ACCELERATING MACHINE LEARNING VIA MULTI-OBJECTIVE OPTIMIZATION

by

ROBERT LIM

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Division of Graduate Studies of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

September 2021

DISSERTATION APPROVAL PAGE

Student: Robert Lim

Title: Accelerating Machine Learning via Multi-Objective Optimization

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Allen Malony	Chair
Boyana Norris	Core Member
Dejing Dou	Core Member
Camille Coti	Core Member
William Cresko	Institutional Representative

and

Andy Karduna	Interim Vice Provost for Graduate Studies

Original approval signatures are on file with the University of Oregon Division of Graduate Studies.

Degree awarded September 2021

DISSERTATION ABSTRACT

Robert Lim

Doctor of Philosophy

Department of Computer and Information Science

September 2021

Title: Accelerating Machine Learning via Multi-Objective Optimization

This dissertation work presents various approaches toward accelerating training of deep neural networks with the use of high-performance computing resources, while balancing learning and systems utilization objectives. Acceleration of machine learning is formulated as a multi-objective optimization problem that seeks to satisfy multiple objectives, based on its respective constraints. In machine learning, the objective is to strive for a model that has high accuracy, while eliminating false positives and generalizing beyond the training set. For systems execution performance, maximizing utilization of the underlying hardware resources within compute and power budgets are constraints that bound the problem. In both scenarios, the search space is combinatorial and contains multiple local minima that in many cases satisfies the global optimum. This dissertation work addresses the search of solutions in both performance tuning and neural network training. Specifically, subgraph matching is proposed to bound the search problem and provide heuristics that guide the solver toward the optimal solution. Mixed precision operations is also proposed for solving systems of linear equations and for training neural networks for image classification for evaluating the stability and robustness of the operations. Use cases are presented with CUDA performance tuning and neural network training, demonstrating the effectiveness

of the proposed technique. The experiments were carried out on single and multi-node GPU clusters, and reveals opportunities for further exploration in this critical hardware/software co-design space of accelerated machine learning.

CURRICULUM VITAE

NAME OF AUTHOR: Robert Lim

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
University of California, Irvine, Irvine, CA, USA
University of California, Los Angeles, Los Angeles, CA, USA

DEGREES AWARDED:

Doctor of Philosophy, Computer and Information Science, 2021, University of Oregon
Master of Science, Computer Science, 2014, University of California, Irvine
Bachelor of Science, Cognitve Science, Specialization in Computing, 2005, University of California, Los Angeles

AREAS OF SPECIAL INTEREST:

Numerical Analysis
Automatic Performance Tuning for GPUs
Multi-Objective Optimization
High Performance Computing

PROFESSIONAL EXPERIENCE:

Intern, U.S. Army Engineer Research and Development Center, Geospatial Research Lab, Alexandria, VA	Sept – Dec, 2019
Intern, Université de Versailles, Versailles, France	Mar – Sept, 2019
Intern, Defence Science & Technology Lab, Salisbury, U.K.	Aug – Sept, 2017
Intern, The Alan Turing Institute, London, U.K.	June – Aug, 2017
Intern, U.S. Army Engineer Research and Development Center, Geospatial Research Lab, Alexandria, VA	June – Sept, 2016
ASTRO Intern, Oak Ridge National Lab, Oak Ridge, TN	June – Sept, 2014
Extreme Blue Technical Intern, IBM, Austin, TX	May – Aug, 2013

GRANTS, AWARDS AND HONORS:

Awards

Graduate Research Fellowship, University of Oregon	2014
SMART Scholarship Award, U.S. Department of Defense	2015
Chateaubriand Fellowship, Embassy of France in U.S.	2017
Gurdeep Pall Fellowship, University of Oregon	2016, 2017

Student Travel Award, Ph.D. Forum

Supercomputing Conference, Denver, CO	Nov 2019
International Conference on Parallel Processing, Eugene, OR	Aug 2018
IEEE Cluster, Chicago, IL	Aug 2015
ACM Object-Oriented Programming, Systems, Languages & Applications, Portland, OR	Sept 2014

PUBLICATIONS:

Lim, R., Oliveira, P., Coti, C., Jalby, W., and Malony, A. "Reduced Precision Computation for Accurate and Robust Learning Systems." *5th Workshop on Naval Applications of Machine Learning*, 2021 (poster)

Lim, R., Norris, B., and Malony, A. "A Similarity Measure for GPU Kernel Subgraph Matching." *31st International Workshop on Languages and Compilers for Parallel Computing*, 2019

Lim, R., Heafield, K., Hoang, H., Briers, M., and Malony, A. "Exploring Hyper-Parameter Optimization for Neural Machine Translation on GPU Architectures." *2nd Workshop on Naval Applications of Machine Learning*, 2018

Lim, R., Norris, B., and Malony, A. "Autotuning GPU Kernels via Static and Predictive Analysis." *46th International Conference on Parallel Processing*, 2017

Lim, R., Norris, B., and Malony, A. "Tuning Heterogeneous Computing Architectures through Integrated Performance Tools." *GPU Technology Conference*, 2016 (poster)

Lim, R., Malony, A., Norris, B., and Chaimov, N. "Identifying Optimization Opportunities within Kernel Execution in GPU Codes." *International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*, 2015

Sreepathi, S., Grodowitz, M., Lim, R., Taffet, P., Roth, P., Meredith, J., Lee, S., Li, D., and Vetter, J. "Application Characterization using Oxbow Toolkit and Pads Infrastructure." *International Workshop on Hardware-Software Co-Design for High Performance Computing*, 2014

Lim, R., Carrillo-Cisneros, D., Alkowaileet, W., and Scherson, I. "Computationally Efficient Multiplexing of Events on Hardware Counters." *Linux Symposium*, 2014

Alkowaileet, W., Carrillo-Cisneros, D., Lim, R., and Scherson, I. "NUMA-aware Multicore Matrix Multiplication." *Parallel Processing Letters*, 2013

ACKNOWLEDGEMENTS

I want to send my sincerest gratitude to the following individuals who have facilitated in the embarkment of this endeavour.

Dissertation committee. Profs. Allen Malony, Boyana Norris, Dejing Dou, Camille Coti, Bill Cresko.

Collaborators. Oak Ridge National Lab: Drs. Jeff Vetter, Megan Grodowitz, Sarat Sreepathi. The Alan Turing Institue: Drs. Kenneth Heafield, Hieu Hoang, Mark Briers. University of Versailles: Drs. William Jalby, Pablo Oliveira.

Compute resources. Argonne National Lab: Drs. Kevin Harms, Phil Carns. NVIDIA: J-C Vasnier, Duncan Poole, Barton Fiske. University of Reims, Champagne Ardenne: Drs. Michael Krajecki, Arnaud Renaud.

Colleagues. TAU Developers: Sameer Shende, Kevin Huck, Wyatt Spear, Nicholas Chaimov, Aurele Maheo, Josefina Lenis, Alister Johnson, Srinivasan Ramesh. Office mates: Chad Wood, Daniel Ellsworth, David Ozog. SMART: Arnold Boedihardjo, Alan Van Nevel, Marisa Garcia, Jessica Holland, Jess Molina, Brandon Cochenour, Karrin Felton.

University of Oregon. CIS Department: Drs. Joe Sventek, Hank Childs, Reza Rejaie. CIS Staff: Cheri Smith, Jan Saunders, Rob Yelle, Charlotte Wise. Neuroinformatics Center: Dr. Don Tucker, Erik Keever. Division of Graduate Studies: Jered Nagel, Lesley Yates-Pollard, Andy Karduna.

Family members. Mom, Dad, Susa, Melody, Tabitha, Lucas, Chad.

I want to acknowledge my advisor, Prof. Allen Malony, who has undoubtedly created an environment for me to thrive in. That car ride to Falling Sky during my recruitment and the conversation we had about GPUs and harsh

journal reviewers was beyond convincing for me to attend the University of Oregon, not to mention the Eugene mist and Ducks Football. I feel very fortunate for the unconditional advisement I received, having a channel to brainstorm ideas, and plowing through the countless deadlines. I also want to acknowledge my dissertation committee, especially Profs. Boyana Norris and Camille Coti, for their unequivocal support throughout various phases in this process.

Many thanks!

To my dad, whose work ethic has inspired me in many ways.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

Figure Page

LIST OF TABLES

Table Page

Table Page

CHAPTER I

INTRODUCTION

Overview

This dissertation work presents various approaches toward accelerating training of deep neural networks with the use of high-performance computing (HPC) resources, while balancing learning and systems utilization objectives. Acceleration of machine learning (ML) is formulated as a multi-objective optimization problem, which seeks to jointly optimize performance and learning objectives, based on its respective constraints. Within each scope, the solver seeks an optimal solution. In machine learning, the solver optimizes a prediction function $h : \mathcal{X} \rightarrow \mathcal{Y}$ from an input space $\mathcal{X}$ to an output space $\mathcal{Y}$, for $x \in \mathcal{X}$, such that $\hat{y} = F_h(x)$ accurately predicts the output and minimizes its empirical risk, defined as the expectation, $R(F_h) = \mathbb{E}[L(F(h(x)), y)] = \int L(h(x), y) \ dP(x, y)$, for some loss function $L(\hat{y}, y)$ estimating $\hat{y}$ from y, and the goal is to find $F_h^* = \arg\min_{F_h \in \mathbb{F}} R(F_h)$, or a prediction function $F_h^* \in \mathbb{F}$ where $R(F_h)$ is minimal. When optimizing for execution performance, the solver seeks to minimize a cost function $g : \mathcal{X} \rightarrow \mathcal{Y}$ by selecting the combination of compute resources $a \in \mathcal{A}$ and transformation options $o \in \mathcal{O}$, $\mathcal{X}(\mathcal{A}, \mathcal{O})$, which yield minimal execution time while maximizing utilization of hardware resources, or $F_g^* = \arg\min_{F_g \in \mathbb{F}} R(F_g)$, where $R(F_g) = \mathbb{E}[F(g(x)), y)] = \int L(g(x), y) \ dP(x, y)$. Joint optimization of machine learning and high performance computing, discussed in Sec. II, is defined as $\min_{F_g \in \mathbb{F}} \min_{F_h \in \mathbb{F}} \mathcal{R}(\mathbb{F})$, for $F_g, F_h \in \mathbb{F}$, where $\mathbb{F}$ represents a vector of objective functions satisfying constraints for its respective domains.

Organization of Dissertation

This section outlines the dissertation format and provides an overview for each chapter.

Background Information. Accelerating machine learning and multi-objective optimization are presented in Chapter II, which motivates the discussion and provides the background information for the dissertation. The subject matter is on the intersection of high-performance computing and machine learning, specifically how the innovations of heterogeneous parallel programming and methods for analyzing massive amounts of data has transformed industries and society, making this an important field to investigate. Performance optimization depends on numerous elements involved in the computation, including both hardware and software. Thus, the computer architectures, the algorithms, and numerical methods related to machine learning are briefly covered in this chapter.

In the chapters that follow, several approaches are presented for optimizing performance and learning by accelerating the computation kernels used by the machine learning algorithms, by tuning the hyper-parameters of these algorithms, and by understanding the numerical representation of the data handled by these algorithms. The next subsections provide a brief overview and the research questions raised for each chapter.

Optimizing Code Generation. Chapter III discusses the work where we proposed metrics for automatic performance tuning of GPU applications. This work seeks to address the following questions:

1. Given the difficult requirements by the user in writing CUDA code, where the user is forced to set threads, blocks, shared memory, and writing efficient

parallel programs, could we automatically come up with ways that discover optimal parameter settings?

2. What metrics can we define statically, such as occupancy, that capture the performance requirements of a computational kernel, and can we use those to help improve our search during automatic performance tuning?

GPU Subgraph Matching. Chapter IV proposes several techniques for performing subgraph matching with GPU kernels. The proposed techniques incorporate the shape and traversal of the graph, its transition probabilities, and hardware information such as the GPU the graph was generated in and instruction mixes. The following questions are proposed:

1. Can we come up with compact ways of representing execution performance information of GPU kernels that captures the essence of runtime information, but at the same time, enable us to reason about an unseen kernel's behavior?

2. Can we define a similarity metric that enables us to match graphs with one another? This similarity metric needs to be GPU architecture independent, provide a correct measure when measuring with itself, and can match graphs of arbitrary shapes and sizes.

Hyper-Parameter Optimization. Chapter V optimizes hyper-parameters for a neural machine translation system. The hyper-parameters explored varies and the study accounts for training stability, trajectories and speed. Other statistics include words processed per second and time to convergence. In this work, we address the following questions:

1. Can we identify which hyper-parameters contribute the most to a model's learning trajectory, while accounting for stability, quality, and speed?

2. Can we identify which hyper-parameters matter most, in terms of system execution performance?

Numerical Representation. Chapter VI examines the numerical requirements when training deep neural networks. In particular, mixed precision operations is proposed that enables users to set the precision and range sizes during training run. The questions raised in this research are the following:

1. What are the precision requirements during various iterations of the phase when training deep neural networks?

2. Can we propose a dynamic mixed-precision approach toward training neural networks, where the precision sizes can be set during the phase of the training run?

Summary and Future Work. Chapter VII closes by summarizing the work that was discussed in previous chapters and presents areas to pursue for future work.

Conclusion

An overview of multi-objective optimization and the dissertation format was presented. Next, motivation and background information relating to accelerated machine learning are presented.

CHAPTER II

BACKGROUND

This chapter provides motivation for the dissertation work and covers the basic concepts needed to be discussed further in the dissertation.

Motivation

Deep Learning Application Domains. The U.S. Department of Energy has outlined the artifical intelligence (AI) objectives to couple machine learning methods with HPC workloads in its concurrent quest for building the first Exaflop supercomputing machine Stevens et al. (2020). The report identifies areas where ML could augment existing scientific workflows, including chemistry, materials and nanoscience; earth and environmental sciences; biology and life sciences; high energy physics and nuclear physics, amongst others. The nascent social media industry, which provides free services to millions of users, has made billions of dollars deploying deep neural networks to analyze petaflops of user data on-line and off-line for its own use of recommendation systems, computer vision, and language comprehension Park et al. (2018). What makes this innovation possible is both the advancement in AI and the compute infrastructure that delivers instantaneous results to users on their end devices.

ML HPC Architectures. This subsection discusses various computer architectures and accelerators that have been developed for machine learning purposes. Because the majority of operations during machine learning are matrix vector products, much effort has been made to fabricate parallel matrix multipliers to accelerate machine learning. Also referred to as neural processors Neural Processor (2020), these architectures include GPUs, CPUs, and custom ASICs, such as tensor processing units (TPU). Not discussed in this dissertation but also part of

	CPU		GPU		TPU	
	Skylake	Cas Lake	Volta	Ampere	v2	v3
Processor						
Cores	28	56	80	96	512	2048
Freq (MHz)	2500	2600	1328	1328	700	940
Peak Perf	2T	3.2T	125T	312T	180T	420T
Memory						
Type	DDR4	DDR4	HBM2	HBM2	HBM	HBM
Off-chip (GB)	120	140	16	40	32	40
BW (GB/s)	16.6	23.4	900	1555	2400	3600
Hardware						
TDP	205	400	300	450	280	450
MXU	n/a	n/a	4×4 (8)	4×4 (4)	128×128 (1)	128×128 (2)

Table 1. Comparing ML computer processors and accelerators.

neural processors are neuromorphic chips, such as IBM True North Modha (2017) and Intel Loihi Davies et al. (2018), and field programmable gate arrays (FPGA). Table 1, drawn from Wang, Wei, and Brooks (2019); Intel Sky Lake (2017); Intel Cascade Lake (2019), displays a general comparison of CPUs, GPUs and TPUs.

Graphic Processing Units. NVIDIA GPUs, which dominate the market share of accelerators for machine learning due to the proven use of GPUs and the CUDA Deep Neural Network (cuDNN) library, has been aggressively introducing new hardware capabilities for fused operations and reduced precision. Designed originally for real-time graphics rendering using fixed-function pipelines with each pixel performing independent operations in parallel, CUDA debuted as a parallel computing platform in 2007 that enabled user defined programs called shaders that combined the vertex and fragment operations, which is the foundation of the data parallel programming units that run on GPUs and currently

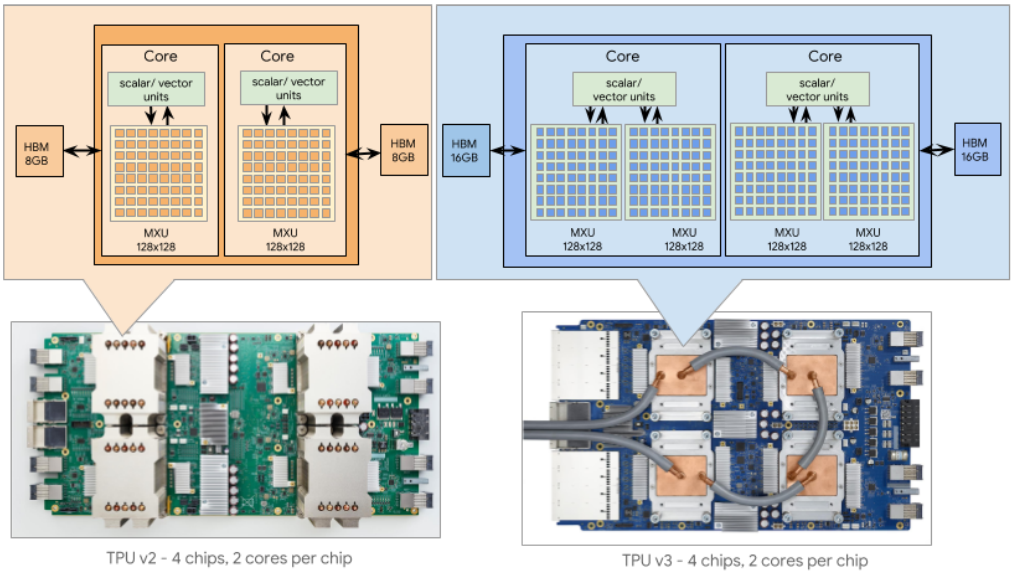

Figure 1. Tensor processing unit (image source Cloud TPU (2019)).

powers some of the world's fastest supercomputers. NVIDIA Volta V100 marks the release of hardware support for tensor cores in 2017 that are capable of executing 4×4 matrices in 16-bit, which uses warps of 32 parallel threads. The NVIDIA Ampere A100, released in 2020, adds the `tensorfloat32` instruction set TF32 (2019a), which utilizes 8 bits for the exponent and 10 bits for the mantissa, and is essentially a 19 bit format in a 32 bit register. Software frameworks such as PyTorch, TensorFlow and MxNet rely on the cuDNN library for acceleration. The library can be downloaded for free with a registered NVIDIA account.

Tensor Processing Unit. The tensor processing unit (TPU), created by Google in 2016, is a high-performance, application-specific integrated chip (ASIC) designed for neural network training with 64 GB high-bandwidth memory and 180 teraflops (TFLOPS) peak performance Cloud TPU (2019). Figure 1 displays a schematic representation of the TPUs for v1 and v2. Organized as

a systolic array of 65536 8-bit matrix multiplier units (MXU) with hardwired activation units and a 24 MB unified buffer, inputs are read once and reused, in contrast to the current approach that loads and stores each value TF32 (2019b). Values are further quantized to 8-bits to increase throughput of the instructions. The TPU performs well for training deep neural networks with large batch sizes and is capable of low latency inferencing.

Scalable CPUs. Intel, known traditionally as a CPU manufacturer, has also been agressively wedging in the market share of high-performance computing and machine learning. The CPU's ability for multi-processing with mixed-mode accelerators can be advantageous for AI models, such as reinforcement learning and agent-based modeling. Intel also open sourced the Math Kernel Library Deep Neural Network (MKL-DNN) in 2019 as part of the oneAPI initiative, a unified programming model that targets all Intel devices, including CPUs, GPUs, and FPGAs.

The Intel Xeon Scalable processors enable multi-node multi-socket connection, with 8-bit multiplies and 32-bit accumulates with instructions, such as 8 bit operations accumulated in 16-bit registers (`VPMADDUBSW`), 16-bit to 32-bit broadcasts (`VPMADDWD`), and neural network instructions (`AVX512_VNNI`), which are 512-bit vectorized intrinsics that perform 8-bit integer operations and accumulates the results to 32-bit registers. Other product lines that Intel provide for AI include Xe GPU accelerators, Arrix FPGAs, and high-performance memory subsystems such as X-point.

Mixed Precision Numerical Methods. Mixed precision algorithms have been successful in providing performance execution benefits, such as increased instruction throughput, less memory footprint and savings in energy. Figure 2

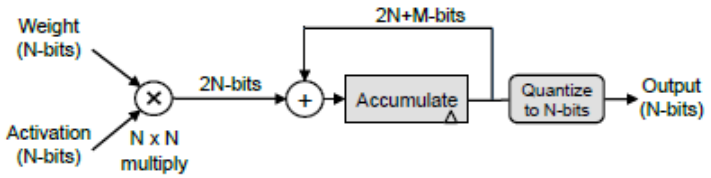

Figure 2. Quantizing multiply-add-accumulate operation.

illustrates how the precision of the output of the activation is reduced to N bits (image source: Sze, Chen, Yang, and Emer (2017)). Reduced precision is commonplace on modern microprocessors and accelerators, such as TPUs that have 8-bit integer arithmetic and GPUs that have mixed precision execution modes, including the Pascal (8-, 16-bit to 32-bit) and the Turing (1-, 4-, 8-, 16-bit to 32-bit) models.

AI and HPC. The advancement of AI has been driven by the algorithmic innovations, massive training sets, and compute capabilities AI and Compute (2018). Figure 3 shows the total amount of compute, in petaflop/s-days, used to train selected AI models. Although one may criticize that the shortcomings of current AI aproaches are exposed with the massive compute requirements, the amount of compute availability often facilitates in algorithmic advances, as seen in the evolving AI approaches, from neural networks to reinforcement learning, and the types of problems being solved, including image classification and competitive gaming.

Background Information

This section covers the prerequisite information needed for discussing the dissertation topics. The topics include multi-objective optimization, machine learning terminology, optimization in machine learning, and stochastic gradient descent.

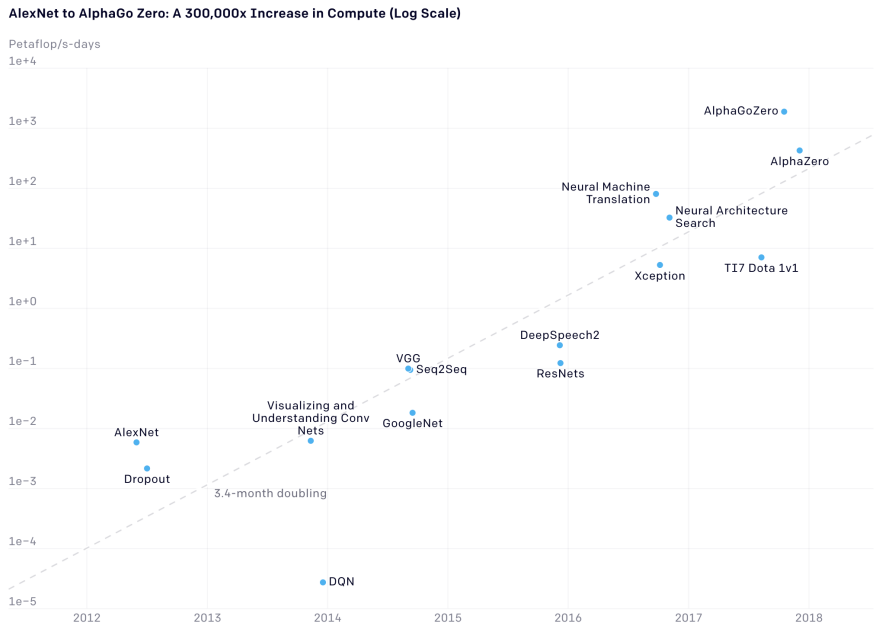

Figure 3. AI training runs, showing 3-4 month doubling time of petaflops. Image source AI and Compute (2018).

Multi-Objective Optimization. Multi-objective optimization seeks to optimize multiple criteria, leveraging overlapping objectives while balancing trade-offs associated with such choices. For accelerating machine learning, multi-objective optimization seeks to achieve quality model learning while leveraging high-performance computing resources.

Multi-objective optimization is formulated as follows. Taken from Miettinen (2012), let k represent the number of objective functions with m inequality constraints and e equality constraints. Let $\mathbf{x} \in E^n$ be a vector of decision variables with n independent variables, x_i, and $\mathbf{F}(x) \in E^k$ be a vector of objective functions, $F_i(\mathbf{x}) : E^n \rightarrow E^1$, where E^1 represents the criteria, or the cost, of the objective.

Multi-objective optimization is defined as follows:

$$\begin{aligned} \min_{\mathbf{x}} \mathbf{F}(\mathbf{x}) &= [F_1(\mathbf{x}), F_2(\mathbf{x}), ..., F_k(\mathbf{x})]^T \\ &\text{subject to} \\ g_j(\mathbf{x}) &\leq 0, j = 1, 2, ..., m, \\ h_l(\mathbf{x}) &= 0, l = 1, 2..., e. \end{aligned} \tag{2.1}$$

The gradient of $F_i(\mathbf{x})$ is $\nabla_{\mathbf{x}} F_i(\mathbf{x}) \in E^n$, where x_i^* is a point that minimizes the objective function, $F_i(\mathbf{x})$. The feasible *design space* $\mathbf{X}$, or the decision space, is defined as the set $\{\mathbf{x} \mid g_j(\mathbf{x}) \leq 0, j = 1, 2, ..., m; h_i(\mathbf{x}) = 0, i = 1, 2, ..., e\}$, whereas the feasible *criterion space* $\mathbf{Z}$, or the attainable set, is defined as $\{\mathbf{F}(\mathbf{x}) \mid \mathbf{x} \in \mathbf{X}\}$. *Feasiblilty* implies that no constraints are involved, whereas *attainability* implies that a point in the criterion space $\mathbf{Z}$ maps to the decision space $\mathbf{X}$. The Pareto optimal is defined as a point $\mathbf{x}^* \in \mathbf{X}$, if and only if there does not exist another point, such that $\mathbf{F}(\mathbf{x}) \leq \mathbf{F}(\mathbf{x}^*)$, and $F_i(\mathbf{x}) < F_i(\mathbf{x}^*)$ for at least one criterion.

ML Terminology. Machine learning is a subcategory of artificial intelligence, and neural networks are a subcategory of machine learning that takes a brain-inspired approach toward learning.

Optimization in Machine Learning. *Optimization* is a mathematical procedure for finding a maximum or a minimum value of a function of several variables, subject to a set of constraints, as in linear programming or systems analysis Chong and Zak (2013). Decision making, where optimization plays a central role, entails selecting the best option amongst a set of alternatives. An objective function, or performance index, provides a "goodness" measure, where the optimization procedure selects the best alternative in light of the given objective function.

Table 2. Selected optimization methods targeting machine learning and high performance computing.

Algorithmic	Solver	Search	Compilation	Node	Distributed
SVM	**SGD**	Random	Loop Transforms	**GPU Thread Block**	**Multi-GPU**
Neural Nets	**Adam**	**Grid**	**Vector-/SIMD**	Memory Hierarchy	**Cluster Parallelism**
Least-squares	**Adagrad**	Bayesian	**IR**	Intrinsics	Block Partition
KNN	FFT	MCTS	**Mixed Precision**	Operator Fusion	Lazy/Eager
Reinforcement Learning	Newton	**Heuristic**	Kernel Fusion	**Rounding**	BFGS, Downpour

Table 2 lists the objectives to optimize for accelerating machine learning. The methods are categorized according to the level of the software stack. Note that this is not an exhaustive list, but a subset that incorporates both ML and HPC. Note, also, that there may be overlaps and that each optimization target may fall under several categories. This dissertation attempts to address the areas that are highlighted.

Algorithmic optimization involves selecting the ML classifier for the task at hand, whether supervised or unsupervised, and its complexity, such as the number of learned parameters. Examples of machine learning algorithms include support vector machines (SVM), neural networks, least-squares methods, K-nearest neighbors (KNN), and reinforcement learning. The solver is the iterative method that provides a performance index during the learning process, and includes stochastic gradient descent (SGD) and its variants, such as AdaGrad and Adam, and second-order methods such as Newton's method, and transformative approaches such as Fast Fourier transform (FFT). Search optimization is concerned with the identification of the maximum or minimum of values. Techniques to

perform search include random, grid, Bayesian, Monte Carlo tree search, and heuristics-based search.

Once the weights have been trained for the model, compilation attempts to optimize the code for execution performance. Code transformations that exploit data locality include loop transformations, vectorization and SIMD approaches, source-to-source translation, reduced precision, and kernel fusion. Single-node optimization targets features available at the computer architecture level, and includes memory hierarchy, intrinsics, and stochastic rounding. Distributed optimization makes use of multiple clusters and multiple accelerators for machine learning, accounting for compute availability, scheduling, checkpointing and problem partitioning. Collectively, this illustrates the complexity and tradeoffs of the landscape when accounting for all factors in optimizing the multiple objectives in machine learning and high-performance computing.

Stochastic Gradient Descent. Stochastic gradient descent is an iterative method that minimizes an objective function F by estimating parameter w for a $F_i(w)$, for the i-th observation Stochastic Gradient Descent (2021). For training neural networks, the weights are learned with each batch of data and updated iteratively. Refer to Appendix A for the derivation of stochastic gradient descent.

