Fall 12-15-2021

A Case Study on HLS Portability from Intel to Xilinx FPGAs

Zhili Xiao

Washington University in St. Louis

James McKelvey School of Engineering

Department of Computer Science and Engineering

Thesis Examination Committee:
Roger Chamberlain
Anthony Cabrera
Christopher Gill

A Case Study on HLS Portability from Intel to Xilinx FPGAs

by

Zhili Xiao

A thesis presented to the James McKelvey School of Engineering
of Washington University in partial fulfillment of the
requirements for the degree of

Master of Science

December 2021
Saint Louis, Missouri

Contents

List of Tables

List of Figures

Acknowledgments

I am grateful to my parents for encouraging me to pursue my goals through this master's degree and supporting me in all my endeavors.

Thanks to my previous academic advisor, Prof. Ed Richter and Prof. Silvia Zhang, and all other professors that have taught me and laid down the foundations of knowledge and skills to complete this research.

I would like to thank Dr. Anthony Cabrera and Dr. Roger Chamberlain, who is also my academic advisor, for the inspiration and guidance that fueled this master's thesis. I also appreciate the help from Clayton Faber and Amy An for advice on my research and help in writing the thesis. I could not accomplish this without their help.

A special thank goes to my friends who accompanied with me during the COVID19 pandemic. It would be a hard time without you listening to me complain and without all the joys and happy moments with you.

Zhili Xiao

Washington University in Saint Louis
December 2021

Dedicated to my wonderful parents and people who have helped me in my life.

ABSTRACT OF THE THESIS

A Case Study on HLS Portability from Intel to Xilinx FPGAs

by

Zhili Xiao

Master of Science in Computer Engineering

Washington University in St. Louis, December 2021

Research Advisor: Professor Roger Chamberlain

Abstract: Field-programmable gate arrays (FPGAs) are a hardware accelerator option that is growing in popularity. However, FPGAs are notoriously hard to program. To this end, high-level synthesis (HLS) tools have been developed to allow programmers to design hardware accelerators with FPGAs using familiar software languages. The two largest FPGA vendors, Intel and Xilinx, support both C/C++ and OpenCL C to construct kernels. However, little is known about the portability of designs between these two platforms.

In this work, we evaluate the portability and performance of Intel and Xilinx kernels. We conduct a case study, porting the Needleman-Wunsch application from the Rodinia benchmark suite written in Intel OpenCL C to Xilinx platforms. We use OpenCL C kernels optimized for Intel FPGA platforms as a starting point and first perform a minimum effort port to a Xilinx FPGA, also using OpenCL C. We find that simply porting one-to-one optimizations is not enough to enable portable performance. We then seek to improve the performance of those kernels using Xilinx C/C++. With rewriting the kernel for burst transfer and other optimizations, we are able to reduce the execution time from an initial 294 s to 2.2 s.

Chapter 1

Introduction

With Dennard scaling no longer effective [1] and Moore's Law in retreat [14], offloading computations from traditional multicore processors to a hardware accelerator is a common approach used in the continuing effort to scale performance and efficiency. FPGAs are an attractive solution, because an FPGA allows the generation of specific hardware to make use of parallelism and specialized operations in the application. However, classical programming of FPGAs using hardware description languages (HDL), such as Verilog and VHDL, requires expertise in digital system designs and a huge amount of effort. Xilinx and Intel, the two FPGA vendors, have tried to improve productivity by offering high-level synthesis (HLS) which allows programmers to design hardware accelerators with FPGAs using familiar software languages, e.g., C/C++ and OpenCL C. Little is known about the portability of designs between these two platforms, which can hinder the further adoption of HLS designs.

To evaluate the portability and performance of Intel and Xilinx kernels, here we extend prior work [7] by porting the Needleman-Wunsch application [15] from the Rodinia benchmark suite [8] written in Intel OpenCL C [25] to a Xilinx FPGA. We use the Intel OpenCL C kernel codes from [25] as a starting point and first performed a minimum effort porting to Xilinx OpenCL C. We then improve the performance of the kernel codes in Xilinx C/C++ by taking advantage of Xilinx C/C++ pragmas and control and by moderately modifying the codes for burst transfer. Eventually, the run time was able to be reduced from 294 s to 2.2 s. This is much closer to, but not quite yet competitive with, the performance of the initial Intel designs. This suggests that to achieve the performance portability of HLS designs across FPGA vendors is not a straightforward task.

Similar to previous efforts [7], this work is a detailed case study of porting an application from the Intel platform to the Xilinx platform, which details the porting efforts and experiences of porting FPGA kernel optimizations from Intel OpenCL to Xilinx HLS and evaluates the performance and portability of the ported kernel. The factors that are distinctive to this work are the following:

- we port an application from a different computing pattern (dynamic programming);
- we start from Xilinx OpenCL C, expanding to Xilinx C/C++;
- we achieve substantial performance improvement through exploration of several optimizations; and
- we analyze the performance to study the reasons for performance gaps that remain.

The outline of the thesis is as follows. Chapter 2 introduces background knowledge of the Neeldleman-Wunsch kernel we chose to port and related works. Chapter 3 describes the application in more detail and our porting methods and optimization efforts. Chapter 4 lists the porting results and analyzes effective and ineffective efforts. Chapter 5 draws the conclusion for HLS portability and points out direction for future works.

The results of this work have been published [21], and the code is available at [22].

Chapter 2

Background and Related Work

The application that we use in this study is the Needleman-Wunsch application [15], which comes from the Rodinia benchmark suite originally created by Che et al. [8]. The intent of Rodinia was to provide a set of applications to evaluate heterogeneous computing systems across accelerator interfaces (e.g., OpenMP and OpenCL) and parallel computing communication patterns (e.g., dynamic programming, structured grid). Zohouri et al. [25] later extended the OpenCL implementations of a subset of the Rodinia benchmarks by designing optimized high level synthesis (HLS) kernels for FPGAs. However, the hardware designs from Zohouri et al. are optimized for Intel FPGA platforms. In this work, we port the Needleman-Wunsch OpenCL kernels from the suite to be synthesizable and performant on Xilinx FPGAs.

Sanaullah et al. [18] uses the Needleman-Wunsch and other common HPC applications to explore the optimization strategies and their effects on FPGAs for Intel OpenCL C. In particular, the authors detailed their optimization strategies and their effect on the singe-work-item (SWI) kernel of the Needleman-Wunsch algorithm. Most of these strategies have been adopted by Zohouri et al.'s original code. In our optimization exploration, we attempted to use their temporary variables strategy to resolve iteration dependencies.

On the Xilinx side, two recent works by Brown evaluated the performance of Xilinx's Vitis HLS tools with the Nekbone mini-app and the Himeno benchmark [2, 3]. In porting the Nekbone AX kernel from Fortran to Xilinx FPGAs via Vitis, the author studied a number of optimizations, including revising the algorithm from von Neumann to dataflow form, optimizing the use of memory banks, loop unrolling, and ping-pong buffering. In porting the Himeno benchmark, they increased the port data width using the `DATA_PACK` pragma and

splitting the dataflow into separate kernels to take use of the Xilinx HLS streams. De Fine Licht et al. [9] documented many transformation strategies to optimize the performance when translating applications from traditional software to Xilinx HLS. The authors categorize these strategies and emphasize the importance of pipelining, scaling, and memory accesses. Brown's and de Fine Licht et al.'s point to us a direction for future work.

