ENERGY-EFFICIENT ACCELERATOR DESIGN FOR EMERGING APPLICATIONS

A Dissertation

by

KYUNG HOON KIM

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Eun Jung Kim
Committee Members,	Daniel A. Jiménez
	Dilma Da Silva
	Paul Gratz
Head of Department,	Scott Schaefer

May 2021

Major Subject: Computer Engineering

Copyright 2021 Kyung Hoon Kim

ABSTRACT

Today, hardware accelerators are widely accepted as a cost-effective solution for emerging applications in computing platforms from servers to mobile devices. Servers often leverage manycore accelerators such as Graphics Processing Units (GPUs) to achieve high performance gain by exploiting simple yet energy-efficient compute cores. The tremendous computing power of GPUs shows great potential to keep up with the emerging applications that demand heavy computation on a large volume of data. However, scaling up single-chip GPUs is challenging due to strict chip power constraints. The data movement overhead over the Network-on-Chip (NoC) becomes a key performance bottleneck in large-scale GPUs that degrades both overall performance and energy efficiency. Mobile devices are inherently even more restricted by energy constraints than servers so that they often leverage low-power accelerators for particular functionalities including inference in Deep Neural Networks (DNNs). However, the emerging applications that typically rely on DNNs require considerable computation due to complex algorithmic operations, which becomes a key energy bottleneck.

To tackle the performance and energy bottlenecks fundamentally, we propose three approaches that focus on minimizing unnecessary data movement and computation. First, we propose a packet coalescing mechanism to coalesce redundant packets over the NoC of GPUs and transfer the coalesced packet in a multicast. Second, we present a packet compression mechanism to directly reduce the packet size based on a dual-pattern compression technique with data preprocessing capability. Third, we propose an optimization methodology for a convolutional neural network (CNN) that uses an early prediction and reduces the complexity of compute kernels in CNNs by guiding them to compute critical features only. In our analysis, the packet coalescing and packet compression approaches show 15% and 33% IPC improvements in a large-scale GPU on average across various modern applications. Besides, the network optimization methodology reduces the inference energy cost of CNNs by 77% on average with an ignorable accuracy drop in a time-series classification problem.

DEDICATION

To my parents and my sister.

ACKNOWLEDGMENTS

I would like to thank the Lord with all my heart for always being with me. It was tough to decide to start my PhD at Texas A&M University. It was a long journey, but I indeed confess it was an unforgettable blessing for me to experience your meticulous love and care deeply every moment. Without your shepherding and words, I could not complete this journey.

I would like to thank my parents and sister for their love, prayers, and encouragement. Also, I am grateful to my friends who have been always there for me at Stony Brook University, Pastor Tae Jun Suk, Young Jun Lee, Dr. Chul Sung, Dr. Yeona Kang, Dr. Nahyun Cho and Nanume church members who have prayed for me.

I would like to thank Dr. Eun Jung Kim for her guidance on research and Dr. Ki Hwan Yum for his review on the papers. I would like to thank my committee members, Dr. Daniel A. Jiménez, Dr. Dilma Da Silva, and Dr. Paul Gratz for their feedbacks and support. I am especially grateful to MS students, Priyank Devpura, Abhishek Nayyar, Andrew Doolittle, and Swathi Changalarayappa for their help with research projects. Also, I thank all lab members who have been with me during my PhD course.

I would like to thank the Department of Computer Science and Engineering and the chair, Dr. Dilma Da Silva for seamless financial support. I also thank Dr. Tanzir Ahmed for giving me a rewarding teaching experience. I would like to thank the staffs in Advanced Micro Devices (AMD), Michael Mantor, Xiaoxiao Liu, Nuwan Jayasena, Shaizeen Aga, and Jonathan Alsop for having me as a part of their fascinating research projects and giving me intellectual mentoring and feedbacks.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Eun Jung Kim (advisor) and Professors Dilma Da Silva and Daniel A. Jiménez of the Department of Computer Science and Engineering and Professor Paul Gratz of the Department of Computer and Electrical Engineering. All other work conducted for the dissertation was completed by the student independently.

Funding Sources

This research was funded in part by a teaching assistantship from the Department of Computer Science and Engineering and NSF award CCF-1423433.

NOMENCLATURE

GPU	Graphics Processing Unit
SM	Streaming Multiprocessor
MC	Memory Controller
NoC	Network-on-Chip
IPC	Instruction Per Cycle
AMAT	Average Memory Access Time
DPC	Dual Pattern Compression
DA-DPC	Data-type Aware-DPC
IoT	Internet of Things
HAR	Human Activity Recognition
DNN	Deep Neural Network
CNN	Convolutional Neural Network
GBT	Gradient Boosting Tree
FSV	Feature Segment Vector
EPV	Early-prediction Parameter Vector
GA	Genetic Algorithm

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

CONTRIBUTORS AND FUNDING SOURCES v

NOMENCLATURE vi

TABLE OF CONTENTS vii

LIST OF FIGURES x

LIST OF TABLES xii

1. INTRODUCTION 1

2. PACKET COALESCING EXPLOITING DATA REDUNDANCY 4

2.1 Introduction 4
2.2 Motivation 6
2.2.1 MC Bottleneck 6
2.2.2 Data Redundancy 8
2.3 Packet Coalescing Unit 10
2.3.1 Overview 10
2.3.2 Coalescing Mechanism 11
2.4 Multicast Support in NoC 13
2.4.1 Overview 13
2.4.2 Multicast in Crossbar 14
2.4.3 Multicast in 2D Mesh 14
2.5 Evaluation 16
2.5.1 Methodology 16
2.5.2 IPC Improvement Analysis 17
2.5.3 Packet Coalescing Analysis 19
2.5.4 Sensitivity Analysis 23
2.5.5 Coalescing in the Global Crossbar 24
2.5.6 Hardware Cost 25
2.6 Related Work 26

2.7 Conclusions . . . 27

3. DUAL PATTERN COMPRESSION USING DATA-PREPROCESSING . . . 28

3.1 Introduction . . . 28
3.2 Background and Approach . . . 30
3.2.1 Hardware-Based Compression . . . 30
3.2.2 Compression Algorithms and Motivation . . . 31
3.2.3 Approach . . . 32
3.3 Compression Mechanisms . . . 33
3.3.1 Preprocessing Using Natural Data Redundancy . . . 33
3.3.2 Compression/Decompression Algorithms . . . 36
3.4 Data Operation Mechanisms . . . 36
3.4.1 Floating-Point Data . . . 36
3.4.2 Character Data . . . 40
3.4.3 Integer Data . . . 42
3.4.4 Data Selector . . . 42
3.4.5 Hardware Design and Cost . . . 43
3.5 Evaluation Methodology . . . 44
3.5.1 Network Bottleneck in a GPU . . . 44
3.5.2 GPU with Packet Compressors . . . 46
3.5.3 Methodology . . . 47
3.6 Evaluation . . . 48
3.6.1 Effect on IPC Performance . . . 48
3.6.2 Compression Performance Analysis . . . 49
3.6.3 Impact of Preprocessing on Compression . . . 50
3.6.4 Energy Analysis . . . 52
3.6.5 Packet Compression under Crossbar . . . 53
3.7 Conclusions . . . 53

4. ENERGY-EFFICIENT CNN INFERENCE EXPLOITING FEATURE CRITICALITY . . . 55

4.1 Introduction . . . 55
4.2 Background and Motivation . . . 57
4.2.1 System Overview . . . 57
4.2.2 Motivation . . . 57
4.3 Early Prediction with Critical Features . . . 60
4.3.1 Feature Subspacing . . . 60
4.3.2 Layerwise Early Prediction . . . 62
4.3.3 Hyperparameter Optimization . . . 63
4.4 Hardware Implementation . . . 65
4.4.1 Functional Primitives . . . 65
4.5 Evaluation . . . 66
4.5.1 Benchmark and Model Selection . . . 66
4.5.2 Evaluation Result . . . 67
4.6 Related Work . . . 70

4.7 Conclusions 70

5. CONCLUSIONS 72

REFERENCES 74

LIST OF FIGURES

FIGURE Page

2.1 Data Redundancy across 29 Benchmarks in 2DMesh 6

2.2 MC Bottlenecks in GPGPUs across 29 Benchmarks (T1: 2DMesh, T2: Crossbar) ... 8

2.3 GPGPU Architecture Incorporating Proposed PCUs 10

2.4 Multicast Router Architecture 15

2.5 System and Coalescing Performance 18

2.6 Comparison of Normalized IPCs 19

2.7 Injected Packets, MC Stall Time and L1 Cache Miss Penalty Reduction and Network Bandwidth Savings 20

2.8 Memory Region with Inter-core Locality 22

2.9 Normalized IPC (Bar) and Inter-core Locality Ratio (Line) as varying the number of RGRs 24

2.10 Normalized IPC (Bar) and Inter-core Locality Ratio (Line) when a minimum L2 hit latency is 120 and 2 cycles 25

3.1 Compression Flow with Data Preprocessing and Encoding 33

3.2 Illustration of Data Redundancy Remaining in Compressed Data by BDI and FPC .. 35

3.3 Example of Data Remap Function and Compression 37

3.4 Data Operation for Short Floating-Point Data (e.g. 12.4) 39

3.5 Example of Data Preprocessing for Character Data 41

3.6 Compression Pipeline for a 128B Cache Block 42

3.7 Analysis of Network Bottleneck in MCs using MC Stall Time Ratio 45

3.8 Large-Scale GPU Architecture with Compressor Integrated in NI of Reply Network 46

3.9 Comparison of IPC Performance and Compressibility 47

3.10 Effects of Floating-Point Data Operation in R.KM With 7 Real-World Text Datasets and 6 Synthetic Datasets 51

3.11 Effects of Character Data Operation in M.SM With 12 Real-World Datasets 51

3.12 Energy Savings in NoC (2DMesh) by DPC and DA-DPC (A: Baseline, B: DPC, C: DA-DPC) 52

3.13 (a) Analysis of Network Bottleneck and (b) Normalized IPC in NVIDIA Fermi using Crossbar and either DPC or DA-DPC 52

4.1 Illustration of a 1D-convolutional network coupled with GBTs exploiting critical features for early predictions 58

4.2 Accuracy and the ratio of used features in a featuremap according to tree numbers .. 59

4.3 Distribution of features according to their criticality in ADL-S 59

4.4 Illustration of a feature computation example under the data dependency between $conv1$ and $pool1$ layers. 61

4.5 Illustration of creating a feature segment ($S_2^l, l = 1..6$) in a backward manner from $pool3$ to $conv1$ 62

4.6 Energy savings (a) and accuracy (b) of the early-prediction network against the baseline. 68

4.7 Energy savings according to layer of early prediction 68

4.8 Heatmap of layerwise feature criticality in ADL-S 69

LIST OF TABLES

TABLE Page

2.1 System Configuration Parameters 17

2.2 RGR Overhead 25

3.1 Qualitative Comparisons of Compressors (CC, MLC, PC: Cache, Memory-Link, Packet Compression) 32

3.2 Summary of Natural Data Redundancy 34

3.3 GPU Configuration Parameters 45

4.1 Parameters for early-prediction network 67

1. INTRODUCTION

Modern computing systems have been continuously innovated with new computing paradigms. In the computing history, we witnessed exponentially increasing performance in succeeding generations of chips for many decades, which was gained due to reduction in transistor size that allows faster operation. With the end of Dennard Scaling, increasing frequency merely was not effective any more due to excessive energy cost. Homogenous multi-core processors have been utilized as an alternative paradigm for decades-until now, but they rarely keep up with recent emerging applications requiring massive processing power. Today, heterogeneous multi-core architecture that exploits accelerators specialized for complex computation such as GPUs [1] and customized hardwares (e.g. FPGA or ASIC) [2] is widely accepted as a cost-effective way to improve overall performance of computing system. Accelerators are becoming more prevalent in a diverse range of systems from servers to mobile devices.

Servers often leverage many-core accelerators such as GPUs to gain high throughput performance for complex compute kernels by exploiting simple yet energy-efficient compute cores and high-bandwidth memory systems. The emerging applications require even more throughput performance to process a large volume of data. Scaling up of single chip accelerators is challenging due to limited power budget. Power supply voltage scaling is not effective due to leakage power constraints. The scaling of transistors with reduced process size gives only limited improvement in power efficiency [3]. Consequently, it is hard to scale up accelerators without breaking the power budget, although the advanced process technology can place more components on a chip. Therefore, energy-efficient architecture is fundamentally essential.

On the other hand, mobile devices utilize low-energy accelerators specialized for important functionalities including inference in DNNs. The accelerators are inherently constrained by strict energy budget under battery-powered mobile devices. Furthermore, the power-efficiency of accelerators becomes even more critical as the emerging applications often relying on DNNs that require heavy computation due to complex algorithmic operations. This will be continued for a long term

due to the state-of-the-art performance that DNNs have shown in many applications. Fundamental decrease in the computational complexity of DNNs is desirable to reduce energy envelope.

To improve energy efficiency in modern accelerators fundamentally, we explore three mechanisms that directly reduce data movement and heavy computation load in a wide-range of applications. After analyzing architectural behaviors of many applications, we identify unnecessary data communication and computation patterns. The proposed mechanisms attempt to minimize the overhead intelligently, thereby reducing overall energy cost.

We first propose a packet coalescing mechanism that minimizes redundant packets over the NoC of GPUs that share the same data. Graphics Processing Units (GPUs) have been widely accepted for diverse general purpose applications due to a massive degree of parallelism. The demand for large-scale GPUs processing a large volume of data with high throughput has been rising rapidly. Many executions on the GPUs place heavy stress on the memory system, creating network bottlenecks near memory controllers. We observe that data redundancy in communication traffic is commonplace across various applications, called inter-core locality. To exploit the data redundancy, we propose a packet coalescing mechanism to alleviate the network bottlenecks by directly reducing the traffic volume. The key idea is to coalesce multiple packets into one without increasing the packet size when they carry redundant cache blocks. To ensure that the coalesced packets are delivered to their respective destinations, we adopt multicast routing for GPUs' interconnection network. Our coalescing approach yields 15% IPC improvement (up to 112%) in a large-scale GPU with 2D mesh across various GPU applications by reducing average memory access time (AMAT) by 15.5% (up to 65.2%) and obtaining network bandwidth savings by 13% (up to 37%).

Second, we introduce a packet compression mechanism that directly reduces the size of packets over NoC of GPUs each carrying redundant data values. The packet coalesing mechanism is effective but limited to the applications with inter-core locality, which motivates us to explore a more general data compression solution applicable for a wide range of applications. Compression techniques are a practical remedy to effectively increase network bandwidth by reducing data

size transferred. We propose a new simple compression mechanism, Dual Pattern Compression (DPC), that compresses only two patterns with very low latency. The simplicity of compression/decompression is achieved through data remapping and data-type-aware data preprocessing which exploits bit-level data redundancy. We demonstrate that our compression scheme effectively mitigates the network congestion in a large-scale GPU. It achieves IPC improvement by 33% on average (up to 126%) across various benchmarks with average space savings ratios of 61% in integer, 46% (up to 72%) in floating-point, and 23% (up to 57%) in character type benchmarks.

Last, we propose a network optimization methodology that reduces the computation cost of CNNs. Convolutional Neural Networks (CNNs) have become immensely popular in many applications due to their state-of-the-art prediction power. While GPUs in desktops or servers are typically chosen as first-choice-hardware for training CNNs, artificial intelligence (AI) accelerators specialized for CNN inference are adopted in mobile devices. However, high energy cost caused by considerable computation is a major hindering factor in leveraging CNNs in the Internet of Things (IoT) devices with a limited power source. In our proposal, an original CNN is transformed into a CNN with early-prediction capability based on Gradient Boosting Trees (GBTs) that make a prediction with features from each feature extraction network layer. Motivated by the observation that some important features are only necessary for early prediction, our methodology reduces an input size for complex compute kernels (e.g., convolution)) enough to compute important features only. A genetic algorithm finds the best hyperparameters as well as input sizes that maximize the energy-efficiency of inference with an ignorable accuracy drop. Our methodology reduces the energy-consumption of CNNs by 77% with an ignorable accuracy drop in the benchmarks of human activity recognition on average.

The rest of this dissertation is organized as follows. We discuss our packet coalescing and compression proposals for many-core accelerators (GPUs) in Chapters 2 and 3, respectively. Next, we describe a CNN optimization proposal for AI accelerators in Chapter 4. We review the background before detailing a proposal in each chapter. Finally, we draw conclusions in Chapter 5.

2. PACKET COALESCING EXPLOITING DATA REDUNDANCY[1]

2.1 Introduction

Modern GPGPUs, equipped with a large number of computation units, provide energy-efficient executions for a wide variety of high throughput data parallel applications. GPGPUs consist of multiple Streaming Multiprocessors (SMs), each comprising multiple compute units, and a set of on-chip Memory Controllers (MCs) connected via scalable Networks-on-Chip (NoCs) [5] [6] [7]. An enormous amount of parallel thread executions in GPGPUs place heavy stress on the memory systems causing the memory bandwidth to become the critical performance bottleneck [5] [6] [7]. The memory bottleneck leads to long memory access latencies in GPGPUs, which are hidden by fine-grained thread context switching [8] [9] [10].

As technology scales, the number of MCs in the GPGPUs does not scale with the SMs due to on-chip pin bandwidth limitation [3]. This exacerbates the memory bottleneck and renders the latency hiding less effective, due to increased AMAT, eventually leading to significant overall system performance degradation. A considerable portion of AMAT is caused by the *MC bottlenecks* where a large amount of reply data from MCs to SMs cannot be injected into the network due to restricted terminal bandwidth at the MC routers even when the data is ready to be sent [5]. The MC bottlenecks are even more aggravated by network hotspots in the NoC near the MCs that cannot transfer a large volume of traffic fast enough due to limited network bandwidth [6]. Therefore, it is critical to explore solutions in the NoC to alleviate the MC bottlenecks.

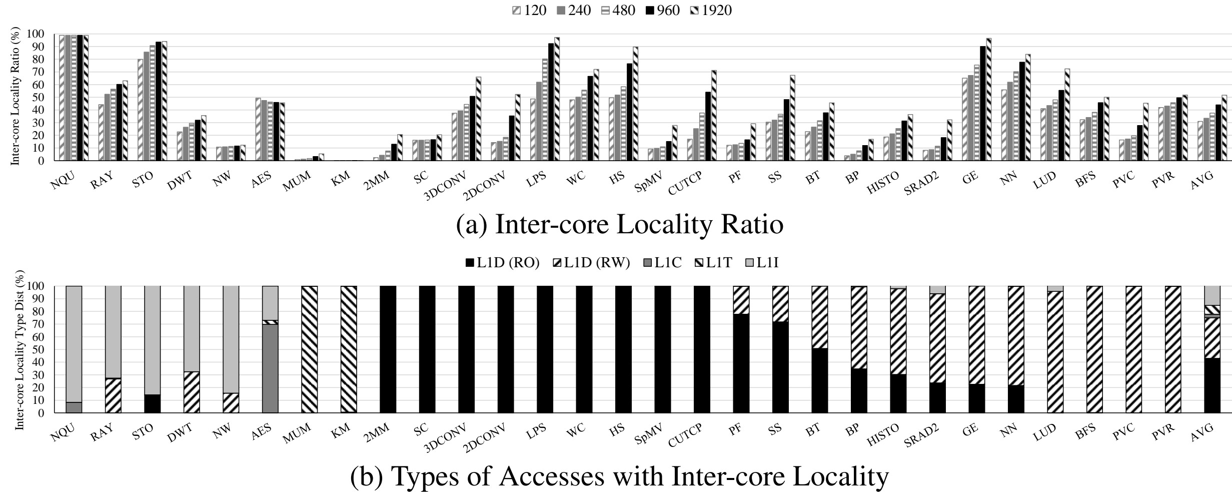
There have been previous studies on designing NoCs tailored to GPGPUs. Bakhoda et al. [5] proposed to provide additional terminal bandwidth using a multiport router design for the MC nodes. Such a design can alleviate the congestion problem at the MC routers by providing additional injection/ejection capabilities but does not reduce the underlying traffic directly. This design also becomes cost-ineffective as the GPGPUs scale up, thereby aggravating the MC router con-

[1]©2017 ACM. Reprinted, with permission, from K. H. Kim, R. Boyapati, J. Huang, Y. Jin, K. H. Yum, and E. J. Kim, Packet coalescing exploiting data redundancy in GPGPU architectures, 07/2017 [4]

gestion. Recent work has attempted to investigate the issues of virtual channel (VC) allocation for request/reply traffic, MC placement, routing algorithm and network topology to find the optimal NoC design for GPGPUs [6] [7]. However, none of these studies tried to solve the MC bottleneck issue in the standpoint of reducing the traffic volume, which we believe is critical to address the issue.

We propose a packet coalescing mechanism that reduces NoC traffic volume by exploiting data redundancy in the GPGPU communication traffic. The proposed mechanism coalesces multiple packets that exhibit data redundancy into a single packet without increasing the packet size, thereby reducing the number of packets injected into the network. Data redundancy in GPGPU communication stems from data sharing among multiple SMs, called *inter-core locality* [11]. We introduce a packet coalescing unit (*PCU*) in the MCs which captures a group of memory requests with inter-core locality from multiple SMs, and generates a single read reply packet destined for the requesting SMs. To make sure that the single packet is delivered to all the SMs, we adopt an existing multicast routing for GPGPUs. In this paper, we make the following contributions.

- We propose a new packet coalescing mechanism that alleviates the MC bottlenecks through traffic volume reduction.
- We adopt a multicast routing algorithm that delivers coalesced packets to SMs. To the best of our knowledge, this is the first work showing a good use of multicast in GPGPU applications.
- We analyze the MC bottleneck issue and inter-core locality common across various applications and characterize applications with inter-core locality.
- We comprehensively evaluate the proposed coalescing technique across various applications from GPGPU-SIM [12], Rodinia [13], Mars [14], Polybench [15], and Parboil [16] benchmark suites. Our coalescing approach yields 15% IPC improvement on average in a large-scale GPGPU with 2D mesh by reducing AMAT by 15.5% and obtaining network bandwidth savings by 13%. Also, our coalescing approach achieves 7% IPC improvement in the NVIDIA Fermi architecture with the crossbar.



(a) Inter-core Locality Ratio

(b) Types of Accesses with Inter-core Locality

Figure 2.1: Data Redundancy across 29 Benchmarks in 2DMesh

2.2 Motivation

In this section, we explain the MC bottleneck issue in a GPGPU with a 2D mesh/crossbar interconnect and motivate our proposed mechanisms based on the observation of widely common data redundancy in GPGPUs.