Algorithm 1 lists the stochastic gradient method, which performs the following steps. A random variable $\mathcal{E}_k$ is generated via a Taylor expansion series, with $\{\mathcal{E}_k\}$ representing a sequence of jointly independent random variables. Given an iterate $w_k \in \mathbb{R}^d$ and the realization of $\mathcal{E}_k$, a stochastic vector $g(w_k, \mathcal{E}_k) \in \mathbb{R}^d$ is computed. Then, given an iteration number $k \in \mathbb{N}$, a scalar stepsize $\alpha_k > 0$ is

Algorithm 1 Stochastic gradient method.

1: Choose an initial iterate w_1
2: **for** $k = 1, 2, ...$ **do**
3: Generate a realization of the random variable $\mathcal{E}_k$
4: Compute a stochastic vector $g(w_k, \mathcal{E}_k)$
5: Choose a step size $\alpha_k > 0$
6: Set the new iterate as $w_{k+1} \leftarrow w_k = \alpha_k g(w_k, \mathcal{E}_k)$

computed. The stochastic gradient estimate for g with S samples is defined as

$$\nabla f_{S_k}(w_k; \mathcal{E}_k) = \frac{1}{|S_k|} \sum_{i \in S_k} \nabla f(w_k; \mathcal{E}_{k,i}). \tag{2.2}$$

Conclusion

This section covered the background information needed to be discussed for the dissertation. Next, we discuss core areas of the dissertation. The core areas of the dissertation include *optimizing code generation*, *control flow subgraph matching*, *optimizing hyper-parameters*, and *numerical representation*.

CHAPTER III

OPTIMIZING CODE GENERATION

This chapter includes previously published co-authored material that was published at the 46th International Conference on Parallel Processing Lim, Norris, and Malony (2017). I was the primary contributor to this work in developing the algorithm, writing the new code, and writing the paper. Dr. Boyana Norris initially identified the need for this work and provided the application that this work was performed in. Dr. Allen Malony assisted in editing the paper.

Abstract

Optimizing the performance of GPU kernels is challenging for both human programmers and code generators. For example, CUDA programmers must set thread and block parameters for a kernel, but might not have the intuition or experience to make a good choice. Similarly, compilers can generate working code, but may miss tuning opportunities by not targeting GPU models or performing code transformations. Although empirical autotuning addresses some of these challenges, it requires extensive experimentation and search for optimal code variants. This research presents an approach for tuning CUDA kernels based on static analysis that considers fine-grained code structure and the specific GPU architectural features. Notably, unlike most autotuning systems, our approach does not require any program runs in order to discover near-optimal parameter settings. We demonstrate the applicability of our approach in enabling code autotuners such as Orio to produce competitive code variants comparable with empirical-based methods, without the high cost of experiments.

Motivation

Heterogeneous computing poses several challenges to the application developer. Identifying which parts of an application are parallelizable on a SIMD accelerator and writing efficient data parallel code are the most difficult tasks. For instance, CUDA programmers must set block and thread sizes for application kernels, but might not have the intuition to make a good choice. With NVIDIA GPUs, each streaming multiprocessor (SM) has a finite number of registers, limited shared memory, a maximum number of allowed active blocks, and a maximum number of allowed active threads. Variation in block and thread sizes results in different utilization of these hardware resources. A small block size may not provide enough warps for the scheduler for full GPU utilization, whereas a large block size may lead to more threads competing for registers and shared memory.

Writing kernel functions require setting block and thread parameters, and the difficulty is in deciding which settings will yield the best performance. One procedure entails testing the kernel with block sizes suggested by the CUDA Occupancy Calculator (OCC) *CUDA Occupancy Calculator* (2016). Although the OCC takes into account the compute capability (NVIDIA virtual architecture) when calculating block sizes and thread counts, inaccuracies may arise because variations in runtime behavior may not be considered, which can potentially result in suboptimal suggested hardware parameters.

How do variations in runtime behavior arise? Accelerator architectures specialize in executing SIMD in lock-step. When branches occur, threads that do not satisfy branch conditions are masked out. If the kernel programmer is unaware of the code structure or the hardware underneath, it will be difficult for them to make an effective decision about thread and block parameters.

CUDA developers face two main challenges, which we aim to alleviate with the approach described in this paper. First, developers must correctly select runtime parameters as discussed above. A developer or user may not have the expertise to decide on parameter settings that will deliver high performance. In this case, one can seek guidance from an optimization advisor. The advisor could consult a performance model based on static analysis of the kernel properties, or possibly use dynamic analysis to investigate alternative configurations. A second concern is whether the kernel implementation is not optimized yet. In this case, advice on parameter settings could still be insufficient because what is really required is a transformation of the kernel code itself to improve performance. For both concerns, static and dynamic analysis techniques are applicable. However, to address the second concern, an autotuning framework based on code transformation is required.

This research presents our static analyzer that can be used by developers, autotuners, and compilers for heterogeneous computing applications. Unlike most existing analysis techniques, our approach does not require any program runs to discover optimal parameter settings. The specific contributions described in this paper include the following.

- A static analyzer for CUDA programs.
- Predictive modeling based on static data.
- Example use cases of the new methodology in an autotuning context.

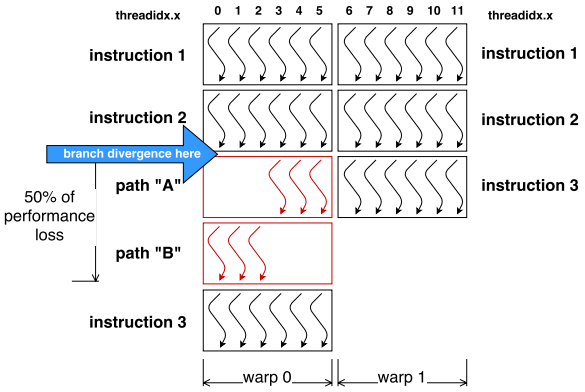

Figure 4. Branch divergence problem and performance loss incurred.

Background

This section briefly discusses the background for our research contributions, including the CUDA programming model, performance measurement approaches, and autotuning.

CUDA Programming Model and Control Flow Divergence. In CUDA kernels, threads are organized in groups called blocks, which consist of one or more warps (each of which has 32 threads). Each block is assigned to one of the GPU's streaming multiprocessors, and each SM is composed of multiple streaming processors, or multiprocessors (MP) that execute individual threads in SIMD.

In a given execution cycle, a SM executes instructions from one of the thread block's warps, where threads within a warp are executed together. However, if threads within a warp take different paths on conditional branches, execution of those paths become serialized. In the worst case, only 1 of the 32 threads within a warp will make progress in a cycle. Figure 4 shows how performance is affected when branches diverge. Measuring the occupancy of a kernel execution can determine whether branch divergence exists and suggest parameter adjustments to the program, a subject of this current work.

GPU Performance Tools. To date, GPU performance tools have mainly focused on the measurement and analysis of kernel execution, reporting time and counters associated with kernel execution. For instance, the TAU Performance System provides scalable, profile and trace measurement and analysis for high-performance parallel applications Shende and Malony (2006), including support for CUDA and OpenCL codes Malony et al. (2011). Even though profile measurements can help answer certain types of questions (e.g., how long did $foo()$ take?), improving performance requires more detailed information about the program structure.

While TAU and other profiling tools provide performance measurement Adhianto et al. (2010); ddt (2016); nvprof (2016), they do not shed much light on the divergent branch behavior and its effects on making good decisions about thread and block sizes. Our work introduces several static analysis techniques that deliver fine-grained information that can be used for predictive modeling. These techniques include the ability to analyze instruction mixes and occupancy for estimating thread and register settings. In a complementary approach (not discussed in this paper), we have also developed dynamic analysis techniques to compute instruction execution frequencies and control flow information Lim, Norris, and Malony (2016).

In the remainder of this section, we discuss how we model different performance-relevant metrics by using primarily static analysis of CUDA binaries.

Autotuning. By themselves, performance models can produce adequate predictions of parameter settings, but can not change the kernel to improve performance. Autotuning systems have been important in exploring alternative parameter choices by providing a kernel experimentation and optimization

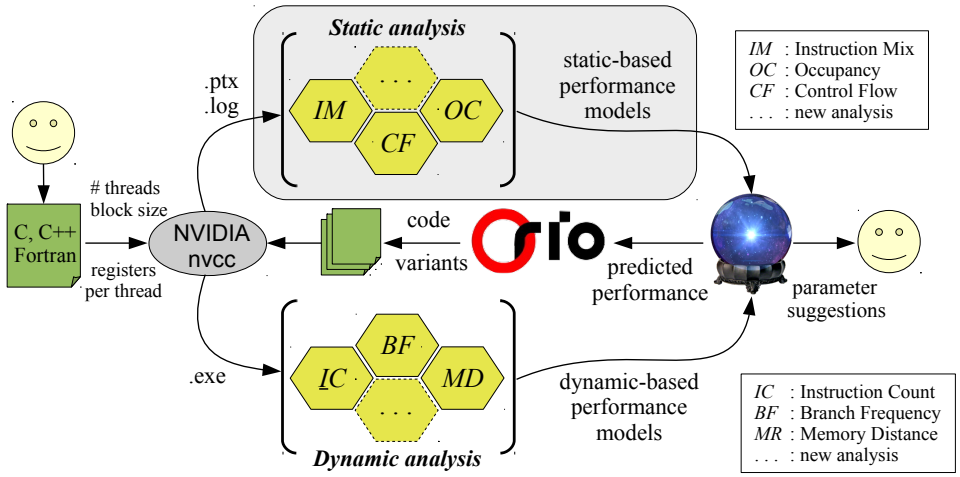

Figure 5. Optimization framework for GPU kernels incorporating static and dynamic analysis, with autotuning and code transformation.

framework. For example, the open-source Orio autotuning framework Hartono, Norris, and Sadayappan (2009) generates many code variants for each kernel computation. The objective of the GPU portions of Orio is to accelerate loops Chaimov, Norris, and Malony (2014); Mametjanov, Lowell, C.C. Ma, and Norris (2012) since loops consume a large portion of program execution time. We use the term kernels to refer to deeply nested loops that arise frequently in a number of scientific application codes. Existing C loops are annotated with transformation and tuning specifications. Transformations are parameterized with respect to various performance constraints, such as block sizes, thread counts, preferred L1 sizes and loop unroll factors. Each kernel specification generates a family of variant translations for each parameter and each variant is measured for its overall execution time, with the fastest chosen as the top performing autotuned translation.

The main challenge in the optimization space search is the costly empirical measurement of many code variants in autotuning systems. The main contribution

of our work is to demonstrate the use of static predictive models in autotuning, reducing the need for experimental performance testing.

Methodology

Figure 5 is a high-level depiction of our framework, which illustrates not only the different processes involved, but also the analysis support and tradeoffs inherent in them. For instance, providing a user with runtime parameters for kernel launch could engage static and/or dynamic analysis, but not necessarily code transformation. Dynamic analysis would be expected to be more costly because experiments would be involved. Transforming the implementation allows new variants to be explored, but these could be analyzed either statically or dynamically, or both. However, it is in the integration of these models with an autotuning system that can transform the kernel code where the greatest power for delivering optimizations is found.

Static Analysis

Our static analysis approach consists of the following steps:

1. Extract kernel compilation information with `nvcc`'s `--ptxas-options=-v` flag.

2. Disassemble CUDA binary with `nvdisasm` for instruction operations executed.

The subsequent sections define metrics resulting from our static analysis approach, including occupancy and instruction mixes. These metrics are then used to significantly reduce or even eliminate the empirical tests in autotuning several kernels.

Occupancy. Threads, registers and shared memory are factors that influence a CUDA kernel's ability to achieve high occupancy. In this section, we will group threads, warps, and blocks into one category for simplifying the discussion, although each term has its own distinct meaning. Threads (T) are the work units performing the computation, whereas warps (W) are the schedulable units for the streaming multiprocessor and blocks (B) consist of groups of warps. Each has memory local to its level. For instance, threads access private registers (R), warps and blocks use shared memory (S), and grids utilize global memory.

The following subsections define factors that contribute to a kernel's GPU occupancy. Table 16 lists the GPUs used in this research, along with hardware features and associated notation. We adopt the naming convention where superscripts denote the source of the variable, with subscripts as constraints of the variable. Compute capability (cc) represents the GPU architecture family (also listed in Tab. 16), meaning `nvcc` will target an architecture based on the assigned compute capability flag (e.g. `-arch=sm_xx`). User input (u) includes threads, registers and shared memory parameters at compile time. Active ($*$) represents the results provided by our static analyzer tool. Occupancy is the metric we are calculating and is defined in the next subsections.

Occupancy Calculation. The objective of the occupancy calculation is to minimize the number of active thread blocks per multiprocessor constrained by hardware resource ψ:

$$B^*_{mp} = \min \left\{ \mathcal{G}_\psi(u) \right\}, \tag{3.1}$$

where $\mathcal{G}(\cdot)$ calculates the maximum allocable blocks for each SM, and $\psi = \{\psi_W, \psi_R, \psi_S\}$ denotes warps, registers, and shared memory. Each $\mathcal{G}_\psi$ will be defined in Eqs. 3.3, 3.4, and 3.5.

Definition of Occupancy. Occupancy is defined as the ratio of active warps on a SM to the maximum number of active warps supported for each SM:

$$occ_{mp} = \frac{W^*_{mp}}{W^{cc}_{mp}} \tag{3.2}$$

where $W^*_{mp} = B^*_{mp} \times W_B$, with B^*_{mp} as defined in Eq. 3.1 and $W_B = 32$ for all GPUs (Tab. 16). Note that in an ideal world, $occ_{mp} = 1$. However, in practice, occupancy rates are on average at 65-75%, and should not be used in isolation for setting CUDA parameters Volkov (2010). Occupancy is one of several metrics we incorporated in our static analyzer.

Theoretical Occupancy. The number of blocks which can execute concurrently on an SM is limited by either warps, registers, or shared memory.

Warps per SM The SM has a maximum number of warps that can be active at once. To calculate the maximum number of blocks constrained by warps $\mathcal{G}_{\psi_W}$, find the minimum of blocks supported per multiprocessor and the rate of warps per SM and warps per block:

$$\mathcal{G}_{\psi_W}(T^u) = \min\left\{B^{cc}_{mp}, \left\lfloor \frac{W_{sm}}{W_B} \right\rfloor\right\} \tag{3.3}$$

where $W_{sm} = W^{cc}_{mp}$ and $W_B = \left\lceil \frac{T^u}{T^{cc}_W} \right\rceil$, with variables as defined in Table 16.

Registers per SM The SM has a set of registers shared by all active threads. Deciding whether registers is limiting occupancy $\mathcal{G}_{\psi_R}$ is described by the following cases:

$$\mathcal{G}_{\psi_R}(R^u) = \begin{cases} 0 & \text{if } R^u > R_W^{cc}, \\ \left\lfloor \frac{R_{sm}}{R_B} \right\rfloor \times \left\lfloor \frac{R_{fs}^{cc}}{R_B^{cc}} \right\rfloor & \text{if } R^u > 0, \\ B_{mp}^{cc} & \text{otherwise.} \end{cases} \tag{3.4}$$

where $R_{sm} = \left\lfloor \frac{R_B^{cc}}{\lceil R^u \times T_W^{cc} \rceil} \right\rfloor$ and $R_B = \left\lceil \frac{T^u}{T_W^{cc}} \right\rceil$. Case 1 represents when the user declares a register value beyond the maximum allowable per thread that is supported for the *cc*, an illegal operation. Case 2 describes when the user provides a valid register value, where we take the product of the number of registers per SM supported over the number of registers per block and the register file size per MP over the maximum register block supported in this architecture. Case 3 is when the user does not provide a value, where the value is set to the thread block per multiprocessor supported by the *cc*.

Shared memory per SM Shared memory per thread is defined as the sum of static shared memory, the total size needed for all `__shared__` variables and dynamic shared memory. If active blocks are constrained by shared memory, reducing S per T could increase occupancy. To compute $\mathcal{G}_{\psi_S}$, take the ceiling of the shared memory per multiprocessor provided by its compute capability over the shared memory per block.

$$\mathcal{G}_{\psi_S}(S^u) = \begin{cases} 0 & \text{if } S^u > S_B^{cc}, \\ \left\lceil \frac{S_{mp}^{cc}}{S_B} \right\rceil & \text{if } S^u > 0, \\ B_{mp}^{cc} & \text{otherwise.} \end{cases} \tag{3.5}$$

where shared memory per block $S_B = \lfloor S^u \rfloor$, shared memory per SM $S_{sm} = S_B^{cc}$, and with cases following similarly to Eq. 3.4.

Instruction Mix Metrics. Instruction mix is defined as the number of specific operations that a processor executes. Instruction mix-based characterizations have been used in a variety of contexts, including to select loop unrolling factors Monsifrot, Bodin, and Quiniou (2002); Stephenson and Amarasinghe (2005), unlike hardware counters which are prone to miscounting events Lim, Carrillo-Cisneros, Alkowaileet, and Scherson (2014). In this work, we use instruction mixes to characterize whether a kernel is memory-bound, compute-bound, or relatively balanced. Refer to Lim, Malony, Norris, and Chaimov (2015) for definitions for $\mathbf{O}_{fl}, \mathbf{O}_{mem}, \mathbf{O}_{ctrl}$, and $\mathbf{O}_{reg}$ according to category type.

The intensity (magnitude) of a particular metric can suggest optimal block and thread sizes for a kernel. Memory-intensive kernels require a high number of registers, where a large block size consists of more registers per block. The tradeoff with big block sizes is that fewer blocks can be scheduled on the SM. Small block sizes will constrain the number of blocks running on the SM by the physical limit of blocks allowed per SM. Compute-intensive kernels perform well with larger block sizes because the threads will be using GPU cores with fewer memory latencies. Small block sizes will result in many active blocks running on the SM in a time-shared manner, where unnecessary switching of blocks may degrade performance. For control-related synchronization barriers, smaller block sizes are preferred

Table 3. Instruction throughput per number of cycles.

Category	Op	SM20	SM35	SM52	SM60
FPIns32	FLOPS	32	192	128	64
FPIns64	FLOPS	16	64	4	32
CompMinMax	FLOPS	32	160	64	32
Shift, Extract, Shuffle, SumAbsDiff	FLOPS	16	32	64	32
Conv64	FLOPS	16	8	4	16
Conv32	FLOPS	16	128	32	16
LogSinCos	FLOPS	4	32	32	16
IntAdd32	FLOPS	32	160	64	32
TexIns, LdStIns, SurfIns	MEM	16	32	64	16
PredIns, CtrlIns	CTRL	16	32	64	16
MoveIns	CTRL	32	32	32	32
Regs	REG	16	32	32	16

because many active blocks can run simultaneously on the SM to hide memory latency.

Pipeline Utilization. Each streaming multiprocessor (SM) consists of numerous hardware units that are specialized in performing a specific task. At the chip level, those units provide execution pipelines to which the warp schedulers dispatch instructions. For example, texture units provide the ability to execute texture fetches and perform texture filtering, whereas load/store units fetch and save data to memory. Understanding the utilization of pipelines and its relation to peak performance on target devices helps identify performance bottlenecks in terms of oversubscription of pipelines based on instruction type.

The NVIDIA Kepler GK100 report objdump (2012) lists instruction operations and corresponding pipeline throughputs per cycle. Pipeline utilization describes observed utilization for each pipeline at runtime. High pipeline utilization

```
/*@ begin PerfTuning (
  def performance_params {
    param TC[]  = range(32,1025,32);
    param BC[]  = range(24,193,24);
    param UIF[] = range(1,6);
    param PL[]  = [16,48];
    param SC[]  = range(1,6);
    param CFLAGS[] = ['', '-use_fast_math'];
  }
  ...
) @*/
```

Figure 6. Performance autotuning specification.

would indicate that the corresponding compute resources were used heavily and kept busy often during the execution of the kernel.

Infer Kernel Execution Time. Because the majority of CUDA applications are accelerated loops, we hypothesize that the execution time of a CUDA program is proportional to the input problem size N. Hence,

$$f(N) = c_f \cdot \mathbf{O}_{fl} + c_m \cdot \mathbf{O}_{mem} + c_b \cdot \mathbf{O}_{ctrl} + c_r \cdot \mathbf{O}_{reg} \tag{3.6}$$

where c_f, c_m, c_b, and c_r are coefficients that represent the reciprocal of number of instructions that can execute in a cycle, or CPI. Equation 3.6 represents how a program will perform for input size N without running the application.

Table 4. A subset of features used for thread block classification.

Feature	**Size**
Thread Count	32 – 1024 (with 32 increments)
Block size [1]	24 – 192 (with 16 increments)
Unroll loop factor	{1 – 6}
Compiler flags	{'', '-use_fast_math'}
Instructions	{FLOPS, memory, control}
Occupancy calculation	{registers, threads, OCC rate, etc.}

[1]Block sizes are compute capability specific.

ORIO Code Generation.

OCC Results

This section reports on the autotuning execution environment for the CUDA kernels listed in Table 5. Results comparing our static analyzer approach with the existing methods are also reported.

Environment. The open-source Orio autotuner was used to generate and autotune CUDA implementations by varying the feature space listed in Table 4. The details of CUDA code generation and autotuning with Orio are described in Mametjanov et al. (2012). The *TC* parameter specifies the number of simultaneously executing threads. *BC* is the number of thread blocks (independently executing groups of threads that can be scheduled in any order across any number of SMs) and is hardware-specific. *UIF* specifies how many times loops should be unrolled; *PL* is the L1 cache size preference in KB.

For each code variant, the experiment was repeated ten times, and the fifth overall trial time was selected to be displayed. The execution times were sorted in ascending order and the ranks were split along the 50th percentile. Rank 1 represents the upper-half of the 50th percentile (good performers), while Rank 2 represents the lower portion (poor performers). On average, the combination of parameter settings generated 5,120 code variants. The GPUs used in this work are listed in Table 16 and include the Fermi M2050, Kepler K20, Maxwell M40, and Pascal P100. Subsequently we will refer to the GPUs by the architecture family name (Fermi, Kepler Maxwell, Pascal). CUDA `nvcc` v7.0.28 was used as the compiler. Each of the benchmarks executed with five different input sizes, where all benchmarks consisted of inputs {32, 64, 128, 256, 512}, except ex14FJ, which had inputs {8, 16, 32, 64, 128}.

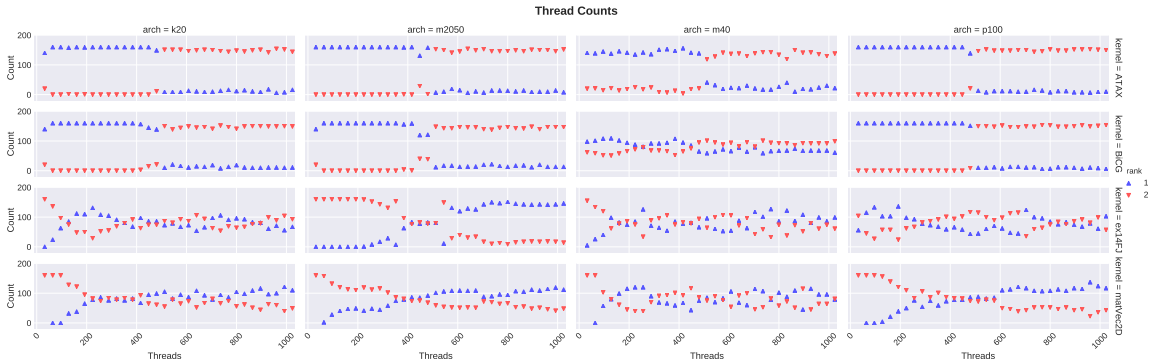

Figure 7. Thread counts for Orio autotuning exhaustive search, comparing architectures and kernels.

To demonstrate our approach, we considered the kernels described in Table 5. Because the chosen kernels (except ex14FJ, which is application-specific) contribute significantly to the overall execution time of many different applications, tuning these kernels can result in significant overall application performance improvements.

Table 5. Kernel specifications.

Kernel	Category	Description	Operations
`atax`	Elementary linear algebra	Matrix transpose, vector multiplication	$y = A^T(Ax)$
`BiCG`	Linear solvers	Matrix transpose, Subkernel of BiCGStab linear solver	$q = Ap$, $s = A^T r$
`ex14FJ`	3-D Jacobi computation	Stencil code kernels	$F(x) = A(x)x - b = 0$, $A(u)v \simeq - \bigtriangledown (\kappa(u) \bigtriangledown v)$
`MatVec2D`	Elementary linear algebra	Matrix vector multiplication	$y = Ax$

Discussion

We empirically autotuned the kernels listed in Table 5 using exhaustive search and uncovered distinct ranges for block and thread sizes, based on ranking. The dynamic analysis of autotuning is displayed in Figure 7, projecting thread settings and frequency for each kernel, and comparing various architectures. In

Figure 8. Time-to-instruction mix ratio, comparing architectures and kernels.

Table 6. Statistics for autotuned kernels for top performers (top half) and poor performers (bottom half), comparing GPU architecture generations.

		Occupancy			Register Instructions			Threads		
		Mean	Std Dev	Mode	Mean	Std Dev	Allocated	25th	50th	75th
ATAX	Fer	77.46	24.18	100.00	39613.1	35673.2	21	152	272	416
	Kep	85.21	19.03	93.75	34833.1	30954.5	27	160	288	416
	Max	90.59	11.87	93.75	104285.9	85207.1	30	160	320	448
	Pas	90.86	12.24	93.75	227899.7	202120.2	30	152	272	392
BiCG	Fer	60.55	15.54	75.00	35321.3	32136.6	27	160	288	416
	Kep	85.14	19.05	98.44	35485.7	31535.9	28	160	288	416
	Max	89.09	11.50	98.44	158963.8	135681.2	32	224	448	736
	Pas	90.93	12.19	93.75	228350.6	201865.8	30	152	272	392
ex14FJ	Fer	53.69	8.83	62.50	98418.5	45166.64	30	608	768	896
	Kep	88.44	9.98	93.75	54345.4	47526.8	31	288	512	768
	Max	89.23	9.61	98.44	4141130.6	158537.4	28	320	608	832
	Pas	89.04	11.10	98.44	4335986.6	409162.6	32	192	480	768
matVec2D	Fer	72.21	14.17	87.50	307425.50	69330.06	23	448	640	864
	Kep	89.29	8.17	96.88	274359.93	65373.88	23	416	640	864
	Max	89.53	9.22	98.43	693752.81	146799.80	18	288	576	800
	Pas	88.42	9.08	90.63	1264815.81	316252.38	18	480	672	864
ATAX	Fer	74.23	15.98	100.00	102946.9	58009.0	21	640	768	896
	Kep	86.27	10.97	93.75	89906.9	51102.5	27	640	768	896
	Max	87.04	10.09	87.50	253714.1	151973.5	30	608	736	896
	Pas	86.77	9.54	87.50	605300.3	337615.5	30	640	768	896
BiCG	Fer	56.12	10.73	66.67	35321.3	32136.6	27	608	768	896
	Kep	86.34	10.93	93.75	89254.3	51141.2	28	608	768	896
	Max	88.55	10.80	93.75	199036.3	149373.1	32	352	608	832
	Pas	86.70	9.57	87.5	605169.4	338092.4	30	640	768	896
ex14FJ	Fer	55.55	14.03	62.50	26321.5	21137.2	30	152	288	448
	Kep	83.05	19.21	93.75	70394.6	51953.5	31	256	544	800
	Max	88.40	12.50	93.75	4079589.4	120401.0	28	224	480	704
	Pas	88.59	11.22	93.75	4359934.4	241618.2	32	352	544	800
matVec2D	Fer	68.93	21.93	87.50	210334.50	47850.90	23	160	352	672
	Kep	82.19	19.54	93.75	219636.56	57185.33	23	160	384	704
	Max	88.09	12.77	93.75	645687.18	137182.93	18	224	480	736
	Pas	89.22	12.89	93.75	877505.0	225900.05	18	160	320	576

Table 7. Error rates when estimating dynamic instruction mixes from static mixes.

		Metrics			
		FLOPS	MEM	CTRL	Itns
ATAX	Fer	0.07	1.69	2.01	3.4
	Kep	0.11	1.75	2.20	3.4
	Max	0.23	0.06	0.12	1.8
BiCG	Fer	0.03	3.68	2.40	1.8
	Kep	0.02	3.80	2.67	1.8
	Max	0.57	1.30	0.06	1.3
ex14FJ	Fer	0.20	0.14	0.00	12.7
	Kep	1.01	0.18	0.21	12.7
	Max	1.97	0.14	0.89	16.3
matVec2D	Fer	0.04	0.92	0.80	4.6
	Kep	0.07	0.97	0.99	4.6
	Max	0.29	0.06	0.36	7.2

general, ATAX and BiCG kernels performed well in lower thread range settings, whereas matVec2D performed better with higher thread settings. The ex14FJ is a more complex kernel[2], and thread behavior patterns for Rank 1 were less apparent.

Table 6 reports statistics on occupancy, registers, and threads for all benchmarks and architectures. The top half represents good performers (Rank 1), whereas the bottom half represents poor performers (Rank 2). In general, occupancy did not seem to matter much, since the reported means were somewhat similar for both ranks, with Fermi achieving low *occ* for all kernels. However, register instructions varied considerably, with Rank 1 consisting of lower mean and standard deviations, versus Rank 2 which had higher values. Thread behavior patterns were apparent when comparing Rank 1 and Rank 2. For instance, one could conclude that ATAX and BiCG prefers smaller range thread sizes, whereas ex14FJ prefers higher ranges.

