OpenCL-based design methodologies for FPGA implementation

March 2020

IMAN FIRMANSYAH

OpenCL-based design methodologies for FPGA implementation

Graduate School of Systems and Information Engineering
University of Tsukuba

March 2020

IMAN FIRMANSYAH

Abstract

FPGA, or Field Programmable Gate Array, has been widely used in many engineering applications such as in medical, automotive, communications, big data and artificial intelligence (AI). Traditionally, hardware description language (HDL) is required to program an FPGA. HDL programming provides an efficient logic resource with low latency. However, it is time-consuming for designs that are more complex. The user may struggle to write sophisticated programs that are performed effectively on FPGAs using HDL program. Currently, OpenCL is implemented for FPGA programming. OpenCL reduces the FPGA development time because it increases the abstraction level of the code. OpenCL implementation on FPGAs yields high-performance results for the computation process. OpenCL is an open and royalty-free framework for accelerating the algorithm executed on a heterogeneous system such as a GPU, CPU, DSP, or FPGA. Recently, FPGAs have also been applied in high-performance computing applications because of their reusability, reliability, high performance, and low power consumption. A new trend continues to grow by implementing CPU, GPU, and FPGA as an accelerator. FPGAs provide an alternative solution due to its capability to communicate with each other, or between FPGA and GPU with low latency. This study focuses on the implementation of OpenCL programming as a type of HLS design on FPGA boards both for HPC applications and engineering applications. Therefore, the study evaluates the capability of FPGA in OpenCL design by conducting the experiments according to external memory bandwidth capability, computational capability, and input/output (I/O) capability. For HPC applications, matrix multiplication is chosen to evaluate the computational capability of FPGA, and Himeno benchmark is chosen as a type of memory-intensive application. To evaluate the I/O capability of FPGA using OpenCL design, a signal generation and measurement is demonstrated using OpenCL program for engineering fields. Because the OpenCL SDK standard does not provide a particular function to access the FPGA's I/O hardware directly, new OpenCL components are developed inside the FPGA Board Support Package (BSP) that can interact with the FPGA hardware directly through an OpenCL I/O channel extension. This component allows an OpenCL kernel on FPGA to read data from and write data to the FPGA I/O. The study shows that OpenCL can be used for high-performance applications and engineering applications.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor Prof. Taisuke Boku and Associate. Prof. Yoshiki Yamaguchi for their dedicated support and guidance in completing the research during my study at the Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba. I would like also to thank the rest of the thesis committees: Prof. Koichi Wada, Prof. Tsutomu Maruyama, Associate. Prof. Toshihiro Hanawa and Prof. Moritoshi Yasunaga for their insightful comments and correction. Further, I am also thankful to all members of the FPGA research group for their collaborative effort during the study.

In this opportunity, I owe my deepest gratitude to the Ministry of Research, Technology and Higher Education (RISTEK-DIKTI) of the Republic of Indonesia for supporting the Program Research and Innovation in Science and Technology (RISET-Pro) scholarship. Finally, I must express my very profound gratitude to my family for continuous encouragement and support throughout my years of study at the University of Tsukuba and through the process of writing this thesis.

Iman Firmansyah

Tsukuba, January 2020

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

In recent years, High-Performance Computing (HPC) infrastructures have been used for computation-intensive or data-intensive applications in government institutions, academic institutions, and industry organizations. GPU provides a highly parallel computation process due to its many-core processors with very high memory bandwidth [1]. It also can be combined along with CPU into hybrid CPU/GPU computing architecture system which has been successfully employed to solve the computation in many different research topics such as high-energy physics, weather prediction, data mining, bioinformatics and so forth. However, not only difficulties in programming for hybrid cluster system need to be overcome due to different hardware and software used for CPU and GPU [2], but also in power consumption requirements. Since the total number of cores inside GPU chip has increased drastically in recent GPUs, the power required for it has grown significantly as well [3]. In HPC applications, a new trend continues to growth by implementing CPU, GPU and FPGA as accelerator. FPGAs provide alternative solution due to its capability to communicate each other, or between FPGA and GPU with low latency [4] [5].

FPGAs, which stands for Field Programmable Gate Arrays, consist of a matrix of configurable logic blocks (CLBs) that are connected through programmable interconnects. FPGAs have been widely implemented in many engineering and scientific applications because of their reusability, reliability, high performance, and low power consumption. FPGAs are used in medical, automotive, and military applications [6], in communications [7], and in nuclear facilities [8]. Recently, FPGAs have also been applied in artificial intelligence (AI) [9] and internet of things (IoT) solutions [10].

In general, FPGA is designed by using HDL such as Verilog HDL and VHDL, which is targeted for logical-level hardware design. These programming languages synthesize logical-level functions, gate-level processing, and physical-level layouts [11]. However, FPGA programming using HDL is time-consuming and becoming difficult to take care of all FPGA design by using HDL because the amount of FPGA circuits is increasing rapidly; for instance, current FPGAs can implement multi-core CPUs, and the design complexity is beyond the range of HDL-based design. Moreover, to develop a complex design using HDL, an FPGA programmer needs to have detailed knowledge of the hardware and software of an FPGA, particularly its programming, simulation, and debugging process.

Currently, high-level synthesis (HLS) is implemented on FPGAs. HLS provides an alternative solution to reduce the development time for FPGA programming. HLS improves the FPGA design efficiency by increasing the abstraction level of the code [12]. HLS also reduces the gap between the FPGA design and the programming process.

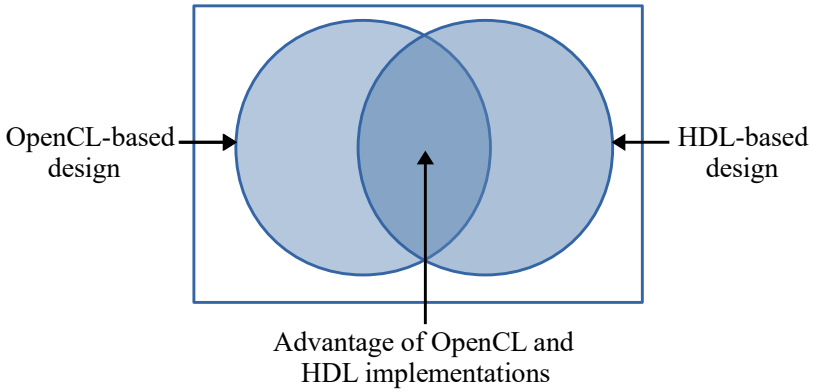

Figure 1.1: Relationship between OpenCL and HDL implementation for FPGA design.

Consequently, the FPGA development time can be reduced. Thus, high-level language design is immensely popular, and FPGA vendors start to introduce High-Level Synthesis (HLS) to their CAD tools. OpenCL, which stands for Open Computing Language, is the open standard for cross-platform, parallel programming for different processors found in personal computers, servers, mobile devices and embedded platforms [13]. OpenCL provides a solution to these problems since the FPGA programmer can employ C syntax in their FPGA projects rather than VHDL/Verilog code. In addition, it can be executed across different platforms consisting of not only FPGAs but also CPUs, GPUs, and DSPs.

1.1 Motivation

Many studies show the implementation of OpenCL design for FPGA, specifically in high-performance computing applications, such as for molecular dynamics simulation, tsunami simulations, data mining, artificial intelligence, big data, and stencil computation. Compared to HDL programming, OpenCL usage for FPGA increases productivity and reduces development time. It is because the OpenCL programming increases the abstraction level of the code. OpenCL for FPGA also provides software libraries, an application programming interface (API), and a communication interface between the host and the FPGA.

Unlike CPU and GPU, FPGA provides connectivity and capability to integrate with external devices such as for interfacing with processors, sensors, external memories, and various components through the FPGA I/O. However, current OpenCL implementation for FPGAs is limited to high-performance computing purposes only. In contrast to an HDL programming, standard OpenCL programming for FPGA does not provide direct access to the FPGA's I/O pin. It is still difficult to employ OpenCL for accessing the FPGA I/O pin to interface with an external component. Therefore, this study aims to obtain the advantage of OpenCL implementation and HDL implementation for FPGA design, as shown in Fig.1.1. The main motivation behind carrying out this study is to open the way for using OpenCL programming for writing parallel programs that can be used not only for high-performance computing purposes but also useful for engineering applications.

1.2 Contributions

This thesis focuses on the implementation of FPGA programming using OpenCL which is motivated by the ease of use in contrast to HDL programming to reduce the development time. The implementation of FPGA-based design both for high-performance computing and engineering applications using OpenCL are presented. To evaluate the capability of OpenCL design for high-performance computing purposes, both external memory bandwidth and computational capability of FPGA are evaluated. Because FPGA is well known for its high bandwidth I/O capabilities such as for communicating or interfacing with external components, the I/O capability of FPGA is also evaluated using OpenCL specifically for engineering applications. This technique opens the way for using OpenCL for writing parallel programs on FPGA to access the FPGA's I/O directly from the OpenCL environment. Therefore, the main study of this thesis is focused on three categories as follow:

- external memory bandwidth, where it is evaluated by performing the memory-intensive application.
- computational capability, where it is evaluated by conducting the compute-intensive application.
- I/O capability, where it is evaluated by measuring and generating a signal through FPGA's I/O using OpenCL. Here, it is shown new OpenCL components that allow the OpenCL kernel to stream data to and from the FPGA's I/O directly.

In the high-performance computing application, first, the performance of the Himeno benchmark as an implementation of the memory-intensive application is evaluated. To increase performance, several kernels are optimized. According to the number of external memory accesses, kernel implementations are mainly divided into two groups: baseline implementation and optimized implementation. For baseline implementation, the OpenCL kernel is directly ported from the sequential C code. Compared to the baseline implementation, the optimized kernel increases the performance significantly by demonstrating the implementation of temporal blocking combined with shift register implementation simultaneously. Second, to evaluate the computational capability of FPGA using OpenCL, a matrix-to-matrix multiplication (GEMM), where it is as an example of compute-intensive application, is choosen to analyze the capability of FPGAs for HPC purposes. The performance-power ratio of multiple FPGAs is also measured. This performance-power ratio of FPGA is useful, particularly when implementing FPGA in an HPC application owing to many current applications that require FPGAs and take advantage of the benefits of FPGAs to achieve high performance with low power consumption.

To show the capability of OpenCL for accessing the FPGAs I/O, an experiment is conducted by developing the OpenCL kernel for signal measurement and generation which is divided into three categories. The first implementation is for signal measurement where the OpenCL kernel measures the input signal and stores the data in global memory of FPGA so that the host can read the data for further analysis. The second implementation is for signal generation, here, the host writes the data to global memory, and then the OpenCL kernel generates an output signal by reading the data from global memory. Finally, the third implementation is for signal measurement and generation. In this implementation, the first kernel reads a signal and writes it to an I/O channel. The second kernel reads the data from the I/O channel and generates a signal simultaneously. Here, data transfer is performed without accessing global memory.

1.3 Thesis organization

The rest of this thesis is organized as follows. First, the OpenCL for the Intel FPGA design is introduced in chapter 2. In chapter 3, the Himeno benchmark is performed as an example implementation of memory-intensive applications on FPGA. In chapter 4, matrix multiplication on a multiple FPGA system is explained as an example of compute-intensive applications. Chapter 5 shows the OpenCL implementation for signal measurement and generation on FPGA. Finally, this is followed by conclusions as presented in chapter 6.

Chapter 2

Field Programmable Gate Array design with OpenCL

2.1 Introduction to FPGA

FPGA, or Field-programmable gate array, is an integrated circuit that can be configured to perform any operation according to the desired application. FPGA consists of registers, lookup tables (LUTs), on-chip memories or Block RAMs (BRAMs), digital signal processor (DSP) blocks, and some hardware interconnects such as external memory controller and PCIe controller, as shown in Fig.2.1. These components are connected via a network of programmable or reconfigurable interconnects. LUTs are used to implement logic functions. To store a large amount of data inside an FPGA, BRAMs are used that serve as data storage for FPGA. For example, Stratix V FPGA consists of 50 Mbits of M20K memory blocks. Meanwhile, Arria 10 FPGA consists of 54 Mbits of M20K memory blocks as BRAMs.

In high-performance computing application, FPGA needs to perform either fixed or floating-point operations. To accommodate this purpose, a specially dedicated circuitry is necessary to perform the computation process on an FPGA. Instead of leveraging the FPGA resource such as LUTs and registers to implement mathematical operations, current FPGAs have been equipped with hardened DSP blocks to achieve high performance by applying the pipeline stage. For example, Intel Arria 10 FPGA has the variable-precision DSP block that delivers the maximum floating-point performance of up to 1.5 TFLOPs. Each DSP block supports either two 18 x 19 multipliers or one 27 x 27 multiplier that can be cascaded to perform a complex multiplication.

2.2 OpenCL for FPGA

In the last decade, FPGAs have the rapid increase in the circuit resources and the improvement of high-speed serial I/O interfaces, and it has helped FPGA-based acceleration to be accepted in various applications. Although Hardware Description Language (HDL) allows us to program FPGAs handily from low-level design such as Register-Transfer-Level (RTL) design, this common practice starts to break because current target applications are much more complicated and larger than the coverage of HDL. FPGA vendors have the interest to introduce High-Level Synthesis to their own FPGA CADs, and FPGAs with HLS become more widely adopted as an acceleration approach from embedded computing to HPC.

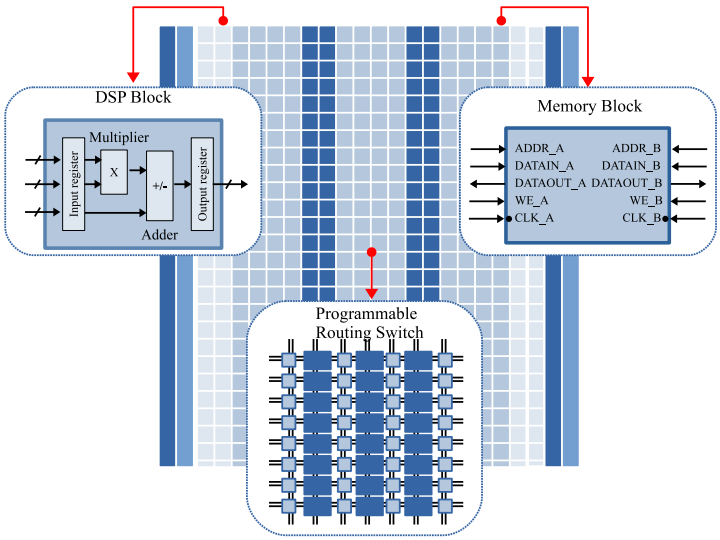

Figure 2.1: FPGA architecture.

OpenCL is one of the good high-level languages which has been spread as a heterogeneous computing framework this decade. Two major FPGA vendors, Xilinx and Intel, provide their OpenCL libraries in their CAD. As for Xilinx FPGA, SDAccel supports high-level FPGA designs by not only C/C++ but also OpenCL, which aims to alleviate the integration between software and hardware [14]. As for Intel FPGA, Intel Software Development Kit (SDK) in Quartus Prime supports OpenCL programming. It can grasp parallel programming algorithm written by the higher level of abstraction than HDL, and then a low-level design will be created for an FPGA [15]. Thus, the expected merits of HLS is not only to enable high-level design but also to create parallelization and to pipeline the circuit design automatically from a sequential processing design.

In other words, an FPGA-based accelerator with HLS will achieve a sufficient performance without the detailed circuit design. Then, interconnects among user logics, IPs, and interfaces on FPGA will be automatically generated, and the modules related to interfaces such as DDR memories and PCIe controllers will also be created automatically or used from prepared HLS libraries. This is just the beginning of the next generation design, and the current situation should be stated by the evaluation of OpenCL design. The summary of OpenCL on FPGA is shown in Fig.2.2 [15]. This figure illustrates units for loading and storing the data for each pipeline which are connected to DDR external memory via a global interconnect. For the local memory access, OpenCL generates an interconnect structure to on-chip M9K RAMs as well [16].

The positive impact of the introduction of OpenCL is to reduce the semantic gap between FPGA design and programming. It means FPGA users can focus only on high-level design such as architecture-level and system-level design. It helps to implement parallel programming applications and will reduce development time. In addition, OpenCL enables the reuse of developed modules on different platforms more easily compared to HDL. The negative prediction is that the FPGA design by the use of OpenCL cannot achieve sufficient performance since it is so difficult for OpenCL to optimize all circuit allocation on an FPGA. But, FPGA vendors start to steer the development environment to OpenCL because they believe OpenCL makes FPGA achieve a certain computing performance. Although the performance must be lower than that by HDL,

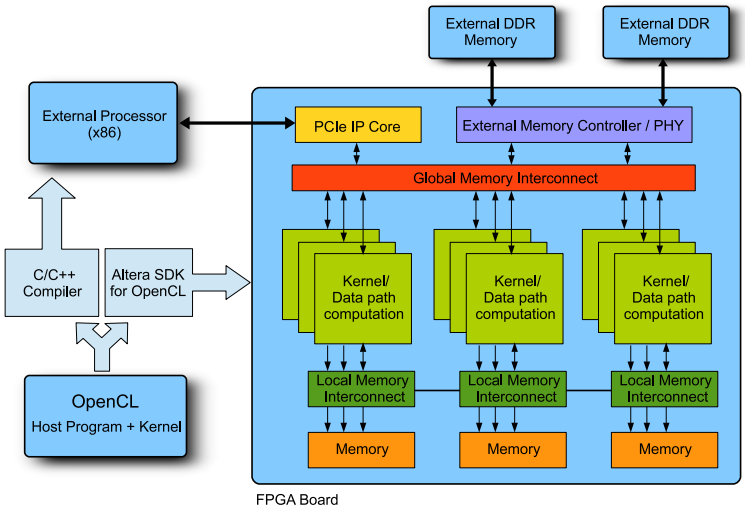

Figure 2.2: OpenCL overview and system implementation.

FPGA vendors think the next discussion will begin on not only performance but also testability, reliability, and development time.

OpenCL platform, particularly in Intel SDK for OpenCL, consists of a host PC and at least one FPGA device. In high-performance computing applications, the platform usually has more FPGA devices attached to PCIe connected to a host, where each FPGA device consists of multiple compute units to execute the OpenCL kernel. For a compute unit, it consists of multiple work-items or processing elements. To execute an OpenCL kernel on one or more FPGA devices, the host creates an OpenCL context so that the host can interact with the FPGA devices such as for managing the memory for the computation and controlling the FPGA for executing the kernel. The host also creates a command queue that provides the communication mechanism between host and FPGA devices [17]. For example, a host encapsulates the command in a command queue for reading and writing the data to FPGA and for executing the OpenCL kernel on FPGA. This execution model can be seen in Fig.2.3.

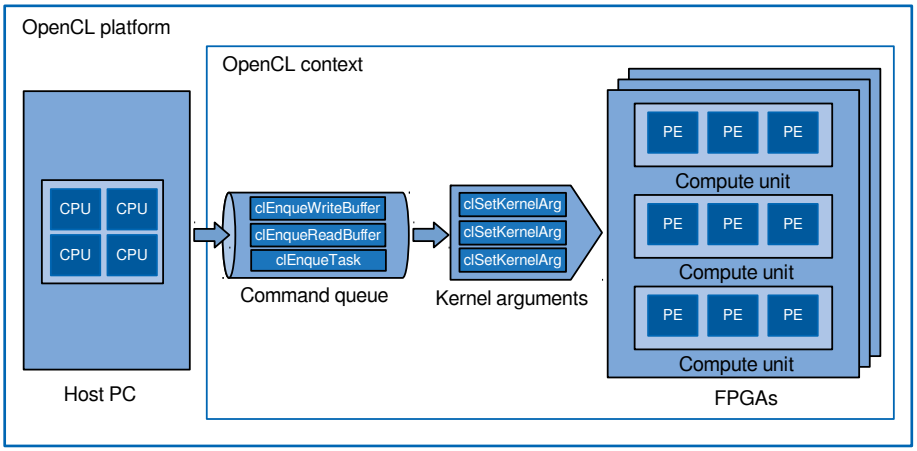

Figure 2.3: OpenCL platform for FPGA implementation.

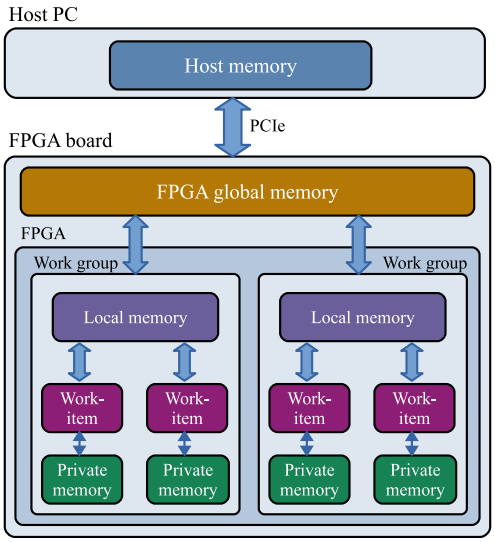

Figure 2.4: OpenCL memory architecture by Altera FPGA.

2.3 OpenCL memory architecture

To implement the OpenCL program on an Altera FPGA, several tasks are executed by host PC. The host is in charge of managing the data transfer from/to FPGA memory, invoking the kernel code to FPGA, handling the communication process and reading the final result from FPGA memory. OpenCL libraries handle these steps so that the users can concentrate on kernel programming code instead of designing the communication protocol between host and FPGA. During the process, each FPGA device can receive multiple command queues sent by the host PC and execute independent commands concurrently [18].

On FPGA, memory size and type need to be allocated beforehand in order to receive the data from the host. Like OpenCL, the Intel SDK for OpenCL defines several types of memory such as global/constant memory, local memory, and private memory [19]. The Intel Offline Compiler for OpenCL (AOC) can use DDR memory on FPGA board as global memory which is configured in a burst interleaved configuration by default. In this memory, constant memory resides as well. Local memory has a smaller size than global memory and has higher throughput with lower latency. Only work items on the same workgroup can access and share this memory. The last type, private memory, is implemented by using FPGA registers. The purpose of private memory is to store single variables or small arrays. This memory provides more bandwidth than other memories [20]. The location of memory types is illustrated in Fig.2.4. In this paper, the computation was implemented on DDR3 global memory, local memory and private memory of each FPGA board. In the rest of this article, we also defined host global memory as host memory, and DDR3 global memory of FPGA was defined as global memory.

