OpenCL devices.

Many GPU and CPU vendors, most of them, has released an OpenCL SDK with ICD, to support the execution of OpenCL on their devices (ARM, AMD, INTEL, NVIDIA). As a matter of fact, each one may yield different performance for different OpenCL tasks on each variation of the CPU and GPU.

1.2.2 Benchmarking OpenCL tasks

We wanted to test, how fast the OpenCL framework can execute many of its tasks. For this reason, while we were developing our SimplePL system, we run a number of benchmarks on many OpenCL devices, from different architectures, on different OS and on different systems. On those benchmarks we developed a small OpenCL application that run many times the typical OpenCL program flow to executed a simple OpenCL kernel. During those runs we were recording the timings of our interest and log them.

We wanted to have at our disposal real OpenCL applications timings on our OpenCL devices. We were not interested on the performance of the kernel execution, as this is related to the kernel that is executed. We were mostly interested on the tasks prior and after the kernel execution, like data transfers and source code build. These operations may some times be too small that can be ignored but for complex kernels or for a big number of data, it can turn out to be significant.

1.2.3 Results usage

Later we could use those recorded timing to compare them with similar tasks timings on our FPGAs, since our SimplePL system implements an OpenCL-like execution model. Our main concern was over the timing difference between downloading a bitfile on the FPGA versus the OpenCL source compile time. Of course, we were aware that, as the performance of many task are related to the speed of the hardware, our FPGA development board had disadvantage.

Additionally, as I personally had little experience with OpenCL applications but I was quite good in C/C++, this was an opportunity to experiment further with the OpenCL ecosystem. We also wanted to get familiar with the ICD drivers implementations and how the OpenCL headers handle multiple ICDs on one system.

We included these results on this document and we used some of them as a comparison between SimplePL on FPGA and OpenCL on GPU/CPU.

2 Introduction to OpenCL on FPGAs

In this section we will briefly explain the OpenCL model and how the model can be implemented on different architectures. Specifically we will explain the components that the model define and how these components match to the components of a GPU or a CPU to support the OpenCL standard. Moreover we will overview how the industry implements nowadays these components on FPGA boards to increase the performance or the power efficiency.

2.1 The OpenCL programing model

OpenCL is an open standard for parallel computing on heterogeneous platforms currently maintained by the Khronos Group. The OpenCL programing model describes a **compute device** with a number of **compute units**. Each compute unit consists of several **processing elements**. In addition, an OpenCL compute device has access to a memory (shared or not) with a four level hierarchy (global, constant, local and private). The hierarchy may not be implemented on hardware. The system (may be a computer) that manage this OpenCL device is referred to, by the OpenCL model, as the **host**. A host can have multiple OpenCL devices.

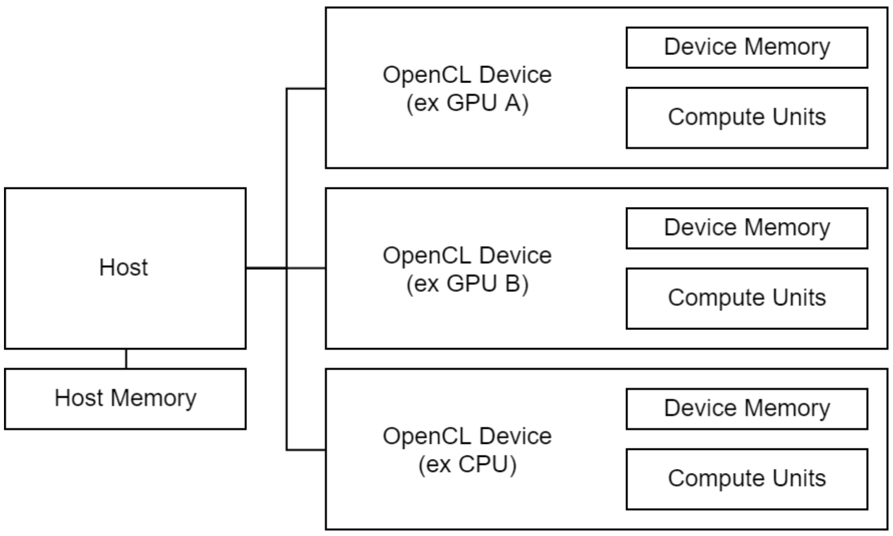


Figure 2.1: The OpenCL Model

Given a system with such a compute device, an application which runs on the host can use the OpenCL API to run a kernel (an OpenCL function) on an OpenCL device and across its multiple compute units. The OpenCL kernel is written in OpenCL and describes the work of a processing element. Before the kernel execution, all data to be processed need to be copied from the host's memory to the device's memory and after the executions the results need to be copied from the device's memory to the host's memory.

OpenCL kernels are usually compiled during the execution of the application by an OpenCL driver (usually a driver form the Vendor of the OpenCL device). This functionality lets the applications support multiple OpenCL devices without the need to produce multiple binaries.

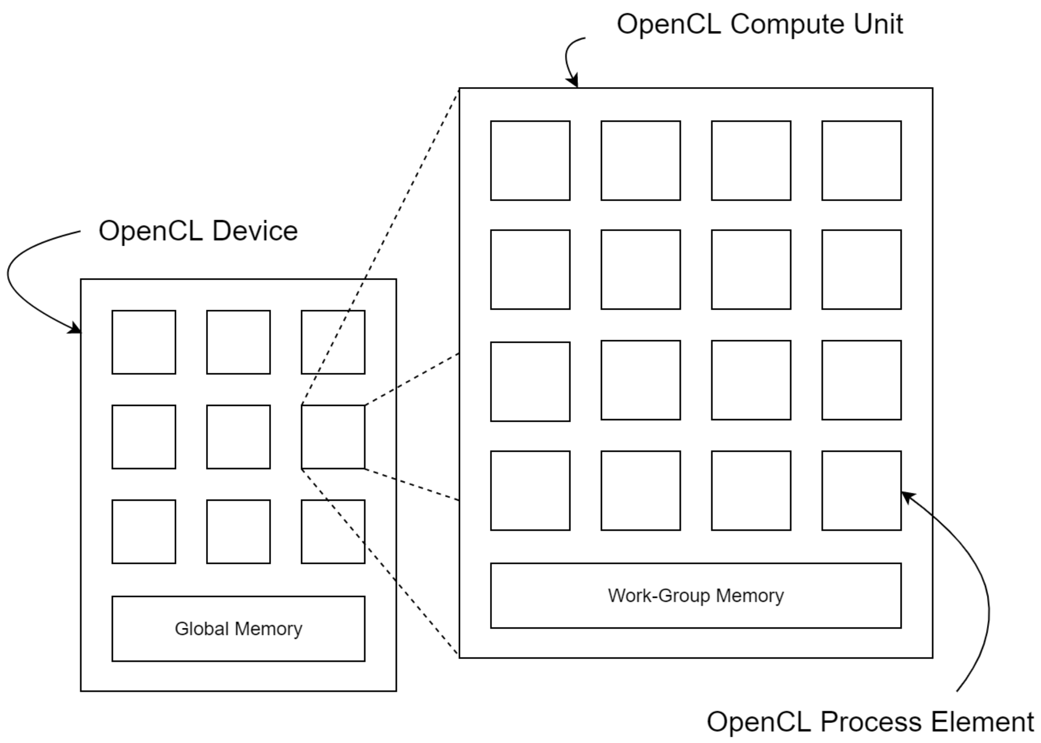


Figure 2.2: OpenCL Work-Groups

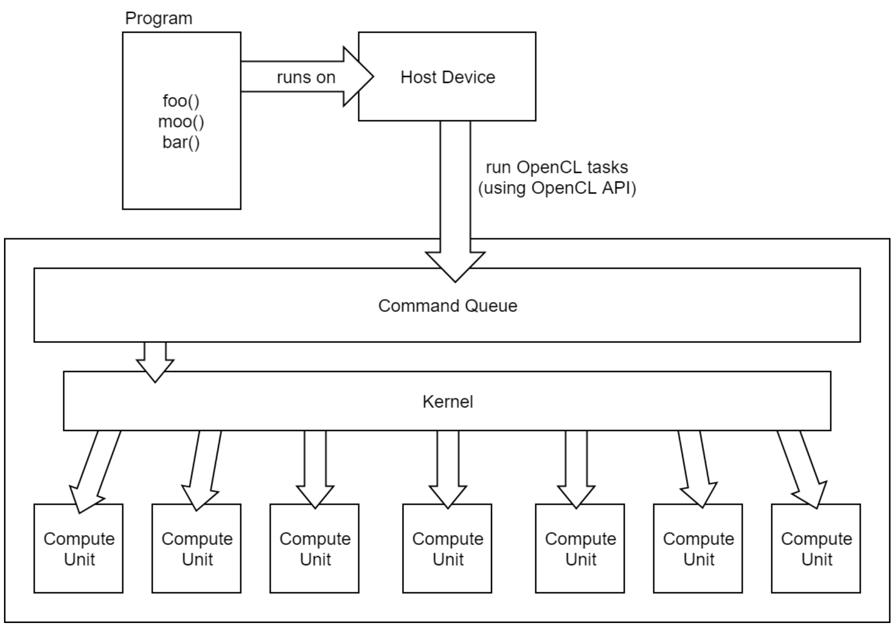


Figure 2.3: The program flow on OpenCL

2.2 OpenCL on GPUs and CPUs

The first OpenCL devices to be implemented were on GPUs the so called GPGPU. NVidia's previous work on their CUDA model made it easy for their GPUs to support the new open standard and this was an opportunity for other GPU vendors to provide GPGPU features on their GPU architectures. Nowadays ala major GPU vendors support the OpenCL standard in a way (maybe not the latest version).

A GPU can easily fit the OpenCL programing model, as it features a quite big internal memory (nowadays 2GB and 4GB GPUs are common) and many core units for graphics computations. Each GPU of a host machine can be a compute device and each of its core units, an OpenCL compute unit. After all the vector architecture of a GPU is one of the main reasons that the OpenCL and other parallel programming languages were created, to take advantage of their SIMD capabilities.

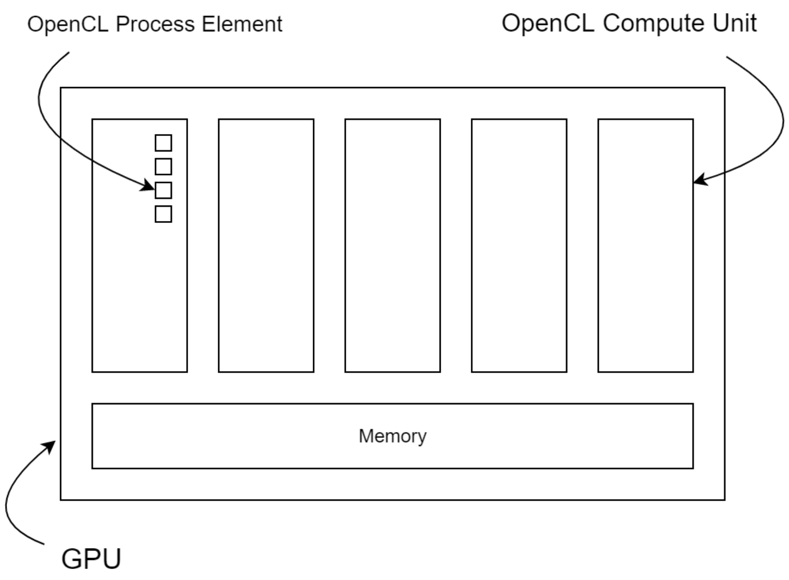


Figure 2.4: GPU as an OpenCL Device

Generally speaking OpenCL take advantage of the parallel execution of tasks to increase performance. Its in our interest to apply such parallelism not only on GPUs but also on our CPUs. Thus, it makes sense that we want to be able to run OpenCL code on our CPUs too if it will increase our application's performance.

CPUs features many CPU cores providing support for execution of parallel threads. This is why a CPU can also fit the OpenCL programing model. After all, a CPU has access to the main memory of the system and each of it's cores can be seen as a single compute unit, or even as multiple compute units if hyper threading is supported. Thus, CPU vendors like Intel and AMD supports OpenCL applications running on their CPU architectures.

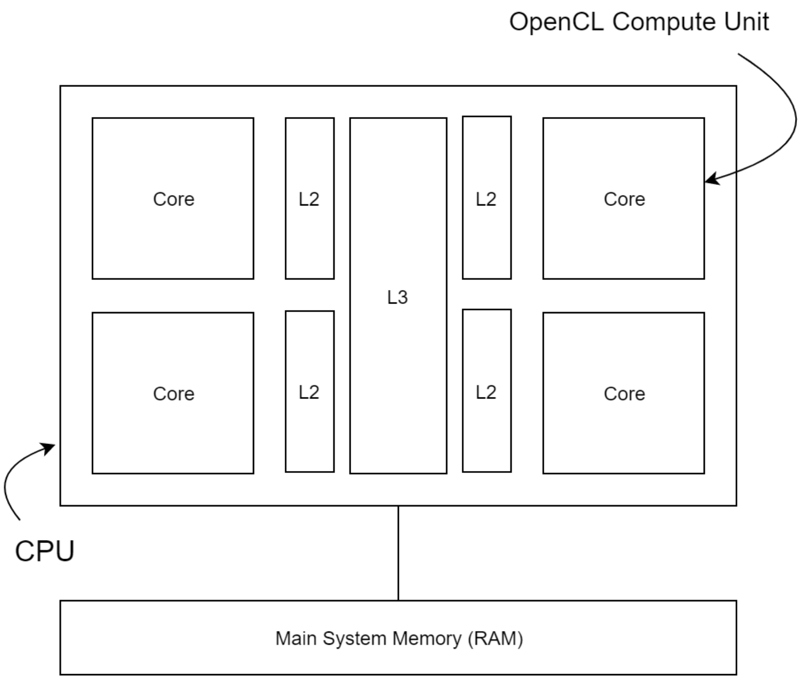


Figure 2.5: CPU as an OpenCL Device

As can be seen, OpenCL has created an interesting ecosystem, where one application can gain easy and simple access to parallel computing. The application can select on which of the host's OpenCL devices it wants to execute a kernel, or just pick one in random. The application does not need to have special knowledge of the OpenCL device as long as it follows the OpenCL standard.

For unsupported devices, interesting projects like pocl[1] (Portable Computing Language) where developed by 3rd party groups. Pocl implements the OpenCL standard by using Clang to parse the kernels and LLVM to execute them on your target device.

2.3 OpenCL on FPGAs

Developing an application on an FPGA is quite harder than usual coding. The development process takes more time than developing a similar application on a CPU. In addition, coding in HDLs demands experienced programmers and the debugging process it quite difficult. As a result, special C-to-HDL tools for HLL synthesis where created. Such tools can convert HLL code (usually C or C-like code) to an HDL code (like VHDL or Verilog). The performance of an application created with an HLS tool is worse (in most cases) than the performance of the same application created directly in HDL, but the development time is faster and the maintained easier.

Since HLS tools exist and FPGA vendors promote them, including an parallel programming language like OpenCL support on them was a good move (potential

[1]Website the “pocl“: `http://portablecl.org/`

performance improvement). So, OpenCL caught the attention of the two major FPGA manufacturers, Xilinx and Altera (Altera is currently owned by Intel). Therefore, each one released their own OpenCL-like development environment in order to support OpenCL applications on their FPGA boards.

Altera released the “Intel FPGA SDK for OpenCL“[2] and Xilinx released “The SDAccel development environment for OpenCL“[3]. Both environment support only specific boards (or provide custom board specifications) and not all their FPGA boards. These boards usually feature a PCIe interface to interact with the host machine and on-board DDR memory, just like a GPU does.

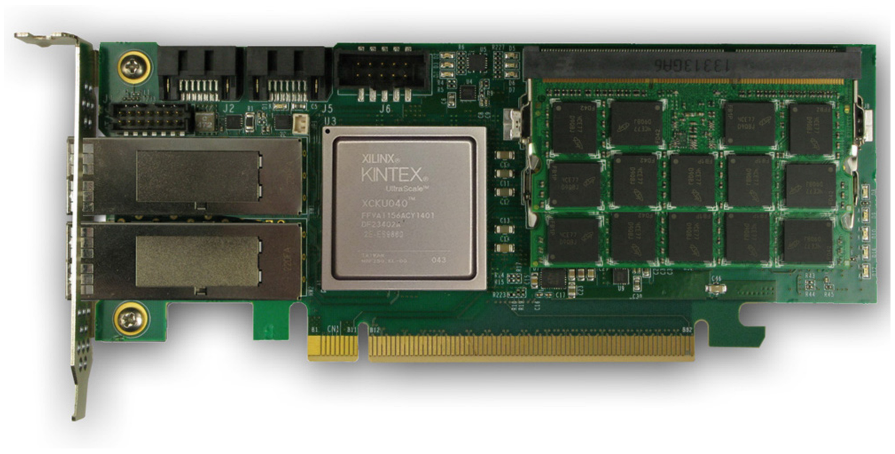


Figure 2.6: FPGA-Based Accelerator Board - ADM-PCIE-KU3 (Kintex Ultra-Scale from Alpha Data)
Image from: `https://www.xilinx.com/products/boards-and-kits/1-4lhig1.html`

[2]Website of “Intel FPGA SDK for OpenCL“:
`https://www.altera.com/products/design-software/embedded-software-developers/opencl/`

[3]Website of “The SDAccel development environment for OpenCL“:
`https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`

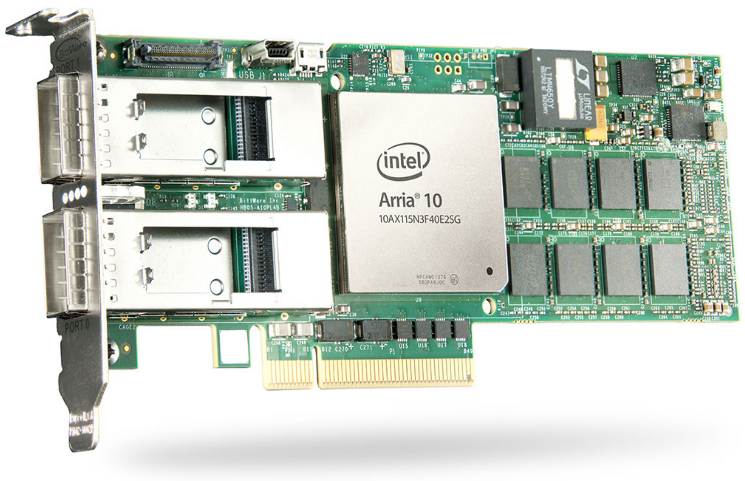


Figure 2.7: Altera FPGA-Based Accelerator Board - A10PL4 (Altera Arria 10 GX FPGA from BittWare)
Image from: `http://www.alteraboards.com/product/a10pl4/`

Of course the development of an OpenCL application for an FPGA is not the same as developing for a GPU but quite similar. For obvious reasons we don't expect a kernel compilation on the application's runtime, all kernels are in one way or an other pre-synthesized for the target FPAG board. Also additional special flags and functions may be needed outside of the OpenCL standard, for optimization or configuration.

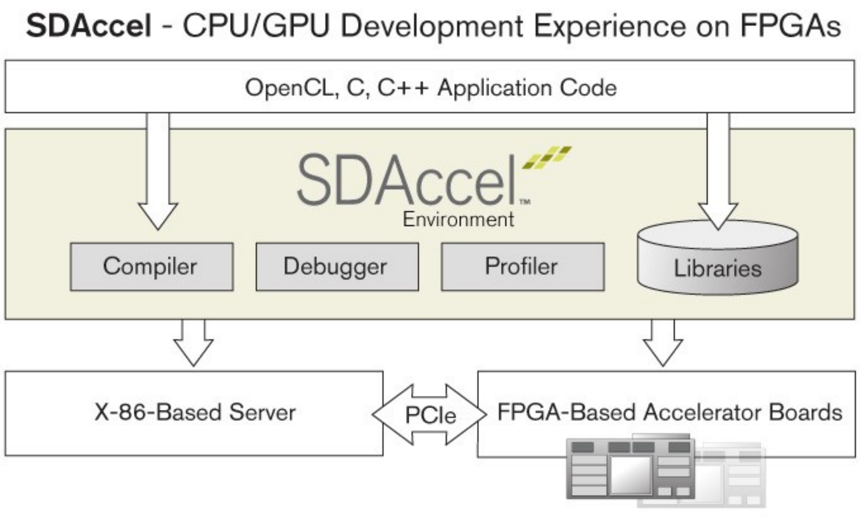

SDAccel - CPU/GPU Development Experience on FPGAs
OpenCL, C, C++ Application Code
SDAccel™ Environment
Compiler
Debugger
Profiler
Libraries
X-86-Based Server
PCIe
FPGA-Based Accelerator Boards

Figure 2.8: The SDAccess - Development Environment
Image from:
`https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`

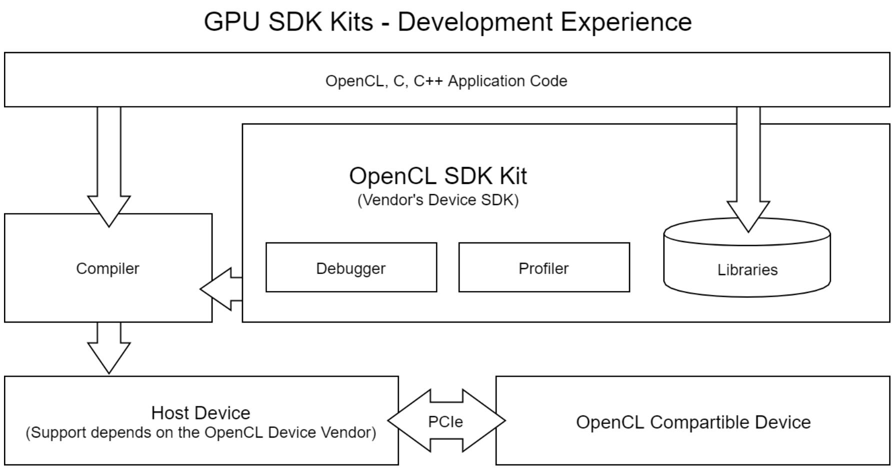

GPU SDK Kits - Development Experience
OpenCL, C, C++ Application Code
OpenCL SDK Kit
(Vendor's Device SDK)
Compiler
Debugger
Profiler
Libraries
Host Device
(Support depends on the OpenCL Device Vendor)
PCIe
OpenCL Compartible Device

Figure 2.9: GPU - Development Environment

2.4 OpenCL Kernels as IP Cores on FPGAs

Along side with its “Vivado Design Suite“[4], Xilinx released “Vivado High-Level Synthesis“[5] (Vivado HLS). With Vivado HLS we can program on C/C++/OpenCL and produce VHDL/Verilog IP cores for Vivado.

For OpenCL development, Vivado HLS creates from an OpenCL kernel's source an IP core of an OpenCL compute unit that implements this core. You can specify inside the kernel's source optimization and configurations using special attributes. For example you can define the size of the compute unit's work-group:
`__attribute__((reqd_work_group_size(size_x, size_y, size_z)))`
or suggest to execute a block of work-item's code in pipeline:
`__attribute__((xcl_pipeline_workitems)){/* Code here */}`

The IP core can be imported into Vivado for creating a design for your system. It can be connected with the target's system process unit and mapped on the system's memory. By doing this you will be able to communicate manually with the compute unit (IP core) by reading and writing on the system's memory in the specific address that it was mapped.

Some C wrappers (start, stop, write, read etc.) are automatically generated by the Vivado if you export the project to the SDK. This is enough for a simple bare metal applications with the need of one compute unit.

[4]Website of “Vivado Design Suite“:
https://www.xilinx.com/products/design-tools/vivado.html

[5]Website of “Vivado High-Level Synthesis“:
https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

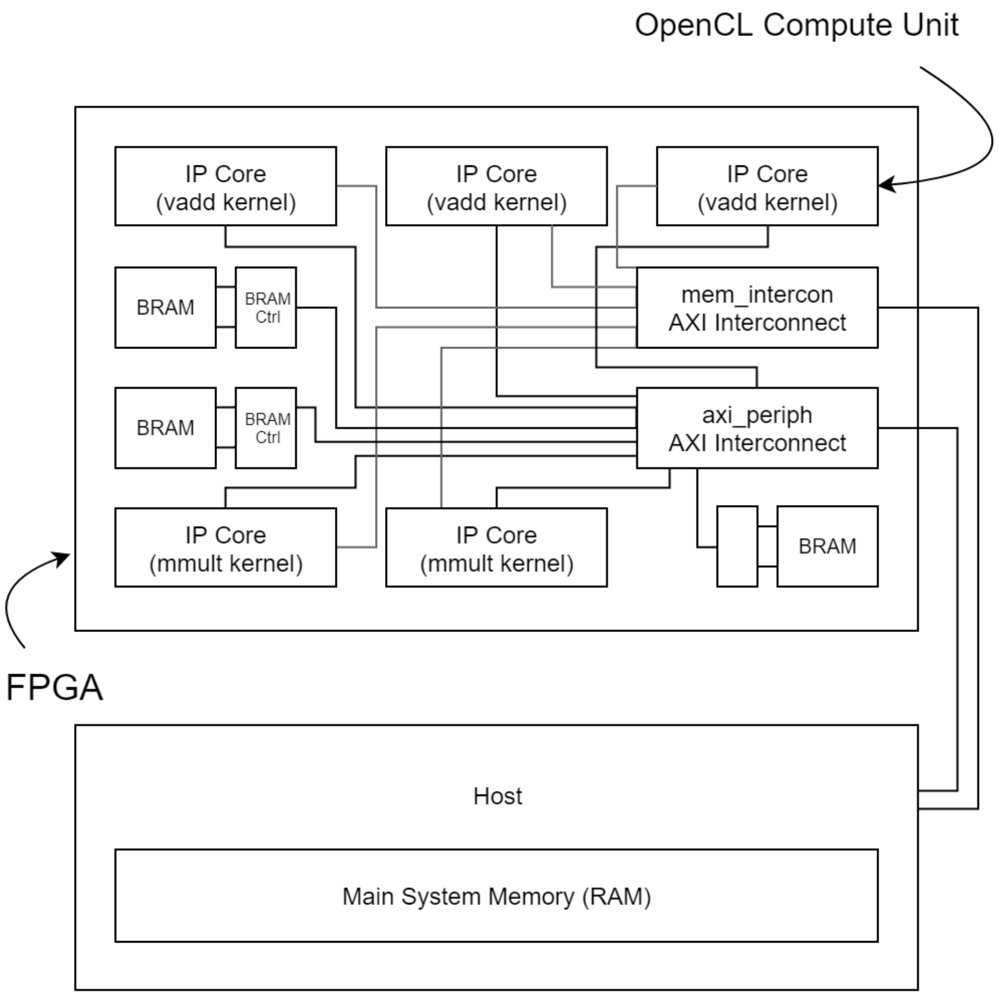


Figure 2.10: FPGA as an OpenCL Device (Using Vivado HLS OpenCL Kernels)

3 Motivation

3.1 The OpenCL open standard

OpenCL is a nice standard of parallel computing that the major Vendors supported and implement it relatively fast. Such well defined open standards are needed in order to have technological progress, since propitiatory monopoly standards are not widely used for obvious reasons. We wanted to support the standard by creating a tool to promote an easy and fast way of usage of OpenCL HLS kernels created by the Vivado HLS.

3.2 OpenCL Libraries

There are many open source OpenCL based libraries for machine learning, image processing, linear algebra, etc. that may run with better efficiency on an FPGA than a GPU. These libraries have many algorithms in kernels that call on demand. Porting each kernel to an FPGA bitstream can not be avoided but the interaction with the kernels inside it could be made easier.

Since the process of executing an OpenCL kernel on an FPGA is relatively the same for every HLS kernel and tends to follow the OpenCL execution process, if there was a library that could handle the bitfiles loading, could easily call the kernels inside the FPGA, could transfer the data hiding the virtual address complexity, spread the work-load across multiple units, and all these in a familiar way to the programmer (OpenCL like API), this would make the porting process easier and faster.

3.3 OS and FPGA Problems

Many FPGA boards have an embedded processor on them (usually an ARM) to run the applications. It's quite common to run bare metal applications on them without an OS since it may only control a design already programmed on the FPGA without the need of libraries and utilities that an OS provides. On the other hand, most computer applications run under an OS to take advantage of all the available libraries and features of an OS. So it make sense that if we want to port some of those applications, we will need to run them under a Linux OS on our FPGA board.

In this case you maybe think that it is easier than running on bare metal but mechanics like virtual address space increase the complexity of our code. There are solutions for such problem but the search for them and their implementation increase the development time. These solutions should be implemented ones in an open source library for everyone to use.

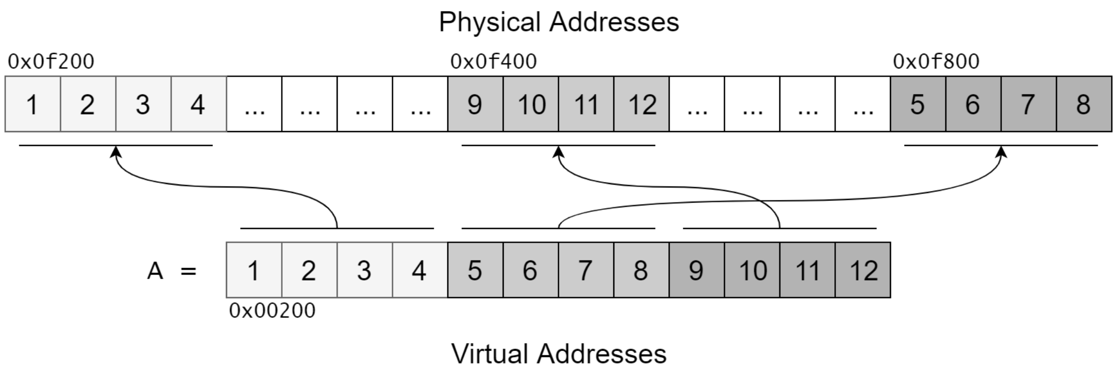


Figure 3.1: Array in Virtual addresses and Physical addresses

```
// Allocate array a
int* a = (int*) malloc(12 * sizeof(int)); // a = 0x00200
// Get physical address
size_t phys_a = virtual_to_physical(a); // phys_a = 0x0f200

// But a[x] does not point to the same address with phys_a[x]
// because the address 0x00200+x does not translate
// to the physical address 0x0f200+x
```

4 SimplePL

An OpenCL-like runtime system for HLS OpenCL Kernels. Simple, because we developed a simple OpenCL-like API. PL, because we control our beloved programming logic. In this section we will present you our simple OpenCL-like system, we will explain the operation of our library and we will introduce you to the API. Furthermore, we will briefly compare an application written for OpenCL and the same application written for SimplePL.

4.1 Introduction

SimplePL implements an OpenCL-like API to manage multiple bitfiles and its components. Since kernels are inside bitfiles, instead of programming kernels on an OpenCL device, we program bitfiles on our FPGA. To manage these bitfiles, a special description file for each bitfile is needed to hold information about the bit file.

Unlike OpenCL, SimplePL lets you manage your device memories. You can define many device memories, on FPGA as BRAM or on the system's main memory.

Buffer parameters can be created from any memory the developer selects, BRAM for small buffers (faster) or on system's memory for big buffers (but slower). FPGA memories (BRAMs) should be pre-synthesized inside the bitfile, therefore their life span ends after we program a new bitfile on our PL.

4.1.1 Programing the FPGA

We want our system to be able to download on demand different bitfiles in order to provide a way to load different HLS OpenCL Kernels. Xilinx provides a linux driver device (depending on your board) with which we can program the programmable logic. On ZedBoard we can write the bitfile on the "/dev/xdevcfg" device.
Terminal command to program the PL on Zedboard in a Linux environment.

```
cat mybitstream.bit > /dev/xdevcfg
```

4.1.2 Physical and Virtual addresses

Our FPGA can only understand physical addresses, but our applications run on virtual addresses. Thus, SimplePL maps all physical addresses (for example the base address of a compute unit) to virtual addresses, using mmap and /dev/mem.

Also, to be able to pass the physical address of a buffer to a compute unit, it stores the system's memories physical addresses.

4.1.3 Contiguous physical memory

Our device memories need to be contiguous physical addresses. We can not allocate a contiguous physical memory from user space, so we need a kernel module that does that for our applications. We could create such a kernel module but as we found a nice implementation of the same functionality and many more, we chose not to reinvent the wheel, and we used the “udmabuf“[1] (User space mappable DMA Buffer) kernel module. Udmabuf lets us define 4 contiguous physical memories on the main memory of the system.

To be able to access the system memory, you need to load the `udmabuf` kernel module on your system before you execute your application. You can specify the number of the buffers (up to 4) and their size in the `insmod` parameters. To create 4 buffers, each one with 2MB contiguous physical memory size, you can run the command:

```
insmod udmabuf.ko udmabuf0=0x200000 udmabuf1=0x200000 udmabuf2=0x200000
udmabuf3=0x200000
```

4.2 Bitfile Description File

To load bitfiles on SimplePL you have to provide their description files, which we will call from now on “Bitfile Description File“. The Bitfile description file is a JSON file with information for each kernel and for each memory inside the corresponding bitfile.

To parse the Bitfile Description File we use the `tiny-json` C library. The `tiny-json` library is a fast and simple C JSON parser. You can find it on Github in `https://github.com/rafagafe/tiny-json`.

4.2.1 Structure

A bitfile description file needs to have a special structure, if not, the bitfile will not be loaded or it may have invalid components (invalid kernel or memory).

4.2.1.1 Name attribute

The first thing stored is a name for the bitfile (this has nothing to do with filenames, it just defines a name for your bitfile). The name attribute should be a String. This attribute is mandatory and a bitfile description file without one, is invalid.

[1]Website of “udmabuf“: https://github.com/ikwzm/udmabuf

Attribute	Type	Mandatory
name	String	yes

Table 4.1: Attribute “name“

```
{
  ...
  "name" : "Just a name for my bitfile",
  ...
}
```

Figure 4.1: Example for attribute “name“

4.2.1.2 Kernels attribute

To define our bitfile's HLS OpenCL kernels, we provide an array of kernels information. Thus, kernels attribute should be an array. This attribute is not mandatory and a bitfile description file without it is valid. You may load bitfiles with no HLS kernels.

Attribute	Type	Mandatory
kernels	Array	no

Table 4.2: Attribute “kernels“

Each item of the kernels attribute describes a kernel.

- The name attribute is used to call the kernel. It accepts a String value and is mandatory.
- The address array defines the physical base address of each compute unit (IP core) of this HLS kernel. As this attribute defines the number of compute units it must be defined.
  - Must have at least one address (at least one compute unit).
  - Addresses must be defined as Hex String (ex “0x0000F“).
- The workgroup array defines the workgroup size of the compute units. It must be defined.
  - It supports 3 dimensions [x, y, z] like OpenCL.
  - At least one dimension must be defined and the rest will be set to 1.
  - Dimension length is an Integer.
  - None dimension length should be zero.
- Size attribute defines the size of memory that each compute unit maps on memory. It must be defined.

  - Value must be an Integer.
  - If value is zero, size will be set to memory page size.
  - (We usually set it to zero).

- Control object defines the control signals offsets of a compute unit.
  - Must be defined.
  - Object's attributes
    - "ctrl" : offset of Control Signals (if not defined is set to "0x0")
    - "gie" : offset of Global Interrupt Enable Register (if not defined, Global Interrupt functions can not be used)
    - "ier" : offset of IP Interrupt Enable Register (if not defined, IP Interrupt functions can not be used)
    - "isr" : offset of IP Interrupt Status Register (not currently in use)
  - Values must be Hex Strings (example "0x48")

- Group object defines the work-group signals offsets of a compute unit.
  - May not be defined, but if is defined, all attributes must be defined.
  - Object's attributes
    - "id_x" : offset of group_id_x signal (number of work-group in 1st dimension)
    - "id_y" : offset of group_id_y signal (number of work-group in 2nd dimension)
    - "id_z" : offset of group_id_z signal (number of work-group in 3rd dimension)
    - "offset_x" : offset of global_offset_x (memory buffers offset in 1st dimension)
    - "offset_y" : offset of global_offset_y (memory buffers offset in 2nd dimension)
    - "offset_z" : offset of global_offset_z (memory buffers offset in 3rd dimension)
  - Values must be Hex Strings (example "0x48")

- Kernel's arguments array.
  - May not be defined.
  - Each item represents a kernel argument and has the following attributes
    - "name" : the name of the argument. Must be a String. Used to set the argument by name. Should be defined.
    - "offset" : offset of the argument register(s). Must be a Hex String. Should be defined.

- ∗ “size“ : size of the argument register. Must be an Integer. It is usually 4 (4 bytes). May not be defined.

All memory sizes and offsets are in bytes.

Attribute	Type	Mandatory
name	String	yes
address	Array	yes
workgroup	Array	yes
size	Integer	yes
control	Object	no
group	Object	no
arguments	Array	no

Table 4.3: Attributes of the items of the attribute “kernels“

Attribute	Type	Mandatory
name	String	yes
address	Array	yes
workgroup	Array	yes
size	Integer	yes
control	Object	no
group	Object	no
arguments	Array	no

Table 4.4: Attributes of the attribute “kernels“ attribute “control“

```
{
  ...
  "kernels" : [
    ...
    {
      "name" : "my_kernel",
      "address" : ["0x43C00000", ... ],
      "workgroup" : [32, 32, 1],
      "size" : 0,
      "control" : {
        "ctrl" : "0x00",
        "gie" : "0x04", "ier" : "0x08", "isr" : "0x0c"
      },
      "group" : {
        "id_x" : "0x10", "id_y" : "0x18", "id_z" : "0x20",
        "offset_x" : "0x28", "offset_y" : "0x30", "offset_z" : "0x38"
      },
      "arguments" : [
        ...
        {"name" : "my_parameter", "offset" : "0x40", "size" : 4},
        ...
      ]
    }
    ...
  ],
  ...
}
```

Figure 4.2: Example for attribute "kernels"

4.2.1.3 Memories attribute

Like before, to define our bitfile's memories (BRAMs), we provide an array of memories information. Thus, memories attribute should be an array. This attribute is not mandatory and a bitfile description file without it is valid. You may load bitfiles with no memories.

Attribute	Type	Mandatory
memories	Array	no

Table 4.5: Attribute "memories"

Each item of the memories attribute describes a BRAM inside our bitfile.

- The name attribute is used to get the memory by name. It accepts a String value and is mandatory.
- The address attribute defines the physical base address of the memory. It must be defined as Hex String and is mandatory.
- The size attribute defines the size of the memory in bytes. It must be defined an Integer and is mandatory.

```
{
  ...
  "memories" : [
    ...
    {
      "name" : "my_fast_bram",
      "address" : "0x40000000",
      "size" : 8192
    },
    ...
  ],
  ...
}
```

Figure 4.3: Example for attribute “memories“

4.2.2 Collecting the information

The information for the Bitfile Description File needs to be retrieved from Vivado and Vivado HLS. The base address offsets of each IP core and each BRAM can be found on the Vivado design’s “Address Editor“. IP core control and arguments offsets can be found inside the the xml file:
“`<hls_project>/solution<number>/impl/ip/<kernel-name>_info.xml`“
A full Bitfile Description File example can be found on the appendix section.

4.3 Data Types

In this section we will describe SimplePL’s data types.

4.3.1 Xilinx-like UInt Data Types

Unsigned integer data types based on their size in bytes, based on Xilinx’s similar data types. These data types are useful when dealing with data inside addresses in order to align. These types keep the same bytes even on different systems.

Data Type	Description	Size in bytes
`pl_u8`	8-bits Unsigned Integer	1 byte
`pl_u16`	16-bits Unsigned Integer	2 bytes
`pl_u32`	32-bits Unsigned Integer	4 bytes

Table 4.6: Data Types: Xilinx-like uint

4.3.2 OpenCL-like Basic Data Types

SimplePL provides the same scalar data types that OpenCL provide.

Scalar Data Type	Type in OpenCL Language	Size
`pl_char`	char	8-bit
`pl_uchar`	unsigned char	8-bit
`pl_short`	short	16-bit
`pl_ushort`	unsigned short	16-bit
`pl_int`	int	32-bit
`pl_uint`	unsigned int	32-bit
`pl_long`	long	64-bit
`pl_ulong`	unsigned long	64-bit
`pl_float`	float	32-bit
`pl_half`	half	16-bit
`pl_double`	double (if supported)	64-bit

Table 4.7: Data Types: OpenCL-like Basic data types

In addition a boolean enumerate type is defined. It can be set to “PL_TRUE“ or “PL_FALSE“.

Data Type	Values
`pl_bool`	`PL_TRUE` or `PL_FALSE`

Table 4.8: Data Types: Boolean enumerator

4.3.3 OpenCL-like Components Data Types

In SimplePL the main components are bitfile, kernel, device memory and buffer. These components have their own data types. Some of them have a similar definition to OpenCL but others does not exits on OpenCL (like bitfile).

Components Data Type	Description
`pl_bitfile`	Reference to a loaded bitfile
`pl_kernel`	Reference to a bitfile's kernel
`pl_memory`	Reference to a bitfile's or system's memory
`pl_buffer`	Reference to a buffer on a memory

Table 4.9: Data Types: Components data types

4.4 Application Programming Interface

In this section we will describe in detailed SimplePL's API.

