UNIVERSITY OF CALIFORNIA SAN DIEGO

Fault-susceptibility Mitigation and Efficient Use of Resources in Programmable Hardware Accelerators

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Atieh Lotfi

Committee in charge:

Professor Rajesh K. Gupta, Chair
Professor Ryan Kastner
Professor Sorin Lerner
Professor Patrick Mercier
Professor Dean Tullsen

2018

The dissertation of Atieh Lotfi is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California San Diego

2018

DEDICATION

To my parents with everlasting gratitude

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This long venture of my PhD journey is a combined effort of many characters, including the scholars whom I had the privilege to meet and work with, family members who constantly supported me, and friends who were always accessible and guided me through this journey. All of them played a crucial role in one or another step of this journey and helped me to achieve my goals. None of my achievements, if any, would have been possible without the unconditional support and help from them. This thesis would therefore be incomplete without expressing my gratitude to all of them.

First and foremost, I owe immense gratitude to my advisor Professor Rajesh Gupta, who gave me the opportunity to conduct the research in an extremely free environment. I dedicate my deepest gratitude for his support, guidance, and encouragements.

I would also like to thank my committee members, Professor Ryan Kastner, Professor Sorin Lerner, Professor Dean Tullsen, and Professor Patrick Mercier for their advice and comments in my dissertation. I would also like to give special thanks to Dr. Nirmal Saxena and Dr. Philip Shirvani, who have given me tremendous advice and support while I was at NVIDIA.

I am blessed to be surrounded by so many good friends and colleagues during my Ph.D. time. I would like to thank many research colleagues who have made this dissertation possible. I would like to thank Abbas Rahimi for his mentorship during my first year. I dedicate especial thanks to Moein Khazraee for our brainstorming discussions, helps, and his encouragements. I thank my colleagues and friends Vahideh Akhlaghi, Jeng-Hau Lin, Manish Gupta, Dhiman Sengupta, Alex Forencich, Nishant Bhaskar, Jason Koh, Omid Assare, Armita Ardeshiricham, Alireza Khodamoradi, Quentin Gautier, Bahram Kherandmand, Shelby Thomas, Michael Barrow, Mohsen Imani, and Xun Jiao for their helps, feedbacks, and our discussions.

Finally, I would like to thank my parents, Homa Rahimi and Kazem Lotfi, for being strong pillars of support throughout my life. They have always encouraged me to explore and pursue opportunities, go above and beyond my potential, think big, and never give up. This dissertation

has all been made possible by their infinite patience. I dedicate special thanks to my lovely sister and brother, Atefeh Lotfi and Ali Lotfi, for their endless kindness and encouragements.

The material in this dissertation is based on the following publications.

Chapter 3 contains reprints of Atieh Lotfi, Abbas Rahimi, Luca Benini, and Rajesh Gupta, "Aging-Aware Compilation for GPGPUs", *ACM Transactions on Architecture and Code Optimization (TACO)*, 2015. The dissertation author is the primary author of this paper.

Chapter 4 contains reprints of Atieh Lotfi, Nirmal Saxena, Richard Bramley, Paul Racunas, Philip Shirvani, "Low Overhead Tag Error Mitigation for GPU Architectures", *International Conference on Dependable Systems and Networks (DSN)*, 2018. The dissertation author is the primary author of this paper.

Chapter 5 contains reprints of Atieh Lotfi, Abbas Rahimi, Amir Yazdanbakhsh, Hadi Esmaeilzadeh, Rajesh Gupta, "GRATER: An Approximation Workflow for Exploiting Data-Level Parallelism in FPGA Acceleration", *Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE)*, 2016. The dissertation author is the primary author of this paper.

Chapter 6 contains reprints of Atieh Lotfi, and Rajesh Gupta, "ReHLS: Resource-Aware Program Transformation Workflow for High-Level Synthesis ", *IEEE International Conference on Computer Design (ICCD)*, 2017. The dissertation author is the primary author of this paper.

Chapter 7, contains reprints of Jen-Hau Lin, Atieh Lotfi, Vahideh Akhlaghi, Zhuowen Tu, and Rajesh Gupta, "Accelerating Local Binary Pattern Networks with Software-Programmable FPGAs", *Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE)*, 2019. The dissertation author is the primary co-author of this paper.

VITA

2008	Bachelor in Computer Engineering, University of Tehran
2012	Masters in Computer Engineering, University of Tehran
2017	Intern, NVIDIA Corporation, Santa Clara, California
2018	Doctor of Philosophy in Computer Science (Computer Engineering), University of California San Diego

- Jeng-Hau Lin, Atieh Lotfi, Vahideh Akhlaghi, Zhuowen Tu, and Rajesh K. Gupta, "Accelerating Local Binary Pattern Networks with Software Programmable FPGAs", *Design, Automation & Test in Europe (DATE)*, 2019.

- Atieh Lotfi, Nirmal Saxena, Richard Bramley, Paul Racunas, Philip Shirvani, "Low Overhead Tag Error Mitigation for GPU Architectures", *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018.

- Atieh Lotfi, and Rajesh K Gupta, "ReHLS: Resource-Aware Program Transformation Workflow for High-Level Synthesis", *IEEE International Conference on Computer Design (ICCD)*, 2017.

- Atieh Lotfi, and Rajesh K Gupta, "RxRE: Throughput Optimization for High-Level Synthesis using Resource-Aware Regularity Extraction (Abstract)", *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays(FPGA)*, 2017.

- Tutu Ajayi, Khalid Al-Hawaj, Aporva Amarnath, Steve Dai, Scott Davidson, Paul Gao, Gai Liu, Atieh Lotfi, Julian Puscar, Anuj Rao, Austin Rovinski, Loai Salem, Ningxiao Sun, Christopher Torng, Luis Vega, Bandhav Veluri, Xiaoyang Wang, Shaolin Xie, Chun Zhao, Ritchie Zhao, Christopher Batten, Ronald G Dreslinski, Ian Galton, Rajesh K Gupta, Patrick P Mercier, Mani Srivastava, Michael B Taylor, Zhiru Zhang, "Celerity: An

Open-Source RISC-V Tiered Accelerator Fabric", *Symp. on High Performance Chips (Hot Chips)*, 2017.

- Atieh Lotfi, Abbas Rahimi, Amir Yazdanbakhsh, Hadi Esmaeilzadeh, and Rajesh K. Gupta, "Grater: an approximation workflow for exploiting data-level parallelism in FPGA acceleration", *Design, Automation & Test in Europe (DATE)*, 2016.

- Atieh Lotfi, Abbas Rahimi, Luca Benini, and Rajesh K. Gupta, "Aging-Aware Compilation for GPGPUs", *ACM Transactions on Architecture and Code Optimization (TACO)*, 2015.

- Atieh Lotfi, Abbas Rahimi, Hadi Esmaeilzadeh, Rajesh K. Gupta, "SqueezCL Squeezing OpenCL Kernels for Approximate Computing on Contemporary GPUs", Approximate Computing Workshop, 2015.

- Mehdi Semsarzadeh, Atieh Lotfi, Mahmoud Reza Hashemi, Shervin Shirmohammadi, "A fine-grain distortion and complexity aware parameter tuning model for the H. 264/AVC encoder", Signal Processing: Image Communication, 2013.

- Atieh Lotfi, Arash Bayat, Saeed Safari, "Architectural vulnerability aware checkpoint placement in a multicore processor", IEEE 18th International On-Line Testing Symposium (IOLTS), 2012.

ABSTRACT OF THE DISSERTATION

Fault-susceptibility Mitigation and Efficient Use of Resources in Programmable Hardware Accelerators

by

Atieh Lotfi

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2018

Professor Rajesh K. Gupta, Chair

Faced with the exponential growth in computing requirements, programmable hardware accelerators, such as GPUs and FPGAs, are becoming increasingly popular in high performance computing systems. In deference to energy efficiency and scalability challenges in these systems, it is crucial to efficiently use hardware resources while maintaining their reliability requirements. To meet system reliability requirements, traditional methods add redundancy in hardware or software. However, these redundancy-based error mitigation techniques suffer from inefficient use of hardware resources. The goal in this dissertation is to devise low-overhead approaches to mitigate the fault-susceptibility of hardware accelerators, and use their available resources efficiently.

For fault-susceptibility mitigation in GPU accelerators, this dissertation proposes a software-based approach that enables isolation of faulty components through task migration. Due to lack of configurable scheduler for GPUs, the proposed solution makes use of *introspective kernels* to enable effective task migration for isolating faulty components. This technique has very low overhead in terms of performance and energy and improves the accelerator lifetime and overall system cost. For FPGA accelerators, faulty component isolation is handled with a directive-based method through the synthesis tool.

This dissertation presents practical optimization methods to efficiently use the available resources on programmable hardware accelerators. These optimizations are performed at different levels of abstractions that are useful for GPUs and FPGAs, and the trade-offs among them are elaborated. For GPUs, optimization opportunities are explored in hardware-level and source-level. For FPGAs, optimizations are studied at the compiler-level, source-level, and algorithm-level. These optimization methods seek to remove unnecessary redundancies from program or hardware. This dissertation demonstrates *practical* and *efficient* approaches for utilizing fault-susceptible programmable hardware accelerators and improving their efficiency in terms of both cost per performance and energy.

Chapter 1

Introduction

With the end of the classical Dennard scaling, the benefits from continued transistor scaling are diminishing due to energy and power constraints, leading to prevalence of "dark silicon", that is, chips portions that must remain dark or unused due to limits of energy use or thermal limits [47] [40]. The dark silicon phenomenon documents diminishing returns in per-transistor speed and energy efficiency with increases in chip sizes. Further, the demand of compute and power intensive tasks are rapidly growing while the general-purpose CPU platforms are not able to keep pace with this increasing demand due to slowdown in scaling. Faced with the large growth in computing requirements and the dark silicon problem, chip designers have started using hardware accelerators to improve performance, cost, and energy efficiency. Among these, programmable hardware accelerators such as graphic processing units (GPUs) or field programmable gate arrays (FPGAs) have been gaining popularity.

For a long time GPUs were used for graphics and gaming purposes, and FPGAs were used for ASIC prototyping and base stations. However, these programmable accelerators are now being used in new systems ranging from embedded systems and IoT to high-performance computing (HPC) and data centers. These new systems create new challenges for these accelerators. The first challenge is that the penalty of inefficient use of their resources becomes more severe, and it becomes more important to efficiently use hardware resources. The second challenge is that the

probability of fault occurrence is increased due to their use at a large scale while there are no error mitigation methods for handling permanent faults in these accelerators.

Regarding the first challenge, these accelerators have complicated programming model. Developing a parallel program for GPUs that considers regular memory access patterns and suitable memory managements is complicated. Moreover, writing a program for FPGAs either by hardware description languages or through the use of high-level synthesis tools requires hardware-aware programming. If our programs are not hardware-friendly, we lose the benefits of using hardware accelerators. In addition, we have seen fast-paced development of code by software developers, e.g. in cloud applications, resulting in codes that are not optimized and tuned for the target platform. Executing an un-optimized code on these accelerators might result in inefficient use of their resources. Therefore, there is a need for automatic tools to further optimize the code and avoid inefficiencies that they may cause.

Regarding the second challenge, these accelerators have no error mitigation method for addressing permanent faults. This means that these accelerators are no longer usable if they become partially defective. This results in disposal of partially defective accelerators which is costly. Various factors such as variation in manufacturing, aging effects and wearout mechanisms, and dynamic variation such as temperature fluctuation and voltage drops may initiate and accelerate the frequency of faults and system failure [11] [53]. As we go towards smaller transistor technologies making a perfectly working die becomes even harder which results in increasing costs of chips by a large factor [62]. In fact, hardware resource failures and down times is a critical concern for large scale high-performance computing platforms, as they adversely affect the performance and quality of service in a system [104].

Since reliability is an important factor in hardware design, different approaches exist to detect and recover from failure. Several approaches add redundancy to perform recovery from failures. These methods incorporate extra components, instructions, or data in the design of a system so that the functionality is not impaired in the event of a failure. Possible ways to build a redundantly reliable system include:

1. Hardware redundancy: These methods replicate all or some part of the hardware device. For example, triple modular redundancy (TMR) [78] is one technique in which the critical components are replicated three times and the results are voted to produce the output. The redundant component can be active at all time and a voter decides the correct output (static hardware redundancy), or the spare components become activated upon the failure of a currently active component (dynamic hardware redundancy). Even though hardware-based redundancy approaches can detect and correct errors fast, they are very expensive in terms of their impact on circuit area, energy, and cost of the system. In addition, for memory elements, redundancy is augmented into data bits so that an error in the data can be detected or corrected. Example of these techniques are using parity bits [54] or error correction codes [34].

2. Software redundancy: These methods perform redundant code execution. For example, SWIFT [98] executes each instruction twice on replicated inputs and compares their outputs. In N-Version programming [36], *N* independently designed versions of a software is executed either sequentially or in parallel and their results are compared by a decision algorithm. While software-based techniques can be more flexible than hardware-based approaches, they have considerable cost in terms of performance and energy efficiency [58].

Hardware and software redundancy add significant overhead on cost, performance, and energy. These approaches repeat a part of computation and this increases the overall system energy consumption. Applying redundancy-based methods are necessary for safety critical applications such as autonomous driving and avionics, but they are not desirable in most other systems. For example, energy efficiency in embedded systems or performance and scalability requirements in high performance computing data centers can not afford these overheads [72].

An alternative approach is core isolation and task migration. This approach works for systems that have a number of components that can work in parallel. Here, the faulty core is isolated, and the tasks running on a faulty core are migrated to another non-faulty one as

soon as a failure is predicted or detected [72]. This approach isolates the faulty component and makes use of other available functional resources in the system to perform its task. For example, [91] [90] [108] [56] propose dynamic workload allocation policies to mitigate core failures in multicore architectures. In these works, the operating system effectively changes workload scheduling method to isolate a faulty core and distribute its tasks to other available cores in the system. In general, this approach has lower overhead than redundancy-based approach, since it does not have expensive redundancies, there is no task duplication that needs to be run at all time, and the fault-affected hardware is still being used. Naturally, this approach is only possible if adequate hardware resources are still available.

In fact, both GPUs and FPGAs are great targets for implementing isolation due to their architectures. GPU architecture contains many parallel cores that run in the Single Instruction Multiple Data (SIMD) fashion. FPGA provides a customizable architecture containing many primitive cells that can be configured through placement and routing. For FPGA accelerators, the synthesis tool allows isolation of faulty components with fine-grained granularity. However, due to lack of operating systems and configurable scheduler for GPUs, core isolation and task migration is not possible for this platform. This result in disposal of partially defective accelerators, which is expensive in terms of cost and replacement time especially in data centers and IoT devices. Enabling error mitigation on accelerators can open up opportunities to use hardware which were considered unusable due to their conditions for a longer time; leading to reducing waste of available hardware resources, increasing the lifetime of defective hardware, as well as reducing the manufacturing cost by accepting more range of defective hardware. Reducing cost while keeping a system energy efficient can significantly help with today's exponentially growing computational requirements.

In this dissertation, we address the aforementioned two challenges. First, we add the missing fault mitigation and isolation support for these accelerators. We develop automatic software-level and directive-based methods to implement isolation and task migration with very low overhead in terms of performance and energy. By isolating faulty components, we can make

use of other available components in the hardware accelerator, and improve the cost and lifetime of hardware. Moreover, efficient use of hardware resources becomes more advantageous in the new use cases of these accelerators. To improve efficiency of using GPU and FPGA accelerators, we perform optimizations in different levels of abstractions that are useful for each of them. For GPUs, we seek opportunities to perform optimization in hardware-level and source-level, to remove unnecessary overheads in hardware or optimize the program running on GPU. For FPGAs, we target optimizations in compiler-level, source-level, and algorithm-level. These optimizations either improve efficiency of high-level synthesis design process, tune the program for hardware, or use a hardware-friendly algorithm and implementation.

To improve efficiency of using GPU accelerators, we seek methods to remove unnecessary redundancy in hardware and software. For removing unnecessary hardware redundancy, we perform a case study to seek opportunities to remove some reliability-related redundancy and replace them with a more efficient design choice. Traditionally, memory structures are augmented with reliability-related redundancy in order to become resilient to errors, which results in area and energy overhead. We study the necessity of these redundancies and the resiliency of unprotected caches in GPU. We further study the usefulness of a very low overhead error mitigation to replace parity bits in some structures. For removing unnecessary redundancy in software, we perform automated optimization of programs that are developed for GPUs. Especially, with the growth in the number of software developers who are not familiar with low-level hardware details, there exists unnecessary computations in these programs. We develop a source-to-source compiler that selectively tunes the bitwidth of variables and operations in a program, and results in efficient use of hardware resources. This optimization is also useful for FPGA accelerators, in which the automatic transformation is done on the high-level design specification. In addition to this optimization, we present two more optimization methods for efficient use of FPGA resources. For improving the efficiency of synthesis results through high-level synthesis (HLS) process, we perform automated transformation in the design specification through compiler and enable task migration and resource sharing without any modification in the HLS tool. This

transformation results in more efficient use of FPGA resources. We also present a case study that uses algorithmic modification to increase efficiency of using FPGA hardware accelerators. All presented approaches can be used to improve efficiency in either a fully functional or faulty hardware accelerator.

1.1 Dissertation Organization and Contributions

In this dissertation, we focus on addressing the two important challenges that are caused by the new use cases of programmable accelerators. We present methods that enable fault mitigation through isolation for GPUs and FPGAs. We further present optimization techniques that result in more efficient use of hardware resources for these commodity programmable accelerators. This dissertation has two main parts. In the first part, we focus on presenting fault mitigation and optimization methods for efficient use of resources in GPU accelerators. First, we propose an efficient method to enable isolation and task migration on GPU accelerators, with very low overhead in terms of performance and energy. Then, we study and propose techniques for more efficient utilization of resources using hardware and software optimizations. The second part, focuses on optimization for FPGA accelerators. We target two optimization approaches that automatically modifies the FPGA design specification. The first approach removes unnecessary computations in the design, and the second one exploits resource sharing opportunities by regularity extraction. We also study the effect of algorithmic optimization on the efficiency of FPGA accelerators. Figure 1.1 illustrates the scope and organization of this dissertation.

The contributions of this dissertation are:

1. Introducing primitive cell isolation for defective FPGAs by a directive-based approach,
2. Enabling component isolation and task migration for faulty GPUs through compiler,
3. Automatic source-level optimization and program transformation for programmable accelerators that results in better resource utilization in hardware,

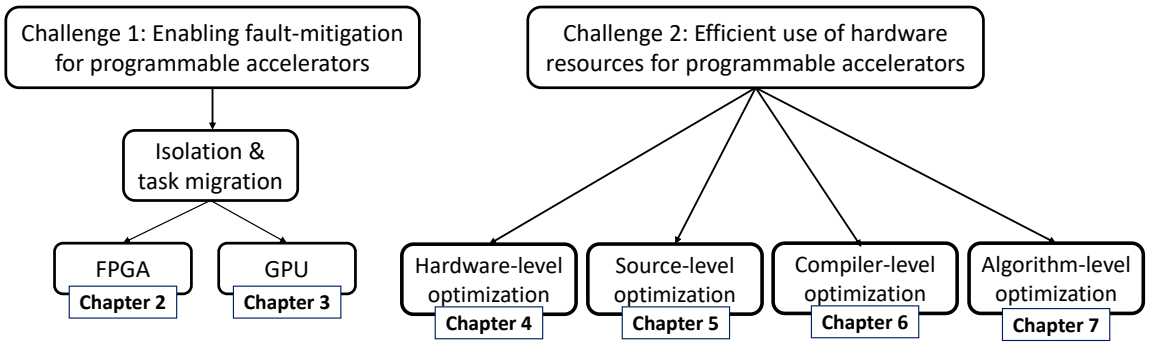


Figure 1.1: Dissertation organization.

4. Enabling automatic task migration to improve resource efficiency and throughput of HLS-based FPGA design,

5. Presenting and evaluating optimizations in different levels, from hardware to algorithm, for efficient use of hardware resources, and analyzing their trade-off and impact.

The organization of this dissertation is as follows:

First, we start by providing background information on GPU and FPGA architecture and design flow in Chapter 2. We also discuss how we can guide the synthesis tool to enable isolation of faulty primitives in FPGAs through a directive-based approach.

For GPU platforms with faulty components, we enable faulty components isolation and task migration in the granularity of stream cores using a just-in-time compilation method. We propose innovations in the static compiled code by introducing the notion of *introspective kernels*. An introspective kernel adaptively monitors the health of a GPU device and triggers runtime workload reallocation scheme to migrate tasks on healthy stream cores. After detection of faulty stream cores (SCs), a just-in-time compilation process replaces the introspective kernel with a *healthy* kernel that responds to the specific health state of the GPU device. This method can efficiently isolate faulty stream cores, and migrate their tasks to other functional cores. We further show the benefit of this approach for a GPU with aged stream cores. By shifting the workload from aged stream cores to healthy ones, the faulty hardware components are gradually healed. This results in efficient use of hardware components and increases the device lifetime

which reduces the overall hardware cost. We discuss this approach in **Chapter 3**.

As mentioned earlier, it is becoming more important to use resources on programmable hardware accelerators in a more efficient way. In the rest of dissertation, we propose methods at different levels to efficiently use the available hardware resources. We perform a case study to find out if there are any redundancy in the design of hardware accelerators that can be removed or replaced with a more efficient design choice. Traditionally, memory structures are augmented with reliability-related redundancy in order to become resilient to errors, which results in area and energy overhead. We evaluate the necessity of these redundancies and the reliability of unprotected caches in GPU. We further study the usefulness of a negligible overhead tag error mitigation mechanism to replace parity bits. This method distributes memory accesses more efficiently with a new cache indexing mechanism to mitigate some pathological address strides that cause error and also increases memory throughput. At a negligible impact on resiliency, this architecture eliminates the need for parity protection in the cache tag SRAM structures. This approach is general and does not induce any constraints or inflexibility to the system and can be integrated in newer generation of accelerators. This approach, which is discussed in **Chapter 4**, can be applied to any system that uses cache memory, including off-the-shelf GPUs.

We further present a source-level optimization approach that targets both GPU and FPGA. This technique identifies and removes unnecessary redundancy in programs developed for GPUs or design specifications developed for FPGAs. The unnecessary redundancy in programs might be caused by two reasons. First, these programs might be developed by software developers who are not familiar with low-level hardware details. Second, the application might have inherent tolerance to some degree of inexactness which can provide further optimization opportunity. Optimizing the program manually, especially in the second case, is not easy. In **Chapter 5**, we introduce GRATER which is a source-level automatic optimizer to reduce unnecessary computations. Our approach automatically simplifies expensive computations in the program. This results in improved performance and reduced hardware resource requirements. For example, by performing fixed-sized computations instead of floating point computations, using single-precision

floating point instead of double-precision floating point, or fewer number of bits for integer computations, GRATER can improve resource utilization and performance. These modifications are all done automatically during compile time without any programmer intervention. GRATER transcompiler is useful for both GPU and FPGA platforms. This optimization method is fast, requires no effort from user, and can be applied to any program or design specification. This approach can be applied to improve the efficiency in both fully functional and faulty hardware accelerator.

To improve the efficiency of HLS-based FPGA design, we present an approach to automatically find reuse opportunities in the design specifications with inherent regularities and implicitly change the scheduling. This approach uses compiler frontend as an independent preprocessing step to explore the design space and adds an automated source-to-source transformation step before HLS. In particular, it shows how inherent regularity in applications can be used to construct a workflow that analyzes the specification, explores the design space for resource optimization opportunity, and transforms the program accordingly. The transformed program can be synthesized using the HLS tool. This approach, which is explained in **Chapter 6**, takes advantages of resource sharing opportunities to reduce resource utilization while keeping latency and energy efficiency similar to the original design.

Moreover, we perform a case study to examine the impact of algorithmic modification to increase efficiency of FPGA accelerators. For the growing field of deep neural network applications, in **Chapter 7**, we show resource saving and power reduction advantages of using a more hardware-friendly algorithm and its harware-aware implementation over off-the-shelf convolutional neural networks. This approach, despite its longer development time, is the most effective solution for improving resource utilization of hardware accelerators. However, this approach is cumbersome compared to automated source-level and compiler-level techniques. Moreover, an algorithmic optimization of this kind does not necessarily work for other applications. This approach indicates that using hardware-friendly algorithms improves the efficiency, in exchange for high development cost. In fact, this is the most effective approach for efficient

use of hardware resources, since we are considering the full stack from algorithm to hardware.

Finally, **Chapter 8** concludes the dissertation and gives future directions. We argue that the use of our source-level and compiler-assisted optimization alongside the isolation and task migration techniques is a *general* approach that can be *easily* applied to fault-susceptible commodity programmable hardware accelerators and improve their efficiency in terms of both cost per performance and energy.

Chapter 2

Background

This section introduces some preliminary concepts and definitions useful to briefly build a background for this work. First, we provide background information on FPGA architecture and design flow. Further, we discuss the directive-based method to enable isolation of faulty primitives in FPGAs. Then we discuss OpenCL programming model which can be used for both FPGA and GPU accelerators. Furthermore, we briefly discuss a sample GPU architecture that we mostly used in the dissertation.

2.1 Background on FPGA and Synthesis Process

Field Programmable Gate Array or **FPGA** is a reconfigurable hardware platform which can be configured after manufacturing. FPGAs are composed of programmable logic and memory blocks which are connected using programmable interconnects. FPGA reconfiguration allows it to perform a wide variety of complex tasks. The basic structure of an FPGA is composed of elements like look-up table (LUT) (which performs logic operations), Flip-Flop (FF) (register element to store the result of operations), memory blocks (BRAM), and digital signal processing (DSP) blocks that perform specific functions. The type and number of elements available on an FPGA depends on the vendor, family, and specific device. FPGA is mainly an island of

elements arranged in a two dimensional grid of sections called *tiles*. There are several different types of tiles: CLB, DSP, BRAM, and interconnect. CLBs (Configurable Logic Blocks) are the resources for implementing logic on the FPGA. A CLB occupies a single tile on the device and is connected to a switch matrix to access the general routing structure. Each CLB is comprised of number of interconnected slices. Many of the tiles can be broken down into smaller components called primitive types. Primitive types are the smallest unit on FPGA. Each primitive is identified by its (X,Y) location on the device (e.g SLICE_X39Y53, DSP48A_X1Y7). One example of a primitive type found within a tile is a slice. For example, in our target FPGA, each slice is made up of two LUTs, two storage units, wide-function multiplexers, carry logic, arithmetic gates, and routing interconnect. Another primitive type is DSP slice. Each slice supports many independent functions, including multiplier, multiplier followed by an adder, or barrel shifter. The slices can also be connected together to form wide math functions. These function could be implemented using more general logic resources such as CLBS. However, the use of DSPs can decrease the amount of general logic resources resulting in higher performance, and efficient device utilization.

High-Level Synthesis (HLS), also known as behavioral synthesis or algorithmic synthesis, is a design process that given a high-level behavioral specification of a digital system and a set of constraints, automatically generates a Register-Transfer Level (RTL) structure that implements the desired behavior [79]. Different HLS tools use different high-level behavioral specification languages like C, C++, SystemC, or OpenCL, which are untimed or partially timed algorithmic descriptions, to describe the design. This description is transformed to a fully-timed and bit-accurate RTL implementation by the HLS flow. HLS process usually consists of a number of tasks which are done in different steps. HLS front-end performs lexical processing, control and data flow analysis, and optimizations in order to build and optimize an intermediate representation to be used in the subsequent steps. In HLS synthesis step, the design decisions are taken, to obtain an RTL description of the target architecture that satisfies the design constraints. By *scheduling*, *resource allocation*, and *resource binding* the number and type of hardware

modules to be used is established and each instruction is scheduled and assigned to one of the hardware resources that can execute it. Finally in the last step of HLS, an RTL implementation is generated containing control and steering logic circuits.

Logic Synthesis and Implementation is the process to synthesize the RTL design and implement the final design on FPGA. Synthesis optimizes a design for a specific implementation target, such as a FPGA chip or a semi-custom ASIC chip. The outcome of high-level synthesis is typically a structural netlist. The netlist contains the primitives on the FPGA and the connections between them. Implementation is the process of taking the previously generated netlist and preparing the design to be configured onto a specific device. It consists of a number of steps as follows. Mapping is the process of pairing the generic logic to the specific primitives (slices, BRAMS, etc.) in the target FPGA. After mapping, placement and routing is performed. During placement, the placer assigns each primitive to a physical site on the device. Then it routes, or connects, the components using the wires in the FPGA as defined by the netlist. Placement and routing typically use sets of heuristics to achieve optimization objectives in an expeditious manner. A placement takes as it inputs the netlist, together with a device map showing the location of each of its functional units, in order to select a legal location on the FPGA for each functional block in the netlist, such that the routing of these blocks is optimized. In general, synthesis tools allow some freedom in the users preference of the placement of circuit. The design is then passed on to the bitstream generator which generates all the information needed to configure a design onto a device.

2.2 Isolating faulty primitives in FPGA

We earlier stated that faulty component isolation on FPGA is straightforward, here we describe how it can be done. We employ a mechanism to periodically detect permanent faults on FPGAs using testing methods [37]. If a permanent fault is detected, the system identifies and reports the exact location of the fault [82, 89]. As soon as the location of the faulty primitive

is known, we can isolate it on FPGA, and perform placement and route again. In order to isolate the faulty block in FPGA, we can use the capability that the synthesis tool provides us to include specific primitive cells in the design. To make sure our design is not mapped to the faulty primitive, we can force the placement process to use it for a dummy operation that we intentionally add to the design. For example, in Xilinx synthesis tool, to isolate a faulty primitive, we define a dummy element in the RTL design and put a **KEEP** directive on it. This way the synthesizer will keep this dummy element in the design despite the fact that it is not connected anywhere and thus would be normally removed by synthesis tools. We then force this dummy element to be placed on the faulty primitive using **set_property LOC** command in the constraint file. This way none of our blocks in the design is placed on that faulty primitive. This process can be easily automated by generating a simple script.

2.3 OpenCL Execution Model on GPU and FPGA

OpenCL is a standard framework for developing parallel programs that execute across heterogeneous platforms consisting of GPUs and FPGAs. OpenCL uses a subset of ISO C99 with added extensions for supporting data and task-based parallel programming models. The programming model in OpenCL comprises of one or more device kernel codes in tandem with the host code. The host code typically runs on a CPU and launches kernels on other compute devices like the GPUs and/or FPGAs through API calls. These kernels execute on compute devices that are a set of compute units (CUs), each comprising of multiple processing elements having ALUs. Each instance of the OpenCL kernel is called a work-item. The work-items execute on a single processing element and exercise the ALU. The OpenCL platform model from the programming model to the framework of the compute devices is illustrated in Fig. 2.1. To launch a kernel, the programmer determines a group of work-items to execute on the device which is referred as an ND-Range. A group of work-items, typically 256 work-items, form a work-group that shares a local memory space. Work-items from one work-group cannot access

the local memory of other work-groups. Work-items are further grouped into wavefront which is composed of 64 work-items, as the unit of scheduling.

