Automatic Static Analysis-Guided Optimization of CUDA Kernels

Mark Lou
Illinois Institute of Technology
Chicago, IL, USA
mlou@hawk.iit.edu

Stefan K Muller
Illinois Institute of Technology
Chicago, IL, USA
smuller2@iit.edu

Abstract

We propose a framework for using static resource analysis to guide the automatic optimization of general-purpose GPU (GPGPU) kernels written in CUDA, NVIDIA's framework for GPGPU programming. In our proposed framework, optimizations are applied to the kernel and candidate kernels are evaluated for performance by running a static analysis that predicts the execution cost of GPU kernels. The use of static analysis, in contrast to many existing frameworks for performance tuning GPU kernels, lends itself to high-level, hardware-independent optimizations that can be of particular benefit to novice programmers unfamiliar with CUDA's performance pitfalls. As a proof of concept, we have implemented two example optimizations and a simple search strategy in a tool called COpPER (CUDA Optimization through Programmatic Estimation of Resources), which makes use of a static resource analysis tool for CUDA from prior work. The prototype tool automatically improves the performance of sample kernels by 2–4% in initial experiments, and demonstrates the feasibility of using static analysis as part of automated performance tuning for GPU kernels.

CCS Concepts: • **Software and its engineering → Software performance**; *Parallel programming languages.*

Keywords: CUDA, optimization, static analysis, tuning

ACM Reference Format:
Mark Lou and Stefan K Muller. 2024. Automatic Static Analysis-Guided Optimization of CUDA Kernels. In *The 15th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM '24), March 3, 2024, Edinburgh, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3649169.3649249

1 Introduction

General-purpose programming on GPUs (GPGPU) is becoming increasingly common, driven by machine learning and data-intensive scientific applications. Running certain computations, or *kernels* on the GPU can drastically improve the performance of certain types of software. In order to make GPGPU programming more accessible, various frameworks for developing GPU kernels have arisen, CUDA, developed by NVIDIA, being one of the more popular. The idea behind CUDA is that programmers write applications in an extended dialect of C/C++, called CUDA C/C++. While the syntax and extrinsic semantics of CUDA C/C++ resemble the sequential language counterparts, the performance characteristics are drastically different due to the nature of GPU hardware. It is possible for a novice CUDA programmer, even one experienced in sequential C or C++ programming, to unknowingly introduce severe performance bottlenecks into kernels.

To make matters worse, there are few to no hard and fast rules for gaining good performance with a GPU kernel, and optimizations are often counterintuitive. As an example, one performance bottleneck in CUDA kernels is called *warp divergence*, which can result in code running more sequentially than expected (the exact nature of this bottleneck is not important for the moment and is described in more detail in Section 4.1). One optimization for reducing the impact of warp divergences, which we investigate further in this paper, is *branch distribution* [6]. In some programs, branch distribution actually *increases* the *number* of warp divergences. Because a warp divergence comes with a certain constant cost, [1] doing so might actually cause *worse* performance. However, in general, this additional overhead is (more than) offset by the increase in parallelism that is gained. Determining whether an optimization is likely to be beneficial or not is a difficult problem, even for experienced GPU programmers. Recent work [11] addressed this problem by providing a performance model for CUDA kernels, as well as a quantitative program logic usable for predicting the impact of several common performance bottlenecks on a CUDA kernel. This program logic has been implemented in a tool called RaCUDA. RaCUDA takes as input a CUDA kernel and a resource metric, which assigns abstract "costs" to various operations performed by a kernel. The tool statically predicts an upper bound on the "cost" of running the kernel, in the units given by the resource metric. The aim of RaCUDA is to help novice programmers to identify and correct performance bottlenecks in their programs.

In this work, we propose taking the goal of RaCUDA a step further to *automatically optimize* input kernels to remove or

PMAM '24, March 3, 2024, Edinburgh, United Kingdom

ACM ISBN 979-8-4007-0599-1/24/03
https://doi.org/10.1145/3649169.3649249

reduce the impact of the targeted performance bottlenecks. The essence of our proposed framework is to iteratively apply optimizations to an input kernel, exploring the space of possible optimizations using some search strategy, and checking for improvement by running RaCUDA. As a proof of concept of this framework, we have implemented a tool, COpPER (CUDA Optimization through Programmatic Estimation of Resources), that (like RaCUDA) takes as input an unoptimized kernel and a resource metric, applies optimizations in a guided manner and returns the best kernel according to the provided resource metric, as estimated by RaCUDA (this may be the original kernel if, as in the branch distribution example above, the attempted optimizations did not result in an improvement). COpPER does not execute the kernel—the optimizations performed are hardware-independent and running the tool does not require a GPU.

To demonstrate our approach, we have equipped COpPER with two optimizations it can perform on kernels: the branch distribution transformation described above and another optimization that copies frequently-used array segments into the GPU's fast *shared memory*. Our approach, however, can work with a wide variety of optimizations, search strategies, and resource metrics. The approach can be used with off-the-shelf optimizations that take CUDA kernels as input and output optimized kernels. However, our approach has the added benefit that optimizations can more tightly integrate with the static analysis in order to specialize to the particular kernel. For example, the optimization that moves arrays to shared memory requires computing upper and lower bounds on the array indices accessed to know what segment of the array to move. These bounds are computed using information already calculated by RaCUDA during an abstract interpretation pass, which are then passed through to the function that does the optimization.

