A Study on Efficient Application Mapping on Parallel Computing Accelerators

Keisuke Dohi

2013

Abstract

Since the invention of electronic computers, their performance has been constantly advanced. The recent progress of micro processors in performance has been mainly achieved by increasing the number of cores on a device, instead of increasing working frequency. In addition, because of increasing of density of semiconductors, not only computational performance but also density of power consumption has been steadily growing, and this trend is expected to restrict the performance of computers as a monolithic system. Therefore, increasing energy efficiency of the computers along with computational performance is now a critical issue. Against a backdrop of these trends, both hardware and software have new challenges; a) suppressed operating frequency; b) large amount of but poor processors; c) power consumption wall; and d) narrow bandwidth technologies for external memory. Both Graphics Processing Unit (GPU) and Field Programmable Gate Array (FPGA) have considerable performance/cost benefits because these are massively manufactured as consumer devices, and thus these are expected to be applicable as parallel computing accelerators in various fields. In addition, this dissertation focuses high-level synthesis tools that generate desired hardware logic using high abstraction layer descriptions. Although parallel computing accelerators include GPUs and FPGAs are available, efficient mapping of desired applications on the accelerators is still an important and difficult issue. Therefore, this dissertation addresses efficient application mapping on parallel computing accelerators using on-chip memories. This dissertation focuses stencil computation and stream-oriented process as design patterns on the accelerators, and demonstrate concrete application mappings from the design patterns.

After describing motivation and background of this study in Chapter 1 to Chapter 3, in Chapter 4, implementation of Smith-Waterman algorithm on GPUs is presented. Operations of the algorithm are massively parallelizable. In addition, the computation requires relatively high ratio of integer arithmetic to memory access. Therefore, costs of synchronizing arithmetic cores and on-chip memory access impact computational performance significantly. Implementing the algorithm on a GPU shows the importance of effective utilization of the warp-level parallelism on a GPU. Central to this technique is a divide and conquer approach to alignment matrix calculation in which a whole pairwise alignment matrix is subdivided. This leads to the efficient calculation of data by 32 threads, and the reduction in the number of load/stores to/from external memory. As a result of evaluation, the implementation of the algorithm achieved a throughput ranging between 9.09 Giga Cell Updates Per Second (GCUPS) and 12.71 GCUPS on a single-GPU version, and a throughput between 29.46 GCUPS and 43.05 GCUPS on a quad-GPU platform, which was the world fastest GPU implementation of the algorithm at that time.

In Chapter 5, an implementation of electro-magnetical simulation by a finite-deference time-domain (FDTD) method for analysis of micro strip antenna characteristics on a GPU is presented. The algorithm is known as a kind of stencil computation that has a high degree of parallelism. The implementation uses Perfectly Matched Layer (PML) as a boundary condition, hence memory accessing pattern and update-equations are changed depend on location of grids. The transformation technique of update-equations for partial boundary cells is also proposed. The empirical experiment showed the memory bandwidth of 62.5 GB/s which corresponds to 55.8 % of the peak of the target GPU.

In Chapter 6, implementation of human detection from video image using Histograms of Oriented Gradients (HOG) feature on an FPGA is presented. Since the HOG features are extracted from luminance gradient of an image they have high robustness for lighting condition and are widely used to detect a human in an input video image. In order to cope a lot of computation costs of the HOG feature extraction, the architecture is designed in a data stream oriented deep pipelined manner. As a result of evaluation, the throughput of 62.5 FPS was achieved without using any external memory modules.

In Chapter 7, implementation of 3-D heat spreading simulation using MaxCompiler, which is a high-level synthesis tool based on Java on an FPGA is presented. This chapter aims to establish estimation models for computational performance and resource utilization using user parameters on high abstraction layer description. In addition, energy consumption of accelerator is measured for various parameter configurations. As a result of evaluation, the best configuration achieved about six times faster than CPU implementation in performance, and the proposed estimation model provided us reasonable estimation about performance and resource utilization.

In Chapter 8, implementation of ellipse estimation from video image using RANSAC algorithm on an FPGA is presented. The algorithm needs to solve simultaneous equations as much as possible to get a reasonable solution. To solver simultaneous equations, three types of algorithms were implemented. As a result of evaluation, the throughput of 62.5 FPS was achieved with 3.34 W power consumption. While the optimal algorithm needs to be chosen depending on the amount of resources on FPGAs and required criteria, the FPGA based system that consists of streamed structure is promised as a better solution for the application.

The results of implementation showed both GPUs and FPGAs have advantages over existing microprocessor architectures in computational performance and energy consumption. At the same time, modification of existing algorithms played a significant role in achieving a high degree of computing efficiency with the parallel computing accelerators. Especially, a view of making the best use of on-chip memory with a deep pipelined manner was crucial.

Acknowledgment

This study has been completed with the help of discussions with many people at Nagasaki University. I would like to express my great thanks to all of them for their support, contributions and encouragement. This study also has been supported by the following scholarships; Support Center for Advanced Telecommunications Technology Research, Foundation, Innovation Training Program Center of Kyushu University, and Nagasaki University.

I express my deepest gratitude to Associate Professor Yuichiro Shibata for his tremendous and long-term support as my supervisor. For almost nine years from an admission examination to Nagasaki University, or at least six years after I was assigned to his laboratory, I have learned a lot from him not only for research work but also about life. Great experiences that I've had, such as attending international conferences, collaborative research at the University of Edinburgh and intern in the Maxeler Technologies at London, would not have been possible without his invaluable supports. I would like to take this opportunity to apologize to him for that I always submitted every papers at the eleventh hour.

I am grateful to Professor Kiyoshi Oguri, my co-supervisor. I have always been impressed at his sincere approach to his research and discussion of computers in the next decade. I remember vividly the first time of his lecture on the stream-oriented architecture. My every studies would not have been achieved without the lecture. My sincere thanks are expressed to Professor Senya Kiyasu for his incisive comments and advice. He also gave a lot of helps for application of a scholarship and the presidential award I received. I with to express my thanks to Associate Professor Takafumi Fujimoto for his valuable advice and constructive comments as a specialist in antenna designing. A part of this study is based on his research, and he provided valuable data for the research.

Special thanks goes to Hideki Yamada, Sayaka Shida, Tomoya Ishimori, Yusuke Kawabata, who have advised and given me guides as senior researchers. I am grateful to the many friends and colleagues who have discussed, supported and encouraged for me. They include the following individuals: Masato Soejima, Tomonori Ooya, Yuta Araki, Hiroaki Murabe, Kazuhiro Negi, Kazuma Hamawaki, Kohei Matsunobu, Kyan Yusuke, Ryo Egawa, Takeshi Kakimoto, Yui Ogawa, Kanako Nakashima, Ryota Iyoya, Yuki Matsuo, Yuma Hatanaka, Kunihiro Ueda, Naoki Kawamoto, Yoshihiko Yamabe, Yoshihiro Nakamura, Yuji Yorita, Jinpei Hamamura, Kaoru Hamasaki, Kota Aoki, Kota Fukumoto, Syun Kashiwagi, Aiko Iwasaki, Koji Okina, Masahito Oishi, Rie Soejima.

Most importantly, I thank my younger brother and sisters for their understanding. Especially I would like to thank my mother Mihoko Dohi for her constant support.

Keisuke Dohi
Nagasaki, January 2014

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the invention of electronic computers, their performance has been constantly advanced. The progress of micro processor architectures had been supported by a significant increase in working frequency offered by rapid development of semiconductor manufacturing technologies. However, after Intel NetBurst microarchitecture [7] was developed in 2000, the working frequency progress began to stagnate. Therefore, recent progress of micro processors in performance has been mainly achieved by increasing the number of cores on a device. Now, parallel computing on computers connected by interconnection is generally used for supercomputing and cluster computing, as well as for mobile computers equipped with multi-core microprocessors [8, 9]. It suggests parallel computing will be used for various fields more and more.

On the other hand, because of increasing of density of semiconductors, not only computational performance but also density of power consumption has been steadily growing, and it is expected the power consumption will be a main factor to restrict the performance of computers as a monolithic system. It raises an issue about increasing energy efficiency of the computers along with computational performance.

Against a backdrop of these trends, both hardware and software have new challenges; a) suppressed operating frequency; b) massively parallel processing; c) high degree of energy efficiency; and d) narrow bandwidth technologies for external memory. Although ASIC based special hardware can achieve the highest performance in many metrics, manufacturing cost and design time of ASICs keep increasing, and developing new ASICs for user applications may be an unrealistic solution in many cases. Therefore, this paper focuses GPU, which was originally developed for graphics processing, and Field Programmable Gate Array (FPGA), which consists of many arithmetic elements and memory elements connected by user-configurable routing wires, to address the above problems. Both devices have considerable performance/cost benefits because these are massively manufactured as consumer devices, and thus these are expected to be applicable in various fields.

GPUs have been grown to achieve high resolution and fast graphic processing. The architectures are specialized for relatively simple control-flow and massively data parallel computations. The architecture is a kind of many-core architectures which consists of many processors with a Single-Instruction-Multiple-Data (SIMD) mechanism and a wide memory bandwidth to external DRAM. Additionally, these processors employ their own programming model in which the many cores have a common instruction flow to achieve a wide bit width to the memory interface and to make it easier to hide memory access latencies. Because not only computer games but also some operating systems require a certain amount of performance in graphic processing, GPUs are widely used for personal computers and has good performance/cost benefits. Therefore, some researches have reported attempts to use GPUs as accelerators for scientific computations since the mid-2000s. After that Graphics Processing Unit (GPU) vendors include NVIDIA and AMD officially released General-Purpose computation on GPU (GPGPU) platforms for these kinds of computations and these platforms have been widely known as acceleration platforms for scientific computations. In addition, specialized products for GPGPU, which have high reliability and additional arithmetic units for the scientific computations, have been released for workstations and supercomputers to achieve higher computation performance.

FPGA is a programmable semiconductor device which consists of an array of arithmetic elements, memory elements, I/O pins and routing wires. Attributes of each element and routing information can be configured by users to make desired digital logic circuits. Generally, a structure of FPGAs is described as an array of Flip Flops (FFs), Look-Up Tables (LUTs) and routing wires to connect these elements. But in recent years, modern FPGAs are an aggregate of various hardware modules which consists of clock managers, SRAM, multipliers, micro processors, PCI Express (PCIe) interface controllers, DRAM interface controllers, Serializer/Deserializer (SERDES) modules, I/O drivers/receivers for various electric specifications and so on in addition to above basic elements. This change is supported by the expanding FPGA market. For example, Xilinx Virtex-7 FPGA family devices are manufactured with TSMC 28nm process [10]. It shows the market has a need for FPGAs that use almost state-of-the-art semiconductor manufacturing process for the products. Against a backdrop of the above changes for FPGAs architecture, FPGAs have been used not only for the prototype of ASIC or glue logics between devices, but also as accelerators which have considerable performance/cost benefits and low energy consumption. One benefit to use FPGAs as accelerators is we can optimize bit-width and number of pipeline stages of arithmetic units and registers for each application. In microprocessor architectures, the number of pipelines stages has a tendency to be a small number because the branch misprediction penalty increases according with the number of pipelines stages. This is because instruction flows are undefined at the designing stage of micro processors. On the other hand, the functionality of FPGAs can be configured by users after manufacturing. It allows us to perform deep-pipelined structure for each application. Data-flow computing is a way of performing efficient computation for FPGAs, where instructions are spread spatially and data flow is focused instead of control flow of applications. Although the data-flow computing have been researched from 80s, applications were limited by lack of enough resources on devices. However, the increase in the resources of modern FPGAs allows us to perform various applications with the data-flow computing manner, and the computer architecture with this paradigm will get attention more and more. Since FPGAs require higher hardware cost to due to its flexible routing structure, Application Specific Integrated Circuit (ASIC) can achieve better performance when the same level manufacturing process is used. However, in addition to its economical benefit, FPGAs are also focused for its dynamically reconfigurable features which can reconfigure the circuit while the application is running. Some researches focus this feature to support virtually large hardware and to realize fault-tolerance systems.

One of challenges for FPGAs is to support high level languages like C or Java to describe desired hardware logic instead of low abstraction layer descriptions by Hardware Description Language (HDL) like Verilog HDL and VHDL. In contrast to software for microprocessors which expands the instructions temporally, hardware design expands for lack of the instructions spatially. It often requires a lot of re-design of the whole architecture for lack of enough hardware resources. From this point of view, FPGAs have problems about programmability, modifiability and reusability. In addition, describing the hardware design by HDL cannot take advantage of software resources for microprocessors. Therefore, in recent years, high-level synthesis tools which allows us to design hardware using high abstraction layer descriptions are focused [11–14]. Although current high-level synthesis tools have some restrictions, it provides us similar development environments to software. One of reasons these high-level synthesis tools started to attract attention is reducing hardware designing cost is sometimes more important issues than designing optimal hardware because of increase in FPGA hardware resources. This is similar to the relationship between assembly languages and high-level languages. The high-level synthesis tools are expected to allow us to reuse the resources and knowledge amassed through existing software development, and to verify the behavior of described hardware quickly on microprocessors. On the other hand, some high-level synthesis tools generate hardware designs from much higher abstraction layer descriptions using Domain-Specific Languages (DSLs), description languages specialized for specific application domains.

Important design strategies to accelerate applications using GPUs and FPGAs are to access off-chip memories effectively and to massively utilize on-chip memories. Both GPUs and FPGAs have on-chip memories consist of SRAM, and these memories are distributed over the device. Therefore, the distance between arithmetic elements and memory elements are close both logically and physically. These memories are comparable to cache memory of micro processors, but in contrast, users can utilize these distributed memories in more explicit ways. In addition, as opposed to the cache which provides data to a small number of arithmetic cores, all the distributed memories tightly coupled with arithmetic elements can be accessed essentially in parallel. It achieves a wide on-chip memory bandwidth, e.g. Xilinx Virtex-6 XC6VSX475T FPGA is reported to have a maximum on-chip memory bandwidth of up to 21 TB/sec [15]. In this way, from the aspect of using parallelly-placed arithmetic elements and memory elements offering a high peak performance in computation and a wide memory bandwidth to external memories, both GPUs and FPGAs should share a core strategy to optimize systems. This study therefore aims to reveal the methodology to efficiently map various applications to these architectures.

The remainder of this paper is organized as follows. Chapter 2 shows backgrounds of this study. In Chapter 3, challenges and purposes of this study are presented. In Chapter 4, implementation of Smith-Waterman algorithm for biological sequence alignment on GPU is presented. In Chapter 5, implementation of electro-magnetical simulation by Finite-Difference Time-Domain (FDTD) method for antenna designing on GPU is presented. In Chapter 6, implementation of human detection from video image using HOG feature on FPGA is presented. In Chapter 7, implementation of heat spreading simulation using MaxCompiler on FPGA is presented. In Chapter 8, implementation of ellipse estimation using RANSAC algorithm on FPGA is presented. Finally, this dissertation is concluded in Chapter 9.

Chapter 2

Background

2.1 GPU

The origin and evoluation of GPU architectures intended to be accelerators of graphics processing. Therefore the architectures had been evolved separately from that of CPUs. The architectures are basically specialized for graphics processing with data-level parallelism, hence the GPUs have benefited from improvement of the transistor count as increase of arithmetic units and on-chip memory. In addition, modern GPUs consist of programmable arithmetic units, called Unified Shaders, instead of fixed-function units. The interest let the GPUs obtain a position as an accelerator for scientific computations exploiting data-level parallelism.

Figure 2.1 shows simplified architecture of NVIDIA GT200b GPU architecture. The architecture can be classified as a Multiple-Instruction Multiple-Data (MIMD) composed of multithreaded SIMD processors. Processors which support multithreaded SIMD instructions, called Streaming Multiprocessor (SM), are connected external memories and a host system via an interconnection network. Each SM has a hardware thread scheduler to execute threads by interleaving. This thread scheduling hides long latencies of arithmetic units and external memory accesses. Therefore, GPUs are equipped with external DRAMs with a highthroughput and a highlatency.

Figure 2.2 shows an overview of the SM architecture. The SM is specialized for execution of SIMD instructions, all the processing units share a single instruction flow. Each processing unit has its own register file which contains 1,024 32-bit registers. The resiter file is not shared by processing units within an SM, while there is 16KB memory which shared within an SM, called shared memory in terms of NVIDIA GPU architecture. The shared memory consists of 16 banks, it thus performs high bandwidth when the SM accesses all the banks without confliction. The shared memory is not shared between SMs. All the SMs within a GPU share memory space on external DRAM. Access to the external DRAM from an SM is basically assumed that all the processing units access to a continuous region of the DRAM, and it performs high bandwidth by coalescing memory access from an SM.

As mentioned above, a GPU contains some types of memory units like CPUs use multilevel of caches. While the caches of the CPUs cannot be controled explicitly by programmers, GPUs offer fine-grained control of the register files and the shared memories which have high bandwidth and locate close to processing units to programmers. This is one of the key differences between CPUs and GPUs.

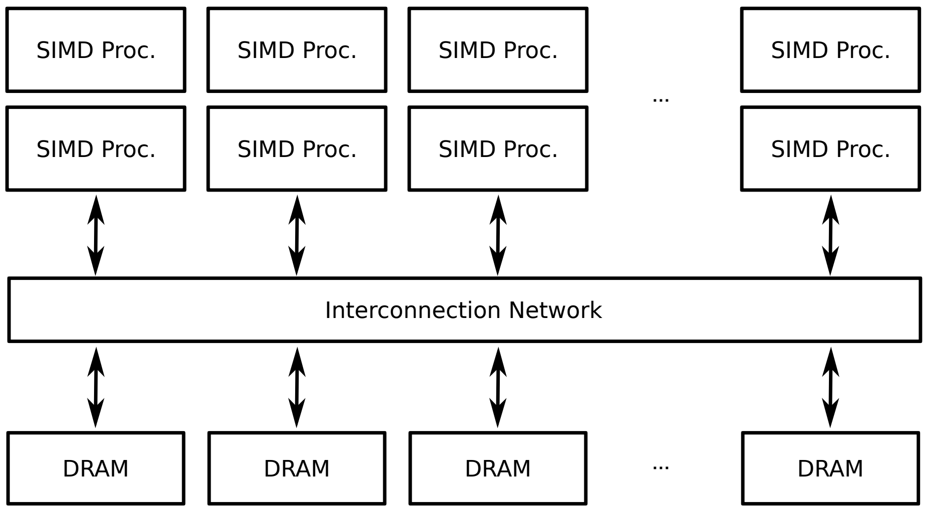

Figure 2.1: Overview of GPU architecture.

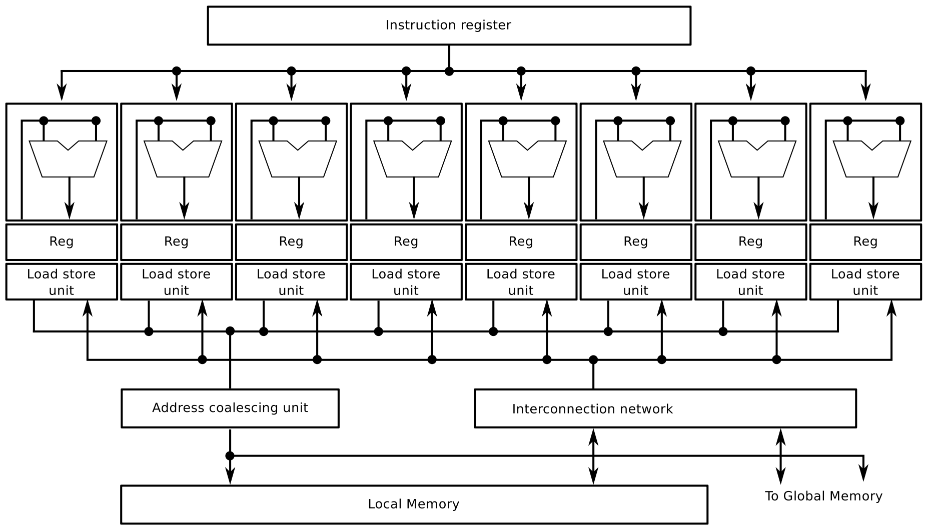

Figure 2.2: Overview of GT200b GPU SIMD Processor.

2.2 CUDA Programming Model

CUDA is a development environment for GPGPU offered by NVIDIA Corporation [16]. The language specification is an extension of C/C++ with a multithread programming model. There are a series of GPUs that are compliant with CUDA, and each CUDA-compliant GPU device has a version known as “CUDA compute capability”. Additionally, the amount of hardware resources can be different from one CUDA-compliant GPU to another.

At the lowest level, the concept of CUDA is parallel processing with hierarchically grouped multiple threads. The group of threads is called “Thread block” and the group of thread blocks is called “Grid”. Each thread executes the same code which is described as a sequential process, called “Kernel”. In CUDA, a kernel execution is called “Kernel call”. Each thread has a unique ID within a grid and is able to execute different instructions and/or operate on different data by using the ID, although they execute the same code. There is a synchronization statement that is essential in parallel computing as it synchronizes all threads within a thread block. There is no synchronization statement that synchronizes all of the threads within a grid. However, it can be synchronized all threads within a grid out of the kernel call by using CUDA API. Additionally, as it is discussed later, actual processing goes along in 32-thread unit, called “Warps” or 16-thread unit that are called “Half-warps”.

Memory in CUDA-compatible GPUs is hierarchical. It is important to understand the characteristics of these memories e.g. amount, bandwidth and region of sharing, to optimize CUDA code [17] [18]. The following presents the memories that is used in the implementation:

≤ Global memory: This memory has a large space (typically 1 to 6 GB or less) and is shared by all of the threads within a grid, but latency is high and bandwidth is lower than on-chip memory [17] . Additionally, it is not cached.

≤ Register: This is on-chip memory that is private to each thread. It is the fastest memory within CUDA and there are 16 K 128-bit registers within a thread block in GT200b GPU cores.

≤ Shared memory: This is on-chip memory that is shared by threads within a thread block. It is as fast as registers but its size is not very large (16 KB per thread block with GT200b GPU cores).

≤ Texture memory: This is read-only memory, shared by all threads within a grid and is cached. It looks like the cached global memory, hence its size depends on global memory.

≤ Constant memory: This is read-only memory like the texture memory. Its size is not large (64 KB per grid with GT200b GPU core) but it is as fast as registers if all threads within a half-warp access the same address.

2.2.1 Memory access issues with CUDA

One of the most important performance considerations is “Coalesced memory access” [18]. In CUDA, memory access instructions for global memory from threads within a half-warp can be packed into one or more wide memory access instructions. If they are packed into just one wide memory access instruction, it is called the coalesced memory access. If global memory accesses are not coalesced, then the performance of global memory access decreases considerably, at worst to one sixteenth [18]. Moreover, “Bank conflict” is an important consideration when shared memory is used. A bank conflict happens if *n* threads within a half-warp attempt to access the same bank at the same time. Here, the performance of shared memory access decreases to one *n*-th [17]. To avoid this problem, the size of array elements that sit on shared memory and the way to access them have to be considered carefully. It should be also considered memory alignment when data are passed between different types of memory. The issue will be followed up in more detail in Section 4.3.2 and 4.3.4.

2.2.2 SIMD elements in SPMD

There are 30 multi-processors (MPs) in a GT200b with GeForce GTX 275, GTX 285 and GTX 295 and each MP has 8 Stream Processors(SPs). In CUDA, a thread block means a group of threads that are executed on the same MP and thus can be synchronized by the “__synchthreads()” statement. A thread block consists of some smaller groups of threads called warps. Threads in the same warp share the same instruction stream and executed simultaneously. Therefore, the GPU architecture looks like an SPMD, but threads within the warp are always synchronized like in SIMD [19]. This is because there is usually no need to use the “__syncthreads()” statement to synchronize all threads in a thread block if each thread doesn’t access the shared memory or global memory that other threads within other warps access.

2.3 FPGA

Field-Programmable Gate-Array (FPGA) consists of arrayed FFs, LUTs, and other hard-macros and programmable interconnections as shown in Fig. 2.3. The LUTs and the interconnections provide us high degree of flexibility, and the hard macros provide area reduction and high operating frequency for commonly used hardware blocks such as memory and multipliers as Block RAM (BRAM) and DSP modules. In addition, I/O blocks for various signal specifications include LVCMOS and LVDS, serializer/deserializer (SERDES) blocks for high speed serial interface from 1Gbps to over 10Gbps, and PCI Express interface block are also integrated into modern FPGA architectures.

LUTs assume important role to realize the high degree of flexibility within FPGAs. *m*-Input *n*-Output LUT can encode any *m*-input *n*-output boolean functions and be implemented as a memory which has *m* address bits wide and *n* data bits wide. Xilinx Virtex-6 FPGA family has 6-input 1-output LUT some of them can be used as a 64x1bit memory.

The programmable routing resources are also important component as well as arithmetic and memory elements. Improvement of semiconductor manufacturing technologies makes routing delays one of the main factors in critical path [20]. It leads a trend that logic elements become more complex to reduce complexity of routing resources. For example, Xilinx Virtex-6 FPGA has a SLICE which consists of four 6-LUTs, eight FFs, wide-function multiplexers, and carry logics while Xilinx Virtex-II Pro FPGA has a one which consists of two 4-LUTs, two FFs, wide-function multiplexers and carry logics.

Partial Reconfiguration is a feature to partially modify the configurations of LUT and routing during other part of device is running. Since the partial configuration can be stored in a memory e.g. SRAM or flush memory, the feature can reduce required resources by switching modules used exclusively. In addition, the feature can support

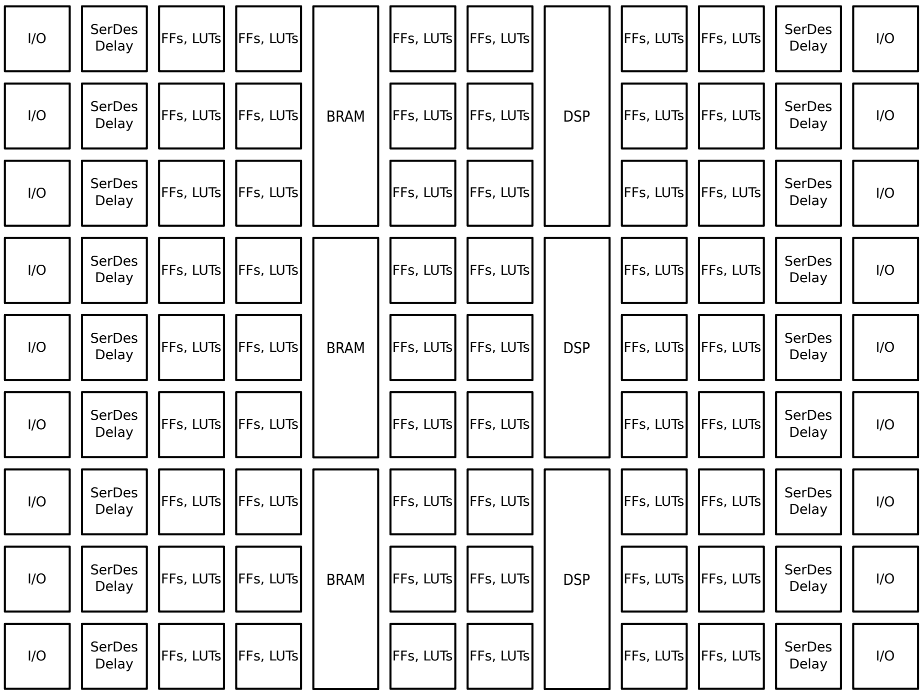

Figure 2.3: Overview of FPGA architecture.

virtual hardware mechanism which emulates bigger hardware than the target FPGAs and reconfiguration of kernel region of FPGA accelerators during the PCI express interface is running.

2.4 MaxCompiler

MaxCompiler is a kind of high-level synthesis tools which is developed by Maxeler Technologies. MaxCompiler intends high-performance computing on special FPGA accelerators and it provides a Java-based development platform with a stream-oriented programming model. MaxCompiler generates modules include arithmetic pipelines for FPGAs, PCIe interface, DRAM interface, FPGA-to-FPGA interface and APIs to access the accelerator from host programs.

Components generated by MaxCompiler classified roughly into (a) kernels, (b) managers, (c) streams and (d) host APIs. The kernels are main components for computation which have pipelined-architecture consisting of arithmetic units and registers. HDLs of the kernels are generated by the MaxCompiler from Java-based user source code. In addition, the compiler hides latencies of arithmetic units and handles stall control for pipelines, so that users can describe desired operations without being aware of pipeline stalls. The managers are components which handle I/O configurations of kernels and provide communication between accelerators and the host. The components are also generated by user descriptions. Attributes such as a clock frequency for kernels, connections between other components, and configurations of interface between the host program and kernels are managed by the managers. The streams are routes which data flow along. Communications between the host, kernels and DRAM are es-

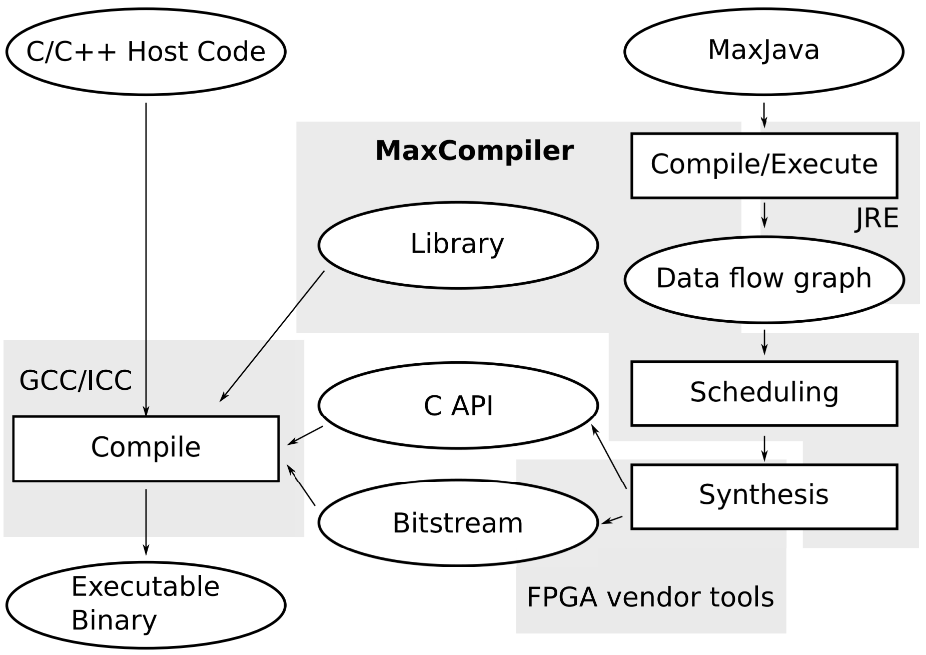

Figure 2.4: Compiling flow of MaxCompiler.

tablished via streams. APIs are generated by the MaxCompiler to control accelerators from applications on the host computer. The APIs are described in C language.

Figure 2.4 shows compiling flow of the MaxCompiler. There are high-level synthesis tools to generate FPGA designs based on data flow computing from descriptions in high level languages which are developed ahead of the MaxCompiler. For example, SRC Computers CARTE generates FPGA design from user descriptions in C or Fortran [13]. In CARTE, user descriptions directly express the form of data flow graph and the compiler tries to translate all the user description into the FPGA design. It imposes constraints heavily what user can describes.

In contrast, the MaxCompiler has a dedicated Java library for describing data flow graphs and user describes relationships between nodes of the graphs with the library. The data flow graphs for the compiler are generated by executing the user description. Therefore, user descriptions for the MaxCompiler do not reflect the structure of operations, but instructions to generate the structure of operation as a kind of domain-specific language. This is the radical difference between the CARTE and the MaxCompiler.

The benefits of the compiler are that user can use standard Java grammar and libraries to describe the data flow graph and execution performance of the code does not affect the performance of the generated data flow graph. These benefits allow us to write easy-to-read source code. One extension from the original Java grammar by the MaxCompiler is overload of operators to describe the graph in a direct way.