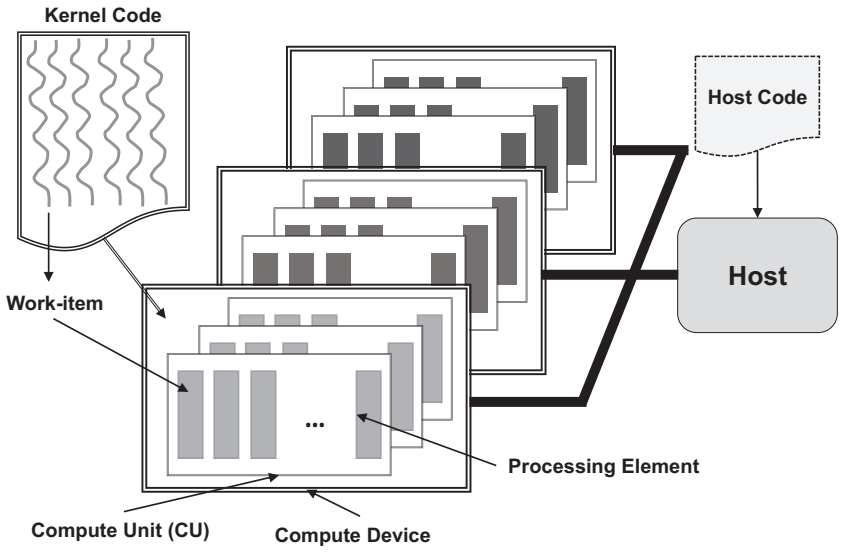


Figure 2.1: OpenCL platform model.

GPUs and FPGAs exploit data-level parallelism differently. GPUs are single-instruction multiple-data (SIMD) devices that exploit data-level parallelism: they group processing elements in a CU to perform the same operation but on their own individual data. On the other hand, FPGAs exploit pipeline parallelism in a CU where different stages of the instructions are applied to different work-items concurrently. FPGAs can further improve the performance benefits by creating multiple copies of the kernel pipelines (synthesized version of an OpenCL kernel). As the kernel pipelines can be executed independently from one another, the performance would scale linearly with the number of copies created owing to the data-level parallelism model supported by OpenCL. Altera OpenCL SDK [1] and Xilinx SDAccel [6] allow programmers to use high-level OpenCL kernels to generate an FPGA design with high performance per Watt [19].

2.4 GPU Architecture

GPUs are large parallel structure of processing cores. The original intent when designing GPUs was to use them exclusively for graphics rendering that often carry same operations on multiple data items and the processed data is usually destined for termination on a screen buffer. However, the massive parallelism of compute operations was quickly recognized as useful beyond large scale graphics rendering, into solving scientific computational problems. To serve these needs, GPUs evolved into General Purpose GPUs or GP-GPUs that feature an instruction set sufficient to carry one entire computation without the need for a "main" general purpose processor to be GPU would normally be an accelerator. In the big picture, the GPU consists of many small processors and has its own data storage hardware. The main GPU vendors, NVIDIA and AMD, use different terminologies for their structures. In this dissertation, we mainly use an AMD GPU for our experiments, expect from chapter 4 that we target an NVIDIA GPU. Since the high-level architecture of GPUs are similar, we introduce our target GCN-based AMD GPU architecture here.

2.4.1 AMD GCN architecture

AMD Graphics Core Next (GCN) architecture is a RISC SIMD architecture that replaces the older VLIW SIMD architecture. In this dissertation, we target Radeon HD 580 RX (Ellesmere) device which has 36 compute units (CUs). The block diagram of this architecture is shown in Fig. 2.2. Every CU has four SIMD units and a wavefront scheduler. Each of the four SIMD units can be scheduled independently. The CU has its own hardware scheduler that is able to assign wavefronts to available SIMD units with limited out-of-order capability to avoid dependency bottlenecks. Each SIMD unit has 16 stream cores (SCs); therefore, it brings a total number of 64 SCs per CU and 2304 SCs per Ellesmere device. The CU has a scratchpad memory, where OpenCL local memory is allocated. In these GPUs, sixteen work-items are executed in SIMD fashion, and the whole wavefront (64 work-items) is executed over four clock cycles.

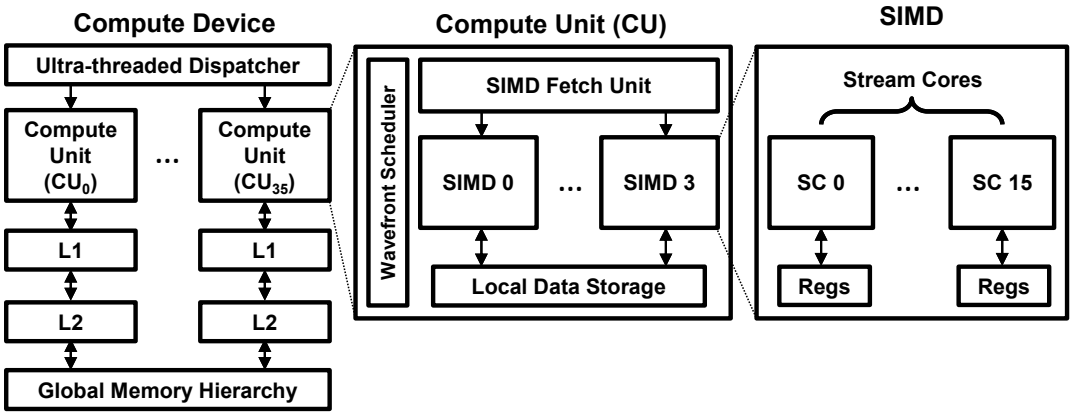


Figure 2.2: Block diagram of the Radeon GCN architecture

Therefore, a work-group is comprised of up to 4 wavefronts that share the execution resources in a CU. To manage these resources, a wavefront scheduler dynamically selects wavefronts for execution. For efficient hardware utilization, the work-item count should be an integer multiple of 64. Each CU executes one or more work-groups at a time. When the CPU launches an OpenCL kernel into the GPU, the work-groups are mapped into the CUs until all of them reach to their maximum occupancy. When a work-group finishes execution, the associated CU allocates a new waiting work-group, and this process is repeated until the entire ND-Range is executed.

Chapter 3

Enabling Task Migration to Isolate Corrupted Cores in GPUs

Graphic Processing Units (GPUs) offer high computational throughput using hundreds of parallel cores. As technology scales down and device dimensions near atomic scales, manufacturing features are no longer as ”chiseled” as in larger dimensions. This makes devices increasingly more susceptible to different types of errors occurred during manufacturing or lifetime of device. In this chapter, we present a software-based methodology to isolate faulty units and mitigate hardware failures of GPUs. This method enables task migration and rescheduling to quarantine the defective hardware units and ensure correct execution. The compilation strategy along with proposed *introspective* and *healthy* kernels can adaptively shift the workload from less reliable units to more reliable units. By isolating defective units, we can make use of other available units and improve the cost and lifetime of the GPU. We evaluate the effectiveness of the proposed method for various OpenCL kernels on AMD GPU architectures.

3.1 Introduction

Due to many factors such as transistor scaling, low voltage, and high frequency, todays processors are more than ever subject to fallouts. Defects can happen during the manufacturing process, some of which might not be detected during factory testing and might progressively be detected during processing. Variability of process parameters in conjunction with aging caused by non-uniform stress is also one source of failure for GPUs with thousands of cores [12]. If a core in GPU gets erroneous, it should no longer perform any task, or otherwise the GPU cannot ensure functional correctness. In fact, the corrupted core should be isolated and its workload should be assigned to other functional cores. However, scheduling and allocation is embedded in GPU hardware, and it can not be modified. Therefore, with current GPUs, the device can no longer be used in case any error is detected.

Parallel execution in GPUs provides an important ability to reallocate workloads in response to the health state of the system. To increase the lifetime of GPU, we introduce a software approach to enable rescheduling and faulty core isolation. Accordingly, we make the following main contributions:

- We propose innovations in the static compiled code by introducing the notion of *introspective kernels*. An introspective kernel adaptively monitors the health state of a GPU device and triggers runtime workload reallocation scheme. On detection of corrupted cores, a just-in-time compilation process replaces the introspective kernel with a *healthy* kernel that responds to the specific health state of the underlying GPU device.

- To isolate the faulty cores, the healthy kernel is customized to seamlessly bypass the workload from the degraded cores by shifting the workload to the healthy counterparts. To reduce the performance penalty due to the time-multiplexing of available cores, the generated healthy kernels can be further tuned according to the number of degraded cores.

- We further show the benefit of this approach for a GPU with aged cores. By shifting the

workload from an aged core to healthy ones, the aged hardware units can be gradually healed. This reduces the overall hardware cost due to efficient use of hardware components and increases its lifetime and improves system cost.

We evaluate our technique on the the AMD graphics core next (GCN) architecture introduced in chapter 2. We implement our approach targeting OpenCL (Open Computing Language) applications. OpenCL kernels can be executed on both AMD and Nvidia GPUs [87]. Moreover, our technique can easily be extended for CUDA-based platform.

3.2 Detecting and locating faulty cores

Our approach can be used for isolating cores in the event of intermittent and permanent faults or prediction of timing-induced faults. This type of faults can be caused by defects or variations during the manufacturing process or aging during the lifetime of system. Due to aging, the threshold voltage of a transistor increases, which causes delay-induced failures and raises the propagation delay of logic gates over time. Due to static variation and different load on different compute units, they age with different pace. The device lifetime is limited by the most aged component in the chip.

We assume that only compute units are vulnerable as memory in GPU is protected by parity or ECC bits. The objective is to build a map indicating which core is reliable and which one is unreliable. For detecting and locating faults, we either can use functional testing techniques, or delay monitoring sensors. The health result through testing or sensors are written in memory and it can be accessed by software.

The first fault detection and location solution is to launch tests at a regular pace. This test consists of a series of benchmarks targeting error detection such as [33] combined with the ability of locating faulty unit at hardware level. The second solution is to use delay monitoring sensors. To ensure necessary observability for non-uniform aging degradation, *in situ* delay monitoring sensors with digital outputs have been proposed and validated on silicon [105]. These

sensors enable high-volume data collection to guide dynamic management schemes and warn of impending device failure. Using compact delay monitoring sensors [105] that provide ΔV_{th} measurement with 3σ accuracy of 1.23 mV for a wide range of temperature, enables large scale data-collection across all the components. Test chips that has been fabricated efficiently consider multiple sensors banks containing up to total 256 NBTI sensors, hence the power overhead of laying out thousands of these sensors would only be a few hundreds of μW at maximum, which is a small fraction of power in our case [105].

3.3 Proposed methodology to quarantine faulty cores

We propose a compilation strategy to enable isolation of faulty stream cores and increase the lifetime of a GPU device through adaptive workload shifting from degraded SCs to other available SCs. A run-time system needs to observe the current health state of SCs to be able to adapt the kernel code accordingly. As described in the previous section, this information is written in part of memory that can be read by software. The AMD Compute Abstraction Layer (CAL) provides an easy-to-use interface to the parallel processor arrays found in AMD GPUs. CAL, part of the AMD accelerated parallel processing software stack, abstracts the hardware details of the AMD stream processor. CAL provides a device driver library that allows applications to interact with the stream cores at the lowest-level. We re-factor the naive kernel code by inserting a custom API, `check_degradation_status()`, to access the test/sensor measurements. This new version of the naive kernel is called an *introspective kernel*, in which every work-item investigates the degradation information of its corresponding SCs. The introspective kernel can query to check health information in memory to find out whether the SC used by the work-item is faulty or not by calling `check_degradation_status()`. Figure 3.1 illustrates the overall compilation flow for adapting kernels. The introspective kernel identifies the reported amount of degradation, and consequently SC isolation might be triggered.

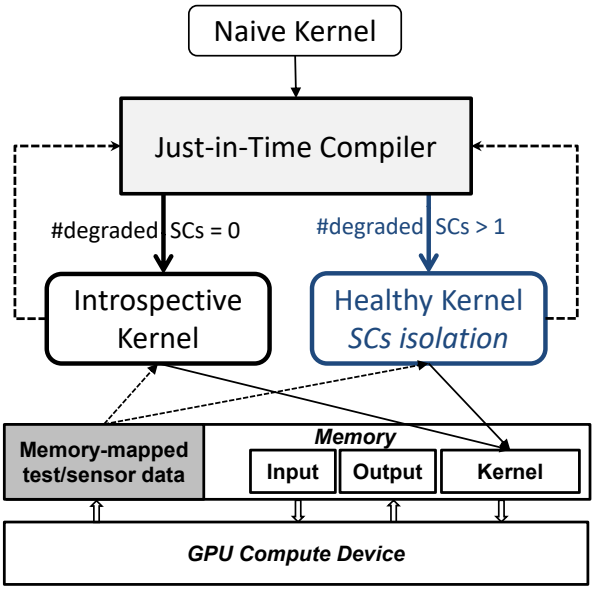


Figure 3.1: Kernel adaptation flow

3.3.1 Stream core isolation

The key idea of SC isolation method is to generate healthy kernels by modifying the normal distribution of workload so that the faulty SC is isolated. This is done by adaptively idling faulty SCs and assigning its work-items to the other healthy SCs in the same CU. For any given kernel, an introspective kernel is compiled and executed. When any of the SCs is degraded, it should be isolated and the workload should be removed from it. Therefore, for each naive kernel, a healthy version is generated in which all the work-items from those degraded SCs are moved to the other healthy SCs within the same CU. Since in an OpenCL kernel, there is no explicit mapping between a work-item and a SC, a set of extra work-items are spawned that exactly perform the same task as those on the degraded SCs. The work-items that have been assigned to any of the degraded SCs will not perform any operation.

Considering the adaptation flow in Fig 3.1, when the introspective kernel runs, each work-item checks the memory corresponding to the test result or the output of delay monitoring sensor for that SC. Fig 3.2 shows the code snippet for the introspective kernel. Besides the normal execution of the naive kernel, the introspective kernel reports the required number of

```
__kernel void IntrospectiveKernel (... default_parameters ...)
{
        //naive kernel execution
        kernel(default_parameters);

        if (check_degradation_status() > 0)
                RWIs++; // triggers SC isolation, increment redundant work-item
}
```

Figure 3.2: Introspective kernel

redundant work-items (*RWIs*), i.e., the number of extra work-items that a work-group requires to bypass the faulty SCs. If this number is more than zero, the just-in-time compiler compiles a healthy version of the naive kernel. The healthy kernel is launched with a different work-item count which simply can be the default work-item count for the naive kernel plus the reported redundant work-item count by the introspective kernel. In other words, for every work-item that is mapped to the degraded SC, a new redundant work-item should be generated to be mapped on another healthy SC. This is doable when the naive work-item count plus the required redundant work-item count is less than 256 (which is the limit for work-item count per work-group in Ellesmere GPUs). For cases that the new work-item count is greater than 256, the work-item count is decreased and the work-group count is increased instead in the healthy kernel. Further, the compiler is able to tune the number of work-items of a healthy kernel based on a specific degradation scenario described in Section 3.4.1.2.

The healthy version of kernel takes the naive work-item count as an extra input parameter as shown in Fig. 3.3. OpenCL kernels are usually written in a way that the work-item ID is used to index a memory location. To preserve functionality, the redundant work-items should exactly imitate those work-items that are not executed because they are mapped to the faulty SC. Therefore, every work-item checks its corresponding memory location filled with the test result or delay monitoring sensor information which forms its 'meta data'. If the corresponding SC is a faulty one, the work-item pushes its ID in a queue and performs no other operation. This queue is a light weight data-structure protected with atomic index and is shared within the

```
__kernel void HealthyKernel(... default_parameters ..., __const int work_item_count)
{
       unsigned int work_item_id = get_local_id(0);
       if (check_degradation_status()){
              push(work_item_id);
              go_to_end_of_kernel_and_do_nothing();
       }
       if (work_item_id >= work_item_count){ //redundant work-items
              virtual_work_item_id = pop();
              //naive kernel execution
              kernel(virtual_work_item_id, ...default_parameters...);
       }
       else //normal work-items
              kernel(work_item_id, ...default_parameters...);
}
```

Figure 3.3: Healthy kernel

work-group. The queue has a local memory of size 128 Bytes to store the local ID of degraded work-items. Since every work-group has a maximum number of 256 work-items, this local memory queue is sufficient for a 50% failure rate for SCs in a CU. Given that a CU can run up to 6 simultaneous work-groups, the healthy kernel consumes 6*128=768 Bytes of 32K shared memory of CU. This queue size does not impact performance of any kernels thanks to its limited memory footprint. Other work-items that are mapped to a non-faulty SC execute normally. The redundant work-items, which have the local ID greater than the naive work-item count, must be executed on behalf of the *resting* work-items. This is done by a virtual ID redirection through a conditional code for ID assignment. Every redundant work-item changes its local ID to the ID of one of the disabled work-items by popping it from the queue, and then executes the kernel with extracted virtual work-item ID. Using virtual ID redirection, the redundant work-item can read the required data from the memory hierarchy exactly the same way as the disabled naive work-item and there is no need to move any data. This forms a temporal fault-aware rescheduling and workload shifting that improves the lifetime of a GPU device while imposing low performance penalty discussed in Section 3.4.1.

Table 3.1: Parameters for the Naive Kernels

Kernel	abbreviation	#WIs per WG	#WGs
SobelFilter	SF	256	1024
BlackScholes	BSC	128	2048
Convolution	CNV	256	1024
FastWalshTransform	FWT	128	2048
FloydWarshal	FW	256	1024
MatrixMultiplication	MM	64	4096
QuasiRandomSequence	QR	128	2048

3.4 Experimental Setup and Results

We focus on the AMD accelerated parallel processing (APP) software ecosystem [10] suitable for stream applications written in OpenCL. The stream kernels are compiled into GPU device-specific binaries using the OpenCL compiler tool-chain which uses a standard off-the-shelf compiler front-end (g++), as well as the low-level virtual machine framework with extensions for OpenCL as the back-end. Table 3.1 lists the kernels, the work-item (WI) count per work-group (WG), the number of work-groups for the naive kernel. We have used Radeon RX 580 GPU for our experiments in this section.

3.4.1 Performance and Energy Overhead

The number of faulty SCs is process-voltage-temperature-workload dependent and changes from chip-to-chip and overtime. Therefore, we assess the performance overhead of our approach for two distinct degradation scenarios:

1. We measure the performance overhead when the number of degraded SCs is increased from 1 to 32 in a single CU. A CU with 32 degraded SCs shows a pessimistic aging scenario where 50% of its resources (the SCs) are degraded. This tests the performance overhead of our technique in the worst case.

2. We also measured the sensitivity of the performance overhead for different number of degraded CUs (from 1 to 10 degraded CUs).

Figure 3.4 measures the performance overhead of the SC healing method when the number of degraded SCs is increased from 1 to 32 for Ellesmere GPU, and the degraded SCs are all located in one CU. As can be seen, if we have one degraded SC in a CU our method incurs 0.14%–8.16% (with geometric mean 1.3%) extra performance overhead depending upon the type of kernel. As we can see, for some applications like *FWT* and *CONV*, this overhead is higher than others. This is because for those applications the number of work-items is a more important factor in the execution time of the application. In fact, finding the optimum number of work-items in a work-group is one important knob in GPU program optimization. Therefore, as we discuss later, we find the best number of work-items in order to reduce this performance overhead. As we increase the number of degraded SCs, this overhead increases. For the very pessimistic scenario, when we have 32 degraded SCs per CU, the performance overhead for different applications changes between 1.6%-13% with geometric mean of 7.1%. This is because the number of working work-items in work-groups are reduced and the number of workgroups that is mapped to CUs are increased. In fact, there are a few factors that affects the performance overhead of healthy kernel, which we discuss later in this section.

Figure 3.5 shows the energy overhead of using healthy kernels in comparison to the naive kernel. The energy overhead comes from the fact that the execution time of healthy kernels are more than the naive kernel. As can be seen, the energy overhead follows the behaviour of performance overhead shown in figure 3.4. For one degraded SC, this number is between 0.045%-7.83% for different applications. For 32 degraded SCs, the energy overhead changes between 1.61%-9.97% with geometric mean 6.37%.

If there is any degraded SC in any CU, the required number of redundant work-items is generated for all the work-groups. The number of work-items and work-groups are only controllable from `clEnqueueNDRangeKernel` API in an OpenCL application; and all workgroups

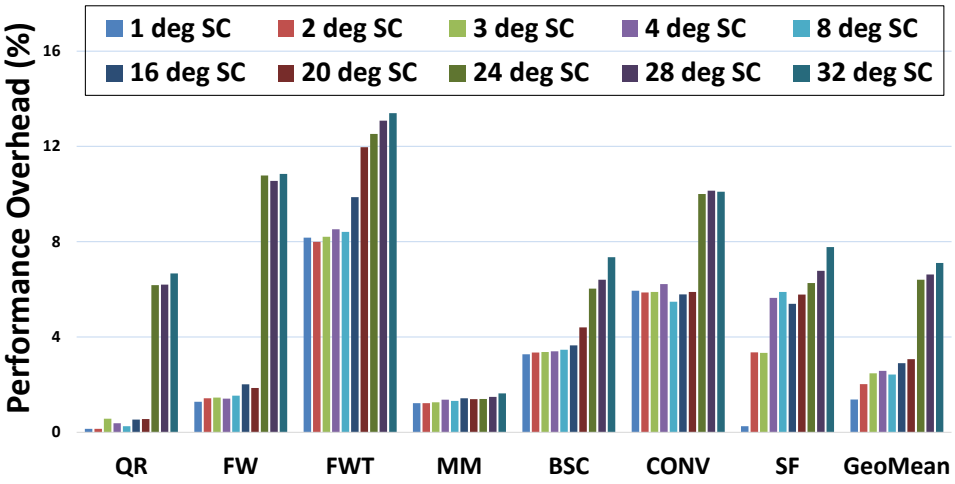


Figure 3.4: Performance overhead of using healthy kernel on Radeon RX 580 compared to the naive kernel when the number of degraded SCs (deg SC) are increased from 1 to 32 (all in a single CU) for different kernels and their average

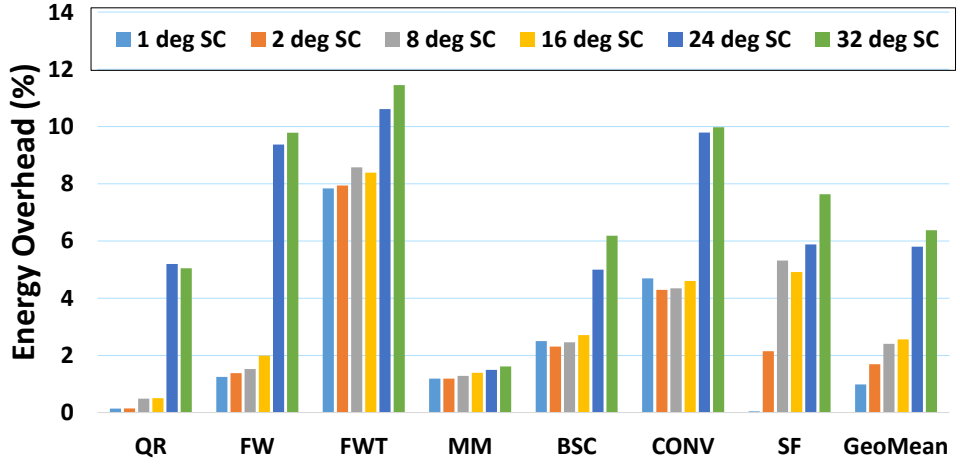


Figure 3.5: Energy overhead of using healthy kernel on Radeon RX 580 compared to the naive kernel when the number of degraded SCs (deg SC) are increased from 1 to 32 (all in a single CU) for different kernels and their average

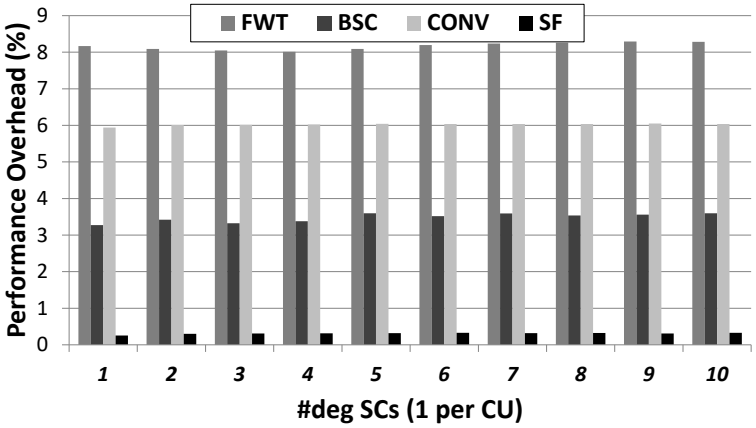
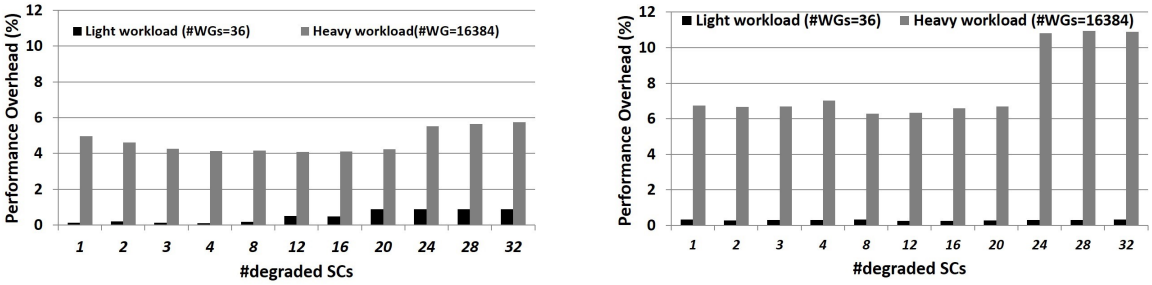


Figure 3.6: Performance overhead of using healthy kernel compared to the naive kernel when the number of degraded CUs is increased from 1 to 10 (one degraded SC per each CU) for four kernels

has the same number of workitems. Consequently, if we have one faulty SC in a CU or one degraded SC in more than one CUs the performance overhead would not vary. Figure 3.6 shows the performance overhead when the number of degraded CUs is increased from 1 to 10 – each CU has one degraded SC.

The difference between performance overhead of different kernels using our approach comes from a couple of factors. The first factor is the number of work-groups which is discussed in Section 3.4.1.1. The other factor is related to the intrinsic characteristics of each kernel. The memory access pattern, and location of barriers for synchronization would affect the performance of the healthy kernel. As an example, if the naive kernel only benefits from intra-wavefront locality, then changing the order of work-items within a work-group may hurt the performance of healthy kernel due to higher cache miss rate.

In the following, we have performed a sensitivity analysis on the execution time of the naive and healthy kernels considering different parameters: the number of work-items, the number of work-groups, the compilation optimization options, and finally the target GPU architecture.



(a) BlackScholes (b) SimpleConvolution

Figure 3.7: Effect of changing number of work-groups (input size) on Radeon RX 580 for (a) Blackscholes (b) SimpleConvolution as the number of degraded SCs is changed from 1 to 32 in a CU

3.4.1.1 Effect of Number of Work-Groups (Input Size)

Fig 3.7 illustrates the effect of changing the input size, i.e., the number of work-groups on the performance overhead of our method using. I) With a small input size (light workload) that utilizes all compute-units only once in the naive version (with 36 work-groups, device utilization = 100%), the performance penalty is very small, under 1%. II) With a large input size (heavy workload) containing a number of work-groups that is comparatively larger that the number of CUs (number of work-groups = 16384 for naive kernels), the performance penalty is larger, between 5%-10% for Convolution and 4%-6% for Blackscholes. The side effect of temporal scheduling in our reshceduling method is pronounced with larger number of work-groups, since more work-groups are mapped to the available SCs.

3.4.1.2 Effect of Number of Work-items for Performance Tuning

Figure 3.8 shows the execution time of naive SobelFilter kernel with different number of work-item for the same workload. As can be seen, number of work-items has a large impact on the performance of the program. For this kernel, we changed the work-item count from 116 to 252, for executing the same workload, and the execution time varies between 16.55 ms and 23.04

ms. This is one of the main reasons that the our method has performance overhead comparing to the naive kernels. The SC healing method is able to tune the number of work-items of

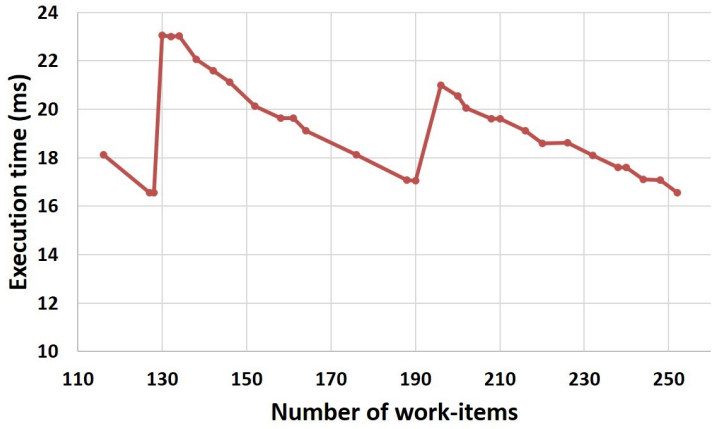


Figure 3.8: Effect of changing work-item count in the naive kernel for SobelFilter

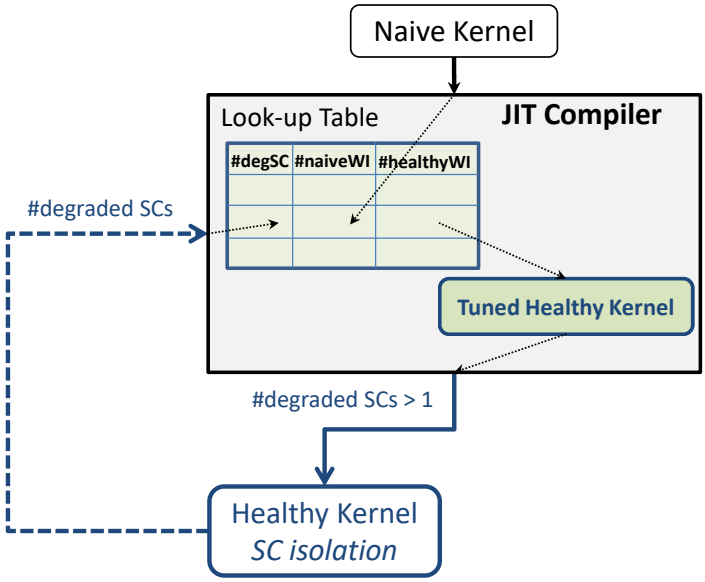


Figure 3.9: Performance tuning for healthy kernels.

a healthy kernel based on a specific degradation scenario to boost performance. The method determines an optimal number of work-items as a function of degraded SCs per CU for a healthy kernel such that its performance penalty is minimum. To implement this feature, a lookup

table (LUT) is designed for each kernel which takes the number of degraded SCs – calculated using the reported *RWIs* – and the number of naive work-items as the inputs and returns the best number of work-items suitable for the degradation scenario such that the performance overhead is minimized. This LUT is constructed through an offline pre-processing phase by measuring the performance using various possible work-item counts for the kernel. The off-line preprocessing is a one-off activity.After constructing LUT, the recompilation process is guided to further *reshape* the healthy kernel for improved performance as shown in Fig. 3.9. This is done as part of the aging-aware kernel adaptation flow in Fig. 3.1.