Brown's recent work [4] also explored the vendor differences in HLS tool chains between Intel and Xilinx by revising an existing advection model kernel for the Xilinx HLS tool flow for both Xilinx Vitis and Intel tool flows with the aim of vendor portability and performance. Therefore, Brown's implementation was Vitis C/C++ tool flow oriented and significant modifications of kernel codes were needed to fit the Intel OpenCL program into the dataflow design. The work also explored the performance difference when scaling to multi-kernel designs, which is not applicable to our dynamic programming kernel.

Weller et al. showed that OpenCL as an HLS supported by both Intel and Xilinx is capable to design energy efficient partial differential equation solver with FPGA [20]. They identified vendor independent design strategies and Vendor specific optimization techniques. As a consequence of those differences, separate kernel codes were written for Xilinx and Intel FPGAs.

To overcome vendor differences, Kenter [12] provided guidance for design patterns that work well for both OpenCL based Xilinx SDAccel and Intel FPGA SDK for OpenCL tool flows and provides insights into the underlying philosophy and mechanism with examples. Kenter et al. also evaluate the portability of OpenCL based FPGA designs between vendors by implementing an finite-difference time-domain application for SDAccel and Intel FPGA SDK [13]. By using preprocessor macros, their implementation can flexibly run on FPGAs from different families. Our major difference from this work is that we evaluate the portability by starting from a design already optimized for the Intel OpenCL FPGA SDK instead of starting from scratch and trying to optimize for both vendors.

In prior work, Cabrera and Chamberlain used the Needleman-Wunsch kernels from Zohouri et al. [25] to evaluate the performance and portability between Intel FPGAs with different memory architectures [6]. They built the OpenCL C kernels that were originally targeting an Intel FPGA connected via PCIe on the Intel HARPv2 platform, which combines a CPU and FPGA on the same chip package. The approach in this work is similar but with a different

focus on evaluating the performance and portability across different FPGA vendors. For evaluation purpose, we compare the performance of the most performant kernel before and after optimizations with Cabrera and Chamberlain's results as shown in Fig 2.1.

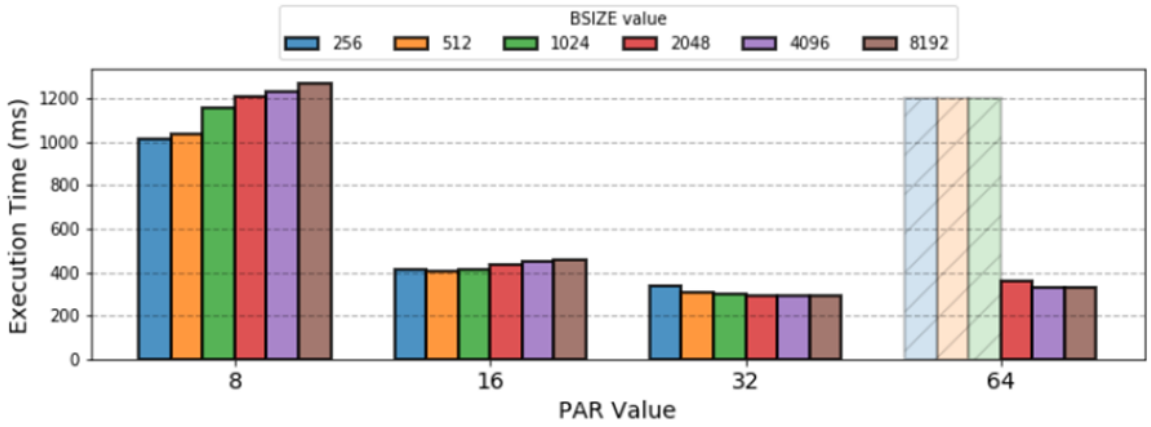

Figure 2.1: Execution times for the best kernel on Intel from prior work, sweeping across the design space of `PAR` and `BSIZE` [6].

We further used the work of Zohouri et al. [25] to evaluate the performance and portability between Intel and Xilinx platforms. This work extends [7] by porting a different class of application (dynamic programming) and utilizing Xilinx C/C++ for kernel design in order to enable design choices not available when using OpenCL C in Xilinx.

Chapter 3

Methods

To evaluate the portability of the two HLS implementations, we leverage the Intel OpenCL implementation of Needleman-Wunsch from the Rodinia benchmark suite modified by Zohouri et al. [25], and use the host and kernel codes as a starting point to build and run the kernels on the Xilinx platform. We first performed one-to-one optimization ports to Xilinx OpenCL C, and then explored how performant the kernel can be in Xilinx C/C++. The Xilinx platform for this work is a Xilinx Alveo U250 Data Center accelerator card, which includes an XCU250 FPGA of the Xilinx UltraScale+ architecture, a Gen3 x16 PCIe interface, and 64 GB of DDR4 off-chip memory. To author designs, we used the Vitis 2020.1 Core Development kit.

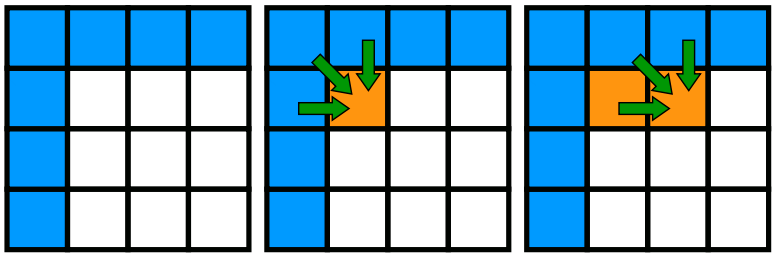

Figure 3.1: Illustration of the baseline version of the Needleman-Wunsch algorithm.

Needleman-Wunsch [15] is a dynamic programming algorithm frequently used in bioinformatics. The goal of the application is to find the global optimal alignment of two biosequences. Figure 3.1 shows a pictorial representation of the Needleman-Wunsch algorithm. Each biosequence is represented by integers which are attached to the output matrix as an extra row and column, as indicated by the blue elements. The score of each element depends on its top, top left, and left neighbors as indicated by the green arrows, the score from a reference

matrix, and a penalty value for mismatch. Due to these data dependencies, an element can only be computed after the score of its top, top left, and left neighbors have been determined and stored.

3.1 Initial Kernel Descriptions

To examine the portability of kernel designs, we chose the baseline kernel and the most performant kernel versions (v1 and v5 following the numbering by Zohouri et al. [25]) for porting. Both kernel versions are singe-work-item (SWI) kernels. The baseline version is just the SWI model itself with no FPGA optimizations at all. The v5 version is the design that has the highest performance and uses the least on-chip resources among the kernel versions according to the reports by Zohouri et al. [24] and Cabrera and Chamberlain [6]. In what follows, we refer to this version as the "best" version. It must be clarified that the design of the "best" version or version "v5" leads to a design space which will be discussed below. By tuning parameters across the design space, the performance of the "best" version will great variability, but it is still much better than the baseline version on Intel platforms. That means, we refer to the design with all different configurations of tuning parameters as the "best" kernel.

The Baseline Kernel The baseline kernel is simply the doubly nested loop outlined in lines 3-8 of Algorithm 1, which iterates through each location in the output matrix and performs the computations as showed in Figure 3.1. As mentioned above, the computation of the score of the current element depends on its top, top-left, and left neighbors as well as the score from the reference matrix and the penalty value, which is shown in line 5-8. Specifically, line 5 and 6 subtract the penalty value from the top and left neighbors, and line 7 adds the reference score at the current location to the top left neighbor. The max in line 8 is an inline function that will return the maximum of the three three results computed by line 5-7.

