ECE 340
Embedded Systems

Spring 2023

Lab 3

FPGAs as Accelerators

1. Introduction

Reconfigurable systems (FPGAs) offer the performance and energy efficiency of hardware with the flexibility of software. Their functionality can be customized during their operational lifecycle and specialized in the particular instance of a task. Furthermore, this specialized system could be reconfigured for different tasks and new algorithms, as well as for new performance, power, and output accuracy requirements. Moreover, Multiprocessor System On Chip (MPSoC) FPGA devices integrate both multiprocessors and the reconfigurable fabric into a single device. Consequently, they provide higher integration, lower power, and higher bandwidth communication between the processor and the fabric. FPGAs have been deployed lately in a wide variety of domains, such as large-scale datacenters for cloud computing as well as High-Performance computers. Cloud companies like Amazon, Microsoft, Baidu, Huawei, etc. are offering FPGA systems to their customers for application development or are using FPGAs internally in their servers to accelerate a variety of workloads (typically Machine Learning).

FPGA fabric can be used to implement hardware accelerators to offload computationally demanding tasks from the CPU. A key enabler for this architectural exploration is High-level synthesis (HLS) technology that enables a designer to focus on larger architectural questions rather than individual registers and cycle-to-cycle operations. A designer captures behavior in a program that does not include timing specifications and an HLS tool creates the detailed RTL micro-architecture. The flexibility of the HLS methodology allows the designer to explore a wide range of architectural optimizations trading off the performance versus FPGA resources.

In this lab, you will implement an application for local sequence alignment (LSAL), an algorithm widely used in bioinformatics. The algorithm will be developed first in software running on x86 and Zedboard Arm processors, and then as a hardware accelerator running on the Zedboard fabric. You will run a software implementation of this application and you will profile the application to assess its performance footprint. However, software running on a processor, no matter how well optimized it is, is usually slower than a well-designed hardware accelerator implementing the same functionality. Based on the outcome of software profiling, you will implement the local sequence alignment as an accelerator to reduce its execution time using Vitis HLS (High-Level Synthesis) and the OpenCL API of the Vitis Unified Software Platform. Optimizations in hardware can provide additional performance gains. This lab will be implemented using the Vitis Software Platform targeting the Zedboard. The Arm processors are running Petalinux (a Linux distribution for Xilinx FPGA boards).

Section 2 of this document describes LSAL which is a dynamic programming algorithm. Section 3 provides a step-by-step optimization methodology that can be used to optimize the performance of the algorithm using the Xilinx Vitis toolset. Note that some of the theory behind this lab is covered by several appendices at the end of the document. Students should study this material before attempting to solve the problems of the lab.

There are two types of deliverables for Lab3. First, the source code for the optimal software and hardware designs organized in two different directories. Second, a detailed report that outlines your

experiments, presents the results using tables and figures, and discusses and explains these results. The quality of your report and your final presentation are important for your final grade.

The lab is designed to give you experience with:

- The design flow of acceleration-based computing.
- Understanding when an application can be mapped to a hardware accelerator.
- Performance analysis for software running on a CPU and hardware running on the FPGA fabric.
- High-Level Synthesis (HLS) techniques and optimizations.
- Understanding the differences between mapping an application to a general-purpose CPU and mapping an application to a hardware accelerator.
- Analyzing performance vs. area tradeoffs in an FPGA design.
- Agile design methodologies including incremental and test-driven development.

2. Local Sequence Alignment Algorithm (LSAL)

LSAL is a local sequence alignment algorithm for identifying relationships between strings of genetic data. In bioinformatics, sequence alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity between two sequences. In particular, local alignments identify regions of similarity within long sequences that are often widely divergent. Given two input strings, a database D of length M, and a query Q of length *N*, the objective is to find regions of similarity between subsequences of all possible lengths. An $M \times N$ similarity matrix *S* is constructed to hold similarity scores for each subsequence comparison. Each element of the matrix ($S_{i,j}$) represents the similarity of strings *D* from index 1 to i and *Q* from 1 to j and is dependent on previous values to the left, diagonal, and above as described in the following equation:

$$S_{i,j} = max \begin{cases} S_{i-1,j-1} + s(d_i, q_j) \\ \max\limits_{k \geq 1}\{S_{i-k,j} + W_k\} \\ \max\limits_{l \geq 1}\{S_{i,j-l} + W_l\} \\ 0 \end{cases} \quad (0 \leq i \leq m-1, 0 \leq j \leq n-1)$$

The first entry, $S_{i-1,j-1} + s(d_i, q_j)$ is the score of aligning d_i and q_j. The function s depicts the similarity score. In our case, we use the following similarity function:

$$s(d_i, q_j) = \begin{cases} +2 \text{ if } d_i = q_j \\ -1 \text{ if } d_i \neq q_j \end{cases}$$

The parameter *W* is the gap penalty. The entry $\max\limits_{k \geq 1}\{S_{i-k,j} + W_k\}$ is the score if q_j is at the end of a gap of length *k*. Likewise, the entry $\max\limits_{l \geq 1}\{S_{i,j-l} + W_l\}$ is the score if d_i is at the end of a gap of length *k*. In our version of the algorithm, we select $k = 1, W_1 = -1$. The similarity matrix is initialized as $S_{i,j} = 0$ when $i < 0$ or $j < 0$. Note that always $S_{i,j} \geq 0$.

		Query [N]							
		T	G	T	T	A	C	G	G
Database [M]	G	0	2	1	0	0	0	2	2
	G	0	2	1	0	0	0	2	4
	T	2	1	4	3	2	1	1	3
	T	2	1	3	6	5	4	3	2
	G	1	4	3	5	5	4	6	5
	A	0	3	3	4	7	6	5	5
	C	0	2	2	3	6	9	8	7
	T	2	1	4	4	5	8	8	7
	A	1	1	3	3	6	7	7	7
		G	T	T	--	A	C		
		G	T	T	G	A	C		

(a)

		Query [N]							
		T	G	T	T	A	C	G	G
Database [M]	G	C	NW	W	C	C	C	NW	NW
	G	C	NW	NW	C	C	C	NW	NW
	T	NW	N	NW	NW	W	W	N	N
	T	NW	NW	NW	NW	W	W	W	N
	G	N	NW	W	N	NW	NW	NW	NW
	A	C	N	NW	N	NW	W	N	NW
	C	C	N	NW	N	N	NW	W	W
	T	NW	N	NW	NW	N	N	NW	NW
	A	N	NW	N	NW	NW	N	NW	NW

(b)

Figure 1. (a) LSAL Similarity matrix for a Database of length M=9, and a Query of length N=8. The cell with the largest similarity value when the algorithm terminates is shown in Green. We start searching for the longest local sequence alignment (shown at the bottom of the similarity matrix) starting from this cell and backtracing till we reach a cell with a value of 0 in the similarity matrix. Note that the alignment is partial since the last G of the database does not match the symbol of the query. (b) The corresponding direction matrix. Note that the priority among the four possible directions in case of a draw is C, NW, N, W. Note that, in most cases, the database length M is much larger than the query length N, i.e. M>>N. T = Thymine, G = Guanine, A = Adenine, and C = Cytosine are the four main nucleobases found in the nucleic acids DNA and RNA.

Figure 1a shows the similarity matrix for two strings of sizes M=9 and N=8. The algorithm starts from the top left cell and computes the values $S_{i,j}$ row-wise until the bottom rightmost cell is reached using the equations above. The maximum similarity score (shown in green in Figure 1a) will be the starting point of the backtracing stage of LSAL.

A second matrix *(*called direction matrix) is formed to hold the direction path through matrix S to produce an alignment. The cell value of the direction matrix indicates the direction followed to reach that cell. The optimal alignment is produced by starting from the position of the maximum score in matrix S and following the directions found at the same index in the direction matrix until a zero in S is found.

The complexity of this dynamic algorithm is $O(M * N)$. The importance of sequence alignment coupled with the explosion of available genetic data and the computational complexity of sequence alignment algorithms has driven a need for a high-performance solution. In typical cases, the size M of the database string can be in the order of millions of symbols.

3. LSAL implementation

LSAL implementations vary by the target platform (CPU, GPU, FPGA, ASIC, etc.), the supported input data sizes, the number of cells processed in parallel, and which part of the algorithm is accelerated. In this lab, you will incrementally develop, evaluate and optimize for performance various implementations of the LSAL algorithm in three different platforms: an x86 CPU, the Arm Cortex A9, and the FPGA. The last two platforms are part of Zedboard which was used for labs 1 and 2.

This chapter provides detailed information and guidance on how the LSAL algorithm can be mapped on these platforms. The lab follows the development flow shown in Figure 2. We **strongly recommend** that you follow these steps.

3.1. Initial LSAL implementation on the x86 CPU

In this phase, you should develop the C code of the LSAL algorithm starting from the given template file. Then, you should compile the C source code (use *-O3* and *-mavx2* flags in *gcc*), and run the executable on the desktops of the lab or your own laptop. We have also included code to record the execution time using the *clock()* instructions.

The code of the *compute_matrices* function should scan the (initially empty) similarity matrix and fill in both the *similarity matrix* and the *direction matrix*. The quality of your initial LSAL implementation will determine how easily this code can be optimized when implemented in hardware in the later stages of lab3.

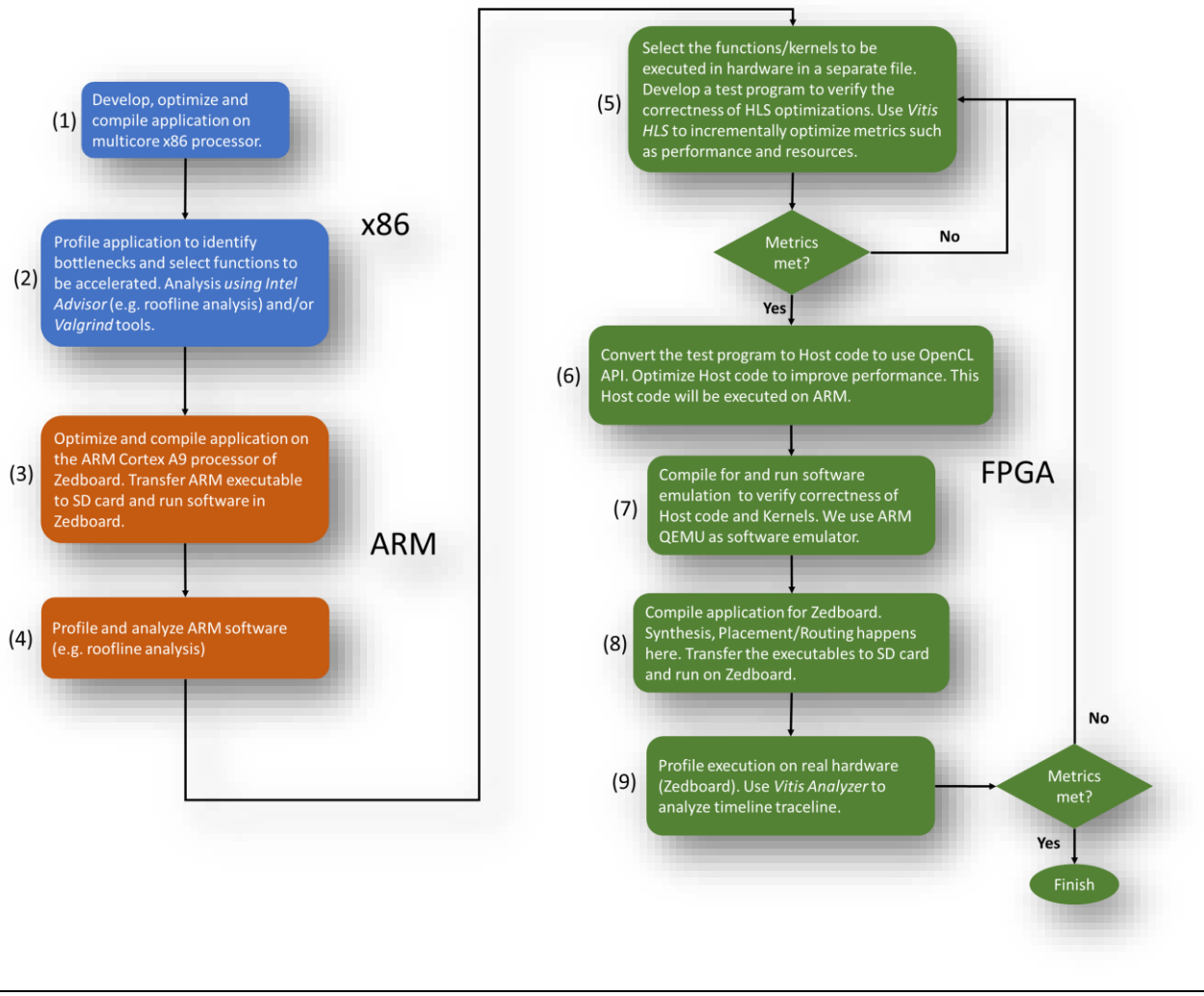


Figure 2. Development flow of lab3. The numbers in parentheses correspond to the material in the subsections of section 3.

