A TOOL FOR AUTOMATIC SUGGESTIONS FOR IRREGULAR GPU KERNEL OPTIMIZATION

by

Saeed Taheri

A thesis submitted to the Graduate Council of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
December 2014

DEDICATION

I dedicate this thesis to my family who supports me in my entire life and to the soul of my father who I always feel his support.

ACKNOWLEDGEMENTS

This project was funded in part by a gift from Nvidia Corporation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Future computing systems, from handhelds all the way to supercomputers, will be more parallel and more heterogeneous than today's systems to provide more performance without an increase in power consumption. Therefore, GPUs are increasingly being used to accelerate general-purpose applications, including applications with data-dependent, irregular memory access patterns and control flow.

The growing complexity, non-uniformity, heterogeneity, and parallelism will make these systems, i.e., GPGPU-accelerated systems, progressively more difficult to program. In the foreseeable future, the vast majority of programmers will no longer be able to extract additional performance or energy-savings from next-generation systems because their programming will be too difficult, i.e., the programmer will no longer possess the necessary expertise to understand and exploit the systems effectively. In this project, the characteristics of GPU codes are quantified and, based on these metrics, different optimization suggestions are made.

CHAPTER 1

Introduction

1.1 Serial vs. Parallel

Traditionally, software has been written for ***serial*** computation. A problem is broken into a discrete series of instructions and instructions are executed sequentially one after another (Fig. 1.1). All of these instructions are executed on a single processor and only one instruction may execute at any moment in time [21]. However, for performance reasons, superscalar CPUs may execute multiple independent instructions together and even out-of-order.

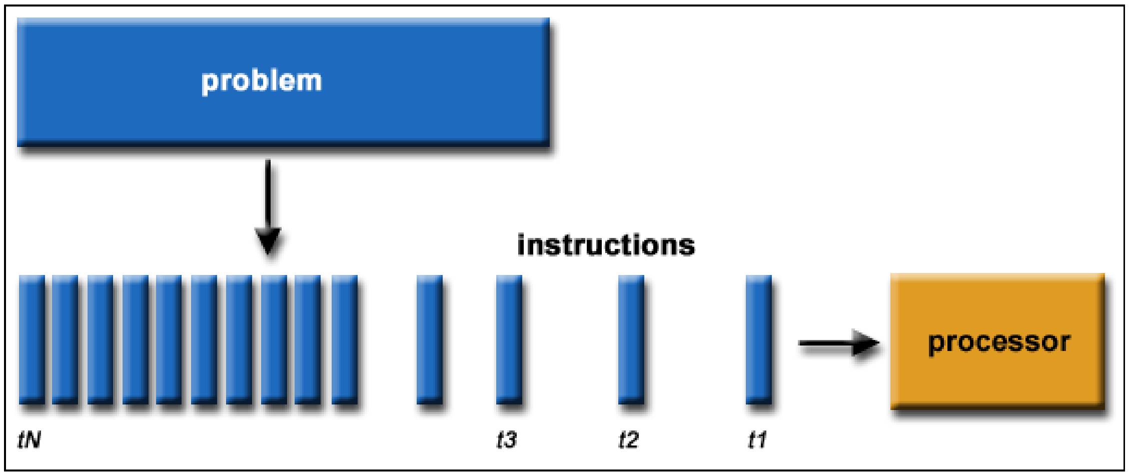


Figure 1.1 Serial computations

In the simplest sense, ***parallel*** computing is the simultaneous use of multiple compute resources to solve a computational problem. A problem is broken into discrete parts that can be solved concurrently. Each part is further broken down to a series of instructions. Furthermore, instructions from each part execute simultaneously on different processors and an overall control/coordination mechanism is employed (Fig. 1.2).

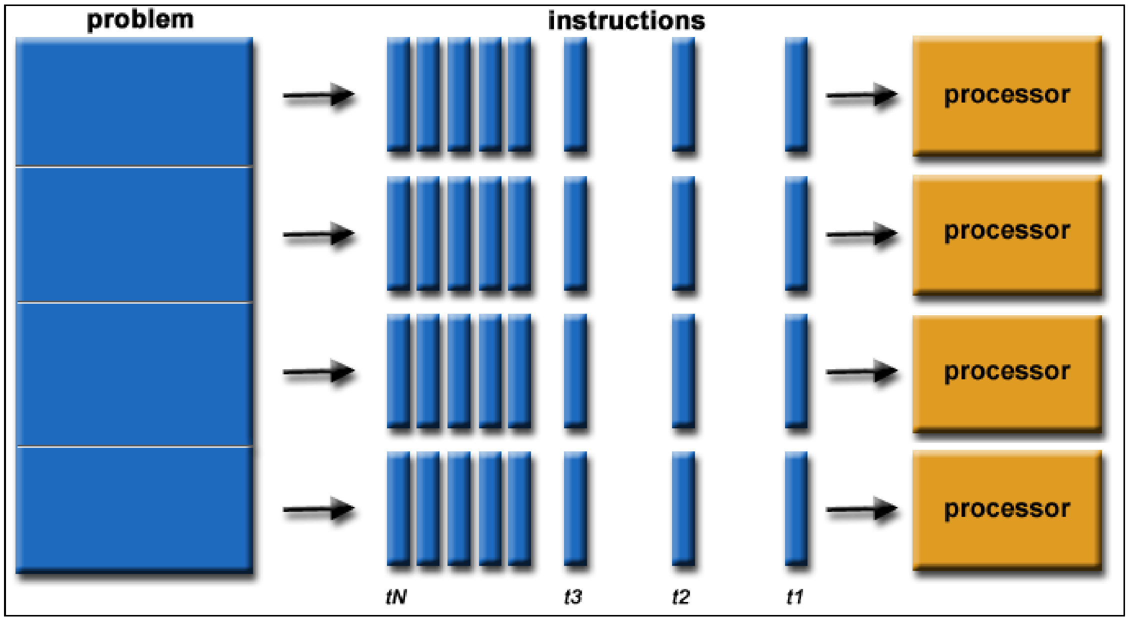


Figure 1.2 Parallel computations

The computational problem should be able to divide into discrete pieces of work that can be solved simultaneously, execute multiple program instructions at any moment in time and be solved in less time with multiple compute resources than with a single compute resource. The compute resources are typically a single computer with multiple processors/cores or an arbitrary number of such computers connected by a network.

1.2 Why Parallel?

Problems are too costly to be solved with the classical approach. Also there is high demand of getting results on specific and reasonable time. In the natural world, many complex, interrelated events are happening at the same time, yet within a temporal sequence. Compared to serial computing, parallel computing is much better suited for modeling, simulating and understanding complex problems which have some characteristics i.e., data reuse and regularity in data accesses and control flow.

Using parallel computing could save time and/or money and/or energy. In theory, assigning more resources to a task will shorten its time to completion, with potential cost savings. Also, parallel computing makes us capable of solving larger and more complex problems. Many such problems are impractical or impossible to solve them on a single computer, especially given limited computer memory. In addition, parallelizing massive computation has additional advantages such as taking advantage of non-local resources [21].

1.3 GPU

A graphics processing unit (GPU) is a specialized processor designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display. GPUs are used in embedded systems, mobile phones, personal computers, workstations, supercomputers, and game consoles. Modern GPUs are very efficient at manipulating computer graphics and image processing, and their highly parallel structure makes them more effective than general purpose CPUs for algorithms where processing of large blocks of data is done in parallel. In a personal computer, a GPU can be present on a video card, or it can be on the motherboard.

The term GPU was popularized by Nvidia in 1999, who marketed the GeForce 256 as “the world’s first ‘GPU’, or Graphics Processing Unit, a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that are capable of processing a minimum of 10 million polygons per second" [22].

1.4 GPU vs. CPU

The CPU or Central Processing Unit is where all the program instructions are executed to process the data. Advancements in modern day CPUs have allowed it to crunch more numbers than ever before, but the advancement in software technology meant that CPUs are still trying to catch up. A Graphics Processing Unit or GPU is meant to alleviate the load of the CPU by handling all the advanced computations necessary to project the final display on the monitor.

Originally, CPUs handled all of the computations and instructions in the whole computer. But as technology progressed, it became more advantageous to take out some of the responsibilities from the CPU and have it performed by other microprocessors. In the days before Graphical User Interfaces (GUIs), the screen was simply a small grid with each box having an 8-bit value that corresponds to a character. This was relatively very easy to do for the CPU, but GUIs have greater resolutions with each pixel having a 16-bit or 32-bit color value.

GPUs were originally developed to render 2D graphics; specifically, to accelerate the drawing of windows in a GUI. But as the need for 3D and faster graphics acceleration grew, the GPU became faster and more specialized in its task. GPUs are now general floating-point processors that can easily crunch computations along with texture mapping tasks.

Hardware wise, GPUs and CPUs are similar but not identical. Fig 1.3 shows the high-level architectures of CPU vs. GPU. As a simple explanation, CPUs have a few powerful core processors while GPUs have many less powerful processors. The specialized nature of GPUs means that it can do its task much faster than a CPU ever can, but it is not able to cover all of the capabilities of the CPU. Multiple GPUs can also be employed to achieve a single goal much like the dual core CPUs currently available [23].

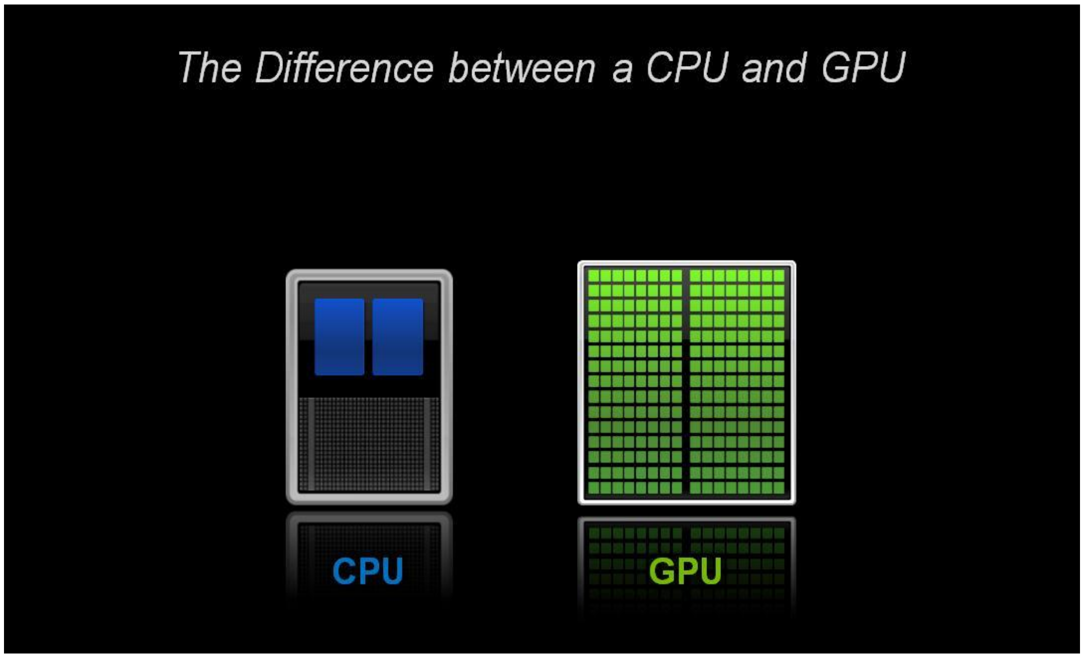


Figure 1.3 CPU vs. GPU [source:Nvidia]

1.5 Thesis Motivation

There are two main difficulties with accelerators such as GPU devices. First, they can only execute certain types of programs efficiently, in particular programs with sufficient parallelism, data reuse, and regularity in their control flow and memory access patterns. Second, it is harder to write effective software for accelerators than for CPUs because of architectural disparities such as very wide parallelism, exposed memory hierarchies, lockstep execution, and memory-access coalescing. Several new programming languages and extensions thereof have been proposed to hide these aspects to various degrees and thus make it easier to program accelerators [25].

