College of
William & Mary
Department of Computer Science

WM-CS-2008-09

A Cross-Input Adaptive Framework for GPU Programs Optimization

Yixun Liu Eddy Z. Zhang Xipeng Shen

Oct. 7, 2008

Submitted to IEEE IPDPS09

A Cross-Input Adaptive Framework for GPU Programs Optimization

Yixun Liu   Eddy Z. Zhang   Xipeng Shen

Abstract

Recent years have seen a trend in using graphic processing units (GPU) as accelerators for general-purpose computing. The inexpensive, single-chip, massively parallel architecture of GPU has evidentially brought factors of speedup to many numerical applications. However, the development of a high-quality GPU application is challenging, thanks to the large optimization space and complex unpredictable effects of optimizations on GPU program performance.

Recently, several studies have attempted to use empirical search to help the optimization. Although those studies have shown promising results, one important factor—program inputs—in the optimization has remained unexplored. In this work, we initiate the exploration in this new dimension. By conducting a series of measurement, we find that the ability to adapt to program inputs is important for some applications to achieve their best performance on GPU. In light of the findings, we develop an input-adaptive optimization framework, namely G-ADAPT, to address the influence by constructing cross-input predictive models for automatically predicting the (near-)optimal configurations for an arbitrary input to a GPU program. The results demonstrate the promise of the framework in serving as a tool to alleviate the productivity bottleneck in GPU programming.

1 Introduction

As a specialized single-chip massively parallel architecture, Graphics Processing Units (GPU) have shown orders of magnitude higher throughput and performance per dollar than traditional CPUs. The properties have recently drawn great interest from researchers and industry practitioners in extending GPU computations beyond their traditional uses in graphics rendering [3, 7, 16–19].

Besides hardware innovations, progresses in programming models have significantly improved the accessibility of GPU for general purpose computing. In particular, the NVIDIA **C**ompute **U**nified **D**evice **A**rchitecture (CUDA) abstracts GPU as a general-purpose multithreaded SIMD (single instruction, multiple data) architectural model, and offers a C like interface—supported by a compiler and a runtime system—for GPU programming. CUDA simplifies the writing of a GPU program.

However, developing an *efficient* GPU program remains as challenging as before if not even more. The difficulties come from four aspects. The *first* is the complexity in GPU architecture. On an NVIDIA GeForce 8800 GT, for example, there are over one hundred cores, four types of off-chip memory, hundreds of thousands of registers, and many parameters (e.g., maximum number of threads per block, thread block dimensions) that constrain the programming. The *second* difficulty is that the multi-layered software execution stack makes it difficult to predict the effects of a code optimization. A special difficulty with CUDA is that currently a GPU program has to be compiled by the NVIDIA CUDA compiler (NVCC) and run on the NVIDIA CUDA runtime system, some details of both of which are not disclosed yet. *Third,* an optimization often has multiple effects, and the optimizations on different parameters often strongly affect each other. *Finally*, some GPU applications are input-sensitive. The best optimizations of an application may be different when different inputs are given to the application. Together, these factors not only make manual optimizations time consuming and difficult to attain the optimal, but also form great hurdles for automatic optimizations, based on either analytical models or empirical search.

The optimization issue is particularly important for GPU programming. Because of the tremendous computing power of GPU, there can be orders of magnitude performance difference between well optimized and poorly optimized versions of an application [3, 17, 18]. Those difficulties thus make GPU program optimization one of the most important bottlenecks in GPU computing.

Several recent studies have tried to tackle the problem through empirical search-based approaches. Ryoo and his colleagues [18] have defined *efficiency* and *utilization* models for GPU pragma to help prune the optimization space. Baskaran et al. [3] have developed a polyhedral compiler model to optimize global memory accesses in affine loop nests, and used model-driven empirical search to determine the levels of loop unrolling and tiling.

Although both studies have shown promising results, neither of them have explored the influence of program inputs on the optimization. Program inputs refer to both the values and other related properties (e.g., dimensions of an input matrix) of the inputs given to a program. In this work, we initiate an exploration in this new dimension, showing that program inputs may affect the effectiveness of an optimization by up to a factor of 6. Based on the exploration, we develop a tool, G-ADAPT (GPU adaptive optimization framework), to efficiently discover near-optimal decisions for GPU program optimizations, and moveover, tailor the decisions to each program input.

