論文 / 著書情報
Article / Book Information

題目(和文)	革新的なFPGAアクセラレータのための効率的な開発基盤
Title(English)	Efficient Development Infrastructure for Innovative FPGA Accelerators
著者(和文)	小林諒平
Author(English)	Ryohei Kobayashi
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第10238号, 授与年月日:2016年3月26日, 学位の種別:課程博士, 審査員:吉瀬 謙二,横田 治夫,宮崎 純,金子 晴彦,渡部 卓雄
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第10238号, Conferred date:2016/3/26, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

TOKYO TECH Pursuing Excellence
TOKYO INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Efficient Development Infrastructure for Innovative FPGA Accelerators

(革新的な FPGA アクセラレータのための効率的な開発基盤)

A dissertation submitted in partial fulfillment of
the requirements for the degree of Doctor of Engineering

Ryohei Kobayashi

March 2016

Graduate School of Information Science and Engineering
Department of Computer Science
Tokyo Institute of Technology

Abstract

Computer systems have been continuously improved throughout the years, and they tend to employ fabrics such as FPGAs and GPUs to accelerate some computing tasks that normally run on general-purpose CPUs. FPGA-based accelerators can achieve higher performance and better power efficiency than implementations on CPUs and GPUs because designers can implement circuits that realize application-specific pipelined hardware and data supply systems. This thesis presents a novel infrastructure supporting efficient development of FPGA-based accelerators. This shows how to build high-performance accelerators targeting several fundamental applications, and enables high-speed RTL simulation to verify that a designed accelerator works as intended.

The first contribution of this work is to propose a high performance FPGA-based accelerator for 2D stencil computation. In the last several decades, stencil computation has been accelerated by using multicore microprocessors and GPUs. However, sustained performance is limited due to memory bandwidth restriction, and also because the computation kernel has small arithmetic intensity. To address this problem, I propose a high performance architecture for 2D stencil computation employing multiple small FPGAs. In this architecture, the data set is divided into multiple blocks and each block is assigned to each FPGA, which means that the data set is stored in FPGA internal memory instead of in an external DRAM. This also means that the according to this architecture, the number of connected FPGAs scales with the size of the data set. The proposed stencil computation hardware was implemented with HDL, and I confirmed that the developed hardware accurately worked. The evaluation result shows that the proposed accelerator achieves even better power efficiency than a typical GPU.

The second contribution of this work is to propose an FPGA-based sorting accelerator, which combines the sorting network and the merge sorter tree. The proposed sorting hardware is customizable by means of tuning design parameters. I also provide an analytical model that accurately estimates the sorting performance depending on the hardware configuration. In other words, designers can estimate sorting accelerator performance in advance and can implement the best one that fulfills cost and performance constraints. I also propose a data compression mechanism for the sorting accelerator to mitigate memory bandwidth limitation. Similar to the stencil-computation accelerator, I developed the proposed sorting accelerator with HDL, and confirmed that the developed hardware actually achieved the estimated performance and higher performance than a typical desktop computer. In order to allow every

designer to easily and freely use this accelerator, the RTL source code is released as an open-source hardware.

Finally I summarized important points for the efficient development infrastructure of FPGA-based accelerators according to the two previous contributions. For development of FPGA accelerators, designers implement logic circuits with HDL and verify the circuit behavior. However, designing large-scale circuits leads to long RTL simulation times, which means that traditional RTL simulators cannot finish the circuit behavior verification within a realistic time frame. To address this problem, I propose a high-speed RTL simulator employing two prior studies. I evaluated it in terms of the RTL simulation speed by using the designs of the two proposed accelerators, and confirmed that it could do the RTL simulation much faster than a commercial one. Also, I discussed that the findings obtained from the development of the two FPGA-based accelerators are useful in other hardware platforms and computation kernels.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Computer systems are essential components to sustain the information age, and are nowadays used for lots of fields from scientific technologies to those we use in daily life, like weather forecast, drug development, financial engineering, artificial intelligence, automotive, smart phones, gaming machines, etc. Human beings have received these benefits that are thanks to performance improvements of computer systems. To make the future of the world more bright and prosperous, it is the key to research how to build innovative computer systems and how to apply them to practical matters.

To achieve high performance computer systems, one of the key components is a microprocessor, which is also called Central Processing Unit (CPU). In the past, a microprocessor consists of a single-core, and the performance improvements of microprocessors had relied on high clock rates and hardware features exploiting inherent instruction-level parallelism (ILP). However the performance improvements reached the limit in the late of 2000s. This is mainly because the power consumption and calorific value are not trivial. Besides it has been challenging to improve ILP by complex hardware components and this approach can cause more power consumption. Therefore, instead of a single-core processor, multi-core and many-core processors have become mainstream, accelerating applications by parallel processing like exploiting thread-level parallelism. The number of cores has been increased year after year due to improvements of semiconductor integration technologies depending on Moore's Law [1], and many-core processors with up to 256 cores begin to appear [2, 3, 4].

Not only CPUs but also other components like memory systems have also been advanced. In particular, graphics performance has been significantly improved thanks to technology evolution of Graphics Processing Unit (GPU). Although GPUs are originally specialized in image processing, in recent years they have been used for the high performance computing, known as General-Purpose computing on GPU (GPGPU). This is because GPU has even more parallel processing elements than CPU, and the high-performance computing kernels frequently have inherent data-level parallelism that GPUs can efficiently exploit. Their large parallel computing capability enables powerful scientific simulations

like linear algebra [5]. In modern supercomputers, GPUs have been seriously employed such as in TSUBAME2.5 [6], HA-PACS [7], etc.

CPU and GPU computation capabilities have been improved year by year, but their performance cannot be 100% delivered depending on application characteristics. For instance, authors in [8] can only exploit 21.8% of peak performance of two Intel Xeon E5220 CPUs for stencil computation used in fluid dynamics. Phillips et al. [9] propose a stencil computation approach using NVIDIA TESLA C1060 GPU, and the sustained performance of a single GPU is 65.6% of the peak performance, and worse, a GPU cluster composed of 16 GPUs can work at only 42.2% of the cluster peak performance. These low performance efficiencies are due to multicores and GPUs architectural aspects including memory bandwidth limitations. Besides CPUs and GPUs are prone to face power efficiency problems [10, 11] in not only stencil computation, and it is necessary to tackle these problems in order to realize computer systems of the future.

To address them, Field-Programmable Gate Array (FPGA) based accelerators have been attractive in recent years. FPGA is a programmable LSI on which designers can implement any desired digital circuit, and FPGA-based accelerators can achieve higher performance and better power efficiency than implementations on CPUs and GPUs because designers can implement circuits that realize application-specific pipelined hardware and data supply systems. For instance, in [12], an FPGA-based system to accelerate data center tasks is proposed. This system improves PageRank throughput of "Bing" as much as 195%. [13] presents a dedicated hardware to accelerate stencil computation that is used in several fields, such as fluid calculation, weather calculation, molecular simulation, etc. The proposed hardware is implemented by using multiple FPGA boards. The accelerator obtains higher computational performance at up to 13.7x compared with Intel Core i7-3930K with six cores operating at 3.2GHz, and the power efficiency is about 7x better than [10]. In June 2015 Intel Corporation announced the acquisition of Altera Corporation that is one of the biggest FPGA vendors in the world. Intel aims to accelerate database operations using FPGA integrated into Xeon processors [14], and will ship its first Xeon server chip with a programmable FPGA from Altera in the first quarter 2016. A lot of companies and research institutes have given considerable attention to FPGAs, and from now on the hybrid computing model based on CPUs, GPUs and FPGAs seriously begins [15].

To exploit the remarkable potentials of FPGA-based accelerators, it is truly necessary to consider how to build them. In other words, designers have to pay attention to an appropriate FPGA device, hardware design, and implementation depending on application characteristics. Another obstacle is to need long simulation times for circuit behavior verification. For development of FPGA accelerators, designers usually implement logic circuits with Hardware Description Language (HDL) and verify that the circuit behavior is as intended. However, designing large-scale circuits leads to long Register Transfer Level (RTL) simulation times, which means that traditional RTL simulators cannot finish the circuit

behavior verification within a realistic time frame. Besides, FPGAs have become larger and larger due to transistor scaling and stacking, and it enables implementation of larger hardware on FPGAs. However the simulation time is also larger, and that is why high-speed simulation environments are required.

This thesis presents a novel infrastructure supporting efficient development of FPGA-based accelerators. The proposed infrastructure shows how to build high-performance FPGA-based accelerators targeting fundamental applications and enables high-speed RTL simulation to verify whether or not a designed accelerator works as intended. The evaluation results show that designed FPGA-based accelerators outperform corresponding implementation on CPUs and GPUs in terms of both performance and power efficiency.

1.2 Contribution

The contributions of this thesis are as follows:

- I propose a high performance FPGA-based accelerator for 2D stencil computation employing multiple small FPGAs.
  - In this architecture, the data set is divided into multiple blocks and each block is assigned to each FPGA, which means that the data set is stored in FPGA internal memory instead of in an external DRAM. This also means that the according to this architecture, the number of connected FPGAs scales with the size of the data set. To realize the proposed accelerator, I developed a computation order optimization mechanism considering location information of each FPGA, a deeply pipelined stream computation unit, and a synchronization mechanism to absorb clock variations between FPGAs. The proposed stencil computation hardware was implemented with Hardware Description Language (HDL), and I confirmed that the developed hardware accurately worked. The evaluation result shows that the proposed accelerator achieves even better power efficiency than a typical GPU.
- I propose an FPGA-based sorting accelerator that combines the sorting network and the merge sorter tree.
  - The proposed sorting hardware is customizable by means of tuning design parameters. I also provide an analytical model that accurately estimates the sorting performance depending on the hardware configuration. In other words, designers can estimate sorting accelerator performance in advance and can implement the best one that fulfills cost and performance constraints. Similar to stencil-computation accelerator, I developed the proposed sorting accelerator with HDL, and confirmed that the developed hardware actually achieved the estimated performance and higher performance than a typical desktop computer. Be-

sides, I propose a data compression mechanism for the sorting accelerator to mitigate memory bandwidth limitation, and the evaluation results show that the sorting accelerator with the mechanism achieves better performance than without it. In order to allow every designer to easily and freely use this accelerator, the RTL source code is released as an open-source hardware.

- I propose a novel infrastructure to show how to build high-performance FPGA-based accelerators targeting fundamental applications and to enable high-speed RTL simulation to verify that developed hardware works as intended.
  - Based on the two previous contributions, I summarized important points for the efficient development infrastructure of FPGA-based accelerators. To shorten long RTL simulation times required to verify a designed circuit behavior, I propose a high-speed RTL simulator. I evaluated it in terms of the RTL simulation speed by using the designs of the two proposed accelerators, and confirmed that it could do the RTL simulation much faster than a commercial one. Also, I discussed that the findings obtained from the development of the two FPGA-based accelerators are beneficial in other hardware platforms and computation kernels.

1.3 Outline of This Thesis

This thesis consists of six chapters and is organized as follows.

In Chapter 2, I describe the fundamental knowledge of the FPGA-based accelerator and the background of my work. At first I briefly introduce the FPGA history, architecture and how to develop desired hardware on FPGAs and then discuss the point of interest for developing FPGA accelerators and the pros and cons of previously proposed development frameworks. Besides, I look into important computation kernels and explain that stencil computation and sorting are appropriate ones to be accelerated.

In Chapter 3, I describe the first main contribution of this work, which is to propose a high performance FPGA-based accelerator for 2D stencil computation employing multiple small FPGAs. In this architecture, the number of connected FPGAs scales with the size of the data set. I detail the design and implementation, and the evaluation results show that the proposed accelerator achieves even better power efficiency than a typical GPU.

In Chapter 4, I propose an FPGA-based sorting accelerator that combines the sorting network and the merge sorter tree, which is the second main contribution of this work. The proposed sorting hardware is customizable by means of tuning design parameters and I also provide an analytical model that accurately estimates the sorting performance depending on the hardware configuration. I detail the design

and implementation, and the evaluation result shows that the developed hardware actually achieves the estimated performance and higher performance than a typical desktop computer. Additionally, I propose a data compression mechanism for the sorting accelerator to mitigate memory bandwidth limitation.

In Chapter 5, I describe a novel infrastructure to show how to build high-performance FPGA-based accelerators targeting fundamental applications and to enable high-speed RTL simulation to verify that developed hardware works as intended. Based on the two previous contributions, I list important points for efficient development infrastructure for FPGA-based accelerators. To realize the high-speed RTL simulation, I employ two previously proposed tools; these are Pyverilog and ArchHDL. I briefly introduce the two tools and describe how to use them in the Proposed RTL simulator. The evaluation results show that it is possible to simulate faster the proposed stencil-computation and sorting accelerators behavior compared with a commercial simulator. Also, I discuss that the findings obtained from the development of the two FPGA-based accelerators are useful in other hardware platforms and computation kernels.

Finally in Chapter 6, I conclude this thesis with the discussion of open-research areas.

Chapter 2

FPGA-based Accelerators

This chapter provides the fundamental knowledge for the FPGA-based accelerator. First I briefly introduce the FPGA history, architecture and how to develop desired hardware on FPGAs and then discuss points of interest for developing FPGA-based accelerators. I also present development frameworks for FPGA-based accelerators previously proposed around the world, along with their pros and cons. Finally, I look into important computation kernels and explain why stencil computation and sorting are appropriate ones to be accelerated, and summarize the motivation of this work.

2.1 FPGA: Field Programmable Gate Array

The FPGA is a semiconductor device on which designers can program/reprogram any digital circuit. In 1985, Xilinx invented the first commercially viable FPGA (the XC2064), and engineers paid attention to it. This is because it is attractive to freely reprogram digital circuits since it was commonly accepted that hardware changes are difficult at that time. However these initial FPGAs were small and costly, and were used for only research prototyping and educational system rather than for commercial products. After several decades, FPGA capabilities have been significantly improved thanks to improvements of semiconductor integration technologies and research outcomes of FPGA architecture, and many companies and research institutes have begun to examine the FPGA availability. Given the FPGA characteristic features, FPGAs are mainly used in the three following cases

1. as a platform for design and verification of Application Specific Integrated Circuits (ASICs) like CPUs,
2. as a hardware component dealing with systems having frequent design changes like image processing systems, network servers, etc.,
3. as a hardware accelerator in order to reduce power consumption while increasing performance.

As mentioned before, I focus on and describe the third case in this thesis.

Table 2.1 Programming technology characteristics

Programming Technology	Volatile	Reprogrammable	Operating Speed
SRAM-based FPGA	Yes	Yes	Fast
Antifuse-based FPGA	No	No	Fast
Flash-based FPGA	No	Yes	Slow

2.1.1 FPGA Architecture

FPGAs are composed of programmable logic blocks and interconnection, which means that designers can electrically rewrite logic block contents and their interconnection configuration. Table 2.1 shows programming technologies for FPGAs [16].

SRAM programming technology is based on static memory cells storing a stream of configuration bits for logic block contents and their interconnection. The advantage of this technology is to strongly receive benefits of the semiconductor processes refinement, since this technology can be implemented under the standard CMOS process. Designers can reprogram desired digital circuits a number of times, but once the device is powered down, the configuration data is lost because SRAM is volatile memory. Therefore, this technology needs to reload the configuration data from external flash or EEPROM devices.

Antifuse programming technology is based on structures that exhibit very high-resistance under normal circumstances. By applying a voltage, these high resistances can be 'blown' to create a low resistance link, and this technology can form digital circuits based on the blown connection. Unlike SRAM programming technology, the configuration data is never lost and the implemented circuit is small and works at high speeds and low energy because this technology does not employ SRAM. However, the largest disadvantage is that antifuse-based FPGAs are one-time programmable devices.

Flash programming technology is based not on static memory cells but flash memory cells storing a stream of configuration bits for logic block contents and their interconnection. Unlike SRAM and antifuse programming technologies, this technology does not lose the configuration data and can reprogram digital circuits any times. However, the manufacturing process is complicated and the operating speed is slow since the read speed from the memory is slower than SRAM.

Given these advantages and disadvantages, SRAM-based FPGAs have nowadays become mainstream. Figure 2.1 shows the SRAM-based FPGA architecture. Most of the today's FPGAs consist of arrays of programmable Logic Blocks (LBs) and hard blocks with fixed functionality, like memory and Digital Signal Processor (DSP), which can work faster and can offer more compact implementations of hardware functions than using LBs. The arrays are surrounded by I/O blocks, which connect the FPGA chip to the outside world. These programmable logic blocks, hard blocks, and I/O blocks are all in-

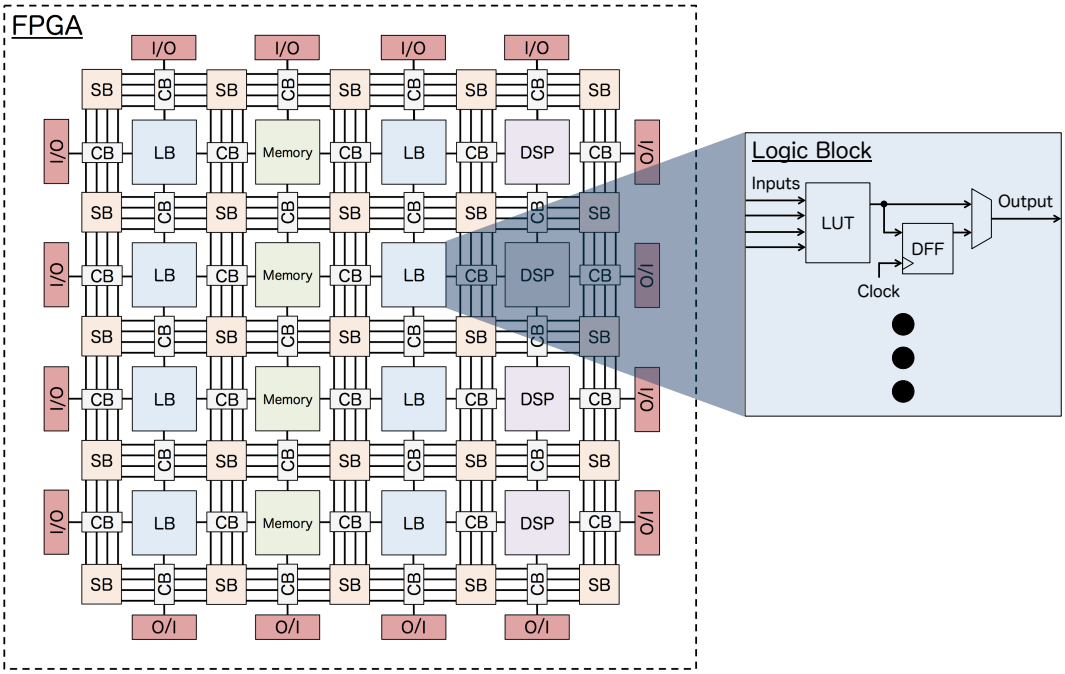


Figure 2.1 SRAM-based FPGA architecture

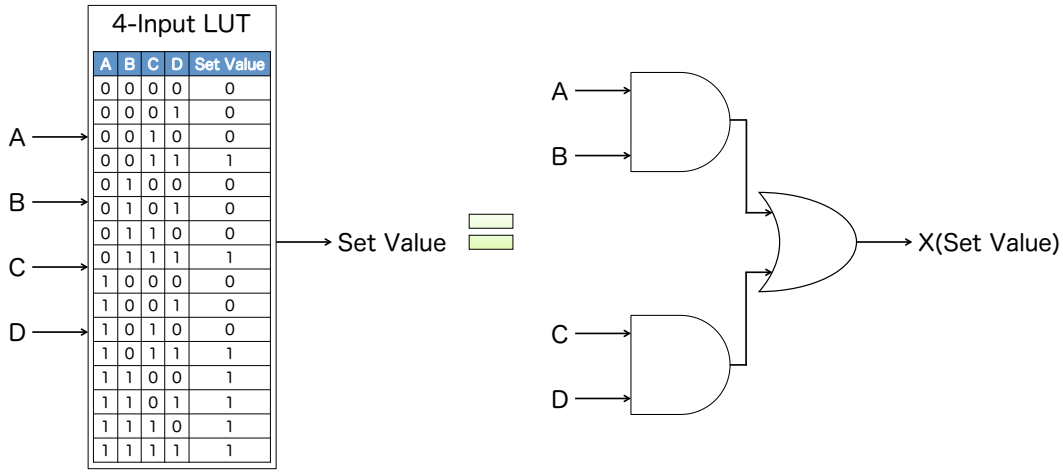


Figure 2.2 4-input LUT realizing X=(A&B)|(C&D)

terconnected using programmable routing fabrics that are Switch Boxes (SBs) and Connection Blocks (CBs). These routing fabrics are composed of 1-bit SRAM cells, and by storing the proper bits in these fabrics, SBs and CBs determine which wire should be connected to another wire and LB/hard block. This two-dimensional mesh interconnected routing architecture is called island-style [17, 18]. Another routing architecture, which is hierarchical [19], was proposed but designers employ island-style for most of the today's FPGAs development because hierarchical usually incurs a significant delay penalty [16].

The programmable logic block is basically composed of Look-Up Tables (LUTs) and D-type Flip Flops (DFF) [18]. LUTs are also composed of 1-bit SRAM cells, and by storing the proper bits in the SRAM cells, various combinational logic functions could be implemented. Figure 2.2 shows one of the examples. The LUT in this figure consists of 16 1-bit SRAM cells and returns a 1-bit value set in an SRAM cell according to the four inputs, which is called 4-input LUT. It realizes a boolean function, which is equal to X=(A&B)|(C&D) where X represents a 1-bit value set in an SRAM cell. To implement sequential circuits like Finite State Machines (FSMs), a DFF is connected to the output of LUTs. The multiplexer is also based on 1-bit SRAM cells, and determines which signal should be emitted depending on the set values.

SRAM-based FPGAs can realize a desired logic circuit by storing proper bits in 1-bit SRAM cells in LBs, SBs, and CBs. To implement a high performance FPGA-based accelerator, it is important to efficiently utilize and connect hard blocks like memory and DSP in addition to LBs.

2.1.2 FPGA Design Flow

Figure 2.3 shows a typical FPGA design flow. At first, designers develop a Register Transfer Level (RTL) design after the hardware specifications are determined. To develop the RTL design, Hardware Description Languages (HDLs) like VHDL and Verilog HDL are mostly used. After that, the RTL design can be verified by using commercial [20, 21, 22, 23, 24] or open-source [25, 26, 27, 28] RTL simulators. This is a truly important step to confirm that the design behavior works as intended.

The next step is to create the FPGA circuit image (bitstream) file the RTL design. This step requires a unique Electronic Design Automation (EDA) tool depending on the targeted FPGA device. For instance using Xilinx, Altera, or Lattice FPGAs require ISE/Vivado Design Suite [29, 30], Quartus Prime (Quartus II) [31], or Lattice Diamond [32] respectively. Although this step needs these FPGA vendor's own development tools, it consists of four processes that are Synthesis, Technology Mapping, Place and Route, and Bitstream Generation shown in the dotted line region in Figure 2.3.

In Synthesis, the developed RTL design is translated into the gate-level netlist shown in Figure 2.4 (a), and designers can simulate the logic circuit behavior in more detail (Gate Level Simulation). The next process, Technology Mapping, fits the synthesized design into targeted FPGA primitives. In Figure 2.4 (b), the design is mapped into the two primitives that is a single LUT and flip flop. This is because the combinational logic composed of two AND gates and an OR gate can be translated into an LUT as described in Figure 2.2. After this process, these mapped FPGA primitives are placed and routed to satisfy the required timing performance, which is illustrated in Figure 2.4 (c). Using the timing information generated by Place and Route, designers can more accurately verify the design behavior in Timing Simulation, compared with RTL and Gate Level Simulation. Finally, the placed and routed design information is used to generate the bitstream for the targeted FPGA configuration.

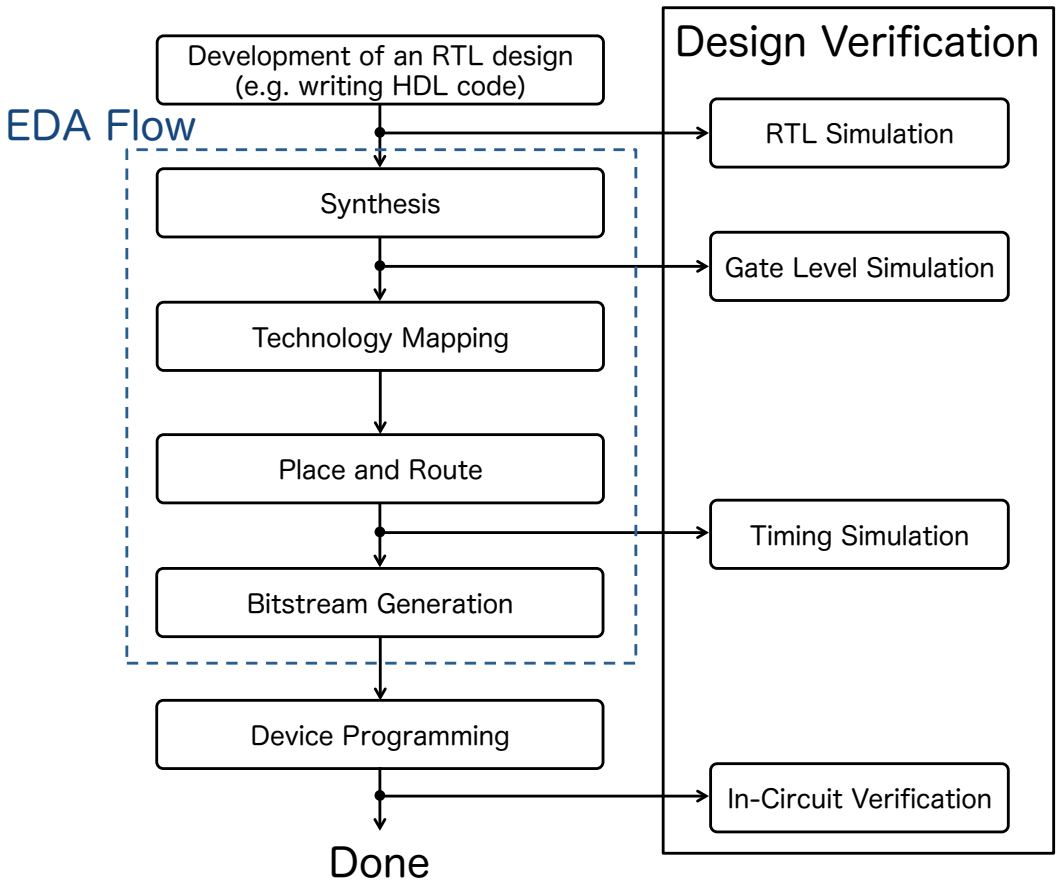


Figure 2.3 FPGA design flow overview

After the EDA flow, designers can program the developed design by downloading the generated bitstream to the targeted FPGA. To confirm that the implemented design on the FPGA correctly works as intended (In-Circuit Verification), designers can use logic analyzers provided by FPGA vendors.

2.2 Points of Interest for developing FPGA-based Accelerators

In this section, I explain the points of interest for developing FPGA-based accelerators. Figure 2.5 shows peak floating-point performance of the three devices for single precision and double precision. This figure is cited from [15]. According to Figure 2.5 (a), GPUs are always GPUs have always the best performance for single precision. The most interesting points of Figure 2.5 (a) is that in 2011 both FPGAs and CPUs have increased their relative performance, but in 2013 relative performance of FPGAs is degraded while CPUs keep improving. This is because many-core CPUs like Intel Xeon Phi began to appear. In Figure 2.5 (b), GPUs have dominated the peak performances, except for the first years and the CPUs started to dominate FPGAs from 2011 for double precision. In 2013 the peak

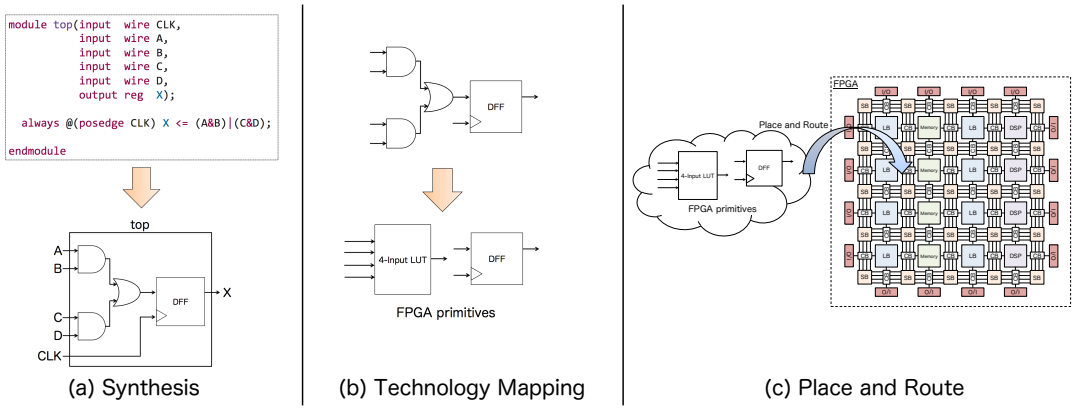


Figure 2.4 This figure depicts how to implement a logic circuit on an FPGA using an EDA tool, which is composed of (a) Synthesis, (b) Technology Mapping, and (c) Place and Route.

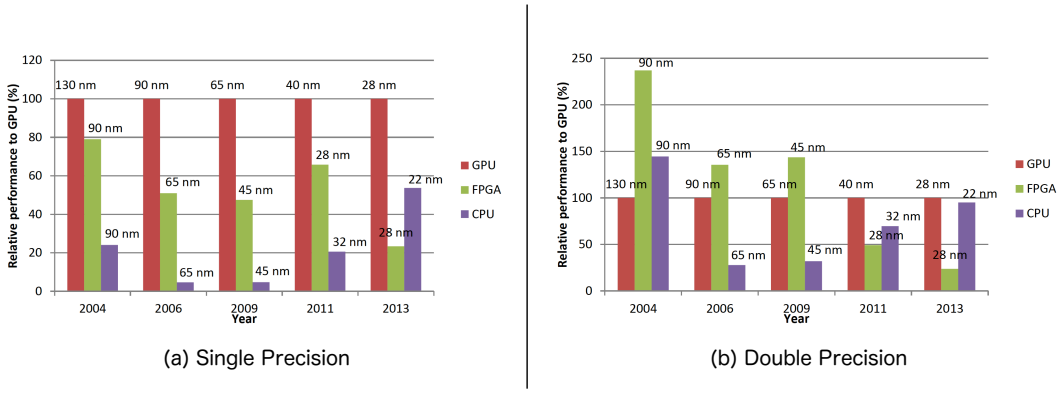


Figure 2.5 Peak performance of FPGA and CPU relative to GPU for (a) single precision and (b) double precision cited from [15]

performances of both GPUs and many-core CPUs came close with about 5%.

Peak performance just gives a theoretical comparison. The sustained performance, which is performance achieved when running a particular application, can vary with each particular application and the power efficiency is similar to it. This is because these computing devices have different architectural aspects including the memory architecture with its external memory bandwidth and the communication network, which are more appropriate for specific applications.

For instance, [33] compares sustained performance between NVIDIA Tesla C2090 GPU, Xilinx ZYNQ XC7Z020 FPGA, and Intel Xeon processor in different encryption algorithms. The authors report that the GPU outperforms the CPU by 13x and the FPGA provides a throughput speed-up of 6~9 over the GPU for 8KB and 16KB plaintext blocks. This is because the FPGA has enough capacities to do streaming computation. For larger plaintext sizes, the GPU and FPGA provide similar throughput. [34] compares the implementation of a quasi-Monte Carlo simulation between Xilinx Virtex-4 FPGA,

NVIDIA 8800GTX GPU and a 2.8 MHz Intel Xeon CPU. According to the results, the FPGA outperforms the CPU-based implementation by 2 orders of magnitude. Also, the FPGA achieves around 3x speed-up compared to equivalent the GPU-based implementation. The Power consumption of the FPGA is 336x more energy efficient than the CPU, and 16x more energy efficient than the GPU.

However, depending on application algorithms, hardware designs, and implementation, FPGAs are not always effective. Authors in [5] compare implementations of Black-Scholes between Intel i7-960 CPU, NVIDIA GTX480 GPU and Xilinx Virtex-6 FPGA and conclude that the GPU is about 1.4x faster than the FPGA and 22x faster than the CPU. In power efficiency, the GPU is about 1.3x more power efficient than the FPGA and almost 40x compared to the CPU. The authors also compare implementation dense matrix multiplication between the computing devices and conclude that the GPU with a sustained performance of 541 GFlop/s is about 3x faster than the FPGA and 6x faster than the CPU. Considering power efficiency, both FPGA and GPU are 3x better than that of the CPU. Therefore, it is necessary to consider an appropriate FPGA device, hardware design, and implementation depending on application characteristics, such as a sequential algorithm, an iterative algorithm, data-parallel algorithm or memory intensive algorithm, etc.

2.3 Development Frameworks for FPGA-based Accelerators

As mentioned before, in order to develop the RTL design, HDLs like VHDL and Verilog HDL are mostly used. However unfortunately, these HDLs hardly have the high-level and abstraction facilities commonly found in modern mainstream languages. Because of this, the FPGA development can be tedious, inefficient and error-prone for non-expert designers, what is worse, it might also affect expert FPGA designers from the productivity point of view [35, 36]. To address this problem, many development approaches have been proposed by many companies and research institutes in the world.

2.3.1 High-Level Synthesis

High-Level Synthesis (HLS) languages and tools are emerging as the most promising technique to provide higher the programmability for FPGA-based hardware development and to make FPGAs more accessible to software developers. HLS techniques translate software languages like C language into RTL designs [37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49].

These days, there are a lot of available commercial HLS languages and tools. Xilinx Vivado HLS [37] can accelerate IP cores creation by enabling C, C++ and SystemC specifications to be directly targeted into Xilinx FPGAs without the need to manually create RTL. This tool has been a commercial product so far, but becomes free of charge and gets included in all Vivado software editions from Vivado Design Suite 2015.4 [50]. Altera OpenCL [38] enables the easy implementation of applications onto FPGAs by abstracting away the complexities of FPGA design, allowing software programmers to write

hardware-accelerated kernel functions in OpenCL C. Not only FPGA vendors but also other companies provide HLS languages and tools. CyberWorkBench [39] is C-based integrated environment for System LSI design developed by NEC Corporation. Impulse C [40] is also a C-based HLS tool developed by Impulse Accelerated Technologies, which is specialized in dataflow-oriented streaming applications. Calypto Catapult C [41] can accept C++ and SystemC and can generate RTL code targeted to FPGAs and ASICs. Lime [42] is a Java-based language that is designed to be executable across a broad range of architectures from FPGAs to conventional CPUs. This language is developed by IBM Research and is used in the Liquid Metal project as a single unified programming language.

Several open-source HLS frameworks also proposed. LegUp [43, 44] is an open source HLS tool being developed at the University of Toronto. This tool accepts a standard C program as input and automatically compiles the program to a hybrid architecture containing an FPGA-based MIPS soft processor and custom hardware accelerators that communicate through a standard bus interface. The Riverside Optimizing Compiler for Configurable Computing (ROCCC) [45] a C to VHDL compiler, which is specifically designed to automatically generate FPGA-based accelerators. Unlike LegUp, this tool aims to compile only critical regions of an application. Synthesijer [46] is a HLS tool that generates VHDL and Verilog HDL code from Java code. This tool does not need any hardware-aware description manner, but some syntax like dynamic memory and recursive functions are unsupported. CoRAM, CoRAM++, PyCoRAM [47, 48, 49] are slightly different from other HLS tools. These tools virtualize the communication and control infrastructure that interfaces the in-fabric hardware acceleration kernels with external memory (DRAM) to improve portability of hardware accelerators. Designers implement the hardware kernels in any hardware design methodology and describe control threads for memory access pattern in software. The designed control threads are translated into control circuits of memory operations. In [47, 48], the control threads are written in C-based language and in [49], Python is used to do this.

2.3.2 System Development Using Domain Specific Language

In addition to HLS techniques, lots of Domain Specific Languages (DSLs) are proposed and published year after year. The most difference from HLS techniques is that designers develop hardware, more or less considering the hardware details like clock timing and interconnection than HLS techniques.