The general idea of this thesis is to make parallel programming easier for programmers who are not experts in GPU programming specially with irregular codes which have data-dependent behavior i.e., data accesses and control flow, and are hard to parallelize. Our goal is to design a tool for GPU codes, irregular codes in particular, to find the performance bottlenecks of the codes and suggest some optimization hints to the user to make the code more efficient. By efficiency, we mean efficiency in both aspect of power consumption and runtime.

In the rest of this thesis document, Ch.2 **Background** illustrates the needed background and the general idea and mentions related work. Ch.3 **Design** explains the general idea and how the flow of experiments is. Ch.4 **Experimental Methodology** introduces what type of devices and what version of software had been used for experiments. Ch.5 **Results and Analysis** shows the results of each experiment alongside with analysis. Ch.6 **Summary and Conclusion** contains a brief summary on the thesis, reasonable conclusions and future works.

CHAPTER 2

Background

2.1 GPU Architecture and Programming

GPUs consist of Streaming Multiprocessors (SM) and each SM contains Processing Elements (PE). Threads run on PEs and blocks of threads are allocated to SMs.

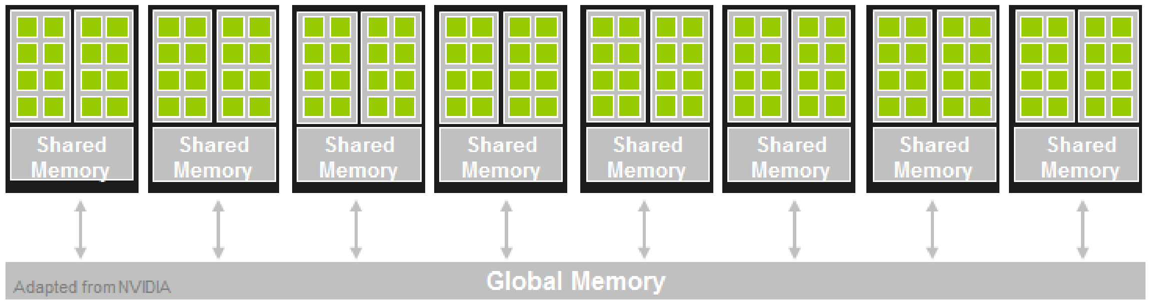


Figure 2.1 Streaming multiprocessors and processing elements

GPU memories are separate from CPU memories. As it is shown in Fig. 1.5 [26], GPUs have a global memory (DRAM), which every thread in each block has access to and also a Constant Memory (DRAM, cached). These two memories are connected to the CPU via the PCI bus. Inside each SM is a shared memory, which is visible to all threads inside a block. Each thread has its own registers, which are limited.

GPUs are designed specifically for graphics and thus are somewhat restrictive in operations and programming. Due to their design, GPUs are only effective for problems that can be solved using stream processing and the hardware can only be used in certain ways [22].

GPUs can only process independent vertices, fragments and streams, but can process many of them in parallel. This is especially effective when the programmer wants to process many vertices or fragments in the same way. In this sense, GPUs are stream processors

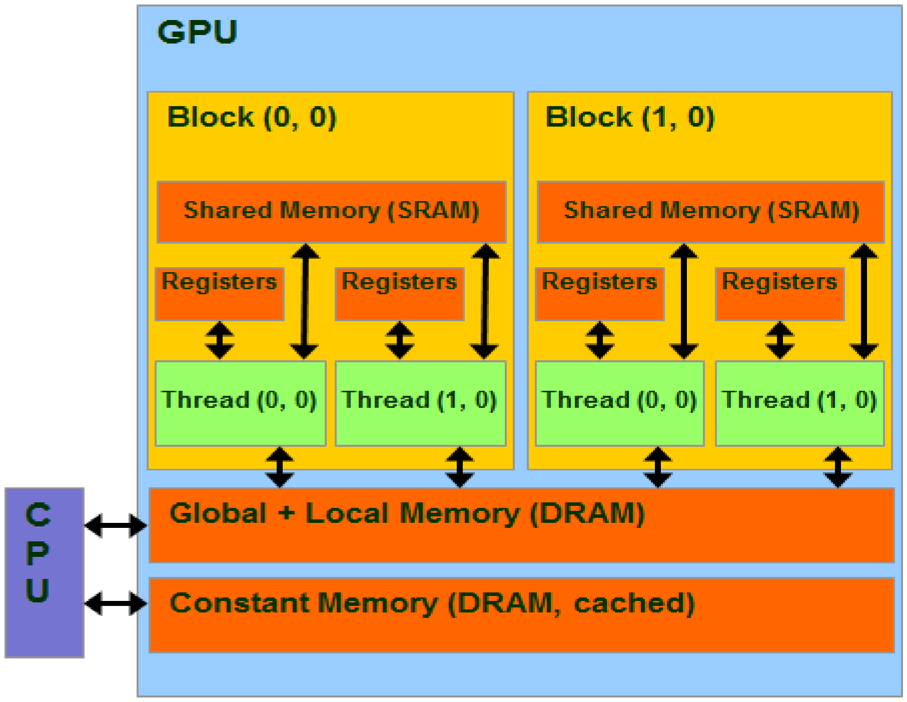


Figure 2.2 GPU memory architecture [source: Nvidia]

that can operate in parallel by running one kernel on many records in a stream at once.

A stream is simply a set of records that require similar computation. Streams provide data parallelism. Kernels are the functions that are applied to each element in the stream. In the GPUs, vertices and fragments are the elements in streams and vertex and fragment shaders are the kernels to be run on them. Kernels can be thought of as the body of loops. On the GPU, the programmer only specifies the body of the loop as the kernel and what data to loop over by invoking geometry processing.

2.2 Irregular Kernels

Recent years have seen a surge of interest in the use of graphics processing units (GPUs) as general-purpose computing accelerators. For programs that map well to GPU hardware, GPUs offer a substantial advantage over multicore CPUs in terms of performance, performance per dollar, and performance per transistor. GPUs also outperform CPUs in energy efficiency on some applications. Due to these benefits, GPUs are appearing as accelerators in many systems.

It is well-known that GPUs are very effective for exploiting parallelism in regular programs that (i) operate on large vectors or matrices, and (ii) access them in statically predictable ways. These codes often have high computational demands, exhibit extensive data parallelism, access memory in a streaming fashion, and require little synchronization. A large number of algorithms from important application areas fit these criteria, including algorithms used in fields ranging from fluid dynamics to computational finance. There exists a broad base of knowledge on the efficient parallelization of these algorithms, and their GPU implementations can be tens of times faster than tuned parallel CPU versions.

However, many problem domains employ algorithms that build, traverse, and update irregular data structures such as trees, graphs, and priority queues. Irregular programs can be found in domains like n-body simulation, data mining, decisions problems that use Boolean satisfiability, optimization theory, social networks, system modeling, compilers, discrete-event simulation and meshing. They are more difficult to parallelize and more challenging to map to GPUs than regular programs [24].

2.3 Main Idea

Several efficient GPU implementations of irregular algorithms have been published, demonstrating that GPUs are capable of accelerating at least some irregular codes relative to multicore CPUs. However, considering novelty of parallel processors and according to difficulties of programming on supercomputers, i.e., using GPGPU, programmers will need access to a system/performance/parallelism expert, but there are only relatively few of them and each one may only be an expert on a certain aspect or application domain. That raises the all-important question of how to best deliver such expertise from different sources to programmers? The main goal of this project is to provide an answer to this question.

I believe the likely solution to be automatic program analysis and recommendation systems. They essentially embody the expert's knowledge and perform the analysis he or she would execute in person to determine how to improve a piece of code. Based on this analysis, the system recommends possible courses of action.

2.4 Related Work

These days, due to slow-down speed of hardware technology improvement, software is playing a more important role to keep computing technology improving. Making applications more efficient in all aspects, leads us to the need of performance measurement. So scientists and experts are trying to design tools for measuring and quantifying performance to some understandable format.

Almost all of the tools are based on traces of events on source code or executable code. They are trying to instrument and measure some metrics/events and then trying to work on the collected data and analyze them. Paradyn [1] was the first tool in automatic performance analysis of HPC dynamic instrumentation to efficiently obtain performance profiles of unmodified executables.

KOJAK [2], Scalasca [3] [13] and Vampir [4] are trace-based tools which support MPI, OpenMP and hybrid platforms. KOJAK automatically deduces the performance properties from the trace files and diagnoses sources of inefficient runtime behavior on a high-level abstraction.

Scalasca [3] [13] is highly scalable and based on wait states that occur in the code (For example, identifying result of unevenly distributed workloads). It uses TAU`s rich instrumentation capabilities [7] (TAU`s API) and processes the performance/trace data in parallel using as many cores as have been used for the target application. Also it scores and summarizes the trace report and shows it on a strong GUI profiler.

Vampir [4] is also a trace-based performance tool for MPI and/or thread/parallel cores. It instruments the source code and because of that has measurement overhead. VampirTrace [5] supports event queue method with a library wrapping approach for CUDA and OpenCL and has been used for GPU performance measurement.

Periscope [6] evaluates performance while the application is still running and searches for previously specified performance problems or properties. It is MPI-based on more focused on efficient communication between cores/processors. Periscope summarizes the measurement phase output and uses summary information instead of tracing.

TAU [7] is a portable tool for performance instrumentation, measurement, analysis and visualization of large scale parallel applications. It has different layer for easier mapping to all parallel architecture and because of that and also the general model for both software and hardware, many other tools use TAU`s API or different layer`s outputs in their approaches. TAU is optimized itself according to platform available features and customizable for different blocks of code. It has 3 major features, source instrumentation, compiler instrumentation and library wrapping. Using the library wrapping benefit of TAU, TAUCuda [8] is created for GPU performance. It has no modification on the source code or binary code.

Recently released tool Score-P [9] is a portable measurement infrastructure for performance measurement tools of HPC. Each of above tools has different measurement output format. For example output format of measurement layer Vampir [4] is OTF and output format of measurement layer of Scalsca [3] [13] is EPILOG/CUBE. Score-P tries to integrate all of these tools into a unified measurement infrastructure. It is compatible with TAU [7], Scalasca [3] [13], Vampir [4] and Periscope [6]. It also covers CUDA. It has flexible measurement without re-compilation, basic and advanced profile generation, event trace recording and online access to profiling data are some of the benefits of Score-P. It supports MPI, OpenMP, and hybrid parallelism (and serial). Also it has enhanced functionality for OpenMP 3.0, CUDA and highly scalable I/O.

HPCToolkit [10] [17] generates statistical profiles using Interval timers and hardware counter overview interrupts and evaluate both application binary and source code.

CUDA Performance Tools Interface (CUPTI) [11] is NVIDIA`s product particularly for CUDA-GPU. A strong library for measuring CUDA code performance according to the device features. It has Callback API which allows you to interject tool code at the entry and exit to each CUDA runtime and driver API call. Also it has Event API which allows the tool to query, configure, start, stop and read event counters on a CUDA enabled device. The PAPI CUDA [12] Component is a hardware performance counter measurement technology for the NVIDIA CUDA platform which provides access to the hardware counters inside the GPU. PAPI CUDA is based on CUPTI support in the NVIDIA driver library. In any environment where the CUPTI-enabled driver is installed, the PAPI CUDA Component can provide detailed performance counter information regarding the execution of GPU kernels.

NVIDIA Visual Profiler [14] is compatible with all CUDA-enabled devices. It finds all bottlenecks with accurate statistics in detail using binary file of CUDA for analysis. Command prompt access to the profiler, remote access and showing all details about time and memory usage by CPU functions and GPU kernels at the same time are some of its features. User can add more hardware metrics for measurement and analysis in case of need of more accurate statistics. After each round of analysis, it shows a brief explanation for each encountered bottleneck. NVIDIA Nsight [15] uses it as profiler tool in Eclipse or Visual Studio. Nsight profiles the code directly from source code and shows the exact line of code which encountered as bottleneck.

eeClust [16] determine relationships between the behavior of parallel programs and the energy consumption of their execution on a compute cluster. It uses Vampir [4] and Scalasca [3] [13] software tools to also record energy-related metrics. The users can then insert energy control calls into their applications which will allow the operating system and the cluster job scheduler to control the cluster hardware in an energy-efficient way. The effectiveness will be evaluated with the help of a small cluster testbed with special energy measurement and control components and synthetic and realistic benchmarks.

