Maximizing the performance of Parallel Applications on Heterogeneous CPU-FPGA System

By

Shuchen Zheng

M.S., Brown University, 2016

Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the School of Engineering at Brown University.

PROVIDENCE, RHODE ISLAND

MAY 2016

AUTHORIZATION TO LEND AND REPRODUCE THE THESIS

As the sole author of this thesis, I authorize Brown University to lend it to other institutions or individuals for the purpose of scholarly research.

Date_____ Signature_____

Shuchen Zheng, Author

I further authorize Brown University to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Date_____ Signature_____

Shuchen Zheng, Author

Signature Page

This thesis by Shuchen Zheng is accepted in its present form by the School of Engineering at Brown University as satisfying the thesis requirements for the degree of Master of Science.

Date_____ Signature_____

Sherief M. Reda, Ph.D., Advisor

Approved by the Graduate Council

Date_____ Signature_____

Peter Weber, Ph.D., Dean of the

Graduate School

Vita

Shuchen Zheng was born in xi'an, Shaanxi Province, P.R. China, in February 1992. She received her undergraduate degree from Beijing Institute of Technology, China in 2014 in Electrical Engineering. Shuchen started to be a Masters student in Brown School of Engineering from 2014, and study after Prof. Sherief M. Reda on CPU-FPGA heterogeneous Computing.

Acknowledgements

I would like to express my thanks to the following people for their continuous support and assistance. The completion of either this thesis or my Master degree could not have been accomplished without them:

Professor Sherief Reda, for his professional advice and providing me with the valuable learning opportunity.

Kapil Dev, for his countless helps in both experimental study and thesis write-up.

Xin Zhan, for his continuous assistance in my coursework and research.

My parents, for their always supporting and love.

Table of Contents

List of Figure

List of Tables

1. Introduction

1.1. Heterogeneous Computing

A bottleneck of general-purpose architectures is that they usually spend most of the time on tasks for which they are not optimized to execute. The underlying reason may be that many believe that many different tasks can adapt to the same kind of hardware computation, with the only discriminant being how long they take to execute. Unfortunately, there is not an all-encompassing architecture can well match all types of data access patterns or algorithms. As a result, only a fraction of the potential peak performance can be achieved on many applications [1].

To better match application needs to appropriate hardware resources, and to provide higher computational performance, heterogeneous computing is being advocated. Heterogeneous computing refers to a single computing system that coordinately use different types of cores, accelerator devices, or memory interfaces to optimize their combined performance or/and power efficiency [2]. The increasingly growth of many- or multi-core processor with parallel, pipelined, and other special-purpose architectures applied to high performance computing systems (HPC) makes it possible to effectively handle computationally intensive problems with heterogeneous computing. Compared with using traditional homogeneous baseline systems, each task can be assigned to an “appropriate” processor or device to maximize total performance. Since the tasks are well matched to the hardware resources, and all the processors/devices can execute concurrently, the use of heterogeneous computing would easily lead to an optimal computational performance.

Figure 1.1.1 shows an ideal example of a program whose subtasks are adapted to different types of architectures [1]. A pipelined system can give twice the performance achieved by executing it on some linear single-core system.

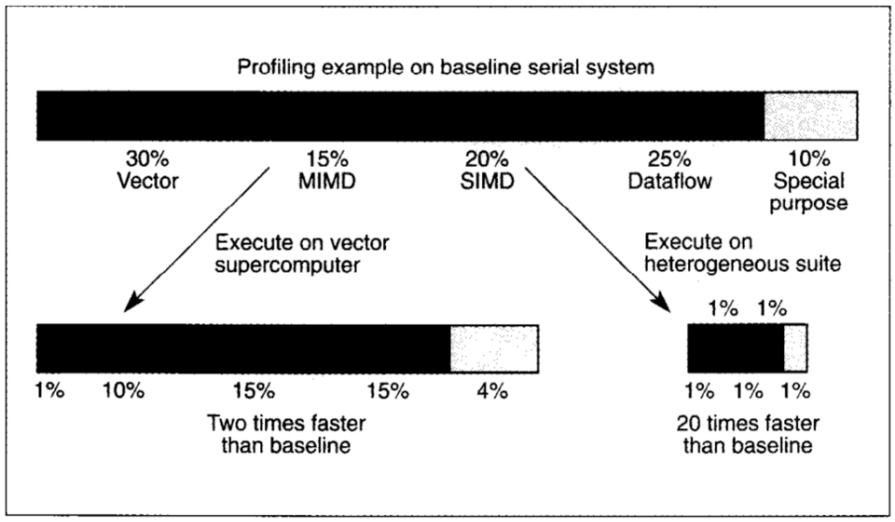


Figure 1.1.1 Hypothetical example of using heterogeneous processing [1].

Heterogeneous computing supports a great number of different applications. Although they may have diverse implementations, all the heterogeneous computing-oriented platforms should be able to perform the following the functions [6]:

1) Detect all available components/devices of the heterogeneous system.
2) Explore and adapt to each component/device's characteristics.
3) Generate instruction blocks (kernels) that can be executed on device.
4) Build and manage the memory objective.
5) Execute each kernel in appropriate device as well as in appropriate order.
6) Return the final result.

The above steps constitute the baseline of a heterogeneous system. CPU-FPGA heterogeneous system is no exception. The above functions are implemented in

OpenCL programming flow, which helps programmers handle this structure with ease.

1.2. FPGA Accelerator

Heterogeneous systems utilize multiple types of processors, like CPUs (central processing unit), GPUs (graphics processing unit), DSPs (digital signal processor) and other microprocessors, to give an optimal performance with highly parallel architectures.

When we only consider the clock speed, there is no doubt that CPU will be the winner. However, CPU perfects in handling serial tasks, which means typically it can only execute one instruction per clock cycle. As a result, CPU's peak performance is much worse than GPU's. Consider a state-of-the-art CPU edition, 6-core Intel Core i7-3930K [12]. And there is an instruction named MAD (Multiply-ADD), which allows two operations to be executed in SSE register in one clock cycle. With a base frequency 3.3 GHz and hyper-threading technology, Intel CPU is possible to execute:

$$2(\text{threads/core}) \times 6(\text{core}) \times 2(\text{double precision floats}) \times 2\ (\text{MAD instruction}) = 48$$

instructions per clock cycle. Consequently, the theoretical performance of Intel Core i7-3930 will be $48 \times 3.3 = 158.4$ GFLOPS.

GPU is well known with its great graphics rendering capabilities and superior performance in computations on very large data sets. With a lot more processing units than CPU, GPU is able to execute much more instructions per clock cycle. Consider a many-core GPU beyond handling graphics only, known as GPGPU, then. Take NVidia GeForce 9800 GTX GPU, which has 128 Shader units at 1.688

GHz, for example [13]. Unlike CPU, GPU is able to execute MAD+MUL instruction, which combines MAD with one more multiply into one instruction, in a single clock cycle. Its theoretical performance can reach $128 \times 3 \times 1.688 = 648$ GFLOPS. Under such measurement, GPUs exhibit higher performance than CPUs. However, GPU also has its drawback in power consumption. It typically consumes power at the scale of 200-300 Watt, which is about twice of the power consumption of CPU.

Finally, let's think over the performance of FPGAs. FPGAs are reconfigurable integrated circuits consisting of programmable routing networks linking together logic array blocks, embedded memory blocks, and digital signal processor (DSP) blocks [5]. From its earlier days, FPGA has served as platforms for configurable computing. Instead of working as SIMD (single instruction, multiple data) devices (such as GPUs) that exploit data level parallelism, FPGAs can create custom instruction pipelines when process the instructions.

FPGA, as a hardware accelerator, exhibits advanced computation capability as software devices like CPUs, GPUs, and DSPs, and preferable energy consumption as ASICs. With the ability to configure local customized storage, FPGAs could reuse the already fetched data. Such efficient use of memory bandwidth brings FPGAs with speed-up over both CPUs and GPUs. Besides their high performance in processing data, as demonstrated in [3], the power consumption of FPGAs platform is only 10% of the GPU and 16% of CPU. Moreover, FPGAs have improved reliability and reduced mean-time-to-failure compared to GPUs because they run at lower temperatures. Based on the above advantages, FPGAs are very promising alternative solution for real application sets with computationally intensive algorithms, considering the hardware parallelism

advantage of FPGAs.

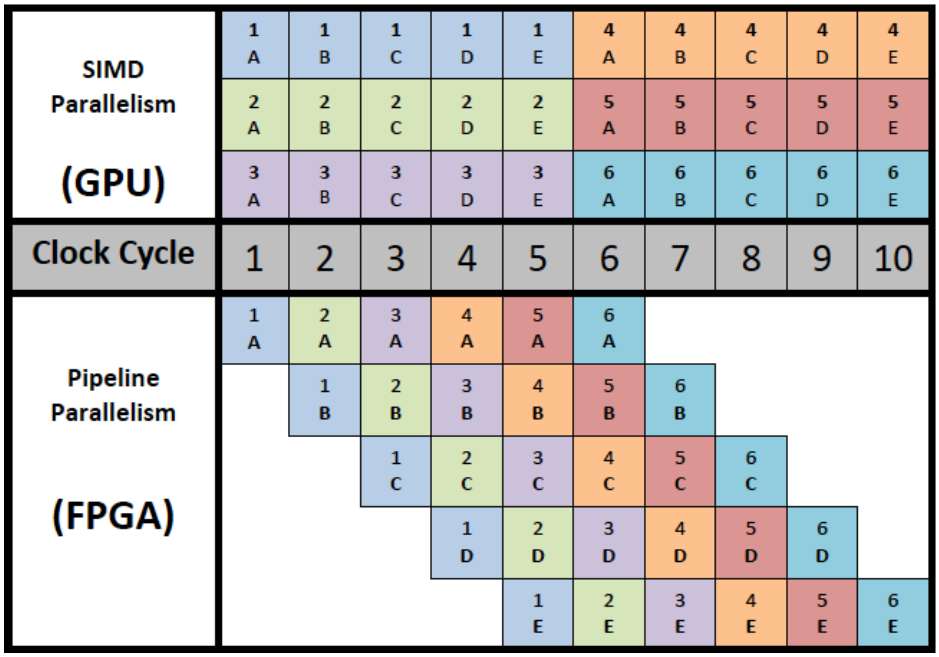


Figure 1.2.1: Comparison of parallelism of GPU and FPGA. [4]

In terms of pipeline parallelism, a function loaded to FPGA can be divided into phases and by inserting pipeline registers between operators, the execution phases can be pipelined. The pipeline can also be replicated for further parallelism. Figure 1.2.1 shows the difference of parallelism between GPU's and FPGA's.

Take *Vector Add* for instance. Figure 1.2.2 shows its GPU implementation. Each thread is going to be assigned a unique address, which marked as its global id, and all the threads will then process a kernel function concurrently.

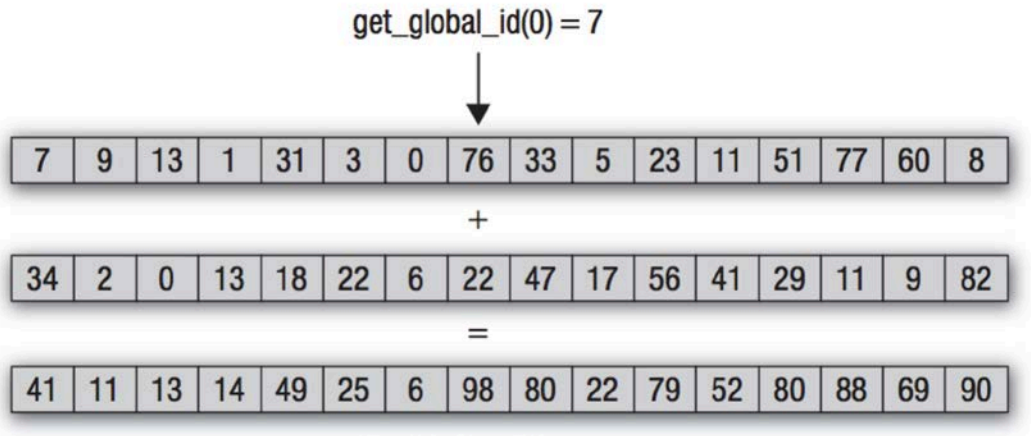


Figure 1.2.2 Mapping get_global_id to a work-item. [6]

Whereas threads execute in parallel on different cores of CPU or GPU, OpenCL compiler for FPGA can translate the executions into deeply pipelined hardware circuits [11]. FPGA implementation of adding two vectors can be separated into three portions: load operands, add, and store result, which form a pipeline dataflow. On each clock cycle each thread will be pipelined to process different operations. As shown in figure 1.2.3, work-item 0 is being stored, while 1 is being added and 2 is being loaded.

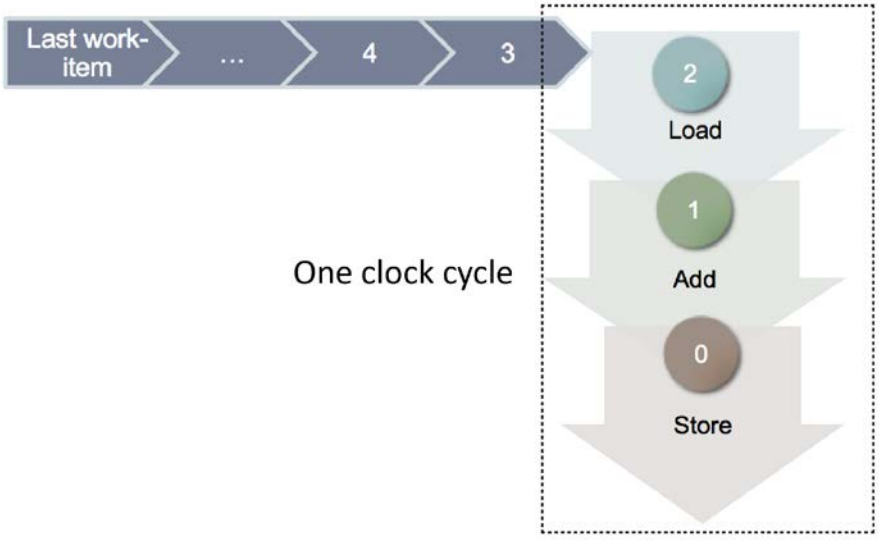


Figure 1.2.3 Pipeline example for Vector Add.

Other more complicated operations will be synthesized to pipelines with more stages, and each pipeline stage can be reused by appropriately overlapping the instructions and data.