2.4 FPGA pipeline parallelism

In OpenCL programming, GPU and FPGA share the same programming model particularly in kernel code which is a set of functions executed by OpenCL device. However, the kernels which are code portable across GPU and FPGA, are not portable

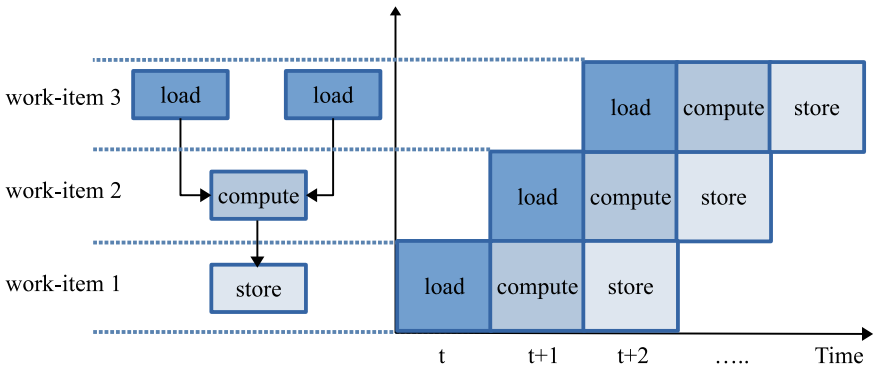

Figure 2.5: Pipeline data path for single work item.

in the performance results, meaning that the implementation of the same kernel yields different performance due to their underlying parallelism mechanism. In addition, they handle the parallelism in a different way. For GPU, where the hardware is composed of cores, the kernels are compiled into a sequence of instructions. These instructions will be executed on hardware cores that are specialized for different functions. In contrast, FPGA exploits the pipeline parallelism where the kernels are compiled into different stages of the instructions. These instructions are then applied to different cores or work items at the same time. As a result, FPGA often delivers better performance per watt than GPU [21].

Intel SDK for OpenCL provides the pipeline parallelism capability in executing the kernels in multi-threaded mode concurrently. It will also construct the FPGA as a parallel device consisting of multiple pipelining execution units. To enhance the performance, some compute units (CU) in a kernel will be multiplied within FPGA circuit resources. This is a spatial parallelism as same as GPUs parallelism and allows to execute multiple workgroups simultaneously. However, the bigger the number of compute units is, the larger the FPGA resource is required. It may cause the working frequency of multiple compute units may be lower than that of the single compute unit [22].

There are two types of kernel execution in Intel SDK for OpenCL, NDRange kernels and single work-item kernels or task kernels. NDRange kernels, where it consists of many work-items that are processed in parallel, share the data during the computation process among many work-items by leveraging local memory which is identified by a

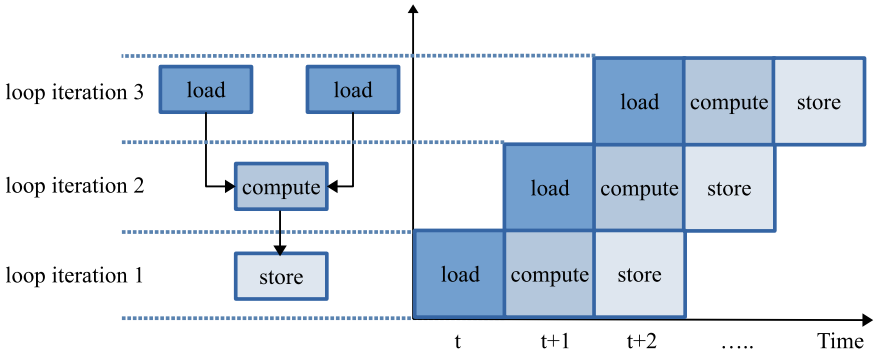

Figure 2.6: Pipeline data path for loop iteration.

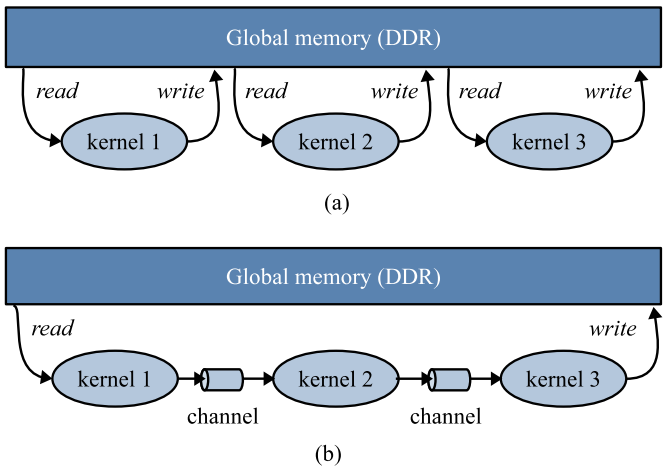

Figure 2.7: Kernel-to-kernel communication (a) without channel through global memory (b) with channel implementation.

local work-item ID. Meanwhile, for the single work-item kernel, the data are shared among multiple loop-iterations using register or private memory. In terms of kernel execution, an NDRange kernel launches work-items in a pipeline manner one after another one as shown in Fig.2.5. On the other hand, because a single work-item kernel has multiple loop-iterations. The kernel executes these multiple loop-iterations simultaneously in different pipeline stages [17], as shown in Fig.2.6.

2.5 OpenCL channel extension

In OpenCL design, particularly for Intel SDK for OpenCL, a kernel needs to communicate with global memory to read and write data for the computation process. When two or more kernels are executed to solve computational problems, more communications and data transfers are performed between the kernels and global memory. Compared to a GPU that supports a high-bandwidth global memory, most FPGA boards are equipped with DDR3 or DDR4 as the global memory. As a result, this causes a reduction in performance owing to the global memory bandwidth bottleneck, as shown in Fig.2.7(a). To overcome this constraint, an OpenCL channel extension is employed to transfer data among the kernels without accessing global memory, as shown in Fig.2.7(b). The channel extension is a first-in-first-out (FIFO) buffer. The channel is implemented using RAM blocks and registers [23] [17]. In this study, the OpenCL channel extension is employed to stream data between the OpenCL kernel and the FPGA's I/O directly from the OpenCL environment.

Chapter 3

FPGA-based Implementation of Memory-Intensive Application using OpenCL

3.1 Introduction

Scientific applications in high-performance computing can be divided into computation-intensive and memory-intensive, or data-intensive operations. The memory-intensive application has a high ratio of memory accesses to instructions. One of the barriers in memory-intensive applications is that the memory bandwidth does not increase at the same rate as the computational performance [24]. The application-development roadmap, which was summarized in Japan in 2012, showed that many scientific and engineering applications in high-performance computing were memory-intensive. The ratio of memory throughput to computing performance for memory-intensive applications, or B/FLOP (B/F), is greater than or equal to 0.5 B/F [25]. Several examples of the memory-intensive applications include live migration techniques for resource consolidation and fault tolerance [26], database management and multimedia processing [27], genome-scale computational [28] and Himeno benchmark.

The Himeno benchmark has been widely implemented on the GPU cluster owing to the high memory bandwidth. However, few studies show the evaluation of the Himeno benchmark on FPGA using high-level synthesis, particularly OpenCL. This is because current FPGA boards are still used DDR memory in their designs. This study focuses on the evaluation of the Himeno benchmark as a memory-intensive application on FPGA using OpenCL by implementing the temporal blocking combined with shift-register implementation for optimization. During the computation, the kernel requires a large amount of data to be read from and written to the global memory of FPGA. Therefore, the performance depends on the memory bandwidth.

3.2 Himeno benchmark

The Himeno benchmark, which was developed by Dr. Ryutaro Himeno, was designed to evaluate the performance of analysis in the incompressible fluid. This benchmark measures the processing speed in the main loops, whereas the Poisson equation is solved by the Point–Jacobi iteration method [29]. The equation of the Point–Jacobi method can be seen in Equation 3.1. This equation is a linear solver for the 3D pressure Poisson equation. It also appears in an incompressible Navier–Stokes solver [30].

The performance is shown in the units of single-precision floating-point operations per second (FLOPS).

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} + \frac{\partial^2 p}{\partial z^2} + \alpha \frac{\partial^2 p}{\partial xy} + \beta \frac{\partial^2 p}{\partial xz} + \gamma \frac{\partial^2 p}{\partial yz} = \rho \tag{3.1}$$

The original version of sequential C code for the Himeno benchmark is shown in Listing 3.1. It contains the 3D arrays, such as arrays $a0$ to $a3$, $b0$ to $b2$, $c0$ to $c2$, $wrk1$, $wrk2$, bnd, and p as a pressure array. According to the code, the p array can also be accessed using the stencil pattern, as depicted in Fig.3.1. In the main compute kernel, there are 34 floating-point operations, 31 read accesses from the external memory, and 1 write access to the external memory. For the naïve implementation, the ideal B/F ratio can be calculated by $(31+1) \times 4\,B/34\,FLOP = 3.764\,B/FLOP$. However, when the cache blocking technique for the p array is applied, the number of reading accesses from the external memory can be reduced to 13. Therefore, the B/F value will improve to $(13+1) \times 4\,B/34\,FLOP = 1.647\,B/FLOP$. Because the B/F ratio is greater than or equal to 0.5, the above computation is a memory-intensive application.

To measure the peak performance related to the memory bandwidth, the F/B ratio is required. Therefore, the peak performance of the Himeno benchmark without and with optimization can be calculated by Equations 3.2 and 3.3, as follows:

$$performance_{wo/_{opt}} = \frac{34[FLOP] \times bandwidth[B/s]}{128[B]} \tag{3.2}$$

when cache blocking is applied, the equation is changed to

$$performance_{w/_{opt}} = \frac{34[FLOP] \times bandwidth[B/s]}{56[B]} \tag{3.3}$$

These equations simplify the peak performance prediction of the Himeno benchmark as the external memory bandwidth is the only defining parameter. The example of the predicted peak performance for FPGAs, CPUs, and GPUs can be seen in Figure 3.2 and Table 3.1 . It can be seen that the larger the memory bandwidth, the higher the performance. From the table, the peak performance can be obtained by implementing the on-chip caches for the computation.

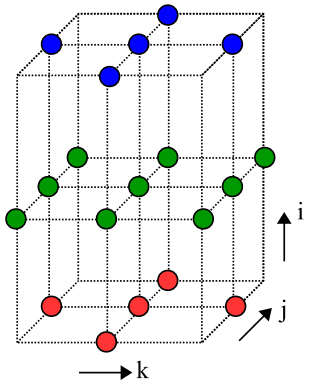

Figure 3.1: 3D stencil of p as a pressure array in Himeno benchmark.

Listing 3.1: C code snippet for Himeno benchmark.

```
........
float jacobi(int nn)
{
  int i,j,k,n;
  float gosa, s0, ss;

  for(n=0;n<nn;++n){
    gosa = 0.0;

    for(i=1 ; i<imax-1 ; ++i)
      for(j=1 ; j<jmax-1 ; ++j)
        for(k=1 ; k<kmax-1 ; ++k){
          s0  = a[i][j][k][0] * p[i+1][j ][k ]
              + a[i][j][k][1] * p[i ][j+1][k ]
              + a[i][j][k][2] * p[i ][j ][k+1]
              + b[i][j][k][0] * (p[i+1][j+1][k ] - p[i+1][j-1][k ]
                                 - p[i-1][j+1][k ]  + p[i-1][j-1][k ])
              + b[i][j][k][1] * (p[i ][j+1][k+1] - p[i ][j-1][k+1]
                                 - p[i ][j+1][k-1]  + p[i ][j-1][k-1])
              + b[i][j][k][2] * (p[i+1][j ][k+1] - p[i-1][j ][k+1]
                                 - p[i+1][j ][k-1]  + p[i-1][j ][k-1])
              + c[i][j][k][0] * p[i-1][j ][k ]
              + c[i][j][k][1] * p[i ][j-1][k ]
              + c[i][j][k][2] * p[i ][j ][k-1]
              + wrk1[i][j][k];
          ss  = ( s0 * a[i][j][k][3] - p[i][j][k] ) * bnd[i][j][k];
          gosa = gosa + ss*ss;
          wrk2[i][j][k] = p[i][j][k] + omega * ss;
        }

    for(i=1 ; i<imax-1 ; ++i)
      for(j=1 ; j<jmax-1 ; ++j)
        for(k=1 ; k<kmax-1 ; ++k)
          p[i][j][k] = wrk2[i][j][k];
  } /* end loop */
}
.......
```

Table 3.1: Predicted peak performance of Himeno benchmark for CPU, GPUs, and FPGAs.

Devices	Memory types	Bandwidth (GB/s)	Wo/ opt (GFlops)	W/ opt (GFlops)
Stratix V DE5-Net FPGA [31]	DDR3-1600 (2 Banks)	23.8	6.32	14.45
Arria 10 DE5a-Net FPGA [32]	DDR4-1200 (2 Banks)	38.4	10.2	23.31
Intel Xeon E5-1650 CPU	DDR4-2133	34.2	9.07	20.74
NVidia GTX1080 GPU [33]	GDDR5 (8 GB)	320	85	194.29
Xilinx Ultrascale+ FPGA [34]	HBM (8 GB)	460	122.19	279.28
Intel Stratix 10 MX FPGA [35]	HBM2 (8 GB)	512	136	310.86
NVidia P100 GPU [33]	HBM2 (32 GB)	549	145.8	333.32
NVidia V100 GPU [33]	HBM2 (32 GB)	900	239.06	546.43

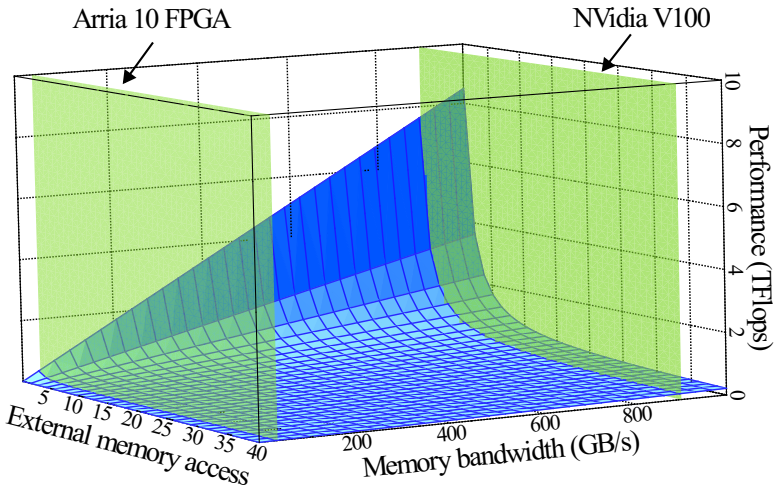

Figure 3.2: Predicted peak performance of the Himeno benchmark according to external memory access and memory bandwidth.

3.3 System implementation

In this study, a single task OpenCL kernel is developed to test the performance of the Himeno benchmark. Compared to the NDRange kernel type, a kernel that is executed by many work-items, the single-task kernel consists of one work item. This type of kernel is also similar to that of sequential C code. However, to employ pipeline-level parallelism in the single task kernel, many loop-iterations are computed using a different pipeline stage or loop-pipelining inside the loops [36].

3.3.1 Stratix V DE5-Net FPGA overview

The implementation of the Himeno benchmark on Altera Stratix V DE5-Net FPGA [31] is demonstrated. The host PC consists of Intel Xeon E5-1650 CPU, 64 GB DDR4-2133, Centos 7.0 Linux 64-bit operating system, gcc version 4.4.7, Altera Quartus 16.1 64-bit and Altera SDK for OpenCL 16.1.0 Build 196. In the OpenCL implementation, the FPGA Board Support Package (BSP) supports the Gen 2x8 lanes. By using OpenCL, the maximum throughput was 2.9 GB/s. The maximum memory bandwidth

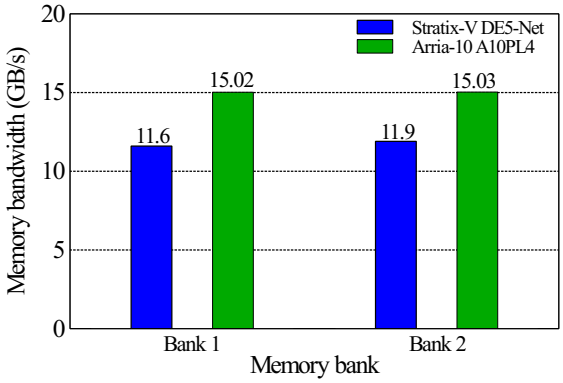

Figure 3.3: Memory bandwidth for Stratix V DE5-Net and Arria 10 A10PL4 FPGA.

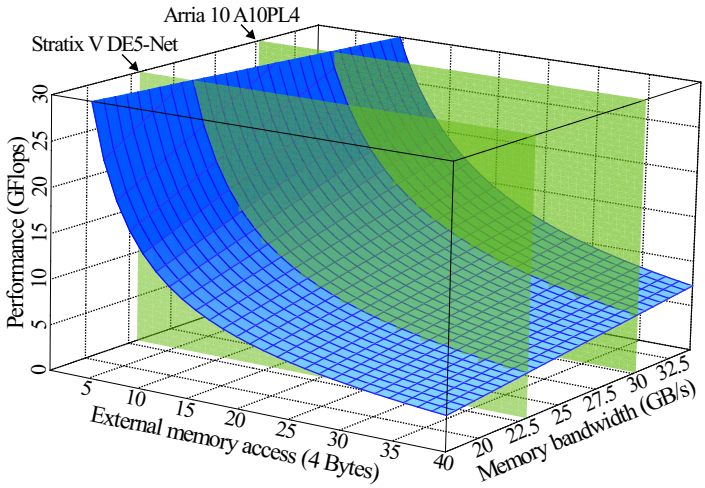

Figure 3.4: Performance estimation of Himeno benchmark for Stratix V DE5-Net and Arria 10 A10PL4 FPGA.

between the two external DDR3 memories and the Stratix V DE5-Net FPGA was also measured. This test was performed using a memory stream kernel. The maximum memory bandwidths achieved 11.6 GB/s and 11.9 GB/s for memory bank 1 and 2, respectively as shown in Figure 3.3.

3.3.2 Arria 10 A10PL4 FPGA overview

The performance of the Himeno benchmark is also evaluated on Arria 10 A10PL4 FPGA with the following specifications: Intel Arria 10 GX FPGA, two banks DDR4 with ECC, and PCIe Gen3x8 [37]. The host PC consists of Intel Xeon CPU E5-2660 v4, Linux Red Hat 4.8.5-28, and Intel SDK for OpenCL version 17.1.2.304. From the experiment, by using OpenCL Board Support Package (BSP), the maximum memory bandwidth for each bank is 15.02 GB/s and 15.03 GB/s for bank 1 and bank 2, respectively as shown in Figure 3.3. Meanwhile, for PCIe throughput, the maximum data transfer between the host and the FPGA is 6.42 GB/s.

3.4 Experimental Results

The peak performance of the Himeno benchmark can be estimated based on the external memory accesses and memory bandwidth. By using Equations 3.2 and 3.3, a 3D model to estimate the performance for Stratix V DE5-Net FPGA and Arria 10 A10PL4 FPGA is developed, as shown in Fig.3.4. In this experiment, the Himeno benchmark for $128 \times 64 \times 64$, $256 \times 128 \times 128$, and $512 \times 256 \times 256$ single-precision floating-point data sizes was performed. According to the size of the external memory accesses, the kernels are divided into two main groups: baseline implementation and optimized implementation.

3.5 Baseline implementation

In this implementation, there are 34 floating-point operations, 31 read accesses from external memory, and 1 write access to the external memory of the FPGA. According to the Equation 3.2, the F/B ratio can be calculated as 0.266. Because the total memory

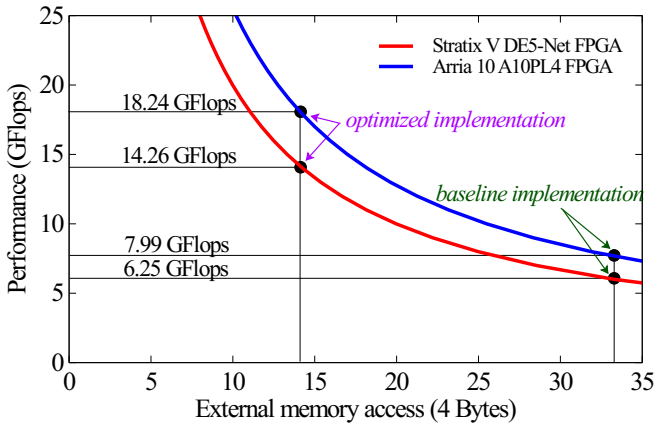

Figure 3.5: Performance estimation in 2D model.

bandwidth is 23.5 GB/s and 30.5 GB/s for Stratix V FPGA and Arria 10 FPGA, respectively, the theoretical performance can be calculated by $23.5[GB/s] \times 0.266[FLOP/B] = 6.25[GFLOPS]$ for Stratix V DE5-Net. Meanwhile for Arria 10 A10PL4 FPGA, the theoretical performance can be calculated by $30.05[GB/s] \times 0.266[FLOP/B] = 7.99[GFLOPS]$, as presented in in Fig.3.5.

In this implementation, the naïve kernel is first evaluated, which is ported from sequential C code to OpenCL kernel code directly without any optimizations, as presented in Listing 3.2. The second kernel is the loop unrolling kernel, where the loop unrolling directive is applied to increase the performance.

3.5.1 Naïve kernel

Here, the original sequential C code is ported to the OpenCL kernel as shown in Listing 3.2. The main difference is the use of a global declaration to store the variable from the host. Additionally, to inform the offline compiler that the type of the kernel is a single task kernel, the $\_attribute\_((task))$ is declared. This declaration invokes the compiler to generate the pipeline stage inside loop-iterations. From the experiment, the peak performance for Stratix V DE5-Net is 2.63 GFLOPS, or 42% of the theoretical performance. On the other hand, Arria 10 A10PL4 achieves 2.43 GFLOPS, or 30% of

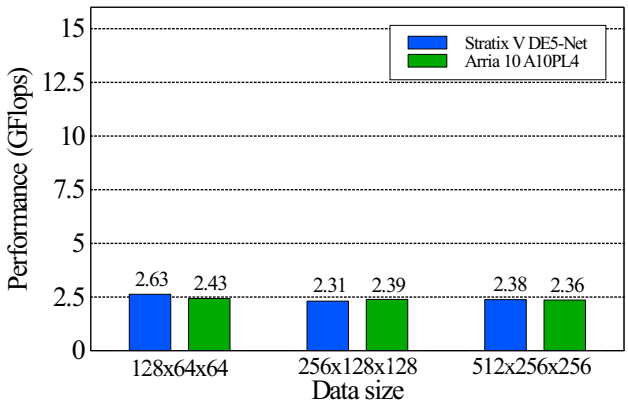

Figure 3.6: Performance for naïve implementation.

Table 3.2: Kernel compilation reports for Stratix V DE5-Net FPGA.

Kernel	Logic	RAM blocks	Memory (Mbits)	DSP blocks	Clock (MHz)
kernel 1	63K(27%)	1,034(40%)	13(25%)	12 (5%)	243.8
kernel 2	83K(36%)	1,075(42%)	12.7(24%)	24 (9%)	255.6
kernel 3	94K(40%)	1,473(58%)	17.4(33%)	24 (9%)	249.8
kernel 4	115K(49%)	990(39%)	11.6(22%)	96 (38%)	252.4
kernel 5	176K(75%)	1,544(60%)	11.7(22%)	192 (75%)	235.2

Table 3.3: Kernel compilation reports for Arria 10 A10PL4 FPGA.