Virtual Institute - High Productivity Supercomputing (VI-HPS) [18] is the collaboration of eleven partner institutions for improving the quality and accelerating the development process of complex simulation codes in science and engineering that are being designed to run on highly-parallel computer systems. Most of known tools for parallel performance and measurement such as TAU [7], Scalasca [3][13] and Vampir [4], designed and created by the partners of this big project. They also have couple of ongoing and completed projects in the field of productivity and performance to improve their previous products. POINT, Score-P, SILC, HOPSA, PRIMA and LMAC are tools for integrating and improving the functionality of performance and measurement tools such as TAU [7] and Vampir [4]. For instance, LMAC adds the functionality of automatically examining performance dynamic for irregular behavior of parallel simulation codes to the established performance analysis tools Vampir [4], Scalasca [3] [13] and Periscope [6].

OpenSpeedshop [19] is performance analysis toolset using program counter (pc) sampling, callstack sampling analysis, hardware performance counters, MPI profiling and tracing, I/O profiling and tracing and floating point exception analysis. It supports MPI, Pthreads, OpenMP and hybrid platforms.

Intel VTune Amplifier XE 2013 [20] is the premier profiler for C, C++, C#, FORTRAN, Assembly and Java. It optimizes serial and parallel performance and locates and analyses the bottlenecks of the code, bandwidth, memory accesses and branches with low overhead and high resolution using on-chip hardware. Analyzing hybrid applications using MPI and OpenMP, supporting cluster computing, remote command line access and perfect GUI are some of its more important features. VTune also tune OpenCL and collect GPU metrics.

A few such systems are still in their infancy and not yet in wide use. Our approach is unique among all of above tools. The focus is on irregular GPU kernels, which are more difficult to make efficient. The main advantage of the proposed tool over other similar tools is the suggestion feature. After analyzing, quantifying and measuring performance metrics, the tool recommends to the user some optimization hint such as using different optimization flags, which make the GPU code more optimized. Also we use Machine Learning approaches to make our suggestions more and to automate as much as possible. Also using machine learning in this tool gives us the ability of extending/modifying the suggestion database and simplifies porting the tool to new systems.

CHAPTER 3

Design

3.1 Overview

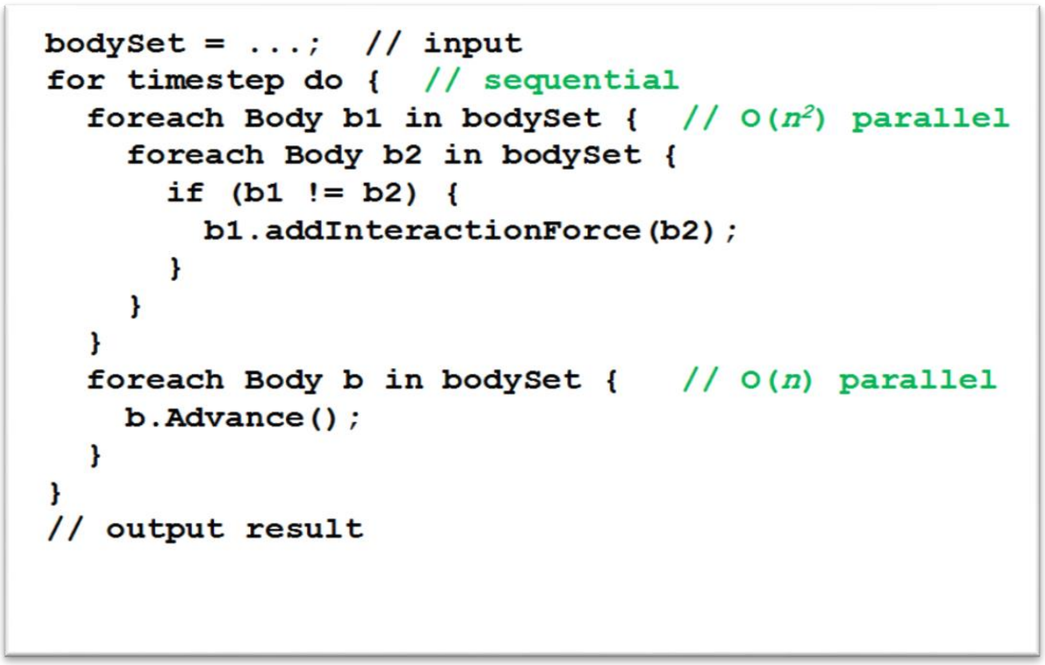
To provide an automated optimization suggestion tool for irregular GPU kernels, we first need to measure how optimized a kernel already is. Our approach to measuring this is to quantify different performance characteristics of each kernel. We use the NVIDIA Visual Profiler [27] for this purpose. It is a profiling tool that can measure a large number of different performance metrics based on hardware performance counters.

The performance quantification results in a large number of features (individual measurements) such as instruction counts, number of cycles, cache hits/misses at different layers, etc. of the kernel code. These feature vectors are input into Machine Learning (ML) methods to classify or rank optimizations. I profiled codes with different sets of optimizations included to train the ML algorithms to hopefully recognize whether an optimization is already present or not and, if not, how much speedup it might provide.

In other words, the goal is to predict by how much each of the trained optimizations (or combination thereof) will improve or hurt the performance of a given CUDA code based on the trained ML model. Based on these predictions, the tool can select which, if any, optimizations to suggest to the user.

3.2 N-Body Problem

In physics, *n*-body simulation is used to compute the motion of individual celestial objects that interact with each other gravitationally [28]. Solving this problem has been motivated by the need to understand the motion of the sun, planets, and the visible stars [29].



```
bodySet = ...;   // input
for timestep do {   // sequential
  foreach Body b1 in bodySet {   // O(n²) parallel
    foreach Body b2 in bodySet {
      if (b1 != b2) {
        b1.addInteractionForce(b2);
      }
    }
  }
  foreach Body b in bodySet {    // O(n) parallel
    b.Advance();
  }
}
// output result
```

Figure 3.1 Simple *n*-body algorithm

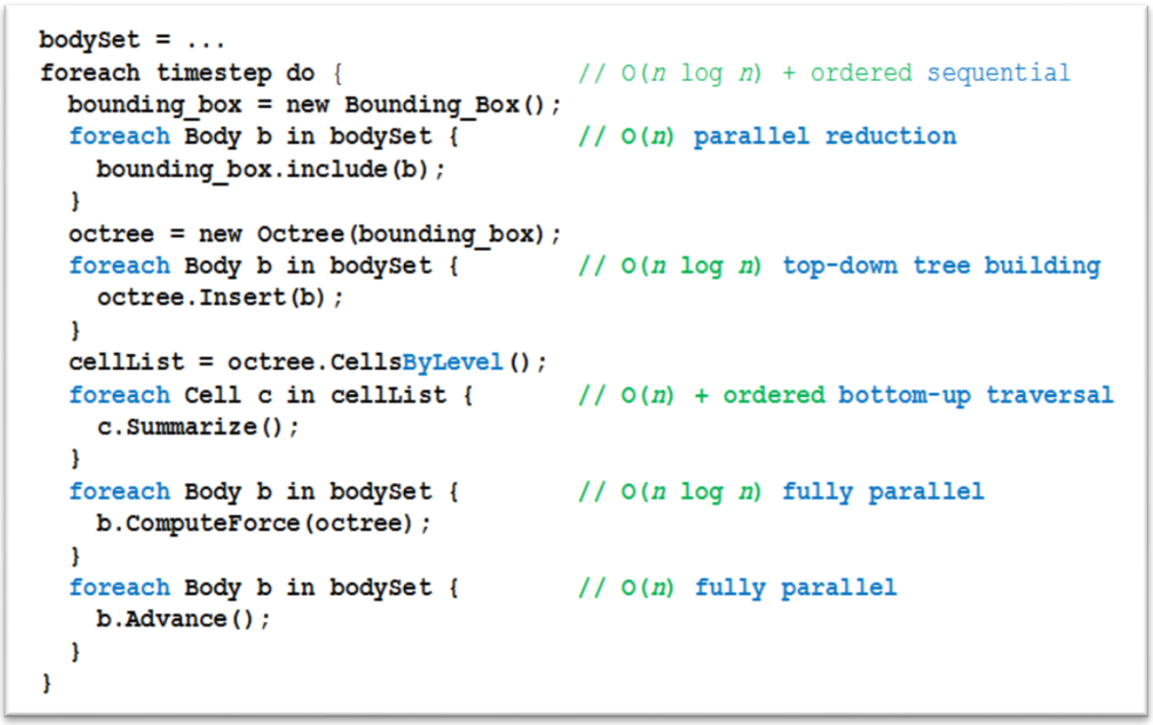
The goal of each step in an *n*-body simulation is to determine the new position of all the bodies by calculating the sum of the forces exerted on each body from all other bodies. For large numbers of bodies, such as in a star cluster, this simulation can be very slow, making parallelization essential. The *n*-body problem is simple to parallelize by dividing the dataset into equal blocks and assigning them to each processor to calculate the force and new position of each body in its block. Fig 3.1 shows a simple parallel algorithm for *n*-body problem that has a complexity of $O(n^2)$. Due to its quadratic time complexity, this algorithm is very slow for large numbers of bodies, even when run in parallel.

To remedy this situation, the Barnes-Hut (BH) *n*-body algorithm has been developed, which has a complexity of *O(n log n)*. However, this algorithm repeatedly builds and traverses an unbalanced tree data structures, resulting in complex and data-dependent program behavior, i.e., irregular control flow and memory-access patterns. This algorithm is difficult to parallelize in general and specifically for GPUs.

3.3 Barnes-Hut Algorithm

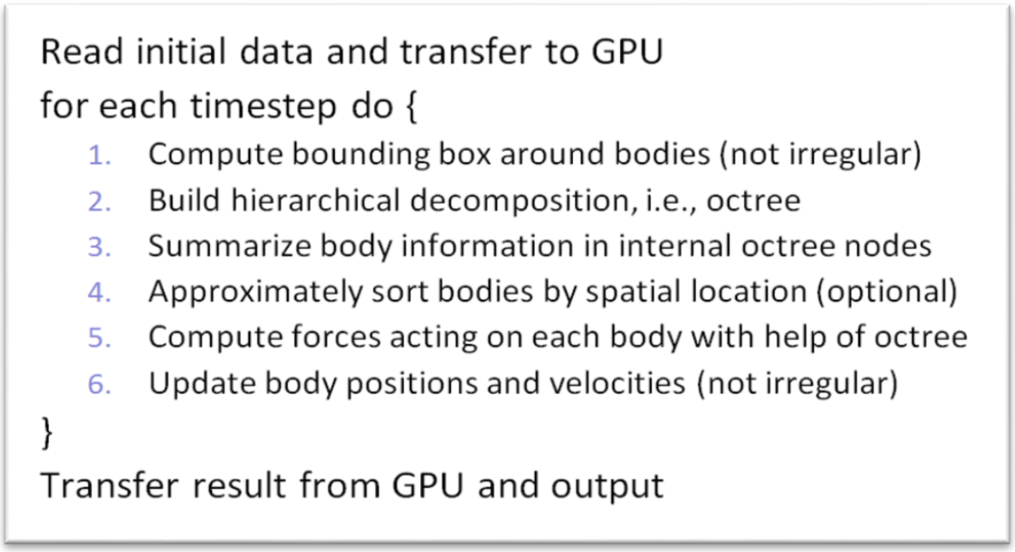
The Barnes-Hut algorithm (by Josh Barnes and Piet Hut) recursively divides the volume around the bodies into cubic cells. The resulting hierarchical decomposition is recorded in an octree (the three-dimensional equivalent of a binary tree). This allows bodies from nearby cells to be treated individually while treating bodies in distant cells together as a single large body centered at the cell's center of mass (or as a low-order multi-pole expansion). This dramatically reduces the number of force calculations that must be computed [30]. The resulting error should be small since the force decreases with the square of distance and the algorithm only uses an approximation for far-away bodies.