At a high level, our work has a similar goal to many papers and systems on *auto-tuning* kernels to apply particular optimizations or select parameters to maximize the performance of a kernel (e.g. [3–5, 9, 10, 14]). The main difference is that these systems generally get feedback on possible improvements by actually running the kernel on a GPU to empirically determine the execution time. This is a good approach for tuning predictable kernels for the best possible performance on hardware, but there are several reasons to prefer static analysis over execution for some applications. First, the analysis of RaCUDA is hardware-independent and geared toward higher-level performance properties. This makes it of more use to programmers, especially novices, who are aiming for good, but perhaps not bleeding-edge, performance across a wide range of hardware. Second, the use of static analysis allows for information-sharing, such as the use of array bounds for copying arrays to shared memory, as described above. The same information could be observed by profiling a running kernel, but only for one particular run of a kernel. If the array indices accessed depend on the input data, for example, information obtained by profiling might result in unsound optimizations. This leads to a final benefit of static analysis: the quantitative program logic underlying RaCUDA comes with a proof guaranteeing a sound upper bound on execution cost, a desirable feature in real-time scenarios where worst-case performance is critical. Indeed, recent work [15] has succeeded in producing adversarial inputs to drastically increase the time and power consumption of neural networks. Even if programmers develop kernels that are robust to these inputs, it is important to know that automatic optimizations cannot introduce vulnerabilities to such attacks—this could not be guaranteed by an optimization framework that simply runs the code to test whether there is improvement on an average input.

On the other hand, the choice of whether to use actual GPU execution or static analysis (or some combination of the two) to assess the benefits of optimizations is largely orthogonal from many of the contributions made by the auto-tuning research, such as how to prune the search space or when to sample (whether "sampling" means executing a candidate kernel or statically analyzing it). Many of these advances can be integrated with our approach. Our purpose in this paper is to introduce one new axis in the design space of kernel optimization methods: the use of static analysis. In order to assess the feasibility of this approach, we introduce and evaluate it by itself rather than in combination with other cutting-edge advances; these combinations would be an excellent subject for future study.

The remainder of the paper proceeds as follows. In Section 2, we give some background information including a more thorough discussion of related work on tuning GPU kernels and more detail on the structure of RaCUDA which is necessary to understand the operation of COpPER. Section 3 gives a brief overview of our proposed framework. We then delve more deeply into the details of COpPER in Section 4, which details the resource metric used as the optimization function as well as other heuristics and assumptions used for our proof-of-concept implementation, and Section 5, which describes the two example optimizations we implemented. In Section 6, we evaluate COpPER on a selection of benchmarks drawn from various sources including prior work and public GitHub repositories. Although a thorough evaluation on real-world code is beyond the scope of this preliminary feasibility study, COpPER is able to apply the two implemented optimizations in several settings with modest but non-trivial (2–4%) benefit. Finally, we conclude with some observations about this study and directions for future work in this area.

2 Background and Related Work

2.1 Related Work

Auto-tuning GPU kernels. Given that the performance of GPGPU kernels (including those written in CUDA, but others as well) is so sensitive to small code changes, parameters,

and hardware details, it is not surprising that there is a great deal of literature on performance tuning of kernels, including doing so automatically. This process of *auto-tuning* is generally aimed at tuning kernels for particular hardware. Many auto-tuning frameworks and algorithms are even specialized to particular applications such as matrix multiplication [5, 9] and stencil computations [4, 10]. Schoonhoven et al. [14] provide a quantitative comparison of various optimization algorithms used for auto-tuning and, in the process, conduct a good survey of the auto-tuning literature.

Most auto-tuning algorithms operate based on a feedback loop of performing optimizations and then executing the new kernel on the GPU to test its performance, whereas we instead statically analyze kernels to predict their resource usage. Our work is therefore more geared toward general, hardware-independent optimizations. Because the aims are different, we will review some of the work most closely related to ours, and refer the interested reader to the literature reviews of these papers for more details.

As in our work, auto-tuning generally reduces to a guided search of a large space of optimizations, with some work aimed at how to best narrow down the search space. Garvey and Abdelrahman [4] use machine learning (ML) models to select optimizations, and other heuristics to guide the remainder of the search of the optimization space. OpenTuner [3] uses a combination of sampling and regression trees to explore the search space. Such techniques could also be applied to the idea of resource analysis-guided kernel optimization, whereas we have largely used a greedy approach.

There is also work on general optimizations for specific performance bottlenecks. Han and Abdelrahman [6] present program transformations for reducing the impact of warp divergence, including the branch distribution optimization we apply in our work. The same authors in later work [7] use ML models to decide whether an optimization on the use of memory is likely to be beneficial or not.

Quantitative Models of GPU Performance. Using static analysis to assess the performance bottlenecks of a kernel, or the performance benefits of an optimization, requires a model of the performance of GPU kernels. Such models are complex and difficult to come by because of the complexities of GPU hardware and the closed-source nature of many of the common architectures and APIs. The quantitative model behind RaCUDA, the static analysis tool on which our prototype is constructed, is based around predicting the impact of three major bottlenecks described in Section 4.1. However, RaCUDA itself does not provide a model for combining this information into an estimate of the running time (or even an abstract cost) of a kernel; that work is done by the *resource metric* that is a parameter to RaCUDA. Deriving a resource metric that can approximate execution time requires a model that translates software features such as those predicted by RaCUDA into execution times. The model of Braun et al. [2], which aims to predict the runtime performance of kernels based on such hardware-independent features, is a promising step in this direction. In a similar vein, GPURoofline [8] is a performance model for GPU kernels specifically intended to guide optimizations of GPGPU code.