2.2.1 MC Bottleneck

A large amount of parallelism in GPGPUs places heavy stress on the limited number of MCs on the chip, especially because L2 cache banks are located only in the MC nodes, and hence every L1 cache miss access is destined for one of the MCs through an interconnection network. The communication patterns in GPGPUs are many-to-few in the request network from many SMs to a few MCs, and reversely few-to-many in the reply network from a few MCs to many SMs [5].

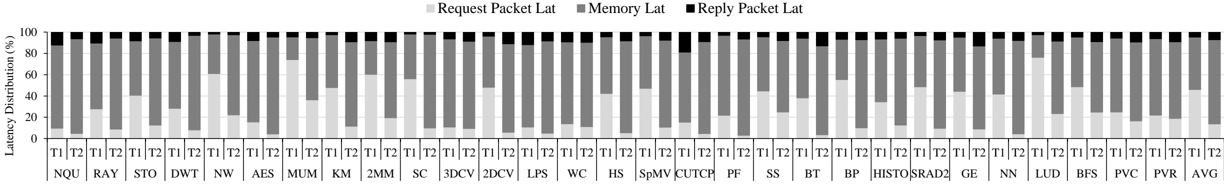
To understand the key reason of the MC bottlenecks, we analyze L1 cache miss penalty of AMAT in two different scales of GPGPUs. We model a large-scale GPGPUs of 56 SMs and 8 MCs with 2D mesh [6], while we do NVIDIA Fermi architecture of 15 SMs and 6 MCs with the crossbar [17]. Through these experiments, we see severe bottlenecks occur in both GPGPU models and the bottlenecks are mainly due to a large volume of traffic highly skewed toward the reply network.

Figure 2.2a[2] shows the breakdown of L1 cache miss penalty measured for all memory requests across 29 benchmarks. The penalty is divided into three latencies: request packet latency, memory latency, and reply packet latency. The request and reply packet latencies are calculated from the time packet's flits are created to the moment when its tail flit is accepted in the final destination. The memory latency is from the time a request packet is accepted by an MC to when a corresponding reply packet is created.

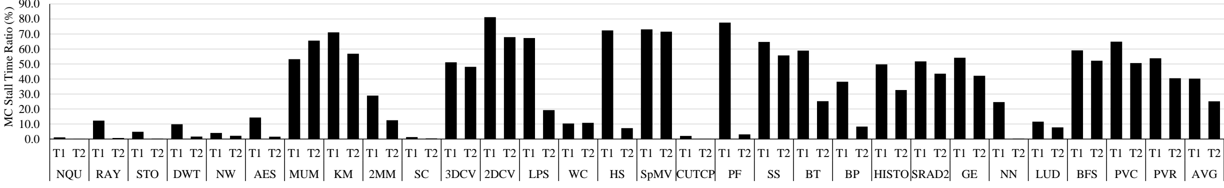
The MC bottlenecks are presented in the request packet latency that is asymmetrically longer than the reply latency. The average request latency is 10 and 2 times higher than the reply latency in 2D mesh and crossbar, respectively. It is due to the backpressure from the highly congested reply network to the request network. Once reply data is read from memory systems, it stays in MC reply queue placed between MC and its Network Interface (NI) input queue, waiting for being sent to the network. As the reply network gets more congested, the MC NI input queue is full, and thus reply data cannot be sent immediately and keeps waiting in MC reply queue. When the reply queue becomes full, an L2 cache cannot accept a request in MC request queue that stores new memory requests from SMs. It is because the reply queue has no more space to store reply data when the request hits the L2 cache. Then, request packets continue to wait in the request network until the MC request queue has available space, resulting in a long request latency.

To quantify the severity of MC bottlenecks, we measure the ratio of average MC stall time out of the total execution time for each benchmark. We count the stall time when MCs cannot inject packets due to the MC NI input queue being full. The average MC stall time ratio in 2D mesh is 40.4% as shown in Figure 2.2b. Such frequent MC stall has been also observed by earlier GPGPU NoC design work [5]. Interestingly, MCs in the crossbar incorporated by modern GPGPUs also stall 25.1% of execution time, particularly in memory-intensive applications. Consequently, the MC bottleneck increases AMAT and, in turn, degrades the overall system performance. Therefore, it is crucial to reduce the stalls by fundamentally reducing the number of packets sent to the MC

[2] To analyze MC bottlenecks in the interconnect perspective, we present the L1 cache miss penalty rather than SM stall cycles. Due to severe response delay by the bottlenecks, the performance is highly affected by the miss penalty, although GPGPUs are designed for hiding latency.



(a) L1 Cache Miss Penalty



(b) MC Stall Time

Figure 2.2: MC Bottlenecks in GPGPUs across 29 Benchmarks (T1: 2DMesh, T2: Crossbar)

NI input queue.

2.2.2 Data Redundancy

Data Sharing among SMs. Inter-core locality occurs when multiple SMs send requests to the same cache block within a relatively short period of time. In order to quantify its potential and temporal locality, and identify its sources, we analyze cache block access patterns in each MC across a wide-range of applications.

Once a read request arrives at an MC router, it is sent to L2 cache. Between them, we capture all read requests that are going to access the same cache block. To do this, we maintain a table where a cache block address is associated with the number of accesses in each entry. If a cache block address of a request does not exist in the table, a new entry is allocated, storing the block address and initializing the number of accesses to one. The entry is deallocated after a fixed-length time window. If a block address of a request does exist, the number of accesses increases by one. To capture all requests, the number of entries is assumed to be unlimited. Five time windows such as 120, 240, 480, 960 and 1920 cycles are adopted by taking multiples of the minimum L2 hit latency (i.e. 120 cycles) [17]. An entry has inter-core locality when it records multiple accesses. The inter-core locality ratio is measured by the percentage of the total number of accesses in all

entries with inter-core locality out of the number of all read requests. As shown in Figure 2.1a, 31% of the requests have inter-core locality on average, when the time window is set to 120 cycles. As the time window is increased to 1920 cycles, the inter-core locality ratio increases up to 51.7%.

Figure 2.1b shows the distribution of access types with inter-core locality at the time window of 960 cycles. Inter-core locality mainly occurs from L1 data (L1D) cache misses by 75% where read-only data takes 43% and read-write data takes 32%. Modern GPGPUs do not support cache coherence protocols among SMs [17], but synchronization method among SMs is often used to avoid data race circumstances on the read-write data. Other sources of inter-core locality are cache misses from read-only caches such as L1 instruction (L1I), constant (L1C) and texture (L1T) cache , which take 16%, 2.8% and 7%, respectively.

Characterization of Applications. The inter-core locality associated with L1D cache misses occurs when an application is written to run many threads accessing shared data structures. We characterize the applications in terms of their computation characteristics on the shared data as follows.

- *Pair-wise Computation.* In MapReduce framework applications, Map stage passes a list of key and value pairs to Reduce stage. Group stage between them sorts the output of Map stage by keys. In the sorting process, threads fetch and compare data elements. When the elements that each SM needs exist in a cache block, inter-core locality occurs. Similarly, SC, CUTCP, HISTO and SpMV have inter-core locality due to pair-wise computation features.
- *Graph Data Computation.* In applications using graph data such as BFS, BT, NN and BP, a data node is explicitly connected with neighboring nodes. In their computation flow, they usually involve checking or obtaining previous data nodes. The inter-core locality occurs when multiple nodes processed by different SMs need data from the same previous node.
- *Stencil Computation.* Applications compute a data point by using neighboring data points. Although SMs are assigned a distinct data tile, the boundary regions, called halo regions [18], around the data tile are redundantly accessed by multiple SMs. HS, PF, SRADV2, 3DCONV, 2DCONV and LPS belong to this type.

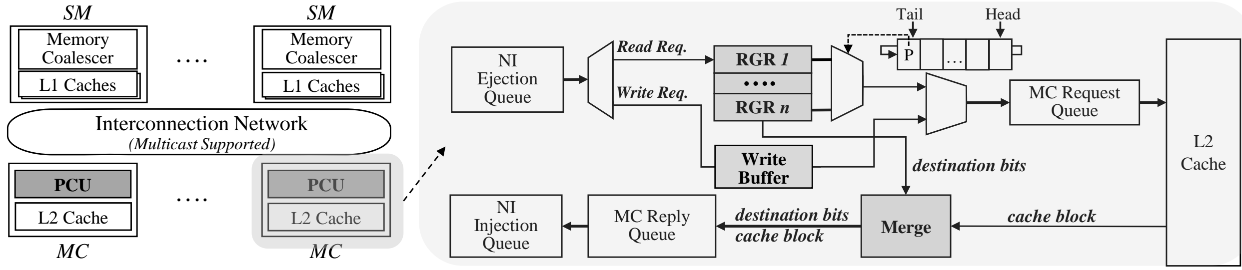


Figure 2.3: GPGPU Architecture Incorporating Proposed PCUs

- *Computation with Row-wise and Column-wise Dependency.* Applications such as LUD, GE and 2MM access row and column data points associated with a data point, to compute the point. As a large dataset cannot be fit in the shared memory of an SM, such row/column data is necessary for multiple SMs.

2.3 Packet Coalescing Unit

In this section, we explain our packet coalescing units (PCUs) that reduce the number of packets by combining multiple packets into one without increasing the packet size.

2.3.1 Overview

Figure 2.3 shows the overall GPGPU architecture integrating PCUs to MCs. In each SM, memory accesses from threads of a warp are combined into fewer accesses by a memory coalescer [17]. When they have L1 cache misses, memory read or write requests packets are sent to MCs through an interconnection network. The MCs respond to them with read reply or write reply packets, respectively. The read reply packets, which are a key factor of MC bottleneck, are coalesced by a PCU, before being injected into the network. A PCU coalesces packets by up to the number of all SMs. To deliver the packet to final destinations, the interconnection network requires multicast capability that we will discuss in Section 2.4. In the following, we explain the details of a PCU.

2.3.2 Coalescing Mechanism

The proposed coalescing mechanism has two key features. First, multiple packets are coalesced into a single one without packet size increase. To coalesce packets one may think of appending packets back-to-back, but it does not attain our goal of reducing traffic volume. Instead, to exploit inter-core locality described in Section 2.2, we attempt to merge only read reply packets carrying the same cache block into a single one. The packet header and payload store multiple destinations and the cache block, respectively. As the unused space of a packet accommodates the destinations, the packet has the same size as a normal (i.e. uncoalesced) packet.

Second, coalescing is performed with low latency overhead. One way of coalescing is to keep track of cache blocks in the MC reply queue and merge ones with the same contents, while they are in the queue. Thus, the longer the packet stays in the queue, the more packets can be coalesced. However, adding extra waiting time to earlier packets is not desirable since it increases end-to-end latency. We determine a group of reply packets to be coalesced when their corresponding requests arrive at an MC. This process is called *Request Grouping*. Once a cache block for a group of requests returns from memory systems, we provide information such as the cache block and SM ids with NI to generate a packet destined for the SMs. This process is called *Reply Merging*. The request grouping leads to low latency overhead for identifying requests accessing the same cache block, while the reply merging does no overhead.

Suppose that a request to a cache block arrives at an MC from SM 1 and there were no requests to the block so far. The request grouping allows the request to access memory systems and records SM 1 as a requesting SM. Until the accessed block returns, if subsequent requests to the same block are sent from SM 2 and 3 to the MC, the request grouping records SM 2 and 3, and it does not allow them to access memory systems. When the block returns, the reply merging sends all recorded SM ids (i.e. 1, 2 and 3) and the cache block to NI that creates a packet destined for the SMs. Now we describe the details of request grouping and reply merging

Request Grouping. The request grouping is performed in a static time window that depends on memory access time. When a request forms a new request group, it is sent to memory systems.

The request group continues to capture all subsequent requests that access the same cache block as the first one does. The number of accesses to the same cache block is bound by the number of SMs since the redundant accesses from each SM are blocked by MSHRs of L1 cache. This grouping is terminated when a cache block for the first request returns. This operation is similar to miss status holding register (MSHR) mechanism of an L2 cache. However, we introduce the request grouping mechanism before L2 cache separately to capture more requests with inter-core locality. GPGPU has a long L2 cache access time (120 cycles [17]) due to the delay of raster operation (ROP) unit coupled with an L2 cache. The request grouping can use it as the minimum time window when a request hits L2 cache. Upon miss in an L2 cache, the request grouping can make use of DRAM access time in addition to the L2 cache access time, which is the maximum time window. The accesses that do not access the main memory are frequently captured by request grouping with the minimum time window due to their temporal locality shown in Figure 2.1a.

To implement the request grouping, we introduce a Request Grouping Register (*RGR*) which groups requests with inter-core locality by storing a cache block address and their requesting SM ids. RGR has 1-bit *valid* field, 41-bit *block address* field, and 64-bit *destination bits* field where each bit position indicates the location of a requesting SM. The RGR that stores requests with inter-core locality has multiple bits of *destinations bits* field set to ones.

We design the request grouping in two stages to perform the grouping while the MC request queue is full. To send read requests to L2 cache in their arriving order, we maintain the PCU pointer ring-buffer where the locations of RGRs are stored according to their allocation order. The PCU head/tail pointers are used to read RGRs in that order. The request grouping mechanism operates in the following manner.

- **Stage 1.** When there is an available RGR, a read request is accepted by PCU. For the read request, all valid RGRs are sequentially accessed to find a match on the block address. If there is a hit in a valid RGR, the requesting SM id is stored in the destination bits field and the request is dropped (not sent to MC request queue). If an RGR miss occurs, an empty RGR (the *valid* field is zero) is located. The requesting SM id is stored in the *destination bits* field

of the RGR. The accessing address is also stored in the *block address* field. The PCU head pointer is set to next available space in the PCU pointer buffer. The RGR location is stored at the space.

- **Stage 2.** Next read request is selected based on an RGR pointed by the PCU tail pointer. When the PCU head and tail pointers are the same, no read request is available. If MC reply queue has available space, a request selector chooses either a read request from the selected RGR or a write request in a write buffer in a round-robin way. When the read request is selected, the block address of the RGR is sent to MC request queue and the PCU tail pointer is set to next valid RGR.

Reply Merging. We introduce Merge unit that combines multiple replies in a single reply. Merge unit stands between L2 cache and MC reply queue. When a cache block returns from L2 cache, the Merge unit obtains the destination bits by accessing the corresponding RGR. Both the cache block and the destination bits are sent to the MC reply queue. At this point, the RGR is reset by clearing its *valid* field for new RGR allocation. The cache block is packetized by NI as a reply packet for SMs encoded in the *destination bits* field. A flit that stores a packet header accommodates the destination bits in its unused space (e.g. 8B in 2D mesh). If the *destination bits* field encodes a single destination, a reply packet is sent to the destination as a unicast packet. Otherwise, the reply packet is a multicast packet sent to all requesting SMs, which we will discuss details in the next section.

2.4 Multicast Support in NoC

In this section, we detail multicast support for both large-scale and NVIDIA Fermi-style GPGPU architectures.

2.4.1 Overview

First, we present an overview of the NoC architectural details for both large-scale and NVIDIA Fermi-style GPGPUs. NVIDIA Fermi architecture uses a global crossbar interconnection network

with destination tag routing [17]. For large-scale GPGPU architectures, we propose to use a 2D mesh interconnection topology among various NoC topologies as in [19] [6] because the global crossbar is not a practical solution due to the complexity of layout and huge power consumption [7]. The efficiency of the proposed packet coalescing mechanisms, which exploit the application behavior of inter-core locality, is independent of the underlying interconnection network topology. For the global crossbar, the request and reply networks are separated by two different crossbar switches. For the 2D mesh, a single network is used for both request and reply communication. To avoid protocol deadlocks, the network is divided into two virtual subnetworks for the respective communication, where VCs are evenly dedicated to each subnetwork [12].

2.4.2 Multicast in Crossbar

To support multicasting in the crossbar, flit replication capability is primarily needed. To enable replication with high throughput, we manifest the matrix-crossbar in the Fermi architecture into a mux-based crossbar. In earlier multicast studies like VCTM [20], replication is performed by reading the same flit out of a VC and sending it to each output port one-by-one upon successful allocation. This has an advantage of a simple crossbar design but incurs serialization delay to multicast flits. Hence, we adopt a mux-based crossbar used by RPM [21] that supports high throughput at the cost of higher energy consumption.

2.4.3 Multicast in 2D Mesh

For multicast support in the 2D mesh topology, we adopt a multicast router supporting tree-based routing, similar to VCTM [20], RPM [21], BAM [22] and Whirl [23]. The routing algorithms in these routers have been optimized for the traffic patterns in Chip MultiProcessors where core-to-core communications are frequent. Jang et al [6] has shown that a Dimension Order Routing (DOR) is simple but effective in GPGPU due to the traffic patterns occurring between SMs and MCs only. Therefore, the multicast router in this paper implements DOR.

We use a 3-stage lookahead router as the baseline router. A traditional NoC router has four stages: Routing Computation (RC), VC Allocation (VA), Switch Allocation (SA) and Switch

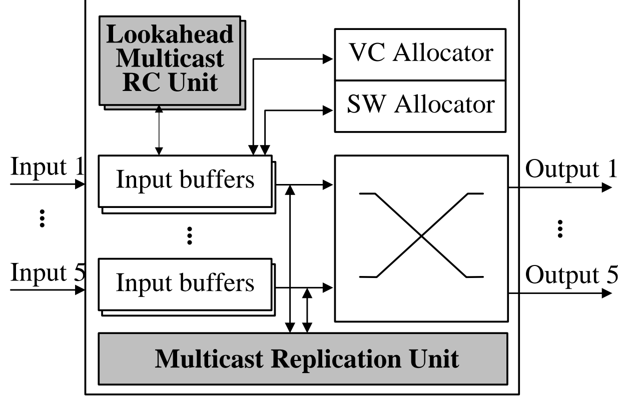


Figure 2.4: Multicast Router Architecture

Traversal (ST). To reduce the pipeline depth, the 3-stage lookahead router performs the routing computation for next hop router in the VA stage [24].

To support multicast routing in the baseline router, we incorporate a replication unit and lookahead multicast RC units as depicted in the shaded units in Figure 2.4. The replication unit copies a multicast packet at a replication point to different directions to make sure that the packet arrives at all final destinations. The multicast RC units decide the output port list to which replicated packets are directed, based on DOR. For each replica, it also splits a destination list of an original packet into a subset being routed via the same output port. Then, VC allocator uses the output port list to get an available VC from the downstream routers for replica packets. As a replica gets a free VC, it goes to the SA stage and a packet is replicated to an output port at the ST stage, storing the destination list for the replica in replica's header [21].

We use multiple RC units to support lookahead routing decision for replicas.When a packet is replicated to multiple directions, we need to make sure that the lookahead routing decision is performed for each replicated packet, which causes additional complexity. For all replicas, each input port is required to have lookahead RC capability for all immediate neighboring routers. Since the multiple RC units work in parallel, they can be overlapped with the VA stage without increasing the critical path.

As a replication scheme, we choose an asynchronous replication scheme [21] where flits tar-

geted for different destinations are forwarded independently. Then VA, SA and ST are done for each flit individually. An input VC keeps storing a flit until it copies the flit to all target output ports [21] [22]. Replicating a packet to multiple output ports may have conflicts with other normal packets already in the router. Replicated flits are handled like normal flits for the VA/SA stages without giving priority on the replicated flits. To enable replication with high throughput, we choose the mux-based crossbar inside the multicast router discussed in Section 2.4.2.

2.5 Evaluation

2.5.1 Methodology

To evaluate the proposed packet coalescing we integrate PCUs into a cycle-accurate GPGPU simulator, GPGPU-Sim 3.2.2 [12]. We modify Booksim [25], the NoC simulation component of GPGPU-Sim, to simulate multicast for crossbar and 2D mesh. To see the impact of routing to coalescing performance, we use two routing algorithms for 2D mesh, XY-XY and XY-YX, where both uses XY routing in the request network, and use XY and YX routing, respectively for the reply network. The number of RGRs that affect coalescing performance is set to 128 for each PCU. We use CACTI model 6.5 [26] to measure latency and energy overhead of RGR. Table 4.1 shows the detailed system parameters we use to model the baseline GPGPU architecture.

We select a variety of applications from multiple benchmark suites: AES, LPS, MUM, NN, NQU, RAY and STO from GPGPU-Sim [12], BFS, BP, B+tree (BT), Discrete Wavelet Transform (DWT), Gaussian Elimination (GE), HS, KM, LUD, NW, Path Finder (PF), SC and SRAD2 from Rodinia [13], CUTCP, HISTO, and SpMV from Parboil [16], PVC, PVR, SS and WC from Mars [14], and 2DCONV, 2MM and 3DCONV from Polybench [15][3]. We choose a mix of compute bound and memory bound benchmarks so as to show the prevalence of data redundancy across diverse applications.

Memory coalescing has a significant effect on reducing the number of memory requests because memory accesses from many threads are merged into smaller ones. Thus, we evaluate our packet coalescing mechanism in the presence of an intra-warp memory coalescer [17]. Also, we

[3] We use abbreviations of benchmarks as presented in their literatures

System Parameters	Details
Shader Core	56 / 15 Cores, 1.4Ghz
Memory Model	8 / 6 MCs, 924 MHz
Warp Scheduler	Greedy-Then-Oldest (GTO)
L1I, L1T, L1C Cache	2KB, 12KB, 8KB
L1D Cache, Shared Memory	16KB, 48KB
L2 Cache	64KB
Min L2, DRAM latency	120, 220 cycles
Topology	8 x 8 Mesh / Crossbar
Virtual Channel	4 VCs per Port (8-Flit Buffer)
Routing	DOR / Destination Tag
Flow Control	Wormhole, Credit-based
Channel Width	128 Bits / 256 Bits

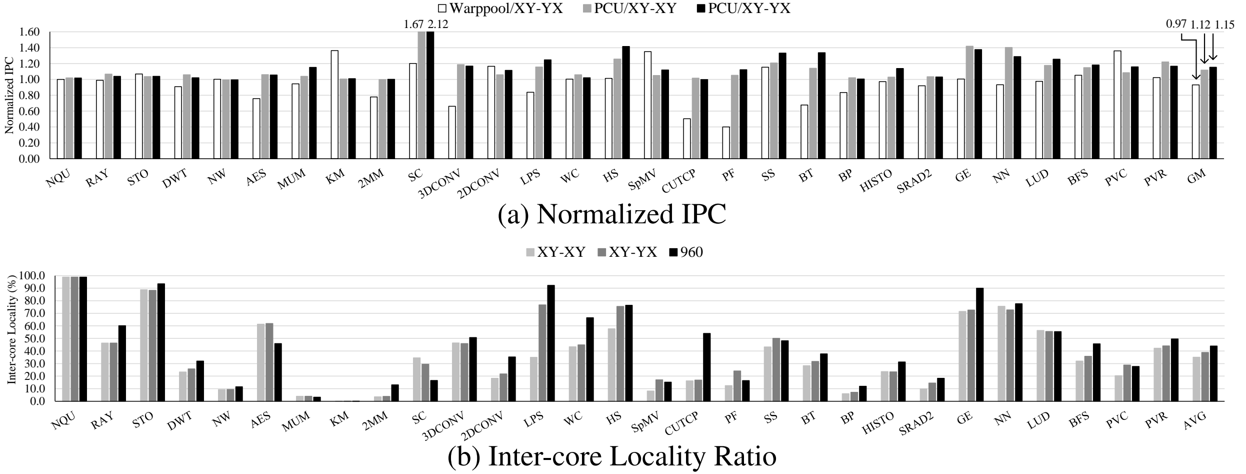
Table 2.1: System Configuration Parameters

compare ours against a novel inter-warp memory coalescer (Warppool) [27] which merges more memory accesses from different warps on top of the intra-warp memory coalescing. As a result, Warppool can be used as an effective means to mitigate the MC bottlenecks. For fair comparison, we implemented the FIFO request selection policy both in our mechanism and in our implementation of Warppool. Note that Warppool also uses prioritization policy proposed by MRPB [28].

