Performance Prediction of Programs on Heterogeneous and Massively-Parallel Architectures

Uri Shomroni
Performance Prediction of Programs on Heterogeneous and Massively-Parallel Architectures

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Uri Shomroni

Submitted to the Senate of the Technion — Israel Institute of Technology
Tishrei 5780 Haifa October 2019
This research was carried out under the supervision of Prof. Avi Mendelson, in the Faculty of Computer Science.

Acknowledgements

I would like to thank

My advisor, Professor Avi Mendelson, for his endless support and patience.
My family, for their encouragement and interest in the work.
Libby Ben-Naftali, for her help in language editing.
# Contents

List of Figures

List of Tables

Abstract 1

Abbreviations and Notations 3

1 Introduction

1.1 Motivation and Existing Works 6

1.2 Goals 7

1.3 Scope 8

1.4 Organization 8

2 Proposed Approach

2.1 Assumptions 11

2.2 Target Characterization 12

2.3 Base Program Characterization 12

2.4 Performance Measurement and Analysis 13

3 Experimental Environment and Software Frameworks 15

3.1 Targets 15

3.2 Workloads 16

3.3 Performance Measurement 17

3.4 Machine Learning 18

3.4.1 Use of Machine Learning in This Work 18

3.4.2 Value Transformations 19

3.4.3 Results Comparison and Additional Experimentation 20

4 Results 21

4.1 Target 1 21

4.1.1 Linear 21

4.1.2 Reciprocal 22

4.1.3 Quadratic 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.4 Logarithmic</td>
<td>24</td>
</tr>
<tr>
<td>4.1.5 Target 1 Results Overview</td>
<td>25</td>
</tr>
<tr>
<td>4.2 Target 2</td>
<td>27</td>
</tr>
<tr>
<td>4.2.1 Target 2 Results Overview</td>
<td>27</td>
</tr>
<tr>
<td>4.2.2 Target 2 Detailed Results</td>
<td>27</td>
</tr>
<tr>
<td>4.3 Target 3</td>
<td>30</td>
</tr>
<tr>
<td>4.3.1 Target 3 Results Overview</td>
<td>30</td>
</tr>
<tr>
<td>4.3.2 Target 3 Detailed Results</td>
<td>30</td>
</tr>
<tr>
<td>4.4 Target 4</td>
<td>33</td>
</tr>
<tr>
<td>4.4.1 Target 4 Results Overview</td>
<td>33</td>
</tr>
<tr>
<td>4.4.2 Target 4 Detailed Results</td>
<td>34</td>
</tr>
<tr>
<td>4.5 Target 5</td>
<td>36</td>
</tr>
<tr>
<td>4.5.1 Target 5 Results Overview</td>
<td>36</td>
</tr>
<tr>
<td>4.5.2 Target 5 Detailed Results</td>
<td>36</td>
</tr>
<tr>
<td>4.6 Target 6</td>
<td>39</td>
</tr>
<tr>
<td>4.6.1 Target 6 Results Overview</td>
<td>39</td>
</tr>
<tr>
<td>4.6.2 Target 6 Detailed Results</td>
<td>40</td>
</tr>
<tr>
<td>4.7 Target 7</td>
<td>42</td>
</tr>
<tr>
<td>4.7.1 Target 7 Results Overview</td>
<td>42</td>
</tr>
<tr>
<td>4.7.2 Target 7 Detailed Results</td>
<td>42</td>
</tr>
<tr>
<td>4.8 Result Summary and Discussion</td>
<td>45</td>
</tr>
<tr>
<td>4.8.1 Meta-Analysis (Feature Set Reduction)</td>
<td>46</td>
</tr>
<tr>
<td>5 Future work</td>
<td>49</td>
</tr>
<tr>
<td>5.1 Further Results</td>
<td>49</td>
</tr>
<tr>
<td>5.2 Improvements to the Method</td>
<td>49</td>
</tr>
<tr>
<td>5.3 Other Applications of the Method</td>
<td>50</td>
</tr>
<tr>
<td>5.3.1 Other Frameworks</td>
<td>50</td>
</tr>
<tr>
<td>5.3.2 Results on Optimized Samples</td>
<td>50</td>
</tr>
<tr>
<td>5.3.3 Application to GPU Optimizations</td>
<td>50</td>
</tr>
<tr>
<td>5.4 Meta-Analysis</td>
<td>51</td>
</tr>
<tr>
<td>5.4.1 Sample Meta-Analysis</td>
<td>51</td>
</tr>
<tr>
<td>5.4.2 Sample Set Composition Meta-Analysis</td>
<td>51</td>
</tr>
<tr>
<td>6 Conclusion</td>
<td>53</td>
</tr>
<tr>
<td>Hebrew Abstract</td>
<td>i</td>
</tr>
</tbody>
</table>
List of Figures

4.1 Target 1 - linear value transformation ................................. 22
4.2 Target 1 - reciprocal value transformation ............................ 23
4.3 Target 1 - quadratic value transformation .............................. 24
4.4 Target 1 - logarithmic value transformation .......................... 25
4.5 Target 1 - threshold tests ............................................. 26
4.6 Target 2 - threshold tests ............................................. 27
4.7 Target 2 - linear value transformation ................................ 28
4.8 Target 2 - reciprocal value transformation ............................ 28
4.9 Target 2 - quadratic value transformation .............................. 28
4.10 Target 2 - logarithmic value transformation ......................... 29
4.11 Target 3 - threshold tests ............................................. 30
4.12 Target 3 - linear value transformation ................................ 31
4.13 Target 3 - reciprocal value transformation ........................... 31
4.14 Target 3 - quadratic value transformation ............................ 31
4.15 Target 3 - logarithmic value transformation .......................... 32
4.16 Target 4 - threshold tests ............................................. 33
4.17 Target 4 - linear value transformation ................................ 34
4.18 Target 4 - reciprocal value transformation ............................ 34
4.19 Target 4 - quadratic value transformation ............................ 34
4.20 Target 4 - logarithmic value transformation .......................... 35
4.21 Target 5 - threshold tests ............................................. 36
4.22 Target 5 - linear value transformation ................................ 37
4.23 Target 5 - reciprocal value transformation ........................... 37
4.24 Target 5 - quadratic value transformation .............................. 37
4.25 Target 5 - logarithmic value transformation .......................... 38
4.26 Target 6 - threshold tests ............................................. 39
4.27 Target 6 - linear value transformation ................................ 40
4.28 Target 6 - reciprocal value transformation ............................ 40
4.29 Target 6 - quadratic value transformation ............................ 40
4.30 Target 6 - logarithmic value transformation .......................... 41
4.31 Target 7 - threshold tests ............................................. 42
4.32 Target 7 - linear value transformation ................................ 43
4.33 Target 7 - reciprocal value transformation ................. 43
4.34 Target 7 - quadratic value transformation .................. 43
4.35 Target 7 - logarithmic value transformation ............... 44
4.36 Target 1 - counter percentile meta-analysis ............... 46
List of Tables

3.1 Target specifications ........................................... 16
3.2 Sample applications ........................................... 17
3.3 Value transformations example ................................. 20
Abstract

Moore’s Law, stating that computation units in computer systems double their density and computation power every two years, is approaching the limit of its physical possibility. To maintain the computation power and power efficiency growth rates, more and more parallel computation and massively parallel processors must be integrated into computer programs and systems. This change has led to massively parallel, throughput-oriented processors becoming increasingly common, and to computations being executed on them being more varied and more complex. These unique processors operate differently than sequential latency-oriented processors, and development targeting them is often performed by specialized experts. Maximizing the benefit of these processors requires algorithms to be implemented differently than the sequential algorithms.

This change is often very time-consuming and is not guaranteed to provide good enough of a return on investment (ROI) in performance compared to the effort it will take. Most programmers do not have a clear view or estimate of when and how much would a program benefit from being ported to a parallel system, what considerations to apply when porting the programs, the effects of choices such as data ordering on the performance of parallel algorithms, and so forth. Existing tools for prediction of the performance of parallel programs require porting the program to the parallel environment in advance, sometimes by automated tools. This porting is a complex action, and executing it in a simplistic or automated manner is liable to provide lesser results, misrepresenting the potential of a well-executed port.