Kernel	Logic	RAM blocks	Memory (Mbits)	DSP blocks	Clock (MHz)
kernel 1	65K(15%)	975(36%)	13(24%)	21 (1%)	198.8
kernel 2	99K(23%)	1,075(40%)	14.5(26%)	84 (6%)	204.2
kernel 3	99K(23%)	1,845(68%)	23.1(42%)	84 (6%)	194.2
kernel 4	79K(19%)	902(33%)	14(26%)	168 (11%)	205.2
kernel 5	107K(25%)	942(35%)	14(26%)	336 (22%)	205.5

the theoretical performance for the $128 \times 64 \times 64$ data size, as shown in Fig.3.6. In this kernel, FPGAs consume 12 and 21 DSP blocks for Stratix V and Arria 10, respectively, as presented as kernel 1 in Table 3.2 and Table 3.3. In this kernel, the Stratix V has a higher performance than that of Arria 10 FPGA owing to the higher working frequency.

3.5.2 Loop Unrolling kernel

In a single work-item kernel type, loop unrolling can be used to boost the performance by applying the *#pragma unroll* $< N >$ directive. Loop unrolling increases the degree of parallelism and allows the kernel to process more data within one FPGA clock cycle [17]. By implementing loop unrolling, the peak performance of the Himeno benchmark on Stratix V increases to 5.79 GFLOPS, or 92% of the theoretical performance. For Arria 10 FPGA, the performance increases to 7.18 GFLOPS, or 90% of the theoretical performance for the $128 \times 64 \times 64$ data size, as shown in Fig.3.7. Table 3.2 and Table 3.3 show the kernel compilation reports for kernel 2. According to the tables, loop unrolling increases the logic resources and the number of DSP blocks. In this kernel, FPGAs require 24 and 84 DSP blocks for Stratix V and Arria 10, respectively.

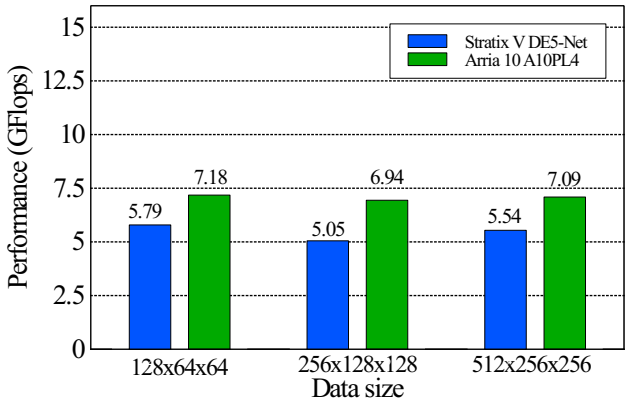

Figure 3.7: Performance for unrolling implementation.

Listing 3.2: Naïve code for the Himeno benchmark using Intel SDK for OpenCL.

```
#define IMAX 256
#define JMAX 128
#define KMAX 128

__kernel __attribute__((task))
void himeno_benchmark(
      __global const float *restrict a1, __global const float *restrict a2,
      __global const float *restrict a3, __global const float *restrict a4,
      __global const float *restrict b1, __global const float *restrict b2,
      __global const float *restrict b3, __global const float *restrict c1,
      __global const float *restrict c2, __global const float *restrict c3,
      __global const float *restrict bnd,__global const float *restrict wrk1,
      __global float *restrict p, __global float *restrict wrk2,
      __global float *restrict gosa_out, float omega, int nn)
{
   #define idxz(i,j,k) (k+(JMAX)*(j+(KMAX)*i))
   int i,j,k,l,n;
   float s0, ss, gosa;

   for(n=0; n<nn; n++){
      gosa = 0.0f;
      for(i=1; i<IMAX-1; i++){
          for(j=1; j<JMAX-1; j++){
             for(k=1; k<KMAX-1; k++){
              s0  =    a1[idxz(i,j,k)] * p[idxz(i+1,j,k)]
                     + a2[idxz(i,j,k)] * p[idxz(i,j+1,k)]
                     + a3[idxz(i,j,k)] * p[idxz(i,j,k+1)]
                     + b1[idxz(i,j,k)] *(p[idxz(i+1,j+1,k)] - p[idxz(i+1,j-1,k)]
                                        - p[idxz(i-1,j+1,k)] + p[idxz(i-1,j-1,k)])
                     + b2[idxz(i,j,k)] * (p[idxz(i,j+1,k+1)] - p[idxz(i,j-1,k+1)]
                                        - p[idxz(i,j+1,k-1)] + p[idxz(i,j-1,k-1)])
                     + b3[idxz(i,j,k)] * (p[idxz(i+1,j,k+1)] - p[idxz(i-1,j,k+1)]
                                        - p[idxz(i+1,j,k-1)] + p[idxz(i-1,j,k-1)])
                     + c1[idxz(i,j,k)] * p[idxz(i-1,j,k)]
                     + c2[idxz(i,j,k)] * p[idxz(i,j-1,k)]
                     + c3[idxz(i,j,k)] * p[idxz(i,j,k-1)]
                     + wrk1[idxz(i,j,k)];
             ss = (s0 * a4[idxz(i,j,k)] - p[idxz(i,j,k)]) * bnd[idxz(i,j,k)];
             gosa = gosa + ss*ss;
             wrk2[idxz(i,j,k)] = p[idxz(i,j,k)] + omega*ss;
      }}}

      for(i=1; i<IMAX-1; i++){
          for(j=1; j<JMAX-1; j++){
             for(k=1; k<KMAX-1; k++){
                p[idxz(i,j,k)] = wrk2[idxz(i,j,k)];
      }}}
}}
```

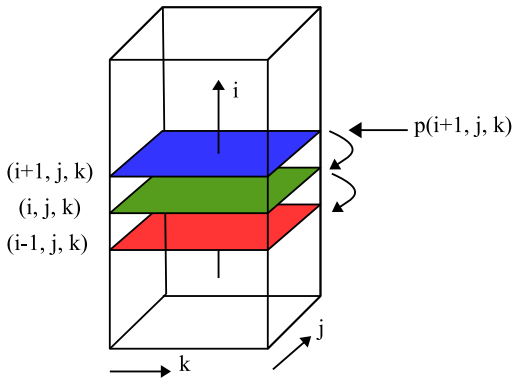

Figure 3.8: Optimization using temporal blocking implementation.

3.6 Optimized implementation

According to the C code for the Himeno benchmark, which is shown in Listing 3.1, the pressure p array is the most frequently accessed array from global memory. Therefore, in this optimized implementation, the array p is cached into the on-chip RAM of the FPGA. Compared to the baseline implementation, the optimized implementation has a higher F/B ratio owing to reductions in external memory accesses. The number of reading accesses from the external memory is reduced to 13. According to the Equation 3.3, the F/B ratio is 0.607. Therefore, the theoretical peak performance can be calculated by $23.5[GB/s] \times 0.607[FLOP/B] = 14.26[GFLOPS]$ for Stratix V FPGA, and $30.05[GB/s] \times 0.607[FLOP/B] = 18.24[GFLOPS]$ for Arria 10 FPGA, as shown in Fig.3.5.

In this optimized implementation, three kernels are developed as follows: temporal blocking kernel, shift register kernel, and temporal blocking combined with shift register kernel.

3.6.1 Temporal blocking kernel

First, the optimization is performed by implementing temporal blocking as shown in Fig.3.8. Three layers of a 2D array are created, namely $p1[j][k]$, $p2[j][k]$, and $p3[j][k]$, by using on-chip RAM of the FPGA that are swapped in the Z-direction for caching

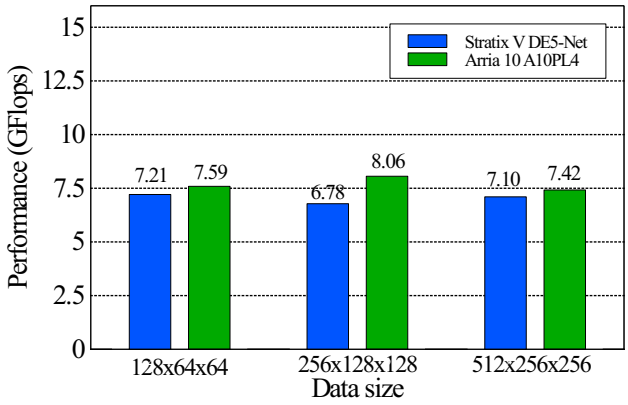

Figure 3.9: Performance for temporal blocking implementation.

Algorithm 1: Optimized kernel using temporal blocking.

```
   Input  : a1..a4, b1..b3, c1..c3, bnd, p, wrk1, wrk2
 1 p1[jmax][jmax];p2[jmax][jmax];p3[jmax][jmax];  // p1, p2, p3 as local memory
 2 data_initialization();
 3 for i = 1 to i = (imax − 1) do
 4 |   for j = 1 to j = (jmax) do
 5 |   |   for k = 1 to k = (kmax) do
 6 |   |   |   p1[j][k]=p2[j][k];
 7 |   |   |   p2[j][k]=p3[j][k];
 8 |   |   |   p3[j][k]=p(i+1, j, k);   // read p array from global to local memory
 9 |   |   end
10 |   end
11 |   for j, k = 1 to j, k = (jmax − 1) do
12 |   |   main_computation();                          // main computation is here
13 |   end
14 end
```

the p array. At the initialization, the kernel copies $p(0, j, k)$ from global memory to $p2[j][k]$ arrays on the on-chip RAM and copies from $p(1, j, k)$ to $p3[j][k]$. After reading the p array $p(1, j, k)$ from global memory to the first layer of variables, the content of the data in second layer was copied to the third layer. For the next iteration, the kernel reads the $p(1 + i, j, k)$ data from global memory and copies it to $p3[j][k]$. Before that, the content of data on $p2[j][k]$ was copied to $p1[j][k]$, and previous data on $p3[j][k]$ was copied to $p2[j][k]$. This process was executed inside the inner loop, as shown in Algorithm 1.

In the experiment, the peak performance increases to 7.21 GFLOPS, or 50.5% of the theoretical performance for Stratix V FPGA. For Arria 10 FPGA, the performance increases to 8.06 GFLOPS, or 44.2% of the theoretical performance, as shown in Fig.3.9. From the kernel compilation report, it is observed that temporal blocking increases the usage of RAM blocks, as described as kernel 3 in Table 3.2 and Table 3.3. The RAM block usage increases to 1,473 (58%) and 1,845 (68%) for Stratix V and Arria 10 FPGA, respectively. The compilation report also shows that the copy operation in the temporal blocking kernel produces data dependency among the variables on the on-chip RAM. Consequently, some iterations are executed serially and the performance is not fully optimized. In the following optimization, it is demonstrated how to avoid the data dependency caused by a copy operation to increase the performance.

3.6.2 Shift-register kernel

Second, the kernel is optimized to remove the data dependency by implementing a shift register pattern as shown in Fig.3.10. The implementation of a shift-register yields a more efficient result compared to the on-chip RAMs accesses [38]. The shift-register also improves the efficiency of external memory accesses via the use of temporal locality, which reduces global memory accesses [39]. In OpenCL design, a shift-register implementation can be seen in Listing 3.3. In this kernel, one element of $p(i-1, j, k)$, $p(i, j, k)$, and $p(i + 1, j, k)$ is shifted from global memory to the $p1[]$, $p2[]$, and $p3[]$ registers, respectively, for every iteration t as shown in Algorithm 2.

To reduce the size of the register, the data is stored on the register for a period of $(2 \times array_{width} + 4)$ cycles, as shown in Fig.3.11. This is called the period of the lifetime of the data [40]. For example, to evaluate the performance for the $(128 \times 64 \times 64)$ data

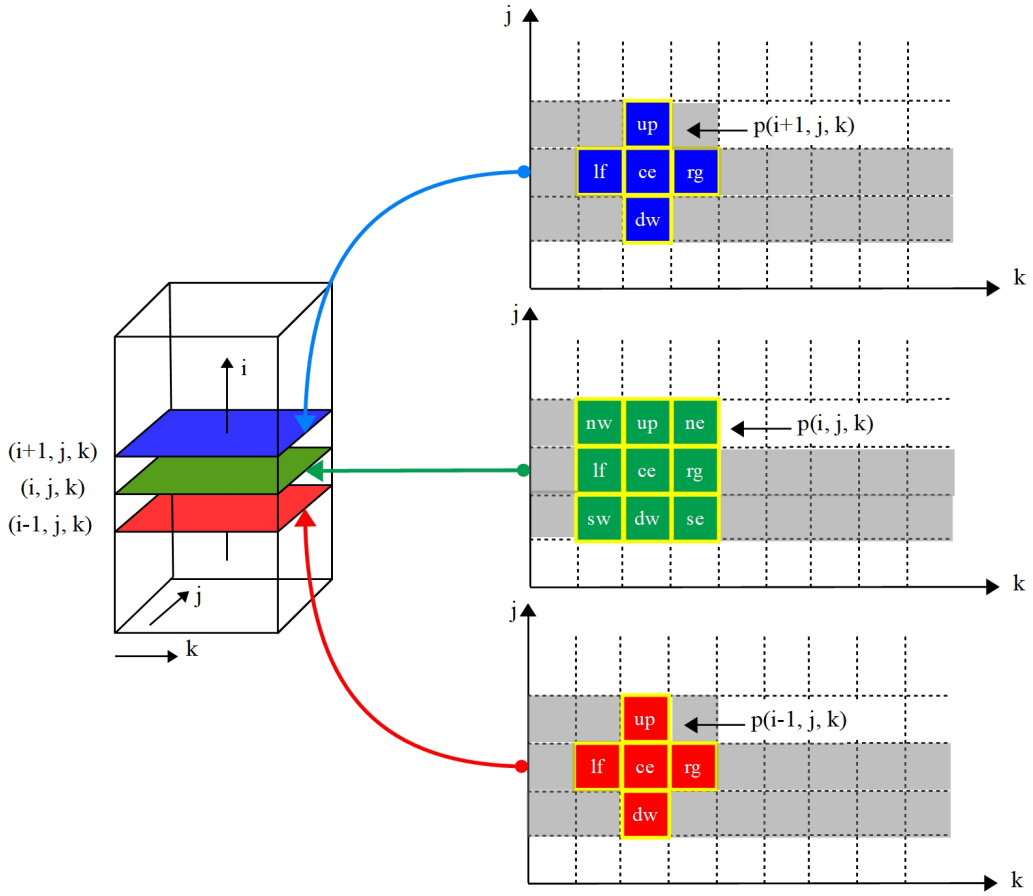

Figure 3.10: Optimization using shift-register implementation.

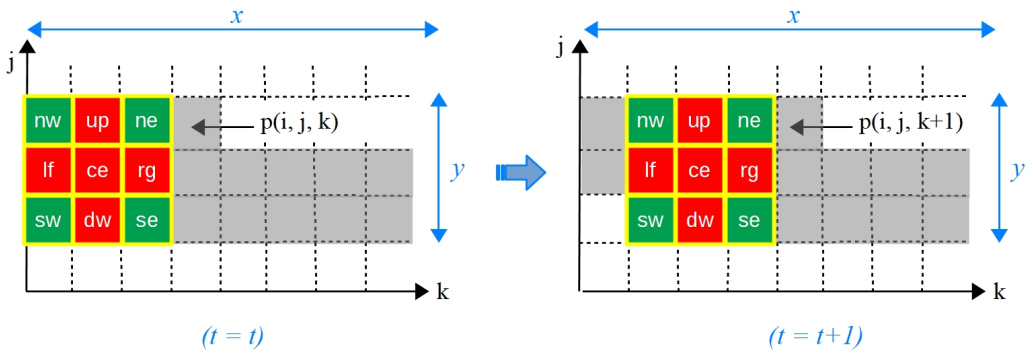

Figure 3.11: Shift-register implementation using OpenCL.

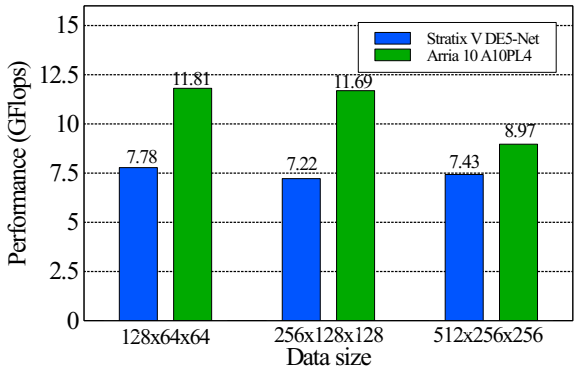

Figure 3.12: Performance for shift-register implementation.

Listing 3.3: Code snippet for the shift-register implementation using OpenCL.

```
1  ..........
2  #define kmax 64
3
4  for (int s = (2*kmax)+4; s > 0; s--)
5  {
6       shiftreg[s] = shiftreg[s-1];
7  }
8  shiftreg[0] = p[(i,j,k)];
9  ..........
```

size, the length of the shift register is $(2 \times 64 + 4)$.

From the experiment, the peak performance increases to 7.78 GFLOPS, or 55% of the theoretical performance for the Stratix V FPGA. Meanwhile, for Arria 10 FPGA, the peak performance increases to 11.81 GFLOPS, or 65% of the theoretical performance, as shown in Fig.3.12. By analyzing the RAM block usage, it is shown that the resource usage decreases to 990 and 902 for Stratix V and Arria 10 FPGA, respectively, as shown as kernel 4 in Table 3.2 and Table 3.3. From the tables, the increase in the number of DSP blocks for both FPGAs is due to the increase in the value of loop unrolling constant. From the experiments, FPGAs consume 96 and 168 DSP blocks for Stratix V and Arria 10, respectively.

3.6.3 Temporal blocking combined with shift register kernel

Third, the kernel is optimized to remove the data dependency and reduce the number of global memory accesses by implementing temporal blocking combined with a shift register pattern simultaneously, as depicted in Fig.3.13. In the temporal blocking implementation, this produces memory dependency, particularly on the on-chip RAM, owing to the copy operation. In contrast, shift register implementation results in memory access efficiency owing to temporal locality. In this kernel, the computation processes are similar to those in the temporal blocking technique. However, the kernel fetches the data from global memory by shifting the data to the shift register, as shown in Algorithm 3.

From the experimental results, the peak performance increases to 10.62 GFLOPS,

Algorithm 2: Optimized kernel using shift-register implementation.

```
   Input  : a1..a4, b1..b3, c1..c3, bnd, p, wrk1, wrk2
 1 p1[2*jmax+4];p2[2*jmax+4];p3[2*jmax+4]; // p1, p2, p3 as private memory
 2 data_initialization();
 3 for i = 1 to i = (imax - 1) do
 4     for j = 1 to j = (jmax - 1) do
 5         for k = 1 to k = (kmax - 1) do
 6             for m = (2 * jmax + 4) to m > 0 do
 7                 p1[m]=p1[m-1];                  // shift register implementation
 8                 p2[m]=p2[m-1];
 9                 p3[m]=p3[m-1];
10             end
11             p1[0]=p(i-1, j, k);                 // shift register implementation
12             p2[0]=p(i, j, k);
13             p3[0]=p(i+1, j, k);
14             main_computation();                     // main computation is here
15         end
16     end
17 end
```

Algorithm 3: Optimized kernel using temporal blocking combined with shift-register implementation.

```
   Input  : a1..a4, b1..b3, c1..c3, bnd, p, wrk1, wrk2
 1 shift_p1;shift_p2;shift_p3;           // private memory with [2*jmax+4] length
 2 p1;p2;                                  // local memory with [jmax][kmax] size
 3 data_initialization();
 4 for i = 1 to i = (imax - 1) do
 5     for j = 1 to j = (jmax - 1) do
 6         for k = 1 to k = (jmax - 1) do
 7             for m = (2 * jmax + 4) to m > 0 do
 8                 shift_p1[m]=shift_p1[m-1];      // shift register implementation
 9                 shift_p2[m]=shift_p2[m-1];
10                 shift_p3[m]=shift_p3[m-1];
11             end
12             shift_p3[0]=p(i+1, j, k);                 // read p array from global
13             shift_p2[0]=p2[j][k];            // shift data from global to private
14             shift_p1[0]=p1[j][k];
15             main_computation();                     // main computation is here
16             p1[j][k] = p2[j][k];                          // copy from p2 to p1
17             p2[j][k] = shift_p3[0];                 // copy from register to p2
18         end
19     end
20 end
```

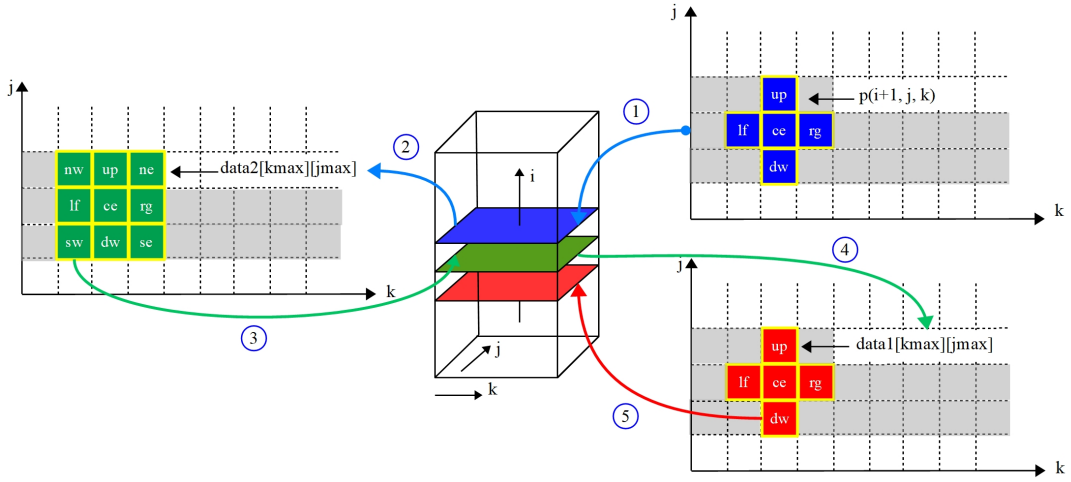

Figure 3.13: Optimization using temporal blocking combined with shift register implementation.

or 74% of the theoretical performance for Stratix V FPGA. For Arria 10 FPGA, the peak performance increases to 13.95 GFLOPS, or 76% of the theoretical performance, as shown in Fig.3.14. We also analyzed the kernel compilation report. From the report, it is observed that the iterations in the execution are launched every cycle and pipelined well. Consequently, the performance increases significantly. Moreover, the data dependency caused by temporal blocking can be omitted. Because of this, in this implementation, the value of the loop unrolling factor was increased to boost the performance. Thus, the logic resources and the number of DSP blocks increases, as shown as kernel 5 in Table 3.2 and Table 3.3. In this kernel, FPGAs consume 192 and 336 DSP blocks for Stratix V and Arria 10, respectively.

3.6.4 Performance of Himeno benchmark on heterogeneous system

To compare the peak performance of the Himeno benchmark on FPGAs, it is presented the performance for CPU and GPU as shown in Fig.3.15. For CPU evaluation, the benchmark was performed on Intel Xeon CPU E5-1650 v3 and 64 GB DDR4-2133 with Centos 7 using gcc version 4.8.5. The peak performance achieved 4.71 GFLOPS.