3.2. Profile and optimization of the x86 CPU implementation. Roofline Analysis.

Roofline Model. Measuring the execution time of the various implementations of the LSAL algorithm is very useful but does not provide insights into the potential for further performance improvements. In this lab, you will obtain a deeper understanding of the bottlenecks of the application and how they can be avoided by using the roofline model to predict performance and track optimizations on different platforms [1]. The roofline model is a graphical representation of the relationship between off-chip memory bandwidth and the performance of the processor and represents the theoretical peak performance for a given platform. This is very useful for comparing different implementations of an application in a given platform. In Appendix *A. Roofline Model*, we explain the basics behind the roofline model and Intel's roofline Insights tool.

In this stage, you will become familiar with and will analyze the x86 LSALimplementation using the *CPU/Memory Roofline Insights* of the *Intel Advisor* toolset [2]. *Intel Advisor* can be downloaded

for free in Linux and works without problems in a Linux VM. You need to compile the code with *gcc* and call the *Intel Advisor* toolset on the executable. Your lab report should present the output of the Roofline Analysis for your LSAL implementation using a variety of input sizes. In this and the following experiments, we propose that you use (at least) the following values for N and M: (N=32, M=32), (N=32, M=65536), (N=256, M=65536), (N=256, M=300,000). Does the position of the *compute_matrices* function in the roofline model change for different input sizes? Explain your findings in the report.

Valgrind. *Valgrind* is a suite of tools used for debugging and profiling. It can be used instead of or in conjunction with *Intel Advisor*. You can use the *callgrind* tool (which is part of the *valgrind* toolset) to record the number of instructions executed, their relationship to source lines, the caller/callee relationship between functions, and the number of such calls. The profile data generated by *callgrind* is written out to a file at program termination. For the following example, we assume that the output file generated is *callgrind.out.3088.* This file is input to *kcachegrind*, a profile data visualization tool used to show the call graph of the *lsal* execution with inputs N=32 and M=65536.

```
gcc -O3 -mavx2 -g -o ./lsal ./lsal.c

valgrind --tool=callgrind ./lsal 32 65536

kcachegrind ./callgrind.out.3088
```

3.3. LSAL implementation on the Arm CPU.

In this step, you will run and profile your code in the Arm Cortex A9 CPU in the Zedboard FPGA. You can use the same source code you developed for x86. The following instructions will guide you through the process of compiling the code targeting the Arm processor and running the executable in the Arm CPU.

- Compile the *lsal.c* source code in your x86 Linux box (or the Linux VM) using the Arm cross-compiler (part of the Xilinx Vitis installation). Cross-compilation is the act of compiling code for one computer system (often known as the target) on a different system, called the host. It is widely used when the target system is too small to host the compiler and all relevant files. You do not want to use Zedboard as a development platform!

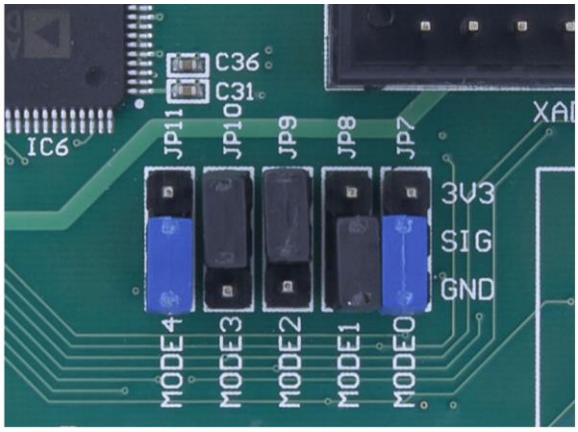
```
/opt/Xilinx/Vitis/2020.2/gnu/aarch32/lin/gcc-arm-linux-gnueabi/bin/arm-
linux-gnueabihf-g++ -o ./lsal_sw_arm ./lsal.c -Wall -O3 -g -std=c++11 -
fmessage-length=0
```

- If this is the first time you use the SD card, please read Appendix E for details. You need to copy the necessary files from your desktop to the two partitions of the SD card before you start using it. The files which reside in *~/courses/ECE340_EmbeddedSystems/bootArtifacts* include the necessary boot files and the root file system. Moreover, you need to copy the *lsal_sw_arm* executable to the SD card.

Also, you may want to read Appendix F to better understand the booting process in the Zynq processor (and, in general, in a CPU).

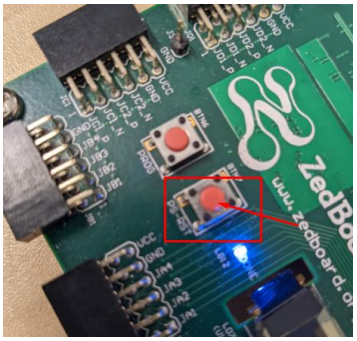
IMPORTANT: You must copy the root filesystem as root (or sudo) user in the root partition.

- It is important that you have the correct jumper settings to be able to boot from the SD Card (see figure below). After you insert the SD Card in Zedboard, switch it on and open *minicom* in your Linux desktop exactly with the same settings as in lab2. Zedboard starts executing the First Stage Bootloader residing in file *BOOT.bin*. Wait for the command prompt to show Zynq>.



When this happens, enter the command *boot* and wait for the login prompt. Several (close to incomprehensible) messages will appear before petalinux finally boots and asks for your login credentials. If it seems that the boot procedure is stuck reset the board by using the PS-RST button on the board (one of the two red buttons).

TIP: To make sure that the boot sequence was successful check if the blue light is on.



Use the following credentials to log in:

user: *root*

password: *root*

You will be logged to petalinux.

Petalinux will automatically mount the boot partition to the directory */mnt/sd-mmcblk0p1*. It corresponds to the BOOT partition and contains all the files needed,

```
cd /mnt/sd-mmcblk0p1
ls -alt
```

- Execute your code in Arm Cortex A9 of Zedboard and record execution time for various input sizes N and M. Note that Zedboard has only 500 MB DRAM, which means that you cannot run for very large array sizes. The memory heap used for *malloc()* is much smaller than in an x86 system.

```
./lsal_sw_arm 256 65536
```

3.4. Profile and Optimization of the Arm CPU implementation.

The Zedboard ecosystem (being a low-power embedded platform) is not as rich as the x86 ecosystem when it comes to profile and analysis tools. The only available tool is the *clock()* function which can be used to measure CPU time. You should run *./lsal_sw_arm* multiple times and compare it with x86 execution (executions when the *clock* is used in x86). How do these two execution times compare? What is the relative execution time ratio when the input size N, M increases?

Ideally, you should construct a roofline model of Arm Cortex A9 and use it to locate execution points. Note also that you can use the *perf* tools available in petalinux to evaluate metrics such as IPC, cache behavior, branch prediction, etc.

3.5. Hardware accelerators using Vitis HLS

Introduction. This subsection is the most important step of lab3. It presents the methodology and toolsets to incrementally design the architecture of the hardware LSAL accelerators using the Vitis HLS toolset. Writing highly optimized synthesizable HLS code is often not a straightforward process. It involves a deep understanding of the application, and the ability to change the code such that the Vitis HLS tool creates optimized hardware structures and utilizes synthesis directives effectively. In this step, each optimization is applied incrementally from the initial design until an implementation is reached that satisfies our performance metrics and fits in the FPGA.

Appendix B summarizes some basic concepts of the Vitis HLS optimization methodology and related directives. For more information on High-Level Synthesis for Xilinx FPGAs, you should consult the latest *Vitis High Level Synthesis User Guide, UG1399* [3], and the very practical *Vivado High Level Synthesis Tutorial, UG871*[1] [5]. We strongly recommend that you finish as many labs of *UG871* tutorial as possible before you continue with the Vitis HLS. Especially, ch.1 (tool flow),

[1] Vivado HLS is an older HLS tool version of Xilinx, but most (though not all) aspects of HLS have been carried through to Vitis HLS. One major difference is that Vitis HLS attempts optimizations such as software pipelining by default even without *#pragmas*.

ch.2 (HLS Introduction), ch. 4 (Interface Synthesis), ch. 6 (Design Analysis), and ch. 7 (Design Optimization).

In this phase, you will develop hardware accelerators for the portion of the LASL that needs to be accelerated, as was determined from the profile analysis done in the x86/Arm CPUs. In fact, you will develop the following two separate C/C++ files:

- A C/C++ file (e.g., *lsal.c*) that implements the top-level kernel function to be mapped into hardware. All functions that are called by the top-level function, will also be mapped into hardware.

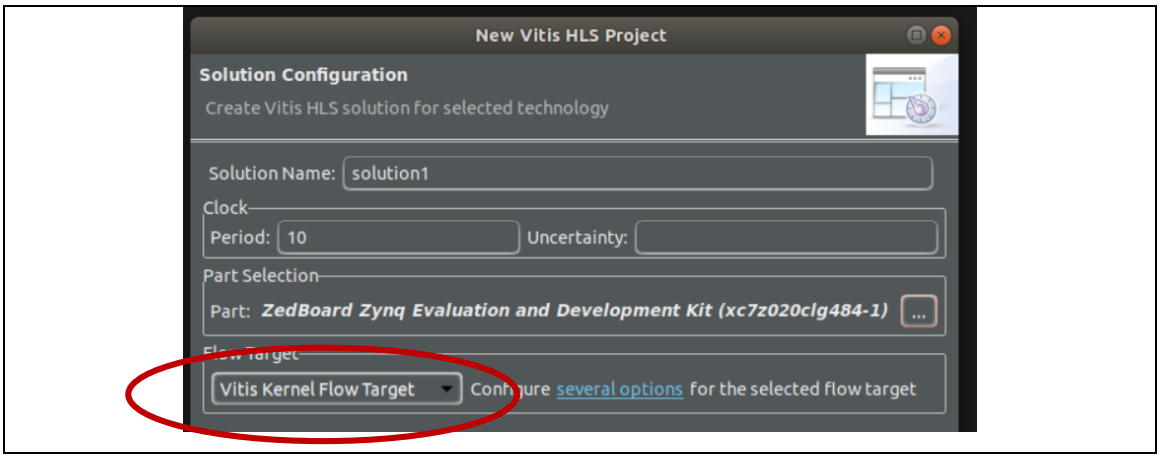
- A test bench (e.g., *lsal_test.c*) which is used to invoke the top-level C/C++ kernel function and validate its correctness (this is very similar in concept to an RTL test bench). The test bench makes all appropriate initializations, file I/O, and invokes the top-level kernel function. Moreover, the test bench also needs to implement the same kernel function in software and use this implementation as a golden version to verify that the hardware version is correct. Vitis HLS re-uses the test bench during C and during RTL verification and confirms the successful verification of the RTL if the test bench returns a value of 0. If any other value is returned by *main(),* including no return value, it indicates that the RTL verification failed. This provides a robust and productive verification methodology.

