Large scale computing applications are an integral part of everyday life. Web search, speech recognition, recommendation systems, and many other applications are powered by machine learning models containing billions of parameters. These large models run on distributed systems with hundreds of Central Processing Units (CPUs) and Graphics Processing Units (GPUs). Similarly, computer simulation models for weather prediction, image processing, fluid dynamics, and others run on supercomputers. However, it is difficult and error-prone to write scalable code for simulations and distributed machine learning. Programmers today must grapple with low-level issues like synchronization, deadlocks, and data locality.

The goal of my research is to enable domain experts, like computational science and machine learning practitioners, to easily and efficiently use high performance hardware. I have designed novel abstractions and languages to describe domain applications naturally without sacrificing on performance. In many cases, my abstractions have outperformed low-level, hand-written code. My work allows domain experts to completely focus on their applications, and still achieve maximal performance on sophisticated, high-performance computing hardware. The major takeaway from my research is that new abstractions can make programming easy and expose new optimizations for better performance. The major contributions of my research are:

1. New optimization and code generation algorithms that generate efficient image processing and scientific computations for CPUs and GPUs while still providing a high level abstraction [15, 16, 17].
2. A programming language and runtime system for expressing distributed machine learning workloads that accelerate the performance of distributed training and inference for widely-used, large machine learning models [12].
3. A system to accelerate graph machine learning applications, that is the first to implement GPU-based graph sampling [18].
4. A foundational semantics for so-called “serverless computing”, which is a new approach to cloud computing [13]. The semantics models subtle bugs that arise and presents ways for programmers to write efficient and correct serverless programs.
5. The first large-scale performance analysis of WebAssembly using significant real-world benchmarks [14].

Impact

Several of my papers have already had impact on the research community:

1. My work on new algorithms and cost models for automatically optimizing image processing and scientific computations for modern CPUs and GPUs has influenced new search algorithms and cost models in this area [2, 3, 9, 11].
2. My work on the first system to execute graph sampling on GPUs, has spurred further research in the area [19].
3. My work on serverless computing has served as the foundation for two other recent works [3, 10].
4. My WebAssembly performance evaluation work has been employed in other empirical work on WebAssembly [6, 7, 21, 23], and influenced the design of several WebAssembly-based systems [8, 20, 22]. This work also influenced the development of WebAssembly runtimes at Google and Intel.

My research was invited for an article in USENIX ;login; has received an ACM SIGPLAN Distinguished Paper Award at OOPSLA, and a Best Paper Award at PACT.

1 Current Research

1.1 Efficient Execution of Large-Scale Scientific Computation on CPUs and GPUs

The summer of 2021 saw flash floods and damaging heatwaves across the United States and Europe due to the effects of climate change. The scale of these events were shocking even to climate scientists who warned the BBC that “We should be alarmed because the IPCC (climate computer) models are just not good enough” [1]. Climate models consist of computations known as stencil computations. Stencil computations also arise in image processing, cellular automata, differential equations, and other contexts. Stencil computations lend themselves to parallelization because they consist of thousands of deeply nested loops, but naive parallelization leads to inefficient use of hardware. Efficient use of hardware involves fusing loops, dividing both inputs and outputs of loops into optimally sized tiles, and dealing with issues like cache misses. Moreover, the optimal configuration of a stencil computation can vary significantly between different CPU and GPU architectures. Therefore, trying to optimize stencil computations by hand can lead to a lot of wasted effort. In my research, I ask the question, how can we automatically optimize stencil computations for CPUs and GPUs?

I have developed algorithms to optimize stencil computations for diverse CPU and GPU architectures. I have developed an algorithm that exhaustively searches the space of all loop fusion possibilities in seconds for CPU-based stencil computations [15, 16]. I also developed new algorithms that maximize the utilization of GPU-based stencil computations [17]. Notably, both approaches present programmers with a simple programming model, while generating high-performance code under the hood.
Automatic Loop Fusion and Tile Size Selection of Stencil Computations In a stencil computation, any combination of loops can be fused, which leads to an exponential number of possible programs. However, only a few of these programs perform well on a particular hardware. Moreover, manually finding these programs is time-consuming or even impossible in many cases. Therefore, my research asks, can we automatically find an optimal stencil program for a given target architecture? I designed Dynamic Programming Fusion (DPFusion) \([15, 16]\), which is an algorithm that exhaustively searches the space of programs with fused loops and varying tile sizes.