4.4.1 Bitfile functions

plInitDevice

Used to initialize SimplePL Library

```
pl_int plInitDevice ()
```

Notes

The initialization process of SimplePL includes the following actions:

Check permissions to the `/dev/xdevcfg` device
Open `/dev/mem` device
Initialize `/dev/udmabufX` buffers if available as system memory

Errors

plInitDevice returns `PL_SUCCESS` if it was executed successfully. Otherwise, one of the errors bellow will be returned:

`PL_XDEVCFG_NOT_FOUND` if `/dev/xdevcfg` was not found
`PL_XDEVCFG_NO_WRITE_PERMISSION` if you have not write permissions on the `/dev/xdevcfg`
`PL_OPEN_DEVMEM_FAILED` if the library failed the open `/dev/mem`

plGetBitfileInfo

Used to load a bitfile description file information on a `pl_bitfile` object

```
pl_int plGetBitfileInfo (pl_bitfile* bitfile,
                         const char* path,
                         const char* name)
```

Parameters

`bitfile`
A pointer to a `pl_bitfile` object. If the function returns `PL_SUCCESS` this object will contain the information of your bitfile.

`path`
Path to the folder that includes the bitfile and the bitfile description file

`name`
Name of the bitfile and the bitfile description file without the file extension.

Notes

The bitfile should be in “`{path}{name}.bit`“ and the bitfile description file in “`{path}{name}.json`“, where “`{path}`“ is the value of “`path`“ parameter and “`{name}`“ is the value of “`name`“ parameter. These paths can be relative paths.

Errors

plGetBitfileInfo returns `PL_SUCCESS` if it was executed successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_bitfile` was `NULL`
`PL_FILE_NOT_FOUND` if "`{path}{name}.bit`" or "`{path}{name}.json`" file was not found.
`PL_FILE_PERMISSIONS_ERROR` if "`{path}{name}.bit`" or "`{path}{name}.json`" file could not be accessed.
`PL_NOT_ENOUGH_MEMORY` if there was not enough memory to load bitfile description file's contents
`PL_FILE_INVALID_FORMAT` if bitfile description file failed to be parsed

plBitfileInitialize

Used to intialize a bitfile programed on the PL.

```
pl_int plBitfileInitialize (pl_bitfile bitfile,
                            pl_mask mask)
```

Parameters

`bitfile`

The `pl_bitfile` object to initialize.

`mask`

A mask to define which components inside the bitfiles to be initialized. The available masks are:

Mask	Description
`PL_BITFILE_INIT_ALL`	Initialize all bitfile's components
`PL_BITFILE_INIT_KERNELS`	Initialize all bitfile's kernels
`PL_BITFILE_INIT_MEMORIES`	Initialize all bitfile's memories

Notes

This function initialize all needed data of a programmed bitfile and its components. We usually use as a `mask` the `PL_BITFILE_INIT_ALL`. This function focus on initializing the bitfile, thus, any error produced by the initialization of any of the component, is ignored.

Errors

plBitfileInitialize returns `PL_SUCCESS` if the bitfile was already initialized or if it was initialized successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_bitfile` was `NULL`
`PL_NOT_PROGRAMED` if bitfile seems not to be programed on the PL.

plBitfileRelease

Used to intialize a bitfile programed on the PL.

```
pl_int plBitfileRelease (pl_bitfile bitfile)
```

Parameters

`bitfile`

The `pl_bitfile` object to be released.

Notes

This function disables the bitfile and its components (kernel and memories) and frees all the allocated objects.

Errors

plBitfileRelease returns `PL_SUCCESS` if the bitfile was released successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_bitfile` was `NULL`

plProgram

Used to program the PL with a bitfile

```
pl_int plProgram (pl_bitfile bitfile)
```

Parameters

`bitfile`

The `pl_bitfile` object to be programed on the PL.

Notes

This function also increase the ProgramedID and saves a copy of it on the `pl_bitfile` object it just programed on the PL.

Errors

plProgram returns `PL_SUCCESS` if the PL was programed successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_bitfile` was `NULL`

`PL_XDEVCFG_WRITE_FAILED` if the library failed to write the bitfile to the xdevcfg driver.

`PL_XDEVCFG_PROGRAM_FAILED` if the PL programming failed.

plIsProgramed

Used to check if this was the last bitfile to be programmed on the PL.

`pl_bool plIsProgramed (pl_bitfile bitfile)`

Parameters

`bitfile`

The `pl_bitfile` object to check if is programmed.

Notes

This function does not check the PL is there is an other bitfile programmed, it compares the `bitfile`'s programedID with the last programedID. ProgramedID is an increment that increases every time the application programs the PL. The programmed `pl_bitfile` keeps a copy of the programedID at the time.

Returns

plIsProgramed returns `PL_TRUE` if this was the last bitfile to be programmed on the PL. Otherwise, returns `PL_FALSE`.

4.4.2 Memory functions

plMemoryInitialize

Intialize a `pl_memory` object.

`pl_int plMemoryInitialize (pl_memory memory)`

Parameters

`memory`

The `pl_memory` object to initialize.

Notes

It maps memory's address into virtual address and prepare it for use. This function is called for each memory by the `plBitfileInitialize` function if memories initialization is covered by the mask.

Errors

plMemoryInitialize returns `PL_SUCCESS` if the memory was already initialized or if it was initialized successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_memory` was `NULL`

`PL_NOT_PROGRAMED` if parent bitfile seems not to be programed on the PL.

`PL_NOT_VALID` if memory's description was not valid and thus could not be loaded

`PL_MEMORY_MAP_FAILED` if the physical address of the memory failed to be mapped on a virtual

plMemoryRelease

Release a `pl_memory` object.

```
pl_int plMemoryRelease (pl_memory memory)
```

Parameters

`memory`

The `pl_memory` object to be released.

Notes

This function disables the memory and frees all the allocated objects. This function is called for each memory by the `plBitfileRelease` function.

Errors

plMemoryRelease returns `PL_SUCCESS` if the memory was released successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_memory` was `NULL`

`PL_MEMORY_UNMAP_FAILED` if the virtual address of the memory failed to be unmapped

plGetSysMemoryByIndex

Get a system's memory (a memory on the main system memory).

```
pl_memory plGetSysMemoryByIndex (size_t index)
```

Parameters

`index`

The number of memory to get. You will get the `index`$^{\text{th}}$ system's memory.

Notes

System memories are provided by the `udmabuf` kernel module. You can create up to 4 memories buffers when you insert the module.

Errors

plGetSysMemoryByIndex returns the `pl\_memory` object of the system memory you asked. Otherwise, it will return `NULL` if the memory was not found.

plGetMemoryByIndex

Get a bitfile's memory (a memory on the PL) based on its index on the kernel list.

```
pl_memory plGetMemoryByIndex (pl_bitfile bitfile,
                              size_t index)
```

Parameters

`bitfile`
: The bitfile that includes the memory.

`index`
: The number of memory to get. You will get the `index`th memory of the bitfile.

Notes

-

Errors

plGetMemoryByIndex returns the `pl_memory` object of the bitfile's memory you asked. Otherwise, it will return `NULL` if the memory was not found or if the `bitfile` parameter was `NULL`.

plGetMemoryByName

Get a bitfile's memory (a memory on the PL) based on its name.

```
pl_memory plGetMemoryByName (pl_bitfile bitfile,
                             char* name)
```

Parameters

`bitfile`
: The bitfile that includes the memory.

`name`
: The name of the memory to get.

Notes

The name of each memory is declared on the bitfile description file.

Errors

plGetMemoryByName returns the `pl_memory` object of the bitfile's memory you asked. Otherwise, it will return `NULL` if the memory was not found or if the `bitfile` parameter was `NULL`.

plClearMemory

Clear the memory but reseting the stack pointer of a memory.

```
pl_int plClearMemory (pl_memory memory)
```

Parameters

`memory`
: The memory to clear.

Notes

This function only resets the stack pointer of the memory, so that new buffers will be created from the start of the memory.

Errors

plClearMemory returns `PL_SUCCESS` if the memory was cleared successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_memory` was `NULL`

4.4.3 Buffer functions

plCreateBuffer

Create a buffer on a memory.

```
pl_buffer plCreateBuffer (pl_memory memory,
                          size_t size,
                          pl_int *errcode_ret)
```

Parameters

`memory`
: The memory inside which the buffer will be created.

`size`
: The size of the buffer to be created in bytes.

`errcode_ret`
: This will be `PL_SUCCESS` if where was no error or will have an error code if an error occurred.

Notes

The size of the buffer will be changed to the next multiple of 4, so that it matches exactly on the addresses.
The buffer will be created on the specified memory on the address:
`{memory base address}+{offset}`
where offset is the stack pointer of the memory. Then the stack pointer will be increased with the size value of the buffer.
The library holds a list of all the buffers created, to recognize when a buffer is given as a kernel parameter. Thus, if no error occurred, this buffer will be added on the list.

Errors

plCreateBuffer returns a `pl_buffer` object if the buffer was created successfully. Otherwise, returns `NULL` and the `errcode_ret` parameter will have one of the errors bellow:

`PL_NOT_READY` if memory is not initialized
`PL_NOT_ENOUGH_MEMORY` if the requested size can not be allocated

plReleaseBuffer

Release a `pl_memory` object.

```
pl_int plReleaseBuffer (pl_buffer buffer)
```

Parameters

`buffer`

The `pl_buffer` object to be released.

Notes

This function release the buffer's memory. It also removes the buffer from library's the buffers list.

Errors

plReleaseBuffer returns `PL_SUCCESS` if the memory was released successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_buffer` was `NULL`

plWriteBuffer

Write data to the device buffer

```
pl_int plWriteBuffer (pl_buffer buffer,
                      size_t offset,
                      size_t size,
                      const void *ptr)
```

Parameters

`buffer`
: The `pl_buffer` object to write to.

`offset`
: The offset in the buffer to write to, in bytes.

`size`
: The size of the data to write, in bytes.

`ptr`
: The pointer to the buffer where data is to be written from.

Notes

This functions memory copies data from the `ptr` to the `buffer`.

Errors

plWriteBuffer returns `PL_SUCCESS` if the memory copy was successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_buffer` was `NULL`

plReadBuffer

Read data from the device buffer

```
pl_int plReadBuffer (pl_buffer buffer,
                     size_t offset,
                     size_t size,
                     const void *ptr)
```

Parameters

`buffer`
: The `pl_buffer` object to read from.

`offset`
: The offset in the buffer to read from, in bytes.

`size`
: The size of the data to read, in bytes.

`ptr`
: The pointer to the buffer where data is to be read to.

Notes

This functions memory copies data from the `buffer` to the `ptr`.

Errors

plReadBuffer returns `PL_SUCCESS` if the memory copy was successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_buffer` was `NULL`

4.4.4 Kernel functions

plKernelInitialize

Intialize a `pl_kernel` object.

```
pl_int plKernelInitialize (pl_kernel kernel)
```

Parameters

`kernel`

The `pl_kernel` object to initialize.

Notes

It maps kernel's address into virtual address and prepare it for use. This function is called for each kernel by the `plBitfileInitialize` function if kernels initialization is covered by the mask.

Errors

plKernelInitialize returns `PL_SUCCESS` if the kernel was already initialized or if it was initialized successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_memory` was `NULL`

`PL_NOT_PROGRAMED` if parent bitfile seems not to be programed on the PL.

`PL_NOT_VALID` if kernel's description was not valid and thus could not be loaded

`PL_MEMORY_MAP_FAILED` if the physical address of the kernel failed to be mapped on a virtual

plKernelRelease

Release a `pl_kernel` object.

```
pl_int plKernelRelease (pl_kernel kernel)
```

Parameters

`kernel`
: The `pl_kernel` object to be released.

Notes

This function disables the kernel and frees all the allocated objects. This function is called for each kernel by the `plBitfileRelease` function.

Errors

plKernelRelease returns `PL_SUCCESS` if the kernel was released successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`

`PL_MEMORY_UNMAP_FAILED` if the virtual address of the kernel failed to be unmapped

plGetKernelByIndex

Get a bitfile's kernel based on its index on the kernel list.

```
pl_kernel plGetKernelByIndex (pl_bitfile bitfile,
                              size_t index)
```

Parameters

`bitfile`
: The bitfile that includes the kernel.

`index`
: The number of kernel to get. You will get the `index`th kernel of the bitfile.

Notes

-

Errors

plGetKernelByIndex returns the `pl_kernel` object of the bitfile's kernel you asked. Otherwise, it will return `NULL` if the kernel was not found or if the `bitfile` parameter was `NULL`.

plGetKernelByName

Get a bitfile's kernel based on its name.

```
pl_kernel plGetKernelByName (pl_bitfile bitfile,
                             char* name)
```

Parameters

`bitfile`
  The bitfile that includes the kernel.

`name`
  The name of the kernel to get.

Notes

The name of each kernel is declared on the bitfile description file.

Errors

plGetKernelByName returns the `pl_kernel` object of the bitfile's kernel you asked. Otherwise, it will return `NULL` if the kernel was not found or if the `bitfile` parameter was `NULL`.

plSetKernelGlobalInterrupt

Set the global interrupt of a kernel.

```
pl_int plSetKernelGlobalInterrupt (pl_kernel kernel,
                                    pl_bool type)
```

Parameters

`kernel`
  The target `pl_kernel` object.

`type`
  The type of the action. Can take the value of:

Action	Description
`PL_ENABLE`	Enable the global interrupt
`PL_DISABLE`	Disable the global interrupt

Notes

This function enables or disables the global interrupt of each compute unit (IP core) of the kernel.

Errors

plSetKernelGlobalInterrupt returns `PL_SUCCESS` if the global interrupt was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`

`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL

`PL_NOT_READY` if the kernel was not initialized

`PL_NOT_VALID` if global interrupt offset was not set on bitfile description file

`PL_FAILED` if the type given was not valid

plSetKernelComputeUnitGlobalInterrupt

Set the global interrupt of a compute unit (IP core).

```
pl_int plSetKernelComputeUnitGlobalInterrupt (pl_kernel kernel,
                                              size_t index,
                                              pl_bool type)
```

Parameters

`kernel`

The target `pl_kernel` object.

`index`

Target the `index`th compute unit.

`type`

The type of the action. Can take the value of:

Action	Description
`PL_ENABLE`	Enable the global interrupt
`PL_DISABLE`	Disable the global interrupt

Notes

This function enables or disables the global interrupt of the `index`th compute unit (IP core) of the kernel.

Errors

plSetKernelComputeUnitGlobalInterrupt returns `PL_SUCCESS` if the global interrupt was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_VALID` if global interrupt offset was not set on bitfile description file
`PL_NOT_FOUND` if compute unit was not found
`PL_FAILED` if the type given was not valid

plSetKernelInterrupt

Set the IP interrupt of a kernel.

```
pl_int plSetKernelInterrupt (pl_kernel kernel,
                             pl_bool type,
                             pl_mask mask)
```

Parameters

`kernel`

The target `pl_kernel` object.

`type`

The type of the action. Can take the value of:

Action	Description
`PL_ENABLE`	Enable the IP interrupt of the mask
`PL_DISABLE`	Disable the IP interrupt of the mask

`mask`

The mask value to apply.

Notes

This function enables or disables the interrupt of each compute unit (IP core) of the kernel. The mask specify the bits to be enabled or disabled.

Errors

plSetKernelInterrupt returns `PL_SUCCESS` if the interrupt was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`

`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL

`PL_NOT_READY` if the kernel was not initialized

`PL_NOT_VALID` if interrupt offset was not set on bitfile description file

`PL_FAILED` if the type given was not valid

plSetKernelComputeUnitInterrupt

Set the interrupt of a compute unit (IP core).

```
pl_int plSetKernelComputeUnitInterrupt (pl_kernel kernel,
                                        size_t index,
                                        pl_bool type,
                                        pl_mask mask)
```

Parameters

`kernel`

The target `pl_kernel` object.

`index`

Target the `index`th compute unit.

`type`

The type of the action. Can take the value of:

Action	Description
`PL_ENABLE`	Enable the interrupt
`PL_DISABLE`	Disable the interrupt

`mask`

The mask value to apply.

Notes

This function enables or disables the interrupt of the `index`th compute unit (IP core) of the kernel. The mask specify the bits to be enabled or disabled.

Errors

plSetKernelComputeUnitInterrupt returns `PL_SUCCESS` if the interrupt was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_VALID` if interrupt offset was not set on bitfile description file
`PL_NOT_FOUND` if compute unit was not found
`PL_FAILED` if the type given was not valid

plSetKernelArg

Set a kernel's argument.

```
pl_int plSetKernelArg (pl_kernel kernel,
                       pl_uint arg_index,
                       size_t arg_size,
                       const void *arg_value)
```

Parameters

`kernel`

The target `pl_kernel` object.

`arg_index`

Target the `arg_index`th kernel argument.

`arg_size`

Size of the value in bytes. Usually just a sizeof the value's data type (`sizeof({data_type})`).

`arg_value`
: The value to set on the argument.

Notes

This function sets the value argument on each compute unit. If the argument is a buffer, the physical address of the buffer should be set. This is why this function check is the argument is a buffer (searches the value on the buffer list) and if found, it uses the buffer's physical address as a value. If the size of the argument is bigger than an address size, the required addresses will be written.

Errors

plSetKernelArg returns `PL_SUCCESS` if the argument was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_FOUND` if the argument was not found
`PL_NOT_VALID` if argument was invalid on bitfile description file

plSetKernelComputeUnitArg

Set a kernel's argument, on a single compute unit.

```
pl_int plSetKernelComputeUnitArg (pl_kernel kernel,
                                  size_t index,
                                  pl_uint arg_index,
                                  size_t arg_size,
                                  const void *arg_value)
```

Parameters

`kernel`
: The target `pl_kernel` object.

`index`
: Target the `index`th compute unit.

`arg_index`
: Target the `arg_index`th kernel argument.

`arg_size`
: Size of the value in bytes. Usually just a sizeof the value's data type (`sizeof({data_type})`).

`arg_value`
The value to set on the argument.

Notes

This function sets the value argument on the selected compute unit. If the argument is a buffer, the physical address of the buffer should be set. This is why this function check is the argument is a buffer (searches the value on the buffer list) and if found, it uses the buffer's physical address as a value. If the size of the argument is bigger than an address size, the required addresses will be written.

Errors

plSetKernelComputeUnitArg returns `PL_SUCCESS` if the argument was set successfully. Otherwise, one of the errors bellow will be returned:
`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_FOUND` if the argument or the compute unit was not found
`PL_NOT_VALID` if argument was invalid on bitfile description file

plSetKernelArgByName

Set a kernel's argument.

```
pl_int plSetKernelArgByName (pl_kernel kernel,
                              const char *name,
                              size_t arg_size,
                              const void *arg_value)
```

Parameters

`kernel`
The target `pl_kernel` object.
`name`
Target the kernel argument with this name.
`arg_size`
Size of the value in bytes. Usually just a sizeof the value's data type (`sizeof({data_type})`).
`arg_value`
The value to set on the argument.

Notes

The name of each argument is declared on the bitfile description file. This function search the argument with the given name, and then calls `plSetKernelArg` with its index. Also, this function sets the value argument on each compute unit. If the argument is a buffer, the physical address of the buffer should be set. This is why this function check is the argument is a buffer (searches the value on the buffer list) and if found, it uses the buffer's physical address as a value. If the size of the argument is bigger than an address size, the required addresses will be written.

Errors

plSetKernelArgByName returns `PL_SUCCESS` if the argument was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_FOUND` if the argument was not found
`PL_NOT_VALID` if argument was invalid on bitfile description file

plSetKernelComputeUnitArgByName

Set a kernel's argument, on a single compute unit.

```
pl_int plSetKernelComputeUnitArgByName (pl_kernel kernel,
                                        size_t index,
                                        const char *name,
                                        size_t arg_size,
                                        const void *arg_value)
```

Parameters

`kernel`
The target `pl_kernel` object.

`index`
Target the `index`th compute unit.

`name`
Target the kernel argument with this name.

`arg_size`
Size of the value in bytes. Usually just a sizeof the value's data type (`sizeof({data_type})`).

`arg_value`
The value to set on the argument.

Notes

The name of each argument is declared on the bitfile description file. This function sets the value argument on the selected compute unit. If the argument is a buffer, the physical address of the buffer should be set. This is why this function check is the argument is a buffer (searches the value on the buffer list) and if found, it uses the buffer's physical address as a value. If the size of the argument is bigger than an address size, the required addresses will be written.

Errors

plSetKernelComputeUnitArgByName returns `PL_SUCCESS` if the argument was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_FOUND` if the argument or the compute unit was not found
`PL_NOT_VALID` if argument was invalid on bitfile description file

plSetKernelGroupIds

THIS FUNCTION IS NOT YET CLEAR, ON HOW TO BE USED
Set a kernel's group ids.

```
pl_int plSetKernelGroupIds (pl_kernel kernel,
                            pl_uint x,
                            pl_uint y,
                            pl_uint z)
```

Parameters

`kernel`
The target `pl_kernel` object.

`x`
The id for the 1st dimension.

`y`
The id for the 2nd dimension.

`z`
The id for the 3rd dimension.

Notes

This function schedules compute units. This function can be used to set the group ids on all the compute units. The group id of each compute unit will be `TODO` were `index` is the number of the compute unit. Usually we use: `x = 0, y = 0, z = 0` to assign different work-groups on each compute unit.

Errors

plSetKernelGroupIds returns `PL_SUCCESS` if the argument was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_FOUND` if the group information was not found

plSetKernelComputeUnitGroupIds

Set a kernel's group ids, on a single compute unit.

```
pl_int plSetKernelComputeUnitGroupIds (pl_kernel kernel,
                                       size_t index,
                                       pl_uint x,
                                       pl_uint y,
                                       pl_uint z)
```

Parameters

`kernel`
The target `pl_kernel` object.

`index`
Target the `index`th compute unit.

`x`
The id for the 1st dimension.

`y`
The id for the 2nd dimension.

`z`
The id for the 3rd dimension.

Notes

This function can be used to set the group ids on a single compute unit. Usually we would use this function to manually schedule the compute units. If for example we have a 1 dimension kernel with global size 64 and 2 compute units with work-group-size of 32 process elements, the first compute unit can execute the work-group 0, 0, 0 and the second compute unit the work-group 1, 0, 0.

Errors

plSetKernelComputeUnitGroupIds returns `PL_SUCCESS` if the argument was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_FOUND` if the group information or the compute unit was not found

plSetKernelGlobalOffsets

Set a kernel's global offset.

```
pl_int plSetKernelGlobalOffsets (pl_kernel kernel,
                                  pl_uint x,
                                  pl_uint y,
                                  pl_uint z)
```

Parameters

`kernel`
The target `pl_kernel` object.

`x`
The offset for the 1st dimension.

`y`
The offset for the 2nd dimension.

`z`
The offset for the 3rd dimension.

Notes

This function can be used to set the global offset on all the compute units. By setting a global offset, all the buffer arguments addresses will increased by the offset.

Errors

plSetKernelGlobalOffsets returns `PL_SUCCESS` if the argument was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_FOUND` if the group information was not found

plSetKernelComputeUnitGlobalOffsets

Set a kernel's global offset, on a single compute unit.

```
pl_int plSetKernelComputeUnitGlobalOffsets (pl_kernel kernel,
                                             size_t index,
                                             pl_uint x,
                                             pl_uint y,
                                             pl_uint z)
```

Parameters

`kernel`
: The target `pl_kernel` object.

`index`
: Target the `index`th compute unit.

`x`
: The offset for the 1st dimension.

`y`
: The offset for the 2nd dimension.

`z`
: The offset for the 3rd dimension.

Notes

This function can be used to set the global offset on all the compute units. By setting a global offset, all the buffer arguments addresses will increased by the offset.

Errors

plSetKernelComputeUnitGlobalOffsets returns `PL_SUCCESS` if the argument was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_FOUND` if the group information or the compute unit was not found

4.4.5 Kernel Execution functions

plRunTask

Run a kernel task.

```
pl_int plRunTask (pl_kernel kernel,
                  pl_bool blocking)
```

Parameters

`kernel`

The target `pl_kernel` object.

`blocking`

Blocking or non-blocking execution. Can take the value of:

Action	Description
`PL_TRUE`	Blocking execution
`PL_FALSE`	Non-blocking execution

Notes

This function just calls the first compute unit, by running:
`plRunKernelComputeUnit(kernel, 0, blocking)`
If non-blocking execution selected, you can wait for it by running:
`plWaitTask(kernel)`

Errors

plRunTask returns `PL_SUCCESS` if the argument was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_FOUND` if the compute unit was not found (impossible)

plWaitTask

Wait a kernel task.

```
pl_int plWaitTask (pl_kernel kernel)
```

Parameters

`kernel`

The target `pl_kernel` object.

Notes

This function just calls the wait first compute unit, by running:
`plWaitKernelComputeUnit(kernel, 0)`

Errors

plWaitTask returns `PL_SUCCESS` if the argument was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_FOUND` if the compute unit was not found (impossible)

plRunKernelComputeUnit

Run a kernel on a compute unit.

```
pl_int plRunKernelComputeUnit (pl_kernel kernel,
                               size_t index,
                               pl_bool blocking)
```

Parameters

`kernel`
The target `pl_kernel` object.
`index`
Target the `index`th compute unit.
`blocking`
Blocking or non-blocking execution. Can take the value of:

Action	Description
`PL_TRUE`	Blocking execution
`PL_FALSE`	Non-blocking execution

Notes

If non-blocking execution selected, you can wait for it by running:
`plWaitKernelComputeUnit(kernel, index)`

Errors

plRunKernelComputeUnit returns `PL_SUCCESS` if the argument was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_FOUND` if the compute unit was not found

plWaitKernelComputeUnit

Wait a kernel's compute unit.

```
pl_int plWaitKernelComputeUnit (pl_kernel kernel,
                                 size_t index)
```

Parameters

`kernel`
: The target `pl_kernel` object.

`index`
: Target the `index`th compute unit.

Notes

This function just wait the `index`th compute unit.

Errors

plWaitKernelComputeUnit returns `PL_SUCCESS` if the argument was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_FOUND` if the compute unit was not found

plRunNDRangeKernel

Schedule a kernel on all the compute units based on the work-size.

```
pl_int plRunNDRangeKernel (pl_kernel kernel,
                           pl_uint work_dim,
                           const size_t *global_work_offset,
                           const size_t *global_work_size,
                           pl_bool blocking)
```

Parameters

`kernel`
: The target `pl_kernel` object.

`work_dim`
: Number of work dimensions (grater than 0 and less or equal 3).

`global_work_offset`
: Array of global offset values. Must have the length of `work_dim` or be `NULL`. If `NULL`, zeros values will be used.

`global_work_size`

Array of work size values. Must have the length of `work_dim` or be `NULL`. If `NULL`, ones values will be used.

`blocking`

Blocking or non-blocking execution. Can take the value of:

Action	Description
`PL_TRUE`	Blocking execution
`PL_FALSE`	Non-blocking execution

Notes

This function schedules the kernel execution by distributing the work-tasks on all the compute units based on the work-size and the work-group-size of the work-groups. It firsts calculate the tasks, and then distributes them on the compute units. If the tasks are more than the compute units, the functions waits until a task complete and schedule the next one. As mentioned, if compute units are not enough, the function will execute in blocking mode until all the work is scheduled. If non-blocking execution selected, you can wait for it by running:
`plWaitNDRangeKernel(kernel)`

Errors

plRunNDRangeKernel returns `PL_SUCCESS` if execution successful. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized
`PL_NOT_VALID` if work dimension were not valid
`PL_FAILED` if an unknown error occurred

plWaitNDRangeKernel

Wait all kernel's compute units.

```
pl_int plWaitNDRangeKernel (pl_kernel kernel)
```

Parameters

`kernel`

The target `pl_kernel` object.

Notes

This function just wait all the compute units.

Errors

plWaitNDRangeKernel returns `PL_SUCCESS` if the argument was set successfully. Otherwise, one of the errors bellow will be returned:

`PL_NULL_POINTER` if `pl_kernel` was `NULL`
`PL_NOT_PROGRAMED` if the bitfile seems not to be programed on the PL
`PL_NOT_READY` if the kernel was not initialized

4.4.6 Help functions

plToolsLoadFileContents

Get the contents of a file.

```
char* plToolsLoadFileContents (char *filepath)
```

Parameters

`filepath`
The path to the target file.

Notes

After usage, the returned string should be released with `free`.

Errors

plToolsLoadFileContents returns the file's contents if successful. Otherwise, it will return a `NULL`.

plToolsCopyFileContents

Copy file contents from source to target.

```
pl_int plToolsCopyFileContents (char *source,
                                char* target)
```

Parameters

`source`
The path to the source file.
`target`
The path to the target file.

Notes

-

Errors

plToolsCopyFileContents returns `PL_SUCCESS` if execution successful. Otherwise, one of the errors bellow will be returned:

`PL_FILE_OPEN_FAILED` if failed to open source or target file

4.5 Comparison of SimplePL with OpenCL

In this section we will compare a program on SimplePL with the same program on OpenCL. The aim of this section is to point out the differences of those two programs and to give you an overview of the library, before we explain it on a deeper level in the upcoming sections.

We will compare the vector addition program, which is a common example for an introduction to OpenCL, thus, we will use it for an introduction to SimplePL. We will not compare kernels, we are focusing in the API similarities and the program flow at this section.

4.5.1 Program flow comparison

To start with, let us compare the program flow in titles between OpenCL and SimplePL.

OpenCL	**SimplePL**
Get Platforms Get Devices	 Initialize Device
Create Context Create Command Queue	
Create Program Build Program Create Kernel	Load bitfile info Program PL Initialize PL Get kernel info
 Create Buffers Enqueue Write Buffers	Get Memories Create Buffers Write Buffers
Set Kernel Arguments	Set Kernel Arguments
Enqueue Kernel Execute Wait Command Queue	Execute Kernel
Enqueue Read Buffers	Read Buffers

As we can see, there are not many differences. The initialization changes, in OpenCL you have to select an OpenCL device while on SimplePL you only have one device, as we are currently support only one programmable logic. Also, we are currently do not support command queues and contexts (we kept it simple). Of course, the program loading process changes as we are loading bit-streams.

Now let us check the API calls for the same program flow.

OpenCL API	**SimplePL API**
clGetPlatformIDs	
clGetDeviceIDs	plInitDevice
clCreateContext	
clCreateCommandQueue	
clCreateProgramWithSource	plGetBitfileInfo
clBuildProgram	plProgram
	plBitfileInitialize
clCreateKernel	plGetKernelByName
	plGetSysMemoryByIndex
clCreateBuffer	plCreateBuffer
clEnqueueWriteBuffer	plWriteBuffer
clSetKernelArg	plSetKernelArg
clEnqueueNDRangeKernel	plRunNDRangeKernel
clFinish	
clEnqueueReadBuffer	plReadBuffer

4.5.2 Program source comparison

In this subsection can compare the actual codes, the fist code is for OpenCL (you need the kernel source to run it) and the second code is for SimplePL (you need the bitfile and the description file to run it).

Here is the Vector Addition program in OpenCL (simple example with no error checking):

```c
/*
 * Vector Addition in OpenCL
 * Based on https://github.com/olcf/vector_addition_tutorials/blob/master/OpenCL/vecAdd.c
 */

// Libraries
#include <stdio.h>
#include <stdlib.h>
#include <CL/opencl.h>

// The kernel source
const char *kernel_source = "..." ;
// You need to define the kernel's source code here

int main() {
    // Length of vectors
    cl_uint n = 512;

    // Host input vectors
    cl_int *h_a;
    cl_int *h_b;
    // Host output vector
    cl_int *h_c;

    // Device input buffers
    cl_mem d_a;
    cl_mem d_b;
    // Device output buffer
    cl_mem d_c;

    cl_platform_id cpPlatform;        // OpenCL platform
    cl_device_id device_id;           // device ID
    cl_context context;               // context
    cl_command_queue queue;           // command queue
    cl_program program;               // program
    cl_kernel kernel;                 // kernel

    // Size, in bytes, of each vector
    size_t bytes = n * sizeof(cl_int);

    // Allocate memory for each vector on host
    h_a = (cl_int*) malloc(bytes);
    h_b = (cl_int*) malloc(bytes);
    h_c = (cl_int*) malloc(bytes);

    // Init rand
    time_t t;
    srand((unsigned) time(&t));

    int i;
    // Initialize vectors on host
    for (i = 0; i < n; i++) {
        h_a[i] = (rand() % 1000);
        h_b[i] = (rand() % 1000);
        h_c[i] = 0;
    }

    size_t globalSize, localSize;
    cl_int err;

    // Number of work items in each local work group
    localSize = 64;

    // Number of total work items - localSize must be devisor
    globalSize = ceil(n/(float)localSize)*localSize;

    // Bind to platform
    err = clGetPlatformIDs(1, &cpPlatform, NULL);

    // Get ID for the device
    err = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &
   device_id, NULL);

    // Create a context
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

    // Create a command queue
    queue = clCreateCommandQueue(context, device_id, 0, &err);

    // Create the compute program from the source buffer
    program = clCreateProgramWithSource(context, 1, (const char **) &
     kernelSource, NULL, &err);

    // Build the program executable
    clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

    // Create the compute kernel in the program we wish to run
    kernel = clCreateKernel(program, "vecAdd", &err);

    // Create the input and output arrays in device memory for our
    calculation
    d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL
    );
    d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL
    );
    d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bytes, NULL,
   NULL);

    // Write our data set into the input array in device memory
    err  = clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0, bytes, h_a,
   0, NULL, NULL);
    err |= clEnqueueWriteBuffer(queue, d_b, CL_TRUE, 0, bytes, h_b,
   0, NULL, NULL);

    // Set the arguments to our compute kernel
    err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
    err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
    err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
    err |= clSetKernelArg(kernel, 3, sizeof(cl_uint), &n);

    // Execute the kernel over the entire range of the data set
    err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize,
    &localSize, 0, NULL, NULL);

    // Wait for the command queue to get serviced before reading back
     results
    clFinish(queue);

    // Read the results from the device
    clEnqueueReadBuffer(queue, d_c, CL_TRUE, 0, bytes, h_c, 0, NULL,
   NULL );

    // Release OpenCL resources
    clReleaseMemObject(d_a);
    clReleaseMemObject(d_b);
    clReleaseMemObject(d_c);
    clReleaseProgram(program);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);

    // Release host memory
    free(h_a);
    free(h_b);
    free(h_c);

    return EXIT_SUCCESS;
}
```

Listing 4.1: Vector Addition in OpenCL

Below is the same Vector Addition program in SimplePL (again, simple example with no error checking):

```
/*
 * Vector Addition in SimplePL
 * Based on https://github.com/olcf/vector_addition_tutorials/blob/
    master/OpenCL/vecAdd.c
 */

// Libraries
#include <stdio.h>
#include <stdlib.h>
#include "simplePL.h"

// The bitfile path
const char *bitfile_path = "..." ;
// You need to add the path to the bitstream and the description file
     here

int main() {
    // Length of vectors
    pl_uint n = 512;

    // Host input vectors
    pl_int *h_a;
    pl_int *h_b;
    // Host output vector
    pl_int *h_c;

    // Device input buffers
    pl_buffer d_a;
    pl_buffer d_b;
    // Device output buffer
    pl_buffer d_c;

    pl_bitfile plBitfile;         // the bitstream object
    pl_kernel kernel;             // kernel
    pl_memory sysmemory;          // system memory for big buffers

    // Size, in bytes, of each vector
    size_t bytes = n * sizeof(pl_int);

    // Allocate memory for each vector on host
    h_a = (pl_int*) malloc(bytes);
    h_b = (pl_int*) malloc(bytes);
    h_c = (pl_int*) malloc(bytes);

    // Init rand
    time_t t;
    srand((unsigned) time(&t));

    int i;
    // Initialize vectors on host
    for (i = 0; i < n; i++) {
        h_a[i] = (rand() % 1000);
        h_b[i] = (rand() % 1000);
        h_c[i] = 0;
    }

    size_t globalSize, localSize;
    pl_int err;

    // Number of total work load
    size_t global = n;

    // Init device
    err = plInitDevice();

    // Get bitfile info
    err = plGetBitfileInfo(&plBitfile, bitfile_path, "vadd");

    // Program pl
    err = plProgram(plBitfile);

    // Initialize bitfile
    err = plBitfileInitialize(plBitfile, PL_BITFILE_INIT_ALL);

    // Get kernel
    kernel = plGetKernelByName(plBitfile, "vadd_int");

    // Get sysmemory
    sysmemory = plGetSysMemoryByIndex(0);

    // Disable global interrupt
    err = plSetKernelGlobalInterrupt(kernel, PL_DISABLE);

    // Create the input and output arrays in device memory for our
    calculation
    d_a = plCreateBuffer(sysmemory, bytes, &err);
    d_b = plCreateBuffer(sysmemory, bytes, &err);
    d_c = plCreateBuffer(sysmemory, bytes, &err);

    // Copy data to device
    err  = plWriteBuffer(d_a, 0, bytes, h_a);
    err |= plWriteBuffer(d_b, 0, bytes, h_b);

    // Set parameters
    err  = plSetKernelArg(kernel, 0, sizeof(pl_buffer), (void *) &d_a
    );
    err |= plSetKernelArg(kernel, 1, sizeof(pl_buffer), (void *) &d_b
    );
    err |= plSetKernelArg(kernel, 2, sizeof(pl_buffer), (void *) &d_c
    );
    err |= plSetKernelArg(kernel, 3, sizeof(pl_uint), (void *) &n);

    // Run kernel (Blocking)
    err = plRunNDRangeKernel(kernel, 1, NULL, &global, PL_TRUE);

    // Copy data from device
    err = plReadBuffer(d_c, 0, bytes, h_c);

    // Release SimplePL resources
    plReleaseBuffer (d_a);
    plReleaseBuffer (d_b);
    plReleaseBuffer (d_c);
    plBitfileRelease (plBitfile); // This also releases kernels and
   memories

    // Release host memory
    free(h_a);
    free(h_b);
    free(h_c);

    return EXIT_SUCCESS;
}
```

Listing 4.2: Vector Addition in SimplePL

5 Results

In this section, we will analyze some of the results we got from the demos we created to test our system.

We will present you some timings we recorded for various tasks. Please note that all these timings were recorded on our test-bench system, a “Zedboard“ development board (Zynq-7000 ARM/FPGA SoC based), which is not a state of the art FPGA board. The following timings were recorded by running a simple matrix multiplication kernel (mmult) which can be found on the Appendix section. In order to minimize external interference, we are focusing on the minimum timings recorded for each task during the benchmark, thus, the best performance we got (not the average), we do though mentions average and maximum timings where needed.

To record the timings, we executed 100 times each kernel with a 10 ms pause between each execution. The timing recording were implemented on C, inside the application’s source code, with the use of a small C library we created, named “benchtimer“. This library was initially created for the benchmarks we executed on CPU/GPU devices that you can see on a later chapter.

5.1 Program PL

In order to use an HLS OpenCL kernel, we first have to download the bitfile that contains it on the FPGA. This process can be compared with the “build program“ process of the OpenCL, where we build our OpenCL kernels from source on runtime.

We recorded the “program time“ as the time to execute the following SimplePL functions:

plGetBitfileInfo
plProgram
plBitfileInitialize
plGetKernelByName

So, we first load and parse the Bitfile Description File. Then, we load the bitfile part by part on a buffer and write it on the PL program driver. Next, we initialize the bitfile’s components and lastly get the kernel by name.

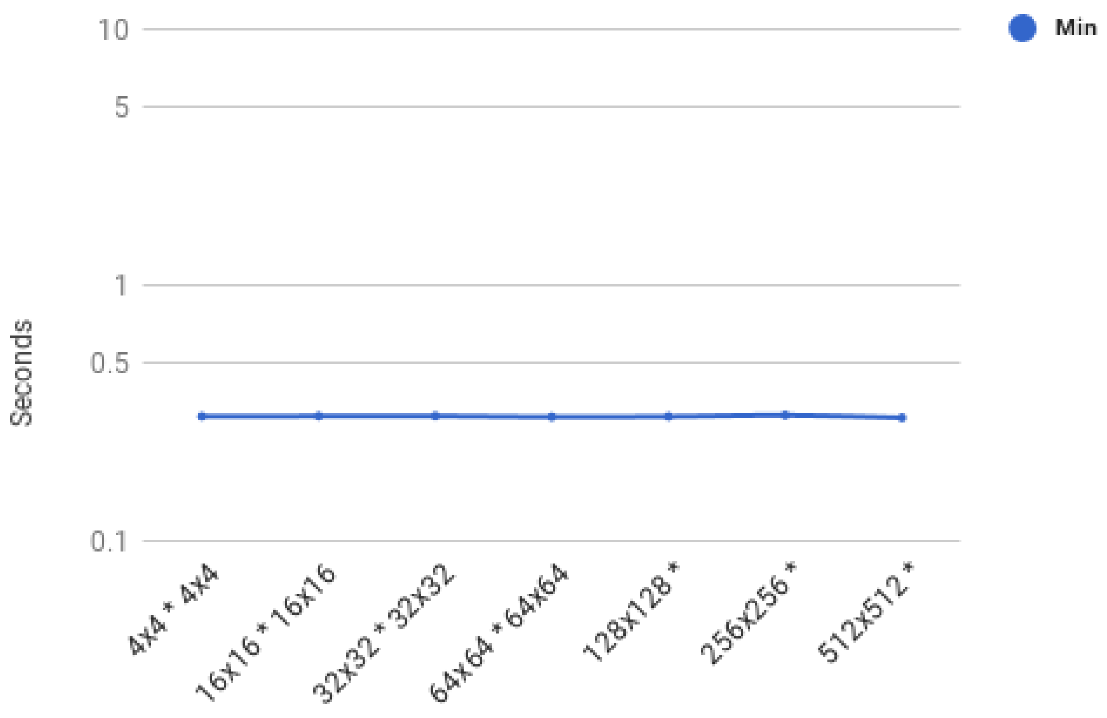


Figure 5.1: Minimum program time of a bitstream in SimplePL (lower is better)

Matrices Sizes	Min	Avg	Max
4x4 * 4x4	0.306580	0.315163	0.351515
16x16 * 16x16	0.307357	0.314225	0.328299
32x32 * 32x32	0.307616	0.313604	0.325809
64x64 * 64x64	0.305072	0.316196	0.53134
128x128 * 128x128	0.306261	0.314989	0.322085
256x256 * 256x256	0.309996	5.981321	22.825676
512x512 * 512x512	0.302770	0.971742	4.439915

Figure 5.2: Program time (in secs) of a bitstream using SimplePL

These results were recorded on a "SanDisk Ultra 16GB Ultra Micro SDHC UHS-I Class 10" which features up to 80 MB/s transfer speeds. We can see that the PL program takes about 300 ms to 320 ms, but in some cases up to 22 secs. This was caused by the fact that, the PL program driver was tying to allocate contiguous memory for the bitstream and this process some times failed with an error:

```
alloc_contig_range: [<hex>, <hex>) PFNs busy
```

It may be a bug or just random behavior. Usually it works with no problems.