We can either choose to form a pipelined execution within one operation or among several operations. Taking two operations for example, if the output of one operation *A* is consumed by a second operation *B*, two operations must be executed sequentially on CPU that B can only start after A has produced the entire set of executions in its output. While in hardware designs, it is possible to run operation *A* and operation *B* simultaneously on different processors, and *B* could start its execution in parallel as long as *A* produced some results.

1.3. Implementing hardware program with OpenCL

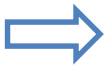
Unsatisfactory programmability and difficulties in debugging largely prevent the wider adoption of FPGA-based accelerators Implementing software programs into hardware requires solid experience and deep understanding for logics and circuits in the past. Therefore, both software engineer and hardware engineer keep exploring new solutions for automated software compilation into hardware. Inspired by such strong need, an open standard tool, OpenCL, proposed by Apple Inc. and maintained within Khronos Group supplies such demand.

Open Computing Language, short as OpenCL, is a standard framework for general-purpose parallel programming heterogeneous systems including CPUs, GPUs, FPGAs, and custom devices that execute across different computing units. With its API and a kernel-oriented programming environment, OpenCL can

achieve all the steps mentioned in Section 1.1. OpenCL specifies an ISO/IEC 9899:1999 C language (C99) based programming language that describes data-parallel kernels and tasksacross different devices [6].

Data parallelism in OpenCL is expressed as an N-dimensional (N = 1, 2, or 3) computation domain (referred to in short as NDRange). NDRange defines the total number of work-items, which execute an instance of a kernel respectively, in parallel. A further explanation for a kernel is a function executed for each work-item. Here is an example of a data-parallel kernel:

```
void trad_mul (int n,
               const float *a,
               const float *b,
               float * result)
{
    int i;
    for (i=0; i < n; ++i)
      result [i] = a[i] * b[i];
}
```



```
__kernel void vector_add(
               __global const float *x,
               __global const float *y,
               __global float *result)
{
  // get index of the work item
  int id= get_global_id(0);
  // add the vector elements
  result[id] = x[id] + y[id];
}
```

Figure 1.3.1 The implementation of vector addition in OpenCL FPGA kernel.

The kernel named *vector_add* functions as summing two vectors of floats. The inputs are referred as x and y, while result is the output. The *get_group_id(dim)* (in this example dim = 0) gets the index of each work-item and returns a one-dimensional NDRange, in which work-items spawn from *0* to *get_global_size(dim) – 1.* Each work-item, which acts as a single thread, is processed by a processing element, and those elements will form to a work-group and be executed within a

compute unit. Such functionality makes synchronization much efficient since only two types of barriers are needed: An explicit barrier for all work-items within a workgroup, and an implied barrier upon kernel completion for all work-items in an NDRange [5].

As a popular industry standard, OpenCL has multiple implementations from Altera, AMD, Apple, Intel, NVIDIA, Xilinx, IBM and many other venders. All the test-benches in this project were implemented with Altera OpenCL. The Altera SDK for OpenCL complies with the OpenCL 1.0 specification for embedded profiles and updates its supports according to the OpenCL 1.1 and 1.2 specifications [7]-[9], which significantly reduces the complexity of converting a software application to its hardware implementation. The system overview is shown in Figure 1.3.2.

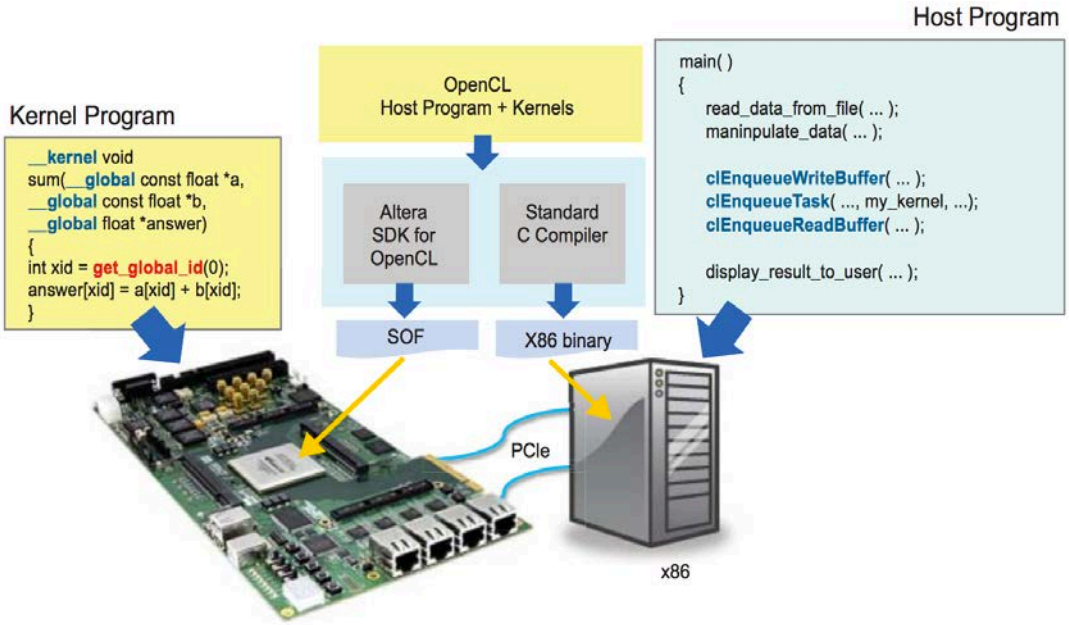


Figure 1.3.2 Altera OpenCL system overview [11].

Altera Software Development Kit for OpenCL maps the OpenCL kernel to FPGA implementation with its offline compiler, Altera OpenCL compiler (AOC). In

process of compiling, AOC will automatically translate OpenCL to Verilog and runtime libraries for the host application API and hardware obstructions [10].

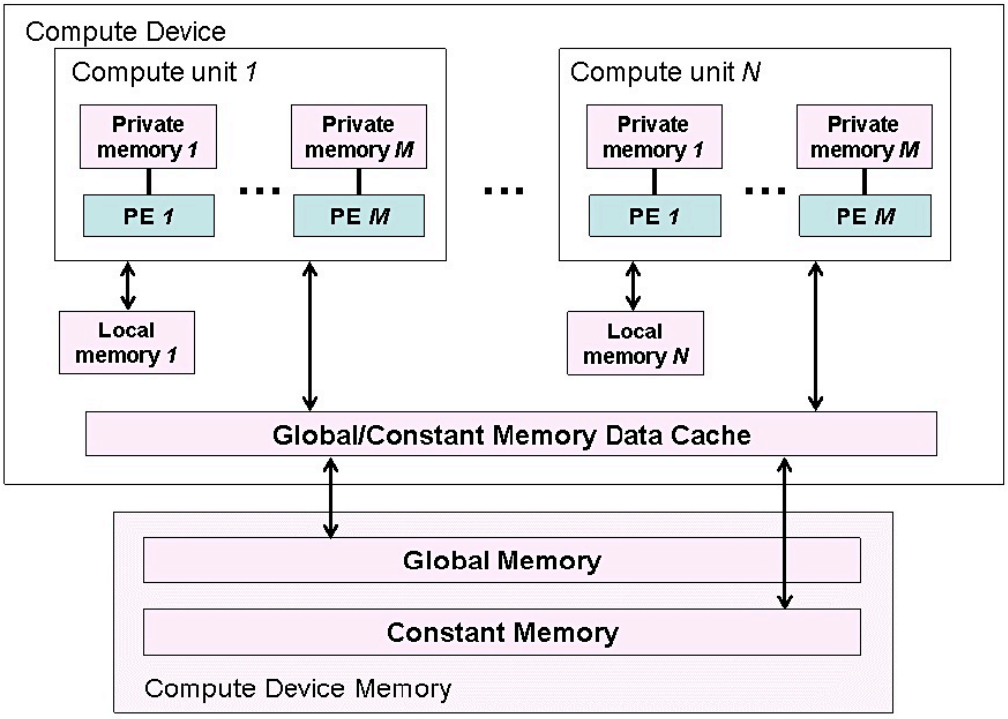


Figure 1.3.3 Altera OpenCL System Implementation [8].

A detailed and complete system implementation model of Altera OpenCL is defined in figure 1.3.3. OpenCL platform consist of a host, one or more OpenCL supported FPGA device(s) and connected by PCIe bus. An FPGA device is further divided into multiple Compute Units (CUs), and a CU is made up by a group of Processing Elements (PEs), which are work-items or SIMD units.

There are totally four memory regions, which ensure the efficiency of requesting the memory. Each work-item has its private memory that cannot be accessed by other work-items, and each compute unit has its local memory, and this memory is local to all the work-items within that work group. Different CUs can share information with external memory via global memory interconnector; loaders and savers are also connected to the global memory and scheduled by a

group of DDR DIMMs. Besides global memory, there is another external memory called constant memory, which is used by host to allocate, initialize or free memory, and stays constant while kernel is processing.

1.4. Research Overview

This thesis explores ways to minimize the execution time by maximizing the overlap and balance between communication and computation on heterogeneous systems with both CPUs and FPGAs. Then we analyze the performance and energy efficiency from using FPGA accelerated application by implementing several benchmarks.

To best study OpenCL workflow and leverage its advantages, we first implement a simple HLS (high level synthesis) benchmark suite, the CHO benchmark set [15], as the kernel functions to FPGA board. HLS is an automated design process that helps designers convert an algorithmic description to RTL (register transfer level) description [18]. Each of the CHO benchmarks is a single kernel and has a test-bench that is part of the host program, and data will be read/written on the host end. With such implementation, we can better understand the way that external I/O interfaces between host and device(s) works.

Based on the above experiments, we could have an overall understanding to OpenCL and FPGA workflow. In order to maximize the advantage of FPGA accelerator, we then focus on the topic of design space exploration, and try to find out the appropriate scale of hardware resources that lead to an optimal performance.

Afterwards, we balance and improve the performance of the benchmarks by overlapping their hardware implementation or data migration part with software implementation. To reach the optimal result, we keep the total data size unchanged but iteratively adjust the size of data sets that sent into software (CPU) and hardware (FPGA) respectively. We also put forward a way to determine the possibility and degree of improving a program's performance with such framework ahead of implementing OpenCL program.

The rest of this thesis is organized as follows. Chapter 2 reviews the background and previous work. Chapter 3 investigates the Altera based FPGA design space exploration. The implementation and results of our suite of benchmarks for FPGA is also described in chapter 3. Chapter 4 focuses on optimizing programs' performance with CPU-FPGA heterogeneous platform. Finally, Chapter 5 sums up our analysis and addresses the future work we are interested in.

1.5. Experimental Setup

For our experimental platform, we use Terasic DE5-Net Development Kit empowered with the top-of-the-line Stratix V GX FPGA board. DE5 is integrated through version 3.0 PCI interface with a motherboard with AMD A10 processor and 16GB DDR3 memory with a transfer bandwidth of a maximum of 12.5 Gbps.

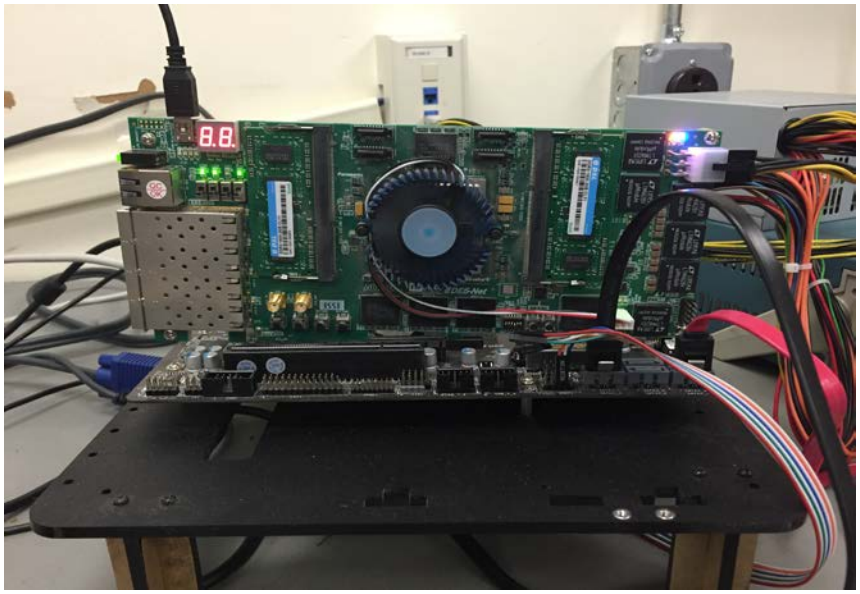


Figure 1.5.1 AMD-Altera Platform.

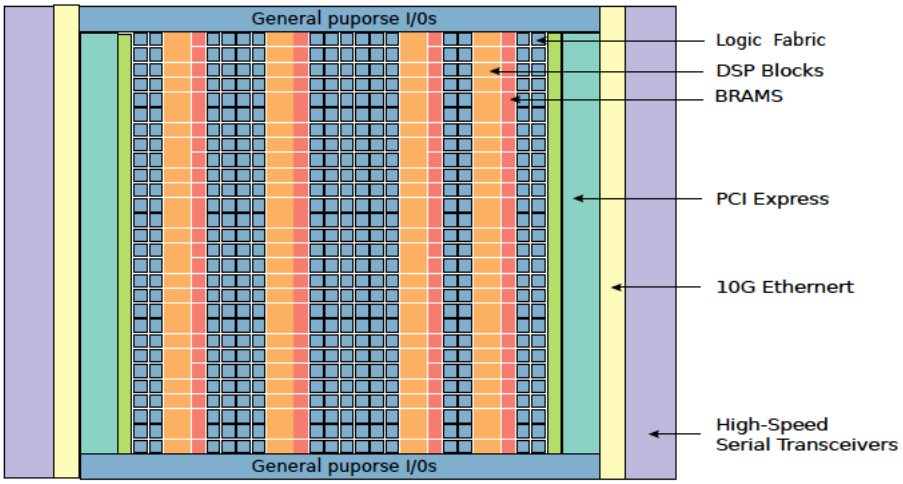


Figure 1.5.2 FPGA's Architecture [15].

We sample the total AC power consumption data of the entire motherboard including DE5 board using digital multimeter.

We implement three benchmarks to compare the performance improvement of hardware implementations under several groups of hardware resource attributes with pure software implementations. And four benchmarks are designed in CPU-FPGA symbiosis computing part. All the experiments in this

thesis are achieved with DE5 and Altera OpenCL, see figure 1.5.1, and figure 1.5.2 gives the architeture of the FPGA.

2. Background

2.1. Related Work

This section mainly discusses the related work on FPAG accelerators.

