공학박사학위논문

Performance Modeling, Performance Tuning and Quantization for GPU Programs

GPU 프로그램을위한 성능 모델링, 성능 튜닝 및 양자화

2021 년 8 월

서울대학교 대학원

전기·컴퓨터 공학부

Thanh Tuan Dao

Performance Modeling, Performance Tuning and Quantization for GPU Programs

GPU 프로그램을위한 성능 모델링, 성능 튜닝 및 양자화

지도교수 이재진

이 논문을 공학박사학위논문으로 제출함

2021 년 4 월

서울대학교 대학원

전기·컴퓨터 공학부

Thanh Tuan Dao

Thanh Tuan Dao의 공학박사 학위논문을 인준함

2021 년 7 월

Abstract

GPUs have played an important role in solving many scientific problems that range across different domains. Writing GPU programs might be easy, but writing them efficiently is much more difficult. To achieve the best performance, it is necessary that the compiler and runtime have advanced techniques to compile and run the program efficiently. These techniques should be transparent to the programmers and help them avoid the burden of having to know many details of the underlying architecture. Among the most important aspects that help improve the performance of a GPU program, we focus on the problem of performance modeling, performance tuning and quantization. Performance modeling estimates the execution time of the program and can be useful in analyzing the program characteristics or partitioning the workload in a heterogenous system. Performance tuning finds the optimal solution from an optimization space in a reasonable time. Quantization reduces the precision needed to execute the program without losing significant output accuracy. The proposed techniques can be integrated into GPU compilers and runtimes to help them be more efficient.

Keywords: Performance Modeling, Performance Tuning, GPU, Deep Learning, Quantization

Student Number: 2013-30839

Contents

Abstract **i**

1 Introduction **1**

1.1 Introduction 1

2 Performance Modeling **4**

2.1 Introduction 4

2.2 Related Work 8

2.3 Background 10

2.3.1 OpenCL Framework 10

2.3.2 GPU Architecture 11

2.3.3 Support Vector Regression 14

2.4 Prerequisites to efficient profiling: An insight to warp scheduling 16

2.5 Performance Estimation 23

2.5.1 Linear Model 24

2.5.2 Model based on Machine Learning 25

2.6 Evaluation 29

2.6.1 Evaluation Setup 29

2.6.2 Performance estimation results 30

2.6.3 The ML-based model on different classes of kernels 37

2.6.4 The performance at different saturation points 37

2.7 Conclusions 39

3 Performance Auto-tuning 41

3.1 Introduction 42

3.2 Related Work 45

3.3 OpenCL and GPU Architectures 47

3.4 Effects of the Work-group Size 49

3.4.1 Occupancy 50

3.4.2 Global Memory Coalescing 51

3.4.3 Cache Contention 56

3.4.4 Amount of Work 57

3.4.5 Work-group Scheduling and Barriers 58

3.4.6 Benchmark Applications 59

3.5 Auto-tuning Work-group Size 61

3.5.1 Workload Tuner 62

3.5.2 Non-coalescing Factor Tuner 64

3.5.3 Concurrency Tuner 66

3.5.4 Exhaustive-search Tuner 70

3.6 Evaluation 70

3.6.1 Overall Tuning Quality 70

3.6.2 Overall Tuning Cost 75

3.6.3 Effect of the Workload Tuner 76

3.6.4 Effect of the Non-coalescing Factor Tuner 77

3.6.5 Effect of the Concurrency Tuner 77

3.7 Conclusions 79

4 Quantization for Deep Learning Programs 80

4.1 Introduction 81

4.2 Related Work 83

4.3 Background 85

4.3.1 Integer Quantization 85

4.3.2 Standard Techniques Used 87

4.4 Quantization Framework 88

4.4.1 Inference Phase 88

4.4.2 Training Phase 89

4.4.3 Adding Noise to the Scale 89

4.4.4 Adaptively Adjusting Precisions 93

4.4.5 Computation of Histogram 97

4.5 Experiments 97

4.5.1 Image Classification Tasks 100

4.5.2 Natural Language Processing 105

4.6 Conclusions 106

5 Conculsion 107

Acknowledgements 123

List of Figures

2.1 OpenCL Architecture. 10
2.2 Schematic block diagram of one SM in the GTX 580 GPU. . . . 13
2.3 Relationship of work-groups, work-items, and warps. 13
2.4 Instructions per microsecond for the first 160 workgroups of a microkernel and `nbody`. 16
2.5 Start/finish times for the first 96 workgroups of a microkernel and `nbody`. 16
2.6 Micro-benchmark containing a loop with a (a) single-precision floating point addition (b) a double-precision floating point addition. 19
2.7 Progress within work-groups in the presence of a staircase. . . . 21
2.8 Micro-benchmark containing a loop with a (a) one (b) two and (c) three double-precision floating point additions. 22
2.9 Sampling at saturation points and the linear model. 24
2.10 Actions and information flow for both models. 28
2.11 Error rates of the linear and ML-based model on the GTX 580 for the execution time of each kernel. 31

2.12 Error rates of the linear and ML-based model on the GTX 280 for the execution time of each kernel. 31
2.13 Error rates of the linear model on the GTX 680 for the execution time of each kernel. 32
2.14 Error rates of the linear and ML-based model on the Radeon HD 6970 for the execution time of each kernel. 32
2.15 Average error rates of the linear and ML-based model at different saturation points. 38

3.1 Proposed auto-tuning framework. 43
3.2 Work-items, work-groups, and warps. 48
3.3 Relationship between the execution time and occupancy of some kernels from Rodinia and NVIDIA SDK. (a) `BFS_1` from Rodinia. (b) `MatVecMulCoalesce2` from NVIDIA SDK. (c) `Fan_2` from Rodinia. 50
3.4 The effect of memory coalescing in terms of occupancy. 52
3.5 The effect of non-coalesced memory accesses for kernel Fan_2 in Rodinia. 55
3.6 Micro-benchmark code to show the relationship between the work-group size and cache contention. 55
3.7 The effect of cache contention. 55
3.8 Transforming the one-dimensional index space in Figure 3.2 by the workload tuner. 62
3.9 The performance of the auto-tuner on GTX Titan X. 68
3.10 The performance of the auto-tuner on GTX Titan X with respect to the occupancy, the number of memory transfers and the instruction count. 68

3.11 The tuning cost of the auto-tuner on GTX Titan X. 68
3.12 The performance of the auto-tuner in comparison with OpenTuner on GTX Titan X. 72
3.13 The performance on FirePro W8000. 74
3.14 The effect of the workload tuner. 74
3.15 The effect of the non-coalescing factor tuner on GTX Titan X. . 74
3.16 The effect of the concurrency tuner on the kernels with high workload variation on GTX Titan X. 75
3.17 The effect of the concurrency tuner on the two-dimensional kernels on GTX Titan X. 75

4.1 The inference phase in our basic 8-bit integer (INT8) quantization framework. 88
4.2 The forward pass in the training phase of our basic INT8 quantization framework. 89
4.3 The histogram of input activations of layer relu0 in EfficientNet-B0 with CIFAR10. 90
4.4 The partial quantization scheme. 92
4.5 Validation accuracy comparison between different precisions used to quantize activations: (a) INT4 (4-bit integers) and (b) INT8. The weight precision is INT8 for all experiments. 94
4.6 8-bit validation accuracy of EfficientNet-B0 on CIFAR10 with the SGD optimizer. 103

List of Tables

2.1 List of features of the ML model for the GTX 280 and the GTX 580 26
2.2 OpenCL kernels 29
2.3 Performance for different error classes. 37

3.1 Hardware specifications of the GPUs used. 49
3.2 Characteristics of the kernels used in this paper. 60
3.3 Values for P and λ_0 for different GPU architectures. 67

1 The networks used in this paper. 98
2 Ablation Validation Accuracy (CIFAR10 and SGD) 101
3 Validation Accuracy for rand+mix for different histogram settings (CIFAR10 and SGD) 101
4 Validation Accuracy for rand+mix for different precision settings (CIFAR10 and SGD) 101
5 Validation Accuracy (CIFAR10 and SGD) 102
6 Validation Accuracy (ImageNet and SGD) 102
7 Validation Accuracy (CIFAR10 and Adam) 104
8 Validation Accuracy (ImageNet and Adam) 104

9 Validation Results for NLP Tasks 105

Chapter 1

Introduction

1.1 Introduction

Graphic Processing Unit (GPU) has gained tremendous popularity in the recent years and has been used for general purpose computing in many domains [1], called GPGPU. There is a huge gap in the theoretical peak bandwidth and the gigaflops performance between GPUs and CPUs nowadays [2]. It is due to the unique architectural featues of GPUs: It consists a massive number of processing units that is suitable for processing embarassingly parallel workload. Typicall, people can write their program using either CUDA or OpenCL programming languages, which are among the most popular languages for GPUs. However, achieving optimal performance, i.e., fully exploiting the redundant amount of computing resources available in the hardware, is a challenging task. The difficulties stem from many performance factors that can occur inside a program and how they can interact with each other. Therefore, a large body of research focus on understanding the performance characteristics of the GPU architecture and modeling the performance of GPU programs [3, 4, 5, 6, 7, 8, 9, 10, 11]. Such techniques are especially crucial in heterogenous systems that partition the workload into different hardwares of different types. To achieve the optimal partitioning, the performance of the program needs to be estimated [12, 13]. Performance modeling, or performance estimation, for GPU programs is difficult due to the lack of understanding the underlying architectural information from the official documentation that the GPU vendors release. For example, how multiple ready-to-execute threads are scheduled (or context-switched) on the same Streaming Multiprocessor is not documented, meanwhile this type of information is critical to construct an accurate predictive model for GPU programs. In the next chapter of this thesis, we propose insightful techniques to accurately model the performance for GPUs.

Currently, there are three main GPU vendors. They are AMD, Intel and NVIDIA. Each of them has their own ecosystem to support their products. These ecosystems include a great number of compilers, profilers and libraries that complicate effort of writing portable performance codes. Even for the same vendor, the fast change in the architecture between GPU generations also poses a challenge to achieve portable performance. Normally, a code that is optimized specifically for a device will not delivery comparable performance for another device and needs to be re-optimized. If the optimization space is large, finding optimal options might be very time consuming. Hence, the performance autotuners come to the play with the ability to select the best options much faster than an exhaustive search. One representative example of the tuning options for GPU programs is the work-group size. Chapter 3 presents a set of techniques to construct an efficient autotuner for GPU programs that can select a good work-group size within a reasonable time.

Modern runtimes [14, 15, 16] do not only support program optimization by parallelization or finding the best runtime options but they also support trading the precision with performance. This process is called quantization [17, 18, 19, 20, 21]. Quantization has played an important role not only for GPUs but also for other high performance devices [22] to improve the performance and reduce the required memory size. This is even more critical for Deep Learning workloads, where the model size can be a hurdle to deploy many state-of-the-art networks onto devices with limited memory capability. For example, the state-of-the-art language model GPT-3 [23] has 175 billions parameters that already exceed the memory capacity of many recent high-end GPUs. Chapter 4 presents an effort to quantize Deep Learning workloads into 8-bit precision using the integer-based quantization.

Chapter 2

Performance Modeling

2.1 Introduction

Today, the word's second-fastest supercomputer, Titan, and many more on the Top500 list [24] are heterogeneous systems comprising both CPUs and GPUs. In order to achieve optimal performance the workload needs to be distributed evenly to the different computing nodes, and to do so an accurate performance model is required.

Static workload distributions [12, 13] based on throughput and threshold values are often far away from the optimal distribution because the performance difference between CPUs and GPUs heavily depends on the program characteristics. A performance estimation model can be used to dynamically distribute the workload inversely proportional to the estimated execution time. While performance modeling for general-purpose CPUs has been researched actively and accurate performance models are available, the state-of-the-art models for GPUs still suffer from a relatively large estimation error. Models for older GPU architectures do not consider the GPU's hardware caches [7, 5, 6], and many are not suited for performance estimation at runtime [3, 4, 5, 6].

A performance model suitable for runtime workload distribution thus needs to capture and characterize the intricate interactions of the most important hardware and software features of modern GPUs. This is undoubtedly a challenging problem because of the GPU's complex hardware that enables its massively parallel processing capability. Aside from a large number of processors, each with multiple scalar execution units, the GPU's hardware thread context switching mechanism and the on-board memory subsystem with different levels of cache memories further complicate the task. Several approaches [5, 9, 7, 11, 8, 6] have been proposed to model GPU performance, including analytical modeling and Machine Learning (ML)-based modeling. The analytical modeling approaches [5, 7, 8, 6] typically rely on micro-architecture information to predict the performance of a program in a handcrafted manner. As GPU architectures continue to evolve dramatically this approach is not that attractive: a minor change in the architecture may require extensive work to adapt the model to the new architecture. On the other hand, instead of evaluating a number of predefined formulas and quickly reporting the execution time, ML models [9, 10, 11] rely on training data to learn the mapping between program features and execution time. ML-based approaches seem to be more appropriate for this task as they are more robust to changes in the GPU architecture.

In this chapter, we present two models to accurately predict the performance of an OpenCL kernel on GPUs. Both the linear and the ML-based model rely on sampling information to overcome the limitations of analytical methods. Sampling incurs an overhead because a small part of the actual workload has to be executed before the performance of the entire workload can be predicted; thus, it is important to keep the sampling overhead to a minimum. In order to determine the earliest sampling points that allow an accurate estimation, it is essential to have a clear understanding how GPUs schedule a workload. Unfortunately, GPU manufacturers do not disclose how the work is distributed to the different compute units of a GPU, or how the more than 1000 active threads are scheduled inside these units. We thus reverse-engineer the scheduling policies of GPUs by analyzing the results of a set of hand-crafted OpenCL micro-benchmarks. Based on these observations we formulate and verify the scheduling policies of modern NVIDIA GPUs which then form the basis of the two models.

The analysis of the GPU's scheduling policy allows us to compute the minimal number of work-items at which the GPU attains maximum throughput. The sampled data is used by a linear model to extrapolate the total execution time of the entire kernel. This linear model works reasonably well because the amount of work per work-item in scientific workloads is typically distributed evenly. On modern GPUs with on-chip caches and memory coalescing, however, the execution time shows much more variance. To cope with such architectures we combine the linear model with an ML-based approach that takes the data of the GPU's hardware performance counters as additional inputs. Training the ML-based model with over 300 data sets allows the model to detect the complex correlations between a workload's sampled data and the actual execution time. The experiments show that the ML-based model is able to improve the accuracy of the linear model significantly.

We evaluate the accuracy of the proposed model with 70 different OpenCL kernels by comparing the estimated kernel execution time with the actual measured execution time. We evaluate the model on different GPU architectures from two representative GPU vendors: NVIDIA and AMD. On the GTX 580 [25], a representative of NVIDIA's Fermi GPU-architecture, we achieve, on average, an error rate of 5.72% for the linear model and 4.76% for the ML-based model. To demonstrate that our model is not tied to a certain GPU architecture, we have applied it to an NVIDIA GTX 280 and an NVIDIA GTX 680. The GTX 280 is a second-generation GPU with no hardware caches. The model achieves an error rate of 7.19% for the linear and 6.42% for the ML-based model. The GTX 680 is based on NVIDIA's latest Kepler architecture [26]. It is currently impossible to read the GPU's performance counters from OpenCL, hence we were only able to run the linear model. An error rate of less than 10% shows that the linear model, although simple, can cover a wide range of GPU generations with acceptable error rates. Interestingly, on the AMD Radeon HD 6970, the model achieves an error rate of 8%. This shows that although the model is formed based on an analysis on a specific NVIDIA GPU, it could be used with not only different GPU generations from NVIDIA, but also from AMD.

The contributions of this chapter are:

- An in-depth analysis of scheduling policies of NVIDIA GPUs that allows us to determine the sampling points with the fewest number of work-items that still allow an accurate performance estimation.
- A simple linear performance model that is very accurate for applications with an evenly distributed workload.
- An ML-based performance model that can cover a wide range of applications and significantly improve the performance of the linear model.
- An implementation and evaluation of the models with 70 OpenCL kernels from four different benchmark suites and OpenCL kernel collections on four different generations of GPU architectures from two different vendors.

The rest of this chapter is organized as follows. Section 2.2 reviews related work. Section 2.3 contains a brief introduction to the OpenCL execution model and the Fermi GPU architecture. In Section 2.4, we provide a comprehensive analysis of the way modern GPUs execute kernels. The performance estimation models described in detail in Section 2.5 are built upon these observations. Section 3.6 evaluates and compares the accuracy of our models, and Section 2.7 concludes this chapter.

2.2 Related Work

Wong *et al.* [27] have analyzed the NVIDIA GTX 280 and revealed a number of undisclosed characteristics by running micro-benchmarks. However, their analysis does not consider contention and does thus not provide much information about how warps are scheduled. Liu *et al.* [8] have developed a performance predictor for a specific application. Although the prediction is precise, this method requires a priori knowledge of the application's throughput. Bitirgen *et al.* [10] proposed an interval-based framework to dynamically select resource allocation decisions on chip multiprocessors. Their model takes program behavior as input and estimates the performance of the program at certain intervals. Though interesting, this interval-based approach is not applicable to GPU performance estimation.

Zhang and Owens [6] have developed a performance model for CUDA programs executing on NVIDIA GeForce 200-series GPUs. Their main goal is to help the programmer identify performance bottlenecks, potential optimizations and architecture improvements. Static program features are collected by the compiler and dynamic features are collected in a simulator [28]. Together with machine-specific characteristics, these features are used to analytically calculate the execution time of CUDA applications. This overhead can easily exceed the running time of the kernel on a GPU itself. Jia *et al.* [4] proposed a regression-based performance model for GPU design space exploration. Training this model requires executing the program over a large number of design points which makes this approach unsuitable for workload distribution.

Kerr *et al.* [29] have proposed a Machine Learning based model to predict performance of CUDA programs on GPUs and CPUs (based on Ocelot compiler). The static features of the program are used to derive the relationship between the program behavior and the performance on a target architecture. This model uses static features of the program so it is limited to an average error of 30%. Luk *et al.* [30] have proposed a linear regression-based model to distribute workload to a heterogeneous system of CPUs and GPUs. Unlike the proposed technique, the linear regression-model needs to be trained for each new kernel encountered.

Hong and Kim [7], Baghsorkhi [5] and Grewe [31] have presented performance models that can be used for workload distribution. These models are based on static information of CUDA/OpenCL programs and require the programs to be written in a parametrized way so that variables related to the problem size can be analyzed symbolically. Another drawback of these models is that they do not consider the presence of caches and are thus not applicable to modern GPUs with caches. Finally, none of these models take into account the interaction between the application and the hardware as well as the effect of compiler optimizations.

The models proposed in this chapter remove the restrictions of static analysis by using dynamic information through sampling. Experiments with a large number of OpenCL kernels from a wide range of applications show that the models can be easily applied to GPU from different architectures and produce a more accurate performance estimate with an average error of less than 5% for the GTX 580, 7% for the GTX 280, 10% for the GTX 680 and 8% for the HD 6970.

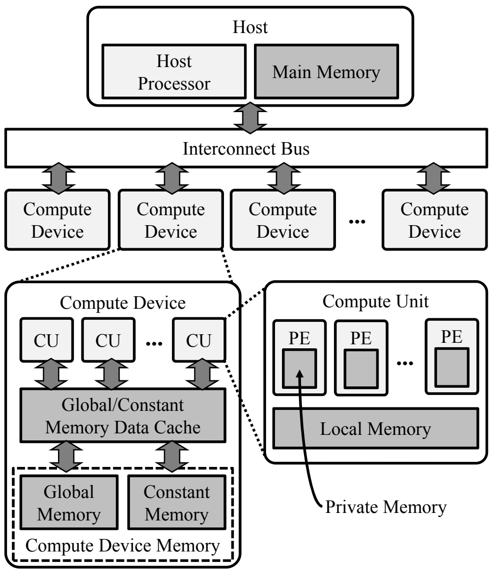


Figure 2.1: OpenCL Architecture.

2.3 Background

2.3.1 OpenCL Framework

In this section, we briefly introduce the OpenCL framework, the NVIDIA's Fermi GPU architecture [32], and provide a short introduction to the Machine Learning techniques applied in this work. To make the discussion consistent we use OpenCL terms to describe the GPU architecture.

In the OpenCL platform model [33] a host processor is connected to one or more *compute devices*. A compute device contains a number of *compute units* (CUs), each of which contains one or more *processing elements* (PEs) (Figure 2.1).

An OpenCL application consists of a *host program* and one or several *kernels*. The host program is executed on the host processor, and the kernels are executed on the devices. For each kernel, the host defines an N-dimensional abstract index space in which the kernel will be executed ($N \in \{1, 2, 3\}$). Each point in this space defines one execution instance of the kernel, called *work-item*. The work-items are organized into groups of equal size, called *work-groups*. The work-groups are executed independently, i.e., concurrently and in any order.

There are four different types of memory accesses in an OpenCL kernel: global memory, constant memory, local memory, and private memory. Global and constant memory accesses have the highest latency since the accessed data resides in the device global memory. The local memory is shared by all PEs in the same compute unit. The private memory is local to a PE. Accesses to the global memory or the constant memory may be cached in the global/constant memory data cache if there is such a cache in the device.

2.3.2 GPU Architecture

NVIDIA's Fermi architecture [32] is designed for massive parallel processing. It comprises a fairly large number of streaming multiprocessors (SMs). Each SM contains 32 streaming processors (SPs) for general computations, 16 load/store (LD/ST) units, and four Special Function Units (SFUs) that handle transcendental operations. There are two levels of data caches: an L1 cache for each SM and an L2 cache shared by all the SMs. Figure 2.2 shows a schematic block diagram of one of the total 16 SMs in the GTX 580 GPU. In OpenCL terminology, the GPU represents a compute device, an SM corresponds to a CU, and an SP to a PE.

When running a work-group on an SM, each work-item is executed by one thread. Threads are bundled into so-called *warps*. One warp consists of 32 threads (or work-items; Figure 2.3). All threads in a warp execute in lock-step. The GPU compiler thus converts conditionally executed code (such as *if-else*-constructs) into predicated code. A *divergence* describes the situation when, at run-time, not all 32 threads take the same control path. When divergence occurs, both parts of the conditionally executed code must be executed. Divergence inevitably leads to reduced performance.

The work-group is the minimal allocation unit of work for an SM. A work-group running on an SM is called an *active work-group* [34]. Similarly, *active warps* denote warps that are currently executing or eligible for execution. If enough hardware resources are available, several work-groups can be active simultaneously in one SM. The number of active work-groups depends on the kernel code, the kernel's index space, and the hardware resources of the SM. The metric to characterize this degree of concurrency is termed *occupancy* [35]. Occupancy is defined as the ratio of the actual number of active warps to the maximum possible number of active warps per SM. This maximum is hardware specific; in the case of the GTX 580, at most 48 warps (and thus $48 * 32 = 1536$ threads or work-items) can be active in one SM at the same time.

The work-groups and warps in a work-group are formed from consecutive work-items. If the size of a work-group is not divisible by the warp size (i.e., 32), every work-group contains one warp that has less than 32 work-items. Thus, to maximize resource utilization, the number of work-items in a work-group should always be a multiple of 32.

The active work-groups that execute concurrently in an SM might have different starting and finishing times. As soon as one work-group finishes, the SM scheduler activates a new work-group. The 32 work-items of a warp are conceptually executed in one cycle. Recall from Figure 2.2 that one SM in the Fermi architecture contains 32 SPs for general-purpose instructions but only 16 load/store units. Every time a warp issues a load/store instruction, its execution would thus have to be split up into two cycles. Instead of doing so, every SM contains *two* warp schedulers that, in every clock cycle, schedule two *half-warps*

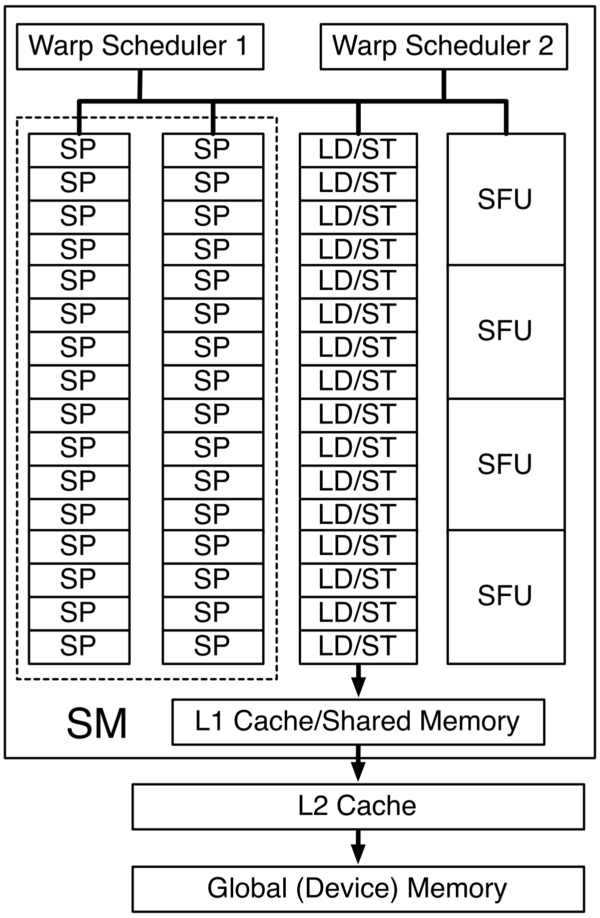


Figure 2.2: Schematic block diagram of one SM in the GTX 580 GPU.