This timings are affected by the read speed of the SD card we used on our system. As an example we inserted a faster SD card on the system, the "Sandisk Extreme Pro SDXC U3 UHS-I" which features up to 95 MB/s read speeds. With the new faster SD card, we recorded PL programming timing from 280 ms to 290 ms.

We had to find out the real PL program time. So we recorded the time to program the FPGA with the bitstream already loaded on the memory. Here are the timings we got.

Matrices Sizes	Min	Avg	Max
4x4 * 4x4	0.182700	0.186153	0.194504
16x16 * 16x16	0.182737	0.187814	0.194330
32x32 * 32x32	0.185373	0.188375	0.193493
64x64 * 64x64	0.182714	0.186459	0.195363
128x128 * 128x128	0.182221	0.189059	0.194191
256x256 * 256x256	0.183090	0.189009	0.201151

Figure 5.3: Raw PL program time (in secs) (data)

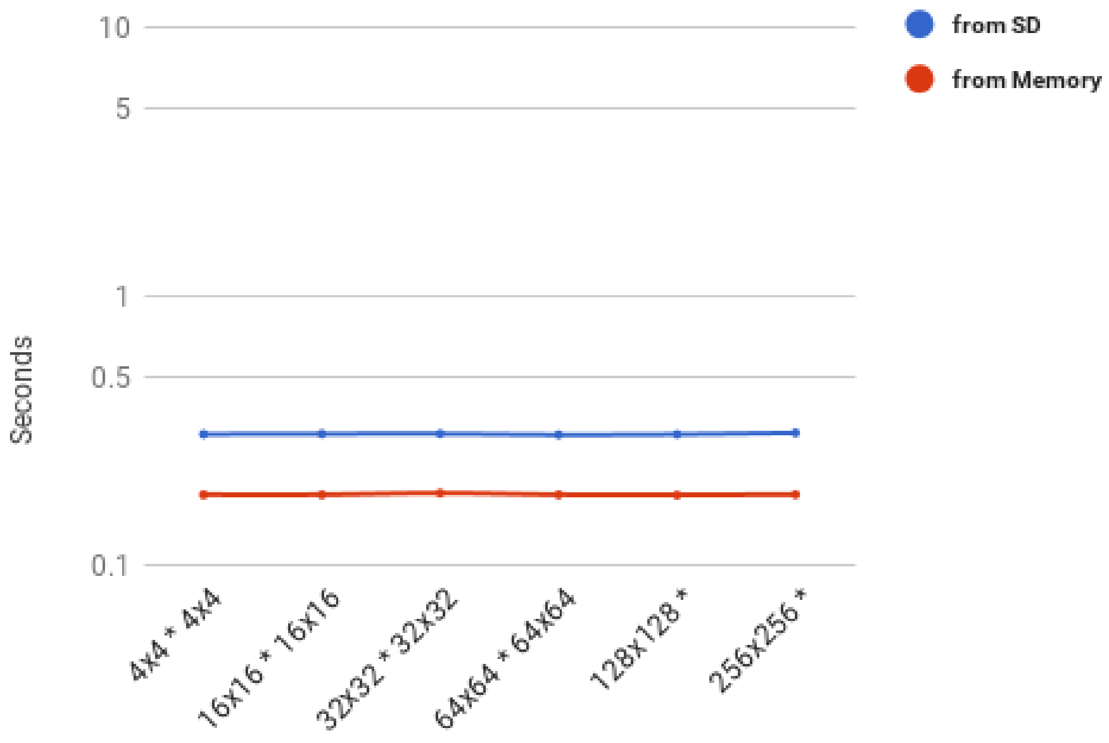


Figure 5.4: PL program times (lower is better)

5.2 Data transfer

As we explained on a previous section, the OpenCL model describes one host memory and one device memory. Data from the host memory has to be moved to the device memory for a kernel to be able to access them and vice versa to be able to get the results.

In this point we have to remind you that in SimplePL, we have 2 types of device memories based on where the memory is physically located. We can create device buffers on the main memory of our system or on BRAMs inside our FPGA. Buffers on the main memory can be bigger as BRAMs size is limited.

We recorded the “send data time“ as the time to write the data on the buffers and set the kernel arguments, so the following SimplePL functions:

plWriteBuffer
plSetKernelArg

The charts below show the data transfer from host to the device, using parts of the system's memory as device memory. Hence, this are timings to transfer data inside the system's memory.

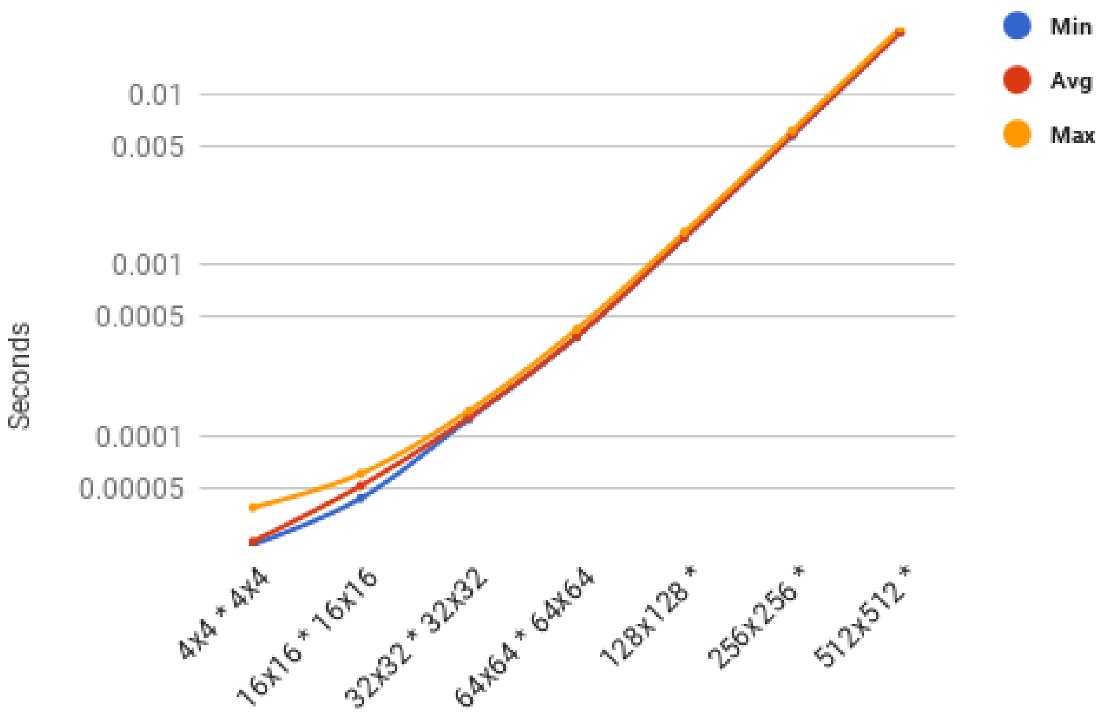


Figure 5.5: Send data times 2*Size*Size*4 + 6*4 bytes (lower is better)

Matrices Sizes	Size	Min	Avg	Max
4x4 * 4x4	152 B	0.000023	0.000024	0.000038
16x16 * 16x16	~ 2 KB	0.000043	0.000051	0.00006
32x32 * 32x32	~ 8 KB	0.000125	0.000128	0.00014
64x64 * 64x64	~ 32 KB	0.000379	0.000382	0.000421
128x128 * 128x128	~ 128 KB	0.001448	0.001456	0.001562
256x256 * 256x256	~ 512 KB	0.005763	0.00583	0.006169
512x512 * 512x512	~ 2 MB	0.023201	0.023245	0.024641

Figure 5.6: Send data times (in secs) 2*Size*Size*4 + 6*4 bytes (data)

We get similar results for the reverse data transfer. Again this is moving data inside the system's memory. We recorded the "receive data time" as the time to read the results data on a buffer, so the following SimplePL function:

plReadBuffer

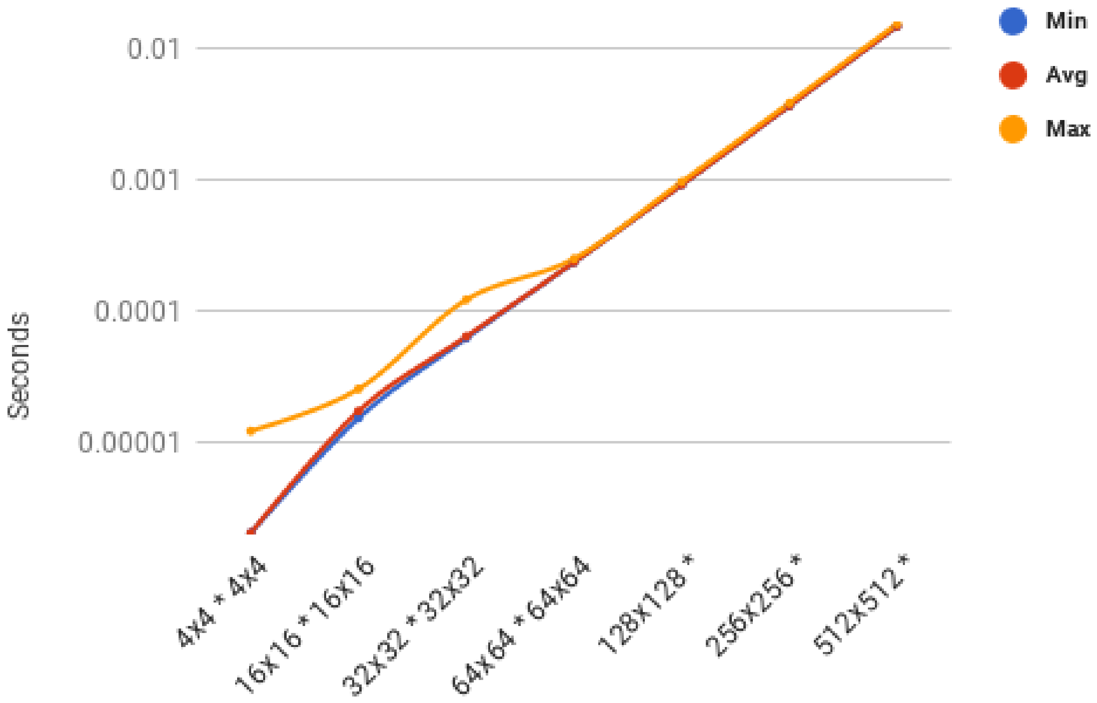


Figure 5.7: Receive data times Size*Size*4 bytes (lower is better)

Matrices Sizes	Size	Min	Avg	Max
4x4 * 4x4	64 B	0.000002	0.000002	0.000012
16x16 * 16x16	1 KB	0.000015	0.000017	0.000025
32x32 * 32x32	4 KB	0.000061	0.000063	0.000120
64x64 * 64x64	16 KB	0.000231	0.000232	0.000247
128x128 * 128x128	64 KB	0.000902	0.000904	0.000952
256x256 * 256x256	256 KB	0.003612	0.003625	0.003806
512x512 * 512x512	1 MB	0.014619	0.014677	0.015293

Figure 5.8: Receive data times (in secs) Size*Size*4 bytes (data)

5.3 Execution

This example was created as a proof-of-work. It was not meant to yield performance, thus we will not evaluate its performance. The bitstream has 9 compute units (for no particular reason), that each one can execute a 32×32 work-group (or smaller). SimplePL knew these informations from the bitfile description file. The kernel can handle up to 4096 dimension length matrices.

We recorded the “execution time“ as the time to schedule and run the compute task execution across our compute units, using the SimplePL function:

plRunNDRangeKernel

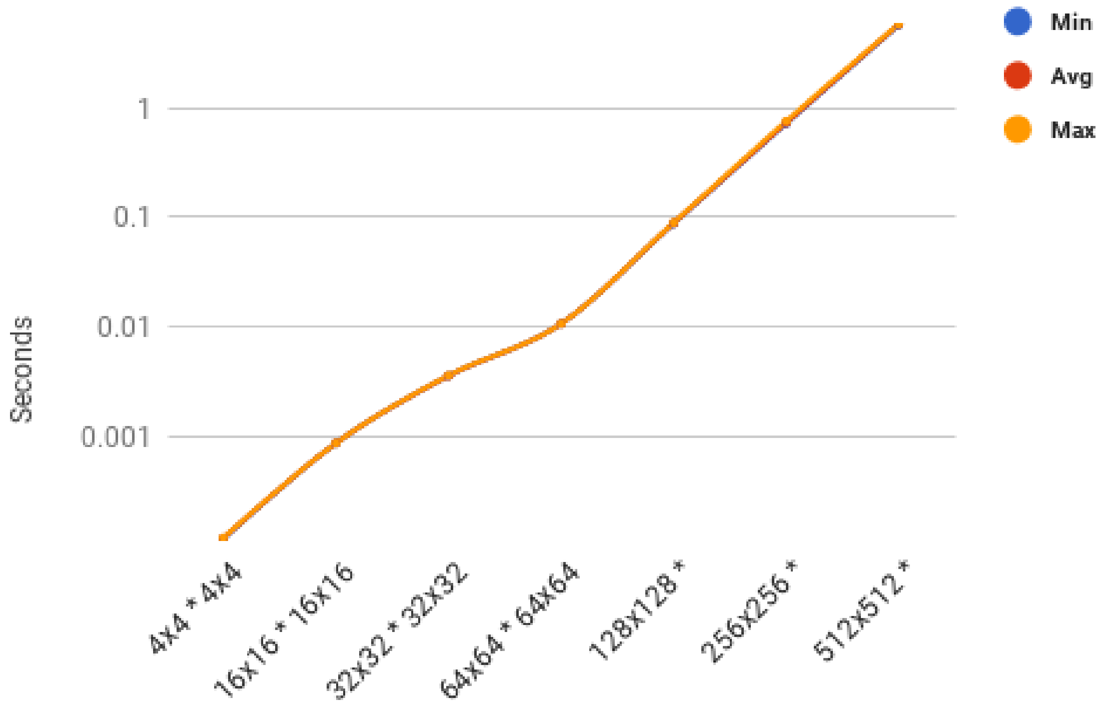


Figure 5.9: Execution times (lower is better)

Matrices Sizes	Work-Groups	Min	Avg	Max
4x4 * 4x4	1 (1 loop)	0.000113	0.000115	0.000116
16x16 * 16x16	1 (1 loop)	0.000854	0.000856	0.000857
32x32 * 32x32	1 (1 loop)	0.003540	0.003542	0.003569
64x64 * 64x64	4 (1 loop)	0.010641	0.010643	0.010653
128x128 * 128x128	8 (1 loop)	0.088266	0.089092	0.090031
256x256 * 256x256	64 (8 loops)	0.728341	0.739033	0.765904
512x512 * 512x512	256 (29 loops)	5.821247	5.907504	5.976080

Figure 5.10: Execution times in secs (data)

Since our bitstream has 9 compute units, if we have more than 9 work-groups, our scheduling will wait for a compute unit to finish. If we assume that the 9 compute units will finish in the same time, we can calculate how many times we will have to restart the compute units with new work-groups, by dividing the number of work-groups with the number of our compute units. We named this info in the table above as “loops“.

6 Future Work

In this section, we will express our ideas about future work on SimplePL.

6.1 Support for HLS C/C++ Functions

We started SimplePL to support HLS OpenCL Kernels, but it can be extended to support HLS IP cores in general. By supporting multiple IP core interfaces we may be able to control IP core generated from a C/C++ function too. Then, SimplePL would be able to call HLS Function on the PL. This feature will let the applications that use SimplePL execute pre-synthesized functions on the hardware. Such a functionality is needed, as we can not always express our algorithms as OpenCL kernels.

6.2 Support for more boards

At the moment, SimplePL only support Zedboard out of the box. Support for more boards can be added by detecting the available multiple PL program driver of the system or defining it on the code. Since, this depends on the Linux driver that each board has to program its FPGA, we can add out of the box support for the most popular ones. To run the SimplePL on an unsupported FPGA board, you now have to adjust the PL program functions and call the correct driver with the correct parameters.

6.3 OpenCL C API Wrappers

Many applications that use OpenCL follows a simple program flow, they get an OpenCL device, create buffers, copy their data, run the kernel and get the results. To port such applications we just have to replace the OpenCL API calls with the respective SimplePL ones. For those applications, we can provide an OpenCL C API wrappers library with a subset of the OpenCL API that can be matched to the SimplePL API. With those wrappers, we would be able to port from GPU/CPU OpenCL applications with probably no code changes.

6.4 Support for more options

Include support of more options to help automatic performance speedup. Depending on the demand, more functions can be implemented on the API, to cover all the basic needs of an application that works with OpenCL HLS kernels. Such changes can be made based on the feedback we get from the community. The feedback will help us find out missing components that could be added on the SimplePL.

7 Lessons learned

During the development of this project, we learned many new things and we faced problems that we had to solve. We red documentations, followed guides and we used our previous knowledge to provide solutions. The learning experience was not always the best, as many times, in new projects / tools there is not enough documentation resources which consumes you development time. Hopefully, projects like this, with open source code and helpful guides will help future developers with similar problems.

7.1 OpenCL

Prior to this work, we had only implemented some OpenCL examples. For this project we leaned OpenCL and how its hole ecosystem works. Our knowledge over vector architectures helped us understand the OpenCL model and how work-groups are works and are implemented.

At first, it was not clear how the hole OpenCL devices works in a system. Most of the getting starting guides does not explain the OpenCL ICD (Installable Client Driver), as a result, the application execution flow was not clear in the beginning. As soon as you understand it , and install your drivers, it is relative simple to execute OpenCL kernels.

7.2 Poor Documentation

From my point of view, releasing a coding tool/library without documentation is as if it was never released. Releasing something with poor documentation makes it difficult to learn it and demands an experienced user. Many tools that are addressed to a specific audience provide insufficient documentation for a new user.

We faced difficulty in using IP cores created from HLS kernel. As you know these IP cores are created using the Xilinx's Vivado HLS tool. Vivado HLS official manuals and guides only reference the capability of creating IP cores from OpenCL kernels. Instead, to understand the usage, the optimization or the implementation of HLS OpenCL kernels, you have to follow unofficial guide, use you HLS C/C++ functions knowledge or test codes from Xilinx's SDAccel's manuals and documentation.

SDAccel shares the same guidelines on writing HLS kernels, but documents nothing over its IP core implementation as the tool handles all the hardware implementation and functionality. At least, during our project's development, we witness improvements in the SDAccel's documentation with full application code examples.

We understand that due to the relative young age of these tools, there are not sufficient documentation yet. Moreover, we think that Xilinx's focused marketing on the SDAccel suite, made the company neglect Vivado HLS.

7.3 Hardware binding with Software

In the past, we work on both software projects and on hardware projects, but rarely did we worked on a project involving both of them in such a low level. It was a the first time that we combined them to create a system that bonds them together. This was an opportunity for us to learn what the development deferences are in such a project.

We had the theoretical knowledge of how these software - hardware bindings and mechanics works, like the virtual addresses, the device mapping to memory or the kernel modules but we never directly used them or implemented them. It was a new experience for us to execute code completely based on our theoretical background and it worked with no problems.

This project gave us the opportunity to develop and work with kernel modules, which play a major role in controlling hardware from software. Thus, we expand our knowledge over the Linux Kernel, its abilities and we now have a better understanding of it.

7.4 Creating a good Structure

When you start the development of a library or a system, you may have to create a model of your system before you start the implementation. Your model has to be well defined from the beginning and sometimes it's good to be extensible so that you can adjust it if needed or expendable for its future versions. This is a huge problem for many inexperienced programmers as they usually have to go back and re-script their code.

In this project we tried to follow the OpenCL model and structure, by extending it were needed. We created a basic schema before the implementation and re-shape it along the way. Keeping a good code structure that follows the model was crucial one of our system's goal was to implement an OpenCL like experience for the programmer. This was also a new experience for us, as our implementation had to fit into the model and follow its rules.

8 OpenCL Benchmarks

8.1 Motivation

Executing OpenCL on GPUs has proven to give quite a performance. Usually, the better the GPU the better the performance we could get. With this in mind, we wanted to benchmark some of the OpenCL device we could get our hands on. We were interested in measuring tasks like data transfer, compile times and execution times. We wanted to included on the benchmarks not only GPU OpenCL devices, but also CPU OpenCL devices. In addition, were interested of getting results of devices from different vendors on many systems.

8.2 Introduction

You can find OpenCL devices on every modern computer. All major GPU and CPU vendors supports OpenCL by installing their respective SDKs (except may be ARM CPUs). We coded a simple OpenCL GEMM (General Matrix Multiply) kernel and we developed an application to execute it, so that we can measure timings of some OpenCL tasks. We executed the same application on different OpenCL devices and on different OS (Windows and Linux). In these benchmarks our aim was compare the speed of the same task on different test-bench machines or different drivers. We have to point out that in our benchmarks, we record the performance of the system that the OpenCL device run, as a result, any poor or good performance we recorded has to be attributed on the hole system and not only on the OpenCL device.

8.3 Benchmarks on CPU/GPU architectures

You can find OpenCL devices on every modern computer. All major GPU and CPU vendors supports OpenCL by installing their respective SDKs (except may be ARM CPUs). We coded a simple OpenCL GEMM (General Matrix Multiply) kernel and we developed an application to execute it, so that we can measure timings of some OpenCL tasks. We executed the same application on different OpenCL devices and on different OS (Windows and Linux). In these benchmarks our aim was compare the speed of the same task on different test-bench machines or different drivers.

8.3.1 Benchmark Kernel

Here is the GEMM kernel that we used to run the benchmarks. It features quite an intensive memory usage. This kernel may favor some of our OpenCL devices in execution performance.

```
// Kernel Code
__kernel void gemm_nn(
    // number of rows of op( A ) and C
    const unsigned int M,
    // number of columns of op( B ) and C
    const unsigned int N,
    // number of columns of op( A ) and number of rows of op( B )
    const unsigned int K,

    const int alpha,
    __global int* A,
    __global int* B,
    const int beta,
    __global int* C
){

    // Get Global ids
    const int x = get_global_id(0);
    const int y = get_global_id(1);

    // Check range
    if(x >= M || y >= N){
        return;
    }

    // Init value
    int value = 0;

    // Calculate value
    for (int i = 0; i < K; i++){
        value += alpha * A[x*K + i] * B[i*M + y];
    }

    // C = alpha + op(A) * op(B) + beta * C
    C[x*M + y] = value + beta * C[x*M + y];
}
```

8.3.2 OpenCL Devices

Since an OpenCL device is not just a hardware device, but also the software that controls it, we consider the same hardware device controlled by a different driver to be a different OpenCL device. To explain this better, we can take for example the “pocl“ implementation on a CPU device, in comparison with the same CPU device controlled by its vendor’s OpenCL drivers.

With all these in mind, these are the OpenCL devices we run our benchmarks on:

Hardware Device	OS	Driver	Type	Info
i5-2410M	Debian	Intel	CPU	2 cores, 4 threads
i7-3540M	Ubuntu	Intel	CPU	2 cores, 4 threads
FX-8350	Windows	AMD	CPU	8 cores, 8 threads
FX-8350	Ubuntu	pocl	CPU	8 cores, 8 threads
GT-540M	Debian	NVIDIA	GPU	96 cores
GTX 660	Windows	NVIDIA	GPU	960 cores
R9 270	Windows	AMD	GPU	1280 cores, 20 compute units
GTX 980	CentOS	NVIDIA	GPU	2048 cores

Of course a GPU's core is not equal in power with a CPU's core, but what the GPU core's lack in power they gain in quantity. As a result, many OpenCL process-elements (may be a hole work-group) will be assigned on 1 CPU core, while 1 process-element will be assigned to 1 GPU core. Thus, as we mention on a previous section, a CPU's core is a compute unit while a GPU's core is a process-element. This is may not always be the case because it's the OpenCL driver's job to take care of the scheduling task in an efficient way.

The device specifications give us info about its OpenCL elements. For example, "R9 270" has 1280 cores forming 20 compute units, as a result we know that each compute unit has $1280/20 = 64$ process elements. The OpenCL driver has to know these numbers to schedule the execution of a kernel and to optimize the OpenCL kernel's code during the build process.

For the benchmarks, we executed 100 times each kernel with a 10 ms pause between each execution. To minimize external interference, we are showing the minimum timing recorded of each task during the benchmark, thus, the best performance we got (not the average).

8.3.3 Build time

Since the majority of the OpenCL applications build the kernels from source on runtime, we should measure this time and check whether it is neglect-able or not. This time is not only related to the OpenCL device. It is related to the complexity of the device's architecture, but also, to the performance of the system's CPU and the OpenCL driver in use.

We recorded the "build time" as the time to execute the following OpenCL functions:

clCreateProgramWithSource
clBuildProgram
clCreateKernel

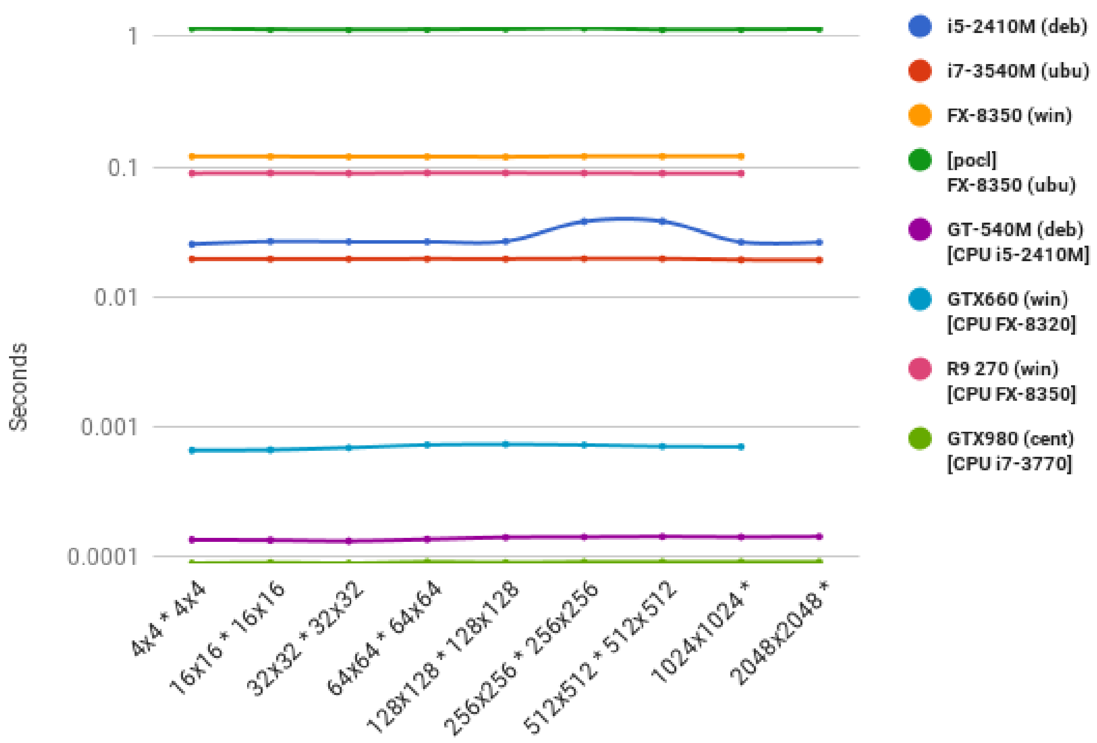


Figure 8.1: Minimum build time of a GEMM OpenCL algorithm (lower is better)

Matrices Sizes	i5-2410M (deb)	i7-3540M (ubu)	FX-8350 (win)	[pocl] FX-8350 (ubu)	GT-540M (deb) [i5-2410M]	GTX660 (win) [FX-8320]	R9 270 (win) [FX-8350]	GTX980 (cent) [i7-3770]
4x4 * 4x4	0.025305	0.019505	0.119754	1.153509	0.000136	0.000660	0.088828	0.000090
16x16 * 16x16	0.026647	0.019510	0.119560	1.136129	0.000135	0.000666	0.089271	0.000091
32x32 * 32x32	0.026486	0.019525	0.119113	1.130714	0.000133	0.000693	0.088716	0.000090
64x64 * 64x64	0.026482	0.019555	0.119303	1.135830	0.000137	0.000728	0.089505	0.000092
128x128 * 128x128	0.026656	0.019542	0.118874	1.144375	0.000142	0.000736	0.089363	0.000091
256x256 * 256x256	0.037921	0.019645	0.119953	1.158543	0.000143	0.000727	0.089112	0.000092
512x512 * 512x512	0.038020	0.019623	0.120016	1.130503	0.000144	0.000710	0.088804	0.000092
1024x1024 * 1024x1024	0.026318	0.019245	0.120137	1.134502	0.000143	0.000705	0.088655	0.000092
2048x2048 * 2048x2048	0.026232	0.019163	-	1.142421	0.000144	-	-	0.000092

Figure 8.2: Minimum build time (in secs) of a GEMM OpenCL algorithm (data)

The size of the data is not a parameter on the build process, therefor, our chart has one flat line for each OpenCL device.

As we can see, "NVIDIA"'s driver features the fastest build timings. In the second place is "Intel"'s drivers followed by the "AMD"'s drivers. The slowest timings were recorded from the "pocl" driver which was at least 10 times slower.

We may have to question our "NVIDIA"'s driver data, as the build time is too small, suggesting that maybe cache mechanics where involved, but, never the less, these were the actual timings we recorded. Let's also peek at the max timings we recorded.

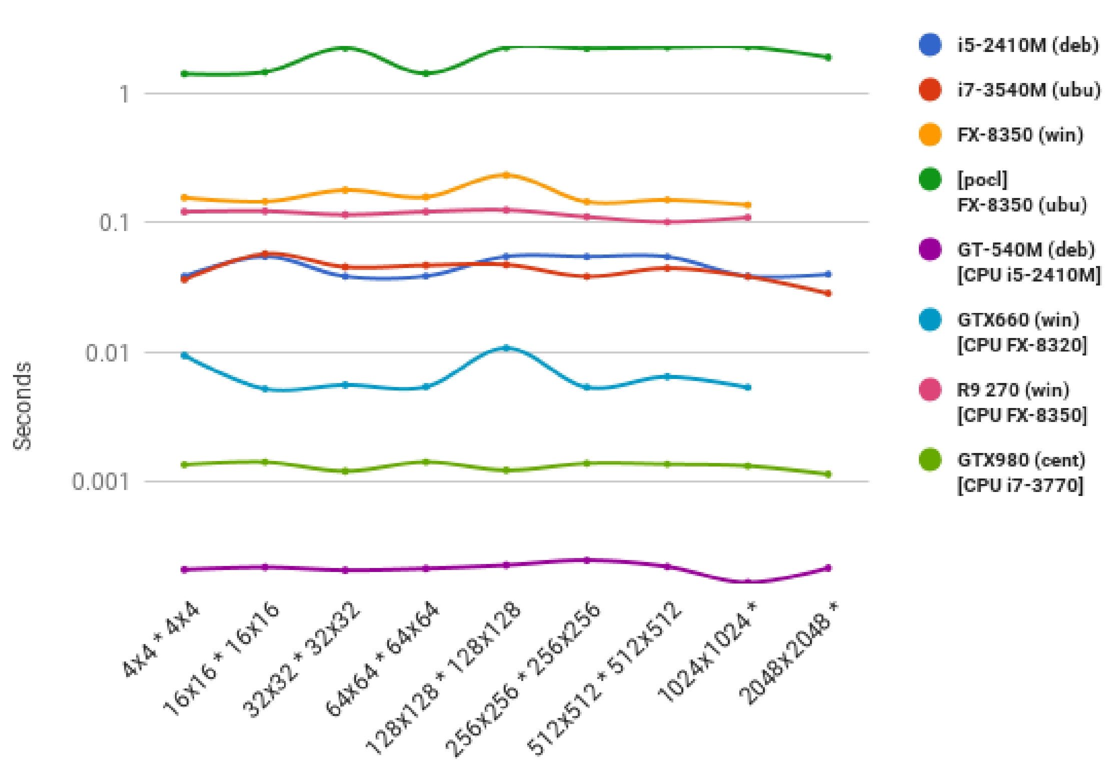


Figure 8.3: Maximum build time of a GEMM OpenCL algorithm (lower is better)

Matrices Sizes	i5-2410M (deb)	i7-3540M (ubu)	FX-8350 (win)	[pocl] FX-8350 (ubu)	GT-540M (deb) [i5-2410M]	GTX660 (win) [FX-8320]	R9 270 (win) [FX-8350]	GTX980 (cent) [i7-3770]
4x4 * 4x4	0.038492	0.036221	0.156323	1.414389	0.000207	0.009359	0.121503	0.001341
16x16 * 16x16	0.054655	0.057401	0.145543	1.461112	0.000216	0.005177	0.122666	0.001405
32x32 * 32x32	0.038407	0.045390	0.178360	2.251992	0.000205	0.005562	0.115340	0.001194
64x64 * 64x64	0.038599	0.046722	0.158100	1.424578	0.000211	0.005375	0.121518	0.001406
128x128 * 128x128	0.054687	0.047226	0.232756	2.271447	0.000224	0.010709	0.125034	0.001213
256x256 * 256x256	0.054528	0.038320	0.144999	2.238416	0.000244	0.005329	0.111008	0.001372
512x512 * 512x512	0.054393	0.044462	0.150182	2.280954	0.000218	0.006422	0.101032	0.001354
1024x1024 * 1024x1024	0.038523	0.038241	0.136993	2.297039	0.000164	0.005317	0.109664	0.001311
2048x2048 * 2048x2048	0.039889	0.028407	-	1.894123	0.000213	-	-	0.001131

Figure 8.4: Maximum build time (in secs) of a GEMM OpenCL algorithm (data)

We still can't be sure about what "NVIDIA"'s driver does, but it's obvious that "pocl"'s driver has to do something to improve its timings.

8.3.4 Send and Receive data times

In OpenCL we need to move the data to the device's memory to process them and then get the results back to the host's memory. In the following chart, we display the minimum times of transferring 3 buffers from the host to the device (3 matrices) we recorded.

We recorded the "send data time" as the time to execute the following OpenCL functions (and wait queue):

clCreateBuffer
clEnqueueWriteBuffer
clSetKernelArg

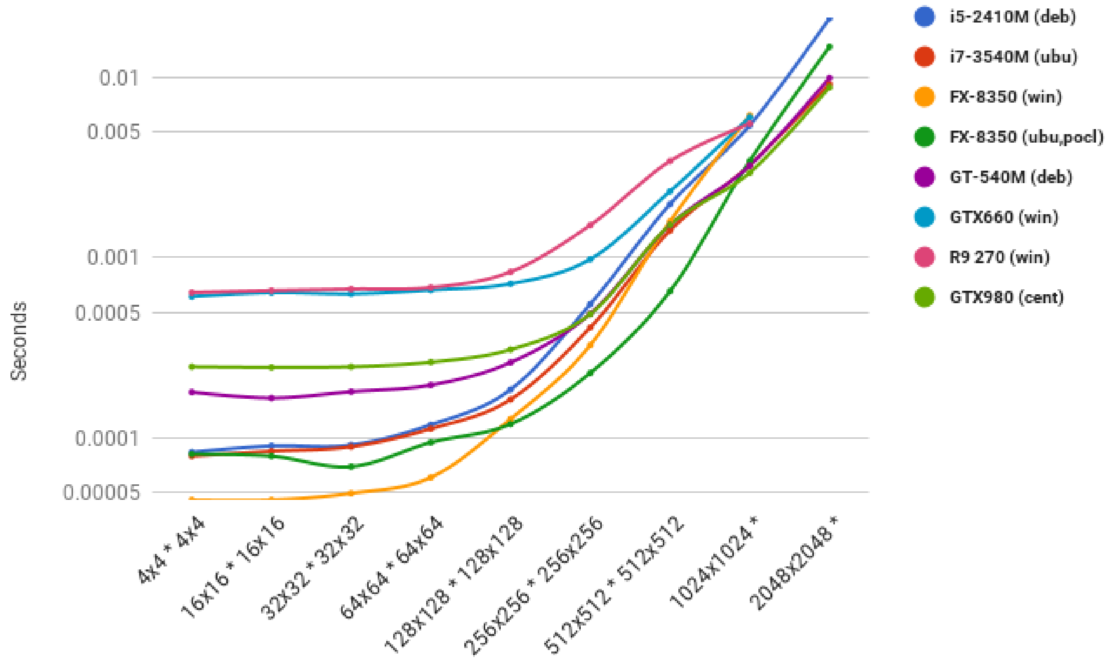


Figure 8.5: Minimum send data times 3*Size*Size*4 + 8*4 bytes (lower is better)

Matrices Sizes	i5-2410M (deb)	i7-3540M (ubu)	FX-8350 (win)	[pocl] FX-8350 (ubu)	GT-540M (deb) [i5-2410M]	GTX660 (win) [FX-8320]	R9 270 (win) [FX-8350]	GTX980 (cent) [i7-3770]
4x4 * 4x4	0.000083	0.000079	0.000045	0.000081	0.000179	0.000609	0.000641	0.000248
16x16 * 16x16	0.000090	0.000084	0.000045	0.000079	0.000166	0.000641	0.000657	0.000246
32x32 * 32x32	0.000091	0.000089	0.000049	0.000069	0.000180	0.000631	0.000670	0.000248
64x64 * 64x64	0.000118	0.000112	0.000060	0.000094	0.000196	0.000662	0.000684	0.000263
128x128 * 128x128	0.000185	0.000163	0.000127	0.000119	0.000262	0.000718	0.000836	0.000309
256x256 * 256x256	0.000553	0.000410	0.000327	0.000229	0.000489	0.000983	0.001522	0.000487
512x512 * 512x512	0.001988	0.001420	0.001597	0.000652	0.001522	0.002345	0.003457	0.001529
1024x1024 * 1024x1024	0.005462	0.003279	0.006205	0.003463	0.003258	0.006071	0.005588	0.002977
2048x2048 * 2048x2048	0.021499	0.009286	-	0.014925	0.010032	-	-	0.008889

Figure 8.6: Minimum send data times (in secs) 3*Size*Size*4 + 8*4 bytes (data)

As we can observe the GPU timings are greater than the CPU timings for relative small data sizes. Also, we can see that the slowest performance was recorded on GPUs running on a Windows machines.

On the other hand, we recorded the “receive data time“ as the time to execute the following OpenCL functions (and wait queue):

clEnqueueReadBuffer

In the following chart, we plot the minimum times of transferring a buffer from the device to the host (1 matrix) that we recorded.