[2]The ex14FJ kernel is the Jacobian computation for a solid fuel ignition simulation in 3D rectangular domain.

Figure 8 illustrates the use of static instruction mixes to predict execution time. Execution time was normalized and sorted in ascending order (x-axis). The mean absolute error was used to estimate execution time based on static instruction mixes. Equation 3.6 was used to calculate the instruction mix ratio, which consisted of weighting instructions according to its number of achievable executed instructions per clock cycle. In general, our model was able to estimate the execution time within a reasonable margin of error, including `ex14FJ` with *MAE* near 1.00, which validates instruction mixes as good indicators of kernel execution performance.

Table 7 reports the error rates calculated, using sum of squares, when estimating dynamic behavior of the kernel from static analysis of the instruction mix. Intensity is also displayed in the last column and is defined as the ratio of floating-point operations to memory operations. Although our static estimator performed poorly for BiCG (memory, control ops), our static analysis, driven by Equation 3.6, closely matches that of the observed dynamic behavior for the other kernels.

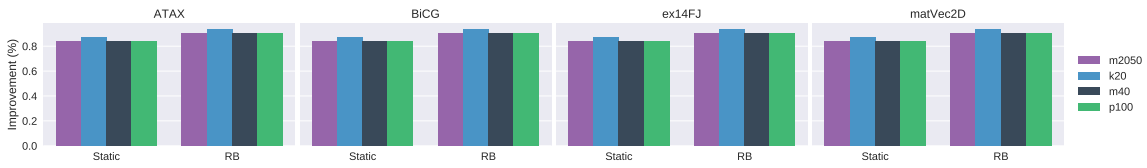

Figure 9. Improved search time over exhaustive autotuning, comparing static and rule-based approaches.

Improved Autotuning with Static Analyzer. Finally, we wanted to determine whether our static analyzer tool could be used to improve the efficiency and effectiveness of Orio. We use the exhaustive empirical autotuning results from

Table 8. Suggested parameters to achieve theoretical occupancy.

		T^*	$[R^u : R^*]$	S^*	occ^*
AT	Fer	192, 256, 384, 512, 768	[21 : 0]	6144	1
	Kep	128, 256, 512, 1024	[27 : 5]	3072	1
	Max	64, 128, 256, 512, 1024	[30 : 2]	1536	1
	Pas	64, 128, 256, 512, 1024	[30 : 2]	1536	1
Bi	Fer	192, 256, 384, 512, 768	[27 : 0]	8192	.75
	Kep	128, 256, 512, 1024	[28 : 4]	3072	1
	Max	64, 128, 256, 512, 1024	[32 : 0]	12288	.71
	Pas	64, 128, 256, 512, 1024	[30 : 2]	1536	1
ex	Fer	192, 256, 384, 512, 768	[30 : 0]	24576	.71
	Kep	128, 256, 512, 1024	[31 : 1]	3072	1
	Max	64, 128, 256, 512, 1024	[28 : 4]	1536	1
	Pas	64, 128, 256, 512, 1024	[32 : 0]	1536	1
ma	Fer	192, 256, 384, 512, 768	[20 : 1]	12288	.92
	Kep	128, 256, 512, 1024	[20 : 11]	3072	1
	Max	64, 128, 256, 512, 1024	[13 : 18]	1536	1
	Pas	64, 128, 256, 512, 1024	[15 : 17]	1536	1

Sec. III as the baseline for validating whether our search approach could find the optimal solution.

Table 8 reports static information for register usage and intensity for each kernel, as well as the thread parameters suggested by our static analyzer, comparing different architectures. T^* displays the suggested thread ranges for the kernel that would yield occ^*. $[R^u : R^*]$ displays the number of registers used and its increase potential. S^* displays (in KB) the amount of shared memory that could be increased to achieve theoretical occupancy.

The basis of our contribution is that the instruction mix and occupancy metrics from our static analyzer gets fed into the autotuner. In general, an exhaustive autotuning consists of $\prod_{i=1}^{m} \mathbf{X}_i$ trials, where $\mathbf{X}_i$ represents a parameter, each having m options. In the case of ATAX, five thread settings were suggested for Fermi and Maxwell, which represents a 84% improvement, and Kepler

representing a 87.5% improvement, with the search space reduced from 5,120 to 640. The search space could be reduced further by invoking our rule-based heuristic. Figure 9 displays the overall results of the improved search module. The first set displays how the static based method improves near 87.5%. When combining with the rule-based heuristic, the search space is further reduced, which results in a 93.8% overall improvement. Figure 10 displays the occupancy calculator for the ATAX kernel, comparing the current kernel and the potentially optimized version.

The model-based search space reduction does involve generating and compiling the code versions, but it does **not** require executing them. Note that empirical testing typically involves multiple repeated executions of the same code version, hence the time saved over exhaustive search is approximately $t * r$, where t is the average trial time and r is the number of repetitions. Even when not using exhaustive search, our new technique can be used as the first stage of the regular empirical-based autotuning process to dramatically reduce the search space, significantly speeding up the entire process and increasing the likelihood of finding a global optimum. Unlike runtime measurement, which requires several runs of each test, static analysis does not suffer from the effects of noise and hence only has to be performed once on each code version. The search space reduced through static binary analysis can then be explored using one of the existing search methods. If it's feasible and desirable to determine the optimal value, then exhaustive search is appropriate, otherwise one of the other methods such as Nelder-Mead simplex or random can be used to strictly control the time spent autotuning.

Related Work

Several prior efforts have attempted to discover optimal code forms and runtime parameter settings for accelerator-based programming models, typically by taking a domain-specific approach. For instance, Nukada and Matsuoka demonstrated automated tuning for a CUDA-based 3-D FFT library based on selection of optimal number of threads Nukada and Matsuoka (2015). Tomov et al. developed the MAGMA system for dense linear algebra solvers for GPU architectures, which incorporates a DAG representation and empirical-based search process for modeling and optimization Tomov, Nath, Ltaief, and Dongarra (2010). The use of autotuning systems based on program transformations, such as Orio Hartono et al. (2009) and CHiLL *CHiLL: A Framework for Composing High-Level Loop Transformations* (2008), enable optimization exploration on more general application code and across accelerator architectures Chaimov et al. (2014). However, the complexity of the optimization space and the cost of empirical search is high. A recent work on autotuning GPU kernels focuses on loop scheduling and is based on the OpenUH compiler Xu, Chandrasekaran, Tian, and Chapman (2016). Our approach attempts to leverage more static code analysis to help better inform an autotuning process, thereby reducing the dependence on pure dynamic measurement and analysis to generate performance guidance.

The NVIDIA CUDA Toolkit NVIDIA (n.d.) includes occupancy calculation functions in the runtime API that returns occupancy estimates for a given kernel. In addition, there are occupancy-based launch configuration functions that can advise on grid and block sizes that are expected to achieve the estimated maximum potential occupancy for the kernel. Because these functions take as input intended per-block dynamic shared memory usage and maximum block size (in

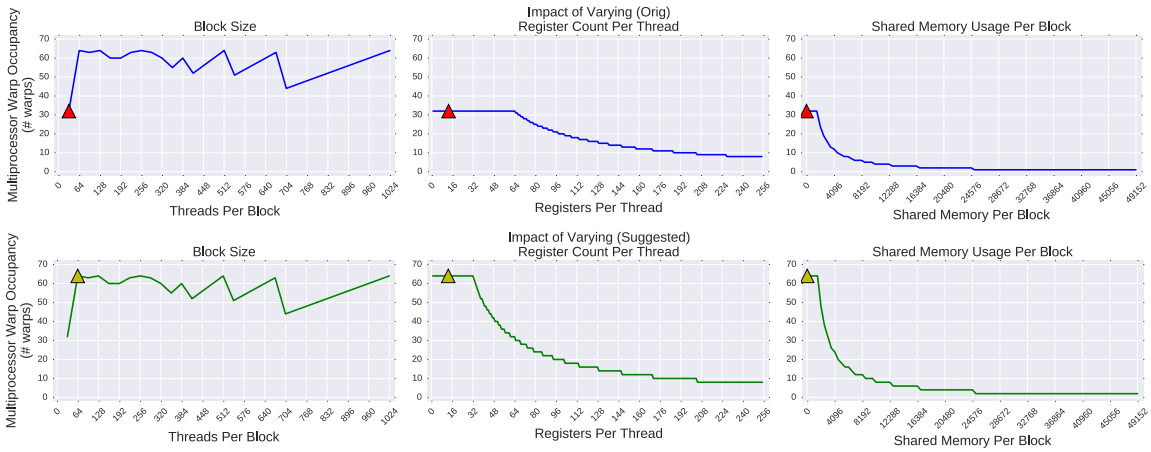

Figure 10. Occupancy calculator displaying thread, register and shared memory impact for current (top) and potential (bottom) thread optimizations for the purposes of increasing occupancy.

addition to knowing user-defined registers per thread), it is possible to retrieve a set of configuration choices. It is important to note that the CUDA Occupancy Calculator/API takes into account the GPU architecture being used. Thus, we can integrate the estimates it generates over the full range of options (e.g., letting registers per thread to be variable) with the other static models.

A project closely related to ours is STATuner R. Gupta et al. (2015), which identifies a feature set of static metrics that characterize a CUDA kernel code and uses machine learning to build a classifier model trained on a CUDA benchmark suite. Kernel codes are compiled in LLVM and static analysis of the generated binary code and IR provide metrics for instruction mix, loops, register usage, shared memory per block, and thread synchronization. The classifier model inputs these metric features for a new kernel to predict which block size would give the best performance. STATuner is shown to give smaller average error compared to NVIDIA's CUDA Occupancy Calculator/API. Only a single block size is predicted by STATuner, whereas the Occupancy Calculator/API offers block size choices

given user input about registers per thread and per-block shared memory. Our approach differs in several respects. First, static analysis is done on the PTX code generated by the NVIDIA nvcc compiler, rather that on the upper level source code (as seen in LLVM). While there are some benefits in incorporating higher-level code information, nvcc produces different PTX code for different GPU architectures, allowing hardware-specific code effects to be seen. Furthermore, our static analysis extracts metrics similar to STATuner, but also builds a CFG to help understand flow divergence Lim et al. (2016). Second, our prediction models are based on estimating performance given the instruction mix, control flow, and problem size. They are not based on learned classifiers. Third, the objective of our work is to integrate predictive models in an autotuning framework, beyond just giving a single block size result to the user.

Milepost GCC Fursin (2011) is a publicly-available open-source machine learning-based compiler for C (but not CUDA) that extracts program features and exchanges optimization data with the cTuning.org open public repository. It automatically adapts the internal optimization heuristic at function-level granularity to improve execution time, code size and compilation time of a new program on a given architecture.

The Oxbow toolkit Sreepathi et al. (2014) is a collection of tools to empirically characterize (primarily CPU) application behaviors, including computation, communication, memory capacity and access patterns. The eventual goal is to build a repository that users can upload and access their datasets, and can provide analysis, plots, suggested parameters, etc.

Discussion

Getting the most performance out of applications is important for code generators and end users, but the process in making the best settings is often convoluted (for humans) and time-consuming (for empirical autotuners). With our static analyzer tool, we show its accuracy in estimating the runtime behavior of a kernel without the high costs of running experiments. Using our tool, we've identified the computational intensity of a kernel, constructed a control flow graph, estimated the occupancy of the multiprocessors, and suggested optimizations in terms of threads and register usage. Finally, we've shown how the integration of our static analyzer in the Orio autotuning framework improved the performance in narrowing the search space for exploring parameter settings.

The field of heterogeneous accelerated computing is rapidly changing, and we expect several disruptions to take place with the introduction of 3D-memory subsystems, point-to-point communication, and more registers per computational cores. Traditional approaches to measuring performance may no longer be sufficient to understand the behavior of the underlying system. Our static analyzer approach can facilitate optimizations in a variety of contexts through the automatic discovery of parameter settings that improve performance.

Future Work

The optimization spectrum is a continuum from purely static-based methods to ones that incorporate empirical search across an optimization landscape.
In general, the objective of our work is on exploring the tradeoffs involving optimization accuracy and cost over this spectrum, with a specific focus on how well purely static methods perform as a guide for autotuning. While static analysis side-steps the need for empirical testing, it is not to say that static models can not

be informed by prior benchmarking and knowledge discovery. We will investigate several avenues for enhancing our static models, including algorithm-specific optimizations and machine learning for code classification.

Furthermore, we regard the methodology we have developed as a knowledge discovery framework where the degree of empirical testing can be "dialed in" during the autotuning process, depending on what the user accepts. By recording the decisions and code variants at each step, it is also possible to replay tuning with empirical testing for purpose of validation. In this way, the framework can continually evaluate the static models and refine their predictive power. We will further develop this capability.

While our static analysis tools will working with any CUDA kernel code, the real power of our approach is in the ability to transform the code in Orio. However, this requires the source to be in a particular input form. We are exploring source analysis technology de Oliveira Castro, Akel, Petit, Popov, and Jalby (2015) to translate kernel code to the input required by Orio, thereby allowing any kernel to be a candidate for CUDA autotuning.

Conclusion

This chapter defined the metrics necessary for optimizing the performance of GPU kernels. Specifically, threads, registers and shared memory, as well as architectural factors were included in the metrics definition. This research revealed that certain computation patterns, whether memory, compute or control bound, have an influence on the parameter settings of a CUDA application. A static model was proposed, based on the instruction mixes, that was able to predict the performance of an execution kernel with a mean absolute error near 1.00. The next

chapter builds on these approaches and defines a similarity measure for matching control flow graphs.

CHAPTER IV

CONTROL FLOW SUBGRAPH MATCHING

This chapter includes previously published co-authored material from a NVIDIA GPU Technology Conference poster Lim et al. (2016) and a workshop paper at the 31st International Workshop on Languages and Compilers for Parallel Computing Lim, Norris, and Malony (2019). I was the primary contributor to this work in developing the algorithm, writing the new code, and writing the paper. Dr. Boyana Norris initially identified the need for this work and provided the application that this work was performed in. Dr. Allen Malony assisted in editing the paper.

Abstract

Accelerator architectures specialize in executing SIMD (single instruction, multiple data) in lockstep. Because the majority of CUDA applications are parallelized loops, control flow information can provide an in-depth characterization of a kernel. `CUDAflow` is a tool that statically separates CUDA binaries into basic block regions and dynamically measures instruction and basic block frequencies. `CUDAflow` captures this information in a control flow graph (CFG) and performs subgraph matching across various kernel's CFGs to gain insights into an application's resource requirements, based on the shape and traversal of the graph, instruction operations executed and registers allocated, among other information. The utility of `CUDAflow` is demonstrated with SHOC and Rodinia application case studies on a variety of GPU architectures, revealing novel control flow characteristics that facilitate end users, autotuners, and compilers in generating high performing code.

Motivation

Structured programming consists of base constructs that represent how programs are written Böhm and Jacopini (1966); Williams and Ossher (1978). When optimizing programs, compilers typically operate on the intermediate representation (IR) of a control flow graph (CFG), which is derived from program source code analysis and represents basic blocks of instructions (nodes) and control flow paths (edges) in the graph. Thus, the overall program structure is captured in the CFG and the IR abstracts machine-specific intrinsics that the compiler ultimately translates to machine code. The IR/CFG allows the compiler to reason more efficiently about optimization opportunities and apply transformations. In particular, compilers can benefit from prior knowledge of optimizations that may be effective for specific CFG structures.

In the case of accelerated architectures that are programmed for SIMD parallelism, control divergence encountered by threads of execution presents a major challenge for applications because it can severely reduce SIMD computational efficiency. It stands to reason that by identifying the structural patterns of a CFG from an accelerator (SIMD) program, insight on the branch divergence problem Sabne, Sakdhnagool, and Eigenmann (2016) might be gained to help in their optimization. Current profiling approaches to understanding thread divergence behavior (e.g., ddt (2016); nvprof (2016); Shende and Malony (2006)) do not map performance information to critical execution paths in the CFG. While accelerator devices (e.g., GPUs) offer hardware performance counters for measuring computational performance, it is more difficult to apply them to capture divergence behavior Lim et al. (2015).

Our research focuses on improving the detail and accuracy of control flow graph information in accelerator (GPU) programs. We study the extent to which CFG data can provide sufficient context for understanding a GPU kernel's execution performance. Furthermore, we want to investigate how effective knowledge of CFG shapes (patterns) could be in enabling optimizing compilers and autotuners to infer execution characteristics without having to resort to running execution experiments. To this end, we present `CUDAflow`, a scalable toolkit for heterogeneous computing applications. Specifically, `CUDAflow` provides a new methodology for characterizing CUDA kernels using control flow graphs and instruction operations executed. It performs novel kernel subgraph matching to gain insights into an application's resource requirements. To the knowledge of the authors, this work is a first attempt at employing subgraph matching for revealing control flow behavior and generating efficient code.

Contributions described in this paper include the following.

- Systematic process to construct control flow graphs for GPU kernels.
- Techniques to perform subgraph matching on various kernel CFGs and GPUs.
- Approaches to reveal control flow behavior based on CFG properties.

Prior Work

Control flow divergence in heterogeneous computing applications is a well known and difficult problem, due to the lockstep nature of the GPU execution paradigm. Current efforts to address branch divergence in GPUs draw from several fields, including profiling techniques in CPUs, and software and hardware architectural support in GPUs. For instance, Sarkar demonstrated that the overall execution time of a program can be estimated by deriving the variances of basic

block regions Sarkar (1989). Control flow graphs for flow and context sensitive profiling were discussed in Ammons, Ball, and Larus (1997); Ball and Larus (1994), where instrumentation probes were inserted at selected edges in the CFG, which reduced the overall profiling overhead with minimal loss of information. Hammock graphs were constructed Zhang and D'Hollander (2004) that mapped unstructured control flow on a GPU Diamos et al. (2011); H. Wu, Diamos, Li, and Yalamanchili (2011). By creating thread frontiers to identify early thread reconvergence opportunities, dynamic instruction counts were reduced by as much as 633.2%.

Lynx Farooqui, Kerr, Eisenhauer, Schwan, and Yalamanchili (2012) creates an internal representation of a program based on PTX and then emulates it, which determines the memory, control flow and parallelism of the application. This work closely resembles ours but differs in that we perform workload characterization on actual hardware during execution. Other performance measurement tools, such as HPCToolkit Adhianto et al. (2010) and DynInst Miller et al. (1995), provide a way for users to construct control flow graphs from CUDA binaries, but do not analyze the results further. The MIAMI toolkit Marin, Dongarra, and Terpstra (2014) is an instrumentation framework for studying an application's dynamic instruction mix and control flow but does not support GPUs.

Subgraph matching has been explored in a variety of contexts. For instance, the DeltaCon framework matched arbitrary subgraphs based on similarity scores Koutra, Vogelstein, and Faloutsos (n.d.), which exploited the properties of the graph (e.g., clique, cycle, star, barbell) to support the graph matching. Similarly, frequent subgraph mining was performed on molecular fragments for drug discovery Borgelt and Berthold (2002), whereas document clustering was formalized in a

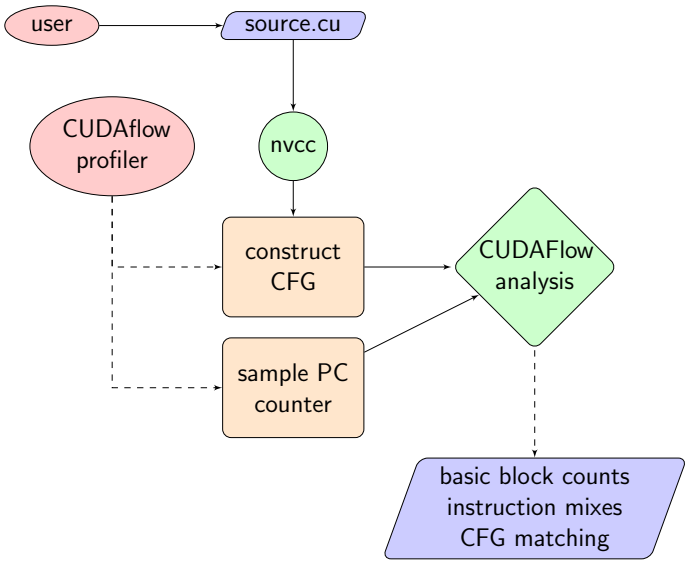

Figure 11. Overview of our proposed `CUDAflow` methodology.

graph database context Huan, Wang, and Prins (2003). The IsoRank authors consider the problem of matching protein-protein interaction networks between distinct species Singh, Xu, and Berger (2007). The goal was to leverage knowledge about the proteins from an extensively studied species, such as a mouse, which when combined with a matching between mouse proteins and human proteins can be used to hypothesize about possible functions of proteins in humans. However, none of these approaches apply frequent subgraph matching for understanding performance behavior of GPU applications.

Background

Our `CUDAflow` approach shown in Figure 11 works in association with the current `nvcc` toolchain. Control flow graphs are constructed from static code analysis and program execution statistics are gathered dynamically through program counter sampling. This measurement collects counts of executed instructions and corresponding source code locations, among other information. In this way, the `CUDAflow` methodology provides a more accurate characterization

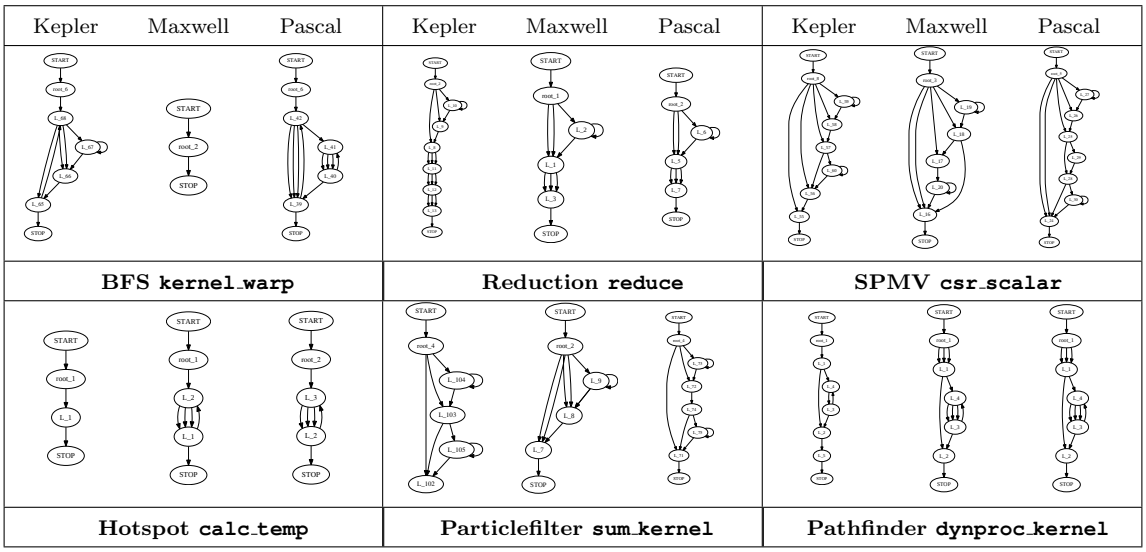

Figure 12. Control flow graphs generated for each CUDA kernel, comparing architecture families (Kepler, Maxwell, Pascal).

of the application kernel, versus hardware performance counters alone, which lack the ability to correlate performance with source line information and are prone to miscounting events Lim et al. (2014). In particular, it gives a way to understand the control flow behavior during execution.

Kernel Control Flow Graphs. One of the more complex parameters used to characterize SIMD thread divergence is by using a control flow graph (CFG) representation of the computation. A CFG is constructed for each GPU kernel computation in program order and can be represented as a directed acyclic graph $G = (N, E, s)$, where (N, E) is a finite directed graph, and a path exists from the $START$ node $s \in N$ to every other node. A unique $STOP$ node is also assumed in the CFG. A node in the graph represents a basic block (a straight line of code without jumps or jump targets), whereas directed edges represent jumps in the control flow.

Each basic block region is incremented with the number of times the node is visited. Upon sampling the program counter, the PC address is referenced internally to determine to which basic block region the instruction corresponds to.

```
.L_41:
   /*04a0*/ DSETP.LE.AND P0,PT,|R6|,+INF,PT;
   /*04a8*/ @P0 BRA `(.L_43);
   /*04b0*/ LOP32I.OR R5, R7, 0x80000;
   /*04b8*/ MOV R4, R6;


   /*04c8*/ BRA `(.L_42);
```

Example control flow graphs for selected SHOC (top) Danalis et al. (2010) and Rodinia (bottom) Che et al. (2009) GPU benchmarks are displayed in Figure 12. Different GPU architecture types will result in the `nvcc` compiler producing different code and possibly control flow, as seen in the CFGs from Figure 12 for Kepler, Maxwell and Pascal architectures. Section III discusses the differences in GPU architectures. The CFG differences for each architecture are due in part to the architecture layout of the GPU and its compute capability (NVIDIA virtual architecture). The Maxwell generally uses fewer nodes for its CFGs, as evident in `kernel_warp`. Our approach can expose these important architecture-specific effects on the CFGs. Also, note that similarities in structure exist with several CFGs, including `csr_scalar` and `sum_kernel`. Part of the goal of this research is to predict the required resources for the application by inferring performance through CFG subgraph matching, with the subgraphs serving as building blocks for more nested and complex GPU kernels. For this purpose, we introduce several metrics that build on this CFG representation.

$$\begin{array}{c} \begin{matrix} R_1 & L_1 & L_4 & L_3 & L_2 & L_5 \end{matrix} \\ \begin{bmatrix} .21 & - & - & - & - & - \\ 0 & .04 & - & - & - & - \\ 0 & .04 & .38 & - & - & - \\ 0 & 0 & 0 & .08 & - & - \\ 0 & 0 & 0 & 0 & .21 & - \\ 0 & 0 & 0 & 0 & .02 & 0 \end{bmatrix} \end{array} \quad \begin{array}{c} \begin{matrix} R_1 & L_3 & L_2 & L_1 \end{matrix} \\ \begin{bmatrix} .30 & - & - & - \\ 0 & .51 & - & - \\ 0 & 0 & 0 & - \\ 0 & 0 & 0 & .21 \end{bmatrix} \end{array}$$

Figure 13. Transition probability matrices for Pathfinder (`dynproc_kernel`) application, comparing Kepler (left) and Maxwell (right) versions.

Transition Probability. Transition probabilities represent frequencies of an edge to a vertex, or branches to code regions, which describes the application in a way that gets misconstrued in a flat profile. A stochastic matrix could also facilitate in eliminating dead code, where states with 0 transition probabilities represent node regions that will never be visited. Kernels employing structures like loops and control flow increase the complexity analysis, and knowledge of transition probabilities of kernels could help during code generation.

A canonical adjacency matrix M represents a graph G such that every diagonal entry of M is filled with the label of the corresponding node and every off-diagonal entry is filled with the label of the corresponding edge, or zero if no edge exists Yan and Han (2002). The adjacency matrix describes the transition from N_i to N_j. If the probability of moving from i to j in one time step is $Pr(j|i) = m_{i,j}$, the adjacency matrix is given by $m_{i,j}$ as the i^{th} row and the j^{th} column element. Since the total transition probability from a state i to all other states must be 1, this matrix is a right stochastic matrix, so that $\sum_j P_{i,j} = 1$.

Figure 13 illustrates transition probability matrices for a kernel from the Pathfinder application (Tab. 12, bottom-rt.), comparing Kepler (left) and Maxwell (right) versions. Note that the Pascal version was the same as Maxwell,

as evident in Fig. 16, lower-right, and was left out intentionally. The entries of the transition probability matrix were calculated by normalizing over the total number of observations for each observed node transition i to j. Although the matrices differ in size, observe that a majority of the transitions take place in the upper-left triangle, with a few transitions in the bottom-right, for all matrices. The task is to match graphs of arbitrary sizes based on its transition probability matrix and instruction operations executed, among other information.

Hybrid Static and Dynamic Analysis. We statically collect instruction mixes and source code locations from generated code and map the instruction mixes to the source locator activity as the program is being run Lim et al. (2015). The static analysis of CUDA binaries produces an objdump file, which provides assembly information, including instructions, program counter offsets, and source line information. The CFG structure is stored in iGraph format Csardi and Nepusz (n.d.). We attribute the static analysis from the objdump file to the profiles collected from the source code activity to provide runtime characterization of the GPU as it is being executed on the architecture. This mapping of static and dynamic profiles provides a rich understanding of the behavior of the kernel application with respect to the underlying architecture.

Methodology

Based on the kernel CFG and transition probability analysis, the core of the `CUDAflow` methodology focuses on the problem of subgraph matching. In order to perform subgraph matching, we first scale the matrices to the same size by taking for graphs G_1 and G_2 the maximal proper submatrix, constructed by $\mathcal{B}(G_i) = \max(|V_1|, |V_2|)$ for a given $G_i = \min(|V_1|, |V_2|)$ using spline interpolation. The similarities in the shapes of the control flow graphs, the variants generated for

Table 9. Distance measures considered in this paper.