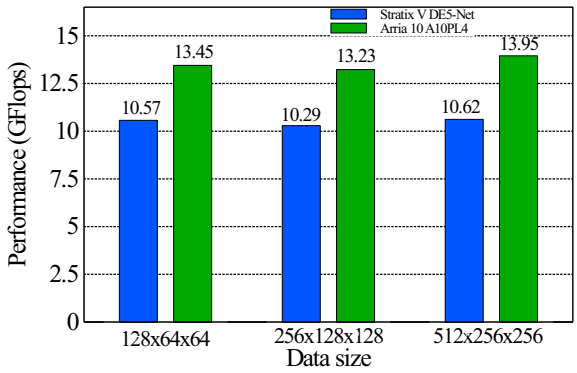

Figure 3.14: Performance for temporal blocking combined with shift-register implementation.

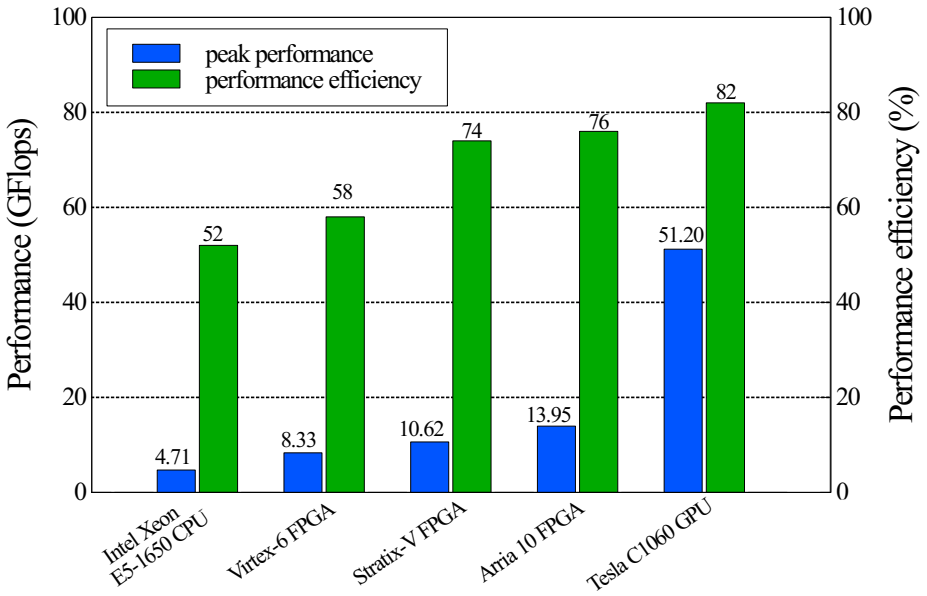

Figure 3.15: Performance of Himeno benchmark on heterogeneous system.

In [41], it is demonstrated the evaluation of the Himeno benchmark using MaxCompiler version 2012.1 on a MAX3 acceleration card that contains a Virtex-6 SX475T FPGA, DDR3 memory, and PCIe gen2x8. The peak performance achieves 8.33 GFLOPS that is obtained by the stream communication via PCIe between the FPGA and the host PC. Our implementations show that the peak performance reaches 10.62 GFLOPS and 13.95 GFLOPS for Stratix V and Arria 10 FPGA respectively. These peak performances outperform the Intel Xeon E5 CPU and Virtex-6 FPGA's performance. In [42], the performance on a single NVidia C1060 GPU is shown, where the peak performance reaches 51.20 GFLOPS. Here, the C1060 GPU outperforms the FPGA's performance owing to the higher memory bandwidth (102 GB/s).

3.7 Conclusions

The optimization in the OpenCL kernel has been demonstrated to increase the performance of Himeno benchmark. In the temporal blocking kernel, it is observed that the usage of RAM blocks increases. Temporal blocking optimization also produces data dependency among the variables on the on-chip RAM due to a copy operation. Consequently, some iterations are executed serially and the performance is not fully optimized. To avoid the data dependency caused by a copy operation in the temporal blocking kernel, a shift-register kernel is implemented. The shift-register improves the efficiency of external memory accesses via the use of temporal locality. This implementation also decreases the RAM block usage and removes data dependency caused by the temporal blocking method. By combining the temporal blocking technique with a shift-register kernel, it is found that the implementation of the shift register effectively removes the memory dependency that is caused by the temporal blocking implementation in the memory-intensive application. Experimental results show that the peak performance of Himeno benchmark for the Stratix V FPGA is 10.62 GFlops, or 74% of the theoretical performance. Meanwhile, the peak performance for the Arria 10 A10PL4 FPGA is 13.95 GFlops, or 76% of the theoretical performance.

Chapter 4

Capability Assessment of a Multiple-FPGA System in General Matrix Multiplication (GEMM) Application using OpenCL

4.1 Introduction

In recent years, High-Performance Computing (HPC) infrastructures have been used by implementing the GPU as an accelerator for computation-intensive and data-intensive applications in government, academic, and industry organizations such as for high-energy physics, weather prediction, data mining, artificial intelligent, and bioinformatics. However, there are problems with regard to power consumption requirements. Since the total number of cores inside a GPU chip has increased drastically in recent models, the power required has grown significantly [3]. The energy-efficient of HPC is the latest trend; the crux of the matter is that computing power limits the performance of many applications such as data streams, big data analysis, and sensor networks [43].

Matrix multiplication has been widely implemented on GPUs, FPGAs, and sometimes both. Currently, matrix multiplication is widely used in machine learning applications specifically for training the algorithm. Previous studies showed the implementation of matrix multiplication using the NVidia Kepler architecture [44], Fermi GPU [45], Cypress GPU [46], and even many different types of GPUs and CPUs by using OpenCL [47]. Matrix multiplication algorithms have been evaluated on various FPGAs as well. For example, moderate success was achieved in the following studies. In [48], a 64-bit ANSI/IEEE Standard 754-1985 matrix multiplication was implemented on a Xilinx Virtex-II Pro FPGA. In [49], the Altera Stratix-V D5 FPGA was used to compute a sparse matrix multiplication. In [50], a hybrid approach on a PC cluster with Xilinx Virtex-5 and Stratix-III FPGAs was introduced using a Message Passing Interface (MPI). In [51], it is shown the implementation of matrix multiplication to accelerate the learning process of Conditional Restricted Boltzmann Machine (CRBM) with Intel FPGAs using OpenCL SDK for OpenCL.

This study evaluates the performance of an FPGA system by using OpenCL. OpenCL usage is motivated by its ease of use in contrast to HDL programming. A

matrix-to-matrix multiplication (GEMM) kernel is chosen as a case study to measure and analyze the computational capability of FPGAs for HPC purposes. The performance-power ratio of FPGA is also measured when executing the GEMM kernel. This performance-power ratio is useful, particularly when implementing an FPGA in an HPC application owing to many current applications that require FPGAs and take advantage of the benefits of FPGAs to achieve high performance with low power consumption.

4.2 General Matrix Multiplication (GEMM)

In this section, a matrix multiplication is introduced that is implemented on an FPGA by using Intel SDK for OpenCL. Two matrix multiplication algorithms are explored both for global memory and local memory implementation.

4.2.1 A brief overview of General Approach

This study focuses on the GEMM implementation, which is a matrix-to-matrix multiplication routine in Basic Linear Algebra Subprograms (BLAS). This was chosen as a good performance index in HPC applications owing to its high computational intensity and regularity [47]. In the experiment, matrices $A_{M,K}$ and $B_{K,N}$ were multiplied, where A was an M-by-K input matrix and B was a K-by-N input matrix. According to the BLAS libraries, general matrix multiplication performs the subroutine as shown in Equation 4.1:

$$C := \alpha * A * B + \beta * C, \tag{4.1}$$

where A and B are the input matrices, C is the output matrix, and α and β are floating-point constants. In this experiment, α and β are equal to 1.

4.2.2 Implementation of matrix multiplication by using Intel SDK for OpenCL

This section describes a matrix multiplication algorithm using OpenCL for the GPU and FPGA. The focus is on the code structure for the OpenCL kernels. There are many algorithms related to matrix multiplication implementation, including the naïve algorithm and the block algorithm. The naïve algorithm, as explained in Listing 4.1, utilizes global memory access to load and store the data. The global work item IDs for the x and y directions are indicated by *get_global_id* (0) and *get_global_id* (1). While the naïve algorithm is straightforward and has slow performance, the block algorithm can achieve better throughput owing to dividing a set of the matrix into several blocks during the execution process.

As explained in Listing 4.2, the block algorithm allows work items inside the same workgroup to calculate a smaller block of the matrix using local memory. The similar matrix multiplication algorithm using OpenCL can also be found in [52]. To implement the block algorithm in OpenCL kernel, the work item ID in local memory is required. This code allows the kernel to access the variable indicated by *get_local_id* (0) and *get_local_id* (1) in local memory instead. This technique also yields an increase in the ratio of data reuse in the multilevel memory hierarchy of the current processors [47].

Listing 4.1: Naïve GEMM using global memory.

```
#define BLOCK_SIZE 64
#define SIMD 4
#define CU 1

__kernel
__attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
__attribute((num_simd_work_items(SIMD)))
__attribute((num_compute_units(CU)))
void matrixMult_global(__global float *restrict A,
                       __global float *restrict B,
                       __global float *restrict C,
                       int A_width, int B_width)
{
    const int globalRow = get_global_id(0);
    const int globalCol = get_global_id(1);

    float Cvalue = 0.0f;
    for (int k=0; k<A_width; k++)
    {
         Cvalue += A[globalCol*A_width + k] * B[k*B_width + globalRow];
    }
    C[globalCol*A_width + globalRow] = Cvalue;
}
```

4.3 System Implementation

In order to perform the high-performance computation based on FPGA boards using Intel SDK for OpenCL, a system is constructed by combining a host PC and four FPGA boards as illustrated in Figure.4.1. The host communicates with the FPGA device through a PCIe bus. In the experiment, the performance of a single Stratix V DE5-Net FPGA board was investigated by measuring the performance of matrix multiplication on global memory and local memory. The next experiment was followed by employing multiple Stratix V DE5-Net FPGA boards to execute the computation kernels simultaneously.

4.3.1 Hardware architecture overview

In this study, the main components of the system are the host PC and Altera Stratix V DE5-Net FPGA boards. The host PC consists of an Intel Xeon E5-1650 CPU, 64 GB DDR4-2133, Centos 7.5 Linux 64-bit operating system, gcc version 4.8.5, Intel Quartus 16.1 64-bit, and Intel SDK for OpenCL 16.1.0 Build 196. For the FPGA devices, the Altera Stratix V DE5-Net FPGA boards are operated to run the computation process. The boards were developed by Terasic with specifications as follows: equipped with one Altera Stratix V GX FPGA (5SGXEA7N2F45C2), 2-GB DDR-1600 SDRAM, PCIe Gen3 slave edge connector, and 32-MB QDR II+ SRAM [31]. In more detail, the Altera Stratix 5SGXEA7N2F45C2 FPGA offers 622k logic elements (LEs), 50 Mbits of embedded memory, 48 transceivers (12.5 Gbps), 256 27-bit x 27-bit DSP blocks, and 2 PCIe hard IP blocks [53].

Listing 4.2: GEMM using local memory.

```
#define BLOCK_SIZE 64
#define SIMD 4
#define CU 1

__kernel
__attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1)))
__attribute((num_simd_work_items(SIMD)))
__attribute((num_compute_units(CU)))
void matrixMult_local(__global float *restrict A,
                      __global float *restrict B,
                      __global float *restrict C,
                      int A_width, int B_width)
{

   __local float localA[BLOCK_SIZE][BLOCK_SIZE];
   __local float localB[BLOCK_SIZE][BLOCK_SIZE];

   int gr_i = get_group_id(0);
   int gr_j = get_group_id(1);

   int i = get_local_id(0);
   int j = get_local_id(1);

   int num_blk = A_width/BLOCK_SIZE;

   int Astart = gr_j * A_width * BLOCK_SIZE;
   int Aincr = BLOCK_SIZE;
   int Bstart = gr_i * BLOCK_SIZE;
   int Bincr = BLOCK_SIZE * A_width;

   float Cvalue=0.0f;

   for (int blk = 0; blk<num_blk; blk++)
   {
         localA[j][i] = A[Astart+j*A_width+i];
         localB[j][i] = B[Bstart+j*A_width+i];

         barrier(CLK_LOCAL_MEM_FENCE);

         #pragma unroll
         for(int k=0; k<BLOCK_SIZE; k++)
         {
               Cvalue += localA[j][k] * localB[k][i];
         }

         barrier(CLK_LOCAL_MEM_FENCE);
         Astart += Aincr;
         Bstart += Bincr;
   }

    C[get_global_id(1)*get_global_size(0) + get_global_id(0)] = Cvalue;
}
```

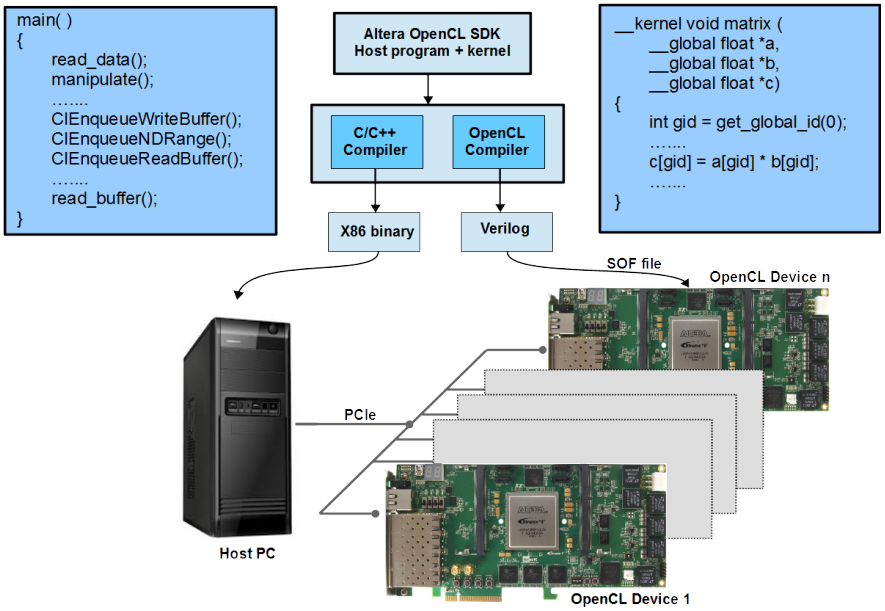

Figure 4.1: Experimental environment: Altera Stratix V DE5-Net FPGA boards and OpenCL programming.

4.3.2 PCIe throughput and DDR3 memory access

An Altera Stratix V DE5-Net FPGA board consists of two internal PCIe hard IP blocks. It is compliant with PCIe Base Gen 1.1, 2.0, or 3.0 for 1, 2, 4, and 8 lanes. Theoretically, the maximum data transfer bandwidths through PCIe are 2.5 GB/s for Gen 1.1, 5.0 GB/s for Gen 2.0, and 8.0 GB/s for Gen 3.0 [54]. However, in the OpenCL application, the Stratix V DE5-Net FPGA board supports the PCIe Gen2×8 lane only in its Board Support Package (BSP). The experiment showed that the PCIe throughput for writing and reading 8192 KB of data using Intel SDK for OpenCL was 2.154 GB/s and 2.903 GB/s.

The Stratix V DE5-Net FPGA board consists of two DDR3 memory banks. Theoretically, the peak performance is 25.6 GB/s (12.8 GB/s $*$ 2 Banks). Since GEMM is a typical memory bandwidth bottleneck, the total memory bandwidth between the Stratix V FPGA and the DDR3 memory is evaluated. To do so, a test was performed using an OpenCL memory stream kernel that transfers the data directly to and from the external DDR3 memory from and to the FPGA. The maximum memory bandwidth for each bank was 11.9 GB/s and 11.6 GB/s for Bank 1 and Bank 2, respectively.

4.4 Matrix multiplication using global memory

The kernel presented in Listing 4.1 is executed on the global memory of the FPGA board. Matrix A is accessed in row-major order using $A\,[y * width + k]$, while matrix B is accessed in column-major order using $B\,[k * width + x]$, where x and y are defined as $get\_global\_id\,(0)$ and $get\_global\_id\,(1)$, respectively. In the experiment, first, the standard compilation was performed in burst-interleaved mode. Second, the kernel was compiled by employing the compilation option to disable the memory cache implementation in global memory access.

The basic matrix multiplication kernel, which is considered as a single computation unit, comprises two operators (one multiplier and one adder) that perform the matrix

Table 4.1: OpenCL kernel compilation reports for global memory access with different SIMD size.

	Logic	RAM blocks	DSP blocks	Clock (MHz)	Estimation (GFlops)
1 SIMD, 1 CU	45,098 (19%)	437 (17%)	5 (2%)	271.01	0.542
4 SIMD, 1 CU	48,632 (21%)	499 (18%)	8 (3%)	273.29	2.186
8 SIMD, 1 CU	52,111 (22%)	464 (18%)	12 (5%)	279.79	4.476
16 SIMD, 1 CU	58,033 (25%)	521 (20%)	20 (8%)	276.16	8.837

multiplication process as described in Listing 4.1. We propose an equation to estimate the performance of matrix multiplication with Stratix V FPGA, as follow:

$$P_{est} = \begin{cases} 2\,[Flop] \times n \times k\text{-}way \times f, & \left(P_{est} < \frac{BW}{4[Bytes/Flop]}\right) \\ \\ \frac{BW}{4[Bytes/Flop]}, & (otherwise) \end{cases} \tag{4.2}$$

where, the performance estimation is P in Flops, n is the number of compute units (CU) attribute, the k-way SIMD performs k vectorization, BW is the maximum memory bandwidth, and f is the actual frequency of the FPGA in Hz. Equation 4.2 shows the performance estimation for single-precision point since the value of [Bytes/Flop] is a constant value.

4.4.1 Performance with cache

The compilation reports for the Naive kernel implementation using burst-interleave mode are presented in Table 4.1 and Table 4.2 for different numbers of SIMD attributes and CU attributes. The experimental results for different numbers of SIMD attributes are shown in Figure.4.2. The peak performances are 0.542 GFlops, 2.179 GFlops, 4.455 GFlops, and 8.668 GFlops for 1 SIMD, 4 SIMD, 8 SIMD, and 16 SIMD, respectively, in a single-precision data type. This result is similar to the performance estimation, as shown in Table 4.1. For different numbers of CU attributes, the peak performances are 0.542 GFlops, 2.236 GFlops, 3.715 GFlops, and 7.511 GFlops for 1 CU, 4 CU, 8 CU, and 16 CU in a single-precision data type, as shown in Figure.4.3. These performances are also similar to the performance estimation, as presented in Table 4.2.

Interestingly, the number of DSP blocks is not proportional to the number of SIMD

Table 4.2: OpenCL kernel compilation reports for global memory access with different CU size.

	Logic	RAM blocks	DSP blocks	Clock (MHz)	Estimation (GFlops)
1 CU, 1 SIMD	45,098 (19%)	437 (17%)	5 (2%)	271.01	0.542
4 CU, 1 SIMD	61,238 (26%)	716 (28%)	20 (8%)	281.52	2.252
8 CU, 1 SIMD	83,791 (36%)	1088 (43%)	40 (16%)	268.88	4.302
16 CU, 1 SIMD	128,768 (55%)	1832 (72%)	80 (31%)	268.24	8.583

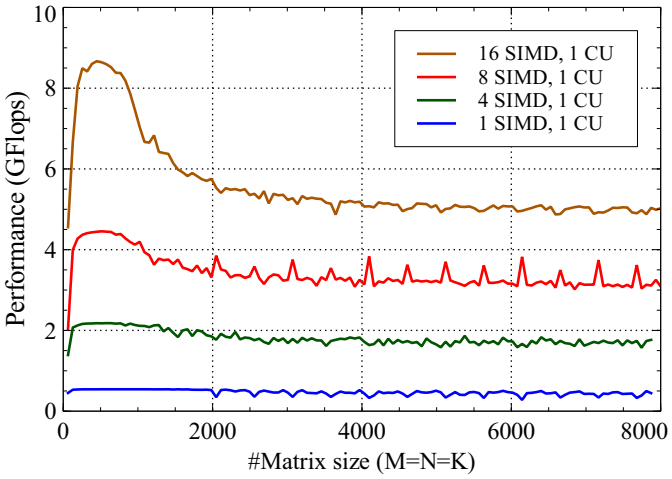

Figure 4.2: Performance of matrix multiplication using different SIMD number.

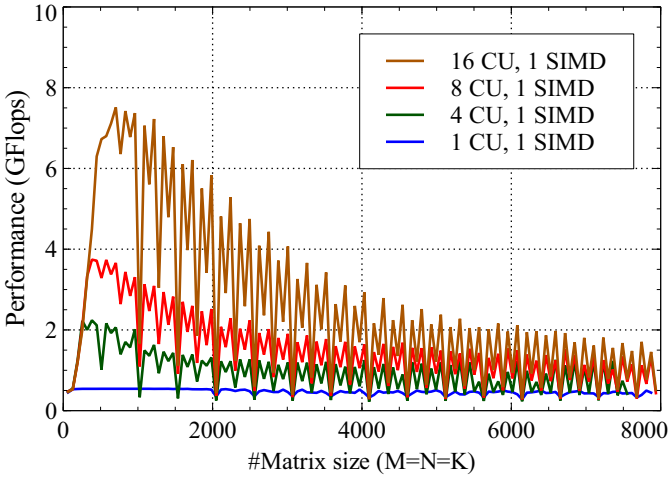

Figure 4.3: Performance of matrix multiplication using different CU number.

attributes. In Table 4.1, the kernel with the 1CU+1SIMD attribute comprises 5 DSP blocks, while the kernel with the 1CU+16SIMD attribute requires 20 DSP blocks. According to the actual mapping of the compiled AOCL project by utilizing the RTL viewer, there are two main component instances inside high-level modules of the kernel. Because of the kernel vectorization, the Intel SDK for OpenCL compiler duplicates the data path only within the compute unit. As a result, the first instance consists of the same number of DSPs regardless of the number of SIMD attributes. In this case, 2 DSPs are required inside the first instance for both kernels. For a kernel with the 1CU+1SIMD attribute, 3 DSPs are required for the computation of the second instance (the total is 5 DSPs). Kernels with the 1CU+4SIMD and 1CU+8SIMD attributes consist of 6 DSPs and 10 DSPs, respectively, for the computation in the second instance (the total is 8 DSPs and 12 DSPs). Last, the kernel with the 1CU+16SIMD attribute consists of 18 DSPs for the computation process (the total is 20 DSPs).