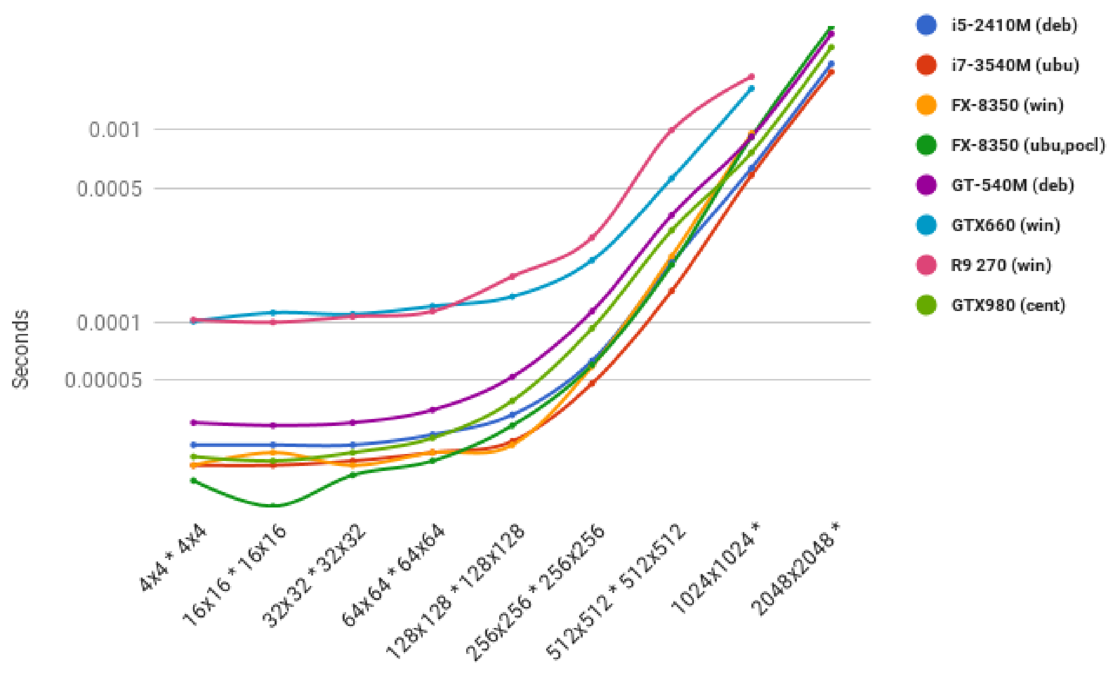


Figure 8.7: Minimum receive data times Size*Size*4 bytes (lower is better)

Matrices Sizes	i5-2410M (deb)	i7-3540M (ubu)	FX-8350 (win)	[pocl] FX-8350 (ubu)	GT-540M (deb) [i5-2410M]	GTX660 (win) [FX-8320]	R9 270 (win) [FX-8350]	GTX980 (cent) [i7-3770]
4x4 * 4x4	0.000023	0.000018	0.000018	0.000015	0.000030	0.000101	0.000103	0.000020
16x16 * 16x16	0.000023	0.000018	0.000021	0.000011	0.000029	0.000112	0.000100	0.000019
32x32 * 32x32	0.000023	0.000019	0.000018	0.000016	0.000030	0.000110	0.000107	0.000021
64x64 * 64x64	0.000026	0.000021	0.000021	0.000019	0.000035	0.000121	0.000114	0.000025
128x128 * 128x128	0.000033	0.000024	0.000023	0.000029	0.000052	0.000136	0.000173	0.000039
256x256 * 256x256	0.000063	0.000048	0.000059	0.000060	0.000114	0.000210	0.000276	0.000093
512x512 * 512x512	0.000205	0.000146	0.000221	0.000199	0.000361	0.000561	0.001001	0.000301
1024x1024 * 1024x1024	0.000635	0.000583	0.000960	0.000922	0.000922	0.001647	0.001899	0.000765
2048x2048 * 2048x2048	0.002212	0.002007	-	0.003449	0.003169	-	-	0.002698

Figure 8.8: Minimum receive data times (in secs) Size*Size*4 bytes (data)

Again, similar results with the host to device data transfer.

8.3.5 Execution time

Lets not see the execution times of the GEMM kernel on each device. In this point we have to say that the kernel is not optimized and it's up to each device's drive on how will the kernel be executed. We remind you that this kernel may not be the best for benchmarking.

We recorded the "execution time" as the time to enqueue the OpenCL kernel's execution (and wait queue):

clEnqueueNDRangeKernel

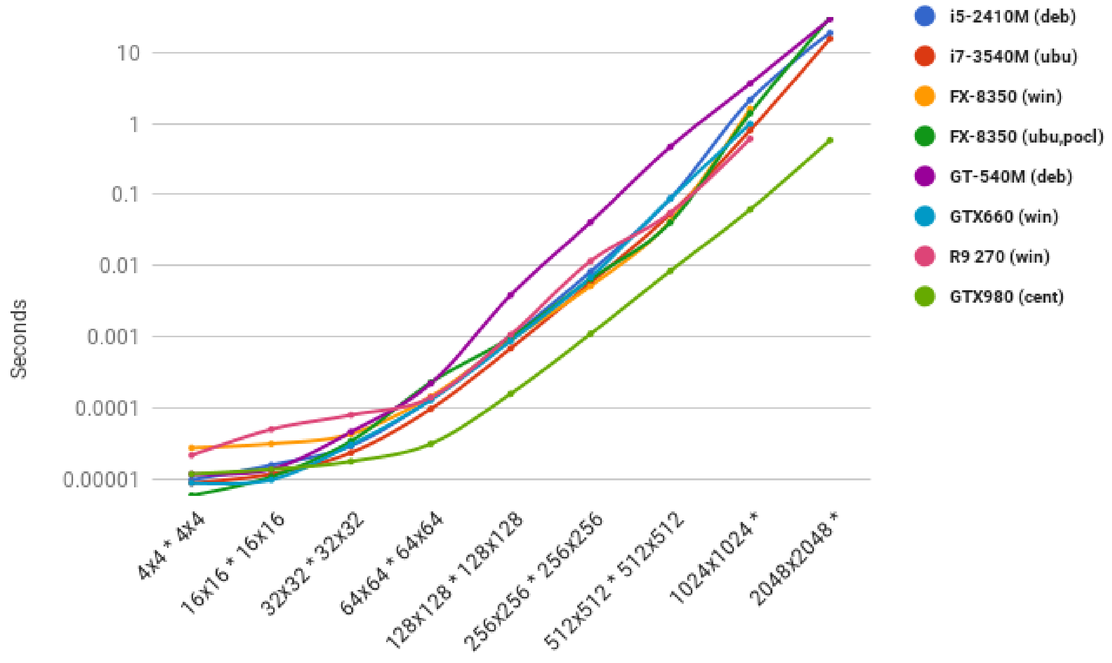


Figure 8.9: Minimum execution times (lower is better)

Matrices Sizes	i5-2410M (deb)	i7-3540M (ubu)	FX-8350 (win)	[pocl] FX-8350 (ubu)	GT-540M (deb) [i5-2410M]	GTX660 (win) [FX-8320]	R9 270 (win) [FX-8350]	GTX980 (cent) [i7-3770]
4x4 * 4x4	0.000010	0.000009	0.000028	0.000006	0.000012	0.000009	0.000022	0.000012
16x16 * 16x16	0.000016	0.000012	0.000032	0.000011	0.000014	0.000010	0.000051	0.000014
32x32 * 32x32	0.000030	0.000024	0.000044	0.000035	0.000047	0.000031	0.000081	0.000018
64x64 * 64x64	0.000136	0.000099	0.000148	0.000232	0.000225	0.000132	0.000143	0.000032
128x128 * 128x128	0.000994	0.000699	0.000921	0.001003	0.003906	0.000888	0.001084	0.000161
256x256 * 256x256	0.008332	0.005827	0.005234	0.006541	0.041294	0.006966	0.011690	0.001113
512x512 * 512x512	0.088418	0.053916	0.042809	0.040923	0.471251	0.088474	0.055598	0.008529
1024x1024 * 1024x1024	2.144968	0.804566	1.621012	1.398856	3.646465	0.989803	0.611013	0.062051
2048x2048 * 2048x2048	18.696455	15.517919	-	30.637685	29.238232	-	-	0.583124

Figure 8.10: Minimum execution times in secs (data)

Rule of the thumb, the more GPU cores the better the performance. In this example (memory intensive), as you can see, CPU devices reach the performance of the GPU devices.

8.4 Compare OpenCL on CPU/GPU with our SimplePL on FPGA

With all these date we now have, we can compare the applications tasks of running OpenCL kernels on CPU/GPU architectures with running SimplePL HLS kernels on your FPGA. Of course, we are comparing the OpenCL devices we benchmarked with our "Zedboard" development board, hence similar tasks on other devices may produce different results. We will not compare executions times, as in this work our kernels were not optimized nor the same.

8.4.1 Build time vs Program time

We compared on a previous section, the program flow of an OpenCL application with a SimplePL application. On this comparison we matched the "build process" with the "PL program process". Therefor, we should compare those times to find out if there is a significant difference between them.

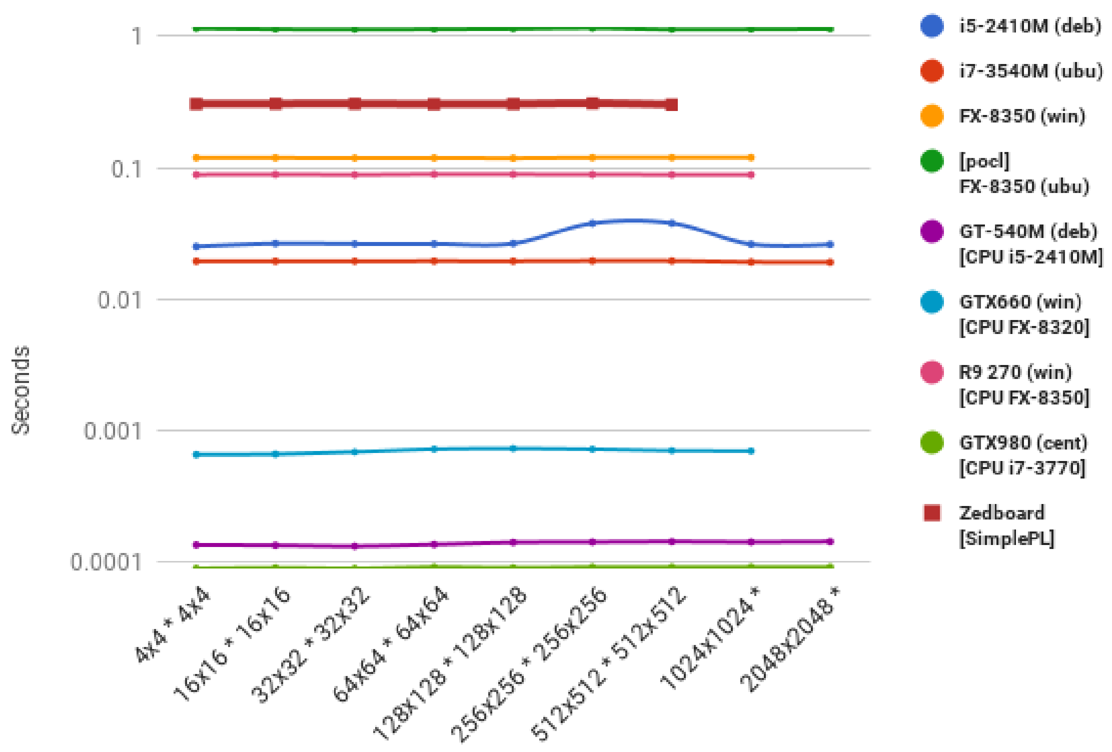


Figure 8.11: Build time and Program time (lower is better)

The fact that our kernels are a bit different will not affect our results, as the PL program time will not change if we change the bitfile, but the OpenCL build time may increase with bigger or higher complexity kernels.

8.4.2 Data transfer

With our results, there are not many things to say about data transfers. Our examples were not optimized to use less memory on both OpenCL and SimplePL applications, hence we may abused it. But, we have to point out that Vivado HLS does not let us choose a memory interface for our OpenCL kernels. In future releases, it may auto select an optimal memory interface or let us configure it.

We should optimize our kernels and use the memory hierarchy mechanics (like work-group memory) in our advantage. Similar optimizations should be applied on our OpenCL HLS kernels, although they have limited memory capacities. As always, we want to be able to feed data to our FPGA systems faster.

9 Conclusions

Our SimplePL system works as intended. We met our goal to provide a simple and familiar OpenCL-like environment that can handle our Vivado HLS OpenCL kernels and decrease even more the development time of Linux applications that use such kernels. We created SimplePL to help us handle all those problems that a Linux application has to face to execute OpenCL on the FPGA, and we experienced first how good it works.

9.1 Simple Usage

SimplePL hides all the complexity of the software to hardware communication, that needs special knowledge from the software developers. In this way, it makes the HLS OpenCL kernels usage for Linux embedded applications easy. Applications can simply call the HLS IP cores on the hardware with just some OpenCL-like function calls.

Hardware developers can create the bitfiles and the description files of many kernels of interest. Then these bitfiles can be distributed and used from applications that wants to execute any of these kernels. The developers of the application does not need to have any knowledge over the FPGA, just to know the kernels inside the bitfiles they are using.

OpenCL's program flow is simple and easy, thus as we based the SimplePL's API on the OpenCL's API, our system also features a simple and easy program flow. We also provide additional API functions for the developers to use, to handle their hardware on a lower level if they want to do so. While we developed all the demos on our systems, we show first hand how easy it was.

9.2 Familiar environment

SimplePL controls IP cores created with Vivado HLS by writing OpenCL kernel codes. This means that the developers of these IP cores knows OpenCL and they are familiar with its API too. If there was an OpenCL implementation for Vivado HLS kernels on their device, they probably would use it.

This is why SimplePL implements an API relative to the OpenCL's one, thus creating a familiar environment for the developers that already knows OpenCL. Such developers, can easily learn SimplePL just by reading some example codes and see the main functions of our API.

By developing with SimplePL, the development process of the application feels like the developing for a device with OpenCL support, just like developing for a GPU device or on SDAccel. The earlier lack of this OpenCL-like environment on embedded FPGA boards that SDAccel does not support, may have been a reason for many developers, not to develop applications for HLS OpenCL kernels on these boards. We think that now, this will change.

9.3 Decrease development time

Due to the fact that SimplePL solve all those problems that an application that wants to interact with HLS OpenCL kernels has to face, we observed that the development time of such applications can be dramatically decreased. Furthermore, as SimplePL handles all those time consuming tasks, like memory management and software to hardware bindings, it provides spare time to the developers to focus on more significant parts of the application. For example, programmers can invest these spare hours to optimize their OpenCL kernel's code for their FPGA, as this will increase the overall performance of their application.

Without SimplePL, developers had to find solutions to overcome the obstacles of managing their bitfiles and communications with the FPGA. Then they had to implement these solutions on their application code. On each application, that wanted to accelerate its performance with the use of HLS OpenCL kernels, developers had to re-implement such solutions from the beginning and many time with maybe only supporting one bitfile.

But with the use of SimplePL, developers just develop those parts of the application that matters, the HLS OpenCL kernels and the main application code, just like on an OpenCL supported device. SimplePL also handles all the workload scheduling on the compute units, so the developers does not have to control the FPGA on a compute unit level and schedule manual the work-groups that needs to be executed.

9.4 Benchmarks conclusions

From our benchmarks we can see that OpenCL runs quite well on CPU devices too, as long as the memory allows it. This is not a surprise, since OpenCL defines a parallel programming model, and thus it can take advantage of the many cores that our nowadays CPUs has to execute our kernels in an efficient way. Modern CPUs are quite advanced and since they support OpenCL, we should not underestimate them when executing OpenCL applications.

As you can see, it is obvious that the OS does affect the performance of the OpenCL applications. This may be caused by the mechanics the OS manages the memory. In fact, Linux seams to perform better, but drivers availability varies depending on the flavor. Of course, different drivers, yield different performances and as these drivers are part of the OS it is logical that they are affected by it too.

We also want to mention that the "pocl"'s performance was quite close to the vendor's OpenCL drivers in our benchmarks. The only visible drawback was the kernel build time which adds an overhead on your first build, but can be hidden if you only build your OpenCL code ones and re-use it for each execution. Thus, pocl seems to be a good solution for supporting OpenCL on an unsupported CPU device as it produce quite good results.

For our FPGA download time with the OpenCL build time comparison, we think the two timings are relative close. The FPGA has a small overhead in comparison with an OpenCL build of a small/simple kernel. We have not tested it, but, partial reconfiguration may decrease those timings. Also, since bigger FPGAs features bigger bitstreams, this time may increase on such FPGAs. Hopefully, the download process may be faster on the latest FPGA boards, with better hardware of with better drivers.

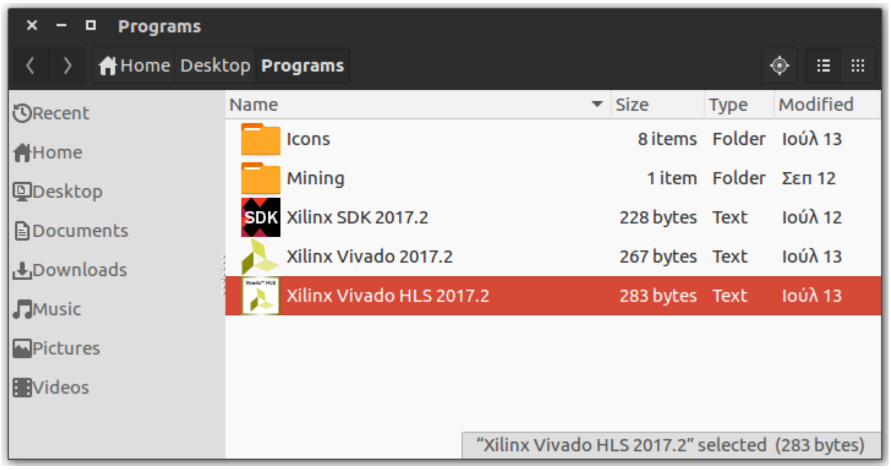
A Vivado Guides

A.1 Vivado HLS Guide

This section will walk you through on how to create an IP core from an OpenCL Kernel on Vivado HLS. We are going to use the Vivado Suite 2017.2 but a similar process can be followed for other versions of the Vivado suite. For this demonstration, we will implement a simple vector addition kernel for the Zynq Zeadboard.

A.1.1 Create a Vivado HLS project

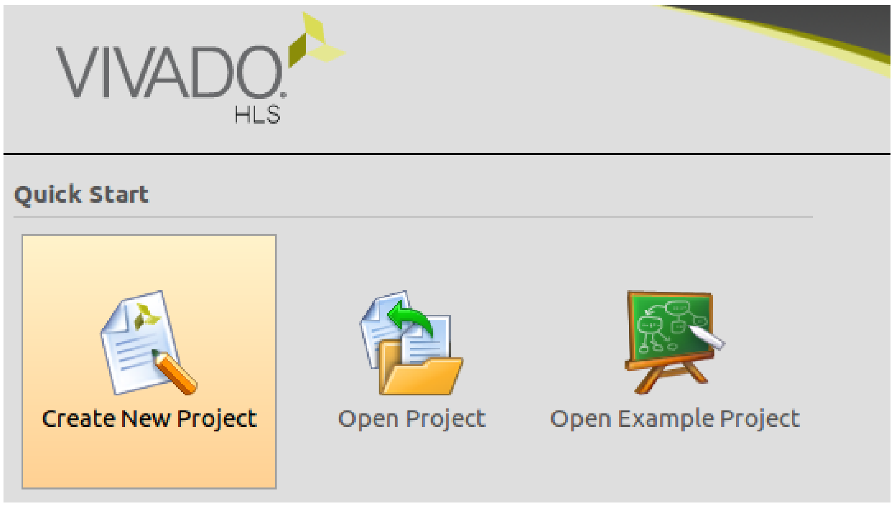
The first thing to do, is to open Vivado HLS.



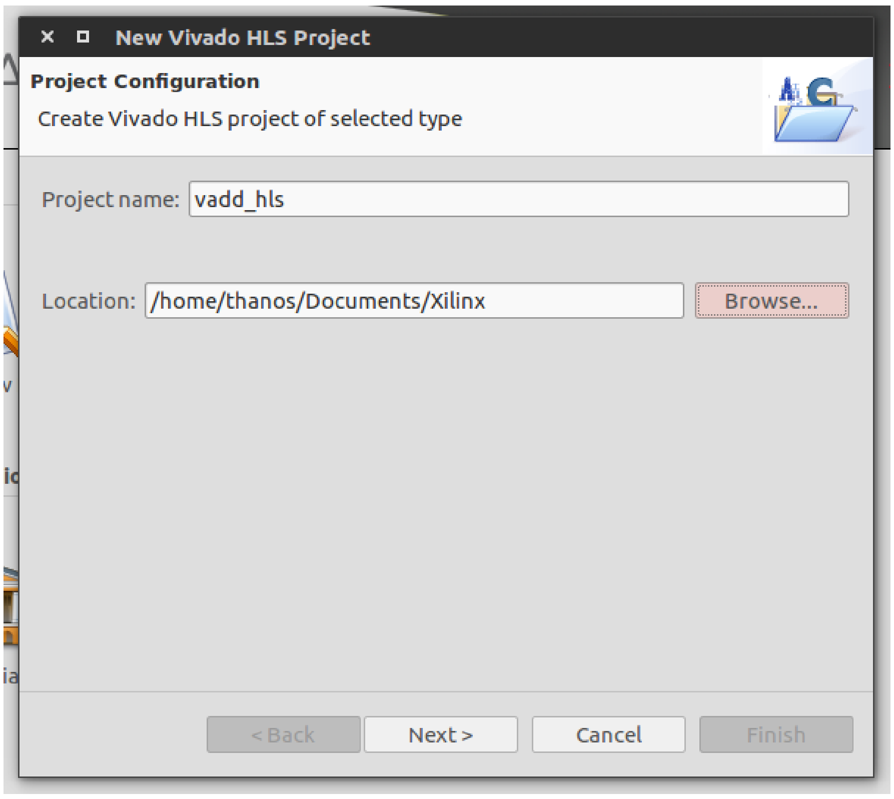
Name	Size	Type	Modified
Icons	8 items	Folder	Ιούλ 13
Mining	1 item	Folder	Σεπ 12
Xilinx SDK 2017.2	228 bytes	Text	Ιούλ 12
Xilinx Vivado 2017.2	267 bytes	Text	Ιούλ 13
Xilinx Vivado HLS 2017.2	283 bytes	Text	Ιούλ 13

"Xilinx Vivado HLS 2017.2" selected (283 bytes)

Select "Create New Project" under the "Quick Start" section.



Set “vadd_hls“ as a project name (or that ever you like) and select a location for the project to be saved. Then click the “Next >“ button.


New Vivado HLS Project

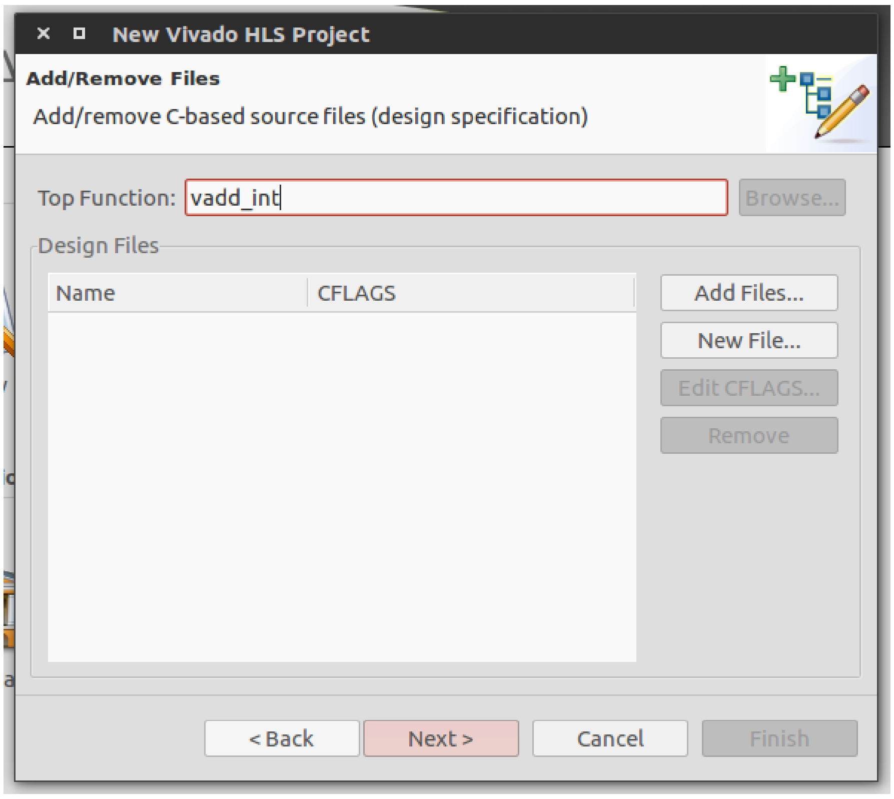
Project Configuration

Create Vivado HLS project of selected type

Project name: vadd_hls

Location: /home/thanos/Documents/Xilinx

Set “vadd_kernel“ as the Top Function (the name of your kernel). Then click the “Next >“ button.


New Vivado HLS Project

Add/Remove Files

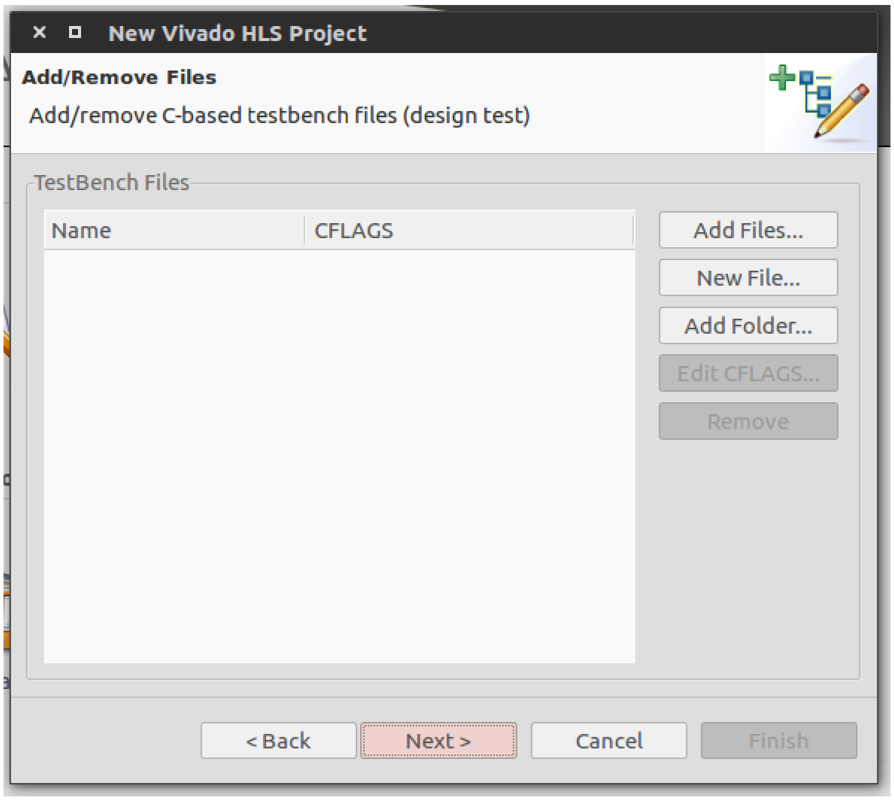
Add/remove C-based source files (design specification)

Top Function: vadd_int

Design Files

Name	CFLAGS

Don't add any files yet (we will add them later), so click "Next >".


New Vivado HLS Project

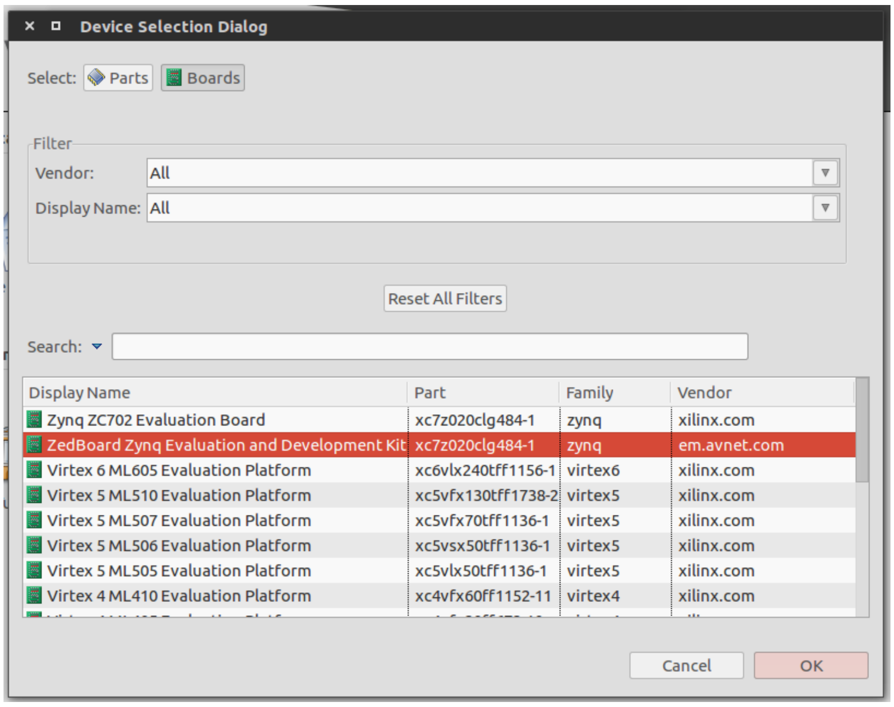
Add/Remove Files

Add/remove C-based testbench files (design test)

TestBench Files

Name	CFLAGS

Now we need to select our target device/board. To target the Zeadboard board, click the "..." button under the Part Selection. Click the "Boards" button on the top, then select "ZedBoard Zynq ..." (or your targeting board) on the list. Then click "OK".



Display Name	Part	Family	Vendor
Zynq ZC702 Evaluation Board	xc7z020clg484-1	zynq	xilinx.com
ZedBoard Zynq Evaluation and Development Kit	xc7z020clg484-1	zynq	em.avnet.com
Virtex 6 ML605 Evaluation Platform	xc6vlx240tff1156-1	virtex6	xilinx.com
Virtex 5 ML510 Evaluation Platform	xc5vfx130tff1738-2	virtex5	xilinx.com
Virtex 5 ML507 Evaluation Platform	xc5vfx70tff1136-1	virtex5	xilinx.com
Virtex 5 ML506 Evaluation Platform	xc5vsx50tff1136-1	virtex5	xilinx.com
Virtex 5 ML505 Evaluation Platform	xc5vlx50tff1136-1	virtex5	xilinx.com
Virtex 4 ML410 Evaluation Platform	xc4vfx60ff1152-11	virtex4	xilinx.com

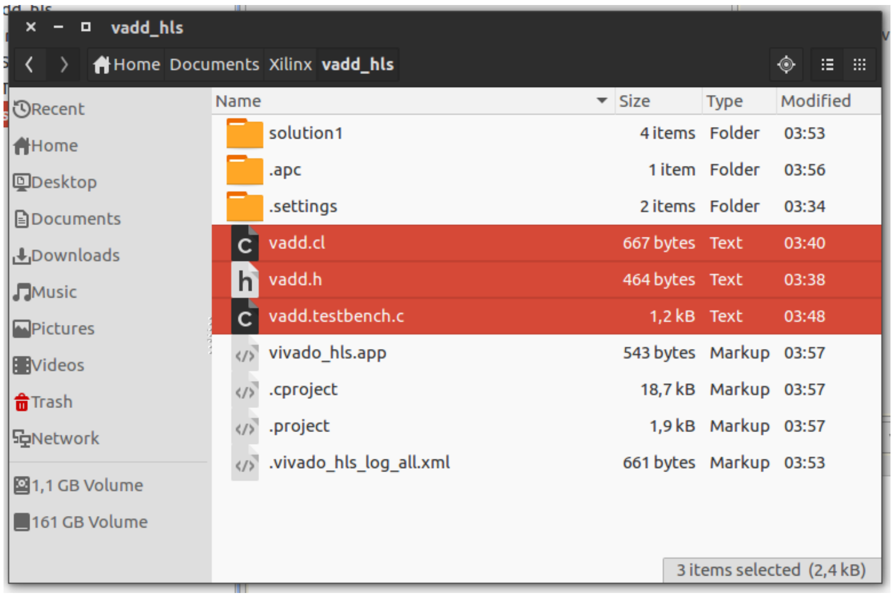
Click "Finish" and your project is ready... but with no source files.

In our Vivado HLS project, we have 3 files.

- Our OpenCL file (vadd.cl)
  - It has our OpenCL kernel code.
  - This is the code that will run on our programmable logic.
  - Should be "*.cl".
- A testbench file (vadd.testbench.c)
  - It has a code to test our OpenCL kernel before and after synthesis.
  - This is the code will only run on Vivado HLS to validate the correct functionality of our OpenCL kernel.
  - Should be "*.c".
- A header file to define shared constants (vadd.h)
  - It is not mandatory.
  - This file only helps our organize our project and define the types that our kernel will use (so that we wan export one kernel for each type without changing the hole kernel code).

– Should be “*.h“.

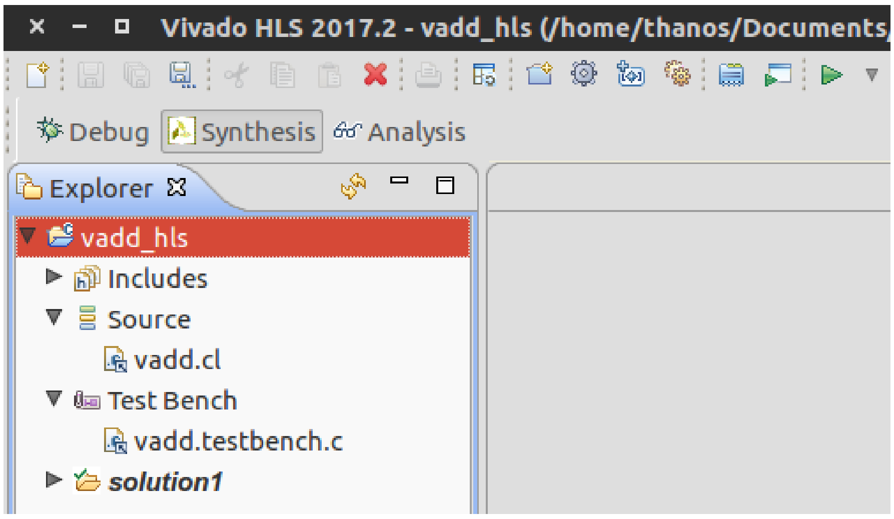
To add these file to the project, first, open the folder of the project and copy inside the 3 files we mentioned, the “vadd.h“, “vadd.cl“ and the “vadd.testbench.c“.



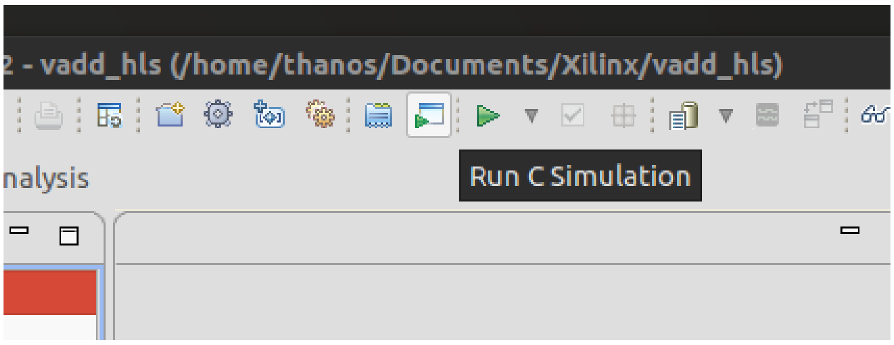
Name	Size	Type	Modified
solution1	4 items	Folder	03:53
.apc	1 item	Folder	03:56
.settings	2 items	Folder	03:34
vadd.cl	667 bytes	Text	03:40
vadd.h	464 bytes	Text	03:38
vadd.testbench.c	1,2 kB	Text	03:48
vivado_hls.app	543 bytes	Markup	03:57
.cproject	18,7 kB	Markup	03:57
.project	1,9 kB	Markup	03:57
.vivado_hls_log_all.xml	661 bytes	Markup	03:53

Now, on the Vivado HLS, add “vadd.cl“ under the “Source“ and add “vadd.testbench.c“ under the “Test Bench“ by right clicking the corresponding folder and clicking “Add file“. We don't need to add the header file on your project as it will be found automatically.

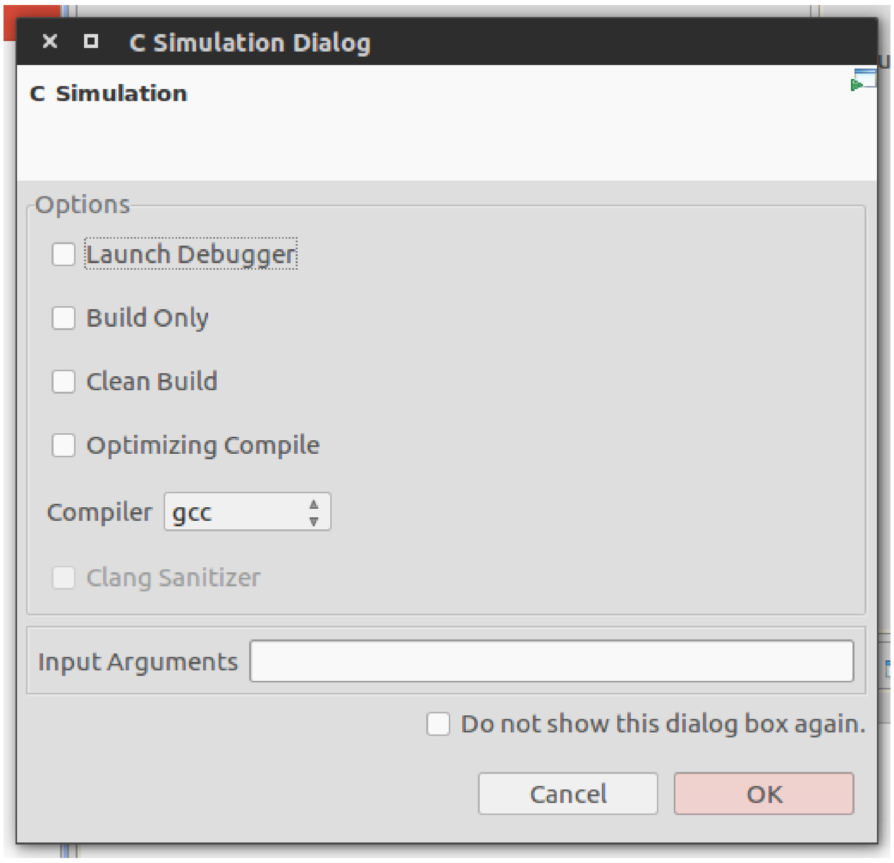
You should now have the following structure to your project:



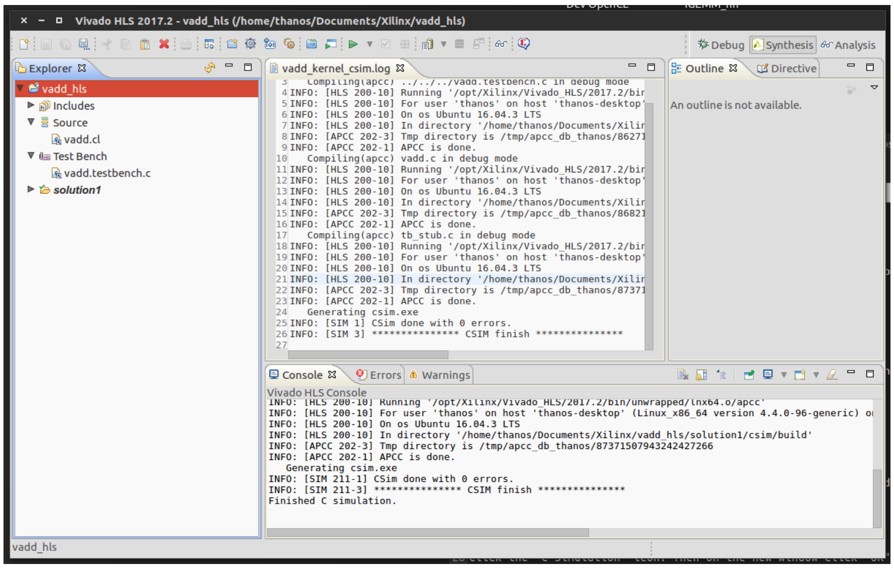
Our next step is to test our code. We want to know if it compiles correctly and if your testbench runs with no problem on our CPU. To do so, we click the “Run C Simulation“ button on the tool-bar.



A new window will appear with addition parameters for the C Simulation (debugger, optimizations) but for now we just click the “OK“ button to run the simulation.