I took the BH code from the LonestarGPU suite [35] and modified it, as well as our NB implementation, to include all possible combinations of six source-code optimizations. The GPU implementation of the Barnes-Hut algorithm encompasses the six steps shown in Figure 3.3, each of which is implemented using one or multiple kernels. Since this implementation is very irregular, it represents a useful case study for testing the ML tool.



```
bodySet = ...
foreach timestep do {                    // O(n log n) + ordered sequential
  bounding_box = new Bounding_Box();
  foreach Body b in bodySet {            // O(n) parallel reduction
    bounding_box.include(b);
  }
  octree = new Octree(bounding_box);
  foreach Body b in bodySet {            // O(n log n) top-down tree building
    octree.Insert(b);
  }
  cellList = octree.CellsByLevel();
  foreach Cell c in cellList {           // O(n) + ordered bottom-up traversal
    c.Summarize();
  }
  foreach Body b in bodySet {            // O(n log n) fully parallel
    b.ComputeForce(octree);
  }
  foreach Body b in bodySet {            // O(n) fully parallel
    b.Advance();
  }
}
```

Figure 3.2 Pseudocode of Barnes-Hut algorithm



```
Read initial data and transfer to GPU
for each timestep do {
    1. Compute bounding box around bodies (not irregular)
    2. Build hierarchical decomposition, i.e., octree
    3. Summarize body information in internal octree nodes
    4. Approximately sort bodies by spatial location (optional)
    5. Compute forces acting on each body with help of octree
    6. Update body positions and velocities (not irregular)
}
Transfer result from GPU and output
```

Figure 3.3 GPU-implementation of BH algorithm

3.4 Profiling

In computer science, the term profiling refers to a form of dynamic program analysis that measures, for example, the usage of particular instructions or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization [32]. I use nvprof, the NVIDIA Visual Profiler [27], to collect supported events and metrics from CUDA kernels [31].

3.5 Speedup

Speedup is a metric for capturing the relative performance improvement when executing a task. The speedup is calculated as the ratio of the runtime before applying an optimization over the runtime after applying the optimization. A speedup above 1.0 means that the optimization resulted in an improvement in performance, i.e., a reduction in runtime.

3.6 Machine Learning

Generally speaking, machine learning is a subfield of computer science and statistics that deals with the construction and study of systems that can learn from data, rather than follow only programmed instructions [34]. Usually, machine learning methods are used for problems that require prediction and/or classification. In my project, I need to predict how much speedup we can achieve by applying an optimization to an irregular GPU kernel.

All machine learning approaches have one thing in common: they all use data attributes as features to perform classification/prediction. Each data entry can be viewed as a data point in an N-dimensional space, where N is the number of attributes each data item has. Assume we have 20,000 data entries for training. By assigning them to data points in the N-dimensional space, we can uniquely represent each data item. This model makes it possible to place any new data point into this space so that it can be classified based on its similarity to other nearby data points.

In this project, the target irregular GPU kernel would be profiled (quantified into numbers) and the ML tool will use its trained model to predict the expected speedup.

3.7 Machine Learning Algorithms

In this section, I briefly explain three popular machine leaning algorithms that I have used in my experiments.

Regression is concerned with modelling the relationship between variables that is iteratively refined using a measure of error in the predictions made by the model. Regression methods are important in statistics and have been cooped into statistical machine learning.

The instance-based learning model is a decision problem with instances or examples of training data that are deemed important or required to the model. Such methods typically build up a database of example data and compare new data to the database using a similarity measure to find the best match and make a prediction. The focus is on representation of the stored instances and similarity measures used between instances.

Decision tree methods construct a model of decisions made based on the values of the attributes in the data. Decisions fork at each level in the tree until a leaf node is reached, where a prediction decision is made based on the training cases that reached the same leaf node. Decision trees are trained on data for classification and regression problems [36].

CHAPTER 4

Experiments

4.1 Overview

The goal of my experiments is to compare the actual speedup of using different optimization and the speedup predicted by the machine learning tool. If the predicted speedup is reasonably close to the actual speedup then it means that by accurate prediction of our tool we can suggest the user which optimization or any combination of them can be used to get higher speedup. In this project, all needed experiments had been divided to three major phases, *i)* profiling, *ii)* machine learning (training the model) and *iii)* analyzing outputs. In profiling phase, I had profiled the BH [35] and NB [35] code with different number of inputs with/without using optimizations to have some numbers as performance indicator. These numbers help us studying the manner and relevance of different optimizations and how much they affect the performance. Different machine learning models can be made out of the data produced in profiling phase. Then by running different codes as test instances and letting the machine learning tool predict the speedup of that test instance based on different models, I became able to compare the actual speedup and expected speedup to see how much accurate our tool is.

I used NVCC V6.0.1 to compile CUDA codes on NVIDIA Tesla K20 GPU device. I wrote all of needed script for running the experiments, parsing and formatting output files using Python 2.6.6. Also for drawing charts I used R-tool V3.0.2.

4.2 Profiling

Using *nvprof* of NVIDIA Visual Profiler V6.5, BH and NB codes have to be profiled and I chose them because BH is highly irregular and could be a good candidate to represent all type of irregularity in the GPU code and NB is the same problem but with complete regular implementation. Profiling these two codes would show us the difference of performance characteristics of each code. Their profiling data is my main dataset that I perform the experiments and evaluate the power of the proposed tool. During the experiments, I wanted to collect different information about the kernels. The more information I have, the better model I can create based on that information. Using a command from *nvprof,* I found 250 metrics or events that can be calculated and supported by this tool. Although it takes a while to measure all available attributes, I decided to measure them because I did not know exactly which events/metrics are useful in next phase and could be a feature in the machine learning process. Once I figured out which features are playing more important role in prediction tool for making decision, I can narrow down the list of events/metrics to the ones that help us more for better prediction.

4.3 Optimizations

The two programs had been modified in a way that makes it possible to individually enable or disable specific optimizations. For NB, we chose the following six code optimizations:

1. FTZ is a compiler flag that allows the GPU to flush denormal numbers to zero when executing floating-point operations, which results in faster computations. While strictly speaking not a code optimization, the same effect can be achieved by using appropriate intrinsic functions in the source code.
2. RSQRT uses the CUDA intrinsic "rsqrtf()" to quickly compute one over square root instead of using the slower but slightly more precise "1.0f / sqrtf()" expression.
3. CONST copies immutable kernel parameters once into the GPU's constant memory rather than passing them every time a kernel is called, i.e., it lowers the calling overhead.
4. PEEL separates the innermost loop of the force calculation into two consecutive loops, one of which has a known iteration count and can therefore presumably be better optimized by the compiler. The second loop performs the remaining iterations.
5. SHMEM employs blocking, i.e., it preloads chunks of data into the shared memory, operates exclusively on this data, then moves on to the next chunk. This drastically reduces the number of global memory accesses.
6. UNROLL uses a pragma to request unrolling of the innermost loop(s). Unrolling often allows the compiler to schedule instructions better and to eliminate redundancies, thus improving performance.

For BH, we selected the following six source-code optimizations.

1. VOTE employs thread voting instead of a shared-memory-based code sequence to perform a 32-element reduction.

2. WARP switches from a thread-based to a warp-based implementation that is much more efficient because it does not suffer from branch divergence and uses less memory as it records certain information on a per warp instead of a per thread basis.
3. SORT approximately sorts the bodies by spatial distance to minimize the tree prefix that needs to be traversed during the force calculation.
4. RSQRT is identical to its NB counterpart.
5. FTZ is also identical to the corresponding NB optimization.
6. VOLA strategically copies some volatile variables into non-volatile variables and uses those in code regions where it is known (due to lockstep execution of threads in a warp) that no other thread can have updated the value. This optimization reduces memory accesses.

4.4 Naming

I compiled each of NB/BH code with different combination of optimizations, different inputs (different number of bodies and different number of time-steps). Also in my experiments I used replication methods to ensure consistency and improve reliability. Any unexpected event or interruption on the device that I was running the experiments on could affect the results and makes them less precise. Because of that, I profiled each of the executables 3 times to make sure that I would get accurate results.

First of all for ease of reading and finding desired file, I converted the presence/absence of each of six optimizations for each code into a bit-string of 0s and 1s with length of 6. The order of different optimizations for different codes are shown in Table 4.1

Code	Bit 5 Most Significant	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0 least Significant
BH	VOTE	WARP	SORT	RSQRT	FTZ	VOLA
NB	FTZ	RSQRT	CONST	PEEL	SHMEM	UNROLL

Table 4.1 Order of optimizations in bit-string

For example, the bit-string 000101 for BH code shows that the optimizations RSQRT and VOLA had been used.

For BH and NB, I used different number of bodies and time-steps as shown in Table 4.2. I tried to choose meaningful and scalable size of inputs in order to make the code use enough resources to keep our analysis more accurate and complete. For BH code which is an irregular GPU code, I profiled the code with 3 different numbers of bodies and also with half of them to see how much decreasing inputs to half, affects the results.

NB		BH	
Body	Time-step	Body	Time-step
50,000	2	125,000	2
100,000	2	250,000	2
100,000	5	250,000	5
200,000	5	500,000	5
-	-	500,000	10
-	-	1,000,000	10

Table 4.2 Inputs for profiling BH and NB codes

For each set of inputs, I had profiled the code with different combination of six optimizations (64 different combinations in total).

The profiling information contains values such as number of cache hits/misses or amount of data transfer (bits) which are highly dependent to the size of input. In order to make the profiling information comparable to each other regardless of input sizes, the dataset need to be normalized. Normalizing data adjusts values measured on different scales to a notionally common scale. For normalizing profiled data, I used $V_n = V_o/A * C$ where V_o is the value of each attribute before normalization, V_n is the value of corresponding attribute after normalization, A is the number of active cycles and C is a constant. Using this equation, the values are input size-independent and all data would be measured on same scale.

Once I got all the results from profiling phase, I can train and create models based on the profiling data (training dataset) and predict the speedup of any test instance (testing dataset). But before testing unknown test instances on the training model, we need to make sure that our model is valid. Cross validation is one of the techniques helps us validate the model.

4.5 Cross Validation

Cross-validation is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent data set. It is mainly used in systems where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice. The goal of cross validation is to define a dataset to "test" the model in the training phase give an insight on how the model will generalize to an independent data set.

One round of cross-validation involves partitioning a sample of data into complementary subsets, performing the analysis on one subset (called the training set), and validating the analysis on the other subset (called the validation set or testing set). To reduce variability, multiple rounds of cross-validation are performed using different partitions, and the validation results are averaged over the rounds.

In my experiments I tried to train and test on different datasets to see how accurate our model is. As I mentioned above, for each set of inputs I did each profiling experiment 3 times. Also for each set of inputs, I profiled the codes for all possible combination of six optimizations (64 combinations). So I have 192 total number of files after profiling for each input size. Shown in Table 4.3, different experiments for evaluating the tool have been set up for BH and NB code.

Experiment	Training Dataset	Dataset Entries	Testing Dataset	Dataset Entries	Tests includes Trainings	Train and test on same input
1	BH	64	BH	192	Yes	Yes
2	BH	64	BH	128	No	Yes
3	BH	128	BH	64	No	Yes
4	BH	192	BH	64	No	No
5	BH	192	NB	192	No	No
6	NB	192	BH	192	No	No

Table 4.3 Different experiments for evaluating the tool

Our machine learning method leverages the algorithms implemented in Weka, a tool for performing data mining tasks [37]. I applied this method to the dataset that had been produced in profiling phase. This tool predicts the speedup that we expect to get applying different optimization flags. For each test instance, this tool using four different prediction methods (Regression, IBK, M5P and Odds Ratio) and predict the speedup of using each of 6 different optimization flags. Afterwards, the expected speedup could be compared to the actual speedup.