Abbrev	Name	Result
Euc	Euclidean	$\sqrt{\sum_{i=1}^{n} \lvert x_i - y_i \rvert^2}$
Iso	IsoRank	$(\mathbf{I} - \alpha\mathbf{Q} \times \mathbf{P})\mathbf{x}$
Man	Manhattan	$\sum_{i=1}^{n} \lvert x_i - y_i \rvert$
Min	Minkowski	$\sqrt[p]{\sum_{i=1}^{n} \lvert x_i - y_i \rvert^p}$
Jac	Jaccard	$\frac{\sum_{i=1}^{n} (x_i - y_i)^2}{\sum_{i=1}^{n} x_i^2 + \sum_{i=1}^{n} y_i^2 - \sum_{i=1}^{n} x_i y_i}$
Cos	Cosine	$1 - \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \sqrt{\sum_{i=1}^{n} y_i^2}}$

each GPU (Table 12) and the activity regions in the transition probability matrices (Fig. 13) provided motivation for this approach. In our case, the dense hotspots in the transition matrix should align with their counterparts if the matrices are similar enough.

Bilinear Interpolation. To scale the transition matrix before performing the pairwise comparison, we employ a spline interpolation procedure. Spline interpolation is general form of linear interpolation for functions of n-order polynomial, such as bilinear and cubic. For instance, a spline on a two-order polynomial performs bilinear interpolation on a rectilinear 2D grid (e.g. x and y) Gonzales and Woods (1993). The idea is to perform linear interpolation in both the vertical and horizontal directions. Interpolation works by using known data to estimate values at unknown points. Refer to Appendix B for the derivation of bilinear interpolation.

Pairwise Comparison. Once the matrix is interpolated, the affinity scores (S_1 and S_2 for graphs G_1' and G_2', respectively) are matched via a distance measure, which includes the Euclidean distance, the IsoRank solution Singh et al. (2007), Manhattan distance, Minkowski metric, Jaccard similarity, and Cosine similarity. The distance measures considered in this work are listed in Table 9.

By definition, $\text{sim}(G_i, G_j) = 0$ when $i = j$, with the similarity measure placing progressively higher scores for objects that are further apart.

CFG Results

Applications. The Rodinia and SHOC application suite are a class of GPU applications that cover a wide range of computational patterns typically seen in parallel computing. Table 10 describes the applications used in this experiment along with source code statistics, including the number of kernel functions, the number of associated files and the total lines of code.

Rodinia. Rodinia is a benchmark suite for heterogeneous computing which includes applications and kernels that target multi-core CPU and GPU platforms Che et al. (2009). Rodinia covers a wide range of parallel communication patterns, synchronization techniques, and power consumption, and has led to architectural insights such as memory-bandwidth limitations and the consequent importance of data layout.

SHOC Benchmark Suite. The Scalable HeterOgeneous Computing (SHOC) application suite is a collection of benchmark programs testing the performance and stability of systems using computing devices with non-traditional architectures for general purpose computing Danalis et al. (2010). SHOC provides implementations for CUDA, OpenCL, and Intel MIC, and supports both sequential and MPI-parallel execution.

Analysis

To illustrate our new methodology, we analyzed the SHOC and Rodinia applications at different granularities.

Application Level. Figure 14 projects goodness as a function of efficiency, which displays the similarities and differences of the benchmark

Table 10. Description of SHOC (top) and Rodinia (bottom) benchmarks studied.

	Name	Ker	File	Ln	Description
SHOC	FFT	9	4	970	Forward and reverse 1D fast Fourier transform.
	MD	2	2	717	Compute the Lennard-Jones potential from molecular dynamics.
	MD5Hash	1	1	720	Computate many small MD5 digests, heavily dependent on bitwise operations.
	Reduction	2	5	785	Reduction operation on an array of single or double precision floating point values.
	Scan	6	6	1035	Scan (parallel prefix sum) on an array of single or double precision floating point values.
	SPMV	8	2	830	Sparse matrix-vector multiplcation.
	Stencil2D	2	12	1487	A 9-point stencil operation applied to a 2D dataset.
Rodinia	Backprop	2	7	945	Trains weights of connecting nodes on a layered neural network.
	BFS	2	3	971	Breadth-first search, a common graph traversal.
	Gaussian	2	1	1564	Gaussian elimination for a system of linear equations.
	Heartwall	1	4	6017	Tracks changing shape of walls of a mouse heart over a sequence of ultrasound images.
	Hotspot	1	1	1199	Estimate processor temperature based on floor plan and simulated power measurements.
	Nearest Neighbor	1	2	385	Finds k-nearest neighbors from unstructured data set using Euclidean distance.
	Needleman-Wunsch	2	3	1878	Global optimization method for DNA sequence alignment.
	Particle Filter	4	2	7211	Estimate location of target object given noisy measurements in a Bayesian framework.
	Pathfinder	1	1	707	Scan (parallel prefix sum) on an array of single or double precision floating point values.
	SRAD v1	6	12	3691	Diffusion method for ultrasonic and radar imaging applications based on PDEs.
	SRAD v2	2	3	2021	...

applications. The size of bubble represents the number of operations executed, whereas the shade represents the GPU type. Efficiency describes how gainfully employed the GPU floating-point units remained, or FLOPs per second:

$$efficiency \quad = \quad \frac{op_{fp} + op_{int} + op_{simd} + op_{conv}}{time_{exec}} \cdot calls_n \tag{4.1}$$

The *goodness* metric describes the intensity of the floating-point and memory operation arithmetic intensity:

$$goodness \quad = \quad \sum_{j \in J} op_j \cdot calls_n \tag{4.2}$$

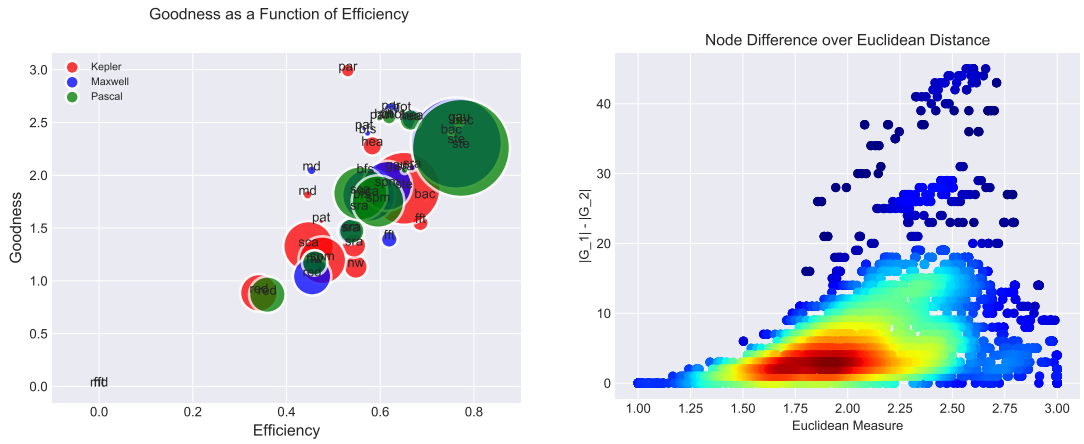

Figure 14. Left: The *static* goodness metric (Eq. B.2) is positively correlated with the *dynamic* efficiency metric (Eq. B.1). The color represents the architecture and the size of bubbles represents the number of operations. Right: Differences in vertices between two graphs, as a function of Euclidean metric for all GPU kernel combinations. Color represents intensity.

Note that efficiency is measured via runtime, whereas goodness is measured statically. Figure 14 (left) shows a positive correlation between the two measures, where the efficiency of an application increases along with its goodness. Static metrics, such as *goodness*, can be used to derive dynamic behavior of an application. This figure also demonstrates that merely counting the number of executed operations is not sufficient to characterize applications because operation counts do not fully reveal control flow, which is a source of bottlenecks in large-scale programs.

CFG Subgraph Matching.

Distribution of Matched Pairs. Figure 14 (right) projects the distribution of differences in vertices $|V|$ for all 162 CFG kernel pairs (Table 10, 2nd col. + 3 GPUs) as a function of the Euclidean measure (application, architecture, kernel), with shade representing the frequency of the score. Note that most matched CFGs had a similarity score of 1.5 to 2.2 and had size differences

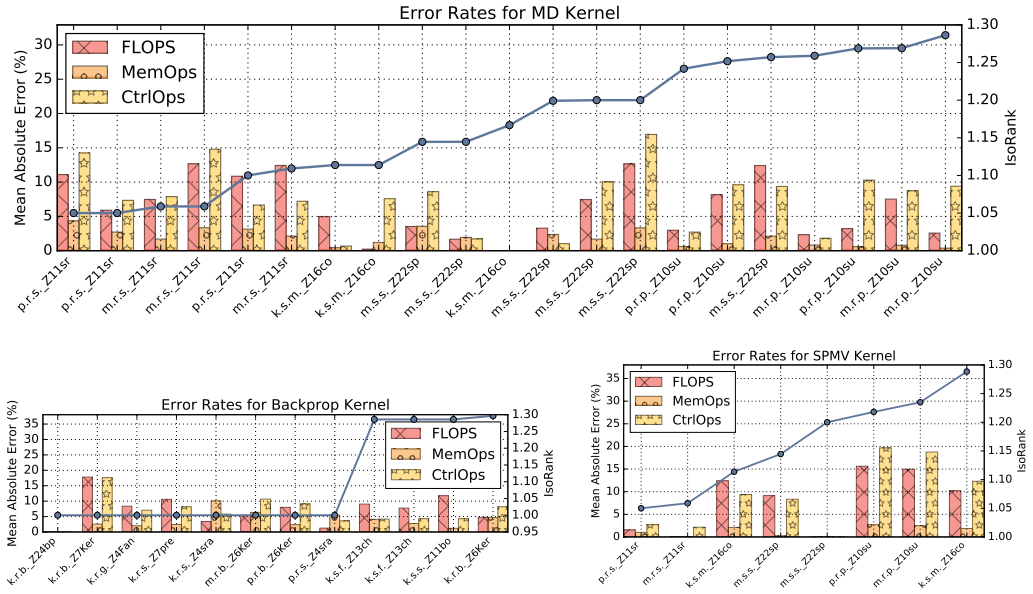

Figure 15. Error rates when estimating instruction mixes statically from runtime observations for selected matched kernels (x-axis), with IsoRank scores near 1.30.

under 10 vertices. Figure 14 (right) also shows that as the differences in vertices increase, similarity matching becomes degraded due to the loss of quality when interpolating missing information, which is expected. Another observation is that strong similarity results when node differences of the matched kernel pairs were at a minimum, between 0 and 8 nodes.

Error Rates from Instruction Mixes. Here, we wanted to see how far off our instruction mix estimations were from our matched subgraphs. Figure 15 displays instruction mix estimation error rates, calculated using mean squared error, for MD, Backprop, and SPMV kernels as a function of matched kernels (x-axis) with IsoRank scores between 1.00 to 1.30. Naming convention for each kernel is as follows: ⟨*gpu_arch.suite.app.kernel*⟩. In general, `CUDAflow` is able to provide subgraph matching for arbitrary kernels through the IsoRank score in addition to instruction mixes within a 8% margin of error. Note that since relative dynamic

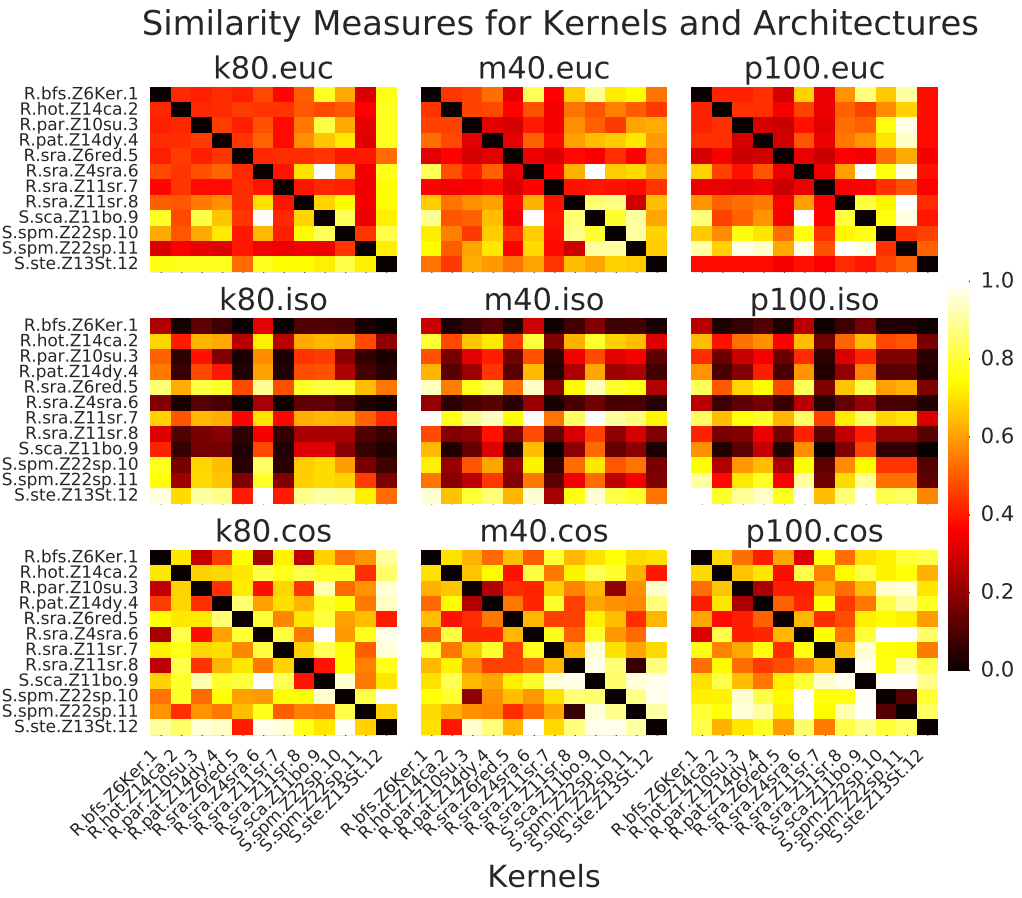

Figure 16. Similarity measures for Euclidean, IsoRank and Cosine distances for 12 arbitarily selected kernels.

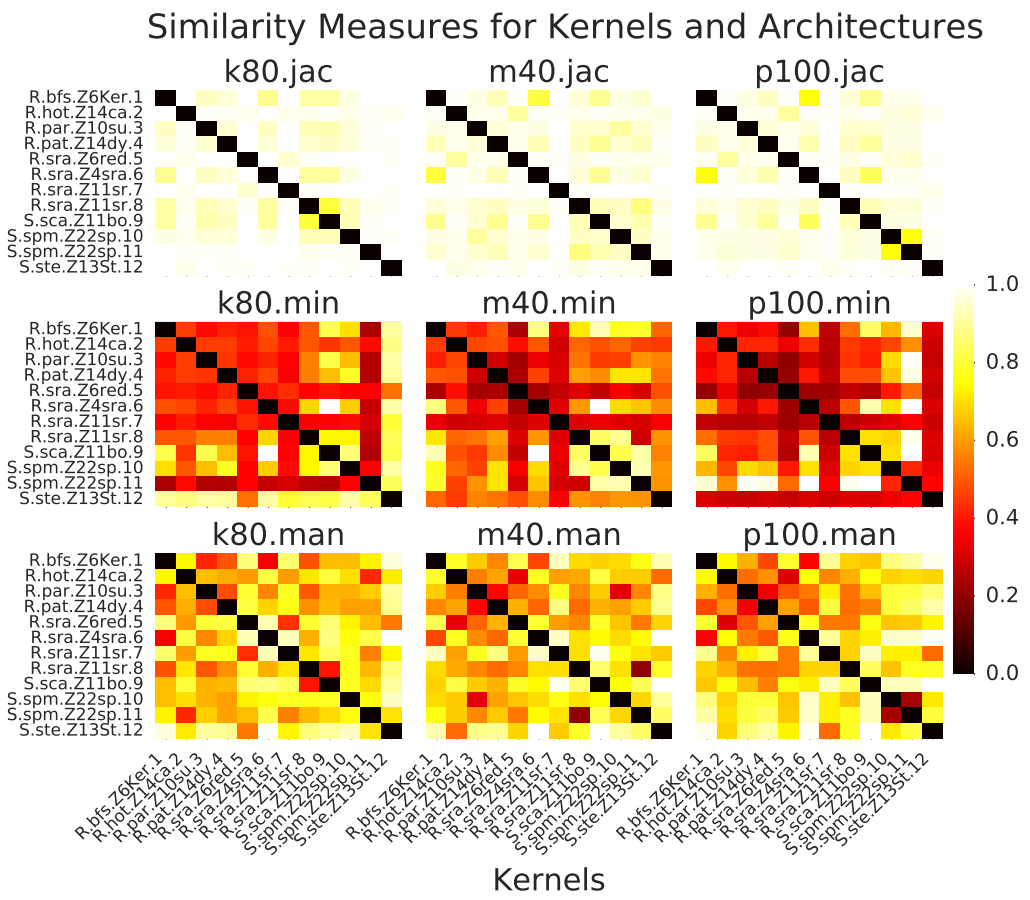

Figure 17. Similarity measures for Jaccard, Minkowski and Manhattan distances for 12 arbitarily selected kernels.

performance is being estimated from static information, the error rates will always be high.

Pairwise Matching of Kernels. Figure 16 shows pairwise comparisons for 12 arbitrary selected kernels, comparing Euclidean (top), IsoRank (middle), and Cosine distance (bottom) matching strategies, and GPU architectures (rows). Figure 17 shows comparisons for the Jaccard measure, Minkowski, and Manhattan distances for the same 12 kernels. Note that the distance scores were scaled to 0 and 1, where 0 indicates strong similarity and 1 denotes weak similarity. In general, all similarity measures, with the exeception of IsoRank, is able to match

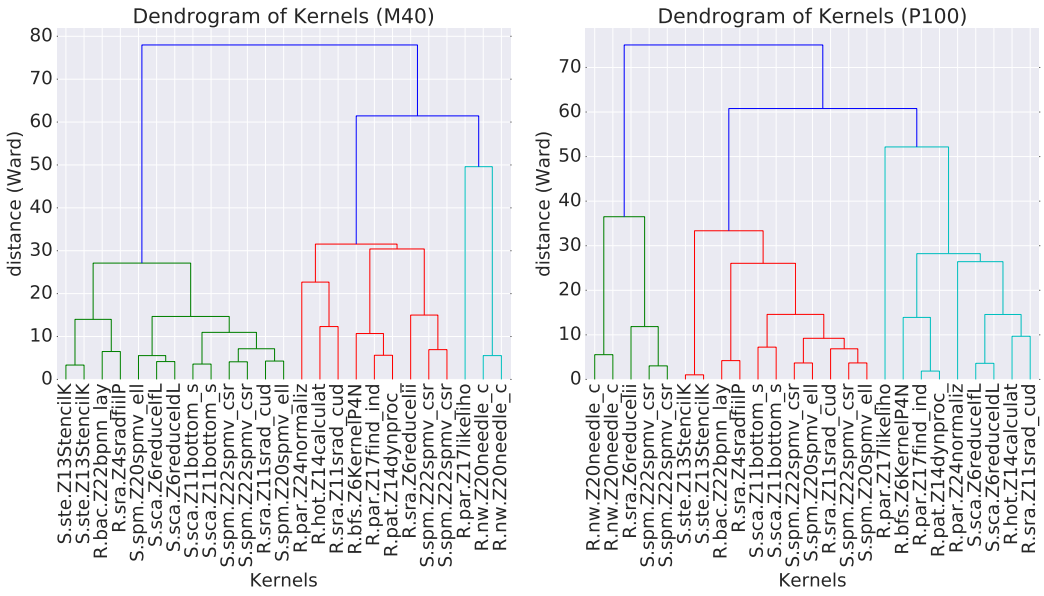

Figure 18. Dendrogram of clusters for 26 kernels, comparing Maxwell (left) and Pascal (right) GPUs.

against itself, as evident in the dark diagonal entries in the plots. However, this demonstrates that using similarity measures in isolation alone is not sufficient for performing subgraph matching for CUDA kernels.

Clustering of Kernels. We wanted to identify classes of kernels, based on characteristics such as instruction mixes, graph structures and distance measures. The Ward variance minimization algorithm minimizes the total within-cluster variance by finding a pair of clusters that leads to a minimum increase in a weighted squared distances. The initial cluster distances in Ward's minimum variance method is defined as the squared Euclidean distance between points: $d_{ij} = d(\{X_i\}, \{X_j\}) = ||X_i - X_j||^2$. Figure 18 shows a dendrogram of clusters for 26 kernels calculated with Ward's method all matched with Rodinia Particlefilter `sum_kernel`, comparing the Maxwell (left) and Pascal (right) GPUs, which both have 4 edges and 2 vertices in their CFGs. `sum_kernel` performs a scan operation and is slightly memory intensive (∼26% on GPUs). As shown, our tool is able to

categorize kernels by grouping features, such as instruction mixes, graph structures, and distance measures that show strong similarity. This figure also demonstrates that different clusters can be formed on different GPUs for the same kernel, where the hardware architecture may result in different cluster of kernel classes.

Finally, we wanted to see if our technique could identify the same kernels running on a different GPU. Figure 19 shows distance measures when comparing three kernels across three GPUs, for a total of 9 comparisons, whereas Figure 20 shows pairwise comparisons for the same three kernels across 3 GPUs, for a total of 27 comparisons (x-axis), considering pairwise comparisons in both directions (e.g. $\text{sim}(G_1, G_2)$ and $\text{sim}(G_2, G_1)$). Figure 19 displays patches of dark regions in distance measures corresponding to the same kernel when compared across different GPUs. As shown in Figure 20, our tool not only is able to group the same kernel that was executed on different GPUs, as evident in the three general categories of clusters, but also kernels that exhibited similar characteristics when running on a particular architecture, such as instructions executed, graph structures, and distance measures.

Discussion.

These metrics can be used both for guiding manual optimizations and by compilers or autotuners. For example, human optimization effort can focus on the code fragments that are ranked high by kernel impact, but low by the goodness metric. An autotuner can also use metrics such as the goodness metric to explore the space of optimization parameters more efficiently, such as by excluding cases where we can predict a low value of the goodness metric without having to execute and time the actual generated code. A benefit to end users (not included in paper, due to space purposes) would be providing the ability to compare an

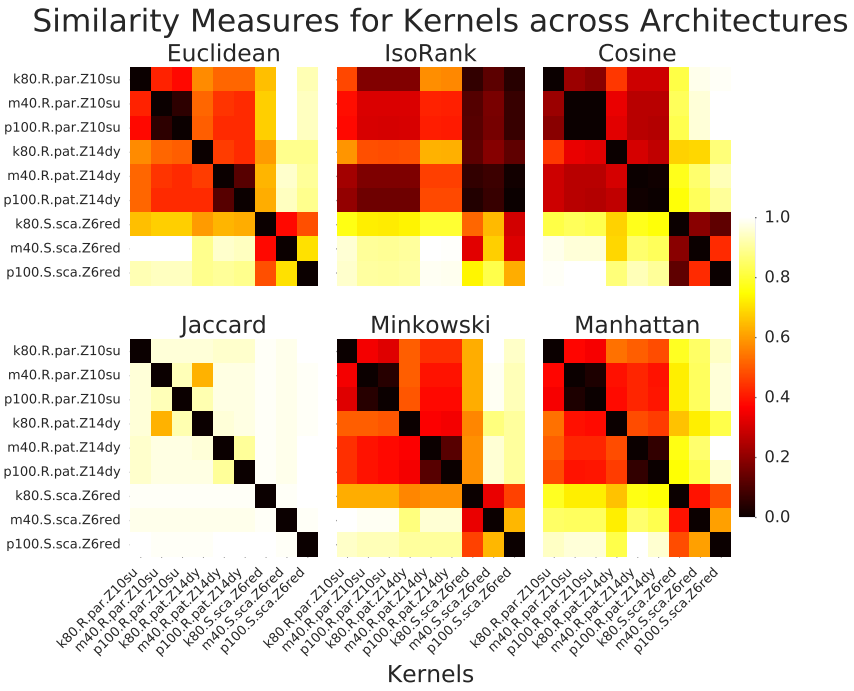

Figure 19. Dendrogram of clusters for pairwise comparison for 3 kernels across 3 GPUs (9 total).

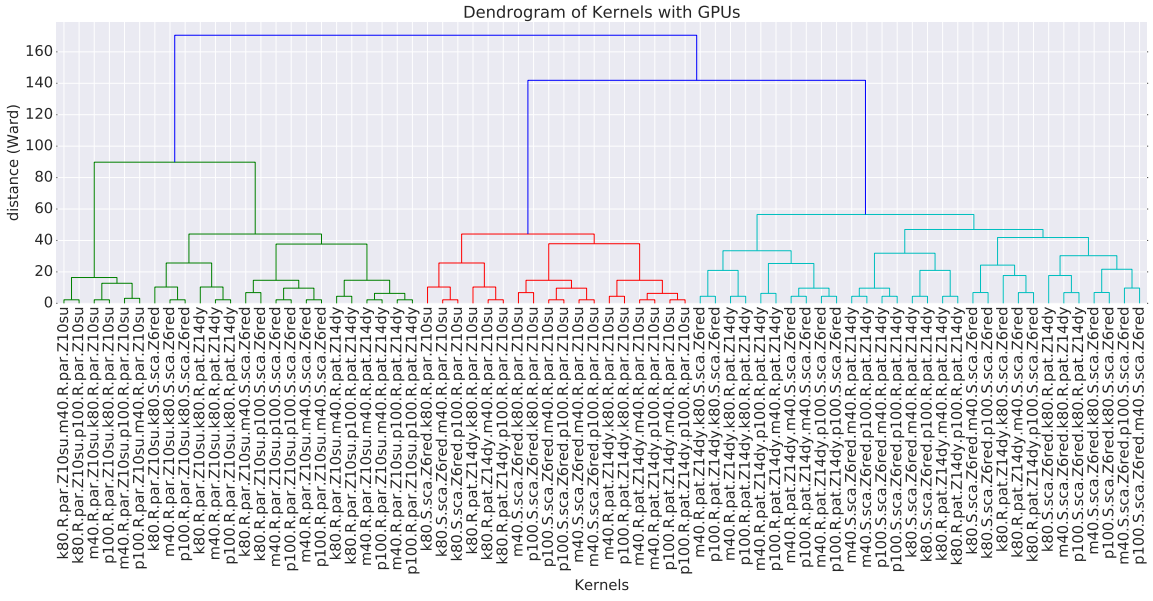

Figure 20. Dendrogram of clusters for pairwise comparison for 3 kernels across 3 GPUs (27 total).

implementation against a highly optimized kernel. By making use of subgraph matching strategy as well as instruction operations executed, `CUDAflow` is able to provide a mechanism to characterize unseen kernels.

We have presented `CUDAflow`, a control-flow-based methodology for analyzing the performance of CUDA applications. We combined static binary analysis with dynamic profiling to produce a set of metrics that not only characterizes the kernel by its computation requirements (memory or compute bound), but also provides detailed insights into application performance. Specifically, we provide an intuitive visualization and metrics display, and correlate performance hotspots with source line and file information, effectively guiding the end user to locations of interest and revealing potentially effective optimizations by identifying similarities of new implementations to known, autotuned computations through subgraph matching. We implemented this new methodology and demonstrated its capabilities on SHOC and Rodinia applications.

Future work includes incorporating memory reuse distance statistics of a kernel to characterize and help optimize the memory subsystem and compute/memory overlaps on the GPU. In addition, we want to generate robust models that will discover optimal block and thread sizes for CUDA kernels for specific input sizes without executing the application Lim et al. (2017). Last, we are in the process of developing an online web portal cknowledge (n.d.); Sreepathi et al. (2014) that will archive a collection of control flow graphs for all known GPU applications. For instance, the web portal would be able to make on-the-fly comparisons across various hardware resources, as well as other GPU kernels, without burdening the end user with hardware requirements or software

package installations, and will enable more feature rich capabilities when reporting performance metrics.

Conclusion

This chapter developed pattern matching techniques that captured runtime information of a GPU program that could be used by compilers and autotuners for optimization, which eliminates unnecessary experimentation runs. The next chapter is in the domain where the optimization landscape is vast. In particular we survey how to tune hyper-parameters for neural networks for a machine translation system. We identify which hyper-parameters matter most, in terms of systems execution performance, and what the cost of tuning that hyper-parameter is.

CHAPTER V

OPTIMIZING HYPER-PARAMETERS FOR NEURAL NETWORKS

This chapter includes previously published co-authored material from a presentation at the 2nd Workshop on Naval Applications of Machine Learning Lim, Heafield, Hoang, Briers, and Malony (2018). This work was performed while I was an intern at The Alan Turing Institute. Dr. Kenneth Heafield supervised me and provided the initial problem to solve. Dr. Hieu Hoang assisted in software install issues, and preprocessing of the datasets, as well as answering questions related to machine translation. Dr. Mark Briers partly supervised me as an intern. Dr. Allen Malony assisted in editing the paper.

Abstract

Neural machine translation (NMT) has been accelerated by deep learning neural networks over statistical-based approaches, due to the plethora and programmability of commodity heterogeneous computing architectures such as FPGAs and GPUs and the massive amount of training corpuses generated from news outlets, government agencies and social media. Training a learning classifier for neural networks entails tuning hyper-parameters that would yield the best performance. Unfortunately, the number of parameters for machine translation include discrete categories as well as continuous options, which makes for a combinatorial explosive problem. This research explores optimizing hyper-parameters when training deep learning neural networks for machine translation. Specifically, our work investigates training a language model with Marian NMT. Results compare NMT under various hyper-parameter settings across a variety of modern GPU architecture generations in single node and multi-node settings,

revealing insights on which hyper-parameters matter most in terms of performance, such as words processed per second, convergence rates, and translation accuracy, and provides insights on how to best achieve high-performing NMT systems.