This thesis proposes an approach to predicting the performance gain from porting a program from the CPU, a latency-oriented processor, to the GPU, a throughput-oriented processor, based on the characterization of the program as measured by performance counters when being executed on the CPU only. The method calls to feed these measurements to a machine learning model trained on a given set of benchmarks, predicting the speedup. The method was designed to allow easy expansion and reproduction of the results, by choice of open standard environments and computation tools. It was then applied to 7 computer systems with a variety of parallel processors, and with a set of benchmark applications common for testing in the field. The predictions in the machine learning model were recreated using several value transformations, to examine the possible correlations between the CPU
performance counters and the results.

The experiments show that the predictions’ accuracy is similar or better than existing methods, that usually require putting in the effort of converting the algorithm as a starting point. With the application of machine learning algorithms, measurements of CPU performance counters have shown a fairly good prediction of the logarithm of the GPU speedup value (within $x^{1/3} - x^3$ of the actual value for most benchmarks). Comparison of the predictions against thresholds has shown over 80% prediction accuracy for all thresholds examined in all experiments. A correlation was found between the number of active CPU performance counters and the accuracy of the predictions. The sample set was also verified to not provide false positive results by performing the experiments with three other value transformations that yielded much lower accuracy. The other value transformations proved very sensitive to the existence of performance counter values outside the range of the machine learning’s training set: which made them estimate values far from the actual results, and even negative values, which are impossible. In contrast, the logarithmic transformation managed to handle these measurements well. In addition, a secondary experiment performed on one of the systems showed that the method, as executed, did not suffer from overfitting due to having too small of a benchmark pool.
Abbreviations and Notations

Computer systems

*Sequential* A property of computer systems and algorithms. Following a set of instructions one after the other, operating on one piece of data at a time.

*Parallel* A property of computer systems and algorithms. Following multiple sets of instructions at a time, or operating on multiple piece of data at a time.

*Massively parallel* A property of computer systems and algorithms. Parallel operations on sets of data measuring in scales of hundreds and up.

*Heterogenous* A property of computer systems and algorithms. Combining multiple operation types, typicaling sequential and (massively) parallel.

*CPU* Central Processing Unit. a sequential processor.

*GPU* Graphics Processing Unit. a parallel processor.

Experiment Data and Environment

*Performance counter* Measurements provided by a piece of hardware describing its operation. Typically provided as averages or counts of specific events or internal values over the period of processing a program. Usually denoted by the piece of hardware that provided them. e.g. “CPU Performance Counters”.

*Benchmark* A computer program used to test and measure various properties of the system’s operation. Benchmarks usually come in sets that attempt to cover various aspects of a system’s operation.

*OpenCL (Open Computing Language)* An open-standard software framework which abstracts massively parallel algorithm operation over hardware of various types and from various vendors.

*OpenMP (Open Multi – Processing)* An open-standard software framework which abstracts massively parallel algorithm operation over CPU hardware.
**speedup**  The relation in runtime between two processors running ported versions of the same algorithm or application. Unless otherwise noted, the baseline value is the CPU value. For example, for a given application and target, $GPU \text{ speedup} = \frac{GPU \text{ run time}}{CPU \text{ run time}}$

**Machine Learning**

*Machine learning*  Algorithms and statistical models which allow computer systems to perform specific tasks without being given explicit instructions, relying on pattern recognition.

*Inference*  A step in machine learning, applying a procedurally-created algorithm to target data.

*Testing set; Validation set*  The input data used in inference.

*Training*  A step in machine learning, the creation of an inference mechanism based on input data.

*Training set*  The input data used in training.

*Features*  The inputs (parameters) of an inference algorithm, also used in training.

*Tag; Value*  The output of an inference algorithm. The input of a training algorithm is a set of (feature vector; value) pairs.

*Linear classifier*  A type of value prediction model which receives numeric values and outputs a result by applying a linear transformation on the input.

*Linear regressor*  A basic type of training algorithm. The inference algorithms created by a linear regressor are linear classifiers.

*Overfitting*  A common issue with machine learning models. If the feature set is too large, or the training set is not varied enough, the training algorithm may produce an inference algorithm that relies too heavily on irrelevant parts of the input.

*False correlation*  A common issue with machine learning models. If the testing set is too similar to the training and validation sets, the results may appear to operate correctly, but are in fact skewed and will fail as soon as a new, and completely different, data point will be introduced.
Chapter 1

Introduction

To keep up with the growing demand for computation power, modern software developers are often required to use massively parallel or heterogenous algorithms. These algorithms are very different in nature and paradigms from “classic” sequential algorithms, but require significant effort to be implemented. A method that allows to predict performance on massively parallel solutions before committing entirely to implementing such a solution would prove very useful, since it is not trivial predicting ahead of time how much performance such an effort would provide (or even if it can be improved at all).

The main computation unit of most computer systems is comprised of one or more single-threaded cores; able to run sequential algorithms, mainly working on a single piece of data with a single operation (instruction) at a time - a SISD (Single Instruction, Single Data) processor. These processors have been developed over time to perform these sequential algorithms most efficiently, by way of adding abilities such as branch prediction, instruction and data pre-fetching, making them latency-oriented. Modern processors usually have several such cores, but the amount of parallelism in them is small, and is handled automatically by the processor or the various operating systems and run-time environments. Typically, the developer will write the algorithm as if it were to be run on a completely sequential processor. In this case, the few elements of parallelism offered by the processor will not be explicit, but rather will be added by the compiler or the processor itself.

However, Moore’s law (observing that the number of transistors in a dense integrated circuit doubles every two years), that enabled computational power to grow for these processors within a single computation unit, is reaching its physical limit - meaning that in order to keep the growth pace in processing larger amounts of data, computers must start applying more parallelism. Using multiple latency-oriented processors to perform massive parallelism is possible, but greatly increases the power budget, as well as the physical size, of the system. Due to this requirement, in recent times massively-parallel processors have become much more prevalent in the general computation scene - these are often derived from processors designated for a
specific task, such as graphics processors. These processors are *throughput-oriented*, and operate by performing a single instruction on multiple data values, or SIMD (Single Instruction, Multiple Data), working more efficiently than their SISD equivalents, as long as the problem is attuned to their abilities. This means the developer of the algorithm has to make it (theoretically) infinitely parallelizable, or as close to it as possible, to maximize the benefit from these processors’ unique properties.

An important consideration of the processor type to use is power consumption - throughput-oriented processors are typically more power-efficient per operation, due to foregoing latency-hiding mechanisms and to unification of instruction logic. Conversely, applying running a sequential algorithm on a throughput-oriented processor will waste power by not utilizing most of the threads in the parallelized hardware.

The benefit of throughput-oriented processors when solving appropriate problems is widely documented and proven, and is taken as an assumption for most optimization research[WS10]. Yet, this performance “increase” varies from several order-of-magnitude performance increases, down to performance decreases for algorithms which are sequential in nature or that can be heavily optimized on a latency-oriented processor. Some systems, wishing to benefit from both worlds, combine the two for a heterogeneous computing environment, containing different types of processors.

### 1.1 Motivation and Existing Works

This property of being suitable to throughput- versus latency-oriented processors, however, can be hard to deduce simply from the problem or algorithm. Most research in the field takes a deductive approach: Port the algorithm to a massively parallel environment, perform some measurements to compare both environments, and deduce whether the problem is better handled with a sequential or a parallel algorithm[GO11, WS12, CBNV13], or by offering to create both versions and use an active or a reactive system to load-balance between them[BSCJ13].

Doing so may rely on an automatically-generated or directly ported version of the original sequential code[GWO13] - possibly resulting in skewed results due to the sequential thinking of the original algorithm - where a properly optimized or specially-crafted approach may yield even better results. It is worth noting that despite this possible skew, most published research shows great performance improvements with massively parallel architectures, which indicates that specially-crafted algorithms may perform even better.

This research acknowledges that properly porting an algorithm to a heterogeneous or massively parallel environment is by no means a simple task, and hence, taking such a port as the base line for performance testing might be too time-consuming: if the algorithm itself, before it is ported, can be shown to not benefit greatly from
porting to a heterogeneous or massively parallel environment; or if it can be shown that the algorithm is best suited for a purely massively-parallel system (avoiding the intricacies of properly managing a heterogeneous system), the porting time can be reduced or the porting itself can be avoided if it is not worth the while. Given the prediction of the return, and knowing the investment required for each option, will allow a potential future user to calculate the return-on-investment (ROI) and avoid investing in fruitless paths.