4.6 Analyzing Output

As mentioned above, the ML tool predicts speedup for each test instance in case of using different flags. The generated output by the ML tool tells us how much speedup we should get if we use those flags. For instance, if the generated value for the flag WARP is 1.73, it means that if we enable the flag WARP and compile the BH code then run it with 125000 bodies and 2 time-steps, we expect the runtime to be 1.73 times faster than doing the same thing but with disabled WARP flag. Then by comparing the predicted value 1.73 with the real speedup, we can measure the accuracy of the ML tool. If the prediction is relatively close to the real speedup then we can suggest to the user to use the mentioned flag in order to get optimized results.

CHAPTER 5

Evaluation, Results, and Analysis

5.1 Overview

This chapter presents the results of the experiments I performed to evaluate the prediction accuracy of the proposed approach. As discussed previously, the output of the tool I wrote are predictions of the speedups we expect to obtain when using different source-code optimizations. To validate the results, I compared the predicted with the actual speedup in many scenarios.

The experiments include different experiments of cross validation. In experiments 1 through 4, I trained and tested the model based on the BH code. In experiments 5 and 6, the model is trained on BH/NB and tested on NB/BH. The number of data entries in each training/testing dataset are multiples of 64 because I opted to include all 64 combinations of the investigated optimizations in each dataset. The following subsections provide more detail about the different experiments and strategies.

Each tuple $< n, t >$ corresponds to 64 data-file entries where n is the input size and t is number of the run. The strategy that I chose for evaluating and comparing the results is the following. For each specific optimization and tuple $< n, t >$, I removed all entries that included this optimization, which always leaves 32 entries that do not include the optimization. Testing on the trained model generates 6 different predicted speedups, one for each of the studied optimizations. The predicted speedup values are then compared to the actual (measured) speedup when actually including this optimization in the code.

Calculating the ratio of the actual speedup (AC) over the expected speedup (EX) shows how close the prediction is to the real speedup. If the predictions are accurate, the ML tool can use them to rank the different optimization, that is, suggest the most promising optimizations (if any) to the user based on the expected speedup.

I show the results in form of *strip* charts. A strip chart plots the data along a line with each data point represented by a star. It is often used for showing the density and distribution of data. For each training model, the resulting strip chart shows 32 data points that represent the ratio of the actual speedup over the expected (predicted) speedup. Note that the speedup predictions do not have to be 100% accurate for the tool to work well. As long as the speedups are approximately correct, the tool will recommend the appropriate source-code optimizations, if any.

5.2 Experiment 1

For each set of inputs, I performed three runs, i.e., I profiled the BH code three times. In the first experiment, I trained the model based on the 64 data-files from a single run and tested all 192 files, including the training data, on the resulting ML model. As I trained and tested on the same dataset, I expected the predictions to be accurate. Each machine learning method used in the prediction tool typically yields a different predicted speedup. To improve readability, I only show the results from the one or two best-performing ML models.

The Y axis of the result charts is the ratio of the Actual Speedup (AC) over the Expected Speedup (EX). The closer the data-points are to 1.0 the more accurate the prediction is. The X axis represents the tuples < *Tr*, *Ts* > where *Tr* is the training model and *Ts* is testing dataset.

Figure 5.1 shows the results of experiment 1 for the six optimizations using the IBK method. As expected, the predictions are very close to the actual speedup.

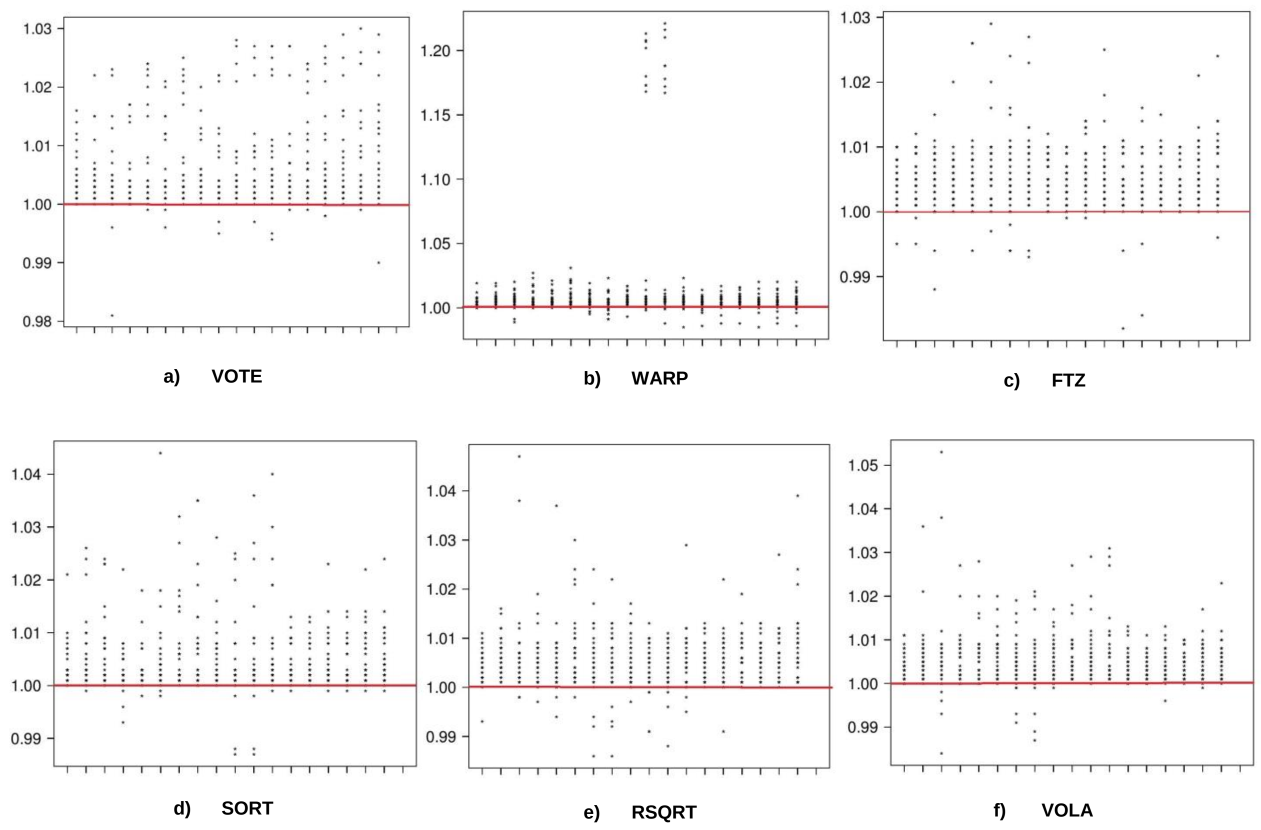


Figure 5.1: Ratios (AC/EX), Experiment 1, IBK method

Most of the data points are above the red line, which means the ratio of the actual speedup over the predicted speedup is greater than 1.0. In fact, almost 95% of the predictions made by the IBK method are less than the actual speedup and are, therefore, underestimated. Nevertheless, all of the data-points fall into the range (0.8, 1.2), which shows that the predictions are close to the actual speedup in all cases. This is expected since the test instances include the training dataset.

Sometimes, using certain optimizations might hurt performance, leading to a slowdown. I compared the actual speedup with the predicted speedup for each optimization to see if they both increase (PP) or decrease (NN) the performance. If both the predicted and the actual speedup are greater than one, it is correct for the recommendation tool to predict a performance gain and thus to recommend the optimization. Similarly, if both the predicted and the actual speedup are less than 1.0, using that optimization would hurt performance and not recommending the corresponding optimization by the tool is the correct behavior. Table 5.1 shows how often the expected speedup and the actual speedup are either both above 1.0 or both below 1.0. Using this metric and the IBK method, on average over 97% of the predictions match the actual behavior. The PN and NP columns in this table show how often the expected speedup and the actual speedup are opposite (i.e., false positives and false negatives).

	PP	PN	NP	NN	Accuracy
VOTE	75.0	0.0	0.0	25.0	100.0
WARP	100.0	0.0	0.0	0.0	100.0
SORT	85.4	0.0	0.0	14.6	100.0
RSQRT	90.3	4.9	0.7	4.2	94.4
FTZ	52.3	7.3	3.6	36.8	89.1
VOLA	100.0	0.0	0.0	0.0	100.0

Total	97.3

Table 5.1: Accuracy of predicted behavior, Experiment 1, IBK method

Figure 5.2 shows the results when using the M5P method for making the predictions. Comparing with the IBK results above, it seems that IBK performs better. For example, the range of the ratios for VOTE is (0.99, 1.03) using IBK and (0.4, 1.5) using M5P. Unlike IBK, which makes mostly underestimated predictions, the ratios of M5P are about evenly distributed below and above 1.0.

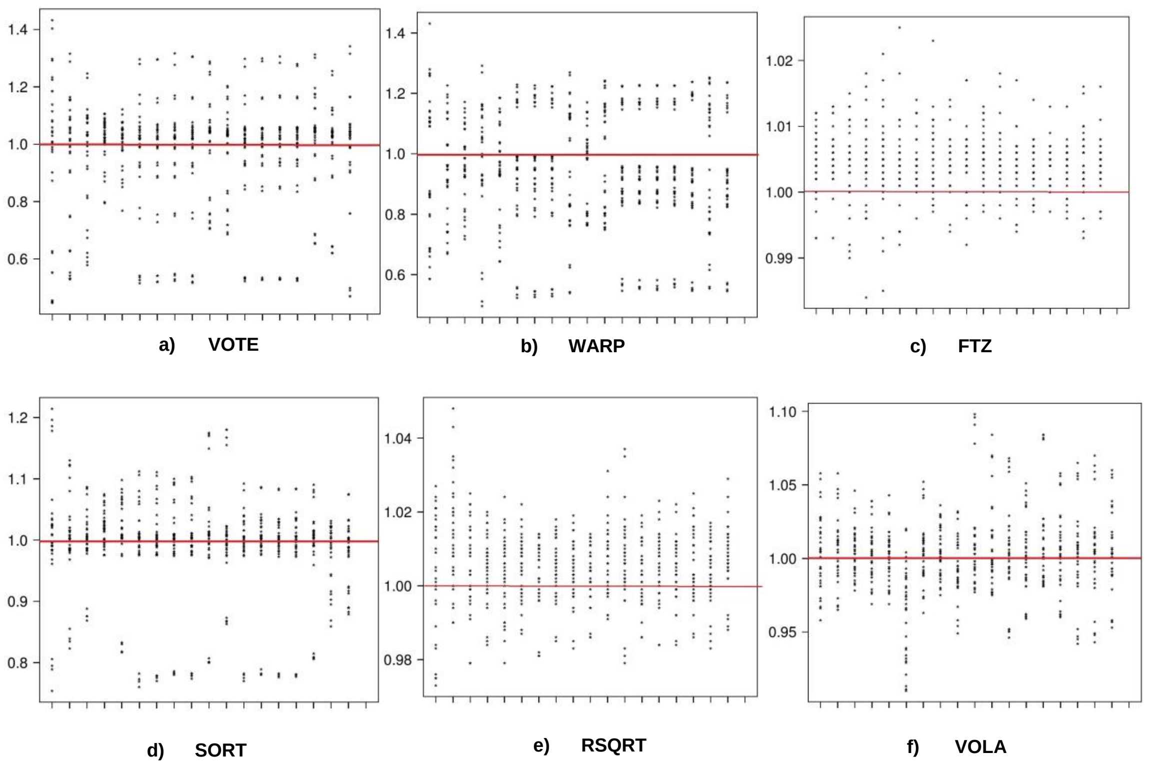


Figure 5.2: Ratios (AC/EX), Experiment 1, M5P method

The accuracy of the predicted behavior is also better in IBK compared to M5P. Table 5.2 shows the percentage of accurate predictions for each optimization. Only 57% of the FTZ behavior is correctly predicted, which severely hurts the overall accuracy. Note that FTZ does not typically help or hurt performance, much making it hard to predict its behavior correctly.