Motivation

The rapid adoption of neural network (NN) based approaches to machine translation (MT) has been attributed to the massive amounts of datasets, the affordability of high-performing commodity computers, and the accelerated progress in fields such as image recognition, computational systems biology and unmanned vehicles. Research activity in NN-based machine translation has been taking place since the 1990s, but statistical machine translation (SMT) soared along with the successes of machine learning. SMT incorporates a rule-based, data driven approach, and includes language models such as word based (n-grams), phrased-based, syntax-based and hierarchical based approaches. Neural machine translation (NMT), on the other hand, does not require predefined rules, but learns lingusitic rules from statistical models, sequences and occurences from large corpuses. Models trained using NNs produce even higher accuracy than existing SMT approaches, but training time can take anywhere from days to weeks to complete. Suboptimal strategies are often difficult to find, given the dimensionality and its effect on parameter exploration.

One of the main difficulties of training neural networks is the millions of parameters that need to be estimated. These parameters are estimated by optimization methods, such as stochastic gradient descent, where the solver seeks to identify the global optima. Due to the combinatorial search space, local optimization in many cases is sufficent to generalize beyond the training

set Goodfellow, Bengio, and Courville (2016) (Ch. 8). Thus, the tuning of hyper-parameters is paramount in accelerating training of neural networks.

In neural machine translation, modeling and training are crucial in achieving high performing systems. A combination of hyper-parameter optimization methods to train a NMT system is investigated in this work. Specifically, this work examines the stability of different optimization parameters in discovering local minima, and how a combination of hyper-parameters can lead to faster convergence.

The following contributions are made in this work:

- We identify which hyper-parameters matter most in contributing to the learning trajectory of NMT systems.
- We analyze our findings for translation performance, training stability, convergence speed, and tuning cost.
- We tie in systems execution performance with hyper-parameters.

Related Work

Hyper-parameter optimization has been an unsolved problem since the inception of machine learning, and becomes even more crucial in training the millions of parameters in neural networks. The past work has investigated techniques for hyper-parameter tuning and search strategies, such as Bergstra, et. al., concluding that random search outperforms grid search Bergstra, Bardenet, Bengio, and Kégl (2011). Likewise, the authors in Shahriari, Swersky, Wang, Adams, and De Freitas (2016); Snoek, Larochelle, and Adams (2012) take a Bayesian approach toward parameter estimation and optimization. However, these efforts apply their strategies on image classfication tasks.

In relation to NMT, Britz, et. al. massively analyze neural network architectures and its variants Britz, Goldie, Luong, and Le (2017). Their approach incorporates a 2-layer bidirectional encoder/decoder with a multiplicative attention mechanism as a baseline architecture, with a 512-unit GRU and a dropout of 0.2 probability. Their model parameters remained fixed and the studies varied the architecture, including depth layer, unidirectional vs bidirectional encoder/decoder, attention mechanism size, and beam search strategies. Likewise, Bahar et. al. compare various optimization strategies for NMT by switching to a different optimizer after $10k$ iterations, and found that Adam combined with other optimizers, such as SGD or annealing, increased the BLEU score by 2.4 Bahar, Alkhouli, Peter, Brix, and Ney (2017). However, these approaches study a standard NMT system. In addition, Wu, et. al. Y. Wu et al. (2016) utilized the combination of Adam and SGD, where Adam ran for a fixed number of iterations with a 0.0002 learning rate, and switched to SGD with a 0.5 learning decay rate to slow down training, but did not perform hyper-parameter optimization.

To the best of our knowledge, there has not been any work comparing different hyper-parameter optimization strategies for NMT. Moreover, our optimization strategies are demonstrated on a production-ready NMT system and explores parameter selection tradeoffs, in terms of performance and stability.

Background

Machine translation involves model design and model training. In general, learning algorithms are viewed as a combination of selecting a *model criterion*, defined as a family of functions, *training*, defined as parameterization, and a procedure for appropriately optimizing this criterion. The next subsections

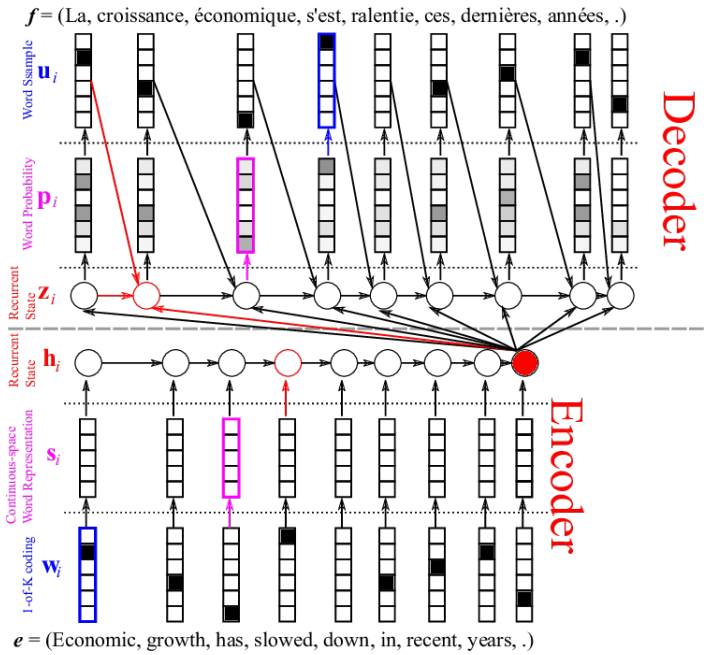

Figure 21. RNN encoder-decoder, illustrating a sentence translation from English to French. The architecture includes a word embedding space, a 1-of-K coding and a recurrent state on both ends.[1]

discuss how sentences are represented with a neural network and the optimization objectives used for training a model for a translation system.

Machine Translation. This subsection discusses how neural networks can model language translation from a source to a target sequence.

Recurrent Neural Networks. Recurrent neural networks (RNN) are typically employed for neural machine translation because of its ability to handle variable length sequences. RNNs capture unbounded context dependencies typical in natural language comprehension and speech recognition systems. For inputs x_t and y_t, connection weight matrices $\mathbf{W}_{ih}$, $\mathbf{W}_{hh}$, $\mathbf{W}_{ho}$, indicating input-to-hidden, hidden-to-hidden and hidden-to-output, respectively, and activation function f, the

[1] https://devblogs.nvidia.com/introduction-neural-machine-translation-gpus-part-2/

recurrent neural network can be described as follows:

$$h_t = f_H(\mathbf{W}_{ih}x_t + \mathbf{W}_{hh}h_{t-1}) \tag{5.1}$$

$$y_t = f_O(\mathbf{W}_{ho}h_t). \tag{5.2}$$

RNNs learn a probability distribution over a sequence by being trained to predict the next symbol in a sequence. The output at each timestep t is the conditional probability distribution $p(x_t|x_{t-1}, ..., x_1)$.

RNN Encoder-Decoder. A RNN encoder-decoder (pictured in Fig. 21) encodes a variable-length sequence into a fixed vector representation, and decodes the fixed vector representation into a variable-length sequence Cho et al. (2014). The RNN encoder-decoder are two separate neural networks that are jointly trained to maximize the conditional log-likelihood, defined as

$$\arg\max_{\theta} \frac{1}{N} \sum_{n=1}^{N} \log p_\theta(t_n|s_n), \tag{5.3}$$

where θ represents the set of model parameters, each $\mathbf{s}_n, \mathbf{t}_n$ is a pair of input and output sequences from a parallel text corpus training set, and the output of the decoder from the encoder is differentiable. A trained RNN encoder-decoder can generate a target sequence given an input sequence.

Neural Machine Translation. Neural machine translation is defined as maximizing the conditional probability, $\arg\max_t p(\mathbf{t}|\mathbf{s}) \propto p(\mathbf{s}|\mathbf{t})p(\mathbf{t})$, for a source $\mathbf{s}$ and target $\mathbf{t}$ sequence, where $p(\mathbf{s}|\mathbf{t})$ represents the translation model, and $p(\mathbf{t})$ represents the language model Bahdanau, Cho, and Bengio (2014); Sutskever, Vinyals, and Le (2014). Taking the log linear of $p(\mathbf{t}|\mathbf{s})$ yields,

$$\log p(\mathbf{t}|\mathbf{s}) = \sum_{n=1}^{N} w_n t_n(\mathbf{t}, \mathbf{s}) + \log\lambda(\mathbf{s}), \tag{5.4}$$

Table 11. Stochastic gradient descent and its variants.

Optimizer	Operations	Description
`SGD`	$\mathbf{g}_t \leftarrow \bigtriangledown_{\theta_t} J(\theta_t)$ $\theta_{t+1} \leftarrow \theta_t - \eta \mathbf{g}_t$	$\mathbf{g}_t$ - gradient cost function, η - learning rate, θ parameters
`AdaGrad`	$\mathbf{g}_t \leftarrow \bigtriangledown_{\theta_t} J(\theta_t)$ $\eta_t \leftarrow \eta_{t-1} + \mathbf{g}_t^2$ $\theta_{t+1} \leftarrow \theta_t - \frac{\eta}{\sqrt{\eta_t}+\epsilon}\mathbf{g}_t$	Divides η by previous gradients, handles sparse data well
`Adam`	$\mathbf{g}_t \leftarrow \bigtriangledown_{\theta_t} J(\theta_t)$ $\eta_t \leftarrow \gamma\eta_{t-1} + (1-\gamma)\mathbf{g}_t^2$ $\hat{\eta} \leftarrow \frac{\eta_t}{1-\gamma^t}$ $\mathbf{m}_t \leftarrow \mu\mathbf{m}_{t-1} + (1-\mu)\mathbf{g}_t$ $\hat{\mathbf{m}} \leftarrow \frac{\mathbf{m}_t}{1-\mu^t}$ $\theta_{t+1} \leftarrow \theta_t - \frac{\eta}{\sqrt{\hat{\eta}_t}+\epsilon}\hat{\mathbf{m}}_t$	$\mathbf{m}_t$ - decay mean of past gradients, $\hat{m}_t, \hat{n}_t$ - biased corrected terms that avoids zero initialization, $\gamma = 0.9$, $\mu = 0.999$, $\epsilon = 10^8$

where t_n and w_n are the n^{th} feature and weight, and $\lambda(s)$ is a normalization constant. The BLEU score provides a measure for optimizing weights during training.

Optimization Objectives. The following subsections describe the tuning of hyper-parameters that affect the performance of training a NMT system. In particular, this work focuses on the optimizers, activation functions, and dropout.

SGD Optimizers. Stochastic gradient descent (*SGD*), commonly used to train neural networks, updates a set of parameters θ, where η is the learning rate and $\mathbf{g}_t$ represents the gradient cost function, $J(\cdot)$. *Adagrad* is an adaptive-based gradient method, where η is divided by the square of all previous gradients, η_t, plus ϵ, a smoothing term to avoid dividing by zero. As a result, larger gradients have less frequent updates, whereas smaller gradients have more frequent updates. Adagrad handles sparse data well and does not require manual

Table 12. Activation units for RNN.

Activation	Operations	Description
`tanh`	$\mathbf{s}_t \leftarrow (e^{\mathbf{x}} - e^{-\mathbf{x}})/(e^{\mathbf{x}} + e^{-\mathbf{x}})$	hyperbolic tangent
`LSTM`	$\mathbf{i} \leftarrow \sigma(\mathbf{x}_t\mathbf{U}^i + \mathbf{s}_{t-1}\mathbf{W}^i)$ $\mathbf{f} \leftarrow \sigma(\mathbf{x}_t\mathbf{U}^f + \mathbf{s}_{t-1}\mathbf{W}^f)$ $\mathbf{o} \leftarrow \sigma(\mathbf{x}_t\mathbf{U}^o + \mathbf{s}_{t-1}\mathbf{W}^o)$ $\mathbf{g} \leftarrow \tanh(\mathbf{x}_t\mathbf{U}^g + \mathbf{s}_{t-1}\mathbf{W}^g)$ $\mathbf{c}_t \leftarrow \mathbf{c}_{t-1} \circ \mathbf{f} + \mathbf{g} \circ \mathbf{i}$ $\mathbf{s}_t \leftarrow \tanh(\mathbf{c}_t) \circ \mathbf{o}$	3 gates, c - internal memory, o output, 2 tanh
`GRU`	$\mathbf{z} \leftarrow \sigma(\mathbf{x}_t\mathbf{U}^z + \mathbf{s}_{t-1}\mathbf{W}^z)$ $\mathbf{r} \leftarrow \sigma(\mathbf{x}_t\mathbf{U}^r + \mathbf{s}_{t-1}\mathbf{W}^r)$ $\mathbf{h} \leftarrow \tanh(\mathbf{x}_t\mathbf{U}^h + (\mathbf{s}_{t-1} \circ \mathbf{r})\mathbf{W}^h)$ $\mathbf{s}_t \leftarrow (1 - z) \circ \mathbf{h} + \mathbf{z} \circ \mathbf{s}_{t-1}$	2 gates, no internal memory, no output gates, 1 tanh

tuning of η. Adaptive moment estimation (*Adam*) accumulates the decaying mean of past gradients, $\mathbf{m}_t$, and the decaying average of past squared gradients, η_t, referred to as the first and second moments, respectively. The moments, $\hat{m}_t, \hat{n}_t$ are biased corrected terms that avoids initializing to zero. γ is usually set to 0.9, with $\mu = 0.999$, and $\epsilon = 10^8$. Table 11 displays SGD, AdaGrad and Adam optimizers.

Activation Functions. Activation functions serve as logic gates for recurrent neural networks that computes the hidden states, and include the hyperbolic tangent, long short term memory (LSTM) Hochreiter and Schmidhuber (1997), and gated recurrent unit (GRU) Cho et al. (2014). Table 12 displays the hyperbolic tangent, LSTM and GRU activation functions.

To address the vanishing gradients problem associated with learning long-term dependencies in RNNs, LSTMs and GRUs employ a gating mechanism when computing the hidden states. For *LSTM*s, note that the input i, forget f and hidden h gates are the same equations except with different parameter matrices. g is a hidden state, based on the current input and previous hidden state. c_t serves as the internal memory, which is a combination of the previous memory, c_{t-1},

Table 13. Dropout versus a standard update function.

Optimizer	Operations	Description
`Update`	$z_i^{(l+1)} \leftarrow \mathbf{w}_i^{(l+1)}\mathbf{y}^l + b_i^{(l+1)}$ $y_i^{(l+1)} \leftarrow f(z_i^{(l+1)})$	Standard update
`Dropout`	$r_j^{(l)} \sim \text{Bernoulli}(p)$ $\hat{\mathbf{y}}^{(l)} \leftarrow \mathbf{r}^{(l)} * \mathbf{y}^{(l)}$ $z_i^{(l+1)} \leftarrow \mathbf{w}_i^{(l+1)}\hat{\mathbf{y}}^l + b_i^{(l+1)}$ $y_i^{(l+1)} \leftarrow f(z_i^{(l+1)})$	r - Bernoulli rv, p - dropout param

multiplied by the input gate. The hidden state, s_t, is calculated by multiplying c_t and the output gate. On the other hand, a *GRU* employs a reset gate r and an update gate u. The reset gate r determines how to combine the new input with the previous memory, whereas the update gate u defines how much of the previous memory to retain. If the reset gates were set to 1's and the update gates to 0's, this would result in a vanilla RNN.

The differences between the two approaches to compute hidden units are that GRUs have 2 gates, whereas LSTMs have 3 gates. GRUs do not have an internal memory and output gates, compared with LSTM which uses c as its internal memory and o as an output gate. The GRU input and forget gates are coupled by an update gate z, and the reset gate r is applied directly to the previous hidden state. Also, GRUs do not have a 2nd non-linearity operation, compared to LSTMs, which uses two hyperbolic tangents.

Dropout. In a fully-connected, feed-forward neural network, *dropout* randomly retains connections within hidden layers while discarding others Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov (2014). Table 13 displays a standard hidden update function on the top, whereas a version that decides whether to retain a connection is displayed on the bottom. $\hat{\mathbf{y}}^{(l)}$ is the

thinned output layer, and retaining a network connection is decided by a Bernoulli random variable $r^{(l)}$ with probability $p(\cdot) = 1$.

Combination of Optimizers. Since the learning trajectory significantly affects the training process, it is required to select and tune the proper types of hyper-parameters to yield good performance. The construction of the RNN cell with activation functions, the optimizer and its learning rate, and the dropout rates all have an affect on how the training progresses, and whether good accuracy can be achieved.

Marian NMT

Marian Junczys-Dowmunt et al. (2018) is an efficient NMT framework written in C++, with support for multi-node and multi-GPU training and CPU/GPU translation capabilities. Marian is currently being developed and deployed by the Microsoft Translator team. Table 14 displays parameters involved with tuning a neural machine translation system, categorized by model, training and validation, with values and types in brackets, and its default value, if any. The types of models in Marian include RNNs and Transformers Vaswani et al. (2017).

The translation system evaluated in this study is a sequence-to-sequence model with single layer RNNs for both the encoder and decoder. The RNN in the encoder is bi-directional and the decoder is sequence-to-sequence. Depth, also referred to as *deep transitions* Koehn (2017), is achieved by stacking activation blocks, resulting in tall RNN cells for every recurrent step. The encoder consists of four activation blocks per cell, whereas the decoder consists of eight activation blocks, with an attention mechanism placed between the first and second block. Word embedding sizes were set at 512, the RNN state size was set to 1024, and

layer normalization was applied inside the activation blocks and the attention mechanism.

Experiments

The experiments were carried out on the WMT 2016 Junczys-Dowmunt and Grundkiewicz (2016) translation tasks for the Romanian and German languages in four directions: EN $\rightarrow$ RO, RO $\rightarrow$ EN, EN $\rightarrow$ DE, and DE $\rightarrow$ EN. The datasets and its characteristics used in the experiments are listed in Table 15, with number of sentence examples in parenthesis. Table 15 shows that for WMT 2016 EN $\rightarrow$ RO and RO $\rightarrow$ EN, the training data consisted of 2.6M English and Romanian sentence pairs, whereas for WMT 2016 EN $\rightarrow$ DE and DE $\rightarrow$ EN, the training corpus consisted of approximately 4.5M German and English sentence pairs. Validation was performed on 1000 sentences of the `newsdev2016` corpus for RO, and on the `newstest2014` corpus for DE. The `newstest2016` corpus consisted of 1999 sentences for RO and 2999 sentences for DE, and was used as the test set. We evaluated and saved the models every 10K iterations and stopped training after 500K iterations.

All experiments used bilingual data without additional monolingual data. We used the joint byte precision encoding (BPE) approach Sennrich, Haddow, and Birch (2015) in both the source and target sets, which converts words to a sequence of subwords. For all four tasks, the number of joint-BPE operations were 20K. All words were projected on a 512-dimensional embedding space, with vocabulary dimensions of 66000 $\times$ 50000. The mini-batch size was determined automatically based on the sentence length that was able to fit in GPU global memory, set at 13000 MB for each GPU.

Table 14. Marian hyper-parameters, with options in brackets.

Model	dimensions	vocab [vect]
		embed [int]
	RNN	dim [int]
		type [bi-dir, bi-unidir, s2s]
		cell: type [gru, lstm, tanh], depth [1, 2, ...],
		transition cells [1, 2, ...]
		skip [bool]
		layer norm [bool]
		tied embeddings [src, trg, all]
		dropout [float]
	transformer	heads [int]
		no projection
		tied layers [vector]
		guided alignment layer
		preproc, postproc, post-emb [dr, add, norm]
		dropout [float]
Training	cost	[ce-mean, ce-mean, words, ce-sum, perplexity]
	after-epochs	[∞]
	max length	[int=50]
	system	GPUs, threads
	mini-batch	size, words, fit, fit-step [int, int, bool, uint]
	optimizer	[sgd, adgrad, adam]
	learn rate	decay: strategy [epochs, stalled, epoch + batches,
		ep+stalled], start, frequency, repeat warmup, inverse
		sqrt, warmup
	label smoothing	[bool]
	clip norm	[float=1]
	exponential	[float=0]
	smoothing	
	guided alignment	cost [ce, mean, mult], weight [float=0.1]
	data weighting	type [sentence, word]
	embedding	vectors, norm, fix-src, fix-trg
Validation	frequency	
	metrics	[ce, ce-words, perplexity, valid-script, translation,
		bleu, bleu-detok]
	early stopping	[int=10]
	beam size	[int=12]
	normalize	[float=0]
	max-length-factor	[float=3]
	word penalty	[float]
	mini-batch	[int=32]
	max length	[int=1000]

Table 15. Datasets used in experiments.

	RO→EN, EN→RO	DE→EN, EN→DE
Train	corpus.bpe (2603030)	corpus.bpe (4497879)
Valid	newsdev2016.bpe (1999)	newstest2014.bpe (3003)
Test	newstest2016.bpe (1999)	newstest2016.bpe (2999)

Beam search was used for decoding, with the beam size set to 12. The translation portion consisted of recasing and detokenizing the translated BPE chunks. The trained models compared different hyper-parameter strategies, including the type of optimizer, the activation function, and the amount of dropout applied, as discussed in Section V. The number of parameters were initialized with the same random seed. The systems were evaluated using the case-sensitive BLEU score computed by Moses SMT Koehn et al. (2007).

We compared models trained on two different types of GPUs (P100 Pascal, V100 Volta), listed on Table 16. The corresponding CPUs are listed on Table 17. Each ran with four GPUs. The dataset was partitioned across 4 GPUs, and a copy of the model was executed on each GPU.

Analysis

This section analyzes the results of the evaluated NMT systems in terms of translation quality, training stability, convergence speed and tuning cost.

Translation Quality. Table 18 shows BLEU scores calculated for four translation directions for the validation sets (top) and the test sets (bottom), comparing learning rates, activation functions and GPUs. Note that entries with `n/a` means that no results were available, whereas entries with `dnf` indicates training time that did not complete within 24 hours. For the validation sets, LSTMs were able to achieve higher accuracy rates, whereas in the test set GRUs and LSTMs were about the same. Also, note that the best performing learning

Table 16. Graphical processors used in this experiment.

	P100	V100
CUDA capability	6.0	7.0
Global memory (MB)	16276	16152
Multiprocessors (MP)	56	80
CUDA cores per MP	64	64
CUDA cores	3584	5120
GPU clock rate (MHz)	405	1380
Memory clock rate (MHz)	715	877
L2 cache size (MB)	4.194	6.291
Constant memory (bytes)	65536	65536
Shared mem blk (bytes)	49152	49152
Registers per block	65536	65536
Warp size	32	32
Max threads per MP	2048	2048
Max threads per block	1024	1024
CPU (Intel)	Ivy Bridge	Haswell
Architecture family	Pascal	Volta

rates were usually at a lower value (e.g. 1e-3). The type of hidden unit mechanism (e.g LSTM vs GRU) and the learning rate can affect the overall accuracy achieved, as demonstrated by Table 18.

Table 19 displays various dropout rates applied for two translation directions RO $\rightarrow$ EN and DE $\rightarrow$ EN, comparing hidden units, GPUs and overall training time. The learning rate was evaluated at 0.001, the rate that achieved the highest

Table 17. Hardware and execution environment information.

Architecture	Haswell	Ivy Bridge
Model	E5-2698 v3	Xeon X5650
Clock speed	2.30 GHz	2.67 GHz
Node count	4, 14	6
GPUs	4 × V100	4 × P100
Memory	256 GB	50 GB
Linux kernel	3.10.0-229.14.1	2.6.32-642.4.2
Compiler	CUDA v9.0.67	
Flags	{'g', 'lineinfo', 'arch=sm_cc'}	

Table 18. BLEU scores for validation (top) and test (bottom) datasets.

		ro→en		en→ro		de→en		en→de	
cell	learn-rt	P100	V100	P100	V100	P100	V100	P100	V100
GRU	1e-3	35.53	35.43	19.19	19.28	28.00	27.84	20.43	20.61
	5e-3	34.37	34.05	19.07	19.16	26.05	22.16	n/a	19.01
	1e-4	35.47	35.46	19.45	19.49	27.37	27.81	dnf	21.41
LSTM	1e-3	34.27	35.61	19.29	19.64	28.62	28.83	21.70	21.69
	5e-3	35.05	34.99	19.48	19.43	n/a	24.36	18.53	18.01
	1e-4	35.41	35.28	19.43	19.48	n/a	28.50	dnf	dnf
GRU	1e-3	34.22	34.17	19.42	19.43	33.03	32.55	26.55	26.85
	5e-3	33.13	32.74	19.31	18.97	31.04	26.76	n/a	26.02
	1e-4	33.67	34.44	18.98	19.69	33.15	33.12	dnf	28.43
LSTM	1e-3	33.10	33.95	19.56	19.08	33.10	33.89	28.79	28.84
	5e-3	33.10	33.52	19.13	19.51	n/a	29.16	24.12	24.12
	1e-4	33.29	32.92	19.14	19.23	n/a	33.44	dnf	dnf

Table 19. Dropout rates, BLEU scores and total training time for test set, comparing systems.

		ro→en				de→en			
cell	dropout	P100	t	V100	t	P100	t	V100	t
GRU	0.0	34.47	6:29	34.47	4:43	32.29	9:48	31.61	6:15
	0.2	35.53	8:48	35.43	6:21	33.03	18:47	32.55	19:40
	0.3	35.36	12.21	35.15	7:28	31.36	10:14	31.50	9:33
	0.5	34.50	12:20	34.67	17:18	29.64	11:09	30.21	11:09
LSTM	0.0	34.84	6:29	34.65	4:46	32.84	12:17	32.88	7:37
	0.2	34.27	8:10	35.61	6:34	33.10	16:33	33.89	13:39
	0.3	35.67	9:56	35.37	11:29	33.45	20.02	33.51	15:51
	0.5	34.50	15:13	34.33	12:45	32.67	20:02	32.20	13:03

BLEU score as evident in Table 18. Generally speaking, increasing the dropout rates also increased training time. This may be the result of losing network connections when applying the dropout mechanism, but at the added benefit of avoiding overfitting. This is evident in Table 19, where applying some form of dropout will result in a trained model achieving higher accuracies. The best performance can be seen when the dropout rate was set at 0.2 to 0.3. This confirms that some form of skip connection mechanism is necessary to prevent the overfitting of models under training.

Figure 23 shows BLEU score results as a function of training time, comparing GPUs, activation units, learning rates and translation directions. Note that in most cases a learning rate of 0.001 achieves the higher accuracy in most cases, at the cost of higher training time. Also, note the correlation between longer training time and higher BLEU scores in most cases. In some cases, the models were able to converge at a faster rate (e.g. Fig. 23 upper left, RO→EN, GRU with learning rate of 0.005 vs 0.001).

Training Stability. Figure 24 shows the cross-entropy scores for the RO → EN and EN → RO translation tasks, comparing different activation functions (GRU vs. LSTM), with learning rates at 0.001. Note the training stability patterns that emerge from this plot, which is highly correlated with the translation direction. The activation function (GRU vs LSTM) during validation also performed similarly across GPUs and was also highly correlated with the translation direction. Cross-entropy scores for the EN → RO translation direction were more or less the same. However, for RO → EN, a LSTM that executed on a P100 converged the earliest by one iteration.

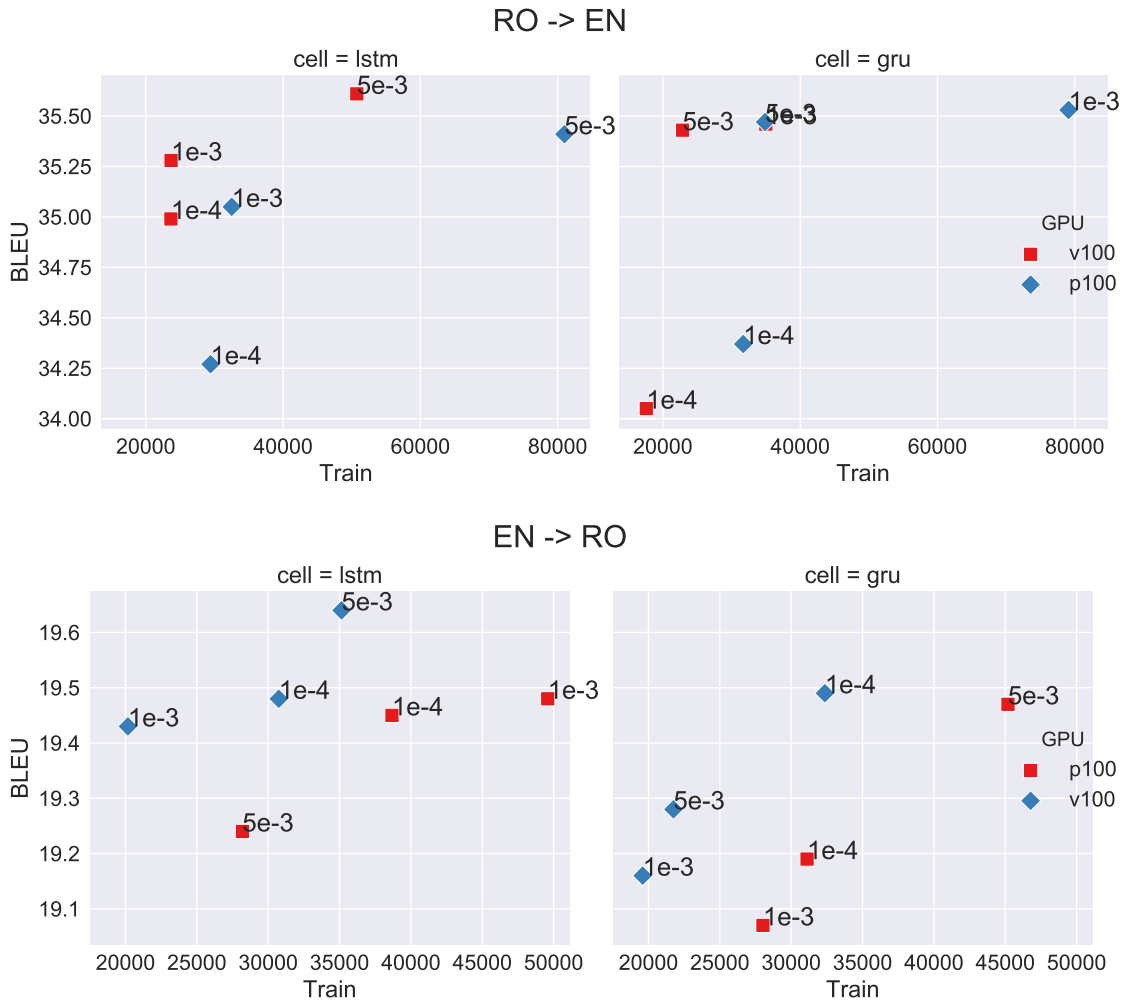

Figure 22. BLEU scores as a function of training time (seconds), comparing GPUs (color), activation units (sub-columns), learning rates and translation directions.