Hardware design DSLs using software language like C++, Java, Python, Haskell, etc. have been proposed before. ArchHDL [51] is a C++11-based library for RTL modeling and simulation. MyHDL [52] is an Python-based DSL that can translate a source code in MyHDL into Verilog or VHDL code. JHDL [53] is a Java-based DSL for reconfigurable systems that allows designers to express circuit organizations that dynamically change over time. Chisel [54] is a Scala-based DSL and hardware designs in Chisel can be translated into C++ simulators or Verilog RTL descriptions. [55] proposes

Veriloggen, which is a library for constructing a Verilog HDL source code in Python. Unlike MyHDL, this tool provides a lightweight abstraction of Verilog HDL AST, and designers can build up a hardware design written in Verilog HDL by using the AST abstraction and the entire functionality of Python. Lava [56] is a Haskell based DSL and has several distributions such as Chalmers Lava, Xilinx Lava, Kansas Lava, and York Lava. Similar to MyHDL and Veriloggen, PyMTL [57] is also a Python-based DSL but can do multi-level hardware modeling composed of functional level, cycle level, and register transfer level. Except for ArchHDL, these DSLs are already open-source.

Some application-specific DSLs have been proposed. Authors in [58] propose a hardware generation framework and system for linear transforms. This approach uses SPIRAL proposed in [59] to automatically produce software implementations of signal transforms, including automatic parallelization and vectorization. HDL Coder [60] generates portable and synthesizable HDL code from MATLAB functions and Simulink blocks. The generated HDL code can be used for FPGA programming or ASIC prototyping and design. Optimus [61] is an optimizing synthesis compiler for streaming applications, which compiles programs written in StreamIt [62] to either software or hardware implementations.

Some companies and research institutes have proposed hardware design DSLs supporting specific programming models. Bluespec SystemVerilog (BSV) [63] is an extended SystemVerilog developed by Massachusetts Institute of Technology, which enhances fine-grained parallel programming approach while abstracting hardware details. MaxCompiler [64] is developed by Maxeler Technologies in order to build high-performance dataflow hardware accelerator. This tool supports stream-oriented programming model using Java. FloPoCo [65] is developed by Institut National de Recherche en Informatique et en Automatique (INRIA), which automatically generates pipelined floating-point units. Using FloPoCo as a back-end, [66, 67] proposes a framework to generate high-throughput pipelined stream processors using their DSL. [66, 67] mentions that unlike MAXCompiler, this tool is designed to easily generate IP cores of stream processors that can be embedded into a common SoC platform on an FPGA.

2.3.3 Discussion

A lot of development frameworks for FPGA-based accelerator have been proposed, but the ease of programmability, performance, resource usage and efficiency can vary from one technology to another, and there is usually a tradeoff between these characteristics [35]. Also, it is not easy to choose the best development framework for implemented applications and this is mostly dependent on programmer experience. These frameworks can verify the designed hardware behavior at higher abstraction layer but it is necessary to verify the behavior at low abstraction layer like RTL or gate-level in order to accurately detect the hardware errata. Therefore development frameworks based on HLS tools and DSLs have been attractive, nevertheless it is common that HDL-based circuit design and the circuit

Table 2.2 Throughput computing kernel characteristics cited from [68]

Kernel	**Application**	**SIMD**	**Thread Level Parallelism**	**Characteristic**
SGEMM	Linear algebra	Regular	Across 2D tiles	Compute bound after tiling
Monte Carlo	Computational finance	Regular	Across paths	Compute bound
Convolution	Image analysis	Regular	Across pixels	Compute bound; BW bound for small filters
FFT	Signal processing	Regular	Across smaller FFTs	Compute bound or BW bound for small filters
SAXPY	Dot product	Regular	Across vector	BW bound for large vectors
LBM	Time migration	Regular	Across cells	BW bound
Constant solver	Rigid body physics	Gather/Scatter	Across constraints	Synchronization bound
SpMV	Sparse solver	Gather	Across non-zero	BW bound for typical large matrices
GJK	Collision detection	Gather/Scatter	Across objects	Compute bound
Sort	Database	Gather/Scatter	Across elements	Compute bound
Ray casting	Volume rendering	Gather	Across rays	4-8MB first level working set; over 500MB last level working set
Search	Database	Gather	Across queries	Compute bound for small tree, BW bound at bottom of tree for large tree
Histogram	Image analysis	Requires conflict detection	Across pixels	Reduction/synchronization bound

behavior verification are repeated as required for development of FPGA-based accelerators. However, designing large-scale circuits leads to long RTL simulation times, which means that traditional RTL simulators cannot finish the circuit behavior verification within a realistic time frame. In the future, FPGAs will be larger thanks to transistor scaling and stacking, leading to more RTL simulation times. Therefore high-speed simulation environments are required.

2.4 Throughput Computing Kernels

In this section, I look into important computation kernels and explain why stencil computation and sorting are appropriate ones to be accelerated.

Recent information technology advances in the past decade have led to an explosion in the amounts of data being generated such as digital documents, stock market data, personal records, electronic commerce sales data, news, etc. As digital data continues to grow rapidly, it is important to process the ever-growing data in a reasonable duration of time.

Authors in [68] mention that processing huge amount of data to distill and deliver appropriate content to users in a timely manner has made throughput computing an important aspect for emerging applications, and they rigorously analyzed CPU and GPU performance differences between a set of important throughput computing kernels. They also analyzed the computational and memory characteristics of four recently proposed benchmark suites and formulated the set of throughput computing kernels that capture these characteristics. Table 2.2 summarizes the throughput computing kernel characteristics. The authors classify these kernels according to (1) their computational and memory requirements, (2) regularity of memory accesses that determines the ease of exploiting data-level parallelism, which

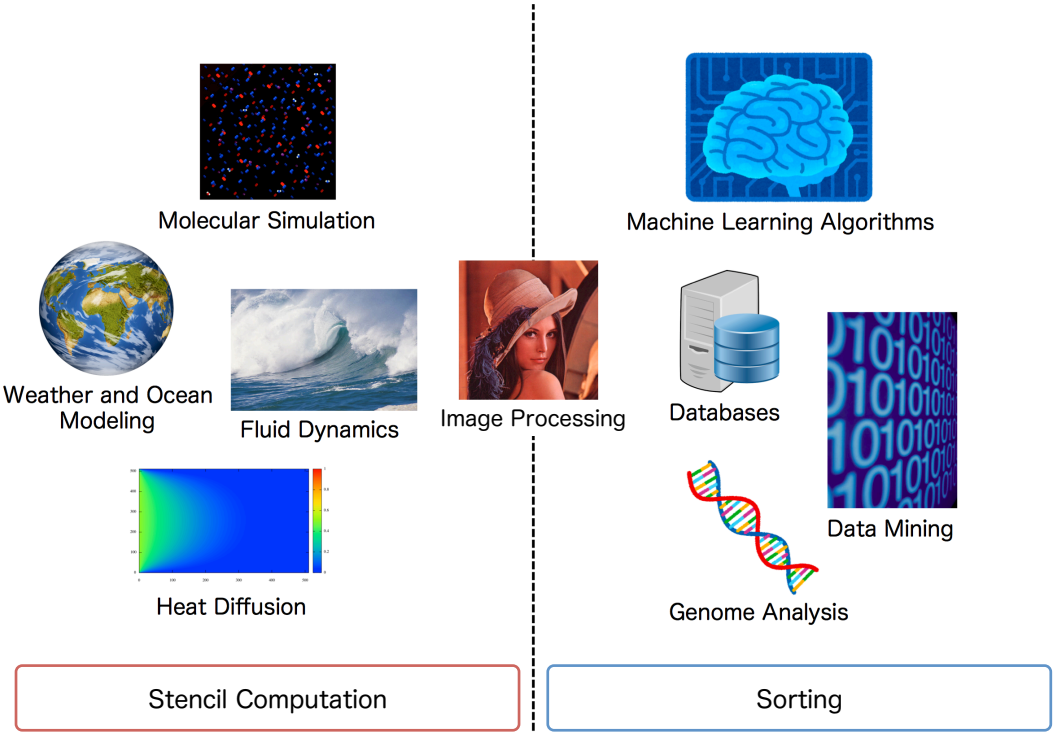


Figure 2.6 Stencil computation and sorting are fundamental operations for many applications.

means the use of Single Instruction Multiple Data (SIMD), and (3) the granularity of tasks, which determines the impact of synchronization. They mention that these characteristics provide insights into the architectural features that are required to achieve good performance.

In this thesis, I select Lattice Boltzmann method (LBM) and Sort as a kernel to be accelerated by FPGAs and discuss that the findings obtained from the two designed FPGA-based accelerators can be applied to the other kernels. As shown in Table 2.2, almost all of the kernels are either compute-bound or bandwidth bound. To demonstrate that FPGA-based acceleration approaches are valuable, it is necessary to clarify that the approaches can achieve better performance than CPUs and GPUs in both compute and bandwidth-bound kernels. Besides, the two kernels are applied to lots of applications shown in Figure 2.6. That is why it is valuable to try to accelerate them, and as a representative of bandwidth and compute-bound kernels, LBM and Sort are selected respectively.

LBM is a class of computational fluid dynamics based on stencil computation whose algorithm updates values associated with points on a multidimensional grid using weighted contributions from a subset of its neighbors in both time and space. Stencil computation plays a crucial role in a variety of different fields of application, ranging from partial differential equation solving, to computer simulation of particles' interaction, to image processing and computer vision [69]. Basically, stencil computation is bandwidth bound due to the algorithm characteristic, which requires to load several data from an

external memory to update a value in each time step.

As already known, sorting is used in a lot of fields, especially databases. [68] picks up radix sort and describes the trendy implementation for CPUs and GPUs. At the time, the best implementation on CPUs is to use SIMD instructions and implement a scatter-oriented rearrangement within cache. On GPUs, radix sort is implemented using the SIMD-friendly 1-bit split algorithm described in [70]. However, the split-based implementation executes even more instructions than a scalar sort, and therefore the overall efficiency of SIMD use relative to optimized scalar code is not high. The authors in [68] mention that totally radix sort has $O(n)$ bandwidth and compute requirements, where n is the number of elements to be sorted, but is usually compute bound because of the inefficiency of SIMD use.

2.5 Summary

In this chapter, I described the fundamental knowledge of the FPGA-based accelerator and the background of my work.

FPGA-based accelerators have been attractive as an alternative computing device in recent years. Although the peak performance is low compared with the other two devices, FPGA-based accelerators can achieve higher performance and better power efficiency than implementations on CPUs and GPUs because designers can implement circuits that realize application-specific pipelined hardware and data supply systems. However in order to receive the remarkable potentials, designers have to decide on an appropriate FPGA device, hardware design, and implementation depending on application characteristics, such as a sequential algorithm, an iterative algorithm, data-parallel algorithm or memory intensive algorithm, etc.

Moreover, many companies and research institutes in the world have proposed development frameworks for FPGA-based accelerator. However there is a tradeoff between the ease of programmability, performance, resource usage and efficiency. Besides their characteristics can vary among the frameworks and it is not easy to choose the best development framework for implemented applications. Therefore it is still common that designers implement logic circuits with HDL and verify the circuit behavior in order to develop FPGA-based accelerators. However, designing large-scale circuits leads to long RTL simulation times, which means that traditional RTL simulators cannot finish the circuit behavior verification within a realistic time frame. In the future, FPGAs will be larger thanks to transistor scaling and stacking, leading to more RTL simulation times. Therefore high-speed simulation environments are required. To address this problem, I propose a novel infrastructure enabling high-speed RTL simulation to verify whether or not the developed hardware works as intended.

Finally I discuss a set of important throughput computing kernels and explain that stencil computation and sorting are appropriate ones to be accelerated by FPGAs. In this thesis, I propose two high performance FPGA-based accelerators targeting them. In order to show that the acceleration methods

are promising, I evaluate the designed accelerators performance in the compute and bandwidth-bound kernel and discuss the applicability of the findings obtained from the development of the proposed accelerators against the other kernels.

Chapter 3

A Scalable Stencil-computation Accelerator by Employing Multiple Small FPGAs

3.1 Motivation

Stencil computation is one of the typical scientific computing kernels. It is applied to diverse areas such as earthquake simulation, digital signal processing and fluid calculation. In the last several decades, stencil computation has been accelerated by using multicore microprocessors and GPUs. However, sustained performance is limited due to memory bandwidth restriction, and also because the computation kernel has small arithmetic intensity. To address this problem, I propose a high performance architecture for 2D stencil computation employing multiple small FPGAs. In this architecture, the data set is divided into multiple blocks and each block is assigned to each FPGA, which means that the data set is stored in FPGA internal memory instead of in an external DRAM. This also means that the according to this architecture, the number of connected FPGAs scales with the size of the data set. I detail the design and implementation of the proposed FPGA-based accelerator, and evaluate it in terms of the sustained performance, scalability, and power efficiency.

The main contributions of this chapter are:

- to propose a high performance computation architecture for 2D stencil computation using multiple small FPGAs,
- to develop an FPGA-based scalable stencil-computation accelerator to realize my proposed architecture,
- to show the architecture usability on the 100-FPGA array system in terms of the sustained performance, scalability, and power efficiency.

```
1  float v0[N][N], v1[N][N];
2
3  for (k = 0; k < IterNum; k++) {
4    for (i = 1; i < N-1; i++) {
5      for (j = 1; j < N-1; j++) {
6        v1[i][j] = (C0 * v0[i-1][j]) + (C1 * v0[i][j-1]) + (C2 * v0[i1][j+1]) + (C3 * v0[i+1][j]);
7      }
8    }
9    for (i = 1; i < N-1; i++)
10     for (j = 1; j < N-1; j++) v0[i][j] = v1[i][j];
11 }
```

Figure 3.1 The pseudo code of kernel part for stencil computation

3.2 Stencil Computation

In the area of scientific computation a class of iterative kernels is frequently used, which uses some values of a time step and calculate a result associated with those values of the next time step. In the class, *stencil computation* [10] updates values according to some **fixed pattern**. Stencil computation is one of the approaches to calculate an approximate solution of partial differential equations, and is used in lots of areas like earthquake simulation, digital signal processing and fluid calculation.

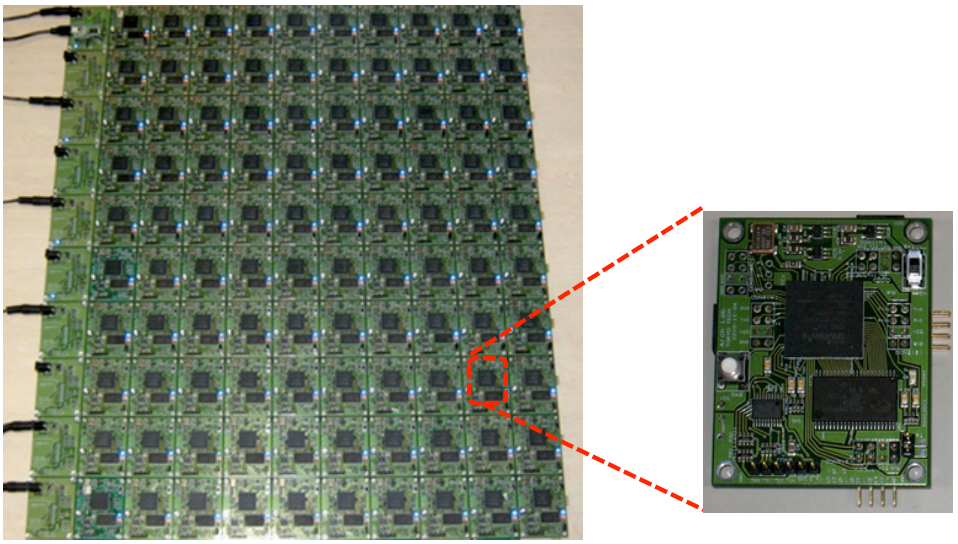
In this thesis I focus on the 2D Jacobi iteration kernel [71] and the formula is given by

$$v_{i,j}^{k+1} = c_0 v_{i-1,j}^{k} + c_1 v_{i,j-1}^{k} + c_2 v_{i,j+1}^{k} + c_3 v_{i+1,j}^{k} \tag{3.1}$$

where $v_{x,y}^{k}$ and $v_{x,y}^{k+1}$ are values of x, y at time step k and $k+1$ respectively, and c_0, c_1, c_2, and c_3 stand for weighting factor. In this thesis, the data type of the weighting factor and values are single precision floating-point. As shown in the formula, a value $v_{x,y}^{k+1}$ is calculated from the summation of the left, up, right and down values multiplied by each weighting factor. If all weighting factors are 0.25, updating values of the next time step stands for the arithmetic mean of the four values of the current time step.

Figure 3.1 shows the pseudo code of kernel part for stencil computation. k and (i, j) are a time step and value coordinate. The two vectors v0 and v1, which are declared at line 1, are used to store the two-dimensional $N \times N$ data set. A value at (i, j) is represented as v0[i][j] or v1[i][j]. At line 6 in Figure 3.1, v1[i][j] is calculated from the summation of v0[i-1][j], v0[i][j-1], v0[i][j+1], and v0[i+1][j] multiplied by each weighting factor. At line 9 and 10 in Figure 3.1, all values of v1 are copied into v0. In this thesis, the process at a time step k (line 4 $\sim$ 10) is called **Iteration**. *IterNum* at line 3 in Figure 3.1 is constant that stands for the number of executed Iterations.

As shown in Figure 3.1, the kernel part for stencil computation is simple, yet needs large computation time since the computation kernel has small arithmetic intensity [72] and the number of Iterations is large. Arithmetic intensity is defined as the number of floating-point operations (Flops) per data size (bytes) read from an external DRAM for cache misses. As described above, stencil computation requires multiple data accesses per unit operation. In a practical application, because the number of memory accesses is even larger than this example, microprocessors and GPUs cannot achieve high



(a) ScalableCore system with 100 units (b) ScalableCore unit

Figure 3.2 Photo of ScalableCore system with 100 units (a) and Scalable Core unit (b)

sustained performance due to memory bandwidth restriction. According to [72], straightforward implementation of stencil computation can achieve only 10% of the peak performance of AMD Opteron microprocessors because of its small arithmetic intensity of 0.5 Flop/Byte. Therefore, several research institutes have proposed FPGA-based accelerators for stencil computation [73, 74].

In this chapter, I propose a scalable stencil-computation accelerator by employing multiple small FPGAs [75]. To realize this proposal, I use ScalableCore system [76] that we have developed. I detail the design and implementation that efficiently executes 2D stencil computation, and evaluate the proposed accelerator in terms of the sustained performance and power efficiency.

3.3 Key Issues of FPGA Array System

In this section, I describe ScalableCore system that is a hardware platform for design and implementation of the proposed stencil-computation accelerator, and discuss a clock variation problem that is due to a different clock oscillator on each FPGA node.

3.3.1 ScalableCore System

Figure 3.2 (a) shows ScalableCore system using 100 FPGAs. ScalableCore system is an FPGA array system employing multiple small FPGAs. Figure 3.2 (b) shows an FPGA node called ScalableCore unit. The board size is 4.67cm $\times$ 6.0cm. The FPGA node works as a stand-alone FPGA board and has

Table 3.1 Worst value and standard deviation of measured clock variations.

Time [sec]	Worst Value [ppm]	Standard Deviation
20	20.47 (x=5, y=3)	4.73
40	20.47 (x=5, y=3)	4.68
80	20.47 (x=5, y=3)	4.73
160	20.59 (x=5, y=3)	4.77
320	20.66 (x=5, y=3)	4.79

an FPGA (Xilinx Spartan-6 XC6SLX16), 512KB SRAM[*1], 40MHz clock oscillator, and configuration ROM. Connection between the FPGA nodes is realized via the external I/O pins on the left, right, top and bottom side.

The leftmost nodes in Figure 3.2 (a) have a DC5V power supply and USB-UART IC chip. Application programs are loaded from a Linux host PC and are transferred to ScalableCore system via USB-UART IC chip on the upper left board. After that, ScalableCore system executes the programs and displays the execution result on the host PC.

ScalableCore system is originally developed for high speed many-core processor simulation. The power consumption of an FPGA is about 1 Watt [76]. Based on the power consumption, I estimate the power efficiency of the proposed stencil-computation accelerator and the estimated power efficiency is 1.7x better than a previous study [77] that uses several large FPGAs to accelerate stencil computation. Because of the examination result, I decide to use ScalableCore system as a hardware platform for the proposed stencil-computation accelerator.

3.3.2 Preliminary Evaluation regarding Clock Variation Problem

As shown in Figure 3.1, stencil computation uses the four adjacent values to calculate the center value of the next time step. Therefore data transfer between FPGA nodes is necessary if the divided tasks are assigned to each FPGA. Besides, the data transfer has to be executed at an appropriate timing because updating values at an Iteration needs the neighborhood values calculated at the previous Iteration.

On the other hand, each FPGA constituting ScalableCore system has an individual clock oscillator that is 40MHz CSX-750PB. The frequency stability of the clock oscillator is ±50ppm. The frequency stability ±50 means that it is guaranteed that the number of deviated cycles per 1 million cycles is ±50 based on the perfect clock. However even if ±50ppm, it may generate a gap of 120,000 cycles per minute because 40MHz CSX-750PB may generate a gap of 2,000 cycles per second. This gap is not trivial and it is necessary to design a robust system that can absorb the clock variation.

In this section, I quantitatively evaluate clock variation of each FPGA node. The original Scal-

[*1] The proposed stencil-computation accelerator does not use it.

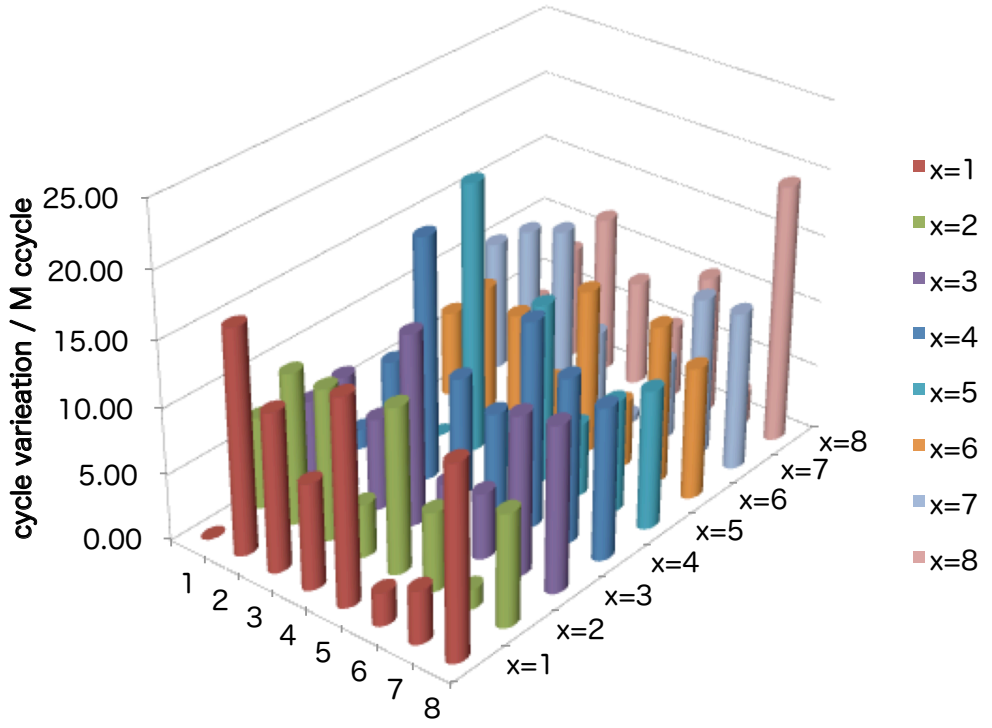


Figure 3.3 Clock variations by measuring 20 seconds

ableCore system emulates many-core processor behavior and the emulated many-core processor works while synchronized under Virtual Cycle proposed in [76]. Using this mechanism, I implement a C application to measure the frequency stability. In this preliminary evaluation, the C program runs on 8×8 ScalableCore system and I evaluate the clock variation of each FPGA node depending on the measurement time.

Figure 3.3 shows clock variation of each FPGA node by measuring 20 seconds. The points on x-axis and y-axis stand for each FPGA coordinate. The z-axis represents the absolute value of the number of deviated cycles per 1 million cycles. This result shows that the all deviated cycles are within 50 cycles according to the clock oscillator specification. The worst deviated cycle is 20.47, which is generated from the FPGA node located at (x=5, y=3).

Table 3.1 shows the worst value and standard deviation of measured clock variations. Time, Worst Value, and Standard Deviation in this table stand for the measurement time, the worst deviated cycle with the FPGA node coordinate, and standard deviation of all clock variations in 8×8 ScalableCore system. The worst deviated cycle and standard deviation hardly change depending on the measurement time. It means that the clock variation is larger at a constant rate depending on the operating time.

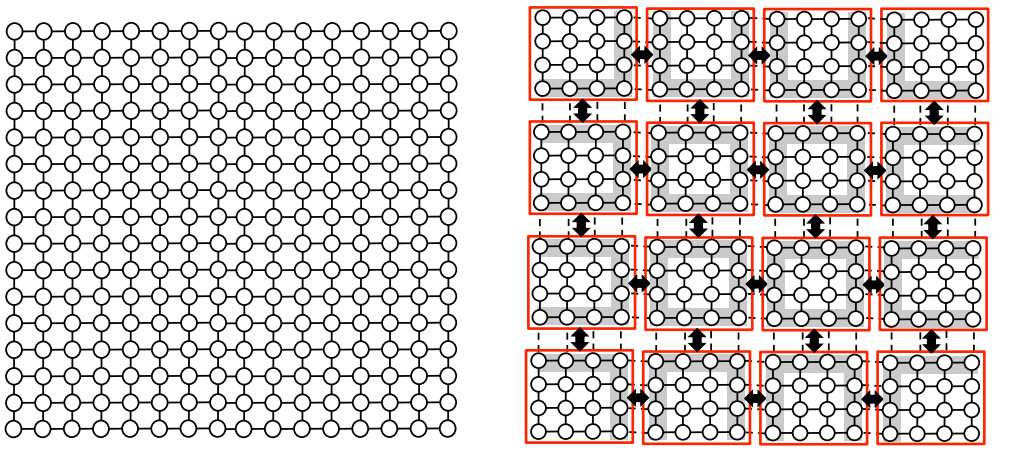


Figure 3.4 Data set decomposition for stencil computation with many FPGAs

As shown in these results, this clock variation is not trivial problem and it is impossible to design the proposed hardware ignoring this important matter. Therefore, it is necessary to design a robust system that can absorb the clock variation.

3.4 Scalable Stencil-computation Methodology Employing Multiple Small FPGAs

Many types of FPGAs are provided from FPGA vendors, and there are some design choices to build a large system that cannot be implemented on a single FPGA. They are (1) to use a few large FPGAs or (2) to use multiple small FPGAs.

In the former, using a few large FPGAs, the designed system can work at high speeds. However this approach requires very long time and to verify the system behavior and to generate an FPGA circuit image file (bitstream), because the implemented digital circuits are larger than using small FPGAs.

Fortunately, the proposed architecture for stencil computation consists of numerous redundant components. This means that it is possible to build the proposed system by downloading the identical circuit data into multiple FPGAs. I detail the design overview in the following sections. Additionally, even if an FPGA node is broken, the system can work successfully by replacing the broken FPGA node with new one. Although the system operation speed is inferior to the former approach, the system can be built at lower cost. Given these advantages, I choose a multi-FPGA based approach.

3.4.1 Data Set Decomposition

Figure 3.4 shows data set decomposition for stencil computation with many FPGAs. The white circles stands for data values. Connection line between values represents each value via the line is neighborhood. Figure 3.4 (a) shows a pre-divided data set for stencil computation that is composed of 16×16 data values. Figure 3.4 (b) shows the data set decomposition to 4×4 FPGAs.

Stencil-computation data set is divided and stored in FPGA internal memory. Figure 3.4 (b) shows that an assigned data set into an FPGA is 4×4 data values. In the practical system, an assigned data set is composed of 64×128 data values, which is the capacity limitation of an FPGA used in my proposed method. The size of the assigned data set is fixed, in other words stencil computation data set can be changed according to the number of FPGA nodes. For instance, 4×4 FPGA array system can execute 256×512 stencil computation.

A data value calculated at a time step k is used for calculating the neighborhood value at the next time step $k+1$. If the neighborhood value is stored in not same FPGA internal memory but an adjacent FPGA, the values used for calculating the neighborhood value have to be sent by the neighborhood value calculation at the next time step. The arrow and gray-toned shaded area represent the data communication between FPGAs and the data values transferred to an adjacent FPGA. As mentioned, the data values have to be sent to an adjacent FPGA in an appropriate timing that is not too late and not too soon.

3.4.2 Computation Order Optimization

To mitigate computation stall, I propose a computation order optimization approach based on the increase in the data communication slack. Figure 3.5 shows calculation order of conventional method (a) and proposed method (b) for the two FPGA stencil computation. Here, the assigned data set to an FPGA are 4×4 data values. In this example, the assigned 16 data values are updated in each Iteration. To simplify the explanation, the data value calculation and update can be done within a single cycle in this example. The practical required cycles are discussed in the following sections.

In Figure 3.5 (a), FPGA (A) and FPGA (B) execute stencil computation in same order. The dotted line region stands for an assigned data set to an FPGA. The circles are data values and the alphabet in the circle stands for an FPGA ID. For instance, FPGA (A) execute stencil computation in the order like A0, A1, A2, ... , A14, A15. As shown in Figure 3.5 (a), computation order of each FPGA is downward directed along with the arrow. FPGA (A) calculates A0 value at 0th cycle and A1 value at 1st cycle. Similar to this, FPGA (B) calculates B0 value at 0th cycle and B1 value at 1st cycle. In this example, each FPGA can obtain the result within a single cycle. After stencil computation at a time step, the computation is moved to the next time step. In this example, stencil computation in each Iteration

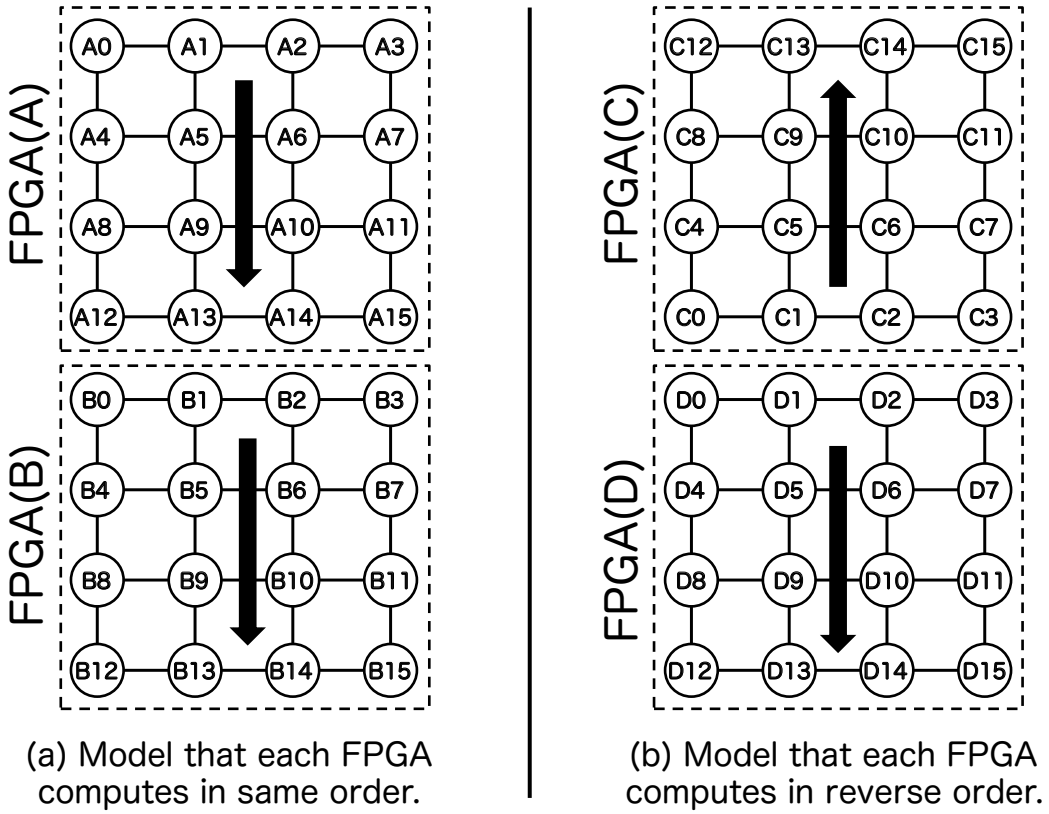


Figure 3.5 Calculation order of conventional method (a) and proposed method (b) for the two FPGA stencil computation

requires 16 cycles. The first Iteration begins at 0th cycle, the second does at 16th cycle, and the third does at 32nd cycle.

In Figure 3.5 (a), the data value B1 calculation requires the data values that are A13, B5, B0, and B2. Therefore A13 has to be sent from FPGA (A) to FPGA (B) for B1 calculation. In this computation order, A13 is calculated at 13th cycle in the first Iteration, and in the second Iteration B1 is calculated at 17th cycle using A13, B5, B0, and B2. It means that A13 has to be sent within three cycles (14th, 15th, and 16th cycle). Similar to A13, it is mandatory for A12, A14, and A15 to be sent within three cycles. In other words, if $N \times M$ data values are assigned to an FPGA, values in the boundary area have to be sent within $N - 1$ cycles after they are calculated.

Figure 3.5 (b) shows the proposed method to address this problem. Here, computation orders of FPGA (C) and FPGA (D) are reverse each other. It means that the computation order of FPGA (C) is upward directed against FPGA (A). In this example, in order not to stall D1 calculation at the second Iteration, C1 has to be sent within 15 cycles (2~16). In other words, the data communication slack is increased from 3 cycles to 15 cycles due to the computation order modification. If $N \times M$ data values are assigned to an FPGA, the slack is $N \times M - 1$ cycles.

Due to this proposed method, it is increased that the data communication slack between FPGAs that is almost equal to the cycles required to calculate all data values at an Iteration. If two computation orders are face-to-face, the slack is same. In other words, if computation orders of FPGA (C) and FPGA (D) are face-to-face, the slack is same as Figure 3.5 (b).

The data communication is necessary between right and left sides. If two FPGA (C) are placed in the left and right, C3 at the left and C0 at the right are neighborhood and the data communication slack is 12 cycles. This slack is given by $(N \times M - M)$ if $N \times M$ data values are assigned to an FPGA.

Therefore, the proposed method gives the sufficient data communication slack that is about an Iteration required cycles. Figure 3.5 shows stencil computation on two FPGAs. If stencil computation is executed on multiple FPGAs, the computation can be realized by placing a pair of FPGA(C) and FPGA (D) as many as required. In Figure 3.5, the number of cycles required to calculate a data value is defined as a single cycle to simplify the explanation of the proposed method. In general, if the number of cycles is k, the data communication slack is $(N \times M - M) \times k$.

3.5 Design of Scalable Stencil-computation Architecture

3.5.1 System Architecture

Figure 3.6 shows the system architecture of an FPGA node for the proposed scalable stencil-computation accelerator.

The system architecture has eight multiply-adder units and a synchronization hardware represented in MADD and Sync in the figure. SER and DES stand for a serializer and deserializer for the data communication between FPGAs respectively. The center numbered blocks from 0 to 9 are FPGA internal memory blocks.

Figure 3.6 also shows the MADD detail. The squares in the MADD are registers, and the multiplier and adder are floating-point arithmetic units with IEEE 754 Standard for single precision, which both have seven pipeline stages. Adding design parameters like the number of pipeline stages, the multiplier and adder can be automatically created from an IP core generator provided by FPGA vendors. The MADD data path consists of 16 pipeline stages because the MADD has two registers, which identical to a data path composed of a multiplier and adder that both have eight pipeline stages.

FIFOs store calculated values at MADDs that have to be sent to adjacent FPGA nodes. After that, the stored data is sent to an adjacent FPGA node via the multiplexer (mux8) and a serializer. For implementation of the serializer and deserializer, two techniques are used that are Clock and Data Recovery (CDR) and NRZI Data Encoding.