As a technology that can automatically translate behavioral level descriptions into RTL (register-transfer level) descriptions, HLS (High-Level Synthesis) frameworks are of a great deal of interest.

Several previous works presented benchmarking of HLS technology. In [14], Y. Hara *et al*. proposed a standard benchmark for C-based high-level synthesis, named CHStone, for HLS researchers to evaluate the effectiveness of their acceleration techniques. They include 12 programs, which were selected from several different domains, in their suite and analyze the source-level features, resource utilization, and sensitivity to resource constrains. However, their implementation is based on a state-of-the-art HLS tool called eXCite instead of OpenCL. Although HLS approaches realize an automated conversion from algorithmic description to RTL (register transfer level) description, this tool is still designed for hardware engineer. HLS tool targets to implement IP (Intellectual Property) cores with C-based language, and then maps those IP cores to an RTL architecture over several clock cycles, which makes it not preferable for implementing large system. Additionally, instead of using pipeline parallelism, HLS tools achieve parallelism by scheduling independent operations in same clock. It is because of the limitation of C-based language – pipeline descriptions are implicit [23]. As a result, such approaches may not make best use of FPGA.

Recently, Ndu *et al.* propose a suite of OpenCL port of the commonly used benchmark, which extends CHStone benchmark suite and named CHO, for qualitative evaluation [15]. Their benchmark suite targets OpenCL 1.0 so that it is compliant to any OpenCL compiler version. The test-bench of each application, which is implemented on the host side, enables them to examine the external I/O interfacing and its bandwidth. They successfully synthesized 4 test functions out of 12, and evaluated how a program's source-level and IR (intermediate representation) -level characteristics impact the resource utilization of hardware implementation as well as the program's performance. The results provide a rough indication of the complexity of each application on FPGA.

Meswani *et al* put forward an implemental methodology that helps researchers explore the optimization opportunities in [17]. This paper raises a creative concept, idiom, which can help programmers determine whether a given application benefits from a certain accelerator. Idioms, or the pattern of computation and memory access, could be useful for describing how a particular application, or a sort of applications can be potentially optimized or sped up. They focus on stream and gather/scatter idioms, and capture the indicative parameters from the source code of a set of benchmarks to determine the performance. The test result shows that the optimization of run time is largely dependent on the data size, the data migration cost and data footprint. A binary instrumentation tool named PEBIL is used to capture the data footprint at the runtime, and the PMaC performance-modeling framework is chosen to provide predictions of large scale HPC application performance. Based on the experimental data, they derived a prediction model concerned about the

memory access time for both GPU and FPGA applications, which can accurately predict each idiom's run time with error rate between 0.3% and 17.3%.

A prediction model only concerning gather/scatter idioms from [18] can achieve more accurate prediction with error rate that is less than 10% for gather/scatter idiom. Their implementation is done on a HC-1 platform. They firstly compile the SGBench benchmark on the host side, and then run the majority executions on host side, while the loops that contain G/S computations run on the FPGAs. The measurements are taken both on the host processor and FPGA coprocessor, which illustrates the relation between the size of address range and memory bandwidth. It is demonstrated that their fine-grained model for FPGA has less than 8.6% absolute error, and further identify that more than 100 instances of G/S idiom are not worth porting, which largely saves the programming time and decreases the likelihood of inaccurate result.

The compelling performance offered by FPGA leads researchers and engineers apply FPGA to diverse fields. In [16], W. Vanderbauwhede *et al.* explore a way to implement a document-filtering algorithm, which is widely used in database and served as a filtering mechanism, with FPGA. They demonstrate the tremendous advantages provided by FPGAs in both performance and power for document filtering is that FPGA implementation for unstructured search achieved an overwhelming speedup than the standard implementation. For the three IR collections they tested, FPGA implementation gains about 10x to 20x than the reference implementation.

Such implementation was also illustrated in [3], Doris *et al.* take advantage of Altera OpenCL and created a kernel to perform the filtering function, and then improve the memory bandwidth efficiency with *memory coalescing*. They also

take advantage of a Bloom Filter to optimize the memory bandwidth, which is caused by read access. Bloom Filter is like a hash map, and it performs a pre-search function that can reduce the times of memory access to the *profileWts* array. Their result shows the performance comparison among Multi-Core CPU, GPU and FPGA, and FPGA outperforms CPU and GPU by a factor of 5.25x and 5.5x respectively when considering the power of external bridge chip and memory power.

3. Design Space Exploration

OpenCL is designed as a software-friendly high level language that enables programing on hardware accelerator. However, it is still a hardware-targeted description. One primary factor that distinguishes OpenCL from software programming language like C or C++ is that OpenCL support statements of setting hardware attributes. Such feature helps programmer exploit the hardware resource under a custom manner. This chapter addresses the problem of optimizing the accelerator's performance with appropriate hardware resource attributes.

3.1. Altera OpenCL Optimization

Altera OpenCL provides programmers several ways of optimizing the kernel function to achieve high performance. To achieve higher performance, it is necessary to optimize the kernel with either automated or manual optimization. In this section, we leverage the pros and cons of those two optimization approaches.

3.1.1. AOC Resource-driven Optimization

A straightforward and simple way to optimize a kernel function is invoking the resource-driven optimizer by applying –O3 flag in the command when the programmer begin to generate kernel file. Resource-driven optimization is a feature of AOC (Altera OpenCL Compiler). By invoking the optimizer, AOC will evaluate the code and determine the possible sharing degree of work-items by analysing various combinations of number of CUs, SIMDs, loop unrolling factor and number of shared resources under the constraints of available hardware

resources and memory bandwidth to identify the optimal choice of these values in terms of work-items executed per second [20].

One of the most compelling benefit of resource-driven optimization is its usage of hardware resource will always conform to the fitting requirement. The default threshold of estimated utilization is 85%, which ensures no usage overflow error occurs while compiling a kernel into the FPGAs. According to AOC Compilation Flow [20], the duration of compilation may last several hours. Even a very simple kernel functione.g., vector addition, takes about 3 hours to compile and generate an Altera Offline Compiler Executable File (*.aocx)* file. On the other hand, unless syntax errors occur, the usage overflow cannot be detected by the compiler at very beginning of the compilation, but in the fitting step. Therefore, it is time-wasting if such error exists.

On the other hand, AOC resource-driven optimization also has several limitations. First, the control flow analyses assume values of kernel arguments are unknown when compiling the kernel. For example, the optimizer assumes that loops with unknown bounds iterate 1024 times, which is meaningless most of the time. Second, the performance estimation might not capture accurately the maximum operating frequency that the hardware compiler achieves. Because all optimizations take place ahead of hardware compilation occurs. Besides, when the number of CUs increases in order to achieve higher throughput, the hardware resource usage also increases due to the increasing frequent global memory accesses.

3.1.2. Performance Optimization Based on Kernel Attributes

Another method to optimize a kernel function is manually setting hardware attributes. To begin with, we need first get better understanding of the hardware attributes the *clEnqueueNDRangeKernel* function [7-9],

cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
cl_kernel kernel,
cl_uint work_dim,
const size_t *global_work_offset,
const size_t *global_work_size,
const size_t *local_work_size,
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)

AOCL supports specifying the following attributes:

reqd_work_group_size(x, y, z): The work-group size that must be used as the local_work_size argument for the kernel, which equals to the number of work-items (x*y*z) that form a work-group.

num_compute_units: Specify the number of CUs that used for processing the kernel. This is used for increasing the throughputs by distributing the work-groups within a kernel across the specified number of CUs, but at the expense of increasing global memory bandwidth among the CUs.

num_simd_work_items: the data path within a compute unit is replicated to increase throughput and can also lead to memory coalescing. For AOCL to implement a SIMD data path, the value of num_simd_work_items must evenly divide the value specified for reqd_work_group_size, and also be a power of 2.

max_unroll_loops: Loop unrolling decreases the number of iterations the AOC executes, at the expense of increased hardware resource consumption. This can be overridden for individual loops using #pragma unroll.

To better investigate the function of kernel attributes, we should first understand OpenCL's platform model as well as the execution model of OpenCL kernel.

OpenCL kernels are defined as *NDRange* style, which refers an N-dimensional (expressed as dim) index space. Once a kernel is requested by the host for execution, its index space is specified. With the command *get_global_id(dim)* or *get_local_id(dim)*, a kernel's instance will be executed at a point indicated by an index within the index space. Such kernel instance is referred to OpenCL language as *work-item*.

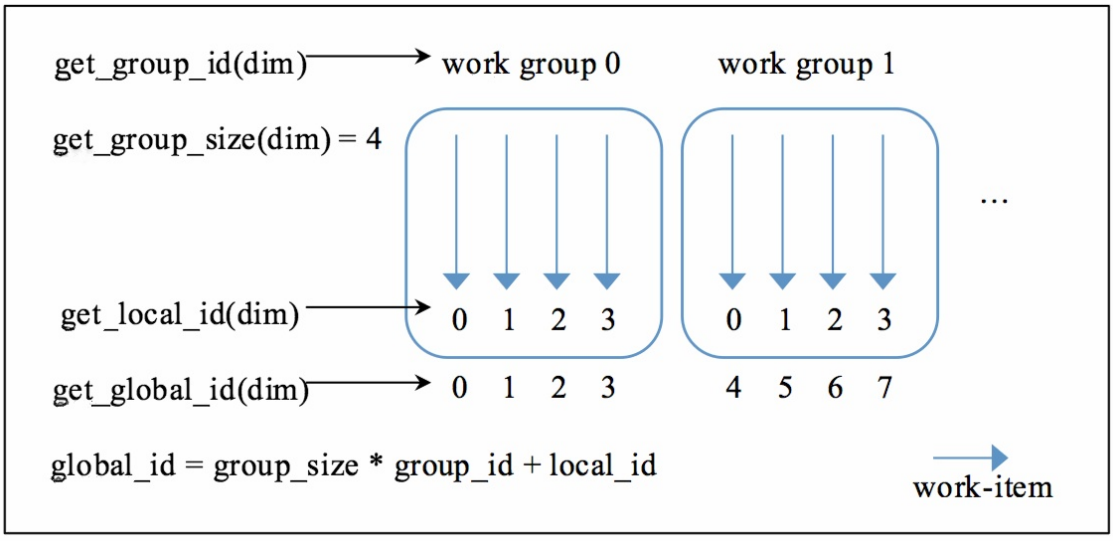


Figure 3.1.2.1 Kernel Execution Model

With this understanding of the kernel execution model, we then evaluate each of the kernel attributes:

1) Loop Unrolling

The loop-unrolling factor will take effect if the kernel program contains any loops. Generally, there will be two cases if any loop exists in the program: 1) a loop with certain bounds, and 2) a loop with unknown bounds. The first case is straightforward, but the second case may lead to insufficient loop unrolling that AOC assumes the loop iterated 1024 times. In this situation, the best solution is to manually set the unroll pragma to a reasonable number, which can override the AOC optimizer's assumption.

2) Kernel Vectorization

Kernel vectorization, which is realized by expanding SIMD, is prior to introducing additional CUs. Considering the compiling rules that the value of *reqd_work_group_size* attribute must evenly divided by the value of *num_simd_work_items*, these two attributes should be set accordingly. The default value of *reqd_work_group_size* is 256, which is the maximum work-group size. However, this default number is not the optimal value in all test cases.

3) Compute Unit Replication

Replicating additional CUs is a way to increase throughput. But at the same time, the times of global memory accesses will be increased. One of key factor of Altera OpenCL Optimization is making efficient use of memory hierarchy. When multiple CUs are generated, consequently more CUs will compete for global memory bandwidth.

AOC optimizer defines higher priority to loop-unrolling and SIMD vectorization than increasing compute unit for the hardware resource reason.

3.2. Evaluation

The result of *Vector Multiplication* indicates that barely running AOC optimizer for a kernel function cannot always achieve its optimal performance. But repeatedly attempting various combinations of kernel attributes to get higher performance is not practical in real implementations. Therefore, exploring a way to identify the proper values of kernel attributes of a program to achieve its optimal performance is crucial for engineers and programmers.

A straightforward concept is that a kernel can be sped up by introducing more hardware resources. When only considering a single hardware attribute to speed-up the kernel, programmers can maximize its value only under constrain of fitting requirement.

But maximizing all the kernel attributes is not possible due to the usage constraints. In most of the cases, we need to leverage among all the attributes and determine an optimal solution.

3.2.1. Double Floating Addition

We first implement CHO benchmark set [15], which is an OpenCL version of CHStone HLS benchmark, to exploring the resource-driven optimization.

The first test we evaluate is an IEC/IEEE-standard double-precession floating-point addition using 64-bits integers. We programmed *dfadd* kernel twice; we firstly compile the kernel directly, and then invoke the command with –O3 optimization. Compilation time for those two ways are nearly the same, but we can find the first stage hardware usage estimation differs a lot, which shows in Table 3.2.1.1. Unfortunately, due to the small data scale, there is no obvious difference in kernel execution time.

Table 3.2.1.1 Usage summary and kernel execution time for dfadd with data size 46.

	Logic utilization	Dedicated logic registers	Memory blocks	DSP blocks	Execution time/ms
Without optimization	19%	8%	16%	0%	0.131
With optimization	70%	26%	14%	0%	0.129

3.2.2. Vector Multiplication

To better analyze the difference before and after applying resource-driven optimization, we implement a vector multiplication application as the second benchmark. The function takes two vectors as inputs, and returns a vector as the output, which stores the product of each two corresponding elements in the two input vectors. The data type is integer, and the vector size is 10^8 $elements$.

We compile the vector-multiplication kernel three times:

1. Set no hardware attribute, and compile the kernel function without any optimization approaching;
2. Set no hardware attribute, but invoke the AOC resource-driven optimizer when compile the kernel;
3. Manually set hardware attributes in the kernel function, and invoke the resource-driven optimizer when compile it.

Each of those three ways of compilation takes around two hours to complete, which makes us believe that applying resource-driven optimization will not increase the overall compilation time. The only difference among those three ways is that the second method spends a few seconds on evaluating several

combinations of hardware attributes for preforming optimizations. However, from the estimated resource usage summary, we can find distinct variation.