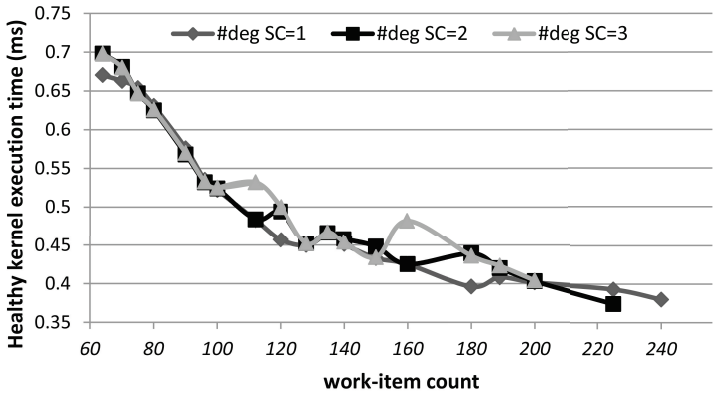


Figure 3.10: Effect of changing (#WI, #WG) on the execution time of healthy kernel for a synthetic kernel with a fixed input size (1209600 integers) on Radeon 580rx

Fig. 3.10 shows the effect of changing work-item count and work-group count on the execution time of a synthetic healthy kernel. This kernel gets a number of integer inputs and performs arithmetic calculations on each entry in the global memory. For a given fixed input size, the work-item count is changed to the all possible values between 64 and 256. The experiment is repeated for different work-item counts when there is 1, 2, and 3 degraded SCs in a CU. As shown, for each pair of degradation scenario and naive work-item count there is an optimal point in a close proximity that yields shorter execution time. This information is stored in the LUT in a discretized manner to infer the best work-item count.

We show the effectiveness of leveraging this performance tuning knob. Figure 3.11 shows the speedup of a tuned healthy synthetic kernel (used in Figure 3.10) compared to a normal

healthy kernel which is unaware of tuning. Theses experiments are repeated for 3 different degradation scenarios (1, 2, and 3 degraded SCs in a CU) and 18 different input sizes with heavy workload, and the speedup of using a tuned healthy kernel to the non-tuned healthy kernel is reported. The naive kernel has 256 work-items per work-group; therefore, the non-tuned healthy kernel decreases its active work-item count to 128 to have enough space in the wavefronts for the redundant work-items. As shown, for any input size, the performance tuning method has a speedup range of 10%–20%.

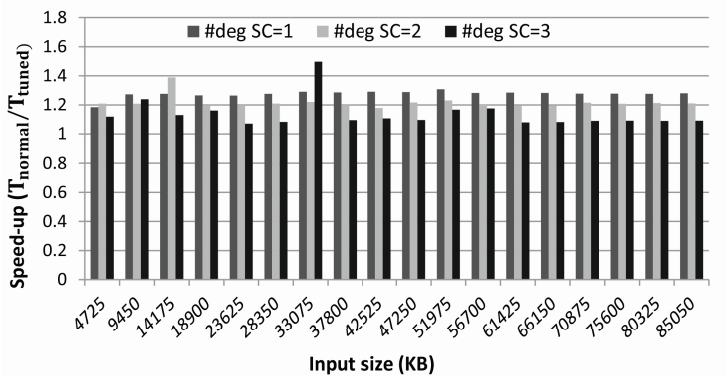


Figure 3.11: Performance benefit of using a tuned healthy kernel over a healthy kernel which is unaware of tuning (The benchmark is the same as Figure 3.10)

3.5 Mitigation of Effects Caused by Aging in Microelectronics

3.5.1 NBTI-Induced Performance Degradation

Aging has been a main factor in performance and reliability degradation of nanoscale devices. Aging decreases the lifetime of system. NBTI is an aging mechanism which manifests itself as an increase in the PMOS transistor threshold voltage (V_{th}) and causes delay-induced failures. When a PMOS transistor is negatively biased ($V_{gs} = -V_{dd}$), the dissociation of Si–H bonds along the silicon oxide interface causes the generation of interface traps, while removal

of the bias (V_{gs} = 0) causes a reduction in the number of interface traps due to annealing [13, 17, 20, 21]. The rate of generation of these traps is accelerated by temperature, and the time of applied stress. The threshold voltage (V_{th}) of the PMOS transistors increases as more traps form, reducing the drive current, which in turn raises the propagation delay of logic gates over time. Thus, the NBTI-induced performance degradation strongly depends on the amount of time during which a PMOS transistor is stressed, that is, when a logic '0' is applied to the gate. The increase in V_{th} is a logarithmic function of the corresponding stress time [68], which is distributed non-uniformly across a logic circuit, leading to 2–5× difference in the degradation rate of V_{th} [115] across a chip. When the stress condition is relaxed, aging can be recovered partially, and the V_{th} decreases toward the nominal value [14, 115].

NBTI is best captured by the Reaction-Diffusion (RD) model [86]. This model describes NBTI in two stress and recovery phases. NBTI occurs due to the generation of the interface traps at the Si-SiO_2 interface when the PMOS transistor is negatively biased (V_{gs} = -V_{dd}) (i.e., stress phase). In the stress condition, some holes in the channel interact with the Si-H bonds in the interface which causes disassociation of Si-H bonds. The resulting hydrogen atom diffuses away and leaves positive traps in the interface. As a result, the V_{th} of the transistor increases which in turn slows down the device. Equation 3.1 shows this increase in the V_{th} due to stress [115]:

$$\Delta V_{th-stress} = (K_v\sqrt{t_{stress}} + \sqrt[2n]{\Delta V_{th-t0}})^{2n} \tag{3.1}$$

where t_{stress} is the amount of time that PMOS transistor is under stress; K_v has dependence on electrical field, temperature (T), and V_{dd}; n is the time exponent parameter which is $1/6$ for H_2 diffusion; and ΔV_{th-t0} is the initial V_{th} variation of PMOS at time zero.

Removing stress from the PMOS transistor (V_{gs} = 0) can eliminate some of the traps by diffusing back dissociative H atoms, which partially recover the V_{th} shift. This is known as the

recovery phase:

$$\Delta V_{th-recov} = \Delta V_{th-stress}(1 - \frac{2\xi_1 t_e + \sqrt{\xi_2 C t_{recov}}}{(1+\delta)t_{ox} + \sqrt{Ct}}) \tag{3.2}$$

where t_{recov} is the time under recovery; t_{ox} is the oxide thickness; t_e is the effective oxide thickness; t is the total time; C has temperature dependence; ξ_1, ξ_2, and δ are constants [115].

[14] derived a long-term cycle-to-cycle model as follows. In this model, the stress and recovery cycles can be simulated for i cycles to find the V_{th} degradation. $\Delta V_{th-stress,i}$ and $\Delta V_{th-recov,i}$ are temporal changes in V_{th} at the end of i-*th* stress and recovery cycles, respectively:

$$\Delta V_{th-stress,i} = (K_v \sqrt{\alpha T_{clk}} + \sqrt[2n]{\Delta V_{th-recov,i}})^{2n} \tag{3.3}$$

$$\Delta V_{th-recov,i} = \Delta V_{th-stress,i}(1 - \frac{2\xi_1 t_e + \sqrt{\xi_2 C(1-\alpha)T_{clk}}}{(1+\delta)t_{ox} + \sqrt{CiT_{clk}}}) \tag{3.4}$$

where α is duty cycle or the ratio of time spent in the stress to one period of stress-recovery; T_{clk} is the period of one stress-recovery cycle; and $i = t/T_{clk}$. The NBTI rate depends on many factors including process-related parameters, temperature, voltage, and workload. Here we focus on the impact of workload or α in the above equations. The duty cycle (α) is controlled by the software to reduce the NBTI-induced effects.

A transistor with a larger V_{th} than expected has lower drive current, and higher delay during a transition. The switching delay of a transistor can be roughly expressed as the alpha-power law:

$$\tau \propto \frac{V_{dd} L}{\mu (V_{dd} - V_{th})^{\alpha'}} \tag{3.5}$$

where μ is the mobility of carriers; $\alpha' \approx 1.3$ is the velocity saturation index; and L is the channel length. Therefore, the delay variation $\Delta\tau/\tau$ can be derived as follows:

$$\Delta\tau/\tau = \frac{\Delta L}{L} + \frac{\Delta\mu}{\mu} + \frac{\alpha'}{V_{dd} - V_{th}} \Delta V_{th} \tag{3.6}$$

Considering only the effect of ΔV_{th} shift and neglecting other terms, the delay degradation $\Delta\tau$ is

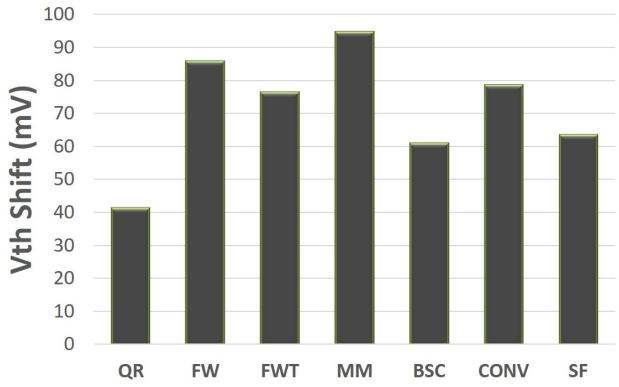


Figure 3.12: V_{th} shift for different kernels at the end of 360 hours on HD 5870 with one degraded SC.

shown in Equation 3.7:

$$\Delta\tau = \frac{\alpha' \Delta V_{th}}{V_{dd} - V_{th-t0}} \tau 0 \tag{3.7}$$

where V_{th-t0} is the original transistor threshold voltage (at time t_0), and τ_0 is its corresponding delay before degradation. We consider the largest ΔV_{th} to calculate the worst case delay degradation [18, 59, 85, 109] in a circuit to assess the potential benefits of proposed NBTI mitigation techniques. In our analysis, we set all the internal node states to '0' during the stress mode to determine the worst case circuit degradation that limits the lifetime of a chip.

3.5.2 Improvement in ΔV_{th}

We consider cycle-by-cycle architectural NBTI analysis [14] in the 65nm PTM technology with V_{gs} =1.2V, T=300K. The stress statistics of the kernels execution obtained from Multi2Sim simulator which is a cycle-accurate CPU-GPU simulation framework [111]. It is common to assume that all PMOS in a circuit degrade by the same amount [18, 59, 85, 109]. Figure 3.12 shows the V_{th} shift at the end of 360 hours due to the naive kernel execution for different kernels. As we can see in this figure, for these kernels, V_{th} can change from 41.2 mV to 94.5 mV after 360 hours of execution.

In order to reduce the V_{th}, as explained above, we can have periods of recovery, in which

we use our healthy kernel and put the degraded SCs in recovery mode. Figure 3.13 shows the behaviour of V_{th} shift based on the model explained above. It shows that it takes a few seconds to reduce V_{th} from 100 mV. Although this NBTI model seems optimistic, but it shows that if we remove stress from transistors, they can be healed. Therefore, we can recover aged SCs in the GPU using the method we have proposed in this chapter.

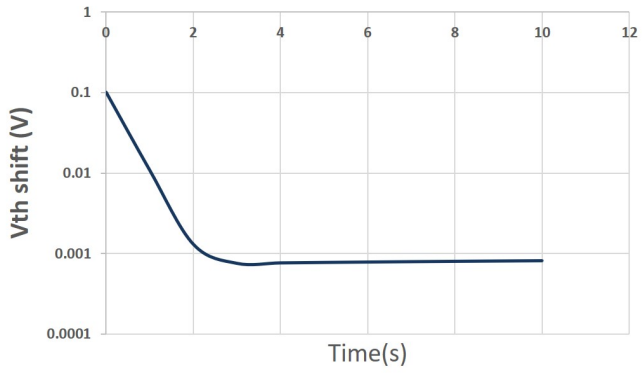


Figure 3.13: V_{th} shift behaviour during recovery phase based on the NBTI model in [14].

3.6 Related Work

For mitigating error effect on GPU, [117] proposed a hardware redundancy method. This method uses two vector lanes or two stream processors along with redundant data storage to perform the same computation. This approach induces high overhead in resource utilization and energy. In contrast to this approach, our method does not require redundant hardware resources. [74] [73] propose task migration by modifying GPU hardware scheduler and the instruction dispatcher to cluster the computational cores and distribute instructions to different cores based on the health state of system. This work requires intrusive modification in hardware, it is inflexible, and can not make error recovery possible for the available GPU devices. Two methods have been proposed to tolerate faults in GPU lanes [38]. They perform intra-cluster and inter-cluster thread shuffling that requires modifications in pipeline and warp scheduler. These methods incur inevitable area and performance overhead for intrusive hardware modification. A

coarse-grained method for mitigating aging-induced degradation for GPUs is proposed in [23] that performs power-gating at the granularity of compute units. They use an online algorithm to find optimal number of compute units which imposes *extra* time overhead to the execution time of each kernel. A linear programming scheme is employed to find a new instruction to replace the cores default NOP instruction for minimizing aging effects [42]. This approach also requires intrusive architectural supports and pipeline modification. Wearout-aware compiler-directed register assignment techniques have been proposed in [9] that attempt to distribute the stress-induced wearout throughout the register file. Even though [9] does not impose architectural overheads and modification, their compiler strategies are limited to healing the register file. In contrast to all these methods, our method is a low overhead software-level solution that can mitigate permanent error and aging effect on GPUs in the granularity of stream cores.

3.7 Chapter Summary

We propose innovations in the static compiled kernel code in conjunction with adaptive rescheduling and workload reallocation strategies to isolate the degraded components and mitigate hard error effect and lifetime uncertainty in GPUs. This method leverages a compiler-directed scheme to generate and launch healthy kernels that respond to the health state of a GPU. By isolating degraded units, we can make use of other available units and improve the cost and lifetime of the GPU. Online monitoring and software calibrations schemes, such as ours, enhance benefits of GPU accelerations as their reliability is an important issue.

3.8 Acknowledgments

Chapter 3 contains reprints of Atieh Lotfi, Abbas Rahimi, Luca Benini, and Rajesh Gupta, “Aging-Aware Compilation for GPGPUs”, *ACM Transactions on Architecture and Code Optimization (TACO)*, 2015. The dissertation author is the primary author of this paper.

Chapter 4

Case Study: Reducing Redundant Hardware Overheads

In the previous chapters, we have proposed effective *compiler-level* solutions for isolating faulty components in GPU and FPGA accelerators. These methods allows us to continue using the faulty accelerator through isolation and task migration. In the rest of this dissertation, we address the other challenge of programmable hardware accelerators and present methods to utilize their available hardware resources more efficiently. In this chapter, we perform our resource optimization study at the level of *hardware*. The goal in this chapter is to find out if there are any reliability-related redundancies in the design of hardware accelerators that can be removed or replaced with a more efficient design choice. Traditionally, memory structures are augmented with reliability-related redundancies in order to become resilient to errors, which results in area and energy overhead. We study the necessity of these redundancies and the resiliency of unprotected *tag* caches in GPU. We further study the usefulness of a very low overhead tag error mitigation method to replace parity bits, which can be applied to read-only and write-through caches in the GPUs. The new cache indexing mechanism distributes memory accesses more efficiently and mitigates some pathological address strides that cause error. This method also increases cache throughput. This architecture can eliminate the need for parity

protection in a portion of tag SRAM cache structures, while it is required to use redundancy for the rest of structures. This approach can be applied to any system that uses cache memory, including off-the-shelf GPUs.

4.1 Introduction

Performing resource optimization in hardware-level at design time is one of the most effective solutions to avoid resource waste. All applications that are run on the hardware would benefit from any optimization that is performed in this level of abstraction. There are traditional reliability-related redundancies that are augmented in the design of hardware to mitigate the effect of error. The most common reliability-related redundancy is found in memory structures. Due to dark silicon effect, the memory portion of chips are getting larger. Therefore, it becomes more effective to reduce unnecessary overheads from memory.

GPUs have multiple levels of cache memory structures to improve performance and reduce energy overhead for accessing data from the main memory. These cache structures are vulnerable to transient hardware errors, especially as their sizes increase and due to operating at low voltage levels. These transient errors can cause failures and corrupt application output. To protect against transient errors, cache memories typically use error detection/correction mechanisms, such as parity/*error correcting codes* (ECC) bits. When a cache line is augmented with parity/ECC bits, every cache access also requires error detection/correction encoding. It is useful to study the necessity of using these parity/ECC bits, and investigate if there is any other method with lower overhead that can be used instead while the reliability requirement is satisfied.

To have a reliable system, the overall FIT (*failures in time*)[1] rate of the system should be lower than a specific value. For example, for HPC purposes, FIT rate of 10 can be acceptable. The overall FIT rate of system is the addition of FIT rate of each component. The FIT rates of

[1] One FIT is defined as a failure rate of 1 per billion hours of operation

each component is proportional to its size. Therefore, the FIT rate is higher for larger cache structures. If a parity/ECC protection is used, the SDC (silent data corruption) FIT rate becomes nearly zero; meaning that the errors that happen will not affect the output. Each cache structure has a tag SRAM structure and a data portion. The tag SRAM has a smaller size, and it has a lower impact on the overall SDC FIT rate of the GPU. This can be a good option for removing parity protections and replacing it with an alternative method with lower overhead. We found this opportunity in the tag structure of read-only instruction caches or write-through data caches to eliminate the need for parity protection. On the other hand the data portion of cache has a higher size and a higher impact on the reliability of system. It is not easy to replace the parity protection with a lower overhead method which can satisfy the required reliability of system. Therefore, it is necessary for the data portion of the cache structures to have the reliability related redundancies.

A single-bit error in the tag memory of a set-associative cache is unsafe if the entries that map to the same set are separated by a Hamming distance of 1. If an error happens in a tag entry and this bit flip results in an incorrect match to the reference tag, the wrong data is read from the cache. This *false hit* event might cause silent data corruption. Therefore, if we change the distribution of tag entries to reduce the chance of mapping two tags with 1-bit Hamming distance to the same set, the probability of an SDC event is reduced. It has been shown that hash functions derived from *Galois Field* [106] primitive polynomials satisfy this requirement. Using this hash function, all address references that hash to the same location would be at least 3-bit Hamming distance away from each other. The Galois Field with base 2 (*GF2*) is very low cost and easily implemented using *eXclusive OR* (XOR) gates in hardware.

We propose an architecture that uses hash functions for set-index calculation to change the distribution of tags in the cache. Specifically, this chapter makes the following contributions:

1. We use Galois-based hash functions, for set-index calculation, in high performance and energy efficient GPU computing, to mitigate false hit events and improve performance.

This work is the first to investigate the effectiveness of hash-based cache indexing schemes for reliability purpose.

2. We provide a sensitivity analysis of cache reliability to key architectural parameters: cache tag width, and associativity. We quantify the reliability and performance of the hash-based cache indexing schemes for the read-only instruction and write-through data caches of the GPU architecture. We conclude that the error probability is a function of workload address trace characteristics and tag SRAM structure.

3. We demonstrate, for the high-performance computing (HPC) benchmarks, the probability of false-hit events in the tag structures is very low and has a negligible impact on the overall SDC FIT rate of the GPU. Furthermore, we show that Galois-based hash functions for tag structures in write-through data caches further reduce the false hit probability by a factor of 10.

4.2 Target GPU Architecture

This section provides a brief background on GPU architecture and its resilience support. Figure 4.1 shows a simplified, representative GPU architecture for NVIDIA Volta generation [8]. The GPU architecture comprises multiple GPU Processing Clusters (6 GPCs) and multiple streaming multiprocessors (14 SMs per GPC) within each GPU. Large SRAM structures like the Register File (RF) and the L1 Data Cache within each SM and the L2 Cache Data are ECC protected. There are a significant number of tag SRAM structures within each SM (L1 Instruction Cache, L1 Data Cache), within each GPC (uTLBs) and within each GPU (TLBs). Some of these tag SRAM structures (L1 Data Cache, for example) are already protected. Since the point of coherence in the GPU is the L2 Cache, it turns out that the L1 Data cache is write-through. The point of articulation of the GPU tag SRAM structures is to highlight the area saving opportunity by evaluating the effectiveness of low overhead tag error mitigation method. For example, in the

case of the L1 Data Cache tag SRAMs, there is an opportunity to save a total of 168K SRAM bits required for parity storage in all SM instances.

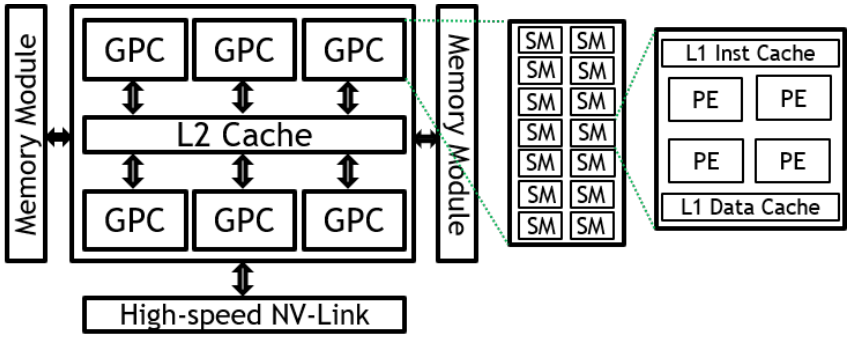


Figure 4.1: Volta GPU block diagram

4.3 Low overhead tag checking

For a tag SRAM array without any parity protection, errors in tag entries could cause *false hits* or *false misses* [102]. We describe these events as follows:

Assume that an error happens in an unprotected tag entry, Tag[i][j] (for set-index i and way j) with value TA1, and changes its value to TA2.

- Assume that the tag SRAM receives a new address reference with set-index i, and value TA1. In the absence of error, this would be a hit. However, because of the transient error, the result is not a hit. This event is called *false miss*.

- Assume that the tag SRAM receives a new address reference with set-index i, and value TA2 (one-bit hamming distance away from TA1). In this scenario, the Tag SRAM would match the incoming reference with its erroneous entry having value TA2. This situation is called *false hit*. False hit can be divided into two conditions:

  – Assume that the *only* entry having value TA2 was the entry that had value TA1 prior to the transient failure. This situation is called *false hit under true miss*.

– Assume that there is another error free entry in set-index i with value TA2. In this situation, a new reference with value TA2 would match two entries in the tag cache. This situation is called *false hit under true hit*.

The tag error event space is shown in Figure 4.2.

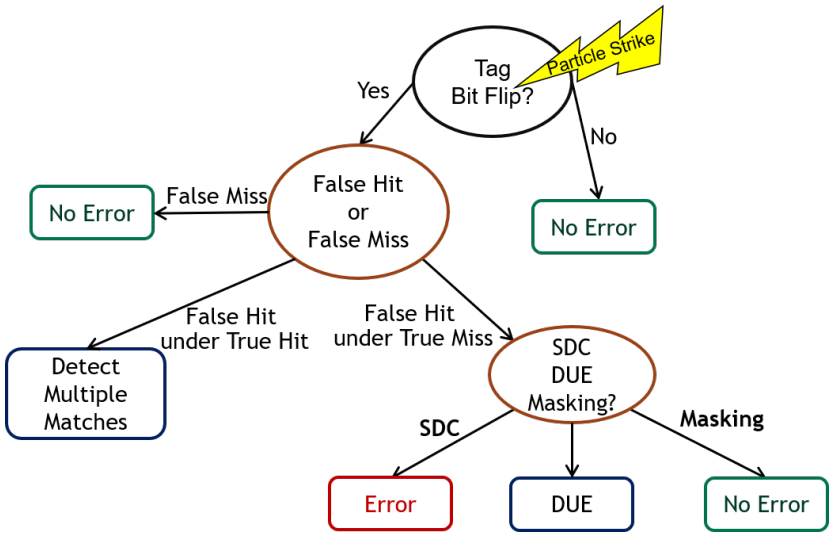


Figure 4.2: Tag error event space

For tag SRAM structures that cache read-only or write-through data, a false miss is not a reliability issue as a back-up copy exists and can be serviced through a refill. There might be only a negligible impact on performance due to false misses as these events are rare. However, a false hit under true miss can cause *silent data corruption* (SDC). If robust circuit design techniques are not used to detect multiple matches for the same row, false hit under true hit condition could also cause SDC. It should be noted that parity augmented tag SRAM arrays for single or odd bit errors effectively suppress the false hit or miss events.

We explored different design alternatives and made an observation that there exists a class of hash functions that map unit-distance address strides across different sets. This class of hash functions that are derived from *Galois Field* primitive polynomials satisfies the properties listed below. We use the example implementation in Figure 4.3 to describe these properties. (Figure 4.3 illustrates an unprotected Tag SRAM structure of a 4-way set-associative cache with

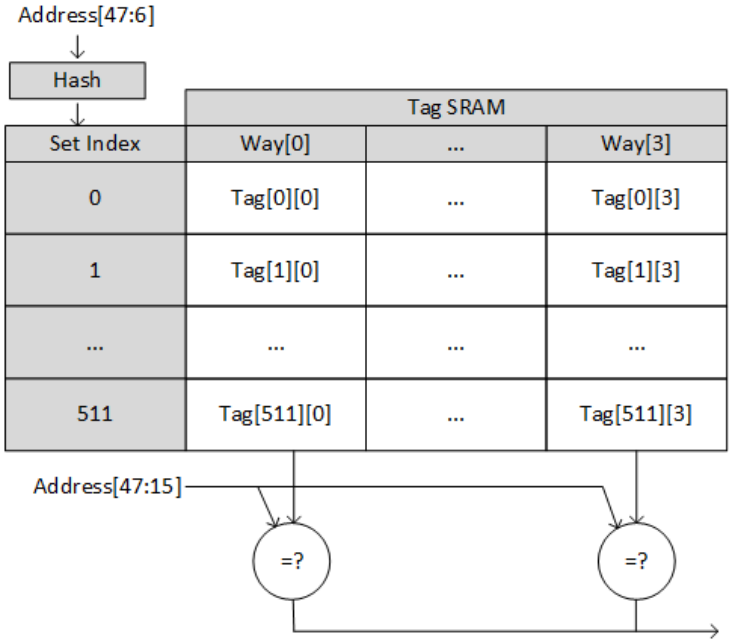


Figure 4.3: No explicit tag error checking

64 byte (B) cache line size and 128KB capacity. Assuming a 48-bit address width, *Address*[5:0] determines the byte within a 64B cache line and *Address*[14:6] defines the set-index in the 4-way tag SRAM array.)

- All address references that change in bit locations *Address*[14:6] (in general, any n contiguous *Address* bits in an n-bit hash function) will map to distinct set-index values. This generalizes the uniform set-index distribution property of hash free set-index calculation.

- All address references *Address*[47:6] that hash to the same set-index value, will be at least Hamming distance = 3 away from each other.

  For example, if *AddressA*[47:6] and *AddressB*[47:6] hash to the same set-index value i then $Hamming\_distance(AddressA, AddressB) > 2$.

- It follows from the previous property that address references with strides of power-of-2 (Hamming distance = 1) or strides of sum of powers-of-2 (Hamming distance = 2) will have different set-index values. This eliminates the false hit scenario that can happen in a hash-free implementation that uses *Address*[14:6] for the set-index.

This hash-based approach eliminates false hits arising from address strides with powers-of-2 or sum of two powers-of-2. In the following sections, we estimate false hit probability for address traces derived from HPC applications. An important side benefit of using a hash-based set-index is the marked improvement in uniformly distributing address references across the sets and the improvement in the cache utilization. We also study the effect on cache hit rate from using these false hit reducing hash-based tag lookup methods.

4.4 Simulation Methodology and Evaluation Metrics

We use a configurable cache simulator to conduct our resiliency studies. This simulator gets an address trace and cache configuration as input and evaluates the cache functionality and resiliency metrics. Traces contain the sequence of addresses submitted to the cache under study during the execution of a program on GPU. We extract input address traces for instruction and data caches for Volta GPU [8]. The cache simulator can calculate the set-index value either using the traditional way (*No-hash* method), or using the hash function (*Hash-based* method).

Other than the classic cache metrics like hit and miss rate, this simulator reports the false hit and false miss rates described in Section 4.3. False hit and false miss might occur when a bit error happens in a cache line. However, instead of injecting faults in cache lines and performing simulations many times for each trace, our simulator measures these metrics *without any explicit fault injection* in only a single run of the input trace. First, we explain how the false hit rate is calculated. A false hit can occur either in the event of a cache miss or a cache hit. We define the notions of $SetEntryFalseHits\_Miss$ and $SetEntryFalseHits\_Hit$ for the number of false hits under a cache miss and a cache hit in each set entry respectively. If a cache access results in miss, we calculate the Hamming distance of the new address reference tag and all the existing valid tags in the corresponding set. If the Hamming distance is 1, this access *might* result in a false hit. Therefore, every time the result of comparison has Hamming distance 1, we increment $SetEntryFalseHits\_Miss$ for that set entry. Another scenario that results in false hit under miss

happens when the address reference tag matches the tag in the corresponding set but the cache line is invalid. In fact, if a bit error occurs in the valid bit position for this entry, a false hit happens. Likewise, if a cache access results in a hit and at the same time the address reference tag is only one-bit Hamming distance away from any other valid tag entries in the same set, we increment the *SetEntryFalseHits_Hit* counter. The pesudo-code for calculating these metrics is shown in Algorithm 3.