The MaxCompiler mainly intends to achieve high-throughput processing by making the best use of large-scale pipelines. Therefore, the Maxcompiler does not implicitly perform resource sharing which can reduce resource usage Users have to explicitly describe the code so that resources are shared, when it is needed.

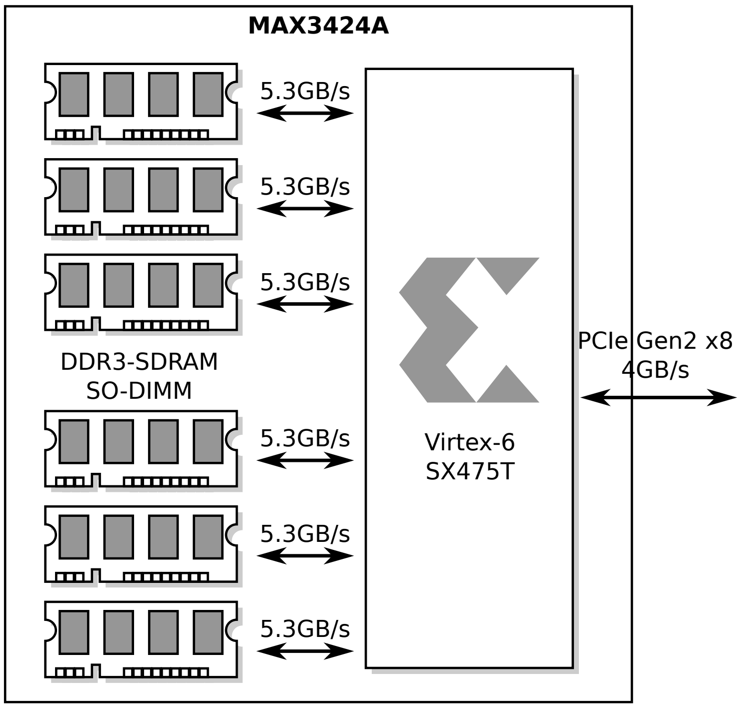

Figure 2.5: Architecture overview of MAX3424A

2.4.1 MAX3424A Data Flow Engine

FPGA accelerators supported by MaxCompiler are called Data Flow Engines (DFEs). The overview of MAX3424A DFE is shown in Fig. 2.5. There are a Virtex-6 SX475T FPGA, six 4 GB DDRIII memories which achieve 31.2 GB/sec peak memory bandwidth in total. The FPGA and the host processor are connected via PCIe gen2 x8 interface.

Chapter 3

Aims of This Study

This chapter describes challenges and aims of this study.

3.1 Challenges

3.1.1 Using On-chip memory

As already mentioned, both GPUs and FPGAs consist of many arithmetic elements and memory elements, and achieve high performance by working these elements in parallel. The on-chip memories provide us larger memory space than registers, and faster and lower latency memory space than off-chip memories. This study focuses highly efficient application mapping on GPUs and FPGAs using on-chip memories, especially for stencil computation and stream-oriented process as described below. Aims of this study include to demonstrate concrete application mappings from design patterns.

3.1.2 High-Level Synthesis Tools

Aspects of the high-level synthesis tools can be classified into two types; 1) Generating hardware design from software code for micro processors; and 2) Generating from their own language or macros. The former tools allow us to re-use existing software libraries, while the latter tools provide us hardware friendly description environment. Although the high-level synthesis tools allow us to design desired hardware by simple descriptions in a highly-abstracted layer, details of relationship between user description and generated hardware might be unclear. This study also aims to establish performance, resources usage and power consumption models from user parameters for stencil computations on the MaxCompiler. These models are expected to provide us reasonable estimations of computational performance and resource utilization from user descriptions in a highly-abstracted level. In addition, these models can be used to reduce development time by narrowing the range of the best user parameter space before synthesizing, placing and routing the designs. This is useful for development of large design especially when design needs a few days for synthesizing.

3.2 Target application category

3.2.1 Stencil computation

Stencil computations are a computational scheme widely used for scientific computations including electro magnetic simulation and computational fluid dynamics. Since the computations iterate the same operations for each point in a multidimensional grid, the computations have been known as its high degree of data-level parallelism. In addition, the computations have a shape of neighborhood dependency to update each grid point in an array. This performs some patterns of data accesses.

The stencil computations on GPUs are widely used for physical simulations. In these computation, all the arithmetic operators require to access the data with high spatial locality. Massive usage of on-chip memory as application specialized data cache improves computational performance. On the other hand, using boundary condition makes governing equations complex and thus proper assignment of computational resources and on-chip memories is one of the most important issues.

3.2.2 Stream-oriented process

Stream-oriented process is a computation aspect which focuses data streams instead of control flows or arithmetic. The computation aims for high efficiency of process by series of pipelined units to decrease random accesses for memories. Since the stream-oriented process expands operations spatially, it leads high operating frequency and simplified control flows. The stream-oriented process can be implemented based on stream-oriented architecture shown in Fig. 3.1 on FPGAs in particular.

The streamed architecture shown in Fig. 3.1 consists of three main parts: registers, a FIFO, and pipeline(s). The registers hold the data for pipeline parts and shift to next registers or a FIFO. The FIFO part holds the data that is not required by pipeline parts and passes the data to the next line of registers. The pipeline part does actual computation and outputs results for next data processing. One advantage of this architecture is any huge memory to store whole input frame data is not used; instead, FIFO to store only a few lines of data is needed. This is a preferable character also in term of energy efficiency. Many previous researches were reported which used this external memory-free architecture, especially for image processing [21–23].

The stencil computations are also widely used on FPGA for scientific simulation and computer visions. In addition, some other types of computations can be implemented as the stencil forms by transforming the definitions. The stream architecture is a key component to perform the stencil computations on FPGA. This approach for FPGA-based stream-oriented processing system to achieve faster, smaller and low energy consumption systems lacks enough knowledge about which applications are suited and how these applications can be mapped to FPGA.

3.3 Purpose and schemes

This dissertation addresses efficient mapping of concrete applications on GPUs and FPGAs based on the stencil computation and stream-oriented process. Details of each project are described as follows.

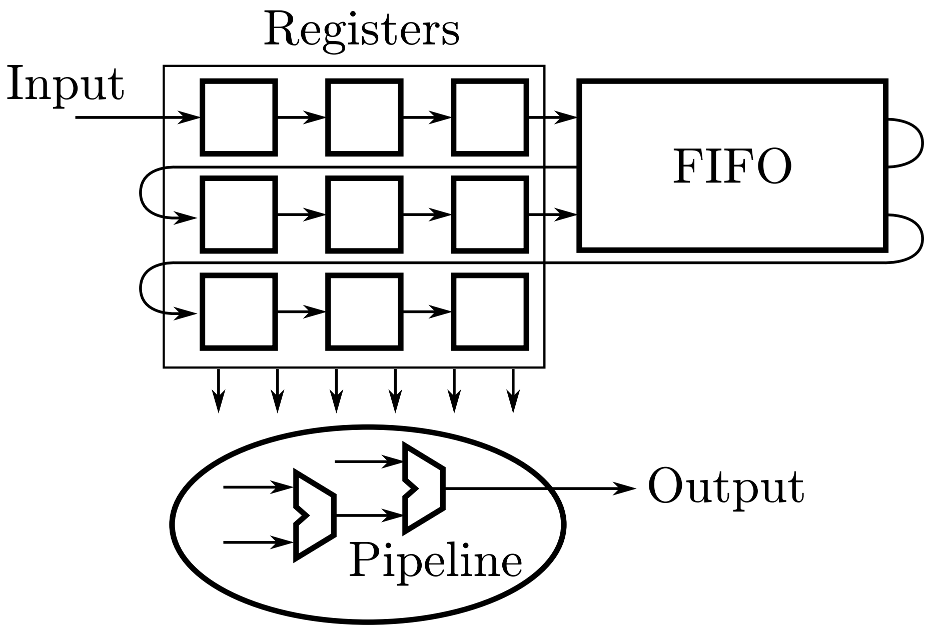

Figure 3.1: Streamed architecture

3.3.1 Smith-Waterman algorithm on a GPU

Smith-Waterman algorithm is a kind of biological sequential alignment algorithms. The algorithm can be computed using a dynamic programming scheme and its operations are massively parallelizable. In addition, the computation requires relatively high ratio of integer arithmetic to memory access. Therefore, costs of synchronizing arithmetic cores and on-chip memory access impact computational performance significantly. Implementing the algorithm on a GPU shows the importance of effective utilization of the warp-level parallelism on GPUs.

3.3.2 FDTD method for a micro strip antenna on a GPU

FDTD method discretizes the Maxwell's equation in spatial and time domains and calculates the electric and magnetic potential at each point on spatial grids or lattices. The algorithm is known as a kind of stencil computations that have a high degree of parallelism. This project addresses to accelerate 3-D FDTD method for antenna designing on a GPU. The implementation uses Perfectly Matched Layer (PML) as a boundary condition, hence memory accessing pattern and update-equations depend on location of grids. This project aims to reveal the implementation methodology the stencil computation with complex boundary conditions.

3.3.3 Human detection system using HOG features on an FPGA

Histograms of Oriented Gradients (HOG) features are extracted from luminance gradients of an image. It has high robustness for lighting conditions. This project aims to extract the HOG features from input video images, and to detect a human in a frame using Adaptive Boosting (AdaBoost) algorithm. In order to cope with a lot of computation costs of the HOG feature extraction, the system is designed in a data stream oriented deep pipelined manner. Performance evaluation includes computation performance, working frequency and power consumption.

3.3.4 Heat conduction simulation using a MaxCompiler

This project aims to define the user parameters for stencil computations on MaxCompiler and MaxGenFD. This project also aims to establish estimation models for computational performance and resource utilization. In addition, energy consumptions of the accelerator which is connected via PCIe to reveal relationship between the user parameters and energy consumption.

3.3.5 Comparison of pupil detection system implementation using HDL and MaxCompiler

This project aims to perform ellipse estimation from input video images using RANdom SAmple Consensus (RANSAC) algorithm. The algorithm needs to solve simultaneous equations as many as possible to get a reasonable solution. To solve simultaneous equations, three types of algorithms are implemented on an FPGA and compared in performance and resource utilization.

Chapter 4

Smith-Waterman Algorithm on GPUs

This chapter describes a multi-threaded parallel design and implementation of the Smith-Waterman (SW) algorithm on GPUs with NVIDIA corporation's Compute Unified Device Architecture (CUDA). Central to this is a divide and conquer approach which divides the computation of a whole pairwise sequence alignment matrix into multiple sub-matrices (or parallelograms) each running efficiently on the available hardware resources of the GPU in hand, with temporary intermediate data stored in global memory. Moreover, the implementation focuses to use thread warps and padding techniques in order to decrease the cost of thread synchronization, as well as loop unrolling in order to reduce the cost of conditional branches. While intermediate data is stored in global memory for large queries, the most inner loop in this implementation will only access shared memory and registers. As a result of these optimizations, this implementation of the SW algorithm achieves a throughput ranging between 9.09 GCUPS (Giga Cell Update per Second) and 12.71 GCUPS on a single-GPU version, and a throughput between 29.46 GCUPS and 43.05 GCUPS on a quad-GPU platform. Compared with the best GPU implementation of the SW algorithm reported to that date, this implementation achieves up to 46% improvement in speed.

4.1 Background

Biological sequence alignment is a widely used and crucial operation in bioinformatics and genetics research. The purpose of it is to find the best possible alignment of a set of sequences, with various real world applications including phylogenetic tree construction, disease diagnosis, and drug engineering. However, biological sequence alignment is also a computationally expensive application as its computing and memory requirements grow rapidly with the size of the databases and queries which leads to massive execution times on standard desktop computers.

Graphics Processing Units (GPUs) have been proposed recently as high performance and relatively low cost acceleration platforms for biological sequence alignment [24]. As modern GPUs have become increasingly powerful, inexpensive and relatively easier to program through high level API functions, they are increasingly being used for non-graphic or general purpose applications (the so-called GPGPU computing). This chapter will present how Compute Unified Device Architecture (CUDA)

GPUs can be used as hardware platforms to accelerate pairwise sequence alignment using the Smith-Waterman (SW) algorithm [25] [26]. The SW algorithm adopts a dynamic programming mechanism which provides the best local alignment between two sequences using exhaustive search, at the expense of a high computational load compared to less accurate heuristics-based algorithms such as the BLAST algorithm [27]. However, with faster computing platforms, this trade-off need not take place as the exhaustive SW algorithm can be implemented efficiently. The remainder of this chapter is organized as follows. First, relevant background on the SW algorithm is presented. Then, the CUDA programming model is presented. After that, this multi-threaded parallel design and implementation of the SW algorithm and pseudo code will be presented. A comparative evaluation between this implementation and the state-of-the-art of GPU implementations as well as various micro-processor based implementations is then presented before conclusions and ideas for future work are laid out.

4.2 The Smith Waterman Algorithm

The SW algorithm is a dynamic programming algorithm which finds the best local fragment between two biological sequences. The search for the optimal local alignment consists of two stages. Firstly, an alignment matrix of two biological sequences (e.g. protein, DNA) is calculated and results in a maximum score. After that, the alignment traces back from the maximum score until a zero element is found. Note that the final trace back procedure will be done only for one or few subject sequences, the one(s) with the highest scores, out of the most subject sequences, and hence is often performed on the host CPU.

More specifically, let $D = d_0 d_1 ... d_{m\ 1}$ denotes a database sequence with length m. Let $Q = q_0 q_1 ... q_{n\ 1}$ denotes a query sequence of length n. Let $W(a_i, b_j)$ denotes the substitution scoring matrix [28] , which gives a score describing the likelihood of substitution between characters a_i and b_j. Let G_{init} and G_{ext} denote penalties for opening a new gap and continuing an existing gap respectively.

With the above, the alignment matrix computation of the SW algorithm is described by the following equations:

$$\begin{aligned}
E_{i,j} &= \max\begin{cases} H_{i,j\ 1} \quad G_{init} \\ E_{i,j\ 1} \quad G_{ext} \end{cases} \\
F_{i,j} &= \max\begin{cases} H_{i\ 1,j} \quad G_{init} \\ F_{i\ 1,j} \quad G_{ext} \end{cases} \\
H_{i,j} &= \max\begin{cases} 0 \\ E_{i,j} \\ F_{i,j} \\ H_{i\ 1,j\ 1} + W(a_i, b_j) \end{cases}
\end{aligned}$$

The values of $H_{i,j}$, $E_{i,j}$ and $F_{i,j}$ are defined as 0 if $i < 0$ or $j < 0$. The gap penalty is called linear if $G_{init} = G_{ext}$, otherwise, it is called affine. From these equations, it is observed that the value of $H_{i,j}$ depends on the values of its upper neighbor $H_{i,j\ 1}$, left neighbor $H_{i\ 1,j}$ and left-upper neighbor $H_{i\ 1,j\ 1}$, as shown in Figure 4.1.

The above operations are massively parallelizable since the anti diagonal elements of the alignment matrix are independent of each other (as labeled with the dot pattern in in Figure 4.1), and hence can be computed in parallel. In addition, the computation of

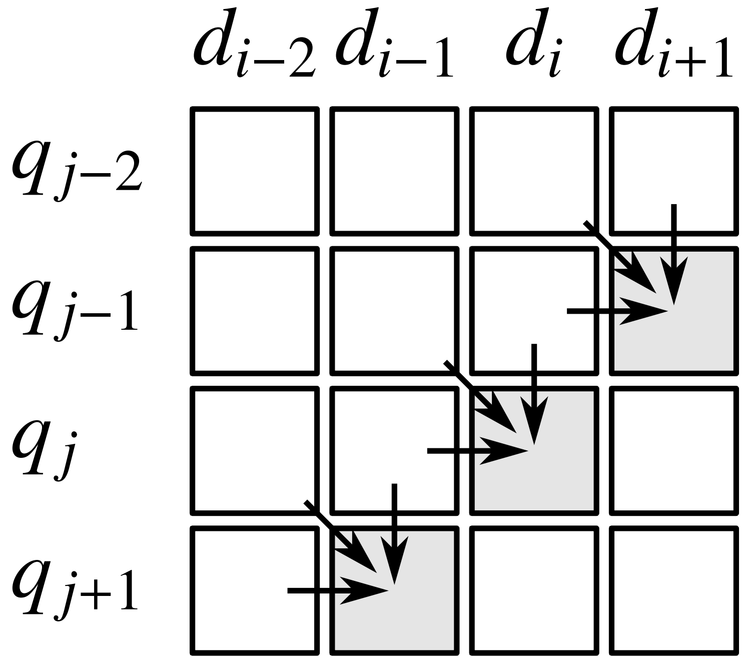

Figure 4.1: Data dependency of the SW dynamic programming algorithm. Arrows indicate dependency for the lower right element.

different alignment matrices between a query sequence and different subject sequences can also be done in parallel. Since GPUs have the ability of allocate thousands of parallel threads to a particular task, it is a very appealing acceleration platform for the SW algorithm. The GPU parallelization task is hence focused on the alignment matrices' calculation.

4.3 Implementation

There are many SW algorithm implementations on various parallel platforms [29] [30] [31] [32] [33] [34], and one of the fastest CUDA-compatible GPU implementations was presented by Liu et al. in [35]. This tried two types of implementations: "Inter-task parallelization" and "Intra-task parallelization". The "Inter-task parallelization" is a method whereby each thread processes one alignment matrix between a database sequence and a query sequence. By contrast, the "Intra-task parallelization" is a method whereby a thread block processes one alignment matrix between a database sequence and a query sequence. In this implementation, each warp is used to process one alignment matrix between a database sequence and a query sequence. To use this technique, it is important to reduce the resource usage per thread, the frequency of global memory access per matrix calculation, and the frequency of the synchronization statement calls.

4.3.1 Calculation method

As already discussed, each warp is used to process one matrix. Since the number of threads per thread block is set to 256, a total of eight matrices are processed by one thread block. This number depends on the target GPU architecture. In general, if the number of threads per thread block is too small or too large, the performance decreases significantly. Moreover, the number of threads per thread block must be as a multiple of 32 [17].

Furthermore, the cost of conditional branch is quite high in GPU, especially when some threads within the same warp take different instruction flows as the threads within warp are synchronized like in a SIMD manner i.e. all threads within a warp have to take the same instruction flow in reality. As shown in Section 4.2, the core of the SW algorithm itself is not very complicated, and consists of two nested loops with simple additions and a maximum function of 2 values. To reduce conditional branches, a traditional approach (other optimization methods have been proposed e.g. in [31] [33].) is used. In the approach of this section, each sequence is padded with null entries(NEs) to have the overall length of a multiple of 32, and the resulting alignment matrix is divided into parallelograms as shown by Figure 4.2. Regions where padded null entries exist lead to decreased performance because the cost of ineffective calculation increases.

The transfer of data, H and F, between each parallelogram is done using global memory and the transfer of data within the parallelogram is done by using shared memory. To use shared memory, each parallelogram was divided to 32-column sub-parallelograms as shown in Figure 4.3. The pseudo code for the calculation of the data from a sub-parallelograms (the most inner loop calculation) is shown in Figure 4.4. The prefixes “r”, “s”, “t” and “c” relate to register, shared memory, texture memory and constant memory, respectively. *r_Max* is the maximum *r_H* score that each thread processes. Data that have to be passed to the downside large parallelogram, H and F are the first 32 elements of *s_H* and *s_F*, stored in global memory. Data that have to be passed to the next sub-parallelogram are the second 32 elements of *s_H* and *s_F* stored in shared memory. *r_H_diag*, *r_Max* and *r_E* are also passed to the next sub-parallelogram. *tid* is the unique ID of each thread and *r_score* is the score of the substitution matrix. Note that global memory is not used within the most inner loop as it is slower than shared memory and registers. After calculating all sub-parallelograms, the alignment score is calculated by returning the maximum *r_Max* of each thread within the warp to the host.

The data that is passed to the next sub-parallelogram (*r_E* and *r_H_diag*) propagates in the lateral direction. Thus, it can be held by registers (since each thread processes data laterally and this data is local to each thread).

4.3.2 Shared memory usage and occupancy

The GPU has 16 KB shared memory and 16 K registers per thread block. All of these resources are open to users, but the “Occupancy” is one of the considerations [17] [18]. Occupancy is the ratio of the number of active warps to the maximum number of warps that each multi-processor can process. The maximum number of warps is determined by GPU hardware. In the case of GT200b, it is 32 warps. The number of active warps is determined by 3 factors: number of threads per MP, register usage and shared memory usage. The register and shared memory usages can be obtained by compiling results. A higher the occupancy, the busier the hardware. So this metric is very important for performance measurement. To get high occupancy, low utilization of shared memory and register is required. In the implementation of this chapter, there are 256 threads per thread block and 6216 bytes of shared memory per thread block and 24 registers per thread are used. Therefore the occupancy is 0.5, and the number of active thread blocks in each MP is 2. As a result of this, it was determined the number of thread

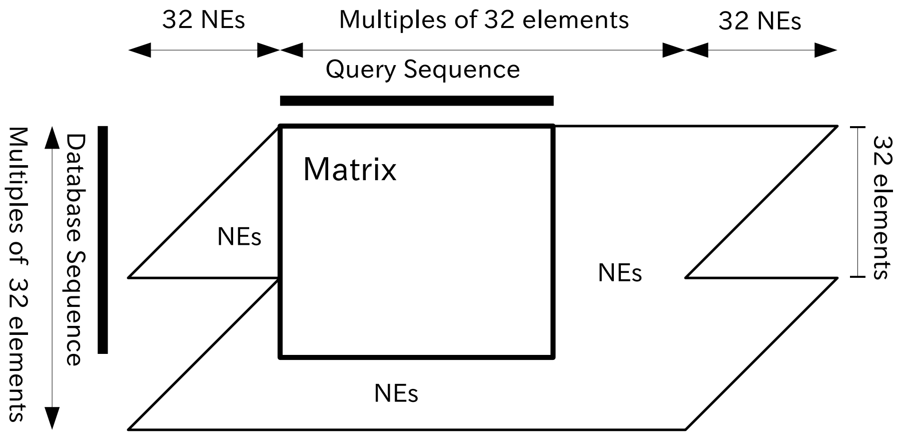

Figure 4.2: Region of calculation of the SW kernel and padding entries of query and database sequences. The square that is enclosed by a heavy line shows alignment matrix.

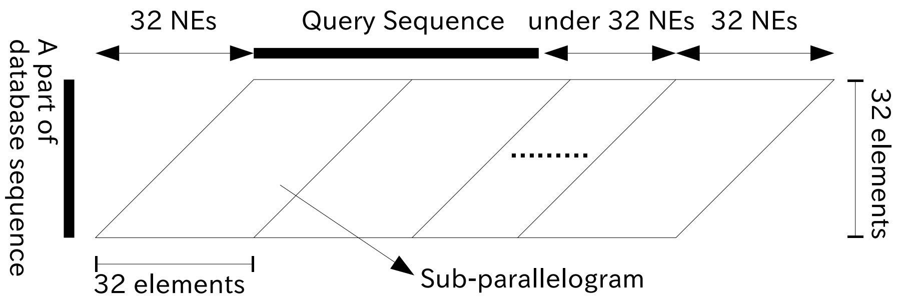

Figure 4.3: Dividing parallelogram to sub-parallelogram. The most inner loops in the SW kernel calculate each sub-parallelogram.

```
for i = 0 to 31 do
    // Load score
    r_query_idx =
    s_query[tid + i + r_query_offset]
    r_score =
    t_score[r_query_idx][r_subject_idx]
    // Calculate F
    r_F = max(
    s_F[tid + i]    c_Gap_extend,
    _H[tid + i]    c_Gap_init)
    // Calculate H and Max
    r_H =
    max(0, r_E, r_F, r_H_diag + score)
    r_H_diag = s_H[tid + i]
    r_Max = max(r_H, r_Max)
    // Store H and F
    s_H[tid + i] = r_H
    s_F[tid + i] = r_F
    //Calculate next E
    r_E =
    max(r_E    c_Gap_extend,
    r_H    c_Gap_init)
end for
```

Figure 4.4: The pseudo code for the most inner loop of SW kernel, when there are 32 threads per warp. The *r_E*, *r_H_diag*, *s_H* and *s_F* are initialized to 0 at the start of kernel execution.

blocks per kernel call to be 60 to make all multiprocessors busy.

In previous SW implementation studies, it was revealed that the maximum bit width of H, F and E is less than 16 bits in most real world applications [34]. In that case, short type variables (i.e. 16 bits) can be used to store these values. This would decrease memory I/O traffic and hence improves performance. However, this also brings about memory bank conflicts if shared memory is used in a straightforward way as shown in Figure 4.5 [17]. It can be easily packed H and F to 32-bit variables as there are some vector types like a “short2” in CUDA API.

4.3.3 Using multiple GPUs

In GTX295, there are two GPU cores in one module. Additionally, some motherboards have multiple PCI-Express x16 bus interfaces, hence allowing for multiple GPUs in one node. OpenMP gives users an environment to use multiple GPUs in one process. A straightforward gain is achieved by dividing the database into multiple chunks and allocating each chunk to one of the GPU cores to process them in parallel.

Since the execution time depends on the length of the query sequence and database sequence, it is very important to balance the execution time between parallel GPUs to achieve high efficiency. For this, the database was divided into equal-length subsets. An easy way to do this dynamically is to assign the longest sequence to the GPU that has the shortest total length of the already assigned sequences. This does not guarantee

```
// (a)
__shared__ short s_H[8][64];
__shared__ short s_F[8][64];
// (b)
__shared__ short2 s_HF[8][64];
// (c)
__shared__ short s_HF[8][128];
// (d)
__shared__ int s_H[8][64];
__shared__ int s_F[8][64];
```

Figure 4.5: Examples of shared memory definition. (a): A bank conflict occurs because two adjacent elements e.g. s_H[0] and s_H[1] sit in the same bank with different threads attempting to access them at the same time. (b): There is no bank conflict here when some threads access each element of the array. (c): Interleaving H and F within a array, there is no bank conflict here and there is no need for packing. (d): There is no bank conflict here, and there is no need for packing. However, a larger memory is needed compared to the other definitions.

the optimal distribution but resulting imbalance in subset sizes is negligible for large databases (490,000 sequences in this implementation).

4.3.4 Other optimization issues

In addition to the aforementioned optimization techniques, this section presents further optimizations for this implementation. First, loop unrolling is used, which is a very well-known optimization technique that is effective in reducing conditional branching overheads [36].

The second optimization technique used is optimized reduction used to find the maximum element of the alignment matrix to be returned to the host. The optimized reduction method used is described in [19]. In it, reduction is achieved with no synchronized statement which improves its execution time considerably.

The third optimization technique used is concurrent memory copy and execution. To use this technique, the implementation separated the calculation sequence to three steps. First, the kernel call step which is the core calculation task executed on the GPU and is the most time consuming step. The results of kernel call are stored in global memory. Second, data is transferred from global memory to the host memory. In order to parallelize data copy and kernel execution, the Stream asynchronous memory copy statement and management organization [17] was used. For this, a double buffer is used in the global memory whereby read/write from/to the buffer is toggled at each iteration.

Third, data loaded from global memory to the host is sorted and tied up with the sequence names in the database file. Here again, the implementation uses a double buffer on the host to allow for concurrent execution of data sorting and data loading from global memory.

Table 4.1: Performance Evaluation of loop unrolling and concurrent memory copy using Blosum50 scoring matrix with a penalty of 10-2k.The performance is in GCUPS.

Queries	Length	Default	Unrolling	Memory copy	Both	Total Speed-up[%]
P02232	144	7.17	8.30	7.71	9.04	26.18
P01111	189	8.38	9.72	8.94	10.48	25.13
P05013	189	8.38	9.72	8.94	10.48	25.14
P09488	218	8.64	10.03	9.15	10.73	24.23
P14942	222	8.79	10.22	9.31	10.92	24.25
P00762	246	8.83	10.27	9.30	10.91	23.58
P10318	362	9.45	11.02	9.81	11.52	21.90
P07327	374	9.77	11.39	10.14	11.90	21.90
P01008	464	10.05	11.73	10.36	12.17	21.14
P10635	497	10.18	11.90	10.49	12.31	20.92
P25705	553	10.24	11.97	10.51	12.35	20.60
P03435	567	10.50	12.27	10.78	12.66	20.61

4.4 Results and Evaluation

The cell updates per second (CUPS) is a commonly used measure for SW execution performance as it normalizes in terms of the database and query sequence sizes. Given a query sequence Q and a sequence database D, the Giga CUPS (GCUPS) is defined as below:

$$GCUPS = \frac{\|Q\| * \|D\|}{t * 10^9}$$

where $\|Q\|$ represents the length of the query sequence, $\|D\|$ the total length of the database sequences, and t means the elapsed time in seconds. In this project, the elapsed time t includes the transfer time of the query sequence from the host to the GPU global memory, the calculation time of the SW algorithm scores on the GPU, result data transfer time from global memory to host, and finally score sorting on the host. The performance of the implementation of this chapter is evaluated by 12 query sequences with lengths ranging from 143 to 567 (see Table 4.1). These sequences include query sequences that are widely used in the literature to test sequence alignment algorithms, including Striped Smith-Waterman [29] and other Smith-Waterman implementations [30] [35]. All queries were run against Swiss-Prot 57.6 which comprises 174,780,353 amino acids in 495,880 sequence entries. Blosum50 scoring matrix was used with a gap penalty of 10-2k. GPU implementations were carried out on a GTX 295 graphics card, which has 30 SPs and 896 MB RAM per GPU, installed on a PC with a Core 2 Duo E7200 2.53 GHz processor, 2 GB DDR2 800 MHz, and running Cent OS 5.0. This graphics card has a core frequency of 576 MHz and a memory clock of 999 MHz.

4.4.1 Optimization techniques

This section evaluates further optimization techniques as shown in Section 4.3.2 and 4.3.4. Note that the shared memory definition used is the method (d) in Figure 4.5.

First, the effects of the loop-unrolling and concurrent memory copy techniques are shown by Table 4.1. The loop-unrolling optimization unrolled 16 iterations resulting in a loop of 2 iterations, and register usage increased from 20 to 22. Both optimization

Table 4.2: Performance Evaluation of shared memory using Blosum50 scoring matrix with a penalty of 10-2k.The performance is in GCUPS.

Queries	(a)	(b)	(c)	(d)
P02232	6.85	8.53	8.54	9.04
P01111	7.94	9.89	9.90	10.48
P05013	7.94	9.89	9.89	10.48
P09488	8.13	10.13	10.13	10.73
P14942	8.28	10.31	10.31	10.92
P00762	8.27	10.30	10.30	10.91
P10318	8.72	10.88	10.87	11.52
P07327	9.01	11.23	11.23	11.90
P01008	9.21	11.49	11.49	12.17
P10635	9.32	11.62	11.62	12.31
P25705	9.34	11.65	11.64	12.35
P03435	9.58	11.94	11.94	12.66

techniques worked well. By unrolling the most inner loop of the SW algorithm, the evaluation result shows 17 % speed up in execution time. Additionally, by using concurrent copy and execution, performance improvement achieved an extra 9 % speed-up of the SW execution. Overall, the evaluation results show over 26 % speed-up by using both of techniques. Because of the use of loop-unrolling, the amount of register usage increased, but this did not affect the occupancy in this instance.

Table 4.2 shows the performance of the four definitions of shared memory data shown in Figure 4.5. This shows method (d) to be the fastest. Method (a) which has the bank conflict problem is the slowest, whereas methods (b) and (c) are in the same range. There is no bank conflict in methods (b) and (c), so it would appear that reason behind the performance drop is not memory bandwidth, but rather the additional computing cost due to data packing or index calculations. Although methods (a), (b) and (c) are slower than method (d), resource usage is lower. In this instance, there is no need to reduce the amount of shared memory usage but methods (b) and (c) are better than method (d) when this is needed.