Figure 3.7 shows relationship of internal memory blocks and values in an FPGA node. The numbered blocks in Figure 3.7 correspond to those in Figure 3.6.

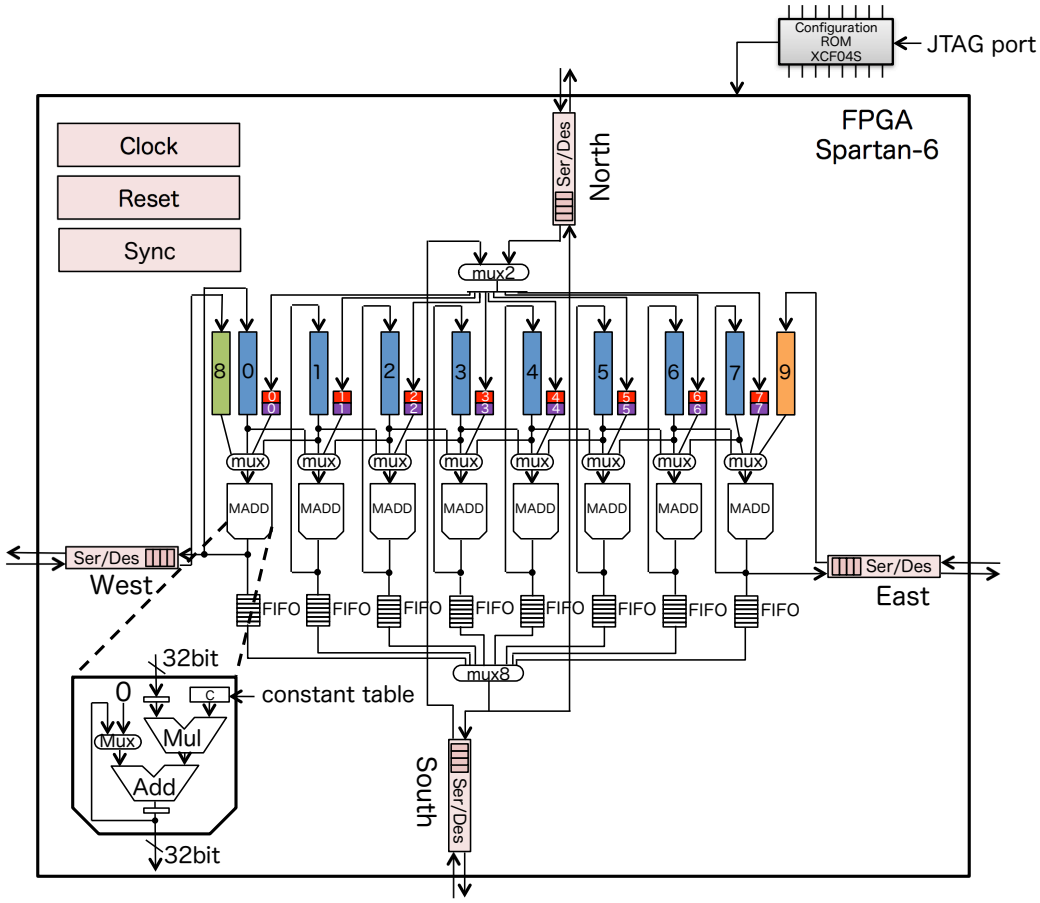


Figure 3.6 System architecture of a single FPGA node for the scalable stencil-computation accelerator

The 2D data set assigned to each FPGA is split along with a vertical direction and is stored in each internal memory from 0 to 7, which is in the dotted line area. In other words, 64×128 data values are assigned to an FPGA, each internal memory stores 8×128 data values split along with a vertical direction. The remained internal memory blocks, which are 8 and 9, are used to store transferred data values from an adjacent FPGA node.

Figure 3.8 shows the pipelined operation of multiply and adder unit for floating-point numbers. The circles and squares in the figure stand for data values and the multiplier calculation results using data values and weighting factor. As described before, it can be seen that the multiplier and adder both have eight pipeline stages. Figure 3.8 (a) and (b) show the numbered data set from 0 to 29 and hardware component names including the multiplier, adder, multiplexer, and wires. In this figure, it is illustrated that the pipelined operation to calculate data values from 11 to 18.

At 0~7 cycles, data values 1~8 read from an internal memory block are input to the multiplier cycle by cycle. The row of Mul input depicted in Figure 3.8 (c) is a data value input to the multiplier at each

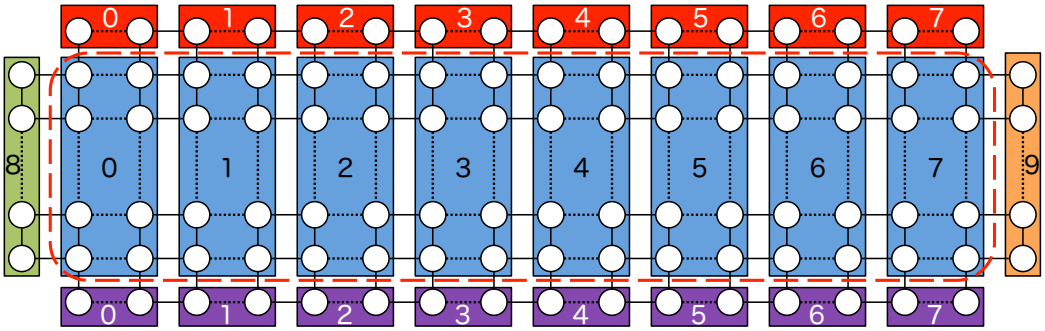


Figure 3.7 Relationship of Block RAM and values in a single FPGA node

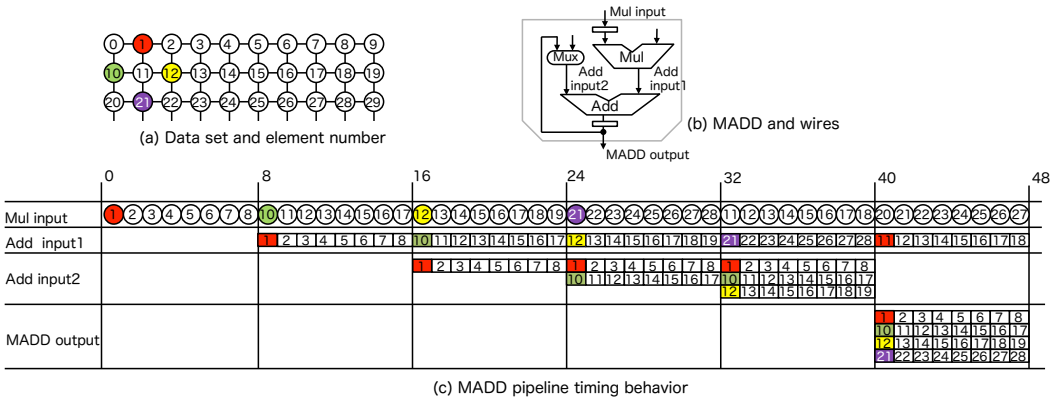


Figure 3.8 Pipelining of multiply and add unit for floating-point numbers

cycle. In other words, the circles numbered from 1 to 8 in the row of Mul input at 0~7 cycles means that the data values 1~8 are input to the multiplier cycle by cycle.

In the next, the data values 10~17 are input to the multiplier and simultaneously the multiplier calculation results are emitted from it. The row of Add input1 depicted in Figure 3.8 (c) is the multiplier calculation results that are input to the adder, and the data values are illustrated in the square form that means the multiplication results using a data value and weighting factor input to the multiplier. For instance, the numbered data value 1 and weighting factor are input to the multiplier at cycle 1, and then the multiplication result is emitted from it at cycle 8.

At 16~23 cycles, the three operations occur at the same time, which are the data values 12~19 are input to the multiplier, and the feedback data values 1~8 and the multiplier calculation results 10~17 are summed at the adder.

At 24~31 cycles, two data values are in the row of Add input2 depicted in Figure 3.8 (c), for instance the numbered squares 1 and 10 are in the row. This means the summation result of the feedback data 1 and multiplication result 10.

Similar to this, MADD calculation results are in the row of MADD output, here the four squares are in there. For instance, the data values 1, 10, 12, and 21 are in there at cycle 40, and this means the calculation result for the data value 11.

Finally the MADD calculation result using the left, right, top and bottom data values, which is multiplied and accumulated, are emitted at cycle 40~47.

When the calculation results are stored in an internal memory block, it has to be careful not to overwrite the data that is used for another calculation at the same time step. In general, temporary buffers like FIFO are used to address this problem, but the proposed architecture has no need to prepare them because the pipeline stages of MADD plays a role as the temporary buffer.

In Figure 3.8, the data values 11~18 are updated at 40~47. These data values are input to the multiplier at 32~39 and then never used for the rest of the MADD calculation at the time step. Therefore, it is guaranteed to correctly update the data values 11~18 without temporary buffers like FIFO. This pipeline scheduling is valid only when the width of data set is equal to the number of pipeline stages of the multiplier and adder. In other words, the proposed architecture employs eight pipeline stages, and that is why the width of data set calculated at MADD is eight.

The pipeline filling rate can be given by $(C - 8/C) \times 100$ where C is the number of required cycles for stencil computation. This architecture can almost achieve 100% of the pipeline filling rate because C is basically massive. Besides, this architecture has no need for temporary buffers to accommodate the update timing, which can save the hardware resource usage. In other words, this architecture can achieve high performance computation with the appropriate hardware resources.

The data communication slack with the pipeline operation can be quantitatively expressed. At first, I define the number of cycles required to calculate a data value, the number of pipeline stages, the height of data set stored in an internal memory block as k, n, and m respectively. In this architecture, these values are given by the following equation.

- $k = 5 \times n + 1 = 41$
- $n = 8$
- $m = 128$

Because in this architecture an internal memory block has 8×128 data values, n and m are 8 and 128 respectively. The number of cycles required to calculate all data values assigned to an FPGA and to store the calculation result in the memory can be given by the following formula.

$$(k - 1) + n + 4n \times (m - 1) = 4112 \tag{3.2}$$

MADD begins to calculate data values for the next Iteration 16 cycles before the calculation result for the lower right data value located in the data set is emitted from MADD. Therefore, the number of

required cycles for an Iteration is given by the following formula.

$$(k-1)+n+4n\times(m-1)-16=4096 \tag{3.3}$$

Using the number of cycles required cycles for an Iteration, the slack to upward or to downward communicate MADD calculation results can be given by the following formula.

$$\begin{aligned}(k-1)+n+4n\times(m-1)-16-k &= 4096-41\\ &= 4055\end{aligned} \tag{3.4}$$

In the next, I explain the data communication slack along with a horizontal direction. The data communication slack from right to left side can be given by the following formula.

$$\begin{aligned}&(k-1)+n+4n\times(m-1)-16-k+3n-1\\ =\;& 4096-41+23\\ =\;& 4078\end{aligned} \tag{3.5}$$

Also, the slack from left to right side can be given by the following formula.

$$\begin{aligned}(k-1)+n+4n\times(m-1)-16-k+1 &= 4096-41+1\\ &= 4056\end{aligned} \tag{3.6}$$

Therefore, the data communication slack with the pipeline operation is equal to the number of required cycles for an Iteration, as described in Section 3.4.2.

As shown in the above formulas, the data communication slack is dependent on the number of data values stored in an internal memory block. In other words, higher speed data communication hardware is necessary as the internal memory capacity is smaller. In this work, I use the data communication hardware used in ScalableCore system. It is tough work to implement and verify data communication hardware operating at high clock frequencies from scratch. Therefore, in this research existing IP cores are used in order to mitigate the design and verification complication.

3.6 Implementation of Scalable Stencil-computation Accelerator

3.6.1 Development Flow of Scalable Stencil-computation Accelerator

Scalable stencil-computation accelerator is developed in the following three steps.

At first, I implement a software simulator in C++. This software simulates stencil computation on multiple FPGAs with cycle level accuracy. I verify the simulator behavior comparing the simulation results with the execution results of a stencil computation program with function level accuracy.

In the next, I implement the stencil-computation hardware in Verilog HDL, based on the software simulator behavior. To verify that the implemented hardware works as intended, I use Icarus Verilog

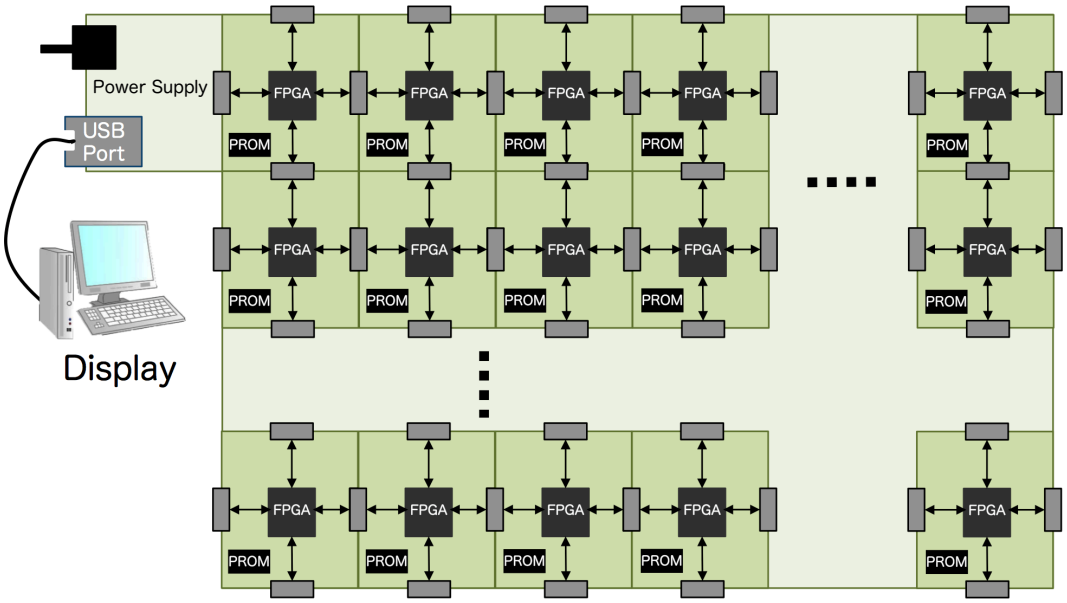


Figure 3.9 The configuration of the implemented scalable stencil-computation accelerator

[25] that is an open-source RTL simulator and compare the RTL simulation results with the cycle-accurate software simulation results.

After the verification, the designed hardware for stencil computation is implemented on ScalableCore system.

Figure 3.9 shows the configuration of the implemented scalable stencil-computation accelerator. The accelerator computation results are displayed on the connected host PC via a USB cable and are compared with a stencil computation program in C language. In this verification, a data value assigned to the upper left FPGA node is sent to the host PC Iteration by Iteration, and then compared with the program results. The verification can be done without displaying all data values calculated in the system by comparing the results for massive Iterations. This is because the computation kernel simulates fluid dynamics like wave or heat propagation, which means that all values eventually become incorrect if a calculation result for a data value is wrong.

3.6.2 Identification of Location Information

As described in Section 3.4.2, the data communication slack is increased by changing the computation order according to FPGA positions. It is necessary to identify that each FPGA is on an odd or even row so that each FPGA decides that its computation order is upward or downward. To deal with the matter, I implement hardware logic to set each FPGA computation order.

Figure 3.10 shows the mechanism to identify odd/even row FPGAs. The arrow in the figure stands for the computation order described in Figure 3.5. The implemented hardware is a simple combinational

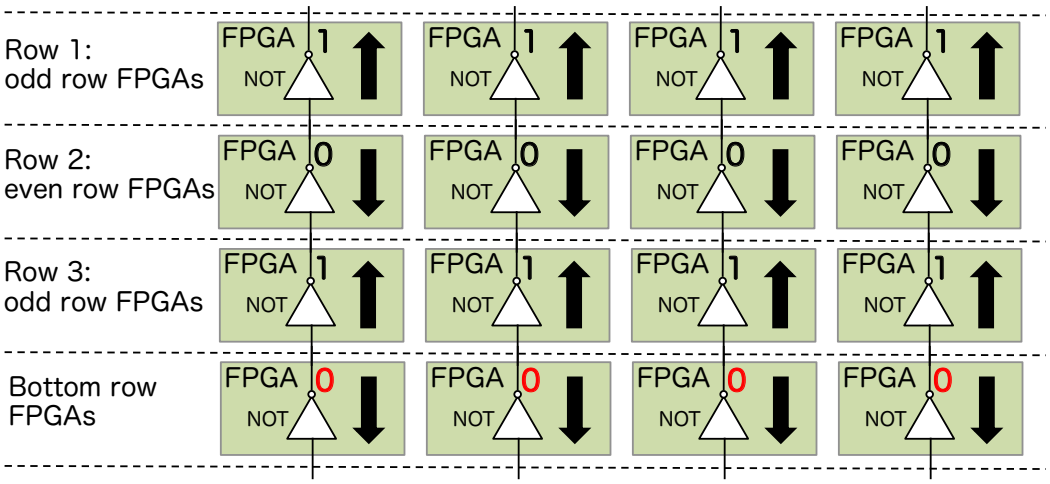


Figure 3.10 The mechanism to identify odd/even row FPGAs

circuit. Each FPGA is connected via a single wire along with a vertical direction. Each FPGA has a NOT gate whose input is downside and output is connected with an upside FPGA.

All FPGA nodes have an identification feature for adjacent FPGA nodes existence. Using this feature, FPGA nodes that have no connection to downward direction identify that they are on the bottom row, and then emit 0 value to upward FPGA nodes. The other FPGA nodes receive the downward input values and emit the inverted values to upward FPGA nodes. As a result, the odd and even row FPGA nodes hold 1 and 0 respectively. Using the information, each FPGA identifies the own location. The benefit of the mechanism is significant small hardware resource usage because each FPGA has only an inverter.

3.6.3 Synchronization Mechanism to Address Clock Variation Problem

The implemented stencil-computation accelerator has no global clock. This means that a synchronization mechanism is mandatory to address the clock variation problem described in Section 3.3.2.

Figure 3.11 shows the synchronization mechanism to deal with the variation of clock oscillators. For this mechanism, FPGA A is defined as the master. The other FPGAs B, C, and D are synchronized with the signal sent from the master, and execute stencil computation. Until they receive the synchronization signal, stencil computation on them is stalled.

The master generate and send the synchronization signal with a period of $\alpha+\beta$ where α and β are the number of required cycles for an Iteration and a margin to absorb the clock variations of FPGA nodes respectively. This margin β is set to absorb them in α.

Figure 3.12 shows the implementation of synchronization mechanism to deal with the variation of clock oscillators. The α and β in Figure 3.12 are same as those of Figure 3.11. The other FPGA

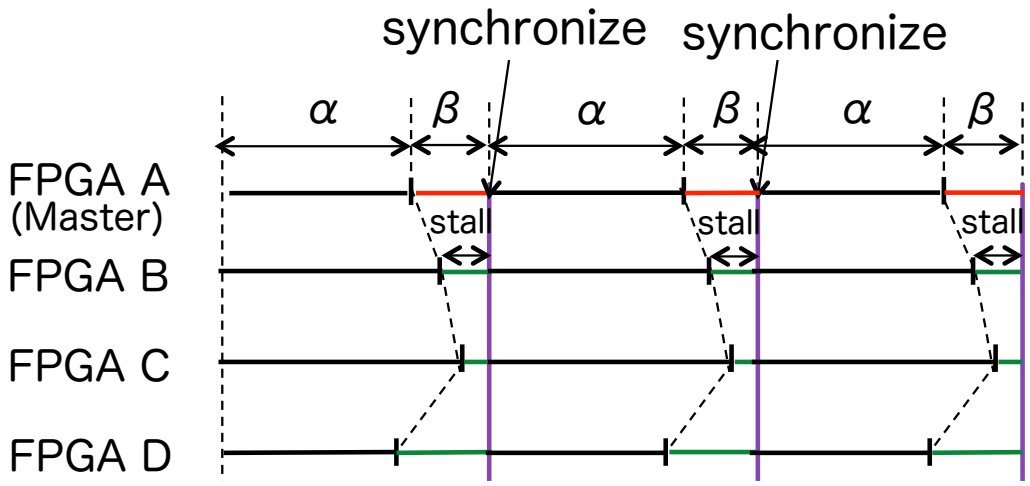


Figure 3.11 Synchronization mechanism to deal with the variation of clock oscillators

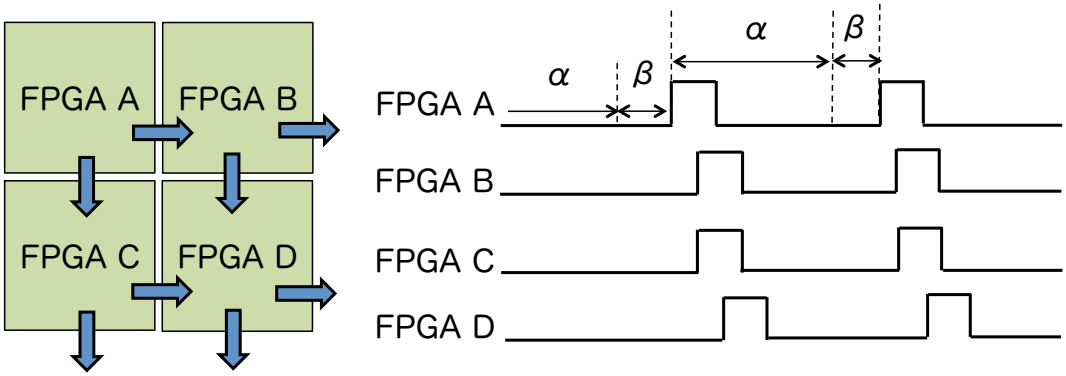


Figure 3.12 Implementation of synchronization mechanism to deal with the variation of clock oscillators

nodes, which are B, C, and D, send the synchronization signal to the right and down FPGA nodes and all FPGA nodes are eventually synchronized with the master node. As mentioned before, The FPGA nodes that receive the synchronization signal restart stencil computation.

The synchronization signal is sent for tens of cycles in order to prevent communication failure to send and receive it. FPGA nodes identify the synchronization signal if they continuously receive the signal asserted at high level for several cycles.

Table 3.2 Hardware resources usage in a single FPGA

Hardware element	Used / Available	Utilization
Slices	2,271 / 2,278	99%
LUTs	7,805 / 9,112	85%
Block RAM	28 / 32	87%
DSP48A1	32 / 32	100%

3.7 Evaluation

3.7.1 Setup

I implement the stencil-computation accelerator in Verilog HDL and synthesize the implemented hardware with Xilinx ISE 14.2. The generated bitstream is downloaded into Xilinx Spartan-6 XC6SLX16 with 64KB internal memory on each node. As mentioned before, in order to implement MADD, I use a generated IP core using Xilinx CORE Generator. For one MADD implementation, four DSP hardware blocks in the FPGA are used. The FPGA has 32 DSP blocks, in other words eight MADDs can be implemented on the FPGA.

For behavior verification of the accelerator, I use a stencil-computation C program using a Softfloat library that has same precision with MADD floating-point operation, and evaluate the computation performance using a C program without the library.

In this evaluation, the number of Iterations is 5,800,000, and the computation results are displayed on the connected host PC via a USB cable in each Iteration and are compared with the C program with the library. As a result, I confirm the computation results are identical to those of the program in all Iterations.

3.7.2 Hardware Resource Usage

Table 3.2 shows hardware resources usage of an FPGA to implement the stencil-computation accelerator. The leftmost, center, and rightmost columns stands for hardware component types, the number of used and available components, and hardware resource utilization.

The LUT and Block RAM utilization are 85% and 87% respectively. Block RAM is internal memory in the Xilinx FPGA. The utilization of Slice is 99%, this is because the serializer, deserializer, identification feature, and synchronization hardware are implemented, in addition to MADD. All DSP hardware blocks (DSP48A1) are used to implement eight MADDs as mentioned before, and that is why the hardware resource utilization is 100%.

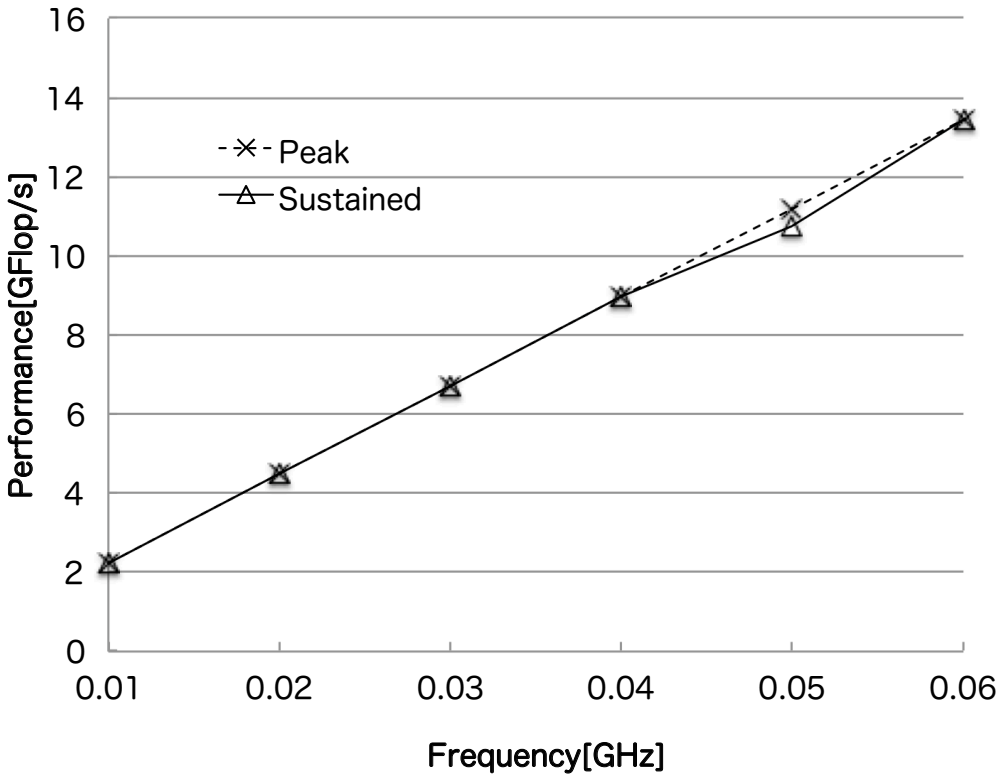


Figure 3.13 Sustained performance and peak performance of stencil calculation in the FPGA array of 16 nodes

3.7.3 Stencil Computation Performance

I define the operating frequency, the number of MADDs implemented on each FPGA, and the number of FPGAs as F GHz, N_{MADD}, and N_{FPGA} respectively. Each MADD can simultaneously execute multiplication and addition cycle by cycle. Due to this, a single MADD can achieve $2 \times F$ GFlop/s peak performance. Using N_{MADD} and N_{FPGA} in addition the peak performance of a single MADD, the hardware peak performance P_{hpeak} GFlop/s of the FPGA array system is given by

$$P_{hpeak} = 2 \times F \times N_{MADD} \times N_{FPGA} \tag{3.7}$$

where N_{MADD} is 8. For instance, P_{hpeak} of the accelerator using 100 FPGA nodes operating at 0.06GHz (60MHz) is 96GFlop/s given by the above formula.

In the next, I define the peak performance for stencil computation. As shown in Figure 3.1, stencil computation requires four multiplications and three additions, which is unbalance between the two operations. Therefore, the average utilization of the multipliers and the adders in MADD is given by

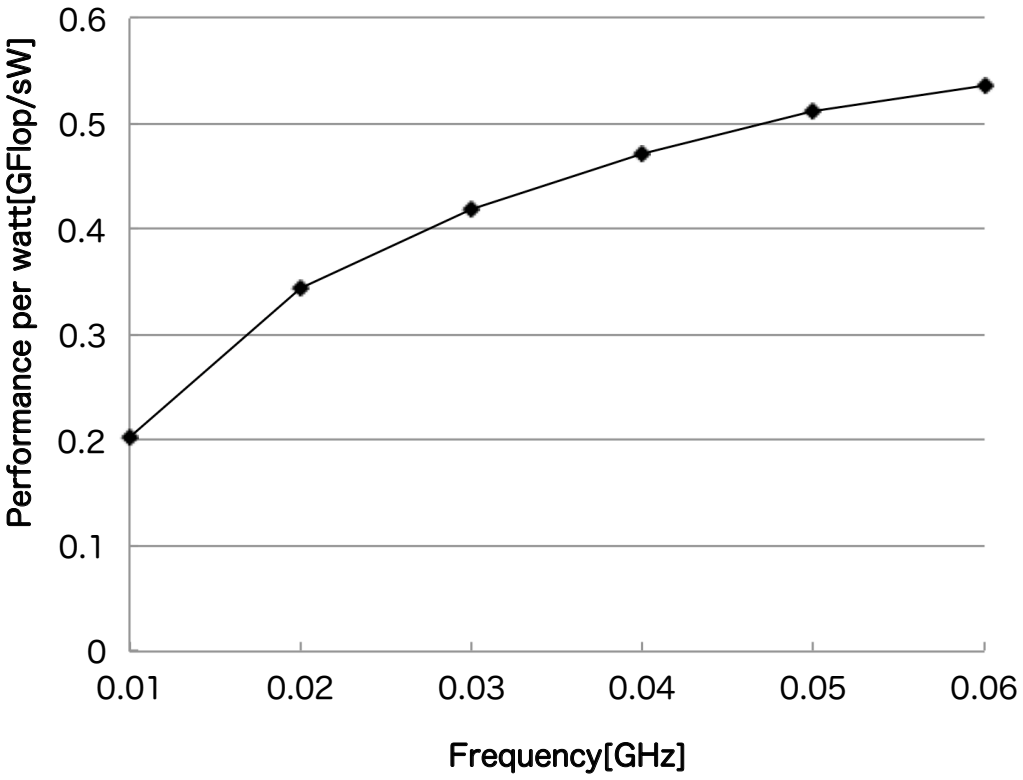


Figure 3.14 Computation performance per watt in FPGA array of 16 nodes

$100 \times (4 + 3)/8 = 87.5\%$. In other words, the peak performance P_{peak} for stencil computation is given by the following formula.

$$P_{peak} = P_{hpead} \times 0.875 \tag{3.8}$$

I evaluate the peak and sustained performance of the stencil-computation accelerator using 16 FPGA nodes, according to the operating frequency. The data set for stencil computation is 256×512 data values. The sustained performance is given by dividing the number of floating-point operations by the computation time. The number of floating-point operations is given by

$$OPs \times GRID \times IterNum = 7 \times 256 \times 512 \times 5,800,000$$

where the number of floating-point operations required to calculate a data value is OPs, and the number of data values and iterations $GRID$ are $IterNum$.

In this evaluation, the computation time is measured using a stopwatch. The number of Iterations 5,800,000 requires about 10 minutes computation time for stencil computation if the operating frequency is 40MHz. The 10 minutes is very long execution time, therefore the measurement error of the stopwatch can be ignored. The computation time is from the initialization completion of the data set

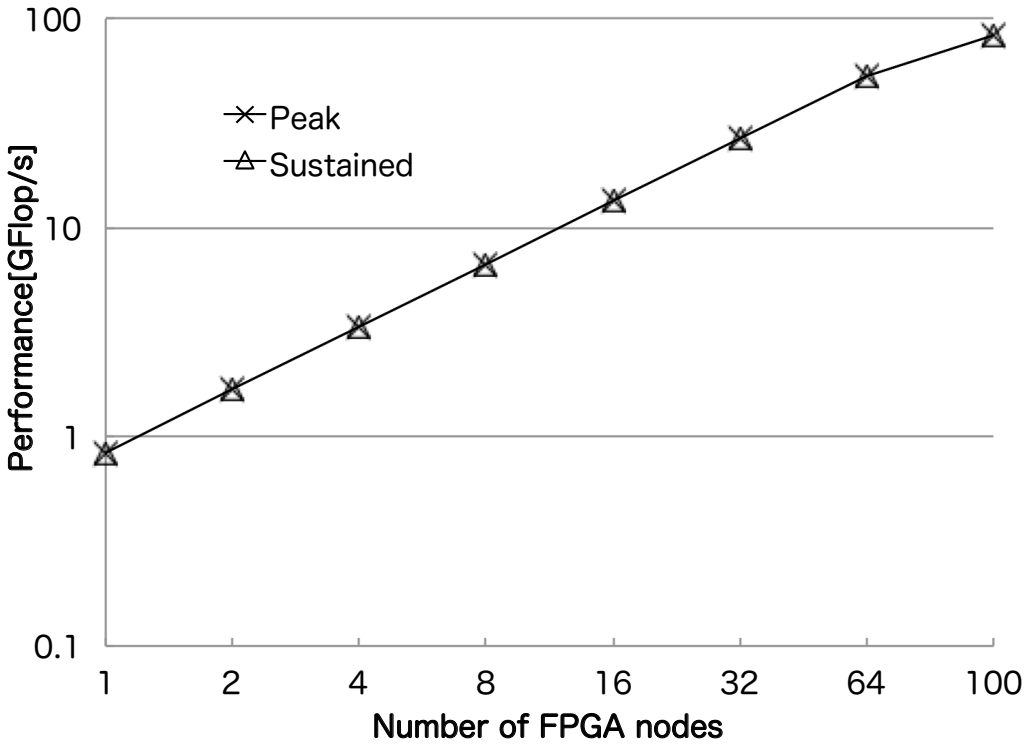


Figure 3.15 Sustained performance and peak performance of stencil calculation running 60MHz

to the computation termination. In this system when the data set initialization is done, Light-Emitting Diodes (LEDs) is luminous, which is the measurement start sign. After that, when the computation is terminated, the computation results are displayed on the host PC, which is the measurement finish sign.

Figure 3.13 shows the peak and sustained performance of the FPGA array system composed of 16 nodes. The peak and sustained performance are almost identical as shown in the figure and it is that obvious the proposed computation methodology has little performance overhead.

In general, a gap between the peak and sustained performance is due to the data preparation like initial load instructions. However, stencil computation requires massive execution time, and that is why the data preparation overhead can be ignored. Authors in [77] states that this performance gap shown in [8, 9] is due to multicores and GPUs architectural aspects including memory bandwidth limitations. To address this problem, I implement the appropriate stencil-computation hardware with pipelined operation. Besides I use FPGA internal memory instead of an external memory DRAM to avoid the performance bottleneck because of the memory bandwidth limitation.

The implemented C program for the performance comparison is compiled with -O3 optimization option. A single thread of Intel Core i7-2700K operating at 3.5GHz executes the compiled program

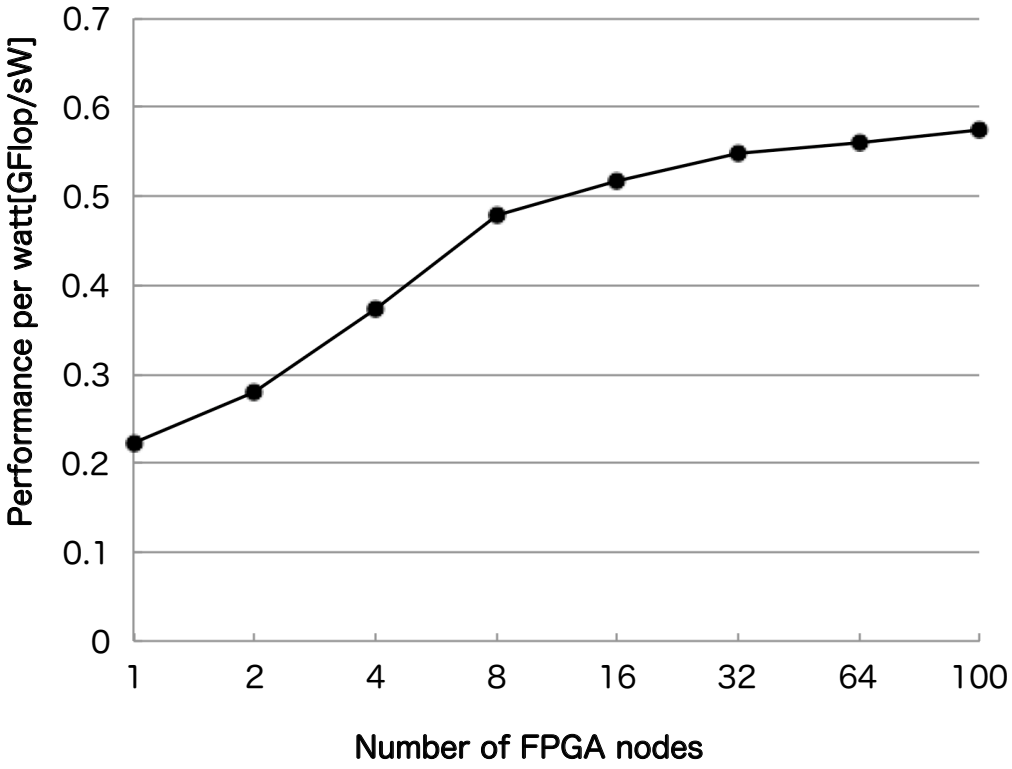


Figure 3.16 Computation performance per watt in FPGA array running 60MHz

and achieves 3.31GFlop/s sustained performance. This result is almost same as 2.8GFlop/s sustained performance reported in [8]. The FPGA array composed of 16 nodes achieves 13.42GFlop/s, which is about 4x better than the single thread result of Intel Core i7-2700K.