Vitis HLS GUI basics. Invoke the Vitis HLS GUI from the Linux prompt as follows:

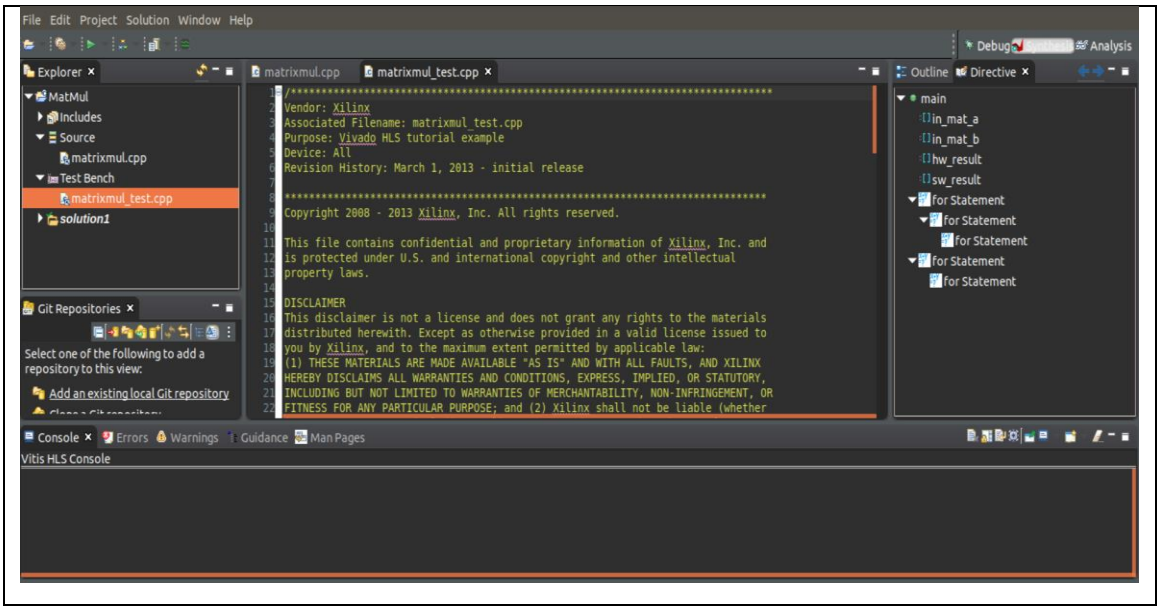
```
vitis_hls
```

Start by *Creating a New Project*, press *Next*, then select the name of the project, and its location, and press *Next*. You will need to add to the project all the C/C++ Files which contain the kernel functions (e.g. *lsal.c*), and to indicate which is the top-level C/C++ kernel function. Press *Next.* In the next windows, you include all the files used by the test bench (e.g. *lsal_test.c*). Press *Next* again.

The figure below shows the next window (Solution Configuration). The Solution Configuration window specifies the technical specifications of the first solution. A project can have multiple solutions, each using a different target technology, package, constraints, and/or synthesis directives. Accept the default solution name (**solution1**), target clock period for synthesis (**10 ns**), and clock uncertainty (defaults to 12.5% of the clock period, when left blank/undefined). Click the part selection button to open the part selection window. Select the **Boards** tab and select **Zedboard** from the list of available boards. Different FPGA devices/boards may have a widely different performance footprint, thus, you must specify Zedboard as your target board to get accurate performance estimates. Finally, you should select the **Vitis Kernel Flow Target** to configure Vitis HLS to generate the compiled kernel object (.xo) for the Vitis application acceleration flow. The Vitis Kernel flow is more restrictive than the Vivado IP flow, and the kernels produced by the HLS tool must meet the specific requirements of the platforms and Xilinx runtime (XRT).



The following GUI is the main window of the Vitis HLS tool. At that point, you should do the tutorials of *UG871* for more information on how to use Vitis HLS functionalities such as C simulation, RTL simulation (optional), and C Synthesis.



LSAL architectural exploration. You will work on the hardware accelerator by using the Vitis HLS toolset.

As a first step, you should implement the *baseline* accelerator using the C/C++ code of x86/Arm CPU execution you developed in sections 3.1 and 3.3. As mentioned, you have to write your own source testbench to invoke the kernel function and, as a first step, to test the correctness of the code. Then, you synthesize the baseline accelerator C/C++ code into RTL, and you study the Synthesis and Analysis reports available in the Vitis HLS. For each such implementation, you should record

basic metrics[2] such as latency, iteration interval (II), clock frequency as well as resource utilization (LUTs, FFs, DSPs, BRAMs). Once you synthesize the HW baseline implementation, you should move on with the following sections according to Figure 2 and run the design in Zedboard. Since optimization is an incremental task, you will come back to section 3.5 after you run the first implementation.

The baseline accelerator may run efficiently in a CPU, but (most probably) is very slow as a hardware accelerator. You should be able to understand the bottlenecks of the baseline HW implementation since this will guide you through the next step of performance optimizations. What is the main limitation of the baseline solution? As additional help (and for extra credit), you can also build the roofline model of the baseline implementation by computing the Arithmetic Intensity and the Performance of the code (see Appendix A).

As we have mentioned in class, hardware design gives you a lot of potential to design different hardware accelerators that span the performance vs. resources space. After detecting the performance bottleneck of the first hardware solution you should apply a series of optimizations to improve performance (i.e. to reduce latency). Some optimizations ideas are the following (note that this is only a partial list):

- The baseline implementation scans sequentially all $N * M$ cells to update the similarity and the direction matrices. This may be acceptable in an SW implementation, but it is clearly sub-optimal in a hardware platform that can support massive computational parallelism. You should detect what are the patterns of parallelism in the LSAL algorithm. In general, parallelism is constrained by data and control dependencies. It may help if you draw data dependencies between source and sink in Figure 1 to help you understand what the degree of parallelism is, and which cells can be updated **concurrently**.

  Once you detect the parallelism, you need to restructure the code of the loop that iterates through the $N * M$ cells so that each loop iteration computes all cells that are executed concurrently. This is a key transformation and requires that you modify how the cells are updated using the West, North, NorthWest values of the similarity matrix that must have been updated in the previous iterations. Moreover, you need to take particular care of the loop boundaries.

- HLS directives that increase bandwidth and/or latency should be applied aiming at a small iteration interval (ideally II=1) and lower iteration latency. Moreover, Vitis HLS supports fixed-point arithmetic using arbitrary precision which can be different than the default 32-, 16-, 8-bit types. All these are important optimization tools regardless of the algorithm used. See Appendix B for more information.

- Bandwidth to global memory is important to sustain computational performance. You need to use memory bursting (*mempcy*) to read/write data between the global DRAM and the BRAMs,

[2] These metrics are best-effort evaluations of the Vitis HLS synthesizer, not measurements from real hardware executions.

and to avoid accesses of single parameters as much as possible. Moreover, the maximum data width from the global memory to and from the kernel is 512-bits. To maximize the data transfer rate, it is recommended that you use this full data width (e.g., using ports of 512-bit arbitrary precision). By default, in the Vitis kernel flow, the Vitis HLS tool automatically re-sizes the kernel interface ports up to 512-bits to improve burst access. See Appendix B for more information.

- The LSAL algorithm processes only four types of input symbols: C, T, G, A. Therefore, using a *char* type to represent these elements disregards this information and is a waste of FPGA resources (Why?). You may want to compress the *query* and the *database* so that they use a minimum number of bits.
- Data reuse within the accelerator is critical to reducing traffic between the kernel and the global memory. The kernel should read from global memory only the symbols needed for a particular stage of the computation and try to reuse symbols as much as possible.

The output of this step is the C/C++ kernel file *lsal.c*.

3.6. Vitis OpenCL Execution Flow

Vitis HLS allows for fast architectural exploration without the long implementation times needed to implement and run the accelerator in Zedboard. We expect that the students will spend a lot of design iterations in Vitis HLS compared with running in real hardware.

Once architectural exploration using the Vitis HLS is concluded with a new (and hopefully faster!) hardware kernel, that kernel should be prepared to run in real hardware (Zedboard) under the control of the Arm Host unit. You will use (after thoroughly studying) the given OpenCL-based source code for the host unit (*lsal_host.c*). The host unit code which is based on the OpenCL programming model is executed on the Arm CPU. Even if the hardware does most of the heavy-duty computation, the Arm core will still execute all tasks related to initialization, the invocation of and communication with the hardware accelerator, and validation of the results produced by the accelerator. The OpenCL API first defines the data structures used in the program (e.g., *query* and *database*), initializes them with random data, performs configuration and triggering of the hardware accelerator, and blocks until the hardware accelerator finishes execution. Then, the Arm core will read the output arrays (*similarity matrix*, *direction matrix*, and *max index*) back from the accelerator. Finally, the software running on the Arm core checks the correctness of the accelerator output. The host unit code compares the output of the hardware accelerator to a golden software implementation of the LSAL algorithm file and reports any mismatches.

The OpenCL API exposes a lot of details about the interfaces between the Arm and the hardware accelerator and is interesting to read and understand (see Appendix C for the details). It is very

similar[3] to any OpenCL host unit code that you could run for a GPU accelerator. In fact, OpenCL code can run without any changes in CPUs, GPUs, and FPGAs.

The output of this stage consists of the two C/C++ files: a) the kernel file *lsal.c* which was developed in section 3.5, and b) the Host unit file *lsal_host.c.*

3.7. Software Emulation

The remaining development stages illustrate how you run the host and kernel code in software emulation and in real hardware (check also Appendix C). Figure 3 shows the recommended directory structure we are proposing for Vitis development.

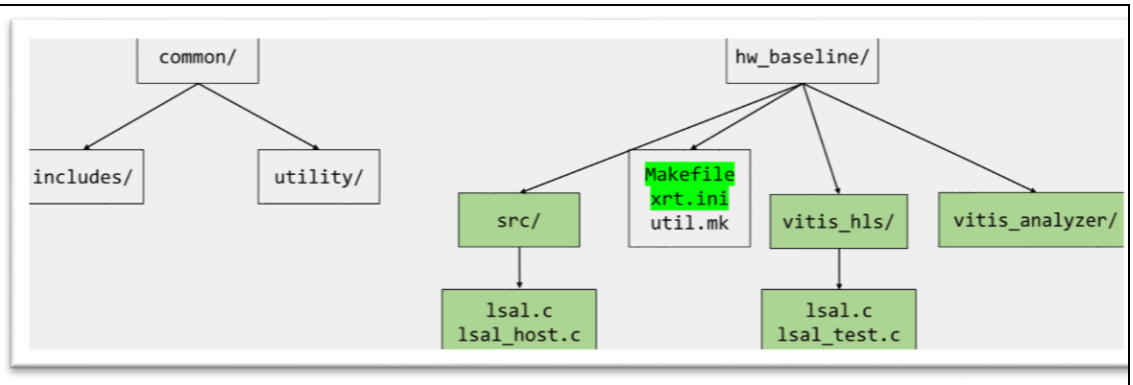


Figure 3. Directory structure used in this lab for application development. The files and/or directories you need to modify are coloured. For each new kernel implementation, you need to build a new directory structure like *hw_baseline*. The development starts with the Vitis HLS, and then the *lsal.c* file has to be copied over to the *src/* directory.

The goal of software emulation (*sw_emu*) is to ensure the functional correctness of the host program and kernels. Even if you have already verified the correctness of the kernel code using the Vitis HLS C simulation (section 3.6), there is no validation of the Host code and its interaction with the kernel code. Software emulation provides a purely functional execution, without any modelling of timing delays, or latency; it does not give any indication of the accelerator performance. The kernel code is compiled and running natively (in x86). The host code is cross-compiled to the Arm processor and running in an emulator (QEMU).