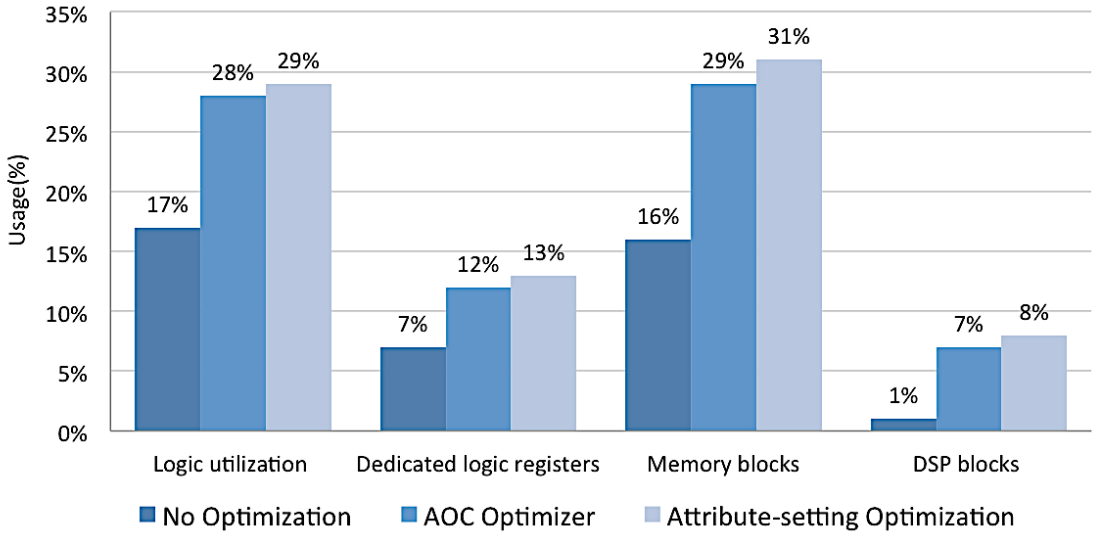


Figure 3.2.2.1 Usage Summary of Vector Multiplication

Figure 3.2.2.1 provides the hardware usage summary of the same kernel function compiling in three different ways. Such differences produced by the varying values of kernel attributes:

Table 3.2.2.1 Kernel Attributes of Vector Multiplication

	# CUs	# SIMDs	Local work size
No Optimization	1	1	1
AOC Optimizer	9	2	1
Attribute-setting	10	2	100

All the default kernel attributes are 1, which in the most cases shows better performance than CPU does. But it still has much scope to be improved. To achieve higher throughputs, the AOC optimizer will assign more hardware resources to the kernel. The addition hardware space would be assigned to those

frequently executed work-items to allow them to complete the operations in fewer cycles.

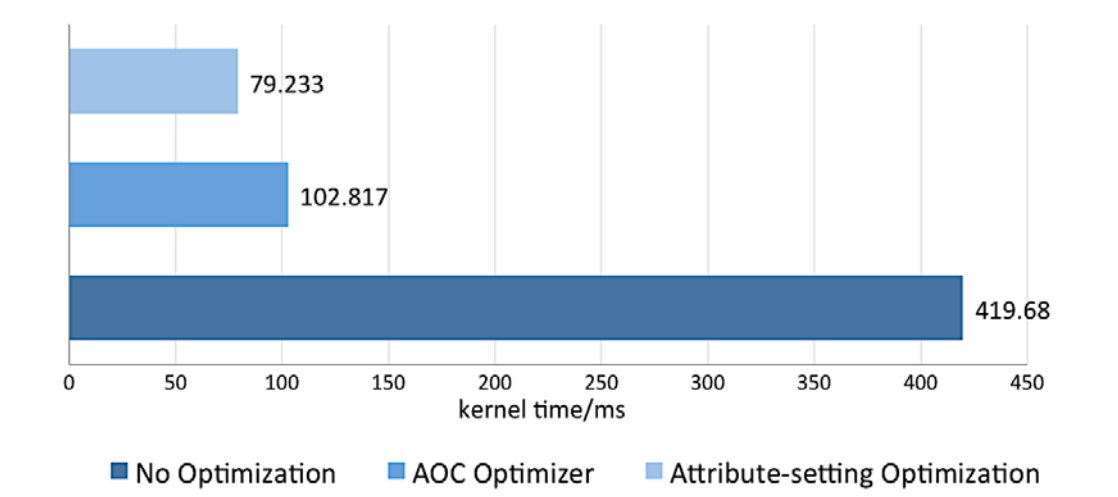


Figure 3.2.2.2 Kernel Execution Time.

Figure 3.2.2.2 gives the kernel's execution time under various optimization methods. From the result, we can easily find that performing optimization, either invoking the AOC optimizer or manually setting the kernel attributes, can speed-up the kernel, but at the expense of occupying more hardware resources.

In this test case, manually setting kernel attributes can achieve 79.233ms to execute multiplication of two vectors of size 10^8, which runtime improves 97.6% over pure CPU execution, 81.12% over non-optimizing FPGA execution, and 22.9% over AOC-optimizing FPGA execution.

3.3. Kernel Optimization Analysis

Vector Multiplication benchmark is tested in this section to help us understand how the kernel attributes cooperate with each other and how each attribute affect the performance. It is defined as:

$$c[i] = a[i] \cdot b[i]$$

where both a and b are lists of numbers.

AOC optimizer evaluates the kernel and than compiles it across 9 CUs with a 2 lane-wide SIMD. The pure kernel runtime when executing 100 million floating-point numbers is 100.8ms. Then we fix all the other attributes unchanged, but only set *reqd_work_group_size* to 100, the kernel runtime decreases to 79.6ms, and performance improves by over 21%.

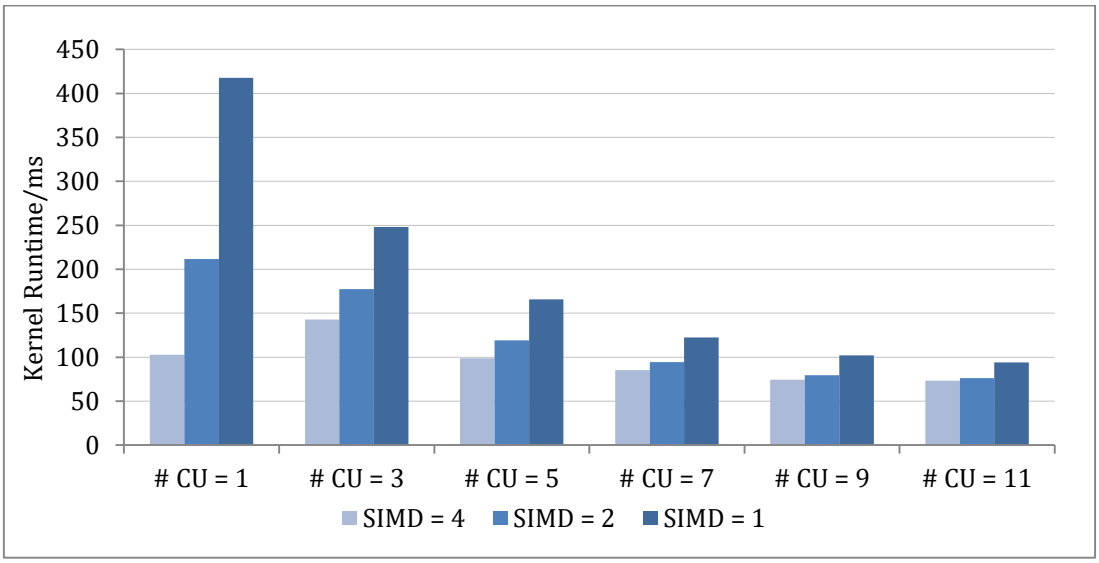


Figure 3.3.1 Pure Kernel Execution Time vs. number of CUs and SIMD.

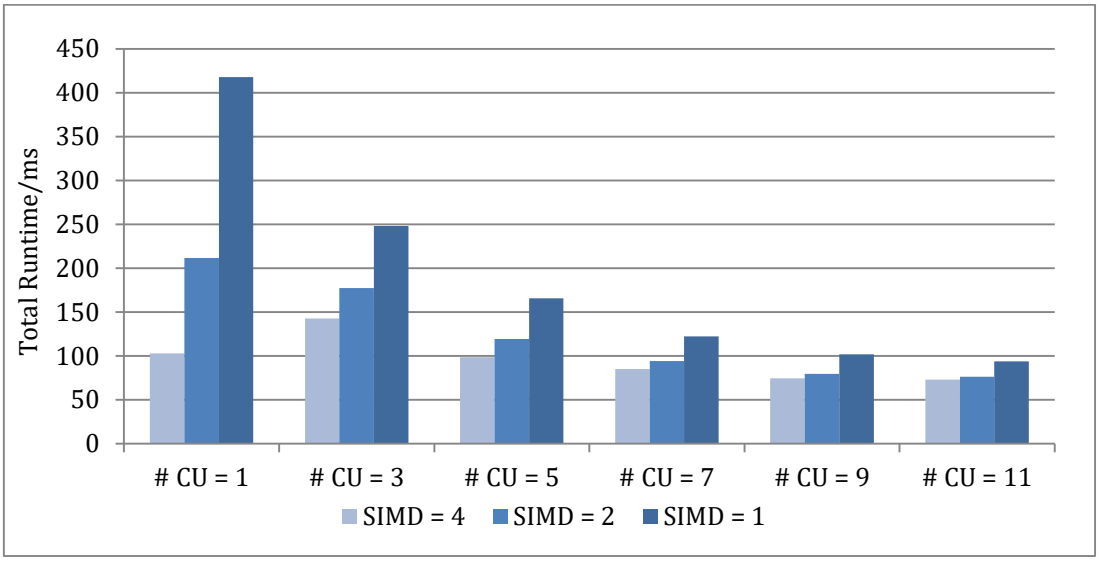


Figure 3.3.2 Total Execution Time vs. number of CUs and SIMD

Figure 3.3.1 gives the variation in pure kernel execution time versus the number of CUs and the number of duplicated SIMDs, and Figure 3.3.2 gives the variation in total runtime, which includes the host-to-device and device-to-host data transform time.

We can easily find there is a descending trend in both FPGA kernel runtime and total runtime as the number of either CU or SIMD decreasing. This tendency is reasonable since the more hardware resources are generated, the more operations can be performed concurrently.

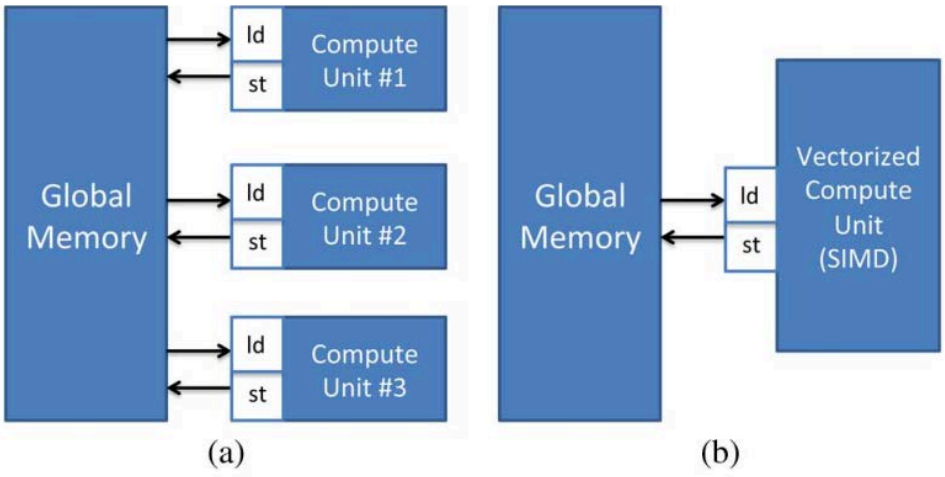


Figure 3.3.3 AOC optimization techniques to increase parallelism. (a) CU replication. (b) Kernel vectorization in one CU [10].

For Altera OpenCL, we can see the difference between two optimization techniques. Every time we introduce one more Compute Unit for a kernel, we will duplicate the number of work-group. The scheduler then dispatches a work-group to the additional Compute Unit, and more work-items will be executed simultaneously. Setting SIMD attribute to larger number can also achieve higher performance. Whereas the CU attribute defines the number of work-groups that can be scheduled, SIMD attribute describes the amount of paralleled executions

performed by a single work-group. Setting SIMD attribute can also defined as kernel vectorization, which increases the datapath of a Compute Unit and enables work-items to execute in SIMD fashion [23]. Although either CU replication or SIMD vectorization can improve a kernel's performance by exploiting more hardware resources, SIMD vectorization should be applied ahead of CU replication.

To clearly show the priority of replicating SIMD we compare the performance and hardware resource usage of following CU-SIMD combination, and show the results in Table 3.3.1.

Table 3.3.1 Performance and Resource Usage Comparison

(CU, SIMD)	Logic Utilization	Dedicated Logic Registers	Block Memory Bits	Total DSP Blocks	FPGA Runtime (ms)
(1, 2)	20%	7.14×10^4	3%	4	215.675
(2, 1)	20%	7.11×10^4	5%	4	365.650
(1, 4)	20%	7.17×10^4	3%	8	104.147
(4, 1)	22%	7.96×10^4	6%	8	187.975

There are two pairs of test results listed in Table 3.3.1. The first pair keeps one attribute as default value 1, and double the other attribute; the second pair also keeps one attribute as default value and increases the other attribute threefold. Both of them give us a clear-cut difference between CU replication and SIMD vectorization in kernel's performance: the runtime of two SIMD vectorization implementation reduced 41% and 44.5% than CU replication respectively, while hardware resource utilization almost remains the same or even less.

Therefore, implementing kernel vectorization can achieve better performance with more efficient design space.

A series of more intuitive of the hardware resource-runtime Pareto frontier are shown in Figure3.3.4, Figure 3.3.5 and Figure 3.3.6.

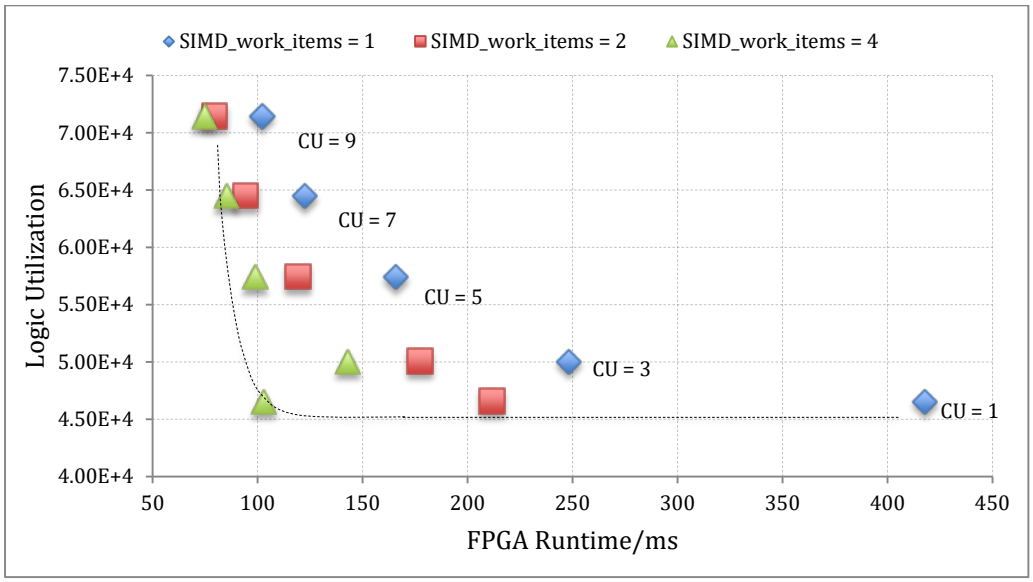


Figure 3.3.4 Logic Utilization-Runtime Pareto Frontier.