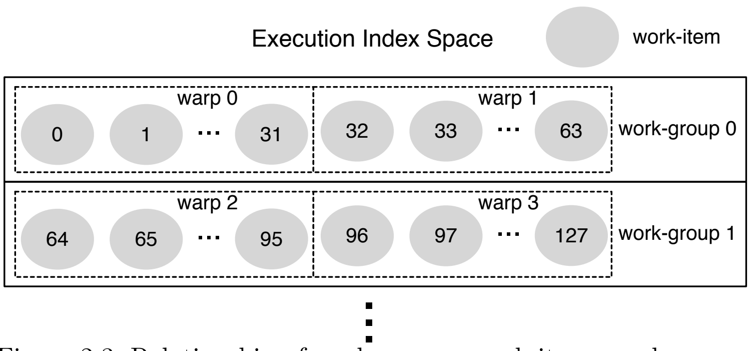


Figure 2.3: Relationship of work-groups, work-items, and warps.

from different warps [34]. One scheduler handles warps with even IDs, and the other one handles warps with odd IDs. Both schedulers issue the instruction to one of the four execution units (2 x 16 SPs, 1 x 16 LD/ST units, 1 x 4 SFUs). Such a setup uses the hardware resources more efficiently and does not require extra management of half-way executed warps because, for example, a general-purpose instruction from one half-warp and a memory operation from the other half-warp can be executed in parallel. However, for transcendental operations that are scheduled on the 4 SFUs, a half-warp will require at least four cycles to be scheduled.

Each SM contains an instruction cache and 64KB of local data memory that can be dynamically configured as scratchpad memory or L1 cache. A unified L2 data cache of 768KB is shared by all SMs in the GPU.

2.3.3 Support Vector Regression

Here, we briefly introduce Support Vector Regression (SVR), the regression learning algorithm used in the ML-based model. We use a non-linear form of a supervised learning algorithm called ϵ-SVR [36]. In this form, the model takes a *feature vector* as its input; this is simply a vector of input parameters. In a *learning phase* the model is trained with a large number of input vectors and the corresponding output values. The model learns the complex interactions between the features of the input vector and the desired objective. We use the *leave-one-out-cross-validation* technique: we train the model with training data from which all data points of the kernel under test have been removed, and then apply the model to the kernel. This method guarantees that the training data set and the test data set are distinct which is important if we only have a limited number of training data points.

In our context, the input data comprises a set of performance features of an OpenCL kernel (provided by the compiler, and obtained through sampling), and the objective is the execution time of kernel.

For the interested reader, the following paragraph provides some more details about ϵ-SVR. To keep the discussion reasonably simple, we describe the linear form; the basic idea is identical in the non-linear form. The training phase is performed on a collection of training data. A data instance has the form $\{x, y\}$ where x is a collection of d representative features for this data instance and y is the objective that is associated with this data instance. The goal is to find a function $f(x)$ that deviates by no more than ϵ from the training target y_i for all y_i while being as flat as possible. In the linear problem statement, the function $f(x)$ has the form

$$f(x) = w \cdot x + b$$

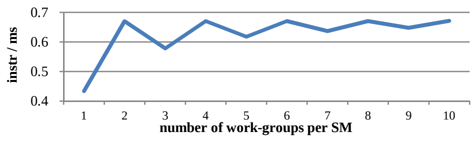
where w denotes the normal vector to the hyperplane and b is the bias with $w \in R^d, b \in R$. The *flatness* in this case means that w is small [37]. Denoting the training data as $(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)$, the objective then becomes

$$\text{minimize} \quad \frac{1}{2}\|w\|^2 \tag{2.1}$$

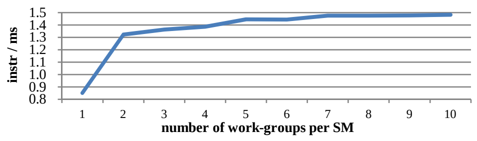
$$\text{subject to} \quad |w \cdot x_i + b - y_i| \leq \epsilon, \forall i \tag{2.2}$$

This convex optimization problem can be solved using its dual formulation by introducing Lagrange multipliers [36]. Condition 2.2 assumes that such a function $f(x)$ exists; however, this is often not the case for large datasets. To overcome this limitation, soft margin SVR [38] was introduced. Details of solving the dual formulation and soft margin SVR are out of the scope of this chapter.

We chose ϵ-SVR because it generalizes well to unseen data. In the training step, rather than only minimizing the observed training error, ϵ-SVR finds the trade-off between the error and the complexity of the objective function. Once

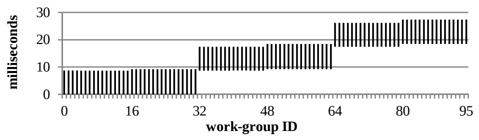


(a) microkernel IPS

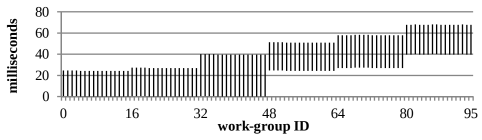


(b) nbody IPS

Figure 2.4: Instructions per microsecond for the first 160 workgroups of a microkernel and `nbody`.



(a) microkernel start/finish times



(b) nbody start/finish times

Figure 2.5: Start/finish times for the first 96 workgroups of a microkernel and `nbody`.

the function $f(x)$ has been determined, it can be used to predict the objective value for yet unseen feature vectors x.

2.4 Prerequisites to efficient profiling: An insight to warp scheduling

In order to accurately estimating the runtime performance of an OpenCL kernel, a deep understanding of the inner workings of a GPU is necessary. In addition, a sampling-based model must keep the number of sampled work-groups to a minimum. This section presents our analysis of the NVIDIA GTX 580 scheduler and provides the foundation for the performance model presented in the following section.

From pieces of information about GPU scheduling gathered from various sources [25, 39, 32, 35] we first state and verify the following two assumptions:

1. On the device-level, the work-group scheduler assigns a new work-group to an SM as soon as an active work-group on that SM finishes executing.

2. On the SM-level, the two warp-schedulers use a round-robin scheduling policy to schedule the active warps. Whenever a warp blocks or has executed for a certain number of cycles, the scheduler picks a new warp.

To accurately measure the execution time of work-items, we add instrumentation code to the very beginning and the very end of real-world and several hand-crafted kernels. The code reads the GPU's clock cycle performance counter. This instrumentation allows us to record the start time, $s_{wi}(i)$ and finish time, $e_{wi}(i)$ for each work-item i. The execution time is then given as $t_{wi}(i) = e_{wi}(i) - s_{wi}(i)$. For a warp w containing N work-items, let

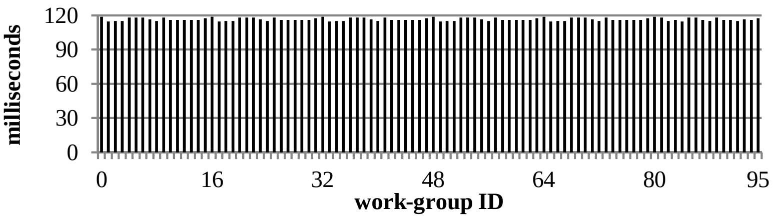
$$\begin{aligned} s_{warp}(w) &= MIN_{k=1}^{N} s_{wi}(k) \\ e_{warp}(w) &= MAX_{k=1}^{N} e_{wi}(k) \\ t_{warp}(w) &= e_{warp}(w) - s_{warp}(w) \end{aligned}$$

The start, finish and execution time for work-groups comprising M warps is defined accordingly:

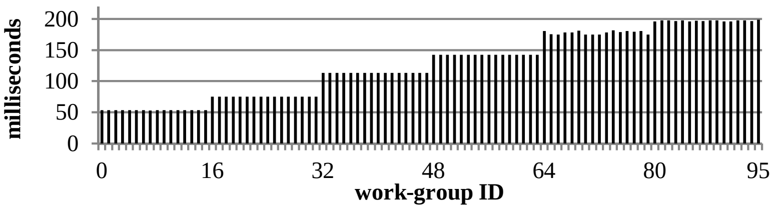
$$\begin{aligned} s_{wg}(j) &= MIN_{k=1}^{M} s_{warp}(k) \\ e_{wg}(j) &= MAX_{k=1}^{M} e_{warp}(k) \\ t_{wg}(j) &= e_{wg}(j) - s_{wg}(j) \end{aligned}$$

Recall that several work-groups can be active within one SM. The compiler computes a kernel's occupancy as a by-product of compiling. Based on the occupancy we can compute how many work-groups can be active simultaneously within on SM. For example, with an occupancy of 1.0 and work-groups containing 256 work-items each, six work-groups can be active at the same time (one work-group contains $256/32 = 8$ warps, and to achieve an occupancy of 1.0, we need 6 work-groups with 8 warps to get 48 warps, the maximum number of active warps per SM). Similarly, if the occupancy is 0.833, five work-groups can be active at the same time. We call the number of active work-groups per SM for a given kernel the *saturation point*, denoted P_{sat}.

When executing one, then two, then up to P_{sat} work-groups on one SM we expect that the throughput, i.e., the number of issued instructions per second (IPS) increases up to the saturation point. At that point, the running kernel exploits as much of the hardware resources as possible, and the warp schedulers within the SM have the biggest freedom when picking ready-to-run warps. Micro-benchmarks consisting only of ALU operations and no memory accesses indeed show the expected behavior. For more complex kernels, however, the IPS can increase even after the saturation point due to cache warm-up effects. In Figure 2.4 (a), the saturation point of the microbenchmark is two, and indeed the throughput achieves a stable maximum at multiples of P_{sat}. In between saturation points, maximum throughput is not achieved because not all hardware resources are fully utilized. For `nbody` in Figure 2.4 (b), the kernel from the NBody application, the saturation point P_{sat} is 3, yet, the throughput clearly increases after the first saturation point. This increase is caused by cache warm-up effects, i.e., the first work-groups suffer from more cold misses and can profit less from memory coalescing than later work-groups. Figures 2.5 plots the start and finish time for each work-group for the microbenchmark and `nbody`. In the case of the microbenchmark, $P_{sat} = 2$ so the the first 32 work-groups are started at the same time (each of the 16 SMs receives two work-groups). We observe that the work-group scheduler distributes the first 16 work-groups one-by-one to each of the 16 SMs. Work-groups 16 to 31 are again evenly distributed to all SMs, and so on. For `nbody` in Figure 2.5(b) $P_{sat} = 3$ so initially, each SM is given three work-groups. These correspond to the first $16 * 3 = 48$ work-groups. For both benchmarks, we observe that whenever a work-group finishes the work-group scheduler immediately assigns a



(a)



(b)

Figure 2.6: Micro-benchmark containing a loop with a (a) single-precision floating point addition (b) a double-precision floating point addition.

new work-group to the corresponding SM. This confirms assumption 1 about the work-group scheduler.

We expect all active work-groups in an SM to start and finish at the same time. Figure 2.5(a) confirms this assumption for the microbenchmark. In the case of `nbody`, however, we observe that the third work-group on each SM takes almost double the time to finish compared to the first work-group (Figure 2.5(b)). We call such a behavior a *staircase.*

We have run a significant number of hand-crafted micro-benchmarks that stress different types of instruction mixes in loops to explain the staircase. Figure 2.6(a) shows the behavior of a kernel that contains a single-precision floating point addition in a loop. There are a total of 96 work-groups in the kernel index space of this benchmark, with each work-group containing 256 work-items. The kernel's occupancy is 1.0, hence six work-groups or 48 warps are active at the same time per SM. In total, there are 16 SMs on the GPU, so all 96 work-groups can run in parallel. This explains why all work-groups start at time 0. Since there is no resource contention between warps, warps block very infrequently (e.g., only on changes in the control flow caused by iterating the loop). According to assumption 2, if a warp does not block, the scheduler will continue to run it until it exceeds a certain number of cycles. After this point the scheduler will schedule another warp that is ready-to-run. Since the kernel does not exhibit a staircase, the selection of the next warp seems to be round-robin.

To trigger resource contention, we replace the single-precision with a double-precision floating point addition. Single-precision floating point operations are executed on one SP, but for double-precision operations, the two rows of sixteen SPs are logically linked and operate as 16 double-precision SPs (Figure 2.2). If one of the half-warps executes a double-precision floating point operation, all 32 SPs are occupied and the other half-warp will block unless it executes a memory or transcendental operation. The execution time of the work-groups is shown in Figure 2.6(b) and clearly exhibits a staircase. If the warp schedulers schedule half-warps in a round-robin fashion, no staircase should appear since all warps will get an equal share of the SM.

To better understand how the warp schedulers select the next warps, we have added additional instrumentation code to the kernel. We not only measure the start and finish time of the kernel, but also record the time after processing 20%, 40%, 60%, and 80% of the workload. The result of this experiment is shown in Figure 2.7. The figure reveals that, while all work-groups start around time 0, they do not progress at the same speed. In fact, the third work-group on each SM does not even start running past the first couple of loop iterations before the first work-group has finished. The same is true for the fourth work-group (it effectively starts executing when the second one finishes), and the fifth

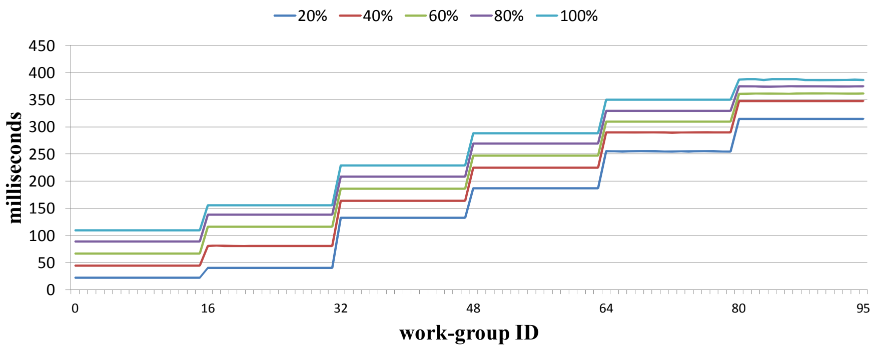


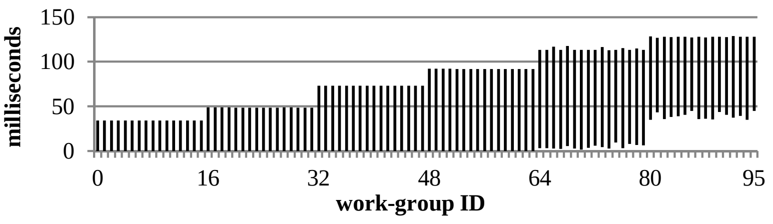
Figure 2.7: Progress within work-groups in the presence of a staircase.

work-group (when the third work-group finishes). Using this result, we refine assumption 2 on warp-scheduling:

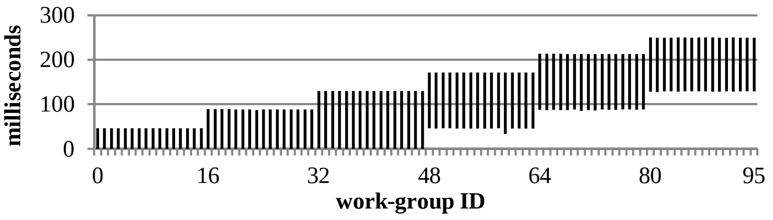
2. On the SM-level, the two warp-schedulers use a round-robin scheduling policy when the active warp has executed for a certain number of cycles. If a warp blocks, the scheduler selects the first warp that is ready-to-run from the work-group with the *lowest ID*.

With this refined assumption, we are able to explain kernels that do not exhibit staircase behavior because warps seldom block and thus run more or less at the same speed thanks to the round-robin scheduling policy. In high-contention situations, however, the warp schedulers give higher priority to earlier work-groups by picking the first warp that is ready to run from the work-group that has been active for the longest time (i.e., has the lowest ID).

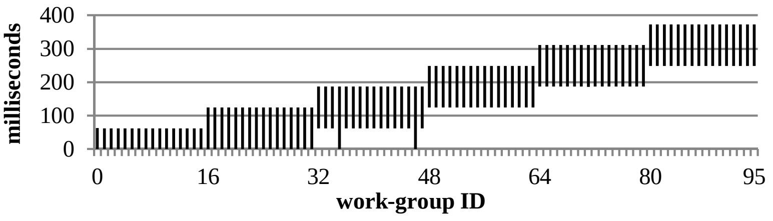
Clearly, the instruction mix will have an effect on what warp will be selected. Figure 2.8 shows the start and end times of all 96 simultaneously active work-groups. We run a loop with one, two, and three double-precision floating point additions in the loop body. We anticipate that the staircase is more pronounced



(a)



(b)



(c)

Figure 2.8: Micro-benchmark containing a loop with a (a) one (b) two and (c) three double-precision floating point additions.

the higher the resource contention. This behavior is indeed observed in Figures 2.8 (a-c): in the case of only one double-precision addition, the mix between operations that cause a warp to block and those that do not is still balanced so that in most cases the scheduler can run a warp until its time quantum expires. As we add more double-precision additions in Figure 2.8(b-c), the contention becomes more severe. The warp schedulers now clearly favor work-groups with a low ID. Indeed, for the case with three double-precision additions, the later work-groups will not even start executing before the first work-group has finished executing (Figure 2.8(c)).

A warp can block for several reasons: it blocks when it encounters an instruction whose operands are not available (data dependency), when the required execution units are occupied (hardware contention), or when control flow occurs. We observe that there are three types of operations that can cause hardware contention: double precision instructions, transcendental instructions (such as sine, cosine, exp, etc.), and memory operations. However, whether the warp will actually block or not also depends on the *instruction mix* of the kernel. If the kernel contains a well-balanced mix of instructions that are handled by different hardware resources, even a kernel with many double-precision floating point operations may not exhibit a staircase.

The following section details the construction of the linear model, the ML-based model and, in particular, the points at which the workload is sampled based on the observations on the the scheduling mechanism.

2.5 Performance Estimation

Based on our analysis of the execution model of Fermi GPUs in the previous section, we first construct a linear model to estimate the execution time of a workload. This simple linear model, however, cannot capture the non-linear effects of memory accesses on performance. Since recent GPUs such as those based on the Fermi architecture [32] implement several levels of hardware caches, we then propose a model based on machine learning techniques to improve the prediction accuracy for modern architectures and non-linear benchmarks. The ML-based model, guided by the linear model, has more relaxed assumptions than the linear model and can thus make a more accurate execution time estimation for the general case.

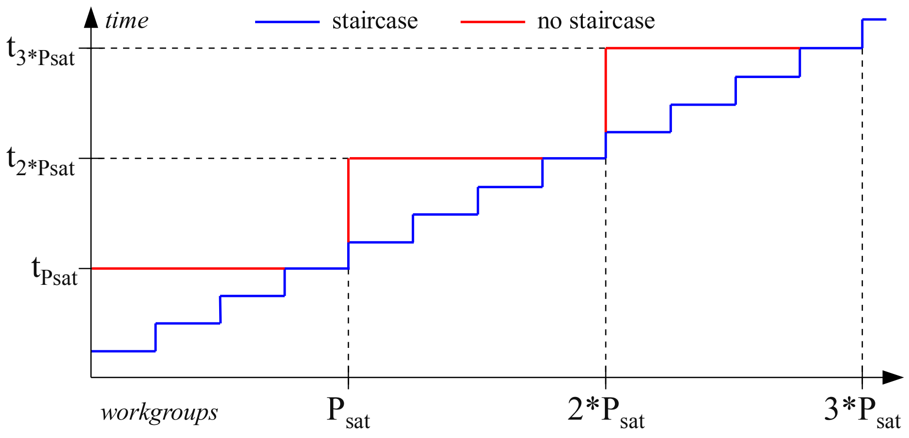


Figure 2.9: Sampling at saturation points and the linear model.

2.5.1 Linear Model

The linear model estimates the total execution time of a kernel by extrapolating the execution time obtained from two sampling points. Our goal is to choose two sampling points with as small a number of work-groups as possible while maintaining an acceptable accuracy for the estimation to keep the overhead to a minimum.

The previous section shows that even though a kernel fully utilizes the hardware resources of a GPU at the saturation point, P_{sat}, the throughput can increase even after P_{sat} due to warming-up effects of the L1- and L2-caches on the GPU (Figure 2.4(b)). We thus sample each kernel *twice*, once at P_{sat1} and once at P_{sat2}, where both P_{sat1} and P_{sat2} are multiples of P_{sat}. The execution time of the first sample, t_{Psat1}, determines the displacement, and the difference in execution time from the second sample to the first, i.e., $t_{Psat2} - t_{Psat1}$ represents the slope. Sampling at integer multiples of the saturation point has the added advantage that the staircase behavior can be ignored in the linear model. In fact, the staircase as explained in Section 2.4 only manifests itself *in between* two saturation points, at (integer multiples of) the saturation point the sampled execution times are identical. Figure 2.9 visualizes this observation. We have found that $P_{sat1} = 2 * P_{sat}$ and $P_{sat2} = 3 * P_{sat}$ are sufficient to avoid the non-linearity of the first few work-groups.

Using two sampling points at multiples of P_{sat}, the execution time of a kernel can be modeled as follows. The first sampling point determines the displacement, and the difference in execution time from the second to the first sampling point yields the increment (Figure 2.9)

$$t_{linear}(N) = t_{Psat1} + \frac{t_{Psat2} - t_{Psat1}}{P_{sat2} - P_{sat1}} \times (N - P_{sat1})$$

Here, $t_{linear}(N)$ denotes the total execution time of the kernel for N work-groups. t_{Psat1} and t_{Psat2} stand for the sampled execution time at saturation points P_{sat1} and P_{sat2}, respectively.

This simple linear model accurately estimates the execution time of a kernel when the following two conditions hold: (1) the workload is evenly distributed and (2) the memory access patterns are similar for all work-groups. The accuracy drops for unevenly distributed workloads or workloads that exhibit a variance in performance caused by caching or memory coalescing effects. The ML-based model tries to eliminate these shortcomings.

2.5.2 Model based on Machine Learning

For the ML-based model, we break the task of estimating the execution time into two smaller tasks. First, we estimate the saturated instructions per second (IPS) of a kernel using a machine learning algorithm. By instructions per seconds we refer to the number of issued instructions per second, not the number of executed instructions. The difference between issued instructions and executed instructions is that issued instructions include instructions that are serialized due

Feature	Description	Source	GPU Model
active work-groups	number of work-groups that can be active at the same time	compiler	GTX 280/580
registers per work-item	number of registers used by a work-item	compiler	GTX 280/580
execution time	execution time of the run	performance counter	GTX 280/580
branches	number of branches	performance counter	GTX 280/580
work-items	number of sampled work-items	performance counter	GTX 280/580
divergent branches	number of divergent branches	performance counter	GTX 280
L1/L2 miss rate	ratio of L1/L2 miss count to the issued instructions count	performance counter	GTX 580
total L1 accesses	L1 miss count plus L1 hit count	performance counter	GTX 580
issued instructions	number of issued instructions	performance counter	GTX 280
executed instructions	number of executed instructions	performance counter	GTX 280/580
bank conflicts	bank conflict count	performance counter	GTX 580
shared loads/stores	shared load/store count	performance counter	GTX 580
global loads/coalesced	number of global total/coalesced load requests	performance counter	GTX 280
warp serializes	number of warp serializes	performance counter	GTX 280
IPS	IPS obtained by the linear model	computed	GTX 280/580

Table 2.1: List of features of the ML model for the GTX 280 and the GTX 580

to hardware contention, memory conflicts, or divergence. Serialization seriously affects performance, thus using issued instructions improves the accuracy of the model. After approximating the saturated IPS, we compute the total number of instructions for the whole kernel and can easily calculate the total execution time. The ML-based model makes less assumptions and covers a wider range of kernels than the linear model since it only requires the kernel to distribute the workload evenly between the work-groups.