2.2 Background: RaCUDA

RaCUDA [11] is a fully automated static resource analysis for CUDA kernels. It takes as input a CUDA program and a *resource metric*, which maps operations to costs in abstract units. For example, RaCUDA includes resource metrics for "warp divergences" which assigns a cost of one every time a warp divergence occurs, and a cost of zero to all other operations. It also includes a "steps" metric, which assigns a cost of one to most sequential operations (weighting memory operations based on expected latency), giving a high-level approximation of execution time.

RaCUDA consists of three main components. The *front end* parses the source code (using FrontC[1]) and converts it to a high-level imperative, resource-annotated intermediate representation (IMP) using the given resource metric to annotate operations with costs. The converted kernel then undergoes an *abstract interpretation*. Information gleaned about values during this analysis is used to estimate CUDA-specific costs (such as the range of memory addresses accessed by a read operation, which impacts the latency of the access) and other information necessary for the bound analysis. Finally, the *bound analysis* builds a set of constraints on the resource usage of program components, which is then solved by an off-the-shelf LP solver. The bound analysis component of RaCUDA is largely unchanged from that of Absynth [12], on which RaCUDA is based.

3 Overview

In this section, we give an overview of the design of CoPPER and how it builds on RaCUDA.

3.1 CoPPER Design

Figure 1 shows the design of CoPPER's pipeline, including some of the internal process of RaCUDA (which we modified somewhat, as described below). CoPPER takes an unoptimized CUDA kernel as input, and runs RaCUDA on it using a custom resource metric that estimates the relative running time impact of different operations (this metric is discussed in more detail in Section 4.1). This initial run of RaCUDA serves several purposes:

1. The resource usage bound produced as a result serves as a baseline cost for the unoptimized kernel; the cost estimates for optimized versions will be compared to this baseline to check for improvement.
2. The CUDA Abstract Syntax Tree (AST) produced by the front end will be fed into the optimizers.

[1] https://github.com/BinaryAnalysisPlatform/FrontC

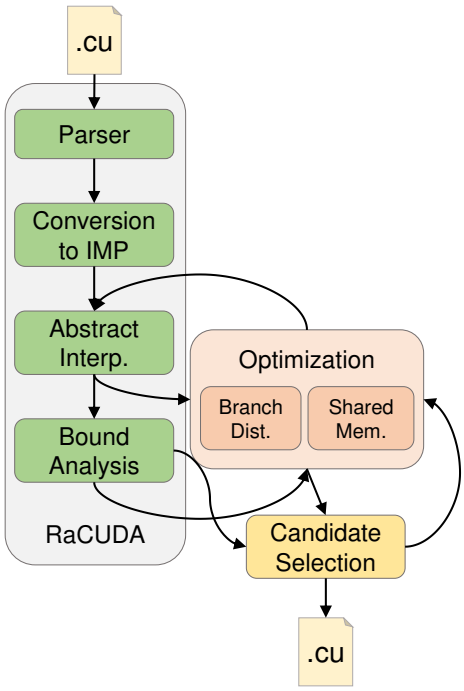

Figure 1. An illustration of the design of COPPER.

3. Some optimizations internally use information gathered during the abstract interpretation phase.

After the initial run of RaCUDA, the CUDA AST, annotated with information derived from the abstract interpretation, is passed to the first optimizer, which generates a number of *candidates*, which are distinct kernels with the optimization applied. The candidates may differ on the degree of agressiveness of the optmization, or other parameters of the particular optimization. COPPER then runs the abstract interpretation and bound analysis of RaCUDA on each candidate to estimate their execution cost. We need to perform the abstract interpretation again on each candidate, despite having performed it already on the unoptimized kernel, because the optimization may have changed runtime values in the kernel, especially the values of variables introduced in the conversion from CUDA to IMP which track cost.

After obtaining a bound (if possible) for each candidate, we compare the bounds and select the best candidate (or the original kernel if none of the candidates have better bounds). Because the bound is a (polynomial) function of the input parameters to the kernel, we compare the bounds using a heuristic comparison function on polynomials which we describe in more detail in Section 4.2. We then pass the winning candidate to the next optimization, and the process repeats until all optimizations have been applied. COPPER then returns the final, optimized kernel.

Note that this design assumes that optimizations are independent. For example, assume that two optimizations, 1 and 2, produce two candidates each (A and B for optimization 1, C and D for optimization 2). We could consider four candidate kernels, with each combination of optimizations (e.g., AC, AD, BC, and BD). To avoid the exponential blowup in the number of candidates, however, we assume that if, e.g., candidate A outperforms candidate B under only optimization 1 and C outperforms D under only optimization 2, then AC is the best of the four candidates.

4 Cost Metrics and Heuristics

4.1 Weighted Steps Cost Metric

The resource metric we use to predict execution cost bounds for kernels is called *weighted steps* and is based on the *steps* metric of Muller and Hoffmann [11]. Every basic CUDA operation (arithmetic operations, assignments, condition checks, etc.) costs one unit, with the following exceptions:

- *Divergent Warps.* CUDA threads operate in groups of 32 called *warps*. All threads in a warp execute the same instruction. If a warp executes a conditional statement where some threads in the warp take one branch and some take the other, the warp is said to *diverge*: both branches are executed, one after the other, with the appropriate threads activated and deactivated while executing each branch. When a warp diverges, our metric charges one unit of cost for the overhead of masking and unmasking threads, and then analyzes the two branches as if they execute in sequence.
- *Global Memory.* In CUDA, all threads share access to a large *global memory*. When multiple threads in a warp access different addresses in global memory, the GPU attempts to group these reads into as few reads as possible of fixed-size (e.g., 128 byte) segments, called *sectors*, as possible. If the 32 threads of a warp all read from (or write to) addresses very far from each other, this access would take 32 sectors. On the other hand, if all of the threads access the same handful of adjacent addresses, this might be possible with one sector. Our cost metric charges $5N$ units for a global memory read or write of N sectors (the multiplicative coefficient is to account for the fact that global memory is slower than *shared memory*, described below).
- *Shared Memory.* Multiple warps can be grouped into *blocks*. Threads within a block share access to a small, fast area of *shared* memory. Shared memory is organized into 32 banks, which may be accessed concurrently. If the 32 threads in a warp all access consecutive locations in shared memory, these accesses all map to different banks, and the locations are all accessed concurrently. However, if two or more accesses target different addresses in the same bank (called a *bank conflict*), these accesses must be sequentialized. Our

metric charges N units for an access to shared memory where at most N accesses target the same bank.

In short, the weighted steps metric aims to simulate the time taken to execute one warp, including the added latency that might be caused by uncoalesced global memory accesses (those in which a warp accesses memory locations that cannot be grouped into a few accesses of consecutive regions) and shared memory bank conflicts. In future work, the relative costs of various operations should be tuned based on empirical observation, and can even be adjusted for a particular GPU. The values used here are simply initial heuristics to get a working prototype.

4.2 Polynomial Comparison Heuristic

Each candidate optimized kernel, including the baseline with no optimizations, is analyzed using the resource metric from Section 4.1 to obtain an upper bound on the cost of execution, as a polynomial function of input parameters. We must then compare these upper bounds to determine whether one candidate is better than another. In this section, we define a comparison function $\preceq$ on two multivariate polynomials f and g with the same number of arguments. In general, it is not possible to define $\preceq$ such that $f \preceq g$ if and only if $f(x_0, \ldots, x_n) < g(x_0, \ldots, x_n)$ for all $x_0, \ldots, x_n$. Nor do we wish to rely solely on asymptotic complexity classes: many optimizations do not change the asymptotic complexity of a program. Intuitively, we want $f \preceq g$ when $f(x_0, \ldots, x_n) < g(x_0, \ldots, x_n)$ for a large domain of commonly occurring values for $x_0, \ldots, x_n$. This domain will, of course, depend on the application and it will likely be beneficial to adjust the heuristic $\preceq$ function as needed for a particular application.

We use the following definition for $\preceq$:

1. If f is of lower degree than g, then $f \preceq g$
2. If f and g are of the same degree n, then compare the sums of the n^{th} degree coefficients of f and g. If this sum is lower for f, then $f \preceq g$. If it is lower for g, then $g \preceq f$. If they are equal (or within a specified absolute difference), proceed to compare the coefficients at the next-higher degree.
3. If this process terminates without determining that $f \preceq g$ or $g \preceq f$, then we say $f \npreceq g$ and $g \npreceq f$. Because we choose a candidate f only if $f \preceq g$, where g is the best candidate so far, this will result in the new candidate not being chosen.

This definition has the property that $f \preceq g$ if and only if $\lim_{x_0,\ldots,x_n \to \infty} \frac{f(x_0,\ldots x_n)}{g(x_0,\ldots x_n)} \leq 1$, but also provides a useful comparison for asymptotically "equal" polynomials where the limit approaches 1.

5 Implemented Optimizations

In this section, we present two algorithms aimed at optimizing CUDA kernels. The first is the Branch Distribution algorithm, which identifies and moves complex, or expensive common code blocks shared between conditional branches. This reduces the impact of branch divergence, which helps improve performance. The second algorithm optimizes kernels by moving certain global memory array parameters into shared memory. It first determines the access bounds for array parameters, identifies which ones can benefit from shared memory, and modifies the original code to use shared memory instead of global.

5.1 Branch Distribution

Branch Distribution [6] reduces the performance impact of some instances of branch divergence by identifying common blocks of code shared between branches, and restructuring the control flow by moving the common code outside of the conditional statements. This algorithm is especially useful when the common code blocks inside the conditional branches are computationally expensive. The expensive statements will be executed only once by all threads in the warp, rather than twice in sequence.

For example, suppose we have a conditional in our kernel, with the potentially expensive, or complex, statement A.

```
if (condition1) { A; B; }
else { A; C; }
```

In this case, if threads in a warp have different values for the condition, the warp would diverge and the two branches would execute sequentially, reducing the performance of the warp. Branch distribution could refactor the code as follows:

```
A;
if (condition1) { B; }
else { C; }
```

In this case, the expensive code block A has been moved outside of the conditions, so all threads would execute the block simultaneously, reducing the impact of the branch divergence and improving performance.

In some cases, branch distribution may not help the performance. For example, if we started off with this code:

```
if (condition1) { A; B; C;}
else { D; B; F; }
```

It would be refactored into this:

```
if (condition1) { A; }
else { D; }
B;
if (condition1) { C; }
else { F; }
```

If the common code B is extremely expensive, then this refactoring is indeed likely to be an optimization. [2] On the other

[2]Note that simply counting the number of potential divergences would miss this optimization, as we have increased the total number of divergences but decreased their impact on performance.

hand, if B has a negligible performance impact, then the overhead of having two (divergent) conditional statements may actually negatively affect our performance. Because of this, we only want to move common code blocks outside of the conditional statements when we are confident that they are sufficiently expensive. The weighted steps metric (Section 4.1) can estimate whether an application of branch distribution is likely to be an optimization or not. However, we wish to reduce the number of candidate kernels that need to be analyzed, and so our implementation uses some heuristics to determine when to apply branch distribution.

