Feasibility Study of Automatic Profile-Based CUDA Optimization

Steve Chapel, James "Augie" Hill

December 19, 2011
Abstract

In recent years, the GPGPU has been adopted as a hardware platform of choice for extremely large-scale compute-intensive scientific studies. These studies may run a small piece of code on enormous data sets for weeks, so achieving even small optimization improvements may save much time and money. Writing and compiling programs for the new generation of highly parallel compute architectures such as GPUs involves the balance of trade-offs to achieve optimal run-time. After fixing the program algorithm constant, program optimizations are generally searched for manually by programmers with expertise of both the GPU architecture and the program in question. It has been shown that the results of these manual optimizations are in general not the global optimum. In this paper, we seek to understand the trade-offs of programs run on nVidia GPUs and test methods of automatically searching the program configuration space for the global optimum. We test our methods on the Parboil CUDA benchmark suite and find that intelligent randomized configuration search performs well for finding near-optimal configurations quickly. The domain-agnostic search methods employed can be improved in future work by including domain-specific heuristics to select configurations to test.

1 Introduction

1.1 Motivation

Computation-based science is becoming more prevalent in many fields such as genomics and materials science, and the GPGPU hardware architecture is being used as a cost-effective platform for addressing these needs. Programmers are finding that optimizing programs for GPGPU architectures involves making trade-offs, for example between memory usage and thread count, that can affect the run time performance of programs in unintuitive ways. This research is motivated primarily by the expected non-linear and difficult to optimize program configuration space. An automated way of intelligently exploring the space of configurations would free programmers from the task of trying these configurations manually.

1.2 Structure

In this section, we introduced and motivated the problem we are attempting to address. In the next section, we describe the nVidia GPU architecture in detail. Neither of the authors of this paper had had any previous experience with GPUs, so this background research into the compute model was a crucial step to moving forward. We then describe our approach to the problem in the third section and the evaluation of our approach in the fourth section. Finally, we discuss our approach and findings in the fifth section.

2 nVidia GPU Hardware

For the sake of brevity, we do not disclose to the reader the nVidia GPU computational model, and note that such information is available in several other resources.

Every GPU manufactured by nVidia is slightly different, but each is based on one of the three generations of general unified architecture. While researching for this project, it was a difficult task to find information describing in detail the hardware upon which we would be working.

Our evaluations were performed using the University of Michigan Center for Advanced Computing (CAC) resources, for which the system of GPUs available on the CAC compute cluster is the nVidia Tesla s1070. The s1070 contains four T10 GPUs, which are based on the second generation of general unified architecture, named GT200. It was not nVidia does not make readily available any detailed specifications for the GT200. We were lucky to find one paper describing an in-depth micro-benchmarking study of the GT200 general unified architecture in addition to the information available by direct device query.

2.1 Device Query

The following are the important data made available via direct query of the GPU’s resources.
As will be made clear in this section, not all information was made available by device query. The remaining information was determined from [4] and [5].

### 2.2 Imposed Limitations

There are many program limitations imposed by the hardware which must be considered by the programmer. This section describes and discusses each of these limitations, notes the specific limiting values on the GT200 general unified architecture and the T10 GPU, and describes how the CUDA programming model can affect the resources used by the program.

In addition to these explicit limitations, there are implicit conditional limitations that are more difficult to express because they are a result of the values of other parameters of the program. We list the explicit limitations and describe the conditional limitations in the following sections.

In general, we found there to be an imperfect mapping from the CUDA program onto the GPU architecture so that the resource usage and computational flow cannot be known by examining the program.