From Table 4.2, increasing the number of CUs replicates the number of kernels in the FPGA. This effects a significant increase in hardware resource utilization such as in the logic, RAM blocks, memory usage, and particularly in the number of DSP blocks. The increase in the number of DSP blocks is proportional to the number of CU

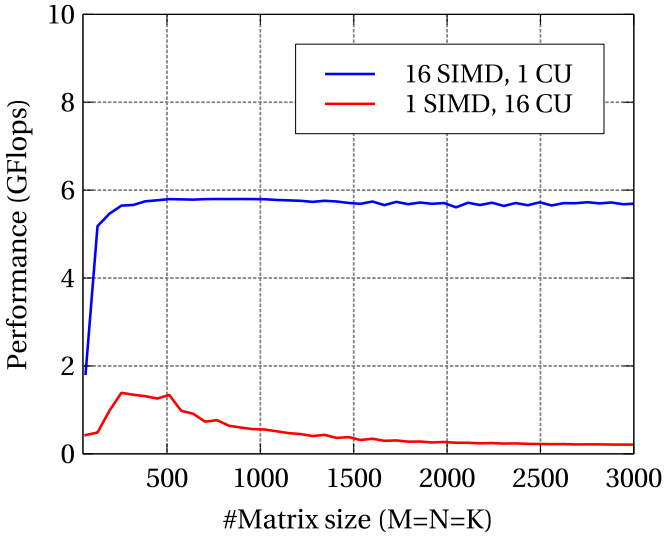

Figure 4.4: Performance of matrix multiplication on global memory without burst-coalesced cached LSUs.

attributes. The number of DSP blocks is 5 DSP blocks for a single CU attribute, 20 DSP blocks for 4 CUs, 40 DSP blocks for 8 CUs, and 80 DSP blocks for 16 CUs.

From the experimental results, both the SIMD attribute and CU attribute increase the peak performance. However, implementing the SIMD attribute achieves a higher performance than the CU attribute. The SIMD attribute generates more efficient hardware by duplicating the data path only and by coalescing the memory accesses. Meanwhile, the CU attribute modifies the number of compute units by replicating the kernel so that it increases the number of times for the kernel to access the global memory as well. This affects the undesired memory access patterns, as shown in Figure.4.3.

4.4.2 Performance without cache implementation

Naïve matrix multiplication is bounded by the DDR3 memory bandwidth of the FPGA. The theoretical peak performance of matrix multiplication for the $2 \times M \times N \times K$ flops of computations can be calculated based on the maximum memory bandwidth [55]. According to section 4.3, the total memory bandwidth for both Bank 1 and Bank 2 is 23.5 GB/s. As a result, the peak performance is limited to 5.9 GFlops.

The peak performances in Figure.4.2 and Figure.4.3 were investigated using the Intel dynamic profiler for OpenCL. For the 16SIMD+1CU attribute, it showed 99.2% cache hits, coalesced memory access, and 100% of memory bandwidth efficiency. For the 16CU+1SIMD attribute, the dynamic profiler yielded 97.9% cache hits, unaligned memory access, and 16.7% of memory bandwidth efficiency. This evidence indicates that the default compilation process still employs the memory cache. The high rate of cache hits (99.2% and 97.9%) increases the peak performance to higher than 5.9 GFlops.

In Intel SDK for OpenCL, a burst-coalesced cached LSU is created when the compiler assumes that the memory access pattern is data-dependent or appears to be repetitive [20]. In this study, matrix multiplication requires the repetitive memory access. Therefore, there is a possibility that a burst-coalesced cached LSU is created. To disable this memory cache, a compilation for the kernel was conducted by making the pointer as "volatile". This implementation created non-cached LSUs. The peak perfor-

Table 4.3: OpenCL kernel compilation reports for global memory access without burst-coalesced cached LSUs.

	Logic	RAM blocks	Memory (Mbits)	DSP blocks	Clock (MHz)
16 SIMD, 1 CU	55,047 (23%)	417 (16%)	2.65 (5%)	20 (8%)	271.88
1 SIMD, 16 CU	127,082 (54%)	943 (37%)	7.725 (15%)	80 (31%)	269.02

mances of global memory without burst-coalesced cached LSU are shown in Figure.4.4. This shows that the peak performance decreased to 5.78 GFlops and 1.51 GFlops for the 16SIMD+1CU attribute and 16CU+1SIMD attribute.

Again, the Intel dynamic profiler was employed to investigate the performance of matrix multiplication in global memory without burst-coalesced cached LSUs. For the 16SIMD+1CU attribute, it showed 0% cache hits, coalesced memory access, and 100% of memory bandwidth efficiency. For the 16CU+1SIMD attribute, the Intel dynamic profiler produced the same 0% of cache hits but different memory access and memory bandwidth. The low performance of the 16CU+1SIMD attribute resulted from unaligned memory access and 10.9% of memory bandwidth efficiency. The compilation reports for both the 16SIMD+1CU attribute and 16CU+1SIMD attribute without burst-coalesced cached LSU are presented in Table 4.3.

4.5 Matrix multiplication using local memory

In this section, the computation kernel using local memory is composed of a single computation unit (CU) from this viewpoint of memory access. In Algorithm 4.2, the CU is much more complicated than the kernels in the previous section because the computational circuits, namely, spatial parallelism, are different. On top of that, some optimizations such as loop unrolling, kernel vectorization, and memory coalescing are implemented. Then, the SIMD parallelism was tested by various parameters to achieve a higher performance based on one CU implementation. In our experiments, the limitation was the number of DSP blocks. Therefore, four-SIMD and one-SIMD implementations were chosen for the computation kernels on single-precision and double-precision floating point, respectively.

In the Stratix V FPGA, a single-precision floating-point multiplier is implemented by using logic, a DSP, and registers, while a single-precision floating point adder is constructed with logic and registers. A double-precision floating-point multiplier is implemented by using four 27-bit DSPs because the multiplication of significands requires wider multipliers [56]. To calculate the performance estimation in the computation kernel where each CU has a Fused Multiply-Add (FMA) with one multiplier and one adder, and by assuming that almost all DSPs and adders work in parallel, the value of $n \times k-$way in Equation 5.2 is equal to the number of multiplications that is obtained from the number of DSP blocks for a single-precision data type. For double precision, the number of $n \times k-$way is almost equal to the number of $DSPs/4$. In this kernel implementation, the communication of global memory is ignored.

In the experiment, matrix A was accessed from global memory to local memory in row-major order, while matrix B was accessed in column-major order, as shown in Listing 4.2. In this study, the size of the local memory was $A_{local}\,[64 \times 64]$ and $B_{local}\,[64 \times 64]$. The obtained parameters and the performance estimation are shown

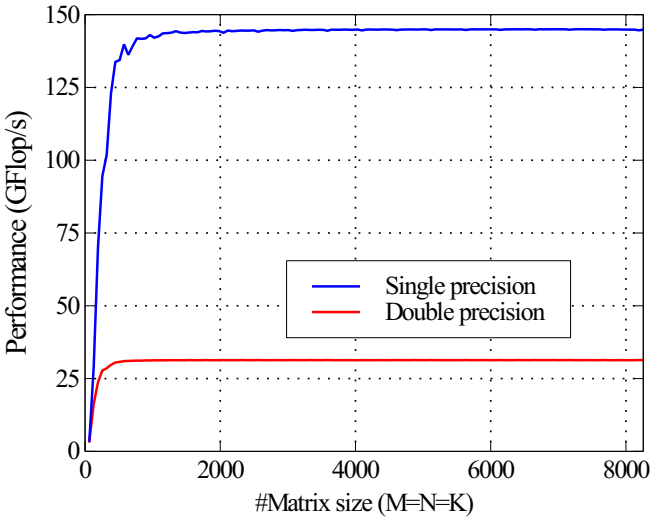

Figure 4.5: Performance of matrix multiplication using local memory.

Table 4.4: OpenCL kernel compilation reports for local memory access.

	Logic	RAM blocks	DSP blocks	Clock (MHz)	Estimation (GFlops)
Single precision	120,853 (51%)	936 (37%)	256 (100%)	283.4	145.10
Double precision	90,212 (38%)	1,085 (42%)	256 (100%)	245.1	31.37

in Table 4.4. As illustrated in Figure.4.5, the performance of matrix multiplication in local memory increases drastically to 144.95 GFlops for single precision. This real performance is similar to the 145.1 GFlops of the performance estimation. For the double-precision data type, the peak performance is 31.3 GFlops, while the performance estimation is 31.37 GFlops.

4.5.1 Performance of multiple Stratix V DE5-Net FPGA boards

To test the performance of multiple FPGAs, the experiment was conducted using four FPGA boards. Unlike a single FPGA, multiple FPGAs perform matrix multiplication by executing the data that is distributed by the host PC to each FPGA board. In this case, the data sent by the host is distributed to the four FPGA boards equally. In the following example, the size of the matrix was $[4352 \times 4352]$ for both matrix A and matrix B. Before the host invoked each FPGA to execute the kernel, the data was distributed to each global memory of the FPGAs, as shown in Figure.4.6. The host dispatched the matrix $A\,[1088 \times 4352]$ and the matrix $B\,[4352 \times 4352]$ to each FPGA board.

In the experiments, the kernel execution time indicated by the red dotted line was measured as shown in Figure.4.7. After invoking the $clEnque\ NDRangeKernel()$ command to start the multiplication process, the host waits for all FPGAs to send their kernel events. These events inform the host that the kernels have been executed successfully. Because the execution time is different for each FPGA, the host waits for all kernels to finish. The total execution time is measured after executing the $clWaitForEvents()$ command. This procedure is followed by reading the buffer using the $clEnqueReadBuffer()$ command to read the calculation results. In this process,

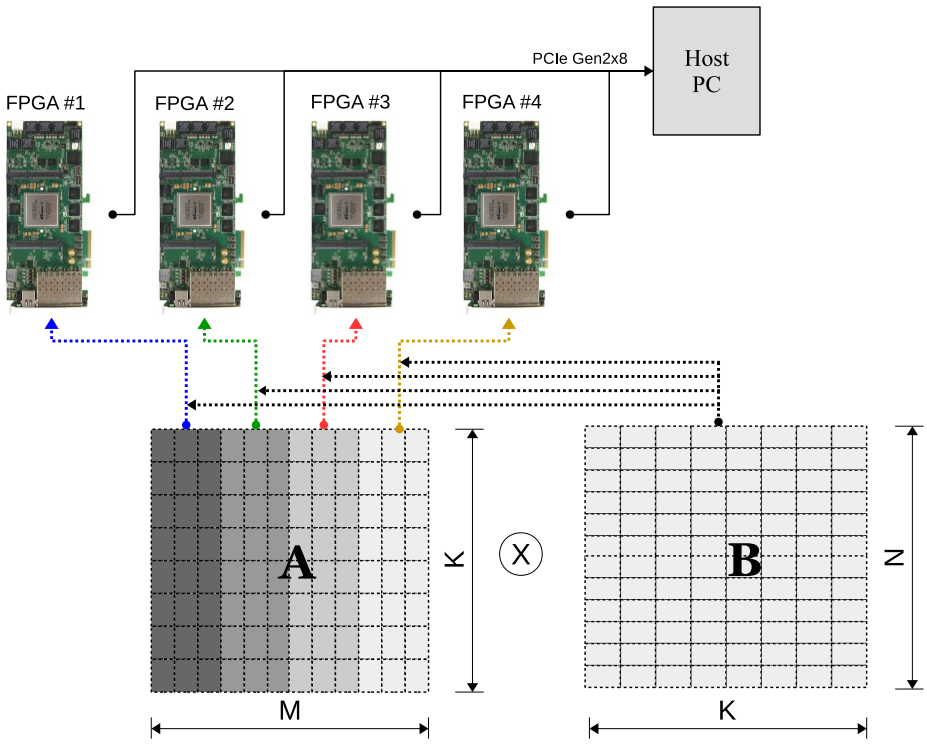

Figure 4.6: Multiple FPGAs approach.

the data transfers from each FPGA memory to host memory are read sequentially. First, the host reads the buffer in FPGA #1. This is followed by FPGA #2, FPGA #3, and finally FPGA #4.

The peak performance for different numbers of FPGA boards is shown in Figure.4.8 and Figure.4.9. These graphs show an increase in performance by employing the FPGA from one board to four boards simultaneously. For the single-precision data type, the performance was measured by executing the kernel in global memory and local memory. The kernel with a configuration of 16 SIMDs and 1 CU was chosen owing to the high peak performance for the global memory access. The performance increases proportionally from 5.795 GFlops to 11.565 GFlops, 17.351 GFlops, and 23.117 GFlops for a single FPGA until four FPGA boards are used, as shown in Figure.4.8.

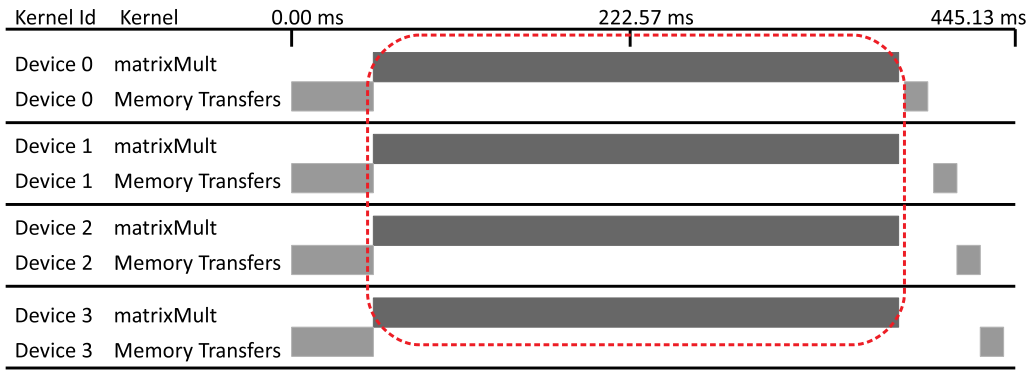

Figure 4.7: Kernel execution in multiple-FPGAs implementation using Intel dynamic profiler for OpenCL.

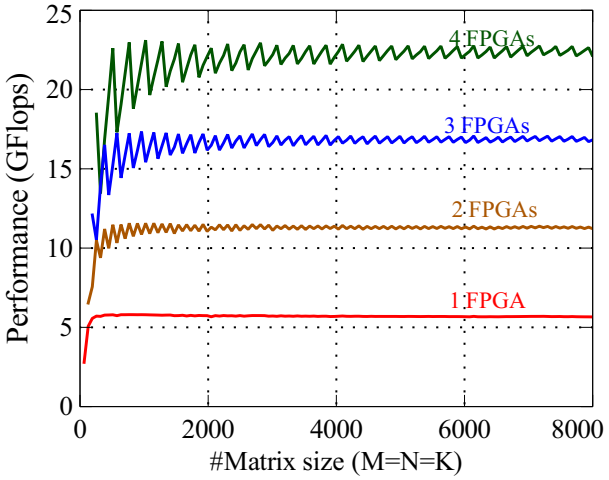

Figure 4.8: Performance of matrix multiplication (16 SIMD, 1 CU) using global memory in single precision.

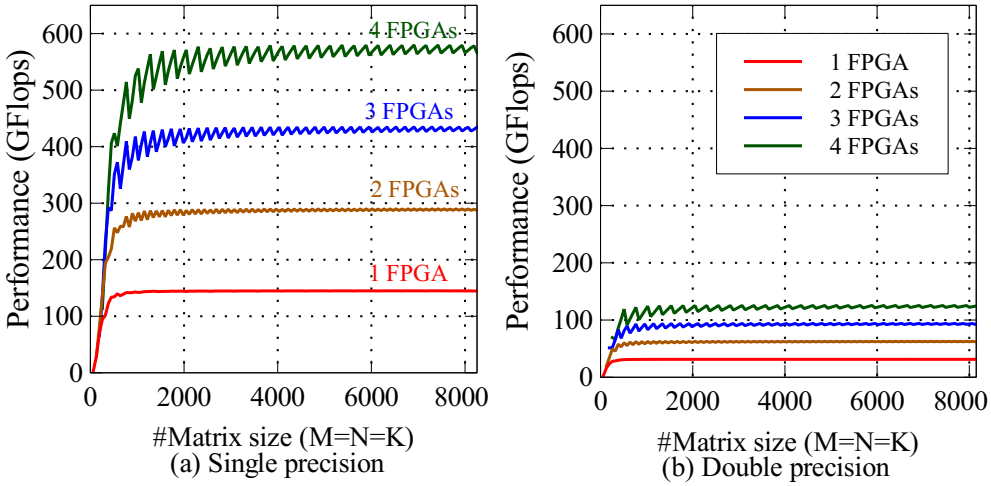

Figure 4.9: Performance of matrix multiplication using local memory.

For local memory access, the performance improves drastically from 144.95 GFlops to 289.61 GFlops, 434.78 GFlops, and 579.41 GFlops from one FPGA to four FPGAs, as illustrated in Figure.4.9(a). For the double-precision data type, it can be observed that the performance increases proportionally from 31.31 GFlops to 62.72 GFlops, 94.08 GFlops, and 125.44 GFlops, as shown in Figure.4.9(b). Even though Stratix V FPGA supports PCIe Gen3x8, this FPGA only supports PCIe Gen2x8 for the OpenCL implementation. Therefore, the kernel execution time was measured without taking into account the data transfer from the host to each FPGA memory.

4.5.2 Performance per power efficiency

In the experiment, the total power including the host PC and FPGA was measured. The HIOKI 3332 Power HiTester was employed to measure the total input power. When testing the system, the voltage was 104 V as measured by the HIOKI 3332 Power HiTester simultaneously. The total power for the host PC and a single Stratix V DE5-Net FPGA required power 103.6 W. For four Stratix V FPGA boards required a

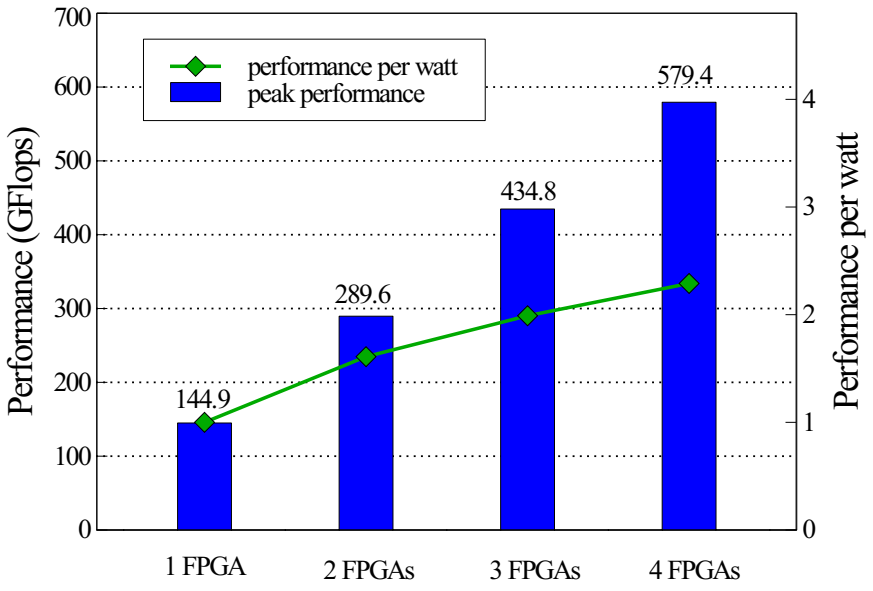

Figure 4.10: Peak performance, and performance per watt of FPGAs relative to single FPGA for single precision data type.

maximum of 180.5 W. The performance per watt of multiple FPGAs relative to single FPGA can be seen in Figure.4.10. The figure shows that the performance efficiency increases to 1.61x, 1.99x, and 2.29x for two, three and four FPGAs respectively.

4.6 Conclusions

The implementation of the GEMM in the global memory and local memory was demonstrated using multiple Stratix V FPGAs. For global memory usage, the peak performance achieved 23.117 GFlops for four FPGA boards. For the local memory implementation, a peak performance of 579.41 GFlops was achieved in a single-precision data type. Meanwhile, for the double-precision data type, the system achieved 125.44 GFlops for four Stratix V DE5-Net FPGAs. In this study, the equation for estimating the performance was introduced. The results show that the peak performances were similar to the performance estimation which is calculated from the OpenCL kernel compilation report. In term of performance per power ratio, the performance efficiency relative to single FPGA increases to 1.61x, 1.99x, and 2.29x for two, three and four Stratix V DE5-Net FPGAs respectively. These results show that the performance of multiple FPGAs increases linearly, also the multiple FPGA implementations consume less power.

Chapter 5

OpenCL Implementation of FPGA-based Signal Generation and Measurement

5.1 Introduction

FPGA has been well known for its high bandwidth capability to access the general purpose I/O with low latency. As a result, FPGA is used in many engineering fields and applications. For example, FPGA implementations can be found in digital signal processing [57], data acquisition [58], communication [59], software-defined radio [60], automotive radar [61], and quantum computing [62]. To program an FPGA, a hardware description language (HDL) is used to generate hardware implementation from the source code onto a register transfer level (RTL). However, FPGA programming using HDL becomes time-consuming due to an increase in design complexity and an increase in FPGA resources. Currently, high-level synthesis (HLS) is implemented on FPGAs as an alternative solution to reduce the development time for FPGA programming. HLS improves the FPGA design efficiency by increasing the abstraction level of the code [12]. HLS also reduces the gap between the FPGA design and the programming process. Consequently, the FPGA development time can be reduced.

This study focuses on the implementation of FPGAs for signal generation and measurement. Therefore, an FPGA needs to access external devices such as an analog-to-digital converter (ADC) and a digital-to-analog converter (DAC) for measuring and generating a signal. To reduce the development time, OpenCL was selected to program the FPGA because OpenCL exploits the concept of parallelism that enables us to develop a parallel program application for FPGAs using high-level language. In addition to this, OpenCL avoids creating complex HDL codes, particularly for the libraries as well as platform-specific tools [63]. However, compared to the HDL program, OpenCL does not provide direct access to the FPGA's I/O, particularly for reading data from an ADC and writing data to a DAC.

To overcome these limitations, ADC and DAC component modules are developed on the *system.qsys* of the FPGA's board support package (BSP), which allows an OpenCL kernel to access the FPGA's I/O. To enable the kernel to communicate with the ADC and DAC components, an OpenCL I/O channel extension was employed. This channel extension allowed the OpenCL kernel to stream data to and from the FPGA's I/O. Therefore, this study demonstrates the capability of the OpenCL program for accessing the FPGA's I/O directly, particularly for signal measurement and generation

applications. It is expected that this research will contribute to the use of the OpenCL program not only for FPGA-based parallel computations, but also for signal and video processing, data acquisition, and control systems through the FPGA's I/O.

The following are several advantages of using OpenCL implementation for the FPGA-based signal generation and measurement compared to using HDL-based design. In HDL implementation, to generate a signal, a ROM-based lookup-table is required to store data to generate a signal. The size of data is also limited by the size of the FPGA's ROM. In some cases, the FPGA needs to be reprogrammed when different signals need to be modified. However, for an OpenCL implementation, the signal data can be stored on global memory (external DDR memory) instead of the FPGA's ROM because the OpenCL framework provides an interfaces and access to the global memory. In this implementation, large data can be stored on global memory where this data is limited by the size of the external memory. Different signals can also be updated quickly without reprogramming the FPGA by invoking the host to transmit the data to the FPGA's global memory. Similarly, the measured signal can also be stored on global memory. Consequently, this allows the host to read the data from the FPGA directly for further processing and analysis. In HDL-based design, these implementations require detailed specifications of the double data rate (DDR) interface for the configuration for the DDR memory controller to access external memory. Moreover, the FPGA simulation and debugging process needs to be performed. To enable communication or to transmit and receive the data between the FPGA and the host, hardware configuration and a device driver for a PCIe hard IP or a 10 Gbps Ethernet controller is required in the HDL-based design. However, for an OpenCL implementation, the PCIe and Ethernet controller are generated automatically. This is because the OpenCL framework consists of firmware, software and device driver between FPGA and the host for connecting, controlling and transferring data [64]. In term of development time, OpenCL implementation takes two weeks of programming the FPGA, particularly for signal measurement and generation, where the most considerable portion involves developing the ADC and DAC component modules using Avalon-ST source and Avalon-ST sink on the FPGA's BSP. Thus, OpenCL implementation reduces the development time and increases productivity.

