A SAMPLING-BASED APPROACH TO GPGPU PERFORMANCE AUTO-TUNING

by

Cheng Xiang Wilson Feng

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2017 by Cheng Xiang Wilson Feng
Abstract

A Sampling-Based Approach to GPGPU Performance Auto-tuning

Cheng Xiang Wilson Feng
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2017

We present a novel strategy for automatic performance tuning of GPU programs. The strategy combines heuristic search with regression trees, a machine learning model, to prune the optimization space. It samples configurations in the space as training data for a regression tree. It then focuses the search on the tree region with the best mean sample performance. Additional regression trees are built using the cumulative samples collected. This process is repeated until the given time budget is exhausted.

We implement our strategy in OpenTuner, an open source automatic tuning framework. We evaluate the strategy using 8 benchmark GPU programs run on an Nvidia GTX 1060 GPU. We compare the effectiveness of our strategy in obtaining good performing configurations against the state-of-the-art AUC-Bandit strategy used by OpenTuner. Experimental results show that our strategy is more consistently able to obtain better performing configurations compared to OpenTuner’s strategy, by up to 34% averaged over 100 auto-tuning experiments.
Acknowledgements

I would like to thank my supervisor Tarek Abdelrahman for his continuous support and guidance throughout my graduate studies. His experience helped me stay on the correct track for my research and achieve good results. His high expectations in research, presentations, and documentations have helped me grow to become a better overall engineer.

I would also like to thank my colleagues, David Han, Joseph Garvey, Andrew Lee and Leslie Barron for giving me their feedback on this work. Their questions and advice helped me refine this work. Last, special thanks go to my family for their support and encouragement throughout my studies.
Contents

1 Introduction 1
  1.1 Research Overview .................................................. 2
  1.2 Contributions ......................................................... 3
  1.3 Organization .......................................................... 4

2 Background 5
  2.1 GPU Architecture ..................................................... 5
    2.1.1 GPU Memory Hierarchy ........................................... 5
  2.2 GPU Programming Models ............................................. 7
  2.3 OpenTuner ............................................................. 9
  2.4 Optimizations .......................................................... 11
    2.4.1 Work-Group Size ................................................. 13
    2.4.2 Local Memory .................................................... 13
    2.4.3 Loop Tiling ....................................................... 14
    2.4.4 Thread Coarsening .............................................. 14
  2.5 Machine Learning ..................................................... 17
  2.6 Supervised Machine Learning ....................................... 17
  2.7 Regression Trees ..................................................... 17
  2.8 Learning Regression Trees .......................................... 18
  2.9 Use of Regression Trees ............................................ 19

3 Auto-tuning Strategy 21
  3.1 Overall strategy ..................................................... 21
    3.1.1 Sampling Configurations ....................................... 22
    3.1.2 Exploration versus Exploitation for the Region of Interest ... 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.3</td>
<td>Pruning the Optimization Space</td>
<td>24</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Exploring the Pruned Region</td>
<td>25</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Iteratively Pruning the Optimization Space</td>
<td>25</td>
</tr>
<tr>
<td>3.1.6</td>
<td>Algorithm Summary</td>
<td>26</td>
</tr>
<tr>
<td>4</td>
<td>Implementation</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Integrating GPU applications into OpenTuner</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Profiling GPU kernels</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>Inter-parameter Constraints</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Modifications to Benchmark Kernels</td>
<td>33</td>
</tr>
<tr>
<td>4.5</td>
<td>Auto-tuning Strategy Implementation</td>
<td>33</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Resetting the Search Region</td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>Evaluation</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>Evaluation Platform</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>Benchmarks</td>
<td>36</td>
</tr>
<tr>
<td>5.3</td>
<td>Performance Metrics</td>
<td>37</td>
</tr>
<tr>
<td>5.4</td>
<td>Hypothesis Validation</td>
<td>38</td>
</tr>
<tr>
<td>5.5</td>
<td>Determining a Good value for $\sigma$</td>
<td>45</td>
</tr>
<tr>
<td>5.6</td>
<td>Comparing Competitive Strategies</td>
<td>45</td>
</tr>
<tr>
<td>5.7</td>
<td>Methodology</td>
<td>46</td>
</tr>
<tr>
<td>5.8</td>
<td>Results</td>
<td>46</td>
</tr>
<tr>
<td>5.9</td>
<td>Summary</td>
<td>49</td>
</tr>
<tr>
<td>6</td>
<td>Related Work</td>
<td>53</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion</td>
<td>57</td>
</tr>
<tr>
<td>7.1</td>
<td>Future Work</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>59</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The demand for energy-efficient high-performance computing has fueled the use of Graphics Processing Units (GPUs) to accelerate general-purpose computation. Industry leaders such as Nvidia, AMD, and Qualcomm provide GPU solutions catered towards different market domains. For example, Nvidia is able to deliver high performance for application domains such as gaming, supercomputing, and deep learning [1]. AMD GPUs is an alternative from Nvidia GPUs with a lower consumer cost on average [2]. Last, Qualcomm focuses on GPUs for mobile devices [3].

Contrary to conventional Central Processing Units (CPUs), which excel in general purpose computing, GPUs excel in highly parallel application domains with their Single Instruction Multiple Data (SIMD) architecture. Their architecture is built around a large number of arithmetic units and thus enables the same instruction to be replicated and executed on different data, resulting in high energy-efficient performance. For example, Nvidia’s Fermi architecture is able to execute one computing instruction using 200 picojoules. In comparison, an energy efficient x86 CPU require 10x more energy [4].

However, a GPU require programmers to optimize their code in order to exploit the GPU’s underlying architecture and, thus, realize its full potential. The act of writing source code in a specific way such that it exploits the full potential of the underlying hardware, while maintaining functionality is known as, optimizing code. For example, determining whether the amount of work done per thread should be expanded or collapsed is known as, thread coarsening [5]. Partitioning a loop into smaller segments such that the data used in the loop is still within the hardware-managed cache [6] is another optimization called, loop tiling. The above process of optimizing GPU code often necessitates GPU programmers to explore many configurations of their GPU code, i.e., variations of the code with different combinations of optimizations applied to it. This can be a tedious and an error prone process; manually restructuring
source code while guaranteeing correctness of its computation. Further, the optimization space (i.e., the space of all possible configurations) is often prohibitively large to exhaustively explore. These challenges have given rise to interest in auto-tuning frameworks; i.e., frameworks that can automatically determine what optimizations to apply to GPU code to achieve near-optimal performance.

Generally, there are two main approaches to auto-tuning. The first is machine learning based. It uses a set of training GPU programs in order to build a machine learning model that can be used to predict what optimizations to apply when presented with a new program [5, 7]. The second approach uses a heuristic strategy that navigates through the optimization space. Starting with some initial configuration(s), a heuristic is repeatedly applied in order to arrive at better performing ones. Examples of such heuristics include simulated annealing [8], genetic algorithms [9] and Nelder-Mead [10], to name a few. This thesis presents an auto-tuning strategy that uses a heuristic that does not require the programmer to be knowledgeable of the underlying GPU architecture.

1.1 Research Overview

In this thesis, we explore a novel heuristic strategy that is based on the sampling of the configuration space. In this strategy, a random sample of configurations in the space is taken. These configurations are used to narrow the space into a smaller region of interest, \( R \). Additional samples are then taken in \( R \) to further narrow it. This process of sampling and narrowing is repeated until \( R \) is small enough to search, either exhaustively or using a simple heuristic. We demonstrate the viability of this strategy and its competitiveness against other heuristic strategies.

The novelty of our proposed strategy stems from its use of both a machine learning model and a heuristic to prune the optimization space and focus on regions that are more likely to contain better performing configurations. More specifically, we use regression trees [11], a well-known machine learning model in order to narrow the space into a region of interest. This is in contrast to how regression trees are normally used (i.e., as a prediction model [11]). An initial sample of configurations is taken across the entire space. A regression tree is built using this sample, the training data. Our heuristic is to focus on the regression tree’s region with the highest mean sampled performance and make it the region of interest, \( R \). Additional samples are taken in this region and the regression tree is rebuilt using all the samples. This process of building a regression tree and sampling in its “best” region is repeated until either \( R \) is small enough or an allocated auto-tuning time budget is exhausted. The best sampled configuration so far is then reported back by the strategy.

Our strategy is not application domain specific. That is, our strategy do not rely on any underlying
assumption or priori knowledge of the GPU program to be auto-tuned. Both OpenCL and CUDA GPU kernels are supported and used in our evaluation. We use a set of optimizations that are commonly applied to GPU kernels such as work-group size in various dimensions, loop tiling factors, thread coarsening factors and use of various memory types (global and local).

We formalize the auto-tuning strategy based on our use of regression trees. We implement this strategy in OpenTuner, an open-source auto-tuning framework [12]. OpenTuner uses a strategy that incorporates an ensemble of well known optimization techniques shown to be effective in auto-tuning [13] [14]: Nelder Mead and Evolution Algorithms. We extend OpenTuner to support GPU optimizations and our strategy.

Experimental evaluation using 8 benchmarks on an Nvidia GTX 1060 GPU shows that, given an auto-tuning time budget, our strategy is able to result in high performing configurations and that it outperforms the default strategy of OpenTuner. Specifically, our strategy results in better performing configurations earlier in the search compared to OpenTuner. Further, our strategy explores roughly the same number of configurations as that of OpenTuner in less time. Even when our strategy explore more configurations, the cumulative time of the explored configurations is less than that of ones explored by OpenTuner’s strategy. This reflects that our strategy focuses the search on regions of the space that have better performing configurations. Experimental results show that our strategy is more consistently able to obtain better performing configurations compared to OpenTuner’s strategy, by up to 34% averaged over multiple auto-tuning experiments. This was calculated by finding the largest difference in best performance found between our strategy and OpenTuner’s for each benchmark, and averaged over a geometric mean.

1.2 Contributions

The main contributions of this thesis are:

- A performance auto-tuning strategy for GPU applications that uses sampling and regression trees to find high performing configurations.

- An evaluation of the proposed strategy, integrated into an open-source auto-tuning framework, OpenTuner. The evaluation shows that strategy out performs that used by OpenTuner.
1.3 Organization

The remainder of thesis is organized as follows. Chapter 2 describes background material on GPU architectures, optimizations, regression trees and OpenTuner. Chapter 3 formulates our auto-tuning strategy. Chapter 4 discusses some implementation details. Chapter 5 presents our experimental evaluation. Finally, Chapters 6 and 7 describe related work and present concluding remarks.
Chapter 2

Background

2.1 GPU Architecture

GPUs are devices that contain a large number of simple arithmetic units grouped together called, cores. These cores are able to operate in parallel and thus result in high computational throughput. In this work, we primarily use the Nvidia GTX 1060. Thus, we focus on its architecture, which is codenamed Pascal. The cores of the GPU are put into groups known as Streaming Multiprocessors (SM). The main difference between the various models in the GTX 10 series is the number of functional SMs. The GTX 1060 has 10 SMs, while the Pascal architecture have GPUs up to a max of 20 SMs [15]. Figure 2.1 shows a more detailed view of what an SM is composed of. Each SM contains 128 cores. SMs schedule up to 64 groups of 32 threads, known as warps, to the cores and they execute in lock-step. The cores within each SM execute the same instruction simultaneously but potentially on different data, i.e., Single Instruction stream Multiple Data stream (SIMD). Figure 2.2 shows an overview of the SM architecture where warps are scheduled for execution.