Algorithm 1 Computing SetEntryFalseHits

```
setIndex = calculate_set_index(Address)
refTag = calculate_tag(Address)
Look up cache for refTag in the setIndex entry of tag cache (TAG[setIndex])
if Access is Hit then
    for i← 1 to Associativity do
        if (HammingDistance(refTag, TAG[setIndex][i])==1) then
            SetEntryFalseHits_Hit[setIndex]++
        end if
    end for
end if
if Access is Miss then
    for i← 1 to Associativity do
        if HammingDistance(refTag, TAG[setIndex][i])==1 then
            if valid[setIndex][i]==1 then
                SetEntryFalseHits_Miss[setIndex]++
            end if
        end if
        if (refTag == TAG[setIndex][i] and valid[setIndex][i]==0) then
            SetEntryFalseHits_Miss[setIndex]++
        end if
    end for
end if
```

This way, we increment the false hit counters every time we get Hamming distance 1 on any bit position in the tag entry, no matter if it is the faulty bit or not. To get the false hit rate considering bit flip in one bit position in each entry, we add a scaling phase at the end of trace execution. After the whole trace is executed, we divide each counter by the number of bits in the set entry *i*. This is summarized in the following equations. In these equations, *i* changes from 1 to the number of sets to calculate *EstimatedFalseHits* for each set.

$$\text{EstimatedFalseHits\_Miss}_i = \frac{\text{SetEntryFalseHits\_Miss}_i}{\text{Associativity} \times \text{TagBitWidth}} \tag{4.1}$$

$$\text{EstimatedFalseHits\_Hit}_i = \frac{\text{SetEntryFalseHits\_Hit}_i}{\text{Associativity} \times \text{TagBitWidth}} \tag{4.2}$$

To estimate the overall false hit rate in the cache, we simply add the *estimated false hits under miss* (*estimated false hits under hit*) calculated above for all sets and divide it by the number of cache accesses to calculate the *false hit rate under miss* (*false hit rate under hit*).

$$\text{FalseHitRate\_Miss} = \frac{\sum_{i=1}^{\#sets} \text{EstimatedFalseHits\_Miss}_i}{\text{Accesses}} \tag{4.3}$$

$$\text{FalseHitRate\_Hit} = \frac{\sum_{i=1}^{\#sets} \text{EstimatedFalseHits\_Hit}_i}{\text{Accesses}} \tag{4.4}$$

It should be noted that false hits under true hits can easily be handled by multiple tag match detection mechanism and is not considered as undetected corrupted output for an unprotected cache.

To evaluate our method, first we generate traces for different instruction and data cache modules in our target GPU. For this purpose, we use the GPU implementation of a set of HPC benchmark applications. A description of each benchmark can be found in Table 4.1.

4.5 Experimental Results

In this section, we show the experimental results for evaluating Hash-based set-index selection in the instruction cache and L1 data cache in the Volta GPU. To evaluate the resiliency of our method, we report the false hit rates. We also report the effect of using hash function on the cache hit rate.

4.5.1 Instruction Cache Tag SRAM Structure Evaluation

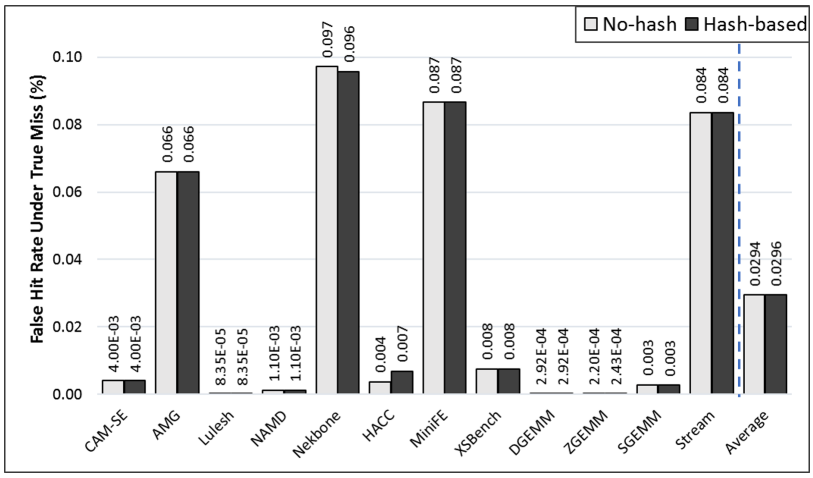
We performed the experiments for two different cache sizes. First, we performed the experiments for a 3-way set-associative instruction cache with 128B cache line size and 12KB capacity. Figure 4.4 shows the false hit under miss and false miss under hit for the No-hash

Table 4.1: Benchmark Description

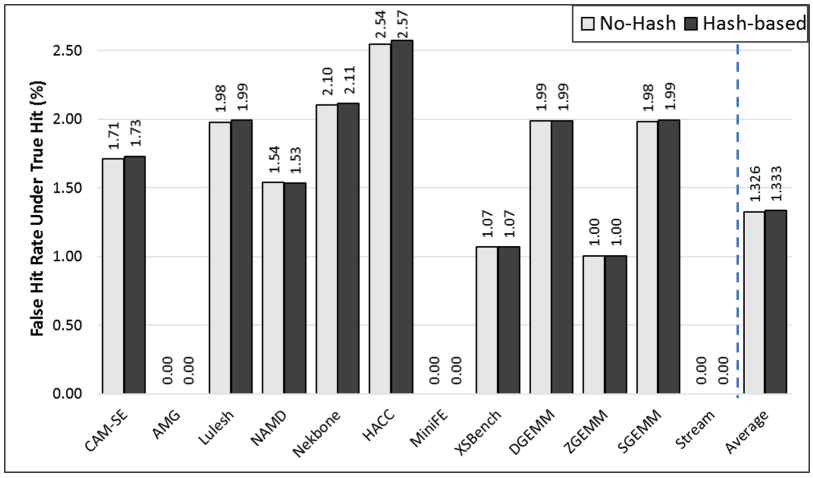
Benchmark	Description
CAM-SE	Atmospheric climate modeling
AMG	Algebraic multi-grid linear system solver for unstructured mesh physics packages
Lulesh	Shock hydrodynamics calculation for unstructured meshes
NAMD	Molecular dynamics
Nekbone	Solves Poisson equation by spectral element method
HACC	Hardware accelerated cosmology code, simulates the formation of structure in collisionless fluids under the influence of gravity
MiniFE	Mimics the finite element generation and assembly
XSBench	Monte Carlo neutron transport
DGEMM	Double-precision matrix-matrix multiplication
ZGEMM	Complex number matrix-matrix multiplication
SGEMM	Single-precision general matrix multiply
Stream	Simple computational kernel (add and multiply)

and Hash-based schemes. Both schemes have false hit rate under true miss less than 0.1% and result in similar false hit rates. For these applications, false hit under true miss is 0.0294% and 0.0296% for No-hash and Hash-based schemes on average (with standard deviation of 0.04% for both). Also as seen in Figure 4.4(b), on average, we get 1.32% and 1.33% false hit under true hit for No-hash and Hash-based schemes respectively (with standard deviation of 0.90% and 0.91%). We also report the effect of using each scheme on hit rate, which can be seen in column 2 and 3 of Table 4.2. The hit rate for these schemes is very similar: 98.1% for No-Hash and 98.09% for Hash-based schemes on average.

In the second experiment, we consider a 3-way set-associative cache with 128B cache line size and 24KB capacity, to measure the effect of cache size on false hit rates. Figure 4.5 shows the false hit under true miss and false miss under true hit for the No-hash and Hash-based schemes. Doubling the cache size improves the hit rate and consequently affects the false hit rate. For these applications, false hit under true miss is 0.022% for both No-hash and Hash-based



(a) False hit under true miss

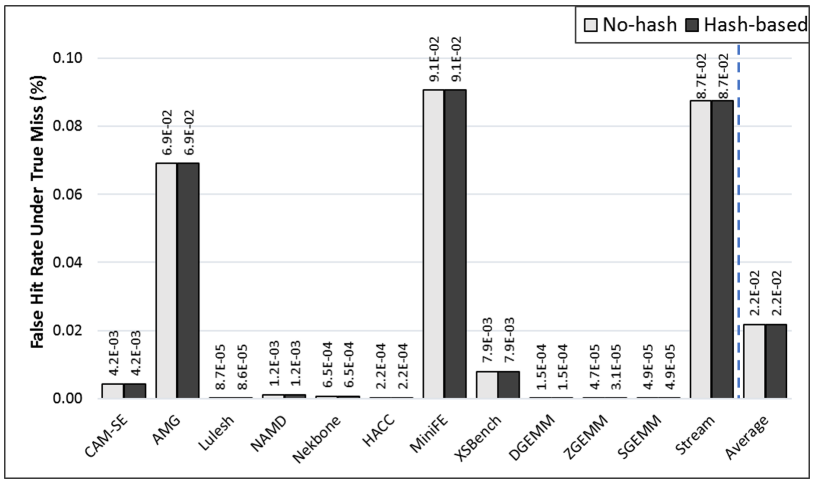


(b) False hit under true hit

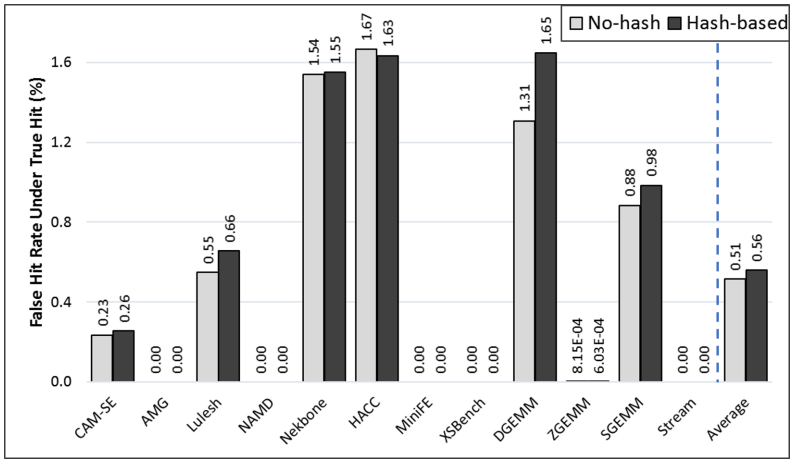
Figure 4.4: False hit for a 3-way12KB set-associative instruction cache: Hash-based vs No-Hash Schemes

schemes on average (with standard deviation of 0.037% for both). Also as seen in Figure 4.5(b), on average, we get 0.51% and 0.56% false hit under true hit for No-hash and Hash-based schemes respectively (with standard deviation of 0.66% and 0.7%). Compared to the smaller size cache, increasing the cache size reduces the false hit rate. For this cache configuration, No-hash and Hash-based schemes have very close hit rates with average 98.43%.

As can be seen, for both cache sizes, No-hash and Hash-based schemes are comparatively equivalent, though No-hash yields marginally better results.



(a) False hit under true miss



(b) False hit under true hit

Figure 4.5: False hit for a 3-Way 24KB set-associative instruction cache: Hash-based vs No-Hash Schemes

4.5.2 Data Cache Tag SRAM Structure Evaluation

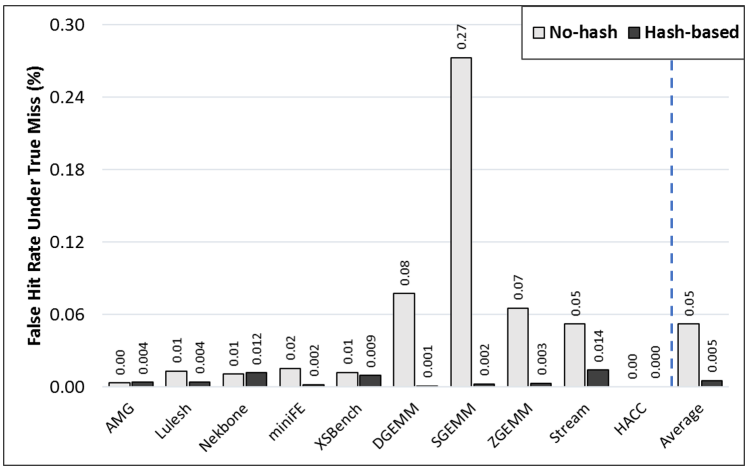
We did the experiments for a 4-way set-associative data cache with 128B cache line size and 128KB capacity. Figure 4.6 shows the false hit under true miss and false miss under true hit for the No-hash and Hash-based schemes. As can be seen, for these applications, false hit under true miss is 0.05% and 0.005% for No-hash and Hash-based schemes on average (with standard deviation of 0.082% and 0.004% respectively). Also, on average, we get 0.46% and 0.07% false hit under true hit for No-hash and Hash-based schemes respectively (with standard deviation of

Table 4.2: Hit rate comparison for No-hash and Hash-based schemes

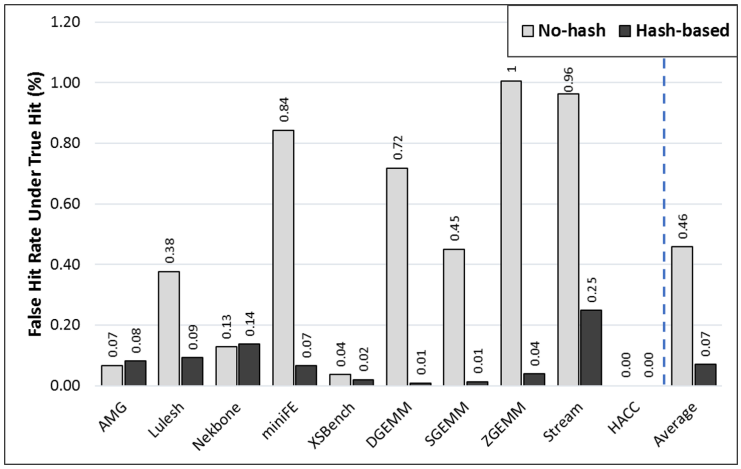
Benchmark	InstCache_Hit rate (%)		DataCache_Hit rate (%)	
	No hash	Hash-based	No hash	Hash-based
CAM-SE	99.72	99.72	93.75	93.75
AMG	95.44	95.44	95.52	95.52
Lulesh	99.99	99.99	96.14	96.21
Nekbone	96.63	96.68	94.05	94.12
HACC	99.86	99.73	99.9	99.9
MiniFE	94.02	94.02	96.89	97.45
XSBench	99.48	99.48	75.52	75.53
DGEMM	99.97	99.97	91.68	93.79
ZGEMM	99.97	99.97	92.95	93.59
SGEMM	99.80	99.80	70.4	87.85
Stream	94.23	94.23	93.75	93.75
Average	98.1	98.09	90.97	92.87

0.39% and 0.076%). For L1 data cache, the Hash-based scheme reduces the false hit rate by 10× on average compared to the No-hash scheme. The Hash-based scheme even improves the hit rate by 2% (No-hash scheme has 90.97% hit rate on average, while the Hash-based scheme improves the hit rate to 92.87% on average). The hit rate result is shown in Table 4.2, columns 4 and 5.

We also repeated the simulation for a smaller cache size (12KB cache, 3-way set-associative, and 128B cache line). We got similar results for this cache configuration. Results indicate that the false hit under true miss is 0.13% and 0.017% for the No-Hash and Hash-based schemes respectively. Also the false hit under true hit, we get 1.16% and 0.54% for No-hash and Hash-based schemes respectively. This also comes with 3% improvement for hit rate (90.1% and 93.1% for the No-hash and Hash-based schemes). In fact, regardless of cache configuration, the Hash-based scheme can improve both the reliability and performance of tag SRAM for write-through data cache.



(a) False hit under true miss



(b) False hit under true hit

Figure 4.6: False hit for a 4-way 128KB data cache: Hash-based vs No-Hash Schemes

4.5.3 Hash Function Area and Timing Analysis

We have added the hash function module to the L1 data cache tag in Volta GPU and synthesized it with Synopsys Design Compiler synthesis tool. Synthesis results show that adding the hash function does not make the critical path longer, and the critical path slack remains at 0. It also made negligible difference in the design area of L1 data cache tag module, actually a 0.4% reduction compared to the original scheme due to synthesis variation it caused. Unless the tag RAM addressing is already the critical path (not the case above), the timing and area effects in other use cases is expected to be similar.

4.5.4 Address Stride Distribution Analysis

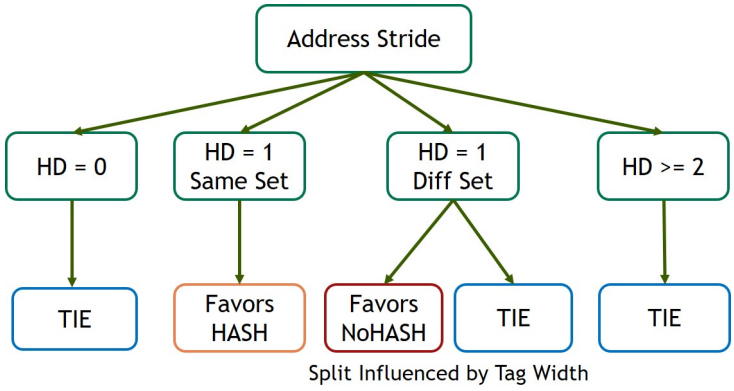
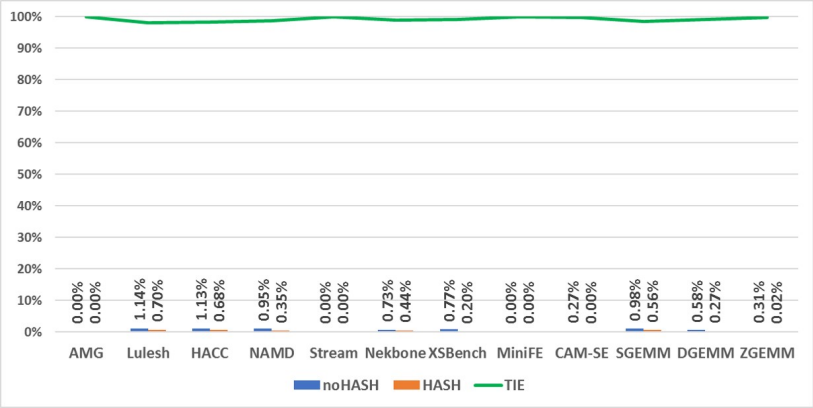


Figure 4.7: Address stride distribution analysis, based on Hamming distance (HD) between tag portions of consecutive addresses

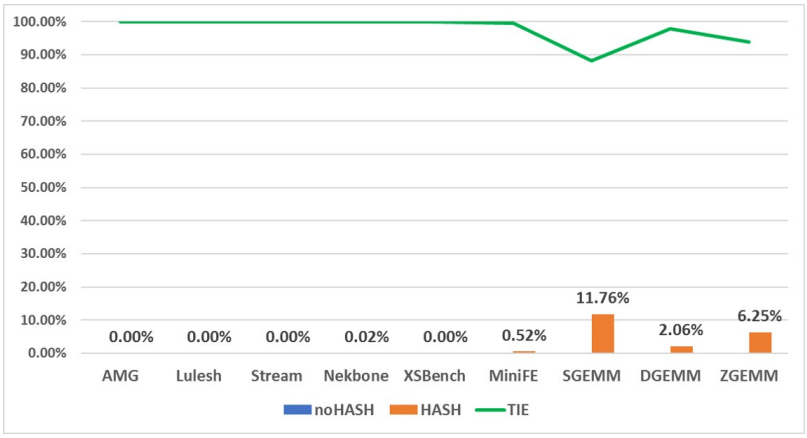
As we have learned from simulation results, the address trace is an important factor for determining if using the hash function for set-index calculation can decrease false hit rate or not. In order to have an indicator of why address strides matter, we perform a first order analysis on address trace distribution. For this analysis, we compute the Hamming distance between tag parts of each two consecutive addresses. Depending on the Hamming distance value, one of the schemes (No-hash or Hash-based) could perform better (Figure 4.7). For example, if Hamming distance between tags is 1, two cases might happen:

- If the set-indexes for these two consecutive address references are the same; the No-hash scheme would map both addresses to the same set. Therefore, if a bit flip happens in the tag entry that is in cache, a false hit occurs. However, the Hash-based maps the new address reference to another set, which decreases the chance of false hit. Therefore, this scenario is in favor of Hash-based scheme.

- If the set-indexes for these two consecutive address references are different; the No-Hash scheme maps these two addresses to two different sets, so a bit flip for the tag entry in the cache does not result in false hit. The same reasoning is true for the Hash-based scheme. So for the large fraction of time, these two schemes behave the same way, which we call it



(a) Instruction cache trace



(b) Data cache trace

Figure 4.8: Outcome distribution (Favor No-hash, Favor Hash, Tie)

a Tie. However, for the Hash-based scheme, the probability that these two sets map to the same locations in the cache tag is $\frac{1}{TagWidth}$, where $TagWidth$ is the number of bits in the tag part of the address. This is because for a given two address references with different set-indexes, in a primitive polynomial-based hash, there can be only one specific tag bit out of the $TagWidth$ bits that can be different in the tag portion of the reference.

With the same reasoning if the Hamming distance between two consecutive tags is 0, two schemes perform the same way, and the situation is a Tie. Also, for the other cases where the Hamming distance between two consecutive tags are more than 2, result is a Tie.

We performed this address stride distribution analysis and the result is shown in Figure 4.8.

As can be seen, the outcome distribution for the instruction cache is almost a Tie, with a slight favor to No-Hash scheme. This is what we have seen also in the results from simulation. For a data cache, either there is a Tie, or the distribution is closer to a Hash-based method as can be seen for benchmarks like MiniFE, SGEMM, DGEMM, and ZGEMM.

4.6 Analytical and Monte-Carlo Simulation for Estimation of False Hit Rates

To get an early understanding of the impact of various tag SRAM attributes (tag width, set associativity, set-index bits, and the cache capacity), we identified that tag width and associativity (i.e., the number of ways) have the most impact on the false hit probability. In fact, it can be shown that for a random address trace, the average probability of false hit is bounded above by: $\frac{\text{Associativity}}{2^{\text{TagWidth}}}$, where *Associativity* is the number of ways in the set-associative cache and *TagWidth* is the number of tag reference bits in the address bit field. Bounding the variance on the false hit probability is a much harder problem as it has a complex dependence on the nature of address references and the initial state of the tag address entries in the tag SRAM. We conducted millions of Monte-Carlo experiments to estimate the average false hit probability (under true miss events) and the standard deviation using random address traces. Tables 4.3 and 4.4 show these results and validate the analytical upper bound on the average probability of false hits. Results also show that in general false hit probability goes down exponentially with increasing tag width and somewhat increases with increasing associativity. As expected, the standard deviation is of the same order of magnitude as the expected false hit probability, suggesting much higher variance compared to a normal distribution.

While the false hit probability for random traces, in practical tag SRAM implementations, is about two orders of magnitude lower (1e-04 vs. 1e-02) than the probability of false hit estimate through actual workload trace simulation, the predicted sensitivity trend on tag width holds. For

Table 4.3: False Hit Probability Sensitivity to Tag Width (Associativity=3)

Tag Width (Bits)	Average False Hit Probability%	False Hit Std Dev%	Upper Bound $\frac{Associativity}{2^{TagWidth}} \times 100$
4	5.7800	5.6e-01	18.75
8	0.3900	9.5e-03	1.17
10	0.1000	9.0e-03	0.29
12	0.0230	2.7e-03	0.07
16	0.0016	4.1e-04	0.005
18	0.0003	2.3e-04	0.001
20	1.3e-04	1.3e-04	2.8e-04

Table 4.4: False Hit Probability Sensitivity to Associativity (Tag Width=20)

Associativity	Average False Hit Probability%	False Hit Std Dev%	Upper Bound $\frac{Associativity}{2^{TagWidth}} \times 100$
1	6.0e-05	6.14e-05	9.5e-05
2	8.5e-05	2.40e-05	1.9e-04
3	7.7e-05	4.02e-05	2.8e-04
4	8.6e-05	3.88e-05	3.8e-04
6	9.8e-05	2.85e-05	5.7e-04
8	1.0e-04	2.34e-05	7.6e-04
16	1.5e-04	1.53e-04	1.5e-03

example, the instruction cache references had 32-bit virtual-address width (with tag width = 20) and the maximum observed false hit probability, under true miss, was 0.1% whereas for L1 data cache with 48-bit virtual-address width (tag width = 33) that maximum observed false hit probability, under true miss, was 0.05%.

4.7 Related Work

4.7.1 Low-cost Soft Error Protection for Cache Structures

A common solution to address soft errors in cache memories is to apply error detection/-correction code such as parity/ECC bits uniformly across all cache lines [50]. Previous work on reducing cache protection overhead either breaks the assumption of uniform protection of all cache lines, or utilizes different mechanisms to protect clean and dirty cache lines. Kim et. al [65] suggest protecting only those cache lines that are most frequently accessed in every cache set to trade between area and level of data integrity. [64] [66] proposed a method to reduce area for protecting L2/L3 caches. Instead of using ECC for all entries, it selectively applies ECC just to dirty cache lines; other clean cache lines are protected by using parity check codes instead. However, our goal is to use hash functions to change the distribution of tag entries in the cache to reduce the probability of undetected errors without using parity protection.

4.7.2 Exploiting Hash Functions for Addressing Memory Structures

Hash functions are widely used in computer architecture to minimize conflict misses in caches [46] or to improve accessing to interleaved multibank memories [118]. A hash function maps an address to a set-index. The easiest hash function to implement for addressing cache memories is the modulo function, which is traditionally used and selects some of the least significant bits of the reference address. However, this way of set-index bit selection creates many conflicts for a number of frequently occurring access patterns, such as large power-of-2 strides. To avoid cache conflicts, alternative hash functions were studied for set-index bit generation to reduce conflict misses by achieving a more uniform cache access distribution across the sets in the cache [15] [110] [61].

Several types of hash functions have been investigated. Depending on the situation, one type of hash function can be more appropriate than another due to properties of access patterns.

The proposed techniques can be classified into static [46] [61] [113] and adaptive [99] indexing schemes.

Among the static methods, XOR-based mapping policies (e.g polynomial [97] and bitwise XOR [46]) are used to obtain a pseudo-randomly placement of blocks. In this way, a better distribution of blocks among cache sets can be obtained which reduces the number of conflict misses. Rau proposed a scheme [97] which describes a method for constructing XOR mapping schemes based on polynomial arithmetic. Polynomial indexing can be explained by considering address A = (a_{n-1}, ..., a_1, a_0) as a polynomial P(x) = $a_{n-1}x^{n-1}$,..., a_1x^1, a_0, where the coefficients are in the Galois Field GF(2) (can take on values 0 or 1). This XOR-based polynomial modulus function has very low complexity (requires only XOR operations) and is suitable for computing a cache index. If the generator polynomial is primitive and the code length is less than the cycle length of the polynomial, then the resulting code's Hamming distance is at least 3. Other works propose the use of more complicated hash functions, like prime-number based [61], which can be used for shared caches to minimize conflict misses at the cost of increased latency and hardware complexity. However, the complexity of these methods makes these techniques unsuitable for first level caches, where latency is critical. For GPUs, researchers have studied the effects of multiple static indexing techniques such as arbitrary modulus indexing [35] and polynomial [60] [63] on performance and energy consumption. In the context of adaptive cache indexing schemes, (ASCIB) [99] monitors the memory access pattern of workloads in CPU at runtime, determines the best indexing bits that are expected to minimize conflict misses for the observed memory access pattern, and periodically reconfigures them accordingly.

As opposed to this existing work, we study the effect of using hash functions on reliability. To the best of our knowledge, we are the first to evaluate the use of Galois-based hash functions, for set-index calculation for cache tags in GPUs. Our goal is to mitigate some pathological address strides that cause failures in the event of soft errors.

4.8 Chapter Summary

GPU architectures have stringent reliability requirements. Yet as a result of technology scaling, they are particularly susceptible to radiation-induced errors. Memory structures including caches and register files are responsible for the majority of transient errors on GPUs. To make these cache structures resilient to errors, entries in the caches are protected by parity/ECC bits, resulting in area and energy overhead. In this chapter, we studied the resiliency of unprotected tag caches in GPU and propose a very low overhead tag checking study that can be applied to read-only and write-through caches in the GPUs. This approach uses hash function for set-index calculation instead of using information redundancy. Our results show the sensitivity of the effectiveness of hash-based methods to the nature of address traces. For instruction address traces, the access patterns (as shown by the address stride distribution analysis) show a tie between Hash-based and No-hash based set-index methods. The Hash-based method mainly helps data cache tag structures, where Hash-based lookup is $10\times$ better than No-Hash lookup in terms of false hit probability. The hash function has almost negligible area and performance overhead. In addition, both simulation and analytical results indicate that the impact of low overhead tag error mitigation with or without set-index hashing on the overall GPU SDC FIT rate is less than 1%. While the simulation studies have been done only for GPU tag SRAM structures, there is nothing in the tag SRAM structures or the address reference patterns that are peculiar to the GPUs; therefore, the results of this study should hold good for CPU tag structures as well. Despite the effectiveness of this method for data cache tag structures, this study shows that the parity/ECC protection is necessary for the rest of cache structures. This is because they have a much higher SDC FIT rate, and as we showed the hash-based method only reduces false hit probability by a factor of 10, which is not enough for those structures. There might be other methods that can be used but perhaps they have higher overhead than using information redundancy itself. Moreover, the area saving from removing these parity bits can be used to make other more vulnerable parts of the design resilient.

4.9 Acknowledgments

Chapter 4 contains reprints of Atieh Lotfi, Nirmal Saxena, Richard Bramley, Paul Racunas, Philip Shirvani, “Low Overhead Tag Error Mitigation for GPU Architectures”, *International Conference on Dependable Systems and Networks (DSN)*, 2018. The dissertation author is the primary author of the paper.

Chapter 5

Automated Source-code Optimization for Efficient Use of Hardware Resources

In this chapter, we propose an automated optimization approach that modifies the application at the source-level and results in more efficient use of hardware resources. Our optimization tool automatically detects and removes unnecessary redundancies that exist in a given GPU program or FPGA high-level design specification. In this chapter, we introduce our resource-aware source-to-source transformation solution. Further, we show its effectiveness on mitigating the performance overhead of core isolation in GPU accelerators with faulty units.

5.1 Introduction

As mentioned earlier, since hardware accelerators are being used in new large scale systems, it is becoming more important to use their available resources in a more efficient way. One efficient and useful solution is to exploit source-level optimization to reduce unnecessary and redundant computation in programs; which results in reducing resource requirements of an application while improving its performance and energy efficiency.

The opportunity for more optimization, that we take advantage of in this chapter, comes

from two trends. First, we have seen fast-paced code development by software developers which are not necessarily optimized and tuned for the target FPGA or GPU accelerator. This results in programs that have unnecessary redundant computations that has no effect on accuracy. Executing an unoptimized code on these platforms results in inefficient use of hardware resources. Second, many modern applications including graphics, multimedia, computer vision, web search, and data analytics exhibit tolerance to imprecision. These applications can accept a range of possible values as correct outputs for a given input. This amenability to *approximation* provides an opportunity to trade small controlled losses of quality (beyond the quantization error caused by limited bitwidth) for higher throughput and better resource utilization. Therefore, there is a need for automated frameworks that can find these redundant computations and modify the program accordingly to better utilize hardware resources.

In this chapter, we devise, GRATER, an automated workflow that optimizes data-type and bitwidth of operations and removes redundant computations. This results in improving resource utilization and achieving higher computational throughput for the target hardware accelerators. The core of our workflow is a source-to-source compiler that takes in an input kernel and applies a novel optimization technique that selectively reduces the precision of kernels data and operations. By systematically tuning the precision of the data and operation, the kernel can be run faster on GPU and FPGA, and the required area to synthesize the kernels on the FPGA decreases. GRATER, especially, provides a readily applicable workflow that exploits the inherent error tolerance of the emerging applications. To effectively explore the possible design space for precision tuning, we devise a genetic programming-based optimization algorithm that assigns various precision levels to different data and operations in the kernel. We exploit genetic programming to evolve kernel variants until one is found with optimal assignments that improves hardware resource utilization while stochastically satisfying the quality-of-result target. GRATER can target both OpenCL and C programs. We especially use OpenCL because it targets both GPU and FPGA platforms. We discussed the background material for OpenCL in chapter 2.