[3] It is called *boilerplate code*, i.e. this code is very similar to any OpenCL code running in the Host unit independent of the target platform (CPU, GPU, FPGA).

Go to the directory where the *Makefile* resides. You may have to edit the Makefile itself to modify the name of the Host file, the kernel file, and the name of the top-level kernel function. Then go to the Linux prompt and type:

```
make all TARGET=sw_emu DEVICE=zedboard_202020_1 HOST_ARCH=aarch32
EDGE_COMMON_SW=/opt/Xilinx/Vitis/2020.2/platforms/sysroot/ COMMON_REPO=../
```

Please note that the DEVICE argument specifies the accelerator platform for the build (Zedboard in this case). This is required because runtime features and the target platform are linked as part of the FPGA binary. The EDGE_COMMON_SW is the *sysroot* folder created from the petalinux process. *Sysroot* is used for cross-compilation. It provides the libraries to be linked when compiling applications for a target system. The COMMON_REPO points to the directory where the *common/* directory resides. You may need to change these paths according to your local directory structure.

The *sw_emu* compilation process calls the Arm *gcc* cross-compiler to compile the Host unit source code *src/lsal_host.c* to the Arm executable *lsal*. Then, it calls *v++* to compile the kernel source code *src/lsal.c* to the *lsal.xo* Xilinx object file, and, then, to link that file with other libraries and produce the *lsal.xlcbin.* After compiling and linking your kernel code to build the *.xclbin*, you need to package the device binary, along with any required supporting files, to build a package that can be run for software emulation. The *v++ -- package* step, or -p, packages the final product at the end of the v++ compile and link process. At the end, it creates a number of directories under *hw_baseline*. See Appendix C for more information on the *sw_emu* Build process.

The output of the process that we are interested in, will be inside the *package.sw_emu* directory. Launch the QEMU emulation environment by running the *launch_sw_emu.sh* script:

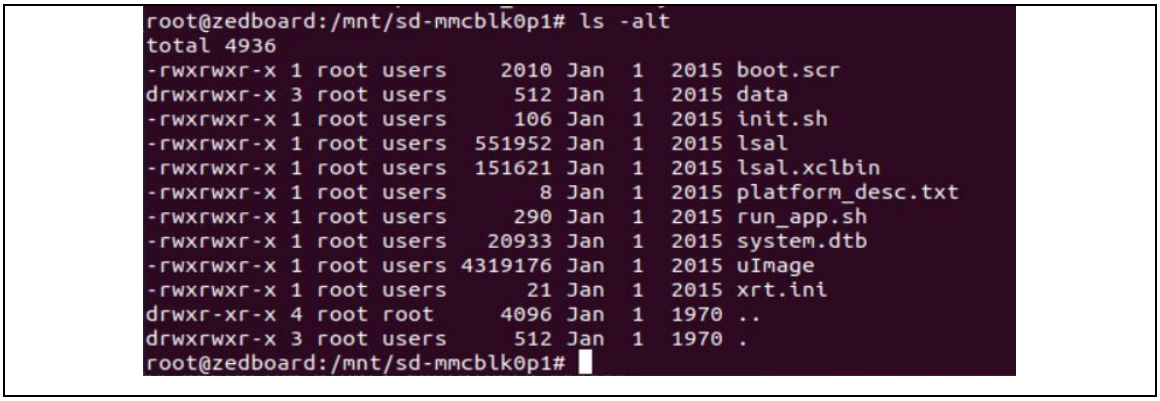
```
./launch_sw_emu.sh
```

This script will invoke the QEMU providing the same environment as if the application were executed in Zedboard. When petalinux boots, use these credentials to login:

```
user: root
```

```
password: root
```

Petalinux will automatically mount the boot partition to the directory */mnt/sd-mmcblk0p1*. It corresponds to the BOOT partition and contains all the files needed. The directory contains a number of files such as the two binaries (*lsal* for the Host unit, and *lsal.xclbin* for the kernel), the *uImage* which is the linux kernel image, the *system.dtb* which is the device tree, etc.



Move to this directory, and execute the application using the *run_app.sh* script that runs the application, after it exports the `XILINX_XRT=/usr` variable. You may have to edit the script to change the invocation of the application (for example, to provide the values N and M).

```
cd /mnt/sd-mmcblk0p1
./run_app.sh
```

The Host code is invoked and is executed under Arm QEMU (Quick Emulator), a generic and open source machine emulator. Xilinx provides a customized QEMU model that mimics the Arm-based processing system present on Zynq-7000 SoC devices. The QEMU model provides the ability to execute CPU instructions in almost real-time without the need for real hardware (see chapter 17 of [4] for more information on QEMU). The developer can check the results and validate the correct functionality. You can exit QEMU by pressing *Cntl+A* and then *X*.

Alternatively, you can run the application manually from the QEMU prompt as follows:

```
export XCL_EMULATION_MODE=sw_emu

export XILINX_XRT=/usr

./lsal ./lsal.xclbin 32 655536  # lsal.xclbin contains the kernel bitstream
```

The execution of the Host code will (hopefully!) print out a message that the kernel results are correct. Otherwise, you must go back to the Host code and make all appropriate corrections. You may ignore the timing information printed out. Software emulation is not time accurate. More information on software emulation can be found in Chapter *Running Emulation* of [4].

3.8. Execution in Zedboard

All previous steps are really preparations for running your code in real hardware. And now comes prime time!

Go to the directory where the *Makefile* resides. You may have to edit the *Makefile* itself to modify the name of the Host file, the kernel file, and the name of the top-level kernel function. Then go to the Linux prompt and type:

```
make all TARGET=hw DEVICE=zedboard_202020_1 HOST_ARCH=aarch32
EDGE_COMMON_SW=/opt/Xilinx/Vitis/2020.2/platforms/sysroot/ COMMON_REPO=../
```

Please note that the DEVICE argument specifies the accelerator platform for the build (Zedboard in this case). This is required because runtime features and the target platform are linked as part of the FPGA binary. The EDGE_COMMON_SW is the *sysroot* folder created from the petalinux process. *Sysroot* is used for ccross-compilation It provides the libraries to be linked when compiling applications for a target system. The COMMON_REPO points to the directory where the *common/* directory resides. You may need to change these paths according to your local directory structure.

The *hw* compilation process calls the Arm gcc cross-compiler to compile the Host unit source code *src/lsal_host.c* to the Arm executable *lsal*. Then, it calls *v++* to compile the kernel source code *src/lsal.c* to the *lsal.xo* Xilinx object file, and, then, to link that file with other libraries and produce the *lsal.xlcbin.* Note that the compilation step is time-consuming since it invokes the synthesis, placement and routing Vivado tools to produce the final bitstream. After compiling and linking your kernel code to build the *.xclbin*, you need to package the device binary, along with any required supporting files, to build a package that can run in Zedboard. The *v++ -- package* step, or *-p*, packages the final product at the end of the v++ compile and link process. At the end, it creates a number of directories under *hw_baseline*. The output of the process that we are interested in, will be inside the *package.hw/* directory. See Appendix C for more information on the *hw* Build process.

- First time we run in Zedboard.

  We need to flash the SD card with a) the kernel image, and the device tree, and b) the *BOOT.bin* (boot loader which also contains the *system.bit* file) and the application executables. We use the *package.hw/sd_card.img* file that contains all the pertinent files to flash the SD card. We write the SD card image *sd_card.img* to the SD card using various tools to do this, such as *Etcher* for Windows or the *dd* command on Linux.

  ```
  sudo dd if=package.hw/sd_card.img of=/dev/sda
  ```

  Please note that *sda* device is indicative, and you must use the one that corresponds to your SD Card in your local system. You can find the device name by using the Linux shell command `df -h`.

  The process above will flash the whole SD Card and will create a) the root filesystem partition that contains the petalinux filesystem, and b) the BOOT partition with the *BOOT.bin* bootloader and the application executables (together with some scripts). The executables are *lsal* (host executable) and *lsal.xclbin* (kernel bitstream).

- Subsequent runs in Zedboard (kernel image and device trees already in the SD card).

  We only will need to copy the contents of the *package.hw/sd_card* directory, inside the BOOT partition of the SD card.

Make sure that you have the correct jumper settings to be able to boot from the SD Card as shown in section 3.3. After you insert the SD Card in Zedboard, switch it on and wait for the command prompt to show Zynq>.

Then enter *boot* and wait for the login prompt.

Use these credentials:

```
user: root
password: root
```

You will be logged in to petalinux.

Petalinux will automatically mount the boot partition to the directory */mnt/sd-mmcblk0p1*. It corresponds to the BOOT partition and contains all the files needed. Move to this directory, and execute the application using the *run_app.sh* script that runs the application, after it exports the `XILINX_XRT=/usr` variable. You may have to edit the script to change the invocation of the application (for example, to provide the values N and M).

```
./run_app.sh
```

The code will execute in Zedboard and will report the execution time.

3.9. Profile using the Vitis Analyzer

The *Vitis Analyzer* is a utility that allows you to view and analyze the reports generated while building and running the application. It is intended to let you review reports generated by both the Vitis compiler when the application is built, and the Xilinx Runtime (XRT) library when the application is executed in Zedboard. The *Vitis Analyzer* can be used to view reports from both the v++ command line flow and the Vitis integrated design environment (IDE).

To trace the Host Application OpenCL API and some limited device side (kernel) activity, edit the *xrt.ini* file as follows:

```
[Debug]
profile=true
timeline_trace=true
```

After the build of the *xclbin* and the sd_card, the *xrt.ini* file is copied in the *sd_card/* directory and will be copied to the SD card. No other change is required for a basic Vitis analysis. Then, during the Zedboard run, three new files are automatically generated and stored in the SD card: *lsal.xclbin.run_summary*, *profile_summary.csv*, and *timeline_trace.csv*. These files should be copied back to the PC (*vitis_analyzer/* directory in Figure 3) and can provide information to the Vitis analyzer.

You will launch the tool using the *vitis_analyzer* command from the Linux prompt.

```
vitis_analyzer
```

To also trace the device and to capture data traffic between the host and kernel, use the following *xrt.ini*

```
[Debug]
profile=true
timeline_trace=true
```

```
[profile]
data_transfer_trace=coarse
```

On top of that, the *v++* link call should also include the flags VPP_PROFILE_FLAGS = *-g --profile.data all:all:all* in Makefile.

For more information on Vitis Analyzer, check the chapter *Using the Vitis Analyzer* [4]. For more information on the *xrt.ini* configuration file, check *https://xilinx.github.io/XRT/2020.2/html/xrt_ini.html*

Appendix A. Roofline Model

The roofline model is an intuitive 2D visual performance chart used to provide performance estimates of a given compute kernel or application running on multi-core, many-core, or accelerator processor architectures, by showing inherent hardware limitations, and potential benefits and priority of optimizations. The model is used to answer questions, such as what the maximum achievable performance in the current hardware resources is if the application works optimally on current hardware resources and whether the application is memory-bound or compute-bound.

The roofline chart plots an application's achieved performance and arithmetic intensity against the machine's maximum theoretic performance. First, roofline analysis requires that the developer computes the arithmetic intensity (AI) of a given implementation, which in essence represents the ratio between the operations that are executed and the total memory footprint in bytes. The arithmetic intensity can be determined statically by just counting the number of operations in the source code (when this is possible), or, dynamically, by instrumenting the source code to count the number of operations executed and the number of bytes transferred between the memory and the CPU. Arithmetic intensity (x-axis) is measured in number of operations (whatever the basic operation is) per byte transferred between CPU and memory.

Second, roofline analysis requires that the developer measures the execution time in GINTOPs/sec (Giga Integer Operations per Second). This is the y-axis.