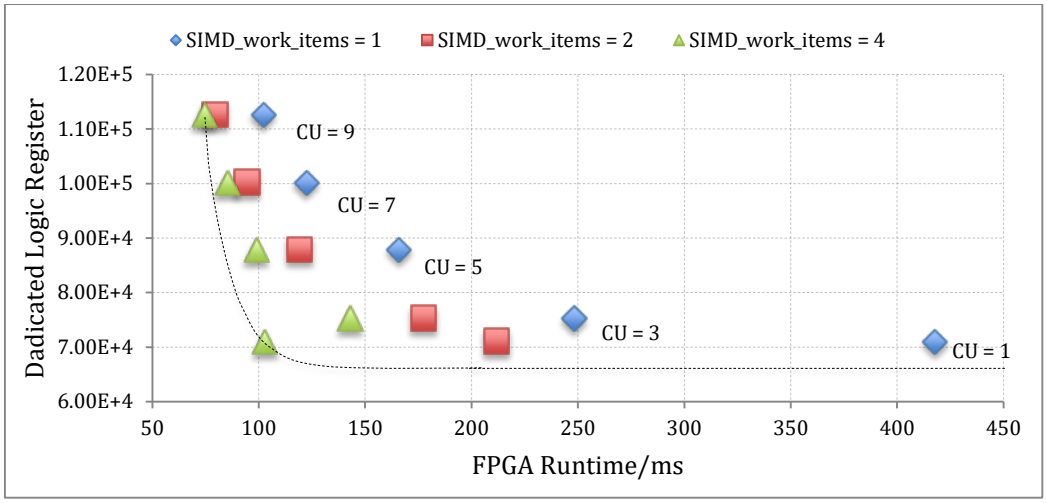


Figure 3.3.5 Dedicate Logic Register-Runtime Pareto Frontiers.

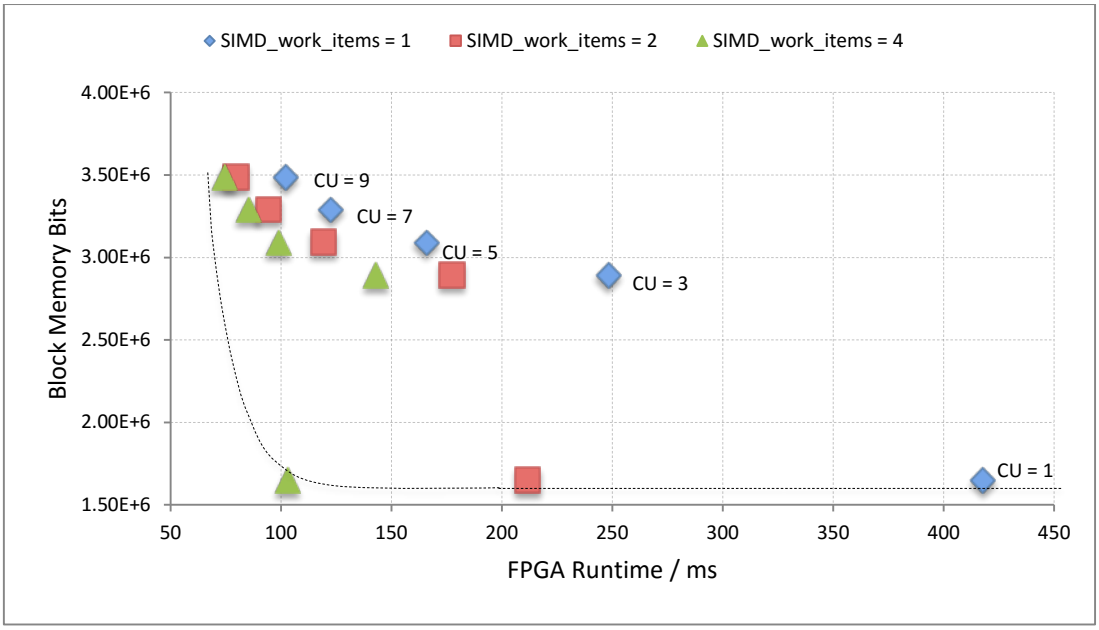


Figure 3.3.6 Block Memory Bits-Runtime Pareto Frontier.

The previous three plots show the importance of replicating SIMD_work_items: as we double the SIMD_work_items attribute, the runtime decreases a lot, but there is no obvious variation among Logic Units, Dedicated Logic Register and Block Memory Bits. While we increase the number of Compute Unit, as we can see, the hardware resources grow exponentially. The runtime of using 1 CU and 4 vector lanes is a little longer than using 9 CUs and 4 vector lanes, which are 102ms and 74ms respectively.

However, not all the hardware resources only change with CU numbers, another crucial hardware resource, like DSP blocks, is proportional to both CU and SIMD attributes. According to our test results:

Table 3.3.2The Impact of SIMD and CU Attributes on DSP Blocks.

	Compute Units				
simd work items	1	3	5	7	9
1	2	6	10	14	18
2	4	12	20	28	36
4	8	24	40	56	72

We can easily find out the connection among SIMD, CU attributes and the number of "active" DSP plocks:

$$DSP\ Blocks = 2 \times CU \times SIMD$$

DSP blocks in FPGA consist of multipliers and many other required functions for processing high-precision DSP applications. Thus, when the application targets on processing complicate DSP functions, both the number of compute unit and simd_work_items should be considered to increase.

Overall, consider both the program's runtime and hardware resource usage, replicating SIMD attribute is prior to introducing more CUs when optimize a program. Otherwise, the inefficient memory access pattern will obstruct achieving optimal performance.

4. Optimize FPGA-CPU Symbiosis

As an accelerator, FPGA exhibits superior parallel execution ability then traditional computer processor. However, performing as an accelerator, FPGA cannot work independently – it must be instruct by a host, which is, in this case, the CPU. The workflow requires kernel functions to be instantiated by host program first, and then be sent to FPGA board. After FPGA executions are completed, all the result must be copied back to the host side. Such flow path could worsen program's runtime if not optimized correctly.

After we examined Altera OpenCL workflow, we find that CPU is idle while FPGA is processing. To maximize the performance of whole applications, we designed a CPU-FPGA overlapping platform.

4.1. CPU-FPGA Overlapping Accelerator

In this section, we leverage the advantage of combining FPGA and CPU to implement OpenCL kernels for several operations, in particular, the operations that can be executed in a reductive fashion. We keep the size of inputs unchanged, which is 10^8, while iteratively reduce the percentage of data that will be executed on FPGA from 100% to 0 by 10%, and identify the optimal combination of CPU and FPGA.

According to AOC programming flow, overlapping CPU-FPGA computations can be generated either after the host side (CPU) call *read_event*, which function as copying data from host to device, or set kernel arguments. Once the host successfully sends data to FPGA and sets kernel arguments, and CPU will

become idle until all the tasks on FPGA are completed. We try to make use of that period and find out an appropriate amount of operations to load to CPU.

We designed five benchmarks to help us evaluate the possibility and overlap degree of using CPU-FPGA overlapping framework. Two factors are considered when we choose our benchmarks: data migration intensity and computation complexity.

4.1.1. Vector Multiplication

For the first step, we illustrate how the computations of FPGA and CPU be overlapped using a simple vector multiplication kernel.

The implementation of vector multiplication is taking two lists of the same size as inputs and adding each pair of the elements, then returning a single vector as the output.

$$c[i] = a[i] \cdot b[i]$$

Here the name, *vector,* is just the container we used in the kernel function, but actually this is a scalar multiplication.

4.1.2. Vector Dot Product

After creating the general kernel for vector operations, we proceed to consider the acceleration cases of exploiting parallel reduction on data intensive computing operations. The second test is taking a dot production of two vectors, in which the inputs are still two vectors while the output result changed from a vector to a single number. Dot product is defined as

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

,where $\boldsymbol{a}$ and $\boldsymbol{b}$ are vectors.

In place of linearly fetching each number from the vector and executing the operation, the parallel algorithm implemented on FPGA can be expressed as follows: the multiplications will be executed across all the initialized work items concurrently and all the result will be stored in a temporary vector, after all the multiplications are finished, the devices side will sum those products in parallel and then return the final result as the output.

4.1.3. $\boldsymbol{L^p}$ *Space*

The third application we tested is $\boldsymbol{L^p}$ *Space*. In finite dimensions, the length of a vector $\boldsymbol{a} = \{a_1, a_2, \dots, a_n\}$ in the n-dimensional real vector space $\boldsymbol{R}^n$ is given by the Euclidean norm:

$$\|\boldsymbol{a}\| = \sqrt[2]{a_1^2 + a_2^2 + \cdots + a_n^2}$$

However, the Euclidean distance may not sufficiently describe an actual distance in many real-life situations, while p-norms can fill this gap and widely applied in many parts of mathematics, engineering and computer science. A $\boldsymbol{L^p}$ norm of vector $\boldsymbol{a}$ is defined as

$$\|\boldsymbol{a}\|_p = (|a_1|^p + |a_2|^p + \cdots + |a_n|^p)^{1/p}$$

We test the performances of $\boldsymbol{L^p}$ Norm with different p values. Instead of sequentially applying square operation on each element, in our FPGA application, all the instantiated work-items do their work concurrently on each cycle.

4.1.4. 1D Convolution

Convolution is one of the most important concepts in Fourier Theory that can be defined on groups other than Euclidean Space. It takes two 1D vectors as

inputs - one of them is an “input signal” or “image”, the other one works as a “filter”- and producing a vector as an “output signal” or “image”. Let’s call our inputs $\boldsymbol{a}$ and $\boldsymbol{f}$, and $\boldsymbol{b}$ as the output, so the convolution of $\boldsymbol{a}$ and $\boldsymbol{f}$ is:

$$b[i] = \boldsymbol{a} * \boldsymbol{f}[i] = \sum_{j=1}^{N} a[j] f\left[i - j + \frac{N}{2}\right]$$

We choose a filter with size $N = 4$, which is much smaller than the input signal. The FPGA implementation unloops the convolution so multiple filters can operate on the signal simultaneously.

4.1.5. Matrix Multiplication

Matrix Operations serves an important role in quantum mechanics, graph theory, and all throughout fields. It is a way of organizing real-life data and hence the calculations can become very easy and practical to tackle using matrix operations. Due to this reason, *Matrix Multiplication* is implemented.

The inputs of matrix multiplication are a pair of matrices, and the output is another matrix. In mathematics, its definition is

$$\boldsymbol{a} \times \boldsymbol{b} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{pmatrix}$$

where $c_{ij} = \sum_{\boldsymbol{k=0}}^{\boldsymbol{n}} a_{ik} * b_{kj}$.

The process of *Matrix Multiplication* can be divided into three pipelined stages: 1) Copying ith row of vector $\boldsymbol{a}$ and jth column of $\boldsymbol{b}$ from host to FPGA; 2) FPGA start operating on the loaded vector, which contributes to c_{ij}; 3) writing data back from FPGA to CPU. Unlike the above benchmarks that copy all the data from CPU to FPGA before FPGA computation begins, *Matrix Multiplication* copies

the input matrix block by block, which overlaps the data migration with FPGA computation.

4.2. OpenCL Data Migration on PCI Express

In most of the cases, computing a function directly in hardware is overwhelmingly efficient than in software. However, merely considering the pure execution time of a program is not enough for us to choose FPGA platform, since data migration between host and device can sometimes be very expensive, and has a direct bearing on a program's time efficiency.

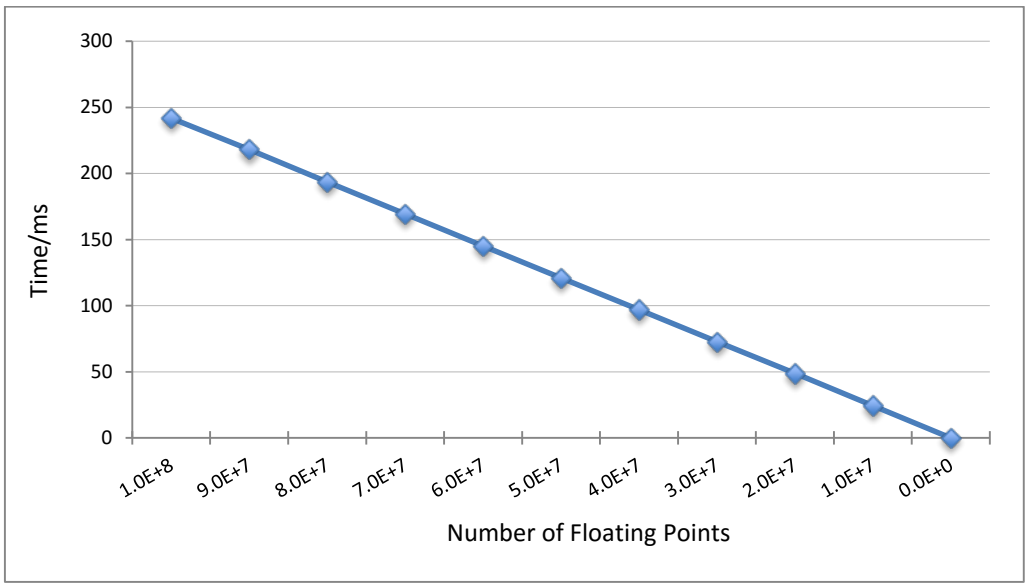


Figure 4.2.1 Data Migration Time for Floating-point

Data migration time is hard to optimize since it is determined by the speed of the PCI Express bus. However, this limitation makes it easy to be predicted. Altera OpenCL provides users with build-in statements, *write_event* and *read_event*, which serve as data-copy methods. We measure the data migration time for our platform by transferring different number of floating points between CPU and FPGA with Altera OpenCL API.