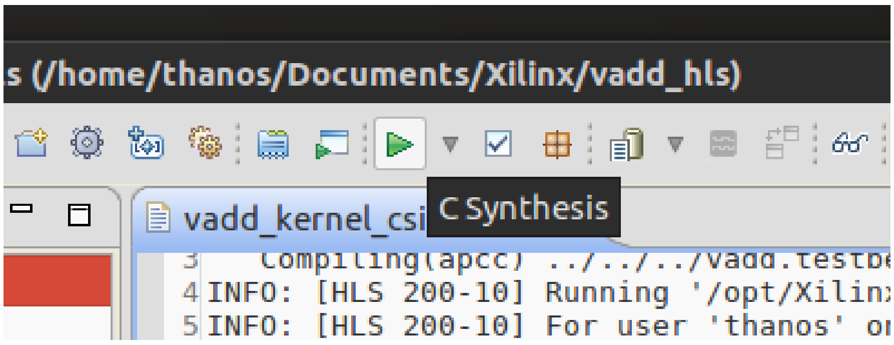
The simulation may take a few seconds and after that it should finish with no errors.



```
   Compiling(apcc) ../../../vadd.testbench.c in debug mode
INFO: [HLS 200-10] Running '/opt/Xilinx/Vivado_HLS/2017.2/bir
INFO: [HLS 200-10] For user 'thanos' on host 'thanos-desktop'
INFO: [HLS 200-10] On os Ubuntu 16.04.3 LTS
INFO: [HLS 200-10] In directory '/home/thanos/Documents/Xilir
INFO: [APCC 202-3] Tmp directory is /tmp/apcc_db_thanos/86271
INFO: [APCC 202-1] APCC is done.
   Compiling(apcc) vadd.c in debug mode
INFO: [HLS 200-10] Running '/opt/Xilinx/Vivado_HLS/2017.2/bir
INFO: [HLS 200-10] For user 'thanos' on host 'thanos-desktop'
INFO: [HLS 200-10] On os Ubuntu 16.04.3 LTS
INFO: [HLS 200-10] In directory '/home/thanos/Documents/Xilir
INFO: [APCC 202-3] Tmp directory is /tmp/apcc_db_thanos/86821
INFO: [APCC 202-1] APCC is done.
   Compiling(apcc) tb_stub.c in debug mode
INFO: [HLS 200-10] Running '/opt/Xilinx/Vivado_HLS/2017.2/bir
INFO: [HLS 200-10] For user 'thanos' on host 'thanos-desktop'
INFO: [HLS 200-10] On os Ubuntu 16.04.3 LTS
INFO: [HLS 200-10] In directory '/home/thanos/Documents/Xilir
INFO: [APCC 202-3] Tmp directory is /tmp/apcc_db_thanos/87371
INFO: [APCC 202-1] APCC is done.
   Generating csim.exe
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

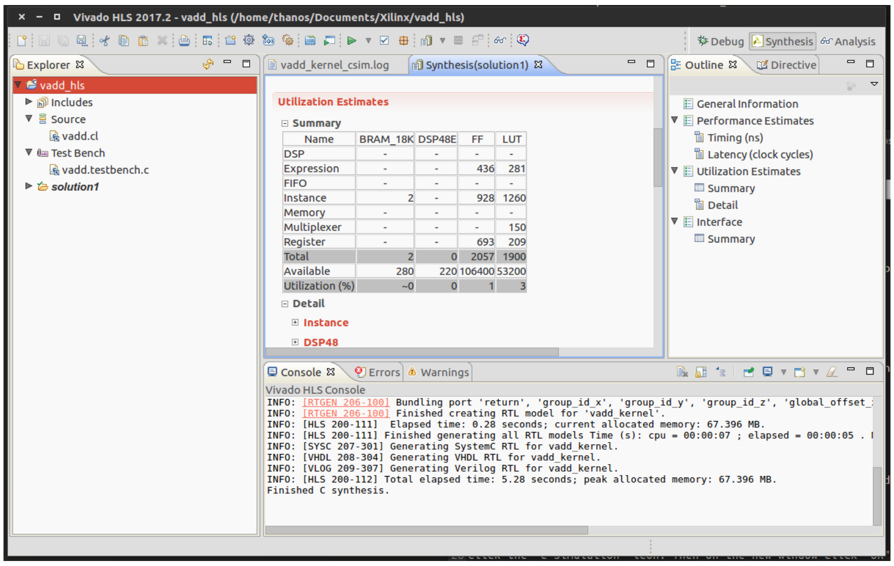
```
INFO: [HLS 200-10] Running '/opt/Xilinx/Vivado_HLS/2017.2/bin/unwrapped/lnx64.o/apcc'
INFO: [HLS 200-10] For user 'thanos' on host 'thanos-desktop' (Linux_x86_64 version 4.4.0-96-generic) o
INFO: [HLS 200-10] On os Ubuntu 16.04.3 LTS
INFO: [HLS 200-10] In directory '/home/thanos/Documents/Xilinx/vadd_hls/solution1/csim/build'
INFO: [APCC 202-3] Tmp directory is /tmp/apcc_db_thanos/87371507943242427266
INFO: [APCC 202-1] APCC is done.
   Generating csim.exe
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

Now that we know that our C code is working like it should be, we can run a "C Synthesis" to convert our C/OpenCL to a Verilog/VHLD IP core. To execute the synthesis we click the "C Synthesis" button on the tool-bar.



```
   Compiling(apcc) ../../../vadd.testbe
INFO: [HLS 200-10] Running '/opt/Xilin
INFO: [HLS 200-10] For user 'thanos' o
```

This may take a while, depending on your system but since our example code is small and simple, it should be done in a few seconds. Upon completion, you will be welcomed by the synthesis results in a new tab.



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	436	281
FIFO	-	-	-	-
Instance	2	-	928	1260
Memory	-	-	-	-
Multiplexer	-	-	-	150
Register	-	-	693	209
Total	2	0	2057	1900
Available	280	220	106400	53200
Utilization (%)	~0	0	1	3

Detail

Instance

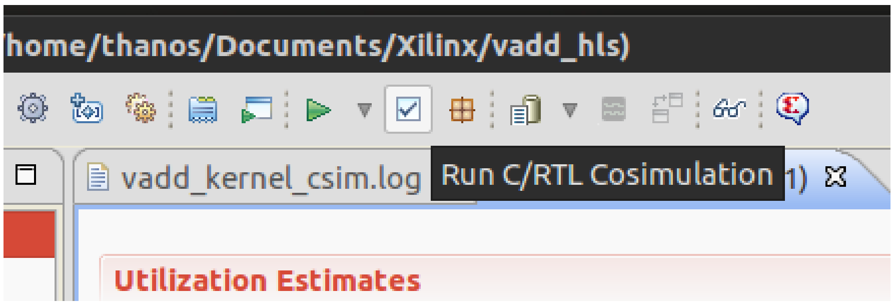
DSP48

Vivado HLS Console

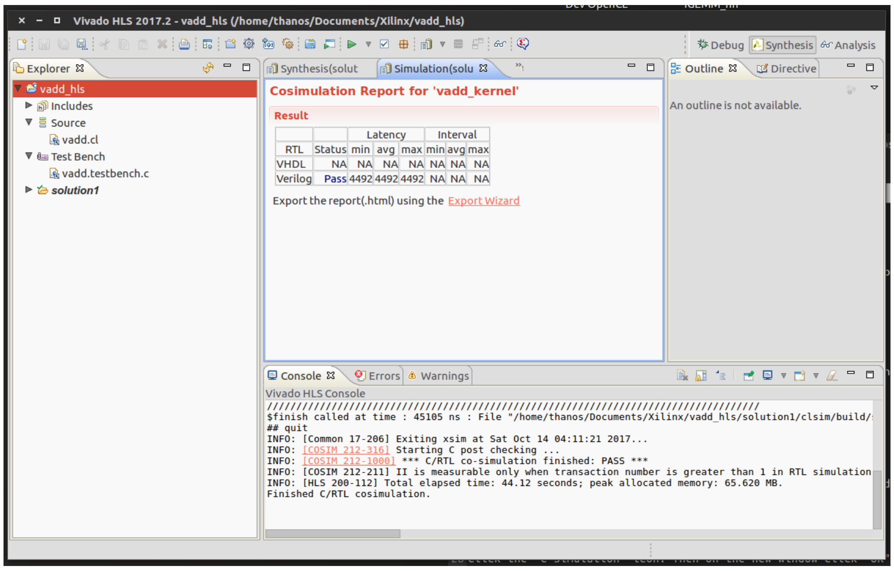
```
INFO: [RTGEN 206-100] Bundling port 'return', 'group_id_x', 'group_id_y', 'group_id_z', 'global_offset_
INFO: [RTGEN 206-100] Finished creating RTL model for 'vadd_kernel'.
INFO: [HLS 200-111]  Elapsed time: 0.28 seconds; current allocated memory: 67.396 MB.
INFO: [HLS 200-111] Finished generating all RTL models Time (s): cpu = 00:00:07 ; elapsed = 00:00:05 .
INFO: [SYSC 207-301] Generating SystemC RTL for vadd_kernel.
INFO: [VHDL 208-304] Generating VHDL RTL for vadd_kernel.
INFO: [VLOG 209-307] Generating Verilog RTL for vadd_kernel.
INFO: [HLS 200-112] Total elapsed time: 5.28 seconds; peak allocated memory: 67.396 MB.
Finished C synthesis.
```

An important table to check inside the synthesis results, is the "Utilization Estimates" table where you can see the portion of the hardware what 1 compute unit of your HLS Kernel will use (1 IP core). You can use the data from this table to estimate how many compute units you can include in your bitfile or if you need to scale your work-groups. Very low usage indicates that you may need to increase the work of each process element of your work-group to increase the performance.

Once again we need to validate the functionality of the generated code. This is done by co-simulating C and RTL. On the tool-bar we need to click the "Run C/RTL Cosimulation" icon. Then a new window will appear, we click "OK". This will execute the testbench and it must produce the same results with the last time (no errors).



If the C/RTL Cosimulation complete successful, we will get a latency estimation of our implementation.



Cosimulation Report for 'vadd_kernel'

Result

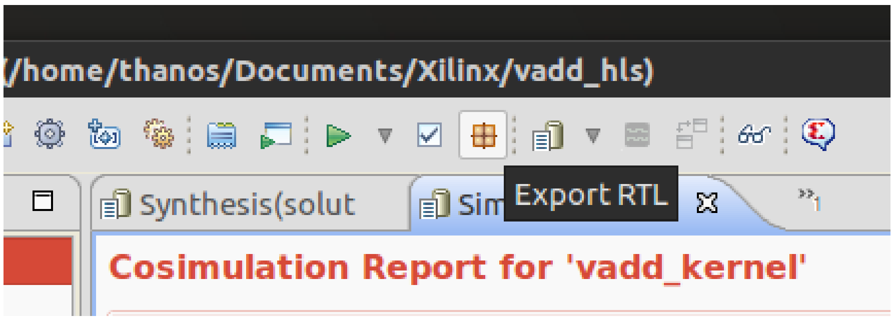
		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	4492	4492	4492	NA	NA	NA

Export the report(.html) using the Export Wizard

Vivado HLS Console

```
////////////////////////////////////////////////////////////////////////////////////
$finish called at time : 45105 ns : File "/home/thanos/Documents/Xilinx/vadd_hls/solution1/clsim/build/
## quit
INFO: [Common 17-206] Exiting xsim at Sat Oct 14 04:11:21 2017...
INFO: [COSIM 212-316] Starting C post checking ...
INFO: [COSIM 212-1000] *** C/RTL co-simulation finished: PASS ***
INFO: [COSIM 212-211] II is measurable only when transaction number is greater than 1 in RTL simulation
INFO: [HLS 200-112] Total elapsed time: 44.12 seconds; peak allocated memory: 65.620 MB.
Finished C/RTL cosimulation.
```

Last but not least, we need to export our IP core, for the Vivado. We click the "Export RTL" icon on the tool-bar and then on the new window we click "Configuration..." to customize our IP info. We fill in the info and then click "OK" on both windows.


(/home/thanos/Documents/Xilinx/vadd_hls)
Synthesis(solut
Sim
Export RTL
Cosimulation Report for 'vadd_kernel'

After the export, an IP core was exported on:
"`<project>/solution<number>/impl/ip/`
`<ventor>_<library>_<kernel-name>_<version>.zip`"

As the IP core is ready, we can start populating the Bitfile Description File with information. Offsets and arguments information can be found inside the xml file:
"`<project>/solution1/impl/ip/<kernel-name>_info.xml`"

A.1.2 Example Vivado HLS codes

Here you can read the example codes that were used on this tutorial.

```
/*
 * Copyright Grammatopoulos Athanasios Vasileios
 * agrammatopoulos@isc.tuc.gr
 * Technical University of Crete
 *
 * Example OpenCL HLS Kernel
 */

#ifndef _KERNEL_VADD_H_
#define _KERNEL_VADD_H_

// Type of matrix data
#define vadd_type int
#define vadd_cl_type int
#define vadd_type_rand (rand() % 1000)

// Required Work Group Size
#define VADD_WORKGROUP_SIZE_X 256
#define VADD_WORKGROUP_SIZE_Y 1
#define VADD_WORKGROUP_SIZE_Z 1

#endif // _KERNEL_VADD_H_
```

Listing A.1: The vadd.h file

```
/*
 * Copyright 2017 Grammatopoulos Athanasios Vasileios
 * agrammatopoulos@isc.tuc.gr
 * Technical University of Crete
 *
 * Example OpenCL HLS Kernel
 */

// Xilinx OpenCL Kernel Library
#include <clc.h>
// Kernel's Static Parameters
#include "vadd.h"

// Set required work group size
__attribute__ ((reqd_work_group_size(VADD_WORKGROUP_SIZE_X,
    VADD_WORKGROUP_SIZE_Y, VADD_WORKGROUP_SIZE_Z)))

__kernel void vadd_kernel(
  __global vadd_cl_type *a,
  __global vadd_cl_type *b,
  __global vadd_cl_type *c,
  uint n
) {
  __attribute__((xcl_pipeline_workitems)) {
    // Get Global id
    const int id = get_global_id(0);
    if (id >= n) return;

    c[id] = a[id] + b[id];
  }
}
```

Listing A.2: The vadd.cl file

```
/*
 * Copyright 2017 Grammatopoulos Athanasios Vasileios
 * agrammatopoulos@isc.tuc.gr
 * Technical University of Crete
 *
 * Example OpenCL HLS Kernel
 */

// C Libraries
#include <stdio.h>
#include <stdlib.h>

// Kernel's Static Parameters
#include "vadd.h"

// Testbench code
int main(int argc, char** argv) {


  // Number of elements
  unsigned int n = VADD_WORKGROUP_SIZE_X;

  // Host matrices
  vadd_type *h_a;
  vadd_type *h_b;
  vadd_type *h_c;
  vadd_type *h_r;

  // Size, in bytes, of each vector
  size_t bytes = n * sizeof(vadd_type);

  // Allocate memory for each vector on host
  h_a = (vadd_type*) malloc(bytes);
  h_b = (vadd_type*) malloc(bytes);
  h_c = (vadd_type*) malloc(bytes);
  h_r = (vadd_type*) malloc(bytes);

  // Init rand
  srand(11888);

  int i;
  // Initialize vectors
  for (i = 0; i < n; i++) {
    h_a[i] = vadd_type_rand;
    h_b[i] = vadd_type_rand;
    h_c[i] = 0;
  }

  // Calculate results
  for (i = 0; i < n; i++) {
    h_r[i] = h_a[i] + h_b[i];
  }

  // Execute the OpenCL kernel function
  hls_run_kernel("vadd_kernel", h_a, 256, h_b, 256, h_c, 256, n, 1);

  // Init errors
  int errors = 0;
  for (i = 0; i < n; i++) {
    if(h_c[i] != h_r[i]){
      errors ++;
    }
  }

  // Return
  return (errors > 0) ? 1 : 0;
}
```

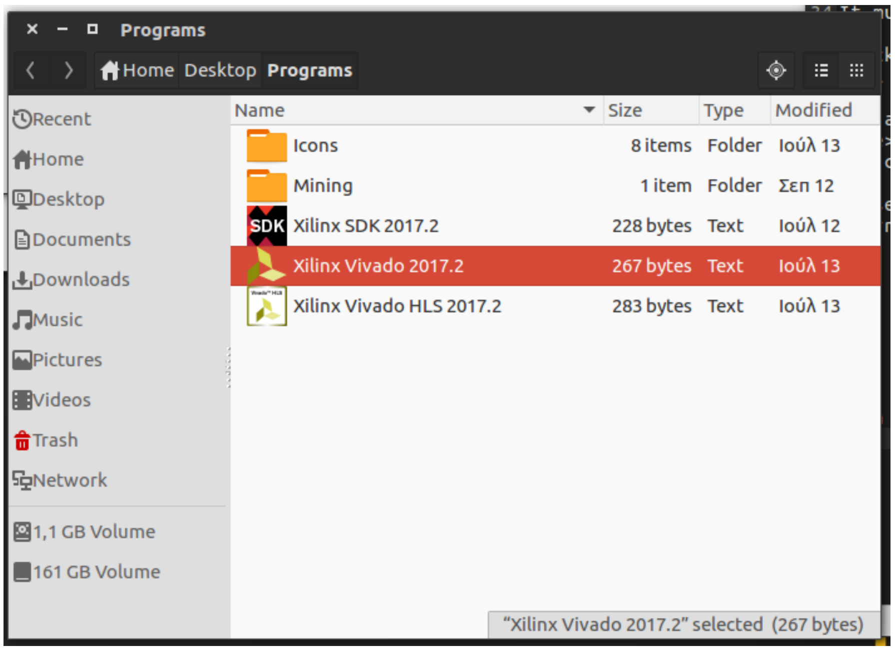
Listing A.3: The vadd.testbench.c file

A.2 Vivado OpenCL IP Core Guide

In this section will show a simple Vivado design with IP cores from an OpenCL Kernel that can produce a bitfile for use with the SimplePL. We are going to use the Vivado Suite 2017.2 but a similar process can be followed for other versions of the Vivado suite. For this design, we will use the vector addition IP cores we implemented on the previous section. Thus, we will target the Zynq Zeadboard.

A.2.1 Create a Vivado HLS project

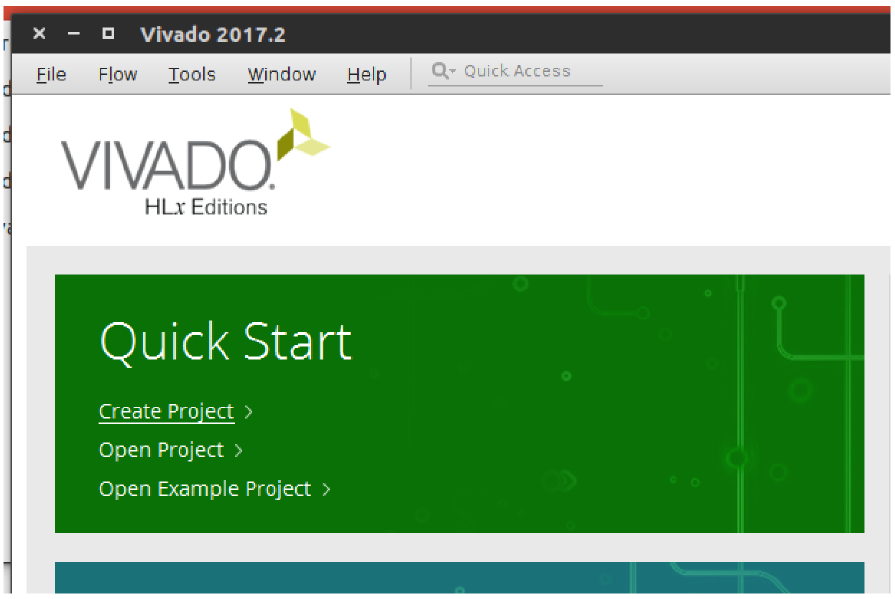
The first thing to do, is to open Vivado.



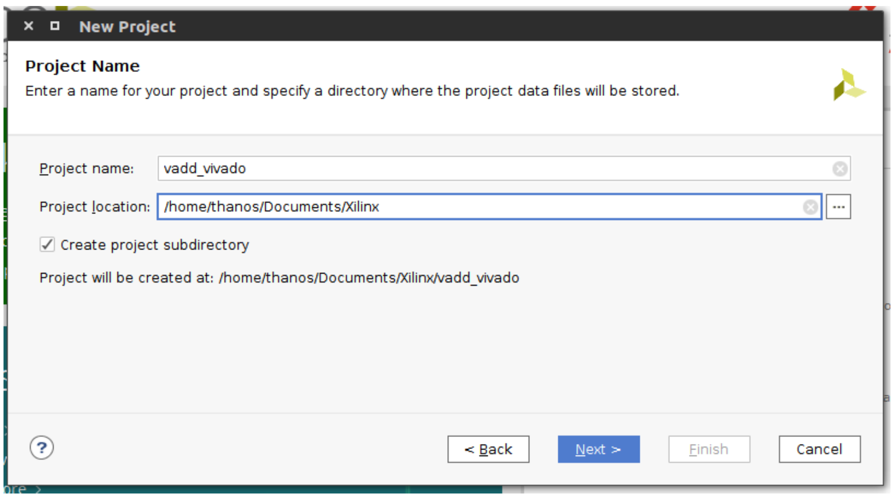
Name	Size	Type	Modified
Icons	8 items	Folder	Ιούλ 13
Mining	1 item	Folder	Σεπ 12
Xilinx SDK 2017.2	228 bytes	Text	Ιούλ 12
Xilinx Vivado 2017.2	267 bytes	Text	Ιούλ 13
Xilinx Vivado HLS 2017.2	283 bytes	Text	Ιούλ 13

"Xilinx Vivado 2017.2" selected (267 bytes)

Select "Create Project >" under the "Quick Start" section.



On the new window click “Next >“ and then you will be asked for your project information. Set “vadd_vivado“ as project name and a location for the project to be saved. Then click “Next >“.

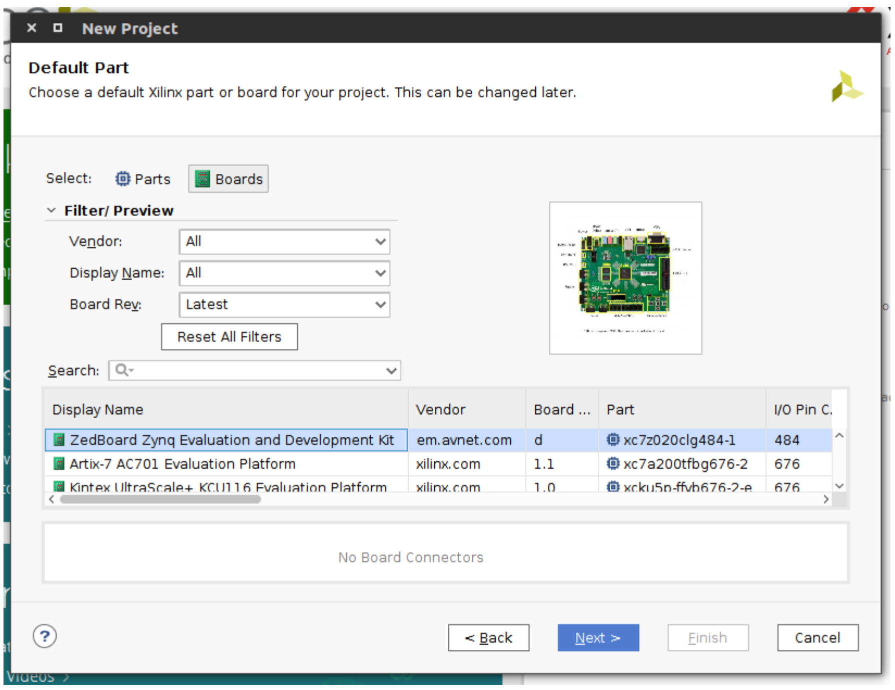


Project Name
Enter a name for your project and specify a directory where the project data files will be stored.

Project name: vadd_vivado
Project location: /home/thanos/Documents/Xilinx
✓ Create project subdirectory
Project will be created at: /home/thanos/Documents/Xilinx/vadd_vivado

On the new window click “Next >“, then “Next >“ and then again “Next >“ (we skipped these 3 steps because we don’t need them).

You should now be on the Board/Parts select step. Select the “Boards“ button on the top, then select “ZedBoard Zynq ...“ on the list. Then click “Next >“.


New Project

Default Part

Choose a default Xilinx part or board for your project. This can be changed later.

Select: Parts Boards

Filter/ Preview

Vendor: All

Display Name: All

Board Rev: Latest

Reset All Filters

Search:

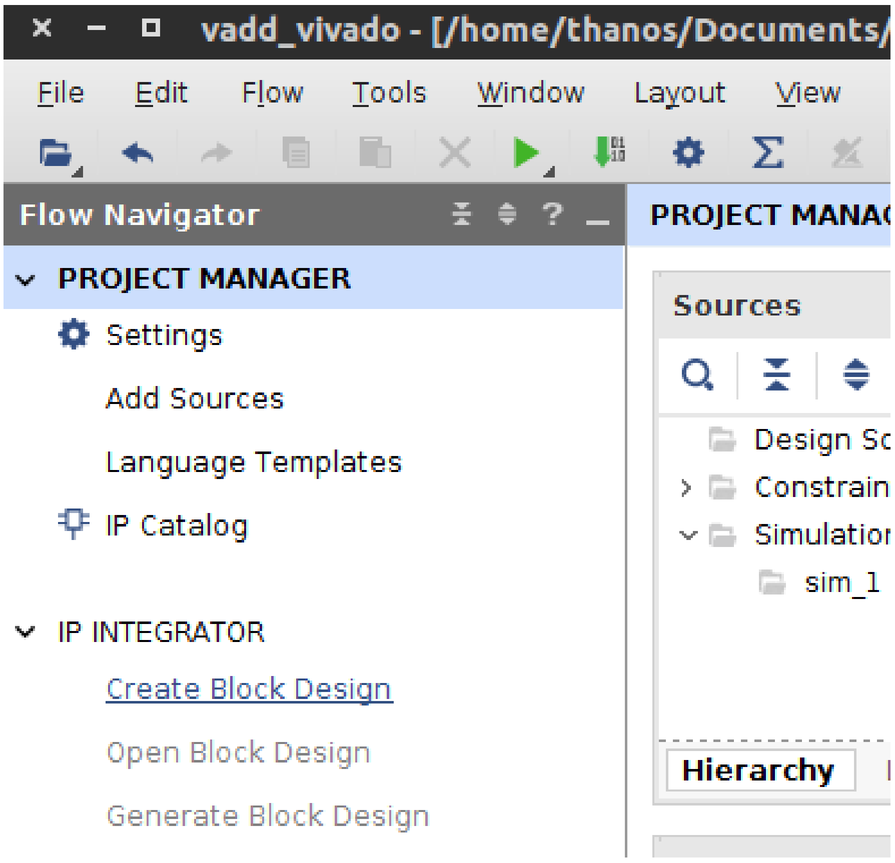
Display Name	Vendor	Board ...	Part	I/O Pin C.
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	d	xc7z020clg484-1	484
Artix-7 AC701 Evaluation Platform	xilinx.com	1.1	xc7a200tfbg676-2	676
Kintex UltraScale+ KCU116 Evaluation Platform	xilinx.com	1.0	xcku5p-ffvb676-2-e	676

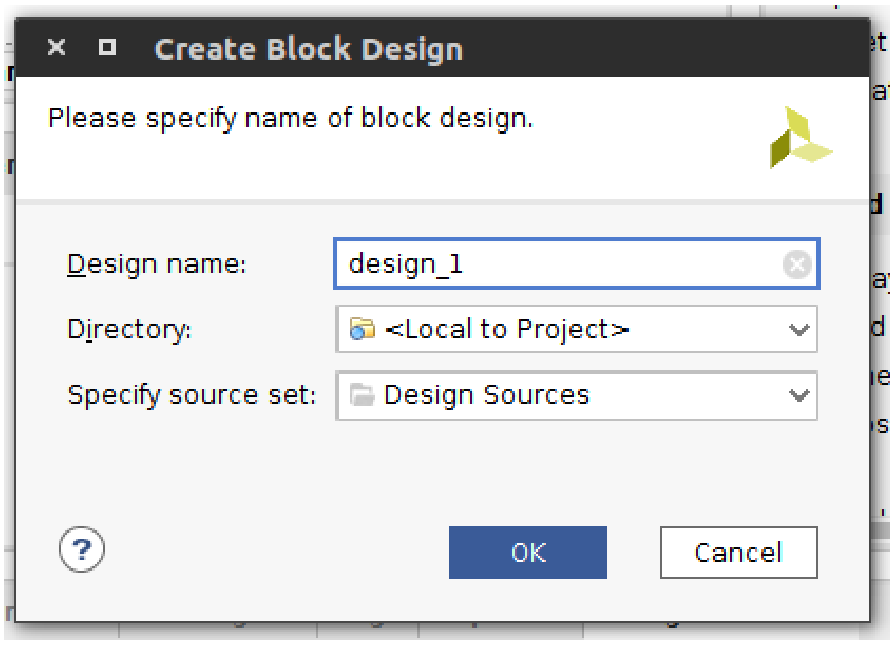
No Board Connectors

< Back   Next >   Finish   Cancel

Click “Finish“ and your project is ready... but it’s empty.

We now need to create a design, so, under the “IP INTEGRATOR“ click “Create Block Design“ and then on the new window click “OK“.




Create Block Design

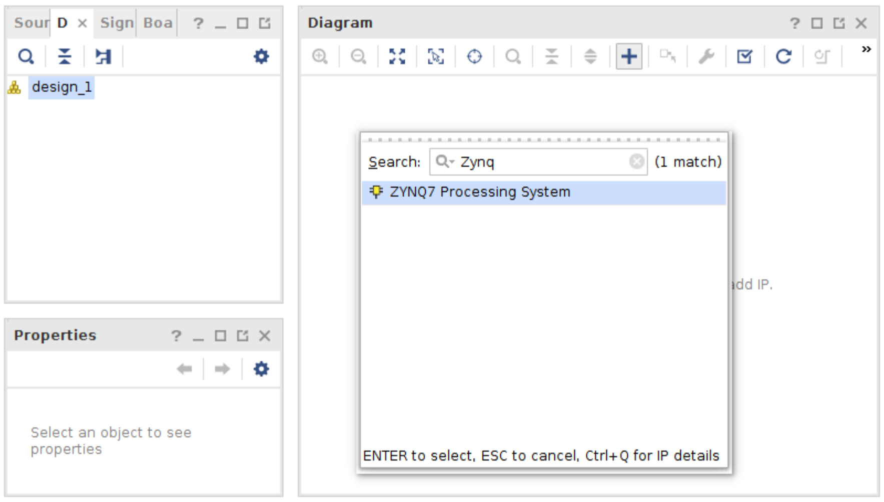
Please specify name of block design.

Design name: design_1

Directory: <Local to Project>

Specify source set: Design Sources

First we need to add and connect the processing system IP. Click the "add IP" button and search for "Zynq", find it and add it. Then click ""Run Block Automation" and on the new window click "OK". This will connect some IO ports of the IP.

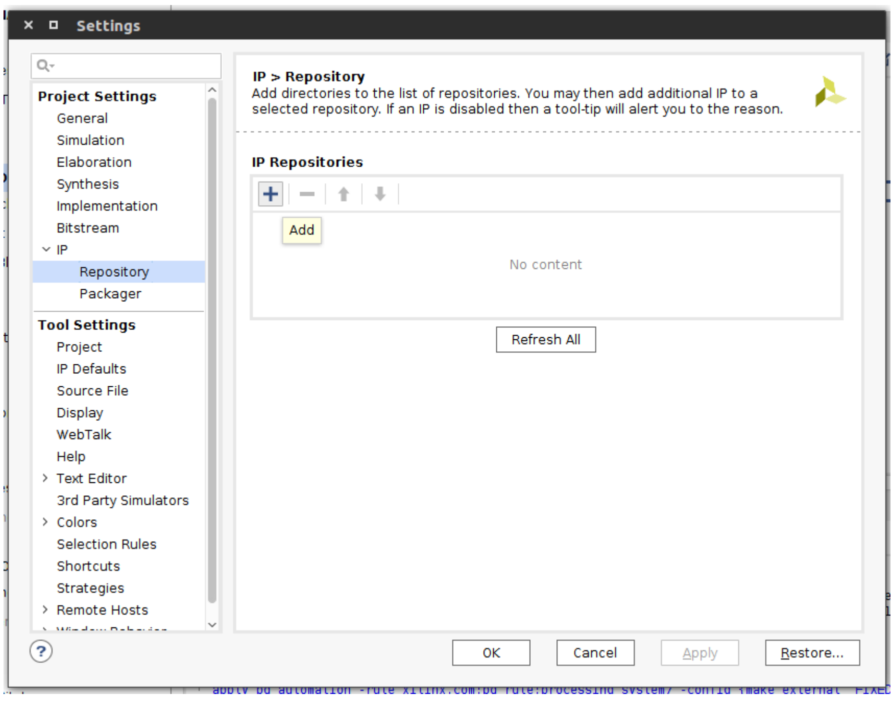


Search: Zynq (1 match)

ZYNQ7 Processing System

ENTER to select, ESC to cancel, Ctrl+Q for IP details

To add the vector addition IP core we implemented on the previous section, we need to let Vivado know where it is located. We can do this by adding an IP repository. Right click the empty space of your design and select "IP Settings ...". Then under the "IP" >"Repository" click the add button. Then select the HLS project "vadd_hls" that we created on the previous section.

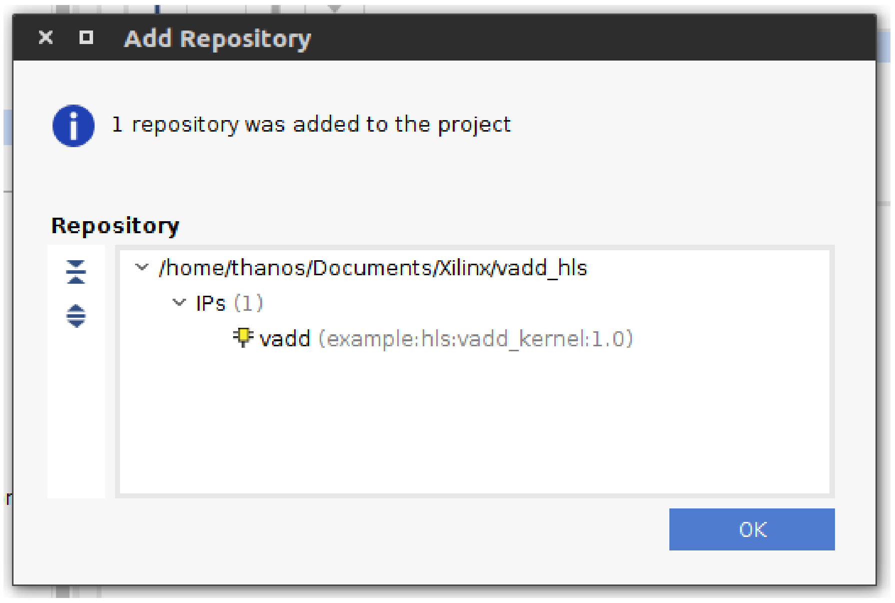


IP > Repository
Add directories to the list of repositories. You may then add additional IP to a selected repository. If an IP is disabled then a tool-tip will alert you to the reason.

IP Repositories

No content

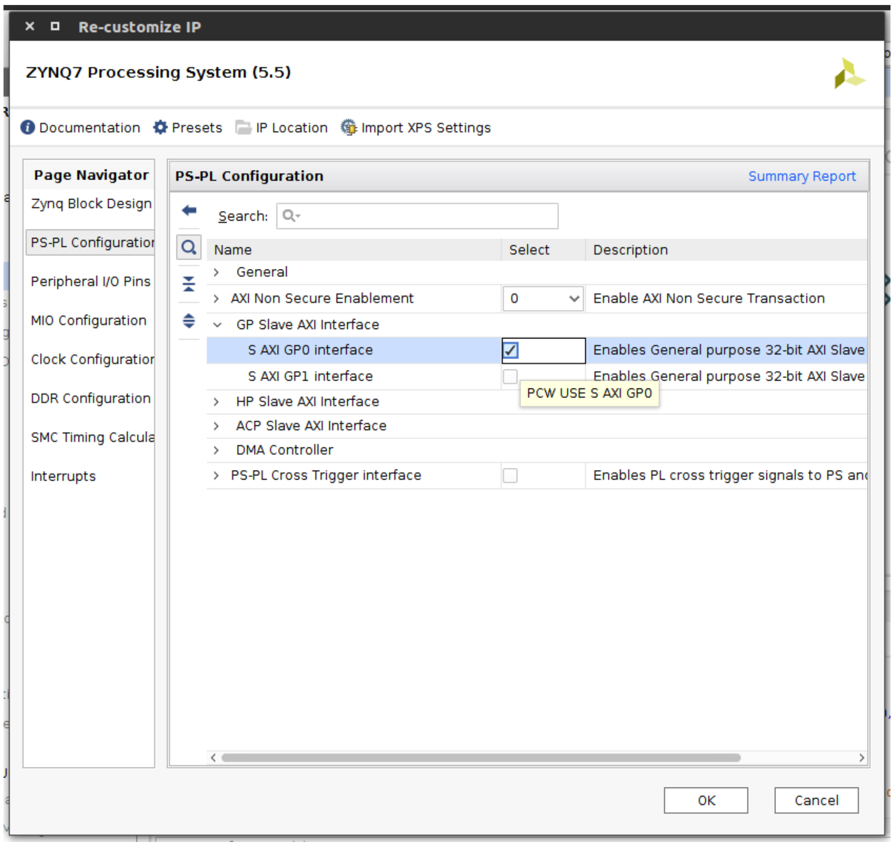
This should find the IP we created on HLS. Click “OK“ and close the settings.



1 repository was added to the project

Repository
/home/thanos/Documents/Xilinx/vadd_hls
IPs (1)
vadd (example:hls:vadd_kernel:1.0)

To continue, we need to enable some signals on the “Zynq processing system“. Double click the Zynq block to open its settings. Then, under the “PS-PL Configuration“ >“GP Slave AXI Interface“ check the “S AXI GP0 interface“ and click the “OK“ button to close the window.



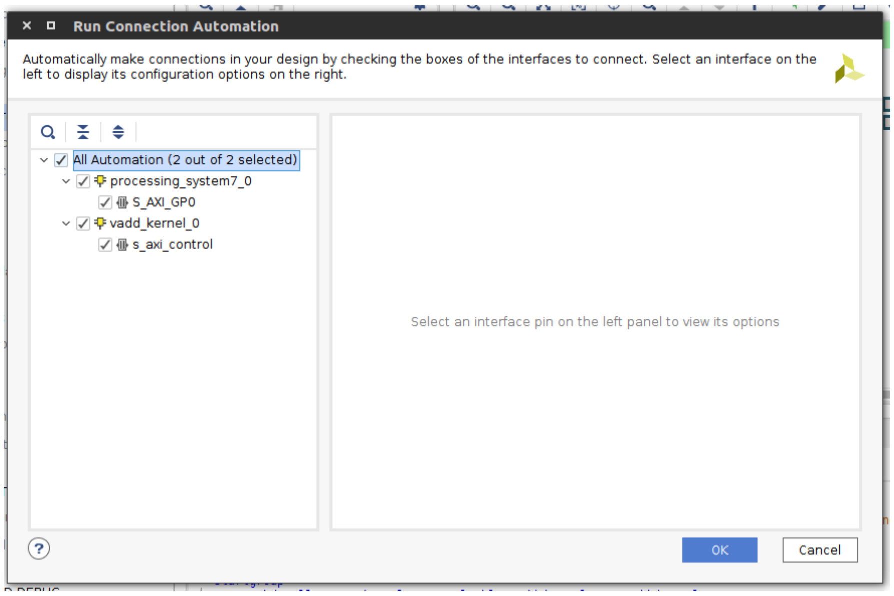
Re-customize IP

ZYNQ7 Processing System (5.5)

Documentation | Presets | IP Location | Import XPS Settings

Name	Select	Description
General		
AXI Non Secure Enablement	0	Enable AXI Non Secure Transaction
GP Slave AXI Interface		
S AXI GP0 interface	✓	Enables General purpose 32-bit AXI Slave
S AXI GP1 interface	☐	Enables General purpose 32-bit AXI Slave
HP Slave AXI Interface		
ACP Slave AXI Interface		
DMA Controller		
PS-PL Cross Trigger interface	☐	Enables PL cross trigger signals to PS an

We are now ready to add our compute units. Click the "add IP" button and search for "vadd", find it and add it. Then click the "Run Block Automation", select all check boxes and click "OK".



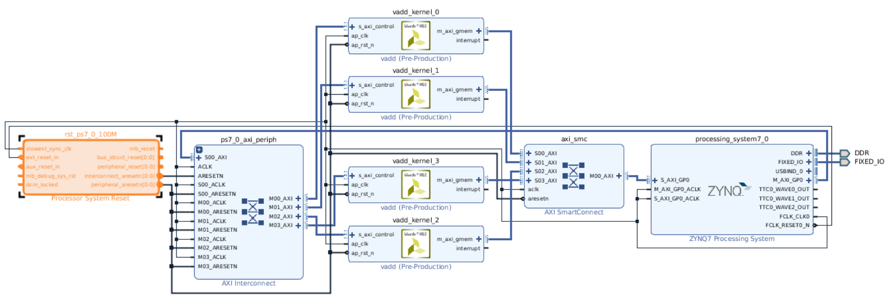
Run Connection Automation

Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

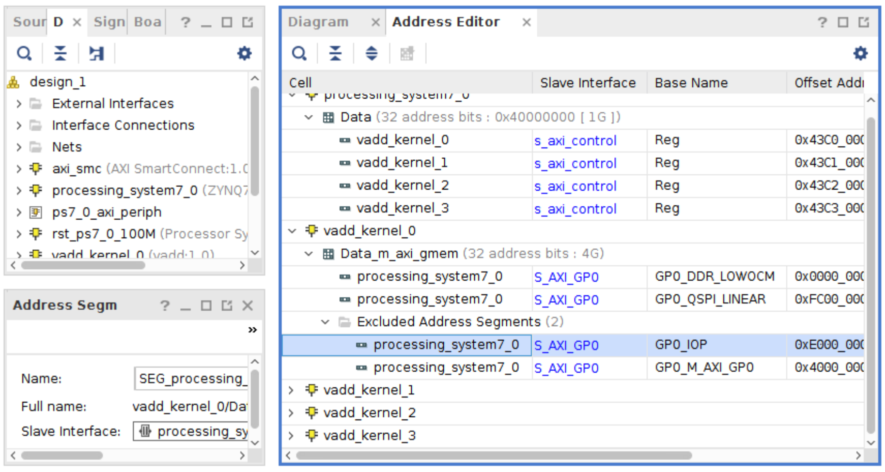
- ✓ All Automation (2 out of 2 selected)
  - ✓ processing_system7_0
    - ✓ S_AXI_GP0
  - ✓ vadd_kernel_0
    - ✓ s_axi_control

Select an interface pin on the left panel to view its options

You can repeat this process to to add more compute units. Let's add 4 compute units for now.