In addition to the kernel AST, the optimization accepts as a parameter a "complexity cutoff" indicating how "expensive" a block of code should be before the optimization will factor it out of a conditional. Our optimization will continue factoring out blocks of code until their complexity falls below the given cutoff. As a simple complexity metric, we use the number of statements, with loop statements counting triple.

The branch distribution algorithm consists of two parts: a function for finding the Most Complex Common Code Block (MCCCB) between two ASTs (e.g., the two branches of a conditional) and a main function that applies the MCCCB algorithm to conditional branches in the AST until all common code blocks over the complexity cutoff are factored out. We now describe these two processes in more detail.

Most Complex Common Code Block (MCCCB). The MCCCB function takes as parameters two blocks of code, as well as the complexity cutoff. We use a dynamic programming approach similar to the longest common substring problem, where the complexity score replaces the substring length in the original algorithm. The function outputs the most complex sub-AST that the two code blocks have in common, and its complexity score.

Branch Distribution Algorithm. This function serves as the entry point for the algorithm. It processes the statements in the kernel one at a time, recursively traversing substatements (such as loop bodies and conditional branches). When the traversal reaches a conditional statement, it calculates the most complex common code block between the two branches (which have, themselves, already had branch distribution applied). If common code is found that passes the complexity cutoff, the function reorganizes the code to move the common code outside the conditional branches.

5.2 Shared Memory

The second optimization we implemented automatically moves arrays from slow global memory to fast shared memory. If the array elements are used frequently, the performance benefits of storing them in shared memory are likely to outweigh the cost of initially moving the data from global to shared memory (and back if they're modified). In order to determine whether an array can be allocated in shared memory, and what portion of the array to move, we must first determine lower and upper bounds on the indices accessed by a kernel, as a function of the thread ID. In order to facilitate this analysis, we have extended the abstract interpretation pass of RaCUDA (see Section 2.1) to annotate expressions in the AST with lower and upper bounds on the runtime value of an expression (in terms of the thread ID and other parameters), if these can be computed from information already tracked by the abstract interpretation.

As an example, the top of Figure 2 shows a motivating example kernel of Muller and Hoffmann [11] that adds and subtracts the vector `A` to even and odd rows of a matrix `B`, respectively (the suffix 2 in the kernel's name refers to the fact that, in the motivating example from which this is drawn, this kernel has already gone through 2 rounds of iterative revision). The array element `A[i]` is accessed once for each row of `B`, so it is a performance optimization to move `A` to shared memory. The result of this optimization, as performed by our implementation, is shown on the bottom of the figure.

The overall algorithm can be broken up into three sections:

1. Generate a hashtable of access bounds for each array parameter in the kernel.
2. Identify which array parameters can be moved into shared memory.
3. Modify the original code to move the data between global and shared memory, and rewrite accessed to the array to target the new, shared memory, array.

Note that our optimization only determines which arrays can be moved to shared memory, not whether doing so would be profitable. For a set A of arrays that can be moved, the optimization will generate a candidate kernel corresponding to each element of the power set of A, in which each combination of arrays has been moved to shared memory. The subsequent RaCUDA analysis passes will determine which of these (including the original candidate in which no arrays are moved) is likely to be most performant.

Hash table Generation. We generate a hash table that maps arrays to information about its access patterns. Each array that is a parameter to the kernel corresponds to an entry in the hash table. We build the hash table by iterating through the code block, populating an intermediate hash table that stores the lower and upper bounds of any expression used to index into each array parameter. In the example of the kernel in Figure 2, both arrays have two accesses. Each access to `A[i]` (both accesses are to the same element, but we do not yet perform this deduplication) has upper and lower bounds of `blockIdx.x * blockDim.x + threadIdx.x` because this is the statically known value of `i`. The expression `j * w + i`, used as the first index to `B`, has an upper bound of `h * w + blockIdx.x * blockDim.x + threadIdx.x` (`h` is an upper bound of `j` and `i` is bounded as above) and a lower bound of `blockIdx.x * blockDim.x + threadIdx.x` (i.e., where `j` is 0). We can bound `(j + 1) * w + i` similarly.

```
__global__ void addSubArray2 (int *A, int *B, int w, int h) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  for (int j = 0; j < h; j += 2) {
    B[j * w + i] += A[i];
    B[(j + 1) * w + i] -= A[i];
  }
}

__global__ void addSubArray2 (int *A, int *B, int w, int h) {
  signed long int lower_bound_A1 = blockIdx.x * blockDim.x;
  __shared__ int A1[BLOCKSIZE];
  for (int temp = 0; temp < 1; temp++)
    A1[threadIdx.x] = A[(blockIdx.x * blockDim.x) + threadIdx.x];
  __syncthreads();
  for (int j = 0; j < h; j += 2) {
    B[j * w + i] += A1[i - lower_bound_A1]];
    B[(j + 1) * w + i] -= A1[i - lower_bound_A1]];
  }
}
```

Figure 2. The unoptimized (top) and optimized (bottom) versions of the addSub2 kernel.

Once the hashtable has been populated, we remove duplicate entries and determine, for each array, how many shared memory arrays should be allocated for it—if a thread accesses multiple disjoint segments of an array, it may be more efficient to store only these segments as separate arrays in shared memory. Each of these segments is assigned a unique number, called a *variant number*. In the case of the example, both arrays have one variant.