From Figure 4.2.1, we can observe that the data transfer time is linear to the size of transferred data. Therefore, it is possible for programmer to calculate the estimate data migration time for a certain amount of data, and then evaluate the possibility of taking advantage of CPU-FPGA heterogeneous platform.

For our experimental platform, the migration speed when apply *writeBuffer* statement is 1.65GB/s. When there are N single floating-point elements to be transferred between host and device, the data migration time will be

$$t = \frac{N \times 4\,Byte}{1.65 \times 10^9 Byte/s}$$

Similarly, programmer can estimate the data migration time for a certain type of input data set, and further consider the computation method as well as the platform applied to their applications.

4.3. Methodology

Based on Amdahl's Law, the speedup of the whole task can be described by portion of computations executed on FPGA x (%) and the speed-up, p, attained from parallelizing in FPGA:

$$Speedup = \frac{1}{1 - x + {}^{x}/_{p}}$$

and when we overlap CPU computations with FPGA computations, the speedup becomes:

$$Speedup = \frac{1}{\max(1 - x, {}^{x}/_{p})}$$

Consider the theoretical speedup of the whole program is limited by:

$$lim_{p \to \infty} Speedup = \frac{1}{1 - x}$$

, which we named as the *Boundary Ratio*.

In our experiments, the *Speedup* of the enhanced faction can be expressed in following two ways based on the overlapping method:

$$Speedup = \frac{1}{\max(1-x, {}^{x}/_{p})} = \frac{t_{cpu}^{N}}{\max(t_{cpu}^{(N-n)}, t_{FPGA}^{(n)}) + t_{migration}^{(n)}}$$

or

$$Speedup = \frac{1}{\max(1-x, {}^{x}/_{p})} = \frac{t_{cpu}^{N}}{\max(t_{cpu}^{(N-n)}, t_{migration}^{(n)}) + t_{FPGA}^{(n)}}$$

where

N is total number of elements to be processed;

n is number of elements to be processed on FPGA;

$t_{cpu}^{(N)}$ is the CPU runtime without using FPGA platform;

$t_{cpu}^{(N-n)}$ is the CPU runtime while applying CPU-FPGA overlapping computation;

$t_{FPGA}^{(n)}$ is the FPGA runtime while applying CPU-FPGA overlapping computation;

$t_{migration}^{(n)}$ is data migration time.

Then the theoretical optimal speedup should be:

$$Boundary\ Ratio = lim_{p\to\infty} Speedup = \frac{t_{cpu}^{N}}{t_{migration}^{(N)}}$$

as all the data are executed on FPGA.

Since generally FPGA has much better computational capability than CPU, the data migration time could be the key factor to determine whether a program can be speed up with CPU-FPGA overlapping computation. If $Speedup \leq 1$, it means

the program cannot be further optimized, otherwise it can be speeded up to some extent.

When n($n \leq N$) elements will be processed on FPGA, if

$$t_{migration}^{(N)} \leq t_{cpu}^{(N)}$$

then:

$$t_{migration}^{(n)} = \frac{n}{N} t_{migration}^{(N)} \leq t_{migration}^{(N)} \leq t_{cpu}^{(N)}$$

$$t_{FPGA}^{(n)} \leq t_{FPGA}^{(N)} < t_{cpu}^{(N)}$$

For illustration, we consider the special case when programs with linear runtime complexity are executed as our experimental objects and evaluate their Speedup degree. However, programmers can further evaluate the Speedup according to their program's time complexity.

If the program execution time is linear to the input data size, we can model the CPU and FPGA time as follows:

$$t_{cpu}^{(n)} = \frac{n}{N} t_{cpu}^{(N)}$$

$$t_{FPGA}^{(n)} = \frac{n}{N} t_{FPGA}^{(N)}$$

Then we consider the two possible situations. The first case is when $t_{migration}^{(n)} > t_{FPGA}^{(n)}(1)$,

$$Speedup = \frac{1}{1 - x + {}^{x}/_{p}} = \frac{t_{cpu}^{N}}{(\max(t_{cpu}^{(N-n)}, t_{FPGA}^{(n)}) + t_{migration}^{(n)})}$$

$$= \frac{t_{cpu}^{N}}{(\max(\frac{N-n}{N} t_{cpu}^{(N)}, \frac{n}{N} t_{FPGA}^{(N)}) + \frac{n}{N} t_{migration}^{(N)})}$$

For this case, since $t_{FPGA}^{(N)}$ could be always less than $t_{cpu}^{(N)}$, and programmer have great chance to further optimize FPGA computation, as long as $t_{migration}^{(N)} < t_{CPU}^{(N)}$, we can prove that:

$$Speedup > \frac{t_{cpu}^{N}}{(\max(\frac{N-n}{N} t_{cpu}^{(N)}, \frac{n}{N} t_{FPGA}^{(N)}) + \frac{n}{N} t_{CPU}^{(N)})}$$

$$Speedup > \begin{cases} \dfrac{t_{cpu}^{N}}{\frac{N-n}{N} t_{cpu}^{(N)} + \frac{n}{N} t_{CPU}^{(N)}} = 1 \\ \dfrac{t_{cpu}^{N}}{\frac{n}{N} t_{FPGA}^{(N)} + \frac{n}{N} t_{CPU}^{(N)}} = \dfrac{N}{n} \dfrac{t_{FPGA}^{(N)}}{t_{CPU}^{(N)}} + \dfrac{N}{n} > 1 \end{cases}$$

Therefore, no matter how we choose n, the program's performance can always be improved.

Then we consider case (2), where speedup is expressed as:

$$Speedup = \frac{t_{cpu}^{N}}{(\max(\frac{N-n}{N} t_{cpu}^{(N)}, \frac{n}{N} t_{migration}^{(N)}) + \frac{n}{N} t_{FPGA}^{(N)})}$$

Without compiling and fitting the OpenCL program onto FPGA, if we could guarantee $t_{cpu}^{(N)} > t_{migration}^{(N)}$, we can also ensure $Speedup > 1$ by choosing appropriate number of data, n. As long as the $Speedup$ is greater than 1, the application can be further optimized with FPGA-CPU overlapping computation.

Therefore, by only comparing the data migration time with CPU-only runtime, we can predict the possibility of optimizing an application with CPU-FPGA platform.

4.4. Experiments

4.4.1. Vector Multiplication

Vector Multiplication is a typical test case which operation is lightweight compared to its data migration. When each of the input vectors contains N elements, then there will be $3N$ elements to be transferred between CPU and FPGA ($2N$ from CPU to FPGA and N elements from FPGA back to CPU). While the total operations is N.

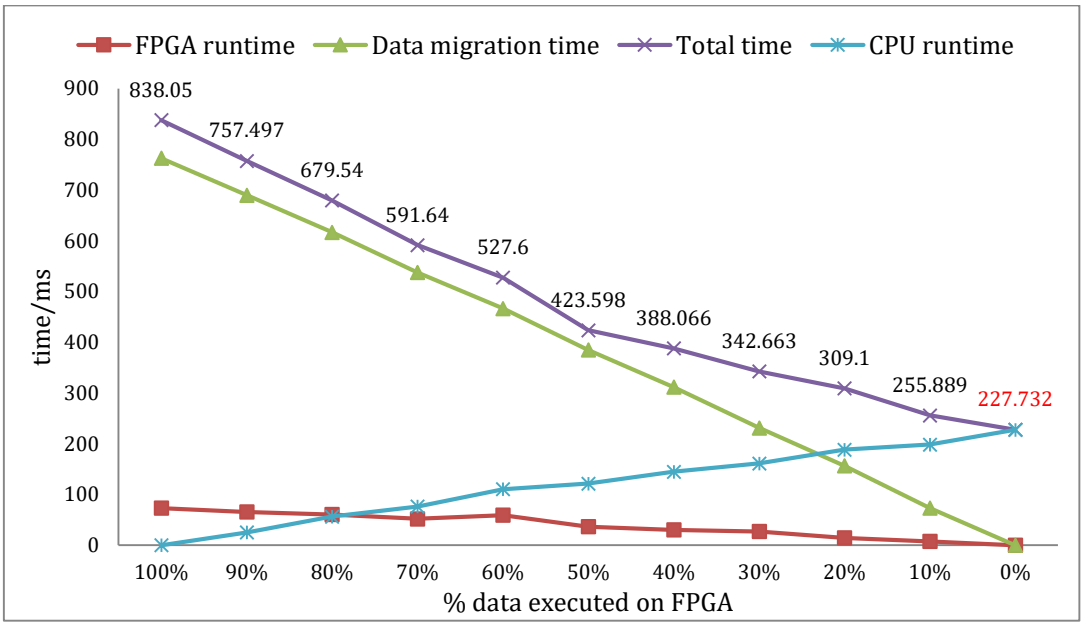


Figure 4.4.1.1 CPU-FPGA Overlapped Computation for Vector Multiplication

From Figure 4.4.1.1, we can find that either CPU runtime or FPGA runtime is basically proportional to its input data size. However, we can also observe that even though FPGA is always faster than CPU, the time spend on copying data is much longer than either FPGA computation or CPU computation. The *Limiting Ratio* is:

$$lim_{p\to\infty} Speedup = \frac{t_{cpu}^{N}}{t_{migration}^{(N)}} = \frac{227.332}{762.79} \cong 0.30$$

For this case, because sending data from the host to the device side as well as copying the result back to the host is a big overhead, which would be the primary determinant of the program's total runtime, the strength of FPGA and CPU overlapped computation cannot make up such expense. Therefore, the optimal implementation, which total runtime is 227.732ms, is CPU-only platform.

4.4.2. Dot Product

After testing Vector Multiplication, we seek for another benchmark with less data migration intensity but higher computational intensity. Dot Product is implemented as our second benchmark.

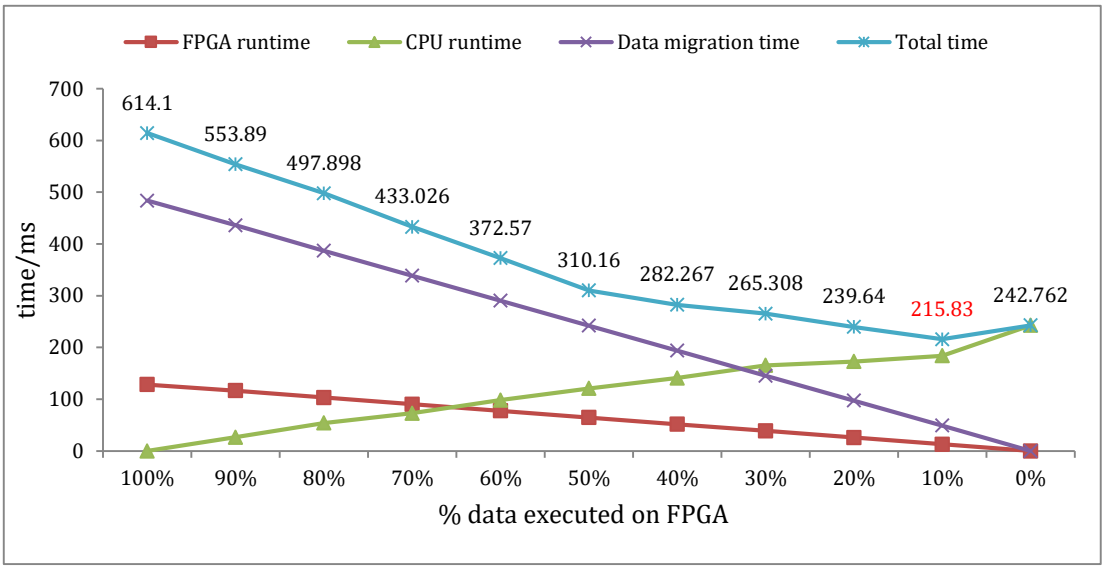


Figure 4.4.2.1 CPU-FPGA Overlapped Computation for Vector Dot Product.

In this experiment, the number of migrated data is $2N + 1$($2N$ from host to FPGA and 1 from FPGA to host CPU), and the number of operation is $2N - 1$(N multiplications and $N - 1$ additions).

Although it costs less time to transfer data between host and device side than *Vector Multiplication*, which data migration still contributes most to the total

execution time. The *Limiting Ratio* of the enhanced fraction for *Vector Dot Product*:

$$lim_{p\to\infty} Speedup = \frac{t_{cpu}^{N}}{t_{migration}^{(N)}} = \frac{227.332}{483.58} \cong 0.50$$

The speedup is less than 1, however, our result shows that the program's runtime is slightly improved from $242.762ms$ to $215.83ms$. And the optimizing point shifts from 0% to 10% as shown in Figure 4.4.2.1 and reduced by 11.1%. It is because the data migration time will decrease as we reduce the amount of data that assigned to FPGA, Then

$$Slim_{p\to\infty} peedup = \frac{t_{cpu}^{N}}{t_{migration}^{(n)}} = \frac{t_{cpu}^{N}}{\frac{n}{N} t_{migration}^{(N)}} \cong \frac{N}{2n}$$

Thus, if there exists an appropriate n that ensures $\frac{N}{2n} > 1$, the program's performance could still be improved.

4.4.3. L^p Space

We further increase the ratio of operation frequency verses data migration quantity in order to explore more chance to optimize the program. In this part, Vector Norm is experimented as the third benchmark.

Because there is only 1 vector need to be transferred to FPGA, the amount of data copied from host to device is N, and only 1 result is going to be written back from device side to host side. In the same time, the number of operation is $2N + 1$, which including Nmutilications, Nadditions and 1 *pow()* function.

In this experiment, data migrate between CPU and FPGA takes less time than processing the same size of data on CPU. Here shows the speedup of the enhanced fraction for $p = 2$ to $p = 6$ are shown in Figure 4.4.3.1:

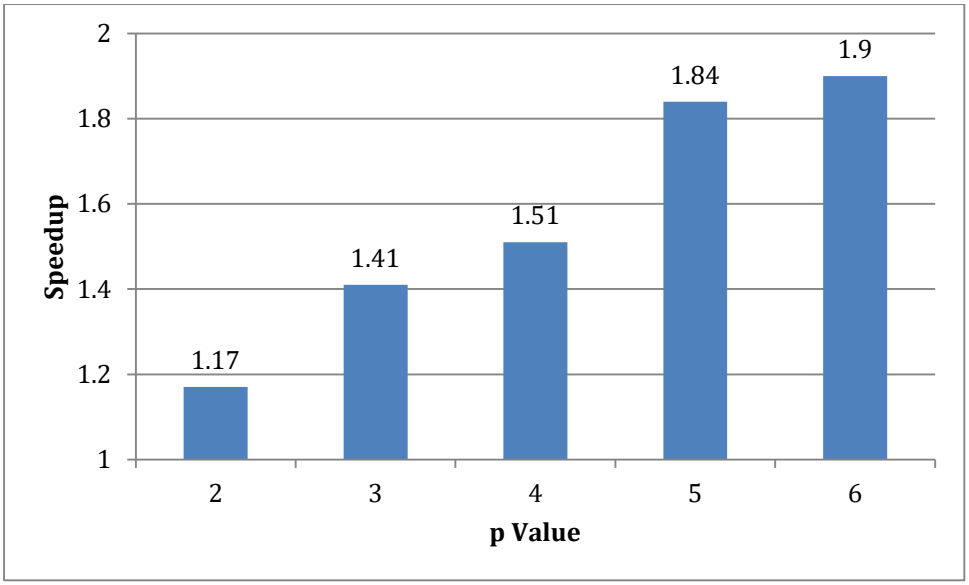


Figure 4.4.3.1 The Theoretical Speedup vs. p-Value.