4.4.2 Evaluation of SW implementation

The implementation results are shown in Table 4.3. For a single GPU implementation, it achieved the highest performance of 12.71 GCUPS, and the lowest performance of 9.09 GCUPS. For a quad-GPU implementation, it achieved the highest performance of 43.05 GCUPS, and the lowest performance of 29.46 GCUPS. The gap penalty has no effect on performance. Next evaluation compares the performance of the implementation of this chapter with CUDASW++-2.0b2 which is one of the fastest SW implementations on CUDA-compatible GPUs [35]. The evaluation is performed with the Swiss-Prot database release 57.6, using Blosum50 scoring matrix with a gap penalty of 10-2k. The results are shown in Figure 4.6. Apart from the case of query P02232, the implementation of this chapter is clearly faster than the CUDASW++-2.0b2 (with the case of query P02232 achieving a similar performance). In the multiple GPU implementation of CUDASW++, the database sequences are transferred for each query sequence. By contrast, the implementation of this chapter transfers the database sequences just one

Table 4.3: Performance Evaluation using swiss-prot release 57.6 and Blosum50 scoring matrix with a penalty of 10-2k.The performance is in GCUPS.

Queries	Length	1 GPU	2 GPUs	4 GPUs
P02232	144	9.09	17.26	29.46
P01111	189	10.53	19.98	34.44
P05013	189	10.53	19.98	34.42
P09488	218	10.78	20.45	35.44
P14942	222	10.97	20.83	36.09
P00762	246	10.96	20.84	36.23
P10318	362	11.57	22.01	38.78
P07327	374	11.96	22.76	40.07
P01008	464	12.21	23.25	41.19
P10635	497	12.36	23.54	41.19
P25705	553	12.39	23.61	41.97
P03435	567	12.71	24.22	43.05

time. It is main the reason for the relatively large performance difference shown on multiple GPU implementation.

Finally, an evaluation shows comparison of the performance of proposed implementation with several recent CPU-based implementations by Farrar [29] [31]. Farrar reported the performance of a SW implementation on a number of processors including Xeon and Cell B.E. The comparison used the performance of Farrar's implementation on two further processors, the first using a PC with a Core 2 Duo E7200 2.53 GHz processor, 2 GB DDR2 800 MHz SDRAM running CentOS 5.0, and the second using a PC with a Core i7 CPU 920 2.67 GHz, 4 GB DDR3 1066 MHz running the CentOS 5.0. Comparison results are shown in Figure 4.7. The data of Xeon and Cell B.E. implementations were taken from Farrar's paper [31]. In the case of a single GPU implementation, the proposed implementation is slower than CPU-based implementations for all query sequences, whereas the multiple GPU-based implementations are faster than the CPU-based implementations for all query sequences.

4.5 Summary

This chapter has presented a novel technique for the implementation of the Smith-Waterman algorithm on CUDA-compatible GPUs.

Compared with previous GPU implementations that calculate the alignment matrix by thread blocks or threads, proposed implementation in this chapter focused on warp level synchronization to decrease the cost of synchronization. Central to this technique is a divide and conquer approach to alignment matrix calculation in which a whole pairwise alignment matrix is subdivided into parallelogram regions. This leads to the efficient calculation of the alignment matrix by 32 threads, and the reduction in the number of loads/stores to/from global memory. Shared memory data definition was very important in order to avoid bank conflicts as the implementation accesses shared memory more frequently (instead of global memory). In addition, this project showed that some other optimization techniques, namely loop-unrolling and double-buffering

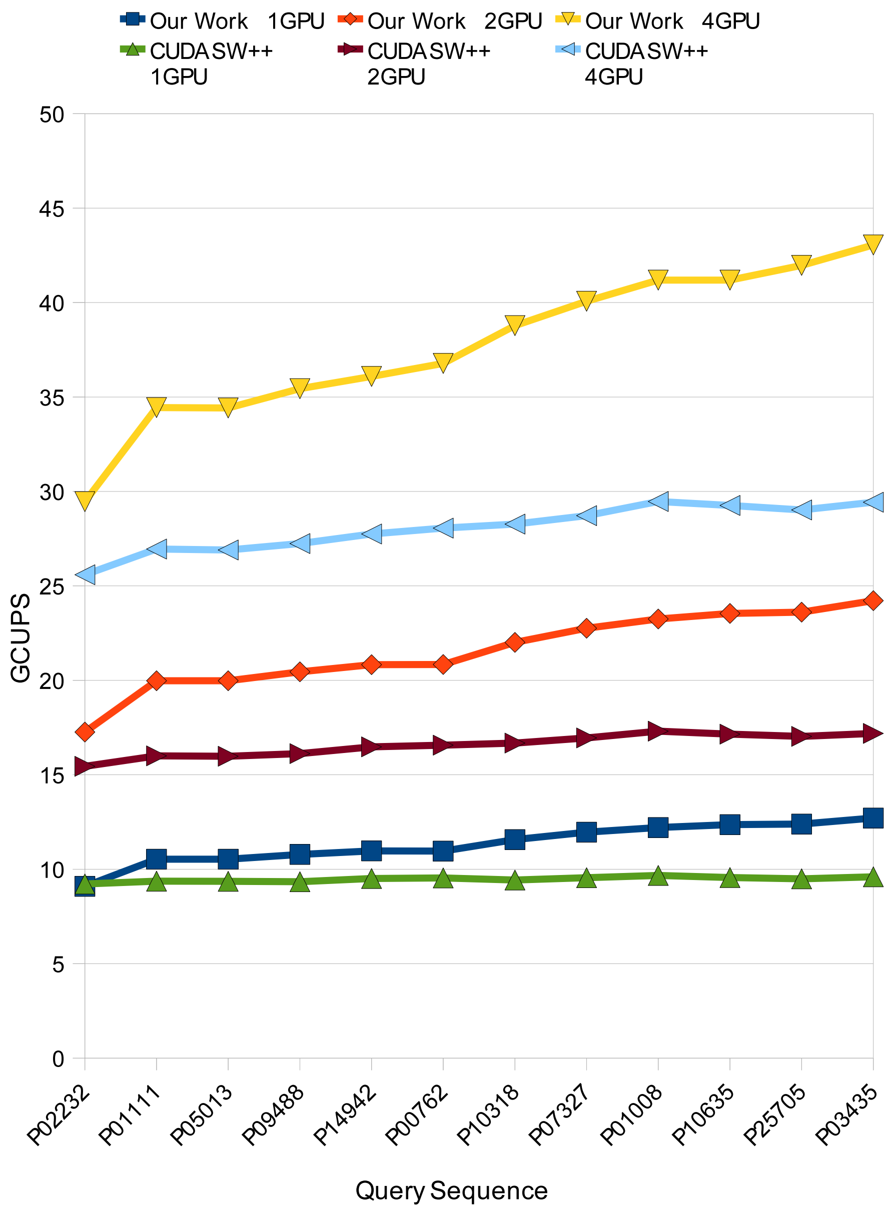

Figure 4.6: Performance comparison between the implementation of this chapter and CUDASW++-2.0b2 using swiss-prot 57.6 and Blosum50 scoring matrix using a penalty of 10-2k.

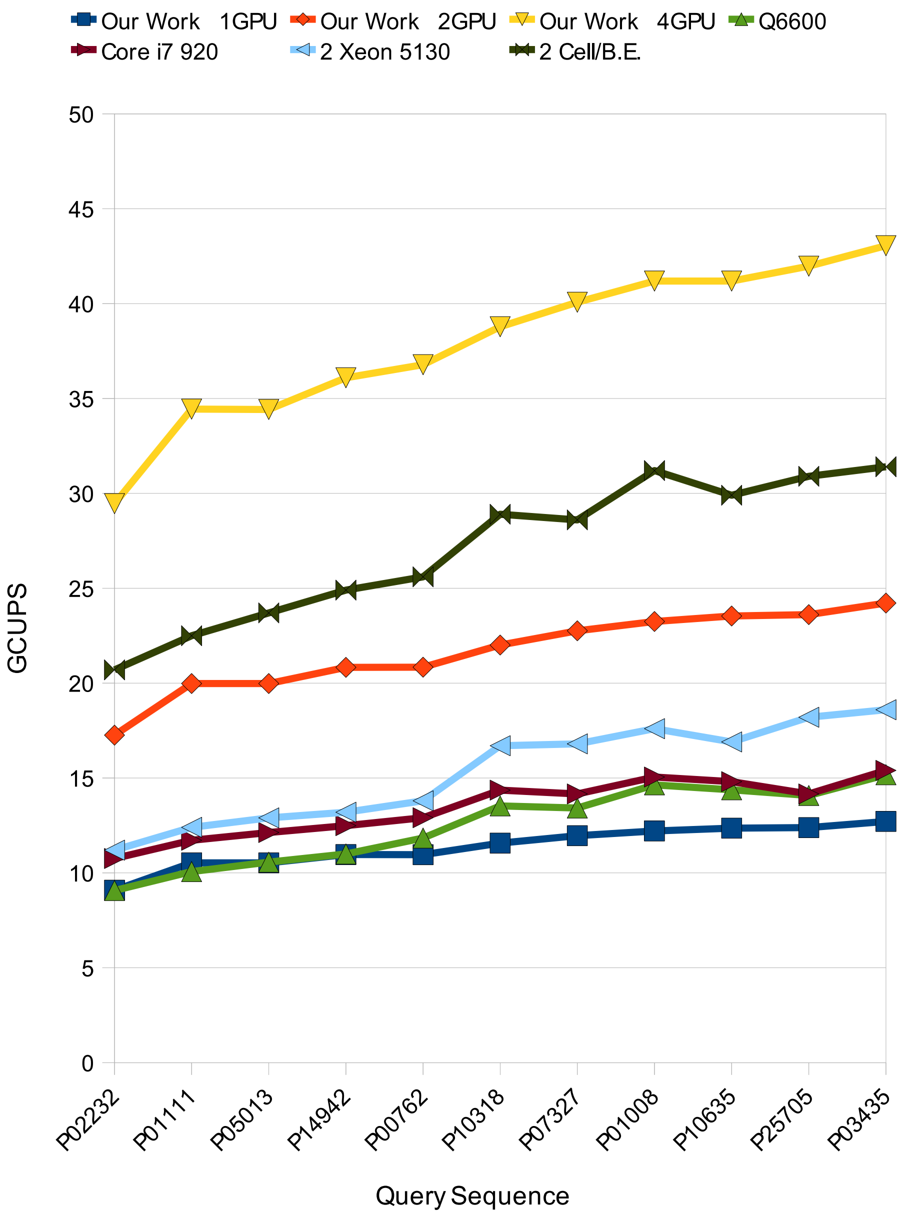

Figure 4.7: Performance comparison between the implementation of this chapter and the striped Smith-Waterman implementation using Blosum50 scoring matrix with a penalty of 10-2k. The CPU implementations used all cores.

to use concurrent memory copy and execution, which are used widely for CPU code optimization, worked very well for GPUs. This project also presented the performance of the SW implementation using multiple GPUs. When 4 GPUs are used, for instance, the performance is much higher than the Farrar's implementation that used two Cell B.E. processors (50 % more), at a much lower cost (75 % less). It could have used more GPUs to scale up the implementation even further if there were more PCI-Express slots on the motherboard.

Chapter 5

FDTD method on a GPU

5.1 Introduction

This chapter discusses optimization techniques to implement 3D Finite-Difference Time-Domain (3D-FDTD) method with Absorbing Boundary Conditions (ABC) on a GPU, for accelerating analysis of antenna characteristics.

Characteristics analysis for antenna design consists of two steps: simulation of electromagnetic propagation in time domain and analysis of simulation results. Acceleration of simulation is needed since most of the execution time is occupied by the electromagnetic simulation. The FDTD method proposed by Yee discretizes the Maxwell's equation in spatial and time domains and calculates the electric and magnetic potential at each point on spatial grids or lattices [37] [38]. Although its computational costs, which are the number of floating point operations and memory accesses in this case, and memory usage increase linearly with the size of a simulation model, the method is widely used since performance of computers have been rapidly improved and the algorithm is simple and easy to understand.

Absorbing boundary conditions(ABCs) are important when using the FDTD method for unbounded problems. Since computers cannot handle an infinite space or infinite number of elements on the grid, the method can only be used within a finite space. To resolve this limitation, several types of ABCs were proposed to attenuate reflection waves from boundaries of the computational space [39] [40] [41]. Among them, the perfectly matched layer(PML) proposed by Bérenger is widely used [42]. However, the PML needs additional floating point operations, memory accesses, and memory resources.

The FDTD method is known as a kind of stencil computation that has a high degree of parallelism but requires a large memory bandwidth. While GPU implementation is attractive as a cost-effective acceleration approach, earlier work has shown that GPU implementation of Absorbing Boundary Conditions (ABCs) tends to be a bottleneck of the simulation [43]. This chapter discusses this issue. In concrete terms, the main contributions of this chapter are as follows:

≤ Efficient GPU implementation of a Non-Uniform grid method is presented to reduce memory requirements and its performance overhead is revealed.

≤ Novel implementation of ABCs is proposed and evaluated, which is coupled with periodic boundary conditions in view of a SIMD nature of GPUs.

≤ An implementation technique based on transformation of update equations of ABCs is introduced to reduce both of the computation amount and memory usage.

The proposed ideas are empirically evaluated by practical simulation of a micro strip antenna (MSA) in terms of performance and accuracy.

The rest of this chapter is organized as follows. Section 5.2 introduces related works. Section 5.3 describes the 3D-FDTD method, the Split PML and basic implementation of 3D-FDTD method on a GPU. Section 5.4, 5.5 and 5.6 describe proposals for the GPU implementation. The memory usage, performance, accuracy and comparison with related works are evaluated in Section 5.7. Finally, Section 5.8 includes the conclusions and some directions for future work.

5.2 Related works

So far, many researchers have reported the results of GPU acceleration of the stencil computation [44] [45], including 3D-FDTD method with ABCs [6] [46] [5]. Nagaoka *et al.* reported a performance comparison a 3D-FDTD method with the Split PML on a CPU, SX-8R super computer and Tesla C1060 GPU [46]. They used a human body model as a calculation target and pointed out that the performance varied with the calculation domain and CUDA thread block size. Implementation of a 3D-FDTD method was evaluated also on a variety of GPUs including CUDA incompatible GPUs [6] . Chu *et al.* achieved the performance of approximately 160 M elements updates per second using UPML [5]. While these works addressed parallelization and acceleration of a 3D-FDTD method for a normal computational space, there have been few reports that mainly discuss implementation techniques of ABCs and a non-uniform gird on GPUs.

5.3 Background

5.3.1 3D FDTD Method

This section briefly describes a 3D FDTD simulation approach in this chapter. Detailed mathematical background can found in other literatures [37] [38]. The approach assumed that a simulation target is isotropy and non-dispersive having the conductivity of $\sigma = 0$. It also assumed permeability is uniform. After applying a typical substitution of central differences for the time and space derivatives for Maxwell's curl equations, time-stepping expression of the approach can be written as for E_x for instance:

$$\begin{aligned} E_x^n\left(i+\frac{1}{2},j,k\right) =& E_x^{n\quad 1}\left(i+\frac{1}{2},j,k\right) \\ &+ \frac{\Delta t}{\varepsilon_x\left(i+\frac{1}{2},j,k\right)\Delta y}\delta_y H_z^{n\quad\frac{1}{2}}\left(i+\frac{1}{2},j+\frac{1}{2},k\right) \\ &\quad \frac{\Delta t}{\varepsilon_x\left(i+\frac{1}{2},j,k\right)\Delta z}\delta_z H_y^{n\quad\frac{1}{2}}\left(i+\frac{1}{2},j,k+\frac{1}{2}\right) \end{aligned} \tag{5.1}$$

where Δy and Δz are cell sizes of each dimension, ε_x is a permittivity in x dimension, Δt denotes the time increment, and $\delta_u(u\ \lfloor\ \}x,y,z|)$ is a difference operator. For example, δ_y is defined as $\delta_y f(i,j,k) = f(i,j,k) \quad f(i,j \quad 1,k)$.

5.3.2 Equations for the Split PML

Since the 3D-FDTD method handles a finite computational space, the computational space is generally surrounded by zero values that correspond the Perfect Electric Conductor (PEC). The PEC reflects waves as a result of computation, and thus ABC are required to attenuate the reflection waves.

As previously mentioned, boundary condition used Bérenger's Split PML [47]. Each component of $\boldsymbol{E}$ and $\boldsymbol{H}$ is split to two components called subcomponents, and the time-stepping equations of each subcomponent are defined as:

$$E^{n}_{xy}\left(i+\frac{1}{2},j,k\right) = C_{E1y}(j)E^{n-1}_{xy}\left(i+\frac{1}{2},j,k\right) + C_{E2y}(j)\delta_y H^{n-\frac{1}{2}}_{z}\left(i+\frac{1}{2},j+\frac{1}{2},k\right) \quad (5.2)$$

where $C_{E1u}(p)$ and $C_{E2u}(p)(u \in \{x,y,z\})$ are introduced for ease of describing the equations. $C_{E1u}(p)$ and C_{E2u} are defined as

$$C_{E1u}(p) = \frac{2\varepsilon_0 - \sigma_u(p)\Delta t}{2\varepsilon_0 + \sigma_u(p)\Delta t}, u \in \{x,y,z\} \quad (5.3)$$

$$C_{E2u}(p) = \frac{2\Delta t}{2\varepsilon_0 + \sigma_u(p)\Delta t} \times \frac{1}{\Delta u}, u \in \{x,y,z\} \quad (5.4)$$

where ε_0 shows permittivity of free space and $\sigma_u(p)$ shows conductivity at the coordinate p on the dimension u. Note that $\sigma_u(p)$ is 0 regardless of the value of p in non PML regions. Since $C_{E1u}(p)$ and $C_{E2u}(p)$ are constants in the time domain, these values can be calculated in advance of the simulation process. The Split PML needs two subcomponents for each component of $\boldsymbol{E}$ and $\boldsymbol{H}$. Thus, the Split PML needs up to 12 additional memory elements compared to normal FDTD calculation.

5.3.3 Basic Implementation of 3D FDTD Method on CUDA-compatible GPU

A general approach to implementation of the 3D-FDTD method on CUDA-compatible GPUs divides the whole simulation space to fixed size *blocks* and makes CUDA thread blocks process each *block* [44] [45]. An implementation in this chapter also divides the whole simulation target of size (S_x, S_y, S_z) into small *blocks* of size (B_x, B_y, B_z) and makes CUDA thread blocks process each *block*. Each component of $\boldsymbol{E}$ and $\boldsymbol{H}$ is stored memory as 3D-array so that dimension x has a unit-stride, dimension y has a larger stride, and dimension z has the largest stride. Fig.5.1 shows the placement of CUDA threads within a *block* and direction of processes. $B_x * B_y$ CUDA threads within a CUDA thread block are placed on a 2D plane and each CUDA thread moves the process along the line with z direction. Therefore, the CUDA thread on the coordinate (x, y) processes a total of B_z cells from $(x, y, 0)$ to $(x, y, B_z - 1)$. When CUDA threads that have the same y coordinate in a block access to components of $\boldsymbol{E}$ and $\boldsymbol{H}$ at the same time, these accesses can be coalesced because these components are continuous on the GPU memory. Since the indices expressed in fractional numbers as like in Eq. (5.1) are inconvenient for straightforward implementation, a set of offset values make the coordinates integer numbers as shown in Table 5.1. Among possible candidates of offset values, offsets in Table 5.1 are for the $\boldsymbol{E}$ and $\boldsymbol{H}$ in this chapter so that accesses to the $\boldsymbol{E}$ and $\boldsymbol{H}$ components are coalesced as much as possible. Thus, Eq. (5.1) can be

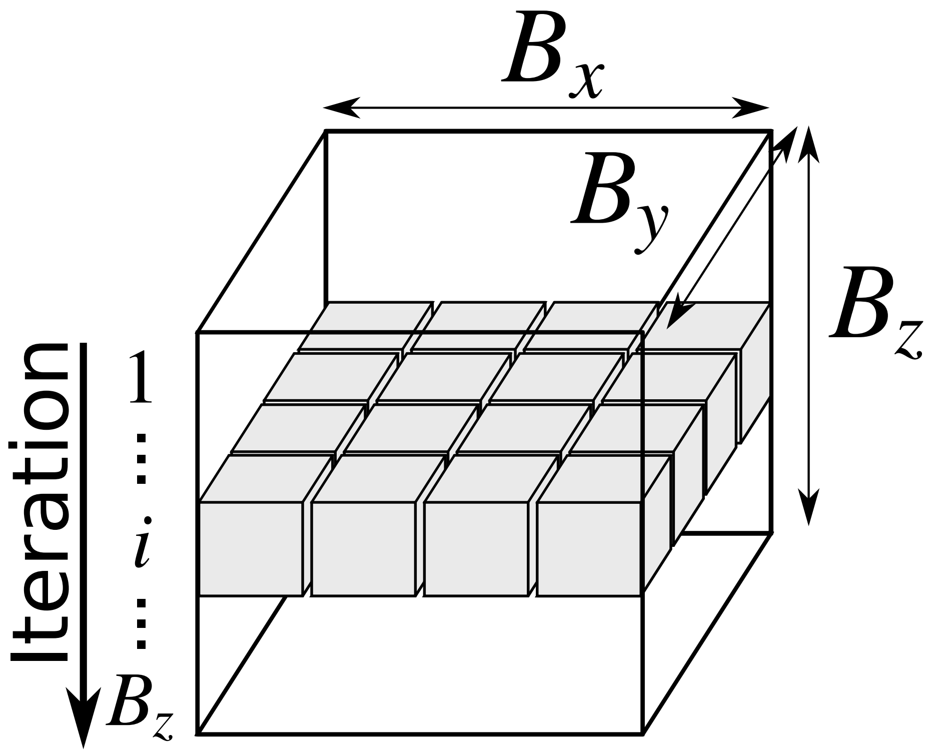

Figure 5.1: Placement of CUDA threads and direction of process within a *block. The gray boxes show CUDA threads.*

Table 5.1: Offsets for components of $\boldsymbol{E}$ and $\boldsymbol{H}$

	Components		
Field	x	y	z
$\boldsymbol{E}$	$(\ 1/2, 0, 0)$	$(0,\ 1/2, 0)$	$(0, 0,\ 1/2)$
$\boldsymbol{H}$	$(0,\ 1/2,\ 1/2)$	$(\ 1/2, 0,\ 1/2)$	$(\ 1/2,\ 1/2, 0)$

rewritten as

$$\begin{aligned} E_x^n\,[i,j,k] =& E_x^{n\ \ 1}\,[i,j,k] \\ &+ \frac{C_{E2y}[j]}{\varepsilon_{xr}[i,j,k]}\delta_y H_z^{n\ \ \frac{1}{2}}\,[i,j,k] \\ &\ \ \frac{C_{E2z}[k]}{\varepsilon_{xr}[i,j,k]}\delta_z H_y^{n\ \ \frac{1}{2}}\,[i,j,k] \end{aligned} \tag{5.5}$$

and the update equation of the Split PML for E_x can also be rewritten as

$$\begin{aligned} E_x^n\,[i,j,k] =& C_{E1y}\,[j]\,E_{xy}^{n\ \ 1}\,[i,j,k] \\ &+ C_{E2y}\,[j]\,\sigma_y H_z^{n\ \ \frac{1}{2}}\,[i,j,k] \\ &+ C_{E1z}\,[k]\,E_{xz}^{n\ \ 1}\,[i,j,k] \\ &\ \ C_{E2z}\,[k]\,\sigma_z H_y^{n\ \ \frac{1}{2}}\,[i,j,k]\,. \end{aligned} \tag{5.6}$$

where coordinates in square bracket intend the coordinates with the offsets. Hereafter, the converted coordinates as shown in Eq. (5.5) and (5.6) are used to describe the implementation.

5.4 The Non-Uniform grid

5.4.1 Introduction of the Non-Uniform grid

Since the size of the FDTD simulation is limited by available size of memory on a GPU device in effect, the liner interpolated Non-Uniform grid [48] [49] is introduced to reduce memory usage for simulation. By using multiple sizes of simulation cells, the Non-Uniform grid approach aims at effective reduction of the memory usage, floating point operations and data memory accesses while sustaining the simulation accuracy. The computation error caused by the use of Non-Uniform grid is discussed in [49]. Applying a linear interpolation to Eq. (5.4), $C_{E2u}(p)$ is described as:

$$C_{E2u}[p] = \frac{2\Delta t}{2\varepsilon_0 + \sigma_u[p]\Delta t} \times \frac{2}{\Delta u[p\ \ 1] + \Delta u[p]}, \qquad u \lfloor\ \}x, y, z| \tag{5.7}$$

where $\Delta u[p]$ is a cell size of coordinate p in dimension u.

5.4.2 Implementation

Important things when introducing the Non-Uniform grid are that how much memory usage can be reduced and how many additional costs, such as floating point operations and data memory accesses, are required. Here, the additional costs to use the method are discussed. In the case of the Uniform grid, it is possible to use $\Delta t/(\varepsilon_0 \Delta u)$ instead of $C_{E2u}[p]$ in Eq. (5.1). Because $\Delta t/(\varepsilon_0 \Delta u)$ is a constant, it need not be fetched from the global memory. Therefore, the additional memory access cost for the Non-Uniform grid is made by fetches of six float values; $C_{E2u}[p]$ and $C_{H2u}[p], (u \lfloor \}x, y, z|)$, that is $6 * 4 = 24$ Bytes, per cell.

Since $C_{E2u}[p]$ and $C_{H2u}[p]$ are constant values through the simulation, the constant memory or texture memory can hold these values instead of the global memory to mitigate the access penalty. In addition, appropriate thread placement and blocking can make further reduction of the accesses. As described in Section 5.3.3, each CUDA thread has the invariant coordinate (x, y) in proposed implementation. Therefore, each CUDA thread needs to fetch the values for $C_{E2x}[i]$, $C_{H2x}[i]$, $C_{E2y}[j]$ and $C_{H2y}[j]$ only once when the CUDA kernel is launched. Moreover, the values of $C_{E2z}[k]$ and $C_{H2z}[k]$ can be shared by all of the threads within the same CUDA thread block, so that a total of B_z fetches are required per CUDA thread block while processing a *block* of (B_x, B_y, B_z).

To summarize, the Non-Uniform grid method needs additional $4 * (2B_x + 2B_y + 2B_z)$ data fetches from the arrays of $C_{E2u}[p]$ and $C_{H2u}[p](u \lfloor \}x, y, z|)$ per *block*. Therefore, the additional memory access amount per grid point becomes

$$\frac{4 * (2B_x + 2B_y + 2B_z)}{B_x * B_y * B_z} \text{ Bytes,} \tag{5.8}$$

which can be mitigated by an appropriate blocking size.

5.5 Novel Absorbing Boundary Condition

5.5.1 Motivation

As already shown, the Split PML as an ABC increases the number of memory accesses and arithmetic operations per cell due to the subcomponents. Generally, the ABC forms thin regions that are smaller than the size of *block*. When computation for the ABC regions and other regions are allocated into the same *block*, instruction and data flows make divergences which results in severe performance degradation due to a SIMD property of GPU architectures. To avoid this situation, the ABC regions and other regions should be placed in different *blocks*. However, in turn, since the ABC regions are thin, the ABC *blocks* need a lot of padding cells to align the blocks especially in the dimensions with a large block size. In addition, at least two ABC *blocks* are needed per dimension. While the padding cells consume resources, they do not make any contributions to the computation. However, the threads assigned for the padding cells execute the same instruction flow as the ones for the ABC regions to keep the SIMD property and thus to avoid the thread divergence in the *block*.

5.5.2 Introduction of the periodic boundary condition

In order to reduce the excess padding cells, this section proposes a novel implementation of an ABC introducing the idea of the periodic boundary condition. In the periodic boundary condition, one side of the simulation space continues to the other side. Thus, using this periodicity, the ABC regions can be gathered on one side of each dimension.

The periodic boundary condition can be introduced by changing the definition of the difference operator δ_u in Eq. (5.1). For example, δ_y is defined as

$$\delta_y f(i, j, k) = f(i, j, k) \quad f(i, (j + S_y \quad 1)\, mod\, S_y, k) \tag{5.9}$$

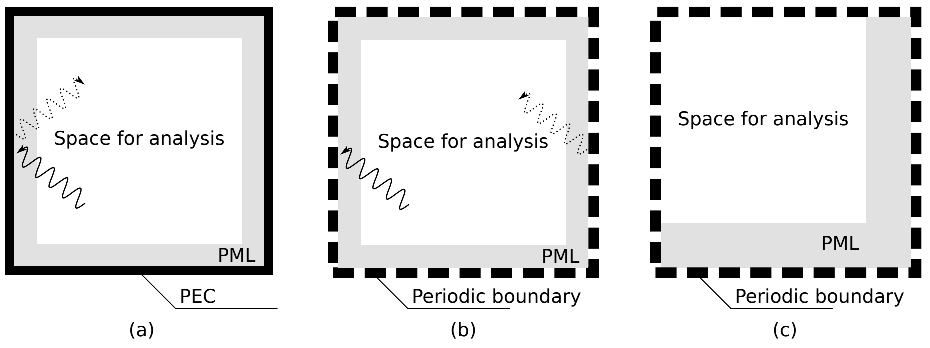

Figure 5.2: (a) The model that has the Split PML and PEC. A dashed line shows reflected wave before attenuation. (b) The model that has the periodic boundary condition instead of the PEC. (c) The proposed model.

where S_y is the size of dimension y.

In the original configuration as shown as Fig.5.2a, emitted waves are finally reflected by the PEC, but before that, they are sufficiently attenuated by Split PMLs so as not to affect the simulation results. Therefore, it is possible to use the periodic boundary condition instead of the PEC as show in Fig.5.2b, since waves that leap the boundary of the simulation space should be also enough weak not to affect the simulation. Additionally, as shown in Fig. 5.2c, the Split PMLs can be gathered on one side in each dimension for efficient blocking.

To summarize, by introducing the periodic boundary condition, the number of the ABC *blocks* for each dimension can be reduced from 2 to 1, and thus the number of padding elements can also be reduced. Note that the number of cells used for the Split PMLs is not changed by introducing the periodicity.

5.6 Transformation of update equations of the Split PML

5.6.1 Motivation

It is independent in terms of dimension if a cell is in an ABC region or not. For example, there can be a cell that is in an ABC region in dimension z but not in dimension y.

For such cells, transformation of update equation of the Split PML Eq. (5.6) is proposed to reduce the required memory size and access amount at the cost of slight increase in the operation count, considering that the simulation size is limited by the available memory size on the GPU device.

Table 5.2: Comparison of memory access requirements of each update equation for E_x. NON_ABC shows the equation that both dimensions are in the normal region, ORIGINAL shows the original equation of the Split PML and TRANSFORMED shows the transformed equation, respectively.

	NON_ABC	ORIGINAL	TRANSFORMED
Read	5	6	6
Write	1	3	2
Total	6	9	8
Ratio	1	1.5	1.333

5.6.2 Transformation

Given that a cell that is inside the normal space region in dimension y, the transformation provides the following equation:

$$\begin{aligned} E_x^n[i,j,k] = & E_{xy}^{n\quad 1}[i,j,k] \\ & + C_{E2y}[j]\,\sigma_y H_z^{n+\frac{1}{2}}[i,j,k] \\ & + C_{E1z}[k]\,E_{xz}^{n\quad 1}[i,j,k] \\ & \quad C_{E2z}[k]\,\sigma_z H_y^{n+\frac{1}{2}}[i,j,k] \end{aligned} \tag{5.10}$$

by substituting $\sigma_y[j] = 0$ to Eq. (5.6). The definition of subcomponents [47] provides the subcomponent $E_{xy}^{n\quad 1}$ as:

$$E_{xy}^{n\quad 1}[i,j,k] = E_x^{n\quad 1}[i,j,k] \quad E_{xz}^{n\quad 1}[i,j,k]. \tag{5.11}$$

By substituting Eq. (5.11) into Eq. (5.10), $E_x^n[i,j,k]$ is defined as:

$$\begin{aligned} E_x^n[i,j,k] = & E_x^{n\quad 1}[i,j,k] \\ & + C_{E2y}[j]\,\sigma_y H_z^{n+\frac{1}{2}}[i,j,k] \\ & + (C_{E1z}[k] \quad 1)\,E_{xz}^{n\quad 1}[i,j,k] \\ & \quad C_{E2z}[k]\,\sigma_z H_y^{n+\frac{1}{2}}[i,j,k]. \end{aligned} \tag{5.12}$$

Comparing Eq. (5.6) and (5.12), it is clear that the transformation can reduce the number of accesses to the subcomponent E_{xy} as well as the total memory usage. On the other hand, while original Eq.(5.10) includes the intermediate term which can be reused for $E_{xz}^n[i,j,k]$, the transformed Eq.(5.12) does not, which means this techniques required one additional operation for each cell. In this sense, this transformation approach is oriented for memory constrained devices like GPUs.