Now that we added our compute units, we need to fix their addresses. Click on the “Address Editor“ tab and include all the excluded segments of every compute unit.

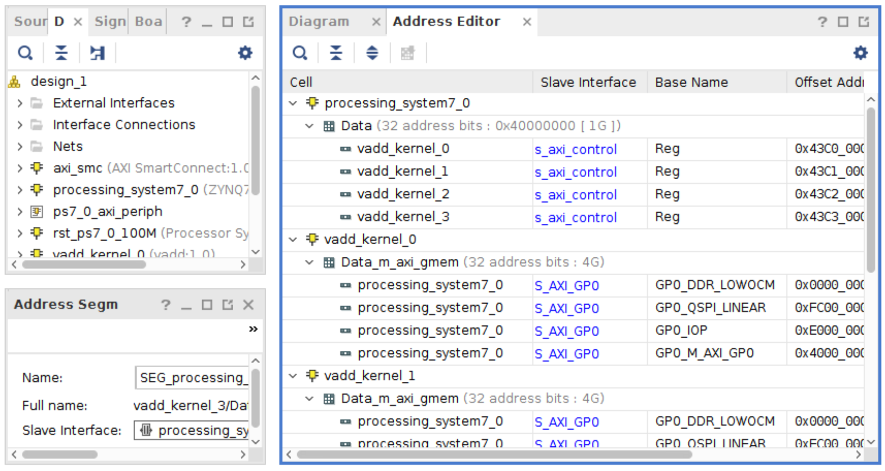

Sour | D | Sign | Boa
design_1
External Interfaces
Interface Connections
Nets
axi_smc (AXI SmartConnect:1.0
processing_system7_0 (ZYNQ7
ps7_0_axi_periph
rst_ps7_0_100M (Processor Sy
vadd_kernel_0 (vadd:1.0)

Address Segm

Name:	SEG_processing_
Full name:	vadd_kernel_0/Da
Slave Interface:	processing_sy

Diagram | Address Editor

Cell	Slave Interface	Base Name	Offset Addr
processing_system7_0			
Data (32 address bits : 0x40000000 [1G])			
vadd_kernel_0	s_axi_control	Reg	0x43C0_000
vadd_kernel_1	s_axi_control	Reg	0x43C1_000
vadd_kernel_2	s_axi_control	Reg	0x43C2_000
vadd_kernel_3	s_axi_control	Reg	0x43C3_000
vadd_kernel_0			
Data_m_axi_gmem (32 address bits : 4G)			
processing_system7_0	S_AXI_GP0	GP0_DDR_LOWOCM	0x0000_000
processing_system7_0	S_AXI_GP0	GP0_QSPI_LINEAR	0xFC00_000
Excluded Address Segments (2)			
processing_system7_0	S_AXI_GP0	GP0_IOP	0xE000_000
processing_system7_0	S_AXI_GP0	GP0_M_AXI_GP0	0x4000_000
vadd_kernel_1			
vadd_kernel_2			
vadd_kernel_3			


Sour | D | Sign | Boa
design_1
External Interfaces
Interface Connections
Nets
axi_smc (AXI SmartConnect:1.0
processing_system7_0 (ZYNQ7
ps7_0_axi_periph
rst_ps7_0_100M (Processor Sy
vadd_kernel_0 (vadd:1.0)

Address Segm

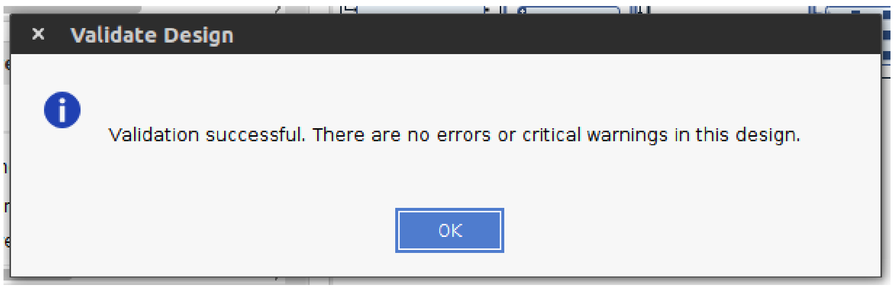
Name:	SEG_processing_
Full name:	vadd_kernel_3/Da
Slave Interface:	processing_sy

Diagram | Address Editor

Cell	Slave Interface	Base Name	Offset Addr
processing_system7_0			
Data (32 address bits : 0x40000000 [1G])			
vadd_kernel_0	s_axi_control	Reg	0x43C0_000
vadd_kernel_1	s_axi_control	Reg	0x43C1_000
vadd_kernel_2	s_axi_control	Reg	0x43C2_000
vadd_kernel_3	s_axi_control	Reg	0x43C3_000
vadd_kernel_0			
Data_m_axi_gmem (32 address bits : 4G)			
processing_system7_0	S_AXI_GP0	GP0_DDR_LOWOCM	0x0000_000
processing_system7_0	S_AXI_GP0	GP0_QSPI_LINEAR	0xFC00_000
processing_system7_0	S_AXI_GP0	GP0_IOP	0xE000_000
processing_system7_0	S_AXI_GP0	GP0_M_AXI_GP0	0x4000_000
vadd_kernel_1			
Data_m_axi_gmem (32 address bits : 4G)			
processing_system7_0	S_AXI_GP0	GP0_DDR_LOWOCM	0x0000_000
processing_system7_0	S_AXI_GP0	GP0_QSPI_LINEAR	0xFC00_000

Here we can also see the offset address of each compute unit on the processing system that we need to save inside the Bitfile Description File.

Lets go back to the “Diagram“ tab and click on the “Validate Design“ button, to check our design for errors. If you followed the guide correctly your design should pass the validation.


Validate Design

Validation successful. There are no errors or critical warnings in this design.

OK

We can also add some memories to our design, let’s add a BRAM. Click the “add IP“ button and search for “BRAM“ and add the “AXI BRAM Controller“.

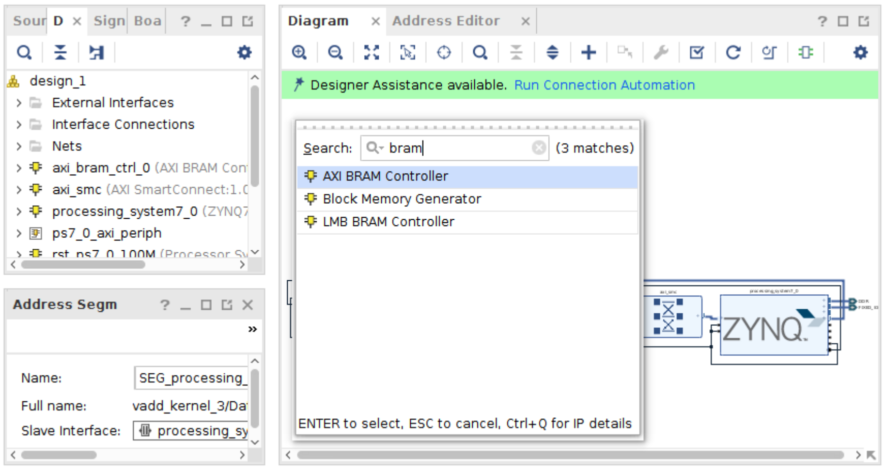


Diagram
Address Editor
Designer Assistance available. Run Connection Automation
design_1
External Interfaces
Interface Connections
Nets
axi_bram_ctrl_0 (AXI BRAM Con
axi_smc (AXI SmartConnect:1.0
processing_system7_0 (ZYNQ7
ps7_0_axi_periph
Address Segm
ZYNQ

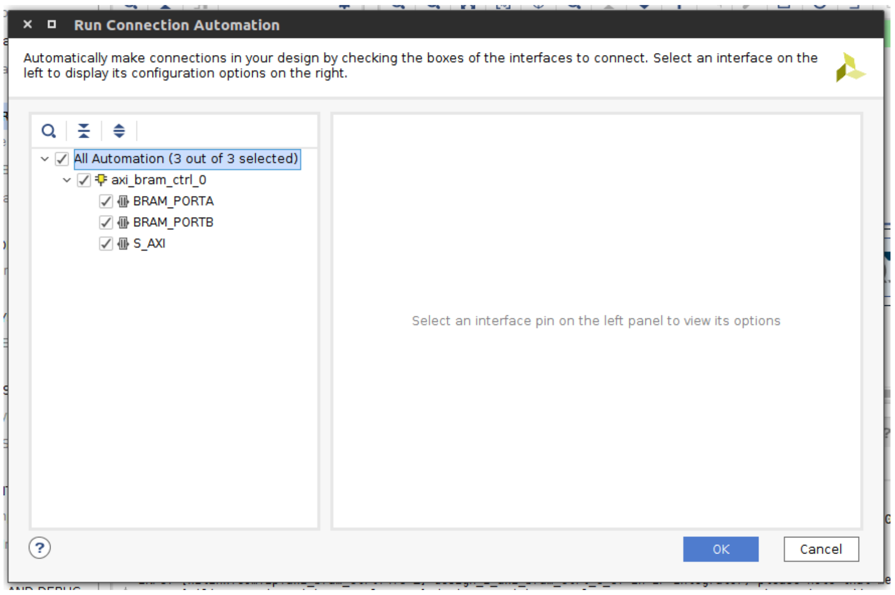
Search: bram (3 matches)

AXI BRAM Controller
Block Memory Generator
LMB BRAM Controller

ENTER to select, ESC to cancel, Ctrl+Q for IP details

Name: SEG_processing_
Full name: vadd_kernel_3/Da
Slave Interface: processing_sy

Then click “Run Block Automation“, select all check boxes and click “OK“.

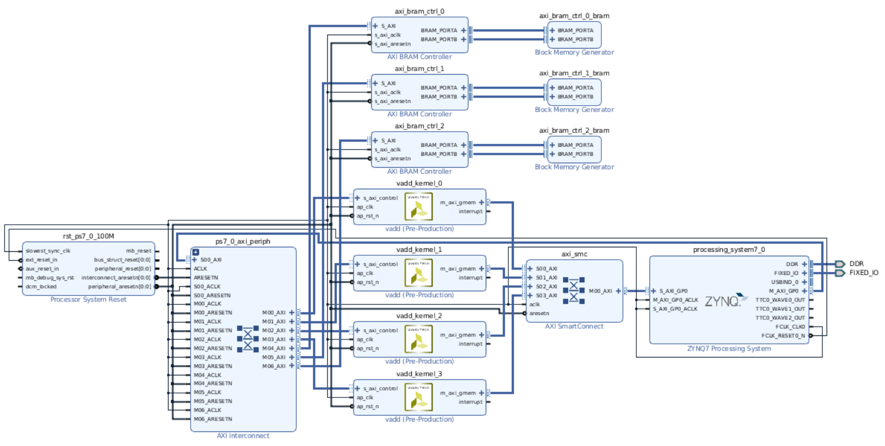

Run Connection Automation

Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

- ✓ All Automation (3 out of 3 selected)
  - ✓ axi_bram_ctrl_0
    - ✓ BRAM_PORTA
    - ✓ BRAM_PORTB
    - ✓ S_AXI

Select an interface pin on the left panel to view its options

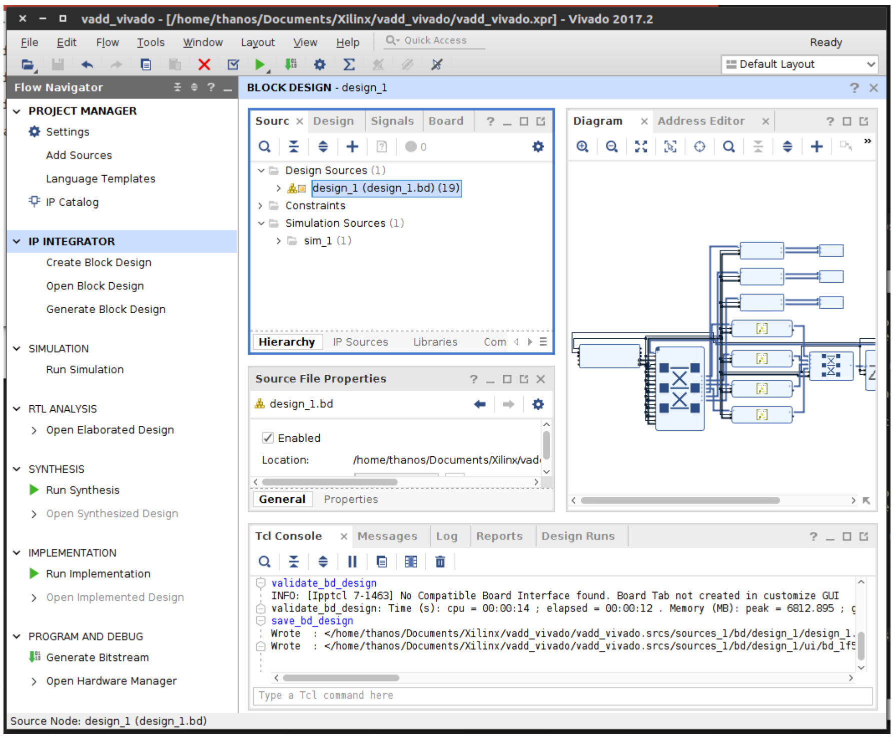
You can repeat this process to add more BRAMs. Lets add 3 BRAMs, one for each buffer of the vector addition.



After adding the BRAMs, click on the “Validate Design“ button, to check our design ones again. Then press `ctrl+s` to save the design.

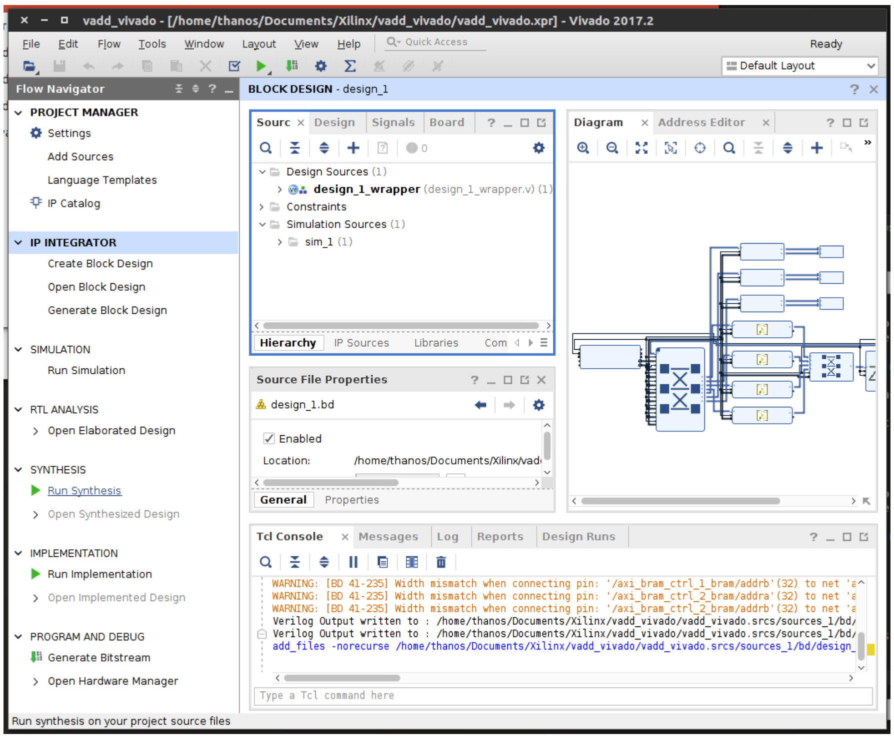
The only thing we now need to do is to export the bitfile of our design.

On the “Source“ tab right click the design and select “Create HDL Wrapper..“ then on the new window click “OK“.



```
validate_bd_design
INFO: [Ipptcl 7-1463] No Compatible Board Interface found. Board Tab not created in customize GUI
validate_bd_design: Time (s): cpu = 00:00:14 ; elapsed = 00:00:12 . Memory (MB): peak = 6812.895 ; g
save_bd_design
Wrote  : </home/thanos/Documents/Xilinx/vadd_vivado/vadd_vivado.srcs/sources_1/bd/design_1/design_1.
Wrote  : </home/thanos/Documents/Xilinx/vadd_vivado/vadd_vivado.srcs/sources_1/bd/design_1/ui/bd_1f5
```

Click under the “Flow Manager“ >“Synthesis“ >“Run Synthesis“ select number of jobs (based on your pc’s cpus) and click “OK“.

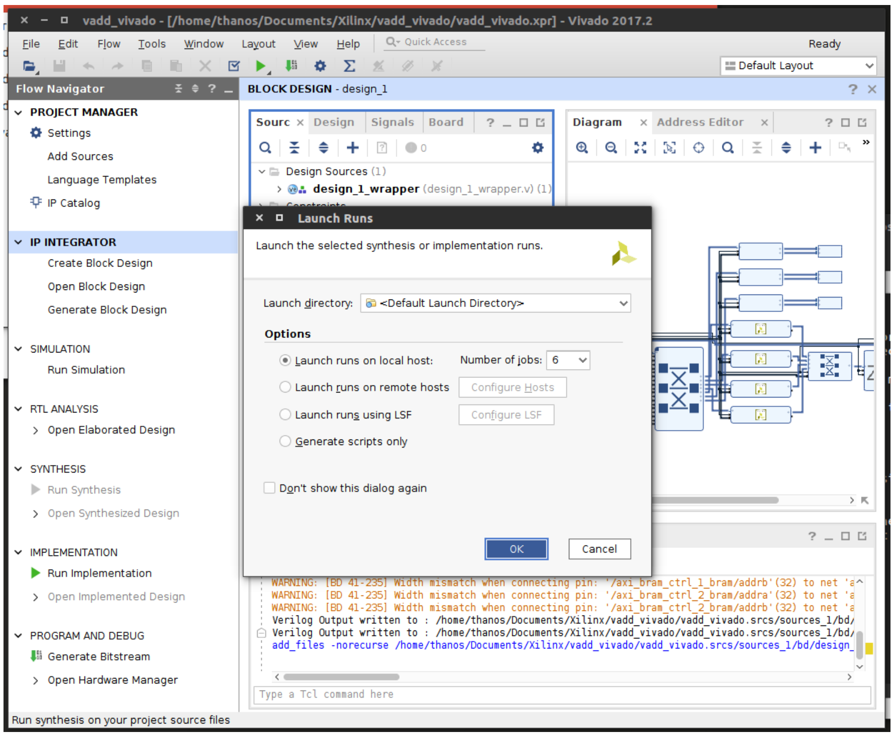


vadd_vivado - [/home/thanos/Documents/Xilinx/vadd_vivado/vadd_vivado.xpr] - Vivado 2017.2

```
WARNING: [BD 41-235] Width mismatch when connecting pin: '/axi_bram_ctrl_1_bram/addrb'(32) to net 'a
WARNING: [BD 41-235] Width mismatch when connecting pin: '/axi_bram_ctrl_2_bram/addra'(32) to net 'a
WARNING: [BD 41-235] Width mismatch when connecting pin: '/axi_bram_ctrl_2_bram/addrb'(32) to net 'a
Verilog Output written to : /home/thanos/Documents/Xilinx/vadd_vivado/vadd_vivado.srcs/sources_1/bd/
Verilog Output written to : /home/thanos/Documents/Xilinx/vadd_vivado/vadd_vivado.srcs/sources_1/bd/
add_files -norecurse /home/thanos/Documents/Xilinx/vadd_vivado/vadd_vivado.srcs/sources_1/bd/design_
```

Run synthesis on your project source files

As soon as the synthesis complete, a new window will pop up and ask you to “Run Implementation“ then click “OK“ and then “OK“.



Launch Runs

Launch the selected synthesis or implementation runs.

Launch directory: <Default Launch Directory>

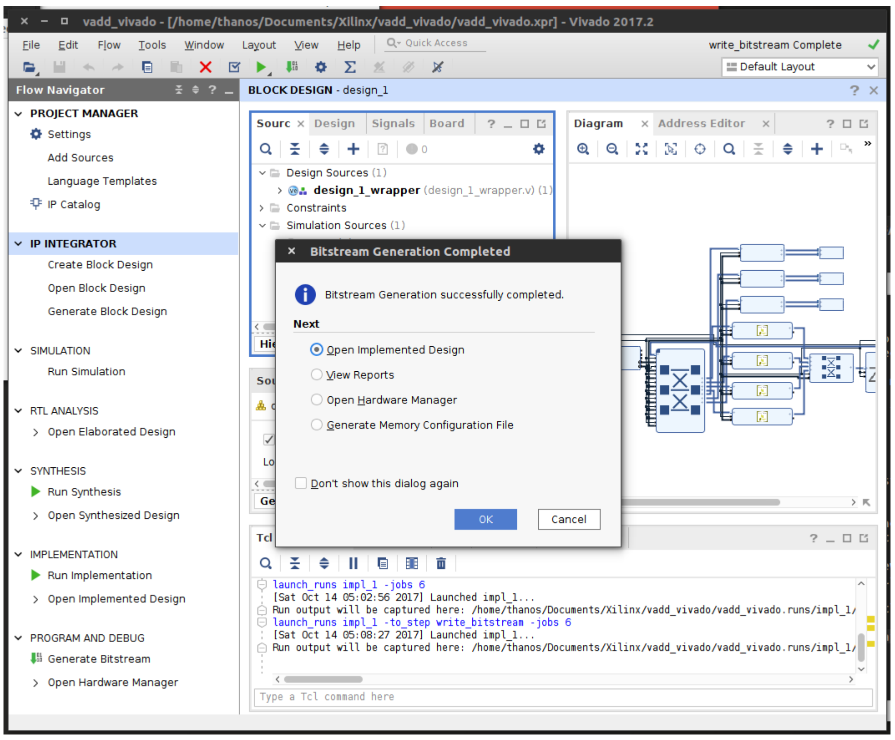
Options

- [✓ Launch runs on local host:] Number of jobs: 6
- Launch runs on remote hosts
- Launch runs using LSF
- Generate scripts only

Don't show this dialog again

When this is completed, on the new window click “Cancel“.

Now click “Program And Debug“ >“Generate Bitstream“, on the new window, click “OK“.



Bitstream Generation Completed

Bitstream Generation successfully completed.

Next

[✓ Open Implemented Design]
View Reports
Open Hardware Manager
Generate Memory Configuration File

Don't show this dialog again

```
launch_runs impl_1 -jobs 6
[Sat Oct 14 05:02:56 2017] Launched impl_1...
Run output will be captured here: /home/thanos/Documents/Xilinx/vadd_vivado/vadd_vivado.runs/impl_1/
launch_runs impl_1 -to_step write_bitstream -jobs 6
[Sat Oct 14 05:08:27 2017] Launched impl_1...
Run output will be captured here: /home/thanos/Documents/Xilinx/vadd_vivado/vadd_vivado.runs/impl_1/
```

Type a Tcl command here

The bitfile can be now found on:
“/<project>/<project>.runs/impl_1/design_1_wrapper.bit“

As we mention on a previous step, from the “Address Editor“ in Vivado you can get the info about the offset addresses of the compute units and the BRAMs.

B Examples

B.1 System preparation

In this small section we will guide you through the preparation of your development system and your on-board system. It is suggested that you work on a Linux based system for development. Also, we are using the Zedboard FPGA board but a similar process can be followed for other similar boards.

B.1.1 Download linux-xlnx

First of all you need to get the latest stable Linux from Xilinx compiled for your board and the sources to be able to compile kernels for it.

In our examples we used “Xilinx Linux v2017.2“ for Zedboard which can be downloaded ready for use from `http://www.wiki.xilinx.com/Zynq_2017.2_Release`. The system files we need to boot Linux are “BOOT.BIN“, “download.bit“, “image.ub“, “system.dtb“ and “zynq_fsbl.elf“.

Further more we need to grap the respective source code for Zedboard Linux from `https://github.com/Xilinx/linux-xlnx/tree/xilinx-v2017.2`.

B.1.2 Setup arm-linux-gnueabihf

To be able to compile our codes for the Zedboard we need to setup the “Linaro Binary Toolchain“ on our system.

We got it from `https://releases.linaro.org/components/toolchain/binaries/latest-5/arm-linux-gnueabihf/`, by downloading the archive for our system. You can also install it by any other way you think it the best.

We installed it on the path “/opt/gcc-linaro/“, so all the binaries are located on “/opt/gcc-linaro/bin/arm-linux-gnueabihf-*“. Thus, whenever we compile our code for our board we will use these binaries.

B.1.3 SD-Card preparation

We need to prepare the SD card from which our Linux board will boot. For easier usage, we formated the SD card to 2 partitions, the “BOOT“ FAT32 partition for the system’s files and the “DATA“ EXT4 partition for our program’s files.

- BOOT partition
  - BOOT.BIN
  - download.bit
  - image.ub

    - system.dtb
    - zynq_fsbl.elf
- DATA partition
    - Your project files

B.1.4 Booting the system

Insert he SD card into your board and set the correct configuration of your board to “boot from SD card“ (check you board’s manual). Then, connect the board’s UART port to your computer and power the boards. Meanwhile, open your preferred serial port terminal on your computer, I personally used “minicom“ but you can use any program that fits your needs.

Wait the board to start. You should now see a “Zynq >“ waiting for you to run something.You can then use the command “run bootcmd“ to start the Linux booting process.

Upon boot, you will be asked to log into your Linux account. Log in as “root“ with the password “root“.

Your board is now ready.

You can mount and enter the second partition of the SD card by using the command:

```
mkdir /mnt/sd && mount /dev/mmcblk0p2 /mnt/sd && cd /mnt/sd
```

B.2 Compiling the Codes

In this tiny section we will guide you on how to compile C/C++ codes to binaries for your board.

B.2.1 Compile the Kernel Sources

To be able to use the system’s memory as device buffers for our applications, we need to compile the “udmabuf“ as a kernel module.

To do so, we first need to prepare our kernel sources. Thus, open a terminal and navigate to the headers source folder.

So he have our arm compile on the path:

```
/opt/gcc-linaro/bin/arm-linux-gnueabihf-
```

We need to set the zynq config file as config:

```
cp arch/arm/configs/xilinx_zynq_defconfig .config
```

Then configure the kernel build:

```
make ARCH=arm CROSS_COMPILE=/opt/gcc-linaro/bin/
arm-linux-gnueabihf- oldconfig
```

Optionally, you can customise the build using menuconfig:

```
make ARCH=arm CROSS_COMPILE=/opt/gcc-linaro/bin/
arm-linux-gnueabihf- menuconfig
```

Then run the compilation:

```
arm-linux-gnueabihf- UIMAGE_LOADADDR=0x8000 uImage
```

B.2.2 Compile a Kernel Module

As we mentioned, we need to compile the "udmabuf" as a kernel module. Get the code of "udmabuf" from its github page urlhttps://github.com/ikwzm/udmabuf.

Now, create a make file to handle the compiling. Change the paths on the code with your systems paths.

```
obj-m += udmabuf.o
# Path to sources
KERNEL_DIR=$(PWD)/linux-xlnx-xilinx-v2017.2
# Path to compiler
COMPILER_PATH=/opt/gcc-linaro/bin/arm-linux-gnueabihf-

all:
  make ARCH=arm CROSS_COMPILE=$(COMPILER_PATH) CC=$(COMPILER_PATH)gcc
     -C $(KERNEL_DIR) M=$(PWD) modules

clean:
  make -C $(KERNEL_DIR) M=$(PWD) clean
```

Then just run it (`make`) to compile the kernel module. The generated `udmabuf.ko` file is the compiled kernel module. You can copy it to the SD card's second partition and use it from the board's linux using the code:

```
insmod udmabuf.ko udmabuf0=0x200000 udmabuf1=0x200000 udmabuf2=0
    x200000 udmabuf3=0x200000
```

which we mention on an earlier section.

B.2.3 Compile an application

The compiling process of a C/C++ application is relative simple. You just use the arm compiler istead of your system's compiler.

Since we have our arm compile on the path:
`/opt/gcc-linaro/bin/arm-linux-gnueabihf-`
we can compile our SimplePL application using the command:

```
/opt/gcc-linaro/bin/arm-linux-gnueabihf-gcc -o myapp -static myapp.c
    ./include/simplePL.c ./include/tiny-json.c ./include/benchtimer.c
    -I'./include/'
```

Where the include folder has all the libraries sources.

B.3 Example Vector Addition

To run this example we first need to compile its code. So, create a folder for this example with the following structure and files:

- `example_vadd`
  - `include`
    - `simplePL.c`
    - `simplePL.h`
    - `tiny-json.c`
    - `tiny-json.h`
    - `benchtimer.c`
    - `benchtimer.h`
  - `vadd.bit`
  - `vadd.json`
  - `vadd.c`
  - `example_vadd`
  - `udmabuf.ko`

Where the `udmabuf.ko` file is the compiled kernel module (we analyzed the process in a previous section) and `example_vadd` is the compiled `vadd.c` code (our application) which can be compiled using the command:

```
arm-linux-gnueabihf-gcc -o example_vadd -static vadd.c ./include/*.c
    -I'./include/'
```

Then you can copy all the folder inside 2nd partition of the SD card with the Linux of your board and boot it. After it boots and you sign in, insert the kernel mod, and then run the example.

```
    # Mount sd
    mkdir /mnt/sd
    mount /dev/mmcblk0p2 /mnt/sd
    # Navigate to sd folder
    cd /mnt/sd

    # Enter example folder
    cd example_vadd/

    # Insert kernel module
```

```
 insmod udmabuf.ko udmabuf0=0x200000 udmabuf1=0x200000 udmabuf2=0
x200000 udmabuf3=0x200000

 # Run example for vectors of 2048 elements
 ./example_vadd 2048
```

The example application codes are bellow.

```c
/*
 * Copyright Grammatopoulos Athanasios Vasileios
 * agrammatopoulos@isc.tuc.gr
 * Technical University of Crete
 *
 * Test vadd using SimplePL Library
 */

// Libraries
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "simplePL.h"
#include "benchtimer.h"


int main(int argc, char *argv[]){
  // Number of elements
  pl_uint n = 1024;

  // Check parameters
  pl_uint param = 0;
  if(argc == 2) {
    param = atoi(argv[1]);
    // If valid
    if (param > 0) {
      n = param;
    } else {
      printf("Invalid parameters.\n");
    }
  }
  printf("Running for {n=%d}.\n", n);

  // Host input vectors
  pl_int *h_a;
  pl_int *h_b;
  // Host output vector
  pl_int *h_c;
  // Result test vector
  pl_int *h_r;

  // Size, in bytes, of each vector
  size_t bytes = n * sizeof(pl_int);

  // Allocate memory for each vector on host
  h_a = (pl_int*) malloc(bytes);
  h_b = (pl_int*) malloc(bytes);
  h_c = (pl_int*) malloc(bytes);
  h_r = (pl_int*) malloc(bytes);

  // Init rand
  time_t t;
  srand((unsigned) time(&t));

  int i;
  // Initialize vectors on host
  for (i = 0; i < n; i++) {
    h_a[i] = (rand() % 1000);
    h_b[i] = (rand() % 1000);
    h_c[i] = 0;
  }

  // Calculate results
  printf("Calculating results on cpu... ");
  benchtimer* execute_cpu = benchtimer_create();
  benchtimer_start(execute_cpu);
  for (i = 0; i < n; i++) {
    h_r[i] = h_a[i] + h_b[i];
  }
  benchtimer_stop(execute_cpu);
  printf("Done (%lf secs)\n", benchtimer_get(execute_cpu));

    size_t globalSize, localSize;
    pl_int err;

  // Init device
  printf("Initializing device... ");
  err = plInitDevice();
  if (err != PL_SUCCESS) {
    printf("Failed [Error %d]\n", err);
    return EXIT_FAILURE;
  } else {
    printf("Done\n");
  }

  // Get bitfile info
  pl_bitfile bitfile;
  printf("Get bitfile info... ");
  err = plGetBitfileInfo(&bitfile, "./bitfiles/", "vadd");
  if (err != PL_SUCCESS) {
    printf("Failed [Error %d]\n", err);
    return EXIT_FAILURE;
  } else {
    printf("Done\n");
  }

  // Program pl
  printf("Program PL... ");
  benchtimer* programing_pl = benchtimer_create();
  benchtimer_start(programing_pl);
  err = plProgram(bitfile);
  benchtimer_stop(programing_pl);
  if (err != PL_SUCCESS) {
    printf("Failed [Error %d]\n", err);
    return EXIT_FAILURE;
  } else {
    printf("Done (%lf secs)\n", benchtimer_get(programing_pl));
  }

  // Initialize bitfile
  printf("Initialize bitfile on PL... ");
  err = plBitfileInitialize(bitfile, PL_BITFILE_INIT_ALL);
  if (err != PL_SUCCESS) {
    printf("Failed [Error %d]\n", err);
    return EXIT_FAILURE;
  } else {
    printf("Done\n");
  }

  // Get kernel
  pl_kernel kernel;
  printf("Get kernel... ");
  kernel = plGetKernelByName(bitfile, "vadd_int");
  if (kernel == NULL) {
    printf("Failed\n");
    return EXIT_FAILURE;
  } else {
    printf("Done\n");
  }

  // Get sysmemory
  pl_memory sysmemory;
  printf("Get system memory... ");
  sysmemory = plGetSysMemoryByIndex(0);
  if (sysmemory == NULL) {
    printf("Failed\n");
    return EXIT_FAILURE;
  } else {
    printf("Done\n");
  }

  // Disable global interrupt
  printf("Disable kernel's global interrupts... ");
  err = plSetKernelGlobalInterrupt(kernel, PL_DISABLE);
  if (err != PL_SUCCESS) {
    printf("Failed [Error %d]\n", err);
  } else {
    printf("Done\n");
  }

  // Create the input and output arrays in device memory for our
    calculation
  printf("Creating buffers... ");
  pl_buffer d_a = plCreateBuffer(sysmemory, bytes, &err);
  if (err != PL_SUCCESS) {printf("Failed a [Error %d]\n", err);return
    EXIT_FAILURE;}
  pl_buffer d_b = plCreateBuffer(sysmemory, bytes, &err);
  if (err != PL_SUCCESS) {printf("Failed b [Error %d]\n", err);return
    EXIT_FAILURE;}
  pl_buffer d_c = plCreateBuffer(sysmemory, bytes, &err);
  if (err != PL_SUCCESS) {printf("Failed c [Error %d]\n", err);return
    EXIT_FAILURE;}
  printf("Done\n");

  // Copy data to device
  printf("Writing buffers... ");
  benchtimer* writing_buffers = benchtimer_create();
  benchtimer_start(writing_buffers);
  err = plWriteBuffer(d_a, 0, bytes, h_a);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
   EXIT_FAILURE;}
  err = plWriteBuffer(d_b, 0, bytes, h_b);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
   EXIT_FAILURE;}
  benchtimer_stop(writing_buffers);
  printf("Done (%lf secs)\n", benchtimer_get(writing_buffers));

  // Set parameters
  printf("Setting kernel parameters... ");
  err = plSetKernelArg(kernel, 0, sizeof(pl_buffer), (void *) &d_a);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
   EXIT_FAILURE;}
  err = plSetKernelArg(kernel, 1, sizeof(pl_buffer), (void *) &d_b);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
   EXIT_FAILURE;}
  err = plSetKernelArg(kernel, 2, sizeof(pl_buffer), (void *) &d_c);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
   EXIT_FAILURE;}
  err = plSetKernelArg(kernel, 3, sizeof(pl_uint), (void *) &n);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
   EXIT_FAILURE;}
  printf("Done\n");

  // Run kernel
  printf("Executing kernel... ");
  size_t global = n;
  benchtimer* execute_fpga = benchtimer_create();
  benchtimer_start(execute_fpga);
  // Blocking
  err = plRunNDRangeKernel(kernel, 1, NULL, &global, PL_TRUE);
  // Non-Blocking
  //err = plRunNDRangeKernel(kernel, 1, NULL, &global, PL_FALSE);
  //if (err != PL_SUCCESS) {printf("Failed Exec [Error %d]\n", err);
    return EXIT_FAILURE;}
  //err = plWaitNDRangeKernel(kernel);
  //if (err != PL_SUCCESS) {printf("Failed Wait [Error %d]\n", err);
    return EXIT_FAILURE;}
  benchtimer_stop(execute_fpga);
  printf("Done (%lf secs)\n", benchtimer_get(execute_fpga));

  // Copy data from device
  printf("Reading buffers... ");
  benchtimer* reading_buffers = benchtimer_create();
  benchtimer_start(reading_buffers);
  err = plReadBuffer(d_c, 0, bytes, h_c);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
   EXIT_FAILURE;}
  benchtimer_stop(reading_buffers);
  printf("Done (%lf secs)\n", benchtimer_get(reading_buffers));

  // Check for errors
  printf("Checking results... ");
  int errors = 0;
  for (i = 0; i < n; i++) {
    if(h_c[i] != h_r[i]){
      errors ++;
    }
  }
  if (errors == 0) {
    printf("No errors\n");
  }
  else {
    printf("%d error(s)\n", errors);
  }

  // Print buffers
  printf("a = [");
  for (i = 0; i < n; ++i) {
    printf("%d ", h_a[i]);
    if(i == 256){printf("... ");break;}
  }
  printf("]\n");
  printf("b = [");
  for (i = 0; i < n; ++i) {
    printf("%d ", h_b[i]);
    if(i == 256){printf("... ");break;}
  }
  printf("]\n");
  printf("c = [");
  for (i = 0; i < n; ++i) {
    printf("%d ", h_c[i]);
    if(i == 256){printf("... ");break;}
  }
  printf("]\n");

  // Release buffers
  printf("Release buffers... ");
  err = plReleaseBuffer (d_a);
  if (err != PL_SUCCESS) {printf("Failed a [Error %d]\n", err);return
    EXIT_FAILURE;}
  err = plReleaseBuffer (d_b);
  if (err != PL_SUCCESS) {printf("Failed b [Error %d]\n", err);return
    EXIT_FAILURE;}
  err = plReleaseBuffer (d_c);
  if (err != PL_SUCCESS) {printf("Failed c [Error %d]\n", err);return
    EXIT_FAILURE;}
```

```
  printf("Done\n");

  // Release bitfile
  // this also release kernels and memories
  printf("Release bitfile's info '... ");
  err = plBitfileRelease (bitfile);
  if (err != PL_SUCCESS) {
    printf("Failed [Error %d]\n", err);
  } else {
    printf("Done\n");
  }

  // Show results again
  printf("\n");
  if (errors == 0){
    printf("Test PASSED.\n");
  } else {
    printf("Test Failed.\n");
  }


  return EXIT_SUCCESS;
}
```

Listing B.1: Example vadd application using OpenCL HLS Kernel

```
{
  "name" : "vadd",
  "kernels" : [
    {
      "name" : "vadd_int",
      "address" : ["0x43C00000", "0x43C10000", "0x43C20000", "0
  x43C30000"],
      "workgroup" : [256, 1, 1],
      "size" : 0,
      "control" : {
        "ctrl" : "0x00", "gie" : "0x04", "ier" : "0x08", "isr" : "0
  x0c"
      },
      "group" : {
        "id_x" : "0x10", "id_y" : "0x18", "id_z" : "0x20",
        "offset_x" : "0x28", "offset_y" : "0x30", "offset_z" : "0x38"
      },
      "arguments" : [
        {"name" : "a", "offset" : "0x40", "size" : 4},
        {"name" : "b", "offset" : "0x48", "size" : 4},
        {"name" : "c", "offset" : "0x50", "size" : 4},
        {"name" : "n", "offset" : "0x58", "size" : 4}
      ]
    }
  ],
  "memories" : [
    {"name" : "bram_a", "address" : "0x40000000", "size" : 8192},
    {"name" : "bram_b", "address" : "0x42000000", "size" : 8192},
    {"name" : "bram_c", "address" : "0x44000000", "size" : 8192}
    ]
}
```

Listing B.2: The Bitfile Description File for the exmaple application

B.4 Example Matrix Multiplication

To run this example we first need to compile its code. So, create a folder for this example with the following structure and files:

- `example_mmult`
  - `include`
    - `simplePL.c`
    - `simplePL.h`
    - `tiny-json.c`
    - `tiny-json.h`
    - `benchtimer.c`
    - `benchtimer.h`
  - `mmult.bit`
  - `mmult.json`
  - `mmult.c`
  - `example_mmult`
  - `udmabuf.ko`

Where the `udmabuf.ko` file is the compiled kernel module (we analyzed the process in a previous section) and `example_mmult` is the compiled `mmult.c` code (our application) which can be compiled using the command:

```
arm-linux-gnueabihf-gcc -o example_mmult -static mmult.c ./include/*.
    c -I'./include/'
```

Then you can copy all the folder inside 2nd partition of the SD card with the Linux of your board and boot it. After it boots and you sign in, insert the kernel mod, and then run the example.

```
# Mount sd
mkdir /mnt/sd
mount /dev/mmcblk0p2 /mnt/sd
# Navigate to sd folder
cd /mnt/sd

# Enter example folder
cd example_mmult/

# Insert kernel module
```

```
insmod udmabuf.ko udmabuf0=0x200000 udmabuf1=0x200000 udmabuf2=0
x200000 udmabuf3=0x200000

# Run example for vectors of 2048 elements
./example_mmult 2048
```