From the above results, we can find there is an upward trend in theoretical optimal speedup as p value increasing. Therefore, the program should have greater chance to be optimized with FPGA-CPU overlapping computation, and we could also assume that the larger the p's value, the greater the speedup.

Supported by the above results, we then measure the performance under each p Value, and produce the following plots:

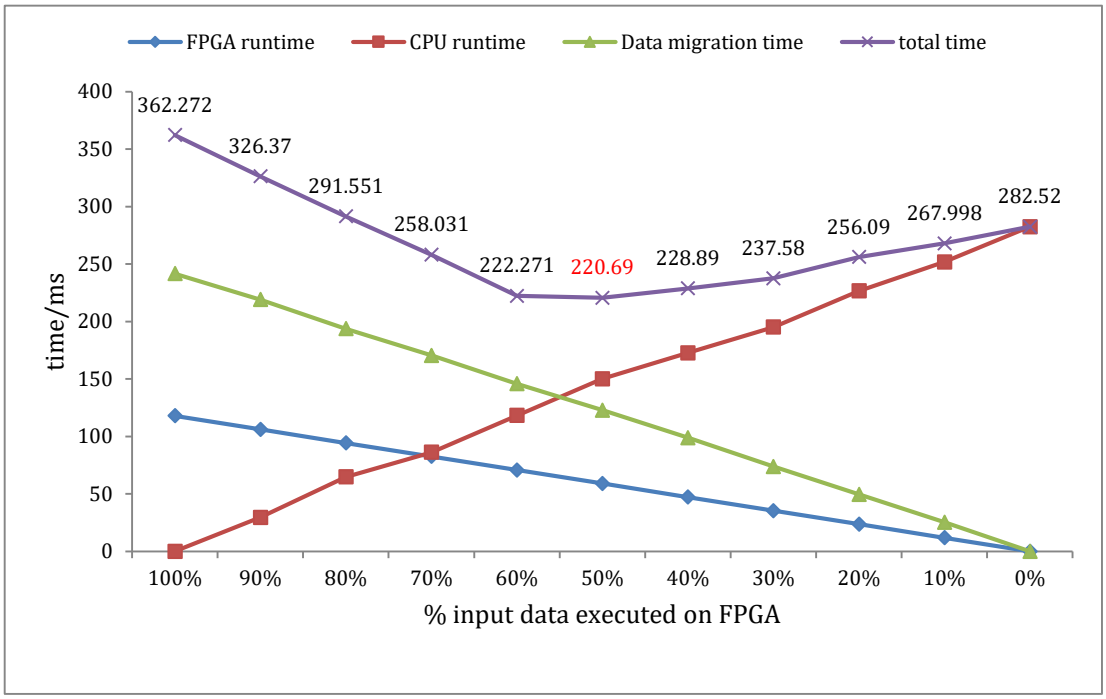


Figure 4.4.3.2 CPU-FPGA Overlapped Computation for L^2Space.

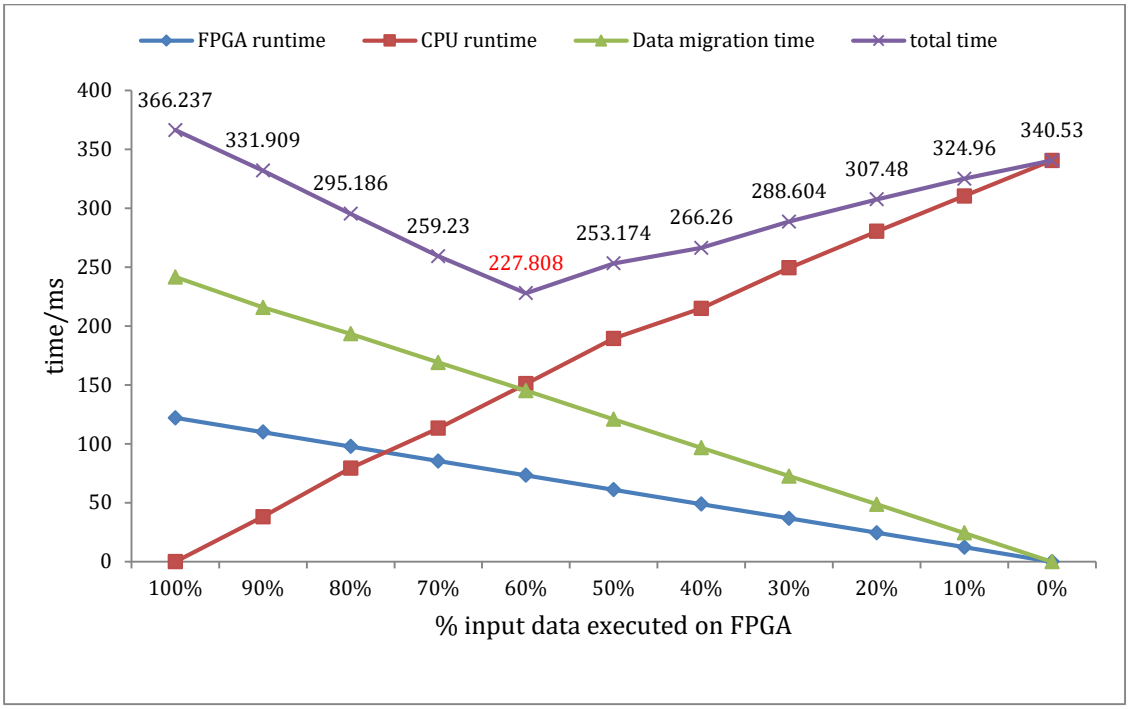


Figure 4.4.3.3 CPU-FPGA Overlapped Computation for $\mathbf{L^3}$ Space.

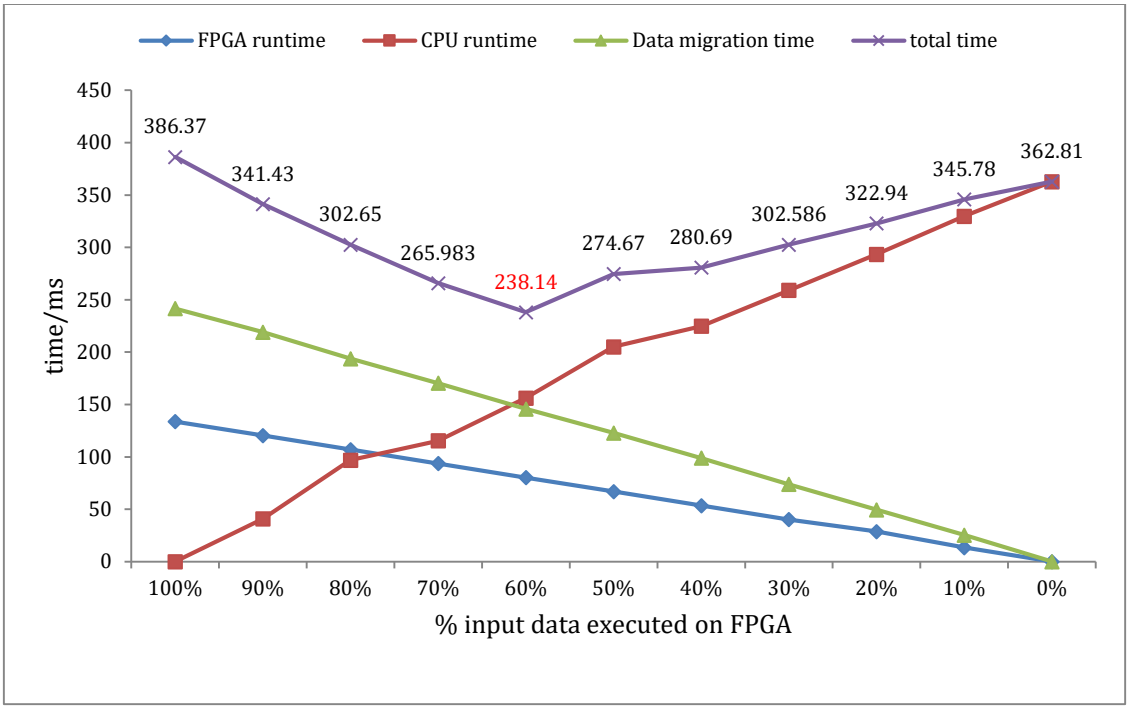


Figure 4.4.3.4 CPU-FPGA Overlapped Computation for L^4 Space.

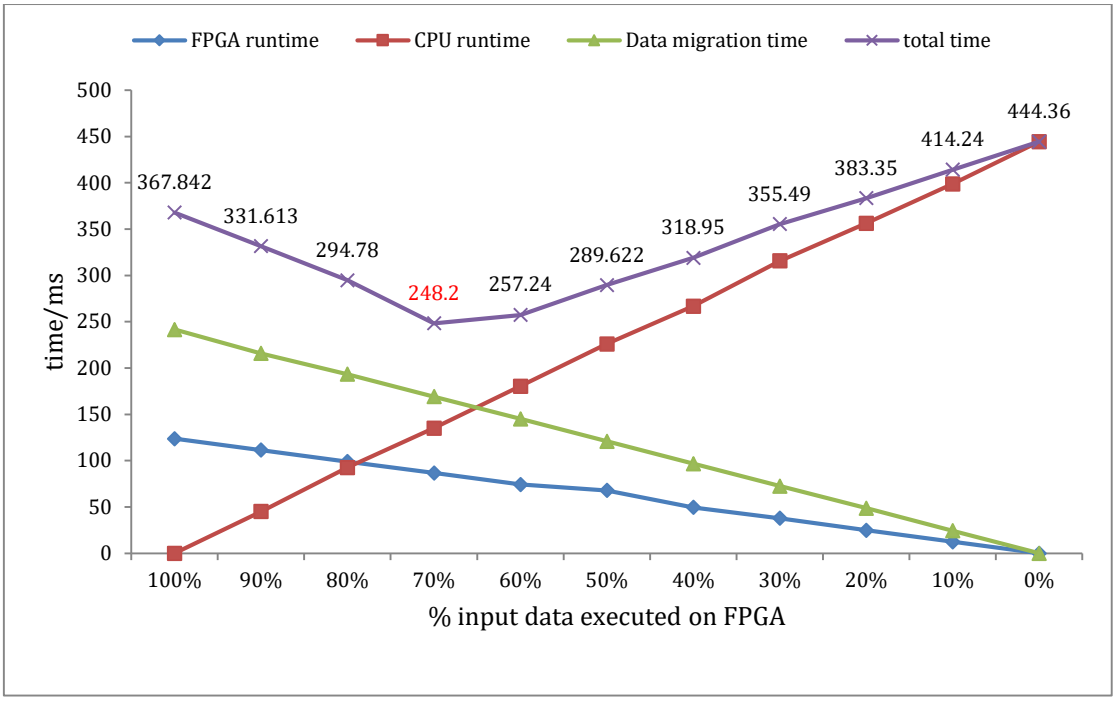


Figure 4.4.3.5 CPU-FPGA Overlapped Computation for $\mathbf{L^5}$ Space.

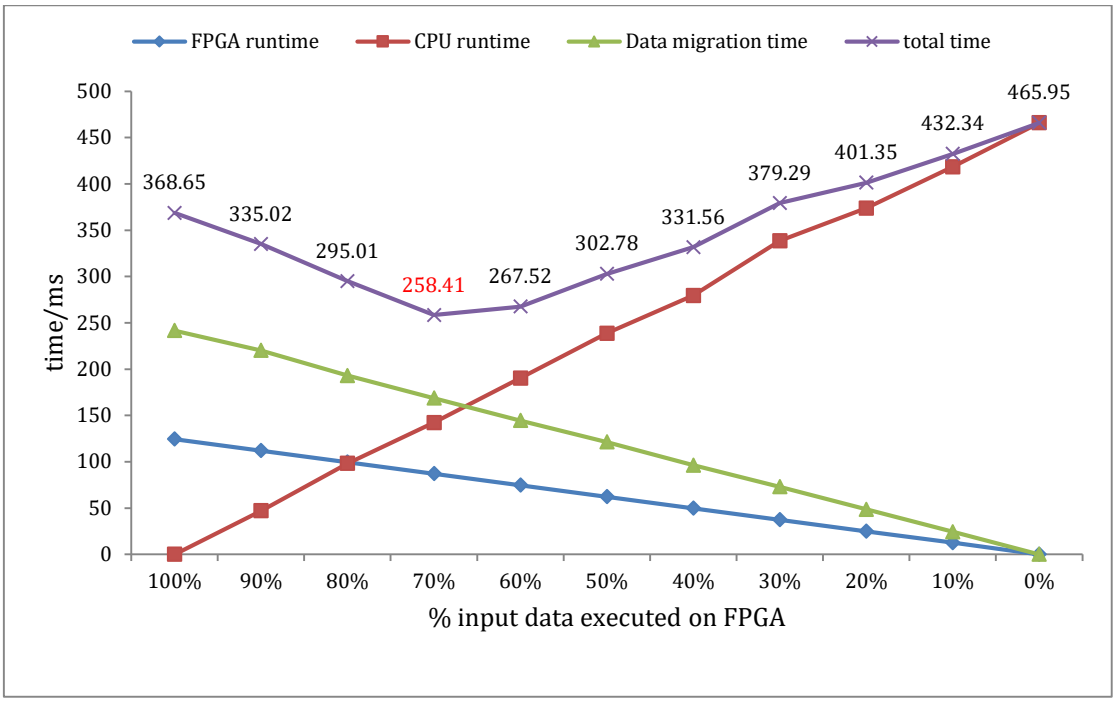


Figure 4.4.3.6 CPU-FPGA Overlapped Computation for $\mathbf{L^6}$ Space.