2.1.1 GPU Memory Hierarchy

GPUs have a memory hierarchy where at each level can impact the overall performance achieved. The following sections describe each level of the hierarchy: Global Memory, Shared Memory, Texture Memory, Registers.
CHAPTER 2. BACKGROUND

Figure 2.1: Nvidia Pascal SM Architecture. Adapted from Nvidia’s GTX1080 whitepaper \[15\]

Global Memory

Global memory has the largest capacity compared to all the other levels. It is a an off-chip DRAM module and thus has very high access latency that results in slow memory transactions. Note that each requested memory transaction will fetch an aligned chunk of data where it will be cached in the hardware managed L1 and L2 caches. If threads access memory locations that are within the range of the chunk of data that has already been fetched, known as \textit{coalesced accesses}, then the memory transaction time will be reduced thanks to the hardware managed caches \[15\].

Shared Memory

Shared memory is a software managed cache that is within each SM. It has less capacity than global memory, but its transaction speed is faster. This cache allows threads within the same SM to share data with each other, but not across SMs. In the Pascal architecture, each SM contains a 96 KB shared memory unit \[15\].
Figure 2.2: Nvidia Pascal GTX1060 Architecture. Adapted from Nvidia’s GTX1080 whitepaper [15]

Texture Memory

Texture memory is another hardware managed cache. It is read-only and benefits from spatial locality of data. Thus threads in the same warp that read from this cache that are close together in a 2-dimensional fashion may benefit in performance. In the Pascal architecture, each SM contains a total of 48 KB L1 and Texture cache shared [15].

Registers

Registers is the fastest memory in the hierarchy. They are organized in a module called the register file, located within each SM. Registers are thread specific and thus cannot be shared across other threads. Each thread can use up to 255 registers while the register file for each SM contains 65536 registers [15].

2.2 GPU Programming Models

There are two GPU programming models used in this work: the Open Computing Language (OpenCL) [16] and the Compute Unified Device Architecture (CUDA) [17]. OpenCL is an open standard for parallel programming of heterogeneous systems based on the C99 programming language. This programming...
model consists of programs for the host, e.g., a Central Processing Unit (CPU), and to devices, e.g., GPUs. In particular, the program launched on a GPU by the host is called a kernel. Work items execute the kernel program in parallel on the GPU. Work groups are organized into a multidimensional grid. Each work item is uniquely identified by its per-dimension unique ID, called global ID. Within each work group, a work item can also be identified by its per-dimension local ID. These values specified for the multidimensional grid is known as a launch configuration for the kernel that the host provides during kernel launch time. For GPUs, each work item within the same work group is executed on the same SM. Within each work group, work items are able to synchronize with each other and access the same software-managed cache, local memory. Consequently, that means the physical hardware limitations of the SM such as number of registers, and amount of local memory allowed restricts the range on the work group size.

The second programming model created by Nvidia, CUDA, is specifically for programming kernels on Nvidia GPUs. The model supports a variety of programming languages, including C. Its construct is similar to OpenCL while some of the terminologies are renamed. Table 2.1 highlights the terminology equivalence between OpenCL and CUDA.

![2D grid example global and local IDs. The numbers parenthesized represents the dimension.](image)
Table 2.1: Equivalent Terminologies, C for CUDA vs OpenCL. Adapted from AMD [18]

<table>
<thead>
<tr>
<th>C for CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>Work-item</td>
</tr>
<tr>
<td>Thread Block</td>
<td>Work-group</td>
</tr>
<tr>
<td>Constant memory</td>
<td>Constant memory</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Local memory</td>
</tr>
<tr>
<td>Local memory</td>
<td>Private memory</td>
</tr>
</tbody>
</table>

2.3 OpenTuner

OpenTuner [12] is a multi-objective program auto-tuning framework. It allows users to specify their program’s domain-specific optimizations that are of tuning interest. Given an auto-tuning time budget, the framework by default attempts to optimize the program using several search techniques such as Nelder Mead [13] and Evolution Algorithms [14]. Throughout the tuning process, the framework determines which of these techniques are more effective in optimizing the desired objective and allocates more time budget to these techniques compared to techniques that are less effective.

Figure 2.4 shows a high level overview of the different components of OpenTuner. The configuration manipulator specifies the optimizations to be tuned and the range of values their parameters may take. Some example parameter types that are already supported include integer, powers of two, boolean, enumerations and floats. OpenTuner also enables its users to extend the frameworks functionality by defining their own parameter types if needed. The search driver dictates which search techniques to use during the tuning session. The framework contains a repository of different techniques. Multiple techniques can be applied during a tuning session by grouping the techniques into a meta technique. OpenTuner’s default search strategy is a meta technique that incorporates the following search techniques: differential evolution, uniform greedy mutation, normalized greedy mutation, and nelder mead. OpenTuner’s default meta technique determines which of the search techniques to use during its tuning session by applying a heuristic called the multi-armed bandit with sliding window, area under the curve credit assignment (AUC Bandit). This heuristic is based on the multi-armed bandit problem [19]. Users are also able to incorporate their own search techniques into this repository and create their own meta techniques as well.

Similarly, the measurement driver dictates what the tuning objective is, e.g., performance or power. Both the search and measurement components interact with the result database that stores information
Figure 2.4: OpenTuner high level overview [12]

on all the program instances evaluated so far during the tuning session.

OpenTuner requires a few inputs to operate. The first is the auto-tuning time budget, which reflects the maximum amount of time that can be spent on auto-tuning the program. The second is a specification of the optimizations to be tuned, expressed as compiler flags, C preprocessor macros, etc. The third is the set of commands needed to compile and run the program. Finally, there is the option to specify how many instances of the program can be ran in parallel.

The following is an example from the OpenTuner tutorial [12] demonstrating how to tune an application based on its runtime. Figure 2.5 is a parameterized version of a matrix multiply program. The parameter of focus to tune here is the block size, defined by BLOCK_SIZE. First we specify to OpenTuner what the optimization we are trying to tune, and the range of values it can vary as shown in Figure 2.6 from lines 10-18. Next we specify how to compile, and execute the application. Note how the compile and execute commands are in the form of strings. This implies that we are able to have multiple commands sequentially since they are treated as command-line executions. In this particular example, we specify the optimization parameter value during the compilation phase with a flag passed in, -DBLOCKSIZE. The compiled binary is named as, tmp.bin. Once the application has been executed, we measure its runtime and return that result. This is demonstrated from lines 20 to 39. Last, we specify the file name which stores the best configuration found during the tuning session as shown from lines 41 to 44. Assuming the file name is mmm_tuner.py, we run OpenTuner to tune this application by executing the command: python mmm_tuner.py --no-dups --stop-after=30. The no-dups flag hides the warning messages if the framework re-evaluates an existing configuration that has already been explored. The stop-after flag
indicates in seconds what the tuning time budget is. Note there exists an optional flag which specifies explicitly which search technique to use. If nothing is specified, then the framework defaults to the AUC Bandit strategy mentioned above.

```c
#include <stdio.h>
#include <cstdlib>

#define N 100

int main(int argc, const char **argv)
{
    int n = BLOCK_SIZE * (N/BLOCK_SIZE);
    int a[N][N];
    int b[N][N];
    int c[N][N];
    int sum=0;
    for(int k1=0;k1<n;k1+=BLOCK_SIZE)
    {
        for(int j1=0; j1<n; j1+=BLOCK_SIZE)
        {
            for(int k1=0;k1<n;k1+=BLOCK_SIZE)
            {
                for(int i =0; i<n; i++)
                {
                    for(int j=j1; j<j1+BLOCK_SIZE; j++)
                    {
                        sum = c[i][j];
                        for(int k=k1;k<k1+BLOCK_SIZE;k++)
                        {
                            sum += a[i][k] * b[k][j];
                        }
                        c[i][j] = sum;
                    }
                }
            }
        }
    }
    return 0;
}
```

Figure 2.5: mmm_tuner.py: Parameterized version of matrix multiply with block factor `BLOCK_SIZE`

One limitation of OpenTuner is that it has no direct support for GPU applications. Thus, part of our effort focuses on providing such support. We give an overview of the implementation issues we tackled in chapter 4.

### 2.4 Optimizations

Program optimization is a process that modifies the program such that it operate more efficiently in some aspect such as higher performance or less resources (e.g., memory) used while retaining the same functional behaviour.

GPU programmers must apply a host of optimizations to their GPU kernels (i.e., code that is
#!/usr/bin/env python
import adddeps # fix sys.path

import opentuner
from opentuner import ConfigurationManipulator
from opentuner import IntegerParameter
from opentuner import MeasurementInterface
from opentuner import Result

def manipulator(self):
    """
    Define the search space by creating a ConfigurationManipulator
    """
    manipulator = ConfigurationManipulator()
    manipulator.add_parameter(
        IntegerParameter('blockSize', 1, 10))
    return manipulator

def run(self, desired_result, input, limit):
    """
    Compile and run a given configuration then return performance
    """
    cfg = desired_result.configuration.data

    gcc_cmd = 'g++ mmm_block.cpp '
    gcc_cmd += '-DBLOCK_SIZE=' + cfg['blockSize']
    gcc_cmd += ' -o ./tmp.bin '

    compile_result = self.call_program(gcc_cmd)
    assert compile_result['returncode'] == 0

    run_cmd = './tmp.bin'

    run_result = self.call_program(run_cmd)
    assert run_result['returncode'] == 0
    return Result(time=run_result['time'])

def save_final_config(self, configuration):
    """called at the end of tuning"""
    print "Optimal block size written to mmm_final_config.json:", configuration.data
    self.manipulator().save_to_file(configuration.data, 'mmm_final_config.json')

if __name__ == '__main__':
    argparser = opentuner.default_argparser()
    GccFlagsTuner.main(argparser.parse_args())

Figure 2.6: Matrix multiply tuning example
launched to the GPU for execution) in order to exploit the underlying GPU architecture and ensure high performance. For example, they must ensure memory coalescing to reduce the number of memory transactions made by GPU cores [20], use local memory to cache (under software control) data that is reused [7], reduce thread divergence that results from branches and loops in GPU code [21] and select kernel launch configurations that balance parallelism with resource usage [22]. The impact of each optimization is often difficult to predict until it is applied to the code and is experimentally evaluated. This is particularly the case when multiple optimizations are combined together.

The following subsections describe some of, but not all, of the common optimizations applied to the kernel benchmarks evaluated in this work.

### 2.4.1 Work-Group Size

A Work-group is a collection of threads organized into multiple dimensions. The threads in each dimension in a work-group are called *work items*. The work-group size is the number of work items in each work-group. The size can impact performance depending on whether it is utilizing all the parallelism available on the GPU. If the size is too small, then the hardware is being under utilized which can degrade performance. If the size is too large, then it can potentially create too many work groups that may not be able to execute at the same time on the GPU and thus negatively impact performance. Large work group sizes may also require too much resources from the hardware such as registers or local memory, hindering performance. Figure 2.7 demonstrates how to specify the number of threads per work group (blocks) and how many work groups in the CUDA GPU programming language. The parameters passed into the round parenthesis are the kernel function parameters.

```c
#define M 2048
2 dim3 block3 (X_BLOCK_SIZE, Y_BLOCK_SIZE);
3 dim3 grid3 ((size_t)(ceil((float)M) / ((float)X_BLOCK_SIZE)), 1);
4 covar_kernel<<<grid3, block3>>>(m, n, symmat_gpu, data_gpu);
```

Figure 2.7: Example on how to specify launch configurations in CUDA.