DPFusion represents a stencil computation as a Data Flow Graph (DFG) of loops, where an edge between loops represents a read-after-write dependency between them. DPFusion starts from the first loop of the graph and obtains two sub-graphs, where one fuses the first loop with its consumer, and the other does not. The first loop of first subgraph is the fused loop and the first loop of second subgraph is the consumer loop. DPFusion represents the execution time of sub-graphs by a cost metric and returns the minimum cost of both sub-graphs as the cost of the graph. The cost of a subgraph is recursively computed and reused to avoid recomputation. I designed a cost function that returns the cost for executing fused loops on multi-core CPUs. The cost function takes into account the memory accesses performed by fused loops, the size of all caches on the CPU, and the number of CPU cores. By plugging this cost function in DPFusion, I was able to automatically optimize image processing applications and multi-grid computations containing up to 100 loops within 30 seconds. The generated code outperforms manually optimized code in the state-of-the-art Halide DSL for these benchmarks on multi-core CPUs by up to 2.46×. This work has influenced the design of new search algorithms and cost models in this area \([2, 4, 9, 11]\).

New Optimization Techniques for Modern GPUs The architecture and instruction sets of GPUs have evolved significantly in the past decade. For example, contemporary GPUs provide low cost synchronization among few threads using warp synchronization and have deeper memory hierarchy by letting threads share data directly using registers instead of memory using warp shuffle. This evolution has forced domain experts to rewrite their programs to obtain maximum utilization of modern GPUs. Existing techniques to execute stencil computations on a GPU were designed a decade ago. This is why, these techniques cannot use both the low-cost synchronization and deep memory hierarchy of modern GPUs leading to under-utilization of modern GPUs and high number of memory accesses. Therefore, my research asks, can we develop new techniques that fully utilize modern GPUs while still providing the same abstraction? I’ve developed algorithms that require no extra input from domain expert and generate code that utilizes features of modern GPUs to obtain full GPU utilization \([17]\).

These algorithms generate efficient code that assigns tiles of the stencil computation to smaller groups of threads (warps), which restricts synchronization to fewer threads (warp synchronization), and fully utilize the deep memory hierarchy of modern GPUs by sharing data of threads using both registers (warp shuffles) and cache (shared memory). I also developed a new cost function to automatically select parameters required to execute a fused loop efficiently on GPUs. These parameters are the number of registers and size of cache (shared memory) per thread, the number of GPU threads to run, and the number of output elements per thread. The optimized code generated by these algorithms for image processing benchmarks outperformed hand-optimized code of Halide by up to 2×. This work received the Best Paper Award at PACT 2020.

1.2 Efficient Training and Inference of Large Machine Learning Models on GPUs Training and inference of large machine learning models on distributed systems takes several days, millions of dollars, and tens of MWh of energy. For example, language models like GPT-3 that power recent advancements in text summarization, text completion, and code generation from code comments, take 34 days to train on 1024 GPUs. Moreover, these models need to be trained again over new datasets. Hence, even small performance improvements in training and inference can lead to huge savings in time, energy, and money. Thus, in my research, I ask the question, can we provide both high-level abstractions for machine learning practitioners and new optimizations for efficient training and inference?

I have developed two systems to efficiently run diverse machine learning models. CoCoNet \([12]\) is a domain specific language to decrease training and inference times of distributed machine learning models. NextDoor \([18]\) is a system that decreases training time of graph machine learning models. Both systems provide new abstractions to express key parts of a machine learning model and then apply novel optimizations that are not possible with existing abstractions.

Co-optimizing Computation and Communication in Distributed ML Distributed techniques for training and inference of large machine learning models divide either the data, the model, or both over nodes of a distributed system. No matter how the data and model are sliced, all techniques require inter-node communication of the result of computations performed by each node. Deep learning frameworks like PyTorch map computation and communication to highly-optimized
library. However, they are not capable of joint optimization, across computation and communication, or even optimization across libraries. For example, separate communication and computation routines can share data only through GPU memory, and cannot use more sophisticated techniques described in Section 1.1. However, writing low-level jointly optimized code for each combination of computation and communication in a multi-node GPU cluster is infeasible. Therefore, I developed a domain-specific language called CoCoNet [12] that allows a programmer to succinctly describe a distributed machine learning program, in terms of communication and computation stages. The CoCoNet compiler then optimizes the computations and communications expressed in this program, and generates optimized low-level code.