More specifically, this work makes three major contributions. *First*, this work exposes the influence of program inputs on GPU program optimizations. We are not aware of any previous studies in this direction. The lack of such explorations may be due to a common intuition that as most GPU applications divide a task into small sub-tasks, the changes in their inputs rarely matter to the optimizations as long as the sub-tasks remain similar. However, our experiments show that almost half of our benchmarks counter the intuition: Their inputs exhibit strong influence on their optimizations. *Second*, we develop a source-to-source compiler-based framework for empirical search for the best optimizations for GPU applications. The framework is distinctive in that it conducts program transformations and optimization space search fully automatically, and meanwhile, offers a set of pragmas for programmers to easily incorporate their knowledge into the empirical search process. *Finally*, we incorporate Regression Trees into the optimization framework to construct cross-input predictive models for automatically tailoring optimizations to program inputs. As far as we know, this is the first framework that allows cross-input adaptive optimizations for GPU applications.

Experiments on NVIDIA GeForce 8800 GT GPU show that the adaptive optimization framework can predict the best optimizations for 7 GPU applications with over 93% accuracy. The adaptive optimization improves the programs performance by as much as several times.

We organize the paper as follows. Section 2 provides some background on GPU and its programming model. Section 3 discusses the challenges in GPU program optimizations. Section 4 describes our solution to those challenges, an adaptive optimization framework. Section 5 reports evaluation of the framework. After discussing related work in Section 6, we conclude the paper with a brief summary.

2 Background on GPU Architecture and CUDA

This work uses the NVIDIA GeForce 8800 GT GPU as the architecture. It is a single-chip massively parallel architecture, with 112 cores, certain amount of on-chip memory, and 512MB off-chip memory. The GPU contains 14 streaming multiprocessors (SMs). Each SM contains 8 streaming processors (SPs) or cores, with the clock rate set at 1.51GHz. Each SM also includes 2 special function units (SFUs) for the fast execution of complex floating point operations, such as sine, cosine. Besides the computing units, on each SM, there are 8192 32-bit registers and 16KB shared memory. Unlike cache, the shared memory has to be managed explicitly in each GPU application.

The off-chip memory includes a 512MB global memory, which is both readable and writable by every SP, and some constant memory and texture memory, which can only be read by the SPs. The constant memory and texture memory are cachable thanks to some on-chip cache, but the global memory is not.

Directly programming such a massively parallel architecture is difficult; CUDA, a programming model developed by NVIDIA, simplifies GPU programming by offering a middle layer with some abstractions. The programming interface of CUDA is ANSI C with some extensions. With CUDA, a GPU application is composed of CPU code and GPU kernels. CUDA abstracts the execution of a GPU kernel as a multithreaded SIMD computation. The threads are grouped into many warps with 32 threads in each. Those warps are evenly partitioned into a set of thread blocks. Each time, the runtime system maps one or more thread blocks to an SM. The warps in those blocks are dynamically scheduled to run on the SM. In GeForce 8800 GT, half of a warp is a SIMD execution unit. If one warp is stalled (e.g., due to memory accesses), the other warps can be switched in with nearly zero overhead. Therefore, the number of warps or thread blocks that are mapped to an SM determines the effectiveness of the pipelining execution in hiding latency. As the thread-block size determines the mapping of blocks on SMs, it is an important parameter in GPU program optimizations.

Threads may communicate in the following ways. Each thread block has a certain amount of shared memory, which is directly accessible only by the threads in that block. Threads in a block can be synchronized by a *__syncthreads* primitive. But the only way for cross-block communication is through global memory, and the only way for safe cross-block synchronization is by the termination of the GPU kernel.

3 Challenges in the Optimization of GPU Programs

Although CUDA simplifies GPU programming, it reduces little if any difficulty in optimizing GPU applications; in some degree, the added layer even complicates the optimization as it makes performance prediction harder.

Optimizations There are mainly two ways to improve the performance of a GPU program: the maximization of the usage of computing units, and the reduction of the number of dynamic instructions. Optimizations to reach the first goal fall into two categories. The first includes those techniques that attempt to increase the occupancy of the computing units. One typical example is to reduce resource consumption of a single thread so that multiple thread blocks can be assigned to an SM at the same time. The multiple blocks may keep the SM busy even when the threads in one block are stalled for synchronization. Transformations for that purpose include the adjustment of the number of threads per block, and loop tiling. The second category contains the techniques that try to reduce latencies caused by memory references (or branches). Examples include the use of cachable memory, the reduction of bank conflicts in shared memory, and the conversion for coalescing memory accesses—that is, the threads in a warp reference a sequence of contiguous memory addresses at the same time.

Optimizations to reach the second goal involve many traditional compiler transformations, such as loop unrolling, common subexpression elimination. Although the CUDA compiler, *NVCC*, has implemented many of these techniques, researchers have seen great potential to adjust some of those optimizations, such as the levels of loop unrolling [3, 18].