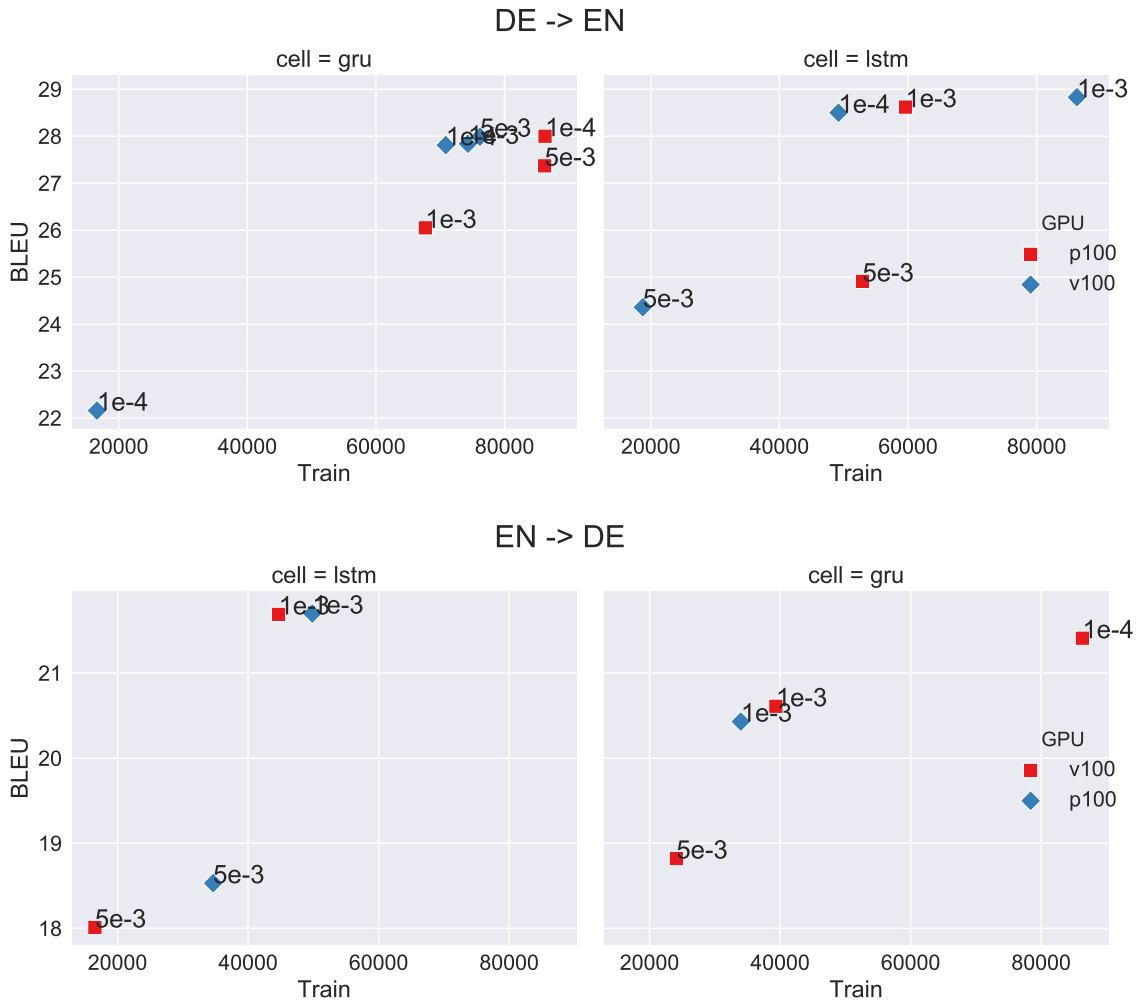

Figure 23. BLEU scores as a function of training time (seconds), comparing GPUs (color), activation units (sub-columns), learning rates and translation directions.

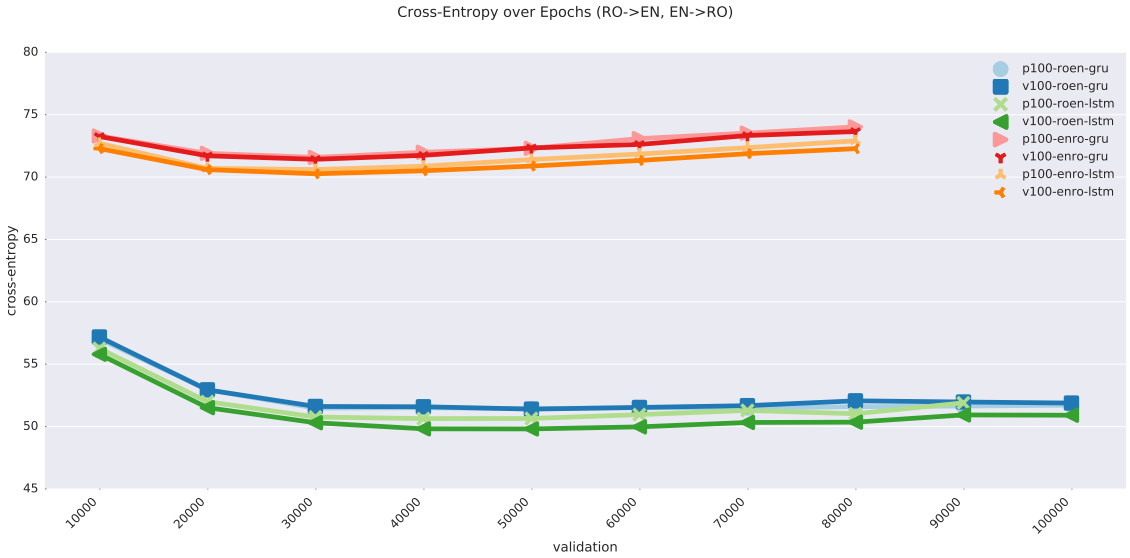

Figure 24. Cross entropy over the number of epochs for RO → EN and EN → RO, comparing activation functions and GPUs.

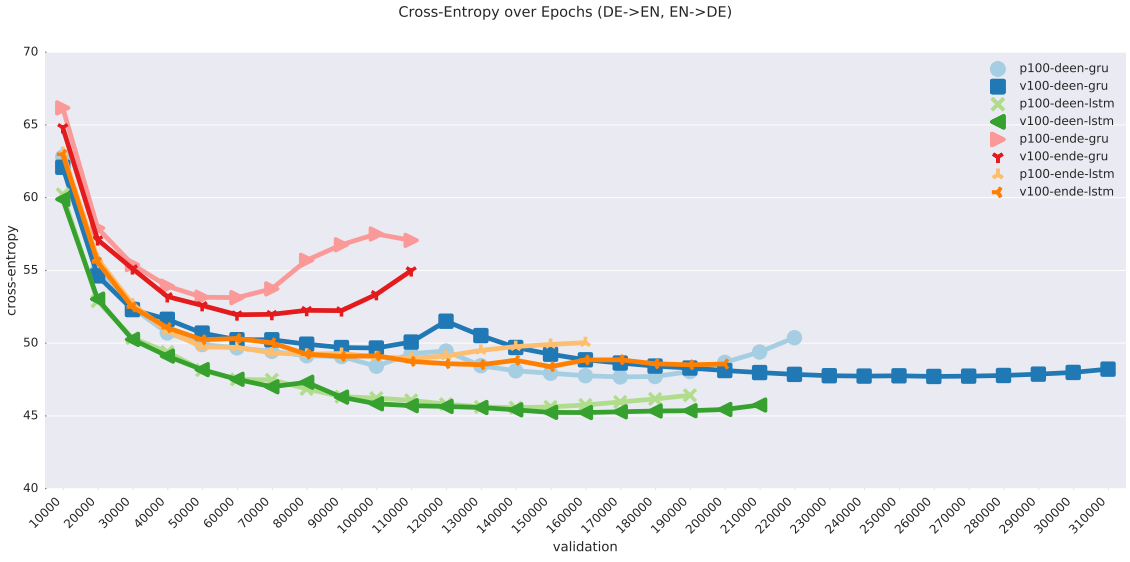

Figure 25. Cross-entropy over the number of epochs for DE → EN and EN → DE, comparing activation functions and GPUs.

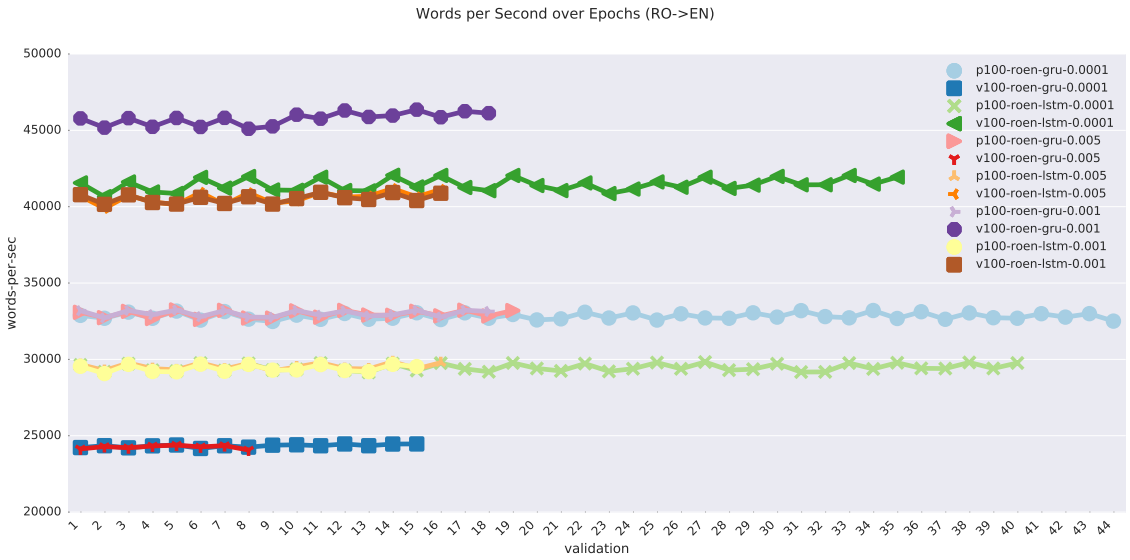

Figure 26. Average words-per-second for the RO → EN translation task, comparing systems.

Figure 25 shows the same comparison of cross-entropy scores over epochs for DE → EN and EN → DE translation tasks. Note that the behavior for this translation task was wildly different for all systems. Not only did it take more epochs to converge compared to Fig 24, but also how well the system progressed also varied, as evident in the cross-entropy scores during validation. When comparing hidden units, LSTMs outperformed GRUs in all cases. When comparing GPUs, the V100 performed better than the P100 in terms of cross-entropy, but took longer to converge in some cases (e.g. v100-deen-lstm, v100-ende-lstm). Also, note that the behavior of the translation task EN → DE for a GRU hidden unit never stabilized, as evident in both the high cross-entropy scores and the peaks toward the end. The LSTM was able to achieve a better cross-entropy score overall, with nearly a 8 point difference for DE → EN, compared with the GRU.

Convergence Speed. Figure 26 shows the average words-per-second for the RO → EN translation task, comparing systems. The average words-per-second

Table 20. Words-per-second (average) and number of epochs, comparing activation units, learning rates and GPUs.

		words-per-sec		validation		words-per-sec		validation	
cell	learn-rt	P100	V100	P100	V100	P100	V100	P100	V100
		ro→en				en→ro			
GRU	1e-3	33009.23	45762.54	18000	18000	29969.14	42746.15	15000	15000
	5e-3	32965.23	24253.14	19000	8000	30223.89	23144.62	17000	10000
	1e-4	32828.61	24341.96	44000	16000	29959.34	23277.51	25000	14000
LSTM	1e-3	29412.87	40534.06	15000	16000	27282.54	38131.13	14000	14000
	5e-3	29536.65	40598.24	16000	16000	27245.42	37384.46	19000	21000
	1e-4	29478.51	41441.37	40000	35000	27002.60	38118.79	25000	25000
		de→en				en→de			
GRU	1e-3	28279.53	38026.87	20000	28000	28367.91	39995.48	10000	10000
	5e-3	28215.40	19819.59	25000	4000	n/a	39944.10	n/a	16000
	1e-4	28367.54	33218.70	26000	32000	dnf	39993.89	dnf	36000
LSTM	1e-3	24995.64	33507.31	16000	17000	25245.67	35122.54	13000	17000
	5e-3	25210.15	33740.92	14000	7000	25049.21	33649.20	9000	6000
	1e-4	dnf	34529.58	dnf	31000	dnf	dnf	dnf	dnf

executed remained consistent across epochs. The system that was able to achieve the most words-per-second was v100-roen-gru-0.001, whereas the one that achieved the least words-per-second was the v100-roen-gru-0.005. Surprisingly, the best and worst performer was the v100-roen-gru, depending on its learning rate, with the sweet spot at 0.001. This confirms 0.001 as the best learn rate that can execute a decent number of words-per-second and achieve a fairly high accuracy, as evident in previous studies, across all systems.

Table 20 also displays words-per-second and validation, comparing activation units, learning rates and GPUs. When fixing learning rate, the V100 was able to execute more words-per-second than the P100, and was able to converge at an earlier iteration. When comparing hidden units, GRUs were able to execute higher words per second on a GPU and converge at a reasonable rate (at 18000 iterations) for most learning rates, except for 5e-3. When looking at LSTMs, words-per-second executed on a V100 was similar, although at a higher learning rate it was able to converge at 42000 iterations, but at the cost of longer training time and slower convergence (35000 iterations).

Table 21. Total training time for four translation directions, comparing systems.

		ro→en		en→ro		de→en		en→de	
cell	learn-rt	P100	V100	P100	V100	P100	V100	P100	V100
GRU	1e-3	8:48	6:21	7:47	5:26	18:47	19:40	9:26	6:41
	5e-3	9:41	4:52	8:38	6:02	23:57	4:36	n/a	10:56
	1e-4	21:58	9:43	12:33	8:59	23:50	21:09	dnf	23:58
LSTM	1e-3	8:10	6:34	7:49	5:36	16:33	13:39	13:50	12:24
	5e-3	9:02	6:34	10:44	8:32	n/a	5:12	9:37	4:35
	1e-4	22:29	14:05	13:46	9:45	n/a	23:57	dnf	dnf

Table 21 shows the corresponding total training time for the four translation directions, comparing GPUs, activation units, and learning rates. The dropout rate was set at 0.2, which was the best performer in most cases (Tab 19). Table 21 shows that the training time increased as the learning rates were decreased. In general, Romanian took a fraction of the time to complete training (usually under 10 hours), whereas German took 18-22 hours to complete training.

Cost of Tuning a Hyper-Parameter. Table 22 displays the average time spent per epoch for the Romanian ↔ English translation task, and Table 23 displays the average time spent per epoch for the German ↔ English translation task, comparing learning rates, activation cells, and GPUs. The mean is displayed in each cell, with the standard deviation in parenthesis and the number of epochs executed in brackets. For both tasks, dropout was set to 0.2. Surprisingly, GRUs take longer on the V100 on average with larger learning rates (5e-3, 1e-4) vs the P100, whereas for LSTMs, the V100s clearly speeds up execution per epoch. Note also that the learning rate does not have a significant change in the average time spent per epoch, except for the case with GRUs executing on the V100 with large learning rates. The learning rate does have an effect on the number of epochs executed, as seen in brackets as the learning rate increases. Table 23 reports on the German ↔ English translation tasks. The same observation can be made for

Table 22. Average time spent per iteration for RO → EN and EN → RO translation directions, comparing systems, with standard deviation in parenthesis and epochs in brackets.

		ro→en		en→ro	
cell	learn-rt	P100	V100	P100	V100
GRU	1e-3	1807.362941 (142.43) [17]	1304.076471 (102.67) [17]	1829.790714 (166.06) [14]	1278.770714 (117.63) [14]
	5e-3	1814.640556 (140.01) [18]	2472.531429 (11.16) [7]	1816.642500 (165.40) [16]	2385.243333 (15.08) [9]
	1e-4	1823.828837 (129.08) [43]	2466.306429 (11.29) [14]	1839.624583 (167.28) [24]	2369.436923 (13.79) [23]
LSTM	1e-3	2032.362857 (155.58) [14]	1470.278 (108.79) [15]	2010.199231 (146.74) [13]	1438.945385 (107.76) [13]
	5e-3	2018.048 (148.21) [15]	1469.054 (110.05) [15]	2014.716667 (144.41) [18]	1474.787500 (100.57) [20]
	1e-4	2026.976154 (147.46) [39]	1445.585882 (106.30) [34]	2037.517083 (140.28) [24]	1443.758333 (99.68) [24]

this task, where GRUs spend less time per epoch compared to LSTMs, and that the average time spent per epoch remains fixed as the learnignrate increases.

Summarize Findings

This work reveals the following, with respect to tuning hyper-parameters:

- Dropout is neccessary to avoid overfitting. The recommended probability rate is 0.2 to 0.3.
- LSTMs take longer than GRUs per epoch, but achieves better accuracy.
- Although the average time spent per epoch remains fixed as learning rates increase, the total number of epochs executed per training run increases as the learning rates increase.
- Tensor core GPUs, particularly the V100, provide more words that can be processed per second, compared to non-tensor core GPUs, such as the Pascal P100.

Table 23. Average time spent per iteration for DE → EN and EN → DE translation directions, comparing systems, with standard deviation in parenthesis and epochs in brackets.

		de→en		en→de	
cell	learn-rt	P100	V100	P100	V100
GRU	1e-3	3430.330526 (124.58) [19]	2555.738148 (95.76) [27]	3432.534444 (128.70) [9]	2535.11 (88.61) [9]
	5e-3	3450.174167 (133.13) [24]	4898.036667 (47.79) [3]	n/a	2432.112000 (87.91) [15]
	1e-4	3425.231600 (129.98) [25]	4907.070667 (51.24) [15]	n/a	2434.452000 (90.02) [35]
LSTM	1e-3	3887.889333 (164.183) [15]	2898.554375 (129.37) [16]	3840.552500 (162.85) [12]	2761.088125 (116.41) [16]
	5e-3	3855.21 (162.27) [13]	2852.335 (121.95) [6]	3859.903750 (167.48) [8]	2886.194 (122.26) [5]
	1e-4	n/a	2814.689000 (118.66) [30]	n/a	n/a

Discussion

The variation in the results, in terms of language translation, hyper-parameters, words-per-second executed and BLEU scores, in addition to the hardware the training was executed on demonstrates the complexity in learning the grammatical structure between the two languages. In particular, the learning rate set for training, the hidden unit selected for the activation function, the optimization criterion and the amount of dropout applied to the hidden connections all have a drastic effect on overall accuracy and training time. Specifically, we found that a lower learning rate achieved the best performance in terms of convergence speed and BLEU score. Also, we found that the V100 was able to execute more words-per-second than the P100 in all cases. When looking at accuracy as a whole, LSTM hidden units outperformed GRUs in all cases. Lastly, the amount of dropout applied on a network in all cases prevented the model from overfitting and achieve a higher accuracy.

The multidimensionality of hyper-parameter optimization poses a challenge in selecting the architecture design for training NN models, as illustrated by the varying degrees of behavior across systems and its performance outcome. This work investigated how the varying design decisions can affect training outcome and provides neural network designers how to best look at which parameters affect performance, whether accuracy, words processed per second, and convergence expectation. Coupled with massive datasets for parallel text corpuses and commodity heterogenous GPU architectures, the models trained were able to achieve WMT grade accuracy with the proper selection of hyper-parameter tuning.

We analyzed the performance of various hyper-parameters for training a NMT, including the optimization strategy, the learning rate, the activation cell, and the GPU across various systems for the WMT 2016 translation task in four translation directions. Results demonstrated that a proper learning rate and a minimal amount of dropout is able to prevent overfitting as well as achieve high training accuracy.

Future work includes developing optimization methods to evaluate how to best select hyper-parameters. By statically analyzing the computational graph that represents a NN in terms of instruction operations executed and resource allocation constraints, one could derive execution performance for a given dataset without running experiments.

Conclusion

This chapter addresses the following questions when training neural networks. Specifically, neural machine translation was evaluated during training for stability, convergence, speed, and cost. Questions that were addressed include how much a hyper-parameter update costed, as well as which hyper-parameters

contributed to learning. The next chapter attempts to combine techniques from the previous chapters for identifying the precision requirements for classifying image applications.

CHAPTER VI

NUMERICAL REPRESENTATION

This chapter includes both previously pubished and unpublished co-authored material. The work includes a poster presentation that was accepted at the 5th Workshop on Naval Applications of Machine Learning Lim, Castro, Coti, Jalby, and Malony (2021), and work in progress involving Dr. Camille Coti, Dr. William Jalby, Dr. Allen Malony and Dr. Pablo Oliveira that started when I was a Chateaubriand Fellow at the University of Versailles. I am the primary contributor to this work in developing the algorithm, writing the new code, and writing the paper. Dr. Coti initially identified the need for this work and provided the application that this work was performed in. Dr. Jalby supervised me while I was interning in Versailles. Dr. Allen Malony assisted in editing the paper. Dr. Oliveira introduced the foundation of numerical representation.

Abstract

This paper investigates training deep neural networks with varying precision lengths while providing the user with guidance on how to best set the precision requirements of a floating point operation. Our approach intercepts floating point operations at the LLVM intermediate representation layer and applies rounding at varying precision lengths. We demonstrate our approach with PyTorch C++ Vision models on the CalTech 101 dataset. Our results are presented in the following manner. We break down the precision requirements per iteration and overall for the training session and compare across various hyper-parameters, including learning rates, mini-batch sizes, and convolution filters. Our results demonstrate that mixed precision is stable in earlier parts of the training phase, whereas reduced precision

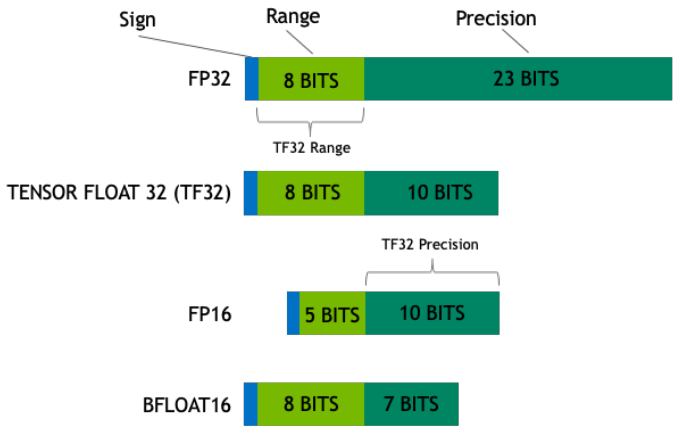

Figure 27. Comparison of floating point representations (image source TF32 (2019a)).

becomes unstable near convergence. Our approach is novel in that it ties in the network architecture and hyper-parameters with variable length floating point precision and enables exploration of precision bounds of an operation.

Motivation

Due to the lengthy amount of time it takes to train machine learning models, increasing floating-point operations per clock cycle can be attained with reduced precision operations, which trades off accuracy with instruction throughput and low latency. Since machine learning involves repeated matrix-vector operations during the forward, backward and update passes, counting multiply-add-accumulate (MAC) operations provide a way to estimate performance of a training run for a given model. Figure 27 displays a comparison of floating point representations, including `tensorfloat32`, `bfloat16`, float and half types. This motivates the discussion to investigate whether more operations can be executed per clock cycle, while maintaining accuracy and correctness of a program using reduced precision.

Modern microprocessors, accelerators and embedded devices provide hardware units for executing reduced precision operations. For instance, NVIDIA Volta and Turing GPUs have hardware capability for executing mixed precision operations since CUDA 8 Mixed Precision Programming (2016), where Volta tensor cores have FP16/FP16 and FP16/FP32 modes, and Turing tensor cores have INT8/INT32, INT4/INT32 and INT1/INT32 execution modes Tensor Core Performance (2019). Intel Cascade Lake provides vectorized intrinsics, `AVX512_VNNI`, for accelerating convolutional neural networks, which performs 8-bit multiplies (`VPMADDWD`) and 32-bit accumulates (`VPADDD`) in one clock cycle using Port 0 and Port 5 simultaneously for a theoretical 4× increase in instruction throughput Intel Cascade Lake (2019).

Floating-Point Numbers in Machine Learning

Numerical portions of machine learning include inputs, model, gradients, activation units and weights De Sa, Feldman, Ré, and Olukotun (2017). Weights and activation units, which represent signals of the gradient when computing backpropagation with SGD, are typical candidates for quantization Intel Low Precision (2018), Mixed Precision Programming (2016), Micikevicius et al. (2017). Floating point representation includes *fixed-point computation* that truncates the floating-point into a fix sized format and *custom quantization points* that varies the length of the precision and range of a floating-point number.

Mixed precision operations incur accuracy loss, depending on the method, and the solution can be improved with iterative refinement. For instance, the authors demonstrated that the inner generalized minimal residual method (GMRES) loop, an iterative method for solving a system of non-linear equations, was computed in 32-bit and the outer GMRES loop was computed in 64-bit with

Table 24. IEEE-754 Numbers and exceptions.

		Outcome	Description
Numbers	Zeroes	$+0$ and -0	Sign determines behavior when dividing by nonzero (e.g. $-\infty$ or $+\infty$)
	Infinities	$+\infty$ or $-\infty$	Div-by-zero, overflow
	NaNs	Not a Number	$(+\infty) - (+\infty)$, $0/0$, $\sqrt{-1}$
	Normal	Normalized	Most common nonzero representable reals
	Subnormal	Denormalized	Values very close to zero, issues regarding rounding errors
Exceptions	Invalid operation	NaN produced	NaN conditions, as above
	Overflow	Operation result	Number too large in magnitude to be represented in data type
	Division by zero	$\frac{x}{\pm 0}$, $x \neq 0$	Produce $\pm\infty$ depending on sign of x and ± 0
	Underflow	Result too small	Generally harmless, but error bounds will differ from normal computations
	Inexact	Real result can't be represented	Rounding (default), care needed for sound analysis

minimal loss in accuracy Baboulin et al. (2009), and more recently with GPUs in 8-bit / 32-bit mixed precision modes Haidar, Wu, Tomov, and Dongarra (2017).

Concerns relating to deep learning with limited numerical precision include overflow and underflow of values, and not-a-number (NaN) resulting from undefined operations, such as adding or subtracting infinite variables, or $\sqrt{-1}$ Goodfellow et al. (2016) (Chapter 4). Table 24 displays floating point numbers and exceptions defined by the IEEE-754 standard. A numerical value can result in zeroes, infinities, NaN, normal, or subnormal. Representing numbers in fixed-point registers require handing exceptional cases. Exceptions result in undefined behavior, either due to the resulting procedure or invalid mathematical definitions, and includes overflow, division by zero, underflow, and inexact.

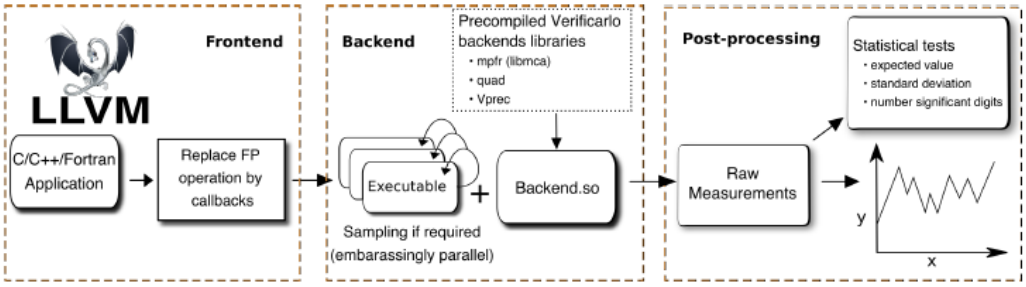

Figure 28. Verificarlo workflow.