Table 5.2 shows the comparison of memory access of update equations for E_x. Since the transformed equation saves one write access compared to the original equation, this technique can improve the performance for the region that only one dimension is in the Split PML. Applying the same transformation to E_y and E_z provides a total of $2^3 = 8$ time-stepping equations for $\boldsymbol{E}$. In addition, it is also provided similar 8 equations for $\boldsymbol{H}$. Note that these transformed equations are mathematically equivalent to the original ones, so that simulation accuracy is not affected.

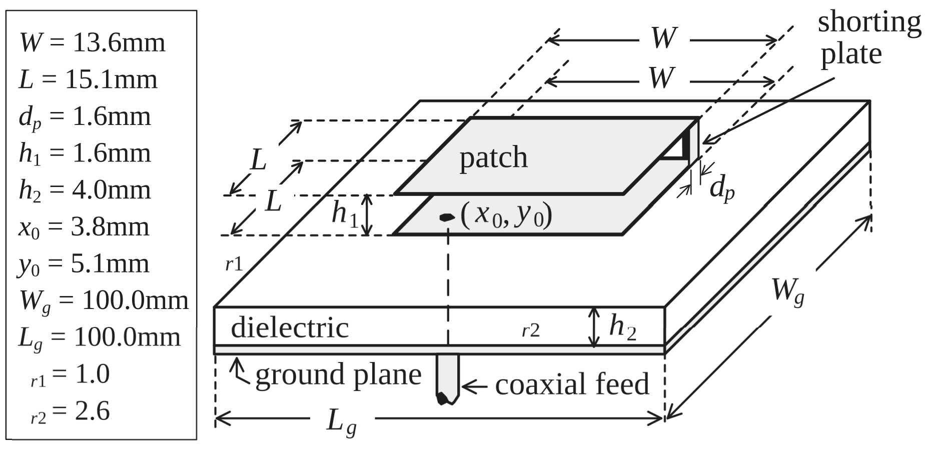

Figure 5.3: The geometry of a stacked rectangular MSA.

5.7 Performance Evaluation and Analysis

5.7.1 Running Example

As a target example of numerical calculation, a stacked rectangular microstrip antenna (MSA) with a shorting plate which has been proposed for dual band operation in [50], was chosen. MSAs are widely used in mobile communications due to their lower profile, weight, and manufacturing costs, as well as their compatibility with integrated circuit technologies. The MSA proposed in [50] radiates a linearly polarized wave at the lower frequency band and a circularly polarized wave at the higher frequency band.

Fig. 5.3 shows the geometry of a stacked rectangular MSA with a shorting plate. The antenna consists of a dielectric substrate and a layer of air with a rectangular patch. The upper and lower patches are the same size. The upper patch is shorted to the lower patch at the apex by a conducting plate. The relative dielectric constant the upper and lower layer are $\varepsilon_{r1} = 1.0$ and $\varepsilon_{r2} = 2.6$. The antenna is excited at the lower patch by a coaxial feed through the lower dielectric substrate at point which lays around the diagonal.

5.7.2 Modeling

Various previous work have reported the *block* size has a strong effect on the performance [43]. Although the best size may depend on the presence or absence of each optimization technique, it used the *block* size of (B_x, B_y, B_z)=(32, 4, 32) based primary evaluation results.

Fig. 5.4 shows the initial simulation model that is based on the MSA shown in Fig. 5.3 with 10 layers of the Split PML. A total of (256 $*$ 216 $*$ 128) cells were used to make the Non-Uniform grid, where 5 kinds of cell sizes were used; 1 mm to 0.1 mm. The size of the *block* is (32, 4, 32) as aforementioned. S_x, S_y and S_z in Fig. 5.4 show the number of cells in each dimension and P_x, P_y and P_z are the number of cells in each dimension of the ABC with padding.

Table 5.3 shows the number of *blocks*, the number of cells and memory usage for

$$
\begin{aligned}
S_x &= 192\,(6)\\
S_y &= 192\,(48)\\
S_z &= 64\,(2)\\
P_x &= 10_{(PML)} + 22_{(padding)}\,(1)\\
P_y &= 10_{(PML)} + 2_{(padding)}\,(3)\\
P_z &= 10_{(PML)} + 22_{(padding)}\,(1)
\end{aligned}
$$

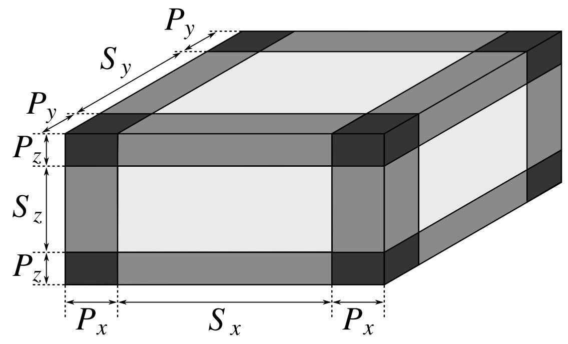

Figure 5.4: Numbers that provided in parentheses show the number of *blocks*. The color of each region shows the group of update equations and darker colors show higher computational costs for memory accesses. 10-cell thick PMLs are used for the boundaries of each dimension. Since the *block* size is 32 in both x and z dimension, 22 padding cells are required for both P_x and P_z. On the other hand, since the *block* size is four in y dimension, three *blocks* are used for P_y to cover the 10-cell thick PMLs, introducing two padding cells.

Table 5.3: The number of *block*, cells and memory usage.

region	# of *blocks*	# of cells	rate	memory	ratio
NON_ABC	576	2,359,296	.333	54.0 MB	.143
PML1	840	3,440,640	.486	236.3 MB	.625
PML2	288	1,179,658	.167	81.0 MB	.214
PML3	24	98,304	.014	6.8 MB	.018
Total	1,728	7,077,888	1	378.1 MB	1

each region of the model. All cells was classified into the four categories according to presence or absence of the ABC; all dimensions are not in the ABC (NON_ABC), any one of three dimensions is in the ABC (PML1), any two of the three are in the ABC (PML2), and all the dimensions are in the ABC (PML3). Approximately 67 % of the cells including padding are in the ABC.

Evaluations in this chapter used an implementation platform with the GeForce GTX 295, which has 30 Streaming Processors and 896 MB GDDR3 memory per GPU core. While the GPU has two GT200b GPU cores, proposed implementation used only one GPU core in this implementation. Single precision floating point operations were utilized through the simulation.

5.7.3 Effect of Non-uniform grid

First, the Non-Uniform Grid method was evaluated in terms of memory usage and execution time. Table 5.4 shows comparison results of the Global memory usage on the GPU. In the case of the Uniform Grid, all the cells have the same size as the finest cells in the case of the Non-Uniform Grid. Although it depends on a simulation model how much memory usage can be reduced, the reduction of 1/36 was achieved in this running example.

Table 5.4: Comparison of memory usage between the Non-Uniform grid and Uniform grid in the running example.

	# of cells	memory usage
Uniform grid	$(1,120 * 1,100 * 224)$ + PML	13,584.6 MB
Non-Uniform grid	$(192 * 192 * 64)$ + PML	378.1 MB

Table 5.5: Execution time for the Non-Uniform grid and the Uniform grid. Both methods processed the same number of cells.$(256 * 212 * 128)$

	Execution time (sec)
Uniform	155.81
Non-Uniform	155.86

Next, the penalty of the Non-Uniform grid was evaluated. As already shown in Section 5.4.2, additional costs of using the Non-Uniform grid arisen from additional fetches of $C_{E2u}[p]$ and $C_{H2u}[p]$ in non ABC regions. To evaluate this penalty, modifications are applied to the implementation to fetch $C_{E2u}[p]$ and $C_{H2u}[p]$, and to use the constant values of $(\Delta t/\varepsilon_0)$ and $(\Delta t/\mu_0)$ instead of them. Table 5.5 shows the result using the simulation model shown in Fig. 5.4. Note that these evaluations focus only on revealing the performance difference here and thus this modification does not preserve the simulation results.

The results show that the Non-Uniform grid degrades the simulation performance per cell by about 0.03% due to additional fetches of $C_{E2u}[p]$ and $C_{H2u}[p]$. In view of the 1/36 reduction of required number of cells, the implementation approach of the Non-Uniform method is efficient for both reduction of memory usage and performance improvement.

5.7.4 The periodic boundary condition with ABC

Fig. 5.5 shows the simulation model with the periodic boundary condition. Compared to the model shown in Fig. 5.4, the number of *blocks* of ABC is smaller in all dimensions. Table 5.6 shows the number of *blocks*, cells and memory usage. Clearly, the periodic boundary condition reduces memory usage. Figs. 5.6a and 5.6b illustrate the execution time of these models. By reducing cells for the ABCs, the periodic ABC achieved 1.8 times faster performance than the Non-Periodic ABC.

The periodic boundary condition significantly reduces the computational region, thus both memory usage and computational performance are improved. The boundary condition was concerned that costly memory access with a long stride would decrease the performance. However, as experiment results show, the method enabled efficient implementation on a GPU.

5.7.5 Effect of Transformation

Table 5.7 shows the number of *blocks*, the number of cells and memory usage after the transformation of the update expressions. The transformation reduces memory usage

$S_x = 192\,(6)$
$S_y = 192\,(48)$
$S_z = 64\,(2)$
$P_x = 20_{(PML)} + 12_{(padding)}\,(1)$
$P_y = 20_{(PML)}\,(5)$
$P_z = 20_{(PML)} + 12_{(padding)}\,(1)$

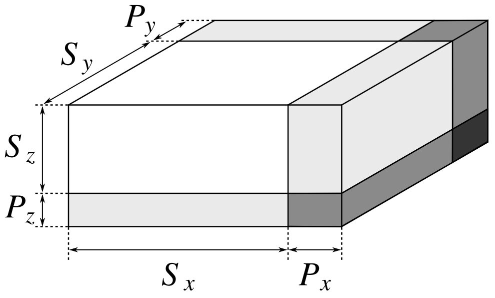

Figure 5.5: The simulation model with the periodic boundary condition to the model shown in Fig. 5.5. Using the periodic boundary condition with ABC, 20 PML cells in total are assigned to P_x and P_z, reducing the nubmer of padding cells 22 to 12. In y dimension, five *blocks* are required for P_y to cover the 20 PML cells.

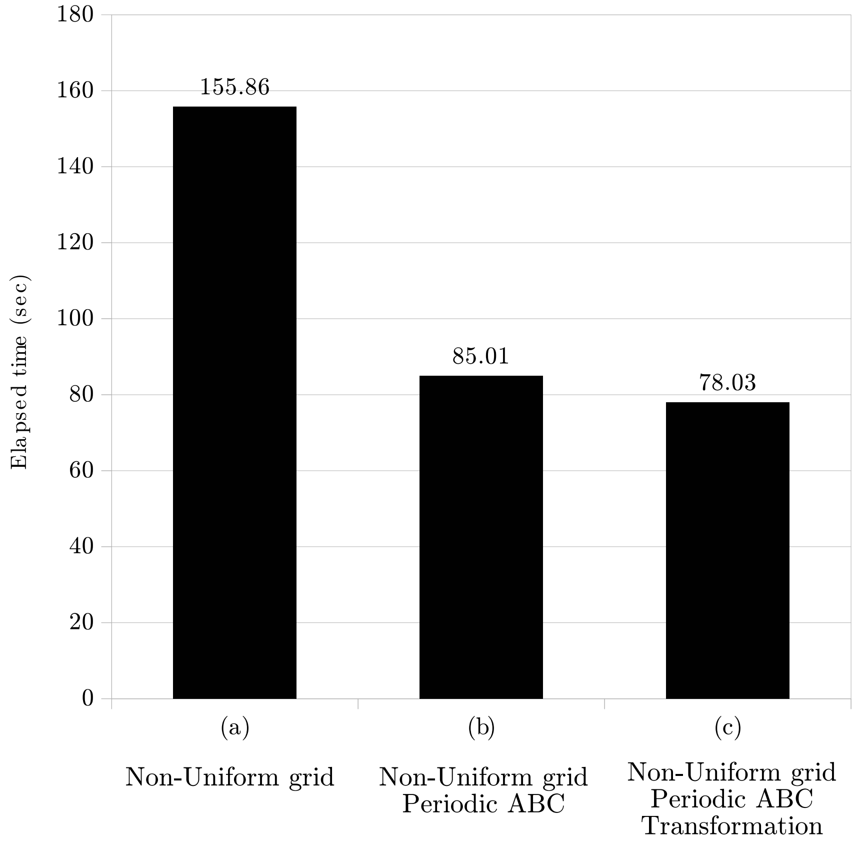

Figure 5.6: Execution time of each implementation.

Table 5.6: The number of *block*, cells and memory usage with the non-uniform grid and the periodic boundary condition.

region	# of *blocks*	# of cells	ratio	memory	ratio
NON_ABC	576	2,359,296	.518	54.0 MB	.263
PML1	444	1,818,624	.399	124.9 MB	.609
PML2	88	360,448	.079	24.8 MB	.121
PML3	5	20,480	.004	1.4 MB	.007
Total	1,033	4,558,848	1	205.1 MB	1

Table 5.7: The number of *block*, cells and percentage of total with the Non-Uniform grid, the periodic boundary condition and transformation of update equations.

region	# of *blocks*	# of cells	ratio	memory	ratio
NON_ABC	576	2,359,296	.518	54.0 MB	.375
PML1	444	1,818,624	.399	69.4 MB	.482
PML2	88	360,448	.079	19.3 MB	.134
PML3	5	20,480	.004	1.4 MB	.010
Total	1,033	4,558,848	1	144.1 MB	1

of PML1 and PML2 so that the proposed implementation needs only 70 % of memory usage in the case of the model shown in Table 5.6.

Figs. 5.6b and 5.6c summarize the performance comparison results for the transformation of the expression. In the running example, the transformation of the Split PML improved the execution performance by 8.9 %.

5.7.6 Simulation Accuracy

In order to evaluate the simulation accuracy of proposed implementation techniques, the characteristics from simulation results are compared with the measured results by the real antenna.

In analysis of antennas, radiation and feed point characteristics are discussed generally. Figs. 5.7 and 5.8 show the calculated axial ratio of a circularly polarized wave as radiation characteristic and return loss as feed point characteristic at the higher frequency band in the dual band MSA, respectively. The measured results are also shown for comparison. The relative error of the frequency at the minimum axial ratio between the calculated and measured results is 0.5 %. The calculated and measured minimum axial ratios are 0.82 dB and 0.43 dB, respectively. In the return loss, the double peaking behaviors are observed in both of the calculated and measured results. The relative errors of the frequencies at the two peaks between the calculated and measured results are 0.3 % and 0.7 %. The calculated return losses at the two peaks are -20.6 dB and -27.6 dB and the measured ones are -19.6 dB and -26.9 dB. In both of the axial ratio and the return loss, the calculated results agree well with the measured ones.

First, in order to assess how periodic boundary condition affected the simulation accuracy, an implementation with the original non-periodic boundary condition was compared with the proposed implementation. As Figs. 5.7 and 5.8 show, slight differences were observed in the trends of axial ratios, but the peak frequency found was identical. On the other hand, the difference in the boundary condition hardly affected the results in terms of the return loss.

Next, aiming at examining numerical error in the simulation, the simulation results were compared with two different grid models with coarse cells: cells with double size (coarse grid #1) and cells with quadruple size (coarse grid #2) in each dimension. That is, these coarse grid models consist of 1/8 and 1/64 cells in total compared to the original model, respectively. As Figs. 5.7 and 5.8 show, the finer the cell size, the more accurate results were obtained for both the axial ratios and the return losses, suggesting the reasonability of proposed simulation scheme.

Final evaluation of the proposed implementation in simulation accuracy explored how arithmetic precision affected the simulation accuracy, by executing the simulation with double precision arithmetic. As shown in Figs. 5.7 and 5.8, almost no difference were observed between the simulation results for the single precision and double precession; the maximum relative error was less than 0.007%. This suggests single precision arithmetic offers enough simulation accuracy as far as the application field is concerned.

5.7.7 Memory bandwidth

Next, achieved memory bandwidth of the proposed implementation was evaluated, since this often tends to become a performance bottleneck for GPU implementation of stencil computation. The simulation model consists of three parts as follows:

≤ MODEL: A main part including the MSA

≤ PML: The Split PML part

≤ PADDING: Padding part.

Fig. 5.9 shows achieved memory bandwidth of whole application and ratios of each part.

MODEL, PML and PADDING accounted 42.3%, 36.8% and 20.9% of the total achieved memory bandwidth, respectively. These ratios differ a little from the ratios of the number of cells shown in Table 5.7. This should come from the fact that the process for PADDING cells is the same as that of the PML cells, and that the PML needs 1.4↤1.9 times greater memory access than the normal cells.

Compared with the peak memory bandwidth of the target GPU, proposed implementation achieved about 55.8% of the peak bandwidth when the accesses for PADDING region are not taken account. If PADDING accesses are included, the achieved bandwidth becomes 70.5% of the peak. These high memory bandwidth or efficiency of 3-D FDTD implementation with ABCs for the practical model is higher than previous works. This suggests the effectiveness of the proposed techniques.

5.7.8 Performance Comparison

Table 5.8 shows performance comparison with related work. Note that Table 5.8 shows peak bandwidth for one of two GPU cores on the GeForce GTX295 because the proposed implementation used only one GPU core. Comparison metrics are chosen carefully to be fair comparison with other implementations which have different parame-

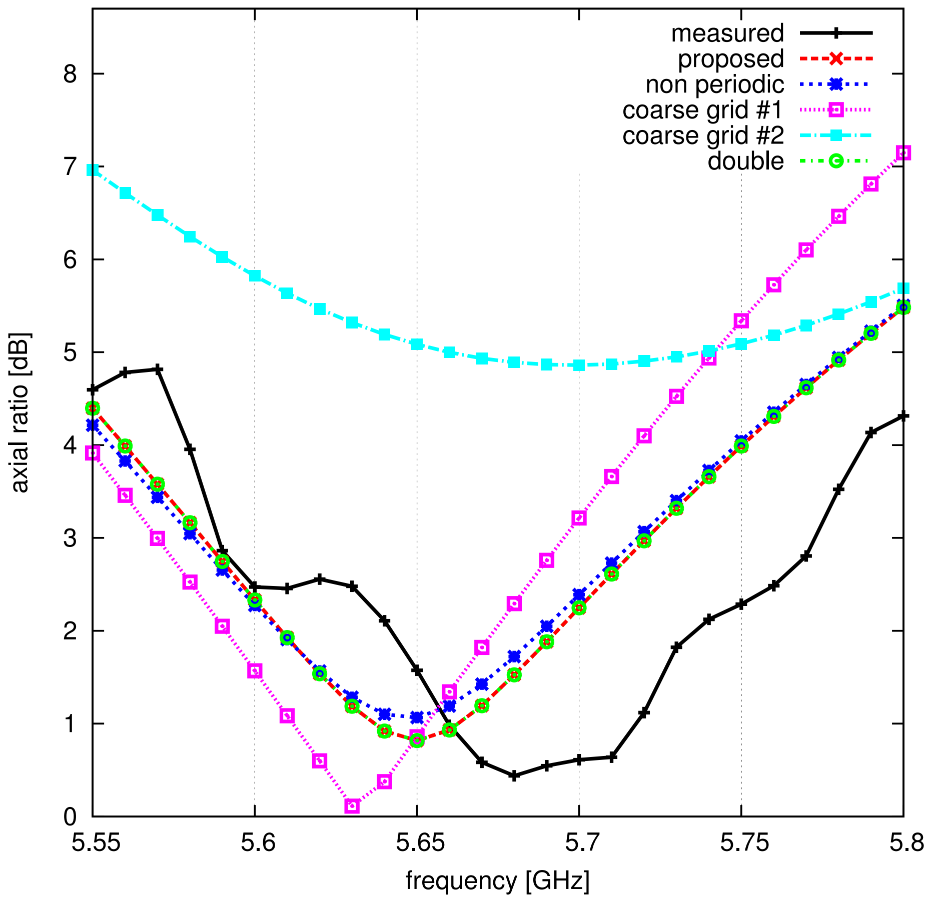

Figure 5.7: Comparison of minimum axial ratio between the calculated and measured results. ‘measured’: Measured data from actual antenna. ‘proposed’: Simulation results with the proposed method. ‘non periodic’: Simulation results with non periodic ABC. ‘coarse grid #1’: Simulation results with double sized cells. ‘coarse grid #2’: Simulation results with quadruple sized cells. ‘double’: Simulation results with double precision floating operations. Note that the two lines, ‘proposed’ and ‘double’, are almost overlapped.

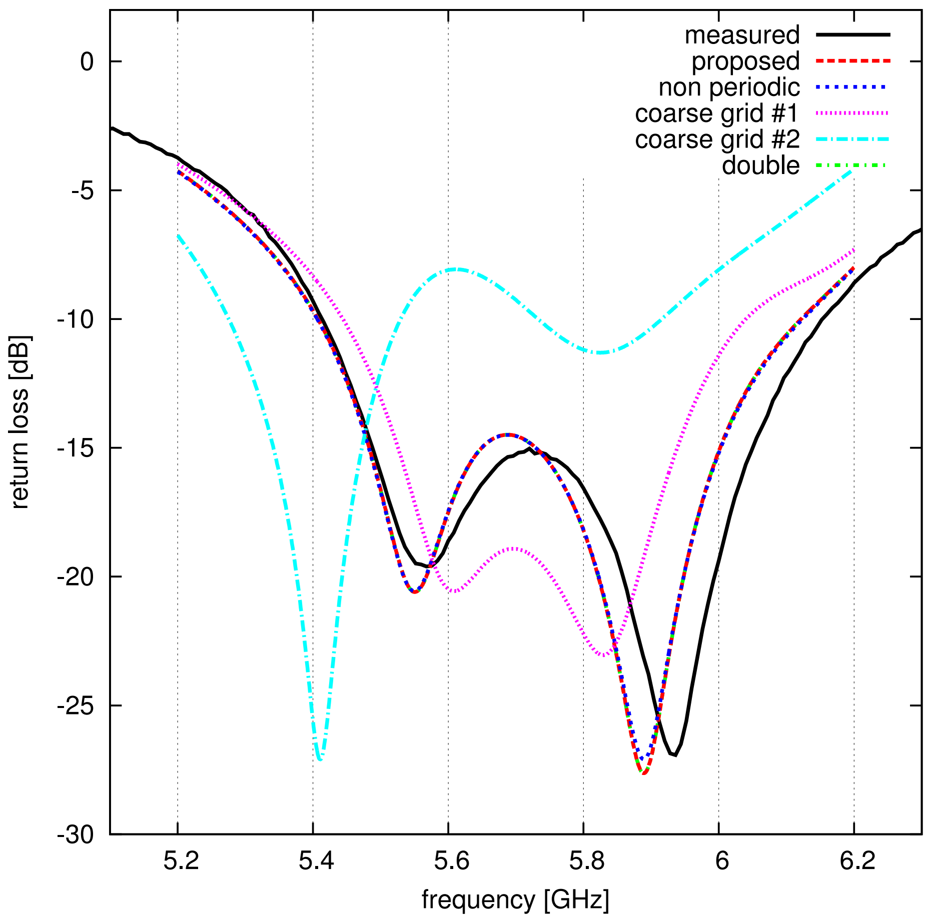

Figure 5.8: Comparison of the return loss between the calculated and measured results. Note that the three lines, ‘proposed’, ‘non periodic’ and ‘double’, are almost overlapped.

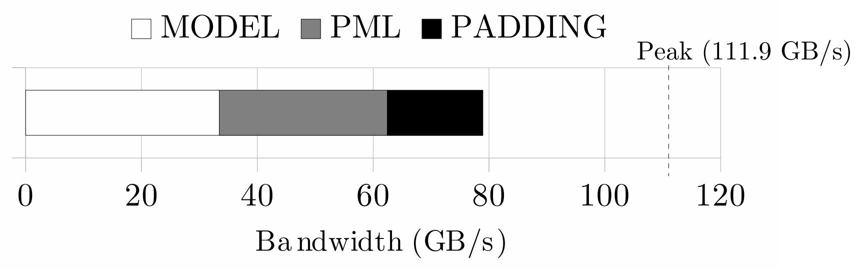

Figure 5.9: Required memory bandwidth for each part of the antenna model. Bandwidth of MODEL, PML and PADDING are 33.4 GB/s, 29.0 GB/s and 16.5 GB/s, respectively.

ters. The metrics include the simulation throughputs in million-point updates per second (Mpoints/s) and the simulation throughputs per peak bandwidth in million-point updates per giga bytes (Mpoints/GB) in order to alleviate differences in the size of simulation models and utilized GPU architectures. While ABC regions are excluded from the model size, the execution time used for the throughput calculation includes those of the ABC regions and Padding regions. The performance for the double precision version of the simulation code was also presented, since arithmetic precision is not explicitly mentioned in the literature for some comparison targets. The proposed implementation shows good results for both metrics among the compared implementation, suggesting effectiveness of the proposed implementation approach for GPU architectures.

The performance with that of CPU-based implementation was also compared. While parallel processing with multi-core, multi-thread and SIMD instructions ware not utilized in this implementation, the code was optimized considering the cache structure. Table 5.8 shows the performance of the GPU implementation achieved about 170 times throughput than the CPU implementation.

5.8 Summary

This chapter addressed the efficient implementation of absorbing boundary conditions of 3D-FDTD method for antenna designing on CUDA-compatible GPU. To reduce memory usage and to improve the simulation performance, a Non-Uniform grid method and the periodic boundary conditions were applied for Split PML implementation. The transformation technique of update-equations for partial ABC cells was also proposed. The empirical experiment showed that the proposed methods almost doubled the simulation performance and eventually achieved the memory bandwidth of 62.5 GB/s which corresponds to 55.8 % of the peak of the target GPU.

Table 5.8: Performance comparison with previous works. The peformance parameters are from announced values [1] [2] [3] [4] unless otherwise noted. °Performance per GPU core. °°Calculated by the values shown in [5]. ‡Exact GPU model number is not mentioned in [6]. Not reported in the literature.

	Proposal	Proposal (double)	CPU	[46]	[5]	[6]
GPU	GeForce GTX295	GeForce GTX295	Core i7 920	Tesla C1060	ATI HD4850	ATI X800‡
Peak Bandwidth (GB/s)	111.9°	111.9°	25.6	102	63.6°°	28.8↔32
Size	$192 * 192 * 64$	$192 * 192 * 64$	$182 * 186 * 69$	$204 * 145 * 499$	$180 * 180 * 180$	160,000 points
ABC	Split PML	Split PML	Split PML	Split PML	UPML	CPML
	10 layers	10 layers	10 layers	8 layers	-	-
Grid	Non-Uniform	Non-Uniform	Non-Uniform	Uniform	Uniform	Uniform
Model	MSA	MSA	MSA	Human Body	Vacuum	Vacuum
Precision	Single	Double	Single	-	Single	-
Throughput (Mpoints/s)	453	140	2.66	45	160	30
Throughput/Bandwidth (Mpoints/GB)	4.05	1.25	0.1	0.43	2.51	0.96↔1.04

Chapter 6

HOG feature extraction on an FPGA

6.1 Introduction

This chapter presents external memory-free FPGA implementation of a real-time image-based human detection system. The image-based human detection generally consists of two stages; calculation of feature amount of given images and pattern classification based on machine learning. In this implementation, histograms of oriented gradients (HOG) [51] and AdaBoost classifiers [52] are used as feature amount and pattern classifiers, respectively. The HOG feature roughly describes object shape of local regions of given images and this is widely used for various object recognition such as pedestrian and car detection [53–55]. High-performance and compact implementation is achieved by making deep pipelined arithmetic structure and a high bandwidth on-chip RAMs. The streamed processing approach described in Section 3.2.2 achieves real-time human detection for input video frames without any external memory.

So far, hardware implementation of HOG-based object detection has been actively investigated. Cao *et al*. [56] presented FPGA implementation of a stop sign detection system using HOG features. By using a simplified 4-bin HOG method, their architecture achieves a processing throughput of 60 frames per second (FPS) for $752 * 480$ images on a Virtex-4 SX35 FPGA. However, this simple detection method is not directly applicable for human detection. Kadota *et al*. [57] presented a novel simplification technique of the HOG feature extraction for efficient FPGA implementation. Their architecture can process $640 * 480$ image at 30 FPS with operating frequency 127.49 MHz on Stratix II FPGA, but detection part is not implemented. After a preliminary version of the HOG implementation was presented, Komorkiewicz *et al*. [58] presented fully-pipelined HOG and SVM implementation without using any external memory. To achieve superior accuracy, they used single-precision floating-point arithmetic for all stages of processing on a Virtex-6 XC6VLX240T FPGA. Their architecture needs multiple clock domains: 25 MHz clock for the HOG feature extraction and up to 237 MHz clock for SVM classifiers. Their architecture is able to process $640 * 480$ images at 60 FPS in real-time. While their system shares some architectural concepts with proposed system here in terms of streamed processing, an aspect of low-cost and low-energy implementation is more emphasized in the approach of this study. Mizuno *et al*. [59] presented HOG and SVM implementation for HDTV resolution video images, which is able to process 1920 ∗ 1080 images at 30 FPS with a Cyclone IV EP4CE115 FPGA operating at 76.2 MHz. Their architecture is based on an SoC style, in which an HOG feature extraction module is connected to a soft-core processor with on-chip buses on the FPGA. Contrasting to the approach, two types of external memories with SDRAM and SRAM are attached to the FPGA and aggressively used to process image data.

The contributions of this chapter are as follows: a) presenting a unified pipelined architecture of HOG feature extraction; b) using AdaBoost classifiers for real-time human detection; c) only single clock frequency is needed for HOG feature extraction and AdaBoost classifying; and d) proposed architecture is constructed without any external memory. Section 6.2 explains fundamentals of the HOG feature extraction. Section 6.3 shows the reduction techniques of the calculation amount for efficient implementation of human detection on an FPGA. Then, Section 6.4 details FPGA implementation with on-chip Block RAM and shift registers. Section 6.5 presents evaluation of the proposed architecture. Finally, Section 6.6 summarizes the chapter.

6.2 Algorithms

This section explains two algorithms for the implementation, the HOG algorithm for feature extraction from an input image and AdaBoost classifiers for real-time human detection. Along with the original HOG described in [51], some extended schemes are used for compact FPGA implementation [57, 60].

6.2.1 HOG features

The histograms of oriented gradients (HOG) use local histograms of oriented gradients of pixel luminance for feature extraction from a given image. In the implementation, the process of HOG feature extraction roughly consists of the following four stages:

1. Luminance gradients calculation
2. Histogram generation
3. Histogram normalization
4. Feature binarization

The first step is to calculate luminance values from a given image using lightness in the HDL color model as luminance for ease of luminance extraction from RGB full color images. In this scheme, the luminance L for each pixel is given by the following equation:

$$L = \frac{\max(R, G, B) + \min(R, G, B)}{2}, \tag{6.1}$$

where R, G and B mean values of each color channel of given image. All the values are presented as 8-bit unsigned integers, i.e., the value from 0 to 255.