#### 2.2.1 Threads

One kernel execution runs on a single thread. Threads are grouped into a warp. Warps are grouped into a block. Blocks are grouped into a grid. There is one grid per invocation (C call) of a kernel (C method). Each invocation specifies the dimensions of the grid and the dimensions of a block, for which there are limitations. There is no limitation on the number of invocations of a kernel. The following table describes the T10 GPU-specific thread limitations.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming Multiprocessors (SM)</td>
<td>30</td>
</tr>
<tr>
<td>Cores Per SM</td>
<td>8 (240 total)</td>
</tr>
<tr>
<td>Global Memory</td>
<td>4 GB</td>
</tr>
<tr>
<td>Constant Memory</td>
<td>64 KB</td>
</tr>
<tr>
<td>Shared Memory per Block</td>
<td>16 KB</td>
</tr>
<tr>
<td>Registers Per Block</td>
<td>16,384</td>
</tr>
<tr>
<td>Max. Threads per Block</td>
<td>512</td>
</tr>
<tr>
<td>Max. Block Dimensions</td>
<td>$512 \times 512 \times 64$</td>
</tr>
<tr>
<td>Max. Grid Dimensions</td>
<td>$65535 \times 65535 \times 1$</td>
</tr>
</tbody>
</table>

#### 2.2.2 Registers

Each streaming multiprocessor has a register file. On the T10 processor, each block has a maximum of 16,384 allocatable registers, while each thread may use at most 128 registers. This implies that if each thread uses the maximum number of registers, the maximum number of threads in a block is limited to 128 - less than the explicit limit of 512 threads per block. The following table describes the T10 GPU-specific register limitations.

<table>
<thead>
<tr>
<th>Resource</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads Per Warp</td>
<td>32</td>
</tr>
<tr>
<td>Threads Per Block</td>
<td>[1, 512]</td>
</tr>
<tr>
<td>Block Dimensions</td>
<td>$[1, 512] \times [1, 512] \times [1, 64]$</td>
</tr>
<tr>
<td>Grid Dimensions</td>
<td>$[1, 65535] \times [1, 65535] \times 1$</td>
</tr>
</tbody>
</table>

We suspect from our research that these limitations refer to the number of registers used at some instant. As the maximum number of allocatable registers is reached, registers are spilled to global memory. This could be a significant source of time lag, as the access of global memory is far slower than register memory.
Registers are extremely fast memory, so utilizing them efficiently could be useful in reducing the runtime of a program. The programmer can generally manage register usage with the number of C variables used within a kernel, though it is difficult to forecast the exact number of registers that will be live at any one time due to instruction expansion.

### 2.2.3 Instructions

There is a hard limit imposed on the number of PTX instructions in a kernel. This number can be checked by compiling to PTX and simply counting the number of instructions for the kernel function. On the GT200 architecture, there are a maximum of 2M PTX instructions per kernel.

The number of instructions can clearly be directly modified by decreasing the number of C instructions in the kernel. Some C instructions may also produce fewer PTX instructions compared to other C instructions while producing the same result.

### 2.2.4 Shared Memory

Memory available to all threads within a block, for which the hardware mapping is a thread processing unit (a set of 8 streaming multiprocessors).

On the T10 processor, this limit on shared memory is listed as 16 KB per block. The programmer may specify particular variables as residing in shared memory using the "__shared__" keyword. It is relatively simple to calculate the amount of shared memory that will be used in a block. If too much shared memory is being used, the variable or perhaps some portion of a data structure could possibly overflow to global memory.

### 2.2.5 Constant Memory

Globally (within the GPU) accessible read-only memory written from the host CPU. Additionally, each streaming multiprocessor has local to it a cache for holding some constant memory. [5] found there to be three levels of constant cache, making constant memory very efficient. On the T10 processor, there is a limit of 64 KB of global constant memory and an 8 KB cache per streaming multiprocessor. The programmer may specify particular variables as residing in constant memory using the "__constant__" keyword.

### 2.2.6 Texture Memory

Not well understood. The amount remains unknown, but the size of the cache local to the streaming multiprocessor is said to be 6-8 KB.

### 2.2.7 Global Memory

Memory accessible by all streaming multiprocessors on the GPU. There is more available, but it tends to be slow relative to all other types of memory. On the T10 processor, there are 4 GB of global memory. There is no global memory cache on this processor.