5.2 Background and Motivating Example

In this section, we show the benefit of bitwidth reduction for FPGA and GPU. We start the discussion with FPGAs. For this purpose, we show the difference in resource utilization of multiply operation with different types and bitwidths when it is synthesized on FPGA. For this experiment, we use Viavdo HLS tool and Virtex 7 FPGA. In this experiment, we changed the bitwidth of operands of multiply from 16-bit int, to 64-bit int, and to float and double data types. As can be seen in table 5.1, int16 (aka *short*) has the lowest resource utilization on FPGA; it only uses one DSP unit to perform the multiplication. The reason is that DSP units are used to implement multiply operation on FPGA and the size of inputs for one DSP unit is 18 bits for our target FPGA. Therefore, we get the same resource utilization for any bitwidth equal or less than 18 bits as inputs of multiplier. Having an int24 multiply is also efficient. Since the bitwidth of each operand (24-bits) is larger than the DSP unit input size, it cascades two 12-bit multiplier units (by utilizing 2 DSPs) and adds their results together; 2 LUTs are also used to implement the control unit. As we increase bitwidth of the operation, more number of DSP units and logic elements are utilized. As can be seen, *double* and int64 take the highest resource utilization. Therefore, it is important to find the lowest required bitwidth for each variable and operation during FPGA design to avoid waste of hardware resources.

Table 5.1: Resource utilization of multiply operation for different types/bitwidth on Virtex7 FPGA

Implementation	Int16	Int18	Int24	Int32	Int64	float	double
LUT	0	0	2	1	15	136	204
FF	0	0	0	3	18	130	304
DSP	1	1	2	4	16	3	11

For GPUs, the most inefficient operation is double precision floating point. For example, Radeon RX 580 has 6175 GFLOPS performance for FP16 (half-precision floating point) and FP32 (single-precision floating point) operations, and 385.9 GFLOPS for FP64 (double-precision

floating point) operation. This means that FP32 operations can be done 16× faster than FP64 operations. This performance difference between FP32 and FP64 is reduced for newer generations of GPU that are used for HPC purposes. For example, Nvidia Pascal GPU [7] has 21.2 TFLOPS performance for FP16, 10.6 TFLOPS performance for FP32, and 5.3 TFLOPS performance for FP64. Therefore, in those GPUs, we can improve the performance by 2× if we optimize a double-precision floating point operation to a single-precision floating point operation. In general, the performance gain from converting FP64 to FP32 can be between 2-32× for different GPU architectures. Some newer generations of GPU like Nvidia Pascal also gain from converting FP32 to FP16. Therefore, it is important for GPUs to reduce the bitwidth required for floating point operations. Usually, when some of the variables in a program requires FP64 precision, the programmer gets the same precision for other floating point operations while that is not really required to get the acceptable results. However, if the same program is written in a mixed-precision way, the performance improves significantly. On the other hand, there is not much difference in performance of FP32 and 32-bit integer operations on GPU. While int16 is faster than int32.

Finding the minimum bitwidth for each variable in a program is a tedious task that often programmers avoid to perform. GRATER can automatically search the useful search space for FPGA and GPU, and generate a program with tuned precision for its variables and operations. In the next section, we discuss our optimization workflow.

5.3 GRATER: Design Optimization Workflow

GRATER has a source-to-source compiler to generate variants of the baseline kernel via source-to-source kernel transformation. The transformation algorithm automatically detects and simplifies parts of the kernel code that can be executed with reduced precision while preserving the desired quality-of-result. To achieve this goal, GRATER takes in as inputs, a *baseline* kernel, a comprehensive set of input test cases, and a metric for measuring the quality-of-result target.

GRATER targets optimizing kernels both in the category of approximate computing applications, in which some degree of variation or inaccuracy is acceptable in the result of their computation; or applications that are not amenable to approximation, in which we want to tune the precision of their variables and operations. GRATER investigates the baseline kernel code and detects data elements, i.e., kernel variables, that provide possible opportunities for increased performance by tuning their precision. GRATER then automatically generates a set of kernel variants with lower precision that produce acceptable results. These modified kernels provide improved performance benefits for GPU accelerators. Moreover, they improve throughput and reduce the circuit area and resource utilization when implemented on FPGAs. GRATER outputs an optimized kernel with tuned precision for its variables and operations whose output quality satisfies the quality-of-result target. Fig. 6.4 illustrates an overview of our workflow.

GRATER uses the precision of the operations and data as a knob to tune performance and resource utilization. The transformation investigates a set of kernels where in each version, the bitwidth of some of variables are reduced. GRATER can optimize and translate both OpenCL and C programs. It mainly takes advantage of the portability of OpenCL code that can be run on both GPU and FPGA. We limit the space of our optimization search across the available variable types in OpenCL or HLS C, due to the nature of a source-to-source transformer that requires to work at the same level of abstraction of the input programming language. GRATER enables Altera OpenCL synthesis tool and Vivado HLS tool chains to benefit from this source-to-source translation by generating standard OpenCL and C kernels. To explain GRATER workflow, we mostly focus on OpenCL kernel tuning as it can be run on both GPU and FPGA. The optimization flow for OpenCL and C kernels are similar and we briefly explain the difference in the following sections.

We assign a precision tag (*PT*) to each variable type. For example, a kernel with data types ranging from double-precision floating point to char has five levels of complexity: {5, 4, 3, 2, 1} are assigned to {`double`, `float`, `int`, `short`, `char`} respectively. The higher the *PT*, the higher the accuracy requirements, and the higher resource consumption. A brute-force

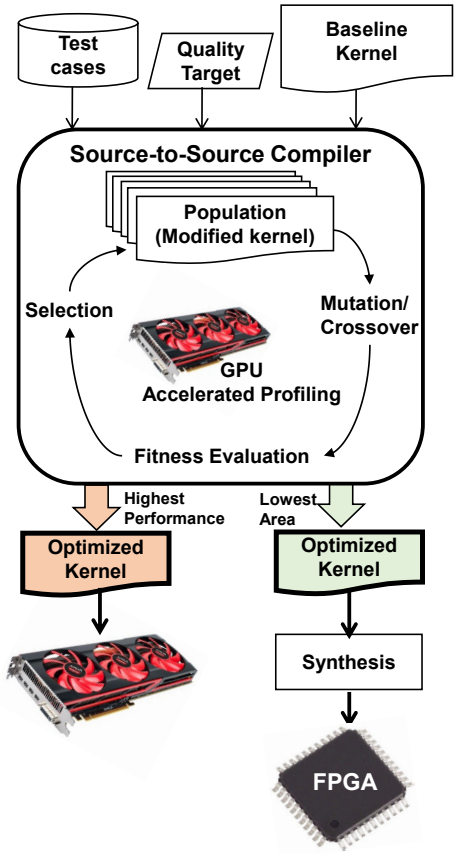


Figure 5.1: Overview of GRATER, our design optimization workflow.

methodology for exploring the kernel variants is to generate a modified kernel for every possible combination of the variable types. For instance, for a kernel with |V| number of `double` variables, a total number of $5^{|V|}$ kernels would be generated where in each version every `double` variable is replaced by different *PTs*. This results in an exponentially growing design space intractable to search. To avoid this huge design space exploration, we devise an algorithm that first detects those variables that are amenable to precision reduction and then applies a genetic-based algorithm to generate the variants of kernel. We discuss the details of our algorithm in the following subsections.

5.3.1 Analysis and Pruning

In the first step, GRATER detects variables in the code that are amenable to precision reduction. To do so, a separate kernel is generated for testing the amenability of each variable. In each kernel, the precision of one variable is demoted by one level (a ΔPT demoting), while other variables have their exact precision, to measure the *significance* of a small precision loss of a variable on the quality of result. This test determines whether the precision of the selected variable can be reduced or not. If the output quality is less than the desired output quality, GRATER excludes this variable from the set of *safe-to-tune* variables and does not modify its precision.[2] Consequently, the variable is eliminated from the candidate list of variables for precision reduction. The pruning algorithm continues the screening process for all the variables in the code (Line 4–10 in Algorithm 2). The pruning algorithm is executed |V| times to determine safe-to-tune variables (TV). This sensitivity test is done with the help of profiling feedback that is accelerated on a GPU.

GRATER then finds the lowest possible precision for each variable in TV (Line 11–14 in Algorithm 2). It generates a modified version of kernel for every variable in TV, where in each kernel, one variable type is replaced by the lowest possible precision (e.g. `char`) while other variables preserve their exact precision (EP) that originally have in the baseline code. If the quality of the generated kernel is less than the desired output quality, then that tentative lowest precision is promoted by one level and the same quality check is repeated. This process is continued until the lower precision bound for each variable is found. At this point, PT value ranges for each safe-to-approximate variable is extracted (from EP to LP).

After finding the lower precision bound for all variables in TV, another modified version of kernel is generated in which all safe-to-tune variables get their lowest possible precision (LP values) found in the previous step. If this kernel meets the quality-of-result target, the solution is found (Line 15–19 in Algorithm 2). Otherwise, a genetic algorithm, described in the following,

[2] GRATER also enables the programmer to annotate critical variables as non-tunable, so that the transcompiler would not change their precision.

Algorithm 2 pseudo-code for the GRATER

```
1: function PRUNE&RELAX(BaselineKernel, QualityTarget, inputSet)
2:     V = {All candidate variables in BaselineKernel}        TV = {}
3:     TopPop = {}        cInput = input_0...input_M from inputSet, M < |inputSet|
4:     for all variables v_i in V do
5:         generate kernel_i s.t. v_i ← ΔPT demoting
6:         run kernel_i with cInput on GPU
7:         if (Quality(kernel_i) ≥ QualityTarget) then
8:             TV = TV ∪ v_i
9:         end if
10:    end for
11:    for all variables v_i in TV do
12:        LP_i = FindLowerPT(v_i, cInput)
13:        EP_i = getExactPT(v_i, cInput)
14:    end for
15:    generate kernel_min s.t. ∀ v_i ← LP_i
16:    run kernel_min with cInput on GPU
17:    if (Quality (kernel_min) ≥ QualityTarget) then
18:        ModifiedKernel = kernel_min
19:    else
20:        ModifiedKernel, TopPop =  GA(BaselineKernel, LP, EP, cInput)
21:    end if
22:    for all input_i in the training inputSet do
23:        run ModifiedKernel with input_i on GPU
24:        if (Quality(ModifiedKernel) < QualityTarget) then
25:            NeedToChangeSolution = True
26:            for all kernel_j in TopPop do
27:                run kernel_j with input_i on GPU
28:                if (Quality(kernel_j) > QualityTarget) then
29:                    ModifiedKernel = kernel_j
30:                    NeedToChangeSolution = False
31:                    Break
32:                end if
33:            end for
34:            if (NeedToChangeSolution) then
35:                cInput = input_i
36:                Goto line 11
37:            end if
38:        end if
39:    end for
40:    return OptKernel
41: end function
```

is run to find the optimized kernel.

5.3.2 Genetic-based Design Space Exploration Algorithm

Genetic algorithm is a powerful stochastic search method which is deployed to find a good solution from a large search space [39]. To operate with a genetic algorithm, we need to take into account the following components: 1) a genetic representation of solutions in a form that can be interpreted as a chromosome, 2) an initial population, 3) a fitness function which

gives an evaluation of the desirability of each chromosome, and 4) genetic operators that change the composition of new generation during reproduction and a selection operator for choosing the survivors.

5.3.2.1 Genetic Representation of Chromosomes

We represent each individual as an array of precision tags with the length of TV list. Each gene in this representation shows the PT of each variable in TV. Every individual can be easily translated to a kernel variant. The precision of the variables and associated operations in the kernel variant is inferred from the assigned PT value in the chromosome.

5.3.2.2 Population

The initial population is randomly generated. Each safe-to-tune variable can have a PT value range with different levels of complexity, started from its lowest precision bound to its exact precision level (LP and EP in Algorithm 2).

All individuals in the population should meet the desired quality-of-result requirement. This can be verified either by executing the kernel or comparing its PT values with the least precision chromosome found. The least precision chromosome found in the population is the one that the PT values of every gene in its chromosome is lower than the PT values of corresponding genes in all other chromosomes. If such a chromosome does not exist in the population, the least precision PT in the population would be the same as LP. In this case, for all generated kernels we need a kernel execution for accuracy check. When the quality measurement test is done by executing a kernel variant, its output is compared with the baseline kernel output on a representative data input. If the output of the kernel variant cannot satisfy the quality-of-result target, this kernel variant is ruled out from the population. Otherwise, it is considered as one of the candidates for the next generation. This kernel profiling and execution process is accelerated on a GPU. This is accomplished by decoupling the quality loss analysis and kernel mapping

thanks to the platform-independent nature of OpenCL. To increase the speed of genetic algorithm, before creating and executing each kernel variant, the generated chromosome is compared to the chromosome with the least PT in the population so far. If all PT values in the newly generated chromosome is higher than or equal to the PT values of the least precision chromosome, this new chromosome can certainly meet the quality-of-result target; otherwise, the corresponding kernel should be executed for accuracy check.

5.3.2.3 Fitness Function

Given a kernel, the fitness function returns a value showing the desirability of the kernel variant. The fitness value is used by the selection operation to decide which individuals would survive to the next generation. Our main objective is to find a kernel variant that minimizes the cost of operations in the kernel. This improves performance on GPU and also reduces resource utilization and improves throughput on FPGA, while meeting the quality-of-result requirement. To achieve this objective, our fitness function computes a weighted summation of its assigned PT values in the chromosome to estimate the area occupancy. For each variable, the weight is determined by a coefficient assigned to each precision tag multiplied by the number of times the variable is used in operations in the kernel. (The coefficients are determined through simulations. For example for FPGA, it is $\{0,1,2,6,12\}$ for PT of $\{1,2,3,4,5\}$. For GPU, it is $\{1,1,4,6,40\}$ for PT of $\{1,2,3,4,5\}$ The higher the precision and the number of times the variable is used in operations, the higher weight it gets. With this definition, the lower the fitness value, the higher performance and the lower area occupancy that configuration has.

It should be noted that GPUs mainly benefit from reducing precision from double-precision to single-precision floating point. In fact, GPUs are good at doing integer and float computation except from 64-bit double precision (*FP64*). This is because GPUs has more integer and single-precision (*FP32*) cores than *FP64* cores as we discussed in section 5.2. On the other hand, FPGAs not even benefit from double-precision to single-precision floating point but

also from floating point to integer and even from reducing bitwidth of integer operations. The difference between this behaviour between GPU and FPGA is due to the fact that GPUs have fixed cores that support certain operation types while the generated circuit on FPGA is fully customized.

5.3.2.4 Selection and Genetic Operators

We use two genetic operators, crossover and mutation, to produce new chromosomes. Crossover combines the first part from one parent chromosome to the second part from the other parent chromosome to produce a child chromosome. In this implementation, the crossover point is selected randomly. Mutation operation randomly modifies PT values of safe-to-tune variables in the chromosome. The new PT value is a random value in the range of LP and EP for the safe-to-tune variable. The newly generated chromosome is only accepted if it meets the quality-of-result requirement; otherwise, the operation is applied again.

There are many possible selection algorithms to select more fit individuals from the new and old population for the next generation. To rank the kernel variants, we use the fitness values as an estimate of the area occupancy. This fitness function is a good estimation for resource utilization of FPGA (without synthesizing and mapping the kernel on FPGA). The selected chromosomes are sorted based on their estimation of area occupancy (fitness value) in each iteration. The top best individuals are always transferred for the next generation (elitism selection). For the rest, individuals are selected based on the proportionate selection where some of them might change with the crossover and mutation operations. For the simulation purpose, the crossover rate, mutation rate, and elitism rate is 0.7, 0.05, and 0.25 respectively. The algorithm runs as long as the user defined number of iterations has not been passed yet or when the best fitness values stop growing any further.

Until here, the genetic algorithm finds the final solution using only a subset of input tests. This solution should be verified with the other input test cases from the training set. If

it meets the quality-of-result requirement for all inputs of this set, this kernel variant is the final solution. Otherwise, either the other top chromosomes in the population is checked or the genetic algorithm is applied again by adding the failed input to the initial test set (Line 23–40 in Algorithm 2).

When this procedure is terminated, the best chromosome with the lowest fitness value is selected and translated to its corresponding kernel. When optimizing for GPU, this kernel has the best performance among the variants. When optimizing for FPGA, it has the least resource utilization on FPGA. To download the kernel on FPGA, this kernel is passed to the Altera SDK tool to be synthesized and mapped on the FPGA.

The accelerated profiling on GPU makes us to explore the design space faster. In fact, executing all kernels and selecting the best on FPGA is very time consuming and even impossible because this process is very slow on FPGAs. Moving the search space exploration from FPGA to GPU saves a considerable amount of time.

5.3.3 GRATER for C and HLS targeting FPGAs

GRATER also works for programs written in C where we use HLS tool to synthesize programs on FPGA. We target Vivado HLS for synthesis where it enables a wider range of bitwidth exploration. In contrast to OpenCL to FPGA tool, this tool has the feature to define any arbitrary size bitwidth for integer operations. By using this tool, we can increase the search space and create more opportunity for optimization. As we have seen in the example in section 5.2, reducing the bitwidth of integer operations (e.g. from *int64* to *int18* or less) can significantly reduce resource utilization. Therefore, depending on the architecture of FPGA, we select the effective bitwidths as our search space in the algorithm. For example, for Xilinx *Virtex* FPGAs, our candidate type and bitwidth would be: {`double`,`float`, `int64`, `int36`, `int32`, `int24`, `int18`, `int16`, `int8`}. We select these numbers because of the input size of operations for DSP units on this FPDA series. GRATER can get the target architecture type to redefine this

search space so that it better matches the target architecture. For profiling generated kernels, we use CPU instead of GPU for this case.

5.4 Experimental Results

5.4.1 Experimental Setup

We focus on a diverse set of application domains, including image processing (r-gaussian, sobel), signal processing (convolution, DCT), physical simulation (n-body, fluid), and finance (b-scholes). These benchmarks are selected from the AMD accelerated parallel processing (APP) SDK v2.9 [10]; The number of variables in these kernels are in the range between 11 and 21.

GRATER source-to-source compiler is implemented in Python, and accepts the baseline kernel implemented in either OpenCL or C, the desired quality metric, and a set of training input test cases as its inputs. We used 150 training inputs for out experiments. GRATER utilizes the AMD Radeon RX 580 (Ellesmere) GPU device for accelerated profiling experiments, and finally generates an optimized kernel with better performance on GPU, and lower area occupancy and better throughput on FPGA, with an acceptable output. The optimized OpenCL kernel is synthesized for Altera DE5 board with a Stratix V FPGA using Altera OpenCL SDK 14.1 tool [1]. It should be noted that the accelerated profiling process on GPU takes order of milliseconds to determine if the kernel can meet the quality-of-result target. While it takes on average more than an hour to synthesize the approximate OpenCL kernels on Stratix V FPGA. In the following, we report the improvements resulted by kernels optimized by GRATER for FPGA and GPU.

5.4.2 Improvements for FPGA

Sections 6.4.2.1 and 5.4.2.2 detail how GRATER can reduce the area and correspondingly increase the throughput for different applications on FPGA. The target FPGA device has 234720 logic elements, 256 DSP blocks, 939 K registers, 2560 M20K memory blocks.

Table 5.2: Resource utilization for *Baseline* and GRATER-optimized kernels on StratixV FPGA

BM	Version	Logic	Registers	DSP	RAM
b-scholes	Baseline	135903	398606	864	899
	GRATER	72764	213308	256	712
conv	Baseline	54884	91613	8	387
	GRATER	52217	80325	3	371
dct	Baseline	52472	85359	18	525
	GRATER	50600	81046	12	455
fluid	Baseline	89170	171832	126	644
	GRATER	88250	166483	85	563
n-body	Baseline	100929	172894	77	208
	GRATER	58680	95156	54	103
r-gaussian	Baseline	64781	110481	42	518
	GRATER	57000	90177	25	473
sobel	Baseline	62448	99425	50	419
	GRATER	48135	74147	50	419
Avg. Area Reduction		18.95%	22.24%	38.45%	15.73%

5.4.2.1 Area Savings

Table 5.2 shows the resource utilization for the *baseline* and GRATER-*optimized* kernel. The *baseline* kernel is the input kernel, and the GRATER-*optimized* is the kernel which is optimized by GRATER.

As shown, the area utilization is reduced by an average of 15%–38% for different FPGA resources using the transformed kernel instead of the baseline one. This gain is achieved by the proper precision tuning of the kernel variable types that brings area saving without scarifying the output quality. For example, the baseline r-gaussian kernel has 12 `float` variables, while in the GRATER-optimized kernel 5 of these variables are replaced by 16-bit integer (`short` type) and 1 of them by 8-bit integer (`char` type). This precision tuning reduces the *logic* utilization

by 12%, *register* utilization by 18%, *DSP* block utilization by 40%, and *RAM* utilization by 8% for this application. For b-scholes this area saving comes from reducing 8 out of 21, 64-bit double-precision floating point variables to 32-bit single-precision floating point variables. This modification reduces *logic*, *register*, *DSP*, and *RAM* utilization by 46%, 46%, 70%, and 20% respectively. The difference in resource saving for different applications is because they contain different number and types of operations, and different number of *safe-to-tune* variables and reducing the bitwidth for each of them affect the area of circuit in a different way.

5.4.2.2 Throughput speedup

As shown in Section 6.4.2.1, GRATER reduces the synthesized area for the approximate kernels on the FPGA. Therefore, the number of parallel kernels (i.e., the kernel pipelines) that can be fitted into the FPGA is increased resulting in higher throughput.[3] Fig. 5.2 shows the corresponding kernel speedup – throughput of the optimized kernel normalized to throughput of the baseline kernel.

As an example, for the n-body kernel the number of kernel pipelines that can be mapped into FPGA is increased from 2 for the baseline kernel to 4 for the optimized kernel. Also the latency of each kernel is improved by a factor of 1.7×. Therefore, the optimize n-body kernel reaches to 3.5× higher throughput compared to the baseline kernel. A geometric mean of 1.83×

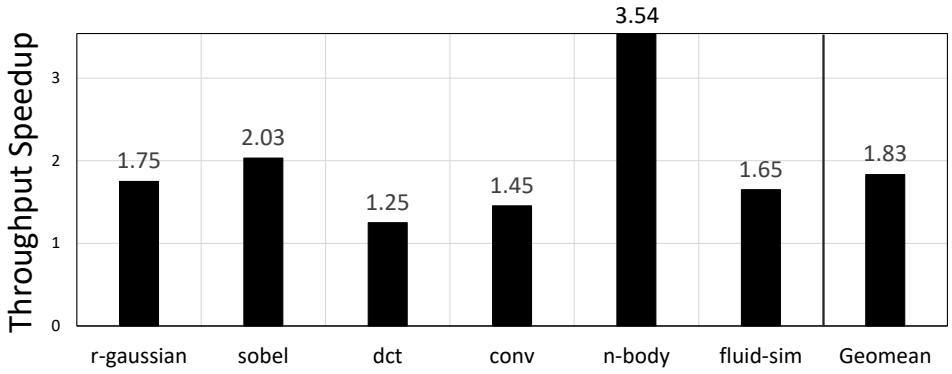


Figure 5.2: Throughput speedup with GRATER on FPGA.

[3]Replication is handled in Altera OpenCL by setting `num_compute_units` as a kernel attribute.

higher throughput is achieved across these evaluated benchmarks.

5.4.2.3 Power

Figure 5.3 shows the improvement in power using GRATER-optimized kernel comparing to the baseline kernel. Since GRATER makes the design smaller and also improves latency of design, it reduces power consumption as well. For our applications, GRATER improves power between 1.5%-12.3% by a geometric mean of 4.3%. This improvement is due to removing redundant computations and resources from the design.

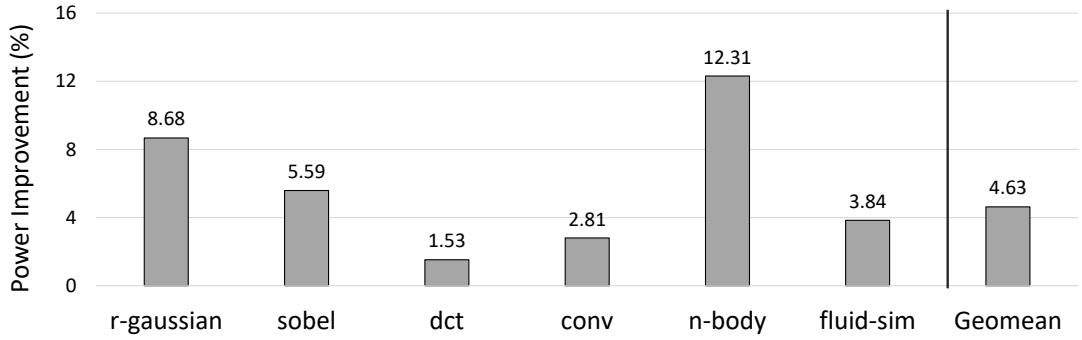


Figure 5.3: Power improvement with GRATER on FPGA.

5.4.3 Improvement for GPU

5.4.3.1 Performance

Radeon RX 580 has 6175 GFLOPS performance for FP16 (half-precision floating point) and FP32 (single-precision floating point) operations, and 385.9 GFLOPS for FP64 (double-precision floating point) operation. This means that FP32 operations can be done 16× faster than FP64 operations. Fig. 5.4 summarizes the performance speedup for executing the optimized kernels on the GPU, comparing to the baseline kernel execution time. The GPU exhibits a maximum speedup of 10× (with a geometric mean of 2.2×). The highest improvement is for b-scholes application, as it has the highest number of double to float conversion. On the other hand, sobel has the lowest improvement, because the conversions are only within integer type.

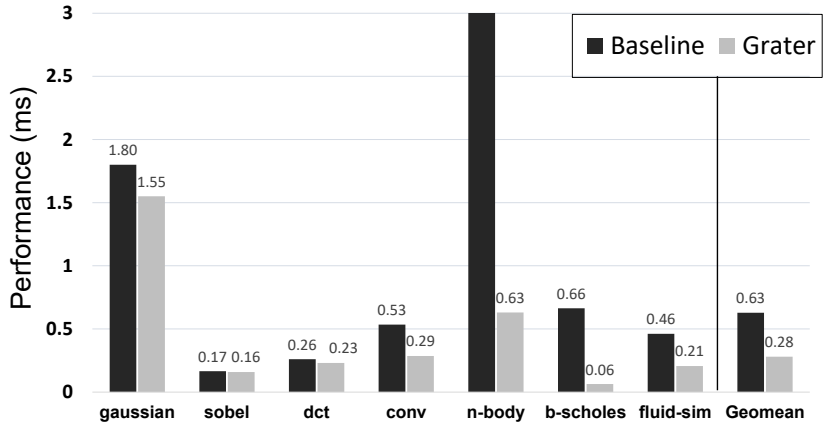


Figure 5.4: Performance speedup with GRATER on GPU.

5.4.3.2 Energy

Fig. 5.5 summarizes the energy improvement of using the optimized kernel on GPU comparing to the energy of baseline kernel. As can be seen in this figure, energy consumption is reduced by 2.2× on average. The energy consumption on GPU follows the same behaviour as performance.

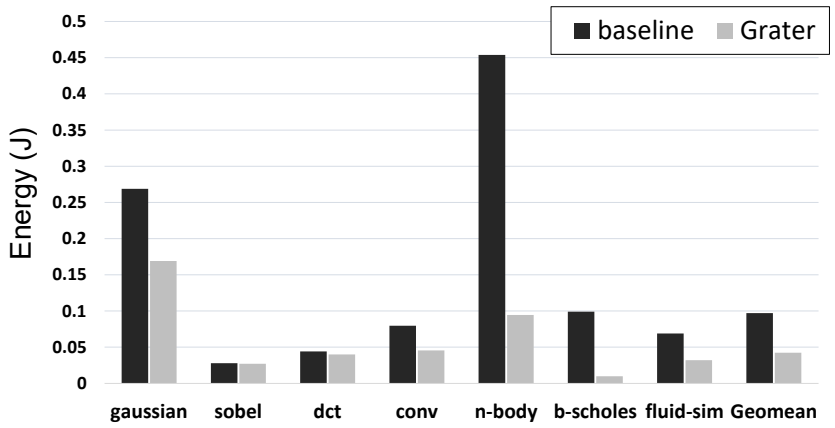


Figure 5.5: Energy improvement with GRATER on GPU.

5.4.4 Quality

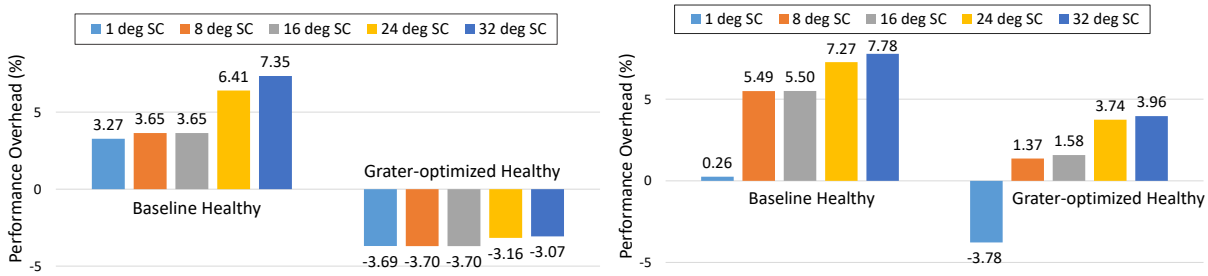
To evaluate the quality loss, we use PSNR for image processing applications and average relative error for the other application domains [81, 100]. We compute the quality loss of

each optimized kernel by comparing against the output elements from the baseline kernel. For simplicity, here we report the quality loss of all applications by average relative error metric. We set the quality loss target to a maximum of 0.7% for image processing applications (which is equivalent to PSNR of a minimum 30 dB) and 1% for other applications which is conservatively aligned with other work on quality trade-offs [57, 67, 83]. We verify the output quality of the optimized kernel with 150 different test input patterns, other than the training input set. In all applications, the maximum quality loss is below 1%. For sobel and r-gaussian there is no quality loss.

The execution time of our proposed algorithm is within few seconds when it can find the solution without running the genetic algorithm to couple of minutes when we run genetic algorithm. GRATER optimizes the code at design time.

5.4.5 Impact of GRATER on GPU Isolation

We performed experiments to show the benefit of using GRATER optimization to reduce the overhead of core isolation in a faulty GPU described in Section 3. For this reason, we select two of benchmarks and compare the overhead of running an un-optimized healthy kernel and a healthy kernel that is optimized with GRATER for GPU. Figure 5.6 shows performance overhead of core isolation for the baseline healthy kernel and Grater-optimized healthy kernel for b-scholes and sobel applications. As can be seen in this figure, using GRATER can reduce the performance overhead of core isolation, and even remove it completely and make it faster. For example, for sobel, the performance overhead when 32 SCs are degraded is reduced from 7% to 3%. For b-scholes, we even can run the healthy kernel 3% faster instead of 7% overhead if we apply GRATER on healthy kernel. Therefore, one method to reduce the performance overhead caused by core isolation is to further optimize the kernel by slightly losing accuracy if possible.