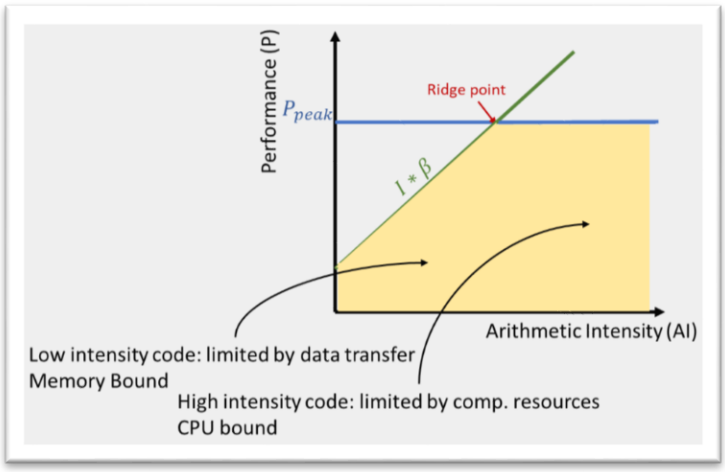


Figure 4. The basic (or naïve) roofline model.

Figure 4 shows that the roofline chart consists of two separate regions which show the performance bottleneck according to the arithmetic intensity of the application. If the AI is large, then the only bottleneck is the peak performance P_{peak} of the platform and the application is compute-bound. P_{peak} is a function of the available compute resources, such as the number of functional units, number of cores, etc., and, obviously, can never be surpassed by any application running on the platform. On the other hand, if AI is small, the performance is constrained by the memory bandwidth β. This can be formulated as follows:

$$P \leq \min\left(P_{peak}, AI * \beta\right)$$

The line $AI * \beta$ is the performance bottleneck for memory-bound implementations. Note that the two lines P_{peak} and $AI * \beta$ depend only on the platform, whereas AI depends only on the application and its specific implementation.

Figure 5 shows the effect of successful optimizations according to the roofline model. The objective is to reduce the dependence of execution time on memory bandwidth (push to the right) and to improve task-level and data-level parallelism (push upwards). Ideally, the developer would want to approach the Ridge Point as close as possible.

The roofline chart of our own LSAL implementation using *Intel Advisor* is shown in Figure 6. *Intel Advisor (*which can be downloaded for free[4]) automates the roofline model and provides several profiling data such as performance, bandwidth, traffic between caches and the memory, etc. You should use the Intel Advisor to analyze your x86 SLAL implementation.

[4] https://software.intel.com/content/www/us/en/develop/articles/oneapi-standalone-components.html#advisor

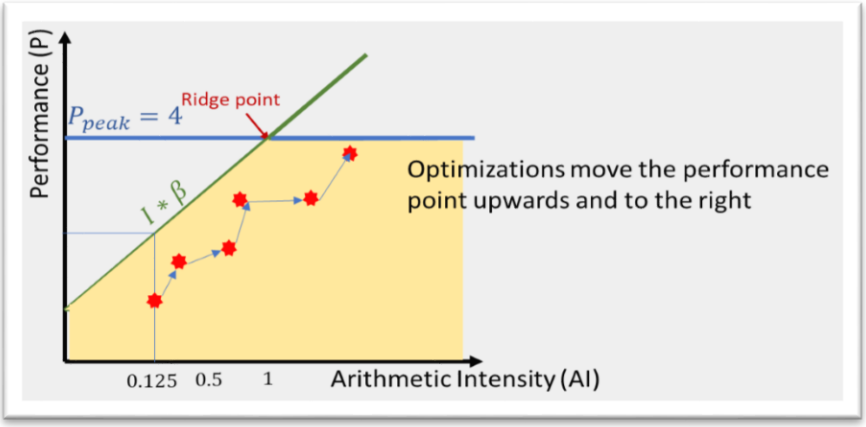


Figure 5. Optimizations can be depicted as continuous upwards and rightwards movement in the roofline model.

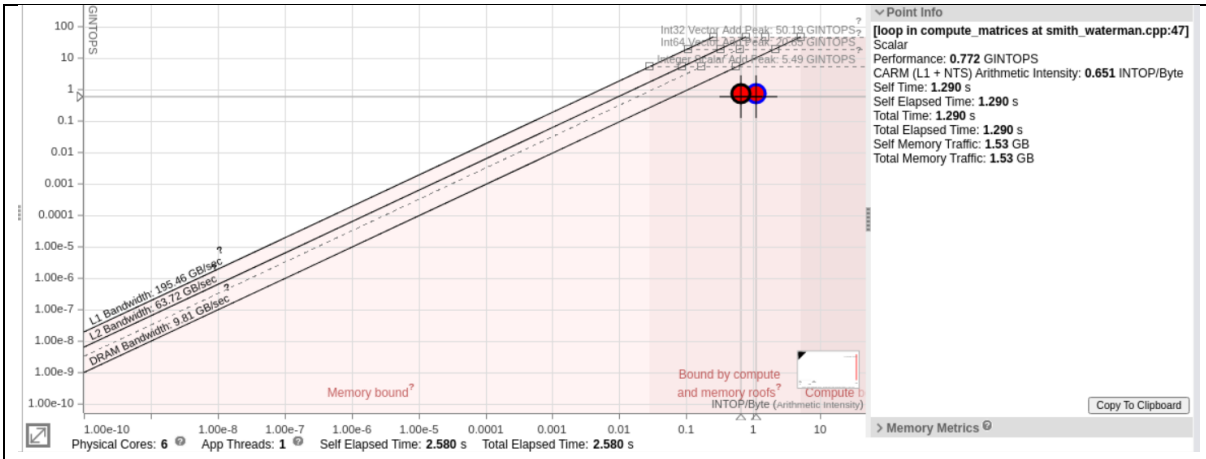


Figure 6. Roofline chart of the LSAL compute kernel executed on an x86 6-core CPU as shown by *Intel Advisor* tool. The two red dots correspond to the execution of the *compute_matrices* function for the L1-based arithmetic intensity (dot to the left) and the DRAM-based arithmetic intensity (dot to the right). The L1-based AI is equal to $\frac{\#INT\ Ops}{\#bytes\ read\ from\ L1\ DCache}$, whereas the DRAM-based AI is equal to $\frac{\#INT\ Ops}{\#bytes\ read\ from\ DRAM}$. Since the $\#bytes\ read\ from\ DRAM$ in a CPU is typically much lower than the $\#bytes\ read\ from\ L1\ DCache$, the DRAM-based AI is usually larger. As we can see, multiple values can characterize AI in a CPU with a memory hierarchy. Note also the multiple horizontal rooflines which correspond to the maximum performance for Integer scalar 32/64 bits and Int32 vector.

Appendix B. Vitis HLS Optimization directives

In the Vitis core development kit, the kernel code is generally the compute-intensive part of the algorithm and meant to be accelerated on the FPGA. During the runtime, the C/C++ kernel executable (converted into a hardware accelerator) is called through the host code executable (executed on the Arm processor). This appendix presents the most basic directives *(#pragmas*) used in Vitis HLS kernel code for performance optimization and kernel interface. For a more thorough treatment, you should consult [3,5].

IMPORTANT: Because the host code and the kernel code are developed and compiled independently, there could be a name mangling issue if one is written in C and the other in C++. To avoid this issue, wrap the whole kernel code with the extern "C" linkage.

```
extern "C" {
 void kernel_function( … ){
  ..
  return;
 }
}
```

Interfaces

The parameters of the top-level C/C++ function of the kernel are synthesized into interfaces and ports that group multiple signals to encapsulate the communication protocol between the kernel and structures external to the kernel. Vitis HLS defines interfaces automatically, using industry standards to specify the protocol used. The type of interfaces that Vitis HLS creates depends on the data type and direction of the parameters of the top-level function, the target flow for the active solution (Vitis vs. Vivado kernel flow), and the default interface configuration settings as specified by INTERFACE pragmas or directives.

Two types of data transfer occur between global memory and the kernels on the FPGA. Data are transferred between the host CPU and the accelerator through global memory banks. Scalar data is passed directly from the host to the kernel (see example below).

```
void kernel_function (int *input, int *output, int width, int height) {

#pragma HLS INTERFACE m_axi port=input offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=input bundle=control

#pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=output bundle=control

#pragma HLS INTERFACE s_axilite port=width bundle=control
#pragma HLS INTERFACE s_axilite port=height bundle=control

#pragma HLS INTERFACE s_axilite port=return bundle=control
      ...
```

Arrays are processed by the kernel, often in a large volume, and should be transferred through the global memory banks on the FPGA board (master AXI interfaces). The host machine transfers a large chunk of data to one or more global memory bank(s). The kernel accesses the data from those global memory banks, preferably in burst mode. After the kernel finishes the computation, the resulting data is transferred back to the host machine through the global memory banks. When writing the kernel interface description, pragmas are used to denote the interfaces going to and coming from the global memory banks.

In the example above, there are two data pointer interfaces. The *input* and *output* interfaces that are connected to the global memory bank are specified in C code by using *HLS INTERFACE m_axi* pragmas as shown in the figure. Also, each memory-mapped interface port needs to have a second interface *pragma*, declared as a scalar input through *s_axilite*, using the same port name and *bundle=control*.

The bundle keyword on the pragma HLS interface defines the name of the port. The system compiler will create a port for each unique bundle name. When the same bundle name is used for different interfaces, this results in these interfaces being mapped to the same port.

Scalar inputs are typically control variables whose value is directly loaded from the host machine. They can be thought of as programming data or parameters under which the main kernel computation takes place. These kernel inputs are write-only from the host side.

You should check chapter 15 of [3] for more details.

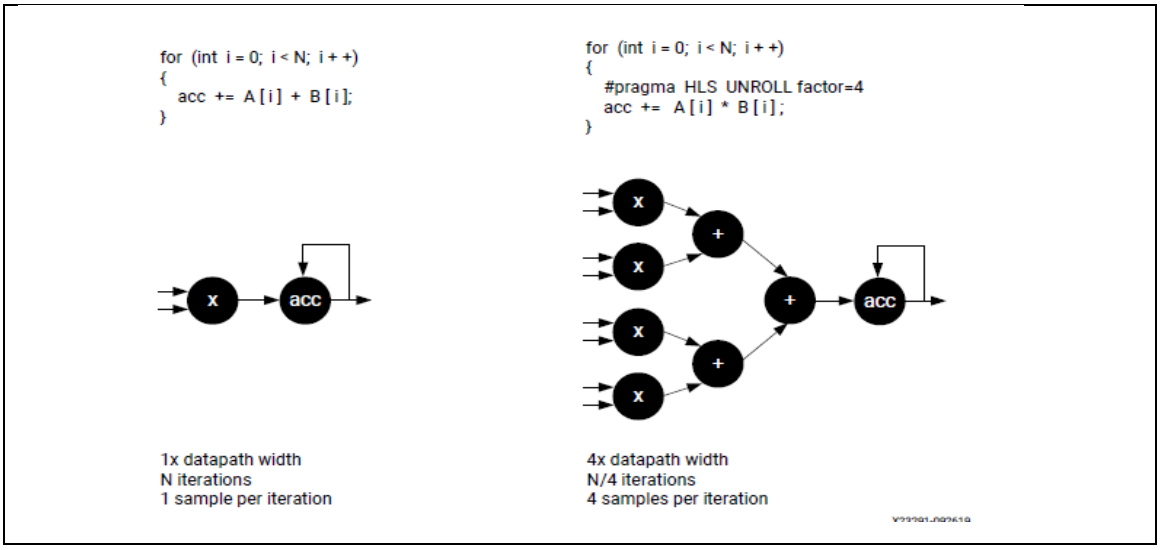
IMPORTANT: Currently, the Vitis core development kit supports only one control interface bundle for each kernel. Therefore, the *bundle= name* should be the same for all scalar data inputs, including the function return. In the preceding example, *bundle=control* is used for all scalar inputs. The above requirements are necessary for the Vitis compiler.