### 2.2.8 Latency & Bandwidth

The communication lines between components also limit the speed of program execution. Latency describes the time it takes to get one unit of requested data from memory, while bandwidth describes the maximum amount of data that can be transferred between components in some amount of time. This seems to be the most difficult hardware limitation for a programmer to grasp, and is probably the limiting factor that is most likely to yield positive results from intelligent randomized configuration search.
3  Approach

Compared to the classical computational model, the limitations imposed by the nVidia model are numerous to the point of being impossible for a programmer to keep track of. We aim to demonstrate that an intelligent profiling-based search of a program’s compile-time configuration space results in a compilation configuration that is negligibly close to the optimal configuration ($\epsilon$-optimal) in a fraction of the time it would take to sweep the entire configuration space. We first define the search space. We then devise constraints that limit the search space. Once we have a space of configurations to be tested, we sweep the entire configuration space, then evaluate various intelligent Monte Carlo methods of searching that space.

Of course, in practice, our approach would perform an on-line search, generating and testing configurations during the search, rather than performing a full sweep before the search begins. Our research investigates different ways this search can be guided so that a near-optimal program can be found quickly.

3.1  Use Case

We have a particular use-case in mind throughout this study. A developer would run the automatic optimization compiler on the hardware for which she is optimizing. She would provide the CUDA program, a set of sample inputs and corresponding outputs (assuming determinism), and a time budget. Within the allotted time, the compiler will search the optimization space, resulting in the binary of the $\epsilon$-optimal compilation configuration. Obviously, the more time budgeted, the smaller $\epsilon$ will be.

3.2  Evaluation Programs

We used two CUDA programs from the Parboil benchmark suite: Matrix Multiplication (MM), and Two Point Angular Correlation Function (TPACF). Parboil provides detailed timing information and verifies the output produced by the compiled program. All programs that returned incorrect output were considered to have infinite run-time. In our graphs, we considered programs that produced incorrect output to have a long run time, because one value of infinite run time during one run of thousands caused the average run time to be infinite.

3.3  Raw Configuration Space

In this section, we list and discuss the configuration parameters and the base range of each parameter independent of the other parameters. We will consider the relationships between parameters in the following section.

3.3.1  Compiler Options

Although most discussions of optimizing CUDA programs focus on source code changes, compiler switches can also be used to provide improved optimization. Many of these compiler switches also involve tradeoffs. For example, limiting the number of registers a function uses can make each thread run more slowly, but doing so may allow more threads to run concurrently and increase overall throughput. Additionally, high levels of optimization can create code that runs faster but contains more instructions than the original code, again limiting the number of threads that can execute concurrently. We also allow the programmer to specify which compiler switches should be involved in the search.

The Parboil suite of benchmarks uses the nvcc compiler, which includes the following compile flags that may involve tradeoffs that can affect GPU performance:

- **--optimize level** Determines the optimization level (0 to 3)

- **--maxrregcount registers** Determines the maximum number of registers per thread

The switches that are passed to the compiler are passed down to the ptxas assembler and to the linker, according to the nvcc manual. [1]
We first implemented a search algorithm that does a complete sweep of the search space by compiling and running every possible variation of the program. We used the results from this complete sweep to evaluate different local search strategies. We can compute how many compilations and runs would be needed to find the optimal configuration using different search techniques.

We were able to do this with Parboil because the benchmarks include routines that compute and report the total amount of time taken by the CPU and the GPU. In order to perform our optimization on other programs, we would need to be able to find a performance metric that our system could measure. One possibility would be to use the `time` command to determine the wall clock running time of a program. Note that we would not be able to use the CPU time, because with CUDA program we typically want to reduce the GPU processing time.

### 3.3.2 Thread Dimensionality

There are many different types of changes to a source code program that a programmer may want to consider when trying to optimize a GPU program. The programmer may want to vary the number of threads created, the amount of unrolling to apply to inner loops, which variables should be allocated to global memory or shared memory, and so on. To make our implementation simple and yet able to deal with all different types of modifications to source code, we have the programmer use the value of a `#define` value to control which variation is used. The programmer then informs our optimizer of the values that can be `#defined` and all the allowable values for each.