(a) BlackScholes (b) SobelFilter

Figure 5.6: Performance overhead comparison for GPU core isolation on Radeon 580rx for the baseline healthy kernel and grater-optimized healthy kernel ((a) b-scholes and (b) sobel) when number of degraded SCs are changed from 1 to 32.

5.5 Related Work

Floating point precision tuning have been used to improve performance of programs. Precimonious [101] tool proposed a delta-debugging approach for floating point programs written in C running on CPU. This tool focuses on floating point precision analysis and provides hints to the programmer on which data type is required while satisfying a given error threshold on the output. STOKE [103] is a JIT assembler for x86-64 instruction set that tunes floating-point kernels based on random search. The optimized assembly can be run on CPU. However, these tools focus only on CPUs and do not deal with accelerators neither they support OpenCL.

Bitwidth optimizations for FPGA is used to improve power and resource utilization of the design. PowerBit [43] performs static range analysis and precision analysis to optimize the bitwidth of datapath and generate an optimized VHDL description. Using range analysis is very conservative and the answer is usually far from optimum. This work does not support high-level languages and program transformation. SOAP [44] is a source-to-source transformation tool for optimization of numerical programs that targets HLS-based FPGA design. They target rewriting floating point expressions to trade the numerical accuracy and resource utilization of design. This work discovers equivalent structures by exploiting the rules of arithmetic, such as associativity and distributivity. This is in contrast with our work that targets bitwidth and type optimization. In fact, this work can be combined with GRATER to further reduce resource utilization of design.

A synthesis method for generating approximate circuits from RTL code is proposed in [84]. The goal of this work is to reduce the power consumption and area. It uses a greedy approach to generate approximate versions, and selects the near-optimal design after simulating and synthesizing all design variants. In contrast to this work, our tool works at higher level and can be used with HLS tools. Floating point to fixed-point conversion is another optimization to limit the area and power consumption of the architecture. Methods have been developed to support the automatic conversion of floating point values to fixed-point ones for a specific domain of applications. For example, the conversion in [32] only operates on programs with regular control structure that can be represented using the polyhedral model. None of the tools provide a complete solution for fixed-point conversion. We did not implement this conversion in GRATER, but it can be easily added to the search space of our tool. For mixed-precision tuning on GPU, common approach requires the programmer to manually substitute the data type in the problem. There is no specific tool that targets automatic precision tuning for GPU code.

In contrast, GRATER focuses on high-level bitwidth and data type optimization for programs targeting FPGA and GPU. This optimization improves throughput and reduces resource utilization on FPGA when the kernel is synthesized, and improves performance when executed on GPU. Performing the optimization in high level enables GRATER to efficiently explore the design space of the kernel. GRATER allows seamless integration of mixed-precision data elements within a software-level kernel to cooperatively work on the same hardware fabric.

5.6 Chapter Summary

In this chapter, we propose an *automated* optimization tool that modifies the application in source-level and results in more efficient use of hardware resources. We devise GRATER, a transcompiler that systematically transforms a kernel to a more optimized version. This optimization removes unnecessary redundancies in computations by tuning the bitwidth of operations. To effectively explore the search space, GRATER employs a genetic-based algorithm

to find suitable optimizations. The use of GRATER for FPGA and GPU accelerators results in improvements in performance, throughput, resource utilization, and energy efficiency. This improvement comes with *no effort* from the user. This automated solution especially helps reduce the overhead caused by isolation and task migration in a faulty accelerator.

5.7 Acknowledgment

Chapter 5, in part, contains reprints of Atieh Lotfi, Abbas Rahimi, Amir Yazdanbakhsh, Hadi Esmaeilzadeh, Rajesh Gupta, "GRATER: An Approximation Workflow for Exploiting Data-Level Parallelism in FPGA Acceleration", *design automation and test in Europe (DATE)*, 2016; and Atieh Lotfi, Abbas Rahimi, Hadi Esmaeilzadeh, and Rajesh Gupta, "SqueezCL: Squeezing OpenCL Kernels for Approximate Computing on Contemporary GPUs", Workshop on Approximate Computing, 2016. The dissertation author is the primary investigator and author of papers.

Chapter 6

Resource-aware Task Migration in HLS-based FPGA Design

In the previous chapter, we proposed a source-level solution that optimizes the program by automatically tuning the bitwidth of its variables and operations, which results in more efficient use of GPU or FPGA accelerators. In this chapter, we propose a workflow for efficient static task migration and resource optimization in HLS-based FPGA design. Despite considerable improvements in existing HLS tools, they still require designer interventions to provide efficient synthesis results. This manual design space exploration and code rewriting and optimization takes significant time and negates the HLS design productivity gains. To overcome this challenge, our workflow uses compiler frontend as an independent preprocessing step to explore the design space and adds an automated source-to-source transformation step before HLS. We propose a compiler-level approach that enables reusing the available resources for common tasks in the design in HLS-based FPGA design. This workflow improves resource usage by automatically taking the opportunity of resource sharing. In particular, it shows how inherent regularity in applications can be used to construct a workflow that analyzes the program, explores the design space for resource optimization opportunity, and transforms the program accordingly. When the transformed program is synthesized using the HLS tool, it uses less hardware resources

with similar latency and energy consumption comparing to the original design. The proposed workflow is useful for either fully-functional or faulty FPGA device especially when the design is not fitted on FPGA.

6.1 Introduction

FPGAs, despite their growing size, are inherently resource-constrained both in physical gates as well as routing and memory resources that ultimately limit the size of the accelerator designs. Fitting a given design or maximizing utilization of FPGA resources is usually done through a lengthy iterative process in programming the accelerator code and steering through a highly parameterized logic synthesis and mapping process that often demands considerable expertise in hardware design. This poses a serious challenge to meet the time-to-market requirement especially for large applications. HLS tools help improve productivity by raising the programming abstraction from hardware description languages to higher level languages like C, C++, OpenCL, or SystemC. Despite the advances in HLS design automation [29], the quality of synthesis results is still not comparable to hand-coded RTL designs, requiring an expert designer effort to optimize and rewrite the source code especially when a design pushes resource limits. Designers often need to manually explore a large design space to find the best design option among many different design alternatives. This manual exploration and code transformations takes significant time, requires knowledge of hardware microarchitecture and the coding style of HLS tool, which negates the HLS design productivity gains.

We presents a practical source-level design exploration and mapping workflow that enables automatic source-to-source transformation to improve resource usage using an HLS design flow. This transformation workflow is especially helpful to automatically make the design smaller as we have even more resource limitations on an FPGA with faulty resources. Different system-level optimizations and code restructuring can be applied on a given application specification, where each transformation impacts differently on resource utilization and perfor-

mance of the design after synthesis. In this section, we focus on a class of optimization and automated code transformations that results in improving hardware resource utilization without noticeable overhead on the performance of design. These transformations seek to reuse the same instance of a hardware component, or *resource sharing*, for parts of the design that have recurring sequence of operations, or computational *patterns* [49]. Common patterns of operations in the design, or *regularities*, can be extracted in every level of circuit design, from layout and netlist representation of HDL description [25] [69] to higher-level application specification [27]. Indeed, an intelligent use of regularity is a key reason why the manual design and optimization often excels the design synthesized by automated HLS tools.

To be sure, off-the-shelf HLS tools (e.g. [5]) are partly capable of exploiting regularities for resource sharing. For example, resource sharing is often done in HLS tools automatically when the exact same function is invoked multiple times from the same caller function (if the callees are not meant to run at the same time). Loops are also other parts that define regularities and HLS tools apply resource sharing by default. However, in the absence of a canonic representation, the implementation of such "common expression" detection and use in current HLS tools varies across tools. In fact, if the repeated sequence of operations are not wrapped in a function or loop, current HLS tools do not consider resource sharing as an optimization strategy, even if those operations do not have overlapping execution time.

Therefore, expanding the automated design space exploration to detect regularities at a higher level design can reduce the design time and improve the resource usage and design quality. At the same time, excessive resource sharing decisions could introduce more overhead than benefit due to need for time-multiplexed control that can be expensive on FPGA implementation targets. If resource sharing is done in inappropriate parts of the design, it might negatively affect resource utilization or performance of the design. We present an automated workflow that can identify the program inherent regularities that are not automatically detected by HLS tools, and decide if sharing resources for instances of those patterns would create a smaller design with little or no performance degradation. This information is used to automatically modify the source

code in a way that guides the HLS tool to perform resource sharing for the selected parts of the code. Using this automatically modified code, the HLS tool can provide a more efficient solution that consumes less resources when synthesized on FPGA. This transformation is especially useful when there are corrupted blocks on FPGA.

Accordingly, we make the following contributions:

1. We present a pre-synthesis regularity extraction and resource-aware task migration workflow for FPGAs that can help fitting the design on the available resources of FPGA. This workflow, called REHLS, automatically finds regularities in the design specification, and evaluates the usefulness of sharing hardware resources among them.

2. We implement a source-to-source transformation tool to automatically generate the improved HLS-C design. When this new specification is used by the HLS tool, another functionally equivalent design with different scheduling and resource usage is generated. This transformation results in reducing area of the design on FPGA with low performance overhead. This results in improving throughput (performance of parallel replicas) of the design.

To the best of our knowledge this is the first work on automated source-to-source transformation for FPGA resource optimization through regularity extraction and resource sharing. This transformation automatically explores the design space and makes appropriate decisions in higher level of abstraction. This reduces design time and especially helps non-experts designers create a more efficient design.

6.2 Motivating Example

In this section, we demonstrate a simple motivational example to explain the benefit of our resource-aware regularity extraction and task migration workflow.

Figure 6.1(a) shows a small example code. This code segment consists of one loop that performs the same sequence of operations with the same dependency and order (labeled as PI1, PI2, and PI3). We call this repeating sequence of operations, a *pattern*. The pattern for this code is shown in Figure 6.1(b).

During synthesis, the HLS tool specifies the operations in the source code (e.g. multiplications, additions, ...) and then maps them to hardware cores that implement these operations (e.g. multipliers, adders, ...). Each of these hardware cores uses different number of FPGA elements (LUT, FF, DSP). Each of PI1, PI2, and PI3 in our example code has 1 multiplication, 1 addition, and 1 division operation. Therefore, for each of them, the HLS tool gets 1 multiplier, 1 adder, and 1 divider. Figure 6.1(c) shows the hardware units (*FU*) that are allocated for implementing PI1, PI2, and PI3. As can be seen, because PI1, PI2, and PI3 have the same computational pattern, the same hardware unit *FU* is used three times. Thus, for this program, the HLS tool generates a design that uses 3 multipliers, 3 adders, and 3 dividers to implement PI1, PI2, and PI3. We call this design the *baseline* design. However, alternative designs can be generated to reduce the resource usage by sharing the hardware unit (*FU*) for these pattern instances.

The first alternative design (*design 1*) is generated by allocating *two* instances of the hardware unit *FU*; one of them is shared between PI1 and PI2 and another one for PI3. With this implementation, only 2 multiplier, 2 adder, and 2 divider is required and four multiplexers are added to the input of the shared FU. (Figure 6.2(b)) In general, when we share hardware resources, computations that use the same hardware resources must be executed sequentially. Therefore, this might increase the latency of design. However, in this example, because PI2 has dependency to PI1 (PI2 can not start its computation before the output of PI1 is ready), the operations in these two loops do not have overlapping lifetime. Therefore, we expect negligible effect on latency and throughput as the result of resource sharing for this design.

The second alternative design (*design 2*) can be generated by allocating only *one* instance of the hardware unit *FU*, which is shared between PI1, PI2, PI3. With this implementation, only 1 multiplier, 1 adder, and 1 divider is required and multiplexers are added to the input of the

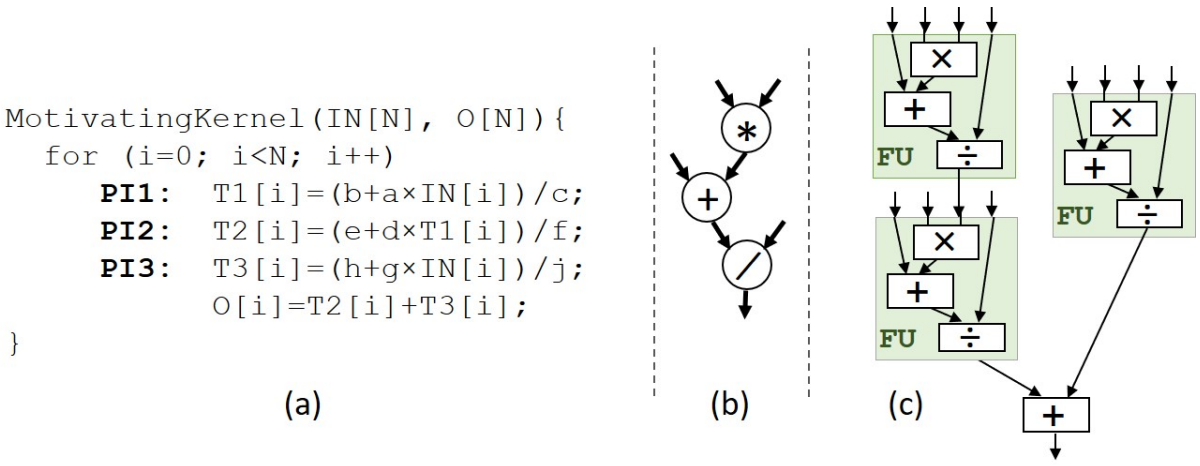


Figure 6.1: (a) example code, (b) its pattern, (c) the design generated for this program

shared FU. (Figure 6.3(b)) This design has the lowest resource utilization among these three designs. However, since PI3 has no dependency to PI1 and PI2 and can be run in parallel with them, the effect of this modification is not negligible on the latency and throughput of the design.

In order to guide the HLS tool to share resources for the instances of this pattern, some transformations on the code is required. The new versions of the code for *design 1* and *design 2* are shown in Figure 6.2(a) and Figure 6.3(a). In the transformed code, each operation in the pattern is defined as a function. Instead of using the operations in C, this function is called for all instances of the pattern. This way the number of hardware units allocated to the instances of pattern can be limited to one through the *ALLOCATION* pragma. (We explain the reason that we perform the transformation this way in section 6.3.0.2.)

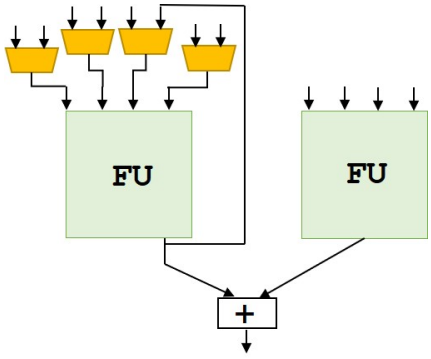
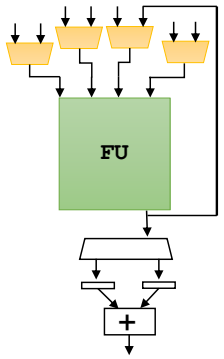


Figure 6.2: *Design 1*. Timing-aware resource sharing for baseline design of Fig 6.1

```
int mul(int a, int b){
   return (a × b);
}
int add(int a, int b){
   return (a + b);
}
int div(int a, int b){
   return (a / b);
}
MotivatingKernel(IN[N], O[N]){
#pragma HLS ALLOCATION instances=mul limit=1 function
#pragma HLS ALLOCATION instances=add limit=1 function
#pragma HLS ALLOCATION instances=div limit=1 function
   for (i=0; i<N; i++)
        PI1: T1[i]=div(add(b, mul(a, IN[i])), c);
        PI2: T2[i]=div(add(e, mul(d, T1[i])), f);
        PI3: T2[i]=div(add(h, mul(g, IN[i])), j);
             O[i]=T2[i]+T3[i];
}
```

(a)



(b)

Figure 6.3: (a) Code for *Design 2*. Aggressive resource sharing for baseline design of Fig 6.1, (b) the design generated for this program

Table 6.1: Resource utilization and timing comparison for Fig 6.1, 6.2 & 6.3

Implementation	LUT	FF	DSP	CP(ns)	Latency(cycles)
Baseline	2936	2919	30	8.28	8601
Design 1	2385	2289	20	8.28	8701
Design 2	1318	1151	10	8.28	11401

To evaluate these implementations, we use Vivado HLS tool [5], and Xilinx Virtex-7 XC7V585T FPGA device. Resource utilization results are after placement and route. Table 6.1 shows the resource utilization and timing comparison of *baseline* (Fig. 6.1) and *Design 1* (Fig. 6.2) and *Design 2* (Fig. 6.3) implementations. From the table, we observe that sharing resources for instances of the detected pattern can significantly reduce resource utilization (Columns *LUT*, *FF*, *DSP*) without affecting the timing results and performance (shown in the last three columns).

As we can observe, *design 1* utilizes 551, 630, and 10 less LUT, FF, and DSP elements comparing to the *baseline* implementation, with only 1.1% overhead on latency of the design. In this example, sharing hardware resources for the selected pattern does not affect the latency of

the design, because these these composite instructions must be executed sequentially due to data dependency of *PI2* to *PI1*. For this example, adding multiplexers does not make the critical path longer, therefore, the clock cycle remains the same. On the other hand, *design 2* utilizes 1618, 1768, and 20 less LUT, FF, and DSP elements comparing to the *baseline* implementation. But this huge benefit comes with 32% overhead on latency of the design. This shows that aggressive resource sharing for this design negatively affects the latency of design, because we change the scheduling from having concurrent units to a sequential design. It should be noted that this transformation only affects the logic elements and do not change the number of memory elements (BRAMs) on FPGA.

This example demonstrates that sharing resources for common patterns in the design *might* be useful. However, regularity extraction and resource sharing would not always result in a better solution. In some cases, it might even increase the resource utilization or highly decrease performance of the design. The effect of resource sharing on resource utilization after this kind of transformation depends on many factors, including the granularity of the selected patterns, the frequency of the pattern, and the type and resource utilization of operators within the selected pattern. Also, if the instances of a pattern are executed simultaneously in the baseline design, sharing resources for them decreases performance. Therefore, it is very important to apply this transformation only when it is useful. In the following section, we discuss about REHLS as our solution that can make this decision and perform required transformation at the program source level.

6.3 Resource-Aware Regularity Extraction and Task Migration Workflow

In this section, we describe our resource-aware regularity extraction and task migration workflow for HLS-based FPGA design. Given a C program with inherent computational

patterns, REHLS explores the opportunities for design optimization through resource sharing, and automatically transforms the program so that the HLS tool can generate a smaller design. Fig. 6.4 illustrates an overview of the automated resource sharing tool. First, the program is converted to LLVM intermediate representation (IR) [70]; the analysis and transformation is done through LLVM passes; and finally, the transformed LLVM IR is converted back to a C program. Because the HLS tools perform some compiler-level optimizations on the source code, we extract the optimized LLVM IR from the HLS tool by synthesizing the baseline program once. The optimized LLVM IR is accessible even from the commercial HLS tool that we used for our experiments. In case the HLS tool does not provide the optimized IR, it can be generated using Clang compiler and can be optimized with the available optimization passes. The main components of REHLS are *program analysis* and *program transformation*. The program analysis step performs *pattern detection* and *pattern selection*. The pattern detection step finds all patterns repeated among different basic blocks in a given program. The pattern selection step decides if sharing hardware resources for the instances of detected patterns can be beneficial or not. The goal is to select all patterns that can share hardware resources between their instances, and improve resource utilization (with low performance overhead) comparing to the baseline design. If any pattern is selected, the program transformation step automatically applies appropriate changes to the LLVM IR of the baseline program, and further transforms it to another C program. The transformed C program along with a generated directive file guide the HLS tool to share resources for the selected parts. These steps are explained in the following subsections.

6.3.0.1 Program Analysis

Given an input C program, the first step is to detect those patterns in the program that can share hardware resources, and further select some of them that result in reducing resource usage of the synthesized design.

Pattern Detection:

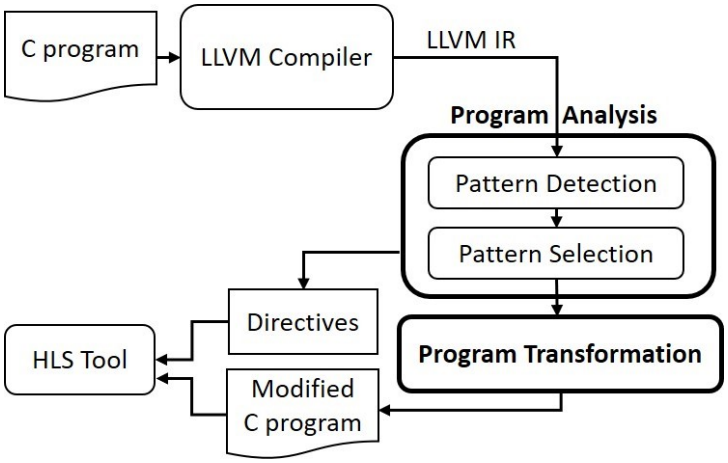


Figure 6.4: Overview of REHLS, our resource-aware regularity extraction tool

REHLS finds all patterns that are repeated inside data flow graphs (DFG) in each function in the program. DFG is a directed acyclic graph G(V, E), in which nodes in V represent operations, and edges in E represent data dependencies or data transfers between operations. Our goal is to find common subgraphs among DFGs. We call the common subgraph a *pattern*. Each pattern graph has a single output node. The number of nodes in a pattern graph is referred to as its size. Given a pair of DFGs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the goal is to enumerate all the subgraphs of G_1 that are isomorphic to subgraphs of G_2, where functionality, type, and bitwidth of corresponding nodes of V_1 and V_2 are the same. We require the nodes in a common graph to reside in the same *basic block*, where a basic block is a contiguous set of instructions with a single entry point and a single exit point.

Our algorithm detects patterns with a breadth-first search approach. To reduce the search space in our pattern detection step, the subgraph enumeration process is incremental, meaning that size *i+1* subgraphs are enumerated when all the size *i* subgraphs are enumerated. If a size *i* subgraph is not frequent enough, it is removed and no further considered for creating new subgraphs of size *i+1*. Our pattern detection pseudo-code is summarized in Algorithm 3. In each iteration *i*, our algorithm finds all patterns of size *i* (with *i* nodes) in all DFGs within a module. To do so, in iteration *i*, we extend the detected patterns of size *i* by adding one of

their neighbor nodes in DFG (Lines 9–12 in Algorithm 3). This method ensures that we only investigate those subgraphs that can be a potential pattern. This is true because if a subgraph of size *i* is not a pattern across multiple DFGs, none of its expansions with size *i+1* could be. The goal in each iteration is to compare the subgraphs of size *i+1* to each other, and find those that are frequent enough. (Lines 13–25 in Algorithm 3). For each subgraph of size *i+1*, if the number of occurrences across different DFGs are more than an acceptable *frequency limit*, we add it to the set of candidate patterns. Otherwise, it is not considered for finding larger subgraphs (Line 23 in Algorithm 3). We also record the instances of every found pattern. This information is required for the pattern selection step. For all newly found patterns of size *i+1*, we also remove all its subgraphs of size *i* that are in the pattern set, if their frequency is the same; because if two patterns have the same frequency, the larger pattern is always more useful for resource sharing purpose, because it results in more area saving (Lines 17–21 in Algorithm 3). With this pruning, we significantly reduce the number of detected patterns in the final set, which makes the pattern selection step easier. After finding candidate patterns of size i+1, we expand each candidate subgraph by adding its neighboring node, and repeat the above steps to identify larger candidates. This process is repeated until either no more new pattern is found (Lines 26–28 in Algorithm 3) or we reach the maximum size of subgraphs we can find.

If the pattern detector finds any candidate pattern in this phase, we investigate the effectiveness of resource sharing and further optimization in the next phase.

Pattern Selection:

So far, we have found a set of computational patterns that are repeated more than a frequency limit in the same function in a given program. Each of these patterns can be a candidate component for resource sharing. The goal is to select all patterns that sharing hardware resources for their instances can reduce the design area. Sharing hardware resources for different instances might change the throughput of final circuit, as we have seen in section 6.2. Our goal is to first select the patterns that are scheduled sequentially since that has a low effect on throughput. If we require more aggressive resource sharing, we also consider the instances of the patterns

Algorithm 3 Pattern Detection Algorithm.

```
Input: Program DFGs 𝔾={G_1, G_2,..., G_N}.
Output: Database of patterns ℙ, and their instances.
ℙ_i: Patterns of size i
𝕊_i: Subgraphs of size i
N : Number of DFGs with more than 1 node
L : Frequency limit
Traverse all DFGs, add all nodes v ∈ 𝔾 to 𝕊_1
for i← 1 to N-1 do
    for all s ∈ 𝕊_i do
        Generate t by adding a neighbour node to s
        Add t to 𝕊_{i+1}
    end for
    for all s ∈ 𝕊_{i+1} do
        freq(s) ← number of other DFGs that have instances of s
        if freq(s) ≥ L then
            add s to ℙ_{i+1}
            for all p ∈ ℙ_i do
                if p is a subgraph of s && freq(s) == freq(p) then
                    Remove p from ℙ_i
                end if
            end for
        else
            Remove s from 𝕊_{i+1}
        end if
    end for
    if (|ℙ_{i+1}| == 0) then
        return
    end if
end for
```

that have overlapping lifetimes. To select patterns for sharing, we estimate the effectiveness of resource sharing for each detected pattern.

If hardware resources are shared for instances of a pattern, those parts must be executed sequentially. Therefore, if instances of a pattern have overlapping lifetimes in the baseline design, the performance of the design degrades after resource sharing. Therefore, if it is important to meet the performance requirement, we select those patterns that are not scheduled in parallel, so that resource sharing does not affect the latency of the design. However, if timing is not a concern and it is more important to make the circuit smaller, we consider all patterns in our selection process. In the first case, for instances of each detected pattern, first we use a dependency analysis pass in LLVM to identify subsets of them that have (direct or indirect) dependency and, therefore, can not be executed at the same time. In fact, for each two instance that can be run in parallel, we keep only one of them in the set. The dependent subset of instances are grouped for further resource-utilization-estimation analysis. With this analysis pass, we ensure that resource

sharing will keep the performance overhead as low as possible. It should be noted that adding multiplexers *might* make the critical path longer and change the clock frequency, but this change is usually not considerable. For the second case, where we don't have timing limitation, we consider all instances of detected patterns for analysis.

We use a greedy algorithm for pattern selection. At each step, the best pattern is chosen based on an area gain metric. For each pattern, we estimate the effect of resource sharing on resource utilization of design. In general, sharing FPGA resources can save area. However, multiplexers are introduced in the inputs of the shared components. These multiplexers (especially larger ones) have non-negligible area. For example, a 32-bit (64-bit) 4-to-1 multiplexer on a Xilinx Virtex-7 FPGA takes 104 (200) LUTs while a 32-bit (64-bit) adder takes only 32 (64) LUTs. Therefore, the granularity, frequency, and type of shared component determines if sharing is beneficial in terms of reducing area or not. Because multiplexers are only added in the inputs of shared components, more complex and more frequent patterns are better candidates for resource sharing. When any candidate pattern P_i is selected for sharing, instead of F_{Pi} instances of the allocated hardware unit, the new design has only one instance of that hardware unit plus some extra multiplexers. For each candidate pattern P_i, we compute an *estimation* of the area gain that can be achieved due to resource sharing for the selected instances of that pattern:

$$AreaGain_{Pi} = (F_{Pi} - 1) \times Area_{Pi} - N_{inputsPi} \times Area_{mux} \tag{6.1}$$

F_{Pi} is the frequency of the candidate pattern P_i that has area $Area_{Pi}$, and $N_{inputsPi}$ is the number of inputs of the candidate pattern. $Area_{mux}$ is the area of a multiplexer of required bitwidth. From the equation, the more complex and larger pattern with more frequency has larger area gain. In this equation, area is defined as the number of hardware elements (FF, LUT, DSP48, BRAM) that the component uses. We only share the logic elements and, therefore, the number of BRAM elements does not change. So we estimate the number of reduced (or increased) FF, LUT, and DSP48 elements due to resource sharing using equation 6.1. We calculate the *areaGain* for

all patterns, and remove those that have negative gain. For the remaining set of petterns that have positive *areaGain*, in each iteration of the algorithm, the pattern with largest *areaGain* is selected.

As a very simple example, if the candidate pattern is 32-bit add operation that has 4 instances, the synthesis result of baseline program (without resource sharing) takes 4×32 = 128 LUT (four 32-bit adders), while the sharing-based approach takes 32 + 2×104 = 240 LUT (1 32-bit adder and two 32-bit 4-to-1 multiplexer). The *areaGain* for this transformation is (3×32-2×104) negative, and sharing-based approach takes larger FPGA elements. Therefore, this pattern is not selected. On the other hand, a 32-bit divider takes 290 FF and 321 LUT elements on FPGA. If the candidate pattern is a 32-bit divide operations that is repeated 4 times, the baseline synthesis result without sharing takes 1160 FF and 1284 LUT (with four 32-bit dividers), while the sharing-based approach takes 290 FF and 529 LUT (with two 32-bit 4-to-1 multiplexer and one 32-bit divider). For this transformation, *areaGain* is positive. Therefore, this pattern can be selected for resource sharing.

After selecting one pattern, we still continue to select other patterns that are good for resource sharing. First, we remove all subgraphs that have overlapping node with any node in the instances of the selected pattern. Then we continue our search for the remaining subgraphs in the pattern set. We continue this process until no more candidate pattern is left in the pattern set. At the end of this phase, we find all independent patterns and their instances that are useful for resource sharing.

6.3.0.2 Program Transformation

After selecting the patterns, the next step is to make the required changes in the code. The transformation is done through an LLVM pass that modifies the LLVM IR. We can consider two methods for defining the patterns in the code. In the first transformation method, each selected pattern is defined as a function and each instance of the pattern is replaced by a function call.

In other words, operations of a pattern are encapsulated as a single function (e.g. *patternFn*); and a pragma is added in the directive file to guide the HLS tool to share instances of function *patternFn*. But we realized that there is a special case for which this transformation results in increasing latency. If the selected pattern is inside a loop which is not perfectly pipelined due to inter-iteration loop dependency, defining the whole pattern as a function might increase the pipeline initiation interval and latency. To clarify this issue, consider the following code:

```
for ( i = 0; i < M; i++ )
    #pragma HLS PIPELINE
            r_norm = r_norm + r[i] × r[i];
```

In this code, the inter-iteration dependency comes from variable *r_norm* that must be computed in one iteration and then be used in the next iteration. Therefore, the initiation interval is equal to the delay of an *add* operation that must wait for *r_norm* to be computed from the previous iteration of the loop. If we transform the code the way we described, it must wait for the whole pattern function to get finished before starting the next iteration; which means that it must wait for one *add* and one *multiply* operations to be done. This increases the initiation interval of pipeline and therefore the overall latency of the design. Therefore, we use another method to define patterns (as shown in section 6.2, figure 6.2(a)). We define each operation inside the pattern as an individual function, and call this function instead of calling the operator in *C* for each instance of the pattern. To share resources for instances of the pattern, for each operation in the pattern, a pragma should be defined (using "**#pragma HLS allocation instances=operationFn limit=1 function**"). This guides the HLS tool to allocate only one instance of the pattern. These pragmas are written in a directive file which is given to the HLS tool as input. These pragmas guides the HLS tool where to apply resource sharing. The transformed LLVM IR is then converted back to C using a resurrected LLVM C Backend. The generated code is a low-level C code that can be synthesized using HLS tool.