Figure 3.14 shows the power efficiency of FPGA array system composed of 16 nodes, according to the operating frequency. The leftmost boards of the FPGA array system play a role of power supply for all FPGA nodes. These boards have AC adapters that are connected to a single power tap. I evaluate the entire power consumption of the system by connecting the tap to Watt Checker (SANWA SUPPLY TAP-TST5). This means that the power consumption includes that of the power supply boards in addition to the FPGA nodes. The power consumption is always constant when the system executes stencil computation.

Figure 3.14 also shows that the power efficiency becomes better as the operating frequency is raised. In general, each FPGA has own sweet spot for the operating frequency. Due to this, the lower operating frequency compared with the sweet spot degrades the power efficiency, and that is why the graph is illustrated in the positively sloped curve.

Figure 3.15 shows the peak and sustained performance of the FPGA array system operating at

60MHz according to the number of FPGA nodes. As mentioned before, the proposed accelerator can change the data set that can be computed in the same computation time by increasing the number of FPGA nodes. An FPGA node has 64×128 data values, in other words the 10×10 FPGA array system can compute 640×1280 data values.

Figure 3.15 also shows that the peak and sustained performance is almost identical. The 100-FPGA array system can compute 640×1280 data values in 396 seconds, but a single thread of Intel Corei7-2700K spends 10,053 seconds. In other words, The 100-FPGA array system achieves about 25x sustained performance compared with a single thread of Intel Corei7-2700K. Compared with a single GPU NVIDIA Tesla C1016 [9], my proposed accelerator using 100 FPGA nodes achieves 83.9GFlop/s, while the GPU achieves 51GFlop/s. In other words, the sustained performance of my proposed accelerator is 1.65x better than a single GPU NVIDIA Tesla C1016.

The stencil-computation accelerator is based on ScalableCore system. Because of the amperage rating restriction, ScalableCore system connects up to 16 nodes along with a horizontal direction. However along with a vertical direction, it is possible to place two-fold FPGA nodes as desired, as long as the power supply boards can be also placed. The two-fold comes from the computation order optimization described in Section 3.4.2. To improve the FPGA array system, one of the effective approaches is to modify how to supply the power to all FPGA nodes. For instance the current FPGA array system only has the leftmost power supply boards, but 32 FPGA nodes connection along with a horizontal direction can be realize and improve the entire performance if the power supply boards can be placed at the rightmost side.

Figure 3.16 shows the power efficiency of the FPGA array system operating at 60MHz, according to the number of FPGA nodes. This evaluation result demonstrates that the power efficiency is better as the number of FPGA nodes is increased. This is because the power consumption of the system includes that of the power supply boards and the power tap overhead in addition to the FPGA nodes. Therefore if the FPGA array is composed of lots of nodes, the power overhead due to the leftmost boards and power tap can be ignored, but if not, the overhead is not trivial. Because the power efficiency becomes worse as the number of FPGA nodes is decreased, the graph is illustrated in the positively sloped curve. In other words, the power overhead is decreased as the number of FPGA nodes is increased, and then the power efficiency is close to the constant value. The stencil-computation accelerator with 100 FPGA nodes achieves about 0.57GFlop/sW, which is 3.8x better power efficiency than NVIDIA GTX280 graphics card [10].

3.8 Related Work

Some research institutes have reported previous studies to aim to optimize stencil computation for multicore microprocessors and GPUs.

Augustin et al. [8] reported stencil computation using Intel Xeon E5220 quad-core processor operating at 2.26GHz. If a single core is used, 2.8GFlop/s is achieved that is 31% of the peak performance 9GFlop/s. Also two E5220 processors can achieve 15.9GFlop/s that is 21.8% of the peak performance 72GFlop/s

Phillips et al. [9] reported stencil computation with NVIDIA TESLA C1060 GPU. A single GPU achieves 51.2GFlop/s that is 65.6% of the peak performance for double precision. The sustained performance is degraded if a GPU cluster is used. In stencil computation for $256 \times 256 \times 512$ data values, the cluster composed of 16 GPUs achieves 42.2% of the peak performance.

Some FPGA-based accelerators for stencil computation have been proposed. [13, 77, 73] propose stencil computation hardware based on programmable systolic array architecture using several Terasic DE3 evaluation boards with ALTERA Stratix Ⅲ EP3SL150 FPGA. Stratix FPGAⅢ EP3SL150 FPGA is a large FPGA that has 142K logic elements and massive internal memory. The memory capacity is 12.4x compared with the Xilinx Spartan-6 FPGA and the cost of the evaluation board is about 34x compared with an FPGA node in my accelerator. The systolic array architecture is based on a pipeline scheduling method used for the cellular automata. As a result, the proposed accelerator achieves linear scalability for multiple devices with a constant memory bandwidth. There are several differences in terms of the proposed architecture and employed FPGA. In [77], the stencil-computation accelerator using nine Terasic DE3 evaluation boards is implemented and achieves 1.3GFlop/sW that is 2.28x better compared with my 100 FPGA nodes accelerator. However the development cost is 3x than my proposal. It means that [77] is better in the power efficiency but my accelerator is better in terms of the development cost. Also, [78] proposes an FPGA array system for the Poisson's equation.

Similar to my work, Mencer et al. [79] developed a scientific computation accelerator called CUBE using 512 FPGAs. These FPGAs are unidimensionally connected. Yoshimi et al. [80] discuss the CUBE effectiveness with edit distance computation algorithm that is a typical stream-oriented application mainly composed of integer calculations.

3.9 Summary

I proposed the scalable stencil-computation methodology employing multiple small FPGAs and the hardware architecture, and demonstrated that it is possible to realize the FPGA array system by using the three key techniques; these are the computation order optimization mechanism considering location information of each FPGA, the deeply pipelined stream computation unit, and the synchronization mechanism to absorb clock variations between FPGAs.

I detailed the design and implementation process. At first, I developed the cycle-accurate software simulator for the hardware architecture in C++, and then implemented the computation unit in Verilog HDL, verifying the computation unit behavior using the software simulator.

The designed hardware was based on 100 FPGAs and I evaluated the FPGA-based accelerator in terms of the sustained performance, scalability, and power consumption. As a result, my proposed accelerator accurately worked and achieved about 0.6GFlop/sW, which is about 3.8 better than a NVIDIA GTX280 graphics card.

Chapter 4

A High Performance FPGA-based Sorting Accelerator with a Data Compression Mechanism

4.1 Motivation

Sorting is one of the most fundamental computation kernels in data management, and lots of approaches to accelerate the kernel have been proposed [81, 82, 83, 84, 85, 86, 87, 88]. These approaches offer significant results, but mostly these studies utilize SIMD instructions of Intel processors [81, 87, 88] to exploit data-level parallelism or experiment on rich hardware environments such as supercomputers [85] or clusters [87]. It is unclear that these approaches are available on low computational performance machines like embedded systems. Besides, Internet of Things (IoT) era is about to seriously begin due to mobile technology progressions, and large amounts of information are more and more generated from mobile devices, wireless sensors, and others. Therefore the future needs a sorting method that is available on any environment from embedded systems to high performance systems like servers.

To address the problem, I propose an FPGA-based sorting hardware called FACE [89], which combines *Sorting Network* and *Merge Sorter Tree*. The proposed sorting hardware is customizable by means of tuning design parameters, and I also provide an analytical model that accurately estimates the sorting performance depending on the hardware configuration. In other words, due to these characteristics designers can estimate sorting accelerator performance in advance and can implement the best one that fulfills cost and performance constraints. In this chapter I detail the design and implementation, and evaluate the proposed sorting accelerator in terms of the sorting performance and the hardware resource usage. To allow every designer to easily and freely use this accelerator, the Register Transfer Level (RTL) source code is available at [90].

My proposed sorting accelerator can be high performance by tuning design parameters, in that case, not only the hardware resource usage but also the memory bandwidth has to be considered. In fact, the

highest performance configuration in [89] suffers from the memory bandwidth limitation. To address this problem, I propose a *data compression mechanism* for the sorting accelerator. Among lots of data compression algorithms, I use an algorithm using the relative difference between values of continuous locations, which is based on [91]. As sorting is proceeded, the relative difference between them becomes smaller. This means that the algorithm is quite suitable for sorting. Besides, the algorithm can be implemented by a simple vector subtraction and addition. That is why I introduce this algorithm, and the data compression mechanism can improve the memory bandwidth utilization while keeping the operating frequency high.

The main contributions of this chapter are:

- To propose a high performance and customizable sorting accelerator with two sorting architectures and also to propose a detailed analytical model, therefore designers can estimate sorting accelerator performance in advance and can implement the best one that fulfills cost and performance constraints,
- To propose a data compression mechanism for the sorting accelerator in order to mitigate the bandwidth limitation of accessing the off-chip memory, and to show that the sorting accelerator with the mechanism achieves better performance than without it,
- To release the RTL source code in Verilog HDL as an open-source hardware in order to allow every designer to easily and freely use this accelerator, and to the best of my knowledge, this is the first open-source sorting accelerator in the world that is high performance, is customizable, and improves the memory bandwidth utilization.

4.2 Sorting Architectures

My proposed sorting accelerator takes advantage of the sorting network and the merge sorter tree. I describe these sorting architectures.

4.2.1 Sorting Network

A sorting network [92] is an algorithm that sorts a fixed sequence of numbers by using a fixed sequence of comparisons. The sorting network consists of two types of items, which are wires and comparators. The wires are running from left to right, carrying values (one per wire) that traverse the network all at the same time. Each comparator connects two wires. When a pair of values, traveling through a pair of wires, encounters a comparator, the comparator swaps the values only if the top wire's value is greater than the bottom wire's value. The sorting network benefits are to sort values in parallel and to be implemented without complicated hardware. That is why the sorting network is a desirable component for building high performance sorting hardware [93, 94, 95].

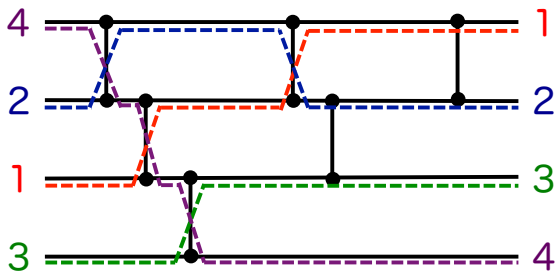


Figure 4.1 Bubble sort network with 4-inputs and 4-outputs

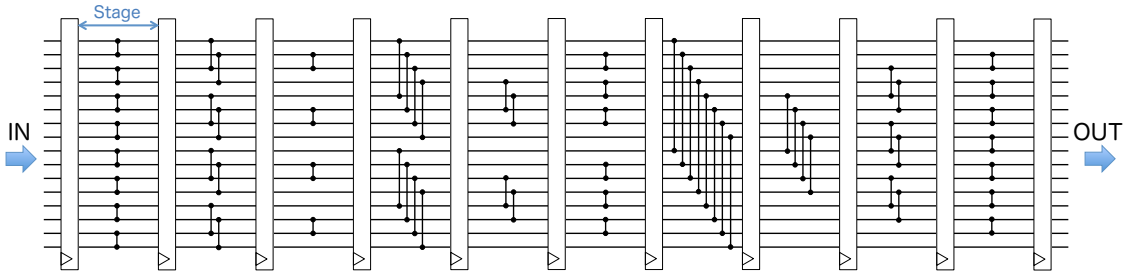


Figure 4.2 Pipelined synchronous Batcher's odd-even merge sort network with 16-inputs and 16-outputs

Figure 4.1 shows a sorting network with 4-inputs and 4-outputs. This network realizes bubble sort, since the largest value is carried to the bottom at first. By changing the connection of the comparators, sorting networks can realize lots of sorting algorithms, such as even-odd merge sort, bitonic sort, bubble sort, insertion sort, etc. In [93], authors implement several sorting networks on an FPGA and conclude that Batcher's even-odd merge sort network is the most efficient in terms of hardware resource usage and throughput. Consequently, our proposed hardware uses this sorting network.

Figure 4.2 shows Batcher's even-odd merge sort network [96] with 16-inputs and 16-outputs. This sorting network consists of 63 comparators and 10 stages. Although it is possible to be implemented as a purely combinational circuit, this case probably causes performance reduction because of large network delay. To address this problem, it is a common way to implement this network as a pipelined circuit by inserting registers between each stage, which prevents a degradation of the operating frequency and improves the network throughput [93]. This network is embedded in our proposed hardware.

4.2.2 Merge Sorter Tree

The merge sorter tree [97] has highly effective performance and good hardware resource usage. The merge sorter tree is a data path that executes merge process and the data path consists of connecting sorter cells as a perfect binary tree. Sorter cells compare two input-values and output one of them,

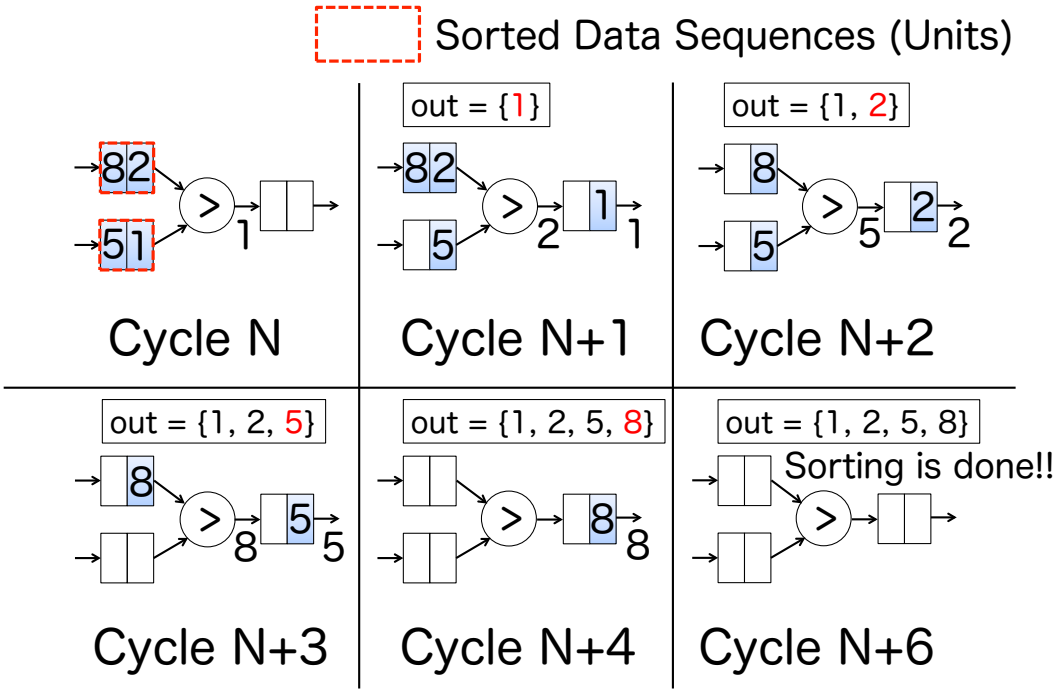


Figure 4.3 Sorting process in merge sorter tree

depending on its comparison result.

Figure 4.3 shows how elements are sorted in the merge sorter tree. The merge sorter tree in Figure 4.3 has two input ports. I define the tree in Figure 4.3 as 2-**way** merge sorter tree. If a merge sorter tree has k input ports, the tree is called k-way merge sorter tree. Now, I explain how elements are sorted in this 2-way merge sorter tree.

First, at Cycle N, each way outputs integers of 2, and 1. Then, 2 and 1 are compared. The data sequences in the leftmost FIFOs must be sorted. I define these data sequences as **Units**. In this example, the sorted element 8 and 2 in the upper FIFO is a Unit, and the element 5 and 1 in the lower FIFO is another Unit. The sorter cell outputs the smaller element depending on the comparison result, unless the output FIFO of the sorter cell is full. At Cycle N+1, 1 is emitted from the root. At the same time, 2 and 5 are compared, and then the sorter cell outputs 2. At Cycle N+2, 2 is emitted from the root. At the same time, the sorter cell outputs 5 depending on the comparison result between 8 and 5.

As shown in Figure 4.3, the Units are merged in the tree, and then the root of the tree emits the sorted data sequence. In other words, the tree merges the two Units, and then generates the one Unit composed of 1, 2, 5, and 8. If k-way merge sorter tree executes this process, the tree can merge k Units and generate a larger Unit.

However, if the number of the Units to be merged is more than k, the data sequence passed through the tree is not fully sorted yet. If so, the tree uses a buffer like off-chip memory in order to store the

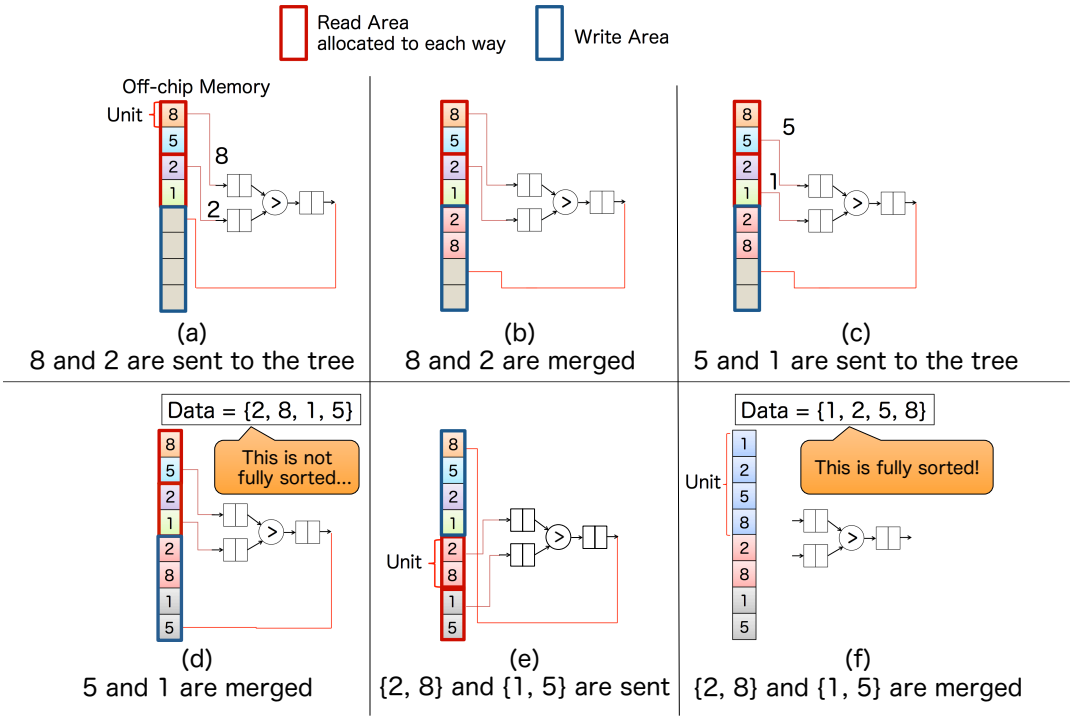


Figure 4.4 The merge sorter tree and off-chip memory. The tree sorts the initial data sequence $\{8, 5, 2, 1\}$ by using the memory.

data sequence. I explain this using a simple example shown in Figure 4.4.

In Figure 4.4, the off-chip memory is divided into two areas, which are Read Area and Write Area. Read Area is used to hold the data sequence that is sent to the merge sorter tree. If k-way merge sorter tree is used, this area is further divided into k areas and each divided area is allocated to each way. Here k is 2, therefore Read Area is divided into two areas. Write Area is used to buffer the data sequence emitted from the tree.

In Figure 4.4 (a), Read Area has an unsorted data sequence, which is $\{8, 5, 2, 1\}$. This data sequence consists of 4 Units shown in Figure 4.4 (a). As mentioned above, Read Area is divided into two areas. Hence, one has $\{8, 5\}$, the other has $\{2, 1\}$. First, 8 and 2 are sent to each way. These elements are merged into one Unit, and then the Unit $\{2, 8\}$ is written into the head of the Write Area, as shown in Figure 4.4 (b).

After that, in Figure 4.4 (c), 5 and 1 are sent and merged. Then, as shown in Figure 4.4 (d), the Unit $\{1, 5\}$ is stored in Write area. This means that the entire data sequence in Read Area is passed through the merge sorter tree and stored in Write Area. As shown in Figure 4.4 (d), the data sequence $\{2, 8, 1, 5\}$ in Write Area is not fully sorted, and has to be passed through the tree again. That is why if the number of the Units to be merged is more than k, the data sequence passed through the tree is not fully sorted yet.

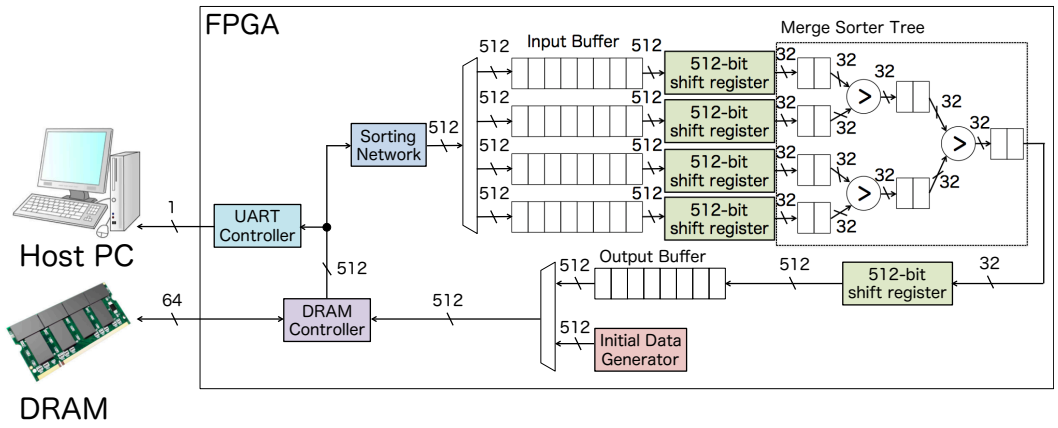


Figure 4.5 Data path of the baseline sorting accelerator of FACE

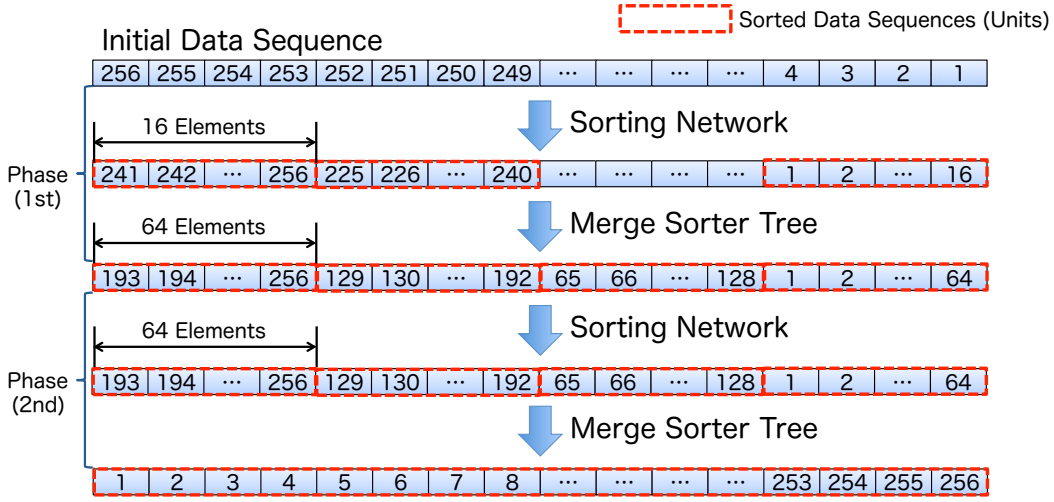


Figure 4.6 Example: sorting 256 elements from 256 to 1

In Figure 4.4 (e) the 2 Units, $\{2, 8\}$ and $\{1, 5\}$, are sent to the ways and are merged in the tree. The operation of the tree is same as Figure 4.3. As shown in Figure 4.4 (f), the data sequence emitted from the tree is fully sorted, which is $\{1, 2, 5, 8\}$.

4.3 Proposed Sorting Accelerator

4.3.1 Data Path

Figure 4.5 shows a data path of the baseline sorting accelerator of FACE. I implement it on an FPGA, and verify that it accurately works by using a host PC. I design two modules of Initial Data Generator and UART Controller for the verification and the performance evaluation.

I explain how the hardware sorts data sequences using Figure 4.6. For simplicity, the initial data

sequence of 256 elements is a reverse-order data sequence from 256 to 1.

A data sequence for sorting is generated from Initial Data Generator. The module can support three data-generation types, which are a random data sequence using Xorshift [98], a sorted data sequence, and a reverse-order sorted data sequence. The data type is 32-bits integer. At first, a data sequence emitted from the module is stored in the external memory via DRAM Controller.

After that initialization, the data sequence in the external memory is loaded and is sent to Sorting Network via DRAM Controller. Sorting Network is Batcher's even-odd merge sort network [96] with 16-inputs and 16-outputs. This means that this network can sort 16 elements. Thus, the initial data sequence turns into 16 sorted data sequences by passed through this network. In other words, the number of Units is 16 and one Unit has 16 elements as shown in Figure 4.6.

The data sequence passed through Sorting Network is stored in Input Buffer that consists of FIFO. The stored elements must already be sorted. The data sequence stored in Input Buffer is sent to 512-bit shift register. This shift register breaks down a 512-bits data into 16 elements, and then sends them to Merge Sorter Tree.

For simplicity, I draw 4-way merge sorter tree in Figure 4.5. By comparing elements at every sorter cell and storing outputs in the FIFOs in each cycle, the merged data sequence is emitted from the root of the merge sorter tree. After passed through the tree, the data sequence composed of 16 Units turns into 4 Units, each of which has 64 elements.

The data sequence emitted from the root of the merge sorter tree is sent to 512-bit shift register, and then is packed into a 512-bits data. After packed, the data sequence is sent to Output Buffer, and then is stored in the external memory via DRAM Controller. However, the data sequence is not fully sorted yet. Thus, it is read from the memory, and then is sent to Sorting Network again. In this time, the network is a mere data path because portions of the data sequence are already sorted.

The data sequence passed through the network is sent to the tree. In the tree, 4 Units are merged into one Unit and then elements of the Unit are emitted from the root of the tree cycle by cycle. This means that all of the emitted elements are fully sorted. The data sequence is stored in the external memory via DRAM Controller, after passed through 512-bit shift register and Output Buffer.

To verify the result, the fully sorted data sequence in the external memory is loaded and is sent to UART Controller via DRAM Controller. UART Controller sends it to the host PC by serial communication. The transferred data sequence is checked, using typical sorting software.

By passing the data sequence through Sorting Network and Merge Sorter Tree twice in this case, it can be fully sorted. I define the process that passes the data sequence through Sorting Network and Merge Sorter Tree as **Phase**. The number of required Phases for fully sorting the data sequence is given by $\log_{\#\,of\,ways} \frac{\#\,of\,elements}{16}$ where 16 is the number of sorted elements at Sorting Network in the first Phase. For instance, in Figure 4.6 the number of required Phases is 2, because the number of ways

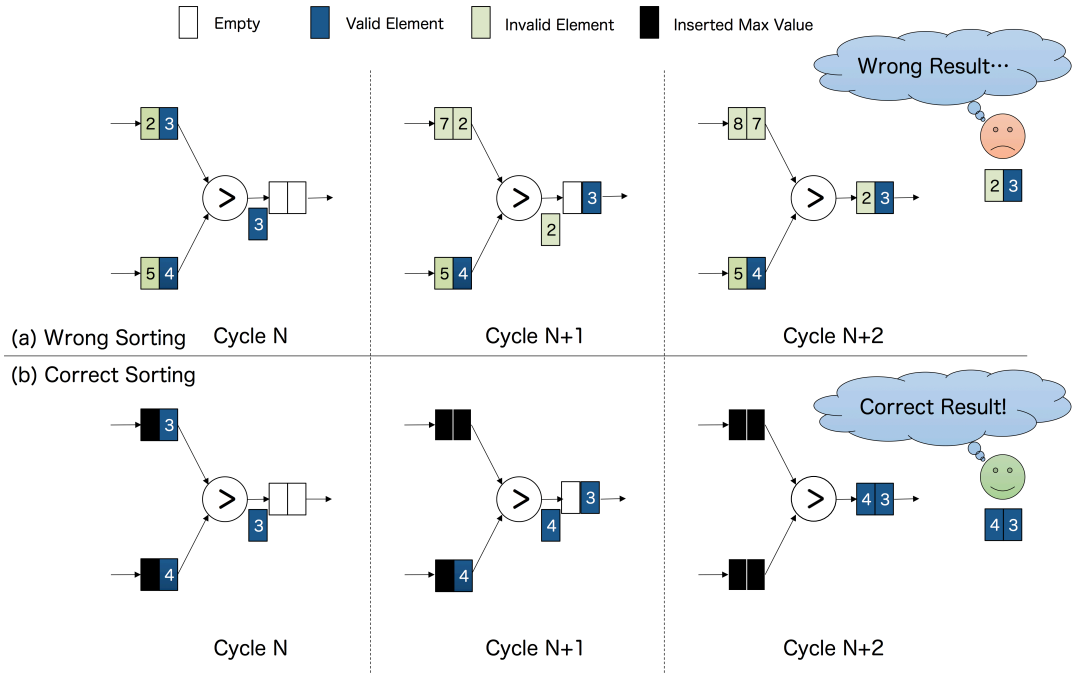


Figure 4.7 Wrong sorting and correct sorting

and elements to be sorted is 4 and 256 respectively.

4.3.2 Control Logic

I describe a control mechanism to sort data sequences.

Depending on the number of ways and elements to be sorted, the number of required Phases is decided. When Units are merged in the merge sorter tree, each Unit needs to be treated separately. If not be separated, that sorting cannot be executed successfully, because invalid elements are mixed into Units.

In Figure 4.7 (a), this example is demonstrated. Figure 4.7 shows a wrong case (a) and a correct case (b) of merging Units (i.e. merging 3 and 4). I define Valid elements as the elements which should be merged into one Unit in the merge sorter tree (i.e. 3 and 4). At Cycle N+1, 4 should be emitted from the sorter cell, because 4 is a Valid element. However, in (a) 2 is emitted, which is an Invalid element, hence this sorting cannot be done successfully.

To address this problem, we have proposed that the maximum value, which depends on the bit width of elements, is inserted after Valid elements [99]. Figure 4.7 (b) shows how this method is applied. By doing so, this sorting can be executed successfully, because 4 is emitted from the sorter cell at Cycle N+1.

To realize this (Figure 4.7 (b)), a circuit that generates the maximum value to separate Units is implemented in Input Buffer as shown in Figure 4.8. Each Input Buffer has a counter, which counts the

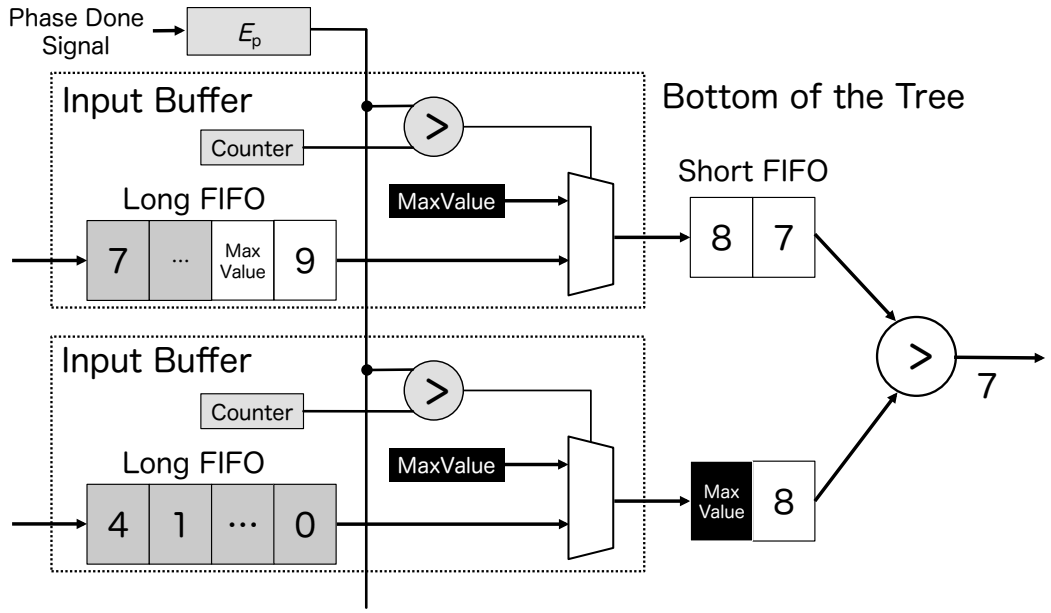


Figure 4.8 Two input buffers and one sorter cell

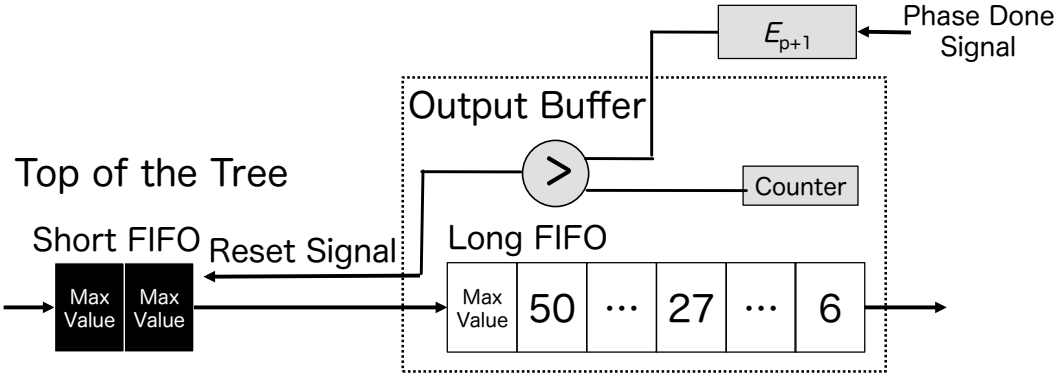


Figure 4.9 How to generate reset signal from output buffer

number of emitted elements from this buffer. I define the number of elements in each Unit in Phase p as E_p. When the counter value exceeds E_p, the maximum value is emitted from this buffer, and this buffer keeps holding subsequent Units.

Output Buffer also has a counter, which counts the number of stored elements in Output Buffer as shown in Figure 4.9. When the counter value exceeds E_{p+1}, all FIFOs in the merge sorter tree, the counter of Input Buffer, and the counter of Output Buffer are reset. After this, the tree begins to merge subsequent Units. Exceeding E_{p+1} means that all elements of a Unit, which is generated in the merge sorter tree, are stored in Output Buffer.

Due to this mechanism, each Unit is treated separately and it can be guaranteed that elements are sorted successfully. In Figure 4.6, in the 1st Phase E_p is 16, E_{p+1} is 64, and in the 2nd Phase E_p is 64,

E_{p+1} is 256.

4.3.3 Performance Model

I analyze theoretical performance of the proposed sorting accelerator. To analyze it, I assume that the DRAM bandwidth is infinity and the latency is 1 cycle. Depending on the number of ways and elements to be sorted, I calculate the number of required cycles to fully sort them. This can be calculated by summation of the number of cycles in each Phase.