Estimation of the saturated IPS. The features of a kernel comprise static features collected by the compiler and dynamic features obtained during sampling runs on the GPU. We use the data from the same two runs as the linear model (at the second and the third saturation point). Table 2.1 lists the kernel features that were used to construct the IPS estimation model for the GTX 580 and the GTX 280.

Training and deploying the learning model. An ML-model requires a large amount of training data during the training phase. To obtain a sufficient amount of data from various kernels, each kernel is run several times, every time with a different number of work-groups. Since the performance factors of a kernel often fluctuate when executing the first few work-groups, several runs at a varying number of work-groups help cover all behavioral differences of the kernel.

Each run produces a set of performance counters. We combine these values with the static features obtained from the compiler and the result produced by the linear model using the second and the third saturation point to form a feature vector. The feature vector is associated with the saturated IPS of the kernel to form the training data. The saturated IPS is obtained by running the kernel with the largest possible input data and recording the IPS.

Estimating the total execution time. Calculating the total execution time for a kernel is straightforward. The features of a kernel include: the static features of the kernel; the performance data obtained from the two sampling runs; and the result given by the linear model. The features are combined to form a feature vector that is fed into ML model. The model outputs the estimated IPS for one work-group. Since we assume that the workload is evenly distributed between the work-groups, we simply multiply the estimated IPS by the number of work-groups as follows

$$t_{ML}(N) = \frac{I_{Psat2} \times N}{P_{sat2} \times IPS_{est}}$$

Here, $t_{ML}(N)$ denotes the total execution time of the kernel for N work-groups. I_{Psat2} is the number of instructions issued on one SM at P_{sat2} work-groups that is recorded when evaluating the linear model. Using sampling information at P_{sat2} leads to more accurate results than at P_{sat1} because of lessened warming-up effects and therefore more precise performance counter data. IPS_{est} is the estimated IPS computed by the ML model.

To summarize, as illustrated in figure 3.1, we describe the processing routine to test an OpenCL kernel (that is not included in the training set):

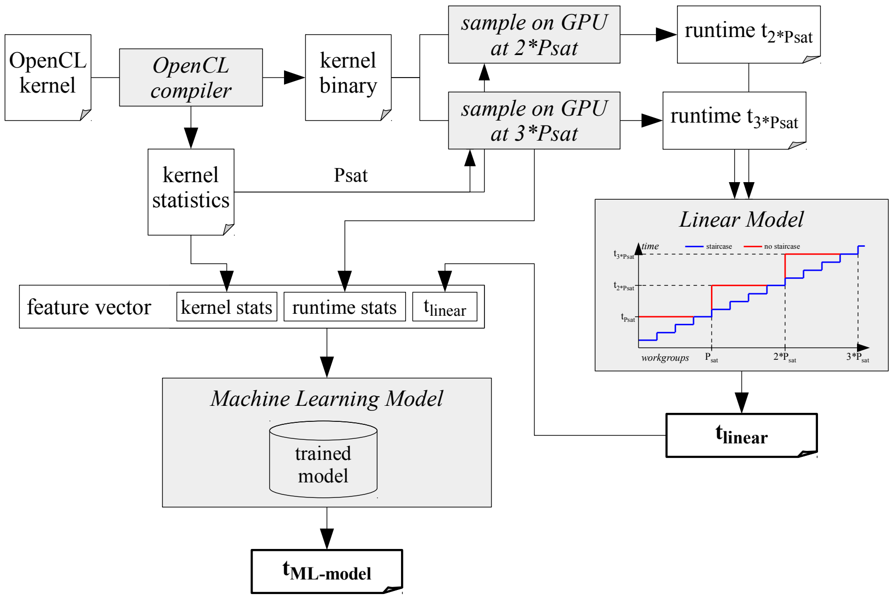


Figure 2.10: Actions and information flow for both models.

1. Based on occupancy and work-group size, analytically calculate the number of concurrently active work-groups, P_{sat}.

2. Extract the data buffer needed by the first work-groups at both sampling points and copy it to the device memory.

3. Sample at P_{sat1} and P_{sat2} work-groups and record the performance counters.

4. Evaluate the linear model to obtain the estimated execution time and compute IPS_{est}.

5. Combine the data from the performance counters with the estimated execution time from the linear model to form a feature vector for the ML-based model. Then use the output of the model, the estimated saturated IPS, to compute the estimated total execution time of the kernel for the

Source	**Application (Kernels)**	**Input**
AMD	Binomial (binomial)	32768 samples
NVIDIA	Blackscholes (blackscholes), BoxFilter (BoxRowsLmem), ConvolutionSeparable (convolutionRows, convolutionColumns), CopyComputeOverlap (VectorHypot), DXTCompress (compress), DotProduct (DotProduct), FDTD3d (FiniteDifferences), HiddenMarkov (ViterbiOneStep), Histogram (histrogram64), Matmul (matmul), MatVecMul (uncoal0, uncoal1, coal0, coal1, coal2, coal3), MedialFilter (ckMedian), Nbody (nbody), QuasiRandomGenerator (Quasi, InvCND), RadixSort (reorderDatakey, radixSortBlockKey), SobelFilter (ckSobel), SortingNetworks (sortLocal, sortLocal1, mergeGlobal, mergeLocal), Transpose (trans_naive), Tridiagonal (pcr, cycle)	default
Parboil	CP (cp), Cutcp (lattice), LBM (StreamCollideX), Mri-q (ComputeQ_GPU), Mri-gridding (binning, reorder, sort, rearrange, gridding), Sad (mb_calc, calc_8, calc_16), RPES (computeX), Tpacf (tpacf)	large
Shock	FFT (fft1D_512, fft1D_512), BFS (bfs), Sgemm (sgemmNN, sgemmNT), Spmv (spmv_csr_vector), Stencil2D (stencilKernel)	-s 4
SNU NPB	BT (initialize2), CG (conj_grad_2, conj_grad_6), EP (ep), FT (cffts1, cffts2, cffts3, indexmap, evolve), LU (setiv), MG (resid, norm2u3, rprj3, psinv, interp)	class=B/C

Table 2.2: OpenCL kernels

entire workload.

2.6 Evaluation

2.6.1 Evaluation Setup

The proposed model has been developed for NVIDIA GPUs with caches. However, its simplicity and generality allow it to be applicable to any GPU that shares similar design concepts. To demonstrate that it can be easily adapted to other GPU architectures, we have evaluated the model on different architectures from the two most significant GPU vendors: NVIDIA and AMD. For NVIDIA, we evaluate the model on three NVIDIA GPUs from three architectural generations: the Kepler-based GeForce GTX 680, the Fermi-based GeForce GTX 580, and the GeForce GTX 280 as a representative for NVIDIA's second-generation GPU architecture without hardware caches. Currently, there are no CUDA/OpenCL drivers for the GTX 680 that support reading the performance counters from within OpenCL programs. For this reason we evaluate only the linear model on the GTX 680. For AMD, we evaluate the model on a recent GPU; the Radeon

HD 6970, which comprises hardware caches.

To test the models, we use 70 kernels extracted from 39 OpenCL applications. The OpenCL applications stem from the Parboil [40], SNU NPB [41], SHOC [42], AMD [43] and NVIDIA [39] benchmark suites. The selected kernels all satisfy two conditions: (1) the kernel is executed with a large number of work-groups (at least several hundreds), and (2) the kernel's execution time is longer than one tenth of a second to minimize the effect of small measurement fluctuations. Table 3.2 lists the applications, the kernels, the source of the application and the input data. All experiments are performed with the NVIDIA OpenCL driver 1.1 and AMD APP SDK v2.9.

By executing each kernel at several saturation points, we extract a total of 683 data instances for NVIDIA GPUs and 578 data instances for AMD GPUs. We use the *leave-one-out cross validation* (see Section 2.3.3) technique to evaluate the ML model: for each kernel to be evaluated, all data sets generated by that kernel are removed from the data set before training the ML model.

To estimate the execution time of a kernel, we sample at the second and the third saturation point. The kernel features and the performance counters are used to compute the execution time for the linear model. The result of linear model is combined with the recorded performance counters to estimate the final execution time using the ML-based model.

2.6.2 Performance estimation results

NVIDIA GPUs

Figures 2.11 and 3.9 show the error rates of the the linear and the ML-based model in terms of estimated execution time compared to the actual execution time for NVIDIA GPUs. The error rates of the linear and ML-based model are,

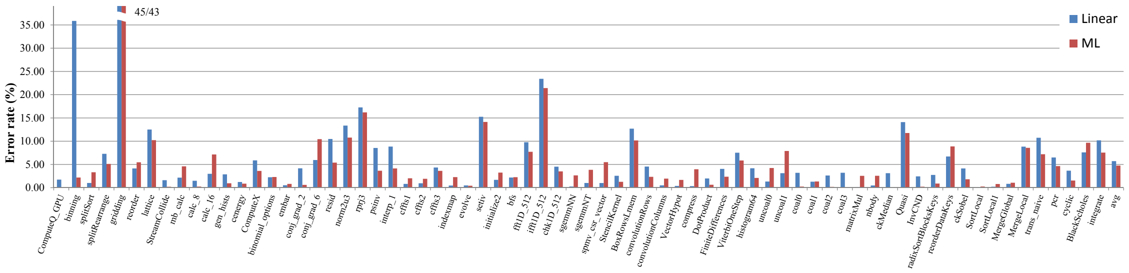


Figure 2.11: Error rates of the linear and ML-based model on the GTX 580 for the execution time of each kernel.

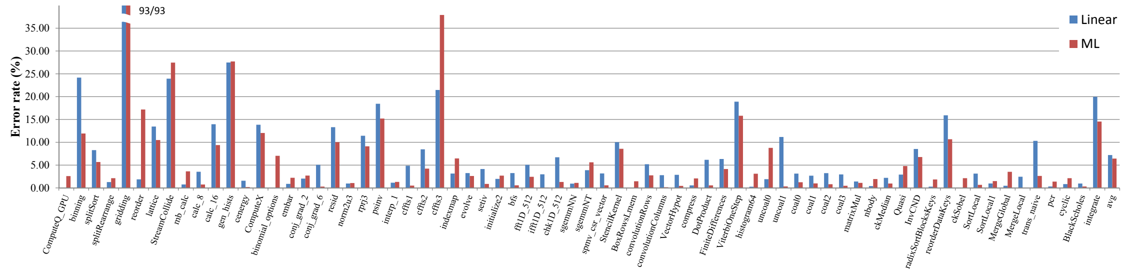


Figure 2.12: Error rates of the linear and ML-based model on the GTX 280 for the execution time of each kernel.

on average, 5.72% and 4.76% for the GTX 580 and 7.19% and 6.42% for the GTX 280, respectively.

As expected, the linear model performs extremely well for the regular kernels, i.e., when all performance factors scale linearly. GPUs perform best when executing regular application, so it is not surprising that the performance counter data of 80% of all kernels we have encountered scale linearly.

For certain kernels, the linear model over- or underestimates the execution time significantly. This comes from several sources. First, even though we have excluded very short running kernels, a few kernels still exhibit a rather short execution time and cause a high fluctuation in the measurements. The

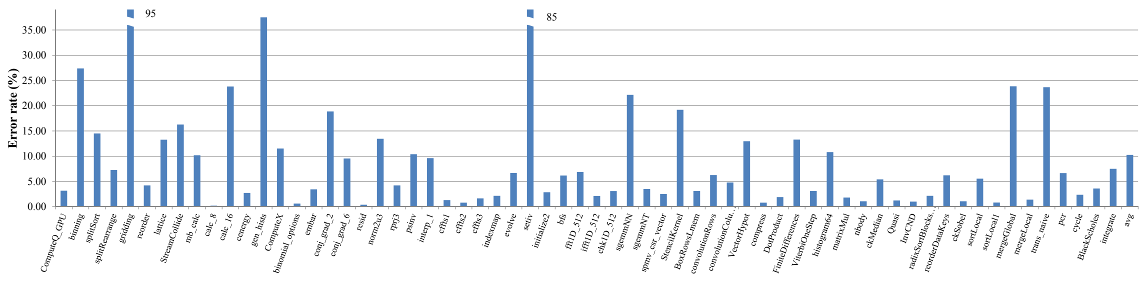


Figure 2.13: Error rates of the linear model on the GTX 680 for the execution time of each kernel.

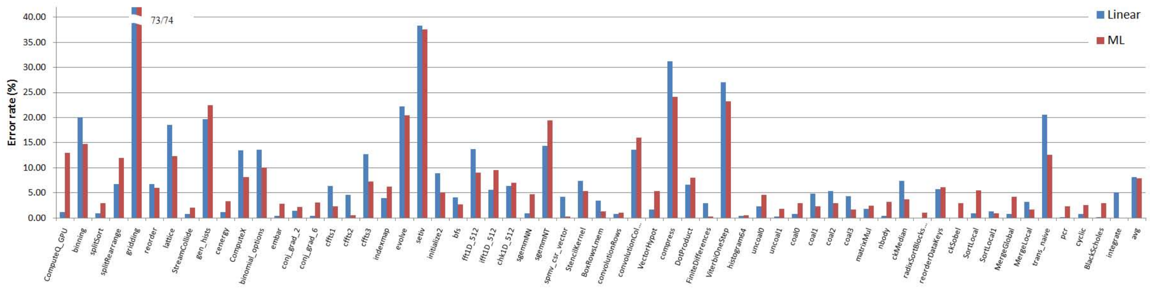


Figure 2.14: Error rates of the linear and ML-based model on the Radeon HD 6970 for the execution time of each kernel.

kernels belonging to this group include `resid`, `rprj3`, `psinv`, `stencilKernel`, `tran_naive`. These kernels show an average error of approximately 10%.

Second, for some kernels we have observed a significant warm-up effect at the beginning of the execution. These effects are caused by the hardware caches, or special instruction scheduling policies that cause the GPU to not evenly distribute the work-groups across the SMs. The learning model observes these non-linear effects through the performance counters and learns to correct the result given by the linear model. On the GTX 580, the significant cases belonging to this type include `ifft1D_512`, `BoxRowsLmem`, `Quasi`, `MergeLocal`, `integrate`, `StreamCollide`, `resid`, `norm2u3`, and `binning`. The ML-based model achieves the biggest improvement for `binning`. This kernel has a large number of warps that are serialized (due to branch divergence) and warp re-issues (due to cache misses) during its execution. These serializations are unevenly distributed across the saturation points. The learning model observes this warm-up effect and is able to produce a much more accurate result. The kernel `Quasi` has high error rate with the linear model because during sampling, the GPU does not seem to evenly distribute the work-groups across the SMs. The GPU performance counters only report data for one SM, hence if the SM being sampled is assigned more work-groups than some other SMs the performance counter numbers are distorted. The ML-based model observes this behavior from the performance counter *sm_cta_launch* and achieves an improved result. The kernels `ifft1D_512` and `BoxRowsLmem` have high error rates for the linear model because the performance factors, such as the number of L1 and L2 cache misses or the number of coalesced memory requests, do not scale linearly. On the GTX 280, the kernels falling into this category include `binning`, `ComputeX`, `cffts3`, `ViterbiOneStep` and `reorderDataKeys`.

The third source for large errors are unevenly distributed workloads across the work-groups. This happens when a kernel contains a conditional branch or a loop whose iteration count depends on the value of some input data, such as the work-group ID or a work-item ID. The kernels that exhibit this behavior are `binning`, `gridding` and `gen_hists`, `StreamCollide`, `uncoal1`, `integrate` on the GTX 280 and `gridding`, `lattice`, `rprj3`, `trans_naive`, `integrate` on the GTX 580. On most of those kernels, the ML-based model performs slightly better than the linear model. This is due to the fact that the learning model attributes this imbalance to warm-up effects and thus computes a more precise throughput.

Comparing the performance of the linear model on GTX 580 and GTX

280, we observe a substantial difference in accuracy between a number of kernels. This includes the kernels `gridding`, `StreamCollide`, `calc_16`, `gen_hists`, `psinv`, `cffts3`, `setiv`, `ifft1D_512`, `BoxRowsLmem`, `ViterbiOneStep`, `uncoal1`, `StencilKernel` and `Quasi`. There are several reasons for these differences: First, certain kernels have an imbalanced number of replays (across the work-groups) on the GTX 580 but are balanced in terms of replays on GTX 280. A replay is the re-execution of an instruction after a cache-miss [44]. The kernels falling into this category are `psinv`, `BoxRowsLmem` and `ifft1D_512`. For these kernels the linear model produces more accurate results on the GTX 280 than the GTX 580. Similarly, the linear model works better on the GTX 580 for kernels that have imbalanced replays on the GTX 280 but are balanced on the GTX 580. Included in this group are `StreamCollide` and `uncoal1`. Second, on the GTX 280 the requirements for memory coalescing are stricter than the GTX 580. For those kernels, `calc_16`, `cffts3` and `ViterbiOneStep`, the warm-up effect is stronger on the GTX 280 and therefore the linear model performs better for the GTX 580 than the GTX 280. Third, due to different architectural parameters, the saturation points are not necessarily equal for the same kernel. `gen_hists` has a large workload imbalance in the last quarter of its work-groups. On the GTX 580, the second sampling point includes work-groups from the last quarter whereas on the GTX 280 the second sampling point includes less work-groups which do not exhibit workload imbalance. For this reason, the linear model has more information on the GTX 580 and is thus more accurate. Similarly, `gridding` is highly imbalanced in terms of the number of instructions per work-group. The occupancy of `gridding` for the GTX 280 and the GTX 580 are different; therefore, the degree of imbalance in the number of instructions at the sampling points is different from the GTX 280 to the GTX 580. Finally, for kernels with a relatively small sampling execution time, the measurement variation is also a source of error, especially for kernels that have large number of work-groups such as `psinv` or `stencialKernel`.

We note that the irregularity in warp serialization on the GTX 280 affects the result of the performance estimation less than the irregularity in memory access patterns on the GTX 580. This explains why the overall error rate of the linear model on the GTX 280 is lower than that on the GTX 580.

We have implemented the model presented by Hong and Kim [7], an analytical model to estimate performance on a GTX 280. Hong's model has an average error rate of 30%, almost five times as much as the ML-based model in Figure 3.9. Note that we use the arithmetic average to compare error rates, while Hong *et al.* report the geometric average in their paper.

To demonstrate the portability of the proposed model, we evaluate the linear model on one of the most recent NVIDIA GPUs, a GTX 680 based on the Kepler architecture, to see how it adapts to a newer GPU. It is currently impossible to read the performance counters from OpenCL, thus we cannot apply the ML-based model to this architecture yet. The linear model obtains a slightly worse overall error rate of 10%. Figure 3.13 details the result on the GTX 680 for each kernel. There are two kernels that suffer from especially high error rates: `gridding` and `setiv`. The kernel `gridding` has an extremely uneven workload distribution across the work-groups as described above. There is no serious workload imbalance in `setiv`. However on the GTX 680, the number of compute units and memory units has been increased, therefore having only 32 work-items per work-group and 8 active work-groups per SM does not use all the GPU resources effectively. To confirm this conjecture we double the number of work-groups per sampling run, i.e., we sample at $2 * P_{sat}$ and $4 * P_{sat}$. This dramatically reduces the error rate for `setiv`.

Note that ML models perform better the more training data is available. We expect the ML-based model to perform better as we add more benchmarks.

AMD GPUs

Figures 2.14 shows the error rates of our model for AMD HD 6970 GPU. The error rates of the linear and ML-based model are, on average, 8.10% and 7.91% respectively. It is interesting that the linear model, although developed based on an analysis of a specific NVIDIA GPU, predicts quite well for most kernels running on AMD HD 6970 GPU. This is because both NVIDIA GPUs and AMD GPUs use the concept of occupancy to indicate how many active groups of work-items can be executed concurrently. This group is called warp in NVIDIA GPUs and wave-front in AMD GPUs. The difference between warp and wave-front is their size, i.e., the number of work-items executed in a *lock-step* manner. This size is simply an input to the linear model. Because the sampling points embody the execution behavior of warps and wave-fronts on the hardware, the linear model can predict very well on both NVIDIA GPUs and AMD GPUs.

The linear model does not predict well on some kernels that include `trans_naive`, `lattice`, `binning`, `gridding`, `setiv`, `gen_hist`, `evolve`, `compress` and `ViterbiOneStep` with an error of more than 20%. There are several reasons for this error. First, as explained in the result section for NVIDIA GPUs, there is a workload imbalance between work-groups for `binning`, `gridding`, `gen_hist` and `trans_naive`. Second, although there is no significant imbalance in the workload, some performance factors do not scale linearly (e.g., the cycles stalled due to a memory access) across different saturation points as in `trans_naive`, `latice`, `setiv`, `compress`, `evolve` and `ViterbiOneStep`.

On the Radeon HD 6970, the ML-based model predicts almost equally as well as the linear model. This is because there is a limited number of performance

Error class	**GTX 280**			**GTX 580**		
	LM	ML	delta	LM	ML	delta
0-5%	1.88	2.36	-0.48	2.03	2.13	-0.10
5-10%	6.65	3.13	3.50	7.59	6.57	1.01
>10%	21.24	18.67	2.57	18.46	13.41	5.06

Table 2.3: Performance for different error classes.

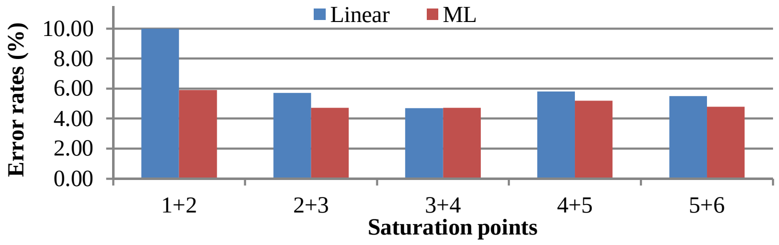
counters we can access using OpenCL. Especially, there is no performance counters related to hardware caches on AMD GPUs that can be accessed through the OpenCL interface. The performance counters used as inputs to the machine learning model on AMD are insufficient to capture the correlation between the execution time and input performance factors, hence its performance is similar that of the linear model.

2.6.3 The ML-based model on different classes of kernels

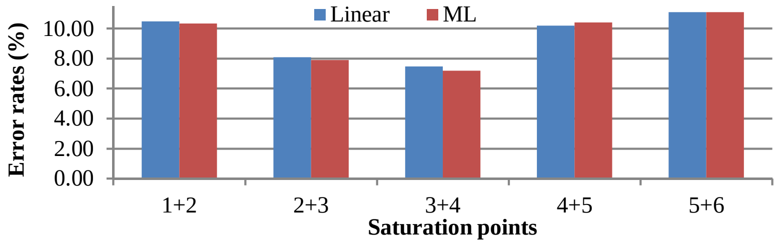
Table 2.3 compares the linear model to the ML-based model in relation to the error class of the kernel for two NVIDIA GPUs. The error class `0-5%` comprises all kernels for which the error of the linear model is 0 to 5%, and the error classes `5-10%` and >`10%` are constructed accordingly. The results confirm that the ML-based model performs better on kernels for which the linear based model has a relatively big estimation error. For regular kernels the linear model slightly outperforms the ML-model. While the ML-based model is capable of adapting to irregularities in the kernels, this flexibility comes at the expense of a small degradation in accuracy for such kernels.

2.6.4 The performance at different saturation points

Figure 2.15 shows the average error rate of the linear and ML-based model at different sampling points for (a) the NVIDIA GTX 580 and (b) the AMD Radeon HD 6970, respectively. The x-axis denotes the first sampling point at x-multiples



(a) NVIDIA GTX 580



(b) AMD Radeon HD 6790

Figure 2.15: Average error rates of the linear and ML-based model at different saturation points.

of the saturation point (P_{sat}); the second sampling point is at $(x+1) * P_{sat}$. For the GTX 580, warm-up effects in the caches lead to a reduced accuracy when sampled at the first saturation point. From the second saturation point on, the warm-up effects can be mostly covered and the accuracy of the model remains similar even with a higher amount of sampling. For the Radeon HD 6970, the linear model achieves error rates of 8-11% for the first five saturation points. This implies that the model benefits from the architectural similarities between GPUs from different vendors. The ML-based model does not perform significantly better; the reason is that AMD GPUs provide much fewer performance counters compared to NVIDIA GPUs. For example, there is not sufficient information about the cache usage for the L1 and the L2 cache. Without these performance counters, the ML model is unable to improve the accuracy of the estimation significantly.

For both architectures, sampling at the second/third sampling point provides reasonable accuracy at the lowest possible overhead. The overhead of sampling compared to the total runtime of the kernels amounts to, on average, 8% for NVIDIA and 15% for AMD GPUs. It is important to note that the sampling overhead is independent of the size of the data input size and will thus be significantly smaller for real-world kernels with very large input data sets.