### 2.4.2 Local Memory

The use of local memory as an optimization refers to loading data from global memory into the shared memory cache. Use of local memory can improve performance when there are repeated accesses to the same data within the same work group. It ensures that data the user specified will be retained in the cache since the shared memory cache is software-managed. This can avoid having desired data to be constantly fetched from global memory due to it being evicted from other hardware-managed caches. It
does however, introduce the overhead of data copying from/to global memory and thus is not always beneficial [23, 7] Large local memory sizes may restrict work group sizes and consequently reduce the amount of parallelism we can expose as well. Figure ?? demonstrates how to use local memory to calculate a sequence of factorial numbers. Each work item initially loads in a piece of data from global memory into local memory. Synchronization within the work group is ensured by the local memory fence barrier. Once the computation is calculated, the final result is written back to global memory.

```c
__kernel void factorial_localMemory(__global float *in, __global float *out) {
    __local float localMem[64];
    int tx = get_local_id(0);
    int gx = get_global_id(0);
    localBuffer[tx] = in[gx];
    barrier(CLK_LOCAL_MEM_FENCE);
    float f = localMem[tx];
    for (int i=tx+1; i<64; i++) {
        factorial *= localMem[i];
    }
    out[gx] = f;
}
```

Figure 2.8: Example on how to use local memory in OpenCL to efficiently calculate factorial computation. Adapted from AMD’s OpenCL programming guide [23].

### 2.4.3 Loop Tiling

Loop tiling is an optimization applied to loop structures in a program. It partitions the loop’s iteration into smaller segments such that the data used in the loop is still within the hardware-managed cache [6]. By enabling data reuse within caches, it reduces the high latency fetches to global memory. Figure 2.9 demonstrates a loop before and after loop tiling has been applied to it.

### 2.4.4 Thread Coarsening

Thread coarsening combines the work done in multiple threads into a single thread. For a GPU, it would be multiple work items merged into a single work item that would perform all the work in the same dimension. By combining work items together, it increases thread work granularity and thus reduces resources on thread overhead [5]. However, thread coarsening increases the use of more registers for each work item. Consequently, this can restrict the work group sizes and reduce the amount of parallelism we can expose. Figure 2.10 demonstrates code before and after thread coarsening has been applied to it.
Chapter 2. Background

Listing 2.1: Before

```c
int j1 = blockIdx.x * blockDim.x + threadIdx.x;
int i, j2;
for (j2 = j1; j2 < PB_M; j2++)
{
    symmat[j1*M + j2] = 0.0;
    for (i = 0; i < PB_N; i++)
    {
        symmat[j1*M + j2] += data[i*M + j1] * data[i*M + j2];
    }
    symmat[j2*M + j1] = symmat[j1*M + j2];
}
```

Listing 2.2: After

```c
int j1 = blockIdx.x * blockDim.x + threadIdx.x;
int i, j2, x, y;
for (j2 = j1; j2 < PB_M; j2+=2)
{
    symmat[j1*M + j2] = 0.0;
    symmat[j1*M + j2+1] = 0.0;
    for (i = 0; i < PB_N; i+=2)
    {
        for (x=j2; x < MIN(j2+2, PB_M); x++)
        {
            for (y=i; y < MIN(i+2, PB_N); y++)
            {
                symmat[j1*M + x] += data[y*M + j1] * data[y*M + x];
            }
        }
        symmat[j2*M + j1] = symmat[j1*M + j2];
        symmat[(j2+1)*M + j1] = symmat[j1*M + j2+1];
    }
}
```

Figure 2.9: Loop Tiling Example With 2x2 Block Size
Chapter 2. Background

Listing 2.3: Before

```c
int j1 = blockIdx.x * blockDim.x + threadIdx.x;
int i, j2;
for (j2 = j1; j2 < _PB_M; j2++)
{
    symmat[j1*M + j2] = 0.0;
    for (i = 0; i < _PB_N; i++)
    {
        symmat[j1 * M + j2] += data[i * M + j1] * data[i * M + j2];
    }
    symmat[j2 * M + j1] = symmat[j1 * M + j2];
}
```

Listing 2.4: After

```c
#define CF_X 2
#define CF_Y 2
#define X_BLOCK_SIZE 128
#define Y_BLOCK_SIZE 8

int xIdx = threadIdx.x;
int yIdx = threadIdx.y;
int cf_count_x, cf_count_y, i, j2;

for(cf_count_x = 0; cf_count_x < CF_X; cf_count_x++, xIdx+=(X_BLOCK_SIZE/CF_X))
{
    for(cf_count_y = 0; cf_count_y < CF_Y; cf_count_y++, yIdx+=(Y_BLOCK_SIZE/CF_Y))
    {
        int j1 = blockIdx.x * blockDim.x * blockDim.y + yIdx * blockDim.x + xIdx;
        for (j2 = j1; j2 < _PB_M; j2++)
        {
            symmat[j1*M + j2] = 0.0;
            for (i = 0; i < _PB_N; i++)
            {
                symmat[j1 * M + j2] += data[i * M + j1] * data[i * M + j2];
            }
            symmat[j2 * M + j1] = symmat[j1 * M + j2];
        }
    }
}
```

Figure 2.10: Thread Coarsening Example With 2x2 Coarsening Factor
2.5 Machine Learning

Machine learning is a field in computer science that focuses on automated pattern recognition [24]. It is enabled by provided data and the use of computer algorithms to build a model that can make predictions or decisions. Machine learning can be typically classified as one of the following types of tasks: supervised learning, unsupervised learning, and reinforcement learning [24]. Our work focuses relates to supervised learning.

2.6 Supervised Machine Learning

Supervised machine learning is an approach that enables us to infer an approximation of a function, \( f \), based on a set of provided training data. This data contains sets of values that \( f \) takes as input, called features, and its corresponding output value. The input space is all the possible input combinations that the features can take on. In order to determine whether \( g \) is a good approximation of the underlying function, a set of data that is mutually exclusive from the training data to evaluate the accuracy of the function. This new set of data is called test data. The approximation function, \( g \), is called a supervised machine learning model. Some model representations include Support Vector Machines (SVM), Linear Regression, and Decision Trees [24]. Our work focuses on one particular model called Regression Trees.

2.7 Regression Trees

A regression tree is a supervised machine learning model that can be used to predict the value of a a multivariate function. It utilizes its training data (i.e., inputs and their corresponding function values) to divide the input space into mutually exclusive regions. Each region approximates the value of the function for its corresponding input space by a constant value. These regions are formed by recursively partitioning ranges of values each input can take, in a way that minimizes some measure of mean error [11]. In our use, the multivariate function represents the execution time of a GPU code and the inputs represent the optimization parameters.

A regression tree model can be represented by a tree-like structure. Non-leaf nodes contain constraints that describe how the space should be bisected into smaller mutually exclusive regions. Leaf nodes represent the partitioned regions of the space. An example regression tree for a 2-dimensional optimization space is shown in Figure 2.11. In this example, the inputs \( x \) and \( y \) represent respectively a GPU code’s total number of work-items in the first and second dimensions of the launch configuration, each ranging from 1 to 1024.
Thus, a region in the input space can be defined by a path from the root to its corresponding leaf node, where the nodes on this path represent the constraints on the optimization parameters that define this region. For example, in Figure 2.11, the highlighted path from the root to the second region from the left is defined as configurations for which $x \geq 512$ and $y < 256$. Configurations within this region are all predicted to have a runtime of 200 milliseconds (ms).

![Regression Trees Example](image)

**Figure 2.11:** A regression trees example

### 2.8 Learning Regression Trees

To construct a regression tree model, take the provided training data and partition it into two regions. The region split is determined by finding a boolean question related to the features such that it minimizes some given cost function related to the output. This creates the first (root) node and two child nodes. We repeat this region splitting procedure recursively on each of the child nodes until some stopping criterion is met. We will be using Mean Squared Error (MSE) as our cost function, which is a common choice. For tree $T$ and node $m$, representing a region $R_m$ with $N_m$ entries of training data, then the cost function MSE would be defined as:

$$c_m = \frac{1}{N_m} \sum_{i \in N_m} y_i$$
\[ H(m) = \frac{1}{N_m} \sum_{i \in N_m} (y_i - c_m) \]

where \( c_m \) is the output mean in node \( m \). Then the sum of squared errors for the whole tree, \( T \), would then be:

\[ S = \sum_{m \in \text{leaves}(T)} H(m) \]

The goal is to split the region such that \( S \) is minimized. A general algorithm is described below:

**Input**: Training Data \( D \)

**Output**: Tree \( T \) and Regions \( R \)

**Parameter**: Minimum number of data entries required in a region, \( q \).

Threshold difference for \( S \), \( \delta \)

**Initialize**: Create tree \( T \) and assign \( D \) to root node

forall Leaf nodes \( m \) in \( T \) do

  Calculate \( c_m \) and \( S \);

  Search all possible values each feature can take on such that the binary split will minimize \( S \);

  if Difference in \( S \) after split < \( \delta \) or number of data entries in new region split < \( q \) then

    Terminate;

  else

    Split current node \( m \) such that \( S \) is minimized and create two new child nodes.

end

**Algorithm 1**: Regression Tree Learning [11]

### 2.9 Use of Regression Trees

Typically, regression trees, like other supervised machine learning models, are used for prediction. That is, once the model is built using training data, it is used to predict the performance of a previously unseen configuration. However, we do not use regression trees for prediction. Instead, we use the model’s by-product information as a means to narrow down a given optimization space. Specifically, we focus on the region with the best predicted performance (the highlighted region in the example in Figure 2.11) and the constraints that define this region (the highlighted path in the example) in order to focus the search in the optimization space.

It is important note that the region we are interested in may vary across different regression tree model instances. This may be caused by what training data is provided as input to the model. Based on the training data, the space may be partitioned into different regions and thus the region with the best predicted performance may represent a different set of feature values. Nonetheless, we hypothesize that
with more training data provided, it allows the regression tree learning algorithm to be able to bisect the space into regions where the mean predicted performance has a smaller variance to it. Thus, the model is a better representation of the overall space itself.
Chapter 3

Auto-tuning Strategy

3.1 Overall strategy

Our strategy samples a subset of configurations from the optimization space. Depending on the percentage of samples taken with respect to the entire space size, it may or may not be probable that a sampled configuration is high performing. Thus, we desire to focus the exploration on a small-sized region of the space that is likely to have high performing configurations. We hypothesize that if a regression tree is built with a relatively small number of samples, then the tree’s region with highest mean performance (referred to as \( R \)) is the most worthy of further exploration.

However, the size of the \( R \) region depends on the number of samples used to build the regression tree. If the number of samples is small, \( R \) is too large to explore. If the number of samples is large, then \( R \) is smaller, but this happens at the expense of sampling and evaluating too many configurations and may render the approach infeasible.

Thus, we iteratively build a series of regression trees that each uses a relatively small number of samples, but progressively reduce the size of the \( R \) region. Starting with the initial tree, additional configuration samples are taken from the \( R \) region of the tree. The combined set of the original and new samples is then used to train a new regression tree model. The new \( R \) region is ideally smaller and has higher performing configurations. We then repeat this process, sampling more configurations in the new \( R \) region and training a new regression tree until the size of \( R \) is small enough, or the auto-tuning time budget is exhausted. At this point, the best performing configuration sampled so far is reported back. The following sections elaborates on the strategy.
3.1.1 Sampling Configurations

We start by sampling $n$ configurations from the optimization space to use as training data for a regression tree machine learning model. An important consideration is how many samples to take, i.e., what the value of $n$ should be. On the one hand, too few samples would result in an ineffective regression tree. The region of interest can potentially be too big and sampling from it will effectively be the same as sampling from the original space. On the other hand, too many samples improve the quality of the tree, but they also consume the auto-tuning time budget more rapidly. Thus, it is important to determine an appropriate number of samples to take. Clearly this number depends on the size and characteristics of the optimization space.

Optimization spaces vary across different GPU kernels. Their sizes are dependent on the number of optimizations applied. Further, the characteristics and shape of the space heavily depend on the nature of the kernel and the impact each optimization has on it. This makes it difficult to apriori determine a fixed number of samples to take or how to sample the space. Since we wish to make no assumptions about the space nor use expert knowledge, we opt for a more generic approach to sampling.