The example application codes are bellow.

```
/*
 * Copyright Grammatopoulos Athanasios Vasileios
 * agrammatopoulos@isc.tuc.gr
 * Technical University of Crete
 *
 * Test vadd using SimplePL Library
 */

// Libraries
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "simplePL.h"
#include "benchtimer.h"


int main(int argc, char *argv[]){
  // Default Matrix sizes
  pl_uint M = 16;
  pl_uint N = 64;
  pl_uint K = 32;

  // Check parameters
  pl_uint A_K = 0;
  pl_uint A_M = 0;
  pl_uint B_N = 0;
  pl_uint B_K = 0;
  if(argc == 5) {
    A_K = atoi(argv[1]);
    A_M = atoi(argv[2]);
    B_N = atoi(argv[3]);
    B_K = atoi(argv[4]);

    // If valid
    if (A_K == B_K && A_M > 0 && B_N > 0 && A_K > 0) {
      M = A_M;
      N = B_N;
      K = A_K;
    } else {
       printf("Invalid parameters.\n");
    }
  }
  printf("Running for {M=%d, N=%d, K=%d}.\n", M, N, K);

  // Host input vectors
  pl_int *h_a;
  pl_int *h_b;
  // Host output vector
  pl_int *h_c;
  // Result test vector
  pl_int *h_r;

  // Size, in bytes, of each vector
  size_t bytes_a = M * K * sizeof(pl_int);
  size_t bytes_b = K * N * sizeof(pl_int);
  size_t bytes_c = M * N * sizeof(pl_int);

  // Allocate memory for each vector on host
  h_a = (pl_int*) malloc(bytes_a);
  h_b = (pl_int*) malloc(bytes_b);
  h_c = (pl_int*) malloc(bytes_c);
  h_r = (pl_int*) malloc(bytes_c);

  // Init rand
  time_t t;
  srand((unsigned) time(&t));

  int i, j;
  // Initialize matrix a on host
  for (i = 0; i < M; i++) {
    for (j = 0; j < K; j++) {
      h_a[K * i + j] = (rand() % 10);
    }
  }
  // Initialize matrix b on host
  for (i = 0; i < K; i++) {
    for (j = 0; j < N; j++) {
      h_b[N * i + j] = (rand() % 10);
    }
  }
  // Initialize matrix c on host and results
  for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
      h_c[N * i + j] = (rand() % 10);
      h_r[N * i + j] = 0;
    }
  }

  // Calculate results
  printf("Calculating results on cpu... ");
  benchtimer* execute_cpu = benchtimer_create();
  benchtimer_start(execute_cpu);
  pl_int value = 0;
  for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
      // Reset value
      value = 0;
      // Calculate value
      for (int l = 0; l < K; l++){
        value += h_a[K * i + l] * h_b[N * l + j];
      }
      h_r[N * i + j] = value;
    }
  }
  benchtimer_stop(execute_cpu);
  printf("Done (%lf secs)\n", benchtimer_get(execute_cpu));

    size_t globalSize, localSize;
    pl_int err;

  // Init device
  printf("Initializing device... ");
  err = plInitDevice();
  if (err != PL_SUCCESS) {
    printf("Failed [Error %d]\n", err);
    return EXIT_FAILURE;
  } else {
    printf("Done\n");
  }

  // Get bitfile info
  pl_bitfile bitfile;
  printf("Get bitfile info... ");
  err = plGetBitfileInfo(&bitfile, "./bitfiles/", "mmult");
  if (err != PL_SUCCESS) {
    printf("Failed [Error %d]\n", err);
    return EXIT_FAILURE;
  } else {
    printf("Done\n");
  }

  // Program pl
  printf("Program PL... ");
  benchtimer* programing_pl = benchtimer_create();
  benchtimer_start(programing_pl);
  err = plProgram(bitfile);
  benchtimer_stop(programing_pl);
  if (err != PL_SUCCESS) {
    printf("Failed [Error %d]\n", err);
    return EXIT_FAILURE;
  } else {
    printf("Done (%lf secs)\n", benchtimer_get(programing_pl));
  }

  // Initialize bitfile
  printf("Initialize bitfile on PL... ");
  err = plBitfileInitialize(bitfile, PL_BITFILE_INIT_ALL);
  if (err != PL_SUCCESS) {
    printf("Failed [Error %d]\n", err);
    return EXIT_FAILURE;
  } else {
    printf("Done\n");
  }

  // Get kernel
  pl_kernel kernel;
  printf("Get kernel... ");
  kernel = plGetKernelByName(bitfile, "mmult_int");
  if (kernel == NULL) {
    printf("Failed\n");
    return EXIT_FAILURE;
  } else {
    printf("Done\n");
  }

  // Get sysmemory
  pl_memory sysmemory;
  printf("Get system memory... ");
  sysmemory = plGetSysMemoryByIndex(0);
  if (sysmemory == NULL) {
    printf("Failed\n");
    return EXIT_FAILURE;
  } else {
    printf("Done\n");
  }

  // Disable global interrupt
  printf("Disable kernel's global interrupts... ");
  err = plSetKernelGlobalInterrupt(kernel, PL_DISABLE);
  if (err != PL_SUCCESS) {
    printf("Failed [Error %d]\n", err);
  } else {
    printf("Done\n");
  }

  // Create the input and output arrays in device memory for our
    calculation
  printf("Creating buffers... ");
  pl_buffer d_a = plCreateBuffer(sysmemory, bytes_a, &err);
  if (err != PL_SUCCESS) {printf("Failed a [Error %d]\n", err);return
    EXIT_FAILURE;}
  pl_buffer d_b = plCreateBuffer(sysmemory, bytes_b, &err);
  if (err != PL_SUCCESS) {printf("Failed b [Error %d]\n", err);return
    EXIT_FAILURE;}
  pl_buffer d_c = plCreateBuffer(sysmemory, bytes_c, &err);
  if (err != PL_SUCCESS) {printf("Failed c [Error %d]\n", err);return
    EXIT_FAILURE;}
  printf("Done\n");

  // Copy data to device
  printf("Writing buffers... ");
  benchtimer* writing_buffers = benchtimer_create();
  benchtimer_start(writing_buffers);
  err = plWriteBuffer(d_a, 0, bytes_a, h_a);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
   EXIT_FAILURE;}
  err = plWriteBuffer(d_b, 0, bytes_b, h_b);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
   EXIT_FAILURE;}
  benchtimer_stop(writing_buffers);
  printf("Done (%lf secs)\n", benchtimer_get(writing_buffers));

  // Set parameters
  printf("Setting kernel parameters... ");
  err = plSetKernelArg(kernel, 0, sizeof(pl_uint), (void *) &M);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
    EXIT_FAILURE;}
  err = plSetKernelArg(kernel, 1, sizeof(pl_uint), (void *) &N);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
    EXIT_FAILURE;}
  err = plSetKernelArg(kernel, 2, sizeof(pl_uint), (void *) &K);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
    EXIT_FAILURE;}
  err = plSetKernelArg(kernel, 3, sizeof(pl_buffer), (void *) &d_a);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
    EXIT_FAILURE;}
  err = plSetKernelArg(kernel, 4, sizeof(pl_buffer), (void *) &d_b);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
    EXIT_FAILURE;}
  err = plSetKernelArg(kernel, 5, sizeof(pl_buffer), (void *) &d_c);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
    EXIT_FAILURE;}
  printf("Done\n");

  // Run kernel
  printf("Executing kernel... ");
  size_t global[2] = {N, M};
  benchtimer* execute_fpga = benchtimer_create();
  benchtimer_start(execute_fpga);
  err = plRunNDRangeKernel(kernel, 2, NULL, global, PL_TRUE);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
    EXIT_FAILURE;}
  benchtimer_stop(execute_fpga);
  printf("Done (%lf secs)\n", benchtimer_get(execute_fpga));

  // Copy data from device
  printf("Reading buffers... ");
  benchtimer* reading_buffers = benchtimer_create();
  benchtimer_start(reading_buffers);
  err = plReadBuffer(d_c, 0, bytes_c, h_c);
  if (err != PL_SUCCESS) {printf("Failed [Error %d]\n", err);return
    EXIT_FAILURE;}
  benchtimer_stop(reading_buffers);
  printf("Done (%lf secs)\n", benchtimer_get(reading_buffers));

  // Check for errors
  printf("Checking results... ");
  int errors = 0;
  for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
      if(h_c[i*N + j] != h_r[N * i + j]){
        errors ++;
      }
    }
  }
  if (errors == 0) {
    printf("No errors\n");
  }
  else {
    printf("%d error(s)\n", errors);
  }

  // Print buffers
  printf("a = [\n");
  if(M * K <= 1024){
    for (i = 0; i < M; i++) {
      printf("   ");
      for (j = 0; j < K; j++) {
        printf("%d ", h_a[K * i + j]);
        printf(" ");
      }
      printf(";\n");
    }
  } else {printf("   [Too big matrix... skipping]\n");}
  printf("]\n");
  printf("b = [\n");
  if(K * N <= 1024){
    for (i = 0; i < K; i++) {
      printf("   ");
      for (j = 0; j < N; j++) {
        printf("%d ", h_b[N * i + j]);
        printf(" ");
      }
      printf(";\n");
    }
  } else {printf("   [Too big matrix... skipping]\n");}
  printf("]\n");
  printf("c = [\n");
  if(M * N <= 1024){
    for (i = 0; i < M; i++) {
      printf("   ");
      for (j = 0; j < N; j++) {
        printf("%d ", h_c[N * i + j]);
        printf(" ");
      }
      printf(";\n");
    }
  } else {printf("   [Too big matrix... skipping]\n");}
  printf("]\n");


  // Release buffers
  printf("Release buffers... ");
  err = plReleaseBuffer (d_a);
  if (err != PL_SUCCESS) {printf("Failed a [Error %d]\n", err);return
    EXIT_FAILURE;}
  err = plReleaseBuffer (d_b);
  if (err != PL_SUCCESS) {printf("Failed b [Error %d]\n", err);return
    EXIT_FAILURE;}
  err = plReleaseBuffer (d_c);
  if (err != PL_SUCCESS) {printf("Failed c [Error %d]\n", err);return
```

```
    EXIT_FAILURE;}
  printf("Done\n");

  // Release bitfile
  // this also release kernels and memories
  printf("Release bitfile's info '... ");
  err = plBitfileRelease (bitfile);
  if (err != PL_SUCCESS) {
    printf("Failed [Error %d]\n", err);
  } else {
    printf("Done\n");
  }

  // Show results again
  printf("\n");
  if (errors == 0){
    printf("Test PASSED.\n");
  } else {
    printf("Test Failed.\n");
  }

  return EXIT_SUCCESS;
}
```

Listing B.3: Example mmult application using OpenCL HLS Kernel

```
{
  "name" : "mmult",
  "kernels" : [
    {
      "name" : "mmult_int",
      "address" : ["0x43C00000", "0x43C10000", "0x43C20000", "0x43C30000", "0x43C40000", "0x43C50000", "0x43C60000", "0x43C70000", "0x43C80000"],
      "workgroup" : [32, 32, 1],
      "size" : 0,
      "control" : {
        "ctrl" : "0x00", "gie" : "0x04", "ier" : "0x08", "isr" : "0x0c"
      },
      "group" : {
        "id_x" : "0x10", "id_y" : "0x18", "id_z" : "0x20",
        "offset_x" : "0x28", "offset_y" : "0x30", "offset_z" : "0x38"
      },
      "arguments" : [
        {"name" : "M", "offset" : "0x40", "size" : 4},
        {"name" : "N", "offset" : "0x48", "size" : 4},
        {"name" : "K", "offset" : "0x50", "size" : 4},
        {"name" : "A", "offset" : "0x58", "size" : 4},
        {"name" : "B", "offset" : "0x60", "size" : 4},
        {"name" : "C", "offset" : "0x68", "size" : 4}
      ]
    }
  ]
}
```

Listing B.4: The Bitfile Description File for the exmaple application

```
/*
 * Copyright 2017 Grammatopoulos Athanasios Vasileios
 * agrammatopoulos@isc.tuc.gr
 * Technical University of Crete
 */

// Xilinx OpenCL Kernel Library
#include <clc.h>
// Kernel's Static Parameters
#include "mmult.h"

// Set required work group size
__attribute__ ((reqd_work_group_size(MMULT_WORKGROUP_SIZE_X,
    MMULT_WORKGROUP_SIZE_Y, MMULT_WORKGROUP_SIZE_Z)))

__kernel void mmult_kernel(
  uint M,      // Number of rows in matrix A
  uint N,      // Number of columns in matrix B
  uint K,      // Number of columns in matrix A and rows in matrix B
  __global mmult_cl_type *A,
  __global mmult_cl_type *B,
  __global mmult_cl_type *C

){
  // Get Global ids
  const int x = get_global_id(0);
  const int y = get_global_id(1);

  // Check range
  if(x >= N || y >= M){
    return;
  }

  // Init values array K + 1 size
  __local mmult_cl_type values[MMULT_HLS_K + 1];
  values[MMULT_HLS_K] = 0;

  // Add data
  __attribute__((xcl_pipeline_loop))
  for (int i = K - 1; i >= 0; i--){
    values[i] = A[K * y + i] * B[N * i + x] + values[i + 1];
  }

  // Save data
  C[N * y + x] = values[0];
}
```

Listing B.5: The mmult OpenCL HLS Kernel

C Library Code

SimplePL has 2 files, header file (`SimplePL.h`) and source file (`SimplePL.c`). Here are their codes.

C.1 SimplePL header

```c
/*
 * Copyright Grammatopoulos Athanasios Vasileios
 * agrammatopoulos@isc.tuc.gr
 * Technical University of Crete
 *
 * SimplePL Library
 * version 1.2.1
 */

#ifndef SIMPLEPL_H
#define SIMPLEPL_H

#ifdef __cplusplus
extern "C" {
#endif

// Load Libraries
#include <stdint.h>
#include <assert.h>
#include <dirent.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stddef.h>



// Type Definitions
//   We need to define the basic types
//   like OpenCL does because the sizeof
//   the default data types in C may varies
//   on different cpu achitectures
typedef char       pl_bool;
typedef int8_t       pl_char;
typedef uint8_t      pl_uchar;
typedef int16_t      pl_short   __attribute__((aligned(2)));
typedef uint16_t     pl_ushort  __attribute__((aligned(2)));
typedef int32_t      pl_int     __attribute__((aligned(4)));
typedef uint32_t     pl_uint    __attribute__((aligned(4)));
typedef int64_t      pl_long    __attribute__((aligned(8)));
typedef uint64_t     pl_ulong   __attribute__((aligned(8)));
typedef uint16_t     pl_half    __attribute__((aligned(2)));
typedef float      pl_float   __attribute__((aligned(4)));
typedef double       pl_double __attribute__((aligned(8)));
//   Also, we define in the same way
//   some insinged integers types based
//   on their size, for use in a bitwise way
typedef uint8_t      pl_u8;
typedef uint16_t     pl_u16;
typedef uint32_t     pl_u32;



// SimplePL basic structure type defs
//   Bitfile typedef
typedef struct _pl_bitfile *     pl_bitfile;
//   Kernel typedefs
typedef struct _pl_kernel *      pl_kernel;       // Kernel's structure
typedef struct _pl_kernel_ctrl *  pl_kernel_ctrl;   // Kernel's
    control structure
typedef struct _pl_kernel_group * pl_kernel_group;  // Kernel's group
     size structure
typedef struct _pl_kernel_arg *   pl_kernel_arg;    // Kernel's
    argument structure
//   Memory typedefs
typedef struct _pl_memory *      pl_memory;       // Memory's structure
typedef struct _pl_buffer *      pl_buffer;       // Buffer's structure


// Kernel Group Struct
//   Information about group size of the kernel
struct _pl_kernel_group {
  // Group id
  pl_u32 id_x;         // Group id x address offset
  pl_u32 id_y;         // Group id y address offset
  pl_u32 id_z;         // Group id z address offset
  // Group offset
  pl_u32 offset_x;       // Global offset x address offset
  pl_u32 offset_y;       // Global offset y address offset
  pl_u32 offset_z;       // Global offset z address offset
};

// Kernel Control Struct
//   Information about control of the kernel
struct _pl_kernel_ctrl {
  // has info
  pl_bool hasCtrl;       // has Control offset
  pl_bool hasGIE;        // has Global Interrupt Enable offset
  pl_bool hasIER;        // has IP Interrupt Enable offset
  pl_bool hasISR;        // has IP Interrupt Status offset
  // info data
  pl_u32 ctrl;         // Control offset
  pl_u32 GIE;          // Global Interrupt Enable offset
  pl_u32 IER;          // IP Interrupt Enable offset
  pl_u32 ISR;           // IP Interrupt Status offset
  // Pointers
  pl_kernel kernel;      // Pointer to kernel data
};

// Kernel Arguments Struct
//   Information a kernel argument
struct _pl_kernel_arg {
  // Info data
  pl_bool isValid;       // If it is valid
  pl_u32 index;        // Index on the argument list
  char* name;          // Name of the argument
  // Mem data
  pl_u32 offset;         // Offset from the base address
  size_t size;         // Argument size in bytes
  // Pointer to kernel
  pl_kernel kernel;      // Pointer to kernel data
};

// Kernel Struct
//   Information of a kernel
struct _pl_kernel {
  // Info data
  pl_bool isValid;       // If it is valid
  pl_bool isReady;       // If it is initialized
  pl_u32 index;        // Index on the kernel list
  char* name;          // Name of the kernel
  // Mem data
  uint compute_units;      // Number of kernel compute units
  pl_u32* address;       // Base addresses of the kernel compute units
  void** address_ptr;      // Pointers to virtual addresses
  size_t workgroup[3];     // Kernel workgroup size
  size_t size;         // Mapped size in memory
  // Kernel data
  pl_bool hasGroup;      // If has group data
  pl_uint num_args;      // Number of arguments
  pl_kernel_arg* args;     // Arguments list
  pl_kernel_ctrl ctrl;     // Control data
  pl_kernel_group group;     // Group data
  // Pointers
  pl_bitfile bitfile;      // Pointer to bitfile data
};

// Memory Struct
//   Information of a memory
struct _pl_memory {
  // Info data
  pl_bool isValid;       // If it is valid
  pl_bool isReady;       // If it is initialized
  pl_u32 index;        // Index on the memory list
  char* name;          // Name of the memory
  // Mem data
  pl_u32 address;        // Base address of the memory
  void* address_ptr;       // Pointer to virtual address
  size_t size;         // memory size in bytes
  size_t stack_offset;     // Stack offset from memory start
  // Pointers
  pl_bitfile bitfile;      // Pointer to bitfile data
};

// Bitfile Struct
//   Information of a bitfile
struct _pl_bitfile {
  // Info data
  pl_bool isReady;       // Initialized flag
  char* name;          // Name of the bitfile
  char* path;          // Path to the bitfile
  size_t programedID;      // If this id match the
  // Kernel data
  pl_uint num_kernels;     // Number of kernels
  pl_kernel* kernels;      // Kernels list
  // Memory data
  pl_uint num_memories;    // Number of memories
  pl_memory* memories;     // Memories list
};

// Buffer struct
//   Information of a buffer
struct _pl_buffer {
  // Mem data
  pl_memory memory;      // Pointer to memory data
  pl_u32 address;        // Base address of the memory
  void* address_ptr;       // Pointer to virtual address
  size_t size;         // memory size in bytes
  void* arg;           // Pointer to address
};



// Define Flags
// General Flags
#define PL_SUCCESS 0
#define PL_FAILED 1
// Init Flags
#define PL_XDEVCFG_NOT_FOUND 111
#define PL_XDEVCFG_NO_WRITE_PERMISSION 112
#define PL_XDEVCFG_WRITE_FAILED 113
#define PL_XDEVCFG_PROGRAM_FAILED 113
#define PL_OPEN_DEVMEM_FAILED 121
// Device Flags
#define PL_NOT_PROGRAMED 411
#define PL_NOT_VALID 412
#define PL_NOT_READY 413
#define PL_NOT_FOUND 421
#define PL_MEMORY_MAP_FAILED 431
#define PL_MEMORY_UNMAP_FAILED 432
// Memory Flags
#define PL_NOT_ENOUGH_MEMORY 211
#define PL_NULL_POINTER 212
// File Flags
```

```
#define PL_FILE_NOT_FOUND 311
#define PL_FILE_OPEN_FAILED 312
#define PL_FILE_PERMISSIONS_ERROR 313
#define PL_FILE_INVALID_FORMAT 314

// Enable/Disable flags
#define PL_ENABLE 0x1
#define PL_DISABLE 0x0
// True/False flags
#define PL_TRUE 0x1
#define PL_FALSE 0x0

// Define Masks
typedef uint32_t pl_mask;
// Mask check macros
#define pl_match_mask(var,mark) ((var) & (mask) == (mask))
// Bitfile init masks
#define PL_BITFILE_INIT_NONE 0x0
#define PL_BITFILE_INIT_ALL 0x1
#define PL_BITFILE_INIT_KERNELS 0x10
#define PL_BITFILE_INIT_MEMORIES 0x100



// Define Address Macros
#define PL_Write_Register(address, offset, data) \
    *(volatile pl_u32*)((address) + (offset)) = (pl_u32)(data)
#define PL_Read_Register(address, offset) \
    *(volatile pl_u32*)((address) + (offset))



// Function Prototypes
//    Main library's functions

// Init Device
//    Prepare library and programmable logic
pl_int plInitDevice ();

// Get Bitfile Info
//    Load a bitfile's informations from json
pl_int plGetBitfileInfo (pl_bitfile* bitfile, const char * path,
    const char * name);

// Check if bitfile is programed
//    Match programed id to check if this is the last programmed
    bitfile
pl_bool plIsProgramed (pl_bitfile bitfile);
// Program PL using bitfile info
//    Program a bitfile to the programmable logic
pl_int plProgram (pl_bitfile bitfile);

// Initialize a bitfile that is programmed on pl
pl_int plBitfileInitialize (pl_bitfile bitfile, pl_mask mask);
// Initialize a bitfile's kernel that is programmed on pl
```

```
pl_int plKernelInitialize (pl_kernel kernel);
// Initialize a bitfile's memory that is programmed on pl
pl_int plMemoryInitialize (pl_memory memory);

// Release a bitfile
pl_int plBitfileRelease (pl_bitfile bitfile);
// Release a kernel
pl_int plKernelRelease (pl_kernel kernel);
// Release a memory
pl_int plMemoryRelease (pl_memory memory);

// Create a buffer
pl_buffer plCreateBuffer (pl_memory memory, size_t size, pl_int *
    errcode_ret);
// Release a buffer
pl_int plReleaseBuffer (pl_buffer buffer);
// Clear memory
pl_int plClearMemory (pl_memory memory);

// Write buffer
pl_int plWriteBuffer (pl_buffer buffer, size_t offset, size_t size,
    const void *ptr);
// Read buffer
pl_int plReadBuffer (pl_buffer buffer, size_t offset, size_t size,
    void *ptr);

// Get kernel methods
pl_kernel plGetKernelByIndex (pl_bitfile bitfile, size_t index);
pl_kernel plGetKernelByName (pl_bitfile bitfile, char* name);
// Get memory methods
pl_memory plGetSysMemoryByIndex (size_t index);
pl_memory plGetMemoryByIndex (pl_bitfile bitfile, size_t index);
pl_memory plGetMemoryByName (pl_bitfile bitfile, char* name);

// Kernel Set Global Interrupts
pl_int plSetKernelGlobalInterrupt (pl_kernel kernel, pl_bool type);
pl_int plSetKernelComputeUnitGlobalInterrupt (pl_kernel kernel,
    size_t index, pl_bool type);
// Kernel Set IP Interrupts
pl_int plSetKernelInterrupt (pl_kernel kernel, pl_bool type, pl_mask
    mask);
pl_int plSetKernelComputeUnitInterrupt (pl_kernel kernel, size_t
    index, pl_bool type, pl_mask mask);

// Kernel Set Argument
pl_int plSetKernelArg (pl_kernel kernel, pl_uint arg_index, size_t
    arg_size, const void *arg_value);
pl_int plSetKernelArgByName (pl_kernel kernel, const char *name,
    size_t arg_size, const void *arg_value);
pl_int plSetKernelComputeUnitArg (pl_kernel kernel, size_t index,
    pl_uint arg_index, size_t arg_size, const void *arg_value);
pl_int plSetKernelComputeUnitArgByName (pl_kernel kernel, size_t
    index, const char *name, size_t arg_size, const void *arg_value);
// Kernel Set Group ID
```

```
pl_int plSetKernelGroupIds (pl_kernel kernel, pl_int x, pl_int y,
    pl_int z);
pl_int plSetKernelComputeUnitGroupIds (pl_kernel kernel, size_t index
    , pl_int x, pl_int y, pl_int z);
// Kernel Set Global Offset
pl_int plSetKernelGlobalOffsets (pl_kernel kernel, pl_int x, pl_int y
    , pl_int z);
pl_int plSetKernelComputeUnitGlobalOffsets (pl_kernel kernel, size_t
    index, pl_int x, pl_int y, pl_int z);

// Kernel Run
pl_int plRunTask (pl_kernel kernel, pl_bool blocking);
pl_int plRunKernelComputeUnit (pl_kernel kernel, size_t index,
    pl_bool blocking);
pl_int plRunNDRangeKernel (pl_kernel kernel, pl_uint work_dim, const
    size_t *global_work_offset, const size_t *global_work_size,
    pl_bool blocking);
// Wait Kernel Run
pl_int plWaitTask (pl_kernel kernel);
pl_int plWaitKernelComputeUnit (pl_kernel kernel, size_t index);
pl_int plWaitNDRangeKernel (pl_kernel kernel);


// Load string from file
char* plToolsLoadFileContents(char *filepath);

// Copy file content to other file
pl_int plToolsCopyFileContents(char *source, char* target);


#ifdef __cplusplus
}
#endif

#endif /* SIMPLEPL_H */
```

Listing C.1: SimplePL header file

C.2 SimplePL source

```
/*
 * Copyright Grammatopoulos Athanasios Vasileios
 * agrammatopoulos@isc.tuc.gr
 * Technical University of Crete
 *
 * SimplePL Library
 * version 1.2.1
 */

// Include Header
#include "simplePL.h"
#include "tiny-json.h"