	PP	PN	NP	NN	Accuracy
VOTE	69.8	5.2	0.7	24.3	94.1
WARP	94.1	5.9	0.0	0.0	94.1
SORT	80.6	4.9	11.1	3.5	84.0
RSQRT	85.4	9.7	1.4	3.5	88.9
FTZ	17.4	42.2	0.7	39.8	57.1
VOLA	100.0	0.0	0.0	0.0	100.0

Total	86.4

Table 5.2: Accuracy of predicted behavior, Experiment 1, M5P method

The remaining ML methods, Linear Regression and Odds Ratio, result in worse performance and are not interesting enough to be shown here.

5.3 Experiment 2

In this experiment, I trained the model on the 64 data-files of a single run and tested the other 128 files obtained from the two other runs. Although the training data is not included in the testing data, the results are expected to be accurate because all files stem from the same program running the same inputs multiple times. Figure 5.3 and Table 5.3 show the results for the six optimizations using the IBK method for experiment 2.

The results are almost identical to experiment 1 with just a slight decrease in accuracy due to excluding the training data from the testing dataset. The results for the M5P method are also very similar to those of experiment 1. The M5P method uses just a few features so excluding the training data from the testing dataset does not affect the results significantly.

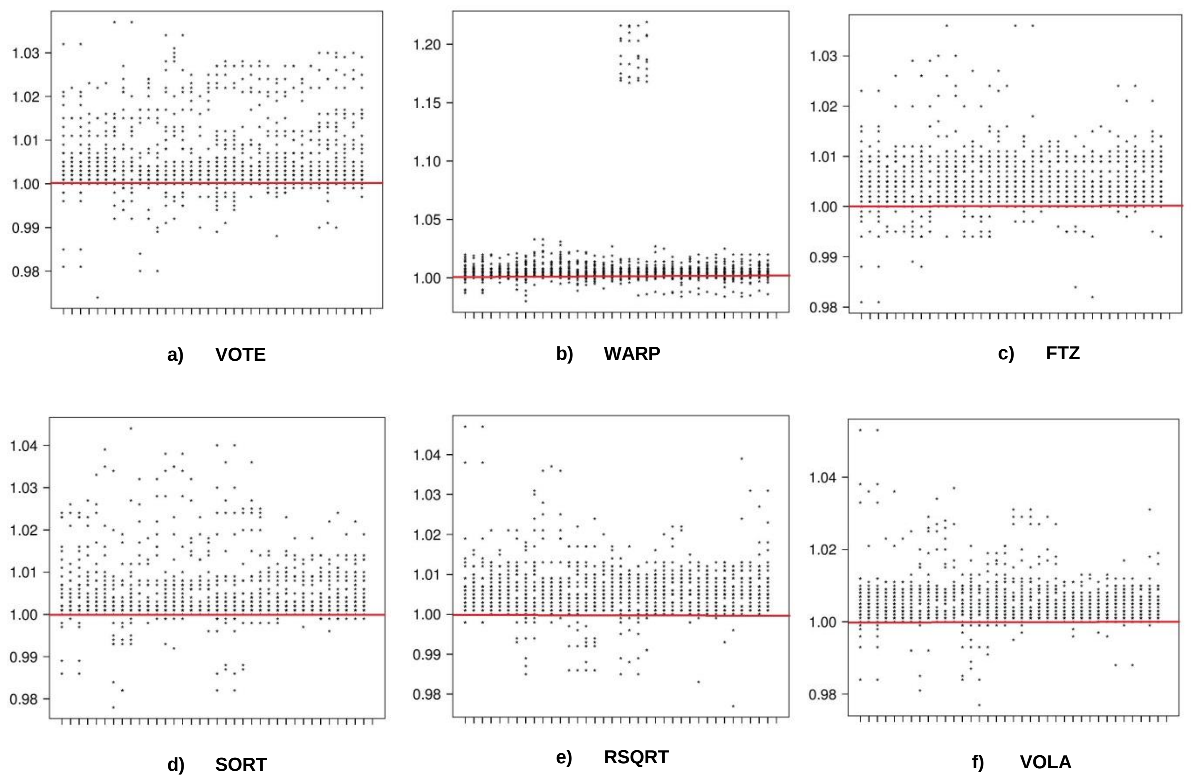


Figure 5.3: Ratios (AC/EX), Experiment 2, IBK method

	PP	PN	NP	NN	Accuracy
VOTE	75.0	0.0	0.0	25.0	100.0
WARP	100.0	0.0	0.0	0.0	100.0
SORT	85.4	0.0	0.0	14.6	100.0
RSQRT	89.1	6.1	1.6	3.3	92.4
FTZ	49.7	9.8	6.3	34.1	83.9
VOLA	100.0	0.0	0.0	0.0	100.0

Total	96.0

Table 5.3: Accuracy of predicted behavior, Experiment 2, IBK method

5.4 Experiment 3

In experiment 3, I trained the model on 128 files and tested on other 64 (experiment 2 used the opposite approach). The hope is that using more training data will improve the resulting model. Figure 5.4 and Table 5.4 show the results for the six optimizations using the IBK method for experiment 3.

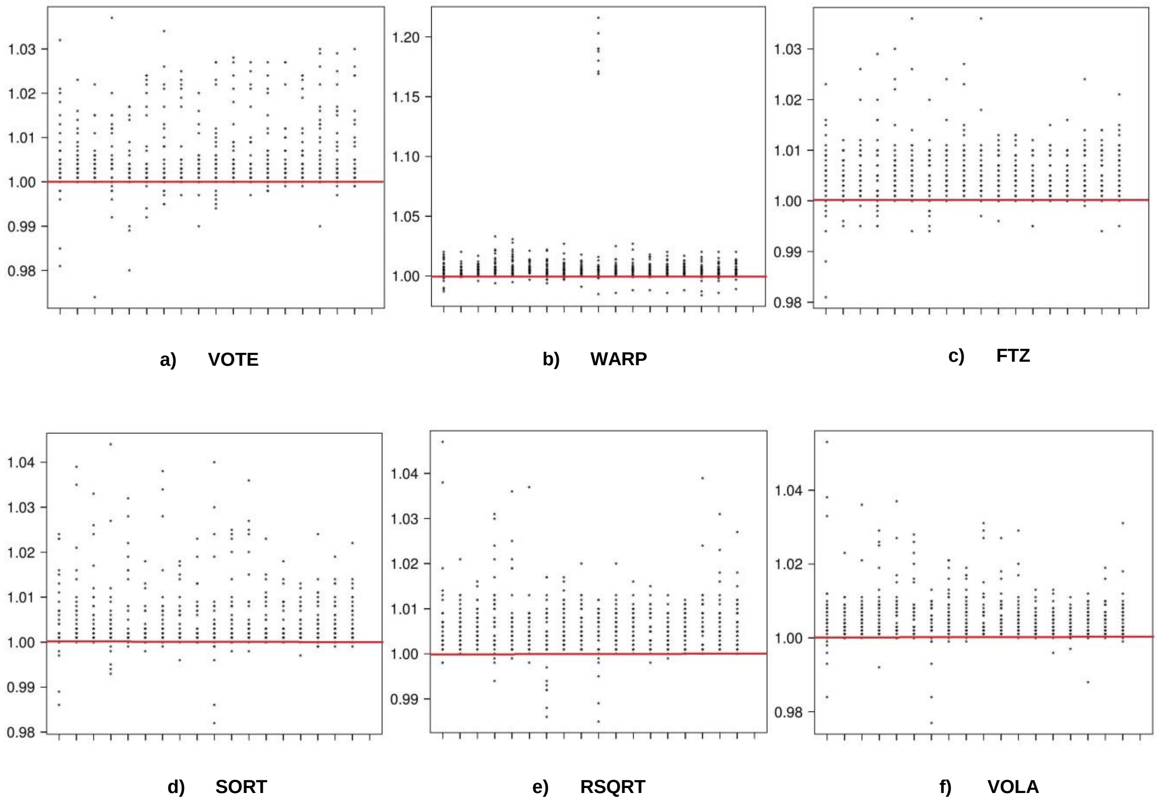


Figure 5.4: Ratios (AC/EX), Experiment 3, IBK method

These results are comparable to the results from the previous experiments in terms of underestimating the speedup and the prediction accuracies. In all cases, the range of the ratios is (0.95, 1.05). If the few outliers in WARP (Figure 5.4.b) are ignored, the results reveal that adding 64 more data-files to the training dataset does not have a significant impact on IBK's predictions. The same is true for the M5P method.

	PP	PN	NP	NN	Accuracy
VOTE	75.0	0.0	0.0	25.0	100.0
WARP	100.0	0.0	0.0	0.0	100.0
SORT	85.4	0.0	0.0	14.6	100.0
RSQRT	89.2	5.9	1.0	3.8	93.1
FTZ	48.6	10.9	4.5	35.9	84.5
VOLA	100.0	0.0	0.0	0.0	100.0

Total	96.3

Table 5.4: Accuracy of predicted behavior, Experiment 2, IBK method

Obtaining about 96% accuracy in the first 3 experiments is expected because training and testing on almost identical data makes the results accurate. In the following experiments, the training dataset is different from the testing dataset in both the program inputs and the programs themselves.

5.5 Experiment 4

In experiment 4, I trained the model with all 192 data-files from one program input and tested on 64 files from each of the other program inputs. Figure 5.5 shows the results of the VOTE optimization with the IBK method. Most of the ratios are around 1.0, meaning that the predicted speedups are close to the actual speedup when adding the VOTE optimization. Unlike in the 3 previous experiments, where most of the IBK ratios were above 1.0, in this experiment the ratios are distributed quite evenly above and below the line. This is also true for the other optimizations (Figure 5.6 and Figure 5.7). The few outliers in Figure 5.5 stem from test cases using the smallest input size, which apparently result in sufficiently different performance metrics to throw off IBK.

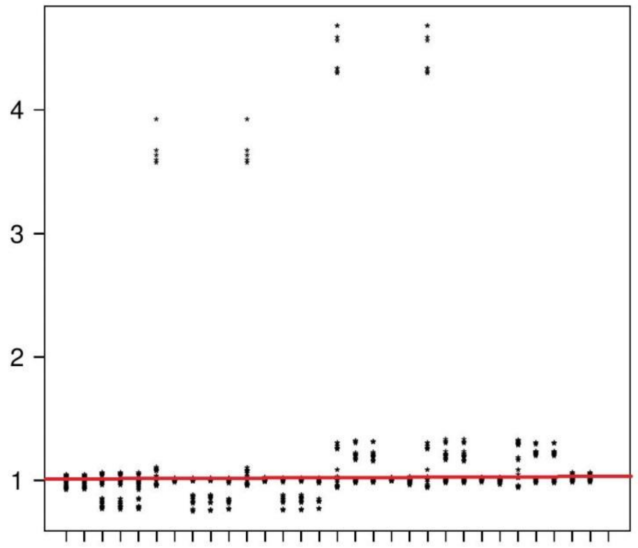


Figure 5.5: Ratios (AC/EX) of VOTE, Experiment 4, IBK method

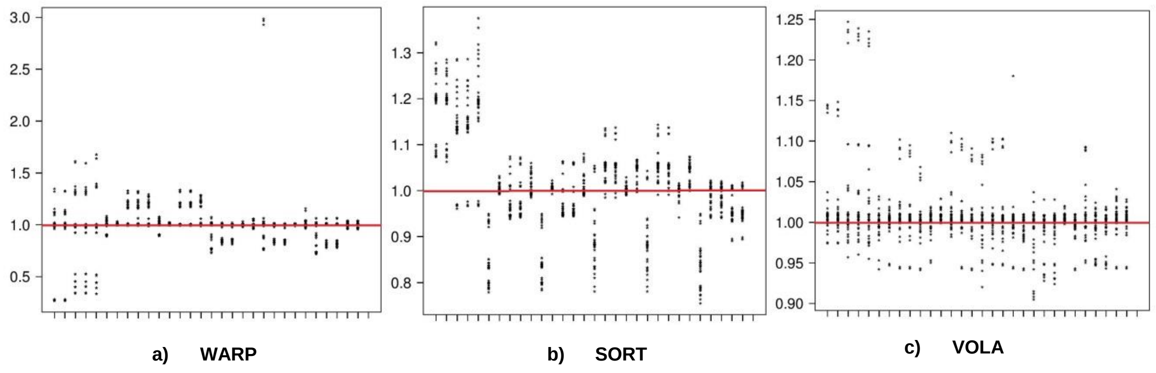


Figure 5.6: Ratios (AC/EX) of WARP, SORT, and VOLA, Experiment 4, IBK method

For the optimizations WARP, SORT, and VOLA, the predictions on smaller inputs are also less accurate than using larger inputs. The plotted ratios are denser close to the 1.0 line for all 3 optimizations in Figure 5.6 because of the higher accuracy with larger inputs. Nevertheless, the results are promising because the range of the ratios in all of these cases is (0.1, 2), which means the tool's prediction speedups are reasonably close to the actual speedups.