In the Parboil benchmark program `mm`, two `#defines` are used to determine the tiling of the matrix for assigning parts of the matrix multiplication to different threads:

```c
#define TILE_WIDTH 16
#define TILE_HEIGHT 8
```

We told our program to try different tiling factors of 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512 for each of these `#defines`.

In addition, we examined the effects of inlining the inner loop of the matrix multiplication. In the original code, this inner loop was:

```c
for (int kk = 0; kk < TILE_WIDTH; kk++)
    c[kk] += a * b_s[j][kk];
```

We added a new `#define` called `UNROLL` and used it to determine how many times to unroll the loop:

```c
for (int kk = 0; kk < TILE_WIDTH; kk += UNROLL) {
    c[kk] += a * b_s[j][kk];
#if UNROLL == 1
#elif UNROLL == 2
    c[kk+1] += a * b_s[j][kk+1];
#elif UNROLL == 4
    c[kk+1] += a * b_s[j][kk+1];
    c[kk+2] += a * b_s[j][kk+2];
    c[kk+3] += a * b_s[j][kk+3];
#endif
}
```

We examined the effects of unrolling the inner loop 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512 times.

### 3.3.3 Memory Management

Another parameter to vary when optimizing CUDA programs is which data is stored in registers, in shared memory, and in global memory. Additionally, prefetching can be used to help hide the latency of accessing global memory. We did not investigate these approaches to optimizing the programs in the Parboil benchmark suite, as they had already been ε-optimized to use the memory hierarchy efficiently and we needed to limit the space of configurations to search because of project time constraints.
3.4 Constraining the Configuration Space

In the MM benchmark, some combinations of configuration were invalid. For example, in the above code, the parameter UNROLL had to be constrained to be no greater than TILE_WIDTH. We handled these cases by rejecting configurations that we knew would lead to invalid results before we began searching through the space of configurations.

The programs in the Parboil benchmark also provide feedback as to the correctness of the output. We could have searched through the entire configuration space, and let the search automatically avoid areas that result in invalid output.

The configuration space for mm consisted of five parameters: TILE_WIDTH, TILE_HEIGHT, UNROLL, --optimize, and --maxrregcount. The first three parameters could take on nine different values: 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512. UNROLL was constrained to be less that TILE_WIDTH for program correctness, and the product of TILE_WIDTH and TILE_HEIGHT was constrained to be 512 or less because this product was used as the number of threads per block. The --optimize parameter could have the values 1, 2, and 3, and --maxrregcount had values that were all multiples of four between 4 and 128 inclusive. There were a total of 21220 runs, which took about four seconds each.

The configuration space for TPACF consisted of four parameters: BLOCK_SIZE, HISTS_PER_WARP, --optimize, --maxrregcount. BLOCK_SIZE took on the values of multiples of 32 from 32 to 512 inclusive. HISTS_PER_WARP took on the values of multiples of four from 4 to 128 inclusive. The values for --optimize and --maxrregcount were as in the mm benchmark. There were a total of 49152 runs, which took about two seconds each.

3.5 Search Methods

In this section, we consider a few methods for intelligently searching the space of compilation configurations. All methods are based on some randomized sampling, also known as Monte Carlo. The best method will consistently find the configuration with the lowest average computation time in the fewest number of tests.

One can think of the space to be searched as an \( n \)-dimensional terrain - a generalization of 3-dimensional or 2-dimensional terrains we humans can visualize. In Euclidean space, the \( x \) and \( y \) dimensions are the values of two configuration parameters being varied, while the \( z \) dimension is the profile time of the program under that configuration. We seek to find the coordinate \((x, y)\) that corresponds to the lowest possible value of \( z \).

We only consider Monte Carlo methods because previous research into optimizing CUDA applications has determined that the evaluation terrain is non-linear with local optima. Non-Monte Carlo methods typically start from some location and move only to neighboring configurations, so these methods are vulnerable to becoming stuck at local optima. By combing random sampling and local search, intelligent Monte Carlo methods are more likely to find the global optimum or at least a better local optimum.