1.2 Goals

This thesis attempts to describe and test a method of predicting the performance of an algorithm when ported to a throughput-oriented processor, based solely on the non-parallelized algorithm. To do so, it will aim to show that by careful analysis of characteristics of a sequential program, it is possible to predict its behavior on a massively parallel or heterogenous system. This will allow software developers to adjust their expectations of the ported algorithm, or even avoid this difficulty in the initial porting (which is required for the existing prediction methods) altogether, if the results are expected to be of less value than the effort.

A previous work by IBM Research\[BFA14\] has attempted to make such predictions, but that work used a CPU algorithm that was already parallelized (using OpenMP) and took measurements on it with by instrumenting the code, thus changing its runtime behavior. It also focused on the computation loops that are parallelized by OpenMP and not on the entire application run-time. This work’s results were largely positive and showed the possibility of making these predictions. However, this thesis theorizes that it is possible to take the prediction even a step further: using only the non-parallelized version of the algorithm and measuring its run-time parameters externally, that is, without modifying the application code.

To test the veracity of this thesis, a set of GPU benchmark programs was chosen and converted to CPU equivalents. Then, CPU performance counter measurements were taken for each of the sequential programs’ runtime, as well as what is the actual performance increase (or decrease) for each one. Finally, given the measurements for all sequential programs, and given the performance changes for all benchmarks save one, a prediction of the removed value was made using a basic machine learning algorithm. This cross-check process was repeated for each benchmark (training the machine learning models on all benchmarks save one, and using the model to predict the one excluded), and the actual results were compared to the predicted values.

Various functions were applied to the performance increase multiplier before using it in the machine learning algorithm, to avoid any assumption on the type of correlation between the CPU performance counters and the GPU speedup value. This also served to help disprove the notion of false correlation (which may arise in
machine learning should the training set and validation set contain programs which are nearly identical) - should some, but not all, of these functions be used to give good predictions on the value, it would show that the correlation is not false for the cases where it happened.

1.3 Scope

In general, this work aims to describe its methods in such a way that future works given more (or different) resources could be made using the same methods to expand the insights gained from it. This is covered in more detail in the fifth chapter.

The method used in this research can be used to make predictions on any given set of applications, and could be adapted to an automated tool which builds a machine learning model and uses it to predict results for all future applications on the same machine. Due to the complexity of converting new applications to run on the GPU, a specific benchmark suite was chosen - any new programs added to the suite was added as both a learning data point and as a test case.

Any metric available on a sequential program could be used in this method. For simplicity and ease of automation, only runtime event counting was performed on the sequential programs. It is possible to provide more metrics (such as information from static analysis of the source code or the raw input for the algorithm), and the prediction model allows arbitrarily adding more values for each application.

One of the systems was chosen and its results were used to perform meta-analysis on the performance counters and the counter set used in the machine learning. This was done to explore the risk of overfitting in the machine learning algorithm. As this analysis is far more time-consuming, this was not applied to the other systems.

1.4 Organization

The second chapter of this work presents the new approach for predicting the potential of massively parallel architectures. This section describes the model used and extends discussion on the methods applied to characterize and measure results (both the training data from the sequential programs and the result data from the parallel programs).

The third chapter focuses the experimental environment used in the work and describes the specifics and details for the hardware and software used in it.

The fourth chapter contains the results of these tests, as performed on the selected targets. Each set of results (real versus predicted) is analyzed, and compared to results from other convolutions on the same sequential measurements (as explained in the “Suggested approach” section above).

The fifth chapter describes how this methodology can be expanded or applied to
other targets, other benchmark suites, and broader definitions of targets, enabling future works to be based off this one.

The sixth and final chapter summarizes the results of the work and the conclusions that can be drawn from their analysis.
Chapter 2

Proposed Approach

This thesis aims to present a new approach to predicting the speedup a program will gain from being ported to the GPU, based only on its execution on a latency-oriented processor, such as a sequential processor. Thus, the prediction is based only on measurements performed on its CPU implementation. The method includes learning the specific execution environment - both CPU and GPU - by measurements on pre-ported algorithms (benchmarks), then using the trained model to predict the speedup for new algorithms.

As previously discussed, the act of converting and porting a sequential algorithm to a massively parallel one is not trivial. Therefore, this research aims to give a completely predictive (rather than deductive) result: take a CPU-based sequential algorithm (the C language was chosen as a representative for this), and without yet porting it to any other architecture or language, attempt to predict the performance benefit, if any, that will come from porting it to the various architectures and systems this research will focus on.

To apply the results of this research, the user must have a sequential implementation of their algorithm on CPU, and can define the platform to which she would like to port the program. As discussed previously, an existing solution is automated vectorization followed by measurements on the converted program - but these automated conversions are often far from optimized for the GPU. However, much of the performance improvement for many algorithms comes only after optimizing the GPU program, thus the estimate should be of the final result and not of the interim state.

2.1 Assumptions

Due to the scope of this work, it will only focus on target computers that have a CPU and a discrete GPU (not integrated GPUs, FPGA accelerators, or heterogeneous targets), without re-optimizing the sample set for the specific architecture, and running the same operating system. The hypothesis of this work is that results on these
sort of targets can still provide meaningful insight into the possibility of predicting performance using only the sequential algorithm.

2.2 Target Characterization

As a first stage, this research characterizes the various architectures that will be taken as the targets of the algorithm. The definition of a target, in this sense, is a single computer system with a CPU and a discrete GPU, which will run the sequential and parallel algorithms, respectively. The experiments described below will be performed on each target separately.

2.3 Base Program Characterization

The second stage of the research focused on measuring characteristics of the sample applications. These are the pairs of CPU and GPU programs to be used in the research. A set of GPU benchmarks and micro-benchmarks was chosen, and each was converted to a sequential version of the same algorithm. This was done to ensure that the GPU algorithms used were well-suited for the massively parallel environment and to avoid cases where incorrect porting would result in skewed predictions.

To understand why GPU algorithms were chosen as the base, consider, for example, an algorithm that adds together the values of an array of numbers into a single number.

When programming this algorithm sequentially, the order of operations does not matter, so a programmer might choose to add the value of the first array cell to the second; add this to the third cell; and so forth until the last cell of the array will contain the sum of all values. \[ S = (((A_1 + A_2) + A_3) + A_4) + \ldots \] This order of operations is highly unsuited for GPU operation, as the dependency of each step on the previous one makes it impossible to parallelize.

When programming for a massively parallel processor, one could, for example, split the values into pairs by index, then sum each pair into its left element. These left elements could then be split into pairs and added the same way, repeating this until the last element of the array contains the sum. \[ S = ((A_1 + A_2) + (A_3 + A_4)) + \ldots \]

While the number of additions remains the same (that is to say, if run sequentially - on the CPU - both algorithms would execute in a similar amount of time) - the parallelization of the second method will make it operate better when run on the GPU.

Once these pairs of sample applications were created, run-time characteristics of their run needed to be extracted. A mechanism for measuring CPU-based performance counters was put in place and applied to each of the benchmarks on each of
the targets.

The CPU-based performance counters extracted describe events encountered by the processor such as thread context switches, various level cache misses, branch mispredictions and so forth. To avoid bias caused by application run time, each counter was then averaged per time unit.

The resulting measurements for each target were then culled by removing all counters that provided the same value for all samples (typically 0).

Furthermore, as the different events vary greatly in frequency (for example, a branch misprediction is a far more common occurrence than a memory page fault). Mixing of very large and very small values in computations (such as the machine learning operation that is used below) risks suffering from loss of precision, so normalization of the values was required. Thus, the value range for each counter on each target was then normalized to the range of \([0.0, 1.0]\). This ensured the maximal value measured across all sample applications was always 1.0. As the machine learning model used is linear, this would not change the concrete results, and only served to make the machine learning operation faster and less prone to rounding errors.

The result of the characterization for each target was a set of measurements for each sample application, consisting of normalized, non-zero event frequencies and a single value representing the run time speed up from the sequential program to the parallel program.