2.5.2 IPC Improvement Analysis

Figure 2.5a compares the normalized IPC of all benchmarks when Warppool is used with routing algorithm XY-YX, and PCUs are used with XY-XY and XY-YX. Each IPC is normalized over the baseline using the corresponding routing combination. Since the request grouping and reply merging in each PCU work together as a mechanism for packet coalescing, we do not show benefit for each separately.

We make two major observations in the IPC performance analysis. First, the proposed coalescing approach is more effective than Warppool. In XY-YX routing, our approach provides 15% IPC improvement on average, while Warppool does IPC performance degradation by 3%. With Warppool, only 8 out of 29 benchmarks (28%) have more than 5% IPC improvement, while others



(a) Normalized IPC

(b) Inter-core Locality Ratio

Figure 2.5: System and Coalescing Performance

have performance degradation or minor improvement. Warppool is a novel idea, but the overhead of merging requests from different warps causes performance degradation in the benchmarks with limited inter-warp locality. Especially, since the merging process is on the critical path of cache accesses, the performance degradation appears more severe for benchmarks with low L1 data cache miss rates (e.g. PF and BT).

Second, the proposed coalescing approach becomes more effective when a better routing algorithm that mitigates reply network hotspots is used. As XY routing in the reply network causes network hotspots near MCs with bottom MC placement, YX routing has been shown more effective [6]. Our approach achieves the highest IPC improvement 15% with XY-YX routing, while it does 12% with XY-XY routing. Such performance gap arises due to worse coalescing performance in XY-XY routing, which is shown in benchmarks such as LPS, HS, SpMV, PF and PVC. Reply packets under XY routing are delivered with delay due to the network hotspots. After the reply packets are accepted by SMs, next requests with inter-core locality are sent to MCs with worse temporal locality, so PCUs are limited in involving more requests in request grouping.

Synergetic Effect of PCU and Warppool. Both PCU and Warppool synergetically improve the overall IPC performance when they work together, as shown in Figure 2.6. We simulate both mechanisms for benchmarks benefitting from Warppool. Both mechanisms achieve IPC improve-

ment by 41% on average, while PCU and Warppool do by 22% and 21%, respectively. Such synergetic effect is due to the difference in the target that each mechanism works for. Warppool attempts to reduce unnecessary memory requests caused by inefficient use of an L1 cache (e.g. cache thrashing), but necessary requests to fill the L1 cache are sent and these still cause the MC bottlenecks. By reducing traffic volume of the corresponding replies with inter-core locality, the benefit from PCU keeps valid with Warppool. However, SC is more effective with PCU only. When Warppool is used only, the requests are waiting in SMs due to the backpropagation of MC bottlenecks, so Warppool effectively works since the latency overhead of merging requests is hidden. However, as our packet coalescing is introduced, this latency hiding is less effective since requests do not wait, thereby degrading the performance.

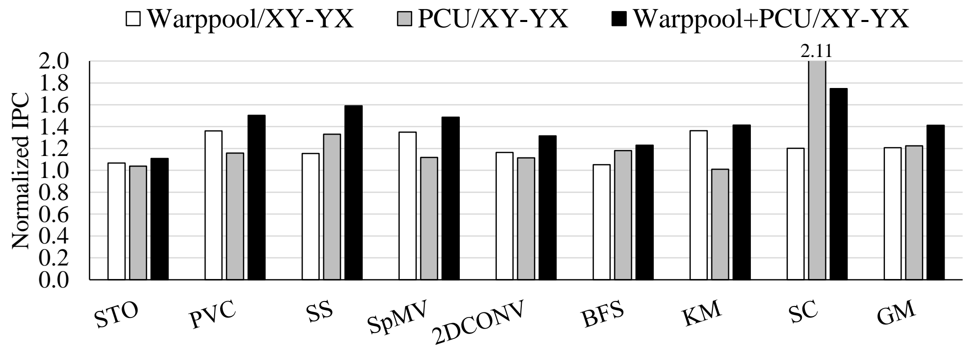


Figure 2.6: Comparison of Normalized IPCs

2.5.3 Packet Coalescing Analysis

The IPC performance improvement is mainly attributed to AMAT reduction by packet coalescing. We first analyze coalescing performance according to routing algorithms, then discuss the MC bottlenecks alleviated by the coalescing, and finally discuss memory regions with inter-core locality on two applications.

Coalescing Performance. We measure the coalescing performance based on the actual inter-core locality ratio measured by the percentage of the number of coalesced packets out of the total

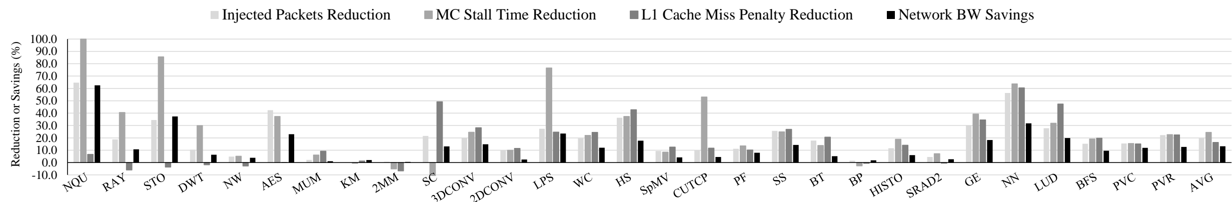


Figure 2.7: Injected Packets, MC Stall Time and L1 Cache Miss Penalty Reduction and Network Bandwidth Savings

number of reply packets. In this analysis, we make two conclusions. First, the coalescing performance is affected by routing algorithms. As shown in Figure 2.5b, the actual inter-core locality ratio is 38.9% under XY-YX on average, while it is 35% under XY-XY. As discussed in Section 2.5.2 and [6], XY-XY routing suffers from more severe bottleneck than XY-YX in the reply network. It causes reply packets to be delivered to each SM with more delay between them. Accordingly, next requests with a potential inter-core locality are sent from each SM more sparsely. As a consequence, some requests lose a chance of grouping under XY-XY routing. However, there are some outliers that have slightly higher inter-core locality under XY-XY. For instance, SC has 34.5% and 29.4% inter-core locality ratio in XY-XY and XY-YX, respectively. Coalescing performance in SC is less sensitive to the temporal locality of accesses due to its long memory latency caused by high L2 cache miss rate (97%). The high MC bottleneck favorably gives larger time window for grouping, so extra requests are additionally grouped to existing RGRs backed up by a higher average number of coalesced packets in XY-XY than XY-YX.

Second, PCUs capture most of the requests with inter-core locality (88.4%) by using memory access time as its time window. To understand this, we conservatively compare the actual inter-core locality ratio to the potential ratio of 960-cycle time window in Figure 2.1a because the average memory latency is 649 cycles under XY-YX. The potential inter-core locality ratio is 44.0% on average, while the actual inter-core locality is 38.9%. By giving extra time to PCU's time window, PCUs are able to capture 5% more requests. However, the extra time becomes as a delay overhead to AMAT of 38.9% requests. This offsets the benefit of AMAT reduction by packet coalescing,

thereby gaining no performance improvement.

Impact of Coalescing. We summarize two conclusions. First, our packet coalescing reduces AMAT by 15.5% and saves network bandwidth by 13%. As the packet coalescing merges multiple packets into one, the number of packets injected into the reply network is reduced by 19.7% on average, which alleviates MC stall time by 24.5% and finally leads to L1 cache miss penalty reduction by 16.3%, as shown in Figure 2.7. As a consequence, AMATs for L1I, L1C, L1T and L1D caches are reduced by 16.1%, 15.9%, 3.8% and 26.2%, respectively on average. The average AMAT reduction of all L1 caches is 15.5%.

Second, SC shows an interesting result where the MC stall time increases by 88% but L1 cache miss penalty is reduced. The impact of the increased stall time is minimal. The MC stall time ratio is 1.3% in the baseline as shown in Figure 2.2b, and increases to just 2.5% when coalescing is used. However, PCUs helps to alleviate bottlenecks caused by long memory latency. While requests accepted by MC keep waiting for their turn for memory accesses in the queue from L2 to DRAM, these backpressures back the MC request queue to be frequently full. MC node in the baseline cannot accept new requests, leaving them to wait in the request network, which makes a bottleneck. However, PCUs continue to accept requests for grouping, while MC is busy with reading data from DRAM. As a result, 29.4% requests are grouped and L1 cache miss penalty is reduced by 49.2%.

Memory Region with Inter-core Locality. Figure 2.8 depicts the entire memory region used by two applications, SS and LUD where the normalized degree of inter-core locality for all cache blocks is illustrated as a heatmap. To measure the degree, the number of requests with inter-core locality for each cache block is counted. Its normalized degree of inter-core locality is calculated as the count value of each block divided by the maximum count value among all cache blocks. To locate each cache block on the plot, the x and y axes indicate the row-wise and column-wise offsets from the base address of a global memory (i.e. 0x80000000).

Figure 2.8a shows almost all cache blocks that store an input matrix have high inter-core locality. It is because LUD kernels have many dependencies on row-wise and column-wise data [13].

LUD has three kernels such as lud_diagonal, lud_perimeter and lud_internal. Among accesses with inter-core locality, 88% and 12% occurs in lud_internal and lud_perimeter, respectively. Interestingly, cache blocks on the top-left region have higher inter-core locality than others. As LUD diagonally processes a matrix from top-left to bottom-right direction over multiple iterations, a range of data that a kernel needs to compute shrinks and thus the number of running SMs gets decreasing. Thus, the data on left side is accessed by more SMs, thus showing a higher degree of inter-core locality.

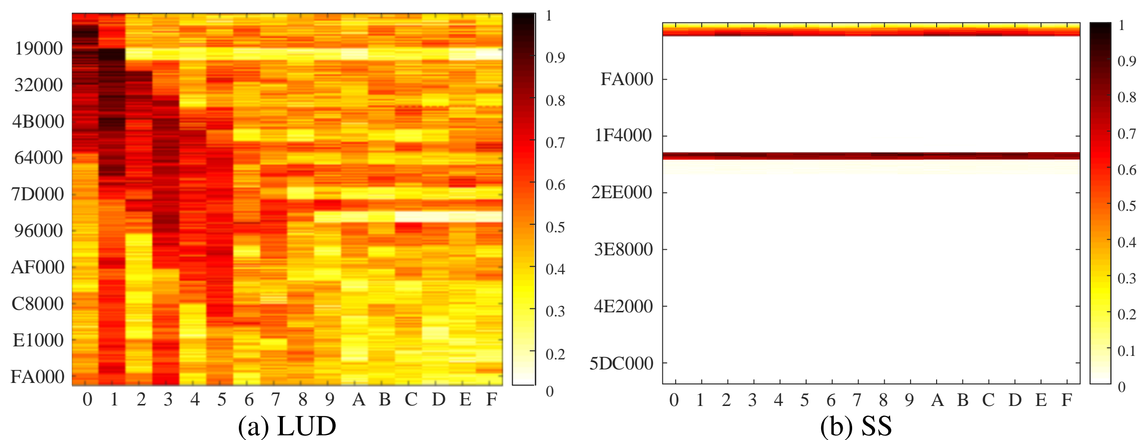


Figure 2.8: Memory Region with Inter-core Locality

SS usually has inter-core locality in two memory regions associated with Map and Group stage, respectively in the MapReduce framework as shown in Figure 2.8b. In the framework for SS, Map stage computes similarity scores for all pair-wise documents by using their feature vectors, while Group stage sorts the pair-wise similarity scores. As such pair-wise computation is performed through multiple thread blocks for large input data, Map stage running across multiple SMs needs to access feature vectors of redundant documents, which appears on the top in Figure 2.8b. 71% of inter-core locality occurs in this stage. Similarly, Group stage also needs to perform a pair-wise comparison between two scores to sort a series of similarity scores. 29% inter-core locality is related to Group stage, which is shown in the middle of Figure 2.8b. As Group stage is necessary

for MapReduce-based applications, PVC, PVR and WC also have inter-core locality in this stage. While the sorting process in the stage requires frequent data movement [14], our coalescing unit effectively eliminates the bottlenecks caused by the data movement.

2.5.4 Sensitivity Analysis

Impact of RGR Size. We evaluate IPC performance and coalescing performance across all benchmarks as varying the number of RGRs: 64, 128, 256, 512 and 1204. Our coalescing technique achieves 39.4% inter-core locality ratio on average and 19% IPC improvement with 1024 RGRs across all benchmarks. Most benchmarks achieve saturated IPC performance at 128 RGRs except four benchmarks shown in Figure 2.9. While three benchmarks such as SC, SpMV and SS gain saturated performance at 256 RGRs, MUM obtains a monotonically increasing IPC improvements until 1024 RGRs are used. As the number of RGRs grows from 128 to 1024 in MUM, the inter-core locality ratio increases from 4% to 16% as shown in Figure 2.9, and the IPC improvement does from 15% to 88%. This happens because MUM has higher memory intensity than others [13].

Impact of L2 Hit Latency. To exploit a long hit latency of L2 cache for request grouping, we place PCUs before L2 cache as discussed in Section 2.3.2. We study the impact of a shorter L2 hit latency on coalescing performance. We model the L2 cache hit latency as 2 cycles based on CACTI model [26]. Figure 2.10 shows the normalized IPCs when the minimum L2 hit latency is set to 120 and 2 cycles, respectively. The IPC values of two configurations are normalized against the baseline with corresponding L2 latency. It also plots inter-core locality ratio as a line for each latency case. The IPC improvement increases up to 24% on average at 2-cycle hit latency, while it is 15% at 120-cycle hit latency. However, the average inter-core locality ratios do not show noticeable differences, which are 39% and 37% in the 120-cycle and 2-cycle latencies, respectively. As the cache access time is reduced in the 2-cycle case, the injection rate of reply packets becomes higher, which causes more severe MC stalls. Our coalescing becomes relatively more effective as a bottleneck alleviator in the 2-cycle case, resulting in higher IPC improvement.

Impact of MC Placement. We compare four configurations such as bottom, top-bottom, edge

and diamond MC placements studied in the previous literature [6]. The IPC values of all different configurations are normalized against the baseline with corresponding MC placements and XY-YX routing. Our coalescing technique achieves similar average IPC improvements, 15%, 15%, 14% in bottom, top-bottom and edge MC placements, while it does lower improvement, 11% in diamond (unplotted). The diamond MC placement is commonly known as the optimal placement [29], but it is not when multicast is used. When MC nodes serve as a replication point of multicast packets, it causes contention between replicated packets and injected packets. As a result, it offsets the benefit of the MC bottlenecks lessened by our coalescing technique.

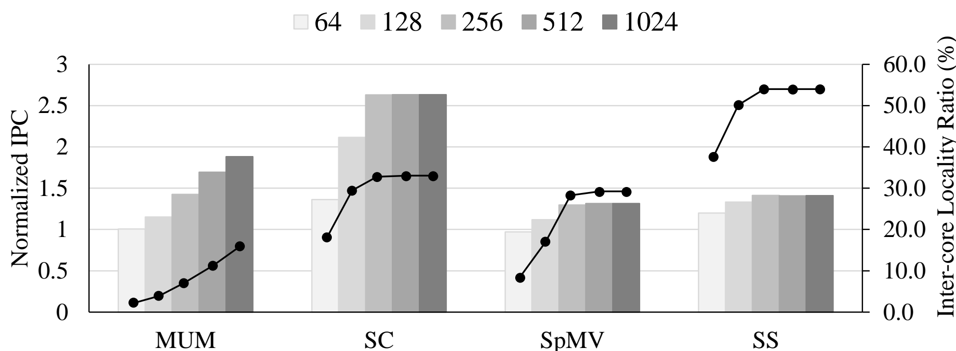


Figure 2.9: Normalized IPC (Bar) and Inter-core Locality Ratio (Line) as varying the number of RGRs

2.5.5 Coalescing in the Global Crossbar

We choose 17 benchmarks with MC bottlenecks under the crossbar such as MUM, KM, 2MM, SC, 3DCONV, 2DCONV, LPS, WC, SpMV, SS, BT, HISTO, SRAD2, GE, BFS, PVC and PVR. Our coalescing technique achieves 30.3% of inter-core locality ratio and yields 7% IPC improvement on average (unplotted). On the other hand, Warppool obtains 28% performance degradation on average and only shows IPC improvement for a few benchmarks such as KM, SC, SS and PVC by 21%, 6%, 10% and 23%, respectively.

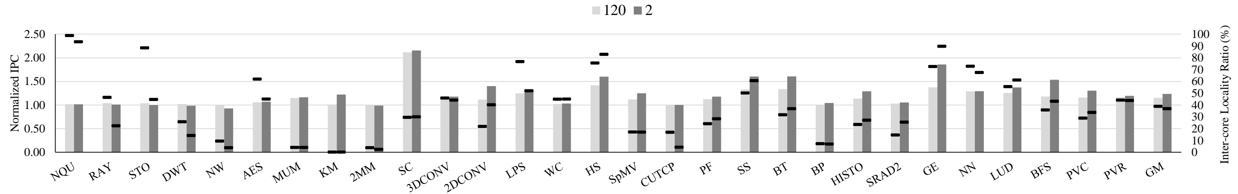


Figure 2.10: Normalized IPC (Bar) and Inter-core Locality Ratio (Line) when a minimum L2 hit latency is 120 and 2 cycles

2.5.6 Hardware Cost

Coalescing Overhead. We analyze the area overhead incurred by the proposed PCUs. Since a PCU uses 128 RGRs and PCU pointers which take 14B and 7 bits, respectively, a PCU per MC incurs the total overhead 1904B. As shown in Table 4.1, an SM has 86KB L1 caches and an MC has 64KB L2 cache. Compared to the total cache infrastructure of 56 SMs and 8 MCs, the total overhead incurred is just 0.28%. The overheads of RGR are summarized in Table 2.2.

Size	Access Time (ns)	Energy (J)	Leakage Pwr (W)
64	0.19	3.66E-12	2.92E-04
128	0.20	6.50E-12	5.37E-04
256	0.21	1.41E-11	1.07E-03
512	0.22	2.56E-11	2.14E-03
1024	0.26	4.16E-11	4.18E-03

Table 2.2: RGR Overhead

Multicast Overhead. The hardware overhead of a DOR-based multicast router has two parts. First, multiple RC units incurs an overhead to support lookahead routing. $3 \sim 4$ RC units per each input port are added to the baseline router. A multicast RC unit needs at most 59 OR gates for 64 destination nodes, so a router needs 944 OR gates which accumulate to the total area overhead of 0.40% per router based on DSENT [30]. Second, we adopted the mux-based crossbar that has been

used by several multicast routers such as RPM [21] and BAM [22], to avoid serialization delay of replication in the matrix-crossbar. The mux-based crossbar has been analyzed to consume more energy than the matrix-crossbar [23]. However, our mechanism can be built on energy-efficient crossbar, mXbar that supports single-cycle replication with small energy overhead or even better energy efficiency compared to the matrix-crossbar [23].

2.6 Related Work

There are previous studies for improving network bandwidth in GPU. Bakhoda et al. [5] proposed a cost-efficient checkerboard router design with multi-ported routers for MCs to increase MC network injection bandwidth, for many read replies. Jang et al. [6] introduced a bandwidth efficient network design for GPU traffic through VC monopolization and partitioning. Ziabari et al. [7] explored asymmetric NoC designs where the reply subnetwork is provided with larger channel width. However, ours differs because we directly reduce the heavy reply traffic exploiting data redundancy. Hsu et al. [31] proposed a packet coalescing mechanism, but this study is applied to request network to rearrange memory requests for enhancing row buffer hits in DRAM. It improves DRAM bandwidth but does not reduce traffic volume in reply network as our coalescing approach.

Reducing memory requests is a promising approach to increase network bandwidth, which eventually reduces reply network traffic. To alleviate high demand on global memory system, an intra-warp memory coalescer [17] and an inter-warp memory coalescer (Warppool) [27] were proposed, which has been compared to our technique. Jia et al. [28] proposed memory request prioritization method for effective caching, but it is limited to cache-sensitive applications. Dongdong et al proposed a DRAM scheduler exploiting inter-core locality to reduce memory access latency [11], which is orthogonal to our packet coalescing technique.

Data compression is the most relevant to our packet coalescing in that it directly reduces data size. Pekhimenko et al addressed a problem of increased dynamic energy caused by frequent communication switching of compressed data traffic [32]. To alleviate the off-chip memory bandwidth bottleneck, Sathish et al applied both lossless compression and lossy compression [33]. The data compression is complementary to our coalescing technique because ours reduce the number of

packets, while compression mechanism reduces the size of each packet.

GPGPU performance has been improved by novel warp scheduling policies. Narasiman et al proposed two-level scheduling that increases core utilization [34], and Jog et al proposed OWL scheduler that improves both L1 hit rate and DRAM bandwidth utilization [35]. As a homogeneous scheduling policy works across SMs, the inter-core locality patterns are maintained, so that the novel schedulers with our packet coalescing can synergetically improve performance.