3.5.1 Random Search

This method will be the control method. It merely consists of randomly sampling from the configuration space. All methods are based on some randomized sampling, also known as Monte Carlo. The best method will consistently find the configuration with the lowest average computation time in the fewest number of tests.

In this method, a configuration is chosen uniformly randomly from the space of configurations. The configuration is tested by being generated, compiled, and run, and the run time is computed. After each test, the configuration that produces correct output with the shortest run time is kept as the best configuration found so far.

3.5.2 Hill Climbing Search

This general method of intelligent search works best when there is some amount of smoothness to the terrain. In this method, we chose random configurations of the program until we find a configuration that runs with correct output. We then test the immediate neighbors of the current configuration (configurations that have one parameter changed to an “adjacent” value relative to the ordering determined by the programmer) and move to the best result as long as it is within some percentage of the run-time at the current configuration. This continues until no more improvements can be made, and a new starting location is chosen at random.
3.5.3 Genetic Search

Genetic search is modeled on evolution, where generations of configurations are tested and reconfigured based on their fitness, and probabilistic mutations of the new generation of configurations introduces additional randomness. We suspect that this method will be more efficient than hill climbing search because the connections between parameter values will be noticed and preserved in subsequent iterations.

4 Evaluation

4.1 Platform

We evaluated our methods on the University of Michigan Center for Advanced Computing (CAC) nyx cluster, which makes available several nVidia Tesla s1070 units. The host CPU architecture is unknown, but we can only assume it is consistent.

4.2 Ground Truth

We gathered ground truth data describing the outcome of every configuration in our selected configuration space. With this data, we were able to quickly test and improve our randomized search methods without having direct constant access to the GPU hardware.

One concern we had going into these experiments was that subsequent tests of a program would have drastically inconsistent run-times. To test this, we ran the TPACF configuration sweep three times in a row. We found that run-time has an average execution range (the difference between highest and lowest run-time) of 3% of the average run-time. This is not a surprising result, and means that any speed-ups we find must be greater than 3% to be significant. We extrapolate this result from the TPACF benchmark to the MM benchmark so as to save execution cycles on the compute cluster.

The following subsections describe the best and worst configurations in the optimization space. In general, we found that the configuration space contained many configurations that were $\epsilon$-close to an optimal configuration. As will be shown, the general topology of the optimization space has a significant effect on the how well the search techniques work. In the case of this research, the numerous $\epsilon$-optimal configurations means that even randomized search is a good enough technique.

4.2.1 MM Benchmark

The following table lists the 5 best configurations.

<table>
<thead>
<tr>
<th>Tile Height</th>
<th>Tile Width</th>
<th>Unroll</th>
<th>Optimize</th>
<th>Max Registers Per Thread</th>
<th>GPU/CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>16</td>
<td>16</td>
<td>2</td>
<td>76</td>
<td>0.054857</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
<td>4</td>
<td>2</td>
<td>56</td>
<td>0.055016</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>8</td>
<td>3</td>
<td>108</td>
<td>0.055076</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>4</td>
<td>1</td>
<td>116</td>
<td>0.055090</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>4</td>
<td>3</td>
<td>104</td>
<td>0.055097</td>
</tr>
</tbody>
</table>

These results show that the configuration run-time is primarily dependent on the tile width program parameter. Within that space, there appears to be an interesting trade-off between the tile height and the amount of unrolling done within the loop.

The following table lists the 5 worst configurations.
### 4.2.2 TPACF Benchmark

The following table lists the 5 best configurations.