Workflow

This subsection covers the workflow of instrumenting deep learning applications with reduced precision. Our methodology of instrumenting floating point operations at reduced precision was implemented at the LLVM intermediate representation (IR) level.

Verificarlo Modes. Verificarlo is a toolkit for assessing and reproducing floating point operations Denis, Castro, and Petit (2015). The reduced precision is emulated in the IEEE-754 format for 32-bit and 64-bit operations, where the exponent and mantissa sizes can be set at arbitrary lengths for each floating point operation or at the function level. The error bounds are where the precision is truncated and can also be set. Figure 28 illustrates the overall Verificarlo workflow. Verificarlo takes in C or C++ code, performs source-to-source translation, where the operations are replaced with its reduced floating point counterpart in the LLVM IR level, and produces a binary executable. The control flow is the same as the original program with the ability of assessing operations as they take place in the program. The code is instrumented and program results are available (both the original and the reduced FP precision) and allows further analysis to take place.

Table 25. Verificarlo backends and options.

Backend	Description	Options
IEEE	No effect on output	debug, debug-binary, print-newline, print-subnormal-normalized, no-backend-name
Monte Carlo (MCA)	MCA on variable, quad type on doubles, double type on floats	mode {ieee, mca, pb [†], rr [‡]}, precision-binary32, precision-binary64, error-mode {rel[§], abs [‖], all}, max-abs-error-exponent, daz [*], ftz [#], seed
MCA-MPFR	MCA using GNU MPFR library	same as MCA
Bitmask	First order model of noise, sets t mantissa bits to 0, 1, rand	mode {ieee, mca, pb, rr}, operator {zero, one, rand}, precision-binary32, precision-binary64, daz, ftz, seed
Cancellation	Cancels out bits arbitrarly	seed, tolerance, warning
Virtual precision	Sets length for both exponent (range) and mantissa (precision)	mode {ieee, mca, pb, rr}, precision-binary32, precision-binary64, range-binary32, range-binary64, error-mode {rel, abs, all}, daz, ftz

[*] denormals are zero, [†] precision bound, [‡] random round, [§] relative, [‖] absolute, [#] flush to zero

Table 25 lists the various backend modes supported, which include Monte Carlo arithmetic (MCA), bitmask, cancellation, and virtual precision. The backend modes can be applied at the *variable* level for inputs, outputs or both, at the *operand* level, or both the variable and operand levels. MCA utilizes quadruple types (128 bits) for double variables and double types on floating-point variables. The bitmask backend sets t mantissa bits to 0, 1 or MCA randomization. The cancellation backend automatically detects a cancellation, and if detected, noise is applied on the cancelled part with MCA. Virtual precision, the study of this work, enables setting of mantissa and exponent at arbitrary lengths, and other options can be applied such as stochastic rounding, denormalization, and flush to zero.

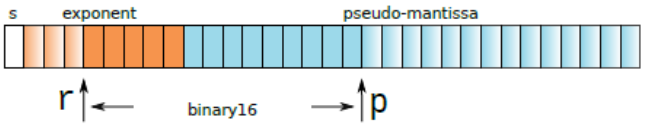

Figure 29. Virtual precision in Verificarlo, showing $r = 5$ and $p = 10$, simulating a binary16 embedded inside a binary 32.

Algorithm 2 Instrumenting functions with Verificarlo.

Data: X inputs, Y outputs, $F : X \to Y$
Result: F' instrumented
procedure FUNC_INST(F, X, Y)
    for $f^i \in$ F **do**
        Count N_{float}, N_{double}
        Allocate $N_{float} + N_{double}$ space in heap
    for $x^i \in X$ **do**
        Add x^i_{type}, x^i_{size}, x^i_{name}, $x^i_{address}$ for `vfc_enter`
    Create callback to `vfc_enter`
     Load x^i rounded values
     Call hooked function with x^i
     for $y^j \in Y$ **do**
         Add y^j_{type}, y^j_{size}, y^j_{name}, $y^j_{address}$ for `vfc_exit`
     Call `vfc_exit`
     Load y^j rounded values
     Return y^j, if needed

Figure 29 illustrates how the virtual precision mode is applied on a floating-point variable, with $r = 5$ and $p = 10$ (image source: Chatelain, Petit, de Oliveira Castro, Lartigue, and Defour (2019)).

Primitive Types from Composite Types. To instrument derived types, a mechanism to infer primitive types, such as floats and doubles, from composite types was added in Verificarlo. Algorithm 2 describes the procedure for instrumenting functions with Verificarlo, whereas Algorithm 3 describes the steps to infer floats from composite types. The routine takes as input a struct type.

Algorithm 3 Infer floating-point types from derived types in Verificarlo.

```
1: Input: Type is a Struct type
2: Output: Number of floats and doubles for derived type T
3: procedure DERIVED_TYPE(Type T)
4:     S ← <Type S>T                                   ▷ type cast to struct
5:     for ∀s ∈ S do                                   ▷ s are members of S
6:         T ← getType(s)
7:         if T_ptr then                               ▷ T is a pointer
8:             T ← getElemPointedTo(T_ptr)             ▷ get callee until primitive type
   reached
9:             if T_fl then float++                    ▷ float
10:            else if T_dbl then double++
11:            else if T_arr or T_vec then
12:                if T_dbl then double+= N_arr
13:                else if T_fl then float+= N_arr
14:            else if T_struct then
15:                DERIVED_TYPE(Type T)
16:        else if T_fl then float++
17:        else if T_dbl then double++
18:        else if T_arr or T_vec then
19:            if T_dbl then double+= N_arr
20:            else if T_fl then float+= N_arr
21:        else if T_struct then
22:            DERIVED_TYPE(Type T)
```

For each member of the struct, the type is checked. If it is a float, a double, or a pointer to such type, the address is logged and the algorithm continues until all members have been examined. Algorithm 3 takes place during Algorithm 2, specifically in Lines 4 to 6, 7 to 8, and 12 to 13.

Rounding. When each float or double is intercepted in Verificarlo, stochastic rounding is applied, which returns a modified version of that value. For instance, D. Stott Parker defines *stochastic rounding* as the probability of rounding x to $\lfloor x \rfloor$ proportional to the proximity of x to $\lfloor x \rfloor$ Parker (1997).

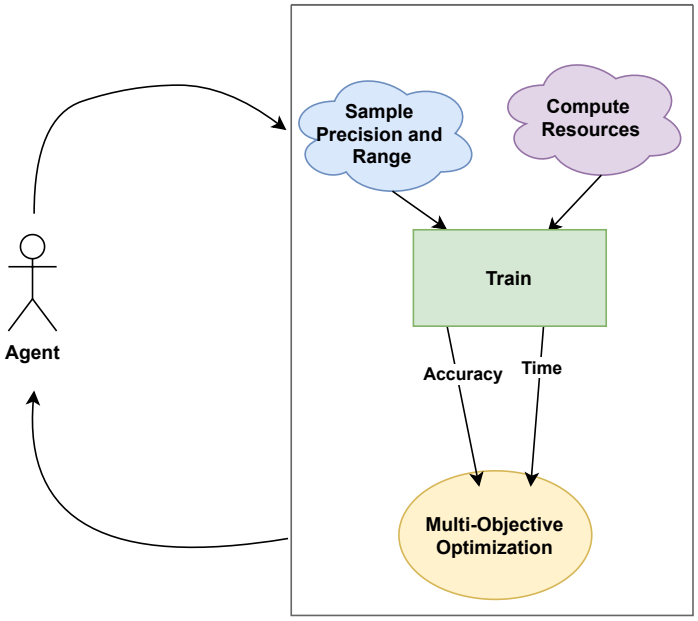

Figure 30. Search for precision and range settings during training.

$$\text{Round}(x) = \begin{cases} \lfloor x \rfloor & \text{with probability } 1 - \frac{x - \lfloor x \rfloor}{\mathcal{E}} \\ \lfloor x \rfloor + \mathcal{E} & \text{with probability } \frac{x - \lfloor x \rfloor}{\mathcal{E}}, \end{cases} \tag{6.1}$$

where $\mathcal{E} \in \mathbb{R}$ represents a random error, uniformly distributed on $(-\frac{1}{2}, \frac{1}{2})$.

Multi-Objective Optimization. We pose the problem of training deep neural networks with reduced precision as a multi-objective optimization problem that seeks to satisfy accuracy and execution speed objectives. Our approach is novel in that it performs training in real time to measure the speed to convergence, as opposed to counting operations. Our approach also provides an ability for models to dynamically set precision and range sizes during training run.

The approach is formulated as follows. Given a precision p, let $F_{acc}(p)$ denote the achieved accuracy on a training run, and let $F_{exec}(p)$ denote the measured execution speed of model m on hardware h, and T be the bounded

execution time. Multi-objective optimization is formulated as

$$\begin{aligned} &\max_{p} F_{acc}(p) \\ &\text{subject to } F_{exec}(p) \leq T \end{aligned} \tag{6.2}$$

Given this formulation, we are interested in the Pareto optimal, defined in Sec. II as a point in the criterion space that satisfies multiple objectives. In this scenario, the Pareto optimal is a model that achieves high accuracy with minimal execution time, or a model that does not decrease its accuracy while maintaining minimal execution time.

Figure 30 displays an overview of the proposed search framework, which consists of an agent that interacts with its environment in a feedback manner. The *agent* performs search by sampling the precision and range sizes of a floating point operation, evaluates the performance of the model under that setting, and updates the model parameters accordingly. The *environment* includes the training process, accounting for compute resources, that measures the time spent per epoch. For a value function $V_{\pi}(S)$, with states S following a sequence of actions $a = 1 : \pi$, where the actions represent a model's accuracy under a specified precision setting, and θ represents the learned model, the goal is to maximize the expected criterion:

$$V_{a;\theta}(s) = E[R], \tag{6.3}$$

where R is the reward function, as defined in Equation 6.2. This process of sampling, evaluating and updating is repeated until θ is reached with a desired number of steps.

Experimental Results

This section reports on instrumenting various neural network models in PyTorch C++ with Verificarlo.

Table 26. Neural networks for image classification evaluated in this study.

Properties	AlexNet	VGG	ResNet	SqueezeNet	ShuffleNet	MNASNet
Top-5	79.6	90.4	93.5	80.6	81.7	91.5
Input Size	227×227	224×224	224×224	224×224	224×224	224×224
Conv Layers	5	13	53	8(×3)+1	13(×3)	4+6(×3)
Filter Size	3, 5, 11	3	1, 3, 7	1,3,7,13	1, 3	1, 3, 5
# Channels	3-256	3-512	3-2048	3-512	24-1024	32-320
# Filters	96-384	64-512	64-2048	16-96	12-512	32-1600
FC Layers	3	3	1	1	1	1
# Channels	256-4096	512-4096	2048	512	24-1024	1280
# Filters	1K-4096	1K-4096	1000	1	12-512	1280
Weights	61M	138M	25.5M	1.24M	7.39M	6.28M
MACs	724M	15.5G	3.9G	0.35G	0.60G	0.53G

Applications and Execution Environment. CalTech 101 is a standard image dataset that includes 101 image categories, ranging from helicopter, chair to camera. There are about 40 to 800 images per category, with most categories with about 50 images.

The applications that were instrumented were PyTorch Vision image classification models, including AlexNet Krizhevsky (2014), MNASNet Tan et al. (2019), ResNet He, Zhang, Ren, and Sun (2016), ShuffleNet Ma, Zhang, Zheng, and Sun (2018), SqueezeNet Iandola et al. (2016) and VGG Simonyan and Zisserman (2014). Table 26, adapted from Sze et al. (2017), displays a summary of the neural network architectures evaluated in this study. Note the trend in compressing models, as evident in the decreased number of weights and MACs in more recent models.

We evaluated each of the models in Torchvision C++. Note that AlexNet, ResNet and VGG models follow the standard convolutional architecture setup, but varies in the number of layers, filter sizes, and channels. SqueezeNet consists of 8 *fire* modules, each of which consists of 3 convolutional layers (1×1, 3×3, 7×7, 13×13). MNASNet consists of 6 *inverted residuals*, each of which consists of 3

Table 27. Intel Xeon Platinum hardware and execution environment information.

Model	8176 (×2)	8180M (×24)
Clock Speed	2.50 GHz	2.10 GHz
Cores	28	28
Threads	56	56
TDP	165W	205W
L2	28 MiB	28 MiB
L3	38.5 MiB	38.5 MiB
Max Mem	768 GiB	1536 GiB

convolutional layers (1×1, 3×3, 5×5). ShuffleNet includes 3 *stages*, each of which consists of 3 (1×1, 3×3) convolutional layers.

The experiments ran on an Intel Skylake with a S2600WF motherboard, which consists of 24 Xeon Platinum 8180M CPUs @ 2.50 GHz, and 2 Xeon Platinum 8176 CPUs @ 2.10GHz. The memory size was 768GB RAM, EDR IB, Intel P4500 1TB NVMe SSD, and Intel P4800X 375GB NVMe Optane SSD. Table 27 lists the hardware specifications for the Xeon Platinum 8180M and 8176 CPUs.

Two sets of experiments were carried out. One set trained the neural networks for 10 epochs, and the second set trained the networks for 30 epochs. For all networks, the mini-batch size for the train set was set to 64 and the mini-batch size for the test set was set to 1000. The learning rate was set to 0.001. The data set was cross validated with a 80/20 split of train and test data. The total train data set size was 7281, and the total test size was 1865.

Results. This subsection discusses the results from this work. First, we analyze the learning trajectories with varying precision lengths. Figure 31 plots the trajectory of accuracy at each epoch, comparing vision models and the precision sizes. The top row represents inbound, or inputs, and the bottom row represents outbound. In general, lowering the precision to 1 bit compared to the

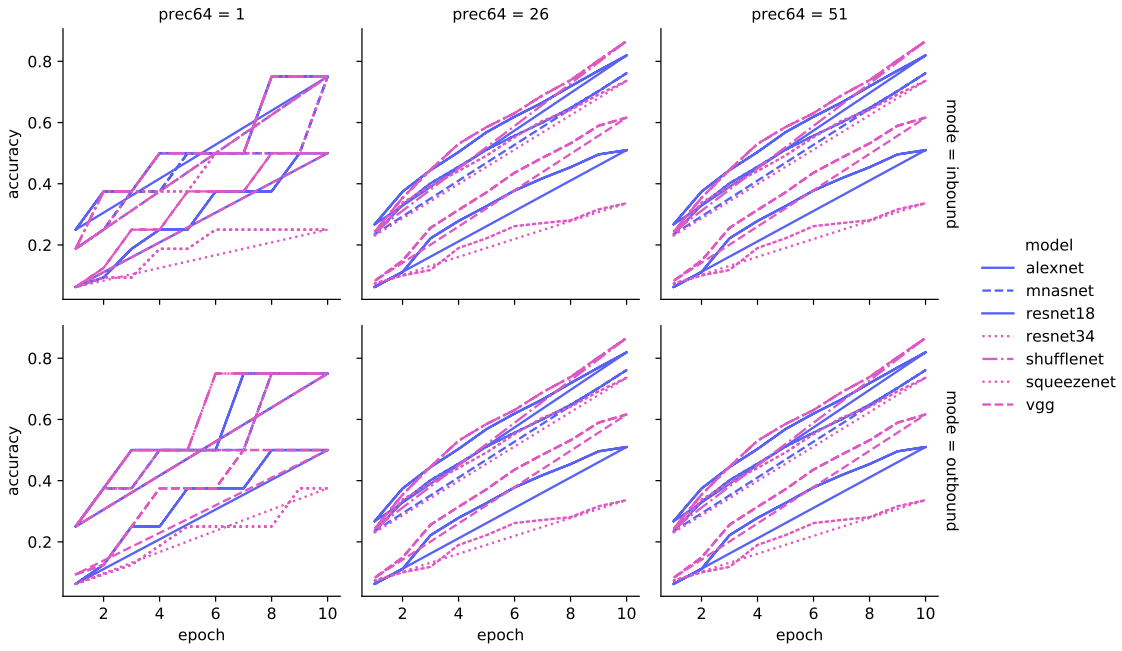

Figure 31. Accuracy per epoch, comparing vision models.

full precision run affected accuracy. What is notable is that lowering the precision to half type had the same effect as running it in full precision mode. This could be related to the recent push of training neural networks in half precision mode, which demonstrates the stability of neural networks in lower precision.

Table 28 breaks down statistics for the first ten epochs, comparing precision sizes, models, and the resulting accuracy and loss. The time reported is in seconds. Note that in some cases, the total time executed in the lower bit precision is faster than the full precision version, such as AlexNet and ResNet34. Note also that in most cases, the half precision types match the accuracy and loss of the full precision types with lower execution times.

Figure 32 plots the change in difference in values, comparing 1-bit and 26-bit mantissa with the full precision as the baseline for AlexNet, MNASNet and VGG. The same observation can be made for the half type that maintains near

Table 28. Statistics for first ten epochs of training, comparing precision sizes and models.

		Inbound			Outbound		
		Time	Loss	Accuracy	Time	Loss	Accuracy
Alexnet	1	801	0.2500	0.5000	762	0.2500	0.500
	26	744	1.9195	0.5100	797	1.9195	0.5100
	51	793	1.9195	0.5100	796	1.9195	0.5100
MNASNet	1	1650	0.1250	0.7500	1605	0.2500	0.7500
	26	1745	0.7954	0.7613	1602	0.7954	0.7613
	51	1744	0.7954	0.7613	1608	0.7954	0.7613
ResNet18	1	1119	0.1250	0.7500	1020	0.1250	0.7500
	26	990	0.6752	0.8193	1014	0.6752	0.8193
	51	1017	0.6752	0.8193	1049	0.6752	0.8193
ResNet34	1	1536	0.1250	0.7500	1405	0.1875	0.7500
	26	1457	0.9195	0.7363	1534	0.9195	0.7363
	51	1452	0.9195	0.7363	1415	0.9195	0.7363
ShuffleNet	1	1258	0.0625	0.7500	1343	0.1250	0.7500
	26	1278	0.4439	0.8662	1210	0.4439	0.8662
	51	1237	0.4439	0.8662	1309	0.4439	0.8662
SqueezeNet	1	896	0.5000	0.2500	823	0.5000	0.3750
	26	793	2.9584	0.3365	809	2.9584	0.3365
	51	828	2.9540	0.3365	798	2.9584	0.3365
VGG	1	3536	0.2500	0.5000	3667	0.2500	0.5000
	26	3424	1.4977	0.6161	3461	1.4977	0.6161
	51	3588	1.4977	0.6161	3566	1.4977	0.6161

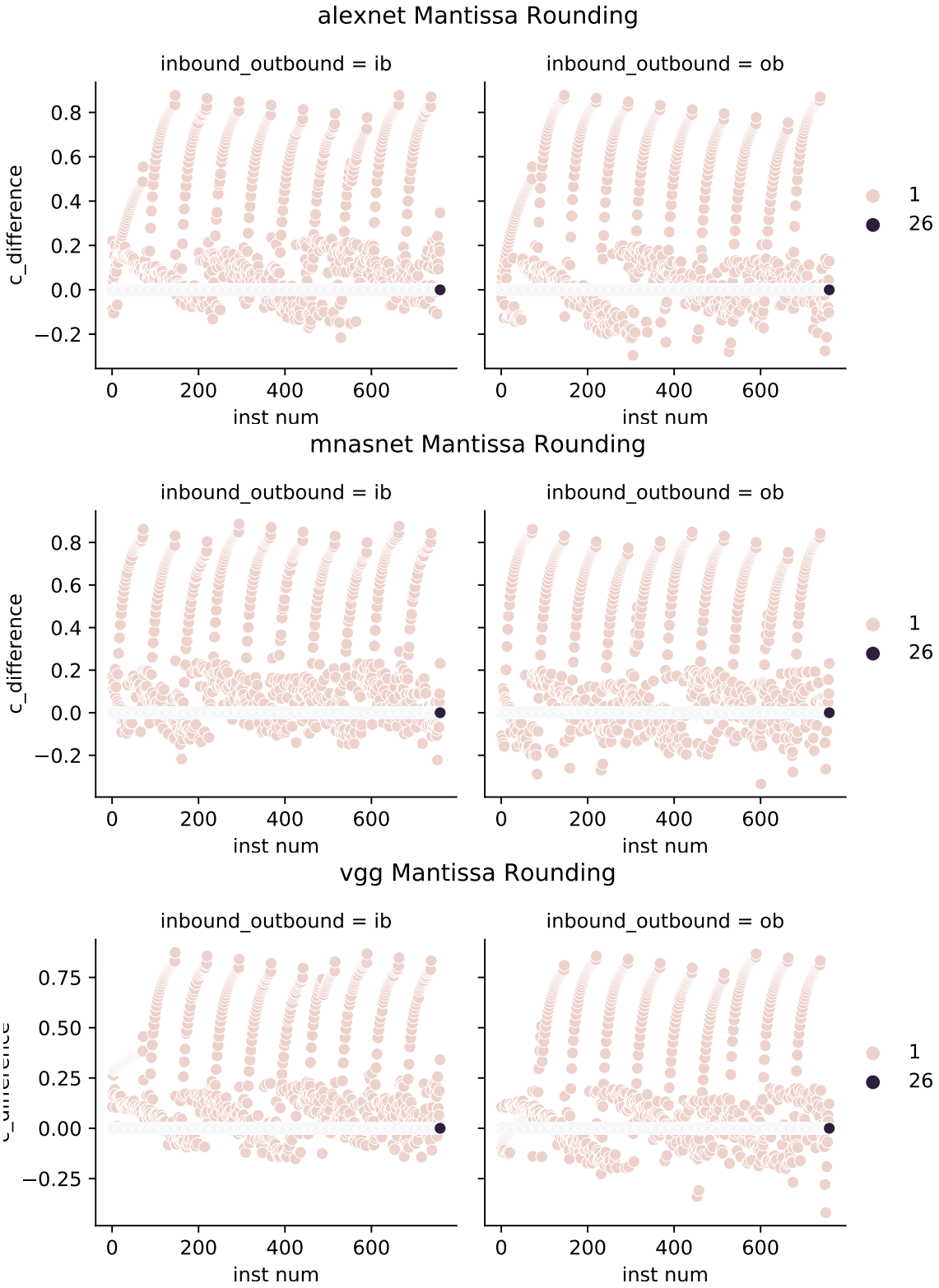

Figure 32. Rounding errors for various mantissa sizes, comparing vision models.

Table 29. Displaying multi-objective results when accounting for accuracy and average time spent per epoch.

	Accuracy	Convergence
AlexNet-1	51.0	801
AlexNet-half	51.0	744
ResNet18-1	75.0	1119
ResNet18-half	81.9	990
ResNet34-1	75.0	1536
ResNet34-half	73.6	1457

zero change in difference in the values, whereas in the 1-bit case, the change varies drastically.

Figure 33, top, displays accuracy as a function of execution time. Observe that in some networks, such as SqueezeNet, ResNet18 and ResNet34, the accuracy of 1 bit is actually on par with the full precision version. Figure 33, bottom, displays loss as a function of time. Loss is defined as $-\sum_{c=1}^{M} y_{o,c}\log(p_{o,c})$, where M is the number of classes, y is the binary indicator if class label c is the correct classification for observation o, and p is the predicted probability that observation o is of class c. In general, the loss for the 1-bit precision is lower than that of the 26-bit and the full bit precision.

Table 29 displays a summarized version of multi-objective optimization when accounting for accuracy and average time spent per epoch. As seen, the precision settings vary depending on the model and size.

Discussion

The techniques evaluated image classification on a variety of neural network models on the CalTech 101 data set. It would be interesting to make a wider comparison with other data sets, such as ImageNet, as well as with neural machine translation, and other scientific applications. The study was contained to image classification with PyTorch C++ models. A limitation of this study is

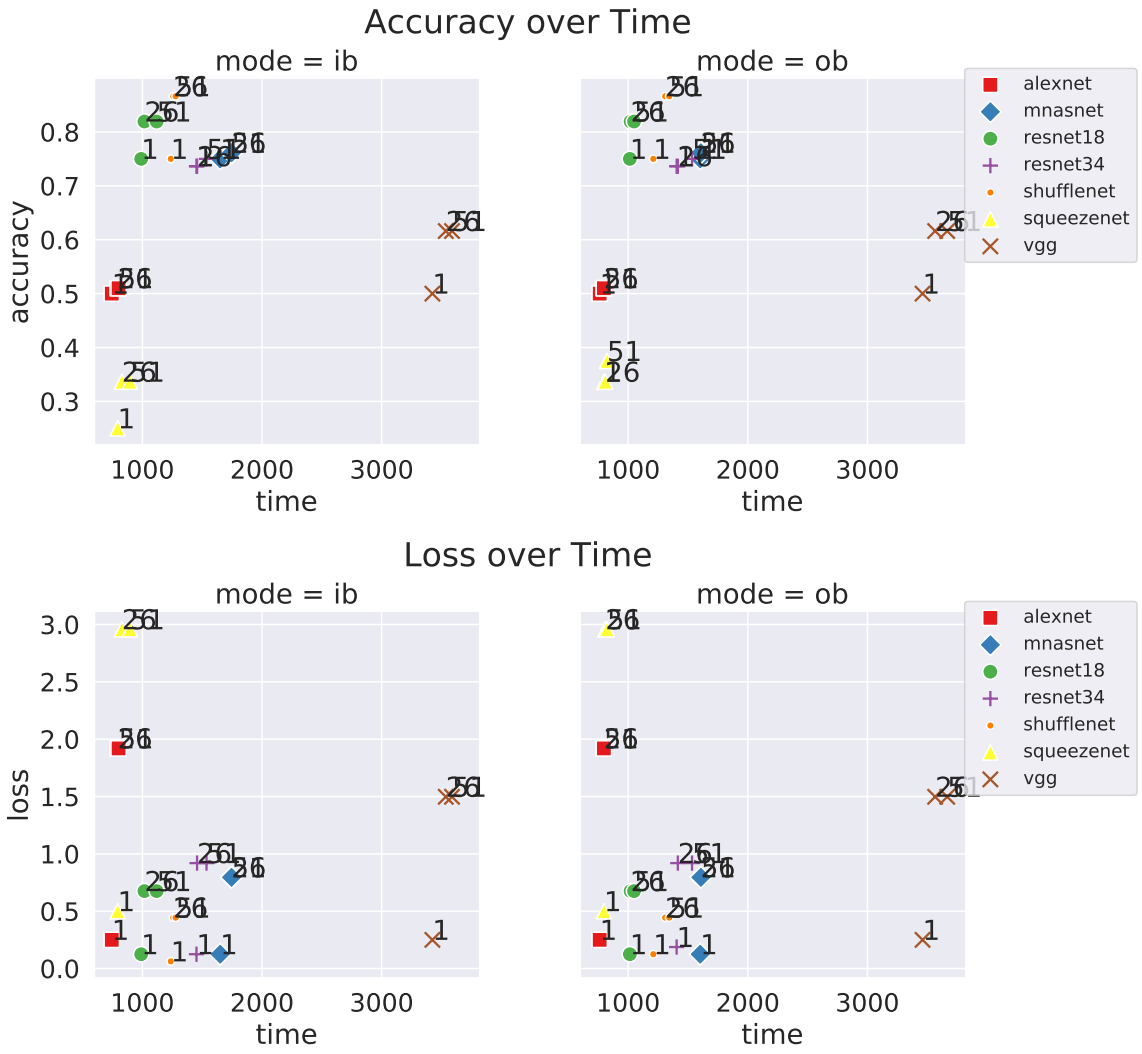

Figure 33. Accuracy over time (top) and loss over time (bottom), comparing vision models.

that the precision settings were made globally for the whole program. This study did not evaluate mixed precision at the variable level. We leave that as future work. Another limitation of this work is that we did not exhaustively explore the precision settings between half and 1 bit. The purpose of this work was to present a proof-of-concept with the tool capabilities that was added in Verificarlo and the types of analysis that can be performed with reduced precision during machine learning.

Prior Work

Techniques to reduce numerical precision have been used for compression, custom quantization points, computing convolutional operations in the logarithmic domain, and stochastic rounding. Deep compression Han, Mao, and Dally (2015) proposes a three-stage compression pipeline that prunes redundant weight connections, quantizes multiple connections to share the same weight, and applies Huffman coding in the fully connected layers to biased effective weights. For the AlexNet neural network, the 256 shared weights were quantized to 8-bits for each convolution layer, and the 32 shared weights were quantized to 5-bits in the fully connected layers without loss in accuracy. They observed that for the last fully-connected layer, most quantized weights were distributed around two peaks. Thus, Huffman coding was used to non-uniformly distribute values, which saved 20-30% in network storage space.

This paper Alistarh, Grubic, Li, Tomioka, and Vojnovic (2017) proposes an approach for quantizing gradients in distributed training of SGD, particularly neural networks. The approach partitions the problem amongst available processors, where each processor broadcasts its unquantized gradients. Then, each processor aggregates the gradients, performs local training with quantized gradients

and, with a uniformly random integer broadcasts the quantized update vector. Their approach compared various neural network models with 1-16 K80 GPUs and found that parallelization decreased epoch time but led to more communication. Their results also compared 4 bit and 8 bit and found that 8 bit quantization was able to maintain the accuracy of the gradients compared to the full precision version, and that 4 bits loses 0.57% for Top-5 accuracy and 0.68% for Top-1 accuracy.