2.7 Conclusions

In this paper, we identify that the performance of GPGPU applications is significantly impacted by MC bottlenecks near the MCs. To address this issue, we propose to reduce the traffic volume in the reply network from MCs to SMs by introducing PCUs in MCs. The key idea is to coalesce read reply packets in MCs when they deliver the same cache block to multiple SMs. To ensure that the coalesced packets arrive at the respective requesting SMs, we support multicast for the interconnection network. To the best of our knowledge, this is the first work showing a good use of multicast in GPGPUs. Our extensive evaluations across a wide range of benchmarks show that PCUs coupled with XY-YX routing obtain 15.5% AMAT reduction (up to 65.2%) and 13% network bandwidth savings (up to 67.8%) in a large-scale GPGPU with 2D mesh, and thus improve overall IPC by 15% (up to 112%) on average. Also, our coalescing approach achieves 7% IPC improvement in a GPGPU with the crossbar.

3. DUAL PATTERN COMPRESSION USING DATA-PREPROCESSING[4]

3.1 Introduction

GPUs have been extensively used in a variety of general purpose applications due to their tremendous computing power. Modern applications require an even more powerful computation capability to process a large volume of data with high throughput. The rising demands have been satisfied by the continuing development of GPUs. In fact, an NVIDIA Fermi GPU (GTX480) [17] released in 2010 started with 480 cores, and the recently released NVIDIA Titan-XP incorporates 3840 cores [1]. Even these GPUs are not sufficient for rapidly evolving AI applications that tackle large datasets [37], so multiple GPUs are often used together to facilitate faster processing [38]. Thus, it is essential to design a large-scale GPU with higher degrees of parallelism.

GPUs are designed to hide long memory access time by overlapping the computation time of active cores with the memory access time of idle cores. However, it has been well-known that long memory latency cannot be hidden due to limited resources in memory [39], cache [40] and network [5] [6]. As more cores are used, the network bandwidth becomes a more critical limiting factor [4] since the network is seriously overwhelmed by excessive memory requests. Most studies have optimized the use of network resources, adapting to unique communication patterns rather than reducing the volume of transferred data [5] [6] [7]. To enhance network bandwidth, it is necessary to devise a cost-effective approach that can fundamentally minimize network traffic.

Data compression is an essential approach for improving effective network bandwidth by reducing packet size (i.e. payload) before being sent through a network. Several compression schemes have been studied, but they entail limitations on their applicability to packet compression. Dictionary-based compression schemes encode data words into corresponding short codes in a dictionary [41–43]. By compacting frequently appearing data words, they obtain high compressibility but are not suitable for packet compression due to insufficient scalability, complex

[4]©2019 IEEE. Reprinted, with permission, from K. H. Kim, P. Devpura, A. Nayyar, A. Doolittle, K. Yum, and E. J. Kim, Dual pattern compression using data-preprocessing for large-scale gpu architectures, 09/2019 [36]

dictionary synchronization, and high latency overhead. On the other hand, pattern-based compression schemes encode data purely based on the occurrences of predefined patterns in the data words. Due to their simplicity, they are inherently amenable for packet compression by reducing latency overhead [39] [44] [45]. However, they lack compressibility for various data types (especially floating-point and character types) and extendability to new data patterns.

In this paper, we propose a compression scheme, called Dual Pattern Compression (DPC), that consists of data preprocessing modules and dual pattern encoders. We observe that value similarity naturally resides in the same bit-positions across meaningful data elements of input data. Our key idea for compression is to exploit this ***bit-level redundancy***. For higher compressibility, the proposed scheme starts by first utilizing data operation mechanisms, defined for three primitive data types, which manipulate input data to artificially create more bit-level redundancy. Then, the preprocessed data is rearranged through data remap mechanisms that create compressible patterns by exploiting bit-level redundancy. Finally, the dual pattern encoder compresses only two patterns (i.e. all ones or zeros) into a single bit. Our DPC shows consistently decent compressibility across various applications by supporting data remapping and data-type-specific preprocessing. The low latency and high compressibility of this scheme also give it the potential to be used in a variety of compression domains.

Our contributions in this paper are summarized as follows:

- We observe that bit-level redundancy is prevalent across various applications and propose a new dual-pattern compression mechanism that exploits the bit-level redundancy with low latency overhead and high compressibility.
- We propose data preprocessing mechanisms that can enhance compressibility by converting integer, floating-point and character data into new format with sufficient data redundancy.
- We address the severe network bottleneck problem of a large-scale GPU by compressing packets with the proposed schemes, thereby achieving an IPC improvement of 33% (up to 126%) on average across various benchmarks.
- To the best of our knowledge, this is the first work that attempts to compress floating-point

and character data types in a pattern-based compression approach. We achieved a 49%∼72% space-savings ratio in real-world floating-point datasets and a 32%∼57% ratio in text datasets.

The remainder of this paper is organized as follows. In Section 3.2, we present the background of compression techniques and describe the overview of our approach. Section 3.3 discusses a basic compression scheme composed of a data remap function and a dual pattern encoder, and we enhance the scheme by adding data operations for three data types in Section 3.4. Section 3.5 describes our evaluation methodology. We analyze the performance results in detail in Section 3.6. Finally, we draw conclusions in Section 3.7.

3.2 Background and Approach

This section starts by summarizing prior study on hardware-based data compression. Then, we motivate our research and describe our overall approach.

3.2.1 Hardware-Based Compression

Data compression algorithms have been applied in various domains: all levels in the memory hierarchy, Network-On-Chip (NoC) and memory links. Several studies for data compression have been conducted in GPU architecture for register file compression [46], cache compression [40] and off-chip memory interface [39] [33]. Unlike these previous studies, we will explore the role of data compression on NoC performance in GPUs.

Packet compression is a cost-effective approach for providing fast and efficient data transfer on a NoC. Packet compression has been well-studied in Chip-Multiprocessors (CMPs) to achieve network latency reduction and power savings in NoCs. Das et al. showed a packet compression based on static compression patterns [47]. Jin et al. proposed a scalable dictionary-based compression scheme that compresses dynamic redundancy patterns [48]. In contrast, the importance of packet compression for advancing NoC architecture of next generation GPUs has not been studied well. Recently, Kim et al. proposed a packet coalescing mechanism to reduce packets with inter-core locality in GPUs [4], but data compression is orthogonal to this approach.

3.2.2 Compression Algorithms and Motivation

Several data compression algorithms have been proposed under two major approaches: dictionary-based and pattern-based. Now we describe the algorithms and discuss their restrictions on packet compression. Later, we compare the key features and target domains of the compressors in Table 3.1.

First, the dictionary-based approach maintains frequent data values in a dictionary table and encodes the values with short codes. C-pack [43] leverages a dictionary dynamically updated with the most frequent values. SC^2 [41] maintains a dictionary where common values are associated with variable-length codes according to the degree of their occurrence. FPH [42] is designed for compressing 64-bit double-type data by referring to a dictionary with Huffman codes for repetitive exponent and mantissa fields. DISH [49] effectively compacts contiguous cache blocks by sharing a dictionary among them for a compressed cache. In this approach, the compressibility is a primary design factor at the consequence of high de/compression latency overhead.

However, the dictionary-based compressors have fundamental limitations inappropriate for packet compression in three aspects.

- A dictionary-based compressor is not a scalable solution for packet compression. In packet compression, all N nodes in a network compress packets they want to send. Each receiver node decompresses the packets coming from other *N-1* nodes. To correctly restore them, a decompressor in each receiver node needs to maintain *N-1* dictionaries.
- Synchronizing dictionaries between a compressor and a decompressor requires an expensive hardware cost. The bulk data transfer for a dictionary created at the training phase is necessary from a compressor to a decompressor [41] [42] or synchronization protocols should be introduced for dynamic dictionary update [43] [48].
- The hardware dependency on a dictionary in compressor/decompressor inherently causes serialized process for each input. In particular, a compressor with long latency overhead adversely becomes a bottleneck for next waiting packets in a highly dense network.

Second, the pattern-based compression approach encodes data matching predefined static pat-

terns into a compressed format with low latency and decent compressibility. FPC [45] compresses data with common value patterns including frequent zeros. BDI [44] observed that high value locality exists among neighboring data, so it compresses them by keeping their difference (called delta-value) against common base values. BPC [39] developed DBX transformation that increases the run-length of repeated zeros and adopted run-length encoding for compressing data.

The pattern-based compressors are more amenable to packet compression, but the following limitations prevent them from being an effective solution for various applications.

- They mainly work well with integer or image data, but their compressibility on floating-point data is limited.
- Despite the rising importance of text processing applications in the area of bioinformatics and data mining, none of the compressors deal with character data.
- The complexity of compressors for gaining high compressibility (e.g. BPC) weaken their applicability in latency-sensitive packet compression.

Compressors		Latency	Comp. Ratio	Data Type			Applied Domain
				Int	FP	Ch	
Dict-based	C-pack	high	high	O	X	X	CC/MLC
	SC	high	high	O	X	X	CC
	FPH	high	high	X	O	X	CC
Pattern-based	FPC	mid	mid	O	X	X	CC/PC
	BDI	low	mid	O	X	X	CC
	BPC	high	high	O	X	X	MLC
	DPC	low/mid	high	O	O	O	PC

Table 3.1: Qualitative Comparisons of Compressors (CC, MLC, PC: Cache, Memory-Link, Packet Compression)

3.2.3 Approach

Our objective is to design a practical compression scheme that can compress multiple types of data with low latency overhead. Figure 3.1 shows an overview of the proposed compression approach. The compression phase consists of a two-level data preprocessing module and dual pattern

encoders. Unlike data compressors that usually exploit data redundancy inherent in data only, we introduce data operation functions that first manipulate input data into a new form to artificially create more data-type-specific data redundancy. The data type is identified during runtime by hardware. Then, data remap functions reorganize the manipulated data into compression-friendly patterns by leveraging bit-level redundancy. Finally, our proposed encoder compacts the preprocessed data by finding the occurrence of two simple patterns. The simplified patterns minimize the latency overhead of de/compression. Similarly, the decompression phase first decodes the encoded data and then restores the decompressed data into the original form in the postprocessing phase.

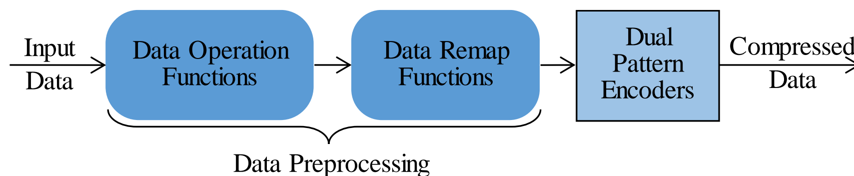


Figure 3.1: Compression Flow with Data Preprocessing and Encoding

3.3 Compression Mechanisms

We first present the data preprocessing mechanism using natural data redundancy, and then we present a simple compression algorithm.

3.3.1 Preprocessing Using Natural Data Redundancy

Data compression is a technique that encodes data into a smaller format by reducing data redundancy. We briefly summarize the common data redundancy according to data types in Table 3.2. First, for an integer type, *zeros* and *repeated values* are widely present in the variables initialized by a program. *Narrow values* with frequent zeros in high-order bits also often appear when variables in a program store a value smaller than the size of a data type. Second, for a floating-point type, redundancy exists in the sign and exponent fields across neighboring floating-point variables when they have similar magnitude with the same sign. *Pixel values* are stored as either an integer

or a floating-point type, and they tend have redundancy due to their color closeness. Third, for a character type, the bit-level redundancy is present across characters. For instance, lower case letters in ASCII are coded from 61_{16} to $7a_{16}$, so their highest three bits are consistently same as 011_2. We call the inherent redundancies caused by common program behaviors or data properties ***natural data redundancy*** in this paper.

Type	Redundant Data Patterns
Integer	Zeros, Repeated values, Narrow values, Pixel values
Floating-point	Values with similar magnitude, Pixel values
Character	Consecutive lowercases or uppercases

Table 3.2: Summary of Natural Data Redundancy

We observe that the existing pattern-based compression algorithms do not adequately eliminate the natural data redundancy in their compressed format. Figure 3.2 illustrates the compressed data by BDI and FPC for 32-byte (32B) pixel values (eight 4B elements) taken from an application, Heart Wall Tracking (R.HW). For simplicity, the examples do not show the encoding pattern type bits. BDI takes two bases: an explicit one (4270_{16}) and an implicit one (0000_{16}). Then it obtains two 1B-delta values for each element by subtracting the two bases from high and low 2B, respectively. It compresses the data by keeping the explicit base and the delta values, and we can observe that *zeros* redundancy still exists in the 1B-deltas computed by the implicit base. Unlike BDI, FPC successfully eliminates the *zeros* by using its supporting pattern (prefix 100_2) [45]. However, the same byte (42_{16}), classified as *Pixel values* redundancy, remains in the compressed data. In both examples, 50% of compressed data are still redundant. To gain better compressibility, it is desirable to fully exploit the remaining data redundancy for further compression.

Preprocessing. To exploit all possible redundancy in compression, we attempt to preprocess input data through a ***data remap function***. We observe that input data is often composed of data elements with a homogeneous natural redundancy type and is within a similar value range. These data elements tend to have an identical value in the same bit position, called ***bit-level redundancy***

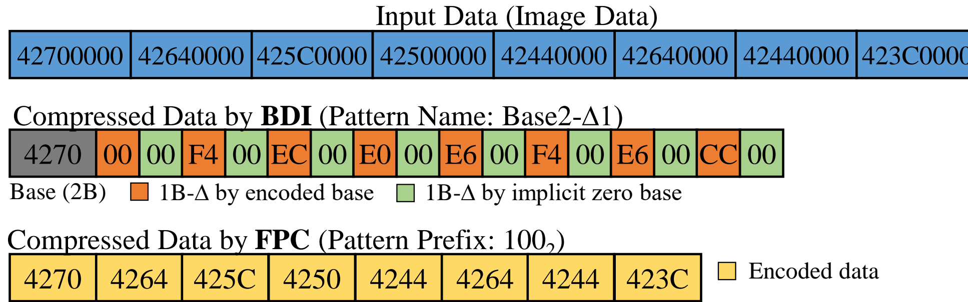


Figure 3.2: Illustration of Data Redundancy Remaining in Compressed Data by BDI and FPC

in this paper. The sequence of bit-values at the same bit position is called a bit-plane in the image processing domain, which has been widely used for compressing an image in a lossy or lossless way [50]. This approach has been taken in an architectural compressor, BPC through DBX transformation. Unlike these approaches, we found that the values on the bit-planes are directly exploitable for compression because consecutive zeros or ones are prevalent in the bit-planes, taking 37% of a cache block on average.

The data remap function statically reorganizes input data as a sequence of the segments that group each bit at the same bit position across different data elements of a cache block. Therefore, we create segments with **two compressible patterns, consecutive zeros or ones**, which will be shortened by our proposed compression algorithms. Figure 3.3 illustrates an example of a data remap function that rearranges the input data of Figure 3.2. For simplicity, we show an example of the data remapping for an input data of 32B. The data remap function creates a byte by grouping the bit-values at the position from 1 to 32 individually. For instance, the most-significant-bits (MSBs) from eight elements form 00_{16}. The final remapped data have the total compressible data of 27B. As a result, unlike BDI and FPC, all redundancies can be considered for compression due to the data remap function.

Postprocessing. The rearranged bits are restored into an original form in the postprocessing. The remap function relocates each bit j in each bit-plane i to the ith bit-location of the jth element.

3.3.2 Compression/Decompression Algorithms

Our dual pattern compression algorithm takes remapped data (128B) as an input, decomposes it into 32 segments (4B granularity), and compresses it on a segment-by-segment basis. A segment is compressed into an encoded data and an encoding status bit. The encoded data stores a single bit, zero or one, if a segment matches consecutive zeros or ones ($00..00_{16}$ or $ff..ff_{16}$), respectively. Otherwise, it stores the original 4B value of the segment. The encoding status bit records if each segment is compressed or not by storing one or zero, respectively. For a given input data, the algorithm produces a compressed output represented as a compression flag (C), a sequence of encoding status (ES), and a sequence of encoded data (ED). The compression flag is used to distinguish if the output is compressed (1) or not (0). A compression example for the remapped data in the previous section is depicted in Figure 3.3.

The decompression algorithm is also straightforward. If the compression flag of input data is set to zero, it produces the remaining data without the flag as an output. Otherwise, it starts decompressing a segment by checking the corresponding encoding status bit. If the encoding status is one, a segment is recovered as 4B consecutive zeros or ones according to its encoded data (zero or one). Otherwise, the segment is restored as its 4B encoded data. All the segments are restored in the same manner. The restored data is converted to the original data after postprocessing.

3.4 Data Operation Mechanisms

The compressor assisted by the data remap function only shows limited compressibility when the datasets have low natural data redundancy. To consistently gain high compressibility across diverse applications, we extend the scheme by introducing the data operation mechanisms for three data types.

3.4.1 Floating-Point Data

The common data redundancy in floating-point datasets is rarely found except for two representative datasets with image/video data and a small set of redundant floating-point data. Our proposed DPC algorithm, as already discussed, effectively compresses image/video data by exploiting their

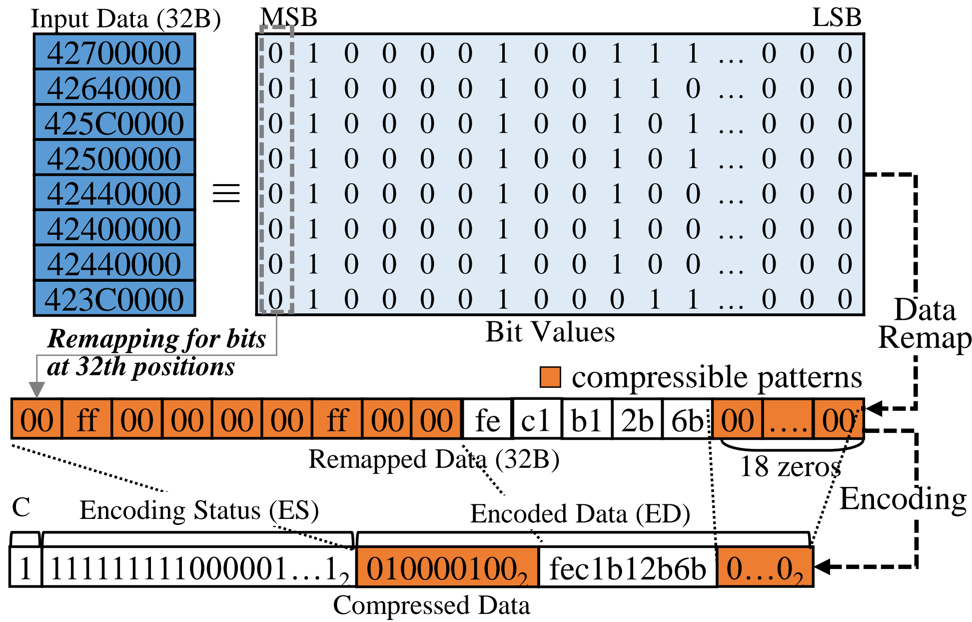


Figure 3.3: Example of Data Remap Function and Compression

natural data redundancy. FPH deals with the datasets where a small set of floating data repeat by using Huffmann-based dictionary [42], although it is not suitable for latency-sensitive applications due to its long latency (See our evaluation).

Unlike these datasets, we observe that the ***short floating-point (SFP) values*** with a small integer and a fraction with limited digits often appear in the real-world datasets of data mining and finance applications (e.g. 12.4 meters and $127.25). It is because real-world measurement data (e.g. speed, weight, length, area) and finance data (e.g. currency) do not need full precision. To compress such floating-point data with high compressibility, lossy compression is taken as an alternative [33], but it is highly risky for finance data because a minor error can cause catastrophic financial loss. Thus, it is essential to find a way of compressing SFP values losslessly.

Preprocessing. The data redundancy among SFPs is seldom exhibited under IEEE-754 standard. To create compressible patterns, we introduce a ***floating-point representation conversion function*** that converts an IEEE-754 standard into a new representation composed of a 1-bit sign field (s), a 23-bit integer part (di) based on a binary code, and a 8-bit fractional part (df) based

on a binary-coded-decimal (BCD) code. This representation helps to create frequent zeros in the integer part, while packing the fractional part losslessly. For example, 12.40 and 127.25, which are encoded as 41466666_{16} and $42fe851f_{16}$ under IEEE-754 representation, are transformed into $00000C.40$ and $00007f.25$ which now become a good target for our compressor.

The representation conversion is carried out by decomposing an IEEE-754 input data into a sign bit, a binary integer and a binary fraction after denormalization, encoding the binary fraction to BCD, and finally concatenating the sign bit, the binary integer and the BCD fraction. However, it is challenging to design the encoding hardware from a binary fraction to a BCD fraction with low latency since it requires a sequential process. In the previous example, a binary fraction $0.0110011001100110011 0_2$ should be transformed to be 0.40_{16}. Thus, we exploit a lookup table where 10-bit binary values are associated with the respective BCD code. Our exhaustive analysis for all numbers in the scope of SFP shows that the high 10-bits in the binary fraction after denormalization are sufficient to distinguish all different cases. The preprocessed data created by this conversion is considered for compression if an input data stores a SFP value and is ignored, otherwise (e.g. very large $di \geq 2^{24}$).

Runtime SFP Detection. Another important challenge is how to detect if an input data stores a SFP value. One way is to rely on programmers' annotation on SFP-typed data, which can weaken practicality. Thus, we propose a runtime SFP detection mechanism that examines if the SFP format obtained after the representation conversion can be recoverable to an original IEEE-754 input data by testing the validity of integer and fraction parts.

The SFP detection mechanism works as follows. First, it checks if an IEEE-754 input data has a small integer value enough to be encoded in an integer part (di) of a SFP format. A SFP-typed data stores a value from 120 to 140 in the exponent field (e), which corresponds to the exponent range of SFP values from the minimum (0.01) to the maximum (9999.99). Second, the mechanism verifies if the BCD fraction (df) of a SFP format recovers the binary fraction decomposed from an IEEE-754 input data. The binary fraction of a SFP-typed data is exhibited as a substream of the binary fraction reversely encoded from the BCD fraction. The algorithm for encoding the BCD

fraction (i.e. decimal fraction) to a binary one is straightforward [51] but requires a nontrivial latency overhead in its directly mapped hardware. To address this issue, the detector also uses the lookup table that associates a BCD code with a pregenerated 31-bit stream, which is necessary for the worst case (i.e. 0.01) that has a 31-bit binary fraction after its IEEE-754 format is decomposed.