Figure 5.7 shows the ratios for FTZ and RSQRT using IBK. The range of the ratios is (0.98, 1.04) and quite evenly distributed regardless of the input size and test cases.

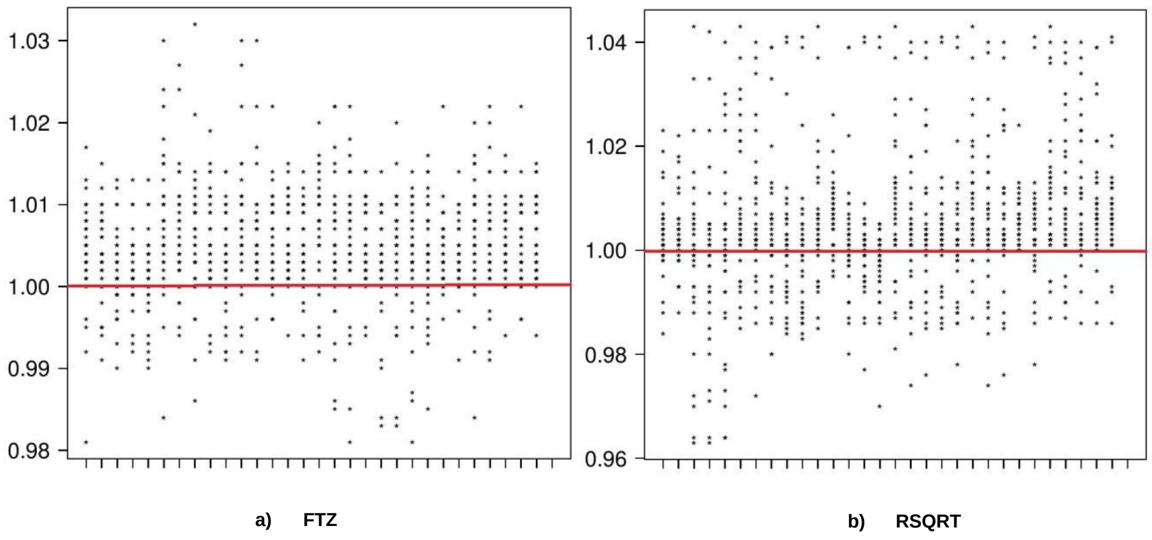


Figure 5.7: Ratios (AC/EX) of FTZ and RSQRT, Experiment 4, IBK method

Table 5.4 shows that the accuracy of positive/negative speedup is still 92% on average. Clearly, training the model on data from one input and testing on data from a different input does not hurt the model's performance substantially.

Figure 5.8 shows the ratios of the predicted speedup over the actual speedup using the M5P method. The ratios are more densely clustered around the 1.0 line than they are for the IBK method. Moreover, the range of ratios has a tighter bound (0.5, 1.5).

	PP	PN	NP	NN	Accuracy
VOTE	72.3	2.7	0.0	25.0	97.3
WARP	100.0	0.0	0.0	0.0	100.0
SORT	81.7	3.8	1.3	13.3	95.0
RSQRT	87.6	7.7	3.2	1.5	89.1
FTZ	44.3	15.1	14.1	26.6	70.8
VOLA	100.0	0.0	0.0	0.0	100.0

Total	92.0

Table 5.5: Accuracy of predicted behavior, Experiment 4, IBK method

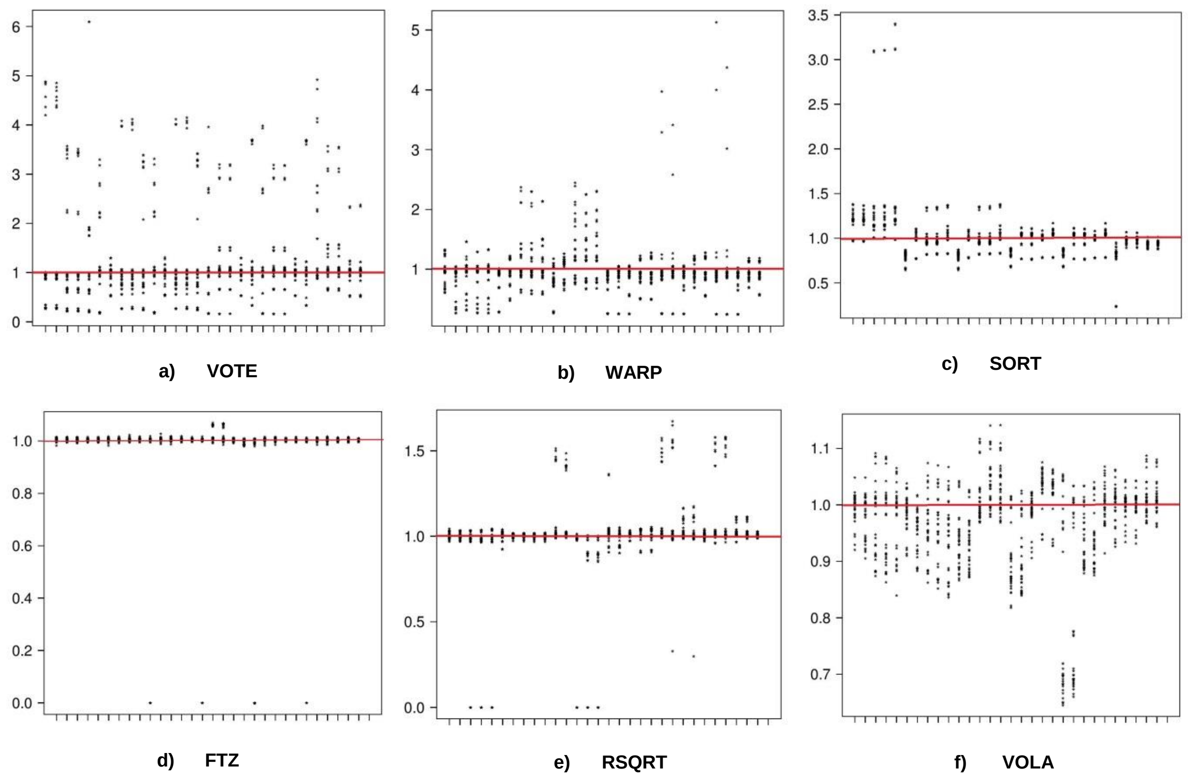


Figure 5.8: Ratios (AC/EX), Experiment 4, M5P method

However, as it is shown in Table 5.4, the accuracy of the prediction behavior is lower than IBK's. This difference between ratios and behavior prediction accuracy shows that the ratio of the actual speedup over the predicted speedup can be close to 1.0 yet the predicted speedup lies on the “other” side of the 1.0 line than the actual speedup. Fortunately, such cases are easily avoided in the recommendation tool by not suggesting optimizations that only result in a small speedup above 1.0.

	PP	PN	NP	NN	Accuracy
VOTE	59.8	15.2	11.7	13.3	73.1
WARP	96.0	4.0	0.0	0.0	96.0
SORT	80.0	5.4	12.1	2.5	82.5
RSQRT	80.8	14.5	2.3	2.4	83.2
FTZ	16.9	42.5	3.0	37.6	54.5
VOLA	100.0	0.0	0.0	0.0	100.0

Total	81.6

Table 5.6: Accuracy of predicted behavior, Experiment 4, M5P method

5.6 Experiment 5

Training the model on a set of programs and testing it on a different program is the ultimate test of my approach. In this experiment, I trained the model on different versions of the BH code and used various versions of the NB code as test cases. This experiment shows much of the results change when I train the model based on data from an irregular GPU code and test it on a regular GPU code. Note that the FTZ and RSQRT optimizations are common to both BH and NB. Hence, I only compare the predicted and actual speedups of these two optimizations as I do not know the actual speedups of the remaining four optimizations when they are applied to NB.

Figure 5.9 shows the results of this experiment using IBK. Almost half of the ratios are below the 1.0 line. The range of the ratios for FTZ is (0.2, 1.7), which shows that the prediction accuracy of the speedup is not as close as it was in the previous experiments (Figure 5.9.a). For RSQRT, the range of the speedups is (0, 3.5), which means there is a significant difference between the actual and the predicted speedup in many cases. As before, the prediction results for test cases with larger input sizes tends to be better. For each model, I tested all 64 data-files of each set of four inputs on the NB code. As GPU codes generally have better performance on large inputs, getting better results for training and testing the model on larger inputs is beneficial.

Considering that I am training and testing on two totally different codes, the results are still promising. As shown in Table 5.5, the accuracy of the predictions for these two optimizations is almost 84%. This means that the tool's suggestions on these optimizations to the user are correct 84% of the time.

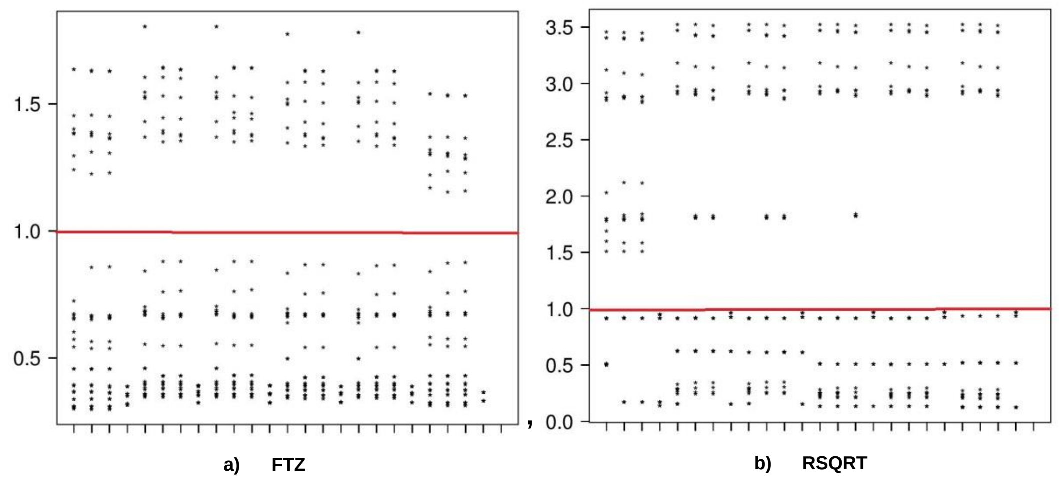


Figure 5.9: Ratios (AC/EX) of FTZ and RSQRT, Experiment 5, IBK method

The accuracy of the M5P method in this experiment for FTZ and RSQRT is only 33%. The reason of this low accuracy is that M5P uses very few features for making decisions. When the training and testing dataset are from different programs, the possibility of accurate predictions based on just a few features is relatively low.