This chapter presents OpenCL kernel implementation for signal measurement and generation. Experiments are conducted by developing the kernel into three categories to evaluate the OpenCL kernel implementation, as follow:

- *signal measurement*: The objective of this kernel is to measure the input signal and store the data in global memory of the FPGA. This allows the host PC to analyze the data for further analysis.
- *signal generation*: OpenCL kernel generates an output signal by reading the data from global memory.
- *signal measurement and generation*: In this implementation, the first OpenCL kernel reads a signal and writes the received signal to an I/O channel. The second OpenCL kernel reads the data from the I/O channel and generates a signal simultaneously where the data transfer is performed without accessing global memory.

5.2 OpenCL for SoC FPGA

In OpenCL programming of a SoC FPGA, the kernel code for the FPGA is also compiled into a different sequence of instructions that are executed by different work

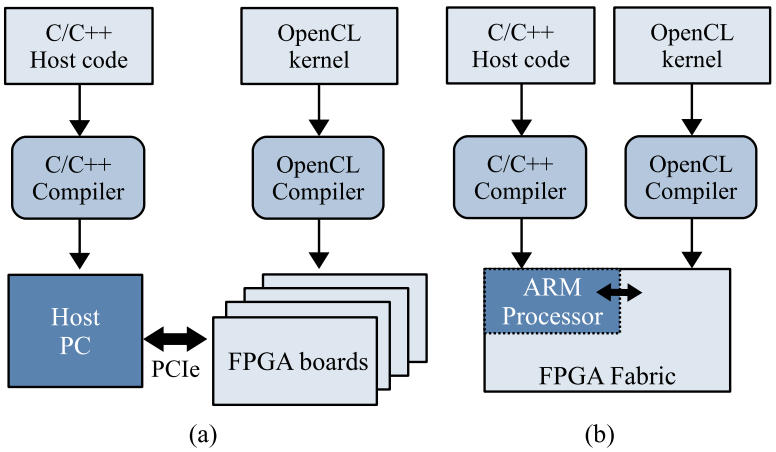

Figure 5.1: OpenCL system with a host CPU and FPGAs (a) data communication through a PCIe, (b) using the internal bus.

items simultaneously. This is because an FPGA exploits pipeline parallelism to execute the kernel. Compared to FPGA for high-performance computing purposes that allows the host to communicate with the FPGA through a PCIe bus, OpenCL for SoC FPGA also provides an application program interface (API) to communicate through an internal bus that is used for data transfer and communication between an FPGA and the advanced RISC machine (ARM) processors, as shown in Fig.5.1.

OpenCL for SoC FPGAs also shares the same memory types for the computation process. The memory types are defined as global/constant memory, local memory, and private memory. The offline compiler for OpenCL employs DDR3 memory on an FPGA board as global memory. The memory type that has higher throughput with lower latency is local memory. During the kernel compilation, local memory is implemented by the block RAMs. This memory is dedicated to work items in the same workgroup. The last type of memory that has faster throughput and smaller size than the others is the private memory. Depending on data size, private memory is implemented by either block RAMs or registers [65].

5.3 Customizing the board hardware for OpenCL components

To allow the OpenCL kernel to access the FPGA's I/O directly, the procedures are divided into three steps as shown in Fig.5.2. First, the ADC and DAC component modules are developed on the FPGA's board support package (BSP) using Qsys system design. Qsys, or currently known as the Platform Designer, is a system-integration tool that improves productivity by automatically generating interconnect logic to connect intellectual property (IP) and subsystems for Intel FPGA. Second, the ADC and DAC component attributes are defined in the channel interface of the *board_spec.xml* file that describes the hardware interfaces to the Intel FPGA SDK for OpenCL. Third, OpenCL I/O channel extension is used for streaming data to and from an FPGA's I/O through the ADC and DAC components.

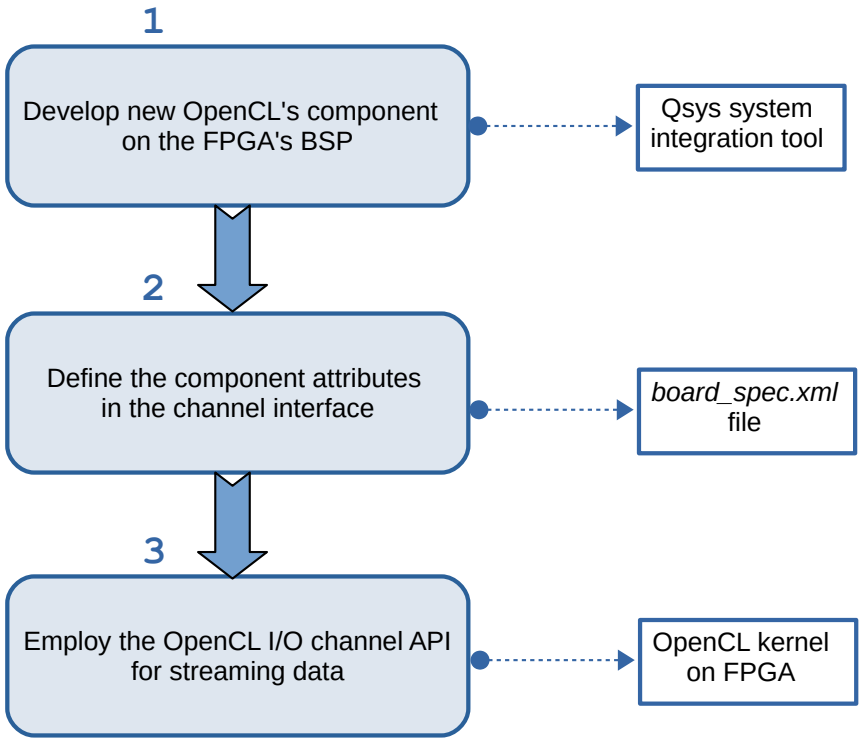

Figure 5.2: A method to develop the new OpenCL component modules.

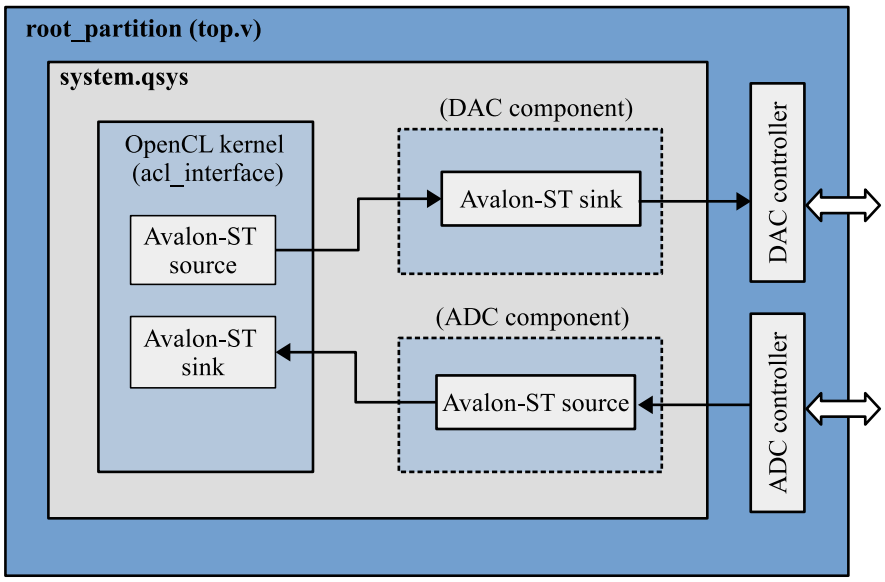

Figure 5.3: System Qsys of a customized board support package (BSP) by adding new ADC and DAC components.

5.3.1 Developing OpenCL's ADC and DAC components

In Fig.5.3, it is shown a *system.qsys* diagram of the FPGA's BSP where the OpenCL ADC and DAC components have been developed. The BSP, which is provided by the FPGA vendor for OpenCL programming, simplifies FPGA programming because it provides an external DDR memory controller for writing and reading data to global memory, and provides a data interface for communications between the host and FPGA.

The first component is the ADC component. This component is created by implementing an Avalon-ST source for streaming data from the external ADC board to the OpenCL kernel. The OpenCL kernel receives the data from this component through an Avalon-ST sink. The block diagram of the ADC component is shown in Fig.5.4(a). The ADC component receives two input signals, the clock (*kernel_clk*) and the reset (*kernel_rst*) from the kernel, has a port for reading the data from the ADC board (*adc_read*), produces two signals for handshaking with the kernel: (*kernel_ready*) and (*kernel_valid*), and includes a port for streaming the data to the kernel (*kernel_out*).

The second component is the DAC component, where it is created by implementing an Avalon-ST sink for streaming data from the OpenCL kernel to the external DAC board. The OpenCL kernel streams the data through an Avalon-ST source to the DAC component. As shown in Fig.5.4(b), this component also has the same signals as the ADC; however, the two signals for handshaking, (*kernel_ready*) and (*kernel_valid*), are in the opposite directions. This component also has a port for receiving streamed data from the kernel (*kernel_in*) and a port for writing data to the external DAC board (*data_out*).

In the Qsys system design, the data ports of the ADC and DAC components need to be exported so that the kernel can read from and write data to the external ADC/DAC board. The *adc_read* port of the ADC component is exported and connected to the data port of the ADC controller, while the *dac_write* port of the DAC component is exported and connected to the data port of the DAC controller. The ADC and DAC controllers are written in HDL and are located inside the root partition (*top.v*) of the BSP to control the ADC/DAC board. The ADC chip on the ADC/DAC board converts an analog signal to a 14-bit digital signal. In contrast, the DAC chip converts a 14-bit digital signal to an analog signal. The system Qsys including OpenCL's ADC and DAC components can be seen in Fig.5.5

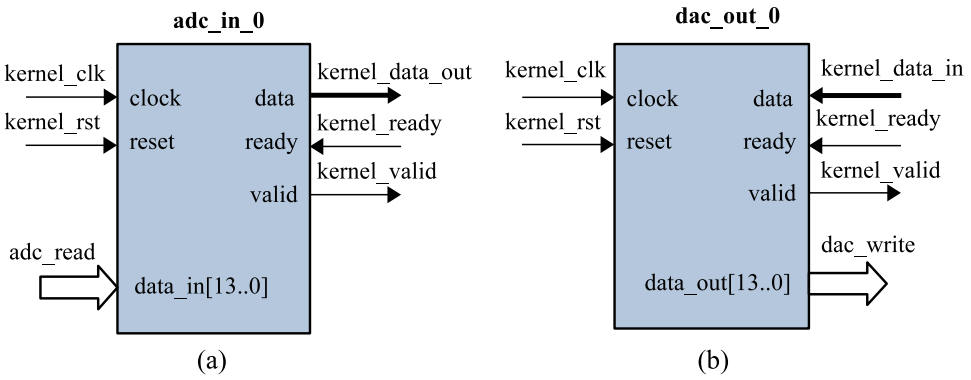

Figure 5.4: (a) ADC component using Avalon-ST source, (b) DAC component using Avalon-ST sink.

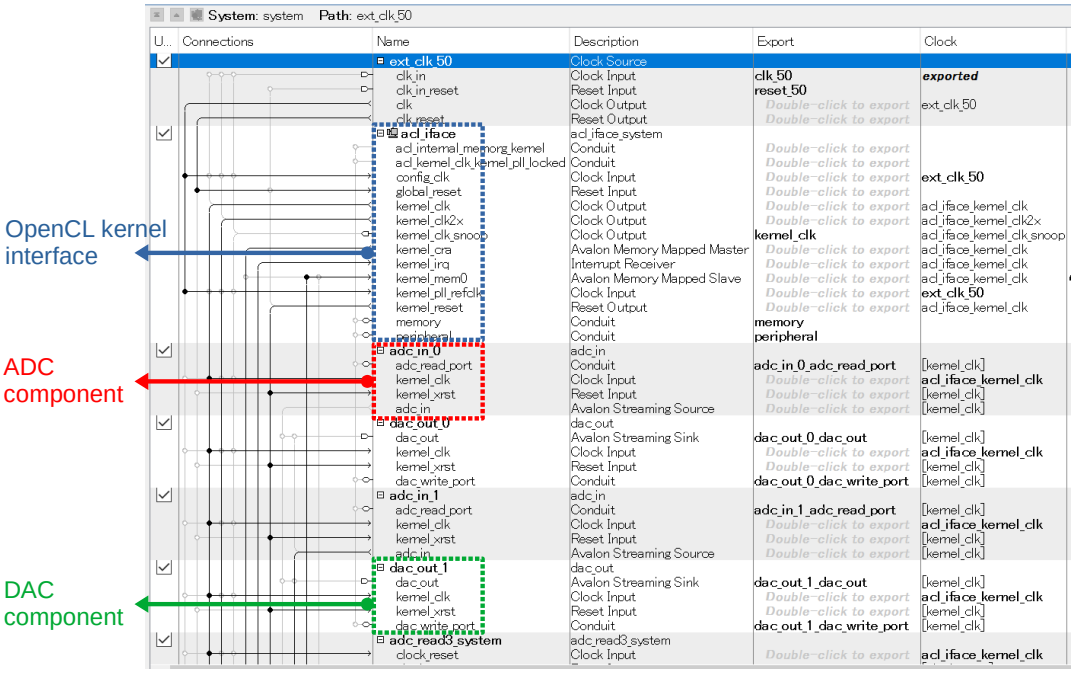

Figure 5.5: System Qsys of a customized board support package (BSP) by adding new ADC and DAC components.

Table 5.1: OpenCL component attributes.

Component attributes	ADC component	DAC component
name	adc_in_0	dac_out_0
port	adc_read	dac_write
type	streamsource	streamsink
width	14	14
chan_id	ch_adc_read	ch_dac_write

5.3.2 Setting OpenCL component parameters

To allow the OpenCL kernel to access these components, the component attributes need to be declared in the *board_spec.xml* file of the BSP. The *board_spec.xml* file is an extensible markup language (XML) file that provides the board description, such as the hardware interface and the component interface, to the Intel SDK for OpenCL. According to the content of the *board_spec.xml* file, a custom circuit for an FPGA is generated by the SDK compiler for OpenCL. Then, this custom circuit is incorporated with the OpenCL kernel [66].

According to the ADC and DAC components, as shown in Fig.5.4(a) and Fig.5.4(b), the component attributes such as *name*, *port*, *type*, *width*, and *chan_id* on the channel interface of the *board_spec.xml* file are specified, as shown in Table.5.1. The *name* attribute specifies the names of the ADC and DAC components. The *port* attribute specifies the data ports of the ADC and DAC components where data are read from and written to the FPGA's I/O. The *type* attribute specifies the type of Avalon-ST bus being used, as shown in Listing 5.1. Because the ADC component reads data from the

Listing 5.1: The content of board_spec.xml file of FPGA's Board Support Package (BSP).

```
<?xml version="1.0"?>
<board version=" " name=" ">
  ........
  ........
  <interfaces>
      <interface name="acl_iface" port="kernel_cra" type="master" width="64" misc="0"/>
      <interface name="acl_iface" port="kernel_irq" type="irq" width="1"/>
      <kernel_clk_reset clk="acl_iface.kernel_clk" clk2x="acl_iface.kernel_clk2x"
          reset="acl_iface.kernel_reset"/>
  </interfaces>

  <channels>
      <interface name="adc_in_0"  port="adc_in"   type="streamsource" width="14"
          chan_id="ch_data_adc_read"/>
      <interface name="dac_out_0" port="dac_out"  type="streamsink" width="14"
          chan_id="ch_data_dac_write"/>
  </channels>

</board>
```

ADC board and streams this data to the OpenCL kernel, a stream source is employed. On the other hand, the DAC component receives streamed data from the OpenCL kernel and writes the data to the DAC board; therefore, a stream sink is used. The *chan_id* attribute is a unique name for the I/O interface on the FPGA board and will be associated to the *io*("*chan_id*") attribute in the OpenCL kernel. The value of 14 in the *width* attribute specifies the 14-bit resolution of the ADC and DAC board. In the experiments, the data type for this channel is specified as *ushort*.

5.3.3 Accessing OpenCL's ADC and DAC components using an I/O channel extension

In the Intel SDK for OpenCL, a $write\_channel\_intel(ch\_0, input\_buf)$ API call is used to write data to the $input\_buf$ variable of a channel ch_0. To read data from the ch_1 channel to an $output\_buf$ variable, a $output\_buf = read\_channel\_intel(ch\_1)$ API call is used, as shown in Listing 5.2. Previous study has shown the implementation of an OpenCL channel extension for data communication. In [67], an implementation of the OpenCL I/O channel extension for data communication using a high-speed FPGA network through the QSFP+ port was demonstrated.

In this study, the OpenCL I/O channel extension is employed to stream data between the OpenCL kernel and the FPGA's I/O through the ADC and DAC components. To read a signal from the ADC board, the channel attribute in the kernel must point to the *chan_id* name of the ADC component. Here, the *chan_id* attribute is specified as *ch_adc_read*. Therefore, the channel attribute in the kernel is declared as *io*("*ch_adc_read*"). A similar method is applied to write data to the DAC board. However, the channel attribute in the kernel is declared as *io*("*ch_dac_write*") so that it points to the DAC component.

Listing 5.2: Writing and reading a data using OpenCL channel extension.

```
1        .....
2            // to write the data in input_buf variable to a channel ch_0
3            write_channel_intel(ch_0, input_buf);
4        .....
5
6        .....
7            // to read the data from a channel ch_1 to output_buf variable
8            output_buf = read_channel_intel(ch_1);
9        .....
```

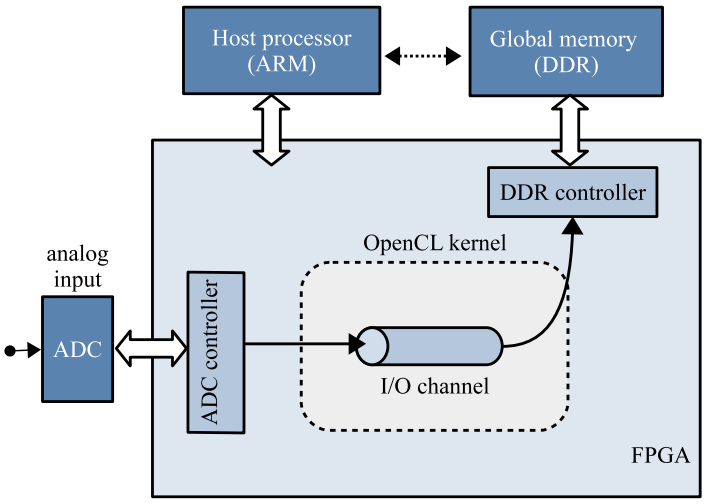

Figure 5.6: I/O channel implementation for signal measurement from ADC to global memory of an FPGA.

5.4 System implementation

This study evaluates the OpenCL kernel using a Cyclone V SoC FPGA board from Terasic. This board consists of the Cyclone V SoC 5CSEMA5F31C6 FPGA, dual-core ARM Cortex-A9 (HPS), 85K programmable logic elements, 4,450 Kbits embedded memory, two 40-pin expansion headers, and two hard memory controllers [68]. For the ADC and DAC chip, we utilized the analog-to-digital/digital-to-analog (AD/DA) board from Terasic. This AD/DA board consists of dual AD channels with 14-bit resolution and dual DA channels with 14-bit resolution [69]. In the experiments, the AD/DA board was connected to the GPIO JP1 and JP2 pins of the Cyclone V DE1-SoC FPGA board. The analog input signal was connected to AD channel A, while the analog output signal was generated from DA channel A. To execute an OpenCL project, Intel SDK for OpenCL version 17.0 was used to compile the kernel. The ARM part of the SoC FPGA executed the host program, which was cross-compiled by Intel's SoC EDS.

5.5 Signal measurement

In this experiment, the OpenCL kernel for signal measurements using the OpenCL ADC component and a channel extension are demonstrated as follow.

Listing 5.3: OpenCL kernel for signal measurement.

```
1  // adc_channel_read.cl
2  #pragma OPENCL EXTENSION cl_intel_channels : enable
3
4  channel ushort ch_data_read
5  __attribute__((depth(0)))
6  __attribute__((io("ch_adc_read")));
7  __attribute__((max_global_work_dim(0)))
8
9  __kernel void adc_channel(__global ushort *restrict datain, int length)
10 {
11    for(int i=0; i<length; i++){
12          datain[i] = read_channel_intel(ch_data_read);
13    }
14 }
```

5.5.1 Experimental design

Fig.5.6 shows the FPGA-based system design for signal measurement using OpenCL. The analog signal, which is converted to digital by the ADC chip, passes through an I/O channel extension. Then, the OpenCL kernel reads and stores the data in the global memory of the FPGA. The global memory is also accessible by the host, which allows the host to read the data for further analysis. To measure the signal, the kernel attributes are declared according to the content of the *board_spec.xml* file. In the experiment, the name for the *chan_id* attribute was specified as "*ch_adc_read*". Therefore, the channel attribute in the kernel was declared as $\_\_attribute\_((io(\text{“}ch\_adc\_read'')))$. The $max\_global\_work\_dim(0)$ attribute was used to inform the OpenCL offline compiler that the kernel type was the single work item kernel.

To read the signal from the channel, the $data\_in[i] = read\_channel\_intel\ (ch\_data\_read)$ API call was used, where the *ch_data_read* was the name of the channel variable. The results were stored in a *datain* buffer on the global memory of the FPGA. To execute this kernel, the host invoked the $clEnqueueTask()$ function in the host code. Meanwhile, the $clEnqueuereadBuffer()$ function was called to read the *data_in* data in global memory. The OpenCL kernel for the signal measurement is shown in Listing 5.3.

5.5.2 Implementation and result

In this first experiment, an arbitrary signal generator is employed to generate different signal types, such as sine, triangle, and square wave signals. Fig.5.7 shows examples of the measured input signal types by the OpenCL kernel (sine wave, triangle wave, and square wave) by leveraging the I/O channel extension in the OpenCL kernel. To evaluate the frequency of the signal, the kernel sampling rate for the measured signal (T) is required. This sampling rate can be calculated from the kernel execution time (t_{kernel}) divided by the length of the data (n), as defined by Equation 5.1. Fig.5.8 shows the signal, which is sampled every T seconds over length n of the dataset.

$$T = \frac{t_{kernel}}{n} \tag{5.1}$$

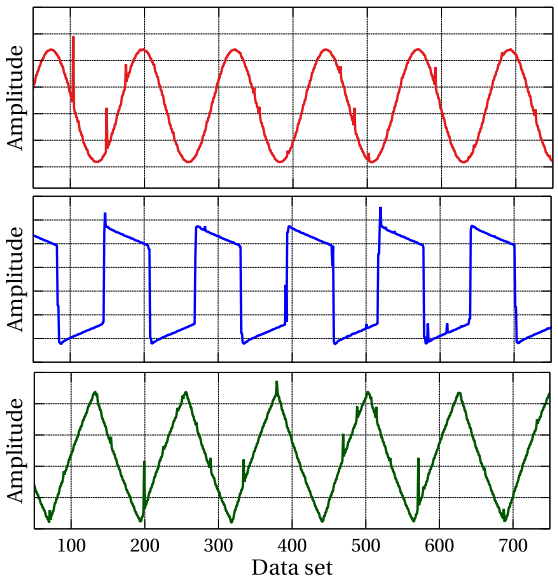

Figure 5.7: Measured input signal by the FPGA from an arbitrary signal generator.