Challenges It is difficult to analytically determine the best optimizations for a GPU application, for three reasons. *First*, it is often difficult to accurately predict the effects of an optimization on the performance of the GPU application. The effects are often non-linear as what Ryoo et al. have shown on the adjustment of thread block sizes [18]. Furthermore, the undisclosed details of the CUDA compiler and other abstractions add more unpredictability. *Second*, different optimizations often affect each other. Loop unrolling, for example, removes some dynamic instructions and exposes more opportunities for scheduler to exploit; but it also increases register pressure in each thread. Given that the number of registers in an SM is limited, it may result in fewer threads an SM can hold, and thus affect the selection of thread-block size. *Finally*, the many limits in GPU hardware add further complexity. The limits in GeForce 8800 GT include that the maximum number of threads per block is 512, the maximum number of threads per SM is 768, the maximum number of blocks per SM is 8, and at each time, all the threads assigned to an SM must use no more than 16KB shared memory and 8192 registers in total. These constraints plus the unpredictable effects of optimizations make it extremely difficult to build an accurate analytical model for GPU optimization.

An alternative solution to determine the best optimizations is through empirical search, whereby the GPU application runs for many times, each time with different optimizations applied. However, it is necessary to remove three obstacles before this solution becomes practical. First, there should be an easy-to-use mechanism for the construction of the optimization space. Furthermore, a compiler must be able to conduct the corresponding transformations automatically. Second, effective space prunes are necessary for the search efficiency, especially when the optimization space is large. Finally, the solution must be able to handle the influence of program inputs. Our study (Section 5) shows that the best values of optimization parameters of some GPU programs are different for different inputs. For example, an optimization suitable for one input to a reduction program degrades the performance of the program on another input by as much as 640%. For such programs, it would be desirable to inject input-adaptability into their executable code.

4 Adaptive Optimization Framework

Our solution to the challenges in GPU program optimization is a cross-input adaptive framework. This section first gives an overview of the framework, and then elaborates every component in the framework.

4.1 Overview

Figure 1 shows the structure of the adaptive optimization framework, *G-ADAPT*. It consists of two parts, corresponding to two stages of the optimization. The task of the first stage, shown to the left of the dot vertical line in Figure 1, is to conduct a series of empirical search in the optimization space of the given GPU program. During the search, a set of performance data, along with the program inputs, are stored into a database. After the first stage finishes, the second stage, shown as the the right part of Figure 1, uses the performance database to recognize the relation between program inputs and the optimal optimization decisions. G-ADAPT then transforms the original GPU code into a program that is able to automatically select the best optimized versions for an arbitrary input.

The first part uses empirical search to overcome the difficulty in modeling GPU program performance;

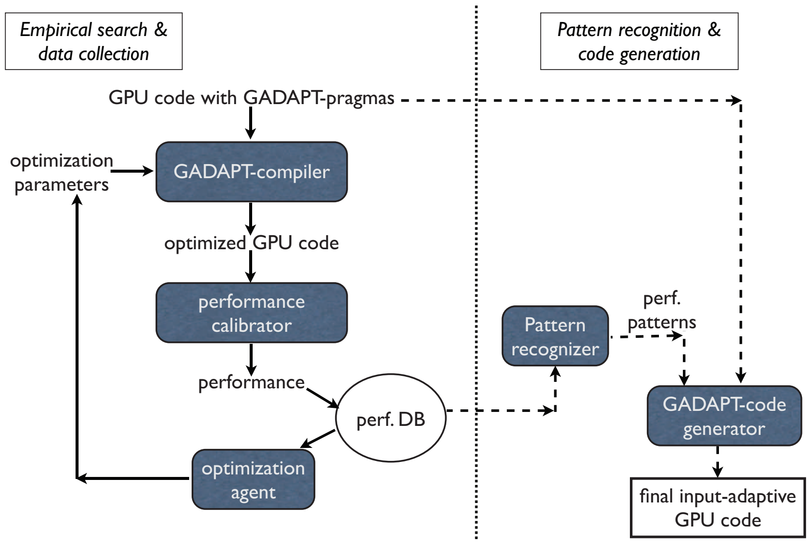


Figure 1. G-ADAPT: An adaptive optimization framework for GPU programs.

the second part addresses the input-sensitivity issue by recognizing the influence of inputs and making GPU program adaptive.

4.2 Stage 1: Heuristic-Based Empirical Search and Data Collection

The first stage is an iterative process. The inputs to the process include the original GPU program (with some special pragmas), along with a set of typical inputs to the GPU program.