The experiment results shown from Figure 4.4.3.2 to Figure 4.4.3.6 confirm our assumption. We collect the results in Table 4.4.3.1 to help us clearly identify the trend.

Table 4.4.3.1 The Impact of Computational Intensity on Speedup.

p value	*Limiting Ratio*	Time reduced/ms	Optimal Point	Real Speedup
2	1.17	61.83	50%	1.28
3	1.41	112.722	60%	1.49
4	1.51	124.67	60%	1.52
5	1.84	196.16	70%	1.79
6	1.90	211.54	70%	1.90

From Table 4.4.3.1, as the theoretical speedup fraction increases, the optimal point shifts to larger percentage, which means the advanced computing capability of FPGA can counteract more overhead of data migration. Besides, our assumption that *Limiting Ratio* would increase with the growth of p value is also confirmed by our experiments, where *Limiting Ratio* reflects the comparison of CPU computational intensity and data migration intensity. This feature demonstrates that if a program's computations are more intensive than its data migrations, it would be further accelerated with CPU-FPGA heterogeneous system.

4.4.4. 1D Convolution

Our next benchmark is 1D Convolution with a filter size 4. Compared with its computation intensity, the data migration intensity could be much light. Hence, we expect this benchmark to be optimized with more FPGA computations.

To predict the possibility of optimizing this benchmark, we still need to evaluate the *Limiting Ratio*:

$$lim_{p\to\infty} Speedup = \frac{t_{cpu}^{N}}{t_{migration}^{(N)}} = \frac{1141.7}{483.42} \cong 2.36$$

This ratio is larger than all the above ratios, which confirms our expectation.

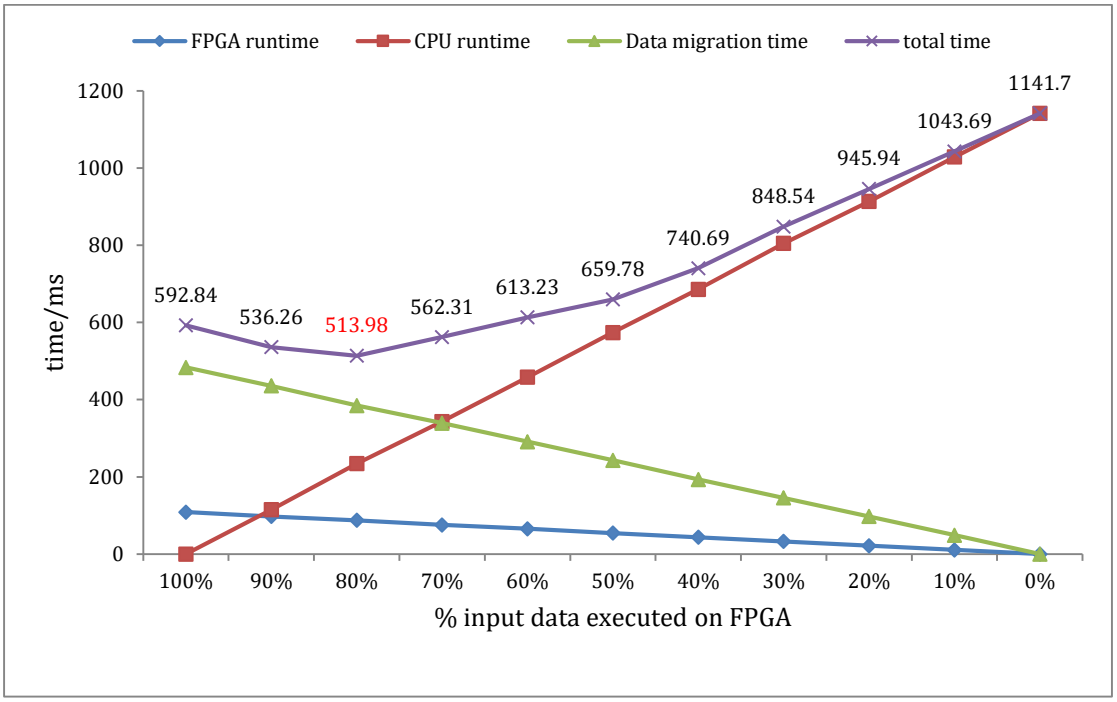


Figure 4.4.4.1 CPU-FPGA overlapped computation for 1D Convolution.

As we can see from Figure 4.4.4.1, the optimal point locates at 80% of input data assigned to FPGA. The total runtime changes from 1141.7ms to 513.98ms, which reduced by 54.98%.

4.4.5. Matrix Multiplication

Our last benchmark is *Matrix Multiplication*. The inputs and output are two 2-dimentional vectors and one 2-dimentinal vector respectively. Since we send two matrices block by block rather than a whole matrix, the data migration time is significantly reduced.

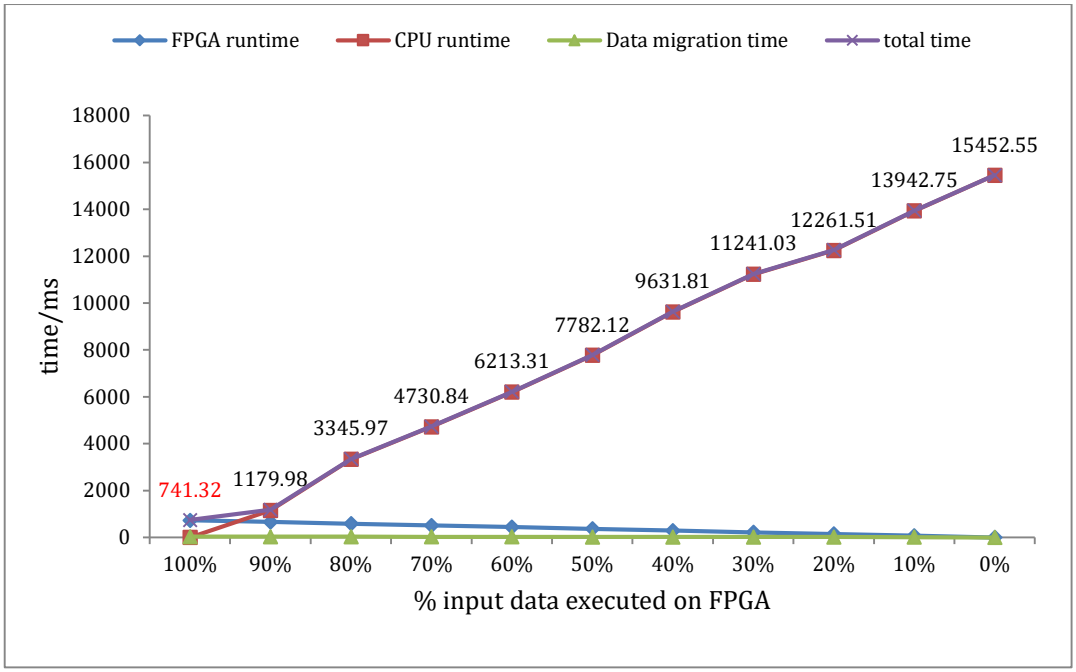


Figure 4.4.5.1 CPU-FPGA overlapped computation for Matrix Multiplication.

As we can see from Figure 4.4.5.1, data transfer time between CPU and FPGA compared to CPU runtime is extremely short, and the *Limiting Ratio* is far greater than 1, which value is:

$$lim_{p\to\infty} Speedup = \frac{t_{cpu}^{N}}{t_{migration}^{(N)}} = \frac{15452.6}{32.122} \cong 481.05 \gg 1$$

Such result indicates that the FPGA is much more capable to handle this program than CPU. The result further verifiers FPGA's capability and the best performance of Matrix Multiplication occurs when all the computations are completed by FPGA, which total runtime reduced by 95%.

By evaluating the results of the above five benchmarks, we can conclude that the degree of optimizing a program with FPGA is largely rely on the comparison between data migration intensity and CPU computation intensity. This comparison can be evaluated by *Limiting Ratio*: If the ratio is larger than 1,

meaning that we can always gain higher performance from FPGA acceleration. For this case, we would better transfer a great portion of data to FPGA while remain a small portion of data on CPU. If the ratio is less than 1, we should further divide this case into two sub-situations. If its value is very small, which means the migration overhead is overwhelming. Then we should stick with CPU and avoid the FPGA acceleration. But is the ratio is a little less than 1, we may achieve higher performance if we let CPU do the majority computations and FPGA execute partial operations.

In this way, programmers can choose the method of optimizing their programs without working on FPGA.

5. Conclusion

The latency of data processing speed relative to the rapid growth of data leads to a strong demanding for high performance computing architecture. Even though the computation ability of multi-core CPU has largely improved compared with traditional single-core CPU, it is still inefficient to processing the growing data and huge computations.

FPGA, as an integrated circuit with powerful computation capability, is put forward as a new processor. Its pipeline parallelism technique is very attractive for processing large-scale data. However, the cumbersome hardware description makes it unpopular in industry and commercial fields. Software developers are also unwilling to use it because of its unfriendly programming language and debugging flow. While problems solved when OpenCL was put forward as a solution. This hardware-targeted programming language is much software-friendly, which is much like C/C++ language. With the help of OpenCL, FPGAs are increasingly applied to diverse fields as powerful accelerator.

In this thesis, we have first compared the performance of non-optimized FPGA computations with optimized FPGA computations with a simple vector addition benchmark, and found that the runtime can be reduced by 75.5%. Then we've deeply investigated the FPGA's performance with Altera OpenCL, and analyzed the pros and cons of two optimization methods: AOC automated optimization and manually setting resource attributes. We also compared two optimization techniques – replicating Compute Unit vs. kernel vectorization – to help programmers better decide the priority of applying those two techniques. Our test results showed that applying kernel vectorization can leads to higher

performance with exploiting the same or less hardware resource than CU replication.

After we have understood the optimization techniques, we presented a practical method that can further optimize the program. This method overlaps the computations of FPGA and CPU, and makes use of CPU's idle period. However, we found that even though CPU-FPGA heterogeneous platform can produce higher performance in most of the cases, data migration may costs much more time and lead to an overall worse performance than CPU-only systems. So we put forward a "Speedup" ratio to help programmers evaluate the possibility of taking advantage of FPGA only with software programming. Our results showed that if the ratio is greater than 1, the program can absolutely be optimized with CPU-FPGA platform, which it is less than 1, the program can probably be optimized. This "Speedup" ratio can also describe the optimization degree of applying FPGA accelerator. Five benchmarks were tested to verify our expectation. The results showed that as the computation intensity of a program increasing and/or data migration intensity decreasing, the program could achieve higher performance.

Our future work will focus on accelerating more complex algorithm, like Convolutional Neural Network (CNN). CNN algorithm is computationally intensive, thus how to reduce the cost as well as ensure its accuracy is a key questions.

In [19], Farabet *et al.* put forward a scalable hardware system to implement convolutional neural networks for large-scale multi-layered synthetic vision systems. It is demonstrated that at the same condition that the convolution filters size of 3x3 and image size of 500x500, either FPGA or ASIC is capable to

run medium complexity tasks with 15W and 1W power consumption respectively, which is lower than 90W of CPU implementation.

We plan to optimize CNN algorithm by loading layers with intensive computations but lighter data migrations to FPGA, while remaining other layers in CPU. Additionally, we hope to achieve deeper overlaps by dividing both data transfer part and FPGA kernels into blocks, and then overlap each two blocks to accomplish optimal performance.

BIBLIOGRAPHY

[1] Freund, Richard F., and Howard Jay Siegel. "Guest Editor's Introduction: Heterogeneous Processing." *Computer* 6 (1993): 13-17.

[2] Maheswaran, Muthucumaru, et al. "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems."*Heterogeneous Computing Workshop, 1999.(HCW'99) Proceedings. Eighth*. IEEE, 1999.

[3] Chen, Doris, and Deshanand Singh. "Invited paper: Using OpenCL to evaluate the efficiency of CPUS, GPUS and FPGAS for information filtering."*Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012.

[4] Accelerate, Opencl on fpgas for gpu programmers.

[5] Settle, Sean O. "High-performance dynamic programming on FPGAs with OpenCL." *Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC)*. 2013.

[6] Munshi, Aaftab, et al. *OpenCL programming guide*. Pearson Education, 2011.

[7] Khronos OpenCL Working Group. "The OpenCL Specification, version 1.0. 29, 8 December 2008."

[8] Khronos OpenCL Working Group. "The OpenCL Specification Version 1.1. Khronos Group, 2011."

[9] Khronos OpenCL Working Group. "The OpenCL Specification Version 1.2. Khronos Group, 2012."

[10] Windh, Skyler, et al. "High-Level Language Tools for Reconfigurable Computing." *Proceedings of the IEEE* 103.3 (2015): 390-408.

[11] Singh, Deshanand. "Implementing FPGA design with the OpenCL standard."*Altera whitepaper* (2011).

[12] Intel® Microprocessor Export Compliance Metrics http://www.intel.com/support/processors/sb/cs-017346.htm

[13] GeForce 9800 GTX Specification http://www.geforce.com/hardware/desktop-gpus/geforce-9800-gtx/specifications

[14] Hara, Yuko, et al. "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis." *Journal of Information Processing* 17 (2009): 242-254.

[15] Ndu, Geoffrey, J. Navaridas, and M. Lujan. "CHO: A Benchmark Suite for OpenCL-based FPGA Accelerators." *University of Manchester Technical Report* (2014).

[16] Vanderbauwhede, Wim, Leif Azzopardi, and Mahmoud Moadeli. "FPGA-accelerated Information Retrieval: High-efficiency document filtering." *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*. IEEE, 2009.

[17] Meswani, Mitesh R., et al. "Modeling and predicting performance of high performance computing applications on hardware accelerators." *International Journal of High Performance Computing Applications* 27.2 (2013): 89-108.

[18] Carrington, Laura, et al. "An idiom-finding tool for increasing productivity of accelerators." *Proceedings of the international conference on Supercomputing*. ACM, 2011.

[19] Farabet, Clément, et al. "Hardware accelerated convolutional neural networks for synthetic vision systems." *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*. IEEE, 2010.

[20] Altera, S. D. K. for OpenCL. Programming Guide. (2014).

[21] Coussy, Philippe, et al. "An introduction to high-level synthesis." *IEEE Design & Test of Computers* 4 (2009): 8-17.

[22] Czajkowski, Tomasz S., et al. "From OpenCL to high-performance hardware on FPGAs." *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*. IEEE, 2012.

[23] Altera, S. D. K. for OpenCL. Optimization Guide. (2014).