We define the number of samples to take to train a regression tree as:

$$n = k \cdot \sigma + b$$

where $n$ is the number of sample configurations to take, $k$ is the number of optimizations that can be applied and $\sigma$ and $b$ are parameters that are user-defined. The value of $n$ is kept constant for each regression tree built by the strategy for an application. We leave exploring varying $n$ as regression trees are built to future work.

The number of sample increases with the number of optimizations to be tuned. This is natural since the size of the optimization space increases as the number of optimizations increases and thus, more samples are needed. The user-defined parameter $\sigma$ is a scaling factor. It provides flexibility to linearly scale the number of samples taken based on the number of optimizations. We use a default value of 1 for this parameter. The user-defined parameter $b$ is used as a base factor to ensure a minimal number of samples used to train the regression tree in the case where there are few optimizations. We use a default value of 10 for this parameter. In chapter 5, we show how the default value of $\sigma$ was chosen through empirical results.

We sample the space using a uniform distribution, with replacement. A sample configuration is formed by independently sampling each optimization, also assuming a uniform distribution, with replace-
ment. This approach to sampling with replacement may potentially result in duplicate configurations used to build the regression tree. We discard these duplicates and not add them to the set of samples used to build the tree. However, given that the sizes of optimization spaces are large for most applications with a sufficient number of optimizations, the number of duplicate samples should be relatively small.

### 3.1.2 Exploration versus Exploitation for the Region of Interest

These user-defined parameters influence the exploration versus exploitation of our heuristic search. The following subsections elaborates on each strategy and their trade-offs.

**Exploration for the Region of Interest**

Exploration in our work is the concept of potentially changing the region of interest our search focuses on. That is, we vary the range of optimization values we search through. The benefit from exploration is that it gives opportunity for our search to stop focusing on a region that has potentially poor performing configurations on average. Instead, it allows us to explore a different region that may potentially have better performing configurations. By decreasing the value of $\sigma$ or $b$, we enable our strategy to further explore potentially better performing regions. Smaller values of $\sigma$ or $b$ implies that we draw less samples from the current region which may be poor performing. Thus, our strategy builds the successive regression tree earlier that results in a new region to be explore.

**Exploitation for the Region of Interest**

Exploitation in our work is the concept of concentrating on the current region of interest. By focusing on the current region of interest, we are sampling more configurations from it. Thus, we increase our chances of obtaining configurations that is more likely to be the expected performance value. The benefit from exploitation is that if the expected performance value is very high, then our search is more likely to be able to obtain a configuration that matches such performance. However, if the expected performance value is not high from the beginning, then it may be detrimental to continuously sample from the same region of interest. By increasing the value of $\sigma$ or $b$, we enable our strategy to further exploit the current region of interest to obtain configurations that meet the expected performance value. Larger values of $\sigma$ or $b$ implies that we draw more samples from the current region which may be high performing. Thus, we increase the probability of finding potentially higher performing configurations than what we have found so far.
Balance Between Exploration versus Exploitation

It is clear that both exploration and exploitation have its benefits and trade-offs. Using figure 3.1 as an example, we can see that depending on which region of interest we are currently in, both exploration and exploitation have its benefits. For example, if we are currently in region B, exploitation may prove to be more beneficial since region C (with the higher performance) is relatively close. If we were to explore too much, we may end up in region D with significantly worse mean performance. On the contrary, if we started out at region D, exploration creates opportunities for us to escape the minima and potentially be able to reach region A or C instead. It is important to find a good balance between these two approaches in our strategy such that we can obtain high performing configurations in an efficient amount of tuning budget. In chapter 5, we show by experimentation how we can optimize between exploration and exploitation for the value of $\sigma$. We leave the optimization of $b$ as future work.

![Figure 3.1: Example of exploration versus exploitation.](image)

3.1.3 Pruning the Optimization Space

We are interested in the leaf node within the regression tree model that has the highest sample mean performance. We refer to it as the node of interest, $\mathcal{N}$. We find the path associated with this node, starting from the root node. The constraints on the optimization values along this path allow us to narrow the range of values the optimization parameters can take and define a corresponding region of interest in the optimization space. That is, the constraints allow us to prune the space, initially starting with the full space, down to a pruned region. This is the region of interest $\mathcal{R}$.

The example in Figure 2.11 can be used to illustrate this process. The node $\mathcal{N}$ is the one with 200 ms
since its sampled average runtime is the lowest amongst all other leaf nodes. The path that connects
the root of the tree to \( N \) allows us to utilize the intermediate nodes’ information to narrow down the
the optimization space. The first such node has a constraint that restricts optimization parameter of \( x \)
to have a range of \([512, 1024]\). The next node on the path restricts optimization \( y \) to be have parameter
values in the range of \([1, 256]\). The final pruned region is shown in the same figure on the right side.

It is possible that not all of the optimizations have their range of values narrowed down. Some of the
optimizations may retain their original value ranges because the regression tree algorithm decided that
these optimizations did not have the biggest influence on performance. Nonetheless, it is still possible
to obtain a pruned region in this case.

3.1.4 Exploring the Pruned Region

With the potentially narrowed down range of values each optimization parameter now takes, new samples
taken in the pruned region \textit{should} have performance that is similar to the mean value associated with
the leaf node of interest. Thus, by sampling from the pruned region, it is expected to draw samples with
performance similar to the mean performance value described in the node of interest. However, it is also
possible to sample configurations that deviate from the expected performance; potentially configurations
that are higher or lower performing than the expected and the current best sample explored so far.
Nonetheless, exploring this region increases the opportunity to find high performing configurations.

3.1.5 Iteratively Pruning the Optimization Space

We further narrow down the size of the region of interest by iteratively building regression trees. Ad-
ditional samples are taken from the region of interest. Combined with the existing samples, they are
used to re-train a new regression tree. Given the density of samples in the region of interest, it is more
likely the latest region of interest is a subspace of the previous, although not necessarily. This process
is repeated until the size of the region is small enough.

Ideally, the regions of interest in the successive regression trees are smaller in size and progressively
have better mean performance values (or at least still have high performing configurations). This is
expected since the successive trees are trained with more and more samples from the space and thus
have more information, particularly on the regions of interest. Thus, successive trees can potentially
partition the region of interest into even smaller sub regions and thus narrower ranges of optimization
values to explore.

It is important to note that there is a possibility where the successive region of interest is not narrowed
down. Although new samples added to the training of the successive regression tree is suppose to assist in the characterization of the optimization space, there is no guarantee that the quality of the new samples will provide any beneficial effects to the one region of the space we are interested in. For example, if new samples taken have run times that are similar to the expected run time of the region we have taken them from, then it may be difficult for the regression tree algorithm to bisect the region. Since the new samples are so closely clustered with the current region of interest already, the criteria to bisect the region based on runtime differences may not be satisfied.

3.1.6 Algorithm Summary

Algorithm 2 gives the pseudocode of our auto-tuning strategy. It takes as input the auto-tuning time budget $\tau$ and outputs the best found configuration $C$ and its runtime. It also has a number of parameters: the number of optimizations $k$, the original optimization space $R_o$ and the scaling and base factors, $\sigma$ and $b$. It also has an internal parameter $r$ that determines the tolerance of the algorithm to having duplicate samples. The elapsed time represents wall clock time.

The region of interest is initially the entire optimization space $R_o$ and the set of samples $S$ is empty. The number of configurations to sample $n$ is determined. As long as the auto-tuning budget $\tau$ is not exhausted, a sample is taken from $R$. If the sampled configuration has not been sampled before, it is added to the set of samples $S$ and is executed to obtain its execution time. We select our best configuration, $C$, as that with the lowest run time in $S$. This is repeated until $n$ new samples are obtained.

If the sampled configuration has been previously sampled, then it is discarded. However, if the number of repeated samples exceed a certain threshold, $r$, we expand the region $R$ to the original space, $R_o$. This is done because $R$ may become significantly small and the number of sample configurations required to train the next regression tree may be larger than the region’s size. Thus we may not be able to sample enough unique configurations to train the next successive regression tree. Since the time budget is not yet exhausted, we sample again from the entire space to possibly better the quality of the regression trees built. This can also be seen as an opportunity for exploration. For example, if the current region of interest we are sampling in is actually a region that is far from optimal, by resetting the sampling space back to the original, we give ourselves an opportunity to potentially escape this local minima.

Once $n$ new configurations have been sampled, and if the time budget has not been exhausted, a new regression tree is trained with $S$. $\mathcal{N}$ and its associated path $P$ are identified and a new region of
interest $\mathcal{R}$ is determined by narrowing down $R_o$ with the constraints of the intermediate nodes along $P$. The algorithm iterates by now sampling from the new $\mathcal{R}$ region until the auto-tuning time budget is exhausted.

**Input**: Auto-tuning time budget, $\tau$

**Output**: Best found configuration, $C$ and its runtime

**Parameter**: Scaling factor $\sigma$

- Number of optimization parameters, $k$
- Base number of samples, $b$
- Original Optimization Space, $R_o$
- Repeated samples threshold, $r$

\[
\mathcal{R} = R_o \\
S = \{\}
\]

\[
n = k \cdot \sigma + b \\
unique\_count = 0 \\
repeat\_count = 0
\]

\[
while\ elapsed\ time < \tau \ do
\]

\[
\text{keepSampling} = true \\
unique\_count = 0 \\
repeat\_count = 0
\]

\[
while\ keepSampling\ and\ elapsed\ time < \tau \ do
\]

\[
s = \text{Sample a configuration from } \mathcal{R}
\]

\[
\text{if } s \notin S \text{ then}
\]

\[
unique\_count++ \\
\text{Execute } s \text{ and record its runtime}
\]

\[
S = S + s \\
C = \text{best performing configuration in } S
\]

\[
\text{else}
\]

\[
repeat\_count++
\]

\[
\text{if } unique\_count == n \text{ then}
\]

\[
\text{keepSampling} = false
\]

\[
\text{else if } repeat\_count > r \text{ then}
\]

\[
\mathcal{R} = R_o \\
repeat\_count = 0
\]

\[
\text{end}
\]

\[
\text{if } elapsed\ time < \tau \text{ then}
\]

\[
\text{Train regression tree } T \text{ with } S
\]

\[
\mathcal{N} = T's \text{ leaf node with best sampled mean performance}
\]

\[
P = \text{Path from } T's \text{ root node to } \mathcal{N}
\]

\[
\mathcal{R} = \text{Narrowed down } R_o \text{ using } P's \text{ constraints}
\]

\[
\text{end}
\]

**Algorithm 2**: Auto-tuning strategy using regression trees
Chapter 4

Implementation

This chapter describes the implementation details of the following: integrating gpu applications into the framework (OpenTuner), adding support to profile GPU kernels, modifications done on existing benchmarks such that it can be auto-tuned by the framework, and integrating the proposed auto-tuning strategy using regression trees into the existing framework.

4.1 Integrating GPU applications into OpenTuner

We implement our auto-tuning strategy within an already existing auto-tuning framework, OpenTuner [12]. This not only saves us the effort to develop a complete framework, but also allows us to compare our strategy to OpenTuner’s ones. Their heuristics have been shown to find configurations with performance similar to that of hand-tuned code [12]. Thus, we compare our approach to the state-of-the-art auto-tuning that reflects what can possibly be achieved by manual tuning.

OpenTuner is built in a modular way to allow users to specify the optimizations, how to compile and execute the application. This facilitates the use the framework to tune the performance new applications. Nonetheless, the framework is mainly intended to tune CPU applications and, thus, it had to be extended to support GPU ones. There are two main implementation issues that arose in this regards. The first of the need to manually instrument GPU kernels to obtain kernel execution time. The second is having to deal with inter-parameter constraints. We elaborate on these issues and how they were addressed in the remainder of this chapter.