2.7 Conclusions

We have presented a linear and an ML-based performance estimation model for GPUs with or without hardware caches. The wide applicability of the models is demonstrated by running the benchmarks on 70 OpenCL kernels on three different generations of NVIDIA GPU architectures and one AMD GPU.

The linear model outperforms existing models and achieves error rates below 10%. On modern GPUs, the complex effects of cached memory accesses are difficult to capture by a linear model. On NVIDIA GPUs, with performance counters reflecting cache usage and branch divergences, the ML-based model detects the correlation of these factors and the execution time and reduces the average estimation error to about 5%. The ML-based model performs especially well on benchmarks that exhibit non-linear throughput and are thus difficult to estimate for the linear model. On the other hand, with AMD GPUs, the ML-based model cannot detect this correlation due to the lack of certain important performance counters.

The proposed models significantly outperform related work on various GPU architectures. The models achieve a very good accuracy for a wide range of benchmarks on different GPU architectures and can thus be used as performance estimation models in a dynamic workload-distribution framework.

Chapter 3

Performance Auto-tuning

3.1 Introduction

Although the benefit of selecting a good work-group size for GPUs have been widely studied for performance portability [45, 46, 47, 48, 49], there is still no satisfactory analysis on the reasons that cause the performance variations according to different work-group sizes. Understanding the reasons is very important for automatic performance tuning and program optimization. Even though several approaches have been proposed for auto-tuning GPU programs [50, 45, 46, 47, 51, 52, 53, 54], the relationship between the work-group size and the overall performance is not analyzed in detail.

To automatically select the work-group size for GPUs, most of current approaches are for a specific class of algorithms (*e.g.*, matrix-multiplication) [50, 45, 46, 52, 54]. Some others require profiling on small input sizes of the programs [51, 53, 55, 56]. These approaches are not applicable to the programs that have few input sizes. Other approaches consider many program optimization techniques at the same time [47, 51, 57], however for tuning the work-group size, they have to search through all possible work-group sizes to find the best work-group size. Evolution search strategies [58] are effective for large-space tuning problem. However, since the search mechanism generally lacks the domain-specific knowledge (e.g., regarding the GPU architecture), it is difficult to find the optimal work-group size.

The state-of-the-art technique [48] has demonstrated an excellent work on finding a number of program parameters, including the work-group size. Their search-space pruning mechanism relies on the number of concurrent threads and the instruction count. It assumes that the programs do not use memory operations intensively, which had been reasonable until Fermi architecture [59] was introduced. However, both AMD and NVIDIA GPU memory systems have

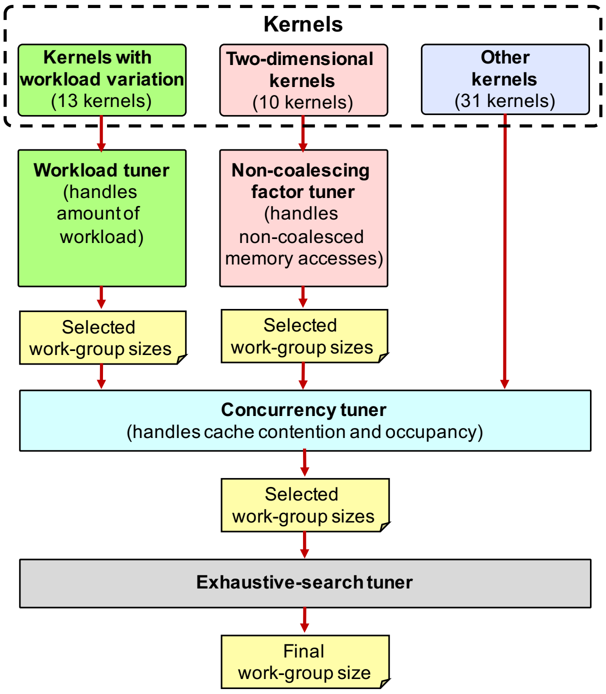

Kernels
Kernels with workload variation (13 kernels)
Two-dimensional kernels (10 kernels)
Other kernels (31 kernels)
Workload tuner (handles amount of workload)
Non-coalescing factor tuner (handles non-coalesced memory accesses)
Selected work-group sizes
Selected work-group sizes
Concurrency tuner (handles cache contention and occupancy)
Selected work-group sizes
Exhaustive-search tuner
Final work-group size

Figure 3.1: Proposed auto-tuning framework.

become more complex, e.g., the presence of caches. GPU programmers have extensively exploited this feature and many programs use heavier memory operations as a result. The heavy use of memory operations in a cached memory architecture leads to a diversity in memory access patterns, which certainly complicates the choice of work-group size. This a clear demand for the GPU auto-tuners to incorporate the ability to handle memory performance factors.

In this paper, we propose an auto-tuning framework that overcomes the drawbacks of previous approaches. Our approach tackles the auto-tuning problem by considering all important performance factors that can affect the choice of a good work-group size. Our approach makes no special assumptions about the program, thus it applies to any program.

We first characterize a large body of OpenCL kernels to identify the performance factors that affect the choice of a good work-group size for GPUs. Based on the characterization, we realize that the most influential performance factors include the number of concurrently executed threads, coalesced global memory accesses, cache contention, and the amount of workload in a kernel.

Based on the observations, we propose a set of auto-tuning techniques that sub-optimally selects the work-group size and shape. By the shape, we imply the two-dimensional shape of the work-group for two-dimensional kernels. Our technique relies on a set of four tuners: workload, non-coalescing factor, concurrency, and exhaustive-search tuners. The input to a tuner is a set of work-group sizes to be searched. Those tuners utilize the profiling information collected by executing the kernel with several representative work-group sizes to prune the search space. Figure 3.1 shows the overall structure of the proposed auto-tuning framework.

The *workload tuner* handles the variation of workload and transforms the kernel index space to another so that the amount of workload does not vary as the work-group size varies. The *non-coalescing factor tuner* handles non-coalesced global memory accesses. It prunes the search space by quickly selecting the work-group sizes that produce the smallest number of memory transactions. The *concurrency tuner* considers the trade-off between the number of concurrently executed threads and potential cache contention to prune the search space. Finally, the *exhaustive-search tuner* executes the kernel with all the work-group sizes in its input and selects the work-group size that produces the smallest execution time.

In summary, contributions of the paper are as follows:

- We characterize OpenCL kernels to provide a clear picture that shows the factors affecting the performance for different work-group sizes.

- We propose a kernel index transformation technique that improves the performance and helps the auto-tuner choose better work-group size.

- We propose a simple metric, called *memory intensiveness*, to identify the kernels that potentially have cache contention.

- We propose a simple method to calculate the number of memory transactions for a work-group and use it to help pruning the work-group sizes. To the best of our knowledge, this proposal is the first auto-tuner that minimizes the number of memory transactions at the work-group level.

- We show the effectiveness of our approach by implementing the auto-tuning techniques and evaluate it with 54 kernels from various sources on three different NVIDIA GPUs and one AMD GPU. Our result shows an improvement in both tuning quality and tuning cost when compared with previous approaches [48, 58].

The remainder of our paper is structured as follows. The next section describes related work. Section 3.3 introduces OpenCL. Section 3.4 characterizes the OpenCL kernels with regard to the work-group size. In Section 3.5, we present auto-tuning techniques for the work-group size. We evaluate our auto-tuner in Section 3.6 and conclude in Section 3.7.

3.2 Related Work

Performance tuning for GPUs is an important research topic. Many studies have been done to tune a set of parameters for a specific GPU program [50, 45, 46, 52, 54]. However, our work focuses on a generic auto-tuner of OpenCL programs for GPUs.

Ryoo *et al.* [48] propose an optimization space pruning method for CUDA programs using static program metrics. Ansel *et al.* [58] introduce a framework called OpenTuner to automatically tune program parameters. OpenTuner framework uses evolution algorithms to search for large space. These works are closest to ours because they can find sub-optimal work-group size in a reasonable time. By quantitatively comparing our result with their result, we show that our approach finds better work-group sizes with similar or smaller cost.

Seo *et al.* [60] propose an automatic OpenCL work-group size selection technique that is tailored to CPU processors and uses search space pruning techniques that are different from ours.

Several compiler frameworks that offer the ability to tune the performance for GPUs have been proposed [47, 51, 57]. These frameworks consider various program optimization techniques. However, to select a work-group size, only an exhaustive search is employed or it is not considered. Some approaches consider auto-tuning the work-group size for more than one input size of the program [51, 53, 55, 56]. However, these approaches require executing the program several times for small input sizes. Our approach however focuses on predicting the work-group size for a single input size.

To characterize the GPU performance, Yang *et al.* [49] also identify the sophisticated relationship between the memory usage and the thread-block size for CUDA programs. Their focus is evaluating compiler transformations rather than pruning the optimization space. Thus, all the possible thread-block sizes need to be fully executed. Rogers *et al.* [61] characterize the effect of the warp-size on NVIDIA GPUs, however the work-group size is not considered. Zhang and Owens [62] characterize some performance factors for NVIDIA GPUs including work-group sizes. Bakhoda *et al.* [63] analyze a number of performance factors for CUDA programs using a simulator. Their work focuses on architectural design factors, not auto-tuning. The effect of work-group size on the performance is not explored either.

Kayiran *et al.* [64] proposes a GPU scheduling mechanism that dynamically determines the number of concurrently executed work-groups to improve the performance. Their scheduling mechanism is also based on the relationship between the resource contention and the optimal number of concurrently executed work-groups, which is similar to a result of our analysis. However, our work focuses on choosing a good work-group size and we also consider other performance factors, such as the number of memory transactions and the amount of workload.

3.3 OpenCL and GPU Architectures

An OpenCL application consists of a host program and one or more *kernels*. A kernel is executed on a GPU in an N-dimensional abstract index space ($N \in \{1, 2, 3\}$) specified by the host program. A point in the index space corresponds to a *work-item* that is an execution instance of the kernel. The work-items are organized into groups of equal size, called *work-groups*. The work-groups are executed independently and the execution order of the work-groups is determined dynamically [65].

A GPU consists of many Streaming Multiprocessors (SMs). Each SM contains a number of processing elements, memory execution units and one or more warp schedulers. When a kernel is executed on the GPU, work-groups in the kernel index space are distributed to SMs. Once a work-group is assigned to an SM, it will execute until all its work-items complete. The GPU hardware groups work-items in a work-group into a smaller chunk, called *warp*, and execute them together in a SIMD manner. All the work-items in a warp always execute the

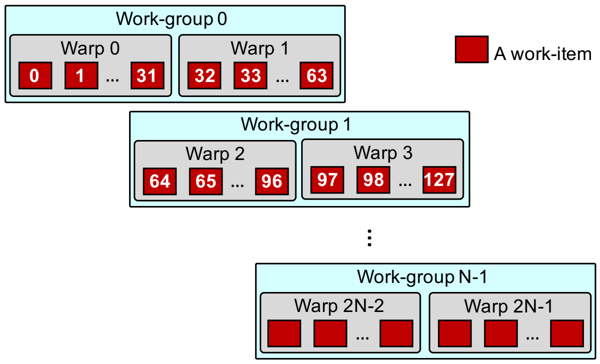


Figure 3.2: Work-items, work-groups, and warps.

same instruction [35, 65]. Figure 3.2 shows a sample one-dimensional index space to illustrate the relationship between work-items, work-groups and warps. Note that the warp IDs are numbered globally instead of being restarted from zero for each work-group as in CUDA warp ID numbering. In practice, the work-group size is typically determined by the programmer.

To keep hardware resources in the GPU busy, an SM can execute more than one warp *concurrently*. The warps being executed on the SM are called *resident warps* or *active warps*. Similarly, a work-group whose warps are active is called an *active work-group*. The GPU resources (*e.g.*, registers) are occupied by the active work-groups. The GPU evenly distributes these resources to the active warps and how much resources each warp receives is determined by the OpenCL kernel compiler. Since each warp holds their own resources (*i.e.*, context), they can progress at their own pace. Context-switching between warps in an SM happens at the hardware level.

Since each warp has its own hardware context in the SM, the number of warps that can be active at the same time depends on how much resources they require. The ratio of the number of active warps to the maximum number of active warps allowed is called *occupancy* [65]. Typically, the higher the occupancy, the better

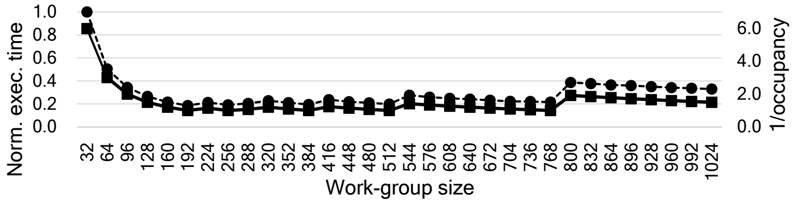
Device name	**GTX 580**	**GTX Titan**	**GTX Titan X**	**Firepro W8000**
Vendor	NVIDIA	NVIDIA	NVIDIA	AMD
Architecture	Fermi	Kepler	Maxwell	Tahiti PRO
Core clock speed (MHz)	1590	902	1080	900
Number of SM	16	14	24	28
Size of L1 (KB)	16	16	24	16
Size of Local Memory (KB)	48	48	96	32
Size of L2 (KB)	768	1536	3072	512
Maximum active warps per SM	48	64	64	40
Maximum active work-groups per SM	8	16	32	40
Number of 32-bit registers per SM	32768	65536	65536	65536
Warp/wavefront size	32	32	32	64

Table 3.1: Hardware specifications of the GPUs used.

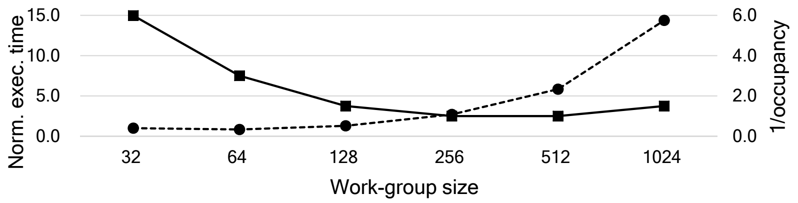
resource utilization. The maximum number of active warps depends on the GPU architecture. GPUs also have a limit in the number of active work-groups. Table 3.1 specifies the hardware specifications of the GPUs used in this paper.

3.4 Effects of the Work-group Size

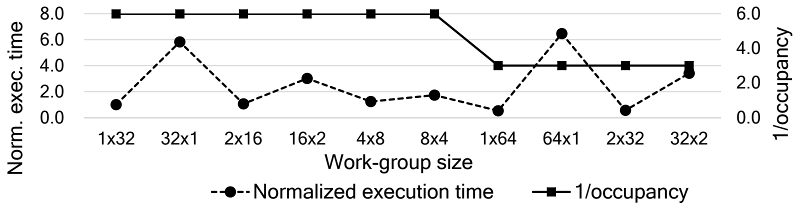
In this section, we pinpoint the performance factors significantly affected by the work-group size. Specifically, they are occupancy, memory coalescing, cache contention, the amount of workload, work-group scheduling and barriers. The experiments in this section are conducted using a GTX 580 GPU. The performance behaviors of the other GPUs listed in Table 3.1 are similar to that of GTX 580. When there is a significant difference between them, we will specify it.



(a)



(b)



(c)

Figure 3.3: Relationship between the execution time and occupancy of some kernels from Rodinia and NVIDIA SDK. (a) `BFS_1` from Rodinia. (b) `MatVecMulCoalesce2` from NVIDIA SDK. (c) `Fan_2` from Rodinia.

3.4.1 Occupancy

The most obvious metric that contributes to the performance of a kernel is the degree of concurrency between warps. This is represented by the occupancy. The work-group size (say G) is one of the three static parameters (together with the number of registers used per work-item denoted by R and the amount of local memory per work-group denoted by M) that the compiler uses to decide the occupancy. When R is fixed and $M = 0$, the occupancy is hard-constrained by G [65, 66].

Figure 3.3 shows the relationship between the execution time and the occupancy of a kernel. Three different kernels `BFS_1`, `Fan_2`, and `MatVecMulCoalesced2` are used. We vary the work-group size. We consider only work-group sizes that are a multiple of the warp size to fully utilize the GPU resources. We plot the execution time and the inverse of occupancy of each kernel to help the reader visualize better.

When we vary the work-group size, the execution time varies. Figure 3.3(a) shows that the changes in the execution time are identical to the changes in the occupancy. This is the typical case where the occupancy is the main contributing factor to overall performance.

Figure 3.3 (b) and (c), the occupancy is not a major contributing factor to performance. In Figure 3.3 (b), a higher occupancy value does not guarantee higher performance. Figure 3.3(c) shows the result for two-dimensional kernel `Fan_2`. Even for the same number of work-items per work-group, the sizes of each dimension affect the performance significantly.

The examples shown in Figure 3.3 have two implications. One is that it is beneficial to choose a good work-group size. For example, the best work-group size for `BFS_1` is 5.22x faster than the worst. The other is that choosing a proper work-group size for each kernel is very difficult because of the complicated interactions between multiple performance factors.

3.4.2 Global Memory Coalescing

Using the global memory in a GPU can affect the choice of a good work-group size because of two things: the ability of the GPU to hide memory latency and the number of memory transactions per warp.

The ability of hiding memory latency. As we mentioned, the work-group size that produces a higher occupancy allows more active warps. A higher number of active warps lets the long latency of memory instructions be more effectively hidden by the latency of compute instructions [65].

When a warp executes a global memory instruction, the GPU coalesces the global memory accesses from the work-items within a warp into as few memory transactions as possible. If accesses generated by a warp cannot be coalesced into a single memory transaction, they are called the *non-coalesced memory accesses* [67, 68].

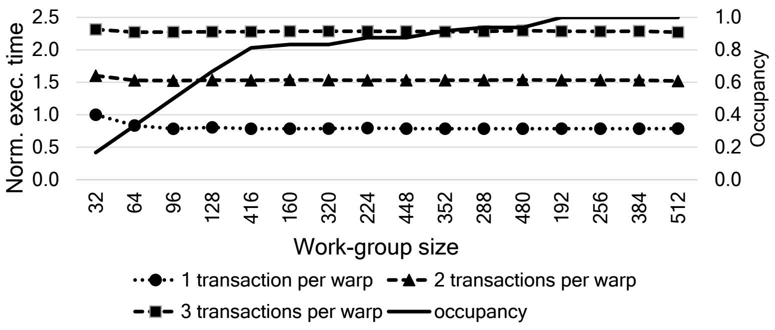


Figure 3.4: The effect of memory coalescing in terms of occupancy.

We clarify how the number of non-coalesced memory accesses per warp may affect the relationship between the work-group size and the ability of hiding memory latency. Specifically, with a large number of non-coalesced memory accesses per warp, the GPU spends most time on memory operations. The memory accesses becomes the performance bottleneck. In this case, a work-group size that produces a high occupancy will be less useful in hiding the latency of the memory instructions.

Effect of non-coalesced memory accesses. To show how non-coalesced memory accesses neutralize the benefit of high occupancy, we construct a micro-benchmark that accesses the global memory with a strided pattern. The stride determines the number of memory transactions generated by a warp. Note that if the stride is bigger than one, the memory accesses are non-coalesced memory accesses. We call the number of memory transactions per warp as the *non-coalescing factor*. Also note that the non-coalescing factor does not vary when the work-group size varies. We vary the non-coalescing factor and measure the performance. The micro-benchmark is constructed in the way that the occupancy does not change as we vary the non-coalescing factor (*i.e.*, the stride). We do not use local memory, and the number of registers used per work-item is small and fixed. Thus, these two factors are not the limiting factor of the occupancy, resulting in a fixed mapping between the work-group size and the occupancy. To obtain a specific occupancy, we look-up this mapping and choose the corresponding work-group size. Figure 3.4 shows the performance of the micro-benchmark. To help reader visualize better, we show the performance in an order of increasing occupancy.

We obtain two observations: First, when the memory requests are fully coalesced (*i.e.*, only one memory transaction per warp), the overall performance varies with the first few occupancies (from 0.167 to 0.667). However, increasing the number of transactions per warp (2 and 3) neutralizes the performance variation. This observation confirms our conjecture that non-coalesced memory accesses neutralize the benefit of high occupancy (*i.e.*, the ability to hide the memory latency). Second, if we consider the performance at the occupancy where the performance saturates (*e.g.*, 0.833), the difference in execution time between consecutive strides is constant. The difference is equal to the execution time of the coalesced case (one transaction per warp). This observation implies that the execution time of the coalesced case is the time spent on memory operations. Moreover, the time spent on compute and control instructions is totally hidden by the time to execute memory instructions.

Memory intensiveness. Based on the observations, we define the *memory intensiveness* of a kernel as the ratio of accumulated cycles spent on processing memory instructions to the total execution time. The memory intensiveness of a kernel represents the degree of how frequently memory instructions are executed in the kernel. A kernel with its memory intensiveness higher than a pre-determined threshold is called a *memory-intensive kernel.* Memory intensiveness is different from the memory access intensity defined by Muralidhara *et al.* [69]. The memory access intensity is defined as the miss rate at the last-level cache. Our memory intensiveness reflects not only the cache miss rate but also the rate of accessing the global memory.

We denote by $AMAT$ the average memory access time per memory transaction, $\#mem$ the number of memory transactions, and T_{mem} the total amount of time taken to process memory instructions in the kernel. We also denote by P the ratio $(\#mem \cdot AMAT)/T_{mem}$. When the memory bus in the GPU is saturated, T_{mem} can be calculated by,

$$T_{mem} = (\#mem \cdot AMAT)/P \tag{3.1}$$

Since memory transactions can overlap with each other [62], P represents the degree of transaction overlapping. If P is greater than one, the GPU allows more than one memory transactions at a time. $\#mem \cdot AMAT$ is the time spent on memory instructions if global memory transactions cannot overlap with each other and have to be performed sequentially.

Since the value of P is hardware dependent, we empirically find P for different GPU architectures. To find P, we measure the execution times for different values of the non-coalescing factor by changing the stride ($STRIDE$) using the same micro-benchmark used in Figure 3.4. When $STRIDE = 1$, let T denote the execution time of the kernel. Then, as we observed before in Figure 3.4, the difference between the execution times of consecutive values of $STRIDE$ is constant and equal to T. By calculating the ratio $(\#mem \cdot AMAT)/T$, we obtain the value P. $AMAT$ and $\#mem$ are derived from the values obtained by the GPU performance counters. If the memory bus is not saturated (*i.e.*, under-utilized), T_{mem} gives an upper-bound of the time spent on memory instructions.

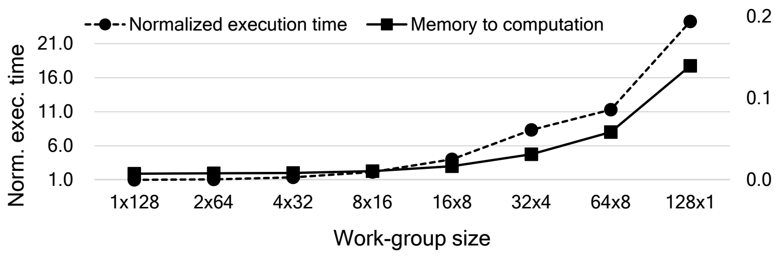


Figure 3.5: The effect of non-coalesced memory accesses for kernel Fan_2 in Rodinia.

```
#define NUM 32
__kernel
void test(__global float * in, __global float * out)
{
  int gID = get_global_id(0);
  float temp = 0.0f;
  for (int i = 0; i < NUM; i++)
    temp += in[STRIDE * gID + i];
  out[gID] = temp;
}
```

Figure 3.6: Micro-benchmark code to show the relationship between the work-group size and cache contention.

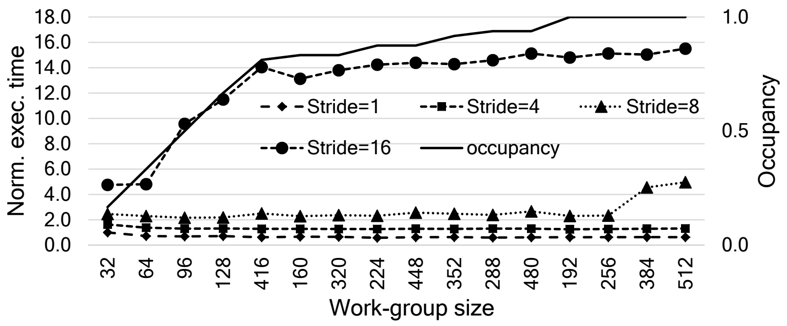


Figure 3.7: The effect of cache contention.

The memory intensiveness, which we denote by λ, can now be calculated as the ratio of T_{mem} to the total execution time of the kernel. If λ is close to one, we say the kernel is memory intensive. We will use λ later to identify the cases where the global memory is the major performance bottleneck.