Using the luminance, 1st-order central-differences in both x and y direction, g_x and g_y, are given by:

$$\begin{aligned} g_x(x, y) &= L(x+1, y) \quad L(x \quad 1, y) \\ g_y(x, y) &= L(x, y+1) \quad L(x, y \quad 1), \end{aligned} \tag{6.2}$$

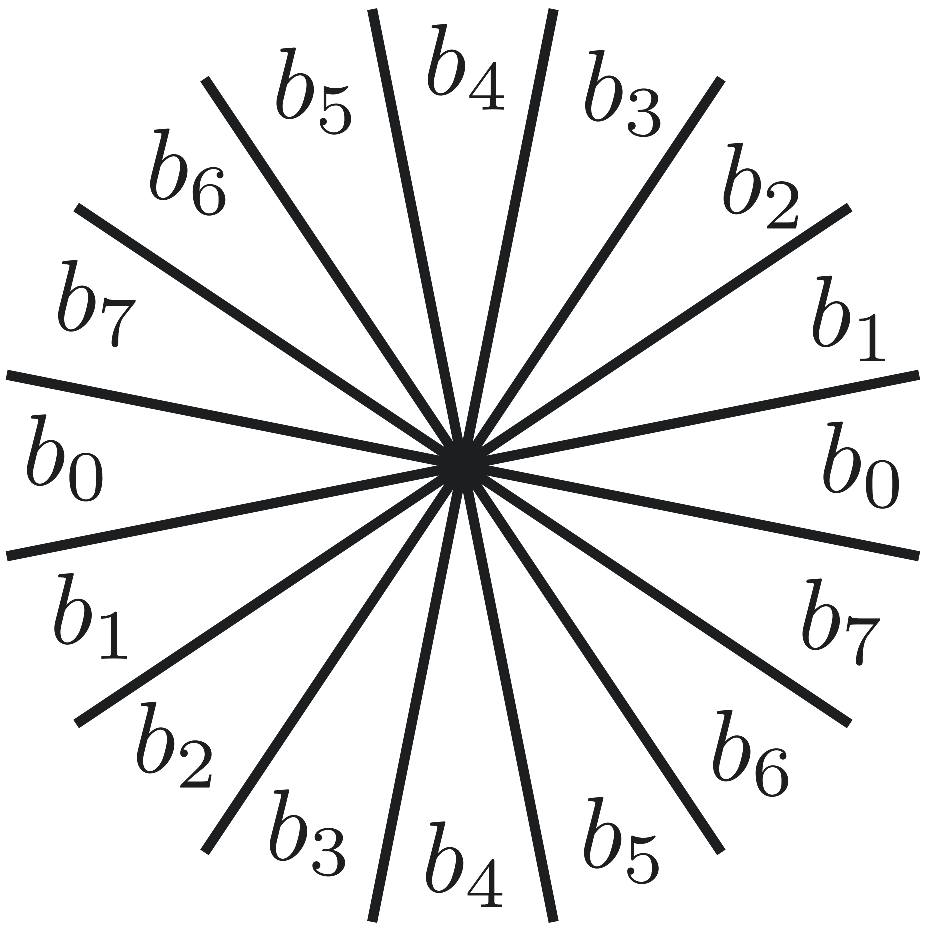

Figure 6.1: Orientation spacing of 8-bin.

where $L(x, y)$, $g_x(x, y)$ and $g_y(x, y)$ mean values of luminance and central-differences at the coordinate (x, y), respectively. Then a magnitude m as well as an orientation θ of the gradient are computed by:

$$
\begin{aligned}
m(x, y) &= \sqrt{g_x(x, y)^2 + g_y(x, y)^2} \\
\theta(x, y) &= \tan^{\;1} \frac{g_y(x, y)}{g_x(x, y)},
\end{aligned}
\tag{6.3}
$$

respectively.

After computing gradient magnitudes and orientations for each coordinate, the second step, histogram generation, is started. A histogram is generated for each *cell*, a square region of $p * p$ pixels, by accumulating the magnitude values according to each orientation of all pixels in a cell. Note that cells have no overlap with neighbors which means that a total of $\frac{w}{p} * \frac{h}{p}$ cells are defined for $w * h$ luminances of the given image. In this implementation, $p = 5$.

To make histograms, the gradient magnitudes are voted into 8 bins according to their orientations as shown in Fig. 6.1. When an orientation θ meets the following

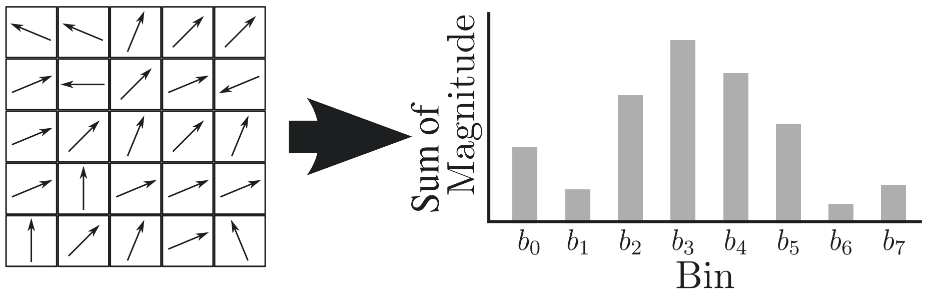

Figure 6.2: Voting magnitude for bins.

condition:

$$\frac{n}{8}\pi \quad \frac{\pi}{16} \geq \theta < \frac{n}{8}\pi + \frac{\pi}{16}, \tag{6.4}$$

the corresponding magnitude is voted to the bin b_n. Note that since the HOG does not focus on gradient directions but orientations, the opposite direction locates in the same bin. Since the gradients are voted into eight bins, eight-dimension feature vector is eventually generated for each cell as shown in Fig. 6.2. Then the feature vector for the cell is described as:

$$\boldsymbol{f} = (f_0, f_1, \dots, f_7) \tag{6.5}$$

where f_n means the sum of voted gradient magnitudes for bin b_n.

The third step, histogram normalization, is one of the most complex processes. A histogram $\boldsymbol{v}$ for the *block* at (i, j), the larger spatial region which consists of $3 * 3$ cells in this implementation, is defined as:

$$\begin{aligned} \boldsymbol{v} = (& \\ & \boldsymbol{f}(i, j), \boldsymbol{f}(i+1, j), \boldsymbol{f}(i+2, j), \\ & \boldsymbol{f}(i, j+1), \boldsymbol{f}(i+1, j+1), \boldsymbol{f}(i+2, j+1), \\ & \boldsymbol{f}(i, j+2), \boldsymbol{f}(i+1, j+2), \boldsymbol{f}(i+2, j+2) \\ &), \end{aligned} \tag{6.6}$$

where $\boldsymbol{f}(i, j)$ means the feature vector for the cell located at the i-th row and j-th column. Since $\boldsymbol{f}$ is eight-dimension vector, $\boldsymbol{v}$ has $8 * 9 = 72$ dimensions. Note that the block scans the entire image in a cell-by-cell manner (stride 1) and thus the number of blocks is equal to that of cells.

The values of histograms in a block are normalized using the L1-norm scheme described in [51]. The normalized histogram $\boldsymbol{v}_n$ is computed as:

$$\begin{aligned} \boldsymbol{v}_n &= \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|_1 + \varepsilon} \\ \|\boldsymbol{v}\|_1 &= \sum \|\boldsymbol{f}\|_1, \end{aligned} \tag{6.7}$$

where ε means regularization constants (here, $\varepsilon = 1$) to support empty histograms and it is known to have little impact on final results over a large range [51].

The final step is feature binarization. As a result of normalization, the normalized histogram has 72 real numbers. So the HOG features for $w * h$ luminance image occupies approximately $\frac{w}{5} * \frac{h}{5} * 72 * 8$ byte of memory capacity to store. This corresponds about 6.75 MB for VGA image, if the 8-byte double-precision floating point format is used for each value of histograms. This makes compact implementation with embedded hardware difficult. Therefore, a binarized HOG scheme described in [60] was employed to reduce the size of the features. In this scheme, each value of normalized histograms is binarized as:

$$v_b = \begin{cases} 1 \text{ if } v_n \sim t_b \\ 0 \; otherwise, \end{cases} \tag{6.8}$$

where v_b means a binarized result of a value v_n in the normalized histogram and t_b means a scalar threshold value. The binarized histogram $\boldsymbol{v}_b$ is generated by applying the binarization to all the values within the normalized histogram $\boldsymbol{v}_n$. With this reduction scheme, the memory capacity required to store the HOG features for luminance of $w * h$ is reduced to $\frac{w}{5} * \frac{h}{5} * 72$ bits. Compared with the original size of $\frac{w}{5} * \frac{h}{5} * 72 * 8$ bytes, $\frac{1}{64}$ of reduction is achieved.

In the boundaries of a given image, special conditions need to be set for generation of both cells and blocks. In this implementation, cells and blocks which cover the outside of the image are ignored. Therefore the numbers of cells and blocks are $\left(\frac{w}{5} \quad 2\right) * \left(\frac{h}{5} \quad 2\right)$.

6.2.2 AdaBoost classifiers

AdaBoost is a machine learning method that combines multiple weak classifiers, each of which only returns a true or false, so that an effective strong classifier is constructed [52]. In the training phase, positive sample images and negative sample images are repeatedly used by changing their weights, to select appropriate weak classifiers.

Since generation of classifiers using sample images is an offline process, AdaBoost classifier generator was implemented as software tools. In this implementation, HOG features that frequently appear in human sample images (positive samples) and rarely observed in other images (negative samples), were employed for weak classifiers. In addition, the block coordinates of such HOG features exist were also utilized. In AdaBoost method, one training phase generates one weak classifier as:

$$C_w = \}\boldsymbol{H}, \boldsymbol{P}|, \tag{6.9}$$

where $\boldsymbol{H}$ means the feature pattern and $\boldsymbol{P}$ means coordinate of the classifier. Given a binarized histogram $\boldsymbol{v}_b$ as a HOG feature, the weak classifier C_w returns true if any one of the nine binarized feature vectors for the cells within a block is exactly matched with the feature $\boldsymbol{H}$ of the classifier. The strong classifier C_s constructed through N_c times of the training can be expressed as:

$$C_s = \}C_{w1}, C_{w2}, \ldots, C_{wi}|. \tag{6.10}$$

Note that since duplicated weak classifiers are eliminated, the number of weak classifiers in one strong classifier is not always equal with the number of trainings. An example of a strong classifier constructed with three training phases is shown in Figure 6.3.

The strong classifier counts up the number of HOG features which weak classifiers return true in a $w_c * h_c$ detection window. After that the region that surrounded by the detection window is identified as a human image region if enough number of weak classifiers return true ($\sim t_c$).

The detection window moves the entire image from the upper left corner in a block-by-block raster scan manner. The required number of window scans for $w_b * h_b$ blocks using $w_c * h_c$ window is $(w_b \quad w_c + 1) * (h_b \quad h_c + 1)$. Therefore, the total number of matching processes required by the N_c weak classifiers is $(w_b \quad w_c+1) * (h_b \quad h_c+1) * N_c$.

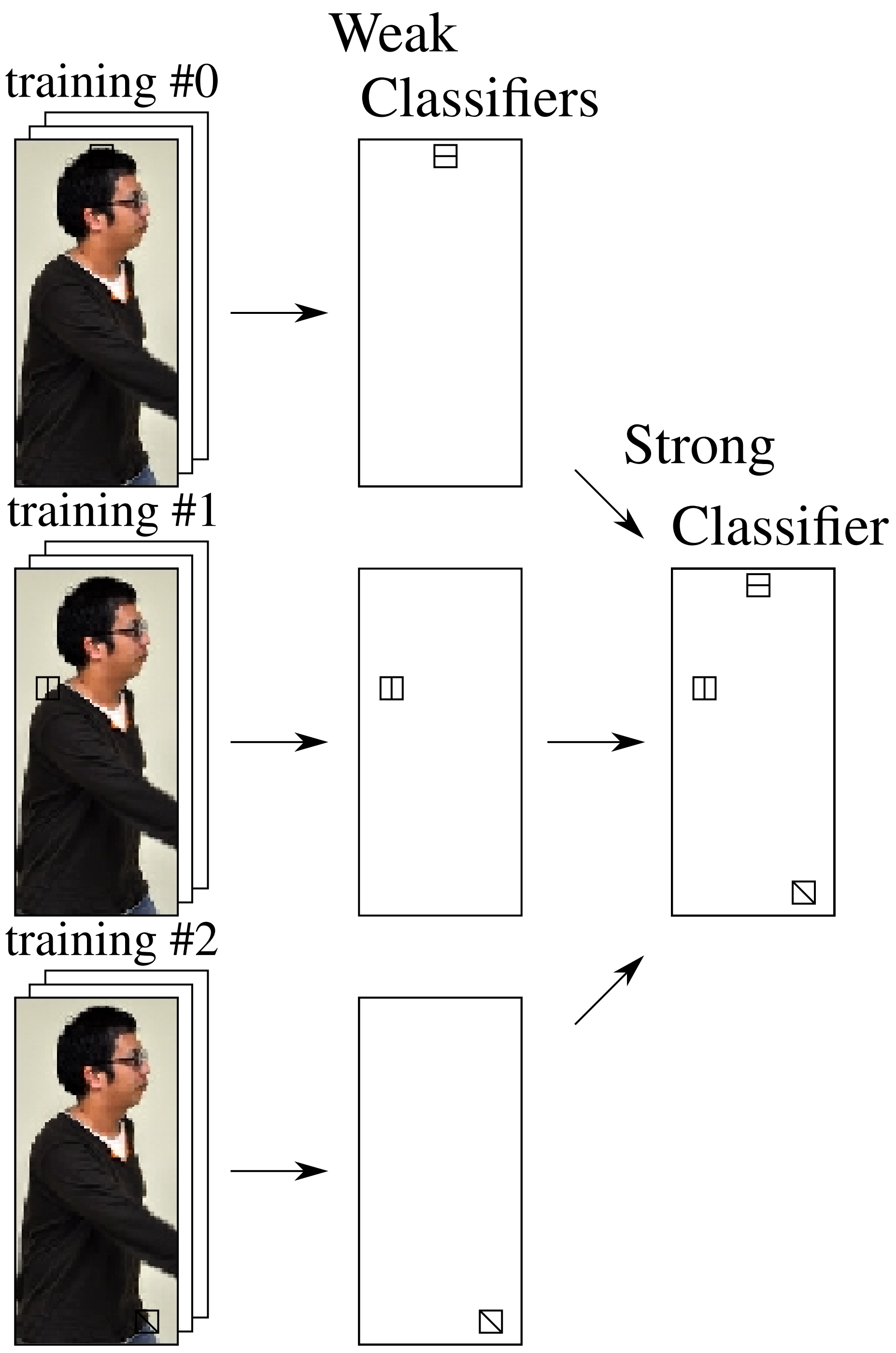

Figure 6.3: Strong classifier generation by three times training.

```
if $(|g_x| \times \tan\left(\frac{7}{16}\pi\right) \geq |g_y|)$ then
    $n \leftarrow 4$
else if $(|g_x| \times \tan\left(\frac{5}{16}\pi\right) \geq |g_y|)$ then
    $n \leftarrow 3$
else if $(|g_x| \times \tan\left(\frac{3}{16}\pi\right) \geq |g_y|)$ then
    $n \leftarrow 2$
else if $(|g_x| \times \tan\left(\frac{1}{16}\pi\right) \geq |g_y|)$ then
    $n \leftarrow 1$
else
    $n \leftarrow 0$
end if
if $(sign(g_x) \neq sign(g_y))$ then
    $n \leftarrow (8 \quad n) \pmod 8$
end if
```

Figure 6.4: Simplified bin selection.

6.3 Simplification of the HOG feature process for hardware

Some of the processes of the HOG feature extraction described in the previous section consist of mathematical functions such as a trigonometric function and division. These functions make the system design difficult to fit in small FPGAs. Thus, approximation schemes were introduced to reduce the calculation complexity.

6.3.1 Approximation for gradient orientations

The first approximation is for choosing the best bin b_n according to a given gradient orientation θ as described in [56]. A naive scheme requires computation of the arc tangent function as shown in Eq. (6.3). Since it is only needed to choose the best bin b_n to vote from eight bins, a more compact scheme to compute Eq. (6.3) and Eq. (6.4) can be introduced. The best bin b_n for given orientation θ can be defined as:

$$g_x \times \tan\left(\frac{n}{8}\pi \quad \frac{\pi}{16}\right) \geq g_y < g_x \times \tan\left(\frac{n}{8}\pi + \frac{\pi}{16}\right). \tag{6.11}$$

Equation (6.11) allows us to choose the bin using simpler functions. Figure 6.4 shows pseudo code for simplified computation of the bin selecting. This scheme only requires four multiplications with constants, four comparisons, comparison of signs, subtraction and modulo operation.

6.3.2 Approximation for normalization of histogram

The second approximation is for normalization process in Eq. (6.7). Kadota *et al.* introduced an approximation scheme for L2-norm normalization in [57]. In this scheme, divisors for the normalization are approximated to power-of-two values, so that the division is replaced by a shift operation. Enhanced-version of this approximation is applied in Eq. (6.7).

```
1:  $\alpha \leftarrow \lceil \log_2(\sqrt{\|v\|_1 + \varepsilon}) \rceil$
2:  if $((1+\frac{3}{4})2^{\alpha-1} < \sqrt{\|v\|_1 + \varepsilon})$ then
3:      $v_n \leftarrow \frac{f}{2^\alpha}$
4:  else if $((1+\frac{2}{4})2^{\alpha-1} < \sqrt{\|v\|_1 + \varepsilon})$ then
5:      $v_n \leftarrow \frac{f}{2^\alpha} + \frac{f}{2^{\alpha+2}}$
6:  else if $((1+\frac{1}{4})2^{\alpha-1} < \sqrt{\|v\|_1 + \varepsilon})$ then
7:      $v_n \leftarrow \frac{f}{2^\alpha} + \frac{f}{2^{\alpha+1}}$
8:  else
9:      $v_n \leftarrow \frac{f}{2^\alpha} + \frac{f}{2^{\alpha+1}} + \frac{f}{2^{\alpha+2}}$
10: end if
```

Figure 6.5: Simplified histogram normalization.

If the denominator of Eq. (6.7) is approximated to 2^α such that $2^{\alpha-1} < (\sqrt{\|v\|_1 + \varepsilon}) \geq 2^\alpha$, the division for the normalization can be replaced by a shift operation. However, this naive approximation to the nearest power-of-two value increases the normalization error. To mitigate the error, sums of the number of the form $1/2^k$, like $1/2^k + 1/2^l$, is used.

The original interval $(2^{\alpha-1}, 2^\alpha]$ is divided into n sub-intervals;

$$\left(2^{\alpha-1}, (1+\frac{1}{n})2^{\alpha-1}\right], \left((1+\frac{1}{n})2^{\alpha-1}, (1+\frac{2}{n})2^{\alpha-1}\right], ..., \left((1+\frac{n-1}{n})2^{\alpha-1}, 2^\alpha\right], \tag{6.12}$$

and shift amounts for each interval are precomputed. Figure 6.5 shows pseudo code for the approximation. In the approximation, the minimum α which meets the condition $(\sqrt{\|v\|_1 + \varepsilon}) \geq 2^\alpha$ is first computed. Then an appropriate interval is chosen from n sub-intervals, and finally each value of normalized histogram is computed by shift and addition. The original interval is divided into four sub-intervals ($n = 4$) and three kinds of power of two values are used in this implementation.

Figure 6.6 shows comparison results of approximation errors between proposed scheme and the naive power-of-two scheme, in the case that a numerator is 361 in Eq. (6.7). The results show that the normalization errors are effectively reduced with the relatively simple additional computation processes.

6.4 Implementation

Fig. 6.7 shows an overview of the proposed human detection system and Fig. 6.8 shows the pipeline structure for HOG feature extraction. The camera module outputs bayer-patterned image data sequentially in a pixel-by-pixel manner, and these data are directly passed to the HOG feature extraction pipeline. Then, extracted HOG features are stored in on-chip BRAMs and are used by the human detection module consisting of AdaBoost classifiers. Decision criteria for each weak classifier (AdaBoost data) are provided by on-chip ROM also implemented with BRAMs. Finally, detection results are indicated with markers on output images and outputted to an external display.

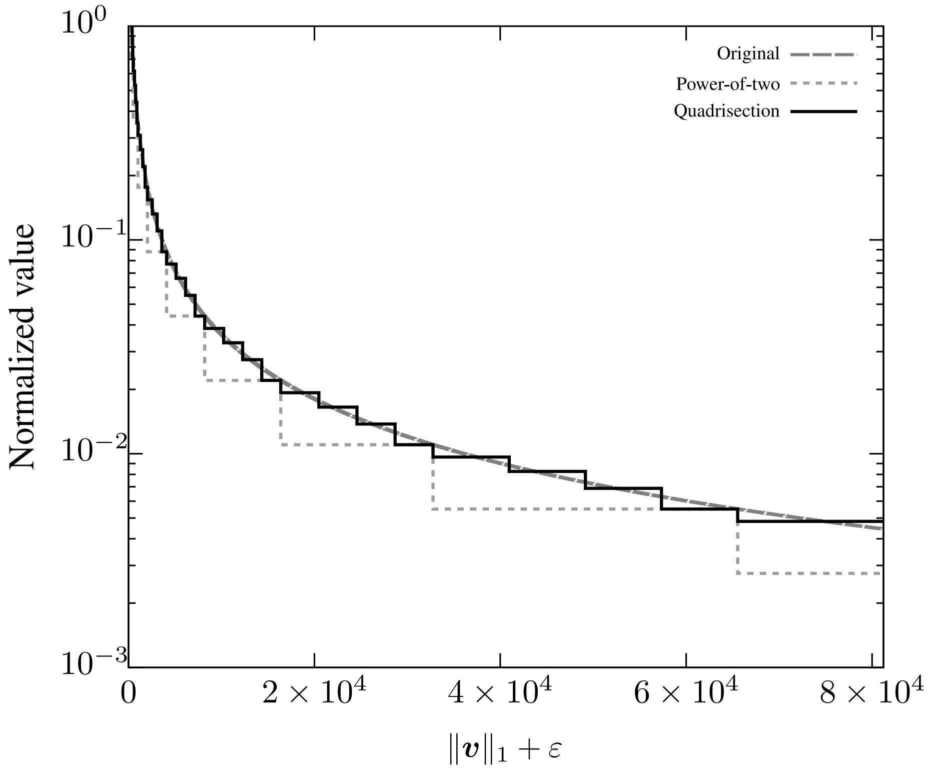

Figure 6.6: Normalization errors.

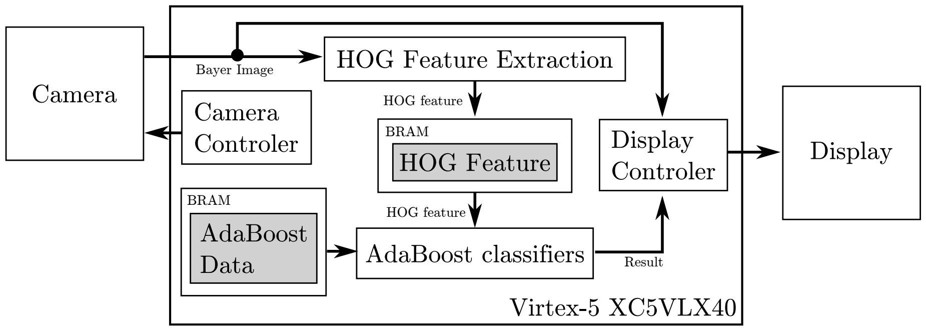

Figure 6.7: Overview of proposed architecture for human detection. Gray boxes show what data are stored on BRAM.

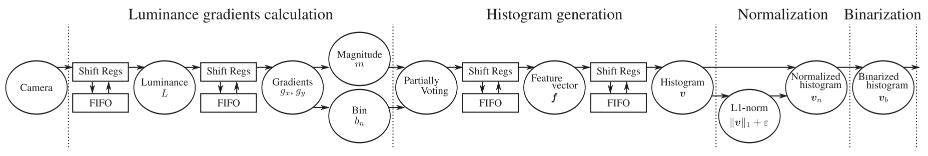
Luminance gradients calculation
Histogram generation
Normalization
Binarization
Camera
Shift Regs
FIFO
Luminance L
Shift Regs
FIFO
Gradients g_x, g_y
Magnitude m
Bin b_n
Partially Voting
Shift Regs
FIFO
Feature vector $\boldsymbol{f}$
Shift Regs
FIFO
Histogram $\boldsymbol{v}$
L1-norm $\|\boldsymbol{v}\|_1 + \varepsilon$
Normalized histogram $\boldsymbol{v}_n$
Binarized histogram $\boldsymbol{v}_b$

Figure 6.8: Overview of HOG feature extraction pipeline

6.4.1 Luminance

As an input device, an OmniVision Technologies OV9620 CMOS camera was used. Since this device produces a raw 8-bit Bayer pattern image consisting of $640 * 480$ valid pixels as shown in Fig. 6.9, it is implemented $2 * 2$-pixel filter to convert a bayer-patterned image to a full color image. Then, luminance values are calculated from the full color image using Eq. (6.1). This filter can be implemented by the streamed architecture shown in Fig. 3.1 with $2 * 2$ of 8-bit registers and a FIFO to store 638 of 8-bit pixels. As a result, a gray scale image of $320 * 240$ of 8-bit luminance values is produced from an input image.

6.4.2 Luminance gradient

As shown in Fig. 6.10, calculation of central-differences of luminance g_x and g_y in Eq. (6.2) requires the streamed architecture with $3 * 3$ of 8bit-registers and a FIFO to store two lines. In a pipeline part, two subtractors for 8-bit integers is needed for computing both g_x and g_y. Computation of the gradient magnitude m in Eq. (6.3) requires two multipliers for 8-bit unsigned integers ($|g_x|$ and $|g_y|$), an adder for 17-bit unsigned integers and a square root operator for 17-bit unsigned integers. Since the maximum value of the gradient magnitude is $361 (= \lceil \sqrt{255^2 + 255^2} \rceil)$, m is expressed as a 9-bit unsigned integer. The bin b_n to be vote can be chosen with the algorithm shown in Fig. 6.4. In order to simplify the implementation, $|g_x|$, $|g_y|$ and results of tangent functions are expressed by fixed-point arithmetic with a 10-bit fraction part. As a result, the computation for the bin b_n requires four multipliers for $|g_x|$ and 10-bit constant unsigned integers, four comparators for 18-bit unsigned integers and other small operators. The bin b_n is expressed as a 3-bit unsigned integer.

Multipliers and a square root operator were generated by Xilinx CORE Generator. Note that all the module is fully-pipelined to compute luminance gradients in the same rate with the camera interface.

6.4.3 Histogram generation for cells

The histogram generation was also implemented using the stream processing approach. Since each cell is not overlapped with others, histograms do not have to be computed every clock cycle. Thus it does not need to handle 25 gradients at the same time.

As shown in Fig. 6.11, the first partial histogram of gradient histograms for five consecutive luminance gradients in horizontal direction is computed using temporary register in partially voting process. Then the stream of the partial histograms goes through FIFO so that partial histograms for five lines are eventually summed up to make the full histogram for the cell in fully voting process.

Since the gradient magnitude m is expressed with a 9-bit value, each orientation of a partial histogram can be expressed as a 11-bit unsigned integer. Thus, required

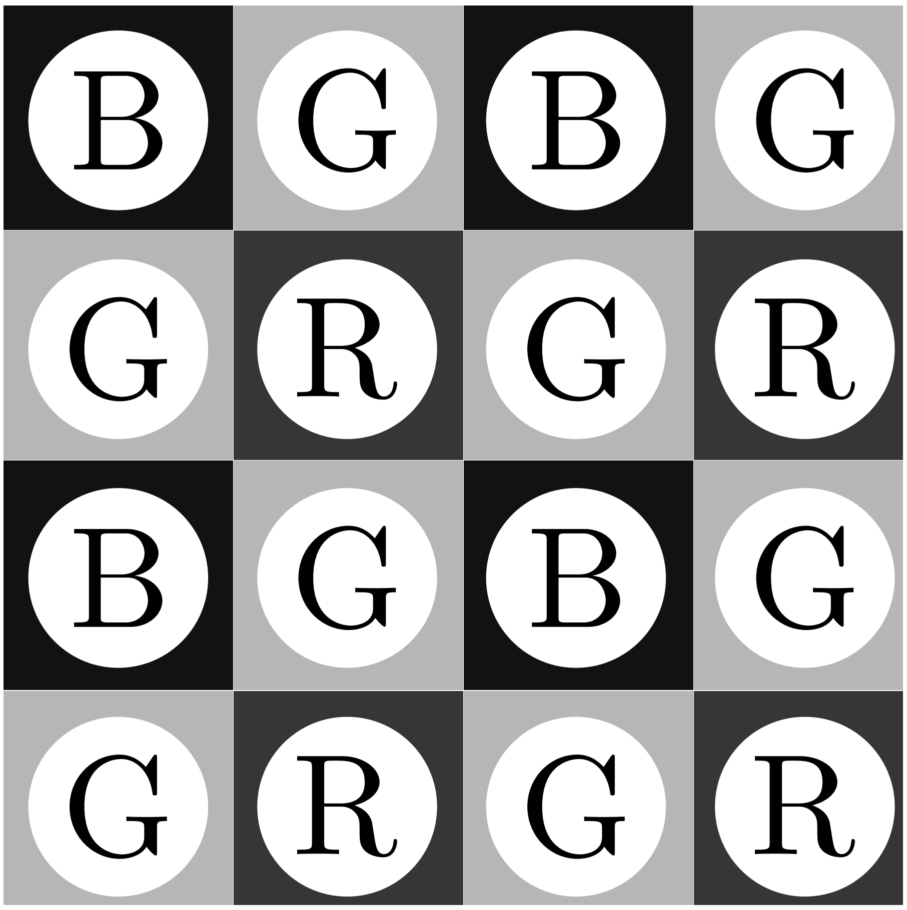

Figure 6.9: Bayer pattern of camera images. Each alphabet means valid color channel at each location.

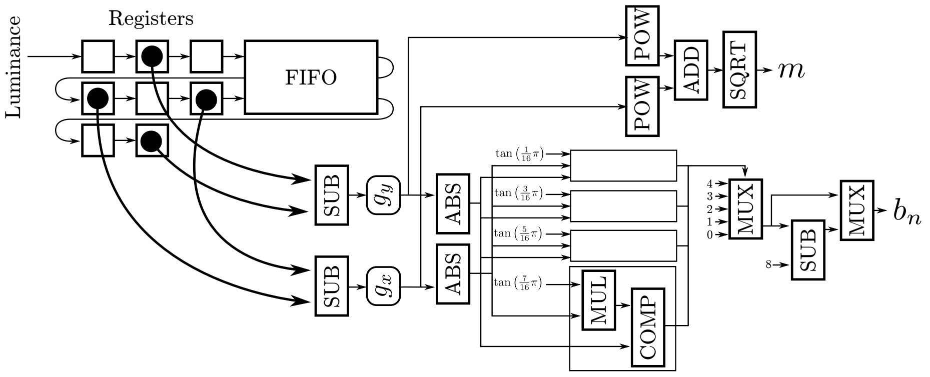
Luminance
Registers
FIFO
SUB
SUB
g_y
g_x
ABS
ABS
POW
POW
ADD
SQRT
m
$\tan\left(\frac{1}{16}\pi\right)$
$\tan\left(\frac{3}{16}\pi\right)$
$\tan\left(\frac{5}{16}\pi\right)$
$\tan\left(\frac{7}{16}\pi\right)$
MUL
COMP
4
3
2
1
0
MUX
8
SUB
MUX
b_n

Figure 6.10: Streamed architecture for computation of the gradient magnitude and the bin.