The following is an example based off of the lavaMD benchmark which will demonstrate how to integrate a new GPU application to be tuned within the framework. Note that the code is similar to the example from Figure 2.6 in Chapter 2. However, the optimizations we wish to tune in this example
def manipulator(self):
    manipulator = ConfigurationManipulator()
    manipulator.add_parameter(IntegerParameter('block', 1, 1024))
    manipulator.add_parameter(EnumParameter('lmem', [0, 1]))
    manipulator.add_parameter(IntegerParameter('outertile', 0, 16))
    manipulator.add_parameter(IntegerParameter('innertile', 0, 16))
    return manipulator

Figure 4.1: Defining the GPU optimization parameters to tune

is GPU-centric and application specific: launch configuration, use of local memory, outer and inner loop tiling block factors. First we specify the optimizations we wish to tune as shown in Figure 4.1. The Configuration Manipulator is responsible for managing all of the optimization parameters and their range of values. Here, the launch configurations have been specified, block, which take on integer values. Followed by the use of local memory or not, lmem, and the loop tiling block factors, (outerTile and innerTile), which takes on boolean and integer values respectively.

Next, we specify how the application is compiled and ran shown in Figure 4.2: updating the optimization values to be applied in the application, and specifying the commands to execute. We implemented a function called updateTemplate which replaces the parameterized and tokenized version of the benchmark kernels with optimization values such that they can be compiled and executed. This is done by parsing through the specified header file, line by line, and replacing the token, $, with the optimization value. Note that self.call_program was a built in functionality by the original framework, but it lacked the support to measure GPU kernel run times explicitly. Therefore, we have implemented our own variant of the function to enable GPU profiling, self.call_gpu_program. One key point to note here is that we return not only the total runtime, but the GPU kernel time, gpu_time, and compile time as well. Another point to note is that by having separate objects for compilation and execution, we are able to measure performance with more granularity. The following sections will describe the work done to support GPU profiling using the GPU kernel time returned.

4.2 Profiling GPU kernels

OpenTuner measures an entire application's runtime but has no means to explicitly measure a GPU kernel's runtime. Since we are tuning the performance of the kernel, we extend the framework to profile kernel functions for both OpenCL and CUDA programming languages.

We manually add instrumentation instructions to each kernel at the source code level. This is necessary because of constraints on Nvidia hardware and its tool kits, which make it not possible to use their proprietary OpenCL profiling tools within the framework. Thus, the use of instrumentation
CHAPTER 4. IMPLEMENTATION

1 parameters = ["block", "lmem", "outerTile", "innerTile"]
2 files = ["macros.h", "macros.h", "macros.h", "macros.h"]
3 keywords = ["#define NUMBER_THREADS",
4 "#define USE_LOCAL_MEM",
5 "#define OUTER_TILE",
6 "#define INNER_TILE"]
7 templates = ["#define NUMBER_THREADS \n",
8 "#define USE_LOCAL_MEM \n",
9 "#define OUTER_TILE \n",
10 "#define INNER_TILE \n"]

11
def updateTemplate(self, sample):
12 for p in range(len(parameters)):
13 file = open(files[p], 'r')
14 lines = file.readlines()
15 file.close()
16
17 keyword = keywords[p]
18 template = templates[p]
19 for line in lines:
20 if line.find(keyword) != -1:
21 temp_key = str(parameters[p])
22 lines[lines.index(line)] = template.replace('\$', str(sample[temp_key]))
23
24 file = open(files[p], 'w')
25 for line in lines:
26 file.write(line)
27 file.close()

28
def run(self, desired_result, input, limit):
29 cfg = desired_result.configuration.data
30
31 if self.isInvalid(cfg):
32 return Result(state='ERROR', time=float('inf'))
33
34 self.updateTemplate(cfg)
35
36 make_cmd = 'make clean; make -j4'
37 make_result = self.call_program(make_cmd)
38 if make_result['returncode'] != 0:
39 return Result(state='ERROR', time=float('inf'))
40
41 run_cmd = './lavaMD -boxes1d 20'
42 run_result = self.call_gpu_program(run_cmd, cfg)
43 if run_result['returncode'] != 0:
44 return Result(state='ERROR', time=float('inf'))
45
46 return Result(time=run_result['gpu_time'], compileTime=make_result['time'],
47 totalRuntime=run_result['time'])

Figure 4.2: Specifying how the GPU application is compiled and ran
instructions allows a common approach for both CUDA and OpenCL kernels. We use the built-in `event` object that each respective language provides [17, 16] to measure kernel run time. Figure 4.3 and 4.4 is an example of how CUDA and OpenCL kernels are profiled, respectively. The kernel prints a formatted output to the standard output stream, specifying the kernel function being profiled. Figure. We modify the OpenTuner framework to parse this formatted output and record kernel run times.

```c
1 cudaEvent_t start_timer, stop_timer;
2 cudaEventCreate(& start_timer);
3 cudaEventCreate(& stop_timer);
4 float kernelTime = 0;
5 
6 cudaEventRecord(start_timer);
7 covar_kernel<<<grid3, block3>>>(m, n, symmat_gpu, data_gpu);
8 cudaThreadSynchronize();
9 cudaEventRecord(stop_timer);
10 cudaEventSynchronize(stop_timer);
11 
12 cudaError err;
13 err = cudaGetLastError();
14 if (cudaSuccess != err) {
15 fprintf("kernel3 failed: %s\n", cudaGetErrorString(err));
16 exit(-1);
17 }
18 
19 cudaEventElapsedTime(&kernelTime, start_timer, stop_timer);
20 printf("KERNEL PROFILE TIME:%f\n", kernelTime);
```

Figure 4.3: Profiling a CUDA GPU kernel

```c
1 cl_event event1;
2 cl_ulong time_start1, time_end1;
3 float total_time1;
4 
5 err = clEnqueueNDRangeKernel(cmd_queue, kernel_s, 1, NULL, global_work, &local_work_size , 0, 0, &event1);
6 
7 if (err != CL_SUCCESS)
8 {
9 fprintf("ERROR: clEnqueueNDRangeKernel() =>%d failed\n", err); return -1;
10 }
11 
12 clWaitForEvents(1, &event1);
13 clGetEventProfilingInfo(event1, CL_PROFILING_COMMAND_START, sizeof(time_start1), &
14 time_start1, NULL);
15 clGetEventProfilingInfo(event1, CL_PROFILING_COMMAND_END, sizeof(time_end1), &time_end1, NULL);
16 total_time1 = ((time_end1 - time_start1) * 1.0e-6f);
```

Figure 4.4: Profiling an OpenCL GPU kernel
4.3 Inter-parameter Constraints

OpenTuner generates a random configuration by taking random values in the ranges of optimization parameters, one optimization parameter at a time. This is problematic for GPU kernels for which the value of one optimization parameter can influence what values other parameters may take. One example of such inter-parameter constraints is the size and shape of the launch configuration. Hardware limitations impose a constraint on the product of the number of work-items in each dimension. This product cannot exceed a certain limit. Configurations that do not observe this limit are not valid and cannot be launched for execution.

In general, it is possible to assign an invalid kernel configuration infinite execution time and allow the framework to ignore such invalid configurations in the auto-tuning process. However, this unnecessarily expends the auto-tuning time budget since invalid configurations must still be compiled and launched to determine that they are invalid. Thus, we add support to the framework to express inter-parameter constraints.

We defined a new parameter class, called `ConstrainedProductIntegerListParameter`. This class takes as input an existing a set of `IntegerParameters` on which we wish to express a constraint on their product.

An `IntegerParameter` is a class defined by the OpenTuner framework to express an optimization parameter that takes on integer values. It contains class members such as lower and upper bounds to support random integer values generated. Valid combinations of parameter values are guaranteed by first generating a cross-product list of all values associated with the parameters. These values are then filtered using the product constraint.

When a random configuration is needed, optimizations that have no constraints associated with them have a value in their ranges randomly chosen, as in the original framework. However, ones that do have constraints have their values randomly selected from the list of filtered combinations to guarantee that the configuration conforms to the constraints specified.

```python
def manipulator(self):
    manipulator = ConfigurationManipulator()
    manipulator.add_parameter(IntegerParameter('x', 1, 256))
    manipulator.add_parameter(IntegerParameter('y', 1, 256))
    manipulator.add_parameter(ConstrainedProductIntegerListParameter(['x', 'y'], [x, y], 25, 1024))
    return manipulator
```

Figure 4.5: Specifying optimizations to be constrained by upper and lower bounds through its product value
Figure 4.5 demonstrates how to use this parameter class to specify other optimizations to be constrained. The new Constrained Product Integer List Parameter takes in Integer parameters, a lower and upper bound specifying the range the product value can take on.

### 4.4 Modifications to Benchmark Kernels

To demonstrate our auto-tuning strategy’s effectiveness in different application domains, we have selected a variety of benchmarks from both the Rodinia [25] and PolyBench-ACC benchmark suites [26]. As mentioned in the previous section, each benchmark kernel has been modified with additional instrumentation instructions at the source code level for profiling purposes. Furthermore, each benchmarks’ optimizations has been parameterized using make file compilation flags and the C preprocessor macro language. Figure 4.6 shows an example of how a benchmark is parameterized using C preprocessor macros. First we use the `#define` macro to specify what optimization and its value to be applied. Using the `#if` statements, it enables us to select which version of the source code we wish to compile in conjunction with how the optimization is applied. In this particular example, a `TILE_SIZE` of 8 has been defined and thus we are selecting to execute the nested for loops with a `TILE_SIZE` by `TILE_SIZE` block factor for our loop tiling optimization.

### 4.5 Auto-tuning Strategy Implementation

Our auto-tuning strategy is implemented using the Python programming language, just like the existing strategies in OpenTuner. We use the Python machine learning package scikit-learn [24] and utilize its implementation of regression trees. We leave the parameters of the package’s regression tree model to their default values.

The framework passes control to the optimization technique whenever a requested configuration has finished executing and evaluated for its objective. Using this functionality, we keep track of how many unique samples we have taken in total so far and how many of which were taken since the last regression tree built. First, we sequentially request for samples where each optimization has its value uniformly randomized. We poll until we have the required amount of samples and proceed to train a regression tree using all of the samples we have evaluated so far. If the sample requested has already been requested from before, it will be ignored and not counted towards the current number of samples required to train the next regression tree. Once we have enough samples required to train a regression tree, we extract the necessary information from the model to narrow down the space as described in the auto-tuning
#define TILE_SIZE 8
__global__ void covar_kernel(int m, int n, DATA_TYPE *symmat, DATA_TYPE *data)
{
int j1 = blockIdx.x * blockDim.x + threadIdx.x;
int i, j2;
int x,y;
int b;

if (j1 < _PB_M)
{
    #if TILE_SIZE > 0
    for (j2 = j1; j2 < _PB_M; j2+=TILE_SIZE)
    {
        for(b = j2; b < MIN(_PB_M, j2+TILE_SIZE); b++){
            symmat[j1*M + b] = 0.0;
        }
    }
    for(i = 0; i < _PB_N; i+=TILE_SIZE)
    {
        for(x=j2; x < MIN(j2+TILE_SIZE ,_PB_M); x++){
            for(y=i; y < MIN(i+TILE_SIZE ,_PB_N); y++){
                symmat[j1 * M + x] += data[y * M + j1] * data[y * M + x];
            }
        }
    }
    #endif
}

for(b = j2; b < MIN(_PB_M, j2+TILE_SIZE); b++){
    symmat[b * M + j1] = symmat[j1 * M + b];
}
}

Figure 4.6: Parameterized GPU kernel example with 8x8 block size loop tiling optimization
strategy chapter: setting the range of values each optimization can take on. Any random configuration requested from this point onwards will have its optimization value be randomized from the narrowed down range described previously.