	PP	PN	NP	NN	Accuracy
FTZ	85.9	0.0	14.1	0.0	85.9
RSQRT	81.3	0.0	18.8	0.0	81.3

Total	83.6

Table 5.7: Accuracy of predicted behavior, Experiment 5, IBK method

5.7 Experiment 6

This final experiment is identical to experiment 5 except I switched the training and testing datasets. Hence, I trained the model on data from the regular NB code and tested it on data from the irregular BH code.

Interestingly, all of the predicted speedups for FTZ using the IBK method are lower than the actual speedups of using FTZ on the BH code because all of the ratios are in the range (0, 0.7) as shown Figure 5.10.a. Moreover, RSQRT has better predicted values in this experiment rather than the previous one (Figure 5.10.b). The range of values of ratios is (0.78, 1.45) and most of the ratios are close to line 1.0. For smaller size of input of training and testing dataset the prediction tool overestimated the speedups.

Table 5.5 shows the accuracy of the predicted values. Comparing the results of the IBK method of this experiment with the corresponding results from the previous experiment, I find that more accurate predictions are made by the tool when the model is trained on irregular code and tested on regular code, which makes sense as irregular codes tend to be more complex.

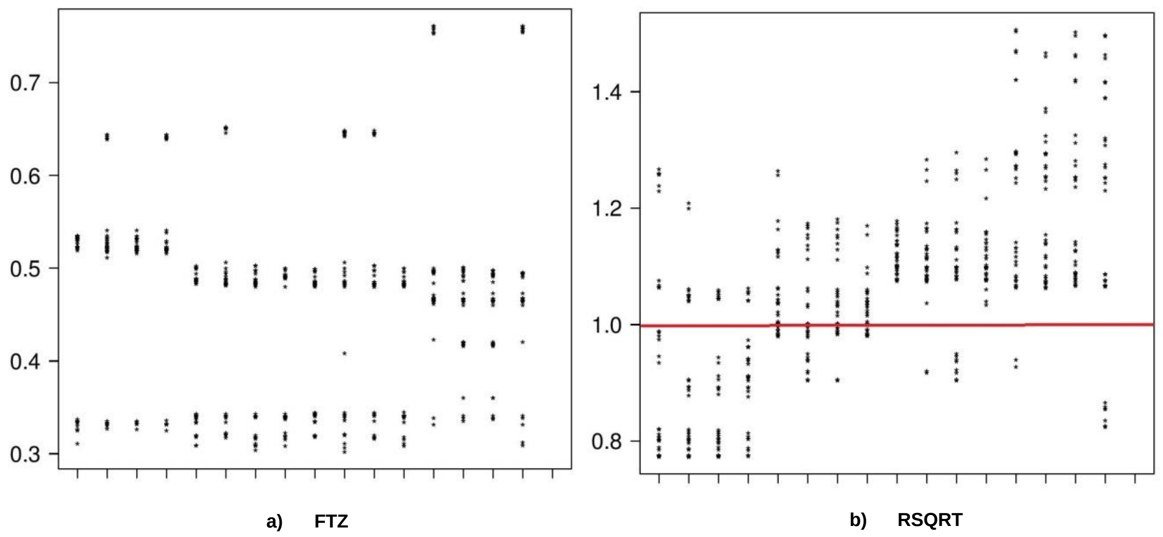


Figure 5.10: Ratios (AC/EX), Experiment 6, IBK method

	PP	PN	NP	NN	Accuracy
RSQRT	51	46.7	0.6	1.8	52.7
FTZ	58.6	0	41.4	0	58.6

Total	55.7

Table 5.8: Accuracy of predicted behavior, Experiment 6, IBK method

There is another important difference between this experiment and all previous experiments. In the first five experiments, the prediction accuracy of IBK is better than that of M5P. However, as shown in Table 5.6, in experiment 6 the overall accuracy of M5P is better than IBK. Hence, there is no one ML model that is always better for my tool. M5P yields better performance because it narrows the features down to a few metrics, which are significant for both irregular and regular codes.

	PP	PN	NP	NN	Accuracy
RSQRT	74.8	22.9	1.4	1	75.8
FTZ	7.6	51	4.7	36.7	44.3

Total	60.1

Table 5.9: Accuracy of predicted behavior, Experiment 6, M5P method

In almost all of experiments, the IBK method of the prediction tool generated better expected speedup than other methods. The reason is IBK method is an instance-based model which builds the model at testing time based on the test instance attributes in addition to the training data attributes. Also as it is shown in almost all of strip charts, the dense of data-points (AC/EX) is more around line 1.0 for larger input sizes.

CHAPTER 6

Summary and Conclusion

6.1 Summary

In this thesis, I designed a tool to suggest optimization hints to the user in order to improve the efficiency of GPU code with an emphasis on irregular codes. The tool needs to be trained on performance data from different programs that do and do not include certain source-code optimizations. During this training, the tool builds machine-learning models for each optimization so that it can later estimate the speedup for each optimization when presented with performance data from other programs. To measure and quantify the performance, I profile different GPU codes with/without certain optimizations and with multiple inputs to gather a large set of performance data. The tool is able to rank the optimizations based on their predicted speedup and suggests the top optimizations to the user if their predicted speedup is above a given threshold. To evaluate the accuracy of the predicted speedups, I compared them to the actual speedups obtained when adding the various source-code optimizations to the code. I performed six different performance evaluation experiments of training models and predicting speedups. In first four experiments, I trained and tested the model on the BH code and obtained up to 97% prediction accuracy. In the remaining two experiments, where I train the model on BH/NB and test it on NB/BH, the tool delivers up to 82% of accuracy.

6.2 Conclusion

Based on the results shown in Chapter 5, the predictions of the machine learning tool are more precise when I trained the model on data-files obtained with larger program input sizes. This makes sense as larger inputs result in more profiling data and more stable-state utilization of the GPU. Expectedly, training the model with more data yields better predictions.

When training the model on code that is different from the tested code, I found that model training based on irregular codes and testing on regular codes results in better predictions than training on regular code and testing on irregular codes. This is likely a combination of two factors. First, regular codes are less complex, making them easier to predict in general. Second, the higher complexity of irregular codes probably provides more diverse training data, which results in better trained ML models.

I studied four different machine learning methods for making the predictions. My results show that there is no such thing as one clear winner. Nevertheless, IBK generally performs very well when predicting the likely speedup of source-code optimizations.

6.3 Future Work

In this thesis, I used differently optimized Barnes-Hut implementations of the n-body problem as a representative irregular GPU code. Of course, using additional (irregular) codes for training would be better. Also, I studied six source-code optimizations as a proof of concept. Large numbers of optimizations can and should be used to better test the capabilities of the tool and to validate the approach. Finally, I used six different inputs. Especially on irregular codes, which can be input sensitive, profiling with more diverse inputs might be beneficial to see how much the inputs affect the accuracy of the predictions.

For the machine learning phase, I investigated four different methods. Other types of machine learning methods could, of course, be employed for predicting the speedups.

LITERATURE CITED

[1] B. P. Miller and J. K. Hollingworth and M. D. Callaghan, The Paradyn Performance Tools and PVM, Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing: Townsend, TN, USA, 25–27 May 1994, pp. 201-210, Society for Industrial and Applied Mathematics, 1994.

[2] Bernd Mohr and Felix Wolf, KOJAK - a tool set for automatic performance analysis of parallel programs, Springer-Verlag, 2003.

[3] Zoltán Szebenyi, Brian J. N. Wylie, Felix Wolf: SCALASCA Parallel Performance Analyses of SPEC MPI2007 Applications. In Proc. of the 1st SPEC International Performance Evaluation Workshop (SIPEW), Darmstadt, Germany, volume 5119 of Lecture Notes in Computer Science, pages 99-123, Springer, June 2008.

[4] W. E. Nagel and A. Arnold and M. Weber and H.-Ch. Hoppe and K. Solchenbach, VAMPIR: Visualization and Analysis of MPI Resources, 1996.

[5] Matthias S. Müller and Andreas Knüpfer and Matthias Jurenz and Matthias Lieber and Holger Brunst and Hartmut Mixand Wolfgang E. Nagel, Developing Scalable Applications with Vampir, VampirServer and VampirTrace, PARCO, Advances in Parallel Computing, Vol. 15, pp. 637-644, IOS Press, 2007.

[6] Michael Gerndt and Karl Fürlinger and Edmond Kereku, Periscope: Advanced Techniques for Performance Analysis, PARCO, John von Neumann Institute for Computing Series, Vol. 33, pp. 15-26, Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[7] S. Shende and A. D. Malony, “The TAU Parallel Performance System,” International Journal of High Performance Computing Applications, SAGE Publications, 20(2):287-331, Summer 2006

[8] Allen D. Malony, Scott Biersdorff, Wyatt Spear, and Shangkar Mayanglambam. 2010. An experimental approach to performance measurement of heterogeneous parallel applications using CUDA. In Proceedings of the 24th ACM International Conference on Supercomputing (ICS ‘10). ACM, New York, NY, USA, 127-136. DOI=10.1145/1810085.1810105 http://doi.acm.org/10.1145/1810085.1810105

[9] http://www.vi-hps.org/projects/score-p/

[10] Laksono Adhianto and Sinchan Banerjee and Michael Fagan and Mark Krentel and Gabriel Marin and John Mellor-Crummey and Nathan Tallent, HPCToolkit: Performance tools for parallel scientific computing, SC'08 USB Key, ACM/IEEE, November 2008.

[11] https://developer.nvidia.com/cuda-profiling-tools-interface

[12] Browne, S., Deane, C., Ho, G., Mucci, P. "PAPI: A Portable Interface to Hardware Performance Counters," Proceedings of Department of Defense HPCMP Users Group Conference, June, 1999.

[13] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, Bernd Mohr: The SCALASCA Performance Toolset Architecture. In Proc. of the International Workshop on Scalable Tools for High-End Computing (STHEC), Kos, Greece, pages 51–65, June 2008.

[14] https://developer.nvidia.com/nvidia-visual-profiler

[15] http://www.nvidia.com/object/nsight.html

[16] Michael Knobloch and Timo Minartz and Daniel Molka and Stephan Krempel and Thomas Ludwig 0002 andBernd Mohr, Electronic poster: eeclust: energy-efficient cluster computing, SC Companion, pp. 99-100, ACM, 2011.

[17] http://www.hpctoolkit.org

[18] http://www.vi-hps.org/projects/

[19] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya and Scott Cranford, Analyzing the Performance of Scientific Applications with Open|SpeedShop, Parallel Computational Fluid Dynamics:, to be North-Holland, (November) May 2009.

[20] http://software.intel.com/en-us/intel-vtune-amplifier-xe

[21] https://computing.llnl.gov/tutorials/parallel_comp/#Who

[22] http://en.wikipedia.org/wiki/Graphics_processing_unit

[23] http://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

[24] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A Quantitative Study of Irregular Programs on GPUs. In Proceedings of the 2012 IEEE International Symposium on Workload Characterization, pages 141-151, 2012.

[25] http://www.hpcwire.com/2014/01/09/future-accelerator-programming/

[26] CS 4378T / CS 5351 Spring 2013 Lecture, Martin Burtscher, Texas State University

[27] https://developer.nvidia.com/nvidia-visual-profiler

[28] http://en.wikipedia.org/wiki/N-body_problem

[29] https://www.princeton.edu/~achaney/tmve/wiki100k/docs/N-body_problem.html

[30] http://en.wikipedia.org/wiki/Barnes%E2%80%93Hut_simulation

[31] http://docs.nvidia.com/cuda/profiler-users-guide/#nvprof-overview

[32] http://en.wikipedia.org/wiki/Profiling_(computer_programming)

[33] http://home.wlu.edu/~whaleyt/classes/parallel/topics/amdahl.html

[34] http://en.wikipedia.org/wiki/Machine_learning

[35] http://iss.ices.utexas.edu/?p=projects/galois/lonestargpu

[36] http://machinelearningmastery.com/a-tour-of-machine-learning-algorithms/

[37] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, Ian H. Witten (2009); The WEKA Data Mining Software: An Update; SIGKDD Explorations, Volume 11, Issue 1.