As described in Section 4.3.1 and Section 4.3.2, multiple Units are merged into one Unit in the merge sorter tree. I call this process **Iteration**. In Figure 4.6, 16 Units generated from the sorting network turn into 4 Units by passed through the merge sorter tree in the 1st Phase. This means that four Iterations are executed in the 1st Phase. In other words, the number of Iterations in the 1st Phase is 4, and that is 1 in 2nd Phase. I define the number of Iterations, ways, and elements to be sorted by the proposed system as I, k, and N respectively. In nth Phase, the number of Iterations for nth Phase is given by

$$I_n = \frac{N}{16k^n} \tag{4.1}$$

where 16 is the number of sorted elements at the sorting network in the 1st Phase.

After Iteration, all FIFOs in the merge sorter tree are reset (Section 4.3.2). Therefore, a few cycles overhead exists between each Iteration. This overhead OH_{iter} is given by

$$OH_{iter} = \log_2 k + 1 \tag{4.2}$$

and OH_{iter} is equal to the number of stages of the merge sorter tree.

The beginning of each Phase also has an overhead. The merge sorter tree cannot sort data sequences unless elements are stored in all of the leftmost FIFOs. Elements have to be stored in these FIFOs immediately, because they are empty at the beginning of each Phase. In other words, this is the overhead. I define the number of required cycles for this buffering as α, and then the overhead OH_{phase} is given by

$$OH_{phase} = k\alpha \tag{4.3}$$

where α is tens of cycles at most.

I define the number of required cycles for nth Phase as C_n. This is given by the following formula.

$$C_n = N + I_n \times OH_{iter} + OH_{phase} \tag{4.4}$$

I explain this formula in three parts. First, the throughput of the merge sorter tree is one element per cycle. Thus, it takes N cycles to emit all elements from the merge sorter tree. Second, in nth Phase, the number of Iterations is I_n. Thus, the number of cycles for the overhead of all Iterations is $I_n \times OH_{iter}$, because OH_{iter} cycles overhead exists between each Iteration. Third, the beginning of each Phase

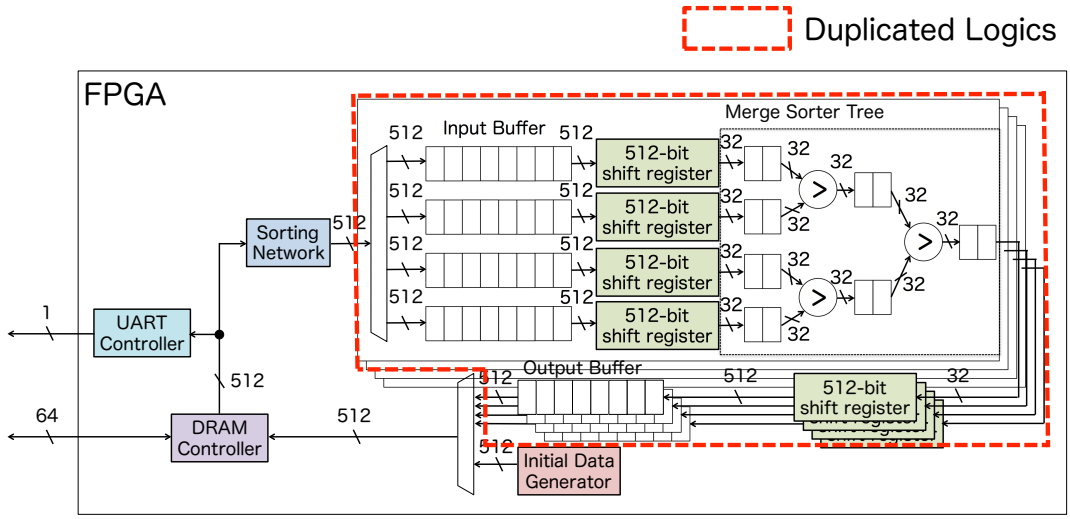


Figure 4.10 Data path of the proposed sorting accelerator with the duplicated merge sorter trees

has OH_{phase} cycles overhead as mentioned. Consequently, C_n can be calculated by summation of the number of these cycles.

Hence, C_{fully}, which is the number of required cycles to fully sort the data sequence, is given by

$$C_{fully} = \sum_{i=1}^{n} C_i \tag{4.5}$$

where n is the number of required Phases. As described in Section 4.3.1, the number of required Phases to fully sort the data sequence is $\log_k \frac{N}{16}$. In other words, C_{fully} can be also given by the following formula.

$$C_{fully} = \sum_{i=1}^{\log_k \frac{N}{16}} \{N + \frac{N}{16k^i}(\log_2 k + 1) + k\alpha\} \tag{4.6}$$

The sorting process time can be estimated by means of dividing C_{fully} by the operating frequency.

4.3.4 Improvement by Duplication of the Merge Sorter Tree

I describe how to improve the proposed sorting accelerator. One of the approaches to achieve this is to improve the sorting logic throughput. I propose duplication of the merge sorter tree. This approach is simple, yet effective for the throughput improvement.

Figure 4.10 shows a data path of the sorting accelerator with the duplicated merge sorter trees. The duplicated trees work in parallel. Thus, the more the tree is duplicated, the higher the sorting logic throughput is.

By taking advantage of the performance model described in Section 4.3.3, it is possible to analyze theoretical performance of the sorting accelerator with the duplicated trees. If the number of duplicated

trees is defined as P, the number of required cycles for nth Phase is given by $\frac{C_n}{P}$. This is because the duplicated trees sort data sequences in parallel. In the last Phase, the parallelism benefit cannot be obtained. Thus, C_{last}, which is the number of required cycles for the last Phase, is given by

$$C_{last} = N + 1 \times OH_{iter} + OH_{phase} \tag{4.7}$$

where the number of Iterations for the last Phase is definitely one. Therefore C_{fully_dup}, which is the number of required cycles to fully sort the data sequence by the sorting accelerator with the duplicated trees, is given by the following formula.

$$C_{fully\_dup} = C_{last} + \sum_{i=1}^{(\log_k \frac{N}{16})-1} \frac{C_i}{P} \tag{4.8}$$

Hence, the sorting process time is estimated, depending on the number of ways, duplicated trees, and elements to be sorted by the sorting accelerator. Besides, designers can implement a sorting accelerator composed of required hardware resources, by means of tuning the number of ways and duplicated trees.

As mentioned before, the higher the sorting logic throughput is, the higher performance the accelerator achieves. Using the operating frequency F and P, the sorting logic throughput is given by $F \times 4Bytes \times P \times 2$ where $F \times 4Bytes$ depends on the throughput of the merge sorter tree. The tree operates at F, and emits one element per cycle from the root, whose data size is 4Bytes. And the constant 2 comes from DRAM read and write.

However, as the sorting logic throughput is higher, the sorting performance becomes sensitive to the memory bandwidth. This means that the memory bandwidth becomes the performance bottleneck. Therefore, it is truly important to consider approaches which can improve the memory bandwidth utilization while keeping the operating frequency high. I present an effective way to realize this in the next section.

4.4 Data Compression for the Sorting Accelerator

4.4.1 Algorithm

To mitigate the bandwidth limitation of accessing the off-chip memory, I adopt *data compression*. Data compression has been successfully adopted in a number of different contexts in modern computer systems as a way to conserve storage capacity and/or data bandwidth (e.g., downloading compressed files over the Internet or compressing off-chip memory) for several decades. Many data compression algorithms are proposed in prior works, but it is necessary to decide the most appropriate algorithm according to data types, applications, and hardware.

In general, data compression algorithms take advantage of redundancy in the data used by applications [100, 101, 102]. However the data handled in sorting is generally random, and there is little

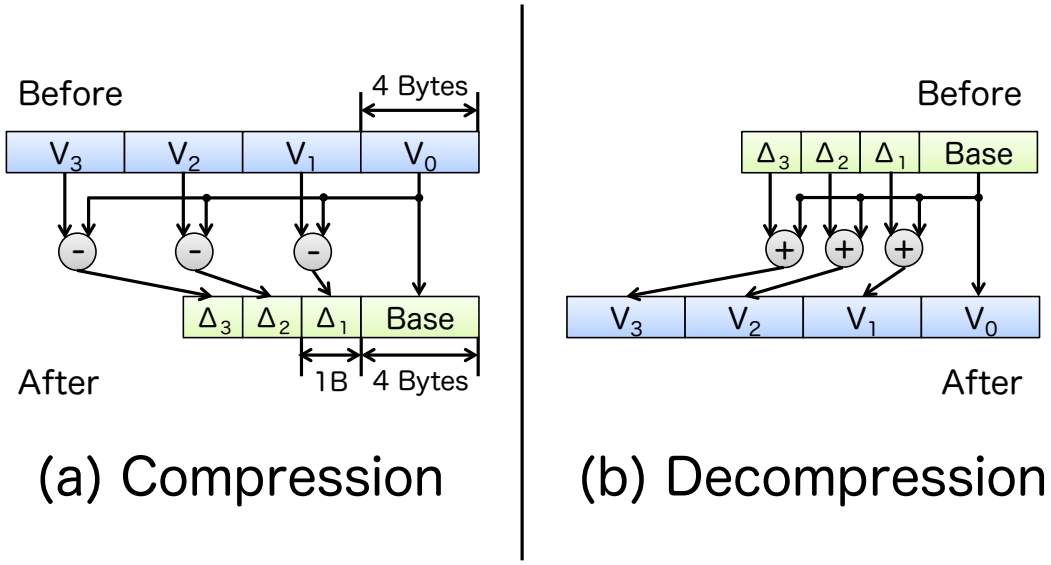


Figure 4.11 Example of the compressor and decompressor method that is described in [91]. In this example, 4Bytes values are compressed into a 4Bytes base and an array of 1Byte Δ.

redundancy in the data used by the application. Therefore such algorithms like [100, 101, 102] are not effective against sorting. Well then, which algorithm is promising for the application?

I focus on a data compression algorithm based on [91], which uses the relative difference between values of continuous locations. As sorting is proceeded, the relative difference between them becomes smaller. That is why the algorithm is quite suitable for sorting. The data compressed by the algorithm is represented in a compact form using a common *base* value and an array of relative differences (*deltas*).

Figure 4.11 shows the example diagram of the compression and decompression method described in [91]. As shown in Figure 4.11, the compression and decompression method can be implemented by a simple vector subtraction and addition. In Figure 4.11 (a), the compressed data is represented in Base V_0 and the array of $\Delta_1 \sim \Delta_3$, using 7Bytes instead of 16Bytes. This results in saving 9Bytes of the originally used space. The compressed data can be easily decompressed by the addition of each delta to Base shown in Figure 4.11 (b).

4.4.2 Adoption of the Data Compression against the Proposed Sorting Accelerator

Figure 4.12 shows the adoption of the data compression against the proposed sorting accelerator. As described in Section 4.3.1, 512-bit shift register packs 32-bits elements emitted from the root of the merge sorter tree into a 512-bits data. In Figure 4.12 if two compressible 512-bits data are successive, the two data are packed into a 512-bits data. For instance, if all 512-bits data packed by the shift

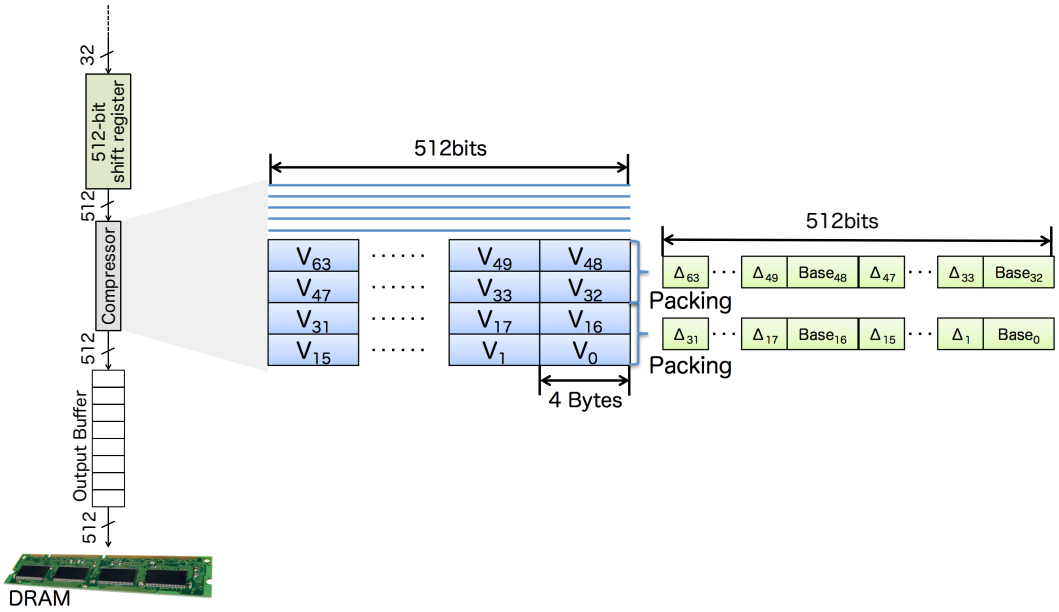


Figure 4.12 Adoption of the data compression

register are compressible, the data amount transferred to the external memory is a half of the original one. This means that it is possible to theoretically obtain double performance if the memory bandwidth is the performance bottleneck. In other words, it is possible to estimate the performance improvement ratio by calculation of the compression ratio against all 512-bits data packed by the shift register under such situation. I define the process to pack two compressible 512-bits data into a 512-bits data as 2x compression if the two data are successive. The 2x compression is given by

$$2xCompRatio = 1.0 + (2.0 - 1.0) \times incidence \tag{4.9}$$

where ”incidence” is the occurrence rate of consecutive two compressible 512-bits data against all 512-bits data packed by the shift register.

4.4.3 Data Path

Figure 4.13 shows the data path of the proposed sorting accelerator with the compressor and decompressor. The compressor packs two compressible 512-bits data into a 512-bits data like Figure 4.14, and the decompressor unpacks compressed data read from the external memory.

The encoding format for the 2x compression consists of four parts, which are Base, Compressed, Void, and Flag shown in Figure 4.14. Base and Compressed represent a base value and an array of deltas. The region of Base and Compressed stands for a compressed original data emitted from 512-bit shift register shown in Figure 4.12. The data emitted from the shift register consists of 16 sorted elements. For instance, in Figure 4.12 the 16 elements from V_0 to V_{15} are sorted, and V_0 is the smallest

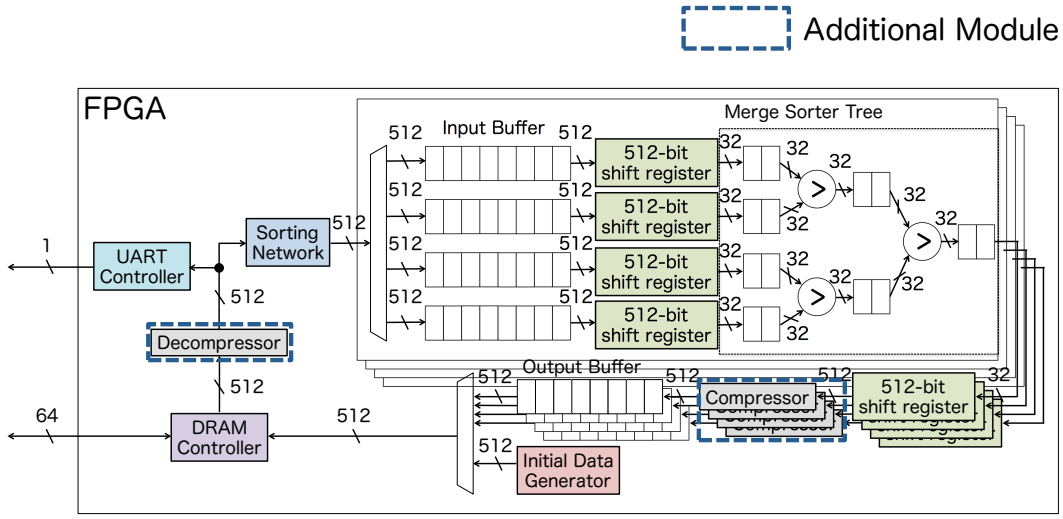


Figure 4.13 Data path of the proposed sorting accelerator with the compressor and decompressor

Name	*Description*
Base	A base value
Compressed	An array of 15 deltas. The delta size is 13bits.
Void	Unused space
Flag	The marker to check whether or not this data is compressed. The marker is 0x0000_0000_1

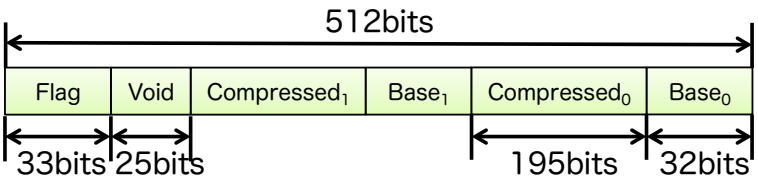


Figure 4.14 The encoding format for the 2x compression

and V_{15} is the largest in the 16 elements. The most simple way is to choose the smallest one as Base and is that the other 15 elements are converted into the 15 13-bits deltas if Δ_{ls}, which is a difference between the largest and the smallest, $\leq$ 0x1fff. And then, if a subsequent 512-bits data is also compressible, the compressor converts the two original data into a 512-bits data shown in Figure 4.14. On this occasion, Flag is set to 0x0000_0000_1 in order to identify that the 512-bits data is encoded. If the compressor cannot convert two original data into a 512-bits data, this module outputs the two original data one by one. The region corresponding to Flag of the two data is never 0x0000_0000_1, because the elements

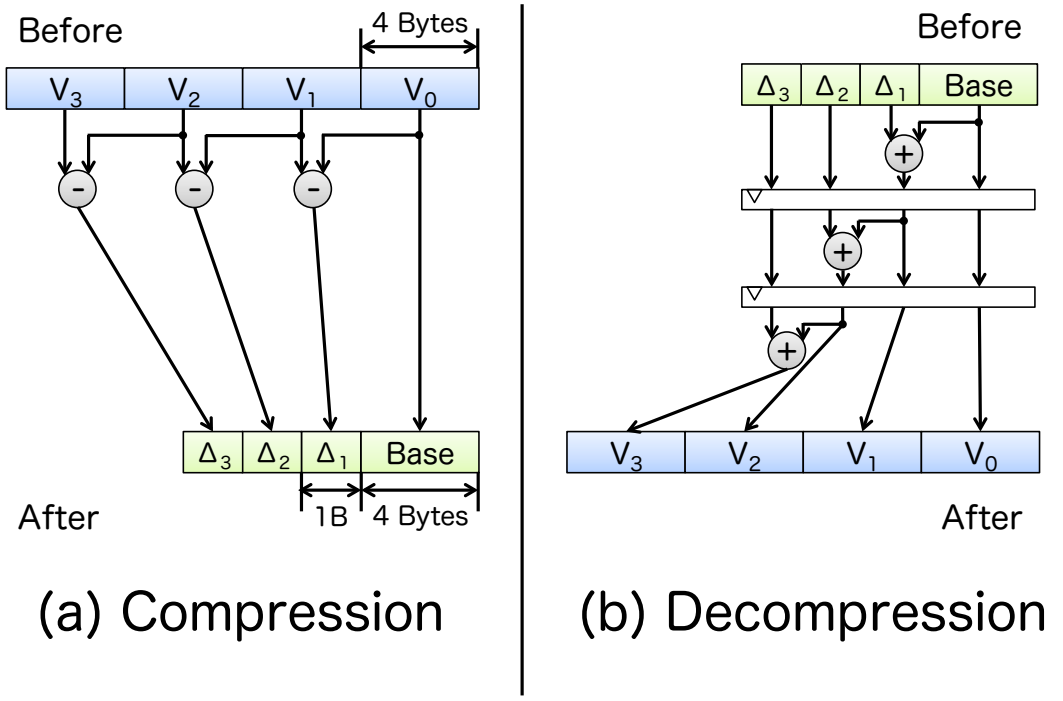


Figure 4.15 Modified compression and decompression designs based on the prior work. In this diagram, 4Bytes values are compressed into a 4Bytes base and an array of 1Byte Δ.

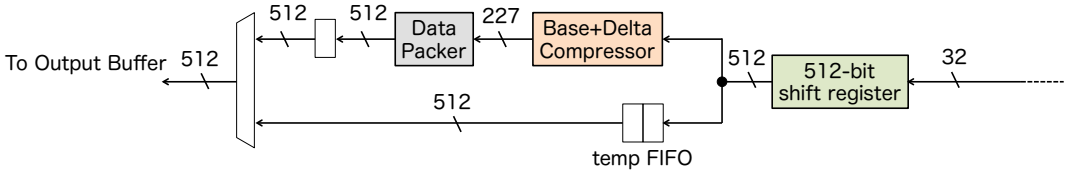


Figure 4.16 Data path of the compressor

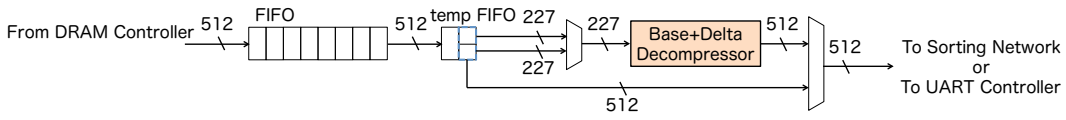


Figure 4.17 Data path of the decompressor

in the two data are sorted in descending order from the Most Significant Bit (MSB). Due to this, the decompressor can identify whether or not the data read from the external memory is encoded by checking the region. I define the remained part in the encoding format as Void that is unused space.

The simple way to generate Compressed cannot efficiently work if a gap between the largest and the smallest is too large. To address this problem, I propose modified compression and decompression designs based on the prior approach shown in Figure 4.11. Unlike Figure 4.11 (a) the compression design subtracts the neighbor value from each value in order to generate the array of $\Delta_1 \sim \Delta_3$. Due to

this, in the 16 elements from V_0 to V_{15} shown in Figure 4.12, V_0 is Base and the other 15 elements are converted into the 15 13-bits deltas if all deltas $\leq$ 0x1fff. Because each delta is smaller than Δ_{ls}, this approach has more likelihood to generate Compressed than the simple way. In the decompression, the pipelined addition is executed unlike the simple vector addition shown in Figure 4.11 (b). Although the decompression design can be implemented as a purely combinational circuit, this probably causes the operating frequency reduction due to the large delay. That is why I choose the pipelined design like the sorting network described in Section 4.2.1.

Figure 4.16 shows the data path of the compressor. This module consists of three components, which are Base+Delta Compressor, Data Packer, and temp FIFO. Base+Delta Compressor executes a simple vector subtraction to compress a 512-bit data emitted from the 512-bit shift register. At the same time, the original data is stored in temp FIFO. If the original data is not compressible, temp FIFO outputs all stored data immediately. Data Packer generates a formatted 512-bits data shown in Figure 4.14 if two compressible 512-bits data are successive. On this occasion, temp FIFO is reset.

Figure 4.17 shows the data path of the decompressor. This module has three components, which are FIFO, temp FIFO, and Base+Delta Decompressor. The FIFO consists of internal memory resources (hard macros) of the FPGA, and the data read from the external memory is stored in the component. The stored data in the FIFO is sent to temp FIFO, and then this component keeps holding the data unless the dequeue signal is asserted. If the data is compressed one, the data splits into two parts, and then Base+Delta Decompressor picks up and decompresses the parts one by one, at the same time the dequeue signal is asserted. If not, the data is sent to Sorting Network or UART Controller directly, and the dequeue is done simultaneously. As mentioned before, the decompressor can identify whether or not the data is compressed by checking Flag shown in Figure 4.14.

4.4.4 Control Logic

As described in Section 4.2.2, the data emitted from the merge sorter tree is sequentially written into the head of the Write Area like Figure 4.18 (a). The figure shows that the data emitted from 4-way merge sorter tree is written into the external memory via DRAM Controller. The buffered data is sent to each way of the tree through Sorting Network in the next Phase. Without the data compression in Figure 4.18 (a), the data sent to each way in the next Phase is correctly written into each region of the memory by tuning data size of each way and grain size of data written into the memory. With the data compression in Figure 4.18 (b), simple sequential write can mixes each region data that should be sent to each way, because each region data can be non-uniform due to the data compression. In that case, sorting cannot be accurately performed because incorrect data is sent to each way.

To address this problem, I present a mechanism named *Throttling*, which tunes grain size of the data written into the external memory. Figure 4.19 shows the overview of Throttling. DRAM Controller

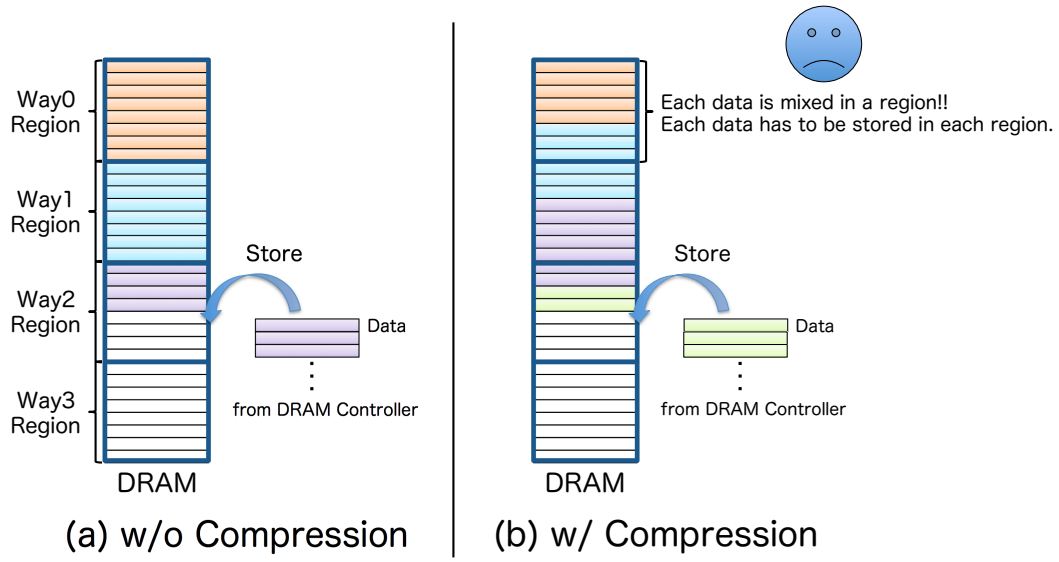


Figure 4.18 The data emitted from the merge sorter tree is sequentially written into the head of the Write Area, without (a) and with (b) the data compression.

writes the data emitted from the tree into the head of the Write Area. At the last of completion of writing data sent to a way in the next Phase, DRAM Controller gradually throttles the grain size. By doing this, emitted data is correctly written into a corresponding region without mixing each region data. The flag to alert that writing data almost finishes is asserted when writing data address crosses Threshold of a region.

After writing data into a corresponding region finishes, DRAM Controller sets the writing address as the head of the next region and then sequentially writes the data emitted from the tree. By repeating this process, all of emitted data is correctly written into the external memory. When setting the writing address as the head of the next region, the writing address is preserved. The address is used as a pointer to identify how much data each way should read in the next Phase.

4.5 Evaluation

4.5.1 Implementation

As a platform for the proposed FPGA accelerator, I use the Xilinx Virtex-7 FPGA VC707 evaluation kit [103]. This kit originally has the Virtex-7 XC7VX485T and 1GB DDR3 SO-DIMM (800MHz/1600Mbps) memory, but I replace this memory with 4GB DDR3 SO-DIMM memory in order to sort larger data sequences.

The sorting logic is implemented in Verilog HDL. To implement DRAM Controller, I use an IP core provided by Xilinx [104]. As a synthesis tool, I use Vivado 2014.4 [30]. In the accelerator without the

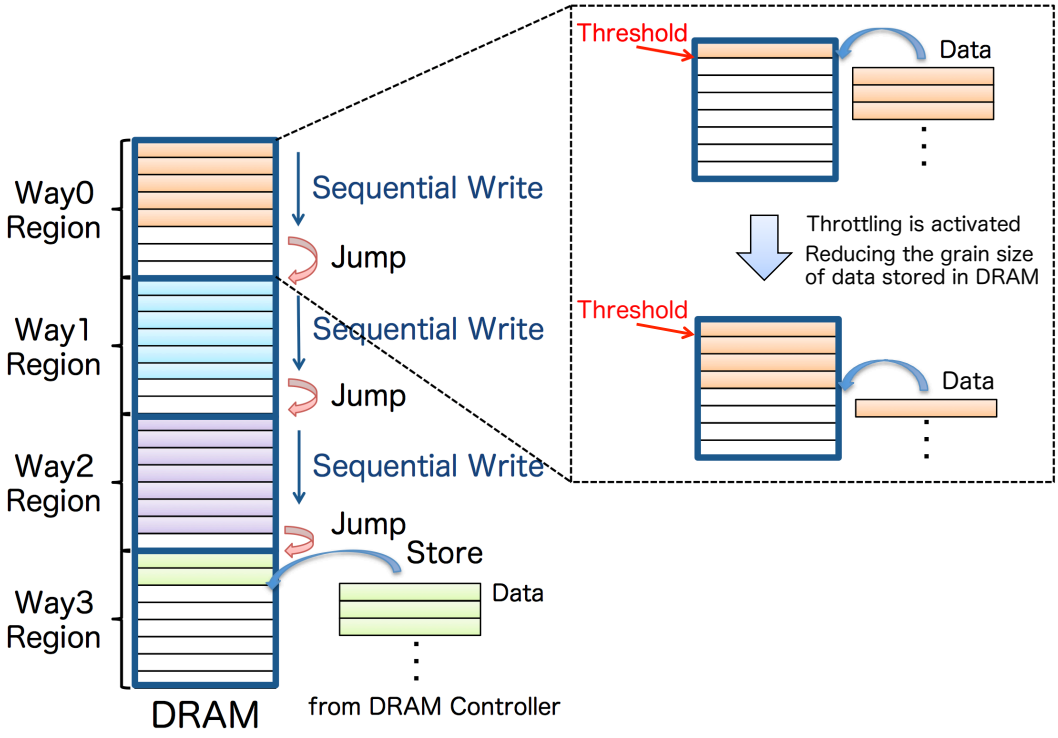


Figure 4.19 Writing the data emitted from the merge sorter tree by Throttling

data compression, I set the synthesis options as default, and set the synthesis options as performance-specific when the accelerator uses the data compression. The placed and routed logic meets all timing constraints. All implemented logics on the FPGA operate at 200MHz and the memory bus operates at 800MHz. Consequently, between the FPGA and the DRAM, the maximum data-transfer speed is 12.8GB/s.

4.5.2 Memory Bandwidth

As mentioned above, in order to achieve high sorting performance, it is necessary to maintain memory bandwidth that can tolerate the sorting logic throughput. The memory bandwidth is tunable by changing grain size of the read and written data from/into the external memory. I evaluate how much memory bandwidth the hardware platform for the sorting accelerator can ensure.

Figure 4.20 shows the memory bandwidth when randomly reading and writing 4GB data. The x-axis shows the grain size of the read and written data from/into the external memory. As shown in Figure 4.20, the larger the grain size is, the larger the memory bandwidth of both cases is. However, as the grain size is larger, the FIFO depth to receive and send data from/to the memory is also larger. This leads to the increase in the hardware resource usage. Therefore, I consider the balance between the memory bandwidth and the hardware resource usage, and set DRAM Write per 8k Bytes and DRAM

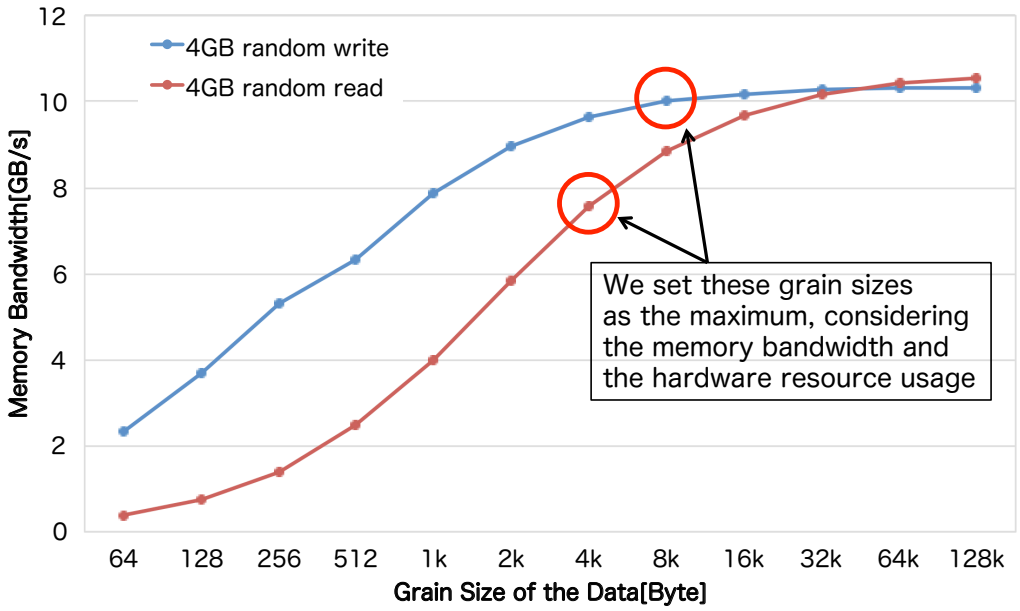


Figure 4.20 The memory bandwidth when randomly reading and writing 4GB data, depending on grain size of the read and written data

read per 4k Bytes as the maximum.

4.5.3 Sorting Performance without Data Compression

I compare the sorting performance of the proposed hardware with Intel Core i7-4770 operating at 3.4GHz. The data-sequence size is 256M elements, whose data type is 32-bits integer. I use merge sort and quick sort for the software running on Intel Core i7-4770, implement them in C language, and compile them with gcc 4.8.2 (-O3 optimization). These two applications are executed as single thread of Intel Core i7-4770. For measurement of the sorting process time, in the case of the sorting accelerator I get sorting execution cycles stored in the hardware counter and calculate the time, and I use gettimeofday in case of the applications.

Figure 4.21 shows the sorting performance comparison between the software and the proposed sorting accelerator. In Figure 4.21, 8-way represents the sorting accelerator with 8-way merge sorter tree and 8-way/2-parallel represents the hardware with two 8-way merge sorter trees. xorshift, sorted, and reverse represent that the initial data-sequence types are a random data sequence, a sorted data sequence, and a reverse-order sorted data sequence respectively. Moreover, Estimated stands for the theoretical performance obtained from the performance model described in Section 4.3.3 and Section 4.3.4.

In xorshift, the 4-way performance is 2.03x and 1.61x, compared with merge sort and quick sort

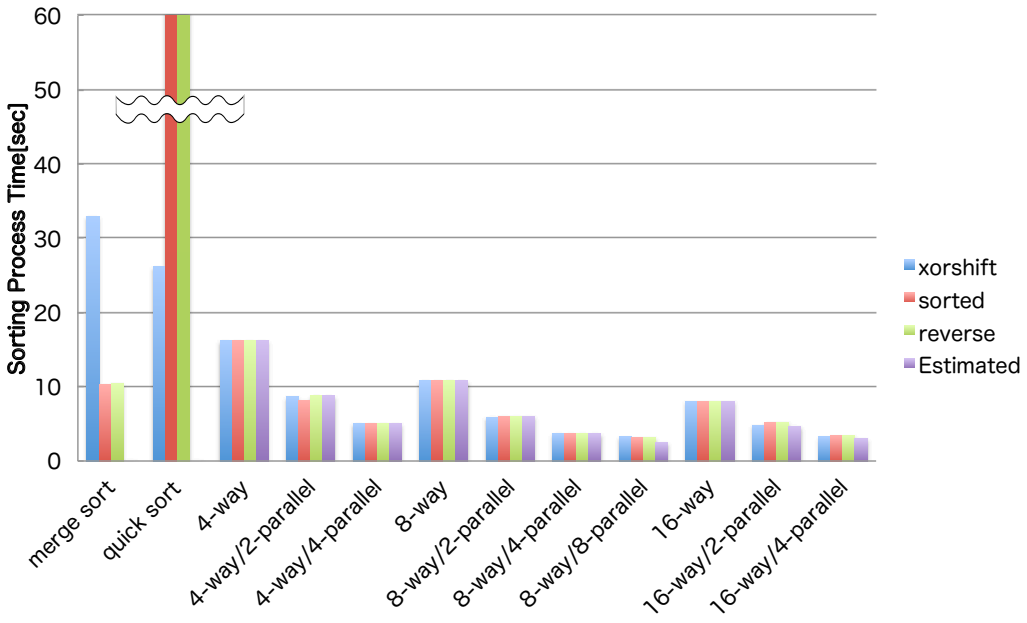


Figure 4.21 Sorting performance comparison between the software and the proposed sorting accelerator

respectively. The 8-way performance is 3.05x and 2.42x; the 16-way performance is 4.07x and 3.24x than the two applications. This shows that the more the number of ways is increased, the higher the sorting performance is. This is because the number of required Phases to fully sort the data sequence is decreased since the more the number of ways is increased, the more the number of elements to be sorted in the merge sorter tree is increased.