OpenTuner generates a random configuration by taking random values in its specified range for each optimization one a time. This may potentially cause the random configuration generated to be considered as invalid. For example, the product of the per dimension work group size value must be less than some constant value that is limited by the hardware platform specifications. In order to ensure that we generate valid configurations and thus having the framework spend less time on this aspect, we have implemented support to specify the product range two optimization parameters can take on. Figure 4.5 shows two parameters to be tuned where x and y represent the work group size in the x-y dimension respectively. The following line of code in the figure ensures that the product of x and y can only range from 25 to 1024.

4.5.1 Resetting the Search Region

Over time as we generate enough sample configurations, the region of interest we are to explore may become significantly small. In this situation, the number of sample configurations required to train the next regression tree may be larger the the region’s size, or very close to it. Thus we may not be able to sample enough unique configurations to fulfil the requirements to train the next successive regression tree. Our solution is to reset the region of interest to sample from back to the original optimization space so that there is a higher chance to sample configurations that have not been already sampled in the current tuning session. Furthermore, sampling from the original optimization space potentially enables the regression tree algorithm to bisect a new region of the space that has high sampled performance for our strategy to explore.
Chapter 5

Evaluation

This chapter describes the evaluation methodology and presents the performance results achieved.

5.1 Evaluation Platform

Our results are collected on a system with an Nvidia GTX 1060 GPU. The host has a 2.7 GHz Intel i5-6400 processor with 8 GB of memory, running Ubuntu 14.04 and using Nvidia driver version 367.48.

5.2 Benchmarks

Table 5.1 lists the benchmarks we use in our evaluation. For each benchmark, the table lists the optimizations applied to it, the range of values of each optimization parameter and the number of configurations in the optimization space.

The benchmarks come from the Rodinia [25] and the PolyBench-ACC [26] benchmark suites. The optimizations are implemented in the code of the benchmarks using C preprocessor macros and compile flags. Specifically, \#define macros are used to specify flags that indicate which optimizations are to be applied and their parameter values. Then, \#if macros are used to select the version of the source code associated with the \#define value to be compiled. The Hotspot, BFS, and Streamcluster benchmarks are implemented in this way by the authors of Starchart [27]. We parametrize the remaining benchmarks similarly.

There are other alternative approaches to parameterizing the benchmarks. For example, the optimization values could have been stored in a text file instead of being a part of the source code files directly. Thus, it allows us to avoid recompiling the a different version of the program every time a
different optimization value is applied to it. However, the drawbacks would include the additional run
time incurred during the execution phase. Note that we are strictly measuring the kernel run time and
thus this alternative approach would not affect our results of interest.

We chose the C preprocessor approach because it is a simplistic model and we are able to select
specific versions of code to be compiled in a much more efficient way versus having to constantly evaluate
some condition to determine which version of code to be executed during run time. Furthermore, by
having constant values presented during compilation phase provides more potential opportunities for the
compiler to apply its own internal optimizations techniques which may result in a speedup gain.

### Table 5.1: Benchmark descriptions and optimizations

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Description</th>
<th>Optimizations</th>
<th>Parameter Ranges</th>
<th>Total # of Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotspot</td>
<td>Temperature simulation for processor architecture</td>
<td>Thread block size X, Thread block size Y, Transpose temperature, Transpose power, Transpose buffer arrays</td>
<td>5-256</td>
<td>57424</td>
</tr>
<tr>
<td>K-means</td>
<td>K-means clustering algorithm</td>
<td>Thread block size, Thread cores factor, Outer loop tiling factor, Inner loop tiling factor</td>
<td>16-1024, 0-1, 0-16, 0-16</td>
<td>2293564</td>
</tr>
<tr>
<td>BFS</td>
<td>Graph traversal - Breadth first search</td>
<td>Thread block size, amount of points per thread, Use parallel arrays, Use share memory, Max regs per thread, Use of cache</td>
<td>16-1024, 0-1, 0-16, 0-1, 1-16, 0/1</td>
<td>10158080</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>Continuous streaming to be clustered</td>
<td>Thread block size, amount of points per thread, Use parallel arrays, Use share memory, Max regs per thread, Use of cache</td>
<td>16-1024, 0-1, 0-16, 0-1, 1-16, 0/1</td>
<td>1310720</td>
</tr>
<tr>
<td>lavaMD</td>
<td>Particle force and potential in 3D space.</td>
<td>Thread block size, Use local memory, Outer loop tiling factor, Inner loop tiling factor</td>
<td>0-1, 0-8, 0-8, 0-8</td>
<td>573376</td>
</tr>
<tr>
<td>GEMM</td>
<td>Generic matrix multiplication</td>
<td>Thread block size X, Tiling length, Use share memory</td>
<td>2-1024, 1-128, 0-3</td>
<td>2745216</td>
</tr>
<tr>
<td>Covariance</td>
<td>Calculates the covariance matrix</td>
<td>Thread block size X, Thread cores factor X, Outer loop tiling factor, Inner loop tiling factor</td>
<td>32-1024, 1-8, 0-16, 0-16</td>
<td>33056976</td>
</tr>
<tr>
<td>Correlation</td>
<td>Calculates the correlation matrix</td>
<td>Thread block size X, Thread cores factor X, Outer loop tiling factor, Inner loop tiling factor</td>
<td>32-1024, 1-8, 0-16, 0-16</td>
<td>33056976</td>
</tr>
</tbody>
</table>

Finally, we add instrumentation instructions at the source code level for profiling purposes; these are
needed for OpenTuner to measure the performance of the kernels, as was described earlier.

### 5.3 Performance Metrics

We evaluate the performance of our auto-tuning strategy using a few metrics. Given an auto-tuning
time budget, we measure the best kernel runtime found within this time budget. This reflects how well
the auto-tuning strategy optimizes a kernel. We compute the speedup of this optimized kernel as the
ratio of the execution time of the unoptimized kernel to the execution time of the optimized one. We
also measure the number of configurations explored by the strategy. This reflects the extent to which
the optimization space is explored to arrive at the optimized configuration. Finally, we measure the
cumulative runtime of the kernels of these configurations to assess the quality of the configurations
explored.

5.4 Hypothesis Validation

In Chapter 3, we hypothesized that the region with the highest mean performance in a regression tree,
$R$, is worthy of further exploration. In this section, we validate that hypothesis.

We examine the size of the region $R$ in the regression tree and the distribution of the performance
of the configurations in this region, as successive regression trees are built by our strategy. To evaluate
the entire configuration space of a benchmark as shown in the table above requires an infeasible amount
of time. For example, the GEMM benchmark has its kernel runtime vary from 5 seconds up to 12
seconds and the cross-product space size is 41472 configurations. Taking the average of the minimum
and maximum runtime results would result in 8 seconds per configuration. This implies that it would
roughly take 331776 seconds, or 92 hours to exhaustively search the full space. The execution time
of these configuration and the optimal configuration are obtained by exhaustively executing all the
configurations in a subspace, only for the purposes of this experiment.

Table 5.2 shows the subspace of optimization values the each benchmark takes on for this experiment.
Note that the total number of configurations is the not cross-product size of all the optimizations. Some
benchmarks have certain optimization constraints, such as Hotspot requiring the product value of multi-
dimensional thread block size to be at least a value of 25. Figure 5.1 shows the size (measured by the
number of configurations) of the $R$ region for each of the first 3 trees built by the strategy. In this
experiment, the number of samples taken per tree $n$ is 1.5% of the space size, except for Hotspot where
20% was used. This is due to its total number of configurations being small, with only 120 configurations
in total. The figure shows that the size of this region sharply decreases as additional trees are built,
which validates our heuristic.

Figure 5.2 to 5.9 show, for each benchmark, the distribution of the execution time of all the con-
figurations in the $R$ in each of the tree. Thus, the horizontal axis shows ranges of the speedup of a
configuration, computed with respect to the optimal solution. The execution time of these configuration
and the optimal configuration are obtained by exhaustively executing all the configurations in a sub-
space, only for the purposes of this experiment. Thus, the higher the speedup, the better and the best
speedup is 1. The vertical axis shows the number of configurations in each speedup range.
Chapter 5. Evaluation

Table 5.2: Benchmark subspace optimizations.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Optimizations</th>
<th>Parameter Ranges</th>
<th>Total # of Configurations</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotspot</td>
<td>Thread block size X</td>
<td>1-1024</td>
<td>120</td>
<td>Thread block size in steps of power of 2</td>
</tr>
<tr>
<td></td>
<td>Thread block size Y</td>
<td>1-1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transpose temperature</td>
<td>0/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transpose power</td>
<td>0/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transpose buffer arrays</td>
<td>0/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-means</td>
<td>Thread block size X</td>
<td>32-1024</td>
<td>11560</td>
<td>Thread block size in steps of power of 2</td>
</tr>
<tr>
<td></td>
<td>Thread block size Y</td>
<td>1-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Outer loop tiling factor</td>
<td>0-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inner loop tiling factor</td>
<td>0-16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFS</td>
<td>Thread block size X</td>
<td>32-1024</td>
<td>2880</td>
<td>Thread block size in steps of power of 2</td>
</tr>
<tr>
<td></td>
<td># of nodes per thread</td>
<td>1-16</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Consecutive nodes</td>
<td>0/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use parallel arrays</td>
<td>0/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Max regs per thread</td>
<td>16-24</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use of cache</td>
<td>0/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Streamcluster</td>
<td>Thread block size X</td>
<td>32-1024</td>
<td>6000</td>
<td>Thread block size in steps of power of 2</td>
</tr>
<tr>
<td></td>
<td># of points per thread</td>
<td>1-15</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use parallel arrays</td>
<td>0/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use share memory</td>
<td>0/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Max regs per thread</td>
<td>16-32</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use of cache</td>
<td>0/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lavaMD</td>
<td>Thread block size X</td>
<td>32-1024</td>
<td>5184</td>
<td>Thread block size in steps of power of 2</td>
</tr>
<tr>
<td></td>
<td>Use local memory</td>
<td>0/1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Outer loop tiling factors</td>
<td>0-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inner loop tiling factors</td>
<td>0-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GEMM</td>
<td>Thread block size X</td>
<td>2-1024</td>
<td>1458</td>
<td>Thread block size in steps of power of 2 from 2-16, then in steps of 32 onwards. Tile range in steps of power of 2 onwards.</td>
</tr>
<tr>
<td></td>
<td>Thread block size Y</td>
<td>2-1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tiling length</td>
<td>1-128</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use share memory</td>
<td>0-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Covariance</td>
<td>Thread block size X</td>
<td>1-1024</td>
<td>8775</td>
<td>Thread block size in steps of power of 2</td>
</tr>
<tr>
<td></td>
<td>Thread block size Y</td>
<td>1-1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thread coarsen factor X</td>
<td>1-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thread coarsen factor Y</td>
<td>1-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Outer loop tiling factor</td>
<td>0-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inner loop tiling factor</td>
<td>0-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Correlation</td>
<td>Thread block size X</td>
<td>1-1024</td>
<td>8775</td>
<td>Thread block size in steps of power of 2</td>
</tr>
<tr>
<td></td>
<td>Thread block size Y</td>
<td>1-1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thread coarsen factor X</td>
<td>1-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thread coarsen factor Y</td>
<td>1-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Outer loop tiling factor</td>
<td>0-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inner loop tiling factor</td>
<td>0-4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following observations generally apply for all the benchmarks. The figures show that while the performance of the configurations vary, it overall improves as regression trees are built. The mean configuration speedup (vertical dotted line in each figure) becomes higher. Further, there are more higher performing configurations as indicated by the shift to the right of the speedup histograms.

It is important to note that although some successive regions do not improve the mean configuration speedup drastically, we do not observe any regression behaviours exhibited. For example, Hotspot’s full space compared to the first regression tree’s region shows marginal improvement. Although it did not show any benefits gained, we did not get a region that is any worse.