The Best Kernel Figure 3.2 illustrates how the best kernel makes use of wavefront parallelism and computes elements on one diagonal line at a time. The kernel processes the

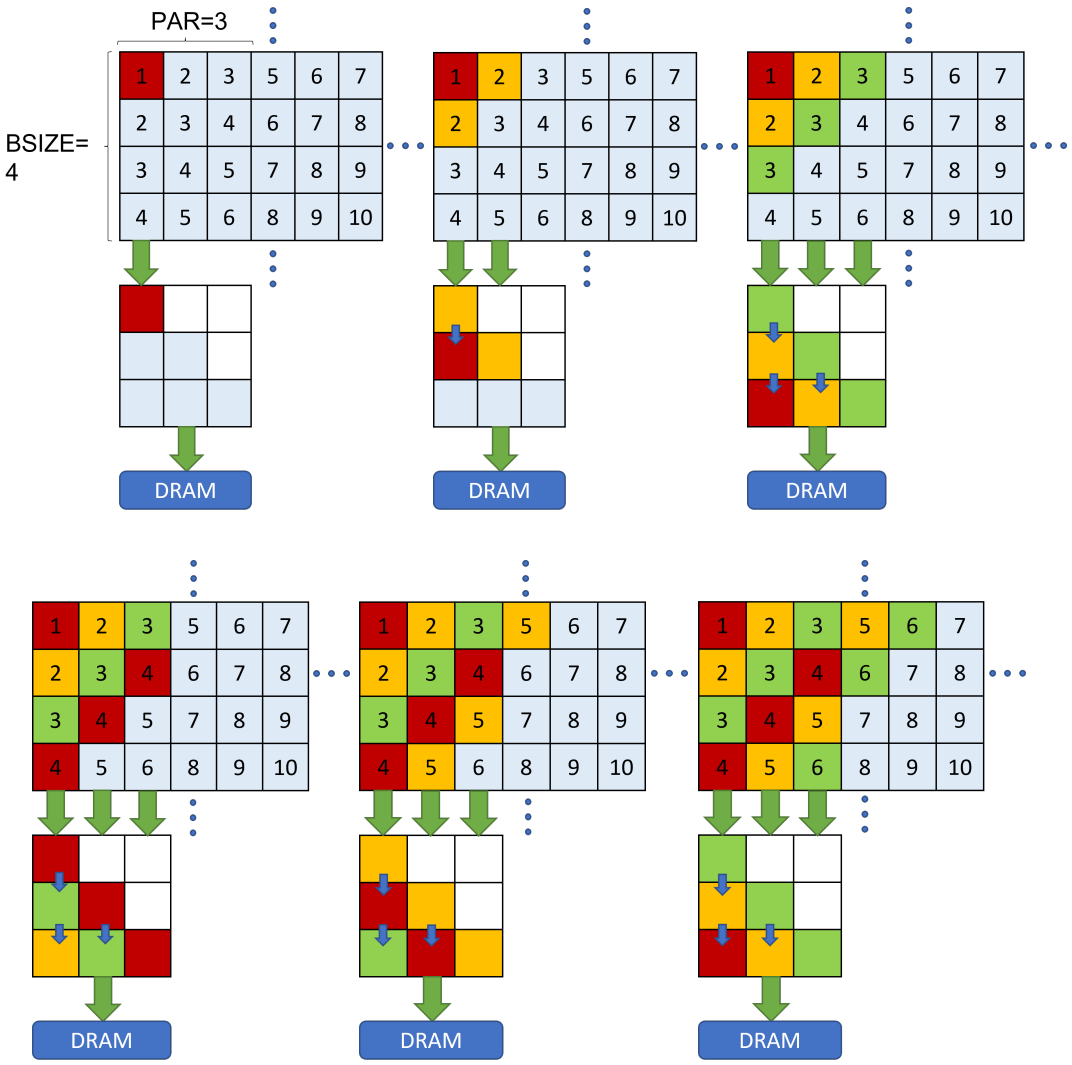

Figure 3.2: Illustration of the best kernel with BSIZE=4 and PAR=3. The number inside in grid is the order in which elements will be processed.

Algorithm 1 Baseline Needleman-Wunsch Algorithm

```
1: int output[dim+1][dim+1], reference[dim+1][dim+1]
2: int penalty
3: for i ← 1 to N + 1 do
4:     for j ← 1 to N + 1 do
5:         top = output[i − 1][j] − penalty
6:         left = output[i][j − 1] − penalty
7:         top_left = output[i − 1][j − 1] + reference[i][j]
8:         output[i][j] = max(top, left, top_left)
```

output matrix as groups of rows, where the size of each group is set by the parameter `BSIZE`. Within a single row group, each group is further divided into chunks of columns to fix the length of diagonal lines. The number of columns in each chunk is defined by the parameter `PAR`. Once the kernel has reached the bottom of the current chunk, it will wrap around to the next chunk until all elements in the current group of rows have been processed.

Another major optimization is the deployment of 2D shift registers of size `PAR` by `PAR` to hold the computation results from the last diagonal lines. This optimization has two advantages: first, it resolves the dependencies by storing the elements in local storage, which avoids expensive accesses to global memory; second, the lower triangular buffers rearrange the elements and coalesce the memory accesses such that elements on the same row can be burst transferred when writing to global memory, which will increase the bandwidth utilization.

The design of the best kernel introduces two design parameters, `BSIZE` and `PAR`. The hardware search space in our experimentation is the Cartesian product of

$$BSIZE \in \{256, 512, 1024, 2048, 4096\}$$

and

$$PAR \in \{8, 16, 32, 64\}.$$

To compare with the performance on Intel FPGAs, the sequence size we use is 23040, which is the same as Zohouri et al. [25] and Cabrera and Chamberlain [6].

3.2 One-to-one Porting to Xilinx OpenCL C

Intel OpenCL and Xilinx OpenCL use the same host interface. We followed the modern C++ conventions used in [7] to rewrite the host code and move the mapping of the DDR banks to the configuration file for Xilinx, instead of establishing the connection in Intel host codes. Using the same methods as in [7], we ported the baseline and the best kernel to Xilinx OpenCL C with minimum changes to simply allow the codes to be executable on the Xilinx platform. Because Xilinx has a partition limit of 1024 for the shift register that is too large to be completely partitioned, we choose not to partition it and let the compiler make its best decision. Besides porting FPGA optimizations like loop unrolling and shift registers, additional changes not documented in [7] are the porting of inline functions and compiler pragma `ivdep`, described next.

Inline Functions

In the baseline kernel version, the max function is an inline function. Inlining a function will make sure the function will not be generated as a hierarchical submodule at the register transfer level (RTL). The integration with surrounding logic and structures could lead to potential optimization during compilation and increase the performance. For Intel OpenCL, inlining a function is the same as in C/C++,

```
inline void foo(){}.
```

For Xilinx, the equivalent OpenCL attribute needs to be placed above the function,

```
__attribute__((always_inline)).
```

Ignore Vector Dependence

In the most performant kernel version, Intel OpenCL uses the pragma `ivdep` on the output matrix to forestall the false load/store dependency assumption on the global memory buffer

for the output matrix. As mentioned in Section 3.1, the dependency on the output matrix has been resolved because of the use of 2D shift registers. For Intel, to ignore the assumed inter loop dependency, the loop is preceded by

```
#pragma ivdep array(data).
```

Although the equivalent attribute was not found in the Vitis document, the SDAccel document suggests that the `xcl_dependence` attribute should be supported by Xilinx OpenCL C. With

```
__attribute__((xcl_dependence(variable ="data", type="inter",
            direction="RAW", dependent="false"))),
```

the read after write loop carried dependency can be resolved and the compiler can lower the initiation interval [12].