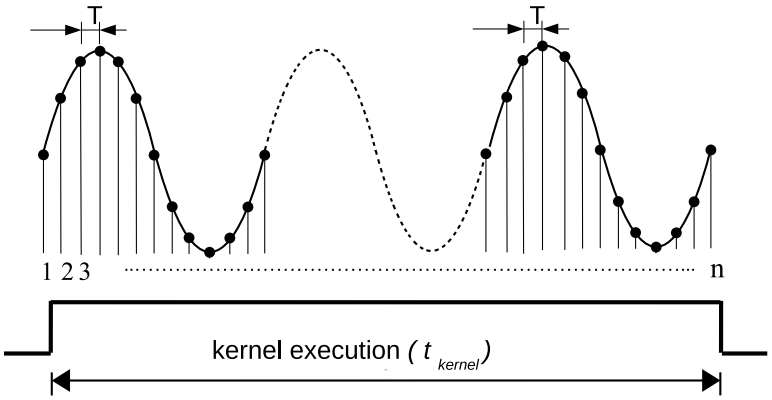

Figure 5.8: Kernel execution time for signal measurement.

In the experiment, the signal generator was set to generate a sine wave (f_{in}) with frequency of 20 MHz. The kernel was programmed to read and store the data to global memory with different lengths (n) as follows: 50K, 60K, and 75K. From the experimental results, the kernel execution time was $t_{kernel\ 1} = 0.686\ s$, $t_{kernel\ 2} = 0.801\ s$, and $t_{kernel\ 3} = 0.974\ s$ for $n = 50$K, $n = 60$K, and $n = 75$K, respectively. By applying the fast Fourier transform (FFT) function, the frequency of the measured signals for different lengths (n) of the dataset are shown in Fig.5.9. The measured frequencies for the 20 MHz input frequency were 16.85 MHz, 17.31 MHz, and 17.8 MHz for $n = 50$K, $n = 60$K, and $n = 75$K, respectively. The results show that the measured frequency is lower than the input frequency. This is because of the slow kernel execution time owing to the use of global memory for storing the measured signal.

To avoid the global memory constraint, the experiment was carried out by storing the data temporarily in the on-chip RAM of the FPGA instead of writing directly to global memory. The kernel execution time decreased to $t_{kernel3} = 0.871\ s$ for $n = 75$K. The result indicates that the use of an on-chip RAM achieves faster execution time.

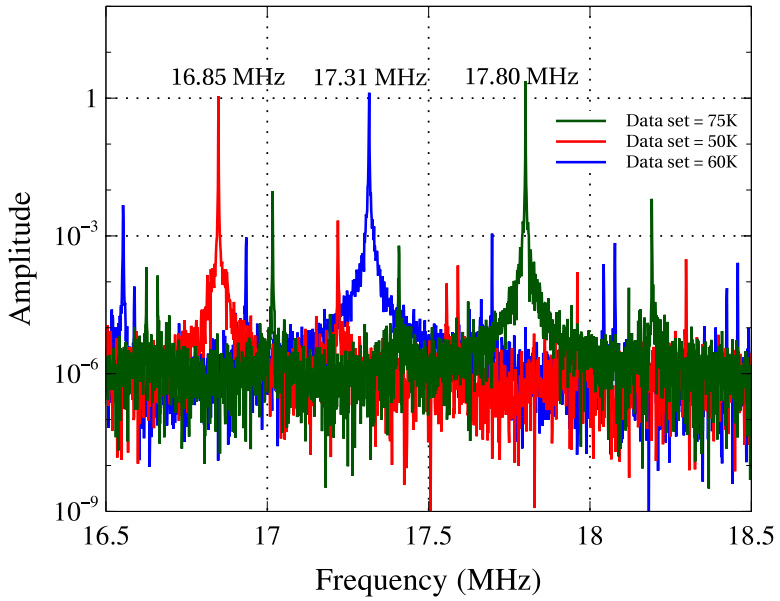

Figure 5.9: Measured frequency using global memory for a 20 MHz frequency input signal.

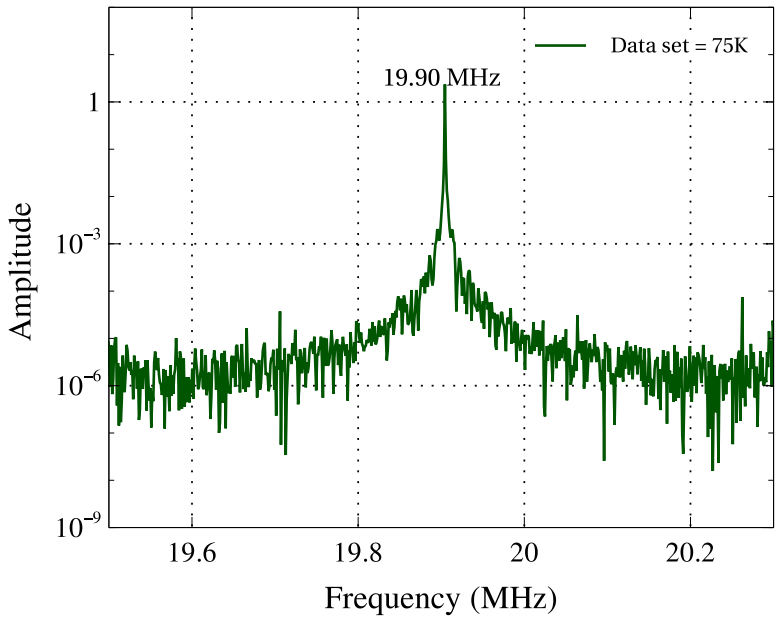

Figure 5.10: Measured frequency using on-chip RAM for a 20 MHz frequency input signal.

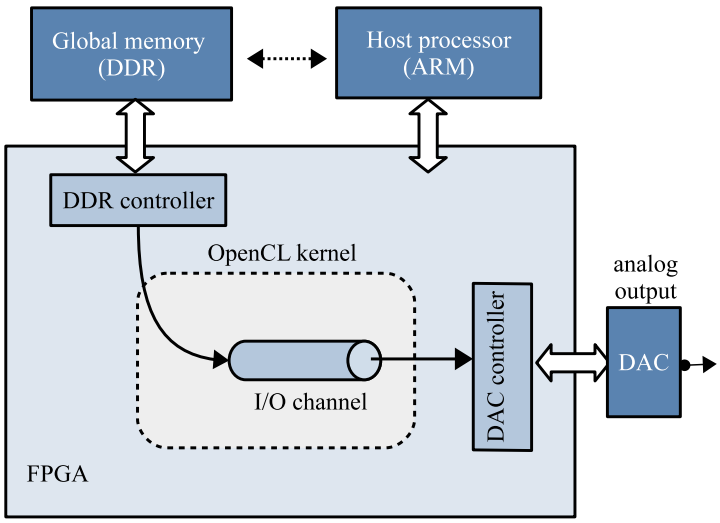

Figure 5.11: I/O channel implementation for signal generation.

By using Equation 1 to calculate the kernel sampling rate, the measured frequency input can be evaluated using an FFT function, with the result shown in Fig.5.10. The measured frequency of 19.9 MHz is closer to the 20 MHz frequency of the input signal than the 17.8 MHz signal measured using global memory access.

5.6 Signal generation

In the following experiment, the OpenCL kernel implementation for signal generation using the OpenCL DAC component and a channel extension is demonstrated. The example implementation of this kernel applies to signal generation for wireless communication or for a radar transmitter.

5.6.1 Experimental design

To generate an output signal, first the host writes one cycle of a sine wave to the global memory of the FPGA by calling the $clEnqueueWriteBuffer()$ function. Second, the host invokes the $clEnqueueTask()$ function to execute the OpenCL kernel. On the FPGA side, the kernel reads the data from global memory and passes it to the DAC board through an I/O channel extension. This process is depicted in Fig.5.11. According to the content of the *board_sprec.xml* file, because the *chan_id* value was specified as "*ch_dac_write*", then the kernel attribute was declared as $\_\_attribute\_((io("ch\_dac\_write")))$. To read the data from global memory and to pass it to the DAC board, a $write\_channel\_intel(ch\_data\_write, sine[i])$ API call was executed, where *ch_data_write* was the name of the channel variable, and *sine* represented variable arrays containing one cycle of a sine wave. The OpenCL kernel for generating the signal is given in Listing 5.4. In the kernel, the *length* variable is defined as the length of the data in one cycle. Fig.5.12 shows how one cycle of a sine wave with length m and amplitude from 0 to 2^{14} is stored in the global memory of the FPGA.

Listing 5.4: OpenCL kernel for signal generation.

```
1  // dac_channel_write.cl
2  #pragma OPENCL EXTENSION cl_intel_channels : enable
3
4  channel ushort ch_data_write
5  __attribute__((depth(0)))
6  __attribute__((io("ch_dac_write")));
7  __attribute__((max_global_work_dim(0)))
8
9  __kernel void dac_channel(__global ushort *restrict sine, int length)
10 {
11    while(1){
12       for (int i=0; i< length; i++){
13          write_channel_intel(ch_data_write, sine[i]);}
14    }
15 }
```

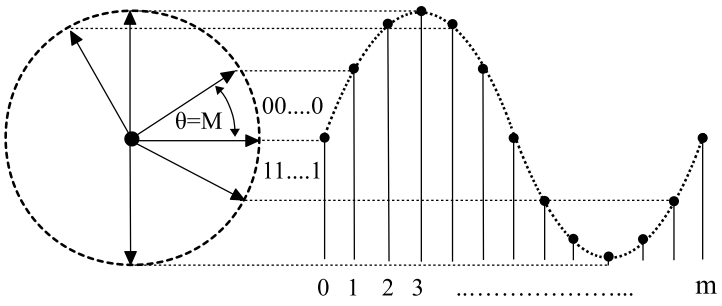

Figure 5.12: One cycle of a sine wave for a dataset of length m.

5.6.2 Implementation and results

To evaluate the kernel, an oscilloscope was employed to measure the output signal from the DAC chip. After executing the kernel, the analog output signal was generated as shown in Fig.5.13. As can be seen, there is a delay after one cycle of the generated signal (red circle), as shown in Fig.5.13(a). This delay is owing to the long latency involved in global memory usage. After one period of the signal, the kernel accesses the non-contiguous memory allocation in the global memory and generates the same signal repeatedly. To overcome this limitation, global memory usage is omitted by copying the data from global memory to the on-chip RAM of the FPGA. Fig.5.13(b) shows the result when the kernel employs the on-chip RAM, indicating that the signal is generated without delay.

To generate the signal at a specific frequency, a formula which is similar to the direct digital synthesis (DDS) architecture is proposed. The DDS technique is used to generate a sinusoidal signal or arbitrary waveform with a programmable frequency. DDS enables us to control the frequency of the signal accurately and to adjust the frequency quickly [70]. A typical DDS architecture consists of a phase accumulator (M), a reference clock f_c, and a DAC. The phase accumulator specifies the phase angle of the output signal. This phase accumulator has N-bit resolution, with a range from 1 to 2^N. The DAC converts the digital value to an analog signal [71] [72], as shown in Fig.5.12.

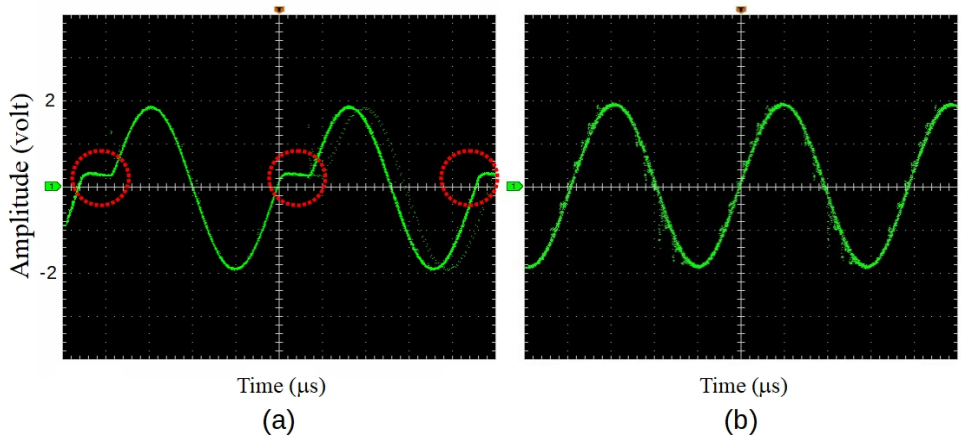

Figure 5.13: Output analog signal (a) with global memory, (b) without global memory implementation.

Table 5.2: OpenCL kernel compilation report.

FPGA resources	Usage
logic utilization	2,434 (8%)
DSP blocks	0 (0%)
memory bits	536K (13%)
RAM blocks	86 (22%)
kernel clock	145.07 MHz

In the experiment, the dataset for one cycle of the sine wave is represented by the data from 0 to 2^{14} because the ADC and DAC have 14-bit resolution. To generate a signal with a specific frequency, the host writes one cycle of a sine wave with the length of the data (m) to global memory, as shown in Fig.5.12. In DDS implementation, the frequency of the signal with $N - bit$ resolution can be calculated using Equation 5.2.

$$f_o = M \times \frac{f_c}{2^N} \tag{5.2}$$

because m is equal to 2^N divided by M, the frequency of the output signal can be estimated using Equation 5.3 as follows:

$$F_{estimation} = \frac{f_{kernel}}{m} \tag{5.3}$$

Here, f_{kernel} is the working frequency of the kernel. This working frequency can be obtained from the OpenCL kernel compilation reports as shown in Table 5.2. From the table, the working frequency of the kernel for this implementation is 145.07 MHz. Fig.5.14 shows the comparison between the frequency estimation and the actual frequency of the signal using a spectrum analyzer. The results show that the frequency estimation is similar to the actual frequency of the signal. The larger the value of m is, the lower the frequency of the output signal. Fig.5.15 shows examples of the output frequency for different m data lengths.

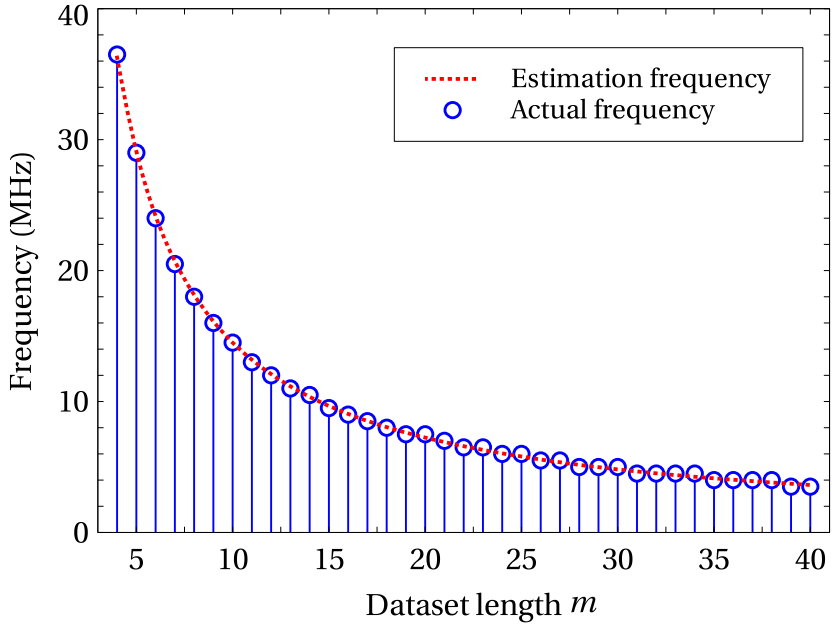

Figure 5.14: Output frequency for dataset length m.

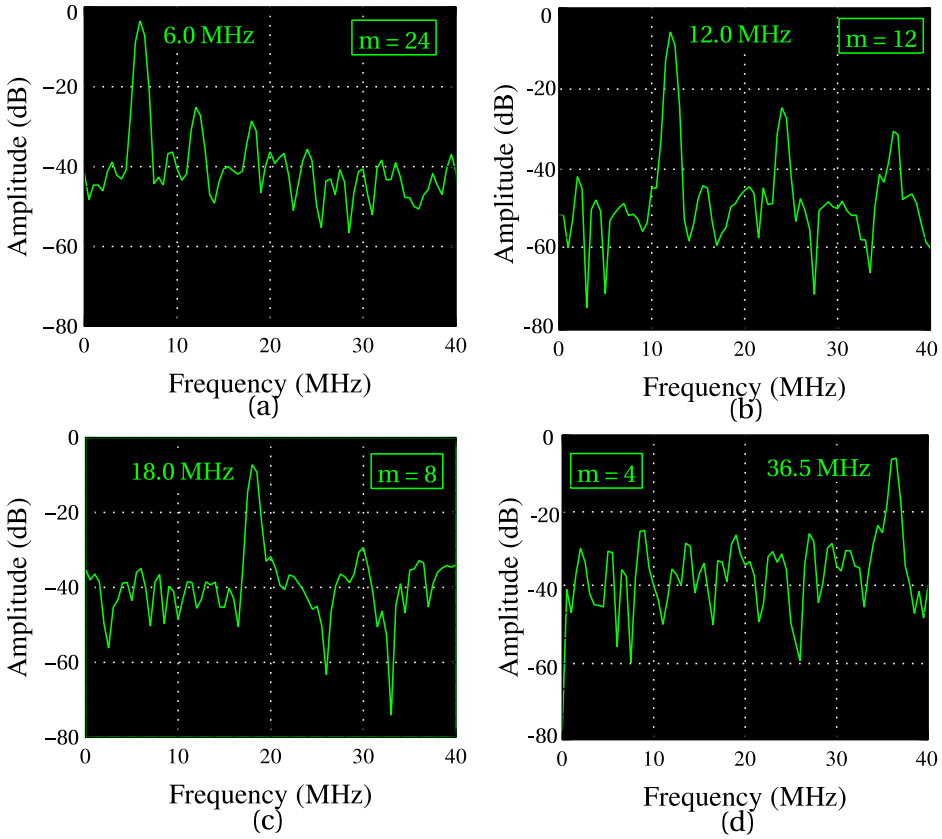

Figure 5.15: Frequency of the output signal for different data lengths: (a) $m = 24$, (b) $m = 12$, (c) $m = 8$, and (d) $m = 4$.

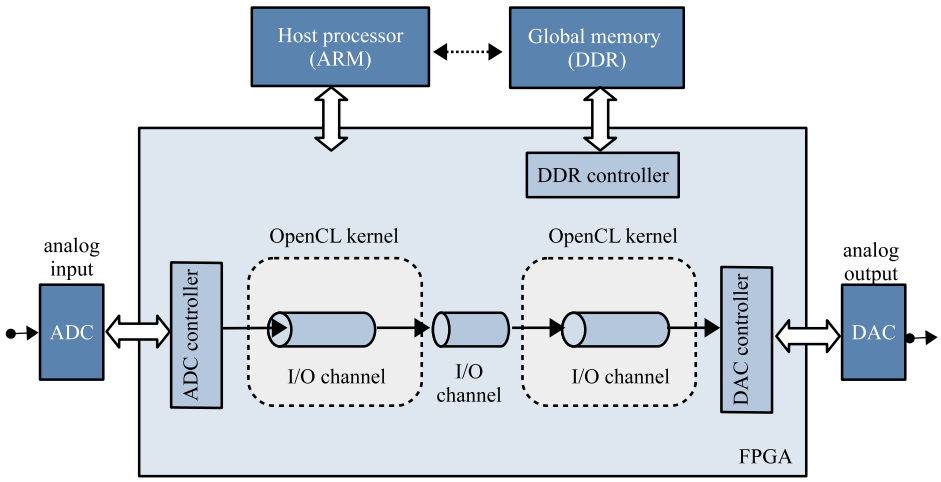

Figure 5.16: I/O channel implementation for measuring a signal, passing data, and generating a copy of the signal.

5.7 Signal measurement and generation

In the third experiment, signal measurement and signal generation are demonstrated using OpenCL ADC and DAC components that are executed simultaneously. Kernel-to-kernel data passing using a channel extension without global memory usage is also presented.

5.7.1 Experimental design

In Fig.5.16, it is shown a simultaneous signal measurement and generation execution. The measured signal is passed through an I/O channel extension and is generated directly without accessing global memory. To measure and to generate a signal simultaneously, three OpenCL kernels are developed in a single OpenCL file, as shown in Listing 5.5. An efficient kernel-to-kernel data communication is demonstrated through a channel extension without accessing global memory for storing and reading data. In the *adc_channel* kernel, the signal from the ADC is read and stored in the *chan_input* channel. In the opposite direction, the *dac_channel* kernel reads the data from the *chan_output* channel and sends the data to the DAC to generate a signal. To pass data between the *adc_channel* kernel and the *dac_channel* kernel, the *in_out* kernel performs a data copy from the *chan_input* channel to the *chan_output* channel. In this implementation, the *autorun* attribute is declared for the *in_out* kernel so that the kernel is automatically executed without a host invocation.

5.7.2 Implementation and result

To evaluate the OpenCL kernel for this implementation, an experiment was conducted by sending an analog input signal generated by an arbitrary signal generator through the ADC chip on the FPGA board. To measure the output signal from the DAC chip, an oscilloscope and a spectrum analyzer were used. Fig.5.17 shows the comparison between the analog input signal and the analog output signal. It can be seen that the input signal is similar to the output signal. The signal measured by

Listing 5.5: OpenCL kernel for signal measurement and generation.

```
1  // adc_dac_channel.cl
2  #pragma OPENCL EXTENSION cl_intel_channels : enable
3
4  channel ushort ch_data_write
5  __attribute__((depth(0)))
6  __attribute__((io("ch_dac_write")));
7
8  channel ushort ch_data_read
9  __attribute__((depth(0)))
10 __attribute__((io("ch_adc_read")));
11
12 channel ushort chan_input;
13 channel ushort chan_output;
14
15 __attribute__((max_global_work_dim(0)))
16 __kernel void adc_channel() {
17    while(1)
18    {
19          write_channel_intel(chan_input, read_channel_intel(ch_data_read));
20    }
21 }
22
23
24 __attribute__((max_global_work_dim(0)))
25 __attribute__((autorun))
26 __kernel void in_out() {
27    while(1)
28    {
29          ushort input = read_channel_intel(chan_input);
30          write_channel_intel(chan_output, input);
31    }
32 }
33
34
35 __attribute__((max_global_work_dim(0)))
36 __kernel void dac_channel () {
37    while(1)
38    {
39          write_channel_intel(ch_data_write, read_channel_intel(chan_output));
40    }
41 }
```

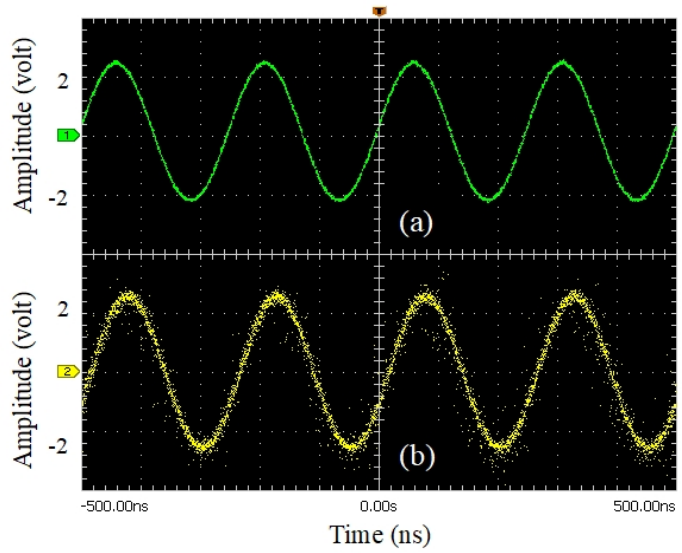

Figure 5.17: Comparison between (a) input sine wave and (b) output sine wave.