These results further validate our heuristic and our hypothesis that successively building regression trees focuses the search on better regions of the space.
Chapter 5. Evaluation

Figure 5.1: Number of configurations in $\mathcal{R}$ regions

(a) Hotspot region sizes in $\mathcal{R}$

(b) Kmeans region sizes in $\mathcal{R}$

(c) BFS region sizes in $\mathcal{R}$

(d) Streamcluster region sizes in $\mathcal{R}$

(e) lavaMD region sizes in $\mathcal{R}$

(f) GEMM region sizes in $\mathcal{R}$

(g) Covariance region sizes in $\mathcal{R}$

(h) Correlation region sizes in $\mathcal{R}$
Chapter 5. Evaluation

Figure 5.2: Hotspot Execution time distribution in $\mathcal{R}$ regions

Figure 5.3: Kmeans Execution time distribution in $\mathcal{R}$ regions
Chapter 5. Evaluation

Figure 5.4: BFS Execution time distribution in $\mathcal{R}$ regions

Figure 5.5: Streamcluster Execution time distribution in $\mathcal{R}$ regions
Chapter 5. Evaluation

43

(a) Full space
(b) First tree
(c) Second tree
(d) Third tree

Figure 5.6: lavaMD Execution time distribution in $\mathcal{R}$ regions

(a) Full space
(b) First tree
(c) Second tree
(d) Third tree

Figure 5.7: GEMM Execution time distribution in $\mathcal{R}$ regions
Figure 5.8: Covariance Execution time distribution in $\mathcal{R}$ regions

Figure 5.9: Correlation Execution time distribution in $\mathcal{R}$ regions
Chapter 5. Evaluation

5.5 Determining a Good value for $\sigma$

First we show how we selected an appropriate $\sigma$ value as part of the algorithm described for sampling in the strategy chapter. For each benchmark, we swept through integer values from 1 to 10 for $\sigma$ and collected the metrics described above. Each sweep is iterated over 20 times to get averaged out results. A maximum auto-tuning time budget of 400 seconds is used. This value We compare the fitted curves to see which model yields us better kernel run times using less amount of elapsed time by the framework (the curve that lower bounds the rest). We repeat this evaluation for each benchmark and record which $\sigma$ model performs best. Figure 5.10 shows the ten performance curves for different values of $\sigma$, for each benchmark. For example, S1, represents the curve with $\sigma = 1$. We can see that for different benchmarks, the optimal value of $\sigma$ varies: an intrinsic characteristic of the benchmark and its optimization space. Thus, there is no single optimal value for $\sigma$ that we can select for all benchmarks. Therefore, our solution is to select the value that occurs most frequently as it is able to optimize towards the majority of the benchmarks. The chosen $\sigma$ value for each benchmark has its curve bolded in the figure. Taking the mode $\sigma$ value across all 8 benchmarks, we have chosen the default value of $\sigma$ to be a value of 1.

It is important to note that all the curves eventually converge to the same best kernel run time. This implies that, with sufficient amount of auto-tuning time budget, regardless of what value $\sigma$ is, our strategy is able to converge on good performing configurations. That is, non optimal values of $\sigma$ used in our strategy is still able to achieve high performance, but under the circumstances of requiring more auto-tuning time budget. In the situation where a programmer wants to use our proposed strategy to auto-tune, but have no priori knowledge of what value of $\sigma$ to use, our results show that given sufficient auto-tuning time budget, it is still possible to discover high performing configurations.

5.6 Comparing Competitive Strategies

In order to assess the competitiveness of our strategy, we compare it to strategies already integrated within OpenTuner. Specifically, We compare to OpenTuner’s ensemble strategy that combines three different optimization strategies, differential evolution, greedy mutation and Nelder-Mead, into a single strategy. In effect this combined strategy uses a multi-armed bandit with sliding window, area under the curve credit assignment (AUC Bandit) algorithm to determine which of the three strategies is best for the current application [12]. Thus, we believe that we compare our strategy to the most effective strategy used by OpenTuner.
5.7 Methodology

We run our auto-tuning strategy with $\sigma = 1$, $b = 10$, and $r = 10$. A maximum auto-tuning time budget of 400 seconds is used. We found that this budget is long enough to allow the evaluated strategies to converge on a solution. Yet, it is small enough to be a relatively small percentage of the total time required to exhaustively search the entire space. Nonetheless, within OpenTuner, a time stamp is recorded every time a better performing kernel runtime is found in the current tuning session. These timestamps are used to determine the best performing configuration discovered at regular auto-tuning time budget intervals. This, in effect, allows us to determine the best configuration discovered for incrementally increasing auto-tuning time budgets. We run each experiment 100 times to address the noise in measurement inaccuracies and randomness in the auto-tuning strategies.

5.8 Results

Figure 5.11 shows, for each benchmark, the average best kernel runtime as a function of the auto-tuning time budget. The runtimes are averaged over the 100 experiments. The error bars show the standard deviation of the best kernel runtimes, across the experiments. Thus, they are an indication of the robustness of each strategy. A smaller error bar implies that the strategy is more likely to find the average kernel runtime shown. A number of observations can be made from the figure.

The first observation is that for all the benchmarks, both strategies eventually converge to the same kernel runtime for a sufficiently long auto-tuning time budget. This reflects that with sufficient time, both strategies converge on good performing configurations. For Kmeans, lavaMD, Streamcluster, Covariance and Correlation, they generally converge approximately around the 200s mark. Hotspot and BFS both converge very quickly at the beginning. Last, the remaining GEMM benchmark tends to approach convergence around the 400s mark. Furthermore, we can draw a conclusion that the amount of time for a GPU program to converge to a high performing run time

The second observation is that our strategy is able converge on this good performing configuration with less auto-tuning time budget compared to OpenTuner’s AUC Bandit strategy. Furthermore, at any given auto-tuning time budget, our strategy finds kernel configurations that have the same or better average best kernel runtime compared to to OpenTuner strategy. There are exceptions for very short time budget, and this is due to the need of the regression tree strategy to sample enough configurations to build the first regression tree. A very short time budget prevents the strategy from effectively doing so. The time required for our strategy to converge on good performing configuration is an intrinsic
characteristic of both the optimizations and the GPU kernel itself. It is not possible for us to have this priori knowledge. Thus, reducing the auto-tuning time budget to be too small may be detrimental to the best performing configuration found. For example, giving an auto-tuning time budget of 20 seconds versus 50 seconds for the streamcluster benchmark shows a huge performance potentially gained with just 30 extra seconds. We leave the exploration of finding an appropriate auto-tuning time budget as potential future work.

The third observation is that, broadly speaking, our strategy has smaller error bars compared to OpenTuner’s. This indicates that the former strategy is more robust than the latter. That is, it is more likely to find the average runtime when the auto-tuning experiment is run once. Other observations can be drawn from the size of the error bars at particular time intervals. Although smaller error bars imply that we have more confidence in the strategy being able to find the expected kernel run time, it also means that we have less of a chance to deviate from the expected value, and potentially a configuration that is better performing. This is an unwanted characteristic early on in the auto-tuning session, but more so ideal as we progress deeper in the exploration. High variance early on in the tuning session implies that there is potential for our strategy to do much better, while small variance late into the session gives us the confidence that our strategy does converge to good performance.

Note that the curves should ideally be monotonically decreasing. However, that is not the case in certain time frames in the figures presented. This is due to the fact that the best kernel runtime found plotted at each interval tick is an average of numerous repeated experiments. Due to the nature of random sampling being involved, the sets of data for some particular experimental runs may have contained a worse runtime found at that specific time interval compared to the other runs. Thus, it skews the averaged out runtime to be worse. It is interesting to note that for three benchmarks, Streamcluster K-means and lavaMD, good performing configurations appear to be found very early in the search. Worse performing configurations appear to be reported shortly after that, before better performing configurations are reported. This behavior is a reflection of both the randomness of the configuration selection and the limited time budget in relation to kernel execution times. It is not a reflection on the performance of the strategy.

With a short time budget (e.g., 10s), only a handful of the sampled configurations are executed. These are shorter running, i.e., better performing, configurations. Depending on the sampling and on the order in which the configurations are executed, the number of better performing configurations executed first varies over the 100 experiments we run, and the average low. However, as can be seen from the error bars, the quality of this best configuration varies too widely make the initial sample reliable. For example, for Streamcluster, less than half of the sampled configuration, are evaluated and
the reported executions times vary by 3X.

When the time budget is larger (e.g., 30s-60s), more of the sampled configurations, particularly worse performing ones, are executed. Therefore, the reported best performing configuration, averaged over the 100 experiments, increases. However, once regression trees are build and the search strategy is applied, the average reported performance improves and has more predictability over the 100 experiments.

When the time budget is larger (e.g., 20s-40s), more of the sampled configurations, particularly worse performing ones, are executed. Therefore, the reported best performing configuration, averaged over the 100 experiments, increases. However, once regression trees are build and the search strategy is applied, the average reported performance improves and has more predictability over the 100 experiments.

These results show that our regression tree strategy: (1) is effective in finding good performing configuration with reasonably small auto-tuning time budgets, and (2) has clear benefit over OpenTuner’s AUC bandit strategy for almost all the benchmarks. Even for benchmarks where there is no benefit (e.g., Hotspot and BFS), our strategy does not under perform that of OpenTuner.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>AUC Bandit Best Speedup</th>
<th>Regression Tree Best Speedup</th>
<th>AUC Bandit Total # of Samples Taken</th>
<th>Regression Tree Total # of Samples Taken</th>
<th>AUC Bandit Total Kernel Runtime (s)</th>
<th>Regression Tree Total Kernel Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotspot</td>
<td>1.099</td>
<td>1.103</td>
<td>221</td>
<td>236</td>
<td>218</td>
<td>209</td>
</tr>
<tr>
<td>K-means</td>
<td>1.026</td>
<td>1.058</td>
<td>33</td>
<td>31</td>
<td>224</td>
<td>230</td>
</tr>
<tr>
<td>BFS</td>
<td>1.226</td>
<td>1.238</td>
<td>245</td>
<td>246</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>1.037</td>
<td>1.052</td>
<td>47</td>
<td>47</td>
<td>144</td>
<td>144</td>
</tr>
<tr>
<td>lavaMD</td>
<td>1.030</td>
<td>1.039</td>
<td>57</td>
<td>96</td>
<td>326</td>
<td>267</td>
</tr>
<tr>
<td>GEMM</td>
<td>0.825</td>
<td>1.847</td>
<td>17</td>
<td>16</td>
<td>305</td>
<td>318</td>
</tr>
<tr>
<td>Covariance</td>
<td>1.494</td>
<td>1.479</td>
<td>82</td>
<td>120</td>
<td>401</td>
<td>373</td>
</tr>
<tr>
<td>Correlation</td>
<td>1.529</td>
<td>1.472</td>
<td>81</td>
<td>116</td>
<td>402</td>
<td>371</td>
</tr>
</tbody>
</table>

Table 5.3: Speedup and number of configurations explored

Table 5.3 provides additional details on the number and performance of the configurations explored by our strategy, again compared to those explored by OpenTuner’s strategy. It shows, for each strategy, the speedup of the best configuration discovered, the number of kernel configurations explored and the total runtime of these configurations. For both strategies, an auto-tuning time of 400s is used (except for covariance and correlation, where 500s are used). There are several observations that can be made.