3.3 Porting to Xilinx C/C++ and Optimization

To optimize the performance of the best kernel, we explored the use of Xilinx C/C++ for architecting kernels, since Xilinx C/C++ affords more fine-grained control over the resulting hardware than is possible with OpenCL C.

HLS INTERFACE

All kernel arguments will be implemented as a port for input or output operations in the RTL design. In Xilinx C/C++, the implementation of these ports must be specified by the HLS `INTERFACE` pragma to assign an I/O protocol . We used the default setting of Vitis, assigning the array pointers to the `m_axi` interface and scalar inputs to the `s_axilite` interface. In addition to assigning the interface, we also explored the effect of the `num_read/write_outstanding` option of the interface, which specifies the maximum number of non-responding read/write requests can be issued before the design stalls to wait for

responses. Without specification, Xilinx will group all ports to the same memory interface. Because the interface will only address access request of one variable at a time, this will cause memory port contention and increase the pipeline initiation interval (II) even when the memories are mapped to different DDR banks. We thus assign a different `bundle` to each array input like Fig 3.3.

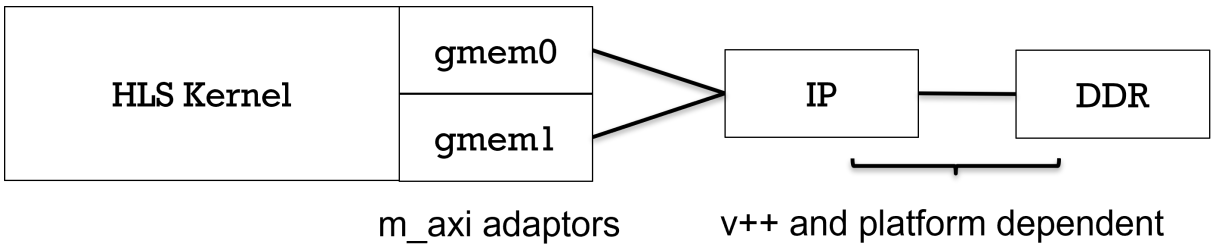

Figure 3.3: A kernel with two different memory adaptors for different inputs.

Loop Unrolling

To port the loop unrolling optimizations, Xilinx has a direct equivalence HLS pragma for loop unrolling,

```
#pragma HLS unroll factor=N.
```

The only difference is that the pragma needs to be placed inside the loop instead of preceding the loop.

Shift Registers

To port the shift registers, we treat the 1D shift registers and 2D shift registers separately. For 1D shift registers, we use the `ap_shift_reg` class in the Xilinx HLS library. To show the differences between the way we port the 1D shift registers in OpenCL C and in Xilinx C/C++, Listing 3.1 shows the set up 1D shift registers in Xilinx OpenCL C and Listing 3.2 shows the use of `ap_shift_reg` for Xilinx C/C++.

```
//fully partition the SR array
int shift_reg[SR_SIZE]
  __attribute__((xcl_array_partition(complete,0)));

//shift
__attribute__((opencl_unroll_hint(SR_SIZE - 1))
for (int i = 0; i < SR_SIZE - 1; ++i){
  shift_reg[i] = shift_reg[i + 1];
}
//new input to the tail of the array
shift_reg[SR_SIZE - 1] = input;
```

Listing 3.1: 1D shift registers in Xilinx OpenCL C.

```
static ap_shift_reg<int,SR_SIZE> shift_reg;
int var1;
//load new input into location 0, read the oldest value at location
   SR_SIZE-1
var1 = shift_reg.shift(input,SR_SIZE-1);

//read location 3 only
var1 = shift_reg.read(3);
```

Listing 3.2: 1D shift registers in Xilinx C/C++.

One major syntax difference is the location of new inputs. It was found that the Xilinx compiler sometimes had trouble inferring 1D shift registers in the style of Listing 3.1, which will degrade the performance. More about this is discussed in Section 4.2.

For 2D shift registers, as in [7], we completely partitioned the shift register arrays, and replaced the Intel OpenCL C loop unrolling pragmas with Xilinx HLS unrolling pragmas, as shown in Listing 3.3. There are several reasons for not implementing the 2D shift registers with `ap_shift_reg`. First, `ap_shift_reg` only supports 1D shift registers. Second, rewriting the columns of 2D shift registers into 1D shift registers would break down the global memory access loops and hinder the inference for burst transfer.

```
//fully partition
int SR[PAR][PAR];
#pragma HLS ARRAY_PARTITION variable=SR complete

//Shift
for (int i = 0; i < PAR; i++){
  #pragma HLS unroll
  for (int j = 0; j < PAR - 1; j++){
    #pragma HLS unroll
      SR[i][j] = write_SR[i][j + 1];
  }
}

//load from data, the global memory buffer
for (int i = 0; i < PAR; i++){
  #pragma HLS unroll
  SR[i][i] = data[read_index];
}
```

Listing 3.3: 2D shift registers in Xilinx C/C++.

Loop Carried Dependence

Similar to porting to Xilinx OpenCL C, we first ported the `ivdep` pragma by placing an `HLS dependence` pragma inside the loop, with `direction=RAW` and `type=inter` for loop carried dependencies,

`#pragma HLS dependence variable=data inter RAW false`.

We then explored the effect of resolving other kinds of dependencies including write after read (WAR) and write after write (WAW) dependencies on the output matrix.

Modifications for Burst Transfer

To enable burst transfer inference, we first isolated global memory access loops from other operations in the computation loop. We changed `i--` to `i++` because one of the precondition for burst transfer in Xilinx is continuous monotonically increasing order. As opposed to the loop switching technique of [9] to combine control flows into one pipeline, we applied loop unswitching techniques to global memory access loops to move the boundary conditionals

outside the loop. The removal of conditionals reduces the loop pipeline initiation interval (II) to 1 such that the burst transfer could be inferred by Vitis [12]. Since the burst transfer size will always equal to `PAR`, the number of columns that will be processed at the same time, we set the max burst read/write length option in `HLS INTERFACE` to 64, the largest `PAR` in our design space.

Exploration for Optimization

Besides the porting efforts above, we also tried to leverage the abundant options and control that Xilinx C/C++ offers. To explore what options are effective, we apply these options to the best kernel with `BSIZE` = 512 and `PAR` = 32. We explored the effects of binding arrays to different storage types to arrays, like FIFO and RAM with different number of ports. This is done by altering the storage `type` option in `BIND_STORAGE` pragma as shown below.

```
#pragma HLS bind_storage variable= <variable> type=<type>.
```

To deal with partition limits and simplify address demultiplexing logic and scheduler activities, we partition arrays with the `cyclic` and `block` option and partition factors of 8, 16, and 32. To reduce loop latency, we pipeline the computation loops using `PIPELINE` pragma. Moreover, we explore the effect of binding ports to different memory banks to avoid memory interleaving accesses and the effect of locating the compute unit to super logic regions (SLRs) through configuration file. The effects of all options and optimization explorations are discussed in detail in Sections 4.2 and 4.3.