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Histograms Per Warp</th>
<th>Optimize</th>
<th>Max Registers Per Thread</th>
<th>GPU/CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>16</td>
<td>3</td>
<td>68</td>
<td>1.198181</td>
</tr>
<tr>
<td>256</td>
<td>16</td>
<td>3</td>
<td>64</td>
<td>1.198312</td>
</tr>
<tr>
<td>256</td>
<td>16</td>
<td>3</td>
<td>36</td>
<td>1.198354</td>
</tr>
<tr>
<td>256</td>
<td>16</td>
<td>2</td>
<td>48</td>
<td>1.198432</td>
</tr>
<tr>
<td>256</td>
<td>16</td>
<td>3</td>
<td>28</td>
<td>1.198473</td>
</tr>
</tbody>
</table>

The following table lists the 5 worst configurations.

<table>
<thead>
<tr>
<th>Block Size</th>
<th>Histograms Per Warp</th>
<th>Optimize</th>
<th>Max Registers Per Thread</th>
<th>GPU/CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>4</td>
<td>3</td>
<td>8</td>
<td>4.755481</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>4.754652</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>4.754580</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>3</td>
<td>8</td>
<td>4.326010</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>1</td>
<td>8</td>
<td>4.325442</td>
</tr>
</tbody>
</table>

It is clear from these results that the TPACF benchmark run-time is primarily dependent upon the block size and histograms per warp. We were only able to optimize within that subspace of the global configuration space. Thus, it would have been fastest to simply ignore the flags while focusing on the program parameters, then search the space of compiler flags independently.
4.3 Randomized Search Methods

We evaluate our search methods primarily on how quickly they can find a near-optimal solution. The following set of figures shows the best (minimal) test found so far as the number of tests increases. The best test time found so far is on the y-axis, while the test number is on the x-axis. These results are the average of 2,000 runs of each algorithm.

While the above figures do put all three methods on the same graph, they do not give a sense of the relative percentage differences in run-time between methods. The following set of figures show the relative merits of hill climbing search (left) and generic search (right) versus random search on the MM benchmark (top) and the TPACF benchmark (bottom).
4.4 Manual vs. Automatic Profile-Based Optimization

The existing manually-optimized configurations for each of the benchmarks we worked with were already very good. For the MM benchmark, the manually-optimized configuration resulted in a 0.058122 second run-time. The randomized search algorithms were able to match this run-time in about 180 tests. As shown in the following graph, the genetic search algorithm found a configuration 5% better than the manually optimized configuration in about 2,000 tests. The global optimum of this benchmark is a 5.6% improvement on the manually optimized configuration.

For the TPACF benchmark, the manually-optimized configuration resulted in a 1.199242 second run-time. Unfortunately, this was already within 0.1% of the globally optimum configuration, so there was no significant optimization to be made from the manual optimization on this benchmark.

5 Conclusions

5.1 Discussion

We are not terribly concerned that there was little we could gain past the manual optimizations of the Parboil benchmark suite. The team that worked on those manual optimizations are among the best experts in the field, so the fact that we could attain any speed-up is an accomplishment.

Our results show that hill climbing search is about 10% better than randomized search in the near term (though not consistently). The results of the comparison between hill climbing and genetic search are inconclusive, as genetic search dominates hill climbing on the MM benchmark, but is dominated by hill climbing on the TPACF benchmark. Of course, all of the methods are equally good in the long term, as they continue searching with some randomness until all configurations have been examined. It is quite surprising how quickly the randomized search methods were able to find an $\epsilon$-optimal configuration. We attribute this result to the search space containing many good configurations.

5.2 Future Work

We believe we were limited by our self-imposed requirement to perform a complete sweep of our configuration space to obtain ground truth data. The intelligent search techniques we seek to employ on CUDA program optimization show their strengths in much larger search spaces. In the future, we would increase the number of dimensions to be searched beyond what would be possible to fully explore.
Our lack of experience with CUDA and GPU architectures in general may have shown in this study. Future work should focus on combing more domain knowledge into the intelligent search algorithms. It may be possible to exploit regularities in the search space and more domain knowledge of GPU architecture to find configurations that run faster with less search. One idea would be to feed the observed run-times into a classifier that could estimate the run-time of a configuration. We could then search configurations that look favorable.

References