Besides, the more the merge sorter tree is duplicated, the higher the sorting performance is. For instance, the 4-way/2-parallel performance is 1.88x and the 4-way/4-parallel performance is 3.21x by compared with 4-way. This is because the duplicated trees sort a data sequence in parallel. In this chapter, 8-way/8-parallel results in the highest performance that is 10.06x and 8.01x compared with merge sort and quick sort respectively.

Although the increase in the number of duplicated trees leads to higher sorting performance, the required memory bandwidth is increased. In other words, to improve the sorting logic throughput, the required memory bandwidth becomes larger. Using P, the sorting logic throughput is given by $200MHz \times 4Bytes \times P \times 2 = 1.6PGB/s$, which is described in Section 4.3.4,

If the merge sorter tree is not duplicated (4-way, 8-way, and 16-way), the sorting performance is equal to the estimated one, because the sorting performance is insensitive to the memory bandwidth. In this case, the sorting logic throughput is $200MHz \times 4Bytes \times 1 \times 2 = 1.6GB/s$. The memory bandwidth utilization of this hardware is 12.5% of the maximum memory bandwidth. This means that there is

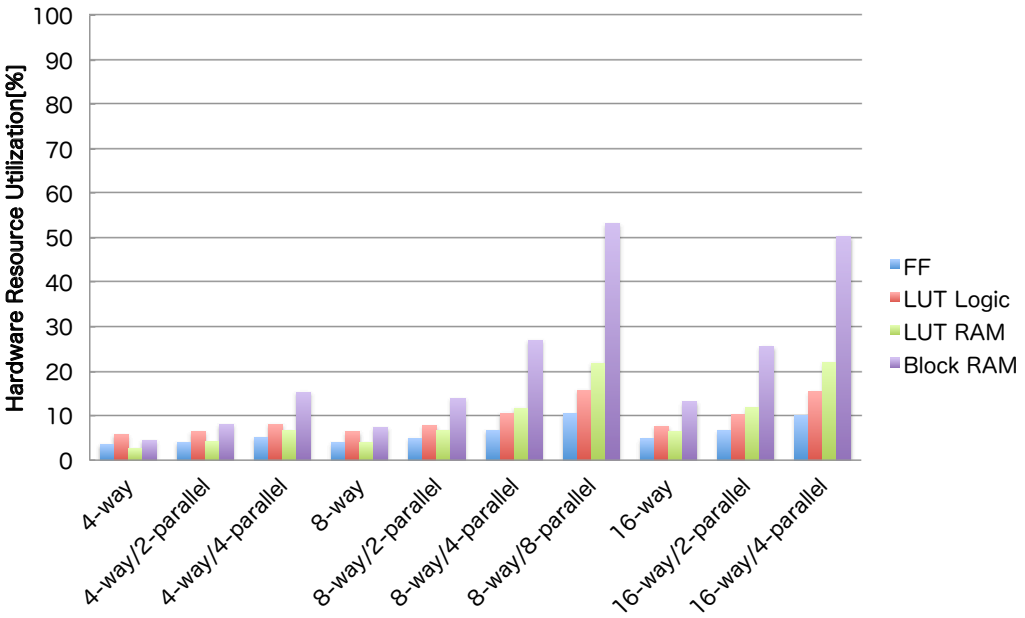


Figure 4.22 Hardware resource usage of the proposed sorting accelerator

sufficient margin in the memory bandwidth, and the sorting logic operates with an efficiency of 100%.

However, the more the merge sorter tree is duplicated, the more the memory bandwidth utilization is close to 100%. This means the sorting performance becomes sensitive to the memory bandwidth. In particular, the sorting logic throughput of 8-way/8-parallel is $200MHz \times 4Bytes \times 8 \times 2 = 12.8\,GB/s$, which is equal to the maximum memory bandwidth. However, the practical memory bandwidth is lower than the maximum due to the overhead of DRAM Controller. When this configuration, I set the maximum grain sizes that represent DRAM Write per 8k Bytes and DRAM read per 4k Bytes shown in Figure 4.20, in order to sustain the highest memory bandwidth. The harmonic mean of the read and write memory bandwidth is 8.62GB/s, which means that the average memory bandwidth is insufficient to tolerate the sorting logic throughput. That is why the sorting performance degradation occurs, which is 70.2% of the estimated one, and the ratio is almost same as that of the average memory bandwidth to the sorting logic throughput.

As shown in Figure 4.21, the performance of xorshift, sorted, and reverse of the sorting accelerator are almost same. This means that the sorting accelerator is independent on the data-sequence type. On the other hand, the software considerably depends on it. Especially, the results of sorted and reverse of quick sort clearly show this aspect because of the worst-case complexity of $O(n^2)$.

Figure 4.22 shows the hardware resource usage of the sorting accelerator. In Figure 4.22, FF, LUT Logic, LUT RAM, and Block RAM represent a flip-flop (FF), a lookup table (LUT) for combinational logic, LUT for distributed memory, and an internal memory (hard macro) of the FPGA. As shown in

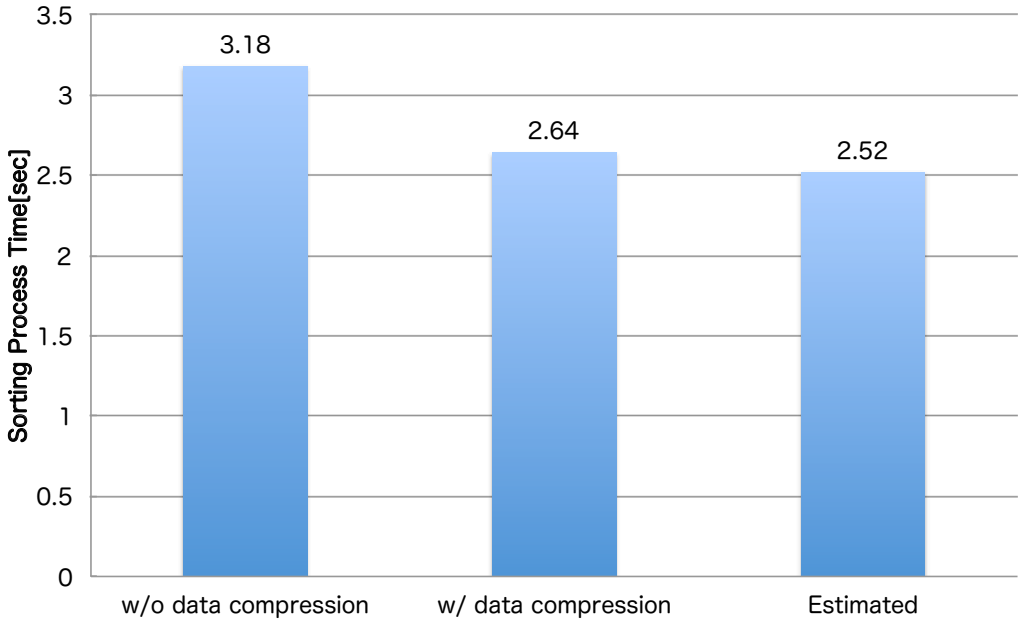


Figure 4.23 The sorting performance of 8-way/8-parallel with and without the data compression mechanism when the initial data-sequence types is a reverse-order sorted data sequence

Figure 4.22, the logic usage except Block RAM is almost within 20%. Because Block RAM is used to implement Input Buffer and Output Buffer, the more the number of ways and duplicated trees is increased, the larger the Block RAM usage is. Also, as the number of duplicated trees is increased, the depth of Input Buffer and Output Buffer is also larger to sustain the required memory bandwidth. Therefore, the Block RAM usage optimization has to carefully consider the balance of the memory bandwidth and the hardware cost.

4.5.4 Sorting Performance with Data Compression

I demonstrate that performance of the sorting accelerator with the data compression. To evaluate how the data compression is effective, I use 8-way/8-parallel suffering from the memory bandwidth bottleneck. 8-way/8-parallel with the data compression operates at 200MHz, and the memory bus operates at 800MHz. The data set is same as Section 4.5.3.

Figure 4.23 shows the sorting performance of 8-way/8-parallel with and without the data compression mechanism when the initial data-sequence types is a reverse-order sorted data sequence. The estimated performance shown in Figure 4.23 is same as 8-way/8-parallel estimated performance shown in Figure 4.21. In this data set, all data is compressed because each delta is very small, which means that Figure 4.23 shows the sorting performance when all data is compressible. While 8-way/8-parallel without the data compression mechanism achieves 74% of the estimated sorting performance, 8-way/8-

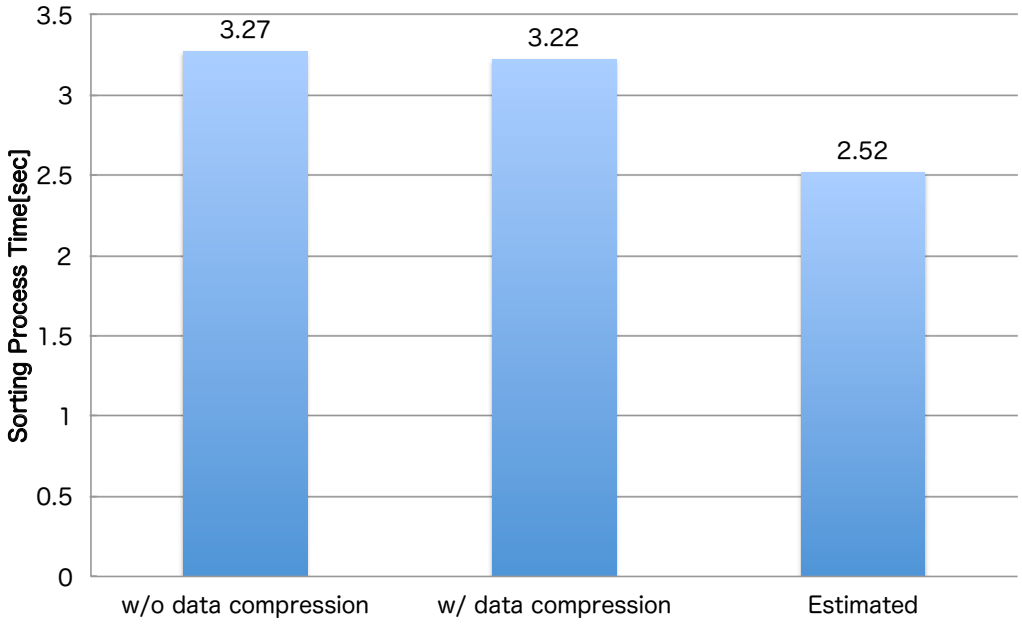


Figure 4.24 The sorting performance of 8-way/8-parallel with and without the data compression mechanism when the initial data-sequence types is a random data sequence using Xorshift

parallel with it does 95% due to alleviation of the memory bandwidth limitation. The small gap between sustained and estimated arises from Throttling overhead, because Throttling gradually reduces the grain size of the data, which leads to the memory bandwidth reduction. Besides, in the first Phase the sorting accelerator reads uncompressed data from the external memory, which is also a part of the gap.

Figure 4.24 shows the sorting performance of 8-way/8-parallel with and without the data compression mechanism when the initial data-sequence types is a random data sequence using Xorshift. With the data compression mechanism, the sorting performance is slightly improved. In order to investigate the reason why the improvement ratio is small, I implement a software simulator to evaluate the data compression ratio in each Phase. Figure 4.25 shows that result. The number of required Phases of 8-way/8-parallel is 8, which is calculated by using the formula described in Section 4.3.1. The compression ratio shown in Figure 4.25 is calculated by using the formula described in Section 4.4.2, for instance if all data is compressible, the compression ratio is 2. Gmean shown in Figure 4.25 represents the geometric mean of all compression ratios from Phase 0 to Phase 7. As shown in Figure 4.25, while no data is compressed in early Phases, the compression ratio is improved as sorting is proceeded. By repeating Phase, each delta becomes smaller and compressible data begins to appear from Phase 4. Gmean is 1.301, and it is clear that in xorshift the performance improvement of 8-way/8-parallel with the data compression is due to the data compression in the latter half of the Phases. In other words, the improvement is not quite because the average compression ratio is low. Besides, Throttling is exe-

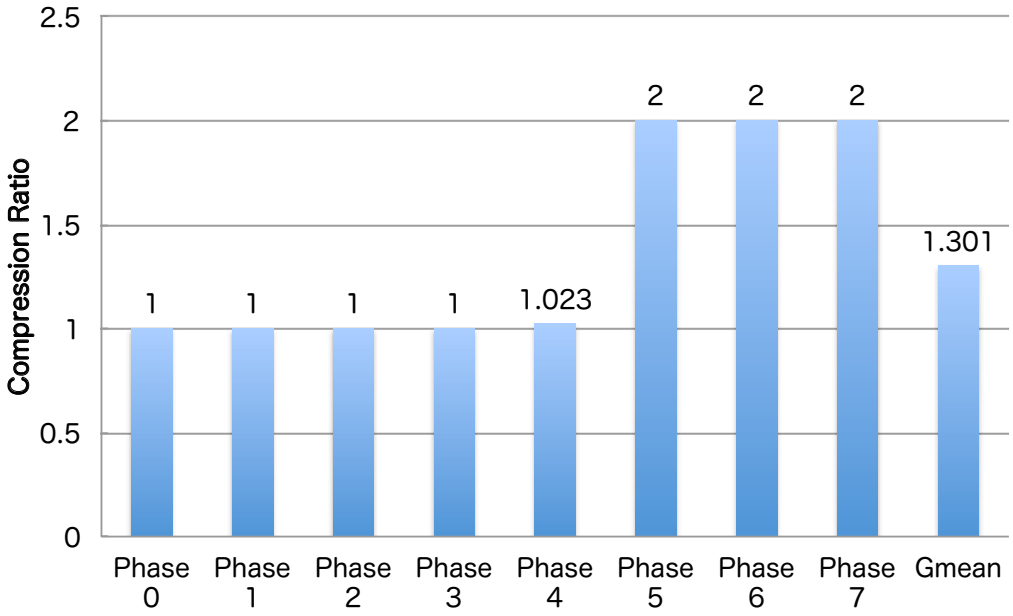


Figure 4.25 The data compression ratio in each Phase of 8-way/8-parallel

cuted in all Phases even if no data is compressed in the early Phases. This overhead also prevents the performance improvement.

Well then, is the data compression mechanism truly effective against a random data sequence with high compression ratio? I investigate it by using data sets with small deltas, which are generated by narrowing values. Figure 4.26 shows that result. The x-axis in Figure 4.26 represents bit width of a value, and the y-axis represents sorting process time and average compression ratio (Gmean) respectively. For instance, when the bit width is 31, the sorting process time and the average compression ratio are same as that of 8-way/8-parallel with the data compression shown in Figure 4.24 and Gmean shown in Figure 4.25. As shown in Figure 4.26, average compression ratio is improved as bit width is reduced. This is because compressible data is increased as each delta becomes smaller by narrowing values. Also, it is obvious that as average compression ratio is improved, sorting process time is shorter. Given that result, the data compression mechanism can improve the sorting performance in a random data set if the average compression ratio is high.

Figure 4.27 shows the hardware resource usage of 8-way/8-parallel with and without the data compression mechanism. The hardware resource usage of 8-way/8-parallel without the data compression mechanism is same as that of 8-way/8-parallel shown in Figure 4.22. With the data compression mechanism, the increase rates of FF, LUT Logic, LUT RAM, and Block RAM are 3.17%, 12.4%, 3.01%, and 1.46% respectively. In particular, the increase rate of LUT Logic is largest between them in order to implement subtractors and adders for the compressor and decompressor. The increase rates of FF

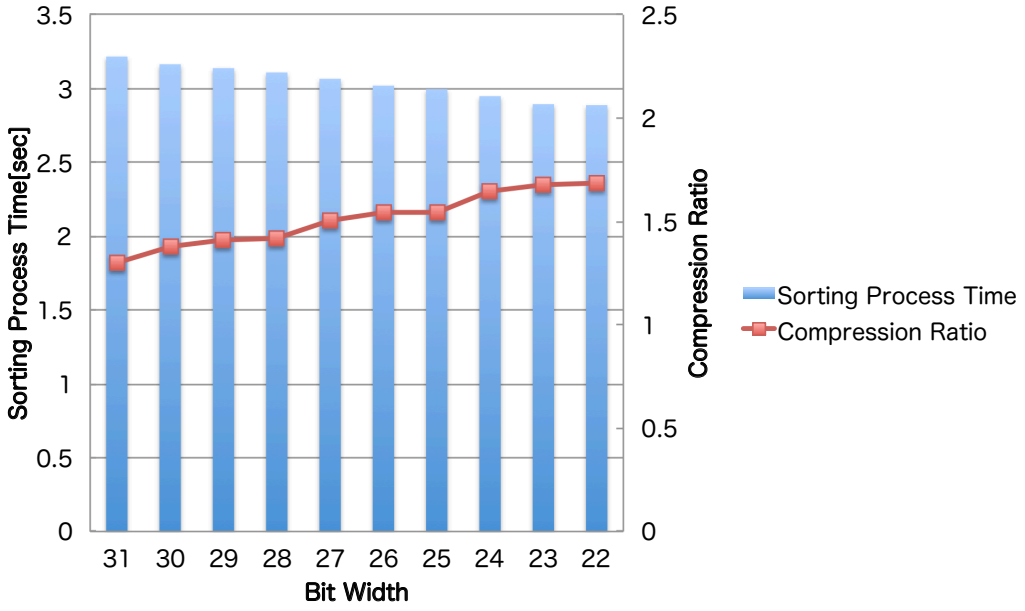


Figure 4.26 The sorting process time and average compression ratio using a random data sequence with small deltas

and LUT RAM are due to implementation of temp FIFO and state machines for the data compression mechanism. The increase rate of Block RAM depends on FIFO of the decompressor. The increase rate of Slice is 14.55%. A Slice is a term used by Xilinx, and it is a logic component including several LUTs and FFs. In other words, the increase rate of LUT Logic is dominant of that of Slice.

4.5.5 Discussion

I evaluated the sorting accelerator in terms of the sorting process time and the hardware resource usage. These results show that the more the number of ways and duplicated trees is increased, the higher the sorting performance is, although it needs more hardware resources. Well then, if same hardware resource usage, which customized hardware achieves the highest sorting performance?

Figure 4.22 shows that the hardware resource usage is almost same among three configurations, which are 4-way/4-parallel, 8-way/2-parallel, and 16-way. This means that if the total number of Input Buffers is same, the hardware resource usage is almost same. In the three customized hardware, the results of 4-way/4-parallel, 8-way/2-parallel, and 16-way are 5.05sec, 5.93sec, and 8.08sec. Thus, 4-way/4-parallel achieves the highest sorting performance. This result means that if same hardware resources are used, the hardware that has more duplicated trees can achieve higher performance. This is due to the sorting logic throughput described in Section 4.5.3.

Figure 4.28 shows relationship between the performance and the hardware resource usage. The x-

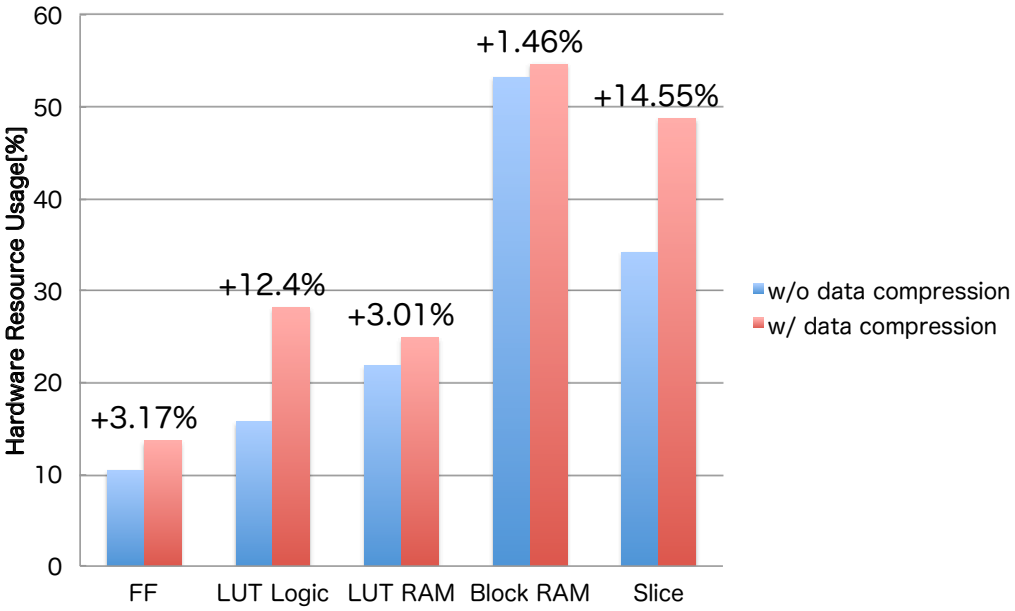


Figure 4.27 The hardware resource usage of 8-way/8-parallel with and without the data compression mechanism

axis and y-axis stand for the number of consumed slices and the speed-up ratio compared with merge sort. As mentioned before, if the total number of Input Buffers is same, the hardware resource usage is almost same — (4-way/2-parallel and 8-way), (4-way/4-parallel, 8-way/2-parallel, and 16-way), (16-way/2-parallel and 8-way/4-parallel), and (16-way/4-parallel and 8-way/8-parallel). However, it is found that the hardware, which has more duplicated trees, consumes slightly more slices. This is because control logics for the duplicated trees are also duplicated.

I draw three areas in Figure 4.28. The left is for cost aware systems. I define the borderline of this area as the number of available slices on the Artix-7 XC7A100T that is used in the Digilent Nexys4 board [105]. The middle is for cost-performance aware systems. I define the upper border of this area as the number of available slices on the Kintex-7 XC7K325T that is used in the Xilinx Kintex-7 FPGA KC705 Evaluation Kit [106]. If more performance-aware systems are required, they need larger devices like the Virtex-7 FPGAs. As shown in Figure 4.28, the designs except 16-way/4-parallel and 8-way/8-parallel are within the left area, and the other designs are within the middle area. This means that most of the presented designs in this chapter can be implemented on low-end devices and our proposed accelerator is available on various environments depending on constraints of the cost and performance. In [107], the proposed sorting accelerator is implemented on a low-end FPGA, and the system achieves 1.28x sorting performance than a desktop computer. I release the RTL source code as an open-source hardware. Hence, designers can customize a sorting accelerator composed of required

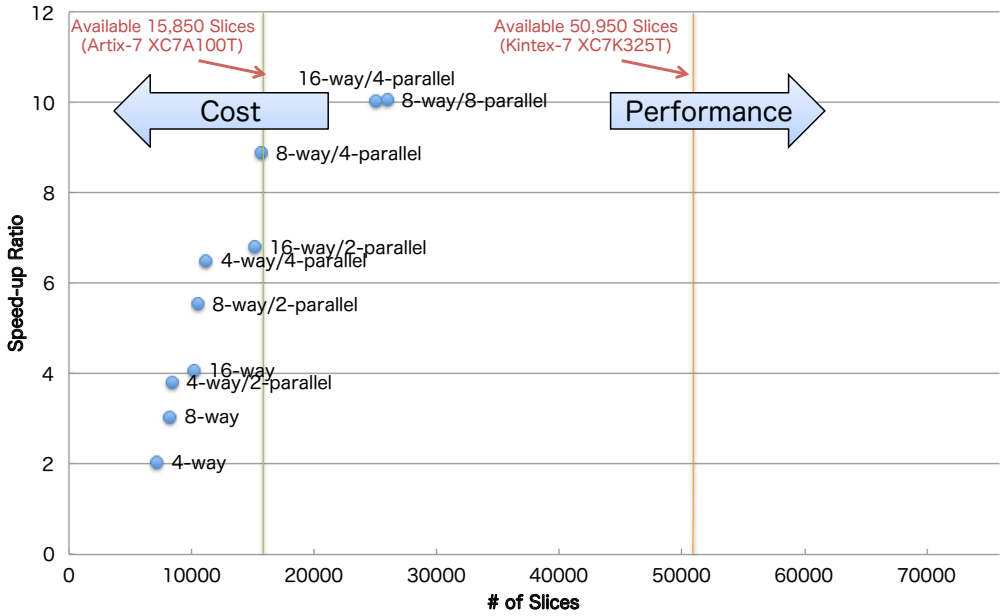


Figure 4.28 Relationship between the performance and the hardware resource usage

hardware resources by means of tuning the number of ways and duplicated trees. However, note that if the number of duplicated trees is increased, it is crucial to consider the memory bandwidth utilization like 8-way/8-parallel.

8-way/8-parallel achieves up to 10.06x speed-up compared with single thread of Intel Core i7-4770 operating at 3.4GHz. In comparison with the systems using the sorting network [93, 94, 95], our proposed system can sort more elements at higher speeds. However, the sorting performance is limited because of the insufficient average memory bandwidth. To overcome it, the data compression mechanism is proposed and attached to 8-way/8-parallel. The experimental result show that 8-way/8-parallel with it achieves 95% of the estimated performance because of alleviation of the memory bandwidth limitation. To implement the data compression mechanism, several hardware resources, especially LUT Logic, are utilized because of the subtractors and adders for the decompressor and compressor respectively.

I rebuild the analytical model according to the sorting throughput, the memory bandwidth, and the average compression ratio. If the memory bandwidth is insufficient against the sorting throughput, I define the number of required cycles to fully sort the data sequence by the sorting accelerator as x. Using the sorting throughput, the memory bandwidth, and x, the ratio m of the memory bandwidth against the sorting throughput is given the following formula.

$$m = 1 - \frac{(x - C_{fully\_dup})}{C_{fully\_dup}} = \frac{memory\ bandwidth}{sorting\ throughput} \tag{4.10}$$

The formula can be solved for x and shown as the following formula.

$$x = 2 \times C_{fully\_dup} - m \times C_{fully\_dup} \tag{4.11}$$

With the data compression mechanism, the memory bandwidth utilization is improved and x can be given by

$$x = 2 \times C_{fully\_dup} - c \times m \times C_{fully\_dup} \tag{4.12}$$

where c is the average compression ratio.

4.6 Related Work

In recent years, FPGAs have benefited from technology process advances to become significant alternatives to ASICs, and lots of companies and research institutes have been interested in them. Due to the trend, several studies have proposed sorting hardware with FPGAs [95, 94, 108, 109, 93, 97].

The sorting network is one of the most famous sorting architectures, and most studies focuses on it [95, 94, 108, 109, 93]. In [93, 94, 95], FPGA-based systems with sorting networks are implemented and evaluated in terms of circuit areas, throughputs, and power consumptions. [109] proposes a Domain Specific Language (DSL) and a compiler to automatically generate sorting networks with optimized throughput and area efficiency. As mentioned before, a sorting network is easy to be implemented in hardware due to simplicity of the architecture, but is unsuitable for larger data sequences. This is because more comparators are required to sort them, and this causes the circuit area increase and the operating frequency degradation. Therefore, the sizes of these data sequences are small. In [93], if the data-sequence size is less than 8, it can be fully sorted only in the sorting network. However if not, the CPU merges these sorted portions.

In addition to the sorting network, the merge sorter tree is proposed in [97, 108]. In particular, [108] proposes a special merge sorter tree that can handle 6 elements per cycle, while our system can do only one element per cycle. Due to this, the system performance is about 7x than our system. However, the merge sorter tree, which can handle multiple values per cycle, is truly difficult to be operated at high clock frequencies. I also implement the tree that can handle 4 elements per cycle, and then in the RTL simulation the sorting accelerator with the tree works successfully. However, the logic can operate at most 140MHz according to the post-place and route timing report. Therefore, to realize the method proposed in [108], it definitely needs high-level optimization techniques. Unlike [108], our sorting accelerator is simple, relatively high speed, customizable, and the RTL source code is released as an open-source hardware. These are significant differences with the prior work and I have never seen such a sorting hardware.

As mentioned before, as the sorting logic throughput is higher, it is truly important consider approaches which can improve the memory bandwidth utilization while keeping the operating frequency

high. However in [108], the proposed hardware has massive memory bandwidth and the authors do not consider that problem. In contrast to [108], I propose a data compression mechanism for the sorting accelerator to mitigate the bandwidth limitation of accessing the off-chip memory. The experimental results show that the sorting accelerator with the mechanism achieves better performance than without it. To the best of our knowledge, no related work proposes data compression mechanisms for sorting hardware and evaluates the effectiveness. This is also a significant difference with these previous studies.

4.7 Summary

In this chapter, I presented the acceleration approach for sorting application. My proposed accelerator uses two sorting architectures that are the sorting network and the merge sorter tree, and I detailed the design and implementation. The most characteristic point of the proposed system is customizable, and I also provided a detailed analytical model that accurately estimates the sorting performance depending on the hardware configuration. Due to these characteristics, designers can estimate sorting accelerator performance in advance and can implement the best one that fulfills cost and performance constraints.

The highest performance configuration, 8-way/8-parallel, sorts 256M 32-bits integer elements at 10.06x and 8.01x faster than merge sort and quick sort respectively. However, the sorting performance is limited because of the insufficient memory bandwidth. To address this problem, I proposed the data compression mechanism based on the algorithm using a base value and an array of deltas. As a result, 8-way/8-parallel with the data compression mechanism can achieve up to 95% of the estimated performance, while 8-way/8-parallel without it does about 70%.

In order to allow every designer to easily and freely use this accelerator, the RTL source code is released as an open-source hardware. To the best of our knowledge, this is the first open-source sorting accelerator in the world that is high performance, is customizable, and improves the memory bandwidth utilization. All the code used to obtain the results in this chapter is also available at https://github.com/monotone-RK/FACE.

Chapter 5

Essential Components for Efficient Development Infrastructure

5.1 Motivation

In this chapter, I describe what is important for efficiently developing FPGA-based accelerators.

As mentioned before, in order to develop FPGA-based hardware, designers get to work on the RTL modeling with HDL like VHDL or Verilog HDL after the design specification is determined. Although development toolkits like HLS are proposed to mitigate the design stress and to improve the productivity, the ease of programmability, performance, resource usage and efficiency can vary from one technology to another, and there is usually a tradeoff between these characteristics as described in Chapter 2. Besides, it is not easy to choose the best development framework for implemented applications and this is mostly dependent on programmer experience. That is why designers still have to do RTL modeling using HDL, and RTL simulation is an important step in ensuring that the designed hardware behavior meets the design specification.

However, designing large-scale circuits and using large data sets for applications running on the circuits can lead to long RTL simulation times, which means that traditional RTL simulators cannot finish the circuit behavior verification within a realistic time frame. Besides, FPGA capacity has been increased year after year, for instance [110] reports that Xilinx FPGA capacity has increased over 6x in less than 5 years. Figure 5.1 shows the graph cited from [110]. FPGAs will be larger thanks to transistor scaling and stacking, which leads to more RTL simulation time. Therefore high-speed RTL simulation environments are essential in the future.

Another important point is that designers have to consider how to efficiently utilize and connect hard blocks like memory and DSP in addition to LBs in order to implement lots of stream computation units on an FPGA that can work at high speeds in parallel. This is mentioned in Chapter 2 and I demonstrated the validity from the development of the two proposed accelerators. In stencil computation and sorting, computation units like MADD and the merge sorter tree are pipelined and duplicated to exploit the

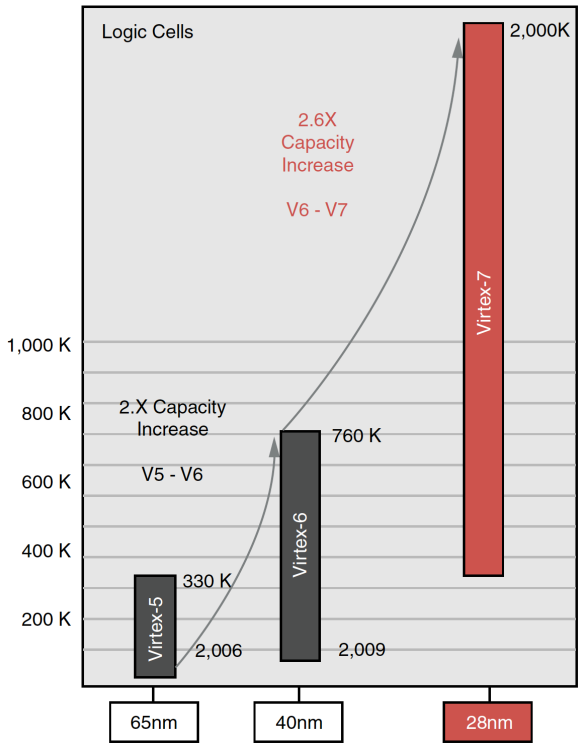


Figure 5.1 The capacity increase in Xilinx FPGAs cited from [110]

computation throughput and parallelism. In particular, stencil-computation accelerator uses the tiling technique, which decomposes the data set, in order to further exploit parallelism. In this chapter, I discuss that these findings are valid on other hardware platforms and in the other computation kernels described in Chapter 2.

5.2 High-speed RTL Simulation Overview

In this chapter, I propose an efficient verification environment enabling high-speed RTL simulation for development of FPGA accelerators called SimVerilog. Figure 5.2 shows the SimVerilog overview. To build SimVerilog I used two previously proposed tools, which are ArchHDL [51] and Pyverilog [111].