Chapter 4

Results

4.1 Minimum Modification Porting Design Space Search

With one-to porting efforts as detailed in Section 3.2, the baseline version's execution time is 315 s, and the best execution time of the best kernel across the design space is 294 s. Table 4.1 lists the best execution times of the baseline and the best kernel on Intel and one-to-one porting of them on Xilinx platforms. Notice that the one-to-one optimization mapping of the baseline kernel has a portable performance on Xilinx as the baseline kernels involves almost zero designs for FPGA specific optimizations. Figure 4.1 shows the run time of the best kernel across the design space ported to Xilinx. Note that the performance of the best kernel varies considerably across the design space, yet the highest performing configuration of the best kernel achieves a speedup of only 1.07× from the baseline.

On the other hand, the best kernel on Intel FPGA with PCIe and HARP system takes only 0.260 s and 0.290 s achieving 784× and 2862× speedups relative to the baseline design, respectively. This is in stark contrast to the 1.07× speedup achieved by the minimum porting effort, let alone some other configurations that have even worse performance than the baseline kernel. Moreover, the huge variations in run time of designs with the same `PAR` is different from the performance pattern as reported in Fig 2.1, where the run time of kernels with the same `PAR` size but different `BSIZE` are similar.

It was also noted that the compiler was not able to synthesize the design with `PAR`=64, which was also observed in Fig 2.1 for `BSIZE` smaller than 2048. For the Intel compiler, the design

Table 4.1: The runtime of the baseline and the best kernel on Intel and one-to-one porting of them on Xilinx platforms.

Version	Platform	Runtime (sec)
Baseline	Stratix V GX A7, PCIe	204
	Arria 10 GX 1150, HARP	830
	Alveo u250, PCIe	315
Best	Stratix V GX A7, PCIe	0.26
	Arria 10 GX 1150, HARP	0.29
	Alveo u250, PCIe	294

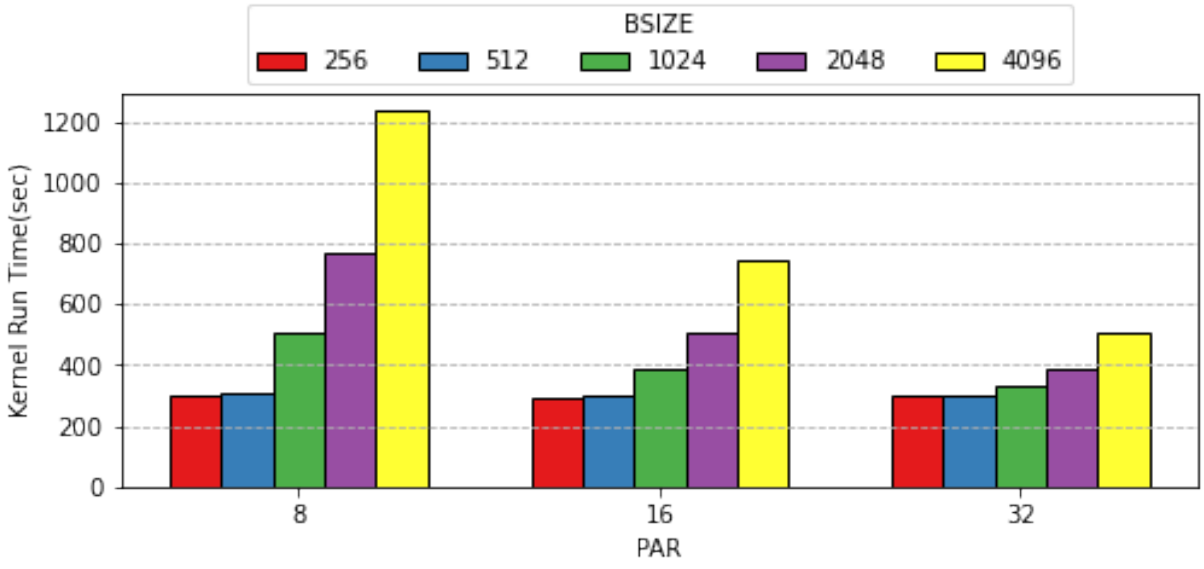

Figure 4.1: Execution times for the best kernel with one-to-one port to Xilinx OpenCL C, sweeping across the design space of `PAR` and `BSIZE`.

is too congested to fit onto the board. But for Vitis, the limitation is because of the partial write inference on the shift registers and the complex scheduling.

Obviously, with minimum effort porting to OpenCL, the Xilinx Vitis compiler interprets the kernel codes differently and the best kernel cannot achieve the same performance and speedup as it has in Intel systems. More modifications are necessary to improve performance. To this end, we use Xilinx C/C++ to author kernels instead of OpenCL C.

4.2 Ineffective Optimization Efforts

Xilinx C/C++ offers finer control and more options than Xilinx OpenCL C. Among the options we explored, there are some ineffective optimization efforts which do not decrease the execution time and even harm performance in some cases. We first try to optimize the shift register structure that cannot be completely partitioned. Even though Xilinx asserts that it supports the inferring of shift registers even without complete partition [19], we found that this was not the case until we enabled burst transfers.

To reduce cycles of operations on shift registers, we use the `BIND_STORAGE` pragma to assign the shift register array to RAM with 1 write port and multiple read ports and FIFO. Before enabling burst transfers, binding arrays to RAM with multiple ports had no performance improvement but consumed more resources, because the RAM cannot satisfy the simultaneous store operations as with the shift registers and therefore breaks the pipeline. Binding to FIFO eventually failed because the compiler cannot find a legal memory core for the store operation on the FIFO. After all, the shift register is not completely the same as the FIFO. To mitigate this, we also explored the cyclic partition on the shift registers with partition numbers = 8, 16, or 32 combined with unrolling factor equal to the partition number as described in [16]. We observed that the cyclic partitioning degrades the performance of the kernel. First, the scheduler is still unable to accommodate the store operations on the array even with cyclic partitions. Second, the scheduling complexity increased because the array's size is not a multiple of the partition factor, which degrades the performance.

However, the inferring of shift registers seems to succeed after we enable burst transfers. No reports on pipeline breaking due to the shift register arrays were found. We attribute

this finding to be a bug of Vitis 2020.1, which seems to have been fixed by 2020.2. We also compared the performance of including the `ap_shift_reg` class for the 1D shift registers after the burst transfer has been enabled. No significant difference in resource and performance was found when using the class versus not using it.

The second option we explored is the number of outstanding read/write requests option in the `INTERFACE` pragma mentioned in Section 3.3. It specifies the number of transactions that can be initiated before waiting for the first to complete, which can effectively hide memory access latency. Applying the option reduces the execution time of the best kernel from hundreds of seconds to 44 s. However, this option is only effective before burst transfer is enabled. It turns out that after enabling burst transfer, the kernel has to stall for external memory before the next burst transaction can be issued because of the dependency. The burst transfer is more effective for hiding latency between atomic memory accesses and reduces the execution time to 14 s.

To try to reduce the pipeline II, we used the temporary variable strategy described in [18] to store the offset variables to resolve loop carried dependencies on reading/writing of the global memory. But the dependency could not be resolved by simply using temporary variables. The scheduler reports show that the pipeline II cannot be further reduced because the dependency is in fact caused by the contention on memory ports, which will be discussed in Section 4.4.

Xilinx empowers users with the ability to do coarse-grained floorplanning by specifying the placement of compute units. Since we have mapped the kernel ports to different DDR banks, the location of the compute unit in different SLR regions can decrease or increase the routing across the boundary and thus affect the timing and clock rate. For Xilinx Avelo 250, SLR0 connects to the port of DDR bank 0, and SLR1 connects to DDR bank 1. Mapping the buffer "reference" to DDR bank 0 and the buffer "data" to bank 1 and explicitly placing the compute unit in SLR0 (2.54 s) or SLR1 (2.57 s) slightly degraded the performance, increasing the run time by about 300 ms. Examination of the implementation log files shows that without specification, Xilinx will spread the compute unit across SLR0 and SLR1 such that the compute unit is close to both memory interfaces, which can slightly reduce the execution time.