The number of memory transactions per warp. Another reason why using the global memory affects the choice of a good work-group size is that different work-group sizes may produce different non-coalescing factors. We do not encounter this behavior for one-dimensional kernels. Instead, this behavior is much more pronounced in two-dimensional kernels. Figure 3.5 shows the effect of different shapes for 128 work-items per work-group. As we see, the performance is almost proportional to the ratio of the number of memory transactions to the number of non-memory instructions. Since the number of non-memory instructions is constant, the quantity that varies is the number of memory transactions. The difference in the number of memory transactions stems from different non-coalescing factors.

3.4.3 Cache Contention

Having multiple warps executed on the same SM may raise the issue of cache contention or thrashing [70] on both L1 and L2 caches. The number of multiple warps to be concurrently executed on the same SM is determined by the occupancy. It is partly determined by the work-group size. Thus, a different work-group size may produce a different degree of cache contention. Note that we do not use local memory, and the number of registers used per work-item is very small in this experiment. Consequently, the only contributing factor to the occupancy is the work-group size.

To trigger the cache space contention, we measure the performance of a micro-benchmark described in Figure 3.6 with different values of $STRIDE$.

$STRIDE$ is the number of memory transactions generated by a warp, *i.e.*, the non-coalescing factor.

The *for* loop loads 32 elements from the global memory. Since the size of `float` is four bytes, all the 32 elements fall into the same 128-byte L1 cache line. Thus, if only one warp executes, the first access introduces L1 cache miss and remaining 31 subsequent accesses hit the L1 cache. The key idea is that when the contention is triggered, the subsequent accesses of the iteration are cache misses because other warps have evicted the cache line brought by the first access of the iteration, resulting in significant performance drop. The contention is triggered by high values of $STRIDE$.

Figure 3.7 shows the execution time when $STRIDE = 1, 4, 8, 16$, sorted by an ascending order of occupancy values. With $STRIDE = 1, 4$, the cache miss rates are small so the execution time is flat. $STRIDE = 8, 16$ triggers the contention. The L1 cache miss rate increases significantly, and the execution time increases when the occupancy increases. A higher occupancy value causes L1 cache contention and thus, hurts performance.

On GTX Titan and GTX Titan X, global memory accesses are not cached in the L1 cache. Instead, the issue that we have just addressed will typically appear on the L2 cache. However, since their L2 caches is much bigger than the L1 cache of GTX 580, the condition for the cache contention will be relaxed.

3.4.4 Amount of Work

The amount of work in a kernel can be different for different work-group sizes. There are two main reasons. First, if the kernel employs a reduction computation pattern that exploits the local memory, the work-group size is typically proportional to the number of reduction steps. A higher number of steps introduces more instructions. This might have a negative effect on the overall performance. Several studies [71, 72, 48] have indicated that reduction computations on GPUs introduce a large number of extra instructions.

Second, some kernels are executed with a fixed number of work-groups. For these kernels, if we modify the work-group size, the total number of work-items will change. However, these kernels often have a loop in each work-item and the number of iterations of the loop is inversely proportional to the work-group size. Thus in total, the effective amount of the workload should be constant as we modify the work-group size. However, if a kernel is written this way, there should be a portion of the code that processes private data (*e.g.*, reading global memory to registers and performing some computation). This portion of the code will be multiplied to the total number of work-items and if it takes up a large portion, it might hurt the performance for large work-group sizes. Kernels `cffts1`, `cffts2`, and `cffts3` in application `FT` from SNU NPB expose this behavior.

3.4.5 Work-group Scheduling and Barriers

The execution time of warps in a work-group may be different because of several reasons: unbalanced workload across warps, different cache miss rates, etc. On the other hand, the GPU schedules a new work-group if all warps inside a currently active work-group complete. Thus, a small work-group size will be more beneficial with regard to this aspect because a new work-group can be scheduled earlier. For a large work-group size, a new work-group has to wait longer.

However, small work-group sizes may have bad effect to the performance if the number of work-groups is big enough. We launch an empty kernel with a large number of work-groups and observe that smaller work-group sizes have higher execution time. This is due to the work-group scheduling overhead, and this overhead may vary for different GPUs.

Barriers. In OpenCL, barriers are a synchronization method between work-items in the same work-group. If the execution of a work-item reaches a barrier, it must wait until all other work-items in the same work-group reach this barrier. Hence, if the warps in a work-group run at different paces, the warps that reach the barrier earlier will be stalled. Thus, for large work-group sizes, the number of warps that have to stall is larger than that of small work-group sizes.

We design a micro-benchmark that satisfies the following condition: when varying the work-group size, the only affected factors are the execution time and the occupancy. Then, we place barriers randomly in the kernel and we compare the performance of having and not having the barriers. For the versions without barriers, when comparing the performance of those work-group sizes having the same occupancy, smaller work-group sizes give slightly better performance. This is attributed to the effect of work-group scheduling described above. Under the presence of barriers, smaller work-group sizes give much better performance. This confirms our conjecture that small work-group sizes are advantageous when the kernel has barriers.

3.4.6 Benchmark Applications

We collect a total number of 54 kernels from five different benchmark-suites: SNU NPB [73], Parboil [74], SHOC [75], Rodinia [76], and NVIDIA SDK [39]. We list the kernel names and their sources that we analyze in Table 3.2. Using these kernels, we categorize the effect of the work-group size according to the important performance factors described earlier. Table 3.2 also shows whether an OpenCL kernel is affected by one of the performance factors mentioned before.

Based on the observations, the occupancy affects most kernels (51 out of 54 kernels). This is natural because a GPU has a large number of processing

Source	**Application**	**Kernel**	**D**[a]	**G**[b]	**O**[c]	**N**[d]	**C**[e]	**W**[f]	**B**[g]
Rodinia	BFS	BFS_1	1	32	×				
		BFS_2	1	32	×				
	Cfd	Initialize	1	32					
		Compute_flux	1	17	×		×		
		Time_step	1	17	×	×			
		Comptute_step_factor	1	32	×				
	Gaussian	Fan2	2	464	×	×			
	Kmeans	Kmeans_kernel_c	1	32	×				
		Kmeans_swap	1	32	×	×	×		
	B+tree	FindK	1	32	×				×
		FindRangeK	1	32	×				×
	LavaMD	Kernel_gpu_opencl	1	32	×	×	×		
	Nn	NearestNeighbor	1	32	×				
	Pathfinder	Dynproc_kernel	1	31	×				×
	Particle_filter	Find_index	1	32	×				
		Normalize_weight	1	32					
	Srad	Prepare_kernel	1	6	×				
		Extract_kernel	1	6	×				
		Reduce_kernel	1	6	×			×	×
		Srad_kernel	1	6	×				
		Srad2_kernel	1	6	×				
		Compress_kernel	1	6	×				
	Streamcluster	Pgain_kernel	1	32	×		×		×
Parboil	Mri-q	ComputeQ_GPU	1	32	×				
	Spmv	Spmv_jds	1	32	×	×			
		Spmv_csr_vector_kernel	1	32	×	×	×		
	Sgemm	MysgemmNT	2	19	×	×	×		
	Stencil	Block2D_hybrid_coarsen_x	2	464	×	×			×
SHOC	BFS	Bfs_kernel_warp	1	32	×				
	MD	Compute_lj_force	1	32	×	×			
SNU NPB	CG	Conj_grad_1	1	32	×	×	×	×	×
		Conj_grad_2	1	32	×	×	×	×	×
		Conj_grad_3	1	32	×	×	×	×	×
		Conj_grad_4	1	32	×	×	×	×	×
		Conj_grad_5	1	32	×				
		Conj_grad_6	1	32	×	×	×	×	×
		Conj_grad_7	1	32	×	×	×	×	×
		Makea_3	1	32	×				
		Makea_6	1	32	×	×	×		
	EP	Embar	1	4	×				
	FT	Cffts1	1	32	×			×	×
		Cffts2	1	32	×			×	×
		Cffts3	1	32	×			×	×
		Evolve	2	464	×	×	×		
	SP	Ninvr	2	200	×				
		Pinvr	2	200	×				
		Txinvr	2	200	×				
		Tzetar	2	200	×				
	IS	Full_verify1	1	32					
		Full_verify2	1	32	×	×		×	×
NVIDIA	MatVecMul	MatVecMulCoalesced2	1	6	×			×	×
		MatVecMulCoalesced3	1	6	×			×	×
	FDTD3d	FiniteDifference	2	464	×				
	MedianFilter	CkMedian	2	340	×				

[a] Dimension.

[b] Work-group sizes (the number of possible work-group sizes).

[c] The kernel is affected by the occupancy.

[d] The kernel has memory access patterns that are not coalesced, and the non-coalescing factor depends on the work-group size.

[e] Cache contention occurs at either L1 or L2 cache.

[f] The amount of workload exposes a significant variation between work-group sizes.

[g] The kernel uses a barrier.

Table 3.2: Characteristics of the kernels used in this paper.

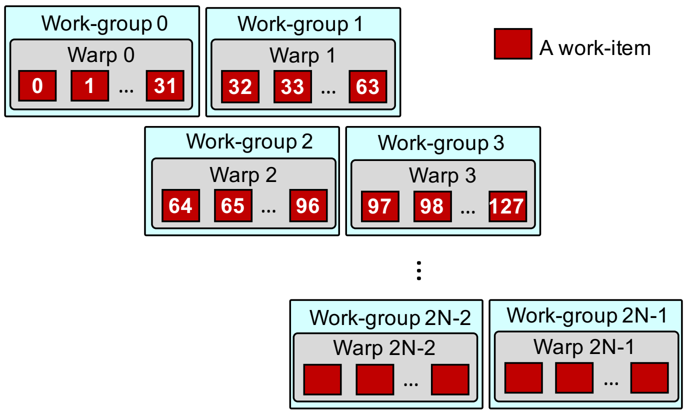
elements. Having as many active warps as possible at the same time increases the utilization of GPU resources and hides long latency instructions. However, as we discussed in Section 3.4.3 and Section 3.4.4, there are two factors that significantly prevent the benefit of high occupancy: *the amount of workload* and *the cache contention*. Furthermore, even for the same number of work-items per work-group, different global memory access patterns and work-group shapes may cause different *non-coalescing factors*, which significantly affect the performance.

3.5 Auto-tuning Work-group Size

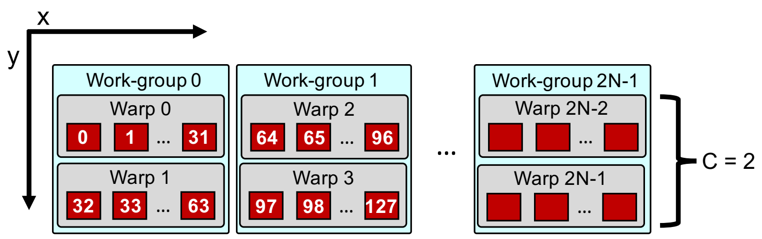
In this section, we use the insights learned from the previous section to automatically find a good work-group size for a given OpenCL kernel. We limit the tuning space to be all the work-group sizes that are legal to compile the kernel and are a multiple of the warp size, $WARP\_SIZE$.

Our auto-tuner is part of an OpenCL runtime. When a kernel is launched, the runtime will profile the kernel with different work-group sizes using the OpenCL profiler [68]. The auto-tuner uses the profiled information to choose the work-group size and report it to the user.

Our auto-tuner takes into account of the following performance factors: the amount of workload, non-coalesced memory accesses, cache contention and occupancy. Figure 3.1 illustrates the overall structure of our auto-tuning framework. The kernels are classified into three categories. The first category includes the kernels that have the workload increased as the work-group size increases. There are 13 such kernels. A kernel of this type can be easily detected by profiling a few work-groups at different work-group sizes and compare the instruction count. The second category includes 10 two-dimensional kernels. The remaining category includes all other 31 kernels. The kernels of each type will be assigned to the corresponding tuner.



(a) After applying Step 1 to the index space in Figure 3.2.



(b) After applying Step 2 to the index space in (a).

Figure 3.8: Transforming the one-dimensional index space in Figure 3.2 by the workload tuner.

3.5.1 Workload Tuner

The input to this tuner is the kernels that expose high variation in their workload for different work-group sizes. It applies an index space transformation to the kernels to eliminate the variation. The transformation is performed in two steps:

Step 1. Make the work-group size to be $WARP\_SIZE$.

Step 2. Group C adjacent work-groups into a bigger work-group.

As mentioned in Section 3.4.4, a smaller work-group size produces a smaller number of reduction steps or a smaller number of work-items, the amount of workload in each work-item is smaller for a smaller work-group size for this type of kernels. Thus, choosing the $WARP\_SIZE$ as the work-group size will minimize the amount of workload. The one-dimensional kernel index space in Figure 3.2 is transformed to Figure 3.8 (a) by Step 1 of the workload tuner. $WARP\_SIZE = 32$ in this example.

However, this small work-group size ($WARP\_SIZE$) often produces poor performance. This is because GPUs have a limit in the maximum number of active work-groups. For example, the GTX 580 GPU only allows 8 active work-groups. Under an assumption that we have an enough number of work-groups, a work-group size of $WARP\_SIZE$ results in 8 active warps (each work-group contains only one warp). This is very small compared to 48, the maximum number of active warps in GTX 580. Thus, it is not enough to maximize the warp-level concurrency in the GPU.

In Step 2, we group the small work-groups into a bigger work-group. This gives us an opportunity to obtain a different, possibly higher occupancy. The question is how many work-groups to be grouped together to obtain good performance. We vary C to tune the new work-group size. This value is exposed as the tuning parameter of the transformed kernel. All possible values of C will be fed into the concurrency tuner later. Note that the transformed kernel is actually a two-dimensional kernel with the size of one dimension fixed to $WARP\_SIZE$, and the size of the other dimension is C. Figure 3.8 (b) illustrates Step 2 of the transformation with $C = 2$.

Different values of C will only produce the same amount of workload for the whole kernel. There are two reasons for this. First, C is independent with the number of reduction steps. The number of the reduction steps is $log_2(work\_group\_size)$. Hence it has been minimized due to Step 1, specifically,

$log_2(WARP\_SIZE)$. Second, for the kernels that have the total number of work-items increased as the work-group size increases, this technique minimizes the total number of work-items and keeps it constant with regard to the change of C. The total number of work-items is minimized because Step 1 chooses the smallest work-group size. It is kept constant for different values of C because when we group C work-groups into a bigger work-group, the total number of new work-groups will be reduced by C times.

In summary, the output of the workload tuner is a transformed kernel index space with a list of work-group sizes ($C \times WARP\_SIZE$) and all possible values of C.

3.5.2 Non-coalescing Factor Tuner

This tuner handles two-dimensional kernels and filters work-group sizes with regard to the non-coalescing factor (*i.e.*, the number of memory transactions). It selects work-group sizes that potentially have a low non-coalescing factor. It solves the following problem: for a fixed number of work-items organized into a work-group of m rows and n columns ($m \cdot n = W$ is a constant), find n (or m) so that the non-coalescing factor is minimized. Since the way a warp is formed is in row major order of work-items in the work-group, we assume that $WARP\_SIZE$ is a multiple of n. In addition, W is a multiple of $WARP\_SIZE$.

A GPU memory segment is a set of 128 consecutive bytes whose starting address is divisible by 128 [77, 65]. The memory requests of each warp are coalesced into as few segments as possible. One memory transaction is issued for the requests in a segment. While the rules that the GPUs use to coalesce global memory requests are provided by the GPU vendors [77, 65], no calculation for work-groups is provided.

This tuner checks each global memory reference in the kernel if its accesses are consecutive. If the accesses are consecutive, the location (*e.g.*, array index) accessed will be represented by the form of $A \cdot ID_x^{global} + B \cdot ID_y^{global} + C$ with $A = 1$ or $B = 1$. C is some constant and $(ID_x^{global}, ID_y^{global})$ is the global ID of a work-item.

To simplify the discussion, we assume that the work-item (0, 0) inside a work-group accesses the starting position of a memory segment. When $A = 1$ and B is some constant, each row accesses n consecutive locations, resulting in a total of $\lceil n \cdot D/128 \rceil$ transactions (note that 128 is the size of a memory segment and we denote D the size of each element). Thus, the entire work-group generates $m\lceil n \cdot D/128 \rceil$ transactions. This can be rewritten as $m\lceil n \cdot D/128 \rceil = \frac{W}{n}\lceil n \cdot D/128 \rceil$. We consider n in each range of $128/D$ consecutive values, starting from 0, as n increases, the term $\lceil n \cdot D/128 \rceil$ does not change. Hence, bigger n will produce smaller number of memory transactions for the work-group. Thus, we should maximize n or minimize m in this case.

When $B = 1$ and A is some constant, we assume that A is bigger than the memory segment size, and a warp spans r rows of the work-group ($r = WARP\_SIZE/n$). Thus, the locations accessed by each row fall into different memory segments. Each row generates n different memory transactions. On the other hand, each column in a warp accesses r consecutive locations (because $B = 1$). Assuming these r consecutive locations start at the starting position of a memory segment, each column in a warp generates $\lceil r \cdot D/128 \rceil$ transactions. Moreover, the whole work-group generates $\frac{m}{r}n\lceil r \cdot D/128 \rceil$ or $\frac{W}{r}\lceil r \cdot D/128 \rceil$ transactions. Thus, we should maximize r or minimize n to minimize the non-coalescing factor.

If both A and B are one, the number of memory transactions generated by a warp becomes $\lceil WARP\_SIZE \cdot D/128 \rceil$. In turn, the whole work-group generates $\lceil WARP\_SIZE \cdot D/128 \rceil \cdot (m/r) = \lceil WARP\_SIZE \cdot D/128 \rceil \cdot (m \cdot$

$n)/WARP\_SIZE$ transactions. Thus, the number of transactions in this case is a constant without regard to the value of m and n. Any values of m and n will do.

The kernel might have more than one global memory references. Assume that either A or B is one, not both. Then, each access pattern will require either minimizing m or minimizing n. There are two cases. First, if all the access patterns require minimizing the size of the same dimension, say m, we easily find a work-group shape that has the smallest non-coalescing factor by selecting the minimum value of m.

Second, one reference prefers minimizing m and another reference prefers minimizing n. In this case, the work-group shape with minimal m or the work-group shape with minimal n do not produce an optimal non-coalescing factor. To find the work-group shape that produces the smallest non-coalescing factor in this case, we rely on profiling all possible work-group sizes. However, this does not happen at all in the kernels of the applications used (very rare in reality).

We denote the work-group size that is found to minimize the number of transactions by $(m_0 \times n_0)$. To find all the work-group sizes that produce this minimum number of transactions, we find all the work-group sizes that have a form of $(p \times q)$ so that p is a multiple of m_0 and q is also a multiple of n_0. Any such work-group size $(p \times q)$ will make the kernel have the same warp formulation with work-group size $(m_0 \times n_0)$, resulting in the same number of transactions. The output of the non-coalescing factor tuner is the list of work-group sizes $(p \times q)$ that have been found.

3.5.3 Concurrency Tuner

The input to the concurrency tuner includes the work-group sizes filtered by the workload tuner, the non-coalescing factor tuner, and the work-group sizes

Device name	**GTX 580**	**GTX Titan**	**GTX Titan X**	**FirePro W8000**
P	16	24.5	36	392
λ_0	0.52	0.62	0.88	∞

Table 3.3: Values for P and λ_0 for different GPU architectures.

that have not been considered by these two tuners.

The effect of cache contention is significant only if two conditions hold: cache contention occurs and the kernel performs memory operations intensively. The second condition can be checked by using the definition of λ that we defined in Section 3.4.2. As λ approaches one, the more likely that cache contention (if occurs) degrades the performance.

To find the threshold, denoted by λ_0, which tells if a kernel performs memory operations intensively, we use the micro-benchmark described in Section 3.4.3. We vary the non-coalescing factor (in this case $STRIDE$) starting from one (the fully coalesced case) and vary the work-group size so that the occupancy increases from the lowest value to the highest value. If we detect a decrease (of more than 10%) in performance when going from a lower occupancy to a higher occupancy, we stop and record the value (say λ_0) of λ at the higher occupancy.

If any work-group size of a kernel produces a λ higher than λ_0, the kernel is classified as a *memory intensive* kernel. For this kernel, if cache contention occurs, the degradation it causes to the performance is likely to dominate the benefit of high occupancy. Hence, this kernel achieves better performance at the work-group size that produces a low occupancy. Table 3.3 specifies the values obtained for P and λ_0 for different GPU architectures. As mentioned in Section 3.4.2, P represents the degree of memory transaction overlapping. If P is greater than one, the GPU allows more than one memory transactions at a time.

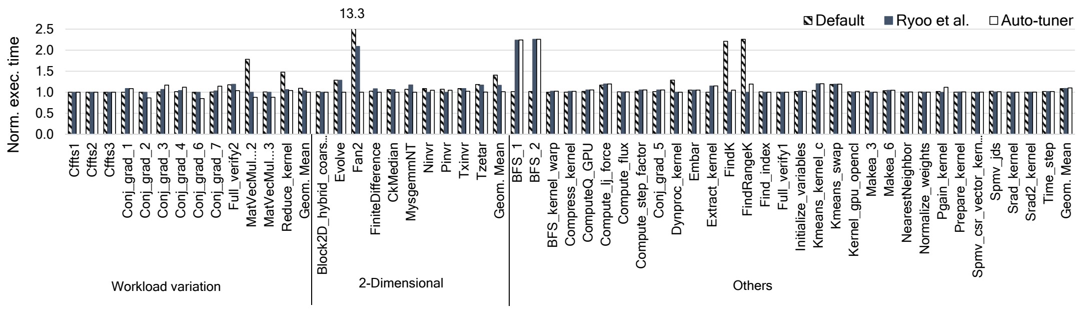


Figure 3.9: The performance of the auto-tuner on GTX Titan X.

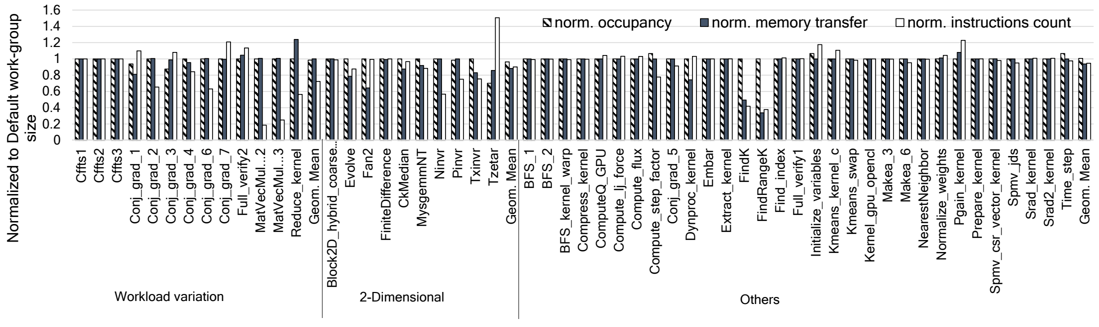


Figure 3.10: The performance of the auto-tuner on GTX Titan X with respect to the occupancy, the number of memory transfers and the instruction count.

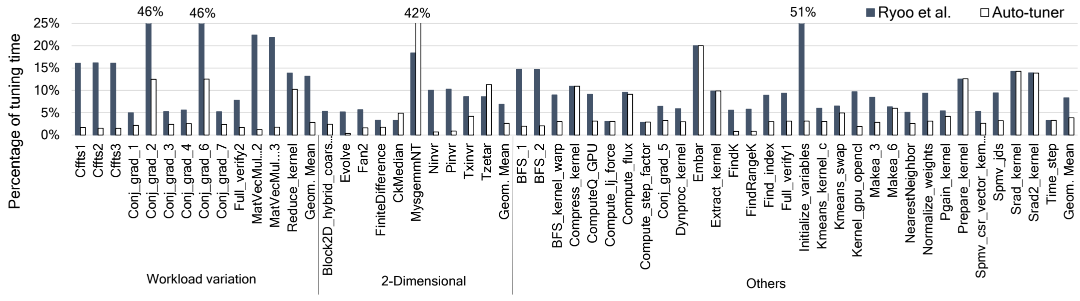


Figure 3.11: The tuning cost of the auto-tuner on GTX Titan X.

The value obtained for λ_0 becomes higher as we traverse from GTX 580, GTX Titan and Titan X. This is because the L1 caches in GTX Titan and

Titan X do not cache global loads. Instead, the L2 cache plays the major role in reducing the latency of memory accesses. Since the L2 cache size is much bigger than the L1 cache size, the condition at which the heavy cache contention occurs in GTX Titan and Titan X is relaxed, resulting in a higher value of λ_0. In addition, the L2 cache size of Titan X is higher than that of GTX Titan. This explains that λ_0 is higher in Titan X.