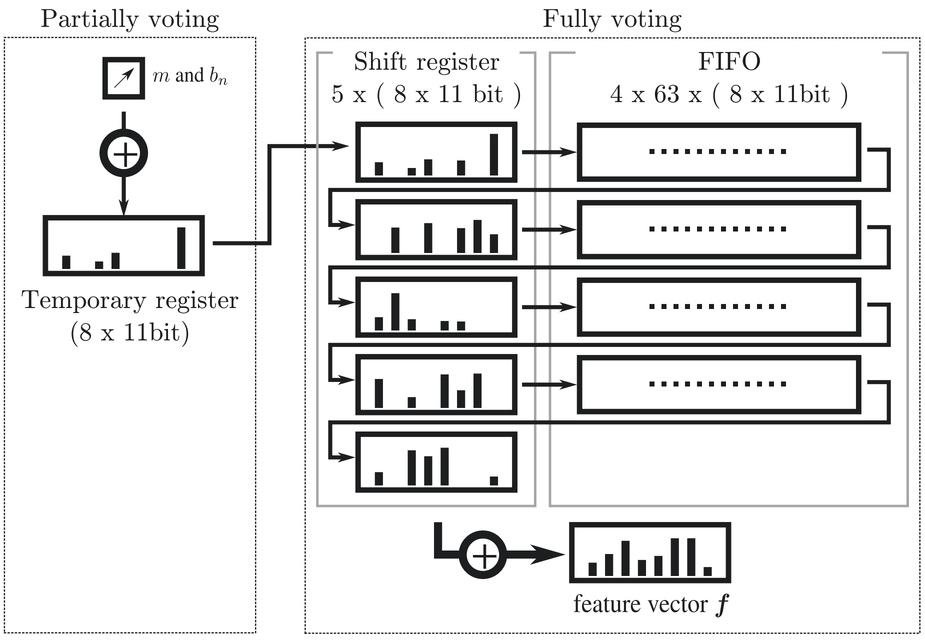

Figure 6.11: Histogram generation for cells.

resources for streaming are registers corresponds to 11 bits $*$ 8 orientations $*$ 5 lines and FIFOs corresponds to 11 bits $*$ 8 orientations $*$ 63 cells per line $*$ 4 lines. The voter requires eight comparators for 3-bit unsigned integers and eight accumulators for 11-bit unsigned integers. Finally, full histogram consists of eight 14-bit unsigned integers.

6.4.4 Histogram normalization in a block

The normalization process is carried out for a moving $3*3$ windows of cell histograms. Again, it can be implemented the streamed structure. For this process, three lines of 3-stage shift registers and 2 lines of 61-stage FIFOs are used to store cell histograms. The histogram $\boldsymbol{v}$ is generated by concatenating all the cell histograms in a window.

Every time a new cell histogram is streamed in, all the 72 values of nine histograms in the $3*$ 3-cell window are summed up to obtain a value of $\|\boldsymbol{v}\|_1$. This addition is done in two clock cycles to avoid degradation of the clock frequency. Since the maximum value of $\|\boldsymbol{v}\|_1$ is $81,225$, $\|\boldsymbol{v}\|_1 + \varepsilon$ can be expressed as a 17-bit unsigned integer when $\varepsilon \geq 49,847$, and the value of ε is '1' in this implementation.

In the next clock cycle, shift amounts for normalization are computed using the approximation scheme shown in Section 6.3.2. Figure 6.12 shows more hardware-oriented pseudo code for the approximation scheme. Lines from 1 to 4 of Fig. 6.12 can be implemented as a Look-Up-Table. Since a value of $\|\boldsymbol{v}\|_1 + \varepsilon$ is expressed with a 17-bit unsigned integer, α is expressed with a 5-bit unsigned integer. In lines 5-16, the shift amounts are computed with small shift operation and comparisons. Lines 6-7 are needed for intervals which are too narrow to divide.

In the 4th clock cycle, values of the normalized histogram are computed by three shift operations and two additions in line 17. Therefore, the normalization in a block is

```
$\alpha$ ′   0
while $2^{\alpha} < ( v\ _{1} + \varepsilon)$ do
    $\alpha$ ′   $\alpha + 1$ √ √
end while
$i$ ′   $( v\ _{1} + \varepsilon) >> (\alpha \quad 3)$
if $(\alpha \geq 2)$ then  √ √
    $S_0$ ′  $\alpha$, $S_1$ ′  $\in$ , $S_2$ ′  $\in$
else if $(i = 8)$ then
    $S_0$ ′  $\alpha$, $S_1$ ′  $\in$ , $S_2$ ′  $\in$
else if $(i = 7)$ then
    $S_0$ ′  $\alpha$, $S_1$ ′  $\in$ , $S_2$ ′  $\alpha + 2$
else if $(i = 6)$ then
    $S_0$ ′  $\alpha$, $S_1$ ′  $\alpha + 1$, $S_2$ ′  $\in$
else
    $S_0$ ′  $\alpha$, $S_1$ ′  $\alpha + 1$, $S_2$ ′  $\alpha + 2$
end if
$\boldsymbol{v}_n$ ′  $\sum_{i=0}^{2}(\boldsymbol{f} >> S_i)$
```

Figure 6.12: Hardware oriented histogram normalization.

accomplished in four clock cycles.

Since the maximum shift amount is 19, each value of histograms is temporarily expanded to a 33-bit fixed-point number with a 19-bit fraction part. As a result of the normalization, each value of histograms is expressed as a 19-bit fixed-point number with a 19-bit fraction part. Which means the normalized histogram for a block is expressed as 72 of 19-bit fixed-point numbers.

6.4.5 Histogram binarization

As described in Section 6.2, the binarization process is relatively simple. The process requires 72 comparators for 19-bit fixed-pointer numbers, and thus it is implemented in a combinational circuit. As a result of the binarization, 72-bit HOG feature for a block is extracted.

6.4.6 Data stream of HOG feature extraction

As summarized in Fig. 6.8, the whole process flow of the HOG feature extraction is fully pipelined. All the HOG features obtained in this process flow are serially stored in on-chip RAMs. The on-chip RAMs can hold all the normalized HOG histograms of $62 * 46$ blocks, which are obtained from a single frame image. Since each normalized HOG histograms is expressed as a 72-bit value, whole HOG feature for a single frame image occupies about 25 kBytes of the on-ship RAMs.

6.4.7 Human detection using AdaBoost classifiers

Fig. 6.13 shows an overview of the human detection module using AdaBoost classifiers. The strong AdaBoost classifier C_s is stored in ROM which actually implemented

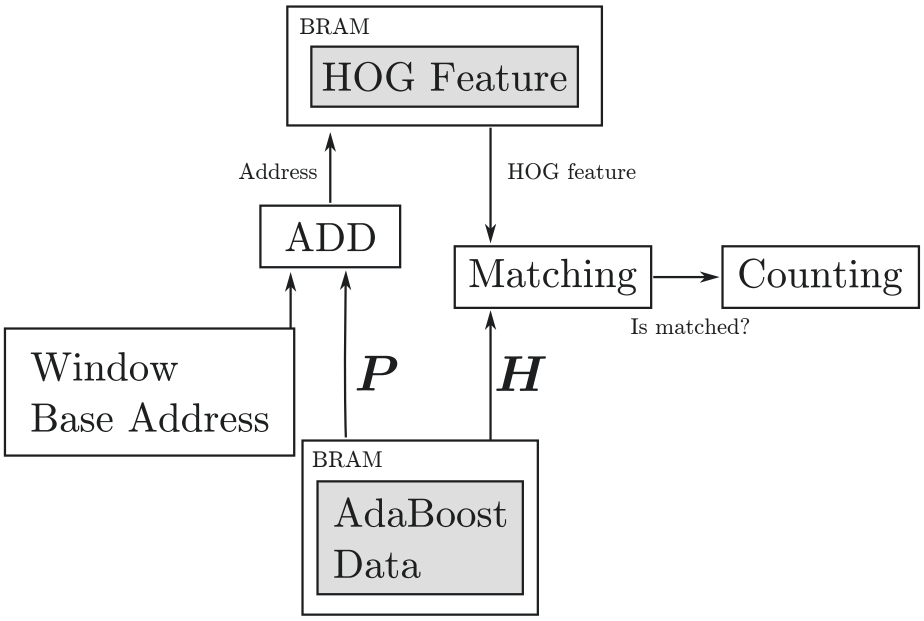

Figure 6.13: Overview of human detection module using AdaBoost classifiers.

with BRAM. Since each weak classifier is expressed as two 8-bit values for a feature pattern $\boldsymbol{H}$ and a block coordinates $\boldsymbol{P}$, N_c weak classifiers occupies approximately $2N_c$ byte of memory capacity to store. What the circuit for the strong classifier needs to do is simply to compare the weak classifiers with HOG features extracted from input images and to count the number of active classifiers.

Since each weak classifier corresponds to a difference block coordinate, random access to on-chip RAMs which stores HOG feature is needed in contrast to the streamed approach exploited in the HOG feature extraction.

In this implementation, the size of the detection window ($w_c * h_c$) is set to $8 * 19$ according to the size of training samples. While a strong classifier is constructed by executing 500 times training trials, a total of 84 weak classifiers were eventually generated with obtain a lot of duplications. Thus 168 bytes of memory capacity were required to store the strong classifier.

The total number of matching processes required for the 84 weak classifiers is $(62 \quad 8 + 1) * (46 \quad 19 + 1) * 84 = 129,360$. The camera device generates one frame image data in 400,000 clock cycles including synchronization intervals. The proposed implementation requires 385,452 clock cycles to extract whole HOG features for one frame data, while the AdaBoost detection process takes 129,360. To finish whole the detection process within 400,000 clock cycles, the feature extraction and the human detection processes are overlapped. As a result, the proposed implementation processes whole computation for one frame in 387,820 clock cycles to enable in-frame real-time processing. Due to the remainder can process additional 7 weak classifiers, $\rfloor 12,180/(55 * 28) \lceil = 7$, this result suggests the strong classifier can be constructed of up to 91 weak classifiers.

Table 6.1: Resource utilization

Resource	Used	Available	Percentage (%)
SLICE	6,607	7,200	91.8
FF	2,255	28,800	7.8
LUT	17,121	28,800	59.4
BRAM/FIFO	36	48	75.0
DSP48E	2	48	4.2

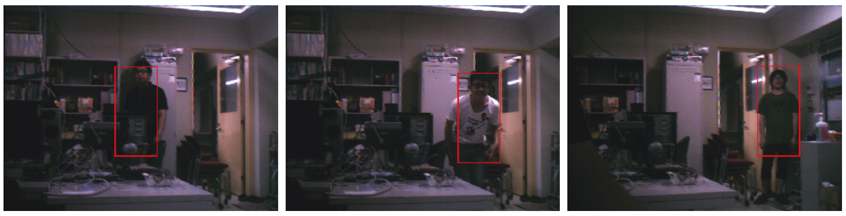
Figure 6.14: Example results of human detection process.

6.5 Implementation results and evaluation

The human detection process described in Section 6.4 was implemented on a Xilinx ML501 board equipped with a Virtex-5 XC5VLX50 FPGA. The design was described in Verilog HDL and a bitstream file was generated using Xilinx ISE design tools 13.4.

Table 6.1 shows implementation results of the design. While the maximum operating frequency of the design achieved 45 MHz, the camera device in the system restricts the system clock to 25 MHz. In spite of the restricted relatively-low clock frequency, the FPGA implementation achieved the throughput of 62.5 FPS for VGA frames and execution latency was also fitted in a single frame time, that is, the real-time performance was accomplished. Furthermore, if a high-speed camera device were used and the FPGA design was operated with the maximum clock frequency of 45 MHz, the execution throughput would be improved up to 112.5 FPS. An adder tree for computing L1-norm in Eq. (6.7) lies on a critical path. Figure 6.14 illustrates examples of experimentation results. These images were obtained by a monitoring mechanism on the FPGA board, which allows us to transmit actual result image data to the host PC. The red frame markers were also generated by the FPGA circuit to display detected regions.

The evaluation includes the quality of results of the approximation scheme by comparing to the original floating-point arithmetic algorithm. In this evaluation, two software simulators in C for both of the schemes were implemented and NICTA Pedestrian database [61] was used for benchmarking. To generate AdaBoost classifiers, 2,000 images from the database were used for offline machine learning, while other 1,000 images were used as the evaluation data. The size of each image is $64 * 80$ pixels. As a result of 500 times of training, a strong AdaBoost classifier which consists of 84 weak classifiers were eventually generated. The threshold value for the histogram binarization was set to 0.04.

Figure 6.15 summarizes the results of comparison as a chart of receiver operator

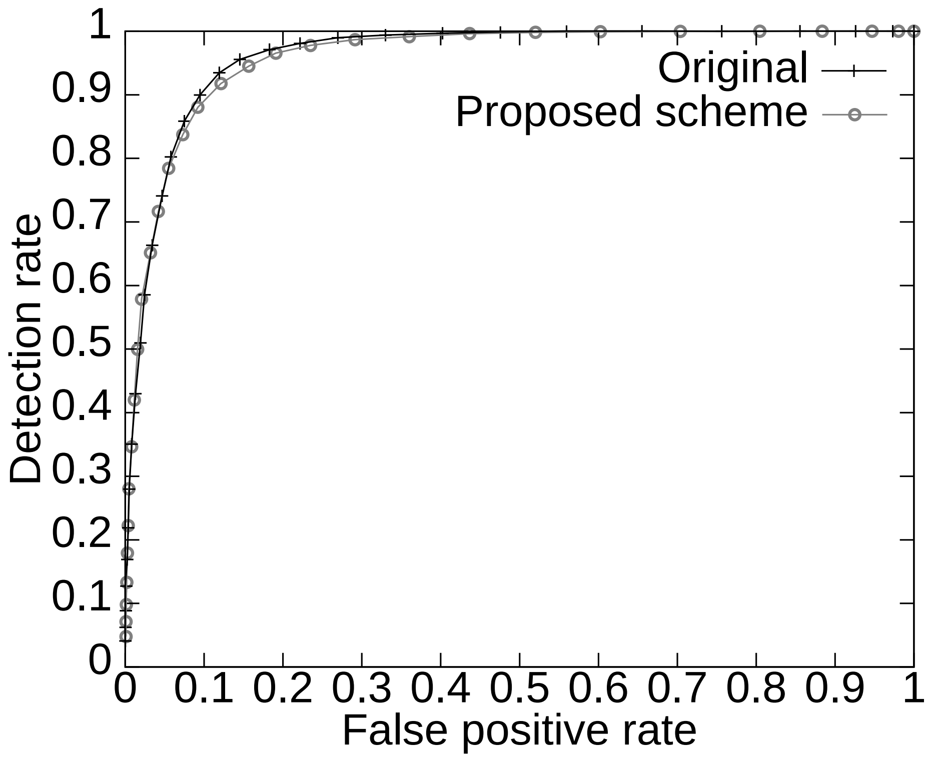

Figure 6.15: ROC curves for the NICTA Pedestrian database.

characteristics (ROC) curve. The chart shows the relationship between the false positive rate (x-axis) and the detection rate (y-axis) of the system. The closer to the upper left area of the chart means better quality of results. The plots were made by changing the threshold number of weak classifiers for detection from 0 to 30. As a result, the simplified scheme for hardware implementation shows 94.5 % of the detection rate with 15.7 % of the false positive rate, while the original one shows 95.6 % of the detection rate with 14.5 % of the false positive rate. Although the detection results for original scheme might be improved by tuning parameters such as the threshold value, the evaluation results suggest the negative impact of simplified scheme for hardware implementation is limited in terms of detection rate.

Finally, the throughput of the FPGA implementation was compared to that of software implementation. The software implementation was compiled by gcc 4.3.1 and run on 2.67 GHz Intel Core i7 920 with 6 GB DDRIII operated by openSUSE 11.2. As a result, the software implementation achieve about 15 FPS which means the FPGA design is 4.2 times faster than the software implementation. Although another software implementation using SIMD instructions is reported to achieve 20 FPS [58], the proposed FPGA design in this study is still 3.13 times faster. Moreover, 7.5 times faster throughput to the software implementation is expected if the camera device operates at the maximum frequency of the design.

Although the proposed external memory free architecture was shown to be efficient for the HOG-based human detection algorithm, this architecture will not be versatile for every image processing application. For example, use of an external frame buffer enables random access to image data and makes it easy to use a soft-core processor to execute a part of tasks. Frame buffers are also useful for introducing multiple clock domains in designs and absorbing differences in throughputs between the domains. On the other hand, applications that have a relatively simple control flow and regular data access patterns, especially a class of algorithms that use moving widow operators are good candidates for the proposed architecture. Avoiding the use of frame buffers, the architecture can reduce energy consumption for the external memories and off-chip data communications as well as implementation size, which is advantageous especially for embedded systems.

6.6 Summary

In this chapter, compact FPGA implementation of real-time human detection using the HOG feature and AdaBoost classifier has been presented. As a result of evaluation, the throughput of 62.5 FPS was achieved without using any external memory modules. If a high-speed camera device was available, the maximum throughput of 112 FPS was expected to be accomplished. While some simplifications were introduced to reduce hardware complexity, the evaluation with ROC curves showed that the negative impact of the simplifications is limited.

Chapter 7

Heat Conduction Simulation using a MaxCompiler

7.1 Introduction

With increase in integration density of semiconductors, Field Programmable Gate Arrays (FPGAs) are getting new applications year by year. On the other hand, user designs on FPGA are still described mainly in hardware description languages (HDLs) like VHDL and Verilog HDL at register transfer level (RTL). Some IP core generators like Xilinx CoreGenerator and Altera Megafunction support users to generate floating-point arithmetic operators, PCI Express (PCIe) interface, DDRIII memory interface and etc. However, adequate parameter settings of IP cores and data scheduling are still up to users.

High-level synthesis tools improve productivity of hardware designing by generating HDL code from high-level specification described in a language such as C and Java. Various tools including commercial tools are available [11, 12], and MaxCompiler developed by Maxeler Technologies is one of them [14, 62]. Using the high-level synthesis, users can easily describe desired applications without much regard for detailed hardware issues. On the other hand, synthesis, placement and routing processes tend to dominate large part of development time. In addition, these time-consuming processes are required to be carried out over and over in order to find the best architectural parameters.

This study has aggressively investigated a stream-oriented processing framework, in which input data are fed into a series of arithmetic operators and processed in a pipelined manner, especially for real-time image processing applications [21–23, 63]. The stencil computation, which is widely used for various kinds of scientific applications, is also known to be suitable for the stream-oriented processing on FPGA-based systems [64–67]. This chapter presents implementation of 3-D heat conduction simulation as the stream-oriented process using the MaxCompiler and evaluates how resource usage and performance are optimized by the high-level synthesis.

The contributions of this chapter include:

≤ Performance modeling of 3-D stencil computing a stream-based FPGA accelerator, formulating how two important user space design parameters have effect on the execution performance and resource amounts.

≤ Evaluation of a set of implementation experiments of a 3-D heat conduction simulation, demonstrating the proposed model gives reasonable estimation of performance and resource usage so that the best combination of the design parameters are easily determined.

≤ Empirical measurements of power consumption of the FPGA accelerator, suggesting proposed design space exploration framework is also valid in terms of optimizing energy efficiency of application execution.

The remainder of this chapter is organized as follows. Section 7.2 shows MaxGenFD, a domain specific framework for 3-D stencil computation on MaxCompiler, and user-space parameters of MaxGenFD. Then, Section 7.3 shows implementation of heat conduction simulation as a benchmark application. Section 7.4 presents evaluation of the design with various configurations. Finally, Section 7.5 summarizes the chapter.

7.2 MaxGenFD

MaxGenFD is an application framework for 3-D stencil computations build on MaxCompiler [14]. The MaxGenFD provides various libraries and features for users to easily develop desired applications taking an advantage of stream-oriented processing. This section describes an architecture overview for 3-D stencil computation which is generated by MaxGenFD and two important parameters that affect the performance: *multi-pipelines* and *multi-steps*.

7.2.1 Overview

In DFE design with MaxGenFD, users can specify size of a computational space dynamically at runtime. First, the user-specified computational region is split into subregions called blocks. Block-splitting is performed in X and Y dimensions, but not in Z dimension. Given computational size (X, Y, Z) and block size (B_W, B_H), computation is performed per (B_W, B_H, Z) block.

Accesses to the off-chip DDRIII memory are performed per tile, (T_W, T_H) 2-D array. Tiles fetched from the memory are stored in on-chip memory called Block RAM (BRAM) and necessary data are sent to the pipelines. Parameters of the block (B_W, B_H) and the tile (T_W, T_H) are specified by users at compile time.

7.2.2 Multi-pipelines and multi-steps

3-D stencil computation can be accelerated by parallelizing arithmetic pipelines and by applying the update equations more than once per off-chip memory access; these methods are called multi-pipelines and multi-steps. Fig. 7.1 shows configuration example of kernels with the multi-pipelines and the multi-steps.

The multi-pipelines shown in Fig. 7.1(b) are a method to parallelize arithmetic pipelines in space and compute multiple grids at the same time. The method improves computational performance if enough memory bandwidths are available. Since the pipelines can share data caches, increase in the number of pipelines has a small impact on the BRAM usage for caches. Note that MaxGenFD requires the number of pipelines to be dividable by the tile width T_W.

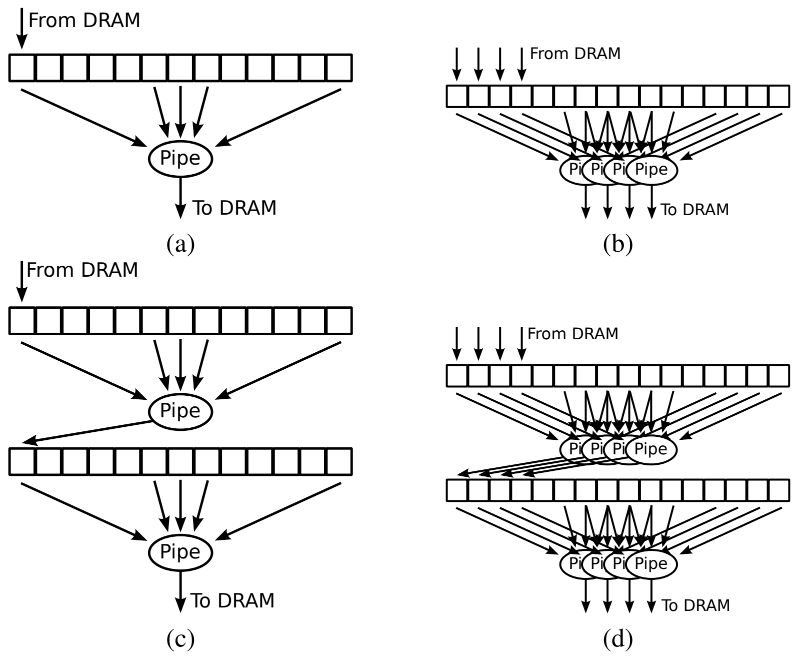

Figure 7.1: Kernel configurations with multi-pipelines and multi-steps. (a) Simple kernel. (b) Kernel with 4 multi-pipelines. (c) Kernel with 2 multi-steps. (d) Kernel with 4 multi-pipelines and 2 multi-steps.

The multi-step shown in Fig. 7.1(c) is a method to cascade multiple steps of arithmetic pipelines so that the update equation is applied more than once per off-chip memory access. In contrast to the multi-pipelines, this method can increase computational performance without increasing bandwidth requirements for off-chip memory. In addition, the number of steps can be tuned in a unit of one step. This enables finner-grained optimization unlike the multi-pipelines. On the other hand, this method has a large impact on BRAM usage in proportion as the number of steps increases.

The multi-pipelines and the multi-steps can be used at the same time. In addition, the number of pipelines N_p and the number of steps N_s can be independently assigned as shown in Fig. 7.1(d). Therefore determining the optimal combination of the parameters considering available resources on the FPGA and characteristics of the update equations is important.

7.2.3 Resource estimation model

The parameters for multi-pipelines and multi-steps can be easily changed by just assigning desired numbers to variables on MaxCompiler. Exploration of the best values of the multi-pipelines and the multi-steps aims to use FPGA resources for valid computation as much as possible. Since these techniques inevitably increase synthesized circuit size and time-cost of synthesis, placing and routing, it is important to narrow down the large parameter space to a realistic size by formulating resource-performance estimation models.

It shall be first addressed constraints for the user parameters by amounts of resources. The number of DSP modules is one of main factors to restrict the degree of the multi-pipelines and the multi-steps. A constraint for N_p and N_s imposed by the number of DSPs is defined as:

$$N_p * N_s * N_{\text{DSP}}^{(1,1)} \geq A_{\text{DSP}} \quad \alpha \tag{7.1}$$

where $N_{\text{DSP}}^{(1,1)}$ is the number of required DSPs for computation on the design of $(N_p, N_s) = (1, 1)$, A_{DSP} is the available number of DSPs on a target FPGA device and α is the number of DSPs which are used except for the arithmetic pipelines. Note that α smaller than A_{dsp}. In addition, on the assumption that BRAM utilization per step is linearly-increasing with N_p, a constraint for the parameters imposed by the number of BRAMs is defined as:

$$\left(N_{\text{BRAM}}^{(1,1)} + C_p * N_p\right) * N_s \geq A_{\text{BRAM}} \quad \beta \tag{7.2}$$

where $N_{\text{BRAM}}^{(1,1)}$ is the number of required BRAMs for computation on the design of $(N_p, N_s) = (1, 1)$, C_p is a constant factor for N_p, A_{BRAM} is available number of BRAMs on the FPGA, and β is the number of BRAMs which is used except for the pipelines. Equations (7.1) and (7.2) give us available combinations of N_p and N_s in the FPGA. Note that C_p should be a small value because spatially parallelized pipelines can share BRAMs.

It is assumed peak performance of a design is limited by the computational ability or memory bandwidth of the design, whichever is smaller. Given that an FPGA has infinite off-chip memory bandwidth, the ideal computational performance P_{compute} can be estimated as:

$$P_{\text{compute}} = \eta N_p N_s F_s \text{ [elements/sec]} \tag{7.3}$$

where η is computational efficiency and F_s is clock frequency of pipelines. Since stencil computations generally use neighboring elements to update each elements, computing all the elements in a block requires values of elements located outside of the block, dubbed halo or ghost zone. Due to the halo region, pipelines require additional clock cycles for a) loading elements at the halo region to shift registers and b) updating elements located the region for next time step. The computational efficiency η is defined as a ratio of written elements to read elements per block:

$$\eta = \frac{G_{\text{write}}}{G_{\text{read}}} \tag{7.4}$$

where G_{write} is the number of elements which are produced and written to off-chip memory and G_{read} is the number of elements which are loaded from off-chip memory per block computation. The number of written elements G_{write} is defined in a straightforward way, since it corresponds to the number of elements in a block:

$$G_{\text{write}} = B_W * B_H * Z \tag{7.5}$$

where B_W, B_H and Z are sizes of a block. On the other hand, definition of the number of read elements G_{read} is a bit more complex:

$$G_{\text{read}} = \left(B_W + 2\left\lceil\frac{S_d N_s}{T_W}\right\rceil T_W\right)\left(B_H + 2\left\lceil\frac{S_d N_s}{T_H}\right\rceil T_H\right)Z \tag{7.6}$$

where T_W and T_H are sizes of a tile, S_d is size of a stencil, a distance between an updated element and the farthermost element which is needed for the calculation. Note that the halo region is increased according to the degree of multi-steps. The ceiling functions are needed to reflect the fact that all the memory accesses are handled by a unit of the tile.

The memory bandwidth performance P_{mem} is described as:

$$P_{\text{mem}} = G_{\text{write}} \frac{T_{\text{mem}}}{B_{\text{mem}}} N_s \text{ [elements/sec]} \tag{7.7}$$

where T_{mem} is the peak memory bandwidth of the system and B_{mem} is required amount of data transfer between FPGA and memory including both read and write operations. The peak memory bandwidth T_{mem} can be described as:

$$T_{\text{mem}} = 8 * N_{\text{mem}} * 2F_{\text{mem}} \text{ [Byte/sec]} \tag{7.8}$$

where N_{mem} is the number of memory channels and F_{mem} is frequency of I/O bus clock. Since the amount of data transfer B_{mem} strongly depends on each application, users have to calculate this value manually. It is sometimes difficult to know the exact value of B_{mem} because data streams may be compressed. In this case, estimation value of the size of compressed data from a target compression rate is used.

Finally, the peak performance P can be expressed as:

$$P = \min\left(P_{\text{compute}}, P_{\text{mem}}\right) \tag{7.9}$$

and the best candidate of the parameters can be found as the pair which maximizes P satisfying (7.1) and (7.2).

7.3 Benchmark application

In this work, a heat conduction simulation was implemented as a benchmark application of 3-D stencil computation with the MaxCompiler and the DFE. The heat equation is expressed as:

$$\frac{\partial T(x, y, z, t)}{\partial t} = \alpha(x, y, z) \dagger^2 T(x, y, z, t) \tag{7.10}$$

where T is a temperature, t is a time variable and α is a thermal diffusivity. Eq. (7.10) can be approximated by a finite-difference equation with the explicit scheme as:

$$\begin{aligned} T_{i,j,k}(t + \Delta t) = T_{i,j,k}(t) + \alpha(x, y, z)\,\Delta t \Bigg(& \frac{T_{i+1,j,k}(t) \quad 2T_{i,j,k}(t) + T_{i \ \ 1,j,k}(t)}{\Delta x^2} \\ & + \frac{T_{i,j+1,k}(t) \quad 2T_{i,j,k}(t) + T_{i,j \ \ 1,k}(t)}{\Delta y^2} \\ & + \frac{T_{i,j,k+1}(t) \quad 2T_{i,j,k}(t) + T_{i,j,k \ \ 1}(t)}{\Delta z^2} \Bigg). \end{aligned} \tag{7.11}$$

7.3.1 Computation kernel

Figure 7.2 shows a part of the main computation kernel on MaxGenFD which applies a stencil computation to the 3-D array. Here, *in_T* is an input stream of temperature T, *alpha* is an input stream of constant α and *out_T* is an output stream of temperature T which becomes the output of this computation. At line 2, the kernel gets N_s which is given as external parameters. At line 4, user-defined function *getStencil()* generates a stencil. At lines 6-8, input stream *curr* is defined as an input temperature filed. At line 10, input stream C is defined for a constant field. At lines 12-19, Eq. (7.11) is computed. At line 21, the results are assigned for an output stream. The multi-step is handled at lines 14-19 and the multi-pipeline is handled at outside of the kernel by MaxGenFD.

```
/◦ Parameters ◦/
int n_step = engine_params.getNumStep();
/◦ Stencil ◦/
Stencil s = getStencil();
/◦ Input stream ◦/
FDVar curr =
 io.waveFieldInput("in_T", 1.0,
                    n_step ◦ (s_size/2));
/◦ Constants stream ◦/
FDVar C = io.earthModelInput("alpha",10,0);
/◦ Computations ◦/
FDVar s_input = curr;
FDVar result = constant.fdvar(0.0);
for ( int t= 0; t<n_step; ++t ) }
 FDVar laplacian =
  C ◦ convolve(s_input, ConvolveAxes.XYZ, s);
 result = s_input + laplacian;
 s_input = result;
|
/◦ Output stream ◦/
io.waveFieldOutput("out_T", result);
```

Figure 7.2: An outline of the computation kernel of the benchmark

Table 7.1: Parameters of target design

Parameter	Value
Block size $B_W * B_H$	$(192 * 120)$
Tile size $T_W * T_H$	$(16 * 12)$
Pipeline freq. F_s	150 MHz
Memory bus freq. F_m	303 MHz
Stencil size S_d	1

7.4 Evaluation

The performance of the implementations was evaluated on a PC with Intel Core i7-2600S 2.8 GHz, DDRIII 16 GB and a MAX3424A DFE running Cent OS 6.4. The Virtex-6 XC6VSX475T FPGA on the MAX3424A DFE has 297,600 LUTs, 595,200 FFs, 4,788 KBytes of BRAMs and 2,016 DSPs. Proposed designs for the FPGA were compiled and synthesized using MaxCompiler 2012.1, MaxGenFD 2012.1 and ISE 13.3. Synthesis, place and route for each parameter were retried up to 16 times using different cost tables until all the timing constraints were met. As a benchmark, a heat conduction simulation for $(512, 512, 512)$ of a computational space with 1,024 iterations was used. Table 7.1 shows common parameters for evaluated designs.