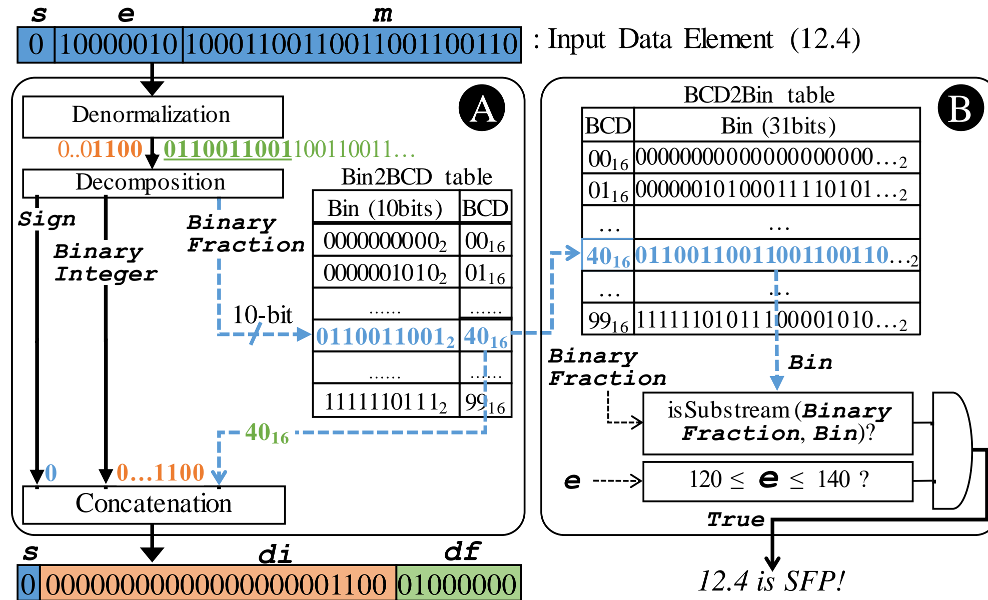


Figure 3.4: Data Operation for Short Floating-Point Data (e.g. 12.4)

Figure 3.4 illustrates the detailed steps of our conversion function and SFP detection with an example of 12.4 (41466666_{16} in IEEE-754 format). First, in the conversion process (Ⓐ), an input data is separated into a binary integer part (orange color) and a binary fraction part (green color) after denormalization. The BCD-code, 40_{16} is obtained by searching high 10 bits of the binary fraction (0110011001_2) in Bin2BCD table. Finally, the sign bit (0), the binary integer part ($00..1100_2$) and the BCD fraction part (01000000_2) are concatenated to create a new representation. Second, in the detection process (Ⓑ), the exponent field of an input (10000010_2) is confirmed to be between 120 and 140. Also, the 31-bit stream in BCD2Bin table associated with the BCD-code, 40_{16} is verified to include a binary fraction part (green color).

Postprocessing. We restore the decimal floating-point representation to IEEE-754 format without loss. First, the function separates the integer value and fraction BCD based on their fixed positions. Second, the function converts the BCD into binary format, which follows the same process described in the SFP detection process. Third, the binary value concatenating the integer and fraction part is normalized to IEEE-754 format.

3.4.2 Character Data

Semantically meaningful character words (e.g. English words) have variable lengths unlike numerical data with a fixed size per type. While software-level text compressors readily compress them due to abundant resource and looser latency constraint, it is not trivial to design an architectural compressor for the variable-length words. Our survey shows that a good range of modern applications such as bioinformatics and data mining often use text datasets such as a small-character style (e.g. DNA and Protein data) and a word-based style (i.e. bag-of-word), while others use a sentence-based style (e.g. natural language). In this paper, we aim to design an architectural compressor mainly targeting the first two types.

Preprocessing. Our compression approach for character data exploits the bit-level redundancy across 1B characters. A bag-of-word style datasets usually store a sequence of words with lowercase characters. The bit-level redundancy across them naturally exist in the three highest bits, since ASCII table assigns sequential code for them. However, since control characters such as null (00_{16}), tab(09_{16}), line feed ($0a_{16}$), carriage return ($0d_{16}$) and white space (20_{16}) often coexist between the words so that it interferes with creating the bit-level redundancy.

To seamlessly create the bit-level redundancy, we introduce a ***code conversion function*** that converts the frequently used interfering code into the rarely used lowercase and vice versa. We define a static code table with five one-to-one mappings from the interfering codes to (60_{16}, $7b_{16}$, $7c_{16}$, $7d_{16}$, $7e_{16}$), respectively. Also, five reverse mappings are added to allow reciprocal transformation for avoiding the aliasing between transformed data and same real data. For instance, real data ’00’ is transformed to ‘60’ code, and real data ‘60’ is transformed to ’00’ code. If a given input character has no matching, the function skips the conversion by directly using the input character

as an output. A detailed example about the code conversion as well as the data remapping for a data from String Match (M.SM), “tee\r\n\0te” is illustrated in Figure 3.5.

Postprocessing. The postprocessing function for character data maintains the same code table shown in Figure 3.5. Thus, it restores an original data by performing the code conversion operation in the same manner.

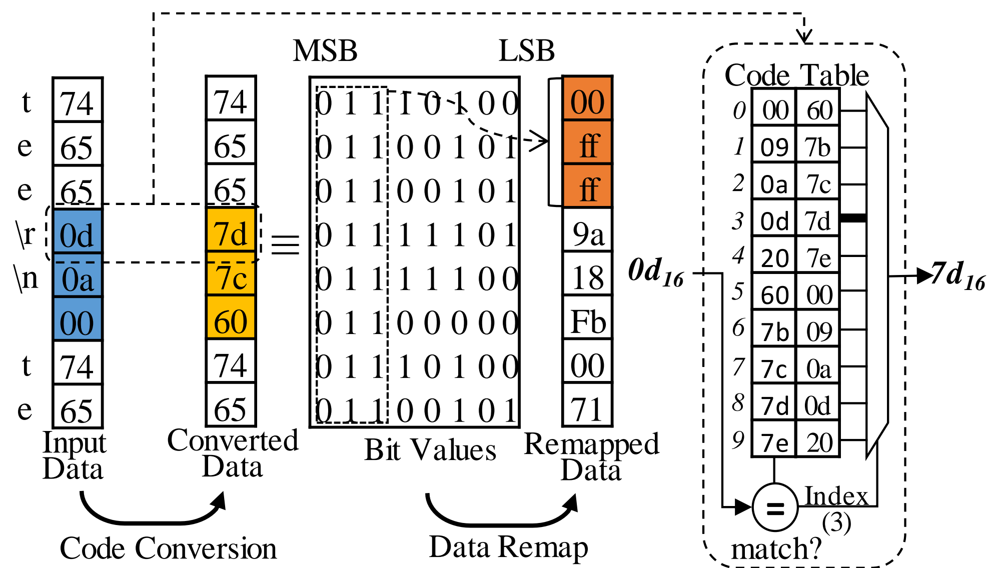


Figure 3.5: Example of Data Preprocessing for Character Data

Dynamic Code Conversion. When text datasets use a small number of alphabets only, the code conversion function replaces the alphabets with sequential codes to increase the bit-level redundancy. For instance, in a DNA sequence dataset, A (41_{16}), C (43_{16}), G (47_{16}), and T (54_{16}) are transformed into 01_{16} to 04_{16}, respectively. Then, as every character has zeros in the five higher-order bits, 62% of an input cache block becomes compressible.

To support this conversion, we introduce a runtime code detector. The detector examines M input characters (e.g. 1024 characters) on execution of an application and stores unique characters in a dynamic code table that stores the mappings from characters to sequential codes and their reverse mappings, similarly to a static table in Figure 3.5. If the number of collected characters is

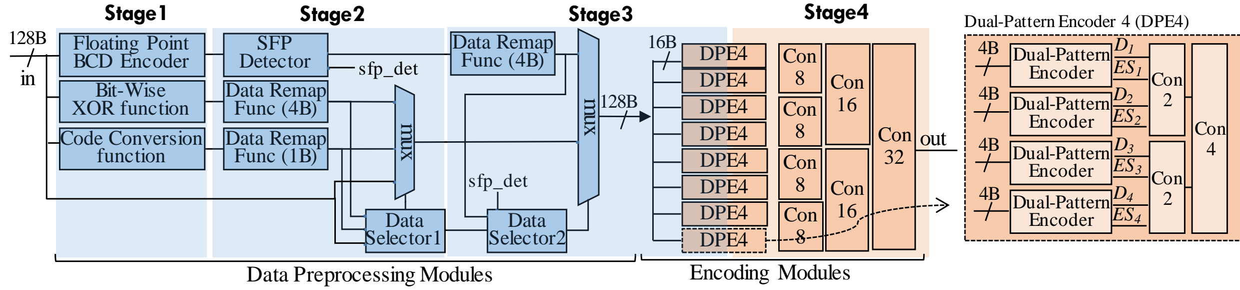


Figure 3.6: Compression Pipeline for a 128B Cache Block

not more than the maximum number of characters (e.g. 8), the code conversion function switches to use the dynamic table and otherwise, the static table.

3.4.3 Integer Data

Applications often use negative integers with small magnitude that have frequent leading ones due to their two's complement representations. When they are coexist with positive integers in a cache block, our data remap function cannot create compressible patterns (i.e. consecutive zeros or ones). We address this problem by introducing ***bitwise exclusive-OR (XOR) function*** that transforms a negative integer to have frequent zeros, while keeping a positive integer having its leading zeros as it is. Given an integer (bit_{31}, .. , bit_0), the function in the preprocessing side creates a new integer (bit'_{31}, .. bit'_0) by performing $bit_{i+1} \bigoplus bit_i$ for a new bit'_i and directly using bit_{31} as a new MSB (bit'_{31}). Reversely, the postprocessing function recovers the preprocessed data element back into an original one by converting an input bit_i to $bit'_i \bigoplus$ recovered bit'_{i+1} and using bit'_{31} to bit_{31}.

3.4.4 Data Selector

So far we have discussed three data operation functions per data type that preprocesses an input data to have more bit-level redundancy. To allow our dual pattern encoders to proceed compression, data selector chooses one of different preprocessed data or an original data by verifying which one has the most compressible patterns. In this manner, the integer and character type data are implicitly classified. As discussed in Section 3.4.1, the preprocessed SFP data is considered by

the data selector only when an input data is verified as SFP type by SFP detector. An original input data can have more compressible patterns, although the remapped data often does in most benchmarks. For instance, a benchmark, R.KM, uses a dataset with frequent zeros and a few non-zeros. The ones in the non-zeros hinder the data remap function from creating compressible patterns, while the zeros in the original data is directly compressible. While the selected data is compressed by dual pattern encoders, its data type is encoded in the header of a compressed data to help the postprocessing side to recover the original data accurately.

3.4.5 Hardware Design and Cost

We implement our DPC compression and decompression schemes as a fully-pipelined module. Figure 3.6 illustrates a four-stage compression pipeline that compresses a 128B cache block. The data preprocessing functions are designed through the first two stages, as discussed in Section 3.3.1, 3.4.1, 3.4.2, 3.4.3. The dual pattern compression is implemented in the next two stages. 32 dual pattern encoders compact the respective 4B data segment of the preprocessed data in parallel as described in Section 3.3.2. The encoded outputs (i.e. status and data) are concatenated hierarchically. Due to the simplicity of dual patterns, the concatenation logic is implemented with low latency overhead.

We synthesize our designs using Synopsys Design Vision [52] with 45nm TSMC standard cell library [53] at 1.4 GHz. A 35% margin of the clock period is used to model uncertainties and wire-load delays. DPC has 2 cycles for the compression: one cycle for both dual-pattern encoding and data remapping part and one cycle for concatenation logic. DA-DPC extends DPC with preprocessing logic and a data selector, which needs two more cycles, so its total compression latency is 4 cycles. The decompression latencies for DPC and DA-DPC are 2 and 3 cycles, respectively. The overall area for DPC and DA-DPC compressors is 43250 μm^2 and 118749 μm^2 respectively, which are roughly equivalent to 60K and 164K 2-input NAND gates, respectively. The area overhead of the runtime code detector is 9339 μm^2. 8 compressors and 56 decompressors are integrated, so the area overhead caused by DPC and DA-DPC versions is just 0.07% and 0.3% against the overall GPU area [54], respectively.

3.5 Evaluation Methodology

Since compression is a fundamental technique that is not architecture-dependent, it can be applied to diverse domains. We choose the packet compression on a NoC of a GPU as a case study to show the effectiveness of a DPC-like compressor that strikes a good balance between compression latency and compressibility. Moreover, using packet compressors/decompressors in a network interface (NI) as a plugin module is practical. We first describe the reason why the packet compression is crucial in a GPU. Then we explain how to integrate compressors with the GPU. Finally we detail our evaluation methodologies.

3.5.1 Network Bottleneck in a GPU

A GPU is composed of streaming multiprocessors (SMs) and MCs that are connected with each other through a NoC. For effective memory accesses, each SM uses cache memories: L1 data cache (L1D), constant cache (L1C), texture cache (L1T) as well as a shared memory. Each MC is also coupled with an L2 cache. For the NoC, we use a 2D mesh topology due to its scalability, simplicity and regularity [4–6]. To prevent a protocol deadlock, we build a single network with two separate virtual channels (VCs) for the request network from SMs to MCs and the reply network from MCs to SMs. The detailed configurations are summarized in Table 4.1.

A large-scale GPU system has been designed in two directions collaboratively. Multi-GPUs (e.g. Tesla-P100) are utilized as a good time-to-market strategy keeping up with fast growing emerging applications. The scaling up for a single GPU has been also performed. The initial GPU model, NVIDIA Fermi with 15 SMs has been evolved as the most recent NVIDIA Titan-XP with 30 SMs. However, the current GPU relies on a crossbar with a fundamental limit on scalability, and thus how to design a scalable and practical interconnect for a large-scale GPU is still an open question.

As a key research problem, we observe that a severe reply network congestion takes place in a large-scale GPU due to a heavy volume of packets as studied in prior work [4–6]. Figure 3.7 shows MC stall time ratio, the ratio of the time that MCs stall out of the total execution time across 30

System Parameters	Details
Shader Core	56 SMs, 1.4Ghz, GTO Scheduler
L1 Cache	L1I(2KB), L1D(16KB), L1T(12KB), L1C(8KB)
L2 Cache	1024KB
Interconnect	8 x 8 Mesh, 1.4Ghz, 2-Cycle Router
Virtual Channel	4 VCs per Port (8-Flit Buffer)
Routing	Dimension-Order Routing (XY)
Flow Control	Wormhole, Credit-based
Channel Width	128 bits
MC Placement	Diamond [5]
Memory Model	8 MCs, 924 Mhz, FR-FCFS scheduling
Min L2, DRAM latency	120, 220 cycles

Table 3.3: GPU Configuration Parameters

benchmarks (see details about benchmarks in Section 3.5.3). We measure the MC stall time when MCs cannot inject new reply packets due to nearby NI input buffers being full. MCs stall 47% out of the entire simulation time on average across 30 benchmarks. This problem is caused by the overwhelming memory requests from many SMs. We also observe 32% of MC stall ratio even in NVIDIA Fermi with a crossbar (See Section 3.6.5). Therefore, we apply the packet compression to the reply network as a fundamental solution that mitigates the network congestion regardless of underlying NoC topologies.

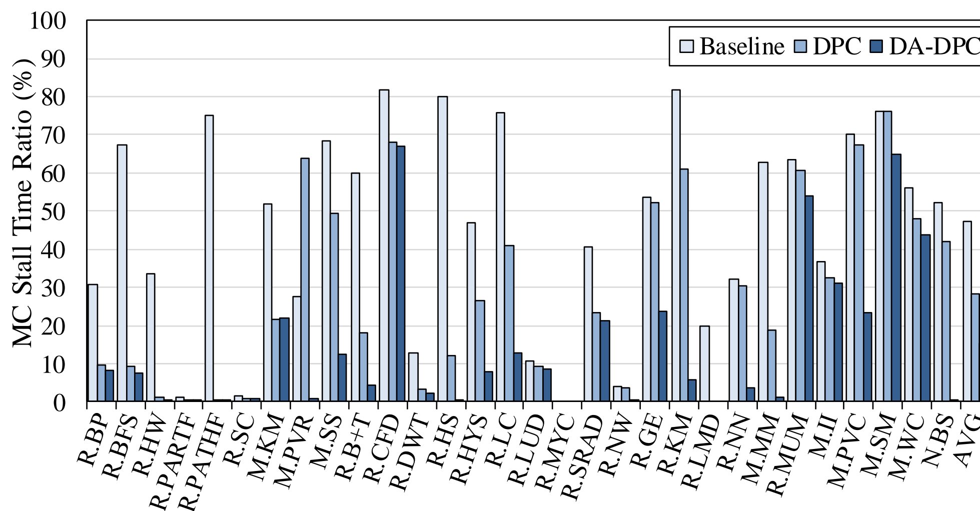


Figure 3.7: Analysis of Network Bottleneck in MCs using MC Stall Time Ratio

3.5.2 GPU with Packet Compressors

Figure 3.8 illustrates a GPU architecture where compressors and decompressors are integrated into an MC NI and an SM NI, respectively. SMs access data from memory through four different types of memory requests depending on their associated cache memories: global, local, texture, constant memory. The packet compression only targets the global and texture accesses which represent the majority of reply memory traffic. To hide compression latency overhead, a compression queue is introduced after a compressor as illustrated in Figure 3.8. The MC reply queue is set to hold eight replies in the baseline. To keep the same area budget, the MC reply queue and the compression queue are set to store four replies, each. The decompression side is also designed in the same manner. Other replies, such as incompressible write replies and instruction replies, bypass both compression and decompression pipelines to obviate unnecessary latency overhead.

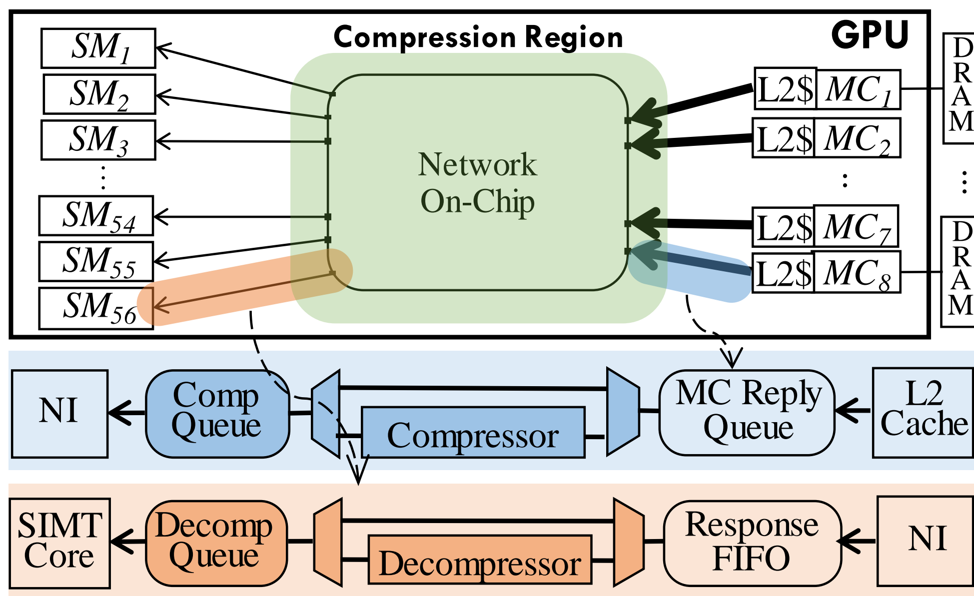
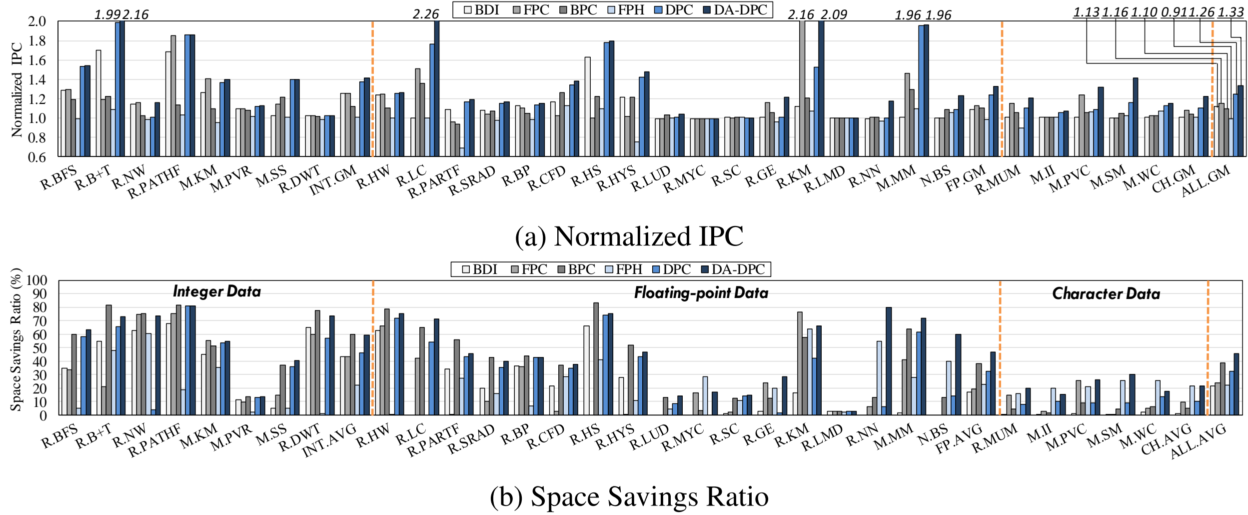


Figure 3.8: Large-Scale GPU Architecture with Compressor Integrated in NI of Reply Network



(a) Normalized IPC

(b) Space Savings Ratio

Figure 3.9: Comparison of IPC Performance and Compressibility

3.5.3 Methodology

Our DPC scheme is implemented in a cycle-accurate GPU simulator, GPGPU-Sim 3.2.2 [12]. We evaluated two versions of our compressors: ***DPC*** that uses a data remap function and dual pattern encoders only (Section 3.3), and data-type-aware-DPC (***DA-DPC***) that integrates three data operation functions on top of *DPC* (Section 3.4).

We compare our compression techniques to four closely related compressors: BDI, FPC, BPC and FPH. We simulated them based on the latency overhead reported in their literature. The compression/decompression latencies of BDI, FPC, BPC and FPH are 2/1, 3/5, 7/11 and 8/20 cycles, respectively. Note that the latency overhead of BPC was obtained at 800 Mhz, but we optimistically take it even though a GPU operates at 1.4 GHz. We also optimistically measure the compression latency for FPH based on CACTI 7.0 [26], excluding the concatenation time. The statistics about repeated words are collected over 20M instructions. The mantissa field is divided into 10-bit and 13-bit for Mantissa-High and Mantissa-Low, respectively and VFT tables of 256 entries are used.