6.4 Experimental Results

In this section, we show the benefit of using our workflow to efficiently improve resource usage of a design on FPGA.

6.4.1 Implementation and Experimental Setup

We use Xilinx Vivado HLS Suite 2015.4 [5] as an exemplary state-of-the-art tool for our experiments. This tool uses the LLVM compiler and compiles a behavioral C/C++ program into RTL hardware design. Before synthesizing the program, it performs special LLVM passes to optimize the IR. Our workflow takes a C program as an input and synthesizes it once using Vivado HLS tool to get the optimized LLVM IR, and to collect synthesis results of the baseline design. REHLS uses this optimized LLVM IR as input and extracts program DFGs. The pattern detection and selection passes are performed on DFGs of each function. After selecting the patterns that are suitable for resource sharing, the required transformation is performed in LLVM IR. Finally, the transformed LLVM IR is converted back to C using a resurrected LLVM C backend called llvm-cbe library. The generated code is a C program that can be synthesized using Vivado HLS tool. This program along with the generated directive file guide the HLS tool to share instances of selected patterns. Although we targeted Vivado HLS for our experiments, the proposed workflow is applicable to any LLVM-based HLS tool.

We evaluate REHLS using a set of computation kernels and applications. A description of each benchmark can be found in Table 6.2. Our benchmarks are implemented for HLS tools and mainly taken from established suites including CHStone [52], autoESL benchmark [2] and PolyBench [4], along with other independent implementations for HLS. We selected these benchmarks because they have inherent computational patterns, making them reasonable candidates for our study. For each benchmark, the number of selected patterns (*NP*) by ReHLS, the size of selected patterns (or the number of operations in the pattern) (*PS*), and the frequency of each pattern (*PF*) are reported in the table. Xilinx Virtex-7 XC7V585T FPGA device is used

Table 6.2: Benchmark Descriptions

Benchmark	Description/Source	NP	PS	PF
adi	Alternating direction implicit solver [4]	2	16,2	2,2
bnn	4-layer bitwise neural network (feedforward) for MNIST [3]	2	6,5	4,3
fdtd	Finite-difference time-domain by anisotropic perfectly matched layer [4]	2	17,14	2,2
gauss	3D gaussian convolution [2]	1	16	4
idct	Inverse discrete cosine transform from JPEG [52]	3	8,13,16	2,2,2
RN	Residual norm	1	2	2
jacobi	Jacobi iterative method [2]	1	8	4

as the target hardware platform in our experiments.

6.4.2 Results

Table 6.3 demonstrates our experimental results. For each benchmark, we report the resource utilization and throughput of using baseline and ReHLS implementations of each application. For each benchmark, the top two rows reflect synthesis results for the baseline and REHLS-optimized programs (*Baseline* and REHLS rows in the table). The third row of each benchmark (*Improvement*) shows the relative improvement of the REHLS-optimized version over the baseline program. (Improvement numbers for resource utilization and performance is the percentage of relative reduction)

6.4.2.1 Area Savings

Columns *LUT*, *FF*, and *DSP* in Table 6.3 show the resource utilization for the baseline, and REHLS-optimized programs. Our transformation does not change the number of memory elements, therefore, the number of utilized BRAM elements is not reported. Our results indicate that REHLS reduces the number of utilized resources on an average of 22% (16% LUT, 18% FF, and 33% DSP elements) comparing to the baseline design. This is achieved by the detection of

patterns and reusing the same hardware resources for them. The amount of reduction depends on different factors, including the type and number of operations in the pattern, and the percentage of resource utilization of instances of patterns comparing to the other parts in the baseline program. For example, in adi, the selected patterns form the major computations in the design, and therefore, sharing hardware resources reduces its area by 44%. On the other hand, the selected patterns in bnn are not that much complex comparing to other computations in the design, therefore, it can only achieve 1.9% area reduction. Figure 6.5 shows the relative reduction in the number of slice logic and DSP elements on the FPGA when comparing REHLS with baseline design. As can be seen our method reduces slice logics by an average of 16.8%.

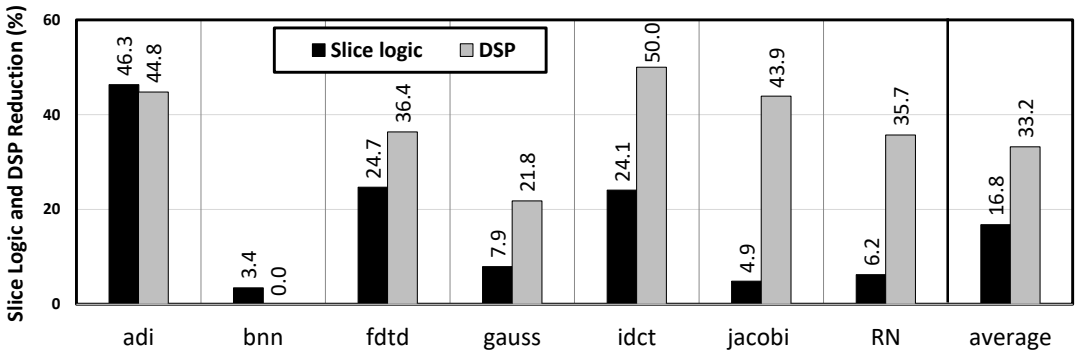


Figure 6.5: Relative reduction in the number of slice logics and DSP elements on Virtex 7. Numbers are in percentage.

6.4.2.2 Performance Overhead

Column *Performance* in Table 6.3 shows the execution time (*ms*) for the baseline and REHLS-optimized designs after synthesis. In the results that is shown in this table, we only selected patterns that sharing resources for them does not change the performance significantly. As can be seen, performance changes of REHLS comparing to baseline design are small (with geometric mean 1.2% and maximum 8.8%), because we only share resources for those instances of patterns that have some dependency. We also can ask the tool to perform more resource sharing with the cost of more performance overhead. For example, if we can accept 17% overhead

Table 6.3: Experimental results on Virtex-7 FPGA

BM	Implementation	LUT	FF	DSP	Performance(ms)
adi	Baseline	34452	33889	192	2353.222
	REHLS	19163	17528	112	2408.8
	Improvement	44.3%	48.2%	41.6%	-2.3%
bnn	Baseline	1557	679	0	13.219
	REHLS	1489	670	0	13.226
	Improvement	4.36%	1.32%	0%	-0.054%
fdtd	Baseline	62721	61560	352	25.149
	REHLS	47202	46410	224	26.86
	Improvement	24.74%	24.6%	36.36%	-6.84%
gauss	Baseline	15312	10145	661	24.63
	REHLS	15080	8361	517	24.67
	Improvement	1.5%	17.5%	21.7%	-0.17%
idct	Baseline	2421	2118	64	0.371
	REHLS	1568	1879	32	0.403
	Improvement	35.2%	11.28%	50%	-8.6%
jacobi	Baseline	14854	6794	82	4620
	REHLS	14454	6140	46	5028
	Improvement	2.69%	9.62%	43.9%	-8.8%
RN	Baseline	3494	2346	14	6.63
	REHLS	3530	1946	9	6.66
	Improvement	-1%	17%	35.7%	-0.5%

in performance for fdtd application, we can save more DSPs (45%) comparing to the baseline implementation.

6.4.2.3 Throughput Speedup

Traditional FPGA design flows usually follow a two-step approach. First, a given application is optimized for best performance and resource utilization. Then the optimized design can be replicated and executed in parallel to fully utilize the available capacity of the target FPGA, and to improve throughput [75]. As shown in Section 6.4.2.1, REHLS reduces the

area of synthesized design for our benchmarks. Therefore, the number of parallel replicas of that design, that can be fitted in the fixed area budget of the FPGA is increased. In addition, because the effect of our transformations on latency is negligible, the increase in the number of mapped applications on FPGA, results in higher throughput.

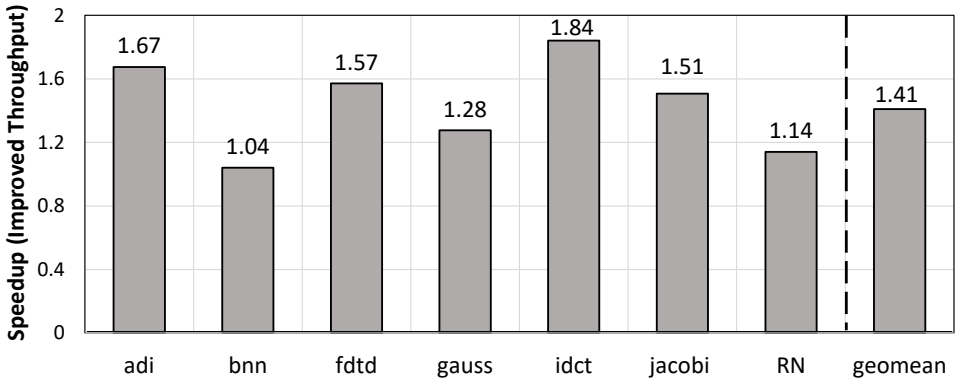


Figure 6.6: Throughput speedup of REHLS-optimized design (Normalized to baseline)

For each benchmark, we can increase the number of design replicas on FPGA until one of the resources available on FPGA reaches its maximum limit. Considering the geometric mean across all the benchmarks, REHLS improves the number of mapped kernels by a factor of 1.45× (maximum 2× in *idct*).

Figure 6.6 shows the corresponding throughput speedup results – throughput of the REHLS-optimized designs normalized to the throughput of the baseline design. As shown, REHLS achieves on average 1.41× higher throughput (between 1.04× and 1.84×) comparing to the baseline design.

6.4.2.4 Energy Overhead

For most of applications, energy only changes slightly. In fact, the power does not change that much and execution time slightly increased. Figure 6.7 shows the energy results normalized to the baseline design energy consumption. For idct, there is a slight energy overhead which results from the performance overhead of this benchmark and very low power improvement. In our benchmarks, we don't see any considerable energy overhead.

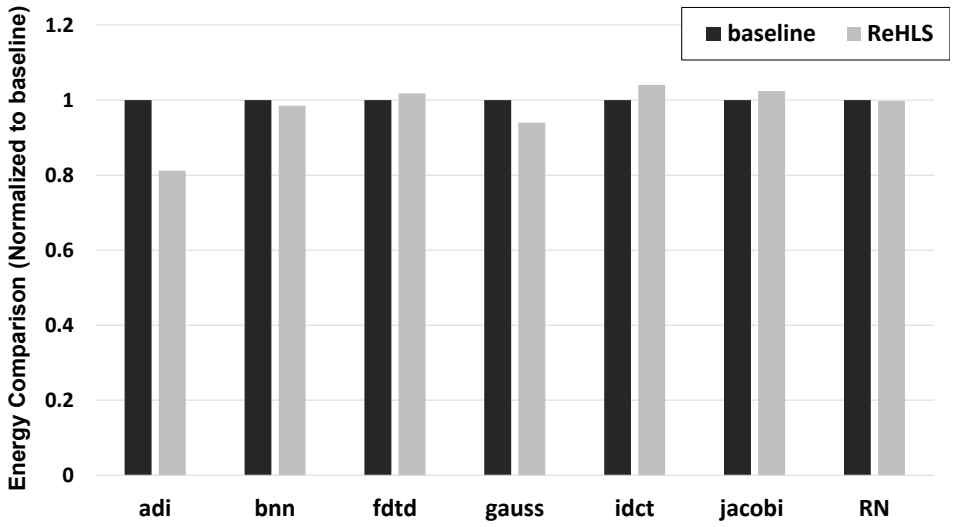


Figure 6.7: Energy comparison of REHLS-optimized design (Normalized to baseline)

6.4.3 Impact of applying both GRATER and REHLS

We also performed experiments to show the benefit of using GRATER optimization, described in Section 5, to improve the efficiency of REHLS. For this purpose, we pick a benchmark and compare the resource utilization and performance of running *baseline* kernel, kernel after applying REHLS, kernel after applying GRATER, and kernel after applying both REHLS and GRATER for FPGA. We perform the experiments that involve GRATER, for two scenarios: First, when we can not tolerate any accuracy loss; and second, when average relative error below 2% is acceptable. The input to these experiments is a C Jacobi application. We use Xilinx Vivado HLS and Virtex 7 FPGA for this experiment.

Figure 6.8 demonstrates performance of each design normalized to the *baseline* design. As shown in chapter 6, using isolation with REHLS slightly reduces the design performance, this can be seen in the leftmost column of this figure. If we use GRATER on the *Baseline* program, where no accuracy loss is allowed, we gain 1.7× performance improvement comparing to the *Baseline* design. As we apply GRATER on REHLS design, we get 1.4× performance improvement instead of performance degradation in REHLS case. This improvement become even better as we accept a slight inaccuracy in the output of application. For this experiment,

we allowed an average relative error of 2% in the output. As can be seen in column *Grater (Approximate)*, this transformation results in 2.6× performance improvement over the *baseline* design. As we apply REHLS on this approximated kernel, the performance increase by 2.13× comparing to the baseline design. Table 6.4 shows resource utilization and power consumption of each design. As expected, both REHLS and GRATER reduce the resource utilization and can even make it better as they combined together. GRATER also slightly improves power consumption of the design. These results show that by applying GRATER, we can reduce the latency overhead caused by resource sharing transformation while we are saving more resources. The improvement even become more if we can accept some degree of inaccuracy in the output of application. It should be noted that in some situations REHLS *might* reduce the chance of optimization by GRATER, as we share resources for all instances of the pattern, and only some of these instances might be amenable to bitwidth reduction. However, if quality loss is acceptable, this increases the chance of reducing resource utilization. In our Jacobi example, there is no conflict between resource sharing an bitwidth tuning optimizations.

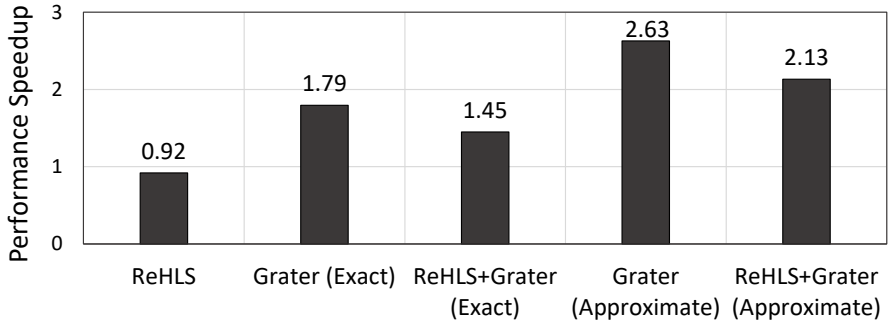


Figure 6.8: Impact of both REHLS and GRATER on Performance

6.5 Related Work

This section discusses work that is related to our resource-aware optimization workflow.

Source-to-source transformation for HLS: In recent years, some source-to-source transformation tools have been developed to perform program optimizations that are not auto-

Table 6.4: Impact of both REHLS and GRATER on resource utilization and power consumption

Implementation	LUT	FF	DSP	BRAM	Power (mW)
Baseline	14854	6794	82	67	520
ReHLS	14456	6140	46	67	522
Grater (Exact)	14281	6201	48	51	503
ReHLS+Grater (Exact)	13977	5764	30	51	503
Grater (Approximate)	13719	5836	34	38	461
ReHLS+Grater (Approximate)	13589	5580	22	38	466

matically done by the HLS tools. The purpose of these tools are to transform the program so that when it is synthesized by the HLS tool, a better design is generated. These tools mainly target loop transformation, memory partitioning, and on-chip buffer restructuring and data reuse optimizations. In this regard, polyhedral loop transformations for high-level synthesis were studied ([28] [92]) that apply affine loop transformation for improving data locality and on-chip memory allocation, and reducing the size of the data reuse buffer in the imperfectly nested loops. Many work tackle the problem of automatic array partitioning and memory banking to improve pipeline initiation interval and throughput of the design [107] [116] [80]. There are also source-to-source transformation tools that target automated expression simplification and bitwidth optimization [45].

In contrast to these works, our tool targets transformations for finding patterns of computations that can benefit from resource sharing to reduce the area of the design. In fact, our work is orthogonal to most of other source-to-source pre-HLS tools and can be added to their solutions to further improve the design.

Regularity extraction and resource sharing: Regularity extraction has been studied extensively in application-specific instruction set processors ([16] [93]) and synthesis literature ([26] [30]) over the past two decades. These works try to extract some of the computational patterns in the design, and use them for reducing resource utilization in synthesis process, or reducing area or latency in custom instruction set selection in ASIPs. Different methods for pattern

detection has been proposed. These methods include string-matching-based approaches [95], grammar induction-based approaches [88], and graph-matching-based approaches [27]. In the field of high-level synthesis, Cong et al. [27] [26] presented a method to extract patterns from behavioral specification. They use the concept of graph edit distance to enumerate similar subgraphs in the data flow graph or control data flow graph representations of the program. After discovering patterns, scheduling and resource binding algorithms of synthesis flow are changed to reduce the resource usage of the generated design. Another work [49] also studied the impact of FPGA architecture on resource sharing and changed the binding stage of HLS flow to apply resource sharing for expensive operations.

All these previous works require to change the HLS tool. However, off-the-shelf HLS tools still can not automatically exploit *non-obvious* regularities in the design; and it is not possible to change synthesis flow and algorithms in commercial HLS tools. Therefore, our method applies the required modifications on the high-level source code itself to exploit inherent regularities in the design, and guide the HLS tool to efficiently share hardware resources when useful. Our pattern detection step is more or less similar to the previous works in this area and we are not trying to improve the pattern detection algorithm. In fact, any of these pattern detection methods can be used in our workflow. We analyze the program to detect and select a set of patterns that are useful targets for resource sharing, and then perform source-to-source transformation to prepare the code for the HLS tool. Otherwise, these transformations must be done manually by the designer. Our tool reduces the design time and designer effort, and improves the synthesis results by automatic source code transformation.

6.6 Chapter Summary

Preparing the code for HLS tools with high-level transformations is not new, but it is typically a manual process. However, some of these transformations can be done efficiently, faster, and error free in an automatic way by the compiler front-end. In this chapter, we presented a

workflow for efficient task migration and resource sharing in HLS-based FPGA design. This pre-HLS workflow reduces the resource usage by identifying and exploiting inherent computational patterns in an input program. This transformation allows a new level of hardware awareness in a source-to-source compiler to improve area of the design. As a future work, we can focus on finding patterns with larger granularity as well as patterns that are functionally similar. The proposed workflow is useful for either fully-functional or degraded FPGA device.

6.7 Acknowledgements

Chapter 6, contains reprints of Atieh Lotfi, and Rajesh Gupta, "ReHLS: Resource-Aware Program Transformation Workflow for High-Level Synthesis ", *IEEE International Conference on Computer Design (ICCD)*, 2017; and Atieh Lotfi, and Rajesh Gupta. The dissertation author is the primary investigator and author of the paper.

Chapter 7

Case Study: Impact of Algorithmic Optimization on Resource Utilization

In order to make efficient use of FPGA resources, so far we have investigated source-level and compiler-level optimizations. We realized these optimizations are beneficial, fast, and efficient; but there is more optimization opportunity in algorithm level. In this chapter, we perform a case study to show the impact of choosing a hardware-friendly algorithm on resource utilization and efficiency. It is important to design the algorithm for an application considering the target hardware and its strengths and limitations. For example, as we discussed earlier, certain operations and data types such as floating point multiplication is more expensive than logical operations when implemented on FPGAs. Therefore, it is more efficient to select the algorithm that suggest simpler operations for a given application. In this chapter, for the growing field of deep neural network, we show resource saving and power reduction of using Local Binary Pattern Network *(LBPNet)* over Convolutional Neural Networks (CNN). *LBPNet* makes the algorithm more hardware-friendly by replacing the costly convolution operations by comparison. In general, designing a suitable algorithm for the target hardware, along with efficient implementation of it, is the most effective approach for efficient use of hardware resources, since this results in considering the full stack from algorithm to hardware.

7.1 Introduction

Optimizing an application in algorithmic level is of great importance for efficient use of accelerators with limited resources. If we can make our algorithms inherently more suitable for the target hardware, we can achieve better resource usage, performance, and power consumption. For example, an algorithm, that contains hundreds of floating point multiplications, is not a good match for FPGA. If we can come up with any algorithmic modification that does not require such expensive computations, we can improve resource utilization and efficiency of design significantly.

An example of expensive algorithms for hardware can be seen in deep convolutional neural networks (CNNs) [71]. CNNs have become an important class of machine learning algorithms widely used in computer vision and artificial intelligence. Modern CNNs may contain millions of floating-point parameters and intensive multiplication and accumulation (*MAC*) operations to recognize a single image. The implementation of convolution operation in these networks overburdens the resource-limited hardware accelerators [96]

There are some approaches to optimize CNN to make the hardware implementation possible. Some work selectively skip arithmetic operations that has less significant values in their operands [48]. Some other perform quantization and bitwidth optimization [94]. However, these works only reduce the model size from the network level. *Binarized neural networks* or *BNN* with binary weights and activations [31] [55] is an alternative to CNNs that reduces computations significantly at the cost of dropping accuracy. Binarization reduces storage and memory bandwidth requirements, and replaces floating point operations with binary operations which can be efficiently implemented and performed on FPGAs. There is also another interesting alternative for CNNs, recently proposed by Lin et. al., called LBPNets [76]. LBPNet made an efficient algorithmic change for deep neural networks, which eliminates the need for computing dot products and convolution operations. LBPNets replaces the expensive MAC operations with simple comparison operations. LBPNets are hardware-friendly and can achieve significant

benefits over CNN models.

In this chapter, we implement LBPNet on FPGA and show the efficiency of algorithmic optimization on resource usage.

7.2 Background

In this section, we briefly review the basic principles of CNNs and LBPNets. Since LBPNet [76] was proposed to be an alternative of the prevailing deep learning method CNN, we start from the preliminary knowledge of CNNs.

7.2.1 Convolutional Neural Networks

A CNN is a machine learning classifier that usually gets a multi-channel image and produces the probabilities of that image belonging to each output class. A CNN consists of a sequence of layers. Each layer takes as input a set of feature maps, performs some computations, and produces a new set of feature maps, which are the inputs of the next layer. The input of the first layer is the input image. During the training phase the parameters of different layers are learned on a set of pre-classified images. After training, the network can be deployed for the classification of unseen images.

Figure 7.1 shows a typical CNN model structure [77]. A CNN model usually consists of *Convolution* layer (Conv), *Fully connected* layer (FC), and *Pooling* layer (Pool). The Convolution layer convolves a K$\times$K weight kernel with the input feature map in a sliding-window manner. The Pooling layer performs a down-sampling operation, which converts the input feature map into smaller output feature map whose every pixel is the max or mean of a K$\times$K window of input pixels. The Fully connected layer performs a dot product on the input vectors of feature maps and a weight matrix. The pattern of input-output of this network is fully connected.

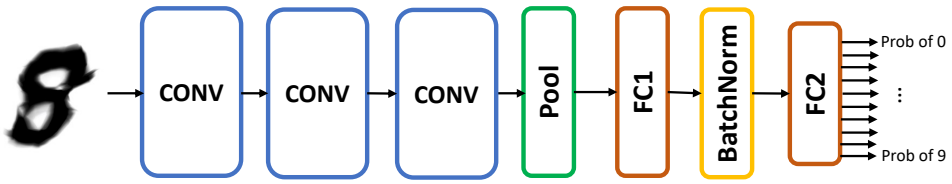


Figure 7.1: The typical convolutional neural network model structure

7.2.2 Local Binary Pattern Network

LBPNet [76] training phase learns a set of local binary patterns. These patterns indicate the location of sampling points in feature maps. Analogous to a Conv layer in CNNs, there are multiple local binary patterns, which record the sampling positions for the comparison with a pivot sampling. For each comparison pair, the pivot and a sampling point locations are used to index two values from the input features. The results of comparisons are written to predefined locations in a bit array. The interesting point in LBP layer is that it has no MAC operation or convolution operation. In addition to LBP layer, LBPNet has Pool layer and FC layers. The layers after LBP layer (or MLP classifier part) are very similar to BNNs and CNNs.

LBPNet has two major benefits over CNNs. First benefit is the convolution-free design of LBP layers. LBP Layer can be easily implemented by simple comparators. The speedup of an LBP layer over a Conv layer with massive MAC operation is therefore guaranteed. LBPNets significantly reduce the hardware resource usage and improves energy efficiency. In FPGA, it takes 62 LUTs to implement an 8-bit multiplier, and 8 LUTs for 8-bit adder, while a comparator requires only 4 LUTs. Second benefit is about the sparse sampling pattern which greatly reduces the model size. An LBP pattern contains $N_{sampling}$ sampling points' locations on a window. Assuming the number of input channel is N_{in}, and the number of output channel is N_{out}, the number of sampling locations is $2 \times N_{in} \times N_{sampling} \times N_{out}$ (it is multiplied by 2 because each location has two dimensions). However, by applying random projection only a part of sampling pairs are compared. Therefore, we only need to store $2 \times N_{sampling} \times N_{out}$ sampling positions and a mapping table of size $N_{out} \times N_{Sampling}$.

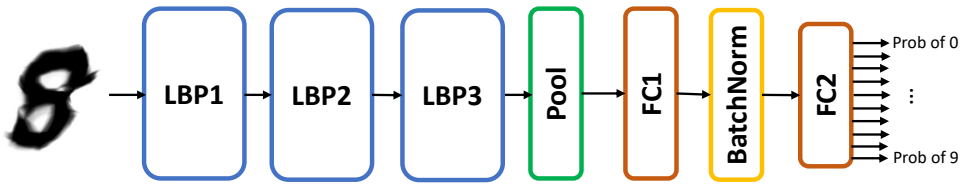


Figure 7.2: The structure of the LBPNet for MNIST.

The LBPNet structure for MNIST dataset is visualized in figure 7.2. In this structure, there are three LBP layers for extracting feature maps. Each LBP pattern contains four sampling points and four pivot points. A *Joint* operation exists at the end of each LBP layer which simply brings the input feature maps to LBP results. For the MLP classifier part, first, an average pooling layer is used. Then two *binarized* FC layers and one batch normalization layer are used to further reduce the dimension of data and extract features for the 10 classes of MNIST dataset. To avoid on-chip floating point arithmetic operations, we perform quantization and binarization. We binarize the weights of both the two fully connected layers to either -1 or 1. Then, we set all -1 to 0 for digital circuitry. The input of the first layer is the averaged value from the AvgPool Layer, which is in floating or fixed numbers. Although we cannot use an XNOR gate to replace the multiplication between the input and a weight, binarized weights enable us to use a multiplexer to select whether to add or subtract the input from an accumulator. The second binarized fully connected layer takes binarized input from the BatchNorm layer. Therefore, we can replace the multiplication with an XNOR operation in the dot-product. Furthermore, we apply the method for batch normalization layer mentioned in FINN [112]. This consists of methods to combine the binarization activation function with the linear transform and calculating a threshold for each input activation off-line.

The model sizes for the MNIST dataset is listed in table 7.1.

Table 7.1: The structure of LBPNet for MNIST.

Layer	Input ch N_{in}	output ch N_{out}	Feature map dim d	Feature map size (Kbit)	Parameter size (Kbit)
LBP1	1	39	32 x 32	-	2.18
Joint1	40	40	32 x 32	163.84	-
LBP2	40	40	32 x 32	-	2.24
Joint2	80	80	32 x 32	327.68	-
LBP3	80	80	32 x 32	-	4.48
Joint3	160	160	32 x 32	655.36	-
AvgPool	160	160	5 x 5	32.00	-
FC1	4000	512	1	4.10	2,056.19
BatchNorm	512	512	1	0.51	8.19
FC2	512	512	1	0.08	5.28
Total					
LBP				1,178.88	8.90
FC				4.69	2,069.66

7.3 FPGA Accelerator Design

7.3.1 Accelerator Architecture

An overview of the accelerator architecture is shown in Figure 7.3. The accelerator consists of four compute units as shown for each different type of layer, data and weight buffers, a memory access controller for off-chip memory transfers, and a controller. The operations of the LBP, Average Pool, Fully-Connected, and BatchNorm layers are performed through the four compute units *LBP*, *AvgPool*, *FC*, and *BatchNorm*, respectively. The LBP unit – dedicated to perform the compare operations in the LBP layer – consists of a set of logical elements, while the other compute units are made up of arithmetic units to perform operations such as addition, multiplication, and division for the other layers. The input data and the parameters of the layers loaded from the off-chip memory are stored into the on-chip *Data buffer* and *Parameters buffer*, respectively. In addition to storing the input data, the Data buffer can accommodate the intermediate results of the layers which are necessary for the computations of their next layer.

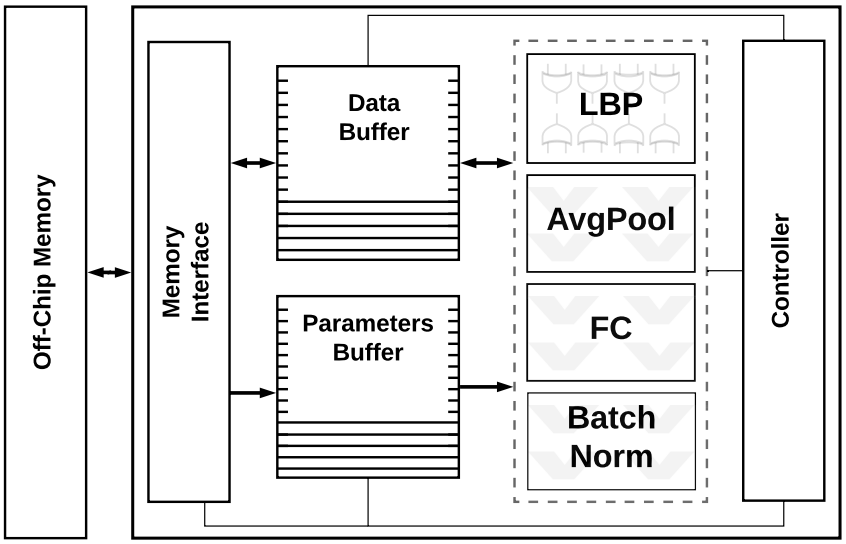


Figure 7.3: System-level architecture for LBPNet accelerator.