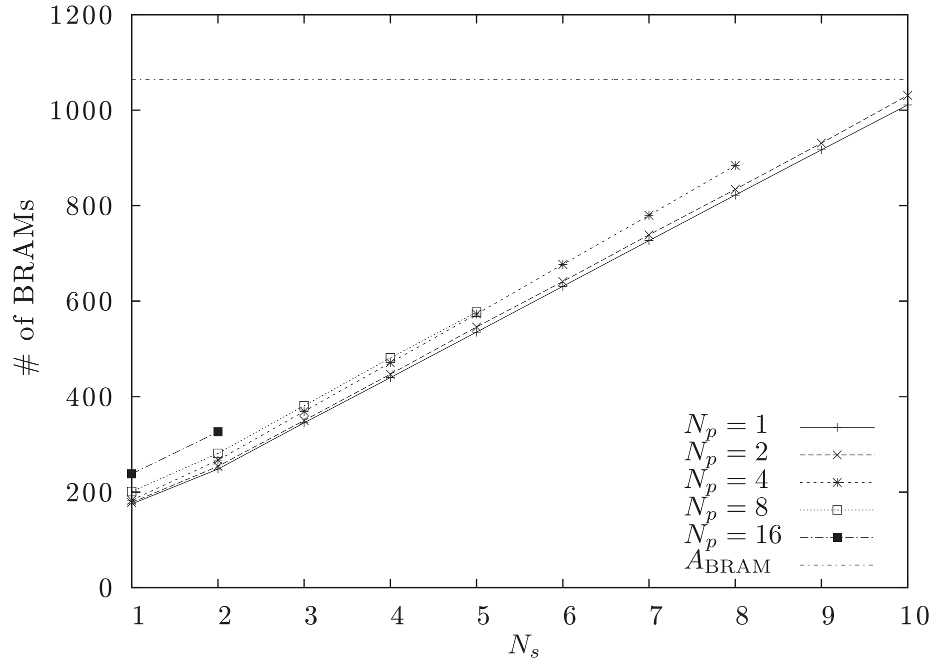

Figure 7.3: BRAM utilization

7.4.1 Resource usage

Resource utilization of all the combinations of N_p and N_s in this chapter is shown in Table 7.2. As a result of synthesis for the design of $\left(N_p, N_s\right) = (1, 1)$, $N_{\mathrm{DSP}}^{(1,1)}$ and α in (7.1) are estimated as 39 and 1, respectively. Then DSP usages N_{DSP} follows $N_{\mathrm{DSP}} = 39 * N_p * N_s + 1$, and (7.1) can be transformed to $N_p * N_s < 51$.

BRAM utilization of all the combinations is shown in Fig. 7.3. Since the lines are close to each other, the amounts of required BRAMs can be assumed to be sensitive to N_s, but not affected by N_p very much. This result shows BRAM usage can be managed by the value of N_s. In addition, the designs with larger N_p need fewer resources among the configurations which have the same value of $\left(N_p * N_s\right)$.

Figure 7.4 shows limitations on the parameter space and synthesis results for each parameter combination. Here, the dots indicate parameter combinations that were estimated as available and actually succeeded in synthesis. The crosses indicate parameter combiations which were failed to synthesis. The circled-dot indicates the best parameter at this time, $(N_p = 8, N_s = 5)$. Four lines on Fig. 7.4 illustrate each limitation imposed by the number of DSPs, the number of BRAMs, the available memory bandwidth and limitation of N_p in MaxGenFD, respectively. The memory limitation line locates at around $N_p = 44$ and no available parameters violate the memory limitation. This is because, while the minimum possible N_p value that exceeds the memory limitation is 64, the parameter $(N_p = 64, N_s = 1)$ is not available due to the DSP limitation. The best synthesized parameter combination is the fourth best one in the estimation model. Although a few parameter combination candidates anticipated to be more efficient were failed to synthesis, the performance model gives us a good direction to easily choose parameters without exhaustive parameter space exploration.

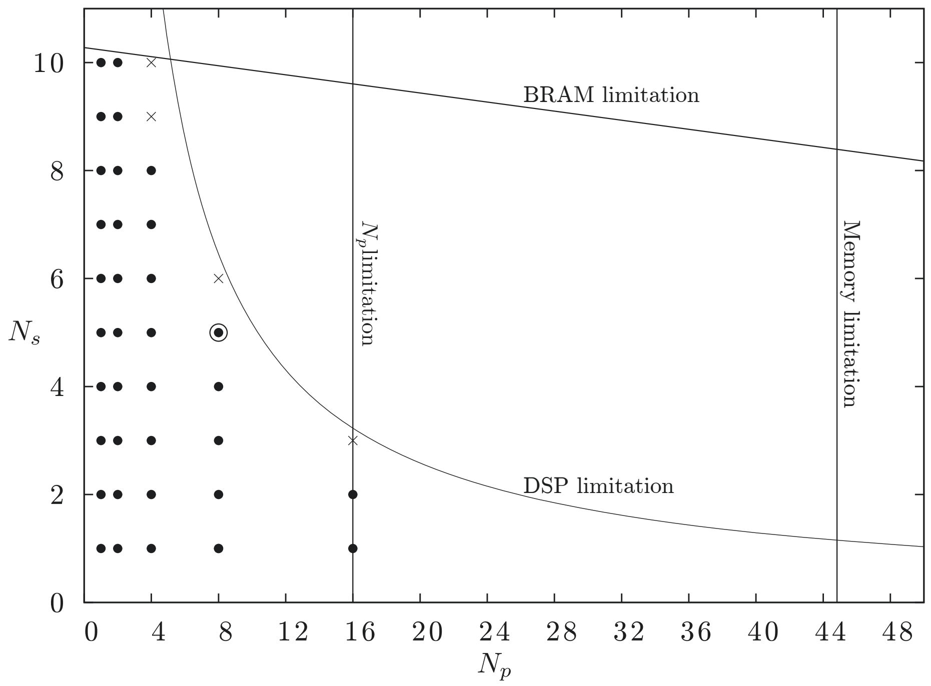

Figure 7.4: Estimated available parameters and synthesis results

Table 7.3 shows combinations of parameters which wire failed to build. Although some of them were expected as best parameters and resource utilization is acceptable, placing and routing phase could not meet design constraints. A combination of parameters ($N_p = 4, N_s = 10$) was failed because of BRAM utilization. The result indicates proposed estimation model has room to be improved in resource estimation taking into account place and routing.

7.4.2 Computational performance

Next evaluation intends to reveal the performances of each configuration shown in Fig. 7.5. The plotted marks indicate measured values and lines indicate estimation values by (7.3). All the configurations in this work have the same computational efficiency $\eta = 0.714$. The evaluation results show the performance is well estimated by the proposed model.

The evaluation also compared the DFE designs with multi-threaded CPU implementation with SIMD instructions on the host PC. The CPU implementation achieved the performance of $6.66 * 10^8$ grids/sec, and it is equivalent to the DFE designs which have $\left(N_p * N_s\right) = 6$. The best configuration $\left(N_p, N_s\right) = (8, 5)$ was about six times faster than the CPU implementation.

7.4.3 Power and energy consumption

Table 7.2 also shows measured values of power consumption as well as energy consumption by the accelerator for each configuration. These data were measured by inserting 10 mΩ shunt resistors into power-supply lines of a PCI-Express slot and an ex-

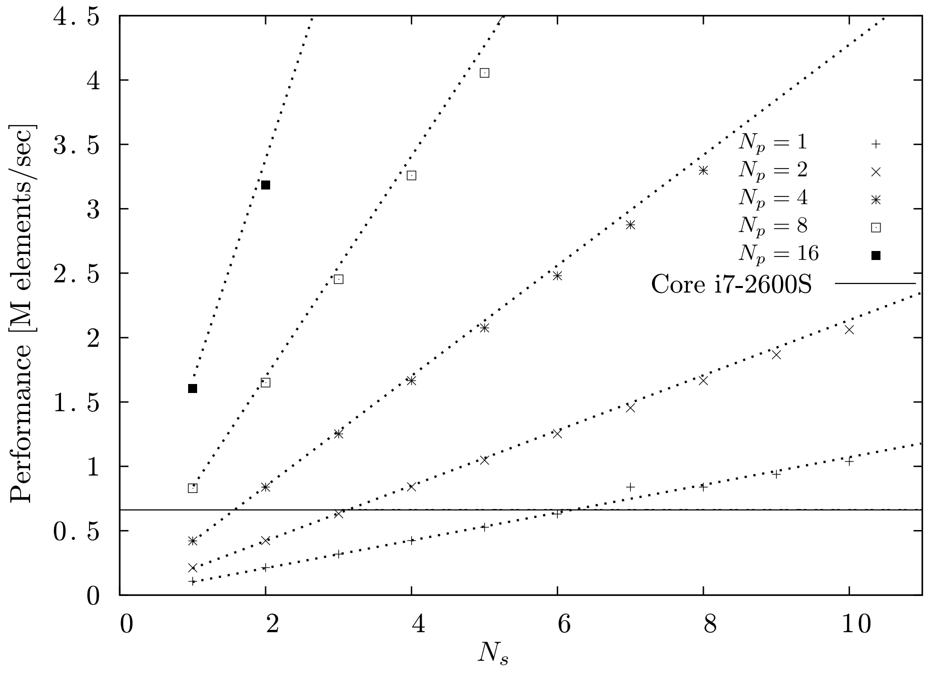

Figure 7.5: Performance of each design.

ternal power-supply connector of the DFE. Voltages between both ends of the resistors were amplified by instrumentation amplifiers, Analog Devices's AD623 and measured by an oscilloscope. Note that these measurements are a kind of early-stage experiment and aim to just compare among implementations of different parameter combinations in a relative manner.

The evaluation results show the power consumptions are mainly sensitive to N_s, suggesting BRAMs are dominant in the total power consumptions. Also in terms of energy consumptions, it was shown that the fastest configuration of the parameters was most efficient in this application.

7.5 Conclusion

This chapter presented the user-space parameters and the performance model for 3-D stencil computation with the MaxCompiler and the MaxGenFD on the MAX3424A Data Flow Engine. This study has used a heat conduction simulation as a benchmark application and evaluated resource utilization, performance and energy consumption for various parameter configurations. Performance comparison with multi-threaded CPU implementation with SIMD instruction shows the DFE implementation is about six times faster than the CPU implementation when a user chooses the best parameters. The empirical experiments demonstrated the fastest configuration was most effective in terms of energy consumption.

Table 7.2: Resource utilization, performance, electric power and energy of each configulation

N_p	N_s	(N_pN_s)	LUTs	FFs	BRAMs	DSPs	Performance [grids/sec]	Power [W]	Energy [J]
1	1	1	49,105	70,074	175	40	1.07e+08	61.32	78,764
1	2	2	54,490	77,285	248	79	2.13e+08	62.28	40,186
1	3	3	58,349	83,804	345	118	3.18e+08	62.76	27,124
1	4	4	65,907	90,434	440	157	4.23e+08	62.76	20,391
1	5	5	70,782	96,401	535	196	5.27e+08	66.60	17,368
1	6	6	75,727	103,328	631	235	6.30e+08	66.84	14,581
1	7	7	82,076	110,125	727	274	8.38e+08	69.36	11,375
1	8	8	88,695	117,145	822	313	8.39e+08	69.96	11,460
1	9	9	93,314	123,686	917	352	9.39e+08	71.64	10,485
1	10	10	99,420	129,822	1,011	391	1.04e+09	72.60	9,594
2	1	2	52,603	75,163	178	79	2.12e+08	62.40	40,453
2	2	4	59,690	85,445	254	157	4.23e+08	63.36	20,586
2	3	6	67,938	95,837	350	235	6.32e+08	63.96	13,909
2	4	8	78,077	107,087	447	313	8.41e+08	64.32	10,511
2	5	10	84,601	115,832	546	391	1.05e+09	65.04	8,513
2	6	12	93,069	128,687	641	469	1.25e+09	66.84	7,349
2	7	14	100,655	136,433	739	547	1.45e+09	71.28	6,756
2	8	16	109,664	150,707	834	625	1.67e+09	71.88	5,915
2	9	18	117,755	157,225	931	703	1.87e+09	73.20	5,379
2	10	20	126,181	167,481	1,031	781	2.06e+09	73.80	4,923
4	1	4	58,949	84,050	183	157	4.21e+08	61.80	20,175
4	2	8	74,231	102,657	267	313	8.38e+08	62.76	10,293
4	3	12	89,764	121,454	369	469	1.25e+09	63.60	6,992
4	4	16	104,261	139,497	471	625	1.66e+09	64.20	5,315
4	5	20	119,424	157,814	573	781	2.08e+09	66.60	4,400
4	6	24	131,246	176,207	677	937	2.48e+09	69.00	3,823
4	7	28	147,648	194,462	780	1,093	2.88e+09	70.20	3,350
4	8	32	185,949	595,200	884	1249	3.30e+09	71.16	2,963
8	1	8	75,543	105,508	201	313	8.30e+08	62.16	1,0293
8	2	16	104,747	140,537	281	625	1.65e+09	63.60	5,297
8	3	24	132,534	175,847	381	937	2.45e+09	66.00	3,702
8	4	32	157,202	211,046	481	1,249	3.26e+09	68.16	2,873
8	5	40	188,686	247,667	577	1,561	4.06e+09	70.80	2,396
16	1	16	111,929	150,289	238	625	1.60e+09	62.76	5,391
16	2	32	165,106	218,620	326	1,249	3.18e+09	68.04	2,940

Table 7.3: Failed parameters

N_p	N_s	$N_p * N_s$	Reason
4	9	36	Failed to place and route
4	10	40	Over BRAMs utilization (1,092 > 1,064)
8	6	48	Failed to place and route
16	3	48	Failed to place and route

Chapter 8

Ellipse Estimation using RANSAC on an FPGA

8.1 Introduction

This chapter presents FPGA implementation of image-based ellipse estimation for an embedded eye tracking system based on Starburst algorithm [68]. While the Starburst is known to be a robust algorithm, it requires high computational performance, making it difficult to be implemented as a compact and portable system. The goal here is to demonstrate highly efficient implementation of the Starburst algorithm on a compact FPGA platform without using any external memories.

The Starburst algorithm mainly consists of three process steps; (1) pre-processing for camera images, (2) extraction of feature points that represent a pupil contour, and (3) estimation of the best fit ellipse for the feature points. For the former two steps, a deep-pipelined stream-oriented image processing architecture is promising, which can achieve a real-time throughput at a relatively low clock frequency and does not require any external memories. This chapter firstly shows how the former two steps of the Starburst algorithm can be restructured to be fitted with this framework.

The last process step in which ellipses are estimated with the RANdom SAmple Consensus (RANSAC) algorithm [69], offers different computational properties. The RANSAC algorithm can robustly estimate an ellipse from a set of extracted feature points including some outliers. This robustness is achieved by a hypothesis-and-verify matching approach that consists of three steps; (1) randomly selecting a fixed number of feature points from the set of points including outliers, (2) generating hypothesis (ellipse parameters) from the selected points, and (3) verifying the generated hypothesis. The method repeats these three steps and finally returns the best hypothesis as a result. The RANSAC algorithm needs to estimate as many ellipses as possible for different point selection during a single camera frame, and this process easily becomes a performance bottleneck.

In contrast to large matrix solvers, few attention has been paid so far to small matrix manipulation on an FPGA. However, this issue is not obvious. For example, [70] reported the efficient algorithm to calculate an inverse of a 4x4 matrix with SIMD instructions was Cramer's rule, which is generally never used because of its high order of computational complexity. As is well known, especially for a small data set, execution performance becomes more sensitive to architectures and does not necessarily reflect computational complexity. Hence, in the latter half of this chapter, it compares three kinds of FPGA implementation of equation solvers and discusses which approach is appropriate for small matrix manipulation on an FPGA.

One of the highest developed FPGA implementation of ellipse estimation has been reported by Martelli et al., aiming for detection of circular road signs [71]. In their implementation, feature points are extracted using histogram stretching, intensity gradients, and the edge extraction and thinning method. However, in order to avoid hardware complication, they imposed a limitation on ellipses that they can detect; ellipses with major axis 0• and 90• from the x axis can only be detected. In contrast, a proposed implementation here does not have any limitations on ellipse to be detected, since it is often needed to detect inclined ellipses in eye tracking. In addition, there has been hardly any literature that focus on efficient solver implementation for a small simultaneous equation system on an FPGA.

The rest of the chapter is organized as follows. Section 8.2 presents the Starburst algorithm which includes pre-processing and the RANSAC. Section 8.3 presents the implementation details. Section 8.4 shows evaluation results and discussion. And finally, Section 8.6 shows the scope for future work and conclude the chapter.

8.2 Algorithms

8.2.1 Reflection removal

The first pre-processing step is the removal of reflections in a pupil, which often cause extraction of undesired feature points as shown in Fig. 8.1(a). To relieve this undesirable situation, a simple bilinear interpolation proposed in [72] was used. Let $\mathrm{I}(x, y)$ denote luminance of the pixel at (x, y). Let $\mathrm{R}(x, y)$ denote a binary reflection map, which is easily obtained based on a threshold luminance. $\mathrm{R}(x, y) = 1$ means that the pixel at (x, y) is in a reflection region and to be interpolated. Two points, (x_r, y) and (x_l, y), are required for the interpolation and their coordinates are calculated as follows:

$$x_r = \min\left\{x^{\infty}\colon \sum_{i=0}^{L\ \ 1} \mathrm{R}(x^{\infty}\ \ i, y) = 0, x^{\infty} > x\right\} \tag{8.1a}$$

$$x_l = \max\left\{x^{\infty}\colon \sum_{i=0}^{L\ \ 1} \mathrm{R}(x^{\infty} + i, y) = 0, x^{\infty} < x\right\} \tag{8.1b}$$

where L is a parameter on the filter size. Using these two points, interpolated luminance $\tilde{\mathrm{I}}(x, y)$ is calculated as:

$$\tilde{\mathrm{I}}(x, y) = \frac{\mathrm{I}(x_r, y) \times (x_r\ \ x) + \mathrm{I}(x_l, y) \times (x\ \ x_l)}{x_r\ \ x_l}. \tag{8.2}$$

Although this method executes the interpolation only in a horizontal direction, the majority of reflections can be effectively removed in practical environments as shown in Fig. 8.1(b).

8.2.2 Extraction of feature points of a pupil contour

The Starburst feature point extraction method starts to radially find feature points from a base point Ps, and returns a set of the nearest points which have larger intensity derivative than a threshold on each ray as shown in Fig. 8.1(c). In addition, another extraction process starts from firstly extracted features towards the base point to improve

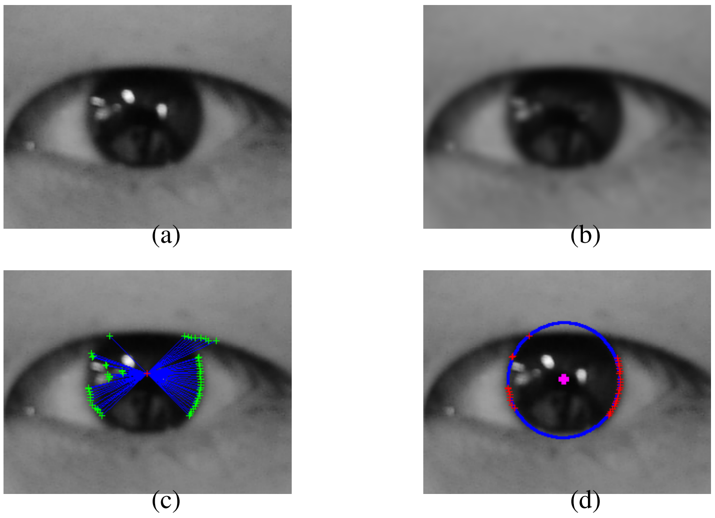

Figure 8.1: Overview of the Starburst algorithm. (a) Original image. (b) Reflection removal and box blur. (c) Starburst feature extraction. Green points denote feature points and the red point denotes the base point for the extraction. (d) Estimated ellipse (blue line) and the center point (purple). Red points denote inliers.

$$\begin{pmatrix} \sum x_i^2y_i^2 & \sum x_iy_i^3 & \sum x_i^2y_i & \sum x_iy_i^2 & \sum x_iy_i \\ \sum x_iy_i^3 & \sum y_i^4 & \sum x_iy_i^2 & \sum y_i^3 & \sum y_i^2 \\ \sum x_i^2y_i & \sum x_iy_i^2 & \sum x_i^2 & \sum x_iy_i & \sum x_i \\ \sum x_iy_i^2 & \sum y_i^3 & \sum x_iy_i & \sum y_i^2 & \sum y_i \\ \sum x_iy_i & \sum y_i^2 & \sum x_i & \sum y_i & \sum 1 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ D \\ E \end{pmatrix} = \begin{pmatrix} \sum x_i^3y_i \\ \sum x_i^2y_i^2 \\ \sum x_i^3 \\ \sum x_i^2y_i \\ \sum x_i^2 \end{pmatrix} \tag{8.4}$$

the robustness. This chapter only focused on the single step Starburst feature extraction for ease of compact hardware implementation.

8.2.3 RANSAC

The RANSAC is an iterative method that finds a model from a data set of points including outliers. At first RANSAC randomly samples a subset from the extracted feature points. The minimum size of subset depends on a target model, and an ellipse needs at least five points.

After sampling a subset, a hypothesis is generated from the subset. The system used the following ellipse equation,

$$x^2 + Axy + By^2 + Cx + Dy + E = 0 \tag{8.3}$$

where A, B, C, D and E are parameters to be estimated. Using the method of least squares, a system of simultaneous equations for estimating an ellipse is obtained as shown in Eq. (8.4).

The generated hypothesis is verified by successively substituting all the feature points and the obtained parameters into Eq. (8.3). The values of left-hand side are con-

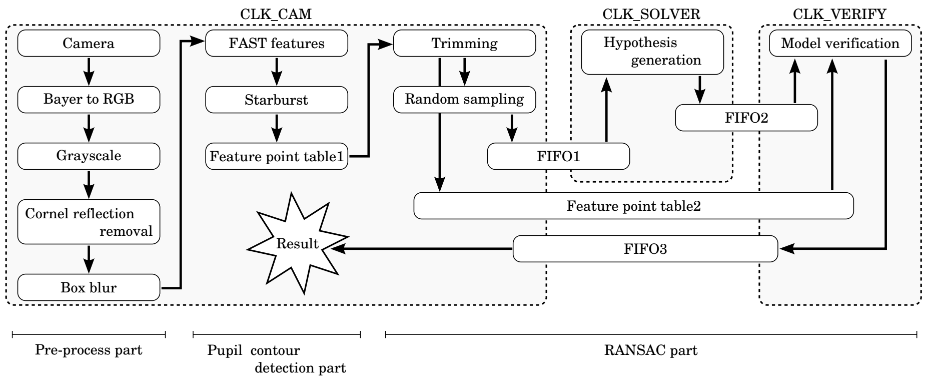

Figure 8.2: Overview of the proposed eye-tracking system.

sidered as *error* and the number of inliers is counted based on a threshold valued. The above three steps, random sampling, hypothesis generating and verifying, are repeated until a new dataset for the next frame image arrives. Finally the best hypothesis that has the maximum number of inliers is returned as estimated parameters.

8.3 Implementation

8.3.1 Design overview

Fig. 8.2 illustrates the proposed system. Pixel data input from the camera interface are streamed to Starburst module through cascaded pre-processing filters. The Starburst module detects up to 128 feature points and the Trimming module eliminates invalid feature points every one frame. The Random sampling module samples five points from the valid points set and stores them into FIFO1. The Hypothesis generation modules execute floating-point arithmetic operations for estimating elliptic parameters. The Model verification module counts the number of inliers for the generated hypothesis and updates the temporal best hypothesis when better hypothesis is found. Note that all the modules only access to memories which are inside an FPGA, e.g., FFs, BRAM and Distributed RAM.

8.3.2 Pre-processing

Although most of the pre-processing can be straightforwardly implemented on the streamed architecture framework described in Section 3.2.2, the reflection removal process is relatively complex. The process is split into two steps; the determination of envelop pixels and the interpolation. As a result, dynamic control flows were mitigated and all the pre-processing modules were implemented on the streamed architecture.

8.3.3 Pupil contour detection

The Starburst feature extraction process is also implemented on the streamed structure. The process is split into three parts; (1) calculation of intensity derivatives for all the pixels, (2) calculation of distances and angles from the center point, and (3) update of the feature points table.

Segmentation test is used to calculate intensity derivatives. The intensity derivatives are calculated by a segmentation test that is inspired by FAST corner detection [73]. By comparing the intensity of a candidate pixel with each point on a 16-point ring that surrounds the candidate pixel, it can extract possible directions in which the candidate pixel is able to be recognized as a feature point.

The distance and angle between the candidate point and center point are calculated using add, multiply and arctan operations. The *i*-th entry of the feature points table holds the coordinate of the feature point that is most recently found on the *i*-th direction and its distance from the center point. The table is updated when a new feature point that is nearer to the center for each direction is found. After scanning all the pixels, the table holds a set of the Starburst feature points for the corresponding frame. In the proposed implementation, the number of entries for the feature points table is 128.

8.3.4 RANSAC

As shown in Fig. 8.2, the RANSAC part includes three clock domains and asynchronous FIFOs and tables are provided for passing data between the clock domains. Using a double buffering technique with the dual-port RAMs, the Hypothesis generation module and the Model verification module work in parallel. The Random sampling module generates addresses for the feature point table randomly using a 32-stage liner feedback shift register.

The Hypothesis generation module consists of two main steps; generation of a system of simultaneous equations (Eq. (8.4)) based the received five feature points and calculation of ellipse parameters as solution of Eq. (8.4). The implementation includes three kinds of solvers for simultaneous equations based on Cramer's rule, Gauss-Jordan elimination and Doolittle LU decomposition.

8.3.5 Hypothesis generation

Cramer's rule

Consider a system of *n* linear equations for *n* unknowns as follows:

$$A\boldsymbol{x} = \boldsymbol{b} \tag{8.5}$$

where A denotes an $n * n$ matrix, $\boldsymbol{x}$ and $\boldsymbol{b}$ denote column vectors. According to Cramer's rule, the values for the unknowns are given by:

$$x_i = \frac{\|A_i\|}{\|A\|} \tag{8.6}$$

where A_i denotes a matrix formed by replacing the ***i***-th column of A by the column vector $\boldsymbol{b}$. The determinant $\|A\|$of an $n * n$ matrix A can be defined as:

$$\|A\| = \sum_{\sigma \lfloor S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} A_{i,\sigma_i} \tag{8.7}$$

where σ denotes a permutation of the set $\}1, 2, ..., n|$, σ_i denotes ***i***-th number of σ and $\operatorname{sgn}(\sigma)$ denotes the signature of σ which is 1 or 1. Due to its high order of computational complexity, the Cramer's rule is generally never considered as a practical solution. However, its dataflow has quite simple and regular structure with rich parallelism.

There are two kinds of tables in the design for the calculation, a memory table called *order table*, whose *j*-th entry contains *j*-th permutation σ_j for column vectors of the matrix ($3 * 5$ bit) and the value of $\operatorname{sgn}(\sigma_j)$ (1 bit) and five memories called *column* *tables* whose i-th memory contains i-th column vector of the matrix. These tables simplify the calculation of the determinant. At first, addresses for column tables and a sign are fetched from the order table. Then, the corresponding elements of the matrix are fetched from the five column tables. After multiplying the five values and the sign, the result is accumulated. Repeating this for 120 times (the number of permutations for $n = 5$), determinant of the matrix is obtained.

For ellipse estimation, six determinant of matrices, $|A|, |A_1|, \ldots, |A_5|$ are required. To calculate $|A_i|$, i-th column table also contains the vector $\boldsymbol{b}$. $|A_i|$ is calculated by using $b_{\sigma_{ji}}$ instead of $a_{i,\sigma_{ji}}$ from the i-th column table at the j-iteration, where σ_{ji} means i-th number of the permutation σ_j.

Since these multiplication steps are independent each other, pipelining can be fully applied. The final accumulation step consisting of an adder with 6-cycle latency allows the pipeline to avoid stalling by interleaving the calculation of determinants of the six matrixes. Hence, after executing 720 sets of multiplication, six determinants are obtained every clock cycle. Finally, the ellipse parameters are calculated by dividing the last five determinants with the first determinant.

This hypothesis generator consists of two 42-bit integer multipliers and two double precision floating point (FP) multipliers, one double precision FP adder, and one double precision FP divider with some format converters. The estimated ellipse parameters are output as single precision FP values.

Gauss-Jordan elimination

Gauss-Jordan elimination, also known as the sweep-out method is one of the commonly used algorithms for solving a system of simultaneous equations. While its computational complexity is higher than that of the Gaussian elimination which is another popular method, Gauss-Jordan elimination does not need backward substitution which is an essentially sequential process.

Given an N-by-$(N+1)$ matrix $M^{(0)} = [A\ \boldsymbol{b}]$ and integer K $(\geq N)$, eliminated matrix $M^{(K)}$ in the K-th step is defined as follows:

$$m_{i,j}^{(K)} = \begin{cases} m_{i,j}^{(K\ 1)} & \text{if } j < K, \\ \frac{m_{i,j}^{(K\ 1)}}{m_{K,K}^{(K\ 1)}} & \text{else if } i = K, \\ m_{i,j}^{(K\ 1)} \quad \frac{m_{i,K}^{(K\ 1)}}{m_{K,K}^{(K\ 1)}} m_{K,j}^{(K\ 1)} & \text{otherwise.} \end{cases} \tag{8.8}$$

In the RANSAC algorithm, the accuracy of the best solution among the repetitive trials is more important than that of each individual solution. Thus, the system requires no pivot exchanging, which makes the control flow sequential and complicated.

A proposed hypothesis generator based on the Gauss-Jordan elimination consists of cascaded five sub-modules, each of which corresponds to calculation of $M^{(K)}$ and consists of three single precision FP operators; adder, multiplier and divider. Each submodule can work in a macro pipelined manner, that is, a new hypothesis can be started to be generated after the first sub-module finish its calculation.

Doolittle LU decomposition

A third solver is based on Doolittle LU decomposition, which is also popular approach. This consists of three steps; decomposing a given matrix into L and U, solving $L\boldsymbol{y} = \boldsymbol{b}$, and solving $U\boldsymbol{x} = \boldsymbol{y}$. Compared to the Gauss-Jordan elimination, the computational complexity of LU decomposition is simplified.

Table 8.1: Resource usage of each implementation.

	CRAMER	GAUSS	LU	Available
FF	20,667	22,267	24,062	28,800
LUT	18,709	19,130	19,725	28,800
BRAM	39	39	34	48
DSP48	47	44	39	48

Table 8.2: Resource usage of solvers and dividers.

	CRAMER	GAUSS	LU
	FF/LUT	FF/LUT	FF/LUT
solver	11,735/9,816	13,335/10,229	15,130/10,816
div	5,982/3,207	6,760/3,840	6,760/3,840
ratio	51%/33%	51%/38%	47%/36%

Given an N-by-N matrix $U^{(0)} = A$ and integer $K(\geq N)$, the K-th step of calculation for matrices $U = U^{(N)}$ and $L = L^{(N)}$ are defined as follows:

$$u_{i,j}^{(K)} = \begin{cases} u_{i,j}^{(K\ 1)} & \text{if } i \geq K, \\ u_{i,j}^{(K\ 1)} \quad \frac{u_{i,K}^{(K\ 1)}}{u_{K,K}^{(K\ 1)}} u_{K,j}^{(K\ 1)} & \text{otherwise.} \end{cases} \tag{8.9}$$

$$l_{i,j}^{(K)} = \begin{cases} l_{i,j}^{(K\ 1)} & \text{if } j < K, \\ \frac{u_{i,j}^{(K\ 1)}}{u_{K,j}^{(K\ 1)}} & \text{else if } i \sim K, \\ 0 & \text{otherwise.} \end{cases} \tag{8.10}$$

The LU decomposition can be executed with a similar architecture to the Gauss-Jordan elimination. The solver includes three sub-modules each of which consists of adder, multiplier and divider for single precision FP operations. After the decomposition, $\boldsymbol{y}$ and $\boldsymbol{x}$ are calculated by the forward substitution and backward substitution, respectively. Each substitution module also requires three operators; adder, multiplier and divider for single precision FP values.