We evaluate a complete set of CUDA applications from the Rodinia [13] and Mars [14] benchmark suite, and Black Schole (N.BS) from CUDA SDK [55]. We abbreviate the benchmarks of Rodinia and Mars by appending their respective prefix R and M to the original abbreviations used

in their literature. We measure the system performance as instruction per cycle (IPC) and the compression performance as the space savings ratio defined as $(1 - \frac{\text{compressed size}}{\text{uncompressed size}}) \times 100$. To collect the performance results, we run the benchmarks to the end or until they reach one billion instructions.

To evaluate the compression performance, datasets should be carefully selected. We first use the datasets of the benchmark suites. To get a deeper insight about the compressibility, we also use real-world datasets from a variety of fields: floating-point datasets (Ecoli/Hepatitis/Iris [56], Enb2012 [57], Forest [58], Nsl-kdd [59] and Sales [60]) and text datasets (Amazon/Nysk/NIPS [56], Tweet [61], Email [62], SentAnal [63], Ngrams [64], URL [65], Proteins/DNA [66], Para [67] and Chr1 [68]).

3.6 Evaluation

We evaluate the effects of six compression schemes on IPC performance and their compression performance in a large-scale GPU using 2DMesh. Then, we analyze the effectiveness of our data-type-aware preprocessing and the energy-savings in a NoC achieved by our scheme. Last, we analyze the effectiveness of our packet compression in GPU with a crossbar.

3.6.1 Effect on IPC Performance

Figure 3.9 (a) shows the normalized IPC when different compressors are used. Each IPC value is normalized against the baseline with no packet compression. We make three conclusions.

First, a full-fledged compressor, DA-DPC achieves a noticeable improvement by 34% on average compared to other compressors, DPC, BDI, FPC and BPC that obtain 26%, 13%, 16% and 11%, respectively. The improvement of DA-DPC is correlated to a good balance between its highest average space savings ratio, 47% and its low latency overhead. The compressed packets result in directly reducing in-flight flits and eventually mitigate the network bottleneck, which is supported by a MC stall time reduction in Figure 3.7. The average MC stall time ratio is reduced from 47% in the baseline to 25% and 16% with DPC and DA-DPC, respectively.

Second, our light-version compressor, DPC achieves higher IPC performance than BDI and FPC; The IPC improvements of DPC, BDI and FPC are 25%, 12% and 15%, respectively. It is

because DPC exhibits high compressibility across broader applications consistently due to the data remap function, while BDI and FPC show biased good compressibility toward some applications. We also observe that DPC is even better than the hybrid compressor (BDI+FPC) choosing the best performing one between BDI and FPC per benchmark that achieves 20% IPC improvement. This result implies that DPC with low latency overhead is a good candidate for latency-sensitive domains where either BDI or FPC are often adopted.

Third, BPC and FPH are not suitable for packet compression due to the latency overhead caused by complex compressor design. BPC, as reported, gains higher space savings ratio than BDI and FPC but shows worse performance improvement. FPH stands in stark contrast to other compressors by adversely degrading the overall performance by 9% due to its low space savings ratio at the expense of high latency overhead.

3.6.2 Compression Performance Analysis

Now we discuss the detailed compression performance per data type. Figure 3.9 (b) compares the space savings ratios among the evaluated compressors across all benchmarks. We categorize the benchmarks into three groups according to their dominant data types. The average space savings ratio for each group is plotted as well as the average ratio for all cases. In this analysis, we make three conclusions.

First, DA-DPC and BPC achieve the highest space savings ratio in integer data oriented benchmarks, around 60%, while DPC, BDI, FPC and FPH achieve 46%, 43%, 43% and 22%, respectively. DA-DPC also achieves high compressibility, and its improvement from DPC is due to preprocessing for negative integers. For instance, DPC achieves 4% space savings ratio in R.NW, while DA-DPC does 74%. BPC shows comparable compressibility, but it does not effectively improve IPC performance due to its overhead as discussed in Section 3.6.1.

Second, DA-DPC outperforms other compressors in floating-point data oriented benchmarks by achieving, 47% space savings ratio on average, while BPC, DPC, FPH, FPC, and BDI achieve 38%, 33%, 23%, 19%, and 17%, respectively. Six benchmarks, R.LC, R.GE, R.KM, R.LMD, R.NN and R.MM have SFP data. For example, the space savings ratios for R.NN and R.GE are

improved from 6% to 80% and from 2% to 29%, respectively by SFP preprocessing. In R.KM, FPC makes an exception by outperforming DA-DPC by 6% since the dataset (KDD_CUP) has highly skewed zeros. FPH, originally designed for floating-point data, shows high performance for benchmarks using dominant floating-point data as expected, but the benchmarks rarely gain IPC improvement due to the latency overhead of FPH.

Third, DA-DPC and FPH are the most effective in character oriented benchmarks. They achieve an average space savings ratio 22%, while BDI, FPC, BPC and DPC do 1%, 10%, 5%, and 10%, respectively. Five benchmarks, R.MUM, M.II, M.PVC, M.SM and M.WC have the memory requests about character data by 16%, 95%, 76%, 99% and 91% of all compressible memory requests, respectively. Their space savings ratios for character data only are 52%, 14%, 16%, 26% and 13%, respectively. R.MUM achieves better performance due to its dataset using a small number of alphabets. Interestingly, FPH, designed for floating-point, compresses character data quite well since the value locality of character data is exhibited in the locations of float-point subfields.

3.6.3 Impact of Preprocessing on Compression

To get more insights on the impact of our preprocessing operations, we evaluate the compression performance with various real-world and synthetic datasets involving short floating-point and character data. The six-different synthetic datasets of SFP values are denoted as I*x*F*y* where x and y are the number of digits in the integer and fraction parts, respectively.

Figure 3.10 shows that DA-DPC achieves 59% space savings ratio on average in R.KM (Kmeans) in all groups of data, while FPH and BPC are limited to 45% and 19%, respectively. For the short floating-point patterns, DA-DPC is more effective due to its low encoding overhead. FPH works effectively when a small number of floating-point data values frequently appear. We have optimistically evaluated FPH by excluding its zero-compression ratio during the training phase. DPC achieves 12% space savings ratio on average since it compresses the natural redundancy of the exponent field only. BDI and FPC also show low space savings ratios, 1% and 13%, respectively.

Figure 3.11 shows the space savings ratio in M.SM using 12 datasets. The compression performance is characterized by three groups of datasets with different alphabet sizes as follows. First,

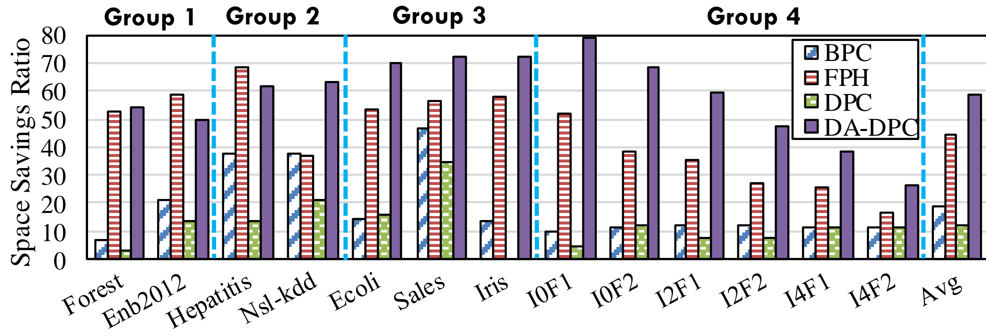


Figure 3.10: Effects of Floating-Point Data Operation in R.KM With 7 Real-World Text Datasets and 6 Synthetic Datasets

DA-DPC with the runtime code detection (See +CodeDet) obtains 31%∼57% space savings ratio in the datasets with 4∼20 alphabets (Group 1), while it obtains 20%∼36% space savings ratios without the code detection (See DA-DPC). Second, DA-DPC gains around 32% in bag-of-words datasets (Group 2). The effectiveness of the code conversion function is clearly presented in that DA-DPC shows better performance than DA-DPC without code conversion (-CodeConv). Third, DA-DPC achieves 13%∼18% space savings ratio in the natural language dataset with a larger number of alphabet (Group 3), which is out of scope in this paper, and we leave further improvement on this scope as our future work. BPC as well as BDI and FPC rarely compress character data.

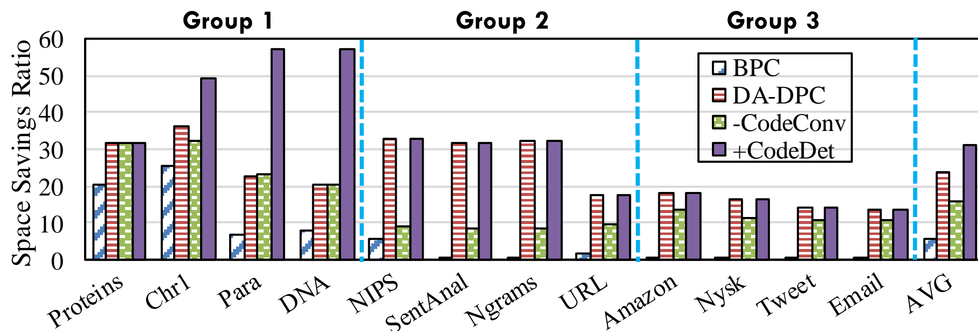


Figure 3.11: Effects of Character Data Operation in M.SM With 12 Real-World Datasets

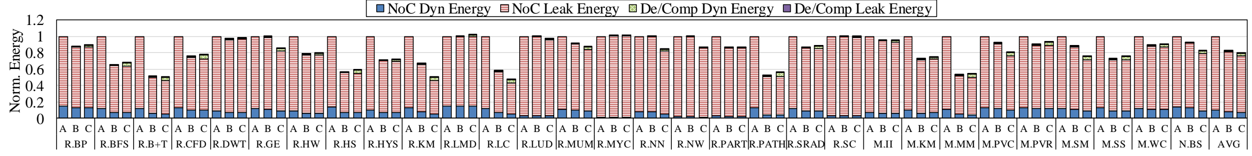
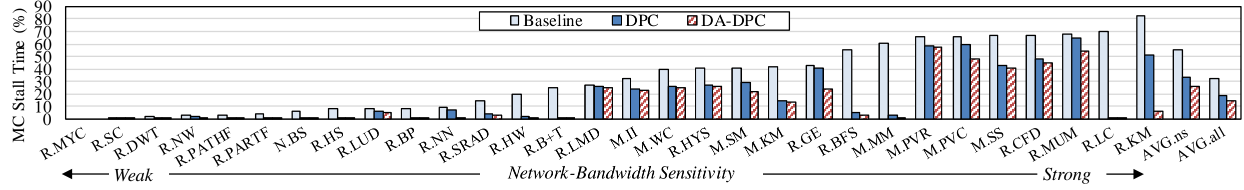
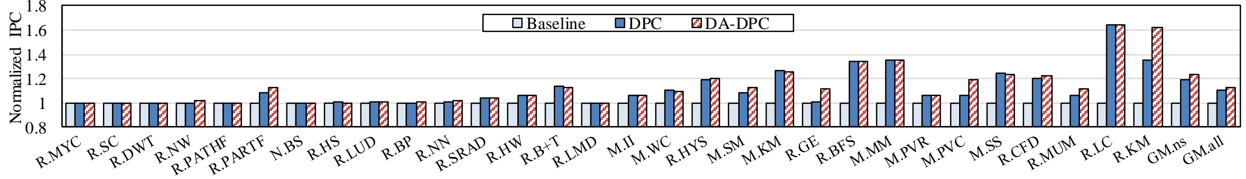


Figure 3.12: Energy Savings in NoC (2DMesh) by DPC and DA-DPC (A: Baseline, B: DPC, C: DA-DPC)



(a) MC Stall Time Ratio



(b) Normalized IPC

Figure 3.13: (a) Analysis of Network Bottleneck and (b) Normalized IPC in NVIDIA Fermi using Crossbar and either DPC or DA-DPC

3.6.4 Energy Analysis

In this section we provide an analysis on the energy savings when our proposed techniques are adopted in a GPU. We first measure the energy cost of our synthesized compression and decompression schemes. DPC requires 0.046 nJ and 0.029 nJ for compressing and decompressing a cache block, respectively. DA-DPC needs 0.23 nJ for compression and 0.103 nJ for decompression. Especially, our DPC shows 4x and 2x better energy efficiency in compression and decompression than BDI [40] commonly accepted as the lightest compressor.

Next, we evaluate the energy consumption in a NoC by integrating DSENT [30] in GPGPU-Sim and including the energy costs of DPC and DA-DPC. Figure 3.12 shows the normalized dynamic and static energy costs of baseline, DPC and DA-DPC. In each benchmark, an energy cost

is normalized against the total energy consumption of its baseline. DA-DPC achieves 21% (up to 53%) total energy savings on 2DMesh by reducing both in-flight flits by 38% and the total execution time by 23% on average. DPC also achieves 18% energy-savings.

3.6.5 Packet Compression under Crossbar

We observe that the bottleneck consistently appears in a GPU with a crossbar. Figure 3.13a shows that NVIDIA Fermi suffers from 32% MC stall ratio on average across all benchmarks ($AVG.all$), even though it is a small-scale GPU with 15 SMs. Especially, network-bandwidth sensitive applications ranging from M.II to R.KM, exhibit high stall ratio 56% (up to 82%) on average ($AVG.ns$) due to their high memory intensity (e.g. R.KM, R.LC, R.MUM, and R.CFD), irregular memory access patterns (e.g. R.GE and R.BFS) and map-reduce operations (e.g. M.SS, M.PVC and M.PVR). Although the crossbar is featured by high bandwidth, a number of reply data, generated by $bursty$ and $frequent$ memory requests from SMs, create a bottleneck state in the NIs and contentions in the network. This implies that the bottleneck stems from the architectural characteristics of GPUs that execute many threads simultaneously.

Figure 3.13b shows the normalized IPC when DPC and DA-DPC are adopted in NVIDIA Fermi. Our DA-DPC effectively addresses the bottleneck problem, thereby accomplishing 23% (up to 64%) IPC improvement for the network-bandwidth sensitive benchmarks ($GM.ns$). The bottleneck is relieved due to the reduced data volume, which is explained by MC stall time ratio reduction from 56% in the baseline to 26% ($AVG.ns$) as shown in Figure 3.13a. Similarly, our DPC also effectively achieves 19% IPC improvement (up to 64%).

3.7 Conclusions

In this paper, we present a simple but effective DPC compression scheme with data preprocessing capability. We observe that the bit-level redundancy naturally exists in cache blocks across various applications. Our DPC scheme preprocesses a cache block to artificially create more bit-level redundancy for three primitive data types. Then, our scheme creates two frequent patterns by transposing the preprocessed cache block in a bit-wise manner and compresses them into a

single bit. Our evaluation shows that DPC scheme effectively improves the network bandwidth of a highly congested NoC in a large-scale GPU by achieving 47% average space savings ratios and 33% IPC improvement across a number of benchmarks. Hence, we conclude that DPC is an effective compression scheme for a variety of compression domains due to its low latency overhead and high compressibility, and we believe that DPC can serve as a versatile compressor by extending data preprocessing layers as per applications.

4. ENERGY-EFFICIENT CNN INFERENCE EXPLOITING FEATURE CRITICALITY

4.1 Introduction

The emerging paradigm of the Internet of Things (IoT) recently penetrates many diverse areas of our daily lives. One of the widely-recognized areas is wearable computing systems for human activity recognition (HAR) adopted in many domains such as healthcare [69], home behavior analysis [70] and smart home [71]. Human activities are identified based on the signals collected from on-body sensors which are practically more preferable than cameras due to their properties such as privacy-preservation, power-efficiency and wide use in popular IoT devices (e.g. smartwatch) [72]. The sophisticated data analysis on the time series data from the sensors is a key factor in the success of HAR.

Nowadays, HAR leverages a convolutional neural network (CNN) as an inference method due to its state-of-the-art prediction quality [72]. Unlike traditional machine learning techniques that often rely on handcrafted and domain-specific low-level features (e.g. statistical information), CNNs achieve high-quality prediction by exploiting internally extracted high-level features essential for identifying complex activities. However, the feature extraction requires heavy computations in CNNs (e.g. convolution operation), inevitably leading to high energy expenditure, which is undesirable for the IoT devices with a limited energy source [72].

To support an energy-efficient inference, we design an early-prediction-based CNN architecture that performs classification by selectively exploiting different levels of features according to the difficulty of input instances. The low-level features extracted at an earlier convolutional layer are sufficient for easy input instances, while the high-level features at later convolutional layers are essential for difficult ones. Therefore, unlike a CNN that typically makes a final prediction with the features from the last feature extraction layer, the early-prediction-based CNN completes a final prediction at different levels of feature extraction layers depending on inputs through a classifier added at each layer. For any given input instance, if a classifier at a layer produces a confident

prediction result, the inference ends with the result. Otherwise, a classifier at the next layer takes over the inference. In this manner, the inference is attempted in the order of layers until the last layer is reached.

There are three difficulties in designing an early-prediction-based CNN architecture. First, it is challenging to improve the power-efficiency of a general classification task at each layer. The previous proposals focused on constructing an early classifier by exploiting all possible features [73] [74], but our goal is to use **critical features only** to reduce unnecessary computation cost. To construct a powerful early classifier, how to choose a good combination of critical features among numerous candidates becomes a key challenge. Second, it is also difficult to determine whether or not the prediction result at a certain layer is confident enough to terminate an inference early. Typically, the confidence level is measured by a classifier during runtime, which is compared against a user-defined threshold found manually offline [73] [74]. However, to choose the best classifier among many candidates each coupled with different set of critical features, it is essentially required to automatically find the threshold level for each candidate. Third, it becomes even more difficult to address the above two challenges in the context of constructing early classifiers for multiple layers which interdependently affect the performance of the early prediction network. It is essential to strike a good balance among the early classifiers, which globally maximizes the inference energy-efficiency without a large accuracy drop compared to an original CNN.

To guide the design of an energy-efficient early-prediction network, we propose the optimization methodology that determines three key information per layer based on a genetic algorithm: a feature subspace, critical features, and a confidence threshold. We define a feature subspace per layer with a subset of an output feature map. We adopt gradient boosting trees (GBTs) as an early classifier where trees are incrementally constructed based on critical features within the feature subspace. A genetic algorithm searches for the least feature subspace, determines critical features by choosing the best size of GBTs, and finds the best confidence threshold. We evaluate our proposal by implementing an inference engine based on low energy design principles. The result shows that we achieve 77% energy-savings with 0.003 accuracy loss on average in six HAR

benchmarks.

4.2 Background and Motivation

4.2.1 System Overview

In this paper, we propose a CNN-based network that facilitates early predictions with critical features for energy-efficient classification as illustrated in Figure 4.1. We build our proposed network by augmenting the baseline CNN comprising three repetitions of a stack of a 1D-convolutional layer and a maxpooling layer for feature extraction, followed by a single-layered fully-connected network. Initially, the baseline CNN is trained to completion. The proposed network redesigns the feature extraction layers into more power-efficient layers by reducing the input dimension for each layer (i.e. an output dimension of a previous layer), that is called a feature subspace. As a result, the computation load per layer is reduced. Unlike the weight pruning [75], we adopt the trained filters of convolutional layers in the baseline as it is. Moreover, our network introduces a gradient boosting tree (GBT)-classifier at every feature extraction layer, so it performs an early prediction by exploiting critical features in a feature subspace. The early prediction process produces not only a predicted class (*hard-label*) but also a confidence level (*soft-label*). The prediction processes in six layers are sequentially executed by an early prediction controller. If the predicted class is confident enough at a certain layer, an inference ends, skipping the processes in the remaining layers. Otherwise, an early prediction process in the next layer is activated. The hyperparameters (e.g. tree size) necessary to configure our network are found through our offline optimization method based on a genetic algorithm.

4.2.2 Motivation

To see the feasibility of the early prediction with critical features, we conduct a motivation study by evaluating the accuracy of the GBT-classifier as varying the number of trees (1$\sim$200) in all layers and measure the ratio of features used by each GBT-classifier among all features in ADL-S benchmark. Given a set of features (i.e. featuremap) as input, a gradient boosting method constructs trees by choosing discriminative features first where a feature corresponds to each scalar

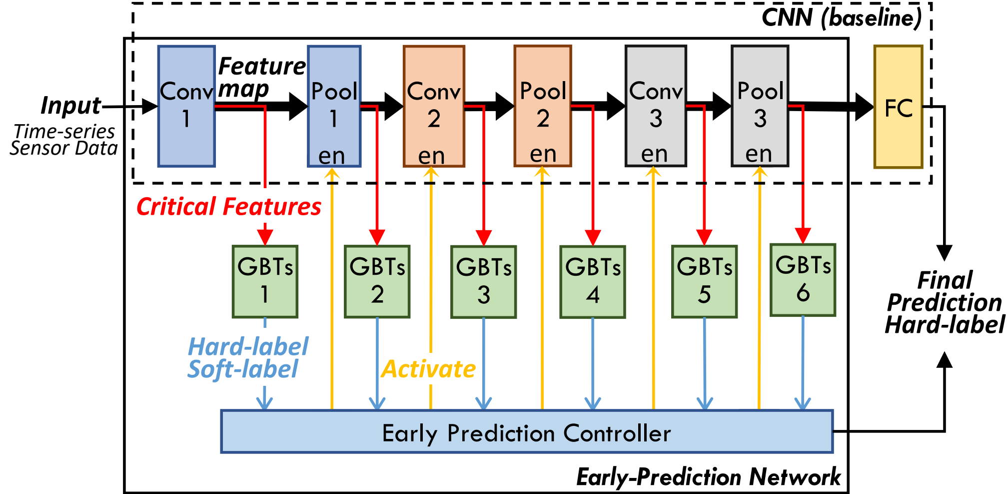


Figure 4.1: Illustration of a 1D-convolutional network coupled with GBTs exploiting critical features for early predictions

data of a featuremap. (See the detailed configuration in Section 4.5.1).

In this study, we made two key observations. First, the classifications at different layers reach the best achievable layerwise accuracy without using all input features. When a gradient boosting method adds more trees, some new features can be chosen or some previously selected features are reused. As shown in Figure 4.2(b), the ratio of used features increases but becomes saturated as the number of trees increases. Likewise, the accuracy improvement according to the number of trees follows the same trend as shown in Figure 4.2(a). When a GBT-classifier uses the maximum number of trees, 70%, 84%, 65%, 74%, 74% and 86% of features are used in $conv1$, $pool1$, $conv2$, $pool2$, $conv3$ and $pool3$, respectively. Second, low-level features are discriminative enough to perform accurate predictions for a good range of inputs without relying on the high-level features at the last layer. This is derived from the best achievable accuracy difference (9%) between the first layer and the last layer in Figure 4.2(a).