We do not detect heavy cache contention that causes performance variation in FirePro W8000 (we denote $\lambda_0 = \infty$ for this reason). In FirePro W8000, since the cache line size (64 Bytes) is smaller than that (128 Bytes) of NVIDIA GPUs, useful cache lines are less likely to be evicted before they are actually used. In addition, the occupancy variation between all possible work-group sizes is very small in FirePro W8000 (normally 5%-20%). We do not observe significant performance variation due to cache contention.

The two conditions described before can be checked using profiling information. We profile the kernel with two work-group sizes: one produces the lowest occupancy and the other produces the highest occupancy. If the cache miss rates produced by these two work-group sizes differ more than 10% (this value can be determined empirically depending on the GPU architecture), we conclude that the cache contention occurs. If the kernel has λ higher than λ_0 at either work-group size, we conclude that the kernel is sufficiently memory-intensive to be hurt by the cache contention.

If the two conditions hold, we select the work-group sizes that minimize the occupancy for the output of the concurrency tuner. Otherwise, we select the work-group sizes that maximizes the occupancy.

3.5.4 Exhaustive-search Tuner

After iterating through all the previous steps, the selected work-group sizes will be validated through an exhaustive search by the exhaustive-search tuner. Specifically, it takes the work-group sizes produced by the concurrency tuner and executes all these work-group sizes and selects one that produces the best performance.

3.6 Evaluation

Table 3.2 describes the kernels used in this evaluation. We do not select those kernels that have only very few number of work-groups because they cannot fully utilize hardware resources in a GPU. We also exclude kernels whose work-group size cannot be varied. We use three NVIDIA GPUs and one AMD GPU for the evaluation. Their specifications are summarized in Table 3.1.

3.6.1 Overall Tuning Quality

GTX 580, GTX Titan, and GTX Titan X. Figure 3.9 shows the result of the proposed auto-tuning techniques on GTX Titan X. We measure the execution time at the selected work-group size and normalize it to the best execution time of the original kernel. We compare our framework with the default work-group size (Default) and the auto-tuner proposed by Ryoo *et al.* (Ryoo et al.)[48]. Default represents the best work-group size of the kernel chosen on a previous architecture. This is typically the case where the programmer chooses a fixed value for the work-group size on the machine that the program is manually tuned.

On GTX Titan X, our work-group size is on average 1.06x slower than the original optimal work-group size (the work-group size that produces the smallest execution time of the original, untouched kernel). On GTX 580 and GTX Titan, our work-group size is on average 1.04x and 1.02x faster than the optimal work-group size, respectively. This is because our tuner transforms the kernel index space with a two-dimensional work-group size that is even better than the original one-dimensional optimal work-group size.

The average on GTX Titan X is slightly worse than two other GTX GPUs. This is because our auto-tuner does not work well on GTX Titan X for two kernels `BFS1` and `BFS2` due to the work-group scheduling overhead that is architecture dependent (Section 3.4.5). `BFS1` and `BFS2` have a very large number of work-groups (*e.g.*, millions), and the overhead is significant in this case. Other than this, the performance on GTX 580 and GTX Titan is similar to that of GTX Titan X.

Our method performs better than both Default and Ryoo et al. The performance gap between ours and Ryoo et al. mainly stems from the case where the kernel performance is strongly affected by the global memory usage, which was not considered in Ryoo et al. The gap is smallest on GTX Titan X since our selected work-group size is only 2% faster than the work-group size found by Ryoo et al. This gap is 7% and 9% on GTX Titan and GTX 580, respectively.

The most significant difference in tuning quality between ours and Ryoo et al. comes from considering the non-coalescing factor. This is manifested in `Fan2`, `evolve`, and `tzetar`. These kernels are 2-dimensional, and their performance is severely affected by the non-coalescing factor.

Performance Breakdown To further analyze the quality of the work-group size selected by our auto-tuner, we detail the performance difference between this work-group size and the default work-group size in Figure 3.10. Specifically, the performance is broken down into the occupancy, the number of memory transfers between the GPU and the global memory, and the number of

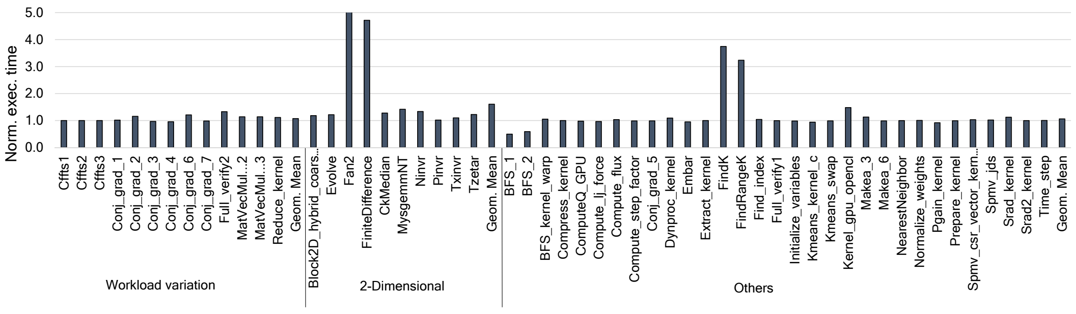


Figure 3.12: The performance of the auto-tuner in comparison with OpenTuner on GTX Titan X.

instructions executed by the GPU of the kernel. The occupancy, the number of memory transfers and the number of instructions of the work-group sized selected by the auto-tuner are normalized to the corresponding number of the default work-group size. Note that we omit the analysis for the cache miss rate because the effect of our auto-tuner to the cache contention is rather insignificant.

First, for the kernels that have the workload increased as the work-group size increases, the work-group sizes selected by the auto-tuner reduce on average 28% workload compared to the default work-group size. This is due to the effect of the transformation employed by the workload tuner. By considering the number of memory transfers between the GPU and the global memory, we realize that by changing the structure of the work-group, the kernel index space transformation can slightly change the number of memory transfers (as in `conj_grad_1` and `Reduce_kernel` with about 20% difference).

Second, for the 2-dimensional kernels, the work-group sizes selected by the auto-tuner reduce on average 10% of the memory transfers. This is because the auto-tuner correctly chooses the work-group shapes that minimize the number of global memory transactions per work-group. In addition, we observe 12% of instruction count reduction. This is attributed to the effect of the exhaustive-search tuner that searches among the work-group sizes having the same non-coalescing factor. The occupancy for this group of kernels is 4% lower than the default occupancy. This indicates that a work-group with high occupancy does not necessarily produce a high performance.

The third group of kernels show the less significant effect of the auto-tuner to the quality of the selected work-group size (only up to 5% difference with the default work-group size). This is because these are mostly regular kernels and the programmers can easily choose a good work-group size that works well for multiple architectures. Therefore, the auto-tuner has little chance to find better work-group sizes.

Comparision With OpenTuner. Figure 3.12 shows the comparison between our auto-tuner and OpenTuner [58]. This auto-tuner uses an evolution search strategy to find the work-group size using a fixed number profiling runs. For a fair comparison, we limit the number of profiling runs to be the same with the number of profiling runs that our auto-tuner spends for each kernel. The execution time of the work-group size selected by OpenTuner is normalized to the execution time of the work-group size selected by our auto-tuner. On average, OpenTuner finds an work-group size that is 15% slower than that found by our auto-tuner. The most significant difference stems from the 2-dimensional kernels. This is attributed to dramatic changes in the landscape of the 2-dimensional work-group size space that the evolution search cannot capture. This result shows that our tuning techniques are generally more efficient than evolution search strategies for tuning the work-group size.

FirePro W8000. Figure 3.13 shows the result of the proposed auto-tuning techniques on FirePro W8000. In this case, we do not show the result of the one-dimensional kernels because there are only four possible work-group sizes

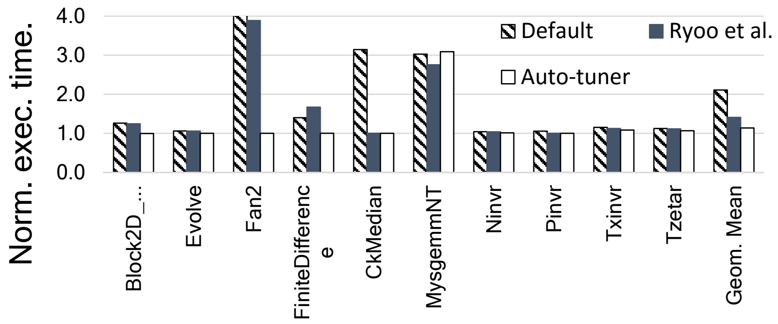


Figure 3.13: The performance on FirePro W8000.

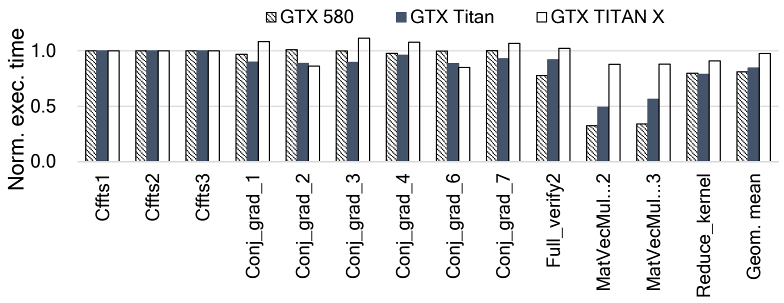


Figure 3.14: The effect of the workload tuner.

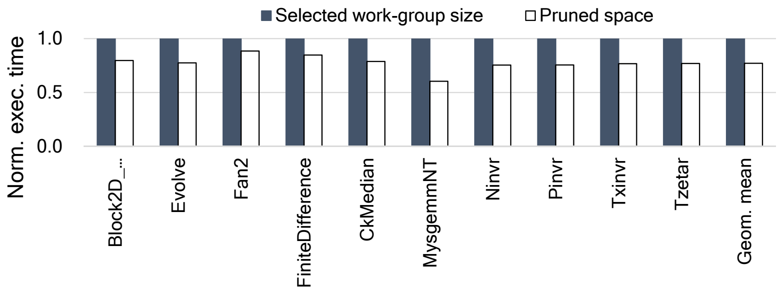


Figure 3.15: The effect of the non-coalescing factor tuner on GTX Titan X.

to exploit for them and the performance variation across the four sizes is very small. Instead, we focus on the two-dimensional kernels where the non-coalescing factor causes significant performance variations.

Since two-dimensional kernels are all sensitive to the choice of work-group shape, the gap between ours and Ryoo et al. becomes much bigger (33%). Our

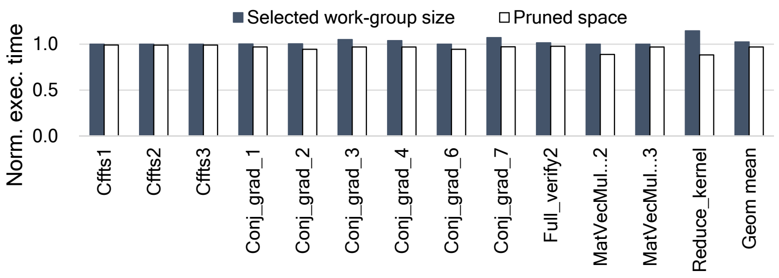


Figure 3.16: The effect of the concurrency tuner on the kernels with high workload variation on GTX Titan X.

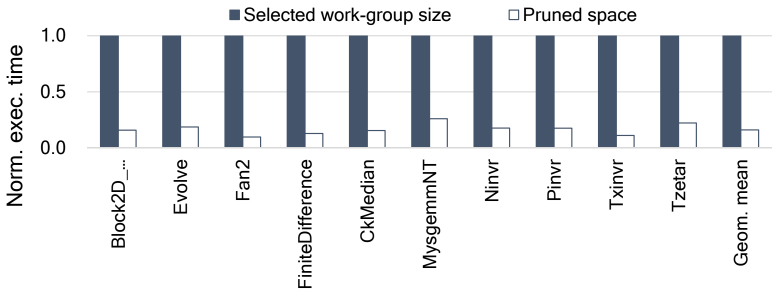


Figure 3.17: The effect of the concurrency tuner on the two-dimensional kernels on GTX Titan X.

technique does not work well with the kernel `MysgemmNT` that has significant unbalanced workload between work-groups. Our auto-tuner does not consider this factor.

3.6.2 Overall Tuning Cost

The tuning cost is the time taken to apply the tuning techniques to an application. This cost is dominated by the execution time spent on profiling the kernels (including the profiling time to determine whether a kernel needs to be hanled with workload tuner). We report the tuning time normalized to the execution time of an exhaustive search that simply executes all possible work-group sizes. Obviously, the more we profile, the closer this rate goes to one.

Figure 3.11 compares the tuning cost for each kernel between ours and Ryoo et al. Our tuner spends 3% and 8% of the time taken for the exhaustive search on GTX Titan X and FirePro W8000, respectively, to find a good work-group size. This is better than the tuning cost of Ryoo et al., which requires 9% and 21% on GTX Titan X and FirePro W8000, respectively. One exception is `MysgemmNT` that has only 19 valid work-group sizes. The small search space makes no difference between ours and Ryoo et al.

We do not compare with OpenTuner in terms of tuning cost because OpenTuner requires a number of profiling runs in order to stop the search.

3.6.3 Effect of the Workload Tuner

Figure 3.14 shows the effect of the kernel index space transformation by the workload tuner on three GPUs: GTX 580, GTX Titan, and GTX Titan X. We show the execution time of the selected work-group size when the output of the workload tuner is fed directly to the exhaustive-search tuner. It is normalized to the best work-group size in the original kernel. The result includes all kernels whose workload increases as the work-group size increases.

Overall, the workload tuner achieves a speedup of 1.23x, 1.17x, and 1.02x on GTX 580, GTX Titan, and GTX Titan X, respectively. The performance is even better than that of the best work-group size. Especially, two kernels `MatVecMulCoalesed2` and `MatVecMulCoalesced3` benefit from our transformation. The workload in these kernels is increased by both the number of reduction steps and the total number of work-items as the work-group size increases. The transformation by the workload tuner enables a kernel to reach the highest occupancy without increasing the workload.

For FirePro W8000, we do not apply the kernel index transformation because even the smallest work-group size (a work-group size of 64) can achieve full occupancy, thus has good performance. This is because the maximum number of active work-groups in FirePro W8000 is very large (40 work-groups). Hence, if the number of active work-groups is not contrained by the number of registers and the amount of local memory, the work-group size 64 can produce 40 active work-groups and thus reaches the maximum occupancy.

3.6.4 Effect of the Non-coalescing Factor Tuner

Figure 3.15 shows the quality of the selected work-group by the non-coalescing factor tuner when its result is fed to the exhaustive tuner. It also shows the search space that the non-coalescing factor tuner prunes, compared to the original space.

Bars marked with Selected work-group size show the execution time of the selected work-group size. It is normalized to the best work-group size. Bars marked with Pruned space show the amount of search space that the tuner has pruned. It is the rate of the accumulated execution time of the kernel with the work-group sizes pruned by the tuner to that with all the work-group sizes.

Overall, the non-coalescing factor tuner is able to select the near-optimal work-group sizes for most of the two-dimensional kernels. On average, the performance of the selected work-group size is the same as that of the best work-group size on GTX Titan X. The amount of search space (in terms of execution time) compared to the original space that non-coalescing factor tuner can prune is 78%.

3.6.5 Effect of the Concurrency Tuner

The concurrency tuner acts like a second-level filter for the workload tuner and the non-coalescing factor tuner. Thus, to see its effect on the final selection, we compare the results obtained in Section 3.6.3 and Section 3.6.4 with the result produced by the whole framework.

Figure 3.16 shows the performance of the selected work-group size by the concurrency tuner and the amount of search space that it prunes. The execution time of Selected work-group size is normalized to that of the best work-group size selected by the workload tuner. In other words, we compare the performance of the auto-tuner with and without the concurrency tuner for the kernels having high workload variation. The comparable performance of the work-group size selected by the concurrency tuner with that selected by the workload tuner indicates that the concurrency tuner does not discard the good work-group sizes among the work-group sizes it takes from workload tuner during its pruning process.

On the other hand, the bar marked with Pruned space indicates how much of the space that the concurrency tuner prunes compared to the total amount of space that the workload tuner produces. The amount of space pruned is more than 97% of the space that the workload tuner produces, on average.

Similarly, Figure 3.17 shows the result for the two-dimensional kernels. The concurrency tuner can preserve the good work-group size provided by the non-coalescing factor tuner because the best work-group size that it produces is very close to the best work-group size that the non-coalescing factor tuner produces for all the two-dimensional kernels. The bar marked with Pruned space indicates the search space that the concurrency tuner prunes compared to the total amount of space of two-dimensional kernels. The contribution of the concurrency tuner to pruning for these kernels is rather limited (only 16% of the original work-group sizes are pruned). This is because the non-coalescing factor tuner has pruned most of the work-group sizes (77% of the original work-group sizes).

3.7 Conclusions

In this chapter, we characterize the relationship between the work-group size and the OpenCL kernel performance on GPUs. By analyzing the micro-benchmarks and a large number of OpenCL kernels, we identify the performance factors that have the most significant impact to the overall performance with regard to the work-group size. The identified performance factors include occupancy, coalesced global memory accesses, cache contention, and the variation in the amount of workload of the kernel. Based on these performance factors, we propose an auto-tuning framework for OpenCL on GPUs. Our framework handles the performance factor one-by-one. To facilitate the tuning framework, we propose a kernel index transformation that both improves the performance of the kernel and helps the auto-tuner choose the work-group size more easily. Our experiments on four different GPUs (three from NVIDIA and one from AMD) show that the auto-tuner can find a good work-group size in a reasonable amount of time. When compared with a state-of-art work-group size tuning methods, we obtain better tuning results in shorter execution time. Our auto-tuner is a good alternative to the GPU performance tuning frameworks that currently use an exhaustive search to tune the work-group size.

Chapter 4

Quantization for Deep Learning Programs

4.1 Introduction

Under the vast development and success of deep learning applications, there is a tendency to deploy state-of-the-art deep learning models into devices with more limited memory capability than conventional servers. Thus, there has been a tremendous effort to effectively quantize the model's weights and activations to reduce the model size. Weight quantization reduces the model size, and activation quantization reduces the memory footprint at the inference phase. To achieve the best result, both weights and activations are often quantized together.

Quantization [17, 18, 19, 20, 21] is not only beneficial for lowering the memory requirement, but it also provides a computational improvement in the inference phase. It is because many devices perform faster at lower precision, such as INT8. For example, Edge TPUs have hardware support for 8-bit integer computation [22]. Recently, TensorRT also has 8-bit integer mode computation support [16, 78].

The key to successful quantization is to choose proper quantization parameters. For integer quantization methods, the two important parameters include the *quantization scale* and the *precision* [18, 79]. These parameters determine how to map a floating-point value to an integer value. Intuitively, the scale characterizes the gap between two consecutive quantized elements, and the precision dictates how many bits in which the quantized integer values can be represented.

Choosing a proper value for these parameters at training time is challenging [21] as it depends on many unknown factors. Previous approaches aim to optimize the quantization scale [17, 18, 19] and the precision [18] by learning them together with the model weights. In general, these approaches can find the parameters that minimize the loss function, *i.e.*, the network learns better from the training data.

However, we approach the quantization in deep learning from an entirely different perspective. We draw a connection between the quantization parameters to the generalization ability of the network. We show that even a simple realization of this idea can improve the quantization quality in many scenarios.

More specifically, we propose two new techniques to adjust the quantization parameters during the training phase so that the inference accuracy is significantly improved. One technique randomly adds some noise to the computation of the quantization scale. The degree of noise to be added is monitored through the training phase. The noise introduces uncertainty to the quantization scale in the training phase. Thus, the model is more robust to unseen data in the inference phase. Adding uncertainty to achieve better generalization has been extensively studied in the literature of machine learning [80, 81, 82].

The other technique learns the precision for each layer to minimize the Kullback Leibler divergence [83] with the target precision. The proposed technique dynamically finds an appropriate schedule between maximizing the learning ability and maximizing the generalization ability of the network.

The contributions of this paper are summarized as follows:

- We introduce a novel technique to improve the model's generalization ability by adding random noise to the quantization scale at the training time.
- We introduce a novel technique to dynamically schedule the integer quantization precision at the training time that improves the validation accuracy.
- Our techniques are simple and intuitive. They can be easily applied to any layer.

- We evaluate our techniques using 8-bit integer quantization. The results are reported with different network architectures, learning tasks, optimization algorithms, and datasets. The results indicate that our techniques are effective and practical. They are applicable to a wide range of model architectures and input data.

4.2 Related Work

Many quantization studies show different levels of success with different precisions. Binary quantization [84] quantizes the weights with two-bit integers. The two-bit integer quantization is insufficient to retain competent accuracy for more complex models and datasets [85]. Misha *et al.* [20] show that the accuracy loss can be recovered by increasing the number of filters for convolutional networks. However, the added filters may result in extra computation time compared with the original model. Some studies [86, 87] show that 16-bit quantization is sufficient to achieve good accuracy with many networks by using simple quantization functions.

8-bit integer (INT8) quantization seems to be a more reasonable balanced trade-off between the accuracy and the performance as recent architectures have better support for INT8. Some studies show that the 8-bit quantization of state-of-the-art models for large datasets like ImageNet is challenging [85, 21], especially for batch-normalization layers [85]. This type of layers performs more expensive operations (*e.g.*, square-root operations) than other types of layers. These operations typically require higher precision than addition and multiplication operations. Since quantizing batch-normalization layers results in a significant accuracy drop in general, it is a common practice to leave this layer unquantized. Dorefa-net [79] provides a fundamental approach to quantizing weights and activations to arbitrary precision. This paper also shows that the last fully connected layer and the output layer should not be quantized to avoid a significant accuracy drop. We also follow these practices in this paper.

The quantization function proposed by Dorefa-net is fundamental to many recent studies. The quantization scale is the key parameter to this function. Traditionally, a tensor's maximum value is used to compute this value [79, 21, 85]. However, depending on the tensor values distribution, this choice may not be optimal [21]. Recently, Choi *et al.* [17] propose a method to learn the quantization scale. However, this method applies to only ReLU-based models. Similarly, some other studies [18, 19] introduce the quantization scale as a learnable parameter. These approaches require a pretrained model (in full precision) and fine-tune the model with quantization. However, there is no evidence that these methods are appropriate for training from scratch.

Quant_noise [88] randomly selects the elements of a tensor to quantize during the training phase. This method allows two precisions in training: 32-bit floating-point (FP32) and the target bit-width used in testing. Our method is different in that we quantize for different integer precisions at different stages of the training phase.

Regarding adding uncertainty to the training phase, many previous studies perform this in different ways [80, 81, 82, 89, 90]. Moving average [89] may cancel certain noise when computing the max value of a tensor. Some work directly add Bernoulli noises [90] to the output of the quantization function. However, none has considered adding the uncertainty specifically to the quantization scale. Even though there has been some effort to quantize both weights and activations of transformer models [91, 92], their quantization scale is based on the maximum value of the tensor.

Regarding optimizing the precision during the training phase, previous work treat the precision as a learning parameter and update it to optimize the conventional training loss function [93, 18]. However, the initial precision need to be initialized within a small range with the target precision.

4.3 Background

In this section, we describe commonly used integer quantization techniques.

4.3.1 Integer Quantization

In an integer quantization method, the floating-point values in the network are replaced by fixed-point numbers. A fixed-point number is generally represented by an integer value and a scale value [21]. The scale value characterizes the gap between two consecutive quantized elements.

We assume that the value x to be quantized is non-negative (*i.e.*, $x \geq 0$) for the simplicity of discussion. Handling negative values will be presented in the following section. Let $q_{n,b}(x)$ be the b-bit integer after quantizing an n-bit floating-point number x with the basic integer quantization. The value b, the precision of the quantization, is the maximum bit width allowed to represent the quantized values. Let s be the scale value. The quantization function q_I for the basic quantization is defined as follows:

$$q_{n,b}(x, s) = clamp(round(x \cdot s), 0, 2^b - 1) \tag{4.1}$$

where *clamp* ensures the quantized value is in the range that can be represented with b bits, which is $[0, 2^b - 1]$ in this case. It is defined by

$$clamp(x, l, u) = \begin{cases} l & \text{if } x < l \\ u & \text{if } x > u \\ x & \text{otherwise,} \end{cases}$$

and the *round* function rounds a number to the nearest integer with half-way values rounded up:

$$round(x) = \lfloor x + 0.5 \rfloor .$$

Function $d_{b,n}(y, s)$ dequantizes the b-bit quantized value y to an n-bit floating-point number, *i.e.*, it maps the b-bit quantized integer y to an n-bit floating-point number:

$$d_{b,n}(y, s) = \frac{1}{s} \cdot y \tag{4.2}$$

where the value y is an integer. The quantity $1/s$ is the gap between two consecutive elements in the dequantized values. This quantity is often referred to as the *step size* of the corresponding quantization function.