### 2.4 Performance Measurement and Analysis

Previous works in the field used various methods in their analysis, most commonly neural networks and machine learning\[GW013, BFA14\] or feature extraction (similarity)\[CBN13\]. These methods have shown good results in statistical analysis and extrapolated result prediction.

Machine learning was chosen as the analysis method, both since it allows for automation in managing and assessing the various parameters, and since it is the standard analysis method in works of this type. This thesis uses machine learning as a tool, and does not aim to research the machine learning itself, so the analysis will stick to common methodologies, such as using 20% of the samples as a training set and applying a fairly basic model. Due to the relatively small number of samples and characteristics, a simple machine learning framework was chosen for this task - a larger feature set may have suffered inaccuracy (but may be adequate if future research uses a wider feature set). The framework was also cross-checked, by being allowed to learn on the sample set excluding each sample (in turn) and the value predicted for that sample by the inference mechanism was then compared to the actual value measured for the sample.
To avoid assuming the type of correlation between the sequential program’s characteristics and the speedup, the experiment was then repeated four times for each target, each time applying a transformation to the speedup value. The transformed speedup was used as the value for the machine learning algorithm separately each time, providing predictions that would imply different correlations.
Chapter 3

Experimental Environment and Software Frameworks

The sample set and the performance measurement tool were chosen from available open options, based on the selection in previous works in the field. These were then processed as described in the “Method” chapter.

A further consideration was given to the measurement being automated and adaptable (e.g. to availability of performance counters based on CPU hardware), allowing testing to be performed on as many targets as possible.

3.1 Targets

As mentioned previously, the targets all run the same operating system, namely Linux Ubuntu 16.04.

The C language (with, in some cases, some elements of C++) was chosen for the target CPU high-level language.

The OpenCL standard was chosen for the target GPU high-level language, for reasons of:

1. Similarity to C language, which is used for the CPU programs
2. Simpler portability between various architectures
3. Easy accessibility to the various architectures (a single base unoptimized port to OpenCL can be run on all supporting systems as well as on the CPU itself)
4. Relatively low-level API allowing for fine-tuned optimizations

OpenCL was chosen for the GPU language (rather than CUDA) to allow targets with GPUs from vendors other than NVIDIA to be used.
<table>
<thead>
<tr>
<th>Target</th>
<th>CPU</th>
<th>GPU</th>
<th>RAM</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target 1</td>
<td>Intel Core i7-6800K(^1)</td>
<td>NVIDIA Quadro GP100</td>
<td>16GB 2133Mhz</td>
<td></td>
</tr>
<tr>
<td>Target 2</td>
<td>Intel Core i7-7700HQ(^2)</td>
<td>NVIDIA Quadro GP100</td>
<td>16GB 2133Mhz</td>
<td></td>
</tr>
<tr>
<td>Target 3</td>
<td>Intel Core i7-7700HQ(^2)</td>
<td>NVIDIA Quadro GP100</td>
<td>16GB 2133Mhz</td>
<td>3</td>
</tr>
<tr>
<td>Target 4</td>
<td>Intel Core i7-6800K(^1)</td>
<td>Geoforce GTX 960</td>
<td>32GB 2133Mhz</td>
<td>4</td>
</tr>
<tr>
<td>Target 5</td>
<td>Intel Core i7-6800K(^3)</td>
<td>NVIDIA Geoforce GTX 1080</td>
<td>32GB 2133Mhz</td>
<td>4</td>
</tr>
<tr>
<td>Target 6</td>
<td>Intel Core i7-6800K(^1)</td>
<td>NVIDIA Geoforce RTX 2080</td>
<td>32GB 2133Mhz</td>
<td>4</td>
</tr>
<tr>
<td>Target 7</td>
<td>Intel Core i7-6800K(^1)</td>
<td>NVIDIA Geoforce RTX 1660Ti</td>
<td>32GB 2133Mhz</td>
<td>4</td>
</tr>
</tbody>
</table>

\(^1\)(3.4GHz, 6 physical cores, 12 virtual cores)

\(^2\)(2.8GHz, 4 physical cores, 8 virtual cores)

\(^3\)Headless configuration (no X server running)

\(^4\)Targets 4-7 are the same computer, replacing only the GPU

Table 3.1: Target specifications

3.2 Workloads

The benchmark suite “Rodinia” (http://lava.cs.virginia.edu/Rodinia/download_links.htm), version 3.0 was chosen as the base program set. As the suite is available in OpenCL, a process of "back-porting" was executed, essentially converting the OpenCL kernels back into sequential C / C++ programs, eliminating operations necessitated by the architecture such as data transferral and OpenCL kernel compilation (these were not eliminated from the samples, as those are part of the ported application’s run time). The resulting C / C++ applications do not use OpenCL at all, and can be run on a machine that does not have an OpenCL runtime installed.

This method (rather than creating a new GPU algorithm from the CPU-optimized version) was used to allow for the assessment to be similar to the one done by a programmer at the decision point between using CPU or GPU architecture. Furthermore, it is very likely that CPU-attuned optimizations would adversely affect GPU performance, due to the different natures of these two processor types. For the targets based on the CPU OpenCL runtime, a similar speedup was expected for all samples, due to similar reasoning.

These custom-ported CPU versions were compared to the CPU-optimized implementations included in the benchmark suite. They have shown almost identical results in run time. For cases where the backported version had shown a performance decrease of more than 10% of its CPU-optimized counterpart, further inspection and comparison showed that a GPU-only mechanism (such as local memory caching or input re-ordering) was accidentally left in place, and was promptly removed. This was done to disprove the notion that since the sequential programs were not optimized for the specific CPU hardware, they might mis-represent the top performance of the sequential algorithms.
3.3 Performance Measurement

The tool ocperf[ocp] was used to capture CPU performance counters during the run time of the sequential applications. This tool was chosen due to it being openly available, and also because it offers a fairly wide array of performance counters. However, being based on perf, it limited the target choice to Linux-based computers.

As a first step, each target had to have its available performance counters enumerated. Examples of events counted by ocperf include L1 and L2 cache misses, low-level cache load operations, page faults, thread context switches, system calls, kernel memory allocation operations and number of scheduler slots issued.

This effort was compounded by the fact that several counters were reported as being operationally actually failing. Including any such counter in a batch would fail the entire measurement operation, even if it included other counters which were otherwise measureable. Thus, each problematic batch would need to be investigated in order to find and remove the failing counters.

As the tool’s accuracy is also adversely affected by the number of counters being measured, the counters also needed to be split into batches and measured in turn - and results for each counter set would need to be consolidated into one large feature set for the sample. Finally, these feature sets were combined (along with the value for each sample) into a single information table, which would be used for the machine learning test.

Finally, an automation method for measurement was devised, which includes as much of this process as possible, requiring only a small amount of manual operation for each target. This was done by python scripts that run these commands, collect their measurements into pre-determined file names, then consolidate the files per sample, then the samples per target - the final product being a single table with the
samples as rows and the various metrics as columns.

3.4 Machine Learning

Machine learning is a broad term describing algorithms and statistical models which allow computer systems to perform specific tasks without being given explicit instructions, relying on patterns and inference. A machine learning operation is typically divided into two phases: The first is training, in which a special algorithm is fed data called a training set, which outputs parameters or a complete algorithm for the task. The second is inference, taking the algorithm created in the training phase, and inputting new data. The machine learning model is usually evaluated by taking a testing set of pre-labeled data and comparing the inference results to the true labels. It is commonplace to randomly select a percentage of the input data as the testing set. This selection is typically done once, to prevent data from the testing set leaking into the prediction model via repeated optimization.

A linear regressor is a basic machine learning models, which makes it accessible and thus widely used. Its input is a set of sample data points, each made up from a feature vector of numeric values (individually called characteristics or features) and a matching classification result (either a Boolean result named a tag or a numeric result named a value). Its output is an inference mechanism called a linear classifier, which predicts the tag or value for feature vectors by applying a linear transformation (essentially, a dot product of the feature vector against a set weight vector which uniquely defines the classifier).

A linear regressor is a training algorithm which outputs a linear classifier as its inference algorithm. It attempts to find the best matching linear transformation so that if its input feature vectors are fed to the classifier, the result will be as close to the matching value as possible. A linear regressor operates by creating a system of linear equations where the variables are the weights of each feature, the coefficients are the feature values, and the bias is the value. Once each pair (feature vector; value) is represented this way, the regressor attempts to procedurally solve (or, if no solution is possible, approximate) the system, yielding the weights for the linear classifier.