Because the size of the intermediate outputs of the layers is small, they can easily be stored on the available on-chip memories. This eliminates the need to transfer intermediate outputs between the accelerator and the off-chip memory. Thus, off-chip memory transfers are only needed for the input image, loading each layers weights, and sending back the final prediction output. This is one of the benefits of LBPNet compared to most other CNN-based accelerators where the size of intermediate results typically exceeds the available on-chip storage. On the other hand, the Parameter buffer can store all the weights for all the LBP layers at once. So only one time data and weight load is required for all the LBP layers and AvgPool layer to compute the input of the first FC layer. On the other hand, there is only enough space to store a portion of the FC layers weights. Each time a new set of weights are loaded into the parameter buffer and a new set of intermediate result is generated. This continues until all FC layer outputs are generated and stored in the on-chip Data buffer. To accelerate the communication and parallelize computations, we pack our 8-bit weights and generate 64-bit words, store these words in the buffers, and unpack them to perform parallel computations.

7.3.2 Execution Flow of the Accelerator

At the beginning, input image and parameters of the three LBP layers are loaded from the off-chip memory to the Data buffer and Parameters buffer. Afterwards, the LBP compute unit performs the corresponding comparison operations starting from the layer LBP1. Accordingly, the output of LBP1 is stored in Data buffer on top of the input data. The process continues until all the LBP layers are performed. Then, the AvgPool unit starts performing a quantized version of average pooling operations on the data to reduce its dimensions. At this point, the parameters stored in the Parameters buffer are not needed any longer, and the space can be freed to store the parameters of other layers.

The next pass operates on layer FC1. Since the parameters of this layer exceed the size of the Parameters buffer, a portion of the parameters are loaded into the on-chip buffer, and the corresponding multiply-and-accumulate (MAC) operations are performed, and the partial outputs are stored in the Data buffer. Then, the process moves to the computations with the next part of parameters by loading them into Parameters buffer and performing the corresponding FC computations. After all the computations of the layer FC1 are completed, the parameters of the BatchNorm layer are brought into the Parameters buffer and overwrite the parameters of FC1. Then, the compute unit BatchNorm performs the batch normalization on the results of FC1 which are available in the Data buffer. The new outputs generated by the BatchNorm unit are stored in the Data buffer. Finally, the computations of the last layer are performed similarly to executing the layer FC1. The last FC unit generates prediction output values. The final label is computed using *ArgMax* operation on the results of the last FC layer and is written back to off-chip memory.

7.3.3 Compute Units Architecture

LBP Layers: The LBP unit is the most critical component of the accelerator responsible for a number of repeated LBP layers. Each unit in the LBP layer is responsible for reading

eight input pixels from the data buffer and performing four comparison operations to generate one output. The position of these eight points are read from the weight buffer; then we can access the corresponding locations in the data buffer, compare every two of them together, and generate the corresponding output pixel by concatenating the four comparison results. As the weights are 8-bits for these layers, and they are stored in 64-bit words, we only need to read two words from the weight buffer which can be done in one cycle. These values indicate the position of points that should be accessed from a tensor in the data buffer. After reading each two-pixel values, a comparison is performed, and 1 bit of the output pixel is generated. This process is performed for every input channel in a pipeline fashion. This process is repeated in a sliding window pattern for the whole image. To improve the latency of the LBP computations, the operations inside the LBP can be parallelized. In this case, we partition the tensor input horizontally, and each processing element performs the aforementioned operations on one part. In order for the processing elements to access to data buffer at the same time, we partition the data buffer BRAM horizontally. There is clearly a trade-off between resource utilization and performance as we change the level of parallelism.

FC Layer: Each cycle we read in N data words and an equal number of weight words. N here is the input parallelization factor (we used 8 in our implementation). We apply appropriate memory partitioning to be able to access to 8 data words in one cycle. N multiplications are done in parallel, and this process is pipelined until an output is generated. As we perform quantization on FC layers, we only have integer MAC units. After the computations on the available set of weights are done, a new set of weights are loaded from the off-chip memory, and the next set of outputs are generated. Note that the level parallelism in FC layer is typically bounded by memory bandwidth of the off-chip connection, rather than the throughput of the accelerator.

BatchNorm Layer: We implement batch normalization layer using a parallel comparison between the data and weights, and multiplexers to generate a binarized output for the next fully-connect layer. In each cycle, eight parallel comparisons are made to generate eight outputs, and this process is performed in a fully pipelined fashion.

AvgPool Layer: This layer is relatively simple. It averages over 5-by-5 windows from input channels. The required memory read, computation, and memory write is fully pipelined.

7.4 Experimental Results

We designed our accelerator for MNIST dataset, which is a dataset of handwritten numbers. Our architecture can be easily adapted for other similar datasets, especially those that use one channel. For other dataset containing colored images, we can convert RGB channels to YUV channels and use one of the channels as the input image to train LBPNets.

7.4.1 Experiment Setup

The modified LBPNets are trained on a GPU machine with NVIDIA Tesla K40, and the training achieves 100.0% accuracy while the test accuracy is 99.34% on MNIST. Compared with the LBPNet paper [76], we have sacrificed some classification accuracy to make LBPNet hardware-friendly through binarizing the MLP classifier.

We have implemented our design in C++ and used Xilinx Vivado HLS and Vivado Suite 2015.4 as the primary tool for synthesizing the accelerator. We evaluate our designs on a low-cost Xilinx Zynq-7000 series (XC7Z020 FPGA) target. This FPGA contains 140 BRAM, 220 DSP, 13055 LUT, and 8148 FF. We performed HLS design space exploration to select the design options that strike a balance between resource utilization and latency. In our final design, we use 64-bit words, and the LBP compute unit consists of four parallel processing units.

7.4.2 Results

The resource utilization for our design is 7954 LUT, 7188 FF, 68 BRAM, and 16 DSP. Our FPGA implementation works at 200 MHz. We evaluate performance of our accelerator for MNIST dataset. The latency break-down for different layers and total execution time is

summarized in Table 7.2. The last column in this table (labeled as *Total Runtime*), shows the execution time (in millisecond) per image for the optimized design. This table also shows the break-down of latencies (number of cycles) per layer in columns 2-5. Column 2 and 4 shows the sum of latencies for all LBP layers and FC layers. Since different LBP and FC layers work on different data sizes, their latency is different. For the MNIST dataset, the latency of three LBP layers are 51745, 78404, 156804 cycles respectively. The latencies in the two FC layers are 259588 and 811 cycles respectively.

Table 7.2: Latency (number of clock cycles) break-down for different layers and total run time for MNIST dataset. The runtime is in millisecond.

LBP	AvgPool	FC	BatchNorm	Total Runtime
286953	72726	260399	75	3.1

We compare our design with off-the-shelf CNN and BNN FPGA implementations. Table 7.3 compares the resource utilization for different FPGA implementations of LeNet and BNN with LBPNet. LeNet, which is a CNN structure for MNIST dataset, has 2 convolutional layers, 2 max-pooling layers, and 2 fully connected multilayer perceptron layers. As shown, our accelerator achieves highest accuracy among all implementations. It utilizes only 48.6% of BRAMs, 7.3% of DSP units, 15.23% of LUTs, and 6.8% of Flip flops on our target FPGA. Comparing to CNN architectures, we mostly have better resource utilization. We also compare our throughput to other works. Throughput is shown in giga-operations-per-second (GOPS). Our accelerator achieves better throughput that CNN-based accelerators. We also have better power consumption when compared to CNN implementations. For example, [41] utilizes 3.32 W power, while our accelerator consumes only 0.5 W to perform classification. Our accelerator is more energy efficient than CNN due to replacing expensive convolution operations with simple logical operations. In general, LBPNet enables us to achieve a good balance between resource utilization and throughput, while maximizing accuracy.

Table 7.3: The comparison of resource utilization, throughput, and accuracy in different implementations of LeNet and LBPNet. Numbers for resource utilization is in percentage.

	LeNet [114]	LeNet [41]	LBPNet
DSP	3.64	43	7.27
BRAM	13.2	66	48.6
LUT	54.64	73	15.23
Flip-Flops	39.02	26	6.76
Throughput(GOPS)	12.73	-	61.62
Accuracy(%)	97.92	99.1	99.34

7.5 Related Work

LBPNet hardware implementation owes much to the groundbreaking work on LBPNet in the machine learning community [76]. This paper contain the discussion on the theory behind LBPNet and its advantages over CNN. There is a great deal of research work on hardware implementation of CNN targeting both ASIC and FPGA. The most well-known work is DianNao line of architectures [22]. Eyeriss contains a comprehensive study and use of dataflows and spatial architectures [24]. Deep Compression [51] proposes a couple of methods for optimizing CNNs by employing pruning, quantization, and customized weight encoding, which reduces the size of network. Despite these optimizations in network level, CNN is not very FPGA friendly, due to the high number of convolution operations.

For a more hardware-friendly version of CNN, other works implemented binarized neural network [112, 119]. The key optimizations for BNN include: single-bit based MAC operation, which can be replaced by efficient XNOR and popcount operations and can avoid conventional multiply and add operations; small size for both parameters and intermediate results, which would enable on-chip caching. However, the accuracy of BNN is lower than LBPNet and CNN. Also, LBPNet has a smaller model size.

7.6 Chapter Summary

Optimizing an application in algorithmic level is of great importance for efficient use of accelerators with limited resources. If we can make our algorithms inherently more suitable for the target hardware, we can achieve better resource usage, throughput, and power consumption. In this chapter, we showed this effect by implementing LBPNets which is a hardware-friendly class of deep neural networks. LBPNet makes the algorithm more hardware-friendly by replacing the costly convolution operations by comparison. The hardware-friendly optimization of LBPNet achieves Kbit model size and a high throughput while maintaining the state-of-the-art accuracy on multiple one-channel datasets. Our results confirms the usefulness of hardware-aware algorithmic modification due to its impact on resource utilization and efficiency. Algorithmic optimization of application and its hardware-friendly implementation, despite its longer development time and high programmer effort, is the most effective solution for efficient use of hardware resources, since this results in considering the full stack from algorithm to hardware.

7.7 Acknowledgement

Chapter 7, in part, contains reprints of Jen-Hau Lin, Atieh Lotfi, Vahideh Akhlaghi, Zhuowen Tu, and Rajesh Gupta, "Accelerating Local Binary Pattern Networks with Software-Programmable FPGAs", *Proc. IEEE/ACM Design, Automation, and Test in Europe (DATE)*, 2019. The dissertation author is the primary co-author of this paper.

Chapter 8

Concluding Remarks

FPGAs and GPUs are increasingly becoming popular as hardware accelerators to improve performance, cost, and energy efficiency of system. The use of these accelerators in new applications such as high-performance computing and data centers creates new challenges for them. Since they are used at large scale, the probability of fault occurrence for these systems is increased. The challenge is that there are no error mitigation methods for handling permanent faults in these accelerators. The second challenge is that it becomes more important to efficiently use hardware resources, and the penalty of inefficient use of their resources becomes more severe.

Regarding the first challenge, we present methods for enabling isolation and task migration on these accelerators. For FPGAs, isolation is enabled by guiding the synthesis tool during the placement process through a directive-based method. For GPUs, we develop an automatic software-level method to implement isolation and task migration with very low overhead in terms of performance and energy. Our method uses just-in-time compilation along with introspective kernels to enable core isolation and task migration. By isolating faulty components on these accelerators, we can make use of other available components in the hardware accelerator. Enabling error mitigation on programmable accelerators can open up opportunities to use hardware which were considered unusable due to their conditions for a longer time. This leads to reducing waste of available hardware resources, increasing the lifetime of defective hardware, as well as

reducing the manufacturing cost by accepting more range of defective hardware.

Regarding the second challenge, efficient use of hardware resources becomes more advantageous as they are being used in new systems. To improve efficiency of using GPU and FPGA accelerators, we perform optimizations in different levels of abstractions that are meaningful for each of them. These proposed methods perform optimization through either source-level optimization, compiler-assisted program modification, hardware modification, or algorithmic modification to remove unnecessary redundancies from program or hardware. For GPUs, we seek opportunities to perform optimization in hardware-level and source-level where the program running on GPU is optimized. For FPGAs, we target optimizations in compiler-level, source-level, and algorithm-level.

The approach for resource optimization in GPU hardware-level tries to find the opportunities for removing unnecessary reliability-related redundancies in the design of hardware accelerators or replacing them with less expensive mechanisms. The source-level optimizer, that is useful for both GPU and FPGA, reduces unnecessary computations and automatically simplifies expensive operations in a program or design specification that is developed for an accelerator. This results in improving performance and reducing hardware resource requirements. For improving the efficiency of HLS-based FPGA design, another approach performs automated transformation in the high-level design specification. The optimization is done through compiler front-end and enables task migration and resource sharing without any modification in the HLS tool. This transformation results in more efficient use of FPGA resources. We also present a case study that performs resource optimization by algorithmic modification, and shows the benefit of using a hardware-friendly algorithm on resource utilization and power consumption of FPGA design.

We can compare these methods based on their efficiency and impact on resource usage of hardware, required labor time for optimization, and scope (number of applications they can impact on). We realize that the algorithmic optimization has a high impact on resource utilization, but it requires the longest development time, and it is not extendable to other applications. On the

other hand, our hardware-level optimization has low impact on resource optimization, requires high labor time, and impacts all the applications that are run on that optimized hardware. The source-level optimization can have moderate impact on resource utilization, it is fast, and works for all applications that are not manually optimized. Finally, the compiler-level pre-HLS optimization method have low to moderate impact on resource utilization, it is fast, and works for only applications with regularities that has the potential for resource sharing.

Table 8.1: Impact on Resource saving Vs. Labor time Vs. Scope for different proposed methods targeting hardware resource optimization

Method	Impact on resource saving	Labor/Optimization time	Scope
Algorithmic modification (Chap. 7)	High	High	Single app
Source-level optimization (Chap. 5)	Moderate	Low	Most apps
Compiler-level optimization (Chap. 6)	Low to Moderate	Low	Apps with regularity
Removing hardware redundancy (Chap. 4)	Low	High	All apps

In conclusion, this dissertation exhibits that the use of automated source-level and compiler-level optimizations alongside the isolation and task migration technique is a *general* and *efficient* approach that can be *easily* applied to fault-susceptible commodity programmable hardware accelerators and improve their efficiency in terms of both cost per performance and energy.

Looking beyond this dissertation, we propose the use of learning approaches to guide compiler for GPU and FPGA program optimization. For example, reinforcement learning can be applied on GRATER for exploring search space of program optimization. By using reinforcement learning, we can learn a model to predict suitable optimizations for a given program that is run on a programmable accelerator. The design space of program optimizations for hardware accelerators are large, therefore, a good model is required for selecting suitable optimizations. This helps a better utilization of hardware resources by applying multiple optimization strategies.

Bibliography

[1] Altera sdk for opencl. http://www.altera.com/products/software/opencl/opencl-index.html.

[2] AutoESL coprocessors benchmark. https://cadlab.cs.ucla.edu/repos/coprocessors/. Accessed: 2017-01-30.

[3] bitwise neural network. https://github.com/hihihippp/bitwise-nn-fpga. Accessed: 2017-01-30.

[4] The polyhedral benchmark suite, version 4.2 (polybench4.2). http://polybench.sourceforge.net.

[5] Vivado high-level synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[6] SDAccel. http://www.xilinx.com/products/design-tools/sdx/sdaccel.html, 2015.

[7] Nvidia tesla p100. June 2017.

[8] Nvidia tesla v100 gpu architecture. June 2017.

[9] F. Ahmed, M.M. Sabry, D. Atienza, and L. Milor. Wearout-aware compiler-directed register assignment for embedded systems. In *Quality Electronic Design (ISQED), 2012 13th International Symposium on*, pages 33–40, March 2012.

[10] Amd app sdk v2.9, http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/, 2013.

[11] R. Baumann. The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction. In *Digest. International Electron Devices Meeting,*, pages 329–332, Dec 2002.

[12] L. Bautista Gomez, F. Cappello, L. Carro, N. Debardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, and M. Sonza Reorda. Gpgpus: How to combine high computational power with high reliability. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–9, March 2014.

[13] K. Bernstein, D.J. Frank, A.E. Gattiker, W. Haensch, B.L. Ji, S.R. Nassif, E.J. Nowak, D.J. Pearson, and N.J. Rohrer. High-performance cmos variability in the 65-nm regime and beyond. *IBM Journal of Research and Development*, 50(4.5):433–449, July 2006.

[14] S. Bhardwaj, W. Wang, R. Vattikonda, Y. Cao, and S. Vrudhula. Predictive modeling of the nbti effect for reliable design. In *Custom Integrated Circuits Conference, 2006. CICC '06. IEEE*, pages 189–192, Sept 2006.

[15] F. Bodin and A. Seznec. Skewed associativity improves program performance and enhances predictability. *IEEE Transactions on Computers*, 46(5):530–544, May 1997.

[16] Philip Brisk et al. Instruction generation and regularity extraction for reconfigurable processors. In *CASES '02*, 2002.

[17] S. Chakravarthi, A.T. Krishnan, V. Reddy, C.F. Machala, and S. Krishnan. A comprehensive framework for predictive modeling of negative bias temperature instability. In *Reliability Physics Symposium Proceedings, 2004. 42nd Annual. 2004 IEEE International*, pages 273–282, April 2004.

[18] T.B Chan, J. Sartori, P. Gupta, and R. Kumar. On the efficacy of nbti mitigation techniques. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

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

[20] G. Chen, K.Y. Chuah, M.-F. Li, D.S.H. Chan, C.H. Ang, J.Z. Zheng, Y. Jin, and D.L. Kwong. Dynamic nbti of pmos transistors and its impact on device lifetime. In *Reliability Physics Symposium Proceedings, 2003. 41st Annual. 2003 IEEE International*, pages 196–202, March 2003.

[21] G. Chen, M.-F. Li, C.H. Ang, J.Z. Zheng, and D.-L. Kwong. Dynamic nbti of p-mos transistors and its impact on mosfet scaling. *Electron Device Letters, IEEE*, 23(12):734–736, Dec 2002.

[22] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[23] X. Chen, Y. Wang, Y. Liang, Y. Xie, and H. Yang. Run-time technique for simultaneous aging and power optimization in gpgpus. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 168:1–168:6. ACM, 2014.

[24] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, Jan 2017.

[25] Amit Chowdhary et al. A general approach for regularity extraction in datapath circuits. ICCAD '98, 1998.

[26] Jason Cong, Hui Huang, and Wei Jiang. A generalized control-flow-aware pattern recognition algorithm for behavioral synthesis. In *DATE '10*, 2010.

[27] Jason Cong and Wei Jiang. Pattern-based behavior synthesis for fpga resource reduction. In *FPGA '08*, 2008.

[28] Jason Cong, Peng Zhang, and Yi Zou. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis. DAC '12, 2012.

[29] Jason Cong et al. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, April 2011.

[30] Jason Cong et al. Pattern-mining for behavioral synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30, June 2011.

[31] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. pages 3123–3131, 2015.

[32] G. Deest, T. Yuki, O. Sentieys, and S. Derrien. Toward scalable source level accuracy analysis for floating-point to fixed-point conversion. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 726–733, Nov 2014.

[33] D. Defour and E. Petit. Gpuburn: A system to test and mitigate gpu hardware failures. In *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 263–270, July 2013.

[34] Timothy J. Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory by. 1997.

[35] Jeffrey R. Diamond, Donald S. Fussell, and Stephen W. Keckler. Arbitrary modulus indexing. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 140–152, Washington, DC, USA, 2014. IEEE Computer Society.

[36] J. B. Dugan and M. R. Lyu. System reliability analysis of an n-version programming application. *IEEE Transactions on Reliability*, 43(4):513–519, Dec 1994.

[37] S. Dutt, V. Verma, and V. Suthar. Built-in-self-test of fpgas with provable diagnosabilities and high diagnostic coverage with application to online testing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(2):309–326, Feb 2008.

[38] W. Dweik, M. Abdel-Majeed, and M. Annavaram. Warped-shield: Tolerating hard faults in gpgpus. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 431–442, June 2014.

[39] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, 2nd edition, 2007.

[40] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[41] Gan Feng, Zuyi Hu, Song Chen, and Feng Wu. Energy-efficient and high-throughput fpga-based accelerator for convolutional neural networks. In *ICSICT*, pages 624–626, 2016.

[42] F. Firouzi, S. Kiamehr, and M.B. Tahoori. Nbti mitigation by optimized nop assignment and insertion. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 218–223, March 2012.

[43] A.A. Gaffar, J.A. Clarke, and G.A. Constantinides. Powerbit - power aware arithmetic bit-width optimization. In *FPT 2006*, pages 289–292, Dec 2006.

[44] Xitong Gao and George A. Constantinides. Numerical program optimization for high-level synthesis. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 210–213, New York, NY, USA, 2015. ACM.

[45] Xitong Gao et al. Automatically optimizing the latency, area, and accuracy of c programs for high-level synthesis. FPGA '16. ACM, 2016.

[46] Antonio González, Mateo Valero, Nigel Topham, and Joan M. Parcerisa. Eliminating cache conflict misses through xor-based placement functions. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 76–83, New York, NY, USA, 1997. ACM.

[47] Nathan Goulding, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Jonathan Babb, Michael Taylor, and Steven Swanson. GreenDroid: A Mobile Application Processor for a Future of Dark Silicon. In *HOTCHIPS*, 2010.

[48] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *NIPS*, 2016.

[49] Stefan Hadjis, Andrew Canis, Jason H. Anderson, Jongsok Choi, Kevin Nam, Stephen Brown, and Tomasz Czajkowski. Impact of fpga architecture on resource sharing in high-level synthesis. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, pages 111–114, New York, NY, USA, 2012. ACM.

[50] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, April 1950.

[51] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *ICLR*, 2015.

[52] Yuko Hara et al. Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[53] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10, May 2013.

[54] M. Y. Hsiao. A class of optimal minimum odd-weight-column sec-ded codes. *IBM Journal of Research and Development*, 14(4):395–401, July 1970.

[55] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *NIPS*, pages 4107–4115, 2016.

[56] K. Shankar A. Kodi A. Louri J. Roveda J. Sun, R. Lysecky. Workload assignment considering nbti degradationin multicore systems. *ACM Journal on Emerging Technologies in Computing Systems*, 10(1):1–22, Jan. 2014.

[57] A.B. Kahng and Seokhyeong Kang. Accuracy-configurable adder for approximate arithmetic designs. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 820–825, June 2012.

[58] Charu Kalra, Daniel Lowell, John Kalamatianos, Vilas Sridharan, and David Kaeli. Performance evaluation of compiler-based software rmt in an hsa environment. In *SELSE*, 03 2016.

[59] U.R. Karpuzcu, B. Greskamp, and J. Torrellas. The bubblewrap many-core: Popping cores for sequential acceleration. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 447–458, Dec 2009.

[60] Mahmoud Khairy, Mohamed Zahran, and Amr G. Wassal. Efficient utilization of gpgpu cache hierarchy. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, pages 36–47, New York, NY, USA, 2015. ACM.

[61] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *Software, IEE Proceedings-*, pages 288–299, Feb 2004.

[62] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. Moonwalk: Nre optimization in asic clouds. *SIGPLAN Not.*, 52(4):511–526, April 2017.

[63] Kyu Yeun Kim and Woongki Baek. Quantifying the performance and energy efficiency of advanced cache indexing for gpgpu computing. *Microprocess. Microsyst.*, 43(C):81–94, June 2016.

[64] S. Kim. Reducing area overhead for error-protecting large l2/l3 caches. *IEEE Transactions on Computers*, 58(3):300–310, March 2009.

[65] Seongwoo Kim and A K Somani. Area efficient architectures for information integrity in cache memories. In *Proceedings of the 26th International Symposium on Computer Architecture (Cat. No.99CB36367)*, pages 246–255, 1999.

[66] Soontae a Kim. Area-efficient error protection for caches. In *Proceedings of the Design Automation Test in Europe Conference*, DATE '06, pages 1–6. IEEE, 2006.

[67] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *VLSI Design (VLSI Design), 2011 24th International Conference on*, pages 346–351, Jan 2011.

[68] S.V. Kumar, C.H. Kim, and S.S. Sapatnekar. An analytical model for negative bias temperature instability. In *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 493–496, Nov 2006.

[69] T. Kutzschebauch. Efficient logic optimization using regularity extraction. In *Computer Design, 2000. Proceedings. 2000 International Conference on*, pages 487–493, 2000.

[70] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, 2004.

[71] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4), 1989.

[72] C. Lee, H. Kim, H. Park, S. Kim, H. Oh, and S. Ha. A task remapping technique for reliable multi-core embedded systems. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 307–316, Oct 2010.

[73] H. Lee, M. Shafique, and M. A. Al Faruque. Aging-aware workload management on embedded gpu under process variation. *IEEE Transactions on Computers*, 67(7):920–933, July 2018.

[74] Haeseung Lee, Muhammad Shafique, and Mohammad Abdullah Al Faruque. Low-overhead aging-aware resource management on embedded gpus. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 67:1–67:6, New York, NY, USA, 2017. ACM.

[75] Peng Li, Peng Zhang, Louis-Noel Pouchet, and Jason Cong. Resource-aware throughput optimization for high-level synthesis. FPGA '15, 2015.

[76] Jeng-Hau Lin, Yunnan Yang, Rajesh Gupta, and Zhuowen Tu. Local binary pattern networks. *arXiv preprint arXiv:1803.07125*, 2018.

[77] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[78] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, April 1962.

[79] Michael C. McFarland, Alice C. Parker, and Raul Camposano. Tutorial on high-level synthesis. In *Proceedings of the 25th ACM/IEEE Design Automation Conference*, DAC '88, pages 330–336, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[80] Chenyue Meng et al. Efficient memory partitioning for parallel data access in multidimensional arrays. DAC '15. ACM, 2015.

[81] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 309–328, New York, NY, USA, 2014. ACM.

[82] H. Modi and P. Athanas. In-system testing of xilinx 7-series fpgas: Part 1-logic. In *MILCOM 2015 - 2015 IEEE Military Communications Conference*, pages 477–482, Oct 2015.

[83] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. Snnap: Approximate computing on programmable socs via neural acceleration. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 603–614, Feb 2015.

[84] K. Nepal, Yueting Li, R.I. Bahar, and S. Reda. Abacus: A technique for automated behavioral synthesis of approximate computing circuits. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6, March 2014.

[85] F. Oboril and M.B. Tahoori. Extratime: Modeling and analysis of wearout due to transistor aging at microarchitecture-level. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12, June 2012.

[86] S. Ogawa and N. Shiono. Generalized diffusion-reaction model for the low-field charge-buildup instability at the si-sio2 interface. *Physical Review*, 51(7):4218–4230, Feb 1995.

[87] Opencl programming guide for the cuda architecture., 2009.

[88] Muhsen Owaida et al. A grammar induction method for clustering of operations in complex fpga designs. In *FCCM '14*, 2014.

[89] Ayan Palchaudhuri and Anindya Sundar Dhar. Built-in fault localization circuitry for high performance fpga based implementations. *J. Electron. Test.*, 33(4):529–537, August 2017.

[90] F. Paterna, A. Acquaviva, and L. Benini. Aging-aware energy-efficient workload allocation for mobile multimedia platforms. *IEEE Transactions on Parallel and Distributed Systems*, 24(8):1489–1499, Aug 2013.

[91] F. Paterna, L. Benini, A. Acquaviva, F. Papariello, A. Acquaviva, and M. Olivieri. Adaptive idleness distribution for non-uniform aging tolerance in multiprocessor systems-on-chip. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 906–909, April 2009.

[92] Louis-Noel Pouchet et al. Polyhedral-based data reuse optimization for configurable computing. FPGA '13, 2013.

[93] Alok Prakash et al. Fpga-aware techniques for rapid generation of profitable custom instructions. *Microprocessors and Microsystems*, 37(3):259 – 269, 2013.

[94] Jiantao Qiu et al. Going deeper with embedded fpga platform for convolutional neural network. In *FPGA*, pages 26–35. ACM, 2016.

[95] D. S. Rao and F. J. Kurdahi. On clustering for maximal regularity extraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(8):1198–1208, Aug 1993.

[96] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *ECCV*, 2016.

[97] B. Ramakrishna Rau. Pseudo-randomly interleaved memory. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 74–83, New York, NY, USA, 1991. ACM.

[98] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, March 2005.

[99] Alberto Ros, Polychronis Xekalakis, Marcelo Cintra, Manuel E. Acacio, and José M. García. Ascib: Adaptive selection of cache indexing bits for removing conflict misses. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pages 51–56, New York, NY, USA, 2012. ACM.

[100] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. Asac: Automatic sensitivity analysis for approximate computing. In *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '14, pages 95–104, New York, NY, USA, 2014. ACM.

[101] C. Rubio-Gonzlez, Cuong Nguyen, Hong Diep Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2013.

[102] N. R. Saxena, C. W. D. Chang, K. Dawallu, J. Kohli, and P. Helland. Fault-tolerant features in the hal memory management unit. *IEEE Transactions on Computers*, 44(2):170–180, Feb 1995.

[103] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 53–64, New York, NY, USA, 2014. ACM.

[104] Bianca Schroeder and Garth A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[105] P. Singh, E. Karl, D. Sylvester, and D. Blaauw. Dynamic nbti management using a 45 nm multi-degradation sensor. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 58(9):2026–2037, Sept 2011.

[106] H.S. Stone. *Discrete mathematical structures and their applications*. SRA computer science series. Science Research Associates, 1973.

[107] Jincheng Su et al. Efficient memory partitioning for parallel data access via data reuse. FPGA '16, 2016.

[108] J. Sun, R. Lysecky, K. Shankar, A. Kodi, A. Louri, and J.M. Wang. Workload capacity considering nbti degradation in multi-core systems. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 450–455, Jan 2010.

[109] A. Tiwari and J. Torrellas. Facelift: Hiding and slowing down aging in multicores. In *Microarchitecture, 2008. MICRO-41. 2008 41st IEEE/ACM International Symposium on*, pages 129–140, Nov 2008.

[110] N. Topham and A. Gonzalez. Randomized cache placement for eliminating conflicts. *IEEE Transactions on Computers*, 48(2):185–192, Feb 1999.

[111] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing . In *Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2012.

[112] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *FPGA*, pages 65–74. ACM, 2017.

[113] H. Vandierendonck and K. De Bosschere. Xor-based hash functions. *IEEE Transactions on Computers*, 54(7):800–812, July 2005.

[114] S. I. Venieris and C. S. Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *FCCM*, 2016.

[115] W. Wang, S. Yang, S. Bhardwaj, S. Vrudhula, F. Liu, and Y. Cao. The impact of nbti effect on combinational circuit: Modeling, simulation, and analysis. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 18(2):173–183, Feb 2010.

[116] Yuxin Wang et al. Memory partitioning for multidimensional arrays in high-level synthesis. DAC '13. ACM, 2013.

[117] Ping Xiang, Yi Yang, Mike Mantor, Norm Rubin, Lisa R. Hsu, and Huiyang Zhou. Exploiting uniform vector instructions for gpgpu performance, energy efficiency, and opportunistic reliability enhancement. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 433–442, New York, NY, USA, 2013. ACM.

[118] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pages 32–41, 2000.

[119] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *FPGA*. ACM, 2017.