To understand the actual contribution of features to classification, we evaluate the feature criticality with the maximum GBTs (i.e. 200). The feature criticality ($Fscore$) is measured based on the number of times that each feature is used to split nodes in a decision tree. Figure 4.3 shows the distribution of features according to their criticality in ADL-S where the criticality range is evenly

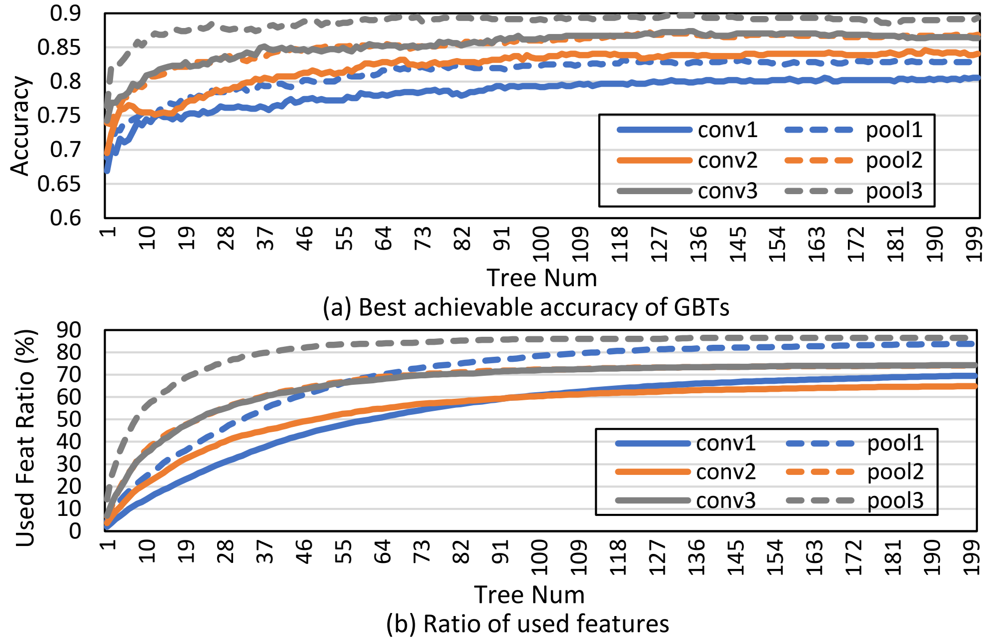


Figure 4.2: Accuracy and the ratio of used features in a featuremap according to tree numbers

divided into ten bins denoted from $C0$ to $C9$. $C0$ is the group of features that are never chosen for the decision trees, whereas $C9$ is a group of the most critical features that are frequently selected. The result shows that reasonably good accuracy at each layer is achieved even though 81% of the entire featuremap (i.e. $C0$ and $C1$) on average are identified as unimportant. Similar observations are captured from different benchmarks UCI-S and ACT-S, which show 79% and 51%, respectively. This motivates us to see the potential of exploiting a subset of features for designing a more energy-efficient feature extraction network.

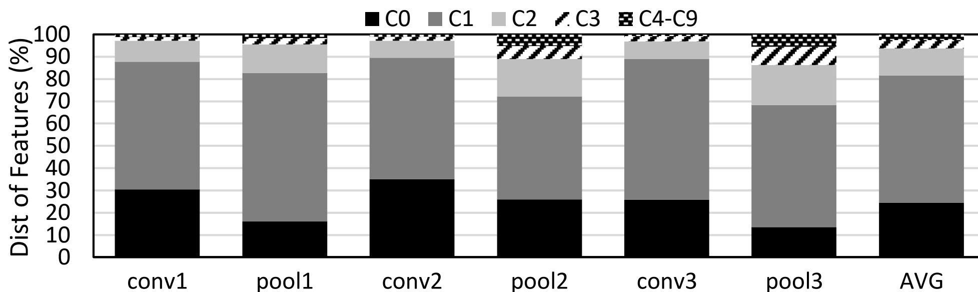


Figure 4.3: Distribution of features according to their criticality in ADL-S

4.3 Early Prediction with Critical Features

4.3.1 Feature Subspacing

Our observation on feature criticality implies that only some features are important for prediction at each layer, which directly helps to reduce unnecessary computation cost by focusing on the important features only. However, it is not sufficient to compute them only because of layer-to-layer data dependency. As a multi-layered feature extraction network is designed in a feed-forward manner, a layer has data dependency on a previous layer. Therefore, even though some features are not immediately used for the early prediction at the current layer, they can be a necessary input for the next layer to compute its critical features.

Figure 4.4 illustrates an example of necessary feature computation when $pool1$ depends on $conv1$. Suppose that a group of features, A and C are known most discriminative in a given featuremap for early prediction at $conv1$, and E and F for $pool1$. Computing E and F at $pool1$ requires B and D at $conv1$ as input, which needs to be produced by $conv1$ even though they are not directly used for prediction in the layer. An important observation is that to save the computation cost at $conv1$, it is desirable that GBTs1 at $conv1$ use B and D rather than A and C. In other words, we can guide a gradient boosting method to first choose features within B and D when constructing GBTs initially. In practice, it is feasible because many different feature combinations eventually show very close classification power.

Therefore, we propose a feature subspacing method that divides the featuremap in all layers into N feature segments subject to global layer-to-layer data dependency. We can reduce the energy cost for feature extraction by choosing a feature subspace, only K segments ($K < N$) sufficient for early prediction and skipping computation for non-selected segments. As the layers in a feed-forward feature extraction network has a data dependency toward a backward direction, our subspacing method does not arbitrarily partition an output featuremap at each layer. Instead, it determines the segments per layer in a backward manner sequentially. First, the featuremap at the last layer is divided into a set of segments. Next, the input regions (i.e. an output region at

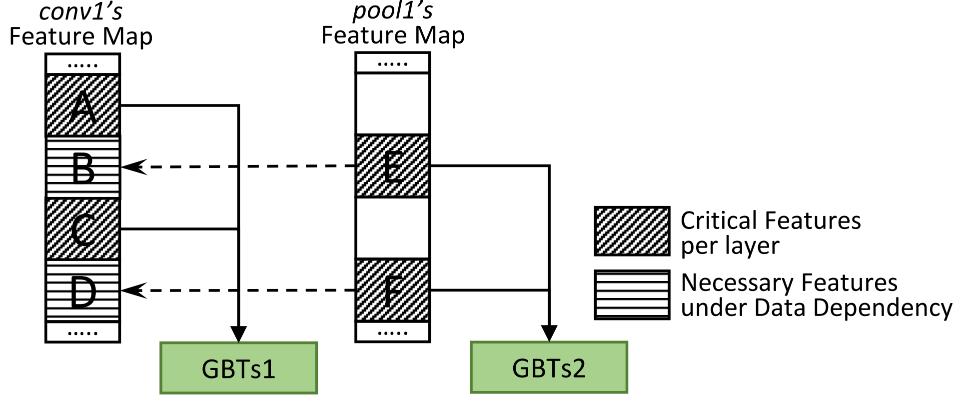


Figure 4.4: Illustration of a feature computation example under the data dependency between $conv1$ and $pool1$ layers.

the second last layer) that the last layer needs to compute each segment is set as segments for the second last layer. We repeat the same process until we get the segments at the first layer.

Figure 4.5 illustrates an example of creating a feature segment in a backward manner from $pool3$ to $conv1$. A featuremap at every layer consists of a time dimension and a channel dimension (C). We first partition a 16×64 featuremap at layer 6 ($pool3$) in the time dimension into feature segments. The tth segment at the time dimension in layer l is denoted as S_t^l. In this example, 16 segments (S_t^6, $t = 1, .., 16$) of shape 1×64 are created. For each segment S_t^6, we search for a corresponding segment, S_t^5 in the featuremap from $conv3$ that S_t^6 depends on. Since S_t^5 is necessary data that $pool3$ needs for computing S_t^6, the shape of S_t^5 depends on the pooling size which is an input window in the time dimension for a pooing operation. Moreover, the start location of S_t^5 in the time dimension depends on the stride size of $pool3$ which is an offset between consecutive input windows. This example uses a pooling size 4 and a stride size 2, so the shape of S_2^5 is 4×64 and its time dimension range is from 2 to 5. In the same manner, the subspace searching is recursively conducted until S_2^1 is found for the first layer ($conv1$). We complete the feature subspacing by repeating this process for all S_t^6 ($t = 1, .., 16$) in $pool3$. As a result, 16 feature segments are created in every layer. How to choose a subset of segments (i.e. K segments) will be detailed later in Section 4.3.3.

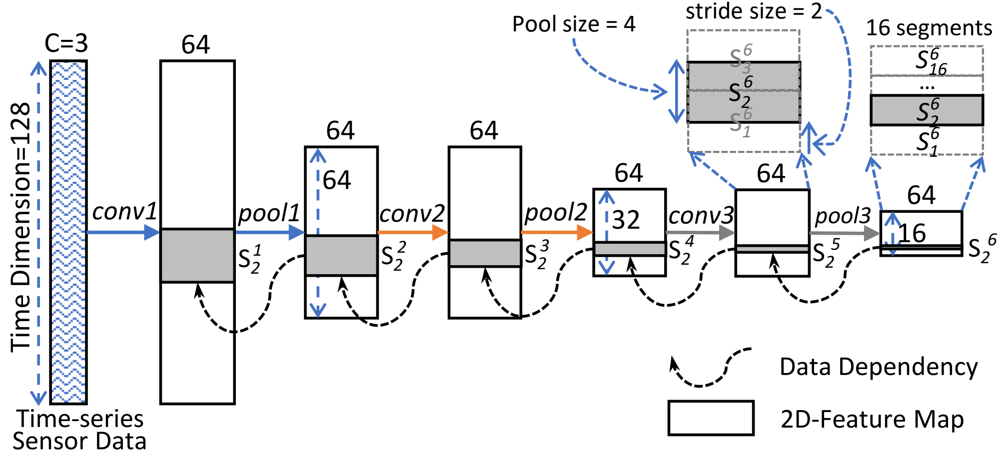


Figure 4.5: Illustration of creating a feature segment (S_2^l, $l = 1..6$) in a backward manner from $pool3$ to $conv1$

4.3.2 Layerwise Early Prediction

Given an input instance, an inference phase is performed through sequential layerwise predictions from $conv1$ to $pool3$ and ends when it obtains a confident result at any layer. To reduce the overhead of the layerwise predictions, we adopt GBTs consisting of T decision trees of depth 6 requiring simple comparison logics and a softmax function as a classifier, unlike other studies that adopt a fully-connected network requiring numerous expensive multiply operations [73] [74]. A GBT-classifier works as follows. Given an input instance, a set of the final prediction scores summed from GBTs, each for a target class is produced, and then it is computed into a set of corresponding probabilities through a softmax function. The class label with the highest probability is chosen as a prediction result (*hard-label*) while the probability (*soft-label*) is chosen as a confidence level of the prediction result. If a soft-label is below a certain threshold level (δ), meaning that the prediction result obtained from the currently processing layer is not confident, the hard-label is not taken as a final prediction result, so a new prediction is attempted in the next layer. Otherwise, the inference terminates at the current layer by taking the hard-label as a final prediction result.

4.3.3 Hyperparameter Optimization

To maximize the energy-efficiency of our early-prediction network without a large accuracy drop, it is critical to optimize the following three types of hyperparameters. First, the feature segments are parameterized as a vector (FSV), $[I_1, I_2, .., I_N]$ where N is the number of feature segments and I_t is set to one if the tth segment of a layer is selected, and zero otherwise. Since the segments are configured based on the global layer-to-layer data dependency, all layers have the same N segments which are parameterized as a single FSV, which simplifies the complexity of parameter searching. Second, the parameter vector for layerwise classifiers, EPV is summarized as $[\delta_1, .., \delta_L, T_1, .., T_L]$ where L is the number of layers in the feature extraction network and δ_l is a probability threshold and T_l is the number of trees for the lth layer. δ_l confirms that a hard-label obtained at the lth layer is confident if a corresponding soft-label is greater than δ_l. We set δ_L to 0 to make sure that the last layer covers the inputs for which the earlier layers do not make a confident prediction.

Overall Algorithm. Finding the best N+2L-1 parameters with naive exhaustive search is impractical. Instead, we present an iterative heuristic that jointly searches for FSV that minimizes the number of necessary segments and EPV that yields the best energy-efficiency without accuracy loss. A key idea of our algorithm is to find the best EPV while eliminating a segment at each iteration from FSV that is initially set to use all N segments. The overall algorithm is summarized in Alg 1.

Assume that the baseline network is first trained. The algorithm chooses the best EPV when N-1 segments are selected at the first iteration. There are N candidates of FSV each choosing a unique combination of N-1 out of N available segments. For each FSV candidate, we generate the features in the selected segments by running the baseline feature extraction network with a training dataset and train GBTs (up to max trees) per layer by using the features as input. We find the best EPV with a genetic algorithm and validate the EPV by evaluating an accuracy loss and an energy-savings with a validation dataset. By choosing the FSV candidate that yields the lowest accuracy loss with its best EPV, we determine N-1 segments available for the second

Algorithm 1: Hyperparams (FSV, EPV) Optimization

```
1  FSV_best = [1, .., 1], EPV_best = [0, .., 0]
2  for i ← N-1 to 0 do
3      Generate new FSVs
4      for j ← 0 to |FSVs|-1 do
5          Train GBTs in N layers with FSVs[j]
6          EPV[j] ← Generate the best EPV with GA
7          AL[j] ← Evaluate EPV[j]
8      end
9      k ← argmin(AL)
10     if AL[k] > δ_user then
11         return FSV_best, EPV_best
12     end
13     FSV_best ← FSV[k], EPV_best ← EPV[k]
14 end
15 return FSV_best, EPV_best
```

iteration where the FSV candidate with N-2 segments is selected in the same manner. We repeat this process until the validated accuracy loss is beyond a user-defined acceptable loss value (e.g. 0.01). Our algorithm takes the last valid FSV candidate and the best EPV found under the FSV as final parameters.

Genetic Algorithm to Generate EPV. To efficiently find the best EPV at each iteration, we use a general optimization technique, genetic algorithm (GA). The algorithm starts with a population of potential solutions (commonly expressed as individuals in GA), but it makes random changes (i.e. crossover and mutation) to the solutions to reproduce a new population of better solutions. The quality of each solution is evaluated with respect to a *fitness function*. Our genetic algorithm finds the best parameters that minimize the fitness score. After repeating this process many times, it tends to converge on a good solution. This algorithm is well-suited for finding a solution EPV. Initially, a population of EPVs is generated with random numbers. We perform crossover and mutation with a probability of 0.5 and 0.2, respectively but implement them in two approaches depending on data types of parameters. First, Ts of an integer type use a single-point crossover and a uniform random mutation with integer numbers from 0 to 200. Second,

δs of a floating-point type adopt a simulated binary crossover and a polynomial mutation with real numbers from 0 to 1.0, while using a distribution index 30.0 and 20.0, respectively [76]. We implement our GA by using an open-source framework [77] and choosing a population size, 300, and a fixed generation size, 200 where a solution is converged to a stable point.

Fitness Function. The fitness function computes a score that combines an accuracy loss and an energy-savings for given an EPV candidate with six sets of GBTs trained under a FSV candidate. The accuracy loss (AL) is calculated by $A_{cnn} - A_{epn}$ where A_{cnn} and A_{epn} are the accuracy of a baseline CNN and an early-prediction network, respectively. The energy-savings (ES) is $1 - E_{epn}/E_{cnn}$ where E_{epn} and E_{cnn} are the energy cost of an early-prediction network and a baseline, respectively. Suppose that a user-acceptable accuracy loss threshold (δ_{user}) is given. If AL meets a constraint of δ_{user} (i.e. $AL < \delta_{user}$), we calculate a score with an equation, AL/ES, which is for obtaining EPV that gains higher energy-savings. Otherwise, we calculate a score based on $AL*ES$, which guides the parameter searching in an opposite direction.

4.4 Hardware Implementation

4.4.1 Functional Primitives

The full-custom hardware implementation for different benchmarks based on the selected FSV and EPV requires enormous efforts. To make the hardware design more practical, we modularize the key functions into hardware primitives each of which covers 1D-convolution, maxpooling, fully-connected network, GBT, softmax, and an early-prediction controller. The primitive is implemented to produce a single scalar output. Moreover, we adopt low energy design principles such as the data flow model and power gating [78]. Each layer is implemented with a group of primitives that run in parallel.

For instance, to implement an early-prediction network, we build a feature extraction network by choosing convolution or pooling primitives that compute the features in the selected feature segments based on Is parameters. Similarly, a GBT-classifier is built with GBT primitives based on Ts parameters. An early-prediction controller uses δ parameters.

In specific, we implement the functional primitives based on fixed-point floating numbers consisting of a 16-bit integer and a 16-bit fraction and verify their correctness with Verilog Compile Simulator (VCS). We also use SAIF files that record switching activities for accurate power estimation. To evaluate the power overhead, we synthesize the functional primitives using Synopsys Design Vision [52] based on Synopsys SAED32nm EDK Digital Standard Cell Library [79] at a 16 MHz clock frequency.

4.5 Evaluation

4.5.1 Benchmark and Model Selection

We evaluate our proposal in a challenging time series classification problem, HAR with three publicly available datasets: ADL [80], ACT [81], UCI [82] where time-series data such as triaxial acceleration and/or triaxial angular velocity is collected from human subjects carrying a smartphone embedding accelerometer and gyroscope sensor while they perform daily activities. The dataset is divided for training, validation, and test by 50%, 25%, and 25%, respectively. The training set is for training a baseline CNN and GBTs, the validation set for searching the best FSV and EPV, and the test set for final evaluation reported in this section.

As for our baseline network, we carefully choose a non-over-parameterized CNN consisting of three convolutional and maxpooling layers followed by a fully-connected layer as illustrated in Figure 4.1. The filter size and pooling size are 4 and the stride sizes for convolution and pooling are 1 and 2, respectively. We design two scales of networks for three datasets where a small network (each with a suffix -S) uses 64 filters all convolutional layers and a large network (each with a suffix -L) uses 64, 128, and 256 filters at $conv1$, $conv2$ and $conv3$, respectively.

Table 4.1 summarizes the parameters of our early-prediction network found by our algorithm in Section 4.3.3 under δ_{user} set to 0.01. Due to space limit, FSV is shortened as a bitstream where I_1 is at the leftmost position and the last I at the rightmost one.

	Feat Subspace (Is)	Number of Trees (Ts)	Thresholds (δs)
ACT-S	010000001010	[144,79,16,200,135,178]	[0.970,0.959,0.742,0.891,0.881]
ADL-S	1000000100000100	[7,69,35,188,6,98]	[0.401,0.910,0.830,0.786,0.384]
UCI-S	0101000000001000	[115,24,16,124,94,49]	[0.992,0.923,0.755,0.942,0.999]
ACT-L	010000011000	[139,10,68,195,132,150]	[0.975,0.587,0.956,0.974,0.643]
ADL-L	1000000001000100	[2,60,200,156,188,115]	[0.72,0.915,0.988,0.855, 0.949]
UCI-L	0000000100011000	[2,15,54,3,3,199]	[0.650,0.939,0.820,0.997,0.556]

Table 4.1: Parameters for early-prediction network

4.5.2 Evaluation Result

Energy Efficiency. Figure 4.6 shows both energy savings and accuracy loss of our early-prediction networks against the baseline. The energy savings of two types of early-prediction networks each configured without or with feature subspacing are presented in Figure 4.6(a). The accuracy loss of early-prediction networks configured with feature subspacing is shown in Figure 4.6(b).

We make two observations as follows. First, the early-prediction network achieves 77% energy savings with 0.003 accuracy loss on average when the feature subspacing is used. In specific, the energy use is reduced by 79%, 77%, 74%, 80%, 79%, 76% in ACT-S, ADL-S, UCI-S, ACT-L, ADL-L, and UCI-L, while accuracy losses are allowed by 0.003, -0.003, 0.004, 0.003, 0.003, and 0.009, respectively. 23%, 13%, 17%, 29%, 4%, 13% of tested inputs are predicted at $conv1$, $pool1$, $conv2$, $pool2$, $conv3$ and $pool3$, respectively on average in six benchmarks.

Second, using our feature subspacing results in higher energy savings by 54% than when it is not used as shown in Figure 4.6(a). It is because the subspacing method not only reduces the power cost spent for feature extraction at each layer but also assists the effective parameter searching of a genetic algorithm. When the feature subspacing is not used, the genetic algorithm converges to the EPV that mainly activates the early prediction at $conv3$ and $pool3$ with more powerful high-level features to preserve accuracy quality. In contrast, the use of our feature subspacing excludes more features from the later layers due to the backward data dependency, causing the layers to have weaker classification power. Therefore, the genetic algorithm is guided to find EPV that allows a

good amount of inputs to be predicted at earlier layers.

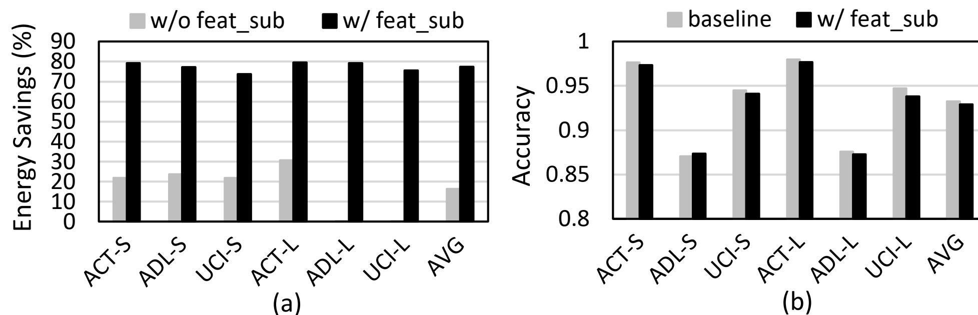


Figure 4.6: Energy savings (a) and accuracy (b) of the early-prediction network against the baseline.

Figure 4.7 shows the energy savings according to the layer of early prediction against the previous studies [73] [74]. The energy cost of early prediction at ith layer sums all energy costs spent from the 1st layer to ith layer. To highlight the true benefit of our proposal, we focus on the energy cost of feature extraction only because our GBT-classifier is more energy-efficient than fully-connected network adopted in [73] [74]. Our proposal achieves the energy savings, 23% at layer 1 and 2, 32% at layer 3 and 4, and 46% at layer 5 and 6, respectively on average. The energy savings becomes higher at later layers. It is because the earlier layers are required to compute more features than the later layers due to the layer-to-layer data dependency as discussed in Section 4.3.1, resulting in the lowest power savings at $conv1$ and the highest one at $pool3$.