3.4.1 Use of Machine Learning in This Work

A best practice in machine learning benchmarks is to randomly choose 80% of the samples as the training set and use the remaining 20% as the testing set to evaluate the model’s quality. Therefore, of the 45 sample applications used, 9 (that is, 20%) were randomly chosen as the testing set, leaving the other 36 as the training set. The programs in the training set were randomized in a way that the set is varied in how much speedup they gained or slowdown they suffered. This was done to test
whether the prediction model could estimate varying values, and not only a certain range.

Both the sample set and the feature set are small compared to a typical machine learning problem, thus more complex machine learning frameworks would run the risk of overfitting or underperforming. The implementation of the linear regressor used was the one provided by the python library TensorFlow[ten].

The performance counter measurements provided by the ocperf tool were used as the features for the machine learning model. To improve the regressor’s reliability, two pre-processing operations were performed on all features: First, all features that have the same value for all samples were eliminated from the feature set. These were typically counters that were never triggered, resulting in the value 0 for all samples. Then, the remaining values were normalized so that the maximum value for each was 1.0, and the minimum possible value remained 0.0. This was done to avoid calculation issues stemming from the loss of precision when adding or multiplying very large and very small numbers together.

3.4.2 Value Transformations

The linear regressor can only attempt find a linear correlation between the features and the values. To test if a correlation could be found by applying a transformation to the values of the machine learning model, the experiment was repeated four times, each with a different value set applied to all applications, using the same source measurements, henceforth referred to as the value transformations.

It is important to note that these transformations are not applied to the features (i.e. the performance counters from ocperf), but only to the value (the speedup ratio from CPU to GPU). The same raw data was used for all the experiments, with only the value of each feature-value vector being modified by the transformation.

(a) The linear value transformation, taking the value set as measured by dividing the CPU run time by the GPU run time, (speedup)

(b) The reciprocal value transformation, using the inverse of the previous value, GPU run time divided by CPU run time (1 / speedup)

(c) The quadratic value transformation, taking the square root of the linear value transformation (sqrt(speedup))

(d) The logarithmic value transformation, taking a 10-radix logarithm of the linear value transformation (log_{10}(speedup)).

Each of these values having a linear correlation with the frequency of certain CPU events would imply different correlations between CPU and GPU run time, so each was tested for all targets.

For example, given an application A that executes in 2000 time units on the CPU and in 5 units on the GPU; and an application B that executes in 10 units on the CPU and in 1000 on the GPU - the input values sets for the linear regressor would
be

<table>
<thead>
<tr>
<th>Value transformation</th>
<th>Program A</th>
<th>Program B</th>
</tr>
</thead>
<tbody>
<tr>
<td>(CPU time)</td>
<td>2000</td>
<td>10</td>
</tr>
<tr>
<td>(GPU time)</td>
<td>5</td>
<td>1000</td>
</tr>
<tr>
<td>(GPU speedup)</td>
<td>( \frac{2000}{5} = 400 )</td>
<td>( \frac{10}{1000} = 0.01 )</td>
</tr>
<tr>
<td>Linear</td>
<td>400</td>
<td>0.01</td>
</tr>
<tr>
<td>Reciprocal</td>
<td>0.0025</td>
<td>100</td>
</tr>
<tr>
<td>Quadratic</td>
<td>20</td>
<td>0.1</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>2 + \log_{10}4</td>
<td>-2</td>
</tr>
</tbody>
</table>

Table 3.3: Value transformations example

The feature vectors for each sample would be identical in all four executions of the regressor.

3.4.3 Results Comparison and Additional Experimentation

As explained previously, the training and inference were done in a “cross-check” method, by excluding the training set the sample set, training the model on the remaining samples, then estimating the performance for the excluded samples. Note that, as the CPU counters available for each target machine are different, the results from different targets were not mixed. This would also serve to help in reproducing the results, as the value and feature sets are exclusive to the target machine.

To verify the success or failure of the prediction model, the predicted results for the testing set were compared to the actual values measured for the samples. The comparison is first described directly as graphs of prediction versus actual value, then these results are tested against hard boundaries. This was done with threshold tests (similar to previous similar works[BFA14]), considering samples that were predicted under the threshold while actually having speedup above it or vice-versa as being mispredicted.
Chapter 4

Results

Results are provided for the first target in detail, and the sections for the following targets will highlight the differences in the configuration and in the results rather than re-explaining the experiment.

4.1 Target 1

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Core i7-6800K (3.4GHz, 6 physical cores, 12 virtual cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>NVIDIA Quadro GP100</td>
</tr>
<tr>
<td>RAM</td>
<td>16GB 2133Mhz</td>
</tr>
<tr>
<td># Counters</td>
<td>1486 total, 224 non-zero</td>
</tr>
</tbody>
</table>

Speedup values for this target ranged from over 700x to under 1/3000x, providing a very wide range of results. The GPU’s tailoring for large data sets gave it a definite edge for the applications more suitable for conversion to massively parallel, while the fairly powerful CPU made the smaller and less-parallel applications have very short run times, resulting in very low “speed up” (actually, slow down) values.

One of the the value transformations (the logarithmic value transformation) yielded more accurate predictions than comparable previous studies, while the others did not perform as well.

4.1.1 Linear

First, the machine learning model was trained with the CPU counter measurements as features, and the GPU speedup value (CPU run time / GPU run time) as the labels. For each application, a linear regressor was used to create a linear classifier, using all the other applications, then used to predict the speedup value for the chosen application. This was repeated three times for each application, and the median value was taken as the final result for the application. The plots below show points using the actual speedup value as the horizontal coordinates and the median predicted value as the vertical coordinates.
In all the following charts, the red guide line indicates a perfect prediction (i.e. the predicted value being exactly equal to the actual value). The closer the points are to the red line, the better results would be considered to be. For value transformations where the actual values range very high, a portion of the x axis was stretched - without changing the y axis - to show the results in more detail. The colored background in the left plot corresponds to the area shown in the right one.

Figure 4.1: Target 1 - linear value transformation

Training on the unmodified speedup value yielded very poor results. Despite several samples having speedup multipliers of over 100, no predicted value exceeded 60.5. In addition, over one fourth of the samples were predicted to have negative values, and nearly half of the samples with a decrease in performance were predicted to grant a speedup of over 20x.

An interesting fact to note is that a negative value is not possible at all. The presence of negative values in the prediction set would imply an estimation error. However, the feature sets for each estimation sample were checked against the ranges of their respective training sets, which led to an interesting outcome: Since the estimator is a linear function, a feature vector in the convex closure of the training set’s feature vectors can be expected to yield a valid result. In fact, all samples whose predictions were negative were outside the convex closure of the training set’s feature vector. Each of these samples had at least 6 features (of 224) whose values lay outside the range between the training set’s lowest and highest values for that feature, and the vast majority had over 10 such features.

4.1.2 Reciprocal

Next, the machine learning model was trained with the same CPU counter measurements as features and the GPU slowdown value (i.e. reciprocal to the GPU speedup value, or GPU run time / CPU run time) as the values. Again, three linear classifiers were created for each application using the other applications, and the median result was taken as the final result for the application. The plotted points use the reciprocal
scale, and the red guide line indicates a perfect prediction (i.e. the predicted value being exactly equal to the actual value). The results would be considered better, the closer the points are to the red line.

Figure 4.2: Target 1 - reciprocal value transformation

Training on the reciprocal gave likewise poor results, with samples that gave better performance on CPU getting very low predicted values (including one that got a negative values), and samples that gave better results on GPU generally receiving values over 50 (despite being expected to have values under 1).

While no negative values appeared for this value transformation on this target, the results that varied the most from the expected value were again those with high numbers of features outside the range of the training set. It is likely that the larger values drew the incorrect estimations upwards and outside the negative range.

4.1.3 Quadratic

With the results from the linear speedup values landing in a range that included negative values and going up to roughly the square root of the maximum actual value, the next attempt was made by applying the square root function to the labels. The machine learning model was trained with the CPU counter measurements as features and the square root of the GPU speedup value (i.e. CPU run time / GPU run time) as the labels. The plotted points use the square root scale, and the red guide line indicates a perfect prediction (i.e. the predicted value being exactly equal to the actual value). The results would be considered better, the closer the points are to the red line.
Training on the square root provided slightly better results, with just over half the samples getting predictions within \(x0.2-5\) of the actual value, but most of the remaining samples still getting negative value predictions or very high predictions for a low-performing samples.