By providing both computation and communication as constructs in the language, CoCoNet is able to apply new joint optimizations across computation and communication. Some of these optimizations are: (i) fusing both communication and computation into a single GPU function call to decrease memory accesses and (ii) fine-grained overlapping of multiple computation and communication operations by modifying the computation to generate data in the order required by the communication. My experiments show that CoCoNet significantly improves the training and inference times of several distributed machine learning workloads. For example, fusing activations into the following communication (AllReduce) and overlapping it with the linear layer computation gives speedup of 1.5× in inference of BERT and GPT-2 models.

Accelerate Graph Sampling for Graph Machine Learning using GPUs

Graph machine learning (Graph ML) powers applications where the data is represented as graphs. Graph ML predicts a property of a vertex based on the properties of the vertex neighbors, for example, Pinterest uses Graph ML to recommend images to a user based on the user’s social network. To decrease the amount of computation involved in a full graph training, Graph ML first generates samples of the input graph using a graph sampling algorithm and then feed these samples to the underlying deep neural network. However, manually writing efficient graph sampling algorithm requires careful thinking of memory accesses and thread scheduling. This is why, before my research, it was not possible to efficiently run graph sampling algorithms on GPUs, and graph neural networks would spend more than half of their training time generating samples. I developed NextDoor [18], which presents new abstractions that facilitate writing a wide variety of graph sampling algorithms, including random walks, neighborhood sampling, and layer sampling in few lines of code and a runtime to execute these algorithms efficiently on GPUs.

In NextDoor’s abstraction, a graph sampling algorithm contains several sets of initial vertices and produces samples of the graph starting from a vertex of each set by taking a series of steps. At each step and for each sample, the algorithm defines a set of vertices that we call transit vertices. The neighbors of transit vertices are chosen based on the sampling criteria of the algorithm and then added to the sample. The key insight in NextDoor’s optimized GPU based implementation is that several samples running concurrently can add neighbors of same transit vertex. This makes it possible to store transit vertices in cache (or even registers) to avoid costly memory loads. Furthermore, NextDoor can use all computation resources by mapping transits to the computation hierarchy (grid, thread block, and warp) based on the number of samples for each transit. NextDoor runs significantly faster than existing graph sampling implementations and improves the training time of graph neural networks by upto 4×. NextDoor is the first system to execute graph sampling on GPUs and this work has influenced further research in this area [19].

1.3 Cloud Computing Abstractions

Cloud computing makes it possible for anyone to get instant access to high-performance hardware. However, writing applications for cloud computing involves extra effort to manage resources and configure the runtime environment. Serverless computing is a new approach to cloud computing where the cloud provider manages these peculiarities, such as load balancing, and resource allocation. For example, when the demand for a service increases, the cloud platform transparently allocates more resources and balances the load, and if the demand falls, the platform automatically deallocates idle resources. Thus, in theory, serverless computing allows programmers to focus on their domain, instead of worrying about the peculiarities of cloud computing. However, in practice, to automatically manage these peculiarities, the serverless platform need to perform several low-level behaviors that are exposed to programmers. For example, to reduce the latency of requests, serverless platforms try to reuse already allocated resources to process multiple requests instead of allocating new resources for each request, but this behavior can lead to incorrect results. Hence, programmers need to spend significant effort to deal with these low-level details. Therefore, my research asks, can we create abstractions to aid domain experts in developing correct and efficient serverless functions? Based on the experience with several serverless computing platforms, I developed an operational semantics model of serverless computing that precisely captures all low-level behaviors of serverless platforms.