To make the RTL simulation more efficient, there are several prior studies that focus on building high-speed RTL simulation environments and C/C++ are often used to build them. Authors in [51] propose ArchHDL, which is a C++11-based library for RTL modeling and simulation, and evaluate the effectiveness under multicore environment. ArchHDL is implemented in C++ yet the coding style is similar to Verilog HDL, and the authors also propose a code translator that converts ArchHDL code into Verilog HDL code. However, hardware is mostly designed in Verilog HDL because it is still the de facto standard. Therefore, I focus on translating Verilog HDL source code into ArchHDL code in

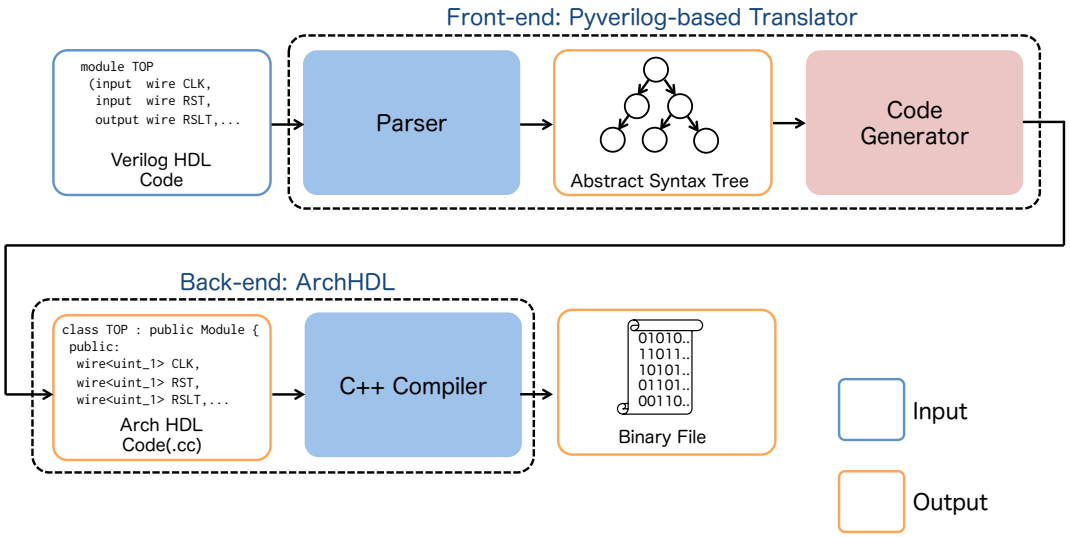
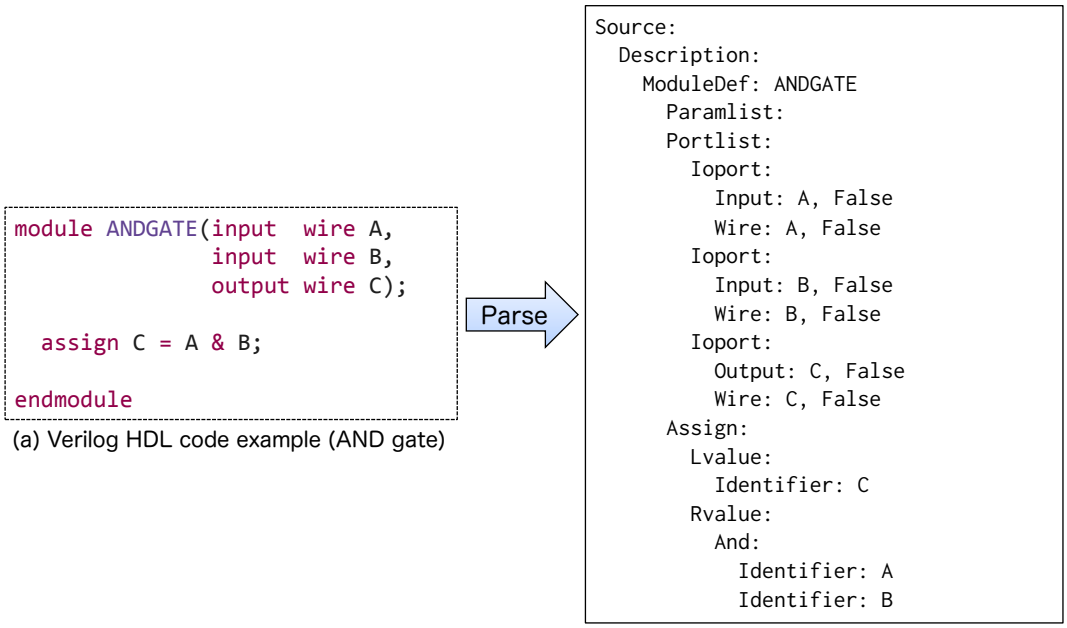


Figure 5.2 SimVerilog overview. I implement Code Generator to produce ArchHDL code from the generated abstract syntax tree by analyzing Verilog HDL source code with Parser.

Figure 5.3 The correspondence relationship between Verilog HDL and ArchHDL

order to use ArchHDL's simulation capability.

To achieve the approach, there are two options: string replacement, or syntactic analysis. The former is simpler, but is too difficult to handle all Verilog HDL source code. Figure 5.3 shows the correspondence relationship between Verilog HDL and ArchHDL. For instance, designers have to expressly describe port connections to show that which port of the instanced sub module is connected to which wire or register, while the Verilog HDL method is similar to a software function call. In order to address the problem, it is necessary to analyze Verilog HDL source code using software tools like compilers, but



(a) Verilog HDL code example (AND gate)

(b) Generated AST

Figure 5.4 Pyverilog analyzes (a) the Verilog HDL source code of AND gate and generates (b) the Abstract Syntax Tree (AST).

there is a lot of cost and effort to develop such tools from scratch. Therefore, I develop a code translator from Verilog HDL to ArchHDL employing a previously proposed tool called Pyverilog [111].

Pyverilog is an open-source hardware design analysis toolkit for Verilog HDL that is implemented in Python. To realize the Pyverilog-based code translator, I use the code-parsing feature that is originally provided by Pyverilog and implement a code generator to produce ArchHDL code from the generated Abstract Syntax Tree (AST) from the feature shown in Figure 5.2.

Combining the translator and ArchHDL, I built SimVerilog. At first the Pyverilog-based translator analyzes the RTL designs described in Verilog HDL, and then generates the AST. It is then used in Code Generator to produce ArchHDL code illustrated in Figure 5.2, and then The generated ArchHDL source code can be compiled with standard GNU or Intel Compiler to generate a simulation binary. Finally designers can do the RTL simulation by running the binary.

In the following section, I present the two previously proposed tools that are fundamental features of SimVerilog.

5.2.1 Pyverilog

Pyverilog [111] offers (1) code parser, (2) dataflow analyzer, (3) control-flow analyzer, (4) visualizer and (5) code generator for Verilog HDL. To implement the front-end of SimVerilog, I use (1) code parser and describe the function and design alternatives for it as follows.

Code Parser

The parser of Pyverilog is a fundamental tool to analyze a source code written in Verilog HDL. The parser generates an abstract syntax tree (AST) from the code of Verilog HDL for later analysis and external tools. In SimVerilog, the generated AST is used in the code generator for ArchHDL. At first, Pyverilog calls Icarus Verilog [25] with -E option as the preprocessor, and then all macros like define and ifdef are extracted and the fabricated source code in the form of text are available. After that, the parser of Pyverilog reads the source code and builds up an AST in the form of nested class objects in Python. Finally the generated AST is ready for the next analysis step. To build the parser, the author chooses PLY (Python Lex-Yacc) [112] as the parser generator (compiler-compiler), which is a lightweight implementation of a lexical analyzer and an LR-parser.

Figure 5.4 shows the code parsing and AST generation by Pyverilog. It can be seen that the source code of AND gate is parsed and the AST is generated. In this example, an AND gate is used in Figure 5.4 (a), and it is correctly analyzed and converted into an appropriate AST representation illustrated in Figure 5.4 (b).

5.2.2 ArchHDL

Basic Concept

ArchHDL [51] is a hardware description language based on C++11. The characteristics of ArchHDL are:

1. To describe combinational circuit as a lambda function,
2. To support non-blocking assignment,
3. To support user-defined datatypes and object-oriented programming style,
4. Cycle based simulation (not event driven),
5. Parallel simulation using OpenMP without decreasing the accuracy,
6. Similar coding style to Verilog HDL,
7. Simple library implementation (only about 300 lines in total).

Figure 5.5 shows the code of xorshift pseudo random value generator [98] and its test bench. This is an example design written in ArchHDL.

ArchHDL library implements Module class and wire/reg class templates. As shown in Figure 5.5, hardware designs can be expressed by inheriting or specializing those templates. An instance of wire and reg class can be regarded as a *wire* and *reg* in Verilog HDL and designers can handle user-defined datatypes for specialization of wire and reg.

Wire class has a member variable of a lambda function which represents combinational circuit (to be set in Assign function, line 17-20). Its operator "()" returns evaluation results of the lambda.

```
1  #include <stdio.h>
2  #include "arch_hdl.h"
3
4  class Xorshift : public Module {
5   public:
6    wire<uint_1> i_rst_x;
7    wire<uint_1> i_enable;
8    wire<uint_32> i_seed;
9    wire<uint_32> o_out;
10
11   reg<uint_32> x;
12   reg<uint_32> y;
13   reg<uint_32> z;
14   reg<uint_32> w;
15   wire<uint_32> t;
16
17   void Assign() {
18     t = [=]() { return x() ^ (x() << 11); };
19     o_out = w;
20   }
21   void Always() {
22     if (!i_rst_x()) {
23       x <<= 123456789;
24       y <<= 362436069;
25       z <<= 521288629;
26       w <<= 88675123 ^ i_seed();
27     } else {
28       if (i_enable()) {
29         x <<= y();
30         y <<= z();
31         z <<= w();
32         w <<= (w() ^ (w() >> 19)) ^ (t() ^ (t() >> 8));
33       }
34     }
35   }
36 };
37
38 class TestTop : public Module {
39  public:
40   static const uint_32 HALT_CYCLE = 30;
41
42   reg<uint_1> HALT;
43   reg<uint_32> cycle;
44
45   wire<uint_1> rst_x;
46
47   wire<uint_32> seed;
48   wire<uint_32> rand;
49   Xorshift xorshift;
50
51   void PortConnect() {
52     xorshift.i_rst_x = rst_x;
53     xorshift.i_enable = rst_x;
54     xorshift.i_seed = seed;
55     rand = xorshift.o_out;
56   }
57   void Assign() {
58     rst_x = [=]() { return (cycle() < 1) ? 0 : 1; };
59     seed = [=]() { return 1; };
60   }
61   void Initial() {
62     HALT = 0;
63     cycle = 0;
64   }
65   void Always() {
66     cycle <<= cycle() + 1;
67     HALT  <<= (cycle() >= HALT_CYCLE);
68
69     if (!rst_x()) {
70     } else {
71       printf("%08x\n", rand());
72     }
73   }
74 };
75
76 int main() {
77   TestTop testtop;
78   do {
79     ArchHDL::Step();
80   } while (!testtop.HALT());
81
82   return 0;
83 }
```

Figure 5.5 Xorshift pseudo random value generator and test bench in ArchHDL

On the other hand, reg class has two member variables named 'current' and 'next' and its operator "()" returns the value of 'current'. Overloaded operator "<<=" assigns right-hand value to 'next', and by calling a member function 'Update', a value of 'next' will be copied to 'current'. Calling an Update function after calculating all 'next' of regs realizes non-blocking assignments like Verilog HDL or VHDL. In the current version, Always function of modules corresponds to always@(posedge) notation in Verilog HDL (line 21-35).

Design files written in ArchHDL can be compiled with standard GNU or Intel Compiler to generate a simulation binary. Figure 5.6 shows the simulation flow of ArchHDL. First, it assigns a lambda function to each wire and initial value to each reg. After that, a simulation enters the main loop which executes simulation one cycle by one cycle. The execution of one cycle is divided into two parts. First

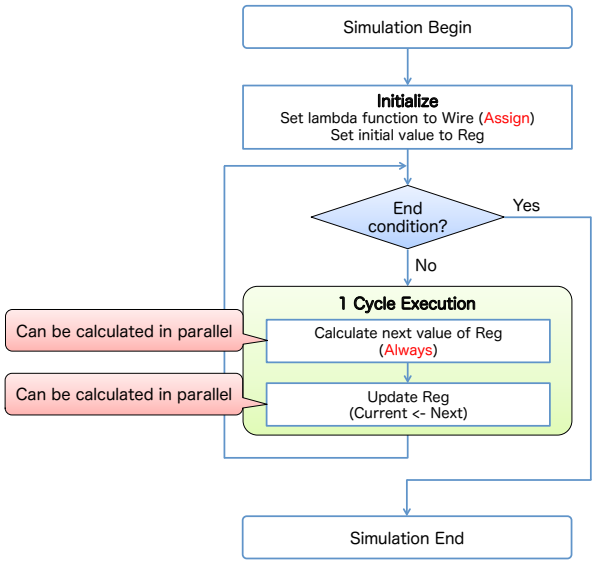


Figure 5.6 Simulation flow chart of ArchHDL

```
1  #pragma omp parallel
2    {
3      update_registers_[thread_num].clear();
4  #pragma omp for
5      for (size_t i = 0; i < modules_size; ++i) {
6        modules_[i]->Always();
7      }
8
9      for (auto reg : update_registers_[thread_num]) {
10       reg->Update();
11     }
12   }
```

Figure 5.7 Simulation kernel of ArchHDL using OpenMP

part computes the 'next' of all regs (call Always function of all module classes), and second part copies the 'next' to 'current' by calling Update function of each reg.

Parallelization using OpenMP

In main loop (light-green area in Figure 5.6), both parts can be calculated in parallel respectively by OpenMP. The accuracy of the simulation is not changed by parallelization. Figure 5.7 shows this part of ArchHDL library code.

In Figure 5.7, vector modules_ holds all pointers to module classes in the design. First for-loop calls Always function of every module in parallel by using OpenMP, which calculate 'next' of regs. At this time, if 'next' has different value from 'current', a pointer to the reg is added to vector update_registers_. Second for-loop calls Update function of the regs in update_registers_. Those regs are reset in every step (line 3).

Vector update_registers_ is prepared for each thread. Due to this, it is possible to eliminate the critical section that adds the pointer of reg to be updated to one vector, which leads the simulation speed-up.

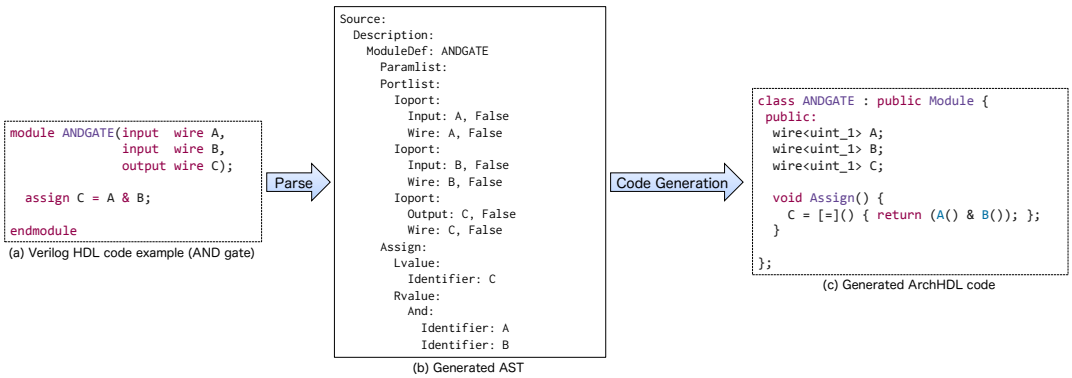


Figure 5.8 The overview of ArchHDL code generation from Verilog HDL source code.

```
class ASTNodeVisitor(object):
    def visit(self, node):
        method = 'visit_'+ node.__class__.__name__
        visitor = getattr(self, method, self.generic_visit)
        return visitor(node)
    def generic_visit(self, node):
        for c in node.children():
            self.visit(c)
```

Figure 5.9 The visitor pattern for the generated AST from Verilog HDL source code in Python

```
class ArchHDLCodeGenerator(ASTNodeVisitor):
    def __init__(self, init_data):
        self.init_data = init_data

    def visit_ModuleDef(self, node):
        """ executes a specific feature for the node of ModuleDef and return the result """
        rslt = self.visit(node.child)
        return rslt

    def visit_Portlist(self, node):
        """ executes a specific feature for the node of Portlist and return the result """
        rslt = self.visit(node.child)
        return rslt

    def visit_Ioport(self, node):
        """ executes a specific feature for the node of Ioport and return the result """
        rslt = self.visit(node.child)
        return rslt

    def visit_Assign(self, node):
        """ executes a specific feature for the node of Assign and return the result """
        rslt = self.visit(node.child)
        return rslt

    def visit_And(self, node):
        """ executes a specific feature for the node of And and return the result """
        return node.operator

    def visit_Identifier(self, node):
        """ executes a specific feature for the node of Identifier and return the result """
        return node.name
```

Figure 5.10 The implementation of ArchHDL code generator inheriting ASTNodeVisitor illustrated in Figure 5.9

5.3 ArchHDL Code Generator

ArchHDL code generator in the front-end of SimVerilog is used for generating a source code written in ArchHDL from the intermediate representation of an AST. Figure 5.8 shows the overview of ArchHDL code generation from Verilog HDL source code. Figure 5.8 (a) and (b) are same as Figure 5.4, and the ArchHDL code (c) is generated from the AST. The feature is implemented in Python using a standard visitor pattern that is to visit each AST node and to call a specific function for each visited AST node

Table 5.1 Evaluation setup

CPU	Intel Xeon E5-2687W (8 cores/16 threads) @ 3.1GHz
Memory	64GB
OS	Ubuntu 14.04.1 LTS x86_64
C++ Compiler	g++ 4.8.2 (-Ofast), icpc 15.0.3 20150407 (-Ofast)
Synopsys VCS	Version H-2013.06_Full64

using its class name recursively.

Figure 5.9 shows the implementation of the visitor pattern to traverse the generated AST nodes and Figure 5.10 shows the implementation of ArchHDL code generator inheriting ASTNodeVisitor illustrated in Figure 5.9. ASTNodeVisitor has two methods: visit and generic_visit. At first ASTNodeVisitor visits an AST node and calls the visit method to search the specific function for the node using its class name (line 3 and 4). If it exists, the function executes its process and returns the result, but if not, the generic_visit method is called (line 5). Existing the corresponding function means whether or not there are the methods having same name as 'method' in line 3 of Figure 5.9 in Figure 5.10. By searching and executing the appropriate function for each AST node recursively, ASTCodeGenerator can get all information of the module.

Using this information, ArchHDL code can be generated. Common source code generators in any language contain a set of template texts of output codes in their internal source codes. Therefore, the amount of the source codes is usually larger than the other parts. In order to separate the implementation of the entire code generator into the code write parts using template texts and the control parts, I use a template engine. In this implementation, I choose Jinja2 [113], a major template engine used in various web applications with Python. Using the template engine, all template texts of ArchHDL codes are removed from the source code in Python. It simplifies the software structure of the code generator.

5.4 Evaluation

In this section, I evaluate RTL simulation speed of SimVerilog compared with Synopsys Verilog Compiler Simulator (VCS) [20]. For this evaluation, I use the stencil and sorting accelerator described in Chapters 3 and 4 as simulated test circuits. To measure the accurate simulation time, I use the chrono library included in C++11 and time command for SimVerilog and VCS respectively, and SimVerilog and VCS are executed 10 times in each RTL simulation and I take the arithmetic mean. Table 5.1 shows the evaluation setup.

5.4.1 Sorting Accelerator

Figure 5.11 shows the proposed sorting accelerator with 4-way merge sorter tree. As described in Chapter 4, this accelerator is customizable by means of tuning the number of ways and duplicated

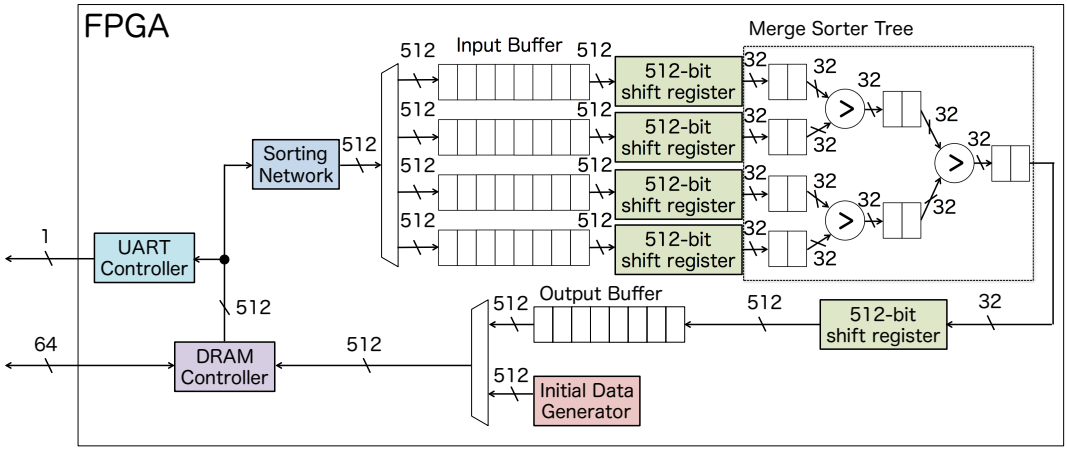


Figure 5.11 The proposed sorting accelerator with 4-way merge sorter tree

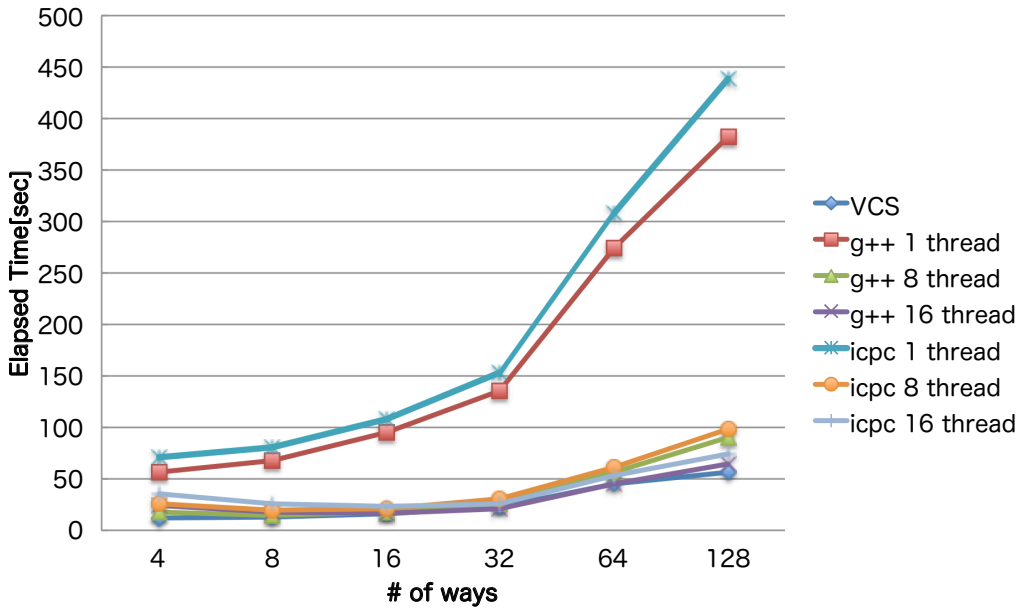


Figure 5.12 Simulation time of VCS and SimVerilog using OpenMP, depending on each hardware running 256K elements sorting

trees. In this evaluation, I measure the simulation time according to the number of ways.

Figure 5.12 shows the simulation time of VCS and SimVerilog using OpenMP, depending on each hardware running 256K elements sorting. The number of threads used in SimVerilog is 1, 8, and 16. 8 and 16 are equal to the number of physical and logical (due to Intel Hyper-Threading Technology) cores. As shown in Figure 5.12, the results using g++ are slightly better than those of icpc. The elapsed time with g++ 16 thread is almost equal to that of VCS.

Figure 5.13 shows SimVerilog simulation speed-up ratio compared with g++ single thread in each

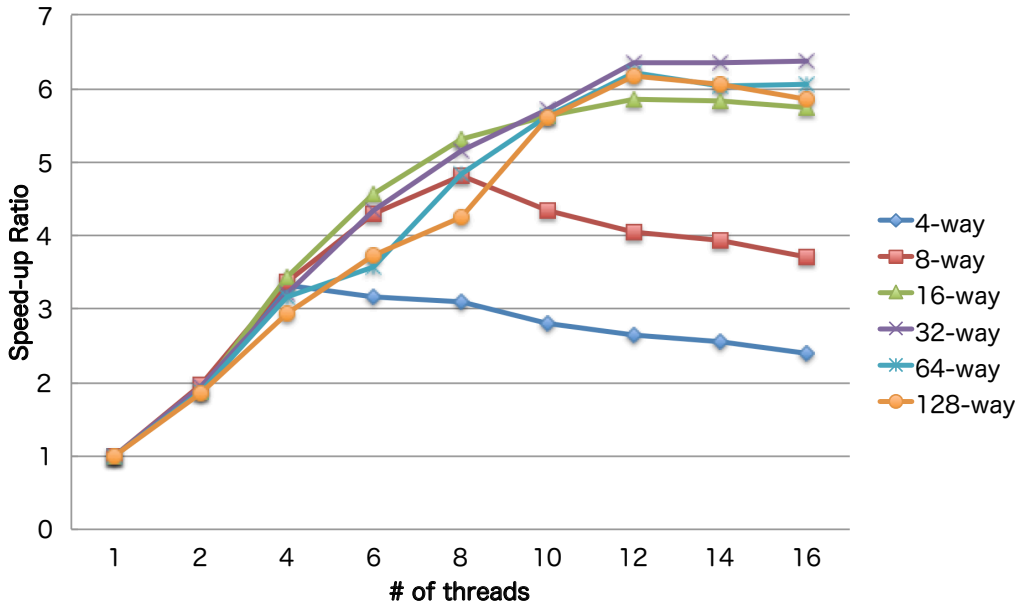


Figure 5.13 Speed-up ratio compared with g++ single thread in each hardware running 256K elements sorting

hardware running 256K elements sorting. The RTL simulation with g++ is slightly better than icpc, and that is why g++ is used for the speed-up ratio comparison. As shown in Figure 5.13, all cases except 4-way achieve higher speed simulations compared with g++ single thread if the number of threads is from 2 to 8. In 4-way, the speed-up is achieved if the number of threads is 2 and 4, but the speed-up is degraded from 6 threads. This is because the parallelized parts are fewer than the other hardware configurations, which means that the number of Modules and regs is insufficient relative to the number of threads. In other words, even if the number of threads is increased, the task parallelism cannot be exploited, and worse, the synchronization overhead between the threads is exposed. From 8 threads, the speed-up ratio of larger hardware basically becomes better. The RTL simulation for the sorting accelerator with 32-way merge sorter tree shows the best speed-up ratio when the number of threads is 16, which is 6.4x speed-up.

It is possible for SimVerilog to achieve high-speed RTL simulations with OpenMP, but even the 16 threads performance is almost equal to VCS because the single thread performance is significantly lower. This is because the sorting accelerator mostly consists of combinational logics like a comparator and wires propagating values depending on the comparison result, in other words the wire components are dominant in the sorting accelerator. This means that the accesses to 'current' in a reg class are large, in other words, memory access is frequent. For instance the speed-up ratio of 128-way is worse than that of 32-way. This is because the memory access overhead becomes dominant even if the RTL

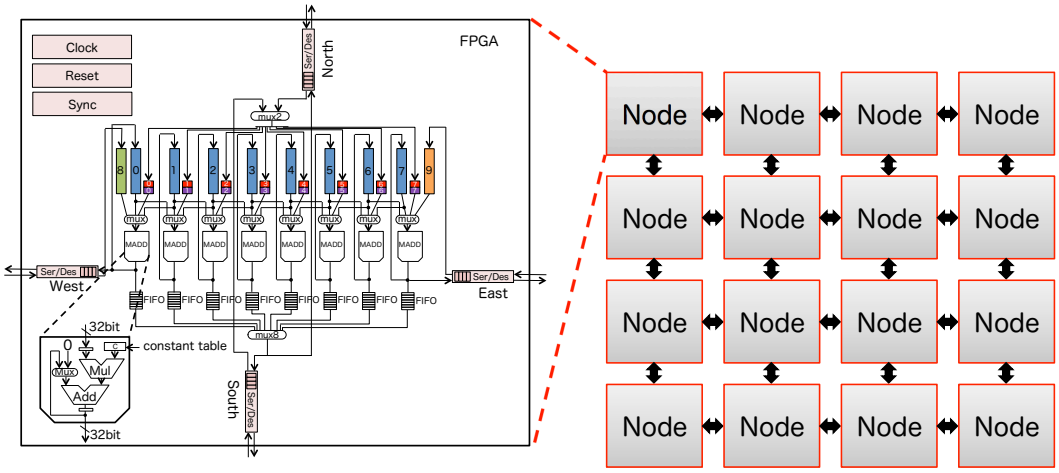


Figure 5.14 The proposed stencil-computation accelerator composed of 4 × 4 nodes

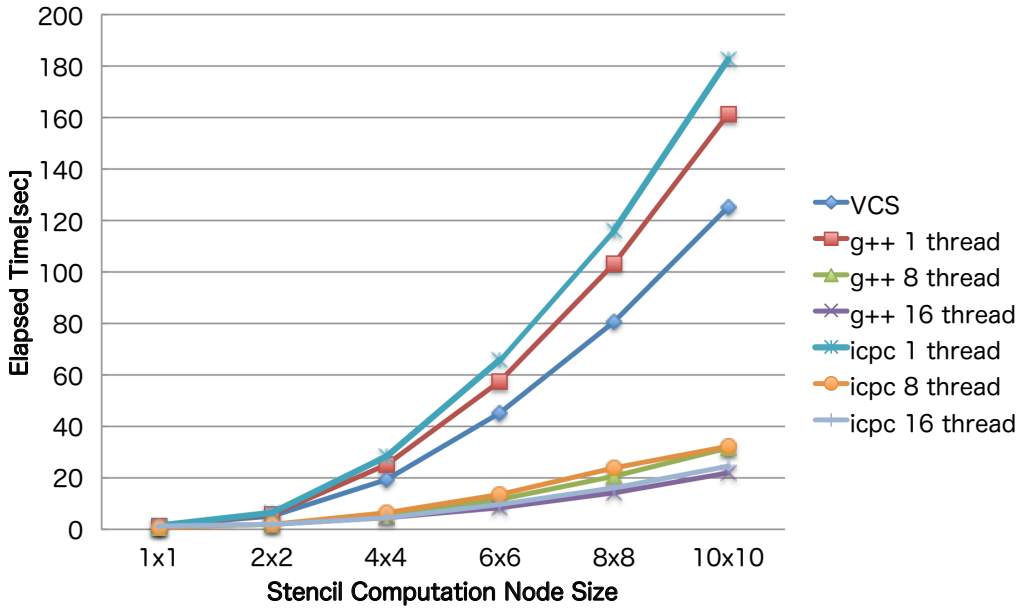


Figure 5.15 Simulation time of VCS and SimVerilog using OpenMP, depending on each hardware configuration for stencil computation

simulation with OpenMP is effective, since the number of wire components is increased according to the number of ways. Given the results, the sorting accelerator is not suitable to SimVerilog.

5.4.2 Stencil-computation Accelerator

Figure 5.14 shows the proposed stencil-computation accelerator composed of 4 × 4 nodes. As described in Chapter 3, the stencil-computation accelerator can be scaled according to the number of FPGA nodes.

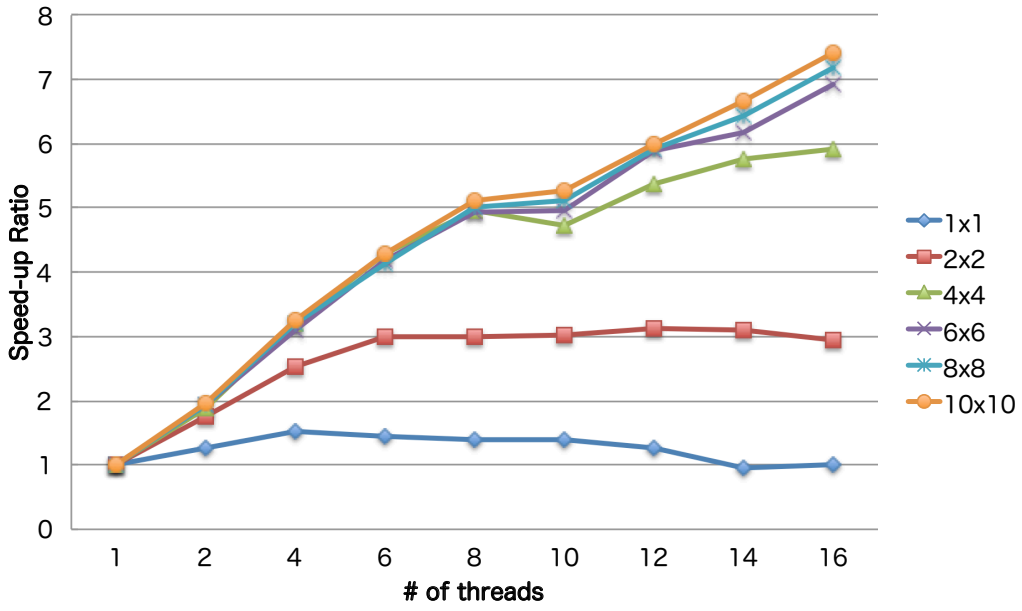


Figure 5.16 Speed-up ratio compared with g++ single thread in each hardware configuration for stencil computation

In this evaluation, I measure the simulation time according to the number of FPGA nodes.

Figure 5.15 shows the simulation time of VCS and SimVerilog using OpenMP, depending on each hardware configuration for stencil computation. For instance 2x2 stands for the accelerator composed of four FPGA nodes. Similar to the sorting accelerator, SimVerilog uses 1, 8, and 16 threads, and it can be seen that the results using g++ are slightly better than those of icpc. As shown in Figure 5.15, SimVerilog using g++ and icpc can both overcome VCS in 8 and 16 threads. Using g++ 16 threads can achieve higher speed simulation from 1.2x to 5.8x compared with VCS. The speed-up ratio against VCS is larger according to the hardware size of the stencil-computation accelerator.

Figure 5.16 shows SimVerilog's speed-up ratio compared with g++ single thread in each hardware configuration for stencil computation. Similar to the sorting accelerator, the compiler is g++. As shown in Figure 5.16, all cases except 1x1 achieve higher speed simulations compared with g++ single thread if the number of threads is from 2 to 8. Similar to 4-way in Figure 5.13, the 1x1 configuration has insufficient Modules and regs that can be parallelized with OpenMP, and there is the synchronization overhead between the threads. From 8 threads, the speed-up ratio becomes better according to the number of nodes in the stencil-computation accelerator. The speed-up ratio in the 10x10 configuration is the largest, which is 7.4x compared with the single thread when 16 threads are used.

Unlike the sorting accelerator, SimVerilog can do considerably faster RTL simulations for the stencil-computation accelerator compared with VCS. This is because the stencil-computation accel-

erator mostly consists of sequential logics such as pipelined multipliers and adders, FIFOs, memory blocks. This means that the stencil-computation accelerator has lots of hardware components that can be parallelized. Of course the memory access overhead exists in this simulation, but SimVerilog can simulate it faster because the effectiveness of parallelization is dominant.

5.5 Related Work

A number of works have studied parallel RTL simulation and some works used GPUs or many-core.

In [114, 115], authors investigate the parallel RTL simulation of SystemC using GPUs. SystemC is a hardware description language implemented as a C++ class library. They are based on discrete event-driven simulation and [115] proposes a method whose aim is to reduce synchronization events. On the other hand, [116] translates Verilog files into GPU source code. The simulation method is based on Chandy-Misra-Bryant (CMB) algorithm, which is an asynchronous parallel simulation protocol. Although these works achieve high speed-up as much as 10 to 100 times, the simulated circuits are simple such as AES or 8b/10b decoding/encoding, and they do not evaluate the methods with practical hardware designs. In contrast, I use complex and large hardware designs such as the stencil-computation accelerator and sorting accelerator, and the evaluation results shows that my proposal is better than the commercial RTL simulator.

In [117], authors studied parallel simulation of SystemC using Intel Single-chip Cloud Computer (SCC) [118] that has 48 cores connected by a 2D mesh network. The simulation algorithm is also based on CMB. They used several sizes of Hermes Multi-Processor System [119] models running MPEG applications for evaluation. It showed that simulation on 48 SCC cores achieved up to 30x speed-up compared to that on 1 SCC core.

5.6 Discussion

In this section, I discuss that my proposals satisfy the key issues for efficient development infrastructure.

5.6.1 SimVerilog Usability

As shown above, SimVerilog can do RTL simulation faster than the commercial tool. However, the current implementation of SimVerilog supports little Verilog HDL syntax, which cannot execute RTL simulations including some syntax such as bit manipulations, multiple clocking, for-loop statements, etc. because ArchHDL cannot directly analyze these statements.

Even if SimVerilog is lacking in this regard, nevertheless SimVerilog can still work correctly unless such statements are used, which means the implemented Pyverilog-based code translator works successfully. If so, SimVerilog behavior is same as traditional RTL simulators like Icarus Verilog [25]

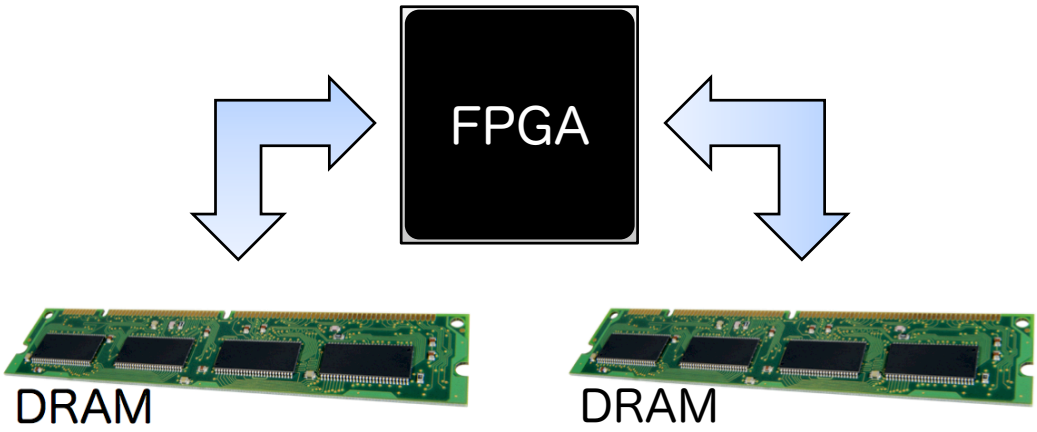


Figure 5.17 Hardware platform example: an FPGA and two DRAMs

or Synopsys VCS [20], which means that the standard output RTL simulation logs on a console are identical. Therefore, SimVerilog is used not only to do high-speed RTL simulations but also to debug Verilog HDL source code.