Similar to the linear value transformation, a negative value is meaningless. The same test described for the linear value transformation predictions was performed, and here, too, negative values only came as predictions for feature vectors which had 6 or more features outside the range between the maximum and minimum of that same feature in the training set.

4.1.4 Logarithmic

Finally, anticipating that each CPU feature’s effect might be multiplicative on the speedup value, the machine learning model was trained with the CPU counter measurements as features, and the logarithm of the GPU speedup value \((\log (\text{CPU run time} / \text{GPU run time}))\) as the labels. Since the classifier is linear, the logarithm base would not matter, so a decimal logarithm was chosen for easier analysis in terms of orders-of-magnitude. The plotted points use the logarithmic scale, and the red guide line indicates a perfect prediction (i.e. the predicted value being exactly equal to the actual value). The results would be considered better, the closer the points are to the red line. The two green lines are placed at 1 unit above and below the red line, signifying prediction errors of under a single order of magnitude.
Figure 4.4: Target 1 - logarithmic value transformation

Training on the logarithm of the speedup provided excellent results, with an average error of 0.578, and only 5 of the 45 samples having an error greater than 1.25. Over half (25/45) of the samples had an error of less than 0.4, and only one sample with an error over 2.

It is important to note, in contrast to the other value transformations, two facts:

First, this is the only value transformation for which a negative value was possible. In fact, about half of the samples had negative values.

Second, unlike the other value transformations, where samples whose feature vectors lay outside the convex closure of the training set’s feature vectors would be given very inaccurate or even impossible value predictions, here these samples would still show fairly good accuracy. This strengthens the notion that this value transformation properly handles the CPU event frequency as a basis for predicting the speedup.

4.1.5 Target 1 Results Overview

Of the four value transformations tested, the logarithmic value provided accurate predictions, surpassing the results of the OpenMP experiments[BA14] in estimating which samples will yield speedup over and under certain thresholds. The following figure shows the number of samples (out of 45) that were accurately predicted over or under each threshold. For example, if the application’s real speedup was 0.15 and it was predicted as 0.25; it would be counted towards the 0.1 column, as both values are greater than 0.1; and towards the 0.5, 1, 2, 5, and 10 columns, since both values are lesser than these thresholds. It would not be counted towards the 0.2 column, since one value is under the threshold and the other is over it.
The other value transformations yielded lesser accuracy, and not exceeding the OpenMP results.
4.2 Target 2

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Core i7-7700HQ (2.8GHz, 4 physical cores, 8 virtual cores)</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA Quadro GP100</td>
</tr>
<tr>
<td>RAM</td>
<td>16GB 2133Mhz</td>
</tr>
<tr>
<td># Counters</td>
<td>1774 total, 215 non-zero</td>
</tr>
</tbody>
</table>

This target uses the same GPU as the first one, but has a significantly weaker and less-advanced CPU. While the CPU is a higher-end series for its time (reporting more performance counters), after culling, it had fewer counters than the first target. Still, the powerful GPU allowed the target to yield speedups in the range of 1/3000x to over 1/400x.

To avoid bias or errors in re-measurement, the different approaches below used the same input data except for the functions applied to calculate the label values.

4.2.1 Target 2 Results Overview

Results for this target followed the same behavior as the first, though overall it had lower accuracy.

4.2.2 Target 2 Detailed Results

As with the first target, all negative value predictions in the linear and quadratic value transformations were made for samples which had many features outside the range of the training set.
Figure 4.7: Target 2 - linear value transformation

Figure 4.8: Target 2 - reciprocal value transformation

Figure 4.9: Target 2 - quadratic value transformation
Figure 4.10: Target 2 - logarithmic value transformation
4.3 Target 3

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Core i7-7700HQ (2.8GHz, 4 physical cores, 8 virtual cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>NVIDIA Quadro GP100</td>
</tr>
<tr>
<td>RAM</td>
<td>16GB 2133Mhz</td>
</tr>
<tr>
<td># Counters</td>
<td>1775 total, 204 non-zero</td>
</tr>
<tr>
<td>Notes</td>
<td>Headless configuration (no X server running)</td>
</tr>
</tbody>
</table>

The hardware configuration of this target is identical to the previous one. However, this target was run without an X server (i.e. no GUI), to test whether this would affect GPU load or CPU load and make any difference in the results. The results proved this to be the case, as the highest GPU speedup remained at over 400x, but the lowest improved to around 1/1200x. However, the same outdated CPU still meant a lower number of CPU counters, providing less training data.

To avoid bias or errors in re-measurement, the different approaches below used the same input data except for the functions applied to calculate the label values.

4.3.1 Target 3 Results Overview

This target showed accuracy closer to the first target, but had many samples close to the 1x speedup threshold mispredicted to the other side of the threshold.

Figure 4.11: Target 3 - threshold tests

4.3.2 Target 3 Detailed Results

Similar to the previous targets, checking the results against the convex closure of the training set’s feature vector showed that the negative value predictions all came from samples outside that range. Examining the logarithmic value transformation’s plot reveals that while quite a few samples were incorrectly predicted over or under the 1x speedup threshold (appearing as dots in the bottom-right and top-left quadrant of that graph), these samples were not mis-predicted far from the threshold, and are mostly still close to 1x.
Figure 4.12: Target 3 - linear value transformation

Figure 4.13: Target 3 - reciprocal value transformation

Figure 4.14: Target 3 - quadratic value transformation
Figure 4.15: Target 3 - logarithmic value transformation
### 4.4 Target 4

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Core i7-6800K (3.40GHz, 6 physical cores, 12 virtual)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Geforce GTX 960 (SM206)</td>
</tr>
<tr>
<td>RAM</td>
<td>32GB 2133Mhz</td>
</tr>
<tr>
<td># Counters</td>
<td>1522 total, 216 non-zero</td>
</tr>
</tbody>
</table>

This target, using a older GPU, had a much tighter range of actual speedup values.

#### 4.4.1 Target 4 Results Overview

It seems that the smaller variance in speedup values considerably improved the accuracy of the predictions. It is notable that while the logarithmic value transformation prediction improved with this target; the other value transformations did not improve significantly; further cementing them as incorrect. As with the previous targets, the negative value predictions all came from feature vectors outside the range of the training set’s convex.

As surmised in the analysis for Target 1’s reciprocal value transformation, the tighter range of speedup values meant that the out-of-convex predictions had lower coefficients pulling them upwards, and this target is the first to show negative predictions for the reciprocal value transformation.

![Figure 4.16: Target 4 - threshold tests](image-url)
4.4.2 Target 4 Detailed Results

Figure 4.17: Target 4 - linear value transformation

Figure 4.18: Target 4 - reciprocal value transformation

Figure 4.19: Target 4 - quadratic value transformation
Figure 4.20: Target 4 - logarithmic value transformation
4.5 Target 5

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Core i7-6800K (3.4GHz, 6 physical cores, 12 virtual cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>NVIDIA GeForce GTX 1080</td>
</tr>
<tr>
<td>RAM</td>
<td>32GB 2133Mhz</td>
</tr>
<tr>
<td># Counters</td>
<td>1491 total, 212 non-zero</td>
</tr>
</tbody>
</table>

This target is the same computer as Target 4, replacing only the GPU. Some of the GPU counters were excluded from the measurement as an in-depth diagnostic revealed they are dummy counters only reporting 0 values.

4.5.1 Target 5 Results Overview

The newer GPU on this target again provided a wider range of speedup values. In turn, this resulted again in slightly less accurate predictions - but still similar to those the first target.