Computational Optimizations

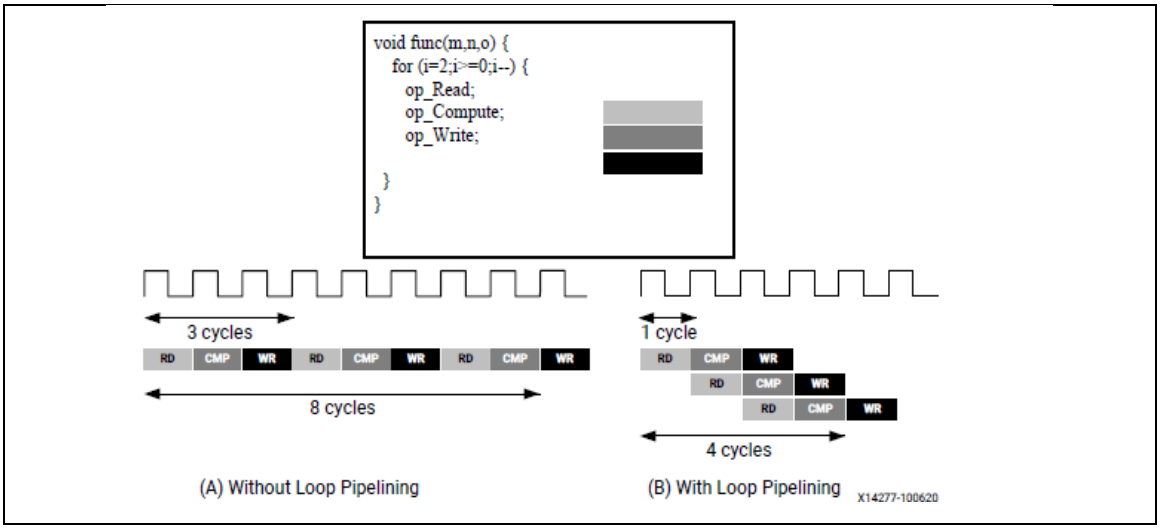
Vitis HLS provides several *pragmas* that can be used by the developer to produce a micro-architecture that satisfies the desired performance and area goals (check chapter 16 of [3] and chapters 4 and 20 of [5] for more details). Most of these *pragmas* are applied at the loop level since most of the execution time is expended in loops. We should also note that by default Vitis HLS always tries to reduce iteration latency, but it will only enter execution of a new iteration when it terminates the current iteration. We should emphasize also that the most substantial optimizations are achieved by combining the HLS *pragmas* with restructuring the source code of the kernel (as you will do in lab3) to make the code more hardware friendly.

Loop Unrolling unwinds a loop, allowing multiple iterations of the loop to be executed together, reducing the loop's overall trip count. Loop unrolling increases the size of the loop body, which allows Vitis HLS to apply more aggressive optimization algorithms to reduce the latency of each loop iteration. The following figure shows that loop unrolling can also be used to eliminate loop carried dependencies (through the *acc* in this case).

```
#pragma HLS UNROLL factor = 4 /* Unroll a loop body 4 times */
```



Loop pipeline allows operations to happen concurrently: each execution step does not have to complete all operations before it begins the next operation. Pipelining is applied to functions and loops. The throughput improvements in function pipelining are shown in the following figure.



Functions or loops are pipelined using the PIPELINE directive.

```
#pragma HLS PIPELINE II=1 /* Attempt to pipeline with II=1 */
```

The directive is specified in the region that constitutes the function or loop body. The initiation interval II defaults to 1 if not specified but may be explicitly specified. Pipelining is applied only to the specified region and not to the hierarchy below. However, all loops in the hierarchy below are automatically unrolled. Any sub-functions in the hierarchy below the specified function must be pipelined individually. Note that Vitis HLS (as opposed to the older Vivado HLS) attempts to pipeline any loop with II=1 by default. You can disable this optimization with the following *pragma*:

```
#pragma HLS PIPELINE off /* Disable pipelining */
```

Unroll and pipelining can be combined to further increase performance.

Data Types. The data types used in a C/C++ function that is synthesized to an RTL implementation, impact the precision, the area, and the performance of the RTL design. The data types used for variables determine the size of the operators required and therefore the area and performance of the RTL. However, in a lot of cases, the standard C/C++ data types have a much larger bitwidth than what is strictly required for hardware computation. For example, variable *idx* in the following code in reality only requires 7 bits instead of 32 bits. This precision reduction translates into large area

```
for (int idx = 0; idx < 100; idx++) {
```

savings and potentially performance improvements in an FPGA hardware implementation.

Vitis HLS supports arbitrary precision fixed-point arithmetic. The developer should trade-off accuracy versus area when deciding to switch to arbitrary arithmetic. As another example, the following code:

```
ap_int<512> *input
```

defines input as a pointer to an array of 512-bit elements. This can be used as a definition of an input port to an accelerator since the Vitis compiler will automatically generate a 512-bit port to the accelerator, hence, increasing bandwidth to the main memory dramatically. The expression:

```
short q = input[23].range(127,112)
```

defines the 16-bit quantity of bits [112, 127] of *input[23]*.

For more information on arbitrary data types, you should read chapter 14 of [3] and/or chapter 7 of [4]. There is also an HLS tutorial in arbitrary precision arithmetic in [5].

Memory Bandwidth Optimizations

Array Partitioning. Loop optimizations such as unrolling, and software pipelining may drastically increase demand for data bandwidth from/to the memory. Aggressive computational optimizations are useless (and, worse, detrimental for resources) if the memory cannot sustain this data bandwidth in every clock cycle. The Vitis compiler maps large arrays to the block RAM in the fabric. These block RAMs have a maximum of two access points or ports. This can limit the performance of the application as all the elements of an array cannot be accessed in parallel when implemented in hardware. The bandwidth can be improved by splitting the array into multiple smaller arrays (multiple block RAMs or even individual registers), effectively increasing the number of ports.

The following *pragma* partitions the first dimension of array A into 64 32-bit registers. This allows the compiler to schedule code that can utilize all 64 elements of the array in a single cycle effectively increasing bandwidth from 2 to 64 elements per clock cycle. The designer should be very careful with this optimization because it tends to deplete FPGA from resources such as registers and LUTs.

```
int A[64];
#pragma HLS ARRAY_PARTITION variable=A dim=1 complete
```

Burst accesses to memory. The first read or write request to global memory is expensive, but subsequent contiguous operations are not. Transferring data in bursts hides the memory access latency and improves bandwidth usage and efficiency of the memory controller. Atomic accesses to global memory must always be avoided unless absolutely required. The load and store functions should be coded to always infer bursting transactions. This can be done using a *memcpy* operation.

The *memcpy* operation is used to transfer array data (*int *a* in the example below) from the global DRAM memory to internal block RAMs in the FPGA (*buff[]).* Accessing *buff[i]* instead of *a[i]* in the loop greatly reduces latency and increases bandwidth since block RAMs are internal to the FPGA.

```
void example (int *a){
// Port a is assigned to an AXI4 master interface
#pragma HLS INTERFACE m_axi port=a
#pragma HLS INTERFACE s_axilite port=return

int i;
int buff[50];

 //memcpy creates a burst access to memory
 memcpy(buff, (const int*)a, 50*sizeof(int));
 for(i=0; i < 50; i++){
  buff[i] = buff[i] + 100;
 }
 memcpy((int *)a,buff,50*sizeof(int));
}
```

Appendix C. Introduction to Vitis OpenCL Execution Flow

The Vitis application acceleration development flow provides a framework for developing and delivering FPGA-accelerated applications using standard programming languages for both software and hardware components. The software component, or host program, is developed using C/C++ to run on the Arm processor with OpenCL API calls to manage runtime interactions with the accelerator. The hardware component, or kernel, can be developed using C/C++, OpenCL, or RTL (Verilog/VHDL). The Vitis software platform accommodates various methodologies, letting you start by developing either the application or the kernel.

We will focus on the details of using this framework for embedded systems, especially for our target device, the Zedboard.

Execution Model

In the Vitis core development kit, an application program is split between a host application and hardware-accelerated kernels with a communication channel between them. The host program, written in C/C++ and using API abstractions like OpenCL, runs on a host processor (an Arm processor for embedded platforms like Zedboard), while hardware-accelerated kernels run within the programmable logic (PL) region of a Xilinx device.

The API calls, managed by Xilinx Runtime (XRT), are used to process transactions between the host program and the hardware accelerators. Communication between the host and the kernel, including control and data transfers, occurs across the AXI bus for embedded platforms. While control information is transferred between specific memory locations in the hardware, global memory is used to transfer data between the host program and the kernels. Global memory is external to the FPGA device and is implemented using DDR3 technology. On the other hand, BRAM is internal to the FPGA device.

For instance, in a typical application, the host (i.e., the Arm CPU) first initializes/prepares data to be operated on by the kernel and then, stores them in global memory. The kernel subsequently operates on the data, storing results back to the global memory. Upon kernel completion, the host reads the data and continues with other tasks (which may include calling the accelerator more times or calling other accelerators).

The target platform contains the FPGA accelerated kernels, global memory, and direct memory access (DMA) for memory transfers. Kernels can have one or more global memory interfaces and are programmable. The Vitis execution model can be broken down into the following steps:

1. The host program prepares the input data needed by a kernel and stores them in the global memory.
2. The host program sets up the kernel with its input parameters.
3. The host program triggers the execution of the kernel function on the FPGA.
4. The kernel performs the required computation while reading data from global memory, as necessary.

5. The kernel notifies the host unit that it has completed its task.
6. The host unit reads the data back to global memory.
7. The host program uses the data and continues operations with other tasks.

Build Process

The Vitis core development kit offers all the features of a standard software development environment:

- Compiler or cross-compiler for host applications running on the Arm processors.
- Cross-compilers for building the FPGA binary.
- Debugging environment to help identify and resolve issues in the code.
- Performance profilers to identify bottlenecks and help optimize the application.

The build process follows a standard compilation and linking process for both the host program and the kernel code. As shown in the following figure, the host program is built using the GNU C++ compiler (g++) or the GNU C++ Arm cross-compiler for MPSoC-based devices, like the Zedboard. The FPGA binary is built using the Vitis compiler (v++). The v++ compiler invokes the Synthesis, Placement, and Routing programs as we saw in lab1.

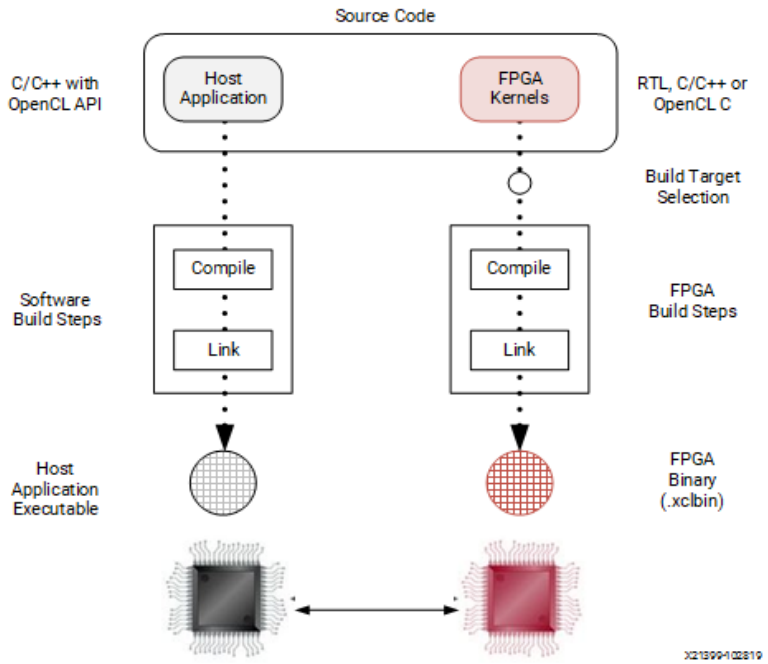


Figure 7. Software/Hardware Build Process

Host Program Build Process. The main application is compiled and linked with the **g++** compiler, using the following two-step process:

1. Compile any required source code into object files (.o).
2. Link the object files (.o) with the XRT shared library to create the executable (.elf).