In practice, we apply the quantization function to a tensor. Each tensor has a common quantization scale for the values stored in it. Thus, to represent a tensor, we can replace the original floating-point values with integers and extra storage space for the quantization scale. A floating-point number represents the scale.

To compute the quantization scale $s_{\mathbf{x}}$ of a tensor $\mathbf{x}$, it is common to base on the maximum value of $\mathbf{x}$'s elements:

$$s_{\mathbf{x}} = \frac{2^b - 1}{MAX(\mathbf{x})} \tag{4.3}$$

From Equation 4.1, the information loss stems from two sources. One is the *round* function that causes the difference between the original floating-point value and the rounded value. The other is the *clamp* function that cuts away the rounded value if it exceeds $2^b - 1$.

Using the MAX values in Equation 4.3 prevents the output of the round function in Equation 4.1 to be out of the bound $[0, 2^b - 1]$, and thus avoid the clamp function from causing the information loss.

However, using MAX might cause a large difference between the original floating-point value and its quantized value in Equation 4.1. In practice, a value smaller than MAX will produce a larger quantization scale, which may improve overall accuracy. Thus, it is crucial to find the trade-off between the quantization space's granularity and the information loss due to the clamp function.

4.3.2 Standard Techniques Used

There are several standard quantization techniques that we employ in this work.

Handling negative values. The quantization function 4.1 assumes that the original floating-point value is non-negative. To handle the case where the original values contain negative numbers, we use the range $[-2^{b-1}, 2^{b-1} - 1]$ in Equation 4.1 that is the range of numbers in b-bit two's complement representation. The scale computation for a tensor $\mathbf{x}$ becomes:

$$s_{\mathbf{x}} = \frac{2^b - 1}{2MAX(|\mathbf{x}|)} \tag{4.4}$$

where $MAX(|\mathbf{x}|)$ is the maximum value of the absolute values of $\mathbf{x}$'s elements.

Affine quantization. One can explicitly introduce the minimum value of the tensor elements to the quantization function [89]. This method in general yields a larger quantization scale than that in Equation 4.3. The scale for a tensor $\mathbf{x}$ can be computed as:

$$s_{\mathbf{x}} = \frac{2^b - 1}{MAX(\mathbf{x}) - MIN(\mathbf{x})} \tag{4.5}$$

where $MIN(\mathbf{x})$ is the minimun value of $\mathbf{x}$'s elements. However, using the minimum value might introduce some additional computation cost [21]

Straight-through estimator. It is not efficient to calculate the exact derivative of the quantization function since it will be zero almost everywhere. Thus, we apply a commonly used gradient approximation method, called straight-through estimator [94], to backpropagate the gradients.

4.4 Quantization Framework

In this section, we detail our methodology to quantize deep neural networks. The goals of our approach are reducing the model size, achieving high accuracy, and running faster in the inference phase. We assume that the original computational precision of a given network is 32-bit floating-point (FP32).

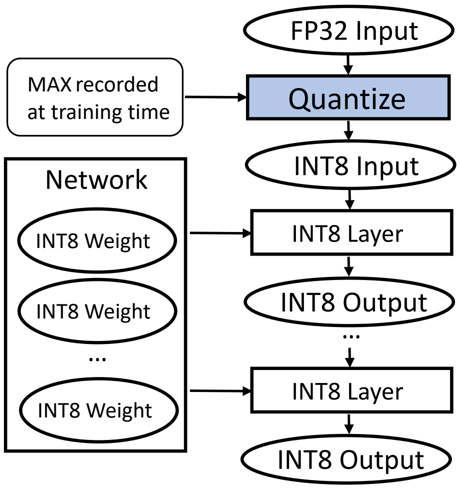


Figure 4.1: The inference phase in our basic 8-bit integer (INT8) quantization framework.

4.4.1 Inference Phase

Figure 4.1 illustrates the inference phase in basic 8-bit integer (INT8) quantization framework. In the inference phase, the model weights are presented in the INT8 format. Only the input is required to be quantized to INT8, and the rest of the network functions in the INT8 mode. To compute the quantization scale for an input tensor, we use the quantization scale that was recorded from the last iteration of the training phase.

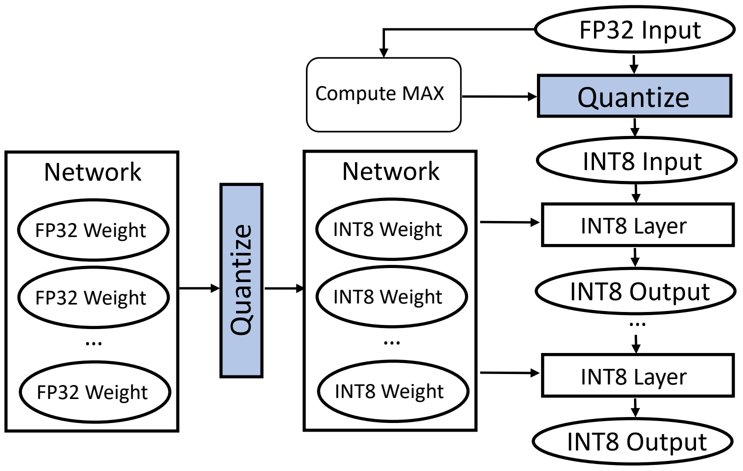


Figure 4.2: The forward pass in the training phase of our basic INT8 quantization framework.

4.4.2 Training Phase

Figure 4.2 shows how our INT8 quantization approach applies to the forward pass in the training phase. Before an INT8 layer performing its computation, its FP32 weights and FP32 input are quantized to INT8. The quantization scale is based on the maximum value. However, since our framework focuses on the inference phase's performance, the backward pass operates in the FP32 mode. The update is committed to the FP32 copy of the model weights.

4.4.3 Adding Noise to the Scale

To improve the quantization quality for activations, we introduce a technique to add noise to the computation of the quantization scale in the training phase. This will help the model generalize better to new data whose value range of the activations are quite different from that of the training data.

Observation I. Figure 4.3 shows the histogram of the input activation values of the first ReLU layer in the EfficientNet-B0 model. The x-axis represents the

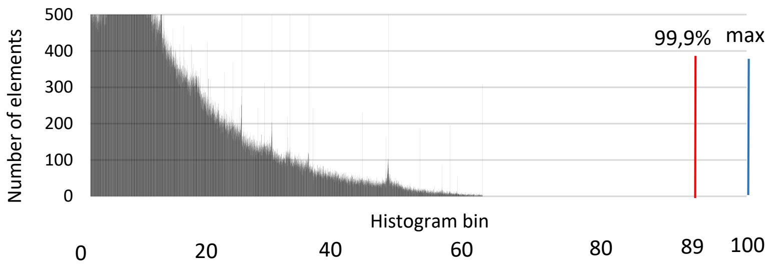


Figure 4.3: The histogram of input activations of layer relu0 in EfficientNet-B0 with CIFAR10.

values and the y-axis represents the frequency. It is noteworthy that 99.9% of the activation map are smaller than the value marked by the red line. In other words, only 0.1% of the activation maps belong to the 11% of the entire range (between the maximum value 16.5 and the value 14.6 at 99.9%). However, it is shown that simply removing these 0.01% activations may severely degrade the accuracy in many cases [21]. Several activations among these 0.01% might affect the accuracy significantly. The degree of sensitivity is different between different layers and also between different input data.

From another view, when the quantization scale is computed based on a smaller value than the maximum element of a tensor $\mathbf{x}$, say α, the activations within the range $(\alpha, MAX(\mathbf{x})]$ will be clamped into the value α by Equation 4.1. This results in two effects. One is that this can be seen as adding certain noise to these activations. From the perspective of stochastic training, adding noise is known to help the model generalize better. The other is that computing the quantization scale based on a value smaller than the maximum element reduces the gap between two consecutive elements in the quantization space as explained in Section 4.3.1. Consequently, choosing a value smaller than the maximum value has two advantages: helping the neural network generalize better and decreasing the quantization granularity. With smaller granularity, the information loss caused by the quantization is also smaller.

Stochastic scales. Based on the observation, instead of using the maximum value, we choose a random value that is smaller than the maximum value to compute the quantization scale.

Let MAX and MIN be the values of the maximum and minimum elements in a tensor $\mathbf{x}$, respectively. The computation of the quantization scale for $\mathbf{x}$ becomes:

$$\begin{aligned} MAX_{new} &= MAX - random(0, p * MAX) \\ s_{\mathbf{x}} &= \frac{2^b - 1}{MAX_{new} - MIN} \end{aligned} \tag{4.6}$$

where p is a parameter that determines how far from MAX the random value is. The function *random* returns a value in the range $[0, p \cdot MAX]$

Since each activation tensor might have different distribution of element values, especially if they belong to different types of layers, we make p specific to each activation tensor. To do so, for each activation tensor of the network, we maintain a histogram of the tensor values at the training time. Based on the histogram, p is chosen so that the ratio of the number of elements in the tensor falling into the range of $[MAX_{new}, MAX]$ to the total number of elements does not exceed a predetermined threshold. This threshold is empirically determined based on the model. We find that a simple value, such as 0.001% or 0.005%, is sufficient for most networks. The histogram that is used to calculate MAX_{new} is generated once after each epoch.

Partial quantization. Common practices [85, 21, 79] suggest that some of the layers should be computed in high precision to avoid significant accuracy reduction. From our experience, we do not quantize the batch-normalization (Batchnorm), layer-normalization (Layernorm), and GeLU layers. These layers perform transcendental operations that require high precision to maintain

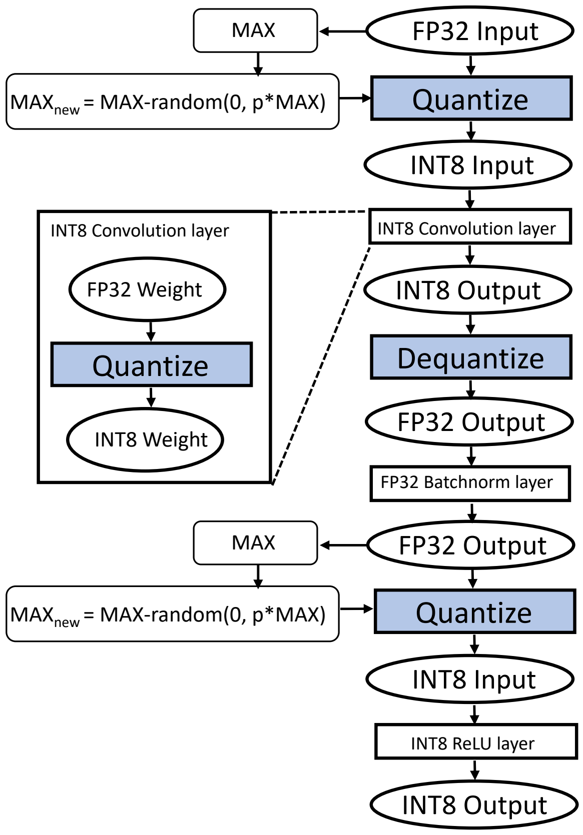


Figure 4.4: The partial quantization scheme.

adequate accuracy. These layers typically occupy only a small portion of the model size. The time taken to process them is also very small.

For such a case, we insert a *dequantization* (Dequantize) operation to convert INT8 values to FP32 values before forwarding them into a layer that is not quantized. Figure 4.4 illustrates how to perform the partial quantization. In Figure 4.4, we do not quantize the batch normalization (denoted by FP32 Batchnorm in Figure 4.4) layer. Thus, the INT8 output of the previous convolution layer will be dequantized to FP32 before being fed into the next Batchnorm layer. Then, the FP32 output of the Batchnorm layer is quantized to INT8 before going into the next INT8 layer, ReLU.

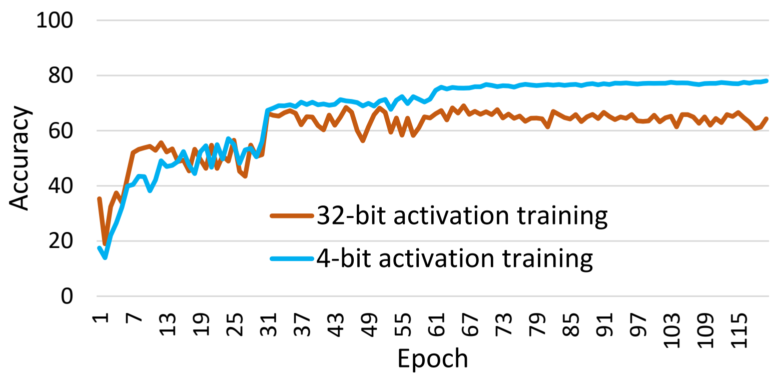
4.4.4 Adaptively Adjusting Precisions

Observation II. We perform a set of experiments to explore the connection between the precision (*i.e.*, the integer bit width) used during the training phase and the validation accuracy. It is common to believe that the training and validation precisions are the same. However, we show that this is not always the case. Specifically, it depends on training stages.

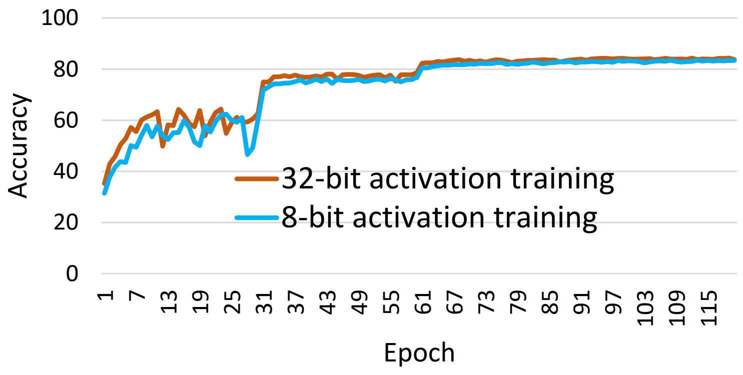
As an example, we quantize EfficientNet-B0 with the CIFAR10 data set. We quantize its convolution and ReLU layers. Their activations and weights (if they exist) are quantized. To obtain the validation accuracy at the specific precision, we insert the quantization modules with the desired precision before each layer to be quantized. This process emulates the accuracy as if the model is executed with real hardware support.

Figure 4.5(a) shows the validation accuracy when we use INT4 for validation when the model is trained with 32-bit integers (INT32) and INT4 in the forward pass. Note that we use FP32 in the backward pass in training. Figure 4.5(b) shows similar experiments but we use INT8 for validation when the model is trained with 32-bit integers (INT32) and INT4 in the forward pass. To see the effect of different precisions used in activation quantization, all experiments in Figure 4.5 use INT8 for weight quantization. The validation accuracy is reported at the end of each training epoch.

Figure 4.5, the teal lines represent the case where the training precision is equal to the validation precision, say Case E, and the orange lines represent the case where training precision is larger than the validation, say Case L. For both Figure 4.5(a) and Figure 4.5(b), the validation accuracy at the end of the training for Case E is better or comparable to that of Case L. This indicates



(a) INT4 validation



(b) INT8 validation

Figure 4.5: Validation accuracy comparison between different precisions used to quantize activations: (a) INT4 (4-bit integers) and (b) INT8. The weight precision is INT8 for all experiments.

that the validation precision is important and hence, it should be used at the training phase. This is consistent with the common knowledge.

However, if we look closer at the divergence between Case E and Case L, Case L performs better than Case E at the early stage of the training. Specifically, for the 4-bit validation case in Figure 4.5(a), the 32-bit version performs better from epoch 1 to epoch 15. Meanwhile, for the 8-bit validation case in Figure 4.5(b), the 32-bit version performs noticeably better from epoch 1 to epoch 19.

Our proposition for this phenomenon is that at the beginning, when the network is not yet stable, *i.e.*, when the update to the weights is large, higher precision are more helpful to learn critical information from the training data. As the training continues, the changes being made to the weights become smaller, the network moves into a more stable state, and the higher precision is no longer helpful. At this point, the training favors the precision that is close to the one used in validation.

Adaptive precision scheduling. The observation suggests adjusting the training precision to obtain better validation accuracy as the training goes by. Here, we propose an approach to learn the training precision. Intuitively, we we start the training with an initial precision that is significantly larger than the validating precision (e.g., 32-bit) and we modify the precision in order to minimize the Kull-back Leibler divergence.

For two discrete probability distributions P and Q, the Kullback-Leibler divergence $D(P||Q)$ defines the relative entropy from Q to P, which is calculated as follow:

$$D(P||Q) = \sum_{x \in \boldsymbol{X}} P(x) log \frac{P(x)}{Q(x)} \tag{4.7}$$

where $\boldsymbol{X}$ is the common probability space of P and Q. An intuitive interpretation of this relative entropy can be seen as the information gain (or loss) obtained if P would be used instead of Q.

In our framework, for each activation tensor, we maintain a parameter b to represent the training precision for this tensor. This value will be the input to the function 4.1. Initially, b is set to 32 (32-bit). This is a good initialization as we observed in the previous section. To search for the optimal values of the training precision distribution, we update b as follow:

$$\begin{aligned} b &= b - \beta \Delta b \\ \Delta b &= \frac{\delta D(P_T||P_b)}{\delta b} \end{aligned} \tag{4.8}$$

Here, P_b is the probability distribution of the tensor as the network is trained using b-bit for quantization and P_T is the desired probability distribution of the tensor at the inference phase using T-bit quantization. Intuitively, we adjust b so that the extra bits needed to encode the distribution using b bits rather using T bits is minimized. Note that, we can also use $D(P_b||D_T)$ to characterize the difference between two probability distributions since this function an asymmetric.

The gradient Δb can be approximated as follow:

$$\begin{aligned} \frac{\delta D(P_T||P_b)}{\delta b} &= \frac{\delta \sum_{x \in \boldsymbol{X}} P_T(x) log \frac{P_T(x)}{P_b(x)}}{\delta b} \\ &= \frac{\delta \sum_{x \in \boldsymbol{X}} P_T(x)(log P_T(x) - log P_b(x))}{\delta b} \end{aligned} \tag{4.9}$$

Since $logP_T(x)$ does not depend on b, we have $\frac{\delta log P_T(x)}{\delta b} = 0$. Hence,

$$\begin{aligned} \frac{\delta D(P_T||P_b)}{\delta b} &= -\frac{\delta \sum_{x \in \boldsymbol{X}} P_T(x) log P_b(x)}{\delta b} \\ &= -\sum_{x \in \boldsymbol{X}} P_T(x) \frac{\delta log P_b(x)}{\delta b} \\ &= -\sum_{x \in \boldsymbol{X}} P_T(x) \frac{1}{P_b(x) ln2} \frac{\delta P_b(x)}{\delta b} \end{aligned} \tag{4.10}$$

Using the chain rule, we have:

$$\frac{\delta D(P_T||P_b)}{\delta b} = -\sum_{x \in \boldsymbol{X}} P_T(x) \frac{1}{P_b(x) ln2} \frac{\delta P_b(x)}{\delta x} \frac{\delta (x)}{\delta b} \tag{4.11}$$

Here, x is the quantized value (from a floating point value x_f) using b bits. Taking the derivatives of 4.1, we have

$$\frac{\delta(x)}{\delta b} = \begin{cases} -\frac{2^{b-1} - 1 log(2)}{2^{b-1}} & \text{if } x_f < MAX \\ 0 & \text{otherwise} \end{cases} \tag{4.12}$$

The quantity $P_b(x)$ can be approximated from a histogram of the values of the tensor that P_b represents. The quantity $\frac{\delta P_b(x)}{\delta b}$ can be effectively approximated by using a simple form of finite difference method:

$$\frac{\delta P_b(x)}{\delta b} \approx \frac{P_b(x+\epsilon) - P_b(x)}{\epsilon} \tag{4.13}$$

where x and $x+\epsilon$ belong to two consecutive bins of the histogram.

We approximate P_T using T-bit quantization from the floating-point tensor at training time.

4.4.5 Computation of Histogram

Both of our techniques use the histogram of the activation tensors. In this section, we discuss how histograms are used in our framework. For each tensor that will be quantized during the training phase, we compute its histogram periodically after each τ iteraions. To cancel the high variation between each iteration, the histograms are averaged over η iterations. We observe that τ and η can take values so that the overhead due to the histogram computation is negligible compared with the total execution time. After each $\tau + \eta$ iterations, we update the parameter p and the precision b for each activation tensor.

In our experiments, we also perform sensitive analysis with various values of τ and η.

4.5 Experiments

In this section, we detail our experiments. To ensure the comparisons are fair, we ensure that the comparisons quantize exactly the same layers for all networks used. We present our experimental results for two different task domains: image processing and natural language processing. We will refer to the method of using the maximum value of the tensor to compute the scale as the MAX method (in

Section 4.4) in our experiments. FP32 denotes the original 32-bit floating point version of the network.

Table 1: The networks used in this paper.

Network	Source	Number of parameters	Task
Resnet18	[95]	11.25M	Image classification
Resnet50	[95]	23.5M	Image classification
Resnet101	[95]	42.5M	Image classification
Resnet152	[95]	58.1M	Image classification
EfficientNet-B0	[96]	2.9M	Image classification
MobilenetV2	[97]	2.3M	Image classification
Transformer	[98]	213M	Machine translation
LSTM	[99]	154M	Machine translation
BERT	[100]	108M	Text classification
GPT-2	[101]	124M	Language modeling

Table 1 lists up the networks used in our experiments. We use multiple versions of the Resnet architecture because they have different numbers of parameters and expose different behavior. It is easy to analyze the effect of our techniques on the network size.

To preserve accuracy, we do not quantize the batch-normalization, last fully-connected, and output layers in CNNs. Also, EfficientNetB0 uses sigmoid to compute the activation. This function requires high precision to obtain good accuracy. Thus, we do not quantize the activation layer that uses sigmoid. For the other networks, we do not quantize the layer-normalization and GeLU layers. In general, the number of parameters in the layers that we do not quantize is less than 0.1% of the total number of parameters. Thus, the proposed quantization technique reduces the network size by approximately 1/4 (from FP32 to INT8).

We compare our techniques with severals state-of-the-art techniques. The first is PACT [17]. PACT introduces a learnable parameter for each ReLU layer.

This parameter determines the quantization scale that is applied to the output of the ReLU layer. Other layers use the MAX method to quantize their input activations. We do not perform PACT on the networks that do not contain any ReLU layer.

The second is Quant_noise [88], denoted *Quant* in our experiment. In this work, quantization is applied to a random portion of the activation map during the training phase with a predetermined probability. For example, in the training phase, 20% of the activation is quantized, and 80% is kept in full precision. In the inference phase, 100% of activations are quantized. We use 20% for the probability as recommended in their paper. This scheme currently only applies to convolutional layers and fully-connected layers. Other layers use the MAX method.

The other two work include Fracbits [93] and DQ [18]. In these work, the precision is modified so that the conventional loss function is minimized and therefore, it can be treated a regular parameter. Heuristic methods are used to approximate the gradient of the precision.

We evaluate two of our schemes. One is to add randomness to the computation of the quantization scale. We denote this method *Rand*. The other is to combine *Rand* with the idea of using mixed precision of integer quantization that we present in Section 4.4.4. We denote this method *Rand+mix*. We use Pytorch1.4 [14] to implement our techniques. For experiments that show high variations due to the random initialization, we repeat each experiment 4 times with different seeds and select the highest result. These experiments include the networks with CIFAR10 and natural language processing tasks. We generally use the default hyper-parameters from the Imagenet benchmark suit from Pytorch repository for the experiments.

4.5.1 Image Classification Tasks

For the image classification task, we evaluate our techniques with two datasets: CIFAR10 and ImageNet. We report the accuracy as the percentage of the correctly predicted images to the total number of images. To show that our techniques are generic, we show the results for two optimization algorithms: Stochastic Gradient Descent (SGD) [102] and ADAM [103].

Ablation study. We conduct experiments on CIFAR10 with SGD optimizer to study the effect of our proposed techniques in an independent manner on two networks: Resnet18 and EfficientNetB0.

To demonstrate the effectiveness of our stochastic method, we compare with two other stochastic based methods. The first method, *Moving Average* [89], uses an moving average variable of the max values of each tensor. The second method, *Bernoulli* [90], directly adds Bernoulli noises to the output of the quantization function.

To demonstrate the effectiveness of our mix-precision method, we compare with a previous work [104] that proposed to schedule the training precision from higher to smaller value. The version *ScheduleA* schedules the precision with a sequence of (32, 16, 8) as in the original paper. We additionally include the version *ScheduleB* that schedules the precision that starts with 32 bit and decreases the precision by one after each epoch until it reaches 8 bit.