Sharable Parameter Identification. After generating the hashtable, we use the access bounds from the array parameters to determine which arrays can be moved to shared memory. In particular, we only copy an array if the size of its segments can be determined at compile time to be a constant multiple of the number of threads in a block.[3] We test whether this is the case for each array variant by subtracting the lower bound from the upper bound, taking the size of a block to be the difference between the upper and lower bounds of the thread ID. In the example, the upper bound of `i` becomes `blockIdx.x * blockDim.x + blockDim.x` and the lower bound is `blockIdx.x * blockDim.x + 0`, for a segment size of `blockDim.x`. Technically, block dimensions are indicated dynamically at the time a kernel is launched. However, in most kernels, these are compile-time constants, so we declare the shared memory variant `A1` as an array of size `BLOCKSIZE`, where `BLOCKSIZE` will need to be filled in with the constant dimension of a block in order to use the resulting code.

Code Modification. Finally, we modify the original code to bring global parameters that were identified as movable into shared memory (and back to global memory at the end of the kernel, if any of the accesses are writes). To do this, we generate code that copies the data between global and shared through this process:

1. Declare shared memory arrays for each array variant.
2. Declare and define variables to capture the lower bound of each variant's accesses.
3. Add code to copy the necessary data for each variant to the new shared array (using a **`for`** loop if a variant's segment is wider than the thread block).
4. Modify the original code to use the shared memory array instead of the original global memory array.
5. Copy the changes from shared memory back to global memory (the reverse of step 3) if writes are detected.

The bottom of Figure 2 shows the optimized version of the `addSub2` kernel: the `for` loop copies values from `A` into the new shared array `A` (indeed, the body of the loop is only executed once, so a loop is not needed), and the accesses to `A[i]` are changed to `A[i - lower_bound_A1]` where `lower_bound_A1` is defined to be the lowest index of `A` accessed by a particular block, as determined by the abstract interpretation pass.

The restrictions on what arrays can be copied to shared memory are fairly conservative in order to ensure that the

[3]This is not a restriction of CUDA; indeed, it is possible (though slightly unwieldy) to allocate arrays in shared memory whose size is not known at compile time. However, the loops that are necessary to copy such array segments into shared memory pose problems for RaCUDA's analysis, so we would not be able to analyze the resulting code for improvement.

resulting code can be analyzed by RaCUDA and that the code rewriting in Step 4 above preserves the semantics of the code. In the future, we hope to be able to relax some of these restrictions.

6 Evaluation

We evaluated CoPPER on a set of benchmarks drawn from prior work and other sources. The RaCUDA analysis and our optimizations aim to improve code written by novices with little knowledge of performance properties of CUDA, and so the goal in the evaluation was to approximate such kernels. With this in mind, we used three sources for benchmarks:

- Synthetic benchmarks used by Muller and Hoffmann [11] to illustrate the performance bottlenecks targeted by RaCUDA. These benchmarks intentionally possess the performance bottlenecks.
- Code downloaded from public GitHub repositories. We filtered for repositories under the "CUDA" topic whose primary language was CUDA, and sorted by lowest number of stars, in order to target repositories of programmers learning or practicing CUDA. Some of these benchmarks were edited to remove features not supported by RaCUDA and, in some cases, to remove optimizations such as the use of shared memory if the programmer had already optimized in this fashion.
- Code produced by ChatGPT [13] with prompts beginning "Pretend you're a student just learning CUDA and you don't know about performance bottlenecks like warp divergence and and shared memory..."

We discuss the benchmarks in more detail below.

- addSub0 and addSub2 were drawn from Muller and Hoffmann [11], and are two versions of a kernel that manipulates a matrix (the code for addSub2 appears in Figure 2). Kernel addSub0 uses a thread per row and conditions on the row modulo 2, resulting in a warp divergence. Kernel addSub2 uses a thread per column and iterates over rows for better locality, but does not use shared memory. The kernels have two parameters, w and h, the width and height of the matrix.
- SYN-BRDIS is a synthetic kernel written by Muller and Hoffmann [11] and inspired by Han and Abdelrahman [6], to motivate the use of branch distribution. It has two parameters, M and N, which are used as bounds of inner and outer loops, respectively.
- DMVM is a kernel for dense matrix-vector multiplication produced by ChatGPT. It doesn't have divergences or use shared memory, but the kernel's locality properties doesn't lend itself to the use of shared memory. The kernel has two parameters, w and h, the width and height of the matrix.
- For the DVMM kernel, we manually modified the DMVM kernel above to instead multiply the vector by the matrix, but otherwise left the code the same.
- ElementWise is a kernel produced by ChatGPT with the goal of adding two matrices. It is worth noting that the logic of the code is incorrect, and repeatedly iterates over the first rows of the matrices (and does not use parallelism). Because of this logic error, the kernel could make use of shared memory.
- The 1DConv benchmark, taken from GitHub, computes a 1-D convolution of an image vector.
- The Rank benchmark, taken from GitHub, computes the rank of each element in an array in numerical order. The array could be stored in shared memory.

We ran CoPPER on all of these benchmarks and recorded the execution time of CoPPER as well as what optimization(s), if any, were found. The experiments were run on a commodity machine with 16 GB of memory and 2.6 GHz processors. Note that CoPPER runs sequentially and uses only the CPU, not the GPU. The results are presented in Table 1. The first two columns show the time to run CoPPER overall and the time spent in RaCUDA, respectively. The first observation about the time taken by CoPPER is that it inherits from RaCUDA a heavy dependence on the complexity of the code. This is to be expected, because CoPPER runs RaCUDA to completion on each candidate. While some benchmarks finish in approximately a second, SYN-BRDIS (the most expensive-to-analyze benchmark in the original paper on RaCUDA [11]) took approximately half an hour and Rank timed out after 10 hours. Across all benchmarks, the vast majority of the time (>98%, and >99% for most benchmarks) was spent running RaCUDA.