FPGA Binary Build Process. Kernels can be described in C/C++, or OpenCL C code, or can be created from packaged RTL designs. Each hardware kernel is independently compiled to a Xilinx object (.xo) file. Xilinx object (.xo) files are, then, linked with the hardware platform to create an FPGA binary file *(.xclbin*) that is loaded into the Xilinx device on the target platform. The *.xlcbin* file contains the FPGA bitstream. The key to building the FPGA binary is to determine the build target you are producing.

Build Targets. The Vitis compiler build process generates the host program executable and the FPGA binary *(.xclbin*). The nature of the FPGA binary is determined by the build target.

- When the build target is software or hardware emulation, the Vitis compiler generates simulation models of the kernels in the FPGA binary. These emulation targets let you build, run, and iterate the design over relatively quick cycles, debugging the application and evaluating performance.
- When the build target is the hardware system, the Vitis compiler generates the *.xclbin* for the hardware accelerator, using the Vivado Design Suite to run synthesis and implementation. It uses these tools with predefined settings proven to provide good quality results. Using the Vitis core development kit does not require knowledge of these tools; however, hardware-savvy developers can fully leverage these tools and use all the available features to implement kernels.

The Vitis compiler provides three different build targets: two emulation targets used for debugging and validation purposes, and the default hardware target used to generate the actual FPGA binary:

- Software Emulation (sw_emu): Both the host application code and the kernel code are compiled to run on the host processor (x86 in our case). This allows iterative algorithm refinement through fast build-and-run loops. This target is useful for identifying syntax errors, performing source-level debugging of the kernel code running together with the application, and verifying the functionality of the system.
- Hardware Emulation (hw_emu): The kernel code is synthesized into a hardware model (RTL) by the Vivado tool and runs in a dedicated simulator (Vivado simulator). This build-and-run loop takes much longer but provides a detailed, cycle-accurate view of kernel activity. This target is useful for testing the functionality of the logic that will go into the FPGA and getting initial performance estimates. Please be aware that, this emulation needs a significant time to complete, especially for large data sets. For that reason, it will not be used in this lab.

- System (hw): The kernel code is compiled into a hardware model (RTL) and then implemented on the FPGA, resulting in a binary that will run on the actual FPGA.

Host Application

In the Vitis core development kit, the host code is written in C or C++ language using the industry-standard OpenCL API. The Vitis core development kit provides an OpenCL 1.2 embedded profile conformant runtime API. In general, the structure of the host code can be divided into three sections:

1. Setting up the environment.
2. Core command execution including executing one or more kernels.
3. Post-processing and release of resources.

Setting Up the OpenCL Environment. The host code in the *Vitis* core development kit follows the OpenCL programming paradigm. To set the environment properly, the host application needs to initialize the standard OpenCL structures: *target platform, devices, context, command queue,* and *program.* The following code snippets are taken from the *lsal_host.cpp* file.

Platform. The OpenCL API call *clGetPlatformIDs* is used to discover the set of available OpenCL platforms for a given system. Thereafter, *clGetPlatformInfo* is used to retrieve specific information about the platform `&platform_id` such as platform name and platform vendor. It is always a good coding practice to use error checking after each of the OpenCL API calls. This can help to debug and improve productivity when you are debugging the host and kernel code in the emulation flow, or during hardware execution.

```
err = clGetPlatformIDs (1, &platform_id, NULL);
if (err != CL_SUCCESS) {
       printf("Error: Failed to find an OpenCL platform!\n");
       return EXIT_FAILURE;
}
printf("GET platform vendor \n");
err = clGetPlatformInfo(platform_id, CL_PLATFORM_VENDOR, 1000,
                    (void *) cl_platform_vendor, NULL);
if (err != CL_SUCCESS) {
        printf("Error:clGetPlatformInfo(CL_PLATFORM_VENDOR) failed!\n");
    return EXIT_FAILURE;
}
printf("GET platform name \n");
err = clGetPlatformInfo(platform_id, CL_PLATFORM_NAME, 1000,
                    (void *) cl_platform_name, NULL);
if (err != CL_SUCCESS) {
       printf("Error: clGetPlatformInfo(CL_PLATFORM_NAME) failed!\n");
       return EXIT_FAILURE;
}
```

Figure 8. OpenCL Platform Initialization.

Devices. After a Xilinx platform is found, the host code needs to identify the corresponding Xilinx devices. The following code demonstrates finding the *device_id* using the API *clGetDeviceIDs*.

```
int fpga = 1;
printf("GET device FPGA is %d \n", fpga);
err = clGetDeviceIDs(platform_id, fpga ? CL_DEVICE_TYPE_ACCELERATOR :
CL_DEVICE_TYPE_CPU, 1, &device_id, NULL);
if (err != CL_SUCCESS) {
       printf("Error: Failed to create a device group!\n");
       return EXIT_FAILURE;
}
```

Figure 9. Device Initialization.

Context. The *clCreateContext* API is used to create a context that contains a Xilinx device that will communicate with the host machine.

```
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

In the code example, the *clCreateContext* API is used to create a context that contains one Xilinx device. Xilinx recommends creating only one context per device.

Command Queues. The *clCreateCommandQueue* API creates one or more command queues for each device. The FPGA can contain multiple accelerators, which can be either the same kernel (in

one or more instances) or different kernels. When developing the host application, there are two main programming approaches to execute kernels on a device:

1. Single out-of-order command queue: Multiple kernel executions can be requested through the same command queue. Xilinx Run Time System (XRT) dispatches kernels as soon as possible, in any order, allowing concurrent kernel execution on the FPGA.
2. Multiple in-order command queues: Each kernel execution will be requested from different in-order command queues. In such cases, XRT dispatches kernels from the different command queues, improving performance by running them concurrently on the device.

The following is an example of standard API calls to create in-order and out-of-order command queues.

```
commands = clCreateCommandQueue(context, device_id, CL_QUEUE_PROFILING_ENABLE,
&err);
if (!commands) {
       printf("Error: Failed to create a command commands!\n");
       return EXIT_FAILURE;
}
```

Program. As described in Build Process, the host and kernel code are compiled separately to create separate executable files: the host program executable and the FPGA binary *(.xclbin*). When the host application runs, it must load the *.xclbin* file using the *clCreateProgramWithBinary* API. In Figure 10, example code shows how the standard OpenCL API is used to build the program from the *.xclbin* file.

```
unsigned char *kernelbinary;
char *xclbin = argv[1];

printf("INFO: loading xclbin %s\n", xclbin);

int size=load_file_to_memory(xclbin, (char **) &kernelbinary);

size_t size_var = size;

cl_program program = clCreateProgramWithBinary(context, 1, &device_id, &n,
             (const unsigned char **) &kernelbinary, &binary_status, &err);

// Function
int load_file_to_memory(const char *filename, char **result)
{
 uint size = 0;
 FILE *f = fopen(filename, "rb");
 if (f == NULL) {
  *result = NULL;
  return -1; // -1 means file opening fail
 }
 fseek(f, 0, SEEK_END);
 size = ftell(f);
 fseek(f, 0, SEEK_SET);
 *result = (char *)malloc(size+1);
 if (size != fread (*result, sizeof(char), size, f)) {
  free(*result);
  return -2; // -2 means file reading fail
 }
 fclose(f);
 (*result)[size] = 0;
 return size;
}
```

Figure 10. Create program using the loaded hardware binary file.

The example performs the following steps:

1. The kernel binary file, *.xclbin*, is passed in from the command line argument, **argv[1].**

   TIP: Passing the *.xclbin* through a command line argument is one approach. You can also hardcode the kernel binary file in the host program, or define it with an environment variable, read it from a custom initialization file, or another suitable mechanism.

2. The **load_file_to_memory** function is used to load the file contents in the host machine memory space.

3. The *clCreateProgramWithBinary* API is used to complete the program creation process in the specified context and device.

Executing Commands in the FPGA. Once the OpenCL environment is initialized, the host application is ready to issue commands to the device and interact with the kernels. These commands include:

1. Setting up the kernels.
2. Buffer transfers to/from the FPGA.
3. Kernel execution on FPGA.
4. Event synchronization.

Setting Up Kernels. After setting up the OpenCL environment, such as identifying devices, creating the context, command queue, and program, the host application should identify the kernels that will execute on the device, and set up the kernel arguments.

The OpenCL API *clCreateKernel* should be used to access the kernels contained within the *.xclbin* file (the "program"). The *cl_kernel* object identifies a kernel in the program loaded into the FPGA that can be run by the host application. The following code example identifies the *compute_matrices* kernel defined in the loaded program.

```
kernel = clCreateKernel(program, "compute_matrices", &err);
if (!kernel || err != CL_SUCCESS) {
       printf("Error: Failed to create compute kernel!\n");
       return EXIT_FAILURE;
}
```

Figure 11. Creating a kernel object that corresponds to the kernel of the loaded program.

Creating buffers. Any non-scalar data type must be passed to kernels buffer objects. You must first create a buffer object using *clCreateBuffer*. It returns a *cl_mem* object, which can be used as a kernel argument. For this lab, we will be using different *clCreateBuffer* arguments, alongside with *clEnqueueMapBuffer***.** An example is shown in Figure **12**.

```
cl_mem input_query;
cl_mem input_database;
input_query = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(char) * N,
       NULL, NULL);
input_database = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(char) * M,
       NULL, NULL);
```

Figure 12. Creating buffer objects in the global memory. The global memory is the main memory of the accelerator. The objects will be read-only by the accelerators.

Buffer Transfer to/from the FPGA Device. Interactions between the host program and hardware kernels rely on transferring data to and from the global memory in the device. The method to send data back and forth from the FPGA is using *clCreateBuffer*, *clEnqueueWriteBuffer*, and *clEnqueueReadBuffer* commands.

```
err = clEnqueueWriteBuffer(commands, input_query, CL_TRUE, 0,
              sizeof(char) * N, query, 0, NULL, NULL);
err = clEnqueueWriteBuffer(commands, input_database, CL_TRUE, 0,
              sizeof(char) * M, database, 0, NULL, NULL);
err = clEnqueueWriteBuffer(commands, output_similarity_matrix, CL_TRUE, 0,
              sizeof(int) * matrix_size, similarity_matrix, 0, NULL, NULL);
err = clEnqueueWriteBuffer(commands, output_direction_matrix, CL_TRUE, 0,
              sizeof(short) * matrix_size, direction_matrix, 0, NULL, NULL);
```

Figure 13. The *clEnqueueWriteBuffer* enqueues commands to write to a buffer object from host memory. For example, the first *clEnqueueWriteBuffer* enqueues a transfer from the *query* array in the Host memory (Arm memory) to the *input_query* object in the Global memory (DRAM accelerator memory). In the Zedboard case, these two memory spaces coincide. For brevity, we do not include error checking.

Setting Kernel Arguments. In the Vitis software platform, two types of arguments can be set for *cl_kernel* objects:

1. Scalar arguments are used for small data transfers, such as constant or configuration type data. These are write-only arguments from the host application perspective, meaning they are inputs to the kernel.
2. Memory buffer arguments are used for large data transfers. The value is a pointer to a memory object created with the context associated with the program and kernel objects. These can be inputs to or outputs from the kernel.

Kernel arguments can be set using the *clSetKernelArg* command as shown below. The following example shows setting kernel arguments for the *compute_matrices* kernel.

Kernel Execution. Often the compute-intensive task required by the host application can be defined inside a single kernel, and the kernel is executed only once to work on the entire data range. Because there is an overhead associated with multiple kernel executions, invoking a single monolithic kernel can improve performance. Though the kernel is executed only once and works

```
err = 0;
printf("set arg 0 \n");
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input_query);
if (err != CL_SUCCESS) {
       printf("Error: Failed to set kernel arguments 0! %d\n", err);
       return EXIT_FAILURE;
}
```

Figure 14. An example of setting the value of kernel argument 0 (the pointer to *input_query*)

on the entire range of the data, the parallelism (and thereby acceleration) is achieved on the FPGA inside the kernel hardware. If properly coded, the kernel is capable of achieving parallelism by various techniques as we have seen in this document.

When the kernel is compiled to a single hardware instance (or compute unit, CU) on the FPGA, the simplest method of executing the kernel is using *clEnqueueTask* as shown below.