The weights and activations were encoded in a base-2 logarithmic representation Miyashita, Lee, and Murmann (2016), since weights and activations have a non-uniform distribution. Their approach proposed computing the convolution operation in logarithmic domain, where either the individual operands or the operation is converted to the `log` domain and quantized. Their experiments compared quantization in the linear and `log` domains. They trained CIFAR-10 with 5-bits weights and 4-bit activations resulting in minimal performance degradation. They also noted that for 3-bits, the `log` domain tolerated a larger dynamic full-scale range, where AlexNet performed 0.2% worse in the `log` domain compared to the linear domain, but VGG, a higher capacity network, performed 6.2% better in the `log` domain and maintained its Top-5 accuracy.

This work evaluated fixed-point representation of 16 bits with round-to-nearest and stochastic rounding modes for training neural networks S. Gupta, Agrawal, Gopalakrishnan, and Narayanan (2015). The weights W^l and biases B^l were quantized to 16-bits and compared with round-to-nearest and stochastic rounding. Aggressive reduced precision may result in loss of gradient information, if updates are significantly smaller than ϵ. In round-to-nearest, any parameter update in the range of $(-\frac{\epsilon}{2}, \frac{\epsilon}{2})$ is always rounded to zero, whereas stochastic rounding

maintains a non-zero probability of small parameter updates to $\pm\epsilon$. Experiments compared MNIST and CIFAR10 datasets and found that, in general, stochastic rounding maintained accuracy compared to round-to-nearest mode.

Automatic mixed precision (AMP) in PyTorch consists of `autocast` and `GradScalar` as modules for executing in low precision PyTorch Mixed Precision (2019). `autocast` will automatically typecast certain operations to `half`, such as convolution and matrix multiplication, whereas other operations will be executed in `float32`, such as softmax and point-wise operations, based on their predefined operation eligibility. The model is converted to `float16` where possible, and a copy of the master weights is kept in `float32` to accumulate per-iteration weight updates. Loss scaling is applied, using the master weights, to preserve small gradient values. The TensorFlow mixed precision, provided by the Keras high-level API TensorFlow Mixed Precision (2021), takes a similar approach toward quantizing variables. However, these approaches perform mixed precision automatically with predefined rules and do not provide a mechanism for the user to specify the precision requirements.

Conclusion

This chapter discusses the precision requirements when training neural networks. Specifically, the chapter seeks to understand how stable the numerical representation is when changing the precision and range sizes. We implemented our work on the LLVM intermediate representation layer and evaluated our work on various PyTorch C++ image applications. We demonstrate our capabilities and were able to identify where in the training phase that the precision is stable, and where it becomes unstable.

The follow-up work that builds upon this work can lead in several directions. For instance, mixed-precision remains fixed throughout the program run. What would be interesting is whether the precision can change throughout the duration of the training run. One approach would be to utilize just-in-time (JIT) compilation of various precision sizes during program execution. This would enable a more dynamic effect of numerical representation and is not limited to machine learning. Another area of exploration is error analysis of accuracy in fault tolerant settings. For instance, the resiliency of the classification models becomes important, especially since it is being incorporated into our daily lives. The security of the models and weeding out false positives become even more urgent.

CHAPTER VII

SUMMARY AND FUTURE DIRECTIONS

The findings of this dissertation are summarized in this chapter. In addtion, directions for future work are also discussed.

Summary

In order to optimize for performance and accuracy, a clear understanding of the optimization landscape is needed. This dissertation work outlines where the potentional opportunites are for optimizing performance while maintaining the learning trajectory curves. Specifically, we evaluated automatic performance tuning for GPUs and developed search heuristics that worked in the static analysis case, which improved our search space by 92%. Next, we evaluated subgraph matching for representing performance profiles for GPU execution. In that study, we were able to define an architecture independent way for matching control flow graphs, and demonstrated that capability with various CUDA programs. Next, we evaluated machine translation and the hyper-parameters that are entailed for tuning a translation system. In that work, we noted that certain hyper-parameters took longer than others. Finally, we investigated reduced precision for increasing exeuction performance for image classification.

Future Work

This section discusses several research directions that can be pursued for the various topics that were covered in this dissertation.

Optimizing Code Generation. In optimizing CUDA code generation, this work Lim et al. (2017) optimized performance, and accounted for threads, blocks, and shared memory. The work did not account for memory behavior

on GPUs, specifically where communication vs. computation optimization opportunities may lie. A more in-depth analysis of loop transformations, such as tiling, fusion, and mixed-precision, could be pursued. Also, pattern matching could be directly employed during Orio automatic performace tuning.

GPU Subgraph Matching. This work Lim et al. (2019) defined the necessary decision support boundaries that characterizes the runtime behavior of a GPU application in the form of a control flow graph, which aides in matching with other unseen GPU kernels. Some areas that need further work include formally providing provable guarantees that pattern matching will always the lead solver to an optimum. In addition, subgraph matching could also be utilized in optimizing hyper-parameters.

Hyper-Parameter Optimization. This work Lim et al. (2018) explored optimizing hyper-parameters for neural machine translation. This work performed grid search when setting hyper-parameters. An area of exploration would be to incorporate a cost metric associated with search methods. Since it is known that machine learning training can take hours to weeks to complete, are there methods that can formally model the whole hyper-parameter training from end-to-end, with adaptive model tuning and checkpointing in-between, without training the network? Also, an area worth investigating is the cost of a hyper-parameter update in relation to hardware counter metrics.

Numerical Representation. Since the recent work on numerical representation Lim et al. (2021) revealed that precision may matter more in certain phases of the training run versus others, this warrants more investigative work on whether dynamic mixed precision could be employed during machine learning training. Several ways of doing that would include JIT-compiling code for certain

precision sizes and running the appropriate precision size. This would provide a more dynamic approach toward numerical representation, versus the approach of fixed sized precision throughout the training run. Another area worth exploring is evaluating the resiliency of applications with mixed precision via fault injection methods. For example, could machine learning models withstand noise and if so, how much noise and at what phases, and how much noise before the overall application is affected?

APPENDIX A

THE FIRST APPENDIX

Stochastic Gradient Descent

A one dimension update of gradient descent involves the following:

$$W(t+1) = W(t) - \eta \frac{dE(W)}{dW} \tag{A.1}$$

The optimal learning rate η_{opt}, or one that gives the fastest convergence, can be derived by a Taylor series expansion on E about current weight W_c

$$E(W) = E(W_c) + (W - W_c)\frac{dE(W_c)}{dW} + \frac{1}{2}(W - W_c)^2 \frac{d^2E(W_c)}{dW^2} \tag{A.2}$$

with $\frac{dE(W_c)}{dW} \equiv \frac{dE}{dW}|_{W=W_c}$. Differentiating both sides with respect to W gives

$$\frac{dE(W)}{dW} = \frac{dE(W_c)}{dW} + (W - W_c)\frac{d^2E(W_c)}{dW^2} \tag{A.3}$$

Set $W = W_{min}$. Note that $dE(W_{min})/dW = 0$, and after rearranging

$$W_{min} = W_c - \left(\frac{d^2E(W_c)}{dW^2}\right)^{-1} \frac{dE(W_c)}{dW} \tag{A.4}$$

Compare with $W(t+1)$ update function, can reach min in one step if

$$\eta_{opt} = \left(\frac{d^2E(W_c)}{dW^2}\right)^{-1}$$

Fig A.1 plots gradient E as function of W. When E is quadratic, the gradient is simply a straight line with value zero at minimum and $\frac{\partial E(W_c)}{\partial W}$ at current weight W_c. $\partial^2 E/\partial^2 W$ is the slope of line, and can be solved in the following way:

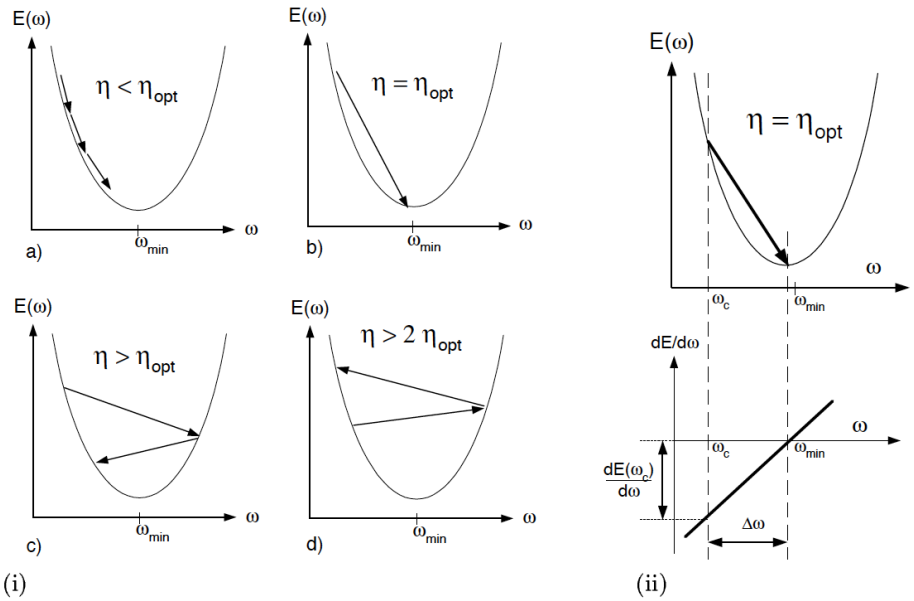

Figure A.1. Gradient descent for different learning rates.

$$\partial^2 E/\partial^2 W = \frac{\partial E(W_c)/\partial W - 0}{W_c - W_{min}}$$

APPENDIX B

THE SECOND APPENDIX

Bilinear Interpolation

Let A represent the canonical adjacency matrix for G_1 and B for G_2, with $A = N \times N$, $B = M \times M$, and $M \geq N$. In other words, we want to scale up A from N to M, which requires constructing interpolated points for every B_{ij}. To find the coordinates for each x and y for a given $B_{(i,j)}$-th element to interpolate, we calculate:

$$x = i \times \frac{M-1}{N-1}, \quad y = j \times \frac{M-1}{N-1}$$

where the $x+1$ and $y+1$ components are given by the floor and ceiling as appropriate, yielding four components: $\{x_1, y_1, x_2, y_2\}$. Note that upon calculating the components, $A_{\{x_1,y_1\}}, A_{\{x_1,y_2\}}, A_{\{x_2,y_1\}}, A_{\{x_2,y_2\}}$ are known points, referenced by the original matrix A.

The solution to the interpolation problem is

$$f(x, y) \approx \omega_0 + \omega_1\, x + \omega_2\, y + \omega_3\, x \cdot y.$$

Solving the linear system gives the coefficients:

$$\begin{bmatrix} 1 & x_1 & y_1 & x_1 y_1 \\ 1 & x_1 & y_2 & x_1 y_2 \\ 1 & x_2 & y_1 & x_2 y_1 \\ 1 & x_2 & y_2 & x_2 y_2 \end{bmatrix} \begin{bmatrix} \omega_0 \\ \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} = \begin{bmatrix} A_{\{x_1,y_1\}} \\ A_{\{x_1,y_2\}} \\ A_{\{x_2,y_1\}} \\ A_{\{x_2,y_2\}} \end{bmatrix},$$

yielding the result:

$$\begin{aligned}
\omega_0 &= \frac{A_{\{x_1,y_1\}} \cdot x_2 \cdot y_2}{(x_1 - x_2)(y_1 - y_2)} + \frac{A_{\{x_1,y_2\}} \cdot x_2 \cdot y_1}{(x_1 - x_2)(y_2 - y_1)} \\
&\quad + \frac{A_{\{x_2,y_1\}} \cdot x_1 \cdot y_2}{(x_1 - x_2)(y_2 - y_1)} + \frac{A_{\{x_2,y_2\}} \cdot x_1 \cdot y_1}{(x_1 - x_2)(y_1 - y_2)}, \\
\omega_1 &= \frac{A_{\{x_1,y_1\}} \cdot y_2}{(x_1 - x_2)(y_2 - y_1)} + \frac{A_{\{x_1,y_2\}} \cdot y_1}{(x_1 - x_2)(y_1 - y_2)} \\
&\quad + \frac{A_{\{x_2,y_1\}} \cdot y_2}{(x_1 - x_2)(y_1 - y_2)} + \frac{A_{\{x_2,y_2\}} \cdot y_1}{(x_1 - x_2)(y_2 - y_1)}, \\
\omega_2 &= \frac{A_{\{x_1,y_1\}} \cdot x_2}{(x_1 - x_2)(y_2 - y_1)} + \frac{A_{\{x_1,y_2\}} \cdot x_2}{(x_1 - x_2)(y_1 - y_2)} \\
&\quad + \frac{A_{\{x_2,y_1\}} \cdot y_2}{(x_1 - x_2)(y_1 - y_2)} + \frac{A_{\{x_2,y_2\}} \cdot x_1}{(x_1 - x_2)(y_2 - y_1)}, \\
\omega_3 &= \frac{A_{\{x_1,y_1\}}}{(x_1 - x_2)(y_1 - y_2)} + \frac{A_{\{x_1,y_2\}}}{(x_1 - x_2)(y_2 - y_1)} \\
&\quad + \frac{A_{\{x_2,y_1\}}}{(x_1 - x_2)(y_2 - y_1)} + \frac{A_{\{x_2,y_2\}}}{(x_1 - x_2)(y_1 - y_2)}.
\end{aligned}$$

This step is carried out for every $\{i, j\}^{th}$ element of B, where $0 \leq i,\, j < M$.

Efficiency and Goodness

Efficiency describes how gainfully employed the GPU floating-point units remained, or FLOPs per second:

$$efficiency \;=\; \frac{op_{fp} + op_{int} + op_{simd} + op_{conv}}{time_{exec}} \cdot calls_n \tag{B.1}$$

The *goodness* metric describes the intensity of the floating-point and memory operation arithmetic intensity:

$$goodness \;=\; \sum_{j \in J} op_j \cdot calls_n \tag{B.2}$$

REFERENCES CITED

[1] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., & Tallent, N. R. (2010). Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, *22*(6), 685–701.

[2] *AI and Compute.* (2018). (`https://openai.com/blog/ai-and-compute/`)

[3] Alistarh, D., Grubic, D., Li, J., Tomioka, R., & Vojnovic, M. (2017). QSGD: Communication-efficient SGD via gradient quantization and encoding. In *Advances in neural information processing systems (nips)* (pp. 1709–1720).

[4] *Allinea DDT.* (2016). (`http://www.allinea.com/products/ddt`)

[5] Ammons, G., Ball, T., & Larus, J. R. (1997). Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices*, *32*(5), 85–96.

[6] Baboulin, M., Buttari, A., Dongarra, J., Kurzak, J., Langou, J., Langou, J., ... Tomov, S. (2009). Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, *180*(12), 2526–2533.

[7] Bahar, P., Alkhouli, T., Peter, J.-T., Brix, C. J.-S., & Ney, H. (2017). Empirical investigation of optimization algorithms in neural machine translation. *The Prague Bulletin of Mathematical Linguistics*, *108*(1), 13–25.

[8] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[9] Ball, T., & Larus, J. R. (1994). Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *16*(4), 1319–1360.

[10] Bergstra, J. S., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems (nips)* (pp. 2546–2554).

[11] Böhm, C., & Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM*, *9*(5), 366–371.

[12] Borgelt, C., & Berthold, M. R. (2002). Mining molecular fragments: Finding relevant substructures of molecules. In *Data mining, 2002. icdm 2003. proceedings. 2002 ieee international conference on* (pp. 51–58).

[13] Britz, D., Goldie, A., Luong, T., & Le, Q. (2017). Massive exploration of neural machine translation architectures. *arXiv preprint arXiv:1703.03906*.

[14] Chaimov, N., Norris, B., & Malony, A. (2014, Dec). Toward multi-target autotuning for accelerators. In *Parallel and distributed systems (icpads), 2014 20th ieee international conference on* (p. 534-541). doi: 10.1109/PADSW.2014.7097851

[15] Chatelain, Y., Petit, E., de Oliveira Castro, P., Lartigue, G., & Defour, D. (2019). Automatic exploration of reduced floating-point representations in iterative methods. In *European conference on parallel processing* (pp. 481–494).

[16] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., & Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *Workload characterization, 2009. iiswc 2009. ieee international symposium on* (pp. 44–54).

[17] *CHiLL: A Framework for Composing High-Level Loop Transformations* (Tech. Rep.). (2008, June). USC Department of Computer Science.

[18] Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

[19] Chong, E. K. P., & Zak, S. H. (2013). An Introduction to Optimization. Wiley.

[20] *Cloud TPU.* (2019). (`https://cloud.google.com/tpu/docs/tpus`)

[21] *Collective Knowledge (CK).* (n.d.). (`http://cknowledge.org`)

[22] Csardi, G., & Nepusz, T. (n.d.). The iGraph software package for complex network research.

[23] *CUDA occupancy calculator.* (2016). `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

[24] Danalis, A., Marin, G., McCurdy, C., Meredith, J. S., Roth, P. C., Spafford, K., ... Vetter, J. S. (2010). The scalable heterogeneous computing (SHOC) benchmark suite. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units* (pp. 63–74).

[25] Davies, M., Srinivasa, N., Lin, T.-H., Chinya, G., Cao, Y., Choday, S. H., ... others (2018). Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, *38*(1), 82–99.

[26] Denis, C., Castro, P. D. O., & Petit, E. (2015). Verificarlo: checking floating point accuracy through monte carlo arithmetic. *arXiv preprint arXiv:1509.01347*.

[27] de Oliveira Castro, P., Akel, C., Petit, E., Popov, M., & Jalby, W. (2015). CERE: LLVM Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. *ACM Transactions on Architecture and Code Optimization (TACO)*, *12*(1), 6. doi: 10.1145/2724717

[28] De Sa, C., Feldman, M., Ré, C., & Olukotun, K. (2017). Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th annual international symposium on computer architecture* (pp. 561–574).

[29] Diamos, G., Ashbaugh, B., Maiyuran, S., Kerr, A., Wu, H., & Yalamanchili, S. (2011). SIMD re-convergence at thread frontiers. In *Proceedings of the 44th annual ieee/acm international symposium on microarchitecture* (pp. 477–488).

[30] Farooqui, N., Kerr, A., Eisenhauer, G., Schwan, K., & Yalamanchili, S. (2012). Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures. In *Performance analysis of systems and software (ispass), 2012 ieee international symposium on* (pp. 58–67).

[31] Fursin, G. e. (2011). Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, *39*(3), 296–327.

[32] Gonzales, R. C., & Woods, R. E. (1993). Digital Image Processing. Addison-Wesley.

[33] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press. (`http://www.deeplearningbook.org`)

[34] Gupta, R., Laguna, I., Ahn, D., Gamblin, T., Bagchi, S., & Lin, F. (2015). STATuner: Efficient Tuning of CUDA Kernels Parameters. In *Supercomputing conference (sc 2015)* (p. poster).

[35] Gupta, S., Agrawal, A., Gopalakrishnan, K., & Narayanan, P. (2015). Deep learning with limited numerical precision. In *International conference on machine learning* (pp. 1737–1746).

[36] Haidar, A., Wu, P., Tomov, S., & Dongarra, J. (2017). Investigating half precision arithmetic to accelerate dense linear system solvers. In *Proceedings of the 8th workshop on latest advances in scalable algorithms for large-scale systems* (p. 10).

[37] Han, S., Mao, H., & Dally, W. J. (2015). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

[38] Hartono, A., Norris, B., & Sadayappan, P. (2009). Annotation-based empirical performance tuning using orio. In *2009 ieee international symposium on parallel & distributed processing* (pp. 1–11).

[39] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 770–778).

[40] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

[41] Huan, J., Wang, W., & Prins, J. (2003). Efficient mining of frequent subgraphs in the presence of isomorphism. In *Data mining, 2003. icdm 2003. third ieee international conference on* (pp. 549–552).

[42] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., & Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*.

[43] Junczys-Dowmunt, M., & Grundkiewicz, R. (2016). Log-linear combinations of monolingual and bilingual neural machine translation models for automatic post-editing. In *Acl* (pp. 751–758). Berlin, Germany: Association for Computational Linguistics. Retrieved from `http://www.aclweb.org/anthology/W16-2378`

[44] Junczys-Dowmunt, M., Grundkiewicz, R., Grundkiewicz, T., Hoang, H., Heafield, K., Neckermann, T., ... others (2018). Marian: Fast Neural Machine Translation in C++. *arXiv preprint arXiv:1804.00344*.

[45] Koehn, P. (2017). Neural machine translation. *arXiv preprint arXiv:1709.07809*.

[46] Koehn, P., Hoang, H., Birch, A., Callison-Burch, C., Federico, M., Bertoldi, N., ... others (2007). Moses: Open source toolkit for statistical machine translation. In *Acl* (pp. 177–180).

[47] Koutra, D., Vogelstein, J. T., & Faloutsos, C. (n.d.). DeltaCon: A principled massive-graph similarity function..

[48] Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*.

[49] Lim, R., Carrillo-Cisneros, D., Alkowaileet, W., & Scherson, I. (2014). Computationally efficient multiplexing of events on hardware counters. In *Linux symposium.*

[50] Lim, R., Castro, P. d. O., Coti, C., Jalby, W., & Malony, A. (2021). Reduced Precision Computation for Accurate and Robust Learning Systems. In *5th workshop on naval applications of machine learning* (p. poster).

[51] Lim, R., Heafield, K., Hoang, H., Briers, M., & Malony, A. (2018). Exploring hyper-parameter optimization for neural machine translation on gpu architectures. *arXiv preprint arXiv:1805.02094*.

[52] Lim, R., Malony, A., Norris, B., & Chaimov, N. (2015). Identifying optimization opportunities within kernel execution in gpu codes. In *European conference on parallel processing* (pp. 185–196).

[53] Lim, R., Norris, B., & Malony, A. (2016). Tuning heterogeneous computing architectures through integrated performance tools. In *Gpu technology conference* (p. poster).

[54] Lim, R., Norris, B., & Malony, A. (2017). Autotuning gpu kernels via static and predictive analysis. In *2017 46th international conference on parallel processing (icpp)* (pp. 523–532).

[55] Lim, R., Norris, B., & Malony, A. (2019). A similarity measure for gpu kernel subgraph matching. In *Languages and compilers for parallel computing (lcpc)* (pp. 37–53). Cham: Springer International Publishing.

[56] *Lower Numerical Precision Deep Learning Inference and Training* . (2018). (`https://software.intel.com/content/www/us/en/develop/articles/lower-numerical-precision-deep-learning-inference-and-training.html`)

[57] Ma, N., Zhang, X., Zheng, H.-T., & Sun, J. (2018). Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the european conference on computer vision (eccv)* (pp. 116–131).

[58] Malony, A. D., Biersdorff, S., Shende, S., Jagode, H., Tomov, S., Juckeland, G., ... Lamb, C. (2011). Parallel performance measurement of heterogeneous parallel systems with GPUs. In *2011 international conference on parallel processing* (pp. 176–185).

[59] Mametjanov, A., Lowell, D., C.C. Ma, C.-C., & Norris, B. (2012). Autotuning stencil-based computations on GPUs. In *Cluster computing (cluster), 2012 ieee international conference on* (pp. 266–274).

[60] Marin, G., Dongarra, J., & Terpstra, D. (2014). Miami: A framework for application performance diagnosis. In *Performance analysis of systems and software (ispass), 2014 ieee international symposium on* (pp. 158–168).

[61] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., ... others (2017). Mixed precision training. *arXiv preprint arXiv:1710.03740*.

[62] Miettinen, K. (2012). *Nonlinear multiobjective optimization* (Vol. 12). Springer Science & Business Media.

[63] Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., ... Newhall, T. (1995). The paradyn parallel performance measurement tool. *Computer*, *28*(11), 37–46.

[64] *Mixed Precision Programming.* (2016). (`https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/`)

[65] Miyashita, D., Lee, E. H., & Murmann, B. (2016). Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*.

[66] Modha, D. S. (2017). Introducing a brain-inspired computer. (`https://www.research.ibm.com/articles/brain-chip.shtml`)

[67] Monsifrot, A., Bodin, F., & Quiniou, R. (2002). A machine learning approach to automatic production of compiler heuristics. In *Artificial intelligence: Methodology, systems, and applications* (pp. 41–50). Springer.

[68] *Neural Processor.* (2020). (`https://en.wikichip.org/wiki/neural_processor`)

[69] Nukada, A., & Matsuoka, S. (2015). Auto-tuning 3-D FFT library for CUDA GPUs. In *Supercomputing conference (sc 2009).*

[70] NVIDIA. (n.d.). *CUDA Toolkit.* `https://developer.nvidia.com/cuda-toolkit`.

[71] *NVIDIA GeForce GTX 680 Whitepaper* (Tech. Rep.). (2012). (`http://bit.ly/2jzzX13`)

[72] *Nvidia Visual Profiler.* (2016). (`https://developer.nvidia.com/nvidia-visual-profiler`)

[73] Park, J., Naumov, M., Basu, P., Deng, S., Kalaiah, A., Khudia, D., ... others (2018). Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*.

[74] Parker, D. S. (1997). Monte carlo arithmetic: exploiting randomness in floating-point arithmetic.

[75] PyTorch. (2019). *Automatic Mixed Precision Package.* (`https://pytorch.org/docs/master/amp.html`)

[76] Sabne, A., Sakdhnagool, P., & Eigenmann, R. (2016). Formalizing structured control flow graphs. In *Languages and compilers for parallel computing (lcpc)* (Vol. 10136). Lecture Notes in Computer Science.

[77] Sarkar, V. (1989). Determining average program execution times and their variance. In *Acm sigplan notices* (Vol. 24, pp. 298–312).

[78] Sennrich, R., Haddow, B., & Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

[79] Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., & De Freitas, N. (2016). Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, *104*(1), 148–175.

[80] Shende, S. S., & Malony, A. D. (2006). The TAU parallel performance system. *International Journal of High Performance Computing Applications*, *20*(2), 287–311.

[81] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[82] Singh, R., Xu, J., & Berger, B. (2007). Pairwise global alignment of protein interaction networks by matching neighborhood topology. In *Research in computational molecular biology* (pp. 16–31).

[83] Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems* (pp. 2951–2959).

[84] Sreepathi, S., Grodowitz, M., Lim, R., Taffet, P., Roth, P. C., Meredith, J., ... Vetter, J. (2014). Application characterization using Oxbow toolkit and PADS infrastructure. In *Proceedings of the 1st international workshop on hardware-software co-design for high performance computing* (pp. 55–63).

[85] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, *15*(1), 1929–1958.

[86] Stephenson, M., & Amarasinghe, S. (2005). Predicting unroll factors using supervised classification. In *Code generation and optimization, 2005. cgo 2005. international symposium on* (pp. 123–134).

[87] Stevens, R., Taylor, V., Nichols, J., Maccabe, A. B., Yelick, K., & Brown, D. (2020). *AI for Science* (Tech. Rep.). Argonne National Lab.(ANL), Argonne, IL (United States). (`https://anl.app.box.com/s/f7m53y8beml6hs270h4yzh9l6cnmukph`)

[88] *Stochastic Gradient Descent.* (2021). (`https://en.wikipedia.org/wiki/Stochastic_gradient_descent`)

[89] Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems* (pp. 3104–3112).

[90] Sze, V., Chen, Y.-H., Yang, T.-J., & Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, *105*(12), 2295–2329.

[91] Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., & Le, Q. V. (2019). Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the ieee/cvf conference on computer vision and pattern recognition* (pp. 2820–2828).

[92] *Tensor Core Performance.* (2019). (`https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9926-tensor-core-performance-the-ultimate-guide.pdf`)

[93] *Tensor Float 32.* (2019a). (`https://blogs.nvidia.com/blog/2020/05/14/tensorfloat-32-precision-format`)

[94] *Tensor Float 32.* (2019b). (`https://en.wikipedia.org/wiki/Tensor_Processing_Unit`)

[95] TensorFlow. (2021). *TensorFlow Mixed Precision.* (`https://www.tensorflow.org/guide/mixed_precision`)

[96] Tomov, S., Nath, R., Ltaief, H., & Dongarra, J. (2010, January). Dense linear algebra solvers for multicore with GPU accelerators. In *International parallel and distributed processing symposium (ipdps 2010).*

[97] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems* (pp. 5998–6008).

[98] Volkov, V. (2010). Better performance at lower occupancy..

[99] Wang, Y. E., Wei, G.-Y., & Brooks, D. (2019). Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint arXiv:1907.10701*.

[100] WikiChip. (2017). *Intel Sky Lake.* (`https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)`)

[101] WikiChip. (2019). *Intel Cascade Lake.* (`https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake`)

[102] Williams, M. H., & Ossher, H. (1978). Conversion of unstructured flow diagrams to structured form. *The Computer Journal*, *21*(2), 161–167.

[103] Wu, H., Diamos, G., Li, S., & Yalamanchili, S. (2011). Characterization and transformation of unstructured control flow in GPU applications. In *1st international workshop on characterizing applications for heterogeneous exascale systems.*

[104] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., ... others (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

[105] Xu, R., Chandrasekaran, S., Tian, X., & Chapman, B. (2016). An analytical model-based auto-tuning framework for locality-aware loop scheduling. In *International conference on high performance computing* (pp. 3–20).

[106] Yan, X., & Han, J. (2002). gspan: Graph-based substructure pattern mining. In *Data mining, 2002. icdm 2003. proceedings. 2002 ieee international conference on* (pp. 721–724).

[107] Zhang, F., & D'Hollander, E. H. (2004). Using hammock graphs to structure programs. *IEEE Transactions on Software Engineering*, *30*(4), 231–245.