The first is that the speedups of the best configuration discovered by our strategy is better than the speedup of the best configuration discovered by OpenTuner’s AUC Bandit strategy, with the exception of correlation and covariance where it is slightly worse. This confirms what Figure 5.11 already shows: given a sufficiently long auto-tuning time budget, our strategy generally discovers better configurations.
The second observation is that our strategy explores roughly the same number of configurations for several benchmarks (e.g., K-means, streamcluster and GEMM), but it does explore more configurations for other benchmarks, e.g., lavaMD, covariance and correlation. However, in spite of the higher number of configurations explored, the runtime of these configurations for our strategy is less than the corresponding time for AUC Bandit. This indicates that our strategy explores better performing configuration by focusing the sampling on better regions of the space.

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Naive Random Sampling Speedup</th>
<th>AUC Bandit Speedup</th>
<th>Regression Tree Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotspot</td>
<td>1.064</td>
<td>1.092</td>
<td>1.096</td>
</tr>
<tr>
<td>K-means</td>
<td>0.781</td>
<td>0.658</td>
<td>0.707</td>
</tr>
<tr>
<td>BFS</td>
<td>1.145</td>
<td>1.213</td>
<td>1.224</td>
</tr>
<tr>
<td>Streamcluster</td>
<td>1.019</td>
<td>0.945</td>
<td>1.018</td>
</tr>
<tr>
<td>lavaMD</td>
<td>1.003</td>
<td>0.815</td>
<td>0.998</td>
</tr>
<tr>
<td>GEMM</td>
<td>0.486</td>
<td>0.770</td>
<td>1.089</td>
</tr>
<tr>
<td>Covariance</td>
<td>0.952</td>
<td>0.769</td>
<td>0.872</td>
</tr>
<tr>
<td>Correlation</td>
<td>0.359</td>
<td>0.886</td>
<td>0.873</td>
</tr>
</tbody>
</table>

Table 5.4: Speedup comparisons

Table 5.4 compares the speedup of the best kernel configuration discovered by our regression tree strategy and OpenTuner’s AUC Bandit strategy, assuming an auto-tuning time budget of 200s. Thus, it demonstrates the effectiveness of our strategy when a limited auto-tuning time budget is used. It further compares to a naive sampling strategy that samples the entire space until the auto-tuning time budget is exhausted and reports the best configuration found. Thus, it represents a most naive approach to auto-tuning and is only used to establish a reference. The table shows that the our strategy, with a limited time budget, is able to obtain better performing configurations than the other two strategies. This, again, validates our strategy.

5.9 Summary

To summarize the results above, we have shown the following:

- Our regression tree strategy is able converge on this good performing configuration with less auto-tuning time budget compared to OpenTuner’s AUC Bandit strategy.

- Our strategy is more robust in terms of finding the expected best average performance compared to OpenTuner’s AUC Bandit strategy.
• In the cases where our strategy is unable to find better performance compared to OpenTuner, the best result we achieve is still competitive.

• Our strategy is able to explore more high performing configurations compared to OpenTuner using the same time budget.
Figure 5.10: Sweep of $\sigma$ values across all benchmarks
Figure 5.11: Average best kernel runtime found vs auto-tuning time
Chapter 6

Related Work

There has been previous work dealing with performance auto-tuning, some of which utilizes machine learning techniques such as regression trees. The following chapter will highlight some of these works.

There are a number of existing auto-tuning frameworks, including Active Harmony [13], Insieme [14], PATUS [28] and OpenTuner [12]. These frameworks use a variety of optimization heuristics, including Nelder-Mead, differential evolution and gradient descent.

Active Harmony is an auto-tuning framework that focuses on tuning program run time using a heuristic based numerical method called Nelder Mead. The heuristic used in this work is based on the gradient direction in which it predicts has lowest run time. The technique iteratively optimizes the objective (e.g., run time) based on the heuristic. The tuning benchmarks used in their work contain parameters that can be varied which results in different run times and thus forms an optimization space.

Insieme is another auto-tuning framework that focuses on tuning program run time, but uses a different technique called Differential Evolution (DE). Contrary to the Nelder Mead approach, DE does not rely on any heuristics. The technique maintains a list of configurations from the optimization space, and creates new configurations by combining existing ones.

Christen, Schenk, and Burkhart presents an auto-tuning framework, PATUS, that focuses on a compilers approach that converts stencil code into an optimized version. It supports the generation of architecture specific stencil code that can incorporates expert knowledge which enables optimization potential beyond capabilities of current compilers.

Furthermore, Ansel, Kamil, Veeramachaneni, Ragan-Kelley, Bosboom, O’Reilly, and Amarasinghe implemented an auto-tuning framework, OpenTuner [12], which combines an ensemble of techniques using a multi-armed bandit heuristic that tunes a given program. Techniques include Nelder Mead,
Our work is based off this framework, where we added support for tuning GPU kernels specifically.

Last, Holewinski, Pouchet, and Sadayappan focused their work on tuning GPU kernels across multiple domains such as synthetic, PDE solvers, image processing, and gradient descent [29]. Optimizations include work done per thread and the launch configuration; similar to some of the optimizations in our work.

Our work focuses on the design of an optimization heuristics as opposed to an auto-tuning framework. As such, it extends and complements this earlier work. Further, we compare our strategy to the ones used by one of these frameworks, OpenTuner, and demonstrate its superiority. We also do not focus specifically on any particular application domain. This is reflected in our evaluation section where the benchmarks used came from a variety of different domains such as graph traversal, physics simulation, and linear algebra.

There is also growing work on the use of machine learning techniques to automatically tune the performance of GPU programs. In this work, a number of training GPU programs are used to train a machine learning model, which is then used to predict how to apply optimizations for a new program. Examples include optimizing GPU code by thread coarsening [5], to use local memory [7] or to predict memory loading technique [30] in stencil code.

Magni, Dubach, and O’Boyle [5] uses a neural network machine learning model predict kernel thread coarsening factors. The neural network model is trained using provided kernel data. The model then predicts what the coarsening factor should be for a new given kernel. The framework utilizes LLVM Clang to parse OpenCL kernel code and applies the appropriate thread coarsening factor. The framework emits transformed OpenCL and then is compiled.

Han and Abdelrahman [7] uses synthetic benchmarks to train a random forest machine learning model that predicts whether a new GPU kernel should use local memory or not. Similarly, Garvey and Abdelrahman [30] utilizes both a random forest machine learning model and heuristics to tune stencil kernels. The synthetic stencil kernels are parameterized with various optimizations. The random forest model is used to predict the optimal memory loading technique optimization. The heuristics are responsible for tuning the remaining optimizations such as work group sizes, and work merging factors. In contrast, our work focuses on optimizing a variety of optimizations and benchmarks that are supposed to represent more realistic GPU applications.

Last, the work of Mametjanov, Balaparakash, Choudary, Hovland, Wild, and Sabin was not focused on performance auto-tuning of neither a specific application domain, nor GPU kernels. Instead, it was for FPGA synthesis, map and place-and-route design tools [31]. Their approach used the concept of
taking samples of optimization parameters and using it as training data for a machine learning model; similar to our work. It is apparent that a sampling approach for auto-tuning has been investigated before and proven effective in other domains. Our work shows that taking samples and using it as training data for GPU kernel auto-tuning is also effective.

Our approach is application domain agnostic for GPU kernels and uses heuristics to prune the optimization space of a single GPU program. Albeit, it trains a regression tree model with samples of this application. Furthermore, our work focuses not on predicting high performing configurations within the optimization space, but a way to narrow down the space into a small region where high performing configurations are likely to be found by random sampling.

Regression trees have been used as a supervised machine learning model to predict the performance of GPU kernels under various GPU optimizations. In particular, Starchart [27] is a framework that takes pre-evaluated GPU configurations as training data and characterizes the optimization space into partitions based on estimated metrics (e.g., run time or power). Given a configuration with an unknown metric value, the framework uses the trained model to predict the estimated metric.

Bergstra, Pinto, and Cox has also used regression trees for auto-tuning a specific CUDA GPU program: filterbank correlation [32]. Similar to our strategy where we iteratively train regression trees, their work utilizes iterative regression trees to fit training data as accurately as possible, called boosted regression trees. Note that the boosted regression trees model is used for prediction on a specific GPU program.

Our work focuses on using regression trees to prune the optimization space for purposes of optimizing a single kernel. A common theme with all the works discussed so far that uses machine learning models is the fact that they are used in a classical manner: prediction or classification. Instead of using a machine learning model for prediction or classification, we are utilizing the model’s by-product information to reduce the size of the optimization space. The resulting region from the optimization space contains high performing configurations we can randomly sample from. Thus, we do not need to predict or classify configurations based on its performance for us to achieve good results in an auto-tuning perspective. We do not limit the number of iterative regression trees to be built and continuously train successive regression trees while the auto-tuning budget permits. Last, our approach does not rely on pre-evaluated training data, but instead collects it by evaluating configurations in real time as part of the auto-tuning session.

From the variety of work mentioned above, the following can be observed. First, optimizing GPGPU kernels is a difficult task due to the difficulty of exploiting hardware platform architectures on different GPU application domains, along with different optimizations being applied to the source code. Second,
there is a plethora of different approaches and strategies that have been proven effective in GPGPU performance auto-tuning. Last, there is no definitive answer as to what is the best and optimal approach to this field of research. Thus, it demonstrates the importance of continuous research to be done on GPU platforms and auto-tuning.
Chapter 7

Conclusion

In this paper, we presented a novel strategy for automatic performance tuning of GPU programs. The novelty of the strategy stems from the use of regression trees, a well-known machine learning model, to prune the optimization space of GPU program configurations. A small set of initial sample configurations are taken in the entire optimization space. These samples are used to build a regression tree. Additional samples are taken in the tree’s region with the best mean sample runtime and a new regression tree is built with the cumulative samples. The aim is to narrow down the size of this region. This process of sampling and building regression trees is repeated until the region is small enough or the time allocated for auto-tuning is exhausted.

We implement our strategy in OpenTuner, an open source auto-tuning framework. We evaluate our strategy against the AUC Bandit strategy used by OpenTuner. We use a set of 8 benchmarks to which various optimizations are applied and evaluate their performance on an Nvidia GTX 1060 GPU. Our experimental evaluation shows that our strategy is effective in obtaining high performing configurations. Compared to OpenTuner’s, our strategy is able to obtain better performing configurations with less auto-tuning time budgets. Further, the narrowing of the search on the best performing region of the regression tree makes our strategy generally examines better performing configurations.

7.1 Future Work

There are several possible pieces of work to extend the current results we have so far. We defined and evaluated our sampling based approach strategy. However, there are several aspect of the strategy that can be further explored, and they form possible directions for future work. First, the values of the base factor $b$ and the repeated samples threshold $r$. Future work may consider exploring the impact of these
The strategy can be evaluated on more and longer running applications. Furthermore, it is worth exploring a potential heuristic for the amount of auto-tuning time budget a GPU kernel should have such that we are likely to be able to find a high performing configuration. One possible approach could be similar to how we swept for a default value of sigma, but instead we iterate through different time budgets over repeated experiments.

The benchmark themselves have various optimizations applied to them. It would be interesting to investigate the possibility of applying more optimizations to them. This gives an opportunity for our strategy to be evaluated under the circumstance where the optimization space is much bigger and more complex.

Another interesting piece of future work is to incorporate our regression tree auto-tuning strategy into OpenTuner’s list ensemble of techniques, and then evaluate the performance of the combined strategies.

The machine learning model, regression trees, have shown to be effective in pruning the optimization space. Another possible direction for future work could be to explore other machine learning models to prune the optimization space and evaluate its effectiveness.

The current implementation of the strategy is an extension of OpenTuner, which is written in the Python programming language. One of the main reasons is because we wanted to compare our strategy to what OpenTuner has already. However, if we do not wish to compare results and we are more focused on the framework’s performance itself, it is possible to create a framework written in a programming language that can yield higher performance, such as C++, so that our auto-tuning time budget can be used more efficiently: less time spent on the framework overhead computation.
Bibliography