SimVerilog plays an important role of efficient development infrastructure for FPGA accelerators, but there is space to improve the usability. This is moved into open research topics.

5.6.2 Finding Applicability

As mentioned in Chapter 2, in order to implement a high performance FPGA-based accelerator, it is important to efficiently utilize and connect hard blocks like memory and DSP in addition to LBs. For instance, DSP blocks are 100% used to implement the stencil-computation accelerator and memory blocks are used in both accelerators to store computation data. It is also obvious to build lots of stream computation units on an FPGA that can work at high speeds in parallel. These are findings obtained from developing the stencil-computation and sorting accelerators. In this section, I discuss that the findings are available on other hardware platforms and in the other computation kernels described in Chapter 2.

For instance, the tiling approach described in Chapter 3 can be the most available on SGEMM. Matrices can be broken into sub-blocks that can be calculated on each FPGA in parallel. Also, building lots of computation hardware is effective to improve performance by parallel processing like the duplicated MADD and merge sorter trees described in Chapters 3 and 4 respectively. Authors in [120] propose a high performance FPGA-based accelerator for deep convolutional neural network. The core computation of the application is convolution. The authors propose a efficient computation logic circuit based on the nested residue number system and duplicate the circuit to improve the computation throughput. Of course, the data path is pipelined.

It is essential to consider not only how to implement high performance logic circuits on an FPGA but also hardware platform like FPGA grade, the number of FPGAs, interconnect and memory bandwidth, etc. Figure 5.17 shows a hardware platform example that has an FPGA and two DRAMs. As described in before, memory bandwidth limitation can restrict computation throughput. One of the options to overcome the problem is to "physically" increase the memory bandwidth, which means use of high-end memory modules like Hybrid Memory Cube (HMC) or memory module addition. In the case of sorting in Chapter 4, the sorting throughput against the memory bandwidth becomes half because the two DRAMs can be used for the read and write independently, which means that there is space to improve the sorting performance.

5.7 Summary

In this chapter, in order to address the FPGA accelerator development problem, I proposed an efficient verification environment enabling high-speed RTL simulation for development of FPGA accelerators. This environment is based on the two previously proposed tools, Pyverilog and ArchHDL.

Pyverilog is used for the implementation of the front-end of SimVerilog, which is a code translator from Verilog HDL source code to ArchHDL source code. Pyverilog originally offers (1) code parser, (2) dataflow analyzer, (3) control-flow analyzer, (4) visualizer and (5) code generator for Verilog HDL, and I use (1) code parser to implement the translator. In the front-end, Pyverilog parses Verilog HDL source code and generates the AST, and then ArchHDL code generator uses the AST to generate ArchHDL source code. I implemented the code generator based on the standard visitor pattern in Python, and used a major template engine Jinja2 used in various web applications with Python in order to simplify the software structure of the code generator.

In the back-end, ArchHDL is used as the RTL simulation engine of SimVerilog. ArchHDL is a C++11-based library for RTL modeling and simulation, which has an especially remarkable feature offering the parallelized RTL simulation with OpenMP. The generated ArchHDL source code can be compiled with standard GNU or Intel Compiler to generate a simulation binary. Finally designers can do the RTL simulation by running the binary.

I evaluate RTL simulation speed of SimVerilog compared with Synopsys VCS. For this evaluation, I use the stencil-computation and sorting accelerator described in Chapters 3 and 4 as simulated test circuits. As a result, although in the sorting accelerator the parallelized SimVerilog performance is almost equal to VCS, SimVerilog achieved the considerable speed-up that is up to 5.8x compared with VCS if the stencil-computation accelerator is used. The speed-up ratio compared with the single thread using g++ is improved according to the number of threads and the hardware size. The speed-up ratio of the 10x10 configuration has the largest speed-up, which is 7.4x compared with the single thread when 16 threads are used.

Also, I discuss that the findings from Chapters 3 and 4 is valid in other hardware platforms and computation kernels. As mentioned before, in order to implement a high performance FPGA-based accelerator, it is important to efficiently utilize and connect hard blocks like memory and DSP in addition to LBs, in other words it is essential to build lots of computation units that can work at high speeds in parallel. I show that the findings are promising in other kernels like SGEMM and convolution, and discuss how my proposed accelerator can accommodate hardware platform changes.

Chapter 6

Conclusion

6.1 Concluding Remarks

In this thesis, I proposed high performance FPGA-based accelerators for the two computation kernels, and a novel infrastructure supporting the efficient development of FPGA-based accelerators based on the findings obtained from the development of the two FPGA-based accelerators. This infrastructure shows how to build high-performance accelerators targeting several fundamental applications and enables high-speed RTL simulation to verify the developed hardware. The evaluation results showed that the proposed FPGA-based accelerators outperformed the corresponding implementation on CPUs and GPUs in terms of both performance and power efficiency.

The contributions of this thesis are as follows:

- I proposed a high performance FPGA-based accelerator for 2D stencil computation employing multiple small FPGAs.
- I proposed an FPGA-based sorting accelerator that combines the sorting network and the merge sorter tree.
- I proposed a novel infrastructure to show how to build high-performance FPGA-based accelerators targeting fundamental applications and to enable high-speed RTL simulation to verify the developed hardware.

The first contribution of this work is to propose a high performance FPGA-based accelerator for 2D stencil computation. Because the computation kernel has small arithmetic intensity, the sustained performance is limited due to memory bandwidth restriction under multicore microprocessors and GPUs. To address this, I proposed a high performance architecture for 2D stencil computation employing multiple small FPGAs and demonstrated that it is possible to implement the FPGA array system by using the three key techniques. The evaluation results showed that the development system correctly worked and achieved even better power efficiency than a typical GPU.

The second contribution of this work is to propose an FPGA-based sorting accelerator, which com-

bines the sorting network and the merge sorter tree. The proposed sorting hardware is customizable by means of tuning design parameters, and I also provided an analytical model that accurately estimates the sorting performance depending on the hardware configuration. The evaluation results showed that the proposed sorting accelerator achieved the estimated performance as long as the memory bandwidth requirement was trivial. Even if it was not, I also proposed a data compression mechanism for the sorting accelerator to mitigate memory bandwidth limitations and the evaluation results showed that the sorting accelerator with the mechanism achieved better performance than without it.

Finally I summarized important points of the efficient development infrastructure for FPGA-based accelerators according to the two previous contributions, which are an efficient verification environment and to consider how to efficiently utilize and connect hard blocks like memory and DSP in addition to LBs in order to implement lots of stream computation units on an FPGA that can work at high speeds in parallel. To build the efficient verification environment, I proposed SimVerilog that is based on the two previously proposed tools. The evaluation results showed that SimVerilog could do the RTL simulation faster than the commercial RTL simulator, and I discussed SimVerilog usability in terms of the RTL simulation speed and debuggability. Also, I discussed the finding applicability, which means that the findings obtained from the development of the two FPGA-based accelerators are valid in other hardware platforms and computation kernels.

6.2 Open Research Topics

There are several remaining topics from this research. I describe some of them as follows:

- to develop higher performance sorting hardware than the prior work and to evaluate the sorting performance including the data transfer,
- to make SimVerilog more practical so that the tool can support more Verilog HDL syntax,
- to automatically generate RTL simulation libraries for SimVerilog so that the tool can run on any environment.

The sorting performance of the proposed accelerator is lower than that of the prior work. To overcome it, I have to develop the merge sorter tree that can handle multiple values at high operating frequencies. Also, the sorting accelerator evaluation does not include the data transfer overhead, which is conducted under ideal situation. To accurately evaluate the usability of the sorting accelerator, I have to evaluate sorting performance including data transfer, such as AXI4, Avalon, NoC, PCIe, etc.

The second and third topics mean that SimVerilog is to be made more practical.

The current implementation of SimVerilog supports little Verilog HDL syntax, which cannot execute RTL simulations including some syntax, e.g. bit manipulations, multiple clocking, for-loop statements, etc. because ArchHDL cannot directly analyze these statements. For instance ArchHDL can express

bit manipulation features with the structure, but it is difficult to translate Verilog HDL code into the form. I have to reconsider and refine the implementation.

Currently I am trying to make SimVerilog automatically generate RTL simulation libraries for SimVerilog so that the tool can run on any environment. Some previous studies proposed RTL simulation methods on GPUs. The simulation library of ArchHDL is based on C++11 standard library functions, which means that it is not easy to port the simulation library to GPU programming environments like CUDA. However, GPUs have significant potential to achieve high-throughput parallel computing as mentioned before. To realize the concept, I try to examine the simulation library implementation referring to the previous studies. This is the third remaining topic.

This thesis presented the point of interest for developing FPGA accelerators and the novel verification environment for efficiently developing them. The RTL source code of the proposed sorting accelerator is available online and I will also publish SimVerilog as an open-source software. I believe that these open-source products and the knowledge provided from this study will advance the hybrid-computing era with CPUs, GPUs, and FPGAs.

Acknowledgement

This work has been supported in part by Core Research for Evolutional Science and Technology (CREST), JST.

Since 2011, I enrolled in this university as a master course student, and have been able to live a meaningful research life thanks to the support of a lot of people, for 5 years composed of 2 years of my master course and 3 years of my doctor course.

First of all, I would like to express my heartfelt gratitude to Associate Professor Kenji Kise. He has been my supervisor and has supported me for 5 years. When I enrolled in this university, I did not know even The LaTeX in spite of a computer science student. However, he has never abandoned me. His constant support, guidance, and encouragement have been radical for me to complete this thesis. Thanks to precious chances he gave me and his popularity, I have met a lot of distinguished researchers in both in domestic and international, and really have enjoyed my research life. Additionally, he has aggressively invited me to play tennis in order to make me healthy. Precisely because I could meet him at a graduate school admission guidance fair held in May 2010, I have been able to have invaluable experience. I would like to appreciate him again.

I also would like to thank all the members at Kise Laboratory. First, I appreciate Mr. Shintaro Sano. Because he took care of me entirely, I could complete this thesis in addition to my master thesis. Dr. Shinya Takamaeda-Yamazaki developed Pyverilog and gave me a lot of precious advice like FPGA knowledge, implementation techniques, researcher attitudes, etc. I am glad to be his junior, but I will surpass him someday. Dr. Shimpei Sato developed ArchHDL, and also took care of me and gave me a lot of advice in terms of both research and private lives. Dr. Naoki Fujieda gave me valuable advice and sometimes softened my heart by talking about our hometowns. Mr. Takayuki Matsumura often made a great mentor for me because our circumstances are same. Mr. Takakazu Ikeda is the first tie that I could have since I enrolled in this university. If he had not been in this laboratory, I might have dropped out. Mr. Tatsuya Kaneko gave me a Mac mini he used, which made my research life more comfortable. Mr. Hiroshi Nakatsuka often went out for a drink with me, because he gave me a rest appropriately, but sometimes interrupted me. Mr. Immanuel Victoria Encarnacion proofread this thesis in terms of English. Mr. Tomohiro Misono explained ArchHDL in detail. Because all of the laboratory members supported me, I have finally made it to the submission of this thesis. I would like to deeply

appreciate them again.

I would like to sincerely thank Professor Haruo Yokota, Professor Jun Miyazaki, Associate Professor Haruhiko Kaneko and Associate Professor Takuo Watanabe for many exact indications and suggestions as members of the thesis committee.

Dr. Yoshitaka Arahori and Dr. Atsushi Keyaki encouraged me when I was in a terrible slump. I appreciate them.

I would like to thank all those whom I met through TKT CAMPUS Asia Program, particularly Associate Professor John Kim, his laboratory members, and those whom I met at Korea Institute of Science and Technology. I could reaffirm my inexperience thanks to this program, and eventually submit this thesis.

I would like to thank all those whom I met at IEICE SIG-RECONF, IEICE SIG-CPSY, and Summer Workshop for their aggressive discussion with me. I am looking forward to doing so again.

I would like to thank those who work at Showa University Hospital, because I had appendicitis and got hospitalized in there when I was a third-year doctor's degree student.

I would like to thank COFFEE ROAST at Ōokayama for selling the great coffee bean, which supported my research life.

Mr. Yoshihiko Abe, Mr. Motonari Inagaki, Mr. Keita Imaizumi, Mr. Chihiro Kuga, Mr. Takaharu Kojima, and Mr. Taichi Homma are old friends of mine since we were children, and they have believed me. Because of it, I eventually reached the submission of this thesis. I really appreciate them.

Mr. Takuya Arakawa, Mr. Kouta Ikuma, Mr. Koichi Imagawa, Mr. Kohei Kubota, Mr. Atsushi Saito, Mr. Kyohei Sasanuma, Mr. Kazuhiro Taguchi, Mr. Hideaki Hyakuma, and Mr. Yuki Yokoyama are friends of mine, since we roomed in the same dormitory. During my research life, they encouraged me while drinking beer. This became my moral support when I was tough. I am very glad that I was able to meet them.

Finally, I would like to express my deepest gratitude to my family. Thank you always for everything, and I will definitely return the favor.

Bibliography

[1] Gordon E. Moore. Readings in computer architecture. chapter Cramming More Components Onto Integrated Circuits, pp. 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[2] Intel xeon phi. http://www.intel.com/.

[3] Tile-mx. http://www.tilera.com/.

[4] MPPA MANYCORE MPPA 256. http://www.kalray.com/.

[5] E.S. Chung, P.A. Milder, J.C. Hoe, and Ken Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pp. 225–236, Dec 2010.

[6] Tsubame2.5. http://www.gsic.titech.ac.jp/en.

[7] Ha-pacs. http://www.ccs.tsukuba.ac.jp/eng/research-activities/projects/ha-pacs/.

[8] Werner Augustin, Vincent Heuveline, and Jan-Philipp Weiss. Optimized stencil computation using in-place calculation on modern multicore systems. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pp. 772–784, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] E.H. Phillips and M. Fatica. Implementing the himeno benchmark with cuda on gpu clusters. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1 –10, april 2010.

[10] 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.

[11] A. Dundar, Jonghoon Jin, V. Gokhale, B. Martini, and E. Culurciello. Memory access optimized routing scheme for deep networks on a mobile coprocessor. In *High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pp. 1–6, Sept 2014.

[12] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Gopal Jan, Gray Michael, Haselman Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James

Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi, and Xiao Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pp. 13–24, Piscataway, NJ, USA, 2014. IEEE Press.

[13] K. Sano, Y. Hatsuda, and S. Yamamoto. Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth. *Parallel and Distributed Systems, IEEE Transactions on*, Vol. 25, No. 3, pp. 695–705, March 2014.

[14] Prabhat K. Gupta. Xeon+fpga platform for the data center. In *Intersections of Computer Architecture and Reconfigurable Logic (CARL 2015)*, 2015.

[15] M. Vestias and H. Neto. Trends of cpu, gpu and fpga for high-performance computing. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pp. 1–6, Sept 2014.

[16] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Found. Trends Electron. Des. Autom.*, Vol. 2, No. 2, pp. 135–253, February 2008.

[17] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-programmable Gate Arrays*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[18] Vaughn Betz, Jonathan Rose, and Alexander Marquardt, editors. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[19] A.A. Aggarwal and D.M. Lewis. Routing architectures for hierarchical field programmable gate arrays. In *Computer Design: VLSI in Computers and Processors, 1994. ICCD '94. Proceedings., IEEE International Conference on*, pp. 475–478, Oct 1994.

[20] Synopsys vcs. http://www.synopsys.com/products/simulation/simulation.html.

[21] Caence. http://www.cadence.com/us/pages/default.aspx.

[22] Mentor modelsim. http://www.mentor.com/products/fpga/model.

[23] Veritak. http://www.sugawara-systems.com/.

[24] Active-hdl. http://www.sugawara-systems.com/.

[25] Icarus verilog. http://iverilog.icarus.com/.

[26] Gplcver. http://gplcver.sourceforge.net/.

[27] Verilator. http://www.veripool.org/wiki/verilator.

[28] Ghdl. http://ghdl.free.fr/.

[29] Ise design suite. http://www.xilinx.com/products/design-tools/ise-design-suite.html.

[30] Vivado design suite. http://www.xilinx.com/products/design-tools/vivado.html.

[31] Quartus prime. https://www.altera.com/products/design-software/fpga-design/quartus-prime/overview.html.

[32] Lattice diamond.

http://www.latticesemi.com/en/Products/DesignSoftwareAndIP/FPGAandLDS/LatticeDiamond.

[33] V. Venugopal and D.M. Shila. High throughput implementations of cryptography algorithms on gpu and fpga. In *Instrumentation and Measurement Technology Conference (I2MTC), 2013 IEEE International*, pp. 723–727, May 2013.

[34] Xiang Tian and Khaled Benkrid. High-performance quasi-monte carlo financial simulation: Fpga vs. gpp vs. gpu. *ACM Trans. Reconfigurable Technol. Syst.*, Vol. 3, No. 4, pp. 26:1–26:22, November 2010.

[35] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, Wei Song, J. Mawer, A. Cristal, and M. Lujan. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pp. 1–8, Sept 2014.

[36] S. Windh, Xiaoyin Ma, R.J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W.A. Najjar. High-level language tools for reconfigurable computing. *Proceedings of the IEEE*, Vol. 103, No. 3, pp. 390–408, March 2015.

[37] Xilinx vivado hls.
http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[38] Altera opencl.
https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html.

[39] Cyberworkbench. http://www.nec.com/en/global/prod/cwb/index.html.

[40] Impulse c. http://www.impulseaccelerated.com/tools.html.

[41] Calypto catapult c. http://calypto.com/en/products/catapult/overview.

[42] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A java-compatible and synthesizable language for heterogeneous architectures. *SIGPLAN Not.*, Vol. 45, No. 10, pp. 89–108, October 2010.

[43] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: High-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pp. 33–36, New York, NY, USA, 2011. ACM.

[44] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, Vol. 13, No. 2, pp. 24:1–24:27, September 2013.

[45] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing modular hardware accelerators in c with roccc 2.0. In *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE*

Annual International Symposium on, pp. 127–134, May 2010.

[46] Synthesijer. http://synthesijer.github.io/web/.

[47] Eric S. Chung, James C. Hoe, and Ken Mai. Coram: An in-fabric memory architecture for fpga-based computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pp. 97–106, New York, NY, USA, 2011. ACM.

[48] Gabriel Weisz and James C. Hoe. Coram++: Supporting data-structure-specific memory interfaces for fpga computing. In *FPL*, pp. 1–8. IEEE, 2015.

[49] Shinya Takamaeda-Yamazaki, Kenji Kise, and James C. Hoe. Pycoram: Yet another implementation of coram memory architecture for modern fpga-based computing. In *Intersections of Computer Architecture and Reconfigurable Logic (CARL 2013)*, 2013.

[50] Vivado design suite user guide release notes, installation, and licensing. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug973-vivado-release-notes-install-license.pdf.

[51] Shimpei Sato and Kenji Kise. Archhdl: A novel hardware rtl design environment in c++. In Kentaro Sano, Dimitrios Soudris, Michael HuÌĹbner, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing*, Vol. 9040 of *Lecture Notes in Computer Science*, pp. 53–64. Springer International Publishing, 2015.

[52] Myhdl. http://www.myhdl.org/.

[53] P. Bellows and B. Hutchings. Jhdl-an hdl for reconfigurable systems. In *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pp. 175–184, Apr 1998.

[54] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pp. 1216–1225, New York, NY, USA, 2012. ACM.

[55] Shinya Takamaeda. A high-level hardware design environment in python (in japanese). *IEICE technical report*, Vol. 115, No. 228, pp. 21–26, sep 2015.

[56] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. *SIGPLAN Not.*, Vol. 34, No. 1, pp. 174–184, September 1998.

[57] D. Lockhart, G. Zibrat, and C. Batten. Pymtl: A unified framework for vertically integrated computer architecture research. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 280–292, Dec 2014.

[58] Peter Milder, Franz Franchetti, James C. Hoe, and Markus Püschel. Computer generation of hardware for linear digital signal processing transforms. *ACM Trans. Des. Autom. Electron. Syst.*, Vol. 17, No. 2, pp. 15:1–15:33, April 2012.

[59] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong,

F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. Spiral: Code generation for dsp transforms. *Proceedings of the IEEE*, Vol. 93, No. 2, pp. 232–275, Feb 2005.

[60] Hdl coder. http://www.mathworks.com/products/hdl-coder/.

[61] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. Optimus: Efficient realization of streaming applications on fpgas. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pp. 41–50, New York, NY, USA, 2008. ACM.

[62] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pp. 179–196, London, UK, UK, 2002. Springer-Verlag.

[63] Rishiyur Nikhil. Bluespec system verilog: efficient, correct rtl from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pp. 69–70, June 2004.

[64] Maxcompiler. https://www.maxeler.com/products/software/maxcompiler/.

[65] Flopoco. https://www.maxeler.com/products/software/maxcompiler/.

[66] Kentaro Sano, Hayato Suzuki, Ryo Ito, Tomohiro Ueno, and Satoru Yamamoto. Stream processor generator for HPC to embedded applications on fpga-based system platform. *Proceedings of the Second International Workshop on FPGAs for Software Programmers*, pp. 43–48, September 2014.

[67] Kentaro Sano. Dsl-based design space exploration for temporal and spatial parallelism of custom stream computing. *Proceedings of the Second International Workshop on FPGAs for Software Programmers*, pp. 29–34, September 2015.

[68] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: An evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pp. 451–460, New York, NY, USA, 2010. ACM.

[69] Riccardo Cattaneo, Giuseppe Natale, Carlo Sicignano, Donatella Sciuto, and Marco Domenico Santambrogio. On how to accelerate iterative stencil loops: A scalable streaming-based approach. *ACM Trans. Archit. Code Optim.*, Vol. 12, No. 4, pp. 53:1–53:26, December 2015.

[70] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–10, May 2009.

[71] K. Morgan. Applied iterative methods, l. a. hageman and d. m. young, academic press, new york, 1981. no. of pages: 386. *International Journal for Numerical Methods in Engineering*, Vol. 19,

No. 4, pp. 625–625, 1983.

[72] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, Vol. 52, No. 4, pp. 65–76, April 2009.

[73] Kentaro Sano, Wang Luzhou, Yoshiaki Hatsuda, Takanori Iizuka, and Satoru Yamamoto. Fpga-array with bandwidth-reduction mechanism for scalable and power-efficient numerical simulations based on finite difference methods. *ACM Trans. Reconfigurable Technol. Syst.*, Vol. 3, No. 4, pp. 21:1–21:35, November 2010.

[74] Wang Luzhou, K. Sano, and S. Yamamoto. Local-and-global stall mechanism for systolic computational-memory array on extensible multi-fpga system. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pp. 102 –109, dec. 2010.

[75] Ryohei Kobayashi and Kenji Kise. Scalable stencil-computation accelerator by employing multiple small fpgas (in japanese). *IPSJ Journal (ACS)*, Vol. 6, No. 4, pp. 1–13, oct 2013.

[76] Shinya Takamaeda-Yamazaki, Shintaro Sano, Yoshito Sakaguchi, Naoki Fujieda, and Kenji Kise. *Reconfigurable Computing: Architectures, Tools and Applications: 8th International Symposium, ARC 2012, Hong Kong, China, March 19-23, 2012. Proceedings*, chapter ScalableCore System: A Scalable Many-Core Simulator by Employing over 100 FPGAs, pp. 138–150. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[77] K. Sano, Y. Hatsuda, and S. Yamamoto. Scalable streaming-array of simple soft-processors for stencil computations with constant memory-bandwidth. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pp. 234 –241, may 2011.

[78] Kazuki SATO, Li JIANG, Kenichi TAKAHASHI, Hakaru TAMUKOH, Yuichi KOBAYASHI, and Masatoshi SEKINE. Performance evaluation of poisson equation and cip method implemented on fpga array (in japanese). *IEICE technical report. Circuits and systems*, Vol. 109, No. 396, pp. 19–24, jan 2010.

[79] O. Mencer, Kuen Hung Tsoi, S. Craimer, T. Todman, W. Luk, Ming Yee Wong, and Philip Leong. Cube: A 512-fpga cluster. In *Programmable Logic, 2009. SPL. 5th Southern Conference on*, pp. 51–57, April 2009.

[80] Masato Yoshimi, Yuri Nishikawa, Hideharu Amano, Mitsunori Miki, Tomoyuki Hiroyasu, and Oskar Mencer. Performance evaluation of one-dimensional fpga-cluster cube for stream applications (in japanese). *IPSJ Journal*, Vol. 3, No. 3, pp. 209–220, 2010-09-17.

[81] Hiroshi Inoue and Kenjiro Taura. Simd- and cache-friendly algorithm for sorting an array of structures. *Proc. VLDB Endow.*, Vol. 8, No. 11, pp. 1274–1285, July 2015.

[82] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent Kulandaisamy, and

Ruchir Puri. Paradis: An efficient parallel algorithm for in-place radix sort. *Proc. VLDB Endow.*, Vol. 8, No. 12, pp. 1518–1529, August 2015.

[83] Orestis Polychroniou and Kenneth A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pp. 755–766, New York, NY, USA, 2014. ACM.

[84] Badrish Chandramouli and Jonathan Goldstein. Patience is a virtue: Revisiting merge and sort on modern processors. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pp. 731–742, New York, NY, USA, 2014. ACM.

[85] Hari Sundar, Dhairya Malhotra, and George Biros. Hyksort: A new variant of hypercube quicksort on distributed memory architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pp. 293–302, New York, NY, USA, 2013. ACM.

[86] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, Vol. 7, No. 1, pp. 85–96, September 2013.

[87] Changkyu Kim, Jongsoo Park, Nadathur Satish, Hongrae Lee, Pradeep Dubey, and Jatin Chhugani. Cloudramsort: Fast and efficient large-scale distributed ram sort on shared-nothing cluster. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pp. 841–850, New York, NY, USA, 2012. ACM.

[88] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast sort on cpus and gpus: A case for bandwidth oblivious simd sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pp. 351–362, New York, NY, USA, 2010. ACM.

[89] R. Kobayashi and K. Kise. Face: Fast and customizable sorting accelerator for heterogeneous many-core systems. In *Embedded Multicore/Manycore SoCs (MCSoc), 2015 IEEE 9th International Symposium on*, pp. 49–56, Sept 2015.

[90] Face. https://github.com/monotone-RK/FACE.

[91] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Base-delta-immediate compression: Practical data compression for on-chip caches. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pp. 377–388, New York, NY, USA, 2012. ACM.

[92] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[93] Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting networks on fpgas. *The VLDB Journal*, Vol. 21, No. 1, pp. 1–23, February 2012.

[94] Valery Sklyarov and Iouliia Skliarova. High-performance implementation of regular and easily scalable sorting networks on an fpga. *Microprocess. Microsyst.*, Vol. 38, No. 5, pp. 470–484, July 2014.

[95] Ren Chen, Sruja Siriyal, and Viktor Prasanna. Energy and memory efficient mapping of bitonic sorting on fpga. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pp. 240–249, New York, NY, USA, 2015. ACM.

[96] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pp. 307–314, New York, NY, USA, 1968. ACM.

[97] Dirk Koch and Jim Torresen. Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '11, pp. 45–54, New York, NY, USA, 2011. ACM.

[98] George Marsaglia. Xorshift rngs. *Journal of Statistical Software*, Vol. 8, No. 14, pp. 1–6, 7 2003.

[99] Takuma Usui, Ryohei Kobayashi, and Kenji Kise. A challenge of portable and high-speed fpga accelerator. In Kentaro Sano, Dimitrios Soudris, Michael HuÌĹbner, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing*, Vol. 9040 of *Lecture Notes in Computer Science*, pp. 383–392. Springer International Publishing, 2015.

[100] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, Vol. 23, No. 3, pp. 337–343, May 1977.

[101] D.A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, Vol. 40, No. 9, pp. 1098–1101, Sept 1952.

[102] P. Deutsch. Deflate compressed data format specification version 1.3, 1996.

[103] Virtex-7 fpga vc707 evaluation kit. http://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html.

[104] Memory Interface Generator (MIG). http://www.xilinx.com/products/intellectual-property/mig.html.

[105] Nexys4 ddr artix-7 fpga board. https://www.digilentinc.com/.

[106] Xilinx kintex-7 fpga kc705 evaluation kit. http://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html.

[107] Takuma Usui, Ryohei Kobayashi, and Kenji Kise. Design and implementation of portable and high-speed fpga accelerator employing usb3.0 (in japanese). *IPSJ SIG Technical Reports. SLDM*, Vol. 2015, No. 36, pp. 1–6, jan 2015.

[108] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA

'14, pp. 151–160, New York, NY, USA, 2014. ACM.

[109] Marcela Zuluaga, Peter Milder, and Markus Püschel. Computer generation of streaming sorting networks. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pp. 1245–1253, New York, NY, USA, 2012. ACM.

[110] Large fpga methodology guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug872_largefpga.pdf.

[111] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In Kentaro Sano, Dimitrios Soudris, Michael HuÌĹbner, and Pedro C. Diniz, editors, *Applied Reconfigurable Computing*, Vol. 9040 of *Lecture Notes in Computer Science*, pp. 451–460. Springer International Publishing, 2015.

[112] Ply (python lex-yacc). http://www.dabeaz.com/ply/.

[113] Jinja. http://jinja.pocoo.org/.

[114] M. Nanjundappa, H.D. Patel, B.A. Jose, and S.K. Shukla. Scgpsim: A fast systemc simulator on gpus. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pp. 149–154, Jan 2010.

[115] S. Vinco, D. Chatterjee, V. Bertacco, and F. Fummi. Saga: Systeme acceleration on gpu architectures. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 115–120, June 2012.

[116] Hao Qian and Yangdong Deng. Accelerating rtl simulation with gpus. In *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*, pp. 687–693, Nov 2011.

[117] C. Roth, S. Reder, H. Bucher, O. Sander, and J. Becker. Adaptive algorithm and tool flow for accelerating systemc on many-core architectures. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pp. 137–145, Aug 2014.

[118] J. Howard, S. Dighe, S.R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V.K. De, and R. Van Der Wijngaart. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *Solid-State Circuits, IEEE Journal of*, Vol. 46, No. 1, pp. 173–183, Jan 2011.

[119] E.A. Carara, R.P. de Oliveira, N.L.V. Calazans, and F.G. Moraes. Hemps - a framework for noc-based mpsoc generation. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pp. 1345–1348, May 2009.

[120] H. Nakahara and T. Sasao. A deep convolutional neural network based on nested residue number system. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pp. 1–6, Sept 2015.

Publication

6.3 Journal Paper

1. **Ryohei Kobayashi**, and Kenji Kise: Scalable Stencil-computation Accelerator by Employing Multiple Small FPGAs (in Japanese), IPSJ Transaction on Advanced Computing Systems, Vol.6, No.4, pp.1–13 (October 2013).

6.4 International Conference Paper

2. Tomohiro Misono, **Ryohei Kobayashi**, and Kenji Kise, Effective Parallel Simulation of Arch-HDL Under Manycore Environment, International Symposium on Computing and Networking -Across Practical Development and Theoretical Research- (CANDAR), pp.140–146, (December 2015).
3. **Ryohei Kobayashi**, and Kenji Kise: FACE: Fast and Customizable Sorting Accelerator for Heterogeneous Many-core Systems, IEEE 9th International Symposium on Embedded Multicore SoCs (MCSoC-15), pp.49–56, (September 2015).
4. Eri Ogawa, Yuki Matsuda, Tomohiro Misono, **Ryohei Kobayashi**, and Kenji Kise: Reconfigurable IBM PC Compatible SoC for ComputerArchitecture Education and Research, IEEE 9th International Symposium on Embedded Multicore SoCs (MCSoC-15), pp.65–72, (September 2015).
5. **Ryohei Kobayashi**, Shinya Takamaeda-Yamazaki, and Kenji Kise: Towards a Low-Power Accelerator of Many FPGAs for Stencil Computations, Workshop on Challenges on Massively Parallel Processors (CMPP 2012) held in conjunction with ICNC'12, pp.343–349 (December 2012).

6.5 Domestic Conference Paper (with Review)

6. Encarnacion Immanuel Victoria, **Ryohei Kobayashi**, and Kenji Kise: 3bOS: A flexible and lightweight embedded OS operated using only 3 buttons, 組込みシステムシンポジウム 2014(ESS2014), pp.126–131 (October 2014).

7. **小林諒平**, 高前田 (山崎) 伸也, 吉瀬謙二: 多数の小容量 FPGA を用いたスケーラブルなステンシル計算機の開発, 先進的計算基盤システムシンポジウム SACSIS2013 論文集, pp.179–187 (May 2013).
8. **小林諒平**, 佐野伸太郎, 高前田 (山崎) 伸也, 吉瀬謙二: メッシュ接続 FPGA アレーにおける高性能ステンシル計算, 先進的計算基盤システムシンポジウム SACSIS2012 論文集, pp.142–149 (May 2012).

6.6 Technical Report

9. 奥村開里, **小林諒平**, 吉瀬謙二: SSD 内の並列性を引き出す I/O スケジューラ, 情報処理学会研究報告 2015-OS-135, No.14, pp.1–8 (November 2015).
10. **小林諒平**, 吉瀬謙二: FPGA を用いた世界最速のソーティングハードウェアの実現に向けた試み, 電子情報通信学会研究報告 RECONF, pp.65–70 (June 2015).
11. **小林諒平**, 吉瀬謙二: FPGA ベースのソーティングアクセラレータの設計と実装 **(優秀若手講演賞)**, 電子情報通信学会研究報告 CPSY, pp.25–30 (April 2015).
12. 臼井琢真, **小林諒平**, 吉瀬謙二: USB3.0 接続の手軽で高速な FPGA アクセラレータの設計と実装, 電子情報通信学会研究報告 RECONF, pp.205–210 (January 2015).
13. **小林諒平**, 吉瀬謙二: ソフトウェアチューニングと HW アクセラレータによるオプティカルフローの高速化 **(第 2 回 ARC/CPSY/RECONF 高性能コンピュータシステム設計コンテスト コンピュータシステム設計部門 優勝)**, 情報処理学会 FIT2014 情報科学技術フォーラム, pp.26–29 (September 2014).
14. **小林諒平**, 高前田 (山崎) 伸也, 吉瀬謙二: メッシュ接続 FPGA アレーを用いた高性能ステンシル計算の設計と実装, 電子情報通信学会研究報告 RECONF, pp.159–164 (January 2013).

6.7 Other Presentation and Poster

15. Takuma Usui, **Ryohei Kobayashi**, and Kenji Kise: A Challenge of Portablea and High-speed FPGA Accelerator, The 11th International Symposium on Applied Reconfigurable Computing (ARC2015), pp.383–392, (April 2015).
16. **小林諒平**, 吉瀬謙二: Ultra High-speed FPGA Accelerator for Sorting Application, 情報処理学会第 77 回全国大会, Vol.1, No.3A-03, pp.25–26 (March 2015).
17. **小林諒平**, 吉瀬謙二: Examination of HDL coding styles to reduce power consumption for FPGAs, 情報処理学会第 76 回全国大会, Vol.1, No.2A-7, pp.25–26 (March 2014).
18. **小林諒平**, 高前田 (山崎) 伸也, 吉瀬謙二: Design of Synchronization Mechanism to Conquer the Clock Oscillator Variation for High Performance Stencil Computation Accelerator, 情報処理学

会第 75 回全国大会, Vol.1, No.3K-7, pp.133–134 (March 2013).

19. **小林諒平**, 佐野伸太郎, 高前田 (山崎) 伸也, 吉瀬謙二: メッシュ接続 FPGA アレーにおけるステンシル計算の検討, 情報処理学会第 74 回全国大会, Vol.1, No.4J-4, pp.107–108 (March 2012).