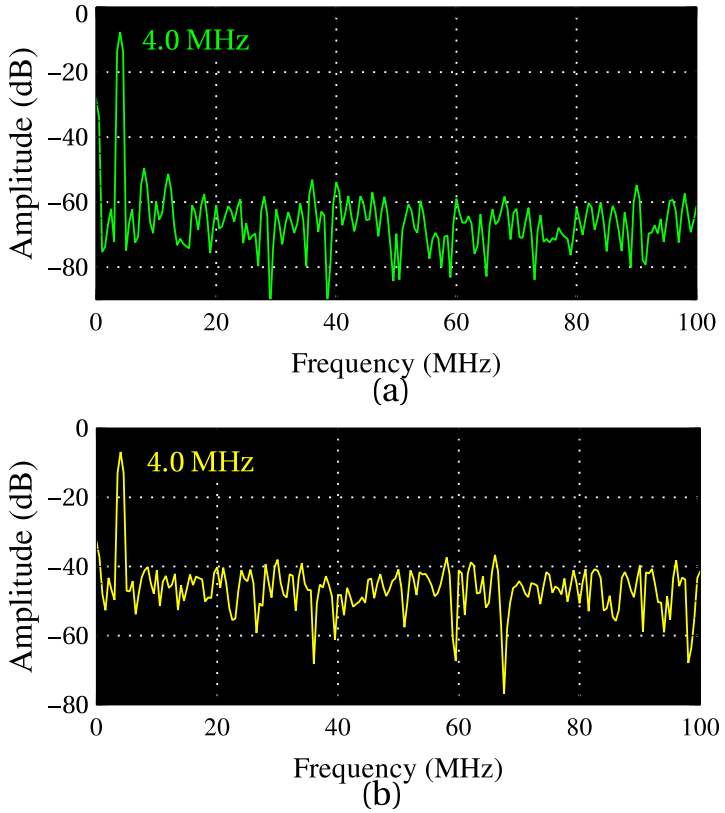

Figure 5.18: Comparison between (a) input frequency and (b) output frequency.

the *adc_channel* kernel is passed through a channel extension and generated by the *dac_channel* kernel. The frequency for both input and output signals were also investigated. Fig.5.18 shows the frequency comparison between the input and the output signals. It can be seen that the frequencies for both signals are the same.

5.8 Conclusions

The implementation of FPGA-based signal measurement and signal generation using OpenCL is demonstrated by using OpenCL ADC and DAC components that can access an FPGA's I/O directly. To allow the OpenCL kernel to stream data to the FPGA's I/O, OpenCL I/O channel extensions are employed for both reading and writing data. In the signal measurement experiment, the measured signal is demonstrated to be similar to the input signal. For signal generation, the frequency of the generated signal is similar to the estimation frequency. The implementation of the I/O channel extension for data transmission between two kernels is also demonstrated. In the experiment, the measured signal is passed to a channel extension and is generated simultaneously without having to access global memory. It can be observed that the frequency of the input signal is similar to the frequency of the output signal. This study has shown that the OpenCL can be used for accessing an FPGA's I/O, particularly for signal measurement and signal generation.

Chapter 6

Conclusions

The implementation of FPGA-based design using OpenCL for both high-performance computing applications and engineering applications has been demonstrated. The experiments were conducted according to the external memory bandwidth, computational capability, and I/O capability. For the high-performance computing applications, the Himeno benchmark was chosen as an example of the memory-intensive use that requires high bandwidth of the external memory. The OpenCL kernels were evaluated under two directions according to the number of external memory access: straightforward implementation and optimized implementation. The former implementation with loop-unrolling achieves 5.79 GFlops and 7.18 GFlops on a Stratix V GX FPGA and an Arria 10 A10PL4 FPGA, respectively. Loop unrolling helped the degree of parallelism to increases and allowed the kernel to process more data within one FPGA clock cycle. On the other hand, the peak performances for the optimized implementation are 10.62 GFlops and 13.95 GFlops for the Stratix V FPGA and the Arria 10 FPGA, respectively, by combining the temporal-blocking kernel with shift-register implementation. It makes the performance increase compared to the straight forward approach. But, the usage of RAM blocks increased in the temporal-blocking kernel because it produces data dependency among the variables on the on-chip RAM due to a copy operation. Consequently, some iterations are executed serially, and the performance is not fully optimized. To avoid the data dependency caused by a copy operation in the temporal-blocking kernel, a shift-register kernel is implemented. The shift-register improves the efficiency of external memory accesses via the use of temporal locality. This implementation also decreases the RAM block usage and removes data dependency caused by the temporal-blocking method. By combining the temporal-blocking technique with a shift-register kernel, it is found that the implementation of the shift register effectively removes the memory dependency that is caused by the temporal-blocking implementation in the memory-intensive application.

The evaluation of the GEMM implementation as an example of compute-intensive applications using global memory and local memory was demonstrated using Stratix V FPGA. For global memory usage, the peak performance achieved 5.78 GFlops by omitting cache usage. For the local memory implementation, a peak performance of 144.95 GFlops was achieved in a single-precision data type. In this study, the equation for estimating the performance was introduced. The results show that the peak performances were similar to the performance estimation, which is calculated from the OpenCL kernel compilation report. In this GEMM implementation, the kernel code also shares similarities with the OpenCL code for GPU. Therefore, the development time in FPGA programming can be reduced by porting the code from the GPU's code.

In multiple FPGA implementations, a peak performance of 579.41 GFlops was achieved in a single-precision data type. For the double-precision data type, the system achieved 125.44 GFlops for four Stratix V DE5-Net FPGAs. These results show that the performance of multiple FPGAs increases linearly. In terms of the performance-power ratio, the performance efficiency of multiple FPGAs relative to single FPGA increases to 1.61x, 1.99x, and 2.29x for two, three, and four Stratix V DE5-Net FPGAs respectively. The results show that multiple FPGA implementations consume less power. This evidence shows that multiple FPGA implementations provide an alternative solution for high-performance computing applications with low power consumption.

To demonstrate the capability of FPGA to access the I/O using OpenCL design, the implementation of FPGA-based design for signal measurement and signal generation is presented. The experiments were performed by developing new OpenCL ADC and DAC components that can access an FPGA's I/O directly. To allow the OpenCL kernel to stream data to the FPGA's I/O, OpenCL channel extensions were employed for both reading and writing data. In the signal measurement experiment, the measured signal was demonstrated to be similar to the input signal. For a signal generation, the frequency of the generated signal was similar to the estimation frequency. To transfer the data between two kernels, the implementation of the I/O channel extension was also demonstrated. In the experiment, the measured signal was passed to a channel extension and was generated simultaneously without having to access global memory. The results showed that the frequency of the input signal was similar to the frequency of the output signal. This study has shown that the OpenCL programming language can be used for accessing an FPGA's I/O, specifically for the signal measurement and signal generation. Because FPGA is well known for its high bandwidth I/O capabilities with low latency, this approach opens the way for using OpenCL not only for high-performance computing applications but also for the engineering applications that need access to the external devices such as for signal processing, image or video processing, software-defined radio, high-speed data acquisitions, and so on. By implementing OpenCL in the FPGA design, it increases productivity and reduces the development time.

Bibliography

[1] Yukihiro Komura. Gpu-based cluster-labeling algorithm without the use of conventional iteration: Application to the swendsen–wang multi-cluster spin flip algorithm. *Computer Physics Communications*, 194, 04 2015.

[2] Tyng-Yeu Liang, Hung-Fu Li, Yu-Jie Lin, and Bi-Shing Chen. A distributed ptx virtual machine on hybrid cpu/gpu clusters. *Journal of Systems Architecture*, 62, 10 2015.

[3] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 280–289, New York, NY, USA, 2010. Association for Computing Machinery.

[4] R. Kobayashi, N. Fujita, Y. Yamaguchi, A. Nakamichi, and T. Boku. Gpu-fpga heterogeneous computing with opencl-enabled direct memory access. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 489–498, May 2019.

[5] N. Fujita, R. Kobayashi, Y. Yamaguchi, and T. Boku. Parallel processing on fpga combining computation and communication in opencl programming. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 479–488, May 2019.

[6] Martin Herbordt, Tom Vancourt, Yongfeng Gu, Bharat Sukhwani, Al Conti, Josh Model, and Doug Disabello. Achieving high performance with fpga-based computing. *Computer*, 40:50–57, 03 2007.

[7] Ruben Ricart-Sanchez, Pedro Malagon, Pablo Salva, Enrique Chirivella Pérez, Qi Wang, and Jose Calero. Towards an fpga-accelerated programmable data path for edge-to-core communications in 5g networks. *Journal of Network and Computer Applications*, 124, 09 2018.

[8] W. Zheng, Rocky Liu, Mudi Zhang, Gao Zhuang, and T. Yuan. Design of fpga based high-speed data acquisition and real-time data processing system on j-text tokamak. *Fusion Engineering and Design*, 89, 02 2014.

[9] Yufei Ma, Naveen Suda, Yu Cao, Sarma B. K. Vrudhula, and Jae sun Seo. Alamo: Fpga acceleration of deep learning algorithms with a modularized rtl compiler. *Integration*, 62:14–23, 2018.

[10] N. Fujii and N. Koike. Iot remote group experiments in the cyber laboratory: A fpga-based remote laboratory in the hybrid cloud. In *2017 International Conference on Cyberworlds (CW)*, pages 162–165, Sep. 2017.

[11] Sejin Jung, Eui-Sub Kim, Junbeom Yoo, Jang-Yeol Kim, and Jong Choi. An evaluation and acceptance of cots software for fpga-based controllers in npps. *Annals of Nuclear Energy*, 94:338–349, 08 2016.

[12] M. Pelcat, C. Bourrasset, L. Maggiani, and F. Berry. Design productivity of a high level synthesis compiler versus hdl. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 140–147, July 2016.

[13] Khronos Group. The open standard for parallel programming of heterogeneous systems. *available online at https://www.khronos.org/opencl/*, accessed in May 2016, 2016.

[14] G. Guidi, E. Reggiani, L. D. Tucci, G. Durelli, M. Blott, and M. D. Santambrogio. On how to improve fpga-based systems design productivity via sdaccel. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 247–252, May 2016.

[15] Intel Corporation. Implementing fpga design with the opencl standard. 2013.

[16] D. Chen and D. Singh. Invited paper: Using opencl to evaluate the efficiency of cpus, gpus and fpgas for information filtering. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 5–12, Aug 2012.

[17] Hasitha Waidyasooriya, Masanori Hariyama, and Kunio Uchiyama. *Design of FPGA-based computing systems with openCL*. 11 2017.

[18] E. Rucci, C. García, G. Botella, A. D. Giusti, M. Naiouf, and M. Prieto-Matias. Smith-waterman protein search with opencl on an fpga. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 3, pages 208–213, Aug 2015.

[19] John Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science engineering*, 12:66–72, 05 2010.

[20] Intel FPGA. Intel sdk for opencl: Best practices guide. *available online at https://www.intel.com*, 2018.

[21] Acceleware. Opencl on fpgas for gpu programmers. *available online at https://www.intel.com*, 2018.

[22] Ze-ke Wang, Shuhao Zhang, Bingsheng He, and Wei Zhang. Melia: A mapreduce framework on opencl-based fpgas. *IEEE Transactions on Parallel and Distributed Systems*, 27:1–1, 12 2016.

[23] S.O. Settle. High-performance dynamic programming on fpgas with opencl. *IEEE Proc. High Perform. Extr. Comp. Conf (HPEC)*, 2013.

[24] M. Korch, T. Rauber, and C. Scholtes. Memory-intensive applications on a many-core processor. In *2011 IEEE International Conference on High Performance Computing and Communications*, pages 126–134, Sep. 2011.

[25] Ken'ichi Itakura, Akihiro Yamashita, Koji Satake, Hitoshi Uehara, Atsuya Uno, and Mitsuo Yokokawa. *Feasibility Study of a Future HPC System for Memory Intensive Applications: Conceptual Design of Storage System*, pages 81–88. 11 2015.

[26] Khaled Z. Ibrahim, Steven Hofmeyr, Costin Iancu, and Eric Roman. Optimized pre-copy live migration for memory intensive applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 40:1–40:11, New York, NY, USA, 2011. ACM.

[27] Chao Huang, Srivaths Ravi, Anand Raghunathan, and N.K. Jha. Generation of heterogeneous distributed architectures for memory-intensive applications through high-level synthesis. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 15:1191 – 1204, 12 2007.

[28] Yun Zhang, Faisal N. Abu-Khzam, Nicole E. Baldwin, Elissa J. Chesler, Michael A. Langston, and Nagiza F. Samatova. Genome-scale computational approaches to memory-intensive applications in systems biology. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, page 12, USA, 2005. IEEE Computer Society.

[29] Himeno benchmark. *http://accc.riken.jp/*.

[30] Yukinori Sato, Y. Inoguchi, Wayne Luk, and Tadao Nakamura. Evaluating reconfigurable dataflow computing using the himeno benchmark. In *2012 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2012*, pages 1–7, 12 2012.

[31] Terasic. De5-net fpga development kit user manual. *available online at http://www.terasic.com.tw/en/*, 2017.

[32] Terasic. Arria 10 de5a-net fpga development kit user manual. *http://www.terasic.com.tw/en/*, 2017.

[33] NVidia. *http://www.nvidia.com.*

[34] M. Wissolik, D. Zacher, A. Torza, and B. Day. Virtex ultrascale+ hbm fpga: A revolutionary increase in memory performance. *available: http://www.xilinx.com*, 2018.

[35] M. Deo, J. Schulz, and J.L. Brown. Intel stratix 10 mx devices solve the memory bandwidth challenge. *http://www.intel.com/*, 2018.

[36] Hasitha Waidyasooriya, Masanori Hariyama, and Kunio Uchiyama. *Design of FPGA-based computing systems with openCL.* 11 2017.

[37] Bittware. Arria 10 gx low profile pcie board with dual qsfp and ddr4. *available: http://www.bittware.com/*, 2018.

[38] Intel FPGA. Intel sdk for opencl programming guide. *https://www.intel.com*, 2018.

[39] Qi Jia and Huiyang Zhou. Tuning stencil codes in opencl for fpgas. *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 249–256, 2016.

[40] Hasitha Muthumala Waidyasooriya, Yasuhiro Takei, Shunsuke Tatsumi, and Masanori Hariyama. Opencl-based fpga-platform for stencil computation and its optimization methodology. *IEEE Trans. Parallel Distrib. Syst.*, 28(5):1390–1402, May 2017.

[41] N. Sato, T. Chigira, K. Toyoda, Y. Iijima, and Y. Yuminaka. Multi-valued signal generation and measurement for pam-4 serial-link test. In *2018 IEEE 48th International Symposium on Multiple-Valued Logic (ISMVL)*, pages 210–214, May 2018.

[42] E. H. Phillips and M. Fatica. Implementing the himeno benchmark with cuda on gpu clusters. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–10, Los Alamitos, CA, USA, apr 2010. IEEE Computer Society.

[43] Bocheng Liu, Chen Qingkui, Jinjing Li, and Liping Gao. Ai bcs: A gpu cluster scheduling optimization based on ske model. *Microprocessors and Microsystems*, 47, 05 2016.

[44] Jakub Kurzak, Pitor Luszczek, Stanimire Tomov, and Jack Dongarra. Preliminary results of autotuning gemm kernels for the nvidia kepler architecture- geforce gtx 680.

[45] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. Fast implementation of dgemm on fermi gpu. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[46] Naohito Nakasato. A fast gemm implementation on the cypress gpu. *SIGMETRICS Performance Evaluation Review*, 38:50–55, 03 2011.

[47] K. Matsumoto, N. Nakasato, and S. G. Sedukhin. Performance tuning of matrix multiplication in opencl on different gpus and cpus. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 396–405, Nov 2012.

[48] Yong Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point fpga matrix multiplication. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, FPGA '05, page 86–95, New York, NY, USA, 2005. Association for Computing Machinery.

[49] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S. Chung, and Greg Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *FCCM*, pages 36–43. IEEE Computer Society, 2014.

[50] B. Holanda, R. Pimentel, J. Barbosa, R. Camarotti, A. Silva-Filho, L. Joao, V. Souza, J. Ferraz, and M. Lima. An fpga-based accelerator to speed-up matrix multiplication of floating point operations. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 306–309, May 2011.

[51] Zoran Jakšić, Nicola Cadenelli, David Buchaca Prats, Jordà Polo, Josep Lluís Berral Garcia, and David Carrera Perez. A highly parameterizable framework for conditional restricted boltzmann machine based workloads accelerated with fpgas and opencl. *Future Generation Computer Systems*, 104:201 – 211, 2020.

[52] Intel FPGA. Matrix multiplication design example. *https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/matrix-multiplication.html*, 2019.

[53] Intel FPGA. Stratix v device overview. *available online at https://www.intel.com/*, 2017.

[54] Intel FPGA. Stratix v avalon-mm interface for pcie solutions user guide. *available online at http://www.intel.com*, 2017.

[55] X. Cui, Y. Chen, and H. Mei. Improving performance of matrix multiplication and fft on gpu. In *2009 15th International Conference on Parallel and Distributed Systems*, pages 42–48, Dec 2009.

[56] Intel FPGA. Stratix v device handbook, volume 1: Device interfaces and integration. *available online at https://www.intel.com*, 2017.

[57] Pilsoo Lee, C. Lee, and Ju Lee. Development of fpga-based digital signal processing system for radiation spectroscopy. *Radiation Measurements*, 48:12–17, 01 2013.

[58] Aboli Audumbar Khedkar and R.H. Khade. High speed fpga-based data acquisition system. *Microprocess. Microsyst.*, 49(C):87–94, March 2017.

[59] K. Venkatraman, Moorthi Sridharan, M. Selvan, and Raja Pitchaimuthu. A comprehensive embedded solution for data acquisition and communication using fpga. *Journal of Applied Research and Technology*, 15, 02 2017.

[60] Liu Qing, Cao Kai, and Lai Ying-yong. Fpga software architecture for software defined radio. *Procedia Engineering*, 29:2133–2139, 12 2012.

[61] Sabrina Zereen, Sundeep Lal, Mohammed Khalid, and Sazzadur Chowdhury. An fpga-based controller for a 77 ghz mems tri-mode automotive radar. *Microprocessors and Microsystems*, 58, 02 2018.

[62] Colm A Ryan, Blake R. Johnson, Diego Ristè, Brian Donovan, and T A Ohki. Hardware for dynamic quantum computing. *The Review of scientific instruments*, 88 10:104703, 2017.

[63] K. Hill, S. Craciun, A. George, and H. Lam. Comparative analysis of opencl vs. hdl with image-processing kernels on stratix-v fpga. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 189–193, July 2015.

[64] Hasitha Waidyasooriya, Yasuhiro Takei, Shunsuke Tatsumi, and Masanori Hariyama. Opencl-based fpga-platform for stencil computation and its optimization methodology. *IEEE Transactions on Parallel and Distributed Systems*, PP, 10 2016.

[65] Intel. Intel fpga sdk for opencl, programming guide. 2018.

[66] Intel. Intel fpga sdk for opencl pro edition custom platform toolkit user guide. 2018.

[67] Ryohei Kobayashi, Yuma Oobata, Norihisa Fujita, Yoshiki Yamaguchi, and Taisuke Boku. Opencl-ready high speed fpga network for reconfigurable high performance computing. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia 2018, pages 192–201, New York, NY, USA, 2018. ACM.

[68] Terasic. De1-soc opencl user manual. *available online at http://www.terasic.com*, 2018.

[69] Terasic. Thdb-ada user manual. *available online at http://www.terasic.com*, 2018.

[70] Stefan Leitner, Haibo Wang, and Spyros Tragoudas. Design techniques for direct digital synthesis circuits with improved frequency accuracy over wide frequency ranges. *Journal of Circuits System and Computers*, 26, 01 2016.

[71] B. Cronin. Dds devices generate highquality waveforms simply, efficiently, and flexibly. *http://www.analog.com*, 2018.

[72] Fundamentals of direct digital synthesis (dds). *http://www.analog.com*, 2018.

List of publications

Reviewed journal

1. Iman Firmansyah and Yoshiki Yamaguchi. “OpenCL Implementation of FPGA-Based Signal Generation and Measurement”. IEEE Access, vol. 7, pp: 48849-48859, DOI: 10.1109/ACCESS.2019.2910391, 2019.

Reviewed conferences

1. Iman Firmansyah, Du Changdao, Norihisa Fujita, Yoshiki Yamaguchi, and Taisuke Boku. ”FPGA-based Implementation of Memory-Intensive Application using OpenCL”. In Proceeding of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies, ACM International Conference Proceeding (ICPS), Article No.16, DOI: 10.1145/3337801.3337806, 2019.

2. Iman Firmansyah, Yusuf N. Wijayanto, and Yoshiki Yamaguchi, ”2D Stencil Computation on Cyclone V SoC FPGA using OpenCL”. In Proceeding of IEEE International Conference on Radar, Antenna, Microwave, Electronics, and Telecommunications (ICRAMET), pp. 121-124, DOI: 10.1109/ICRAMET.2018.8683924, 2018.

3. Iman Firmansyah, Yoshiki Yamaguchi, and Taisuke Boku, ”Performance evaluation of Stratix V DE5-Net FPGA board for high performance computing”. In Proceeding of IEEE International Conference on Computer, Control, Informatics and its Applications (IC3INA), pp. 23-27, DOI: 10.1109/IC3INA.2016.7863017, 2016.

Poster presentations

1. Iman Firmansyah, Yoshiki Yamaguchi, and Taisuke Boku. ”Capability assessment of a multiple-FPGA system for high-performance computing”. ISC High Performance, Frankfurt, 2016.

2. Iman Firmansyah and Yoshiki Yamaguchi, ”A flexible high-performance computing on an FPGA cluster”. JSPS 8th HOPE Meeting with Nobel laureates, Tsukuba, 2016.