4.3 Effective Optimization Efforts

Using the run time profile of the best kernel, we found that Xilinx failed to infer burst transfer from the original kernel code. Therefore, only atomic transactions to/from the global memory could be issued and more than 10000 ns of global memory access latency was incurred. By rewriting the best kernel in ways described in Section 3.3, the average global memory transaction size is increased to `PAR` integers and the latency is reduced to about 300 ns.

Although the performance improvement achieved by the memory banking is minimal, separating global arrays into different memory interfaces with the `bundle` option can effectively reduce the memory port contention and reduce the pipeline II from 2·`PAR` (caused by the read of the reference matrix and the output matrix) to `PAR`.

After the rewrite for burst transfers, the Vitis compiler will only pipeline the memory access loops. We added the `PIPELINE` pragma to the whole computation loop to pipeline both burst transfers and computations. With the read after write dependency on the output matrix resolved by `#pragma HLS dependence variable=data inter RAW false`, the execution time for the best kernel with `BSIZE`=512 and `PAR`=32 is reduced to 6.85 s.

A closer look at the compilation log shows that besides the RAW dependency, Xilinx also assumes WAW dependencies on the output matrix. With `#pragma HLS dependence variable=data inter false`, all loop carried dependencies can be resolved and execution time is reduced to 2.63 s. Examination of the run time reports shows that resolving all loop carried dependencies will increase the kernel frequency from 105 MHz to 235 MHz, which is the major cause of the speedup.

Xilinx allows users to map memories to different memory banks to avoid interleaving global memory access through building commands or configuration files. Fig 4.2 shows the floor plan of global memories and SLR regions on the Alveo U250. With no manual mapping of memory banks, all global memories are accessed through DDR bank 0, and the resulting design will be placed in SLR 0 which is closest to bank 0. This design decreases the clock rate. By mapping the reference matrix to DDR bank 0, the output matrix to DDR bank 1,

and the smaller input vector to PLRAM, the clock rate is increased to 295 MHz, and the execution time is reduced to 2.2 s.

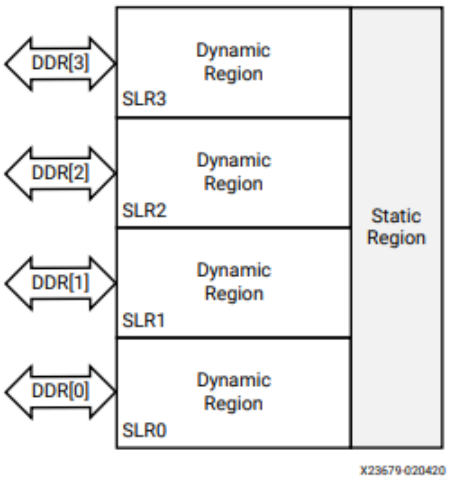

Figure 4.2: The floor plan of the Xilinx Alveo U250 [23].

4.4 Xilinx C/C++ Performance Analysis

Table 4.3 lists the execution time, the FPGA resource consumption, and the speedup relative to the baseline of Xilinx C/C++ kernels across the design space. It is hard to make a direct comparison between the hardware resources for Xilinx and Intel, as they use different logic slices (see Table 4.2). But both of them use only a small amount of their respective FPGA resources. Because Xilinx was able to identify the shift registers of the C/C++ kernels, BRAM usage does not increase as the `PAR` size increases as we observed for OpenCL kernels. Figure 4.3 illustrates the execution time of the best kernel across the design space. We observed a similar performance pattern as [6] (see Fig 2.1), where designs with the same `PAR` have a similar execution time, and the performance of designs with `PAR`=8 are significantly worse than others.

The C/C++ kernels with burst transfers enabled and pipelined are much more performant than the one-to-one optimization port OpenCL kernels. The execution time is reduced to 2.2 s, achieving 143× speedup relative to the baseline version. However, the Xilinx C/C++ kernels are still 10× slower than the Intel OpenCL C kernels. To know why there continues

Table 4.2: Resources available on the Intel and Xilinx Platforms [10] [23] [11]

FPGA	ALM/CLB	Register (K)	M20K (Blocks | Mb)	DSP	External Memory
Stratix V GX A7, PCIe	234,720	939	2,560 | 50	256	2x DDR3-1600
Arria 10 GX 1150, HARP	427,200	1,709	2,560 | 50	1,518	2x DDR4-2133
Alveo u250, PCIe	182,000	3,456	N/A	12,288	4x DDR4 2400MT/s

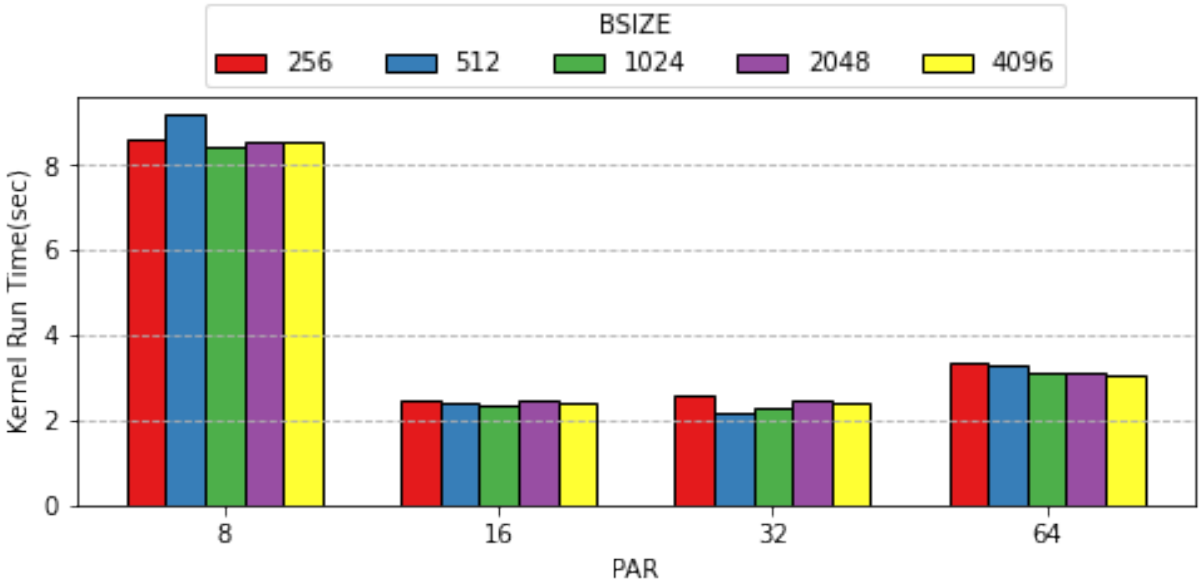

Figure 4.3: Execution times of the best kernel in Xilinx C/C++, sweeping across the design space of `PAR` and `BSIZE`.

to be a performance gap and for potential performance improvement in future work, we next utilize the synthesis report and the run time report to the analyze the execution time and study the performance bottleneck.