```
err = clEnqueueTask(commands, kernel, 0, NULL, &enqueue_kernel);
```

XRT schedules the workload, or the data passed through OpenCL buffers from the kernel arguments, and schedules the kernel tasks to run on the accelerator.

Waiting for events. All OpenCL enqueue-based API calls are asynchronous. These commands will return immediately after the command is enqueued in the command queue. To pause the host program to wait for results, or resolve any dependencies among the commands, an API call such as *clFinish* or *clWaitForEvents* can be used to block the execution of the host program. The following command waits for the termination of the *compute_matrices* execution. Note that the *enqueue_kernel* event has been defined in the previous *clEnqueueTask.*

```
clWaitForEvents(1, &enqueue_kernel);
```

Post-Processing and Cleanup. At the end of the host code, all the allocated resources should be released by using proper release functions. If the resources are not properly released, the Vitis core development kit might not able to generate a correct performance related profile and analysis report.

```
clReleaseMemObject(input_database);
clReleaseMemObject(input_query);
clReleaseMemObject(output_direction_matrix);
clReleaseMemObject(output_max_index);
clReleaseMemObject(output_similarity_matrix);
clReleaseProgram(program);
clReleaseKernel(kernel);
clReleaseCommandQueue(commands);
clReleaseContext(context);

free(query);
free(database);
free(similarity_matrix);
….
```

Figure 15. The cleanup stage

Summary. As discussed in earlier topics, the recommended coding style for the host program in the Vitis core development kit includes the following points:

1. Add error checking after each OpenCL API call for debugging purposes, if required.

2. In the Vitis core development kit, one or more kernels are separately compiled/linked to build the *.xclbin* (bitstream) file. The API *clCreateProgramWithBinary* is used to build the *cl_program* object from the kernel binary.
3. Use buffer for setting the kernel argument (*clSetKernelArg*) before any enqueue operation on the buffer.
4. Preferably use the out-of-order command queue for concurrent command execution on the FPGA.
5. Invoke the accelerator with *clEnqueueTask*.
6. Use event synchronization commands, *clFinish*, and *clWaitForEvents* to resolve dependencies of the asynchronous OpenCL API calls.
7. Release all OpenCL allocated resources when finished.

Appendix D. Vitis Compiler Configuration File

A configuration file can be used to specify the Vitis compiler options. A configuration file provides an organized way of passing options to the compiler by grouping similar flags together and minimizing the length of the **v++** command line. Some of the features that can be controlled through configuration file entries include:

- HLS options to configure kernel compilation.
- Connectivity directives for system linking such as the number of kernels to instantiate or the assignment of kernel ports to global memory.
- Directives for the Vivado Design Suite to manage hardware synthesis and implementation.

In general, any **v++** command option can be specified in a configuration file. However, the configuration file supports defining sections containing groups of related commands to help manage build options and strategies. For more information on the config file, see chapter 22 of [4].

Clock Frequency

The clock frequency that a kernel function will use is configurable. Most importantly, you must ensure that your design is synthesizable on a target frequency. You can explore these constrain during the Vivado HLS development stage, by setting the Clock Period at Synthesis Setting and checking the estimated clock frequency in the HLS synthesis report.

Normally, Vivado designs can freely use any clock frequency in specific ranges (e.g. 10 MHz – 600 MHz), and this option can be configured through Vivado configuration options and constraints. However, in a Vitis development environment, the available clock frequencies must be set at the platform creation stage, and be hard-coded into the platform. That means that the developer cannot pick any frequency, but only the ones that the platform provides. If no frequency is provided to

Vitis tools, it chooses a default clock frequency. The default frequency is configured at the platform creation stage.

For this lab, Zedboard has been set up to provide 2 clock frequencies:

- 100 MHz (Clock ID: 0)
- 142 MHz (Clock ID: 1)

You can choose the frequency before compilation, by modifying the **design.cfg** file. There are lines for each clock frequency for HLS, paired with a clock ID. You can comment on the active configuration and comment on the one that you would like to use. You can also verify the available clocks and clock ids by using the *platforminfo* tool by issuing the command *platforminfo zedboard_202020_1*.

Note: When your target is software emulation there is no need to configure the clock frequency. You can confirm that, if you view **design_emu.cfg** file that is used for emulation target.

Appendix E. Zedboard Petalinux with SD Card

There are many ways to boot the Zedboard and use the ARM CPU in different environments. In Lab 2 you explored the standalone usage of the ARM (bare metal), without any Operating System layer on top of the hardware. In this lab, you will work inside a lightweight Linux environment, specifically used for embedded applications that run on Xilinx boards, called Petalinux.

To use the capability to boot from the SD card image, the SD card must contain two partitions:

- BOOT Partition (FAT32) - Contains 4 necessary files
    1. *BOOT.BIN:* Contains all the necessary data that is initially loaded in the system, as well as the FPGA bitstream (see Appendix F).
    2. *System.dtb:* It is the device tree file that describes the hardware devices and the various interconnections between them.
    3. *uImage:* The Linux kernel image.
    4. *Boot.scr:* A helper script that contains information about the other boot files.
- RootFS Partition (ext4) – Contains the Petalinux root filesystem. There are various ways to store and load the RootFS in Petalinux, for example, to load it in RAM at boot time or a separate partition in the SD Card. In this lab, we prefer the latter.

We provide you with the necessary pre-built files to copy to the two partitions of the SD card, as a quick start to test and try the Petalinux environment. The files reside in */home/embedded/courses/ECE340_EmbeddedSystems/bootArtifacts*. There are two directories:

1. BootPartition
2. RootFSPartition

You must copy the files in each directory to the corresponding partition. If the SD card does not contain the necessary partition, please follow this guide to format it appropriately:

```
https://xilinx-
wiki.atlassian.net/wiki/spaces/A/pages/18842385/How+to+format+SD+card+for+SD+b
oot
```

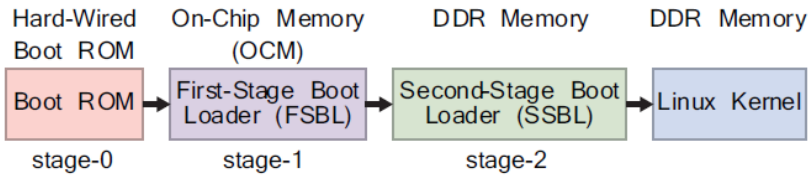
You can build the necessary boot files and the RootFS partition using the Petalinux flow. It is not in the scope of this lab to show you how to build the Petalinux artifacts from scratch, but feel free to explore it on your own (check Xilinx UG1144).

Appendix F. Booting Zynq

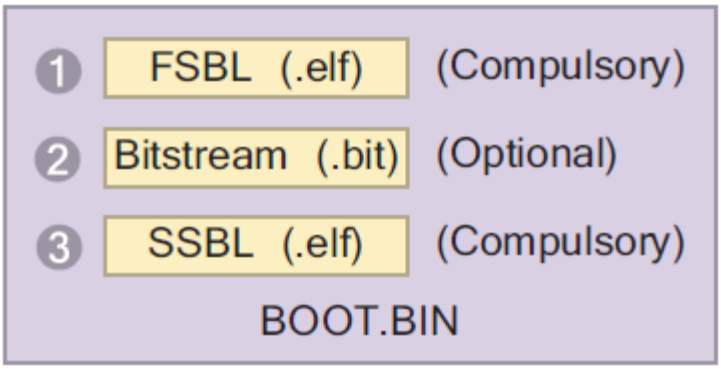
Zynq devices boot over a number of stages, starting with the boot ROM which is initialized at power-on. The value of the boot mode strapping pins of the device determines the boot mode. The boot mode defines from which of the supported interfaces JTAG, NAND Flash, NOR Flash, QSPI Flash, or SD card, the First Stage (Stage 1) Bootloader (FSBL) will be loaded. Once the boot mode has been determined, the boot ROM will read the boot header, and given the configuration parameters, will authenticate the image and load the First Stage Boot Loader (FSBL) image from the specified interface to the Zynq OnChip Memory (OCM). Once the image is transferred to the OCM, the control of the CPU is handed over to the FSBL.

The individual stages of the Zynq Linux boot process are detailed in the following figure.

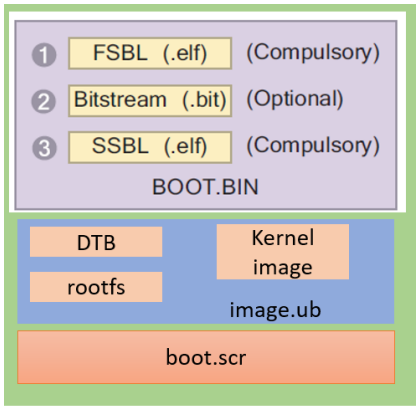
- **Stage 0**: On power-on reset, system reset or software reset, a hard-coded boot ROM is executed on the primary processor. The FSBL is loaded to the OCM.
- **Stage 1**: The FSBL is responsible for a number of initialization functions which include initializing the CPU with the PS configuration data, transferring the bitstream (.bit) from the boot interface (in our case the SD card), and programming the PL, loading the second-stage bootloader (*SSBL*) or initial user applications into DDR memory, and beginning execution of the second-stage bootloader/initial user application code. Before the control of the CPU is handed over to the second-stage bootloader, the FSBL disables the cache and the MMU, as well as invalidates the instruction cache, as U-boot assumes that they are disabled upon start.
- **Stage 2**: This is the user application software (.elf) that will run on the processing system in case of a bare metal system. In our case (petalinux), this is the second-stage bootloader. In this case, the SSBL is U-Boot and it is responsible for loading the compressed Linux kernel image, the system device tree, and the ramdisk image into memory. Once these images are loaded into memory, U-Boot will initialize the execution of the Linux kernel.



The following figure shows the zynq boot image file *BOOT.bin* which resides in the boot interface (SD card). It contains 2 compulsory executable and linkable format (elf) files, the FSBL (*zynq_fsbl.elf*) and SSBL (*u_boot.elf*) files, and the bitstream (*system.bit*) file. The FSBL and SSBL files, as their name suggests, contain the final stages of the bootloader which is used to load petalinux on the device. The bitstream is the file that is used to configure the programmable logic of the Zynq-7000 AP device. The ordering of the files that are stitched together to form the boot 17image is important; the optional bitstream file, if required, must be placed after the FSBL file and before the SSBL file.



The following figure shows the contents of the boot medium that need to be copied to the SD card. The DTB file (Device Trees Binary) is used to describe the hardware architecture and address map to the Linux kernel, and was created as part of petalinux-config.



Boot Medium (SD card)

References

1. Samuel Williams, Andrew Waterman, David Patterson,. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM 52(4): 65-76 (2009)*

2. https://software.intel.com/content/www/us/en/develop/documentation/advisor-user-guide/top/optimize-cpu-usage/cpu-roofline-perspective.html

3. Vitis High-Level Synthesis User Guide (UG1399).

4. Vitis Application Acceleration Development (UG1393)

5. Vivado High Level Synthesis Tutorial (UG871).