![Figure 4.21: Target 5 - threshold tests](image)

4.5.2 Target 5 Detailed Results

Like in all the previous targets, all impossible value predictions in the linear, reciprocal and quadratic value transformations were given to samples where the feature vector was outside the convex of the training set’s features. As with target 4, the tighter range of actual values caused the reciprocal value transformation to show some negative-valued predictions. The convex testing for this target showed that several of the samples had an exceptionally large number of features outside the training set’s range (over 25 of 212, up to 48 features for one of the samples) - the very distant predictions (under -75 in the linear and reciprocal plots, under -20 in the quadratic plot, and under -5 in the logarithmic plot) all came from these samples.
Figure 4.22: Target 5 - linear value transformation

Figure 4.23: Target 5 - reciprocal value transformation

Figure 4.24: Target 5 - quadratic value transformation
Figure 4.25: Target 5 - logarithmic value transformation
4.6 Target 6

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Core i7-6800K (3.4GHz, 6 physical cores, 12 virtual cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>NVIDIA GeForce RTX 2080</td>
</tr>
<tr>
<td>RAM</td>
<td>32GB 2133Mhz</td>
</tr>
<tr>
<td># Counters</td>
<td>1491 total, 212 non-zero</td>
</tr>
</tbody>
</table>

This target is the same computer as Targets 4 and 5, only with the GPU replaced. The high-end, latest-generation GPU was not fully utilized by the OpenCL applications, since OpenCL does not give access to its unique ray-tracing and tensor cores.

4.6.1 Target 6 Results Overview

Since OpenCL could not make use of the unique parts of this GPU’s architecture, the actual values for it preformed similar to a middle-range GPU, as many of its resources lay idle. This, in turn, gave more accurate results for the logarithmic value transformation (like target 4). The smaller range also made it so that the negative-valued predictions were not as extreme as in target 5; and that once more, no negative predictions were made for the reciprocal value transformation.

Figure 4.26: Target 6 - threshold tests
4.6.2 Target 6 Detailed Results

Figure 4.27: Target 6 - linear value transformation

Figure 4.28: Target 6 - reciprocal value transformation

Figure 4.29: Target 6 - quadratic value transformation
Figure 4.30: Target 6 - logarithmic value transformation
4.7 Target 7

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Core i7-6800K (3.4GHz, 6 physical cores, 12 virtual cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>NVIDIA GeForce GTX 1660Ti</td>
</tr>
<tr>
<td>RAM</td>
<td>32GB 2133Mhz</td>
</tr>
<tr>
<td># Counters</td>
<td>1491 total, 212 non-zero</td>
</tr>
</tbody>
</table>

This target is the same computer as Targets 4 through 6, replacing only the GPU. The GPU is modern but mid-range in its computation power, and yielded a fairly tight speedup results range.

4.7.1 Target 7 Results Overview

The modern GPU used here provided higher speedup values for some of the samples with lower, but positive speedup values on other GPUs. However, being middle-range in its power, the highest speedup values were also pulled back compared to other targets. This caused the accuracy (by threshold) for the logarithmic value transformation to be pulled upwards for the middle thresholds, while overall lowering the accuracy of the other value transformations.

![Figure 4.31: Target 7 - threshold tests](image)

4.7.2 Target 7 Detailed Results

Similar to target 5, many of the feature vectors for this target were far outside the convex of the training set, which resulted in the same type of extreme mispredictions: low negative values for the linear, reciprocal, and quadratic value transformations. The two samples that had the most out-of-range features (39 and 40 of 212), also showed logarithmic value transformation predictions of under -5. However, both these samples had actual values under -1 (0.1x), so they still passed the threshold accuracy tests.
Figure 4.32: Target 7 - linear value transformation

Figure 4.33: Target 7 - reciprocal value transformation

Figure 4.34: Target 7 - quadratic value transformation
Figure 4.35: Target 7 - logarithmic value transformation
4.8 Result Summary and Discussion

Overall, the results seem to indicate that, as hypothesized, values of CPU counters on the sequential algorithm can be used to predict the speedup that can be gained from converting to a massively-parallel algorithm. While the number of sample applications used was not very large, estimating using the logarithmic scale provided results that were typically within one order-of-magnitude of the actual value. Presumably, the larger the training set will be, the better these results would prove.

However, the results of individual samples varied greatly based on the hardware configuration of the target; thus the estimator should be trained on the same target where the estimations will be made for the best accuracy. Still, the collection and training can be fairly easily automated to collect and generate an estimator - this only leaves a balance between higher accuracy of the estimates (more samples used for training) and shorter estimator creation time (fewer samples used for training).

It is notable that the main difference in accuracy between targets seems to be related to the CPU model. This might indicate that certain CPUs provide more performance counters that are relevant to these estimates (or fewer irrelevant ones, avoiding overfitting).

The fact that the linear, reciprocal and quadratic value transformations gave less accurate estimates reinforces the notion that the logarithmic transformed values are well-correlated. This is because the lower accuracy for these value transformations eliminates the possibility of samples being too similar and creating perfect matches for the linear classifier - if that were the case, all transformations would present good correlation. CPU event frequency having a linear correlation to the logarithm of the speedup factor would indicate that there are multiple event types that can predict a certain speedup factor, and that the effect of these events is multiplicative between them.

This notion was further solidified by the fact that the value transformations other than the logarithmic would predict impossible values: and that these impossible predictions were, without exception, made on samples where the feature set lay outside the convex closure of the training set’s feature vectors. Despite using the same feature vectors, the logarithmic value transformation would still, for the most part, make fairly accurate predictions. It was only tripped up by samples that had an extremely high number of these out-of-range features, and even then, its misprediction was usually by taking the actual value and exaggerating it: predicting a large negative value for a sample with a small negative actual value, or predicting a large positive value for a small positive actual value.
4.8.1 Meta-Analysis (Feature Set Reduction)

Having established the logarithmic value transformation as providing the best results, further exploration was performed to attempt to reduce the training time; as well as attempting to detect (and reduce) overfitting. This was performed only on Target 1. To this extent, the following experiment was performed:

First, a linear regressor was trained on the training set using the full suite of features (excluding, as always, any culled features). Then, the linear classifier outputted by the regressor was queried to provide the coefficients it gave each feature. These were sorted in descending absolute value order, then a set of new linear regressors was trained, each removing an additional half decile of the lowest-coefficiented features. That is to say, the first regressor was trained with the 95% of the features whose coefficients (in the 100% training) were largest. The second was trained with the highest 90% of the features. The third with 85%, and so on until the last regressor trained with 5% of the features.

The linear classifier generated by these regressors were then compared to the original classifier using the same accuracy measurement used previously.

Figure 4.36: Target 1 - counter percentile meta-analysis

The results showed that while the higher proportions did not suffer much from the reduction of the feature count, they also did not improve the accuracy, indicating
that the original model did not suffer much from overfitting. The middle and lower proportions showed lower accuracy values, indicating that many of the features are assisting in the prediction.
Chapter 5

Future work

5.1 Further Results

This research also aims to describe and allow expansion of the method itself, enabling future research to add new targets and test whether they provide similar reliability in prediction. As with all machine learning-based estimators, the addition of more CPU-GPU sample pairs is expected to increase the accuracy of the predictions.

5.2 Improvements to the Method

For the sake of simplicity, this research aimed to take the simplest path in the described experiment, as no previous work was done in an attempt to make estimations such as these. However, it is possible that with a deeper understanding of the mechanisms used, better results can be obtained without changing the sample or target sets. Such changes might include:

- Using machine learning models that are more advanced than a linear classifier, such as neural networks or other classifier types.
- Using different machine learning frameworks or improving the usage of TensorFlow for faster training and estimation or more accurate results.
- Culling certain inputs (CPU counters) as irrelevant or misleading - this experiment only culled CPU counters that provided the same value for all samples (typically 0).
- Pre-processing certain inputs (CPU counters) by applying functions to them - this experiment only normalized ranges (by dividing each counter by its maximum value).
- Better stabilizing the run environment for measurements.
- Changing the operating system used for measurement.
5.3 Other Applications of the Method

Other than repeating the experiment exactly as described in this paper on more targets, more samples and more hardware combinations, several more research avenues arise if the results of the experiment are considered:

5.3.1 Other Frameworks

The experiment can be repeated for other acceleration frameworks, such as CUDA or OpenACC. Many of these frameworks provide automated conversion tools that may facilitate the creation of CPU-GPU sample pairs. Since this experiment attempted to keep the algorithm for each sample application as close as possible between the two versions, it is expected that these other acceleration frameworks would provide a similar correlation and estimators could be generated for them as well.