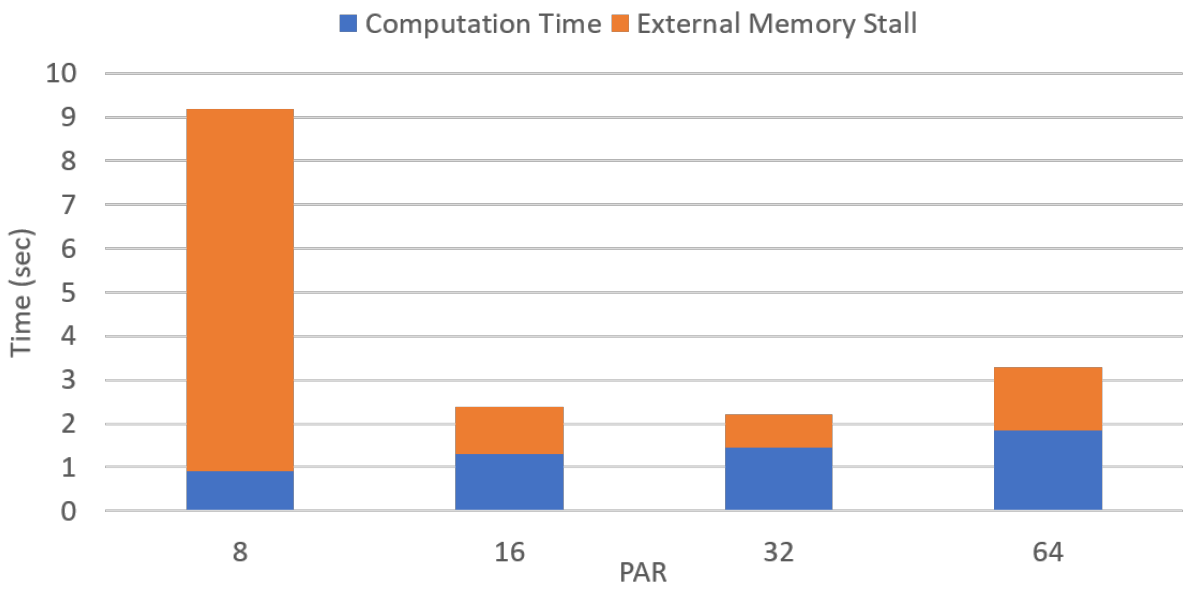

Figure 4.4: Execution times for the best kernel with `BSIZE` = 512 and different values of `PAR` in terms of computation time and external memory stalls.

The synthesis report shows that the computation loop can only be pipelined with II=`PAR`, because it cannot resolve the loop carried dependency of reads on the reference matrix or the output matrix. Of course, there is no real "read after read" dependency here, and the true reason is that the interface will only deal with one burst transaction at a time and that each burst transaction needs `PAR` cycles (one cycle for each element). Reducing the `PAR` of the design will reduce the pipeline cycle, but it will also reduce the number of elements processed per loop and potentially increase the memory access latency. For example, reducing `PAR` to 8 will increase the latency to over 1000 ns. On the other hand, further increasing the `PAR` will not further decrease the memory latency, but will make the design more congested and result in the decrease of clock rate. Figure 4.4 shows the composition of the execution time of the best kernel with `BSIZE` = 512 and different `PAR` in terms of the external memory stalls and the rest of time, which is the actual time spend on computation. Therefore, if we reduce the external memory stalls as reported by the run time report, we will find that the actual

Table 4.3: Results for the baseline and the best version with different configurations in Xilinx C/C++.

Kernel Version	PAR	BSIZE	Time (sec)	f_{max} (MHz)	LUT	Register	BRAM	DSP	Speedup
Baseline	N/A	N/A	314.788	300	3752 (0.22%)	5073 (0.15%)	2 (0.07%)	30 (0.24%)	1
Best	8	256	8.598	300	10300 (0.6%)	12773 (0.39%)	4 (0.15%)	6 (0.05%)	37
		512	9.179	300	10409 (0.6%)	12837 (0.39%)	4 (0.15%)	6 (0.05%)	34
		1024	8.467	300	10746 (0.62%)	12876 (0.39%)	4 (0.15%)	6 (0.05%)	37
		2048	8.542	300	12571 (0.73%)	12825 (0.39%)	4 (0.15%)	6 (0.05%)	37
		4096	8.540	300	13782 (0.8%)	12817 (0.39%)	4 (0.15%)	6 (0.05%)	37
	16	256	2.486	295	16564 (0.96%)	22376 (0,68%)	4 (0.15%)	6 (0.05%)	127
		512	2.390	300	17617 (1.02%)	21942 (0.67%)	4 (0.15%)	6 (0.05%)	138
		1024	2.375	300	19229 (1.11%)	22239 (0.68%)	4 (0.15%)	6 (0.05%)	133
		2048	2.449	300	18920 (1.09%)	22499 (0.68%)	4 (0.15%)	6 (0.05%)	129
		4096	2.426	300	20678 (1.2%)	22191 (0.67%)	4 (0.15%)	6 (0.05%)	130
	32	256	2.599	245	33022 (1.91%)	41720 (1.27%)	4 (0.15%)	6 (0.05%)	121
		512	2.200	295	33608 (1.94%)	41900 (1.27%)	4 (0.15%)	6 (0.05%)	143
		1024	2.294	285	33913 (1.96%)	41920 (1.27%)	4 (0.15%)	6 (0.05%)	137
		2048	2.497	270	32524 (1.88%)	41571 (1.26%)	4 (0.15%)	6 (0.05%)	126
		4096	2.437	275	36061 (2.09%)	41851 (1.27%)	4 (0.15%)	6 (0.05%)	129
	64	256	3.345	210	62210 (3.6%)	81196 (2.47%)	4 (0.15%)	6 (0.05%)	94
		512	3.283	215	64481 (3.75%)	81218 (2.47%)	4 (0.15%)	6 (0.05%)	96
		1024	3.147	225	64004 (3.7%)	81102 (2.46%)	4 (0.15%)	6 (0.05%)	100
		2048	3.118	230	63711 (3.69%)	81198 (2.47%)	4 (0.15%)	6 (0.05%)	101
		4096	3.056	255	67478 (3.9%)	81139 (2.46%)	4 (0.15%)	6 (0.05%)	103

computation time is similar for different `PAR` size as indicated by Figure 4.4. The actual time spent on the computation on Xilinx FPGA is around 1 second even with clock frequency at 300 MHz, which is still 3× slower than the overall execution time on Intel platforms. We therefore conclude that the performance bottleneck is the coupling of the pipeline II and `PAR` caused by the memory port contention. To achieve better performance, we must decouple the II and `PAR`, and effectively hide or eliminate the external memory stalls.

Aside from the performance portability, we also noticed a significant difference in build time. As `BSIZE` decreases and `PAR` increases, the build time increases. Examination of the logs shows that the increase mainly comes from place and route. But compared with the unsynthesizable Xilinx OpenCL C version with `PAR` = 64, Vitis took no longer than 5 hours to build the bitstream across the C/C++ design space.

Chapter 5

Conclusion and Future Work

This work presents our efforts to port an application kernel that has already been optimized for Intel FPGA to the Xilinx platform and our evaluation of its performance and portability. We found that most FPGA optimizations including the 1D and 2D shift registers can be successfully ported with relatively low effort. Inter loop dependency optimization can also be easily ported as Vitis is a best effort multi-pass compiler and will try to resolve all different kinds of loop carried dependency itself. But in terms of performance, one-to-one kernel optimization ports is not enough. The rigorous constraint on pipeline II for burst transfer inference of the Xilinx compiler requires significant rewriting of the code that works well on Intel FPGAs, which is also the biggest contributor of the performance difference. By rewriting the kernel in the style that the Xilinx compiler prefers, we enabled burst transactions to/from the global memories and reduced the average transaction latency to around 300 ns and the overall execution time to 2.2 s, achieving a 143× speedup relative to the baseline kernel and 134× speedup relative to the one-to-one optimizations port version. But even with the rewriting for burst transfer, there is still an order of magnitude gap in the performance between the Xilinx kernel and the Intel kernel.