```

```

// System memory struct
typedef struct {
  // System's memory objects
  pl_memory mem;
  // fd pointer to the /dev/buffer
  int fd;

  // Paths
  char path_buffer[255];
  char path_info[255];
  char path_phys_addr[255];
} pllib_sys_memory;

// Basic pl data struct
typedef struct {
  // A fd pointer to the /dev/mem
  int dev_mem_fd;
  // Save system's page size
  int page_size;

  // If lib is ready for use
  int isReady;
  // If PL is programmed
  int isProgramed;
  // A simple increment to identify
  // which bitfile is currently
  // programed on the PL
  size_t programedID;

  // System's memory objects
  pllib_sys_memory sysmem[4];

  // List of buffers
  pl_buffer* buffers;
  uint buffers_stack;
} pllib_device_struct;
// Initialise basic data
static pllib_device_struct pllib_device = {
  .dev_mem_fd = 0,
  .page_size = 0,

  .isReady = PL_FALSE,
  .isProgramed = PL_FALSE,
  .programedID = 1
};

#define PL_MAX_BUFFERS 32



// Memory List functions
static void plBufferList_listBuffer (const pl_buffer buffer) {
  for (int i = 0; i < PL_MAX_BUFFERS; ++i) {
    if (pllib_device.buffers[i] == NULL) {
      pllib_device.buffers[i] = buffer;
      if(pllib_device.buffers_stack <= i){
        pllib_device.buffers_stack = i + 1;
      }
      return;
    }
  }
}
static void plBufferList_unlistBuffer (const pl_buffer buffer) {
  for (int i = 0; i < PL_MAX_BUFFERS; ++i) {
    if (pllib_device.buffers[i] == buffer) {
      pllib_device.buffers[i] = NULL;
      if(pllib_device.buffers_stack == i + 1){
        pllib_device.buffers_stack--;
      }
      return;
    }
  }
}
static pl_bool plBufferList_isBuffer (pl_u32 address) {
  for (int i = 0; i < PL_MAX_BUFFERS && i < pllib_device.
    buffers_stack; ++i) {
    if (address == (pl_u32) pllib_device.buffers[i]) {
      return PL_TRUE;
    }
  }
  return PL_FALSE;
}

// Init sys buffers
//   Get the corresponding udmabuf<index> buffer
//   Initialize it's info and variables
//   Link it on a pl_memory object
//   Map required pointers
static void plInitDevice_initSysMemory (int index, pllib_sys_memory*
    sys_memory) {
  // Allocate memory
  sys_memory->mem = (pl_memory) malloc(4 * sizeof(struct _pl_memory))
    ;
  // Temp pointer to memory
  pl_memory memory = sys_memory->mem;

  // Init values
  memory->isValid = PL_FALSE;
  memory->name = NULL;
  memory->address = 0;
  memory->size = 0;
  memory->bitfile = NULL;
  memory->address_ptr = NULL;
  memory->stack_offset = 0;

  // Create paths
  sprintf(sys_memory->path_buffer,  "/dev/udmabuf%d", index);
  sprintf(sys_memory->path_info,    "/sys/class/udmabuf/udmabuf%d/",
    index);
  sprintf(sys_memory->path_phys_addr, "/sys/class/udmabuf/udmabuf%d/
    phys_addr", index);

  // Check if exists
  if (access(sys_memory->path_buffer, F_OK) == -1 ) {
    // Error file dont exists
    return;
  }
  // Open and mmap buffer
  sys_memory->fd = open(sys_memory->path_buffer, O_RDWR | O_SYNC);
  if (sys_memory->fd == -1) {
    // Error failed to open file
    return;
  }
  // Map kernel base address
  memory->address_ptr = mmap(NULL, 0x200000, PROT_READ|PROT_WRITE,
    MAP_SHARED, sys_memory->fd, 0);

  // Parse name
  memory->name = strdup(sys_memory->path_buffer);
  // Parse address
  char* phys_addr = plToolsLoadFileContents(sys_memory->
    path_phys_addr);
  memory->address = (int) strtol(phys_addr, NULL, 0);
  free(phys_addr);
  // Parse size
  memory->size = 0x200000;

  // Set as valid
  memory->isValid = PL_TRUE;
  // Set as ready
  memory->isReady = PL_TRUE;
}

// Init Device
//   Initialize library's variables
//   Map pointer to mmap and xdevcfg
//   Initialize system buffers
pl_int plInitDevice () {
  // If already initialized
  if (pllib_device.isReady == PL_TRUE) {
    return PL_SUCCESS;
  }

  // Check if xdevcfg exist
  if (access("/dev/xdevcfg", F_OK) == -1 ) {
    return PL_XDEVCFG_NOT_FOUND;
  }

  // Check if xdevcfg can be written
  if (access("/dev/xdevcfg", W_OK) == -1 ) {
    return PL_XDEVCFG_NO_WRITE_PERMISSION;
  }

  // Open /dev/mem
  pllib_device.dev_mem_fd = open("/dev/mem", O_RDWR | O_SYNC);
  if (pllib_device.dev_mem_fd == -1) {
    return PL_OPEN_DEVMEM_FAILED;
  }

  // Allocate list of buffers
  pllib_device.buffers_stack = 0;
  pllib_device.buffers = (pl_buffer*) malloc(PL_MAX_BUFFERS * sizeof(
    pl_buffer));
  for (int i = 0; i < PL_MAX_BUFFERS; ++i) {
    pllib_device.buffers[i] = NULL;
  }

  // Get sys buffers
  for (int i = 0; i < 4; ++i){
    plInitDevice_initSysMemory(i, &(pllib_device.sysmem[i]));
  }

  // Save page size
  pllib_device.page_size = getpagesize();

  // Set as initialized
  pllib_device.isReady = PL_TRUE;

  // Return
  return PL_SUCCESS;
}



// Get Bitfile Info - Parse Kernel Group
//   Parse JSON with data about kernel group offsets
//   Get X,Y,Z id offsets
//   Get X,Y,Z offset offsets
static pl_kernel_group plGetBitfileInfo_parseKernelGroup(json_t const
    * data) {
  // Get properties
  json_t const* id_x_field = json_getProperty(data, "id_x");
  json_t const* id_y_field = json_getProperty(data, "id_y");
  json_t const* id_z_field = json_getProperty(data, "id_z");
  json_t const* offset_x_field = json_getProperty(data, "offset_x");
  json_t const* offset_y_field = json_getProperty(data, "offset_y");
  json_t const* offset_z_field = json_getProperty(data, "offset_z");

  // Validate properties
  if (
    id_x_field == NULL || json_getType(id_x_field) != JSON_TEXT ||
    id_y_field == NULL || json_getType(id_y_field) != JSON_TEXT ||
    id_z_field == NULL || json_getType(id_z_field) != JSON_TEXT ||
    offset_x_field == NULL || json_getType(offset_x_field) !=
   JSON_TEXT ||
    offset_y_field == NULL || json_getType(offset_y_field) !=
   JSON_TEXT ||
    offset_z_field == NULL || json_getType(offset_z_field) !=
   JSON_TEXT
  ) {
    return NULL;
  }

  // Allocate memory
  pl_kernel_group group = (pl_kernel_group) malloc(sizeof(struct
    _pl_kernel_group));

  // Set data
  group->id_x = (int)strtol(json_getValue(id_x_field), NULL, 0);
  group->id_y = (int)strtol(json_getValue(id_y_field), NULL, 0);
  group->id_z = (int)strtol(json_getValue(id_z_field), NULL, 0);
  group->offset_x = (int)strtol(json_getValue(offset_x_field), NULL,
    0);
  group->offset_y = (int)strtol(json_getValue(offset_y_field), NULL,
    0);
  group->offset_z = (int)strtol(json_getValue(offset_z_field), NULL,
    0);

  // Return
  return group;
}

// Get Bitfile Info - Parse Kernel Control
//   Parse JSON with data about kernel control offsets
//   Get control offset
//   Get interrupts offset
static void plGetBitfileInfo_parseKernelCtrl (json_t const* data,
    pl_kernel_ctrl ctrl) {
  // Get properties
  json_t const* ctrlfield = json_getProperty(data, "ctrl");
  json_t const* giefield = json_getProperty(data, "gie");
  json_t const* ierfield = json_getProperty(data, "ier");
  json_t const* isrfield = json_getProperty(data, "isr");

  // Init values
  ctrl->hasCtrl = PL_TRUE;
  ctrl->ctrl = 0x0;
  ctrl->hasGIE = PL_FALSE;
  ctrl->GIE = 0x0;
  ctrl->hasIER = PL_FALSE;
  ctrl->IER = 0x0;
  ctrl->hasISR = PL_FALSE;
  ctrl->ISR = 0x0;

  // Parse control
  if (ctrlfield != NULL && json_getType(ctrlfield) == JSON_TEXT) {
    ctrl->hasCtrl = PL_TRUE;
    ctrl->ctrl = (int)strtol(json_getValue(ctrlfield), NULL, 0);
  }
  // Parse Global Interrupt Enable
  if (giefield != NULL && json_getType(giefield) == JSON_TEXT) {
    ctrl->hasGIE = PL_TRUE;
    ctrl->GIE = (int)strtol(json_getValue(giefield), NULL, 0);
  }
  // Parse IP Interrupt Enable
  if (ierfield != NULL && json_getType(ierfield) == JSON_TEXT) {
    ctrl->hasIER = PL_TRUE;
    ctrl->IER = (int)strtol(json_getValue(ierfield), NULL, 0);
  }
  // Parse IP Interrupt Status
  if (isrfield != NULL && json_getType(isrfield) == JSON_TEXT) {
    ctrl->hasISR = PL_TRUE;
    ctrl->ISR = (int)strtol(json_getValue(isrfield), NULL, 0);
  }
}

// Get Bitfile Info - Parse Kernel Argument
//   Parse JSON with data about kernel control offsets
//   Get control offset
static void plGetBitfileInfo_parseKernelArg (json_t const* data,
    pl_kernel_arg arg) {
  // Get properties
  json_t const* namefield = json_getProperty(data, "name");
  json_t const* offsetfield = json_getProperty(data, "offset");
  json_t const* sizefield = json_getProperty(data, "size");

  // Init values
  arg->isValid = PL_FALSE;
  arg->name = NULL;
  arg->offset = 0;
  arg->size = 0;

  // Validate properties
  if (
    namefield == NULL || json_getType(namefield) != JSON_TEXT ||
    offsetfield == NULL || json_getType(offsetfield) != JSON_TEXT ||
    (sizefield != NULL && json_getType(sizefield) != JSON_INTEGER)
  ) {
    return;
  }

  // Parse name
  arg->name = strdup(json_getValue(namefield));
  // Parse offset
  arg->offset = (int)strtol(json_getValue(offsetfield), NULL, 0);
  // Parse size
  if (sizefield != NULL) {
    arg->size = json_getInteger(sizefield);
  }
  // Set as valid
  arg->isValid = PL_TRUE;
}

// Get Bitfile Info - Parse Kernel Arguments List
//   Parse JSON with data about kernel arguments
static pl_kernel_arg* plGetBitfileInfo_parseKernelArgs (json_t const*
    list, int* length) {
  // Init variables
  pl_u32 index = 0;
  json_t const* item;

  // Count kernel arguments
  for (item = json_getChild(list); item != 0; item = json_getSibling(
    item)) {
    if (JSON_OBJ == json_getType(item)){
      index ++;
    }
  }
  *length = index;

  // Allocate memory for the kernel arguments
  pl_kernel_arg* args = (pl_kernel_arg*) malloc((*length) * sizeof(
    pl_kernel_arg));

  // For each kernel argument
  index = 0;
  for (item = json_getChild(list); item != 0; item = json_getSibling(
    item)) {
    if (JSON_OBJ == json_getType(item)){
      // Parse kernel argument
      args[index] = (pl_kernel_arg) malloc(sizeof(struct
    _pl_kernel_arg));
      plGetBitfileInfo_parseKernelArg(item, args[index]);
      args[index]->index = index;
      index++;
    }
  }

  return args;
}

// Get Bitfile Info - Parse Kernel
//   Parse JSON with data about kernel
static void plGetBitfileInfo_parseKernel (json_t const* data,
    pl_kernel kernel) {
  // Get properties
  json_t const* namefield = json_getProperty(data, "name");
  json_t const* addressfield = json_getProperty(data, "address");
  json_t const* workgroupfield = json_getProperty(data, "workgroup");
  json_t const* sizefield = json_getProperty(data, "size");
  json_t const* controlfield = json_getProperty(data, "control");
  json_t const* groupfield = json_getProperty(data, "group");
  json_t const* argumentsList = json_getProperty(data, "arguments");

  // Init values
  kernel->isValid = PL_FALSE;
  kernel->name = NULL;

  kernel->compute_units = 0;
  kernel->address = NULL;
  kernel->address_ptr = NULL;
  kernel->workgroup[0] = 0;
  kernel->workgroup[1] = 0;
  kernel->workgroup[2] = 0;
  kernel->size = 0;

  kernel->hasGroup = PL_FALSE;
  kernel->num_args = 0;
  kernel->args = NULL;
  kernel->ctrl = NULL;
  kernel->group = NULL;

  kernel->bitfile = NULL;

  // Validate properties
  if (
    namefield == NULL || json_getType(namefield) != JSON_TEXT ||
    addressfield == NULL || json_getType(addressfield) != JSON_ARRAY
   ||
    workgroupfield == NULL || json_getType(workgroupfield) !=
  JSON_ARRAY ||
    sizefield == NULL || json_getType(sizefield) != JSON_INTEGER ||
    controlfield == NULL || json_getType(controlfield) != JSON_OBJ ||
    (groupfield != NULL && json_getType(groupfield) != JSON_OBJ) ||
    (argumentsList != NULL && json_getType(argumentsList) !=
  JSON_ARRAY)
  ) {
    return;
  }

  // Count compute_units (number of addresses)
  json_t const* item;
  for (item = json_getChild(addressfield); item != 0; item =
   json_getSibling(item)) {
    if (JSON_TEXT == json_getType(item)){
      kernel->compute_units ++;
    }
  }
  if (kernel->compute_units == 0) {
    return;
  }


  // Get kernel size
  int i = 0;
  for (item = json_getChild(workgroupfield); item != 0 && i < 3; item
    = json_getSibling(item)) {
    if (JSON_INTEGER == json_getType(item)){
      kernel->workgroup[i] = json_getInteger(item);
      if (kernel->workgroup[i] == 0) {
        return;
      }
      i++;
    }
  }

  // Parse controls
  kernel->ctrl = (pl_kernel_ctrl) malloc(sizeof(struct
   _pl_kernel_ctrl));
  plGetBitfileInfo_parseKernelCtrl(controlfield, kernel->ctrl);
  kernel->ctrl->kernel = kernel;
  // Parse group
  if (groupfield != NULL) {
    kernel->group = plGetBitfileInfo_parseKernelGroup(groupfield);
    if (kernel->group != NULL) {
      kernel->hasGroup = PL_TRUE;
    }
  }
  // Parse name
  kernel->name = strdup(json_getValue(namefield));
  // Parse address
  i = 0;
  kernel->address = (pl_u32*) malloc(kernel->compute_units * sizeof(
    pl_u32));
  kernel->address_ptr = (void**) malloc(kernel->compute_units *
    sizeof(void*));
  for (item = json_getChild(addressfield); item != 0; item =
    json_getSibling(item)) {
     if (JSON_TEXT == json_getType(item)){
      kernel->address[i] = (int)strtol(json_getValue(item), NULL, 0);
      kernel->address_ptr[i] = NULL;
      i++;
    }
  }
  // Parse size
  kernel->size = json_getInteger(sizefield);
  if (kernel->size == 0) kernel->size = pllib_device.page_size;
  // Parse arguments
  kernel->num_args = 0;
  if(argumentsList != NULL){
    kernel->args = plGetBitfileInfo_parseKernelArgs(argumentsList, &(
    kernel->num_args));
    for (int i = 0; i < kernel->num_args; ++i) {
      kernel->args[i]->kernel = (void *) kernel;
    }
  }
  // Set as valid
  kernel->isValid = PL_TRUE;
}

// Get Bitfile Info - Parse Kernels List
static pl_kernel* plGetBitfileInfo_parseKernels (json_t const* list,
    int* length) {
  // Init variables
  pl_u32 index = 0;
  json_t const* item;

  // Count kernels
  for (item = json_getChild(list); item != 0; item = json_getSibling(
    item)) {
     if (JSON_OBJ == json_getType(item)){
      index ++;
    }
  }
```

```c
  *length = index;

  // Allocate memory for the kernels
  pl_kernel* kernels = (pl_kernel*) malloc(index * sizeof(pl_kernel))
    ;

  // For each kernel
  index = 0;
  for (item = json_getChild(list); item != 0; item = json_getSibling(
    item)) {
    if (JSON_OBJ == json_getType(item)){
      // Parse kernel
      kernels[index] = (pl_kernel) malloc(sizeof(struct _pl_kernel));
      plGetBitfileInfo_parseKernel(item, kernels[index]);
      kernels[index]->index = index;
      index++;
    }
  }

  return kernels;
}

// Get Bitfile Info - Parse Memory
static void plGetBitfileInfo_parseMemory (json_t const* data,
    pl_memory memory) {
  // Get properties
  json_t const* namefield = json_getProperty(data, "name");
  json_t const* addressfield = json_getProperty(data, "address");
  json_t const* sizefield = json_getProperty(data, "size");
  json_t const* argumentsList = json_getProperty(data, "arguments");

  // Init values
  memory->isValid = PL_FALSE;
  memory->name = NULL;
  memory->address = 0;
  memory->size = 0;
  memory->bitfile = NULL;
  memory->address_ptr = NULL;
  memory->stack_offset = 0;

  // Validate properties
  if (
    namefield == NULL || json_getType(namefield) != JSON_TEXT ||
    addressfield == NULL || json_getType(addressfield) != JSON_TEXT
    ||
    sizefield == NULL || json_getType(sizefield) != JSON_INTEGER
  ) {
    return;
  }

  // Parse name
  memory->name = strdup(json_getValue(namefield));
  // Parse address
  memory->address = (int)strtol(json_getValue(addressfield), NULL, 0)
    ;
  // Parse size
  memory->size = json_getInteger(sizefield);
  if (memory->size == 0) memory->size = pllib_device.page_size;
  // Set as valid
  memory->isValid = PL_TRUE;
}

// Get Bitfile Info - Parse Memories List
static pl_memory* plGetBitfileInfo_parseMemories (json_t const* list ,
    int* length) {
  // Init variables
  pl_u32 index = 0;
  json_t const* item;

  // Count memories
  for (item = json_getChild(list); item != 0; item = json_getSibling(
   item)) {
    if (JSON_OBJ == json_getType(item)){
      index ++;
    }
  }
  *length = index;

  // Allocate memory for the memories
  pl_memory* memories = (pl_memory*) malloc(index * sizeof(pl_memory)
   );

  // For each memory
  index = 0;
  for (item = json_getChild(list); item != 0; item = json_getSibling(
   item)) {
    if (JSON_OBJ == json_getType(item)){
      // Parse memory
      memories[index] = (pl_memory) malloc(sizeof(struct _pl_memory))
   ;
      plGetBitfileInfo_parseMemory(item, memories[index]);
      memories[index]->index = index;
      index++;
    }
  }

  return memories;
}

// Get Bitfile Info
pl_int plGetBitfileInfo (pl_bitfile* bitfile, const char * path,
   const char * name) {
  // Check pointer
  if (bitfile == NULL) {
    return PL_NULL_POINTER;
  }

  // Get string lengths
  size_t len_path = strlen(path);
  size_t len_name = strlen(name);
  // Allocate memory for paths
  char* path_bitfile = malloc(len_path + len_name + 4 + 1);
  char* path_info = malloc(len_path + len_name + 5 + 1);
  // Construct paths
  strcpy(path_bitfile, path);
  strcat(path_bitfile, name);
  strcat(path_bitfile, ".bit");
  strcpy(path_info, path);
  strcat(path_info, name);
  strcat(path_info, ".json");

  // Check if files exists
  int error = 0;
  if (error == 0 && (access(path_bitfile, F_OK) == -1 || access(
    path_info, F_OK) == -1)) {
    error = PL_FILE_NOT_FOUND;
  }

  // Check file permissions
  if (error == 0 && (access(path_bitfile, R_OK) == -1 || access(
    path_info, R_OK) == -1)) {
    error = PL_FILE_PERMISSIONS_ERROR;
  }

  // If an error exist return it
  if (error != 0) {
    free(path_bitfile);
    free(path_info);
    return error;
  }

  // Read bitfile info
  char* info_string = plToolsLoadFileContents(path_info);
  free(path_info);
  if (info_string == NULL) {
    free(path_bitfile);
    return PL_NOT_ENOUGH_MEMORY;
  }

  // Parse bitfile info
  json_t pool[512];
  json_t const* root = json_create(info_string, pool, 512);
  if (root == NULL) {
    free(path_bitfile);
    free(info_string);
    return PL_FILE_INVALID_FORMAT;
  }

  // Try to get data
  json_t const* namefield = json_getProperty(root, "name");
  json_t const* kernelsList = json_getProperty(root, "kernels");
  json_t const* memoriesList = json_getProperty(root, "memories");

  // Check if data exist
  if (
      namefield == NULL || json_getType(namefield) != JSON_TEXT ||
      (kernelsList != NULL && json_getType(kernelsList) != JSON_ARRAY)
      ||
      (memoriesList != NULL && json_getType(memoriesList) != JSON_ARRAY
      )
    ) {
      free(path_bitfile);
      free(info_string);
      return PL_FILE_INVALID_FORMAT;
    }

    // Create a bitfile object
    pl_bitfile b;
    b = (pl_bitfile) malloc(sizeof(struct _pl_bitfile));
    b->name = NULL;
    b->path = path_bitfile;
    b->num_kernels = 0;
    b->kernels = NULL;
    b->num_memories = 0;
    b->memories = NULL;
    b->programedID = 0;

    // Set name
    b->name = strdup(json_getValue(namefield));

    // Parse kernels
    int i = 0;
    if (kernelsList != NULL) {
      b->kernels = plGetBitfileInfo_parseKernels(kernelsList, &(b->
      num_kernels));
      for (i = 0; i < b->num_kernels; ++i) {
        b->kernels[i]->bitfile = (void *) b;
      }
    }

    // Parse memories
    if (memoriesList != NULL) {
      b->memories = plGetBitfileInfo_parseMemories(memoriesList, &(b->
      num_memories));
      for (i = 0; i < b->num_memories; ++i) {
        b->memories[i]->bitfile = (void *) b;
      }
    }

    // Return data
    *bitfile = b;
    return PL_SUCCESS;
}



// Check if bitfile is programed
pl_bool plIsProgramed (pl_bitfile bitfile) {
    // If this is not the bitfile programmed on pl
    if (pllib_device.programedID != bitfile->programedID) {
```

```
    return PL_FALSE;
  }
  return PL_TRUE;
}

// Program PL using bitfile info
pl_int plProgram (pl_bitfile bitfile) {
  // Check pointer
  if (bitfile == NULL) {
    return PL_NULL_POINTER;
  }

  // Program device using xdevcfg
  pl_int result = plToolsCopyFileContents(bitfile->path, "/dev/
    xdevcfg");
  if (result != PL_SUCCESS) {
    return PL_XDEVCFG_WRITE_FAILED;
  }

  // Check if program done
  char * prog_done = plToolsLoadFileContents("/sys/class/xdevcfg/
    xdevcfg/device/prog_done");
  if (prog_done == NULL || prog_done[0] != '1') {
    pllib_device.isProgramed = PL_FALSE;
    return PL_XDEVCFG_PROGRAM_FAILED;
  }

  // Set as programmed
  pllib_device.isProgramed = PL_TRUE;

  // Increase programedID and set new
  pllib_device.programedID++;
  bitfile->programedID = pllib_device.programedID;

  // Return
  return PL_SUCCESS;
}



// Initialize a bitfile that is programmed on pl
pl_int plBitfileInitialize (pl_bitfile bitfile, pl_mask mask) {
  // Check pointer
  if (bitfile == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != bitfile->programedID) {
    return PL_NOT_PROGRAMED;
  }

  // If already initialized
  if (bitfile->isReady == PL_TRUE) {
    return PL_SUCCESS;
  }

  // Prepare variables
  int i = 0;
  size_t address;
  size_t offset;

  // If init kernels
  if (pl_match_mask(mask, PL_BITFILE_INIT_ALL) || pl_match_mask(mask,
     PL_BITFILE_INIT_KERNELS)) {
    // Map kernels physical addresses to virtuals
    for (i = 0; i < bitfile->num_kernels; ++i) {
      plKernelInitialize(bitfile->kernels[i]);
    }
  }

  // If init memories
  if (pl_match_mask(mask, PL_BITFILE_INIT_ALL) || pl_match_mask(mask,
     PL_BITFILE_INIT_MEMORIES)) {
    // Map memories physical addresses to virtuals
    for (i = 0; i < bitfile->num_memories; ++i) {
      plMemoryInitialize(bitfile->memories[i]);
    }
  }

  // Set as ready
  bitfile->isReady = PL_TRUE;

  // Return success
  return PL_SUCCESS;
}

// Initialize a bitfile's kernel that is programmed on pl
pl_int plKernelInitialize (pl_kernel kernel) {
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  // Get bitfile
  pl_bitfile bitfile = (pl_bitfile) kernel->bitfile;

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != bitfile->programedID) {
    return PL_NOT_PROGRAMED;
  }

  // If not valid
  if (kernel->isValid != PL_TRUE) {
    return PL_NOT_VALID;
  }

  // If already initialized
  if (kernel->isReady == PL_TRUE) {
    return PL_SUCCESS;
  }

  int i;
  size_t address;
  size_t offset;

  // For each unit
  for (i = 0; i < kernel->compute_units; ++i) {
    // Calculate address and offset
    address = (kernel->address[i] & (~(pllib_device.page_size - 1)));
    offset = (kernel->address[i] - address);

    // Map kernel base address
    kernel->address_ptr[i] = mmap(NULL, kernel->size + offset,
   PROT_READ|PROT_WRITE, MAP_SHARED, pllib_device.dev_mem_fd, address
   );

    // Check for errors
    if (kernel->address_ptr[i] == MAP_FAILED) {
      kernel->address_ptr[i] = NULL;
      return PL_MEMORY_MAP_FAILED;
    }

    // Move pointer on offset
    kernel->address_ptr[i] += offset;
  }

  // Set as ready
  kernel->isReady = PL_TRUE;

  // Return success
  return PL_SUCCESS;
}

// Initialize a bitfile's memory that is programmed on pl
pl_int plMemoryInitialize (pl_memory memory) {
  // Check pointer
  if (memory == NULL) {
    return PL_NULL_POINTER;
  }

  // Get bitfile
  pl_bitfile bitfile = (pl_bitfile) memory->bitfile;

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != bitfile->programedID) {
    return PL_NOT_PROGRAMED;
  }

  // If not valid
  if (memory->isValid != PL_TRUE) {
    return PL_NOT_VALID;
  }

  // If already initialized
  if (memory->isReady == PL_TRUE) {
    return PL_SUCCESS;
  }

  // Calculate address and offset
  size_t address = (memory->address & (~(pllib_device.page_size - 1))
    );
  size_t offset = (memory->address - address);

  // Map memory base address
  memory->address_ptr = mmap(NULL, memory->size + offset, PROT_READ|
    PROT_WRITE, MAP_SHARED, pllib_device.dev_mem_fd, address);

  // Check for errors
  if (memory->address_ptr == MAP_FAILED) {
    memory->address_ptr = NULL;
    return PL_MEMORY_MAP_FAILED;
  }

  // Move pointer on offset
  memory->address_ptr += offset;
  // Set as ready
  memory->isReady = PL_TRUE;

  // Return success
  return PL_SUCCESS;
}



// Release a bitfile
pl_int plBitfileRelease (pl_bitfile bitfile) {
  // Check pointer
  if (bitfile == NULL) {
    return PL_NULL_POINTER;
  }

  int i;

  // For each kernel
  for (i = 0; i < bitfile->num_kernels; ++i) {
    plKernelRelease (bitfile->kernels[i]);
    free(bitfile->kernels[i]);
  }
  free(bitfile->kernels);

  // For each memory
  for (i = 0; i < bitfile->num_memories; ++i) {
    plMemoryRelease (bitfile->memories[i]);
    free(bitfile->memories[i]);
  }
  free(bitfile->memories);

  // Release strings
  free(bitfile->name);
  free(bitfile->path);

  // Set as not ready
  bitfile->isReady = PL_FALSE;
  // Set as not programed
  bitfile->programedID = 0;

  // Release bitfile
  free(bitfile);

  // Return success
  return PL_SUCCESS;
}

// Release a kernel
pl_int plKernelRelease (pl_kernel kernel) {
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  int i;
  size_t address;
  size_t offset;

  // For each unit
  for (i = 0; i < kernel->compute_units; ++i) {
    // Calculate address and offset
    address = (((size_t) kernel->address_ptr[i]) & (~(pllib_device.
   page_size - 1)));
    offset = (((size_t) kernel->address_ptr[i]) - address);

    // Un map kernel
    if (munmap((void *) kernel->address_ptr[i], kernel->size + offset
   ) != 0) {
      return PL_MEMORY_UNMAP_FAILED;
    }
  }

  // Release name
  free(kernel->name);

  // For each argument
  for (int i = 0; i < kernel->num_args; ++i) {
    // Release name
    free(kernel->args[i]->name);
    // Set as not valid
    kernel->args[i]->isValid = PL_FALSE;
  }
  // Free arguments
  free(kernel->args);

  // Free control
  free(kernel->ctrl);

  // Disable flags
  kernel->isValid = PL_FALSE;
  kernel->isReady = PL_FALSE;

  // Return success
  return PL_SUCCESS;
}

// Release a memory
pl_int plMemoryRelease (pl_memory memory) {
  // Check pointer
  if (memory == NULL) {
    return PL_NULL_POINTER;
  }

  // Calculate address and offset
  size_t address = (((size_t) memory->address_ptr) & (~(pllib_device.
    page_size - 1)));
  size_t offset = (((size_t) memory->address_ptr) - address);

  // Un map memory
  if (munmap((void *) memory->address_ptr, memory->size + offset) !=
    0) {
    return PL_MEMORY_UNMAP_FAILED;
  }

  // Release name
  free(memory->name);

  // Disable flags
  memory->isValid = PL_FALSE;
  memory->isReady = PL_FALSE;

  // Return success
  return PL_SUCCESS;
}



// Create a buffer
pl_buffer plCreateBuffer (pl_memory memory, size_t size, pl_int *
    errcode_ret) {
  // Check if memory is not ready
  if (memory->isReady != PL_TRUE) {
    *errcode_ret = PL_NOT_READY;
    return NULL;
  }

  // Make size multiple of 4
  if (size % 4) {
    size = size + (4 - size % 4);
  }

  // Check if enough memory
  if ((memory->stack_offset + size) > memory->size) {
    *errcode_ret = PL_NOT_ENOUGH_MEMORY;
    return NULL;
  }

  pl_buffer buffer;
  // Allocate buffer memory
  buffer = (pl_buffer) malloc(sizeof(struct _pl_buffer));
  // Allocate memory
  buffer->address = memory->address + memory->stack_offset;
  buffer->address_ptr = memory->address_ptr + memory->stack_offset;
  // Move stack pointer
  memory->stack_offset += size;
  // Set size
  buffer->size = size;
  // Set memory
  buffer->memory = memory;
  // Set pointer to adderss
  buffer->arg = (void*) &(buffer->address);

  // TODO : buffer allocation on memory is not so good

  // Save buffer address
  plBufferList_listBuffer(buffer);

  // Return buffer
  *errcode_ret = PL_SUCCESS;
  return buffer;
}

// Release a buffer
pl_int plReleaseBuffer (pl_buffer buffer) {
  // Check pointer
  if (buffer == NULL) {
    return PL_NULL_POINTER;
  }

  // Delete buffer address
  plBufferList_unlistBuffer(buffer);

  // Clear buffer memory
  free(buffer);

  // Return success
  return PL_SUCCESS;
}

// Clear memory
pl_int plClearMemory (pl_memory memory) {
  // Check pointer
  if (memory == NULL) {
    return PL_NULL_POINTER;
  }

  // Reset stack offset
  memory->stack_offset = 0;
```

```

  // Return success
  return PL_SUCCESS;
}



// Write buffer
pl_int plWriteBuffer (pl_buffer buffer, size_t offset, size_t size,
    const void *ptr) {
  // Check pointer
  if (buffer == NULL || ptr == NULL) {
    return PL_NULL_POINTER;
  }

  // Calculate target
  char* b = buffer->address_ptr + offset;

  // Copy data to the buffer

  // Serial copy?
  //for (int i = 0; i < size; ++i) {
  //  b[i] = ((char*) ptr)[i];
  //}

  // Better let the compiler handle it
  memcpy(b, ptr, size);

  return PL_SUCCESS;
}

// Read buffer
pl_int plReadBuffer (pl_buffer buffer, size_t offset, size_t size,
    void *ptr) {
  // Check pointer
  if (buffer == NULL || ptr == NULL) {
    return PL_NULL_POINTER;
  }

  // Calculate target
  char* b = buffer->address_ptr + offset;

  // Copy data from the buffer

  // Serial copy?
  //for (int i = 0; i < size; ++i) {
  //  ((char*) ptr)[i] = b[i];
  //}

  // Better let the compiler handle it
  memcpy(ptr, b, size);

  return PL_SUCCESS;
}

```

```


// Get kernel methods
pl_kernel plGetKernelByIndex (pl_bitfile bitfile, size_t index) {
  // Check pointer
  if (bitfile == NULL) {
    return NULL;
  }

  // Check kernel number
  if (index < bitfile->num_kernels && index >= 0) {
    return bitfile->kernels[index];
  }

  // Not found
  return NULL;
}
pl_kernel plGetKernelByName (pl_bitfile bitfile, char* name) {
  // Check pointer
  if (bitfile == NULL) {
    return NULL;
  }

  // Search in kernels
  for (int i = 0; i < bitfile->num_kernels; ++i) {
    if (bitfile->kernels[i]->isValid == PL_TRUE && strcmp(bitfile->
   kernels[i]->name, name) == 0) {
      return bitfile->kernels[i];
    }
  }

  // Not found
  return NULL;
}

// Get system memory
pl_memory plGetSysMemoryByIndex (size_t index) {
  // Check memory number
  if (index >= 0 && index < 4) {
    return pllib_device.sysmem[index].mem;
  }

  // Not found
  return NULL;
}

// Get memory methods
pl_memory plGetMemoryByIndex (pl_bitfile bitfile, size_t index) {
  // Check pointer
  if (bitfile == NULL) {
    return NULL;
  }

  // Check memory number
  if (index < bitfile->num_memories && index >= 0) {
    return bitfile->memories[index];
  }

  // Not found
  return NULL;
}
pl_memory plGetMemoryByName (pl_bitfile bitfile, char* name) {
  // Check pointer
  if (bitfile == NULL) {
    return NULL;
  }

  // Search in memories
  for (int i = 0; i < bitfile->num_memories; ++i) {
    if (bitfile->memories[i]->isValid == PL_TRUE && strcmp(bitfile->
   memories[i]->name, name) == 0) {
      return bitfile->memories[i];
    }
  }

  // Not found
  return NULL;
}



// Interrupts
// Kernel Set Global Interrupts
pl_int plSetKernelGlobalInterrupt (pl_kernel kernel, pl_bool type) {
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->
   programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check data
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }
  if (kernel->ctrl->hasGIE != PL_TRUE) {
    return PL_NOT_VALID;
  }

  int index;

  // Set register
  if (type == PL_ENABLE) {
    // Enable
    for (index = 0; index < kernel->compute_units; ++index)
      PL_Write_Register(kernel->address_ptr[index], kernel->ctrl->GIE
    , 1);
  }
  else if (type == PL_DISABLE) {
    // Disable
    for (index = 0; index < kernel->compute_units; ++index)
      PL_Write_Register(kernel->address_ptr[index], kernel->ctrl->GIE
    , 0);
  }
  else {
    // Unknown type
    return PL_FAILED;
  }

  // Return success
  return PL_SUCCESS;
}
pl_int plSetKernelComputeUnitGlobalInterrupt (pl_kernel kernel,
    size_t index, pl_bool type) {
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->
    programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check data
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }
  if (kernel->ctrl->hasGIE != PL_TRUE) {
    return PL_NOT_VALID;
  }
  if (kernel->compute_units <= index) {
    return PL_NOT_FOUND;
  }

  int i;

  // Set register
  if (type == PL_ENABLE) {
    // Enable
    PL_Write_Register(kernel->address_ptr[index], kernel->ctrl->GIE,
    1);
  }
  else if (type == PL_DISABLE) {
    // Disable
    PL_Write_Register(kernel->address_ptr[index], kernel->ctrl->GIE,
    0);
  }
  else {
    // Unknown type
    return PL_FAILED;
  }

  // Return success
  return PL_SUCCESS;
}

// Kernel Set IP Interrupts
pl_int plSetKernelInterrupt (pl_kernel kernel, pl_bool type, pl_mask
    mask) {
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->
    programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check data
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }
  if (kernel->ctrl->hasGIE != PL_TRUE) {
    return PL_NOT_VALID;
  }

  // Check type
  if (type != PL_ENABLE && type != PL_DISABLE) {
    return PL_FAILED;
  }

  pl_u32 value;

  // For each unit
  for (int index = 0; index < kernel->compute_units; ++index){
    // Get register
    value = PL_Read_Register(kernel->address_ptr[index], kernel->ctrl
    ->IER);

    // Set register
    if (type == PL_ENABLE) {
      // Enable
      PL_Write_Register(kernel->address_ptr[index], kernel->ctrl->IER
    , value | mask);
    }
    else {
      // Disable
      PL_Write_Register(kernel->address_ptr[index], kernel->ctrl->IER
    , value & (~mask));
    }
  }

  // Return success
  return PL_SUCCESS;
}
pl_int plSetKernelComputeUnitInterrupt (pl_kernel kernel, size_t
    index, pl_bool type, pl_mask mask) {
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->
    programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check data
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }
  if (kernel->ctrl->hasGIE != PL_TRUE) {
    return PL_NOT_VALID;
  }
  if (kernel->compute_units <= index) {
    return PL_NOT_FOUND;
  }

  // Check type
  if (type != PL_ENABLE && type != PL_DISABLE) {
    return PL_FAILED;
  }

  pl_u32 value;

  // Get register
  value = PL_Read_Register(kernel->address_ptr[index], kernel->ctrl->
    IER);

  // Set register
  if (type == PL_ENABLE) {
    // Enable
    PL_Write_Register(kernel->address_ptr[index], kernel->ctrl->IER,
    value | mask);
  }
  else {
    // Disable
    PL_Write_Register(kernel->address_ptr[index], kernel->ctrl->IER,
    value & (~mask));
  }

  // Return success
  return PL_SUCCESS;
}

```

```


// Get argument index by name
static pl_uint plGetKernelArgIndexByName (pl_kernel kernel, const
    char *name) {
  // For each argument
  for (int i = 0; i < kernel->num_args; ++i) {
    if (kernel->args[i]->isValid == PL_TRUE && strcmp(kernel->args[i
    ]->name, name) == 0) {
      // Return index
      return i;
    }
  }

  // Return not found
  return kernel->num_args;
}

// Kernel Set Argument
pl_int plSetKernelArg (pl_kernel kernel, pl_uint arg_index, size_t
    arg_size, const void *arg_value) {
  // Check pointers
  if (kernel == NULL || arg_value == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->
    programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check kernel and argument
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }
  if (arg_index >= kernel->num_args) {
    return PL_NOT_FOUND;
  }
  if (kernel->args[arg_index]->isValid != PL_TRUE) {
    return PL_NOT_VALID;
  }

  // Value variable
  pl_u32* value = (pl_u32*) arg_value;
  // Check if buffer
  if (plBufferList_isBuffer((pl_u32) *((pl_buffer*) arg_value)) ==
    PL_TRUE) {
    value = (pl_u32*) (*((pl_buffer*) arg_value))->arg;
    arg_size = 0;
  }

  // Argument address variable
  void* arg;

  // For each unit
  for (int index = 0; index < kernel->compute_units; ++index) {
    // Get argument pointer
    arg = kernel->address_ptr[index] + kernel->args[arg_index]->
    offset;
    // Get size
    if (arg_size == 0) {
      arg_size = kernel->args[arg_index]->size;
    }

    // Write registers
    for (int offset = 0; offset < arg_size; offset += 4) {
      PL_Write_Register(arg, offset, *(value + offset));
    }
  }

  // Return success
  return PL_SUCCESS;
}
pl_int plSetKernelComputeUnitArg (pl_kernel kernel, size_t index,
    pl_uint arg_index, size_t arg_size, const void *arg_value) {
  // Check pointers
  if (kernel == NULL || arg_value == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->
    programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check kernel and argument
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }
  if (arg_index >= kernel->num_args) {
    return PL_NOT_FOUND;
  }
  if (kernel->args[arg_index]->isValid != PL_TRUE) {
    return PL_NOT_VALID;
  }
  // Check if compute unit exist
  if (kernel->compute_units <= index) {
    return PL_NOT_FOUND;
  }

  // Value variable
  pl_u32* value = (pl_u32*) arg_value;
  // Check if buffer
  if (plBufferList_isBuffer((pl_u32) *((pl_buffer*) arg_value)) ==
    PL_TRUE) {
    value = (pl_u32*) (*((pl_buffer*) arg_value))->arg;
    arg_size = 0;
  }

  // Argument address variable
  void* arg;

  // Get argument pointer
  arg = kernel->address_ptr[index] + kernel->args[arg_index]->offset;
  // Get size
  if (arg_size == 0) {
    arg_size = kernel->args[arg_index]->size;
  }

  // Write registers
  for (int offset = 0; offset < arg_size; offset += 4) {
    PL_Write_Register(arg, offset, *(value + offset));
  }

  // Return success
  return PL_SUCCESS;
}

// Kernel Set Argument
pl_int plSetKernelArgByName (pl_kernel kernel, const char *name,
    size_t arg_size, const void *arg_value) {
  // Get index by name
  pl_uint arg_index = plGetKernelArgIndexByName (kernel, name);
  // Check result
  if (arg_index == kernel->num_args) {
    return PL_NOT_FOUND;
  }
  // Return by index
  return plSetKernelArg (kernel, arg_index, arg_size, arg_value);
}
pl_int plSetKernelComputeUnitArgByName (pl_kernel kernel, size_t
    index, const char *name, size_t arg_size, const void *arg_value) {
  // Get index by name
  pl_uint arg_index = plGetKernelArgIndexByName (kernel, name);
  // Check result
  if (arg_index == kernel->num_args) {
    return PL_NOT_FOUND;
  }
  // Return by index
  return plSetKernelComputeUnitArg (kernel, index, arg_index,
    arg_size, arg_value);
}




// Kernel Set Group ID
pl_int plSetKernelGroupIds (pl_kernel kernel, pl_int x, pl_int y,
    pl_int z) {
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->
    programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check is kernel is ready
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }

  // Check if group data exist
  if (kernel->hasGroup != PL_TRUE) {
    return PL_NOT_FOUND;
  }

  // For each unit
  for (int index = 0; index < kernel->compute_units; ++index) {
    // Set Group id
    PL_Write_Register(kernel->address_ptr[index], kernel->group->id_x
    , x);
    PL_Write_Register(kernel->address_ptr[index], kernel->group->id_y
    , y);
    PL_Write_Register(kernel->address_ptr[index], kernel->group->id_z
    , z);
  }

  // Return success
  return PL_SUCCESS;
}
pl_int plSetKernelComputeUnitGroupIds (pl_kernel kernel, size_t index
    , pl_int x, pl_int y, pl_int z) {
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->
    programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check is kernel is ready
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }

  // Check if group data exist
  if (kernel->hasGroup != PL_TRUE) {
    return PL_NOT_FOUND;
  }
  // Check if compute unit exist
  if (kernel->compute_units <= index) {
    return PL_NOT_FOUND;
  }

  // Set Group id
  PL_Write_Register(kernel->address_ptr[index], kernel->group->id_x,
    x);
  PL_Write_Register(kernel->address_ptr[index], kernel->group->id_y,
    y);
  PL_Write_Register(kernel->address_ptr[index], kernel->group->id_z,
    z);

  // Return success
  return PL_SUCCESS;
}

// Kernel Set Global Offset
pl_int plSetKernelGlobalOffsets (pl_kernel kernel, pl_int x, pl_int y
    , pl_int z) {
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->
    programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check is kernel is ready
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }

  // Check if group data exist
  if (kernel->hasGroup != PL_TRUE) {
    return PL_NOT_FOUND;
  }

  // For each unit
  for (int index = 0; index < kernel->compute_units; ++index) {
    // Set Group id
    PL_Write_Register(kernel->address_ptr[index], kernel->group->
    offset_x, x);
    PL_Write_Register(kernel->address_ptr[index], kernel->group->
    offset_y, y);
    PL_Write_Register(kernel->address_ptr[index], kernel->group->
    offset_z, z);
  }

  // Return success
  return PL_SUCCESS;
}
pl_int plSetKernelComputeUnitGlobalOffsets (pl_kernel kernel, size_t
    index, pl_int x, pl_int y, pl_int z) {
```

```
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->
     programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check is kernel is ready
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }

  // Check if group data exist
  if (kernel->hasGroup != PL_TRUE) {
    return PL_NOT_FOUND;
  }

  // Check if compute unit exist
  if (kernel->compute_units <= index) {
    return PL_NOT_FOUND;
  }

  // Set Group id
  PL_Write_Register(kernel->address_ptr[index], kernel->group->
     offset_x, x);
  PL_Write_Register(kernel->address_ptr[index], kernel->group->
     offset_y, y);
  PL_Write_Register(kernel->address_ptr[index], kernel->group->
     offset_z, z);

  // Return success
  return PL_SUCCESS;
}



// Run Kernel compute unit
pl_int plRunKernelComputeUnit (pl_kernel kernel, size_t index,
     pl_bool blocking) {
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->
     programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check is kernel is ready
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }

  // Check if compute unit exist
  if (kernel->compute_units <= index) {
    return PL_NOT_FOUND;
  }

  // Get control register
  pl_u32 value = PL_Read_Register(kernel->address_ptr[index], kernel
   ->ctrl->ctrl) & 0x80;
  // Set start register
  PL_Write_Register(kernel->address_ptr[index], kernel->ctrl->ctrl,
   value | 0x01);

  // If blocking enable
  if (blocking == PL_TRUE) {
    // Wait while not done
    do {
      // Get value
      value = PL_Read_Register(kernel->address_ptr[index], kernel->
   ctrl->ctrl);
    } while (!((value >> 1) & 0x1));
  }

  // Return success
  return PL_SUCCESS;
}
// Wait Kernel compute unit
pl_int plWaitKernelComputeUnit (pl_kernel kernel, size_t index) {
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->
   programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check is kernel is ready
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }

  // Check if compute unit exist
  if (kernel->compute_units <= index) {
    return PL_NOT_FOUND;
  }

  // Get control register
  pl_u32 value;

  // Wait while not done
  do {
    // Get value
    value = PL_Read_Register(kernel->address_ptr[index], kernel->ctrl
   ->ctrl);
  } while (!((value >> 1) & 0x1));

  // Return success
  return PL_SUCCESS;
}



// Run Kernel Task
pl_int plRunTask (pl_kernel kernel, pl_bool blocking) {
  // Call the first compute unit
  return plRunKernelComputeUnit(kernel, 0, blocking);
}
// Wait Kernel Task
pl_int plWaitTask (pl_kernel kernel) {
  // Wait the first compute unit
  return plWaitKernelComputeUnit(kernel, 0);
}



// xyz struct
typedef struct {uint x; uint y; uint z;} xyz;

// Run a ND range kernel
pl_int plRunNDRangeKernel (pl_kernel kernel, pl_uint work_dim, const
    size_t *global_work_offset, const size_t *global_work_size,
    pl_bool blocking){
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->
    programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check is kernel is ready
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }
  // Check dimentions
  if (work_dim <= 0 || work_dim > 3) {
    return PL_NOT_VALID;
  }

  // Init variables
  size_t offset[3] = {0, 0, 0};
  size_t size[3] = {1, 1, 1};
  int i;

  // Get offsets
  if (global_work_offset != NULL) {
    for (i = 0; i < work_dim; ++i) {
      offset[i] = global_work_offset[i];
    }
  }
  // Get sizes
  if (global_work_size != NULL) {
    for (i = 0; i < work_dim; ++i) {
      size[i] = global_work_size[i];
    }
  }

  // Calculate ranges
  size_t groups[3];
  groups[0] = (size[0] + kernel->workgroup[0] - 1) / kernel->
    workgroup[0];
  groups[1] = (size[1] + kernel->workgroup[1] - 1) / kernel->
    workgroup[1];
  groups[2] = (size[2] + kernel->workgroup[2] - 1) / kernel->
    workgroup[2];

  // Allocate data list
  size_t length = groups[2] * groups[1] * groups[0];
  xyz* data = (xyz*) malloc(length * sizeof(xyz));

  uint x, y, z, l = 0;
  for (z = 0; z < groups[2]; ++z){
    for (y = 0; y < groups[1]; ++y){
      for (x = 0; x < groups[0]; ++x){
        data[l].x = x;
        data[l].y = y;
        data[l].z = z;
        l++;
      }
    }
  }

  // Keep a state of the compute units
  pl_bool* cu_running = (pl_bool*) malloc(kernel->compute_units *
    sizeof(pl_bool));
  for (i = 0; i < kernel->compute_units; ++i) {
    cu_running[i] = PL_FALSE;
  }

  // Distribute the work on the compute units
  pl_u32 ctrl;
  pl_int err;
  l = 0;
  while (l < length) {
    // Loop compute units and give work to anyone that is free
    for (i = 0; i < kernel->compute_units && l < length; ++i) {
      // Get value
      ctrl = PL_Read_Register(kernel->address_ptr[i], kernel->ctrl->ctrl);
      // If compute units is done
      if (cu_running[i] == PL_FALSE || ((ctrl >> 1) & 0x1)) {
        // Set offset
        err = plSetKernelComputeUnitGlobalOffsets(kernel, i, offset[0], offset[1], offset[2]);
        if(err != PL_SUCCESS) return PL_FAILED;
        // Set group id
        err = plSetKernelComputeUnitGroupIds(kernel, i, data[l].x, data[l].y, data[l].z);
        if(err != PL_SUCCESS) return PL_FAILED;
        // Start
        PL_Write_Register(kernel->address_ptr[i], kernel->ctrl->ctrl, ctrl | 0x01);
        // Set work task assigned
        l++;
        cu_running[i] = PL_TRUE;
      }
    }
  }

  // If blocking enable
  if (blocking == PL_TRUE) {
    // For each compute unit
    for (i = 0; i < kernel->compute_units; ++i) {
      if(cu_running[i] == PL_TRUE) {
        // Wait while not done
        do {
          // Get ctrl
          ctrl = PL_Read_Register(kernel->address_ptr[i], kernel->ctrl->ctrl);
        } while (!((ctrl >> 1) & 0x1));
      }
    }
  }

  // Return success
  return PL_SUCCESS;
}
// Wait Kernel compute unit
pl_int plWaitNDRangeKernel (pl_kernel kernel) {
  // Check pointer
  if (kernel == NULL) {
    return PL_NULL_POINTER;
  }

  // If this is not the bitfile programmed on pl
  if (pllib_device.programedID != ((pl_bitfile) kernel->bitfile)->programedID) {
    return PL_NOT_PROGRAMED;
  }

  // Check is kernel is ready
  if (kernel->isReady != PL_TRUE) {
    return PL_NOT_READY;
  }

  // Get control register
  pl_u32 value;

  // For each compute unit
  for (int i = 0; i < kernel->compute_units; ++i) {
    // Wait while not done and not idle
    do {
      // Get ctrl
      value = PL_Read_Register(kernel->address_ptr[i], kernel->ctrl->
    ctrl);
    } while (!((value >> 1) & 0x1) && !((value >> 2) & 0x1));
  }

  // Return success
  return PL_SUCCESS;
}



// Load string from file
char * plToolsLoadFileContents(char *filepath) {
  char * buffer = 0;
  long length;
  FILE * f = fopen (filepath, "rb");

  if (f) {
    fseek (f, 0, SEEK_END);
    length = ftell (f);
    fseek (f, 0, SEEK_SET);
    buffer = malloc (length);
    if (buffer){
      fread (buffer, 1, length, f);
    }
    fclose (f);
  }

  if (buffer) {
    return buffer;
  }
  else {
    return NULL;
  }
}

// Copy file content to other file
pl_int plToolsCopyFileContents(char *source, char* target) {
  size_t len;
  char buffer[128];
  FILE *in, *out;
  if ((in = fopen(source, "rb")) != NULL) {
    if ((out = fopen(target, "wb")) != NULL) {
      do {
        len = fread(buffer, 1, sizeof(buffer), in);
        if (len != 0) {
          fwrite(buffer, 1, len, out );
        }
      } while (len == sizeof(buffer));
      fclose(out);
      fclose(in);
    }
    else {
      fclose(in);
      return PL_FILE_OPEN_FAILED;
    }
  }
  else {
    return PL_FILE_OPEN_FAILED;
  }

  return PL_SUCCESS;
}
```

Listing C.2: SimplePL source file

D References

Altera, FPGA vendor, `https://www.altera.com/`

CUDA, Parallel computing platform and API, `https://developer.nvidia.com/cuda-zone`

Kernel Module Guide, The Linux Kernel Module Programming Guide, `http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html`

OpenCL, Open standard for parallel programming of heterogeneous systems, `https://www.khronos.org/opencl/`

OpenCL 1.2, OpenCL 1.2 Reference Card, `https://www.khronos.org/files/opencl-1-2-quick-reference-card.pdf`

OpenCL AMD, AMD OpenCL SDK, `http://developer.amd.com/tools-and-sdks/opencl-zone/`

OpenCL Altera, Altera OpenCL Development Environment, `https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html`

OpenCL Apple, Apple OpenCL on Mac, `https://developer.apple.com/opencl/`

OpenCL example, OpenCL Vector Addition, `https://www.olcf.ornl.gov/tutorials/opencl-vector-addition/`

OpenCL Headers, Khronos Group Github OpenCL Headers, `https://github.com/KhronosGroup/OpenCL-Headers`

OpenCL Intel, Intel OpenCL SDK, `https://software.intel.com/en-us/articles/opencl-drivers`

OpenCL nVidia, NVIDIA OpenCL SDK, `https://developer.nvidia.com/opencl`

OpenCL Xilinx, SDAccel Xilinx OpenCL Development Environment, `https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`

OpenCL on Zynq, Bo Joel Svensson, `http://svenssonjoel.github.io/writing/ZynqOpenCL.pdf`

PetaLinux, Linux for Xilinx boards, `http://www.wiki.xilinx.com/PetaLinux`

pocl, Portable Computing Language, `http://portablecl.org/`

tiny-json, Json parser in C, `https://github.com/rafagafe/tiny-json`

udmabuf, User space mappable dma buffer device driver for Linux., `https://github.com/ikwzm/udmabuf`

Vivado Design Suite, HDL design tool, `https://www.xilinx.com/products/design-tools/vivado.html`

Vivado HLS, C/C++/OpenCL High-Level Synthesis tool, `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`

Xilinx, FPGA vendor, `https://www.xilinx.com/`

Xilinx Linux 2017.2, FPGA vendor, `https://www.xilinx.com/`

Xilinx Linux Kernel, Linux Zynq 2017.2 Release, `http://www.wiki.xilinx.com/Zynq_2017.2_Release`

Xilinx OpenCL C Example, SDAccel Environment Optimization Guide, `https://www.xilinx.com/html_docs/xilinx2017_2/sdaccel_doc/topics/introduction/con-opencl-c-example.html`

Xilinx SDAccel Examples, Example codes for SDAccel, `https://github.com/Xilinx/SDAccel_Examples`

Xilinx Wiki, Wiki and Guides, `http://www.wiki.xilinx.com/`

Zedboard, Zynq-7000 ARM/FPGA SoC Development Board, `http://store.digilentinc.com/zedboard-zynq-7000-arm-fpga-soc-development-board/`