To let programmers reason about their code with high-level semantics, I also developed a simplified “naive” semantics. However, there are specific conditions, described informally in the past, under which it is safe for programmers to only use the naive semantics. I formalized these specific conditions and proved the weak bisimulation relation between the naive semantics and the operational semantics under these conditions. Two recent works [3 10] are based on the operational semantics developed in this work. This work received an ACM SIGPLAN Distinguished Paper Award at OOPSLA 2019.
1.4 Performance Analysis of Web Programming Languages

Web browsers are the most popular platforms for running portable applications. For example, image processing, scientific simulations, and machine learning inference applications when written in JavaScript can be executed on all major web browsers. However, the performance of native applications written in native languages, like C and C++, is much worse when compiled to JavaScript. To improve the performance of native applications in browsers, browser vendors designed WebAssembly, which is a portable compilation target for native languages. An early study on WebAssembly’s performance on a benchmark suite of small scientific computing functions showed that WebAssembly is faster than JavaScript and only 10% slower than native code [5]. However, these scientific functions are not real-world applications, which are the intended target for WebAssembly. Therefore, my research asks, how can we run native real-world applications in WebAssembly and what is their performance against native code? I developed an operating system kernel called BROWSIX-WASM [14] that runs inside web browsers and provides a system call interface for native applications compiled to WebAssembly.

Based on BROWSIX-WASM, I developed an infrastructure to execute diverse applications of SPEC Benchmarks compiled to JavaScript and WebAssembly. I then conducted the first comprehensive performance analysis of WebAssembly using SPEC Benchmarks. I found that in line with the early study WebAssembly is faster than JavaScript, but in contrast to the early study WebAssembly is 50% slower than native code. I then conducted an analysis of WebAssembly’s language design and WebAssembly compilers in web browsers to identify the reasons for the slowdown of WebAssembly over native code. This study on WebAssembly’s performance has influenced the design of several WebAssembly-based systems [3,20,22] and has inspired other empirical work on WebAssembly [6,7,21,23]. This work also influenced the development of WebAssembly runtimes at Google and Intel. This work was invited as an article to USENIX ;login: Fall 2019 issue.

2 Future Research

My long term research vision is to work with computational scientists and machine learning practitioners, understand their needs, and develop tools that help them get the most out of current and next-generation high-performance hardware. Towards this goal, I am keen to immediately start working on the following research topics.

Easy and Efficient Programming on GPUs Writing correct but in-efficient programs is easy, but writing correct and efficient programs requires considerable time and effort. For CPU programming there are a diverse set of tools, like debuggers and profilers, that help writing correct and efficient programs. However, there is a dearth of similar tools for GPUs. To help domain experts easily deal with several correctness and performance issues arising in GPU programs, I want to create performance and correctness debugging tools and aim to work with domain experts to help them adopt these tools.

I will develop a framework for writing dynamic analysis tools that provide domain experts with full information of execution of a GPU program and imposes minimal overhead. I aim to work with domain experts to write several domain specific tools using this framework to find performance issues, like doing random memory accesses and not utilizing full memory hierarchy of GPUs. Since the performance of a GPU code can depend on input sizes and values of the input, I aim to develop fuzzing techniques that can generate diverse inputs and find performance issues in diverse use cases of domain applications. To let these tools work on a variety of GPU programs, I aim to develop formal semantics models for popular GPU programming languages and then based on these semantics develop dynamic analysis and fuzzing techniques. I believe these works will strongly improve the landscape of general purpose and domain specific GPU programming.

Programmer Tools for Cloud Computing Serverless Computing is a promising programming model that abstracts out peculiarities of a cloud platform, such as managing resources and balancing load. However, in order to provide this abstraction, a serverless platform needs to perform several low-level behaviors that makes it difficult for programmers to write correct and fast serverless functions. Traditional CPU programming has a diverse set of tools, like debuggers and profilers, available to find the source of correctness and performance bugs but there are no such tools for debugging serverless functions. Based on my work on formal semantics of serverless platform [13], I will design a runtime tracing technique that can determine if the serverless function can deal with the low-level behaviors of serverless platforms. Since this tracing technique can determine robustness of serverless function only after the deployment, I will focus on creating a debugger for serverless functions running on a remote serverless platform. Using this debugger, domain experts will be able to quickly explore different inputs and execution scenarios of their functions before deploying the functions.

Currently, to utilize multiple CPUs and GPUs, domain experts need to manually divide their computation in different serverless functions and invoke each function separately. This approach is in-efficient because sandboxing techniques used by serverless platforms has high latency overhead and individual requests for each function will overwhelm the network. Therefore, I aim to develop a framework for executing parallel programs, like machine learning workloads, efficiently on serverless platforms. I also aim to develop new sandboxing techniques for serverless platforms based on the formal safety guarantees of WebAssembly that can run faster than existing sandbox environments.
References