For benchmarks that did not time out, CoPPER found optimizations for all except DMVM and ElementWise. We consider this a correct result for DMVM, as a manual inspection of the kernel shows that moving the matrix or vector to shared memory would not be beneficial: as written, each element of the matrix and vector is loaded from global memory only once (in DVMM, naively swapping the roles of the vector and matrix causes each element of the vector to be loaded for each iteration of the loop, making it beneficial to move the vector to shared memory). In ElementWise, the first row of the matrix could be moved to shared memory, but CoPPER does not find this optimization because of its restriction to fixed-length array segments.

For benchmarks where an optimization was found, the last three columns of the table give the upper bound on the weighted steps cost metric computed by CoPPER for the input kernel and the optimized kernel, as a function of the input parameters described above. The last column gives the type of optimization performed (for all kernels, only one optimization, either branch distribution or global-to-shared, turned out to be beneficial).

Next, we selected two kernels (addSub2 and SYN-BRDIS) for a deeper evaluation of the benefits of the optimization.

Table 1. Optimization time and optimizations found for the benchmarks.

Benchmark	Time (s)	RaCUDA (s)	Opt. found?	Orig. bound	Optimized bound	Type(s) of opt
addSub0	7.527	7.522	Yes	$7 + 1078w$	$8 + 746w$	Branch Dist.
addSub2	0.414	0.408	Yes	$166 + 152h$	$185 + 108h$	Global-to-shared
SYN-BRDIS	1653.316	1653.301	Yes	$15 + 70N + 116MN$	$16 + 61N + 93MN$	Branch Dist.
DMVM	1.438	1.433	No			
DVMM	1.651	1.642	Yes	$50 + 128h$	$113 + 84h$	Global-to-shared
ElementWise	53.883	53.881	No			
1DConv	8.049	8.038	Yes	812	624	Global-to-shared
Rank	>10h	—	—			

Table 2. Performance of unoptimized (Unopt.) and optimized (Opt.) versions of the addSub2 kernel.

h	Unopt. (ms)	Opt. (ms)	% Impr.
1000	0.419	0.427	-1.91
2000	0.798	0.791	0.88
3000	1.213	1.181	2.64
4000	1.591	1.555	2.26
5000	1.975	1.927	2.43
6000	2.373	2.308	2.74
7000	2.773	2.700	2.63
8000	3.153	3.078	2.38
9000	3.556	3.468	2.47
10000	3.951	3.846	2.66

For both kernels, we executed both the optimized and unoptimized versions of the kernel, with suitable setup code. The experiments were run on the same machine as the benchmarks above, which has an NVIDIA GeForce GTX 1650 GPU. The results for addSub2 are in Table 2, for varying values of h (the bound is unaffected by w). The last column gives the percentage improvement provided by the optimization. For smaller values of h, the optimization gives little benefit. Indeed, for $h = 1000$, the optimized version runs almost 2% slower. However, for larger values of h, the optimized version converges to approximately 2.5% faster, a modest improvement. This behavior is consistent with the bounds found by RaCUDA for the optimized and unoptimized kernels: the constant factor of the bound is larger in the optimized version, because of the overhead introduced by moving values from global memory to shared memory. This constant overhead is outweighed by the improvement in the linear bound for larger values of h. The improvement is, however, smaller than would be expected from the bounds (as h grows, we would expect the difference to converge to

$$\frac{152h - 108h}{152h} \approx 29\%$$

Table 3. Performance of unoptimized (Unopt.) and optimized (Opt.) versions of the SYN-BRDIS kernel.

M	N	Unopt. (ms)	Opt. (ms)	% Impr.
128	128	4.130	3.994	3.29
128	256	8.170	7.915	3.12
128	384	12.209	11.842	3.01
128	512	16.239	15.770	2.89
128	640	20.277	19.696	2.87
128	768	24.318	23.624	2.85
128	896	28.350	27.552	2.81
128	1024	32.388	31.47	2.83
256	128	8.227	7.941	3.48
384	128	12.321	11.905	3.38
512	128	16.411	15.854	3.39
640	128	20.508	19.810	3.40
768	128	24.604	23.771	3.39
896	128	28.708	27.728	3.41
1024	128	32.815	31.683	3.45

This indicates that the weighted steps heuristic used is an imperfect model of the cost of execution and could be improved with real-world data.

The results for the SYN-BRDIS kernel are shown in Table 3, first varying N and then varying M while keeping the other parameter constant. The branch distribution optimization results in an improvement for all values tested, as predicted by RaCUDA: the constant overhead introduced by the optimization is quite small. Somewhat counterintuitively, the improvement seems to decrease slightly as N increases for fixed M, but more experimentation is needed to determine whether this is significant or an experimental artifact. As with addSub2, the improvements are smaller than predicted by the weighted steps metric, but are not trivial.

7 Conclusion and Future Work

In this work, we have proposed a framework for using static resource analysis to guide the automatic optimization of

general-purpose GPU kernels written in CUDA. Static resource analysis provides a sound, hardware-independent basis for determining the performance benefits of optimizations. As a proof of concept, we implemented the COPPER tool with a simple search strategy and two optimizations. COPPER is able to achieve modest speedups in benchmarks with performance bottlenecks.