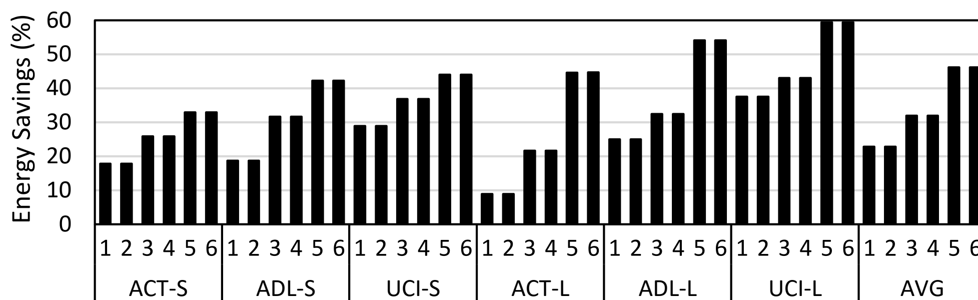


Figure 4.7: Energy savings according to layer of early prediction

Feature Criticality Analysis. Figure 4.8 visualizes the criticality of features in the layerwise feature maps shown with the time dimension (x-axis) and the filter dimension (y-axis). In the colored heatmap for six layers of ADL-S, the feature data points closer to black are less critical, whereas the ones closer to white are more critical. While the features within a feature subspace are presented with various criticality levels, the others are shown as contiguous black regions. In addition, the result shows that our proposal effectively excludes unnecessary feature computation by having $C0$ ratios, 38%, 30%, 53%, 53%, 70%, and 83% in $conv1$, $pool1$, $conv2$, $pool2$, $conv3$, $pool3$, respectively, which are higher than our motivation study shown in Figure 4.3. Furthermore, there are still high $C1$ ratios, 51%, 51%, 31%, 26%, 14%, and 4% in the six layers. This implies that there is a potential to achieve more energy savings through finer-grained subspacing, which we leave as our future work.

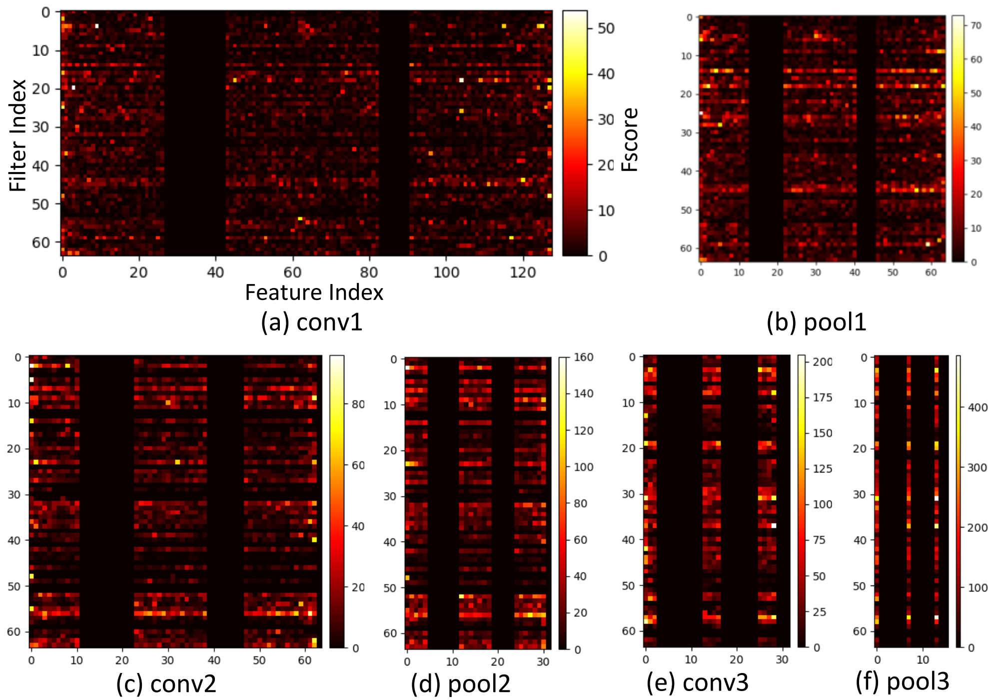


Figure 4.8: Heatmap of layerwise feature criticality in ADL-S

Inference Time. We observe that our early-prediction network reduces the worst inference

time due to simple GBT-based classifiers, even though our main focus is on improving energy-efficiency. It is natural to reduce the average inference time due to easy input instances for which inference is finalized in the intermediate layers. Furthermore, in the worst case that need high-level feature at the last layer, our proposed network has less inference time than the baseline due to simple GBT classifiers, which cannot be easily achieved in the previous studies based on multiple fully-connected layers [73] [74]. The latency overhead of all GBT-classifiers added in every feature extraction layer is less than a FC layer in the baseline implemented with the data flow model consisting iterative multiply operations. The time reduction ratio of the worst case classification in ACT-S, ACT-S, ACT-S, ACT-L, ADL-L, and UCI-L are 25%, 49%, 47%, 68%, 75% and 79%, respectively on average in six benchmarks.

4.6 Related Work

There are two main approaches to reduce the energy cost of complicated CNNs by optimizing models. First, compressing CNNs is the most common approach that prunes unimportant model parameters [75], which can skip the computations relevant to the removed parameters. In contrast, ours chooses critical features only while keeping model parameters intact, which can synergetically reduce energy cost with the pruning approach. Second, the prediction adapting to input difficulty is another approach to reduce the average energy cost [73] [74]. Unlike this approach, ours performs early prediction with critical features only and relevant parameters are found automatically.

Our proposal is also relevant to the feature selection that chooses a subset of features for traditional learning models whose performance is substantially impacted by the quality of input features [83]. Unlike the previous studies, our proposal shows a good use of feature selection to reduce the computation overhead of a feature extraction network in CNNs.

4.7 Conclusions

The energy-efficient inference is critical for IoT devices with limited battery life. We propose a design methodology that transforms a CNN into an early-prediction network exploiting critical features to reduce the energy cost for inference. This is achieved by the feature subspace and the

early-prediction parameters that are intelligently chosen by a genetic algorithm to maximize the energy-efficiency without an accuracy drop. We demonstrate the effectiveness of our proposal by showing 77% energy-savings against a baseline CNN with an ignorable accuracy drop on average in six HAR benchmarks.

5. CONCLUSIONS

Hardware accelerators are becoming more critical than ever as a cost-effective computing platform for emerging applications from servers to mobile devices. Servers often leverage many-core accelerators such as GPUs to achieve high performance gain by exploiting simple yet energy-efficient compute cores. Larger-scale GPUs are essential for the emerging applications that require the process of a considerable amount of data. However, it is challenging to scale up single-chip GPUs. In specific, the data movement overhead over the GPUs' NoC becomes a critical performance bottleneck in large-scale GPUs. Unlike servers, mobile devices often leverage low-power accelerators for inference in DNNs. The energy efficiency of the accelerators is critical in mobile devices, but they are required to compute complex algorithms of DNNs, which becomes a key energy bottleneck. In this dissertation, we explore the solutions to address the performance bottleneck incurred by unnecessary data movement in GPUs and the energy bottleneck caused by heavy computation in AI accelerators.

First, we propose a packet coalescing mechanism that minimizes redundant packets over the NoC of GPUs [4]. Massive multi-threading in GPUs place heavy stress on the memory system, creating network bottlenecks near memory controllers. We observe frequent inter-core locality across various applications where data redundancy in communication traffic is commonplace. We propose a packet coalescing mechanism that coalesces multiple redundant packets into one without increasing the packet size. The coalesced packets are delivered to their respective destinations through multicast routing. Our coalescing approach yields 15% IPC improvement (up to 112%) in a large-scale GPU with 2D mesh across various GPU applications by obtaining network bandwidth savings by 13% (up to 37%).

Second, we propose a simple compression mechanism, Dual Pattern Compression (DPC), that compresses only two patterns with very low latency [36]. Unlike our packet coalescing proposal that reduces the number of packets [4], this compression approach reduces every single packet's size by compressing redundant values among data carried by a packet. Our compres-

sion/decompression is designed with simpler logic than the previous proposals but gives higher compressibility due to data remapping and data-type-aware data preprocessing which exploits bit-level data redundancy. We demonstrate the effectiveness of our proposal in a large-scale GPU by showing IPC improvement by 33% on average (up to 126%) across various benchmarks with average space savings ratios of 61% in integer, 46% (up to 72%) in floating-point and 23% (up to 57%) in character type benchmarks.

Last, we propose a network optimization methodology that reduces the energy cost of CNNs caused by considerable computation. The proposed methodology transforms an original CNN into a new CNN with early-prediction capability based on GBTs classifiers that make a prediction with important features only from each feature extraction network layer. A genetic algorithm finds the best hyperparameters that maximize the energy-efficiency of inference with an ignorable accuracy drop for the early-prediction-based CNN. We achieve the energy-savings by 77% on average over the baseline CNN with an ignorable accuracy drop in human activity recognition benchmarks.

REFERENCES

[1] NVIDIA, "TITAN Xp Graphics Card with Pascal Architecture NVIDIA GeForce."

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, (New York, NY, USA), pp. 1–12, ACM, 2017.

[3] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco, "Gpus and the future of parallel computing," *Micro, IEEE*, vol. 31, pp. 7–17, Sept 2011.

[4] K. H. Kim, R. Boyapati, J. Huang, Y. Jin, K. H. Yum, and E. J. Kim, "Packet coalescing exploiting data redundancy in gpgpu architectures," in *Proceedings of the International Conference on Supercomputing*, ICS '17, (New York, NY, USA), pp. 6:1–6:10, ACM, 2017. `http://doi.acm.org/10.1145/3079079.3079088`.

[5] A. Bakhoda, J. Kim, and T. M. Aamodt, "Throughput-effective on-chip networks for many-core accelerators," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*, pp. 421–432, 2010.

[6] H. Jang, J. Kim, P. Gratz, K. H. Yum, and E. J. Kim, "Bandwidth-efficient on-chip interconnect designs for gpgpus," in *Proceedings of the 52nd Anual ACM/EDAC/IEEE Design

Automation Conference (DAC 2015), pp. 1–6, 2015.

[7] A. K. Ziabari, J. L. Abellán, Y. Ma, A. Joshi, and D. Kaeli, "Asymmetric noc architectures for gpu systems," in *Proceedings of the 9th Anual International Symposium on Networks-on-Chip (NOCS 2015)*, pp. 25:1–25:8, 2015.

[8] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA-39)*, pp. 416–427, 2012.

[9] A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Orchestrated scheduling and prefetching for gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA-40)*, pp. 332–343, 2013.

[10] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. Mowry, and O. Mutlu, "A case for core-assisted bottleneck acceleration in gpus: Enabling flexible data compression with assist warps," in *Proceedings of the 42nd ACM/IEEE Anual International Symposium on Computer Architecture (ISCA-42)*, pp. 41–53, 2015.

[11] D. Li and T. M. Aamodt, "Inter-core locality aware memory scheduling," *IEEE Computer Architecture Letters*, vol. 15, pp. 25–28, Jan 2016.

[12] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Proceedings of International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*, pp. 163–174, 2009.

[13] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron, "A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads," in *Proceedings of International Symposium on Workload Characterization (IISWC 2010)*, pp. 1–11, 2010.

[14] W. Fang, B. He, Q. Luo, and N. K. Govindaraju, "Mars: Accelerating mapreduce with graphics processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 608–620, April 2011.

[15] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *Innovative Parallel Computing (InPar), 2012*, pp. 1–10, May 2012.

[16] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, vLi Wen Chang, N. Anssari, G. D. Liu, and W. mei W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Tech. Rep. IMPACT-12-01, University of Illinois at Urbana-Champaign, East Lansing, Michigan, March 2012.

[17] NVIDIA, "Fermi: Nvidia's next generation cuda compute architecture," 2009.

[18] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus," in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, (New York, NY, USA), pp. 256–265, ACM, 2009.

[19] J. Balfour and W. J. Dally, "Design tradeoffs for tiled cmp on-chip networks," in *Proceedings of the 20th Annual International Conference on Supercomputing (ICS 2006)*, pp. 187–198, 2006.

[20] N. E. Jerger, L.-S. Peh, and M. Lipasti, "Virtual circuit tree multicasting: A case for on-chip hardware multicast support," in *Proceedings of THE 35th Anual International Symposium on Computer Architecture (ISCA-35)*, pp. 229–240, 2008.

[21] L. Wang, Y. Jin, H. Kim, and E. J. Kim, "Recursive partitioning multicast: A bandwidth-efficient routing for networks-on-chip," in *Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip (NoCS 2009)*, pp. 64–73, 2009.

[22] S. Ma, N. Jerger, and Z. Wang, "Supporting efficient collective communication in nocs," in *Proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture (HPCA-18)*, pp. 1–12, 2012.

[23] T. Krishna, L.-S. Peh, B. M. Beckmann, and S. K. Reinhardt, "Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*, pp. 71–82, 2011.

[24] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[25] N. Jiang, D. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. Shaw, J. Kim, and W. Dally, "A detailed and flexible cycle-accurate network-on-chip simulator," in *Proceedings of 2013 IEEE Inernational Symposium on Performance Analysis of Systems and Software (ISPASS 2013)*, pp. 86–96, 2013.

[26] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40)*, pp. 3–14, 2007.

[27] J. Kloosterman, J. Beaumont, M. Wollman, A. Sethia, R. Dreslinski, T. Mudge, and S. Mahlke, "Warppool: Sharing requests with inter-warp coalescing for throughput processors," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*, pp. 433–444, 2015.

[28] W. Jia, K. Shaw, and M. Martonosi, "Mrpb: Memory request prioritization for massively parallel processors," in *Proceedings of the 20th Anual International Symposium on High Performance Computer Architecture (HPCA-20)*, pp. 272–283, 2014.

[29] D. Abts, N. Enright, J. J. Kim, D. Gibson, and M. Lipasti, "Achieving predictable performance through better memory controller placement in many-core cmps," in *Proceedings of the 36th Annual International Symposium on Computer architecture (ISCA-36)*, pp. 451–461, 2009.

[30] C. Sun, C.-H. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L.-S. Peh, and V. Stojanovic, "Dsent - a tool connecting emerging photonics with electronics for opto-electronic

networks-on-chip modeling," in *Proceedings of the 6th IEEE/ACM International Symposium on Networks-on-Chip (NOCS 2012)*, pp. 201–210, 2012.

[31] C.-T. Chen, Y.-C. Huang, Y.-Y. Chang, C.-Y. Tu, C.-T. King, T.-Y. Wang, J. Sang, and M.-H. Li, "Designing coalescing network-on-chip for efficient memory accesses of gpgpus," in *Network and Parallel Computing*, pp. 169–180, 2014.

[32] G. Pekhimenko, E. Bolotin, N. Vijaykumar, O. Mutlu, T. C. Mowry, and S. W. Keckler, "A case for toggle-aware compression for gpu systems," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 188–200, March 2016.

[33] V. Sathish, M. J. Schulte, and N. S. Kim, "Lossless and lossy memory i/o link compression for improving performance of gpgpu workloads," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, (New York, NY, USA), pp. 325–334, ACM, 2012.

[34] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving gpu performance via large warps and two-level warp scheduling," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 308–317, Dec 2011.

[35] A. Jog, O. Kayiran, N. Chidambaram Nachiappan, A. K. Mishra, M. T. Kandemir, O. Mutlu, R. Iyer, and C. R. Das, "Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (New York, NY, USA), pp. 395–406, ACM, 2013.

[36] K. H. Kim, P. Devpura, A. Nayyar, A. Doolittle, K. Yum, and E. J. Kim, "Dual pattern compression using data-preprocessing for large-scale gpu architectures," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 675–685, 2019.

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, June

2016.

[38] NVIDIA, "NVIDIA's Tesla P100 Whitepaper," 2016.

[39] J. Kim, M. Sullivan, E. Choukse, and M. Erez, "Bit-plane compression: Transforming data for better compression in many-core architectures," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 329–340, June 2016.

[40] A. Arunkumar, S. Y. Lee, V. Soundararajan, and C. J. Wu, "Latte-cc: Latency tolerance aware adaptive cache compression management for energy efficient gpus," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 221–234, Feb 2018.

[41] A. Arelakis and P. Stenstrom, "Sc2: A statistical compression cache scheme," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pp. 145–156, 2014.

[42] A. Arelakis, F. Dahlgren, and P. Stenstrom, "Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods," in *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pp. 38–49, 2015.

[43] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, pp. 1196–1208, Aug 2010.

[44] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pp. 377–388, 2012.

[45] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," Tech. Rep. Technical Report 1500, Computer Sciences Department, University of Wisconsin-Madison, April 2004.

[46] S. Lee, K. Kim, G. Koo, H. Jeon, W. W. Ro, and M. Annavaram, "Warped-compression: Enabling power efficient gpus through register compression," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 502–514, June 2015.

[47] R. Das, A. Mishra, C. Nicopoulos, D. Park, V. Narayanan, R. Iyer, M. Yousif, and C. Das, "Performance and power optimization through data compression in network-on-chip architectures," in *IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 215–225, Feb 2008.

[48] Y. Jin, K. H. Yum, and E. J. Kim, "Adaptive data compression for high-performance low-power on-chip networks," in *41st IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 354–363, Nov 2008.

[49] B. Panda and A. Seznec, "Dictionary sharing: An efficient cache compression scheme for compressed caches," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, Oct 2016.

[50] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

[51] V. P. Nelson, H. T. Nagle, B. D. Carroll, and J. D. Irwin, *Digital Logic Circuit Analysis and Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995.

[52] H. Bhatnagar, *Advanced ASIC Chip Synthesis: Using Synopsys Design Compiler Physical Compiler and Prime Time*. Norwell, MA, USA: Kluwer Academic Publishers, 2nd ed., 2002.

[53] Taiwan Semiconductor Manufacturing Company, "45nm cmos standard cell library v120a," 2009.

[54] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA-40)*, pp. 487–498, 2013.

[55] NVIDIA, "CUDA C/C++ SDK Code Samples," 2011.

[56] M. Lichman, "UCI machine learning repository," 2013.

[57] "Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools," *Energy and Buildings*, vol. 49, pp. 560 – 567, 2012.

[58] P. Cortez and A. Morais, "A data mining approach to predict forest fires using meteorological data," in *Proc. EPIA 2007*, pp. 512–523, 2007.

[59] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the kdd cup 99 data set," in *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications*, pp. 1–6, July 2009.

[60] S. C. Tan, P. San Lau, and X. Yu, "Finding similar time series in sales transaction data," in *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, pp. 645–654, Springer, Cham, 2015.

[61] Z. Cheng, J. Caverlee, and K. Lee, "You are where you tweet: a content-based approach to geo-locating twitter users," in *Proceedings of the 19th ACM international conference on Information and knowledge management*, pp. 759–768, ACM, 2010.

[62] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 13–24, Feb 2007.

[63] M. Hu and B. Liu, "Mining and summarizing customer reviews," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 168–177, ACM, 2004.

[64] M. Davies, "N-grams data from the corpus of contemporary american english (coca),"

[65] R. Sinha, J. Zobel, and D. Ring, "Cache-efficient string sorting using copying," *Journal of Experimental Algorithmics (JEA)*, vol. 11, pp. 1–2, 2007.

[66] P. Ferragina and G. Navarro, "Pizza&chili corpus-compressed indexes and their testbeds," *September*, 2005.

[67] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki, "Storage and retrieval of individual genomes.," in *Recomb*, vol. 5541, pp. 121–137, Springer, 2009.

[68] J. Sirén, "Compressed suffix arrays for massive data.," in *SPIRE*, vol. 5721, pp. 63–74, Springer, 2009.

[69] X. Liu, L. Liu, S. J. Simske, and J. Liu, "Human daily activity recognition for healthcare using wearable and visual sensing data," in *2016 IEEE International Conference on Healthcare Informatics (ICHI)*, pp. 24–31, IEEE, 2016.

[70] P. Vepakomma, D. De, S. K. Das, and S. Bhansali, "A-wristocracy: Deep learning on wrist-worn sensing for recognition of user complex activities," in *2015 IEEE 12th International conference on wearable and implantable body sensor networks (BSN)*, pp. 1–6, IEEE, 2015.

[71] A. Wang, G. Chen, C. Shang, M. Zhang, and L. Liu, "Human activity recognition in a smart home environment with stacked denoising autoencoders," in *International conference on web-age information management*, pp. 29–40, Springer, 2016.

[72] J. Wang, Y. Chen, S. Hao, X. Peng, and L. Hu, "Deep learning for sensor-based activity recognition: A survey," *Pattern Recognition Letters*, vol. 119, pp. 3–11, 2019.

[73] P. Panda, A. Sengupta, and K. Roy, "Conditional deep learning for energy-efficient and enhanced pattern recognition," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 475–480, IEEE, 2016.

[74] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd International Conference on Pattern Recognition (ICPR)*, pp. 2464–2469, IEEE, 2016.

[75] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, pp. 1135–1143, 2015.

[76] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[77] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, jul 2012.

[78] A. Wang, L. Chen, and W. Xu, "Xpro: A cross-end processing architecture for data analytics in wearables," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pp. 69–80, 2017.

[79] R. Goldman, K. Bartleson, T. Wood, K. Kranen, C. Cao, V. Melikyan, and G. Markosyan, "Synopsys' open educational design kit: capabilities, deployment and future," in *Microelectronic Systems Education, 2009. MSE'09. IEEE International Conference on*, pp. 20–24, IEEE, 2009.

[80] B. Bruno, F. Mastrogiovanni, A. Sgorbissa, T. Vernazza, and R. Zaccaria, "Analysis of human behavior recognition algorithms based on acceleration data," in *2013 IEEE International Conference on Robotics and Automation*, pp. 1602–1607, IEEE, 2013.

[81] J. R. Kwapisz, G. M. Weiss, and S. A. Moore, "Activity recognition using cell phone accelerometers," *ACM SigKDD Explorations Newsletter*, vol. 12, no. 2, pp. 74–82, 2011.

[82] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "A public domain dataset for human activity recognition using smartphones.," in *Esann*, vol. 3, p. 3, 2013.

[83] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of machine learning research*, vol. 3, no. Mar, pp. 1157–1182, 2003.