Besides the compiler options we explored in this work, Xilinx C/C++ offers additional compiler options that can potentially improve the performance. For example, using the `latency` option in the `INTERFACE` pragma to issue read and write requests in advance could reduce external memory stall cycles. Merging sequential loops with the `LOOP_MERGE` pragma can eliminate cycles between loops. However, many of these techniques can only reduce the depth of each loop iteration. As the performance analysis in Section 4.4 shows, the overall execution time will not be reduced to less than 1 s if the pipeline II is not reduced.

We could possibly break the performance bottleneck by rewriting the memory access into functions and use the `DATAFLOW` pragma as described in [9] and [2] and by applying the `AGGREGATE` pragma, which is similar to the `DATA_PACK` directive that is no longer supported in Vitis 2020.1 to utilize the full port width. We also noticed that Vitis 2020.2 has a new functionality called automatic port width widening. This new feature allows the Xilinx compiler to automatically infer a widen port if the memory alignment condition can be satisfied.

For this work's purpose of evaluating portability of HLS codes, we choose to stop and conclude that simply performing one-to-one optimization ports and rewriting loops for burst transfer is not enough to make the HLS design's performance portable between vendors. As suggested above, competitive performance with Intel's platform could be achieved on Xilinx platform. Better comparison in terms of operations per clock cycle and resource usage could be made, and more insights into the vendor differences can be provided. Further exploration for optimization as suggested above is left for future work.

References

[1] Mark Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, Winter 2007.

[2] Nick Brown. Exploring the acceleration of Nekbone on reconfigurable architectures. In *Proc. of IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pages 19–28. IEEE, 2020.

[3] Nick Brown. Weighing up the new kid on the block: Impressions of using Vitis for HPC software development. In *Proc. of 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 335–340. IEEE, 2020.

[4] Nick Brown. Accelerating advection for atmospheric modelling on xilinx and intel fpgas. In *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 767–774, 2021.

[5] Nick Brown and David Dolman. It's all about data movement: Optimising FPGA data access to boost performance. In *Proc. of IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pages 1–10. IEEE, 2019.

[6] Anthony M Cabrera and Roger D Chamberlain. Exploring Portability and Performance of OpenCL FPGA Kernels on Intel HARPv2. In *Proc. of International Workshop on OpenCL*, pages 3:1–3:10. ACM, 2019.

[7] Anthony M Cabrera, Aaron R Young, Jacob Lambert, Zhili Xiao, Amy An, Seyong Lee, Zheming Jin, Jungwon Kim, Jeremy Buhler, Roger D Chamberlain, and Jeffrey S. Vetter. Toward Evaluating High-Level Synthesis Portability and Performance between Intel and Xilinx FPGAs. In *Proc. of International Workshop on OpenCL*, pages 7:1–7:9. ACM, 2021.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[9] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1014–1029, May 2021.

[10] Intel. Intel Arria 10 Device Overview. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_overview.pdf, 2020.

[11] Intel. Stratix V 5SGXA7 FPGA Specifications. https://www.intel.com/content/www/us/en/products/sku/210327/stratix-v-5sgxa7-fpga/specifications.html, 2021.

[12] Tobias Kenter. Invited Tutorial: OpenCL design flows for Intel and Xilinx FPGAs: Using common design patterns and dealing with vendor-specific differences. In *Proc. of 6th International Workshop on FPGAs for Software Programmers*, 2019.

[13] Tobias Kenter, Jens Förstner, and Christian Plessl. Flexible FPGA design for FDTD using OpenCL. In *Proc. of 27th International Conference on Field Programmable Logic and Applications (FPL)*, September 2017.

[14] Chris A Mack. Fifty years of Moore's law. *IEEE Transactions on Semiconductor Manufacturing*, 24(2):202–207, 2011.

[15] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[16] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types. In *Proc. of 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 393–407, 2020.

[17] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types. *CoRR*, abs/2004.04852, 2020.

[18] Ahmed Sanaullah, Rushi Patel, and Martin Herbordt. An empirically guided optimization framework for FPGA OpenCL. In *Proc. of International Conference on Field-Programmable Technology (FPT)*, pages 46–53. IEEE, 2018.

[19] Vitis Unified Software Development Platform Documentation. Vitis HLS Methodology Guide. https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/migrating_to_vitis_hls.html, 2021.

[20] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. Energy efficient scientific computing on fpgas using opencl. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 247–256, New York, NY, USA, 2017. Association for Computing Machinery.

[21] Zhili Xiao, Roger D Chamberlain, and Anthony M Cabrera. Hls portability from intel to xilinx: A case study. In *Proc. of IEEE High-Performance Extreme Computing Conference (HPEC)*, 2021.

[22] Zhili Xiao, Roger D Chamberlain, and Anthony M Cabrera. Source code. `https://github.com/zhilixiao/rodinia-nw`, 2021.

[23] Xilinx. Alveo u200 and u250 data center accelerator cards data sheet. `https://www.xilinx.com/support/documentation/data_sheets/ds962-u200-u250.pdf`, 2021.

[24] Hamid Reza Zohouri. *High Performance Computing with FPGAs and OpenCL*. PhD thesis, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2018.

[25] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 409–420. IEEE, 2016.

Vita

Zhili Xiao

Degrees

B.S. Summa Cum Laude, Computer Engineering, May 2020
M.S. Computer Engineering, December 2021

Publications

Anthony M. Cabrera, Aaron R. Young, Jacob Lambert, Zhili Xiao, Amy An, Seyong Lee, Zheming Jin, Jungwon Kim, Jeremy Buhler, Roger D. Chamberlain, and Jeffrey S. Vetter. "Toward Evaluating High-Level Synthesis Portability and Performance between Intel and Xilinx FPGAs," in *Proc. of 9th International Workshop on OpenCL* (IWOCL), April 2021. DOI: 10.1145/3456669.3456699

Zhili Xiao, Anthony M Cabrera, Roger Chamberlain. "HLS Portability from Intel to Xilinx: A Case Study," in *Proc. of IEEE High-Performance Extreme Computing Conference* (HPEC), September 2021. DOI: 10.1109/HPEC49654.2021.9622785

Jacob Wheelock, William Kanu, Marion Sudvarg, Zhili Xiao, Jeremy Buhler, Roger D. Chamberlain, James H. Buckley. "Supporting Multi-messenger Astrophysics with Fast Gamma-ray Burst Localization," in *Proc. of IEEE/ACM HPC for Urgent Decision Making Workshop* (UrgentHPC), November 2021. DOI: 10.1109/UrgentHPC54802.2021.00008

Clayton J. Faber, Tom Plano, Samatha Kodali, Zhili Xiao, Abhishek Dwaraki, Jeremy D. Buhler, Roger D. Chamberlain. "Platform Agnostic Streaming Data Application Performance Models," in *Proc. of IEEE/ACM Workshop on Redefining Scalability for Diversely Heterogeneous Architectures* (RSDHA), November 2021. DOI: 10.1109/RSDHA54838.2021.00008

December 2021