Much work remains to be done. For one, a full evaluation of an improved and expanded version of COPPER would involve a larger set of benchmarks more representative of the kernels we target: simple code written by novice CUDA programmers. Such an evaluation could use a systematic study of public code repositories or of student submissions in, for example, an undergraduate parallel systems course. These evaluations are beyond the scope of this first study, which simply aims to establish the feasibility of static analysis as a tool in guiding the optimization of CUDA kernels.

It is also important to note that our goal is not to compete with the existing work being done in tuning GPU kernels. Indeed, this work has made major strides in improving aspects such as the sampling and search strategy, from which our framework could benefit. Future work should evaluate the use of static analysis as a replacement for kernel execution in a state-of-the-art auto-tuning framework (and/or the integration of major ideas from these frameworks into our system). In addition, as discussed in the evaluation, our optimization framework is only as good as the resource model it (and in particular, RACUDA) uses to predict the performance of candidate kernels. The quantitative model behind RACUDA is based around predicting the impact of warp divergences, global memory accesses, and shared memory bank conflicts. The original evaluation of RACUDA shows that it is fairly precise in quantitatively predicting these bottlenecks. However, combining this information into a precise estimate of kernel execution time is beyond the scope of RACUDA. One approach would be to combine RACUDA's predictions of certain properties of the code with some of the models described in Section 2, which could turn these predictions into more concrete predictions of kernel execution time.

Acknowledgments

This work was partially supported by the National Science Foundation under award number CCF-2007784.

References

[1] Piotr Bialas and Adam Strzelecki. 2016. Benchmarking the Cost of Thread Divergence in CUDA. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Ewa Deelman, Jack Dongarra, Konrad Karczewski, Jacek Kitowski, and Kazimierz Wiatr (Eds.). Springer International Publishing, Cham, 570–579. https://doi.org/10.1007/978-3-319-32149-3_53

[2] Lorenz Braun, Sotirios Nikas, Chen Song, Vincent Heuveline, and Holger Fröning. 2021. A Simple Model for Portable and Fast Prediction of Execution Time and Power Consumption of GPU Kernels. *ACM Trans. Archit. Code Optim.* 18, 1, Article 7 (dec 2021), 25 pages. https://doi.org/10.1145/3431731

[3] Wilson Feng and Tarek S. Abdelrahman. 2017. A Sampling Based Strategy to Automatic Performance Tuning of GPU Programs. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1342–1349. https://doi.org/10.1109/IPDPSW.2017.46

[4] Joseph D. Garvey and Tarek S. Abdelrahman. 2015. Automatic Performance Tuning of Stencil Computations on GPUs. In *2015 44th International Conference on Parallel Processing*. 300–309. https://doi.org/10.1109/ICPP.2015.39

[5] Dominik Grewe and Anton Lokhmotov. 2011. Automatically Generating and Tuning GPU Code for Sparse Matrix-Vector Multiplication from a High-Level Representation. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units* (Newport Beach, California, USA) *(GPGPU-4)*. Association for Computing Machinery, New York, NY, USA, Article 12, 8 pages. https://doi.org/10.1145/1964179.1964196

[6] Tianyi David Han and Tarek S. Abdelrahman. 2011. Reducing Branch Divergence in GPU Programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units* (Newport Beach, California, USA) *(GPGPU-4)*. ACM, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/1964179.1964184

[7] Tianyi David Han and Tarek S. Abdelrahman. 2014. Automatic Tuning of Local Memory Use on GPGPUs. *CoRR* abs/1412.6986 (2014). arXiv:1412.6986 http://arxiv.org/abs/1412.6986

[8] Haipeng Jia, Yunquan Zhang, Guoping Long, Jianliang Xu, Shengen Yan, and Yan Li. 2012. GPURoofline: A Model for Guiding Performance Optimizations on GPUs. In *Euro-Par 2012 Parallel Processing*, Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 920–932.

[9] Yinan Li, Jack Dongarra, and Stanimire Tomov. 2009. A Note on Auto-tuning GEMM for GPUs. In *Computational Science – ICCS 2009*, Gabrielle Allen, Jaroslaw Nabrzyski, Edward Seidel, Geert Dick van Albada, Jack Dongarra, and Peter M. A. Sloot (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 884–892.

[10] Azamat Mametjanov, Daniel Lowell, Ching-Chen Ma, and Boyana Norris. 2012. Autotuning Stencil-Based Computations on GPUs. In *2012 IEEE International Conference on Cluster Computing*. 266–274. https://doi.org/10.1109/CLUSTER.2012.46

[11] Stefan K. Muller and Jan Hoffmann. 2021. Modeling and Analyzing Evaluation Cost of CUDA Kernels. 5, POPL, Article 25 (1 2021), 31 pages. https://doi.org/10.1145/3434306

[12] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded Expectations: Resource Analysis for Probabilistic Programs. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 496–512. https://doi.org/10.1145/3192366.3192394

[13] OpenAI. 2024. ChatGPT. https://chat.openai.com/chat. (Jan. 2024 version).

[14] Richard Arnoud Schoonhoven, Ben van Werkhoven, and Kees Joost Batenburg. 2023. Benchmarking Optimization Algorithms for Auto-Tuning GPU Kernels. *IEEE Transactions on Evolutionary Computation* 27, 3 (2023), 550–564. https://doi.org/10.1109/TEVC.2022.3210654

[15] Ilia Shumailov, Yiren Zhao, Daniel Bates, Nicolas Papernot, Robert Mullins, and Ross Anderson. 2021. Sponge Examples: Energy-Latency Attacks on Neural Networks. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. 212–231. https://doi.org/10.1109/EuroSP51992.2021.00024