Table 2 shows the experiment results. Our *Rand* outperforms other stochastic based methods, especially for EfficientNetB0. On the other hand, our *mix* outperforms the two versions that schedule the precision in a fixed manner.

To analyze the sensitivity of the accuracy to the choices of τ and η in the Section 4.4.5, we perform the experiments over three different combinations of these values and the result is shown in Table 3. The combination ($\tau = 100$

Table 2: Ablation Validation Accuracy (CIFAR10 and SGD)

Network	FP32	MAX	Moving Average	Bernoulli	Rand	ScheduleA	ScheduleB	Mix
Resnet18	94.53	94.1	94.46	93.83	94.5	94.48	94.54	94.67
EfficientNetB0	85.75	83.52	84.01	84.03	86.6	86.51	86.2	86.85

$\eta = 4$) produces the most reasonable results so we use these values for all other experiments.

Table 3: Validation Accuracy for rand+mix for different histogram settings (CIFAR10 and SGD)

Network	$\tau = 200$ $\eta = 2$	$\tau = 100$ $\eta = 4$	$\tau = 50$ $\eta = 8$
Resnet18	94.8	94.8	94.83
Resnet50	93.92	94.91	93.8
EfficientNetB0	86.02	86.85	85.13
MobileNetV2	90.39	91.02	90.52

As the update rule specified by Equation 4.8 suggests the the precision might be a floating point value, we also compare the results when using the precision as float and integer. In the case of integer, the precision is rounded up to the nearest integer before going to the quantization function. Using the precision as a float or an integer might have an impact to the complexity of the implementation with real hardware support (in the training phase). The comparison is presented in Table 4. As we can see, there is no significant difference between the two settings. Therefore, we use integer precision in all other experiments.

Table 4: Validation Accuracy for rand+mix for different precision settings (CIFAR10 and SGD)

Network	Floating point precision	Integer precision
Resnet18	94.8	94.77
EfficientNetB0	86.85	86.79

Table 5: Validation Accuracy (CIFAR10 and SGD)

Network	FP32	MAX	PACT	Quant	Fracbits	DQ	Rand	Rand+mix
Resnet18	94.53	94.1	94.2	94.41	94.44	94.04	94.23	**94.72**
Resnet50	94.66	93.42	93.92	92.05	93.54	94.54	93.66	**94.71**
Resnet101	95.03	92.75	93.59	93.15	93.84	94.26	92.88	**94.37**
Resnet152	94.95	93.38	93.76	93.47	92.95	90.55	92.95	94.33
EfficientNet-B0	85.75	83.52	N/A [1]	83.83	84.26	82.55	83.68	**86.85**
MobileNetV2	90.77	89.1	87.81	89.46	89.66	90.51	89	**91.02**

Table 6: Validation Accuracy (ImageNet and SGD)

Network	FP32	MAX	PACT	Quant	Fracbits	DQ	Rand	Rand+mix
Resnet18	69.9	68.53	**69.67**	69.55	69.61	69.1	69.2	69.63
Resnet50	75.7	73.31	**75.61**	74.41	74.01	75.1	73.32	75.53
EfficientNet-B0	68.86	67.34	N/A[1]	66.78	67.8	66.0	66.13	**68.3**
MobileNetV2	65.79	60.38	60.86	63.01	62.66	63.12	60.58	**63.66**

CIFAR10 and SGD. As shown in Table 5, *Rand* slightly improves the accuracy over the MAX method on four out of six networks. *Rand* and MAX together do not perform well on Resnet50, Resnet101, Resnet152 and EfficientNet-B0 and significantly degrade the accuracy compared with the FP32 version. However, when combined with mixed precision (*Rand+mix*), the improvement in accuracy is noticeable, especially for EfficientNet-B0 with 3.13% improvement over the MAX method. *Rand+mix* obtains the best results for 5 out of 6 networks and have comparable accuracy to the FP32 version. Especially, it achieves better accuracy than the FP32 version for MobileNetV2 and EfficientNet-B0.

To illustrate the effectiveness of our method during the training process, Figure 4.6 shows the validation accuracy of EfficientNet-B0 for different methods: MAX, PACT, Quant_noise and *Rand+mix*. The validation accuracy is reported at the end of each training epoch. As shown in the figure, *Rand+mix* dominates

[1]PACT does not apply to these networks.

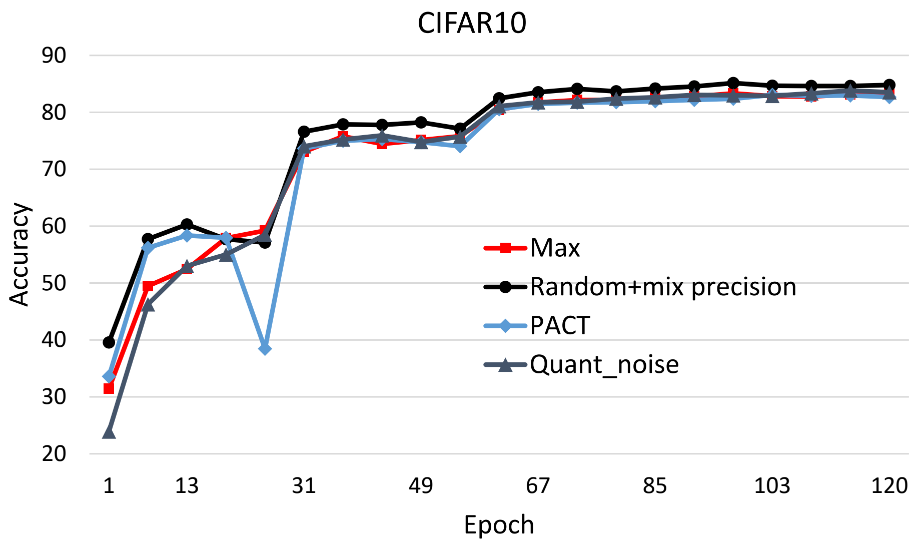


Figure 4.6: 8-bit validation accuracy of EfficientNet-B0 on CIFAR10 with the SGD optimizer.

the other three methods during the whole training time. In the first 30 epochs, *Rand+mix* uses 32-bit precision, and MAX uses 8-bit precision for quantization in the training phase. As a result, the validation accuracy of *Rand+mix* is much better than that of MAX in this period. This justifies the effectiveness of using high precision at the beginning of the training.

ImageNet and SGD. Table 6 shows the results on ImageNet dataset with the SGD optimizer. We select four representative networks. As for the current experiment data, *Rand* improves the accuracy over MAX for three out of four networks except for EfficientNet-B0. This shows the potential of using stochastic quantization scale, though using it alone does not out perform previous work (such as PACT). The combination *rand+mix* achieves the best accuracy for two out of four networks with significant improvement over the second best (around 0.6%). For Resnet18 and Resnet50, it is only slightly worse than the best, which is PACT (up to 0.07%).

Table 7: Validation Accuracy (CIFAR10 and Adam)

Network	FP32	MAX	Rand	Rand+mix
Resnet18	89.6	89.24	**89.57**	89.44
Resnet50	90.2	**89.87**	89.68	87.46
Resnet101	92.3	91.9	**92.05**	87.8
Resnet152	92.5	92.16	**92.95**	87.89
EfficientNet-B0	86.31	86.07	86.2	**86.4**
MobileNetV2	89.1	88.22	87.86	**88.44**

Table 8: Validation Accuracy (ImageNet and Adam)

Network	FP32	MAX	Rand	Rand+mix
Resnet18	64.12	63.86	63.76	**64.24**
Resnet50	70.09	70.46	**70.6**	70.5
EfficientNet-B0	64.95	64.41	64.42	**64.88**
MobileNetV2	57.45	56.76	56.78	**57.14**

CIFAR10 and ADAM. Table 7 shows the results of our methods compared with the FP32 and MAX using the CIFAR10 dataset using Adam optimizer. We use the same hyper parameters as in the experiments with SGD, i.e., the only difference is the optimizer. Except for Resnet50 and MobileNetV2, *Rand* improves the accuracy over the MAX method. However, for the four resnet-based networks, the accuracy of *rand+mix* significantly degrades compared with MAX and FP32. First, when comparing *Rand+mix* and MAX, we observe that the training of *Rand+mix* converges much slower. We believe that this stems from two things. One is ADAM's poor ability of generalization compared to SGD [105]. This phenomenon is more severe for deeper architectures with a small dataset, such as CIFAR10 [105]. The other is the fast progress at the beginning of the training compared with SGD [105]. Thus, the ADAM optimizer learns the training data in high precision so quickly that when the precision is reduced, it encounters difficulties in bridging the gap between two different quantization precisions.

ImageNet and ADAM. Table 8 shows the results of our methods in ImageNet dataset compared with FP32 and MAX using the ADAM optimizer. *Rand* only slightly improves the accuracy of two out of four networks over the MAX baseline. The results are rather similar to that of Imagenet and SGD. *Rand+mix* performs best for two out of four networks.

Table 9: Validation Results for NLP Tasks

Network	FP32	MAX	Quant_noise	Rand	Rand+mix
Transformer	35.53	35.20	35.3	35.23	**35.52**
LSTM	29.7	29.61	N/A [1]	29.86	**29.96**
BERT	85.29	85.23	85.67	**85.83**	85.21
GPT-2	20.1	20.82	N/A [1]	**19.72**	21.81

4.5.2 Natural Language Processing

In natural language processing, we evaluate four networks: Transformer, LSTM, BERT, and GPT-2. The first two networks solve the problem of machine translation. We use the dataset IWSLT14 German-English to evaluate these two networks and report the BLEU score. Their implementations are based on the implementations from Facebook [106]. We evaluate BERT using MPRC task [107], which is one of the tasks in the GLUE benchmark suite[108]. Following the practice, we report the F1 score for this network. We evaluate GPT-2 with the task of language modeling on the dataset wikitext-2 [109]. We report the perplexity (PPL) for GPT-2. Since a widely validated implementation of the pretraining (training from scratch) script is not available at the time of writing this paper, we use the fine-tuning script from Hugging Face [110]. We start the training with quantization using the FP32 pre-trained models. BERT and GPT-2 are the only networks that we use the FP32 pre-trained versions.

[1]Currently, Quant_noise does not apply to these networks.

As shown in Table 9, our method *Rand* improves the accuracy over the MAX method for three out of four networks. The most significant result is GPT-2 with more than 1 PPL point reduced using *Rand*. Overall, the best results can be achieved by using one of the two schemes we present here. Our results are comparable to or better than the original FP32 version.

4.6 Conclusions

In this chapter, we present two techniques for integer quantization in deep learning. One is adding random noise into computing the quantization scale. This added noise makes the deep learning models more robust to unseen data, thus improves their generalization capability. The other adjusts the integer quantization precision in the training phase. The use of multiple precisions makes the models able to learn better from the training data and achieve better accuracy in general. We evaluate our techniques using various models on two different domains: image processing and natural language processing. The results for different tasks show that our techniques are effective and applicable to a wide range of model architectures and input data.

Chapter 5

Conculsion

In this thesis, we presented three main techniques for modeling, autotuning and quantizing GPU programs. To accurately modeling the performance, we reverse-engineered the GPU scheduling policy using micro-benchmarks. We then use a Machine Learning method to build the performance model based on the training data obtained from the performance counters.

We presented an efficient autotuner for selecting a good work-group size for GPU kernels. We also used a set of micro-benchmarks to single out the most critical performance factors on GPUs. Using these insights, we proposed our autotuner.

Lastly, we presented a set of techniques used in integer quantization. Our techniques improve the accuracy compared to previous work using the same precision. We evaluated our techniques using a large number of network architectures and various learning tasks and obtained potential results.

Bibliography

[1] D. Luebke and M. Harris, "General-purpose computation on graphics hardware," in *Workshop, SIGGRAPH*, vol. 33, 2004.

[2] A. R. Brodtkorb, T. R. Hagen, and M. L. Saetra, "Gpu programming strategies and trends in gpu computing," 2012.

[3] S. Hong and H. Kim, "An integrated gpu power and performance model," in *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 280–289, 2010.

[4] W. Jia, K. A. Shaw, and M. Martonosi, "Stargazer: Automated regression-based gpu design space exploration," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pp. 2–13, IEEE, 2012.

[5] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 105–114, 2010.

[6] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in *2011 IEEE 17th international symposium on high performance computer architecture*, pp. 382–393, IEEE, 2011.

[7] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," in *Proceedings of the 36th annual international symposium on Computer architecture*, pp. 152–163, 2009.

[8] W. Liu, W. Muller-Wittig, and B. Schmidt, "Performance predictions for general-purpose computation on gpus," in *2007 International Conference on Parallel Processing (ICPP 2007)*, pp. 50–50, IEEE, 2007.

[9] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. De Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *Proceedings of the 22nd annual international conference on Supercomputing*, pp. 368–377, 2008.

[10] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pp. 318–329, IEEE, 2008.

[11] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee, "An approach to performance prediction for parallel applications," in *European Conference on Parallel Processing*, pp. 196–205, Springer, 2005.

[12] M. Fatica, "Accelerating linpack with cuda on heterogenous clusters," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 46–51, 2009.

[13] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu, "Adaptive optimization for petascale heterogeneous cpu/gpu computing," in *2010 IEEE International Conference on Cluster Computing*, pp. 19–28, IEEE, 2010.

[14] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *arXiv preprint arXiv:1912.01703*, 2019.

[15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

[16] H. Vanholder, "Efficient inference with tensorrt," 2016.

[17] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "Pact: Parameterized clipping activation for quantized neural networks," *arXiv preprint arXiv:1805.06085*, 2018.

[18] S. Uhlich, L. Mauch, F. Cardinaux, K. Yoshiyama, J. A. Garcia, S. Tiedemann, T. Kemp, and A. Nakamura, "Mixed precision dnns: All you need is a good parametrization," *arXiv preprint arXiv:1905.11452*, 2019.

[19] R. Gong, X. Liu, S. Jiang, T. Li, P. Hu, J. Lin, F. Yu, and J. Yan, "Differentiable soft quantization: Bridging full-precision and low-bit neural networks," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 4852–4861, 2019.

[20] A. Mishra, E. Nurvitadhi, J. J. Cook, and D. Marr, “Wrpn: wide reduced-precision networks,” *arXiv preprint arXiv:1709.01134*, 2017.

[21] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, “Integer quantization for deep learning inference: Principles and empirical evaluation,” *arXiv preprint arXiv:2004.09602*, 2020.

[22] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

[23] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, “Language models are few-shot learners,” *arXiv preprint arXiv:2005.14165*, 2020.

[24] Top500.org, “Top500 supercomputer sites - november 2013,” 2013.

[25] NVIDIA, “Nvidia geforce gtx 580.,”

[26] NVIDIA, “Nvidia geforce gtx 680.,”

[27] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, “Demystifying gpu microarchitecture through microbenchmarking,” in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pp. 235–246, IEEE, 2010.

[28] S. Collange, D. Defour, and D. Parello, “Barra, a parallel functional gpgpu simulator,” 2009.

[29] A. Kerr, G. Diamos, and S. Yalamanchili, "Modeling gpu-cpu workloads and systems," in *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, pp. 31–42, 2010.

[30] C.-K. Luk, S. Hong, and H. Kim, "Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 45–55, IEEE, 2009.

[31] D. Grewe and M. F. O'Boyle, "A static task partitioning approach for heterogeneous systems using opencl," in *International conference on compiler construction*, pp. 286–305, Springer, 2011.

[32] NVIDIA, "Nvidia fermi compute architecture white paper.,"

[33] K. O. Group, *The OpenCL Specification Version 1.1. Khronos Group.*

[34] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide.* Pearson Education, 2011.

[35] N. C. Team *et al.*, "Nvidia compute ptx: Parallel thread execution," *ISA version*, vol. 1, 2009.

[36] V. Vapnik, *The nature of statistical learning theory.* Springer science & business media, 2013.

[37] A. J. Smola and B. Schölkopf, "A tutorial on support vector regression," *Statistics and computing*, vol. 14, no. 3, pp. 199–222, 2004.

[38] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[39] NVIDIA, *NVIDIA OpenCL SDK code samples.*

[40] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[41] S. Seo, G. Jo, and J. Lee, "Performance characterization of the nas parallel benchmarks in opencl," in *2011 IEEE international symposium on workload characterization (IISWC)*, pp. 137–148, IEEE, 2011.

[42] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pp. 63–74, 2010.

[43] A. ATI, "Stream software development kit (sdk) v2. 1," 2010.

[44] NVIDIA, "Nvidia cuda profiler.,"

[45] C. Jiang and M. Snir, "Automatic tuning matrix multiplication performance on graphics hardware," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pp. 185–194, IEEE, 2005.

[46] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *International Parallel & Distributed Processing Symposium*, pp. 1–12, IEEE, 2010.

[47] S. Lee and R. Eigenmann, "Openmpc: Extended openmp programming and tuning for gpus," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–11, IEEE Computer Society, 2010.

[48] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded gpu," in *Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 195–204, ACM, 2008.

[49] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," in *ACM Sigplan Notices*, vol. 45, pp. 86–97, ACM, 2010.

[50] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, p. 4, IEEE Press, 2008.

[51] Y. Liu, E. Z. Zhang, and X. Shen, "A cross-input adaptive framework for gpu program optimizations," in *International Parallel & Distributed Processing Symposium*, pp. 1–10, IEEE, 2009.

[52] A. Nukada and S. Matsuoka, "Auto-tuning 3-d fft library for cuda gpus," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, p. 30, ACM, 2009.

[53] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, "Portable performance on heterogeneous architectures," in *ACM SIGPLAN Notices*, vol. 48, pp. 431–444, ACM, 2013.

[54] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pp. 1–11, IEEE, 2008.

[55] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *ACM SIGPLAN Notices*, vol. 50, pp. 379–390, ACM, 2015.

[56] S. Muralidharan, A. Roy, M. Hall, M. Garland, and P. Rai, "Architecture-adaptive code variant tuning," in *ACM SIGPLAN Notices*, vol. 51, pp. 325–338, ACM, 2016.

[57] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 13–24, ACM, 2013.

[58] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, (Edmonton, Canada), August 2014.

[59] F. NVidia, "Nvidia's next generation cuda compute architecture," 2009.

[60] S. Seo, J. Lee, G. Jo, and J. Lee, "Automatic opencl work-group size selection for multicore cpus," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pp. 387–398, IEEE Press, 2013.

[61] T. G. Rogers, D. R. Johnson, M. O'Connor, and S. W. Keckler, "A variable warp size architecture," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 489–501, ACM, 2015.

[62] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture*, pp. 382–393, IEEE, 2011.

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

[64] O. Kayıran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: optimizing thread-level parallelism for gpgpus," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pp. 157–166, IEEE Press, 2013.

[65] NVIDIA, "Cuda programming guide," 2012.

[66] NVIDIA, "Occupancy calculator," 2012.

[67] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "Gpumech: Gpu performance modeling technique based on interval analysis," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 268–279, IEEE, 2014.

[68] NVIDIA, *NVIDIA Compute Visual Profiler User Guide*, May 2011.

[69] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 374–385, ACM, 2011.

[70] T. G. Rogers, M. O'Connor, and T. M. Aamodt, "Cache-conscious wavefront scheduling," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 72–83, IEEE Computer Society, 2012.

[71] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, *et al.*, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 451–460, ACM, 2010.

[72] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 73–82, ACM, 2008.

[73] S. Seo, G. Jo, and J. Lee, "Performance characterization of the nas parallel benchmarks in opencl," in *Proceedings of the 2011 IEEE Symposium on Workload Characterization*, pp. 137–148, IEEE, 2011.

[74] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[75] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-*

Purpose Computation on Graphics Processing Units, pp. 63–74, ACM, 2010.

[76] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE Symposium on Workload Characterization*, pp. 44–54, IEEE, 2009.

[77] NVIDIA, "Opencl programming guide," 2013.

[78] S. Migacz, "8-bit inference with tensorrt," in *GPU technology conference*, vol. 2, p. 5, 2017.

[79] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *arXiv preprint arXiv:1606.06160*, 2016.

[80] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[81] C. M. Bishop, "Training with noise is equivalent to tikhonov regularization," *Neural computation*, vol. 7, no. 1, pp. 108–116, 1995.

[82] G. An, "The effects of adding noise during backpropagation training on a generalization performance," *Neural computation*, vol. 8, no. 3, pp. 643–674, 1996.

[83] J. R. Hershey and P. A. Olsen, "Approximating the kullback leibler divergence between gaussian mixture models," in *2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07*, vol. 4, pp. IV–317, IEEE, 2007.

[84] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in neural information processing systems*, pp. 3123–3131, 2015.

[85] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," in *Advances in neural information processing systems*, pp. 5145–5153, 2018.

[86] D. Das, N. Mellempudi, D. Mudigere, D. Kalamkar, S. Avancha, K. Banerjee, S. Sridharan, K. Vaidyanathan, B. Kaul, E. Georganas, *et al.*, "Mixed precision training of convolutional neural networks using integer operations," *arXiv preprint arXiv:1802.00930*, 2018.

[87] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, pp. 1737–1746, 2015.

[88] A. Fan, P. Stock, , B. Graham, E. Grave, R. Gribonval, H. Jegou, and A. Joulin, "Training with quantization noise for extreme model compression," 2020.

[89] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv preprint arXiv:1806.08342*, 2018.

[90] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," *arXiv preprint arXiv:1802.05668*, 2018.

[91] G. Prato, E. Charlaix, and M. Rezagholizadeh, "Fully quantized transformer for machine translation," *arXiv*, pp. arXiv–1910, 2019.

[92] A. Tierno, "Quantized transformer," tech. rep., tech. rep., Stanford University, Stanford, California, 2019.

[93] L. Yang and Q. Jin, "Fracbits: Mixed precision quantization via fractional bit-widths," *arXiv preprint arXiv:2007.02017*, vol. 1, 2020.

[94] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv preprint arXiv:1308.3432*, 2013.

[95] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[96] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.

[97] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017.

[98] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, pp. 5998–6008, 2017.

[99] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.

[100] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[101] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[102] H. Robbins and S. Monro, “A stochastic approximation method,” *The annals of mathematical statistics*, pp. 400–407, 1951.

[103] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[104] B. Zhuang, C. Shen, M. Tan, L. Liu, and I. Reid, “Towards effective low-bitwidth convolutional neural networks,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7920–7928, 2018.

[105] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, “The marginal value of adaptive gradient methods in machine learning,” *arXiv preprint arXiv:1705.08292*, 2017.

[106] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, “fairseq: A fast, extensible toolkit for sequence modeling,” in *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[107] W. B. Dolan and C. Brockett, “Automatically constructing a corpus of sentential paraphrases,” in *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*, 2005.

[108] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, “Glue: A multi-task benchmark and analysis platform for natural language understanding,” *arXiv preprint arXiv:1804.07461*, 2018.

[109] S. Merity, C. Xiong, J. Bradbury, and R. Socher, “Pointer sentinel mixture models,” *arXiv preprint arXiv:1609.07843*, 2016.

[110] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen,

C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, “Transformers: State-of-the-art natural language processing,” in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020.

Acknowledgements

I would like to express my sincerest thanks to my advisor, Professor Jaejin Lee, for his generous support during my PhD study. His patience, enthusiasm and great understanding in the field have motivated me to perform the research righteously. There were times when I had doubts, but he helped me clear them away by showing me passion and trust, like a father. I am fortunate to have Professor Jaejin Lee as my advisor and I could never thank him enough.

Besides my advisor, I would like to deeply thank the other members of my thesis committee: Prof. Jin-soo Kim, Prof. Soo-Mook Moon, Prof. ChangHee Jung, Prof. Hyungmin Cho. Their insightful comments and questions were helpful to improve my thesis.

I had an opportunity to work with Professor Bernhard Egger in my early stage of my PhD study. His wise advice and encouragement were motivational for me to settle the first step in this thesis.

I wholeheartedly thank my colleagues at Thunder Research Group. From the bottom of my heart, I will never forget that they have taken care of me as a member of the family since my first day in Korea. I would not have made this far without their enormous help. Additionally, special thanks to Wookeun Jung for having in-depth discussion and valuable comments about my thesis.

I would like to thank my family for always being there for me. Most of all, I want to thank my son, who has given me the strength to overcome any difficulties since the day he was born. Thank you for being a great, strong and always a strong boy. I thank my wife for her understanding and continuous support for the family throughout the hardest time. I thank my sister for visiting me in Korea. Lastly, I would not be able to carry on without unconditional love from my parents. There would be no possible way to tell how much they have sacrificed for their children and grandchildren.