8.4 Evaluation and Discussion

8.4.1 Environment

An evaluation system was implemented on an ML501 prototype board equipped with a Xilinx Virtex-5 XC5VLX50 and an OmniVision OV9620 CMOS camera device using ISE 13.4 tool sets. Since the proposed FPGA implementation employs the streamed structure, pre-processing and the Starburst feature extraction achieved the performance of 62.5 fps, provided that the maximum frequency of the CLK_CAM exceeds 25 MHz which is the maximum frequency to communicate with the camera device. In the experiment, the clock frequencies for both CLK_SOLVER and CLK_VERIFY were set to 100 MHz. Note that any on-board memory devices were not used.

8.4.2 Resource usage

Table 8.1 shows resource usages for each implementation with the available resource amount on XC5VLX50 FPGA, and Table 8.2 shows how much portion of each solver is occupied by dividers.

The solver based on Cramer's rule (CRAMER) showed the lowest resource usage in FFs and LUTs despite of only this solver requires double precision FP operators due to accuracy requirements. This compact implementation is due to CRAMER needs the smallest number of operators among the three solvers and most of the required operators are multipliers which can be efficiently built with DSP48E hard macro modules.

As Table 8.2 shows, FP dividers were dominant in terms of resource usage for each solver. Although the Gauss-Jordan elimination (GAUSS) and the LU decomposition (LU) use single precision FP operators, the largest part is still occupied by dividers since they use multiple dividers. For GAUSS and LU solvers, it seems difficult to reduce the resource usage without performance degradation. However, for the CRAMER solver, a fully pipelined FP divider is used only for five division operations. Thus, there should be room to reduce the required resources while keeping the performance by changing the structure of the divider.

8.4.3 Power consumption

Table 8.3 shows performance and power consumptions of each implementation. The power consumptions were measured by inserting a 1- ohm shunt resistor between the board and DC power supply. Note that these measured values include the power consumed by the camera device, HDMI display interface, debug interface (including memories), and so forth.

The CRAMER showed the highest power consumption, which is about 5.6% higher than the lowest one, despite the solver showed the lowest resource usage. The comparison results suggest that power consumptions are more sensitive for utilization of DSP48Es and BRAMs rather than FFs and LUTs. The power consumption of the system was only 3.34 watts even for the CRAMER, which demonstrates the effectiveness of FPGAs in terms of a power performance ratio. Capability of tight and efficient integration of dedicated arithmetic and I/O interface makes a huge contribution to the advantage of the FPGA implementation.

8.4.4 Throughput

A metric *throughput* is defined as the number of hypothesis generation per second. As shown in Table 8.3, the GAUSS solver showed the highest throughput and the lowest latency and this value corresponds to approximately 9 times of that for the CRAMER solver. This is due to the high order of computational complexity of the Cramer's rule. However, the performance difference was smaller than the difference in the computational complexity because of the higher pipeline usage rate of arithmetic units in the CRAMER solver.

For the RANSAC algorithm, improving the throughput of the hypothesis generation improves the accuracy of ellipse fitting. As shown in Fig. 8.3, even the CRAMER solver, which achieved the lowest throughput, successfully estimates reasonable ellipse parameters for practical images. In this sense, it can be considered that the throughputs obtained by the three solvers are enough for ellipse estimation, while detailed accuracy evaluation will be needed for future work.

Table 8.3: Performance of each implementation.

	CRAMER	GAUSS	LU
Power [W] (Solver/Total)	0.33/3.34	0.28/3.29	0.15/3.16
Latency [us]	8.69	3.05	4.99
Throughput	$1.1 * 10^6$	10^7	$3.8 * 10^6$

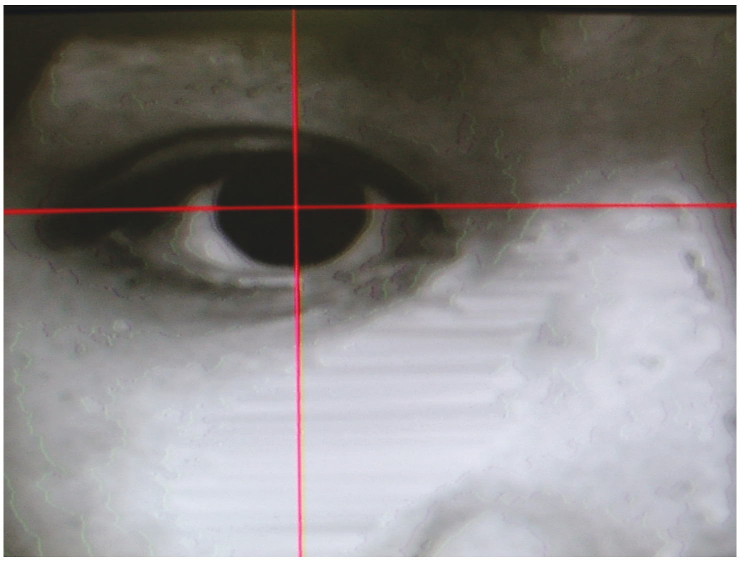

Figure 8.3: Ellipse estimation result using the CRAMER solver. Crossing point shows the center of eye.

8.5 Comparison between RTL design and the MaxCompiler

The application was also implemented with MaxCompiler 2012.1.1 on MAX2336B DFE. The implementation used one of two FPGAs on the DFE and kernel frequency was set 100 MHz.

8.5.1 Resource utilization

Resource utilization of each module generated by the MaxCompiler is shown in Table 8.4. The HypothesisGen and Verify modules which play a central role in the RANSAC algorithm used half of LUTs and FFs and 80% of the DSP48E modules for whole design. In the preprocess modules, the reflection removal, the BoxBlur, and the feature extraction modules used many resources. The MaxCompiler automatically generates data scheduling modules and these occupy 30% of LUTs and 56% of FFs for preprocess modules. In addition to user description, the PCIe interface generated by MaxCompiler occupies 25% of LUTs and FFs for whole design. The PCIe interface includes a data transfer mechanism between host computer and the DFE.

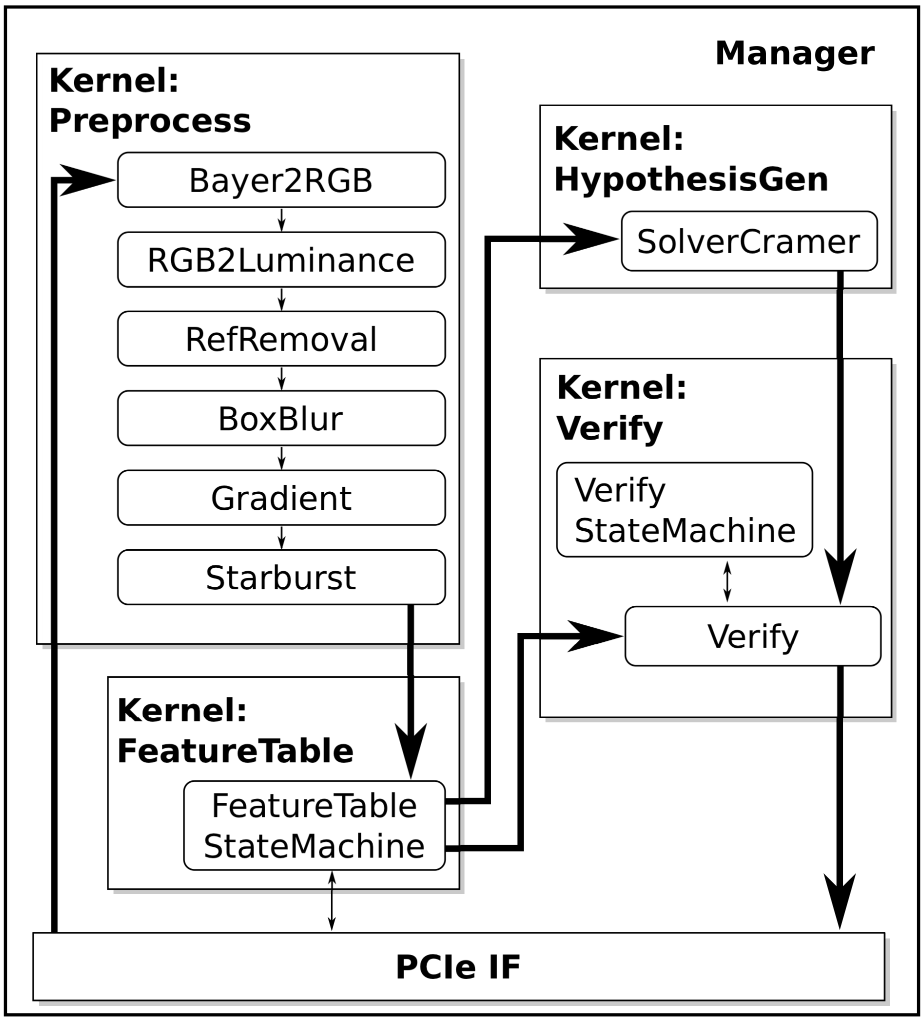

Figure 8.4: Overview of MaxCompiler design.

Table 8.4: Resource utilization - MaxCompiler

Module	LUTs	FFs	BRAMs	DSP48Es
Preprocess				
- Bayer2RGB	165	161	0	0
- RGB2Luminance	185	185	0	0
- RefRemoval	609	834	1	4
- BoxBlur	1,052	852	0	0
- Gradient	240	192	0	0
- Starburst	1,003	1,059	2	9
- Others	1,478	3,505	0	0
FeatureTable	739	800	0	2
HypothesisGen				
- MatrixGen	3,642	4,223	0	42
- Cramer	8,027	11,239	0	40
Verify	4,737	5,985	8	14
FIFOs	742	1,083	9	0
PCIe	7,407	10,465	19	0
Others	1,883	1,922	14	0
Total	31,909	41,905	82	111
Available	207,360	207,360	324	192

Table 8.5 shows resource utilization of the implementation in Verilog HDL. The utilization shows a trend that the implementation in Verilog HDL needs smaller area than the implementation with MaxCompiler. This is due to: (1) MaxCompiler does not provide resource sharing; (2) Kernels of MaxCompiler massively use DSP blocks; (3) MaxCompiler generates deeper-pipelines to achieve higher frequency; (4) Control of streams needs additional resources; and (5) Difference between actually generated design and indentation of user description.

8.5.2 Throughput

Throughput comparison between the HDL and the MaxCompiler implementation was performed with the clock frequency of 100 MHz for the ellipse estimation part and throughput of 62.5 FPS for input video stream. The throughput of the Verilog HDL implementation was about $1.10 * 10^5$, while that of the MaxCompiler implementation was about $1.05 * 10^5$. This result shows that the MaxCompiler implementation achieved the equivalent level in performance with the HDL implementation.

One reason for the throughput decrease by the MaxCompiler compared with the HDL is that the MaxCompiler implementation processes the feature extraction and ellipse estimation steps sequentially. While the HDL implementation processes these steps in parallel, controls are returned back to the host processor from DFE at each video frame in the MaxCompiler implementation. The both steps therefore have to wait until the host processor invokes the DFE.

Table 8.5: Resource utilization - Verilog HDL

Module	LUTs	FFs	BRAMs	DSP48Es
Preprocess				
- Bayer2RGB	107	141	1	0
- RGB2Luminance	41	28	0	0
- RefRemoval	1,120	1,600	4	2
- BoxBlur	778	1,226	2	0
- Gradient	403	426	2	0
- Starburst	543	476	0	2
FeatureTable	115	154	1	0
HypothesisGen				
- MatrixGen	2,937	2,691	0	0
- Cramer	6,892	9,051	6	28
Verify	2,803	2,306	1	14
Others	2,970	2,568	22	1
Total	18,709	20,667	39	47
Available	28,800	28,800	48	48

Table 8.6: Line-based comparison of Verilog HDL and the MaxCompiler.

Module	MaxCompiler [Lines]	Verilog HDL [Lines]
Preprocess	1,300	3,530
FeatureTable	441	632
HypothesisGen	261	1,309
Verify	412	312
Others	631	3,321
Total	3,045	9,104

8.5.3 Lines of source codes

Table 8.6 shows the number of lines of source codes for the MaxCompiler and Verilog HDL implementation for the application. The table does not include the lines of automatically generated codes. There are noticeable differences between the MaxCompiler and HDL implementation in the results for Preprocess and HypothesisGen kernels. Both kernels form deep-pipelined structure, while other kernels include FeatureTable and Verify kernels are described as state machines. This result shows the MaxCompiler is easier to describe pipelined-structure than the HDL, but not for state machines.

8.6 Summary

This chapter presented that deep-pipelined FPGA implementation of real-time ellipse estimation for eye tracking system is achieved high performance (62.5 fps) and low power consumption (3.34 watt) without using any external memories. The ellipse estimation consists of various processes, which include pre-processing, feature extraction and the RANSAC algorithm. This study focused the hypothesis generation process, which solves a system of simultaneous equations repeatedly. The result of evaluation shows resource usage, power consumption and throughput of three solvers. While the optimal algorithm needs to be chosen depending on the amount of resources on FPGAs and required criteria, the FPGA based system that consists of streamed structure and hypothesis generator with FP operators is promised as a better solution for the application.

Chapter 9

Conclusion

This dissertation aimed to reveal the methodology to efficiently map various applications to parallel architectures consisting of an array of arithmetic logic and memory elements, such as GPUs and FPGAs. Especially, this dissertation focused on making the best use of on-chip memories to achieve a high degree of execution efficiency. This study will contribute to reduce the cost of implementation of applications on accelerators to achieve high computational performance and low energy consumption.

The stencil computation was a main target of this study and various applications which can be expressed as a kind of stencil computations were implemented on the parallel architectures as case studies. The stencil computation is efficiently processed on GPUs by decomposing the computational space into blocks, and is processed on FPGAs as stream-oriented process. This study demonstrated concrete application mapping methodologies in this design pattern. In addition, the high-level synthesis tool, which enables us to design hardware using high abstraction layer, was also focused. Some important user space parameters of the MaxCompiler for stencil computations were pointed out, and estimation models to describe relationships among the parameters, computational performance and resource utilization were presented.

The first project was the implementation of Smith-Waterman algorithm on GPUs. This implementation achieved a throughput of 12.66 GCUPS on the NVIDIA GTX 295 GPU, and it was the fastest performance in speed as a GPU implementation of the algorithm at the time. The implementation also achieved a throughput of 43.05 GCUPS when 4 GPUs are used. These performances were led by decreasing the cost of synchronization focused on the warp level synchronization. The key to the implementation was a divide and conquer approach to the problem to process each sub-problem by a warp. In addition, memory accessing optimization, including data caching by on-chip memory and coalescing of external memory access from a warp, contributed to the increase in performance. As well as these optimizations, loop unrolling and instruction scheduling which are known as general optimization techniques not only for GPU but for existing computer architectures also had a large impact on the performance.

The second project was the implementation of 3-D FDTD electromagnetic simulation on a GPU. Since this project aims to accelerating analysis of a microstrip antenna, the simulation must handle the artificial fundamental equations for boundary conditions as well as the equations for the physical phenomenons. The split PML boundary condition used for the implementation requires up to 12 additional elements for each grid point and additional computations. In addition, the boundary condition needs various types of equations depending on computational regions. The implementation therefore focused on the optimizations on the boundary regions. In particular, the periodic boundary condition was introduced into the split PML boundary condition to decrease the load of computation and memory access by focusing the granularity of control flow of GPUs. The achieved throughput for the implementation was 55.8 % of the peak memory bandwidth of the GPU when the accesses for padding region were not taken account. If padding accesses were included, it corresponded to 70.5 % of the peak. Additionally, comparison of antenna characteristics obtained from the simulation with measured results showed reasonable concordance between the two. This result indicates validity of the implementation.

The third project was the implementation of a human detection system by the HOG features from a video stream on an FPGA. This project was one of the earliest FPGA implementation of human detection systems with streamed HOG feature extraction. The most complex part in the system is the HOG feature extraction process. This study used the stream-oriented architecture consisting of FFs, distributed memories, and BRAMs. In addition, approximate calculations were introduced to reduce the resource utilization as well as bit-width optimization at each stage of processes. As a result, the implementation achieved a real-time throughput of approximately 60 FPS, showing a potential of the stream-oriented approach. The implementation needed about 75 % of BRAMs on the FPGA for FIFOs of the stream-oriented architecture. This result indicates the amount of BRAM will be the main restriction for stream-oriented video-stream processing for small-size FPGAs.

The fourth project was the implementation of heat spreading simulation using MaxCompiler, a high-level synthesis tool, and the MaxGenFD, a framework for stencil computation on the MaxCompiler. However, the MaxCompiler and the MaxGenFD offer various design alternatives for stencil computation on FPGA accelerators, forcing users of the tools to determine adequate design parameters for pipeline parallelization. This project aims to define important user space parameters and to establish the estimation models for computational performance, resource utilization, and a parameter space. In addition, this project measured the power consumption of the accelerator connected via the PCI Express bus, by inserting shunt resistors into power lines of the accelerator. The result of the measurement showed that BRAM utilization gives the largest impact on energy consumption when two implementations have the same performance in speed.

The last project was the implementation of ellipse fitting process on a video stream using the RANSAC algorithm on an FPGA. This project consists of two steps: feature points extraction from the video stream and ellipse fitting using the feature points. Since the former process includes the Starburst feature extraction algorithm which requires random memory access, the algorithm has been modified from the original one to map the algorithm on the stream-oriented architecture. That modification to the design has achieved exclusion of external memory and random access to the memory for the algorithm using additional logic elements and on-chip memories. At the ellipse fitting step, three types of algorithms to solve a system of simultaneous equations were implemented. As a result of evaluations showed that an implementation of the Cramer's rule has lowest resource utilization and enough accuracy for the application. In addition, this study also implemented the application using the MaxCompiler. The MaxCompiler allowed us to implement the application from one third of description of HDL in lines, while HDL implementation needed fewer resources because of resource sharing and other optimizations. This is due to the MaxCompiler focused on generating high-throughput designs rather than low resource utilization.

Through the empirical experiments of the five projects, Smith-Waterman algorithm, FDTD method for electro-magnetical simulation, HOG feature extraction, 3-D heat spreading simulation and ellipse estimating using the RANSAC algorithm, this study specifically demonstrated a basic strategy to efficiently implement various applications on both GPUs and FPGAs. In addition, evaluation results of each project showed both architectures have advantages over existing microprocessor architectures in computational performance and energy consumption. The above results were achieved by not only straightforward mapping of existing algorithms on GPUs and FPGAs, but also needing modifications of the algorithms. The modifications especially intended to adapt the algorithms to the stream-oriented process by eliminating random accesses for effective use of on-chip memories and logic elements. On the other hand, there is no bible for automatic modification of existing algorithms for GPUs or the stream-oriented architectures on FPGAs, and therefore sometimes the modifications require a lot of effort from designers and implementers. That issue can be reduced by promotion of a *design blocks catalog* which consists of relatively small design examples used inside a large design framework like the stream-oriented architecture. In addition, it is important challenge to create design frameworks not only the stream-oriented architecture but also others, suck as the one intends to tree/graph data structures. This study is expected to be referred as important knowledge to efficiently utilize fast and distributed on-chip memories, which can be applied to implement various applications on parallel computing accelerators.

Bibliography

[1] NVIDIA. GeForce GTX 295 Overview. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-295`.

[2] NVIDIA. Tesla C1060 Computing Processor Board. `http://www.nvidia.com/docs/IO/43395/BD-04111-001_v06.pdf`.

[3] Intel. Intel Core i7-900 Desktop Processor Extreme Edition Series and Intel Core i7-900 Desktop Processor Series Datasheet, Vol 1.

[4] ATi. ATi RADEON X800 3D Architecture White Paper.

[5] T. Chu, J. Dai, D. Qian, W. Fang, and Y. Liu. A Novel Scheme for High Performance Finite-Difference Time-Domain (FDTD) Computations Based on GPU. *Algorithms and Architectures for Parallel Processing*, pp. 441–453, 2010.

[6] M.J. Inman, A.Z. Elsherbeni, J.G. Maloney, and B.N. Baker. GPU based FDTD solver with CPML boundaries. In *IEEE Antennas and Propagation Society Int'l Symp.*, pp. 5255–5258. IEEE, 2007.

[7] D. Koufaty and D.T. Marr. Hyperthreading technology in the netburst microarchitecture. *Micro, IEEE*, Vol. 23, No. 2, pp. 56–65, 2003.

[8] Qualcomm. "snapdragon s4 processors: System on chip solutions for a new mobile age; white paper". 2013.

[9] NVIDIA. "whitepaper nvidia tegra 4 family gpu architecture". 2013.

[10] Brent Przybus. Xilinx redefines power, performance, and design productivity with three new 28 nm fpga families: Virtex-7, kintex-7, and artix-7 devices. *Xilinx White Paper*, 2010.

[11] Impulse Accelerated Technologies. Impulse C. `http://www.impulseaccelerated.com/`.

[12] Xilinx. Vivado HSL Design. `http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/index.htm`.

[13] SRC Computers. CARTE PROGRAMMING ENVIERONMENT. `http://www.srccomp.com/`.

[14] Maxeler Technologies. MaxCompiler. `http://www.maxeler.com/`.

[15] Maxeler Technolgies. *MaxCompiler Tutorial*, 2013.2.1 edition, 2013.

[16] CUDA ZONE. `http://www.nvidia.com/object/cuda_home.html`.

[17] NVIDIA CUDA programming guide 2.3.

[18] NVIDIA CUDA Best Practices Guide 2.3.

[19] NVIDIA Optimizing Parallel Reduction in CUDA.

[20] ”XILINX”. ”achieving higher system performance with the virtex-5 family of fpgas”. 2006.

[21] Hidenori Matsubayashi, Shinsuke Nino, Toru Aramaki, Yuichiro Shibata, and Kiyoshi Oguri. Retrieving 3-d information with FPGA-based stream processing. In *Proc. 16th ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, pp. 261–261, February 2008.

[22] K. Dohi, Y. Yorita, Y. Shibata, and K. Oguri. Pattern compression of FAST corner detection for efficient hardware implementation. In *Proc. IEEE 21st Int. Conf. Field Programmable Logic and Applications*, pp. 478–481, September 2011.

[23] K. Dohi, Y. Hatanaka, K. Negi, Y. Shibata, and K. Oguri. Deep-pipelined fpga implementation of ellipse estimation for eye tracking. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 458–463. IEEE, 2012.

[24] Transcoso M. Charalambous, P and A. Stamatakis. Intial experiences porting a bioinformatics application to a graphics processor. *In Processings of 10th Panhellenic Conference on Informatics.*, 2005.

[25] T Smith and M Waterman. Identification of common molecular subsequences. *J Molecular Biology*, Vol. 147, pp. 195–197, 1981.

[26] O Gotoh. An improved algorithm for matching biological sequences. *J Mol Biol*, Vol. 162, pp. 707–708, 1982.

[27] SF Altschul, W Gish, W Miller, EW Myers, and DJ Lipman. Basical local alignment search tool. *J Mol Biol*, Vol. 215, No. 3, pp. 403–410, 1990.

[28] S. Henikoff and JG Henikoff. Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences*, Vol. 89, No. 22, p. 10915, 1992.

[29] M Farrar. Striped smith-waterman speeds database searches six times over other simd implementations. *Bioinformatics*, Vol. 23, No. 2, pp. 156–161, 2007.

[30] A Szalkowski, C Ledergerber, P Krahenbuhl, and C Dessimoz. SWPS3 - fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2. *BMC Research Notes*, Vol. 1, p. 107, 2008.

[31] MS Farrar. Optimizing smith-waterman for the cell broadband engine.

[32] SA Manavski and G Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, Vol. 9, No. Suppl 2, p. S10, 2008.

[33] T Rognes and E Seeberg. Six-fold speedup of smith-waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, Vol. 16, No. 8, pp. 699–706, 2000.

[34] K. Benkrid, Ying Liu, and A. Benkrid. A highly parameterized and efficient fpga-based skeleton for pairwise biological sequence alignment. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, Vol. 17, No. 4, pp. 561–570, 2009.

[35] Yongchao Liu, Douglas Maskell, and Bertil Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, Vol. 2, No. 1, p. 73, 2009.

[36] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[37] K. Yee. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *IEEE Trans. on antennas and propagation*, Vol. 14, No. 3, pp. 302–307, 1966.

[38] Allen Taflove and M E. Brodwin. Numerical solution of steady-state electromagnetic scattering problems using the time-dependent Maxwell's equations. *IEEE Trans. on Microwave Theory and Techniques*, Vol. 23, No. 8, pp. 623–630, 1975.

[39] S.D. Gedney. An anisotropic perfectly matched layer-absorbing medium for the truncation of FDTD lattices. *IEEE Trans. on Antennas and Propagation*, Vol. 44, No. 12, pp. 1630–1639, 1996.

[40] J.A. Roden and S.D. Gedney. Convolution PML (CPML): An efficient FDTD implementation of the CFS-PML for arbitrary media. *Microwave and optical technology lett.*, Vol. 27, No. 5, pp. 334–339, 2000.

[41] S.A. Cummer. A simple, nearly perfectly matched layer for general electromagnetic media. *Microwave and Wireless Components Lett., IEEE*, Vol. 13, No. 3, pp. 128–130, 2003.

[42] J.P. Bérenger. *Perfectly matched layer (PML) for computational electromagnetics*. Morgan & Claypool Publishers, 2007.

[43] Keisuke Dohi, Yuichiro Shibata, Kiyoshi Oguri, and Takafumi Fujimoto. GPU implementation and optimization of electromagnetic simulation using the FDTD method for antenna designing. *SIGARCH Comput. Archit. News*, Vol. 39, pp. 26–31, December 2011.

[44] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pp. 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[45] Paulius Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pp. 79–84, New York, NY, USA, 2009. ACM.

[46] T. Nagaoka and S. Watanabe. A GPU-based calculation using the three-dimensional FDTD method for electromagnetic field analysis. In *Annu. Int'l Conf. of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pp. 327–330. IEEE, 2010.

[47] J.P. Bérenger. A perfectly matched layer for the absorption of electromagnetic waves. *J. Comput. Phys.*, Vol. 114, No. 2, pp. 185–200, 1994.

[48] CJ Railton and JP McGeehan. Analysis of microstrip discontinuities using the finite difference time domain technique. In *IEEE MTT-S Int'l Microwave Symp. Digest*, pp. 1009–1012. IEEE, 1989.

[49] Huiling Jiang and Hiroyuki Arai. Analysis of Computation Error in Antenna's Simulation by Using Non-Uniform Mesh FDTD. *IEICE Trans. on communications*, Vol. 83, No. 7, pp. 1544–1553, 2000-07-25.

[50] Takafumi Fujimoto and Kazumasa Tanaka. Stacked rectangular microstrip antenna with a shorting plate for dual band (vics/etc) operation in its. *IEICE transactions on communications*, Vol. 90, No. 11, pp. 3307–3310, 2007-11-01.

[51] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, pp. 886–893. IEEE, 2005.

[52] Y. Freund and R. Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pp. 23–37. Springer, 1995.

[53] N. Dalal, B. Triggs, and C. Schmid. Human detection using oriented histograms of flow and appearance. *Computer Vision–ECCV 2006*, pp. 428–441, 2006.

[54] P. Dollár, C. Wojek, B. Schiele, and P. Perona. Pedestrian detection: A benchmark. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 304–311. IEEE, 2009.

[55] C.H. Kuo and R. Nevatia. Robust multi-view car detection using unsupervised sub-categorization. In *Applications of Computer Vision (WACV), 2009 Workshop on*, pp. 1–8. IEEE, 2009.

[56] T.P. Cao and G. Deng. Real-time vision-based stop sign detection system on fpga. In *Computing: Techniques and Applications, 2008. DICTA'08. Digital Image*, pp. 465–471. IEEE, 2008.

[57] R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto, and Y. Nakamura. Hardware architecture for hog feature extraction. In *Intelligent Information Hiding and Multimedia Signal Processing, 2009. IIH-MSP'09. Fifth International Conference on*, pp. 1330–1333. IEEE, 2009.

[58] M. Komorkiewicz, M. Kluczewski, and M. Gorgon. Floating point hog implementation for real-time multiple object detection. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pp. 711–714. IEEE, 2012.

[59] Kosuke Mizuno, Yosuke Terachi, Kenta Takagi, Shintaro Izumi, Hiroshi Kawaguchi, and Masahiko Yoshimoto. Architectural study of HOG feature extraction processor for real-time object detection. In *Signal Processing Systems (SiPS), 2012 IEEE Workshop on*, pp. 197–202. IEEE, 2012.

[60] Y. Yamauchi, C. Matsushima, T. Yamashita, and H. Fujiyoshi. Relational hog feature with wild-card for object detection. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pp. 1785–1792. IEEE, 2011.

[61] G. Overett, L. Petersson, N. Brewer, L. Andersson, and N. Pettersson. A new pedestrian dataset for supervised learning. In *Intelligent Vehicles Symposium, 2008 IEEE*, pp. 373–378. IEEE, 2008.

[62] Oliver Pell, Oskar Mencer, KuenHung Tsoi, and Wayne Luk. Maximum performance computing with dataflow engines. In Wim Vanderbauwhede and Khaled Benkrid, editors, *High-Performance Computing Using FPGAs*, pp. 747–774. Springer New York, 2013.

[63] K. Negi, K. Dohi, Y. Shibata, and K. Oguri. Deep pipelined one-chip FPGA implementation of a real-time image-based human detection algorithm. In *Proc. Int. Conf. Field-Programmable Technology*, pp. 1 –8, December 2011.

[64] Yukinori Sato, Yasushi Inoguchi, Wayne Luk, and Tadao Nakamura. Evaluating reconfigurable dataflow computing using the himeno benchmark. In *Proceedings of International Conference on ReConFigurable Computing and FPGAs*, pp. 1–7, 2012.

[65] Heiner Giefers, Christian Plessl, and Jens Förstner. Accelerating finite difference time domain simulations with reconfigurable dataflow computers. In *Proceedings of 4th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, pp. 33–38, 2013.

[66] Kentaro Sano. FPGA-Based Systolic Computational-Memory Array for Scalable Stencil Computations. In Wim Vanderbauwhede and Khaled Benkrid, editors, *High-Performance Computing Using FPGAs*, pp. 279–303. Springer New York, 2013.

[67] Tomoyoshi Kobori and Tsutomu Maruyama. A high speed computation system for 3d fchc lattice gas model with fpga. In Peter Cheung and GeorgeA. Constantinides, editors, *Field Programmable Logic and Application*, Vol. 2778 of *Lecture Notes in Computer Science*, pp. 755–765. Springer Berlin Heidelberg, 2003.

[68] Dongheng Li, D. Winfield, and Derrick J. Parkhurst. Starburst: A hybrid algorithm for video-based eye tracking combining feature-based and model-based approaches. In *Proc. IEEE Vision for Human-Computer Interaction Wrokshop at CVPR*, pp. 1–8, June 2005.

[69] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, Vol. 24, No. 6, pp. 381–395, June 1981.

[70] Streaming SIMD Extensions - Inverse of 4x4 Matrix. Technical report, Intel Corp., 1999.

[71] S. Martelli, R. Marzotto, A. Colombari, and V. Murino. FPGA-based robust ellipse estimation for circular road sign detection. In *Proc. IEEE 6th IEEE Workshop on Embedded Computer Vision*, pp. 53–60. IEEE, June 2010.

[72] Zhaofeng He, Tieniu Tan, Zhenan Sun, and Xianchao Qiu. Toward accurate and fast iris segmentation for iris biometrics. Vol. 31, No. 9, pp. 1670–1684, September 2009.

[73] Edward Rosten and Tom Drummond. Fusing points and lines for high performance tracking. In *IEEE Int. Conf. Computer Vision*, Vol. 2, pp. 1508–1511, October 2005.