5.3.2 Results on Optimized Samples

This similarity in algorithms between the CPU and GPU versions, however, is not the case for many real-world applications. While such an estimator can prove very useful in the decision whether to convert a naive or proof-of-concept algorithm to run in a massively parallel environment - existing CPU-based algorithms will typically be optimized for their running environment. Therefore, there is merit in conducting a similar experiment, but having the sample pairs be highly optimized for CPU and GPU execution, respectively. If such an experiment still provides good estimations, the resulting estimation method can be used for existing software solutions more reliably.

5.3.3 Application to GPU Optimizations

In addition, given that the estimation given is for a raw conversion of the input CPU algorithm; GPU algorithms are usually very varied in what optimizations they require. This could include memory transfer optimization, memory access pattern optimization (for locality or for latency hiding), register pressure reduction, memory pressure reduction for each of the GPU’s various memory levels, and many other optimization methods. Once a GPU algorithm is created, GPU performance counters may be used to estimate which of these factors is the bounding factor or bottleneck. This factor is also very liable to change between different hardware configurations (e.g. when running the same code, a GPU with less VRAM might have shared memory pressure as the bounding factor while one with more VRAM might be bound on host transfer speed or register pressure).

Most hardware vendors provide profiling tools and performance counters that aid in identifying these bottlenecks. Thus, instead of only estimating the speedup
provided by porting to a massively-parallel environment from CPU counters - by providing different value definitions for the machine learning model method can be used to estimate either (a) the value of certain indicative GPU performance counters or, taking it further, (b) classification of the algorithm into groups based on the optimizations that will be required once ported to GPU. This could provide more information prior to converting the algorithm. If proven to be successful, this could even result in the ability to estimate the theoretical maximum speedup gainable from moving to a massively parallel algorithm (and see also “Results on optimized samples” above).

5.4 Meta-Analysis

There are also ways to analyze the information collected for this research along different axes than the ones described. These were not explored, mainly due to resource limitations, namely a limited number of available targets.

5.4.1 Sample Meta-Analysis

This research did not examine the specific samples used. It is possible to observe the specific samples across multiple targets, and to try to deduce whether these samples are more or less suited to this estimation method. It is possible that certain samples are more predictable or less predictable than others, and using results from several targets it might be possible to predict behavior of the same sample on other targets. However, such analysis requires a much larger target set for comparison, and was thus deemed not to fall within the scope of this research.

5.4.2 Sample Set Composition Meta-Analysis

Given enough results on varied targets, it could be possible to also judge the necessity of certain samples’ presence in the training set for the machine learning operation, and to research into whether and why some samples are required in order to provide good estimates for others. This can be done by reducing the training set beyond what was done in the base experiment (which only removed the sample being estimated from the training set) and comparing the results for a single sample with different training sets. If the results’ accuracy varies greatly, it might indicate that the training set can be trimmed for faster estimator construction (and training data collection). If it does not, it would indicate that increasing the size of the training set would be the best method of improving the estimator’s accuracy.
Chapter 6

Conclusion

During this research, we attempted to prove it is possible to predict an algorithm’s performance on a massively parallel architecture by measuring details about its performance on a sequential architecture. To do so, we have chosen pairs consisting of a CPU C++ implementation and a GPU OpenCL implementation of the same algorithm. We have performed measurements of various CPU performance counters for each CPU implementation, as well as measure the factor of run time (i.e. speedup or slowdown multiplier) between the two implementations, to be used as “truth values”. These data were then fed into a basic machine learning algorithm to produce a linear classifier, which is intended to predict the speedup of a sample it was not trained on - from just the CPU measurements.

Trying various approaches to the problem, we discovered that taking the logarithm of the speedup as the “truth value”, the classifiers were able to predict the speedup factor well. Using a threshold test used by other works of this nature showed that over 80% (and typically more) of the programs predicted were correctly placed in relation to the threshold. Despite the fact that this experiment was made using simpler methods, not using source code instrumentation and parallelized CPU implementations, its accuracy was similar to experiments previously performed using those methods in the measurement.

Taking other “truth values” proved less accurate - but this also serves to validate that the sample set taken was varied enough, since otherwise all truth value sets would provide good results. These results re-emerged in the same manner accross multiple testing targets, varying the CPU and GPUs in both power and microarchitecture generations. Doing so has shown that the more advanced the CPU was (and therefore, the more counters were available), the more accurate the predictions given. This further solidifies the notion that the counters are proving useful in predicting the GPU results. This was further reinforced by the fact that the other value transformation exhibited impossible value predictions in direct correlation with the number of features that were not within the range of the training set. The logarithmic value transformation was able to handle the same out-of-range inputs and still
output accurate predictions.

This research has shown that this avenue of deduction for algorithms has merit, and that it could be possible to determine the benefit of porting an algorithm to a massively parallel environment before even writing a single line of code specific to that environment. As this method could easily be expanded up and automated, it might serve as a useful tool in decision-making related to the choice of target platform by developers.

However, we have also discussed further avenues of research to build upon these results, as it is clear that they can be greatly improved or made more efficient. It is clear that such future research could even further enhance the usefulness of these methods in decision-making for developers.
Bibliography


The results of the overall evaluation of the new method showed that the CPU and GPU in the implementation of the machine learning algorithms, measurements of the performance index were obtained - on average, the predictions of the logarithm of the machine learning performance in the training set were accurate. The error between the predicted and the real values was found to be between 3% and 5% of the range of the average errors of all tests. On the other hand, the logarithm data model was effective in handling values at the top. In addition, we tested a new regression model, and we found that the new model was better than the traditional models for the new data set.
תקציר

"חוקمور", לפיו יחידות העיבוד של מחשבים הופכות את כוח החישוב שלהן כל שנה, מתקרבים לפני שהיכן הפיזיקלית של מעבדה. כדילשמור על קצב הגדלת ייצור המודולים והשכירות, רכישת מעבדה מסדרה-פרחית (massively-parallel) שייכת למערכות מחשבים המתאימות לפורמטים ותוחלת שירות המערבי בתקופה של שנה. הmares החישובים ומקבלי השכר מתאימים לשもらった מעבדים שהופכים לPorno רחביים ומחונים יותר. מעבדים אלה מתומכים בחישובים כמו מעבדים שינויזה מובילה inclination. השיפורים שבוצעו למעבדים המסיביים והחישובים שמ אינם מתבצעים על-ידי מעבדים מוכרים לstasy, והפיתוח עבורם מתבצע לרוב על ידו של מומחה לתחום.

ניצול יעיל של מעבדים אלה הדור הרışı וה xxxx בשתייה מחקרים מחשבים וחישובים בטכנולוגיה מחcciones מחשבים ובאולוריים מחקרים באלגוריתמים войны невית של מעבדים."dı ל ula מחשבים על פי ערכתי המ区管委会 מחשבת במעבדה משנה את מעבדה.

CPU- מעבד הזהboro משורה לשיפור בезультטים שהועברה המשימה, מעבד מנוון הזמנה, והתREFIXught על אולוריים מחקרים מחağını מחקרים מחשבים ובאולוריים מחקרים באלגוריתמים войныневית של מעבדים."district מחשבים על פי ערכתי המведен מחשבת במעבדה משנה את מעבדה.

לגרגון רוש מbringing מחשבים על פי ערכתי המведен מחשבת במעבדה משנה את מעבדה.

Technion - Computer Science Department - M.Sc. Thesis MSC-2020-04 - 2020
המחקר玻צע בجامיות של פרופסור אבי מנדלסון. מקוליעו למדריך המחבר.

תודה

ברצוני לתרום

לאנשה של פרופ', אבי מנדלסון, על ה兴旺ה והممתקים והאינטנסיביים.

למשתתפים על העזרה והעניקים שדיבעו במחקרים.

ל嘴里 בן-เทศל על עזרתו בעריכה למשנה.
ניבי ביתיעסום של תוכניות על מעבדים מקבליים וטרוגונימיים

והבר על מחקר

לשם مليולי תקן שלتحرורכתלקבלת ההוזור
אזורים למполнение במרץ המועדים

אוריש שומורני

הותקן לתוך עמידת — מכון מדעי ולהיושב
ת iar ומשתף 2019
ניבי ביצועים של תוכنيיה על מעבדים מקבליים והטרוגניים

אוריש שומורי

Technion - Computer Science Department - M.Sc. Thesis MSC-2020-04 - 2020