In the iterative process, the adaptive framework, for each of the given inputs to the GPU program, automatically searches for the best values of optimization parameters that can maximize the program performance. The process results in a performance database, consisting of a set of $< \overrightarrow{input_j}, \overrightarrow{performance_j} >$ tuples.

Three components are involved in the iterative process. For a given input to the GPU program, in each iteration of the process, a compiler produces a new version of the program, a calibrator then measures the performance of the program on the given input, and the measuring result is used by an optimization agent to determine what version of the program the next iteration should try. When the system finds the best optimization values for that input, it stores the result into the performance database, and starts the iterations for another input.

There are several issues that need to be addressed to make the empirical search efficient and easily customizable to particular GPU applications. The issues include how to derive optimization space from the application, how to characterize program inputs, and how to prune the search space to accelerate the search. The three components in this stage work together to solve those problems as described as follows.

4.2.1 Optimization Pragmas and G-ADAPT Compiler

We classify the optimization parameters in GPU applications into three categories, corresponding to three different levels. In the first category is the execution configuration of the program—that is, the number of threads per block and the number of thread blocks for the execution of each GPU kernel. The second category includes the parameters that determine how the compiler transforms the program code. Loop optimization parameters, such as loop unrolling levels and size of loop tiles, belong to this category. The third category includes other implementation-level or algorithmic decisions, such as the selection of different algorithms for implementing a function. These parameters together form the space for the empirical search.

Different applications have different parameters to optimize; some parameters may be implicit in the program, and the ranges of some parameters may be difficult to be automatically determined because of aliases, pointers, and the entanglement among program data. Specially examples include the optimization space for complex non-affine loops and the parameters in the third category.

So even though compilers can automatically recognize some parameters in the first two categories, for automatic search to work, it is necessary to have a mechanism to easily expose all those kinds of parameters and their possible values for an arbitrary GPU application.

In this work, we employ a set of pragmas, named G-ADAPT pragmas, to support the synergy between programmers and compilers in revealing the optimization space. There are three types of pragmas. The first type is dedicated for the adjustment of scalar variable (or constant) values that control the execution configurations of the GPU application. The second type is for compiler optimizations. The third type is for implementation selection. The pragmas allow the inclusion of search hints, such as the important value ranges of a parameter and the suitable step size. For example, a pragma, "#pragma erange 64,512,2" above the statement "#define BLKSZ 256", means that the search range for the value of BLKSZ is from 64 to 512 with exponential (the first "e" in "erange") increases.

We develop a source-to-source compiler, named the G-ADAPT compiler, to construct and explore the optimization space. The G-ADAPT compiler is based on Cetus [14], a C compiler infrastructure developed by the group led by Eigenmann and Midkiff. With some extensions added to Cetus, the G-ADAPT compiler is able to support CUDA programs, the G-ADAPT pragmas, and a set of program transformations (e.g., redundant elimination, and various loop transformations.)

The G-ADAPT compiler has two-fold responsibilities. At the beginning of the empirical search, the compiler recognizes the optimization space through data flow analysis, loop analysis, and analysis on the pragmas in the GPU application. In each iteration of the empirical search, the compiler uses one set of parameter values in the search space to transform the application and produces one version of the application.

4.2.2 Performance Calibrator and Optimizing Agent

The performance calibrator invokes the CUDA compiler, *NVCC*, to produce an executable from the GPU program generated by the G-ADAPT compiler. It then runs the executable (on the current input) to measure the running time. After the run, it computes the occupancy of the executable on the GPU. The occupancy reflects the degree to which the executable exerts the computing power of the GPU. A higher occupancy is often desirable, but does not necessarily suggest higher performance. The occupancy calculation is based on the occupancy calculating spreadsheet [1] provided by NVIDIA. Besides hardware information, the calculation requires the information on the size of shared memory allocated in each thread, the number of registers used by each thread, and the thread block size. The calibrator obtains the information from the ".cubin" files of the GPU program and the execution of the executable [1].

The calibrator then stores the parameter values, along with the running time and occupancy, into the performance database. It checks whether the termination conditions (explained next) for the search on the

[1]The ".cubin" files are generated by *NVCC* with the usage of registers and shared memory per thread block exposed.

current input have been reached; if so, it stores the input, along with the best parameter values that have been found, into the performance database.

The responsibility of the optimization agent is to determine which point in the optimization space should be explored in the next iteration of the search process. The size of the optimization space can be very large. For K independent parameters with D_i denoting the number of possible values of the ith parameter, the optimization space is as large as $\prod_{i=1}^{K} D_i$. The exponential growth implies that for an application with many loops and implementation options, the space may become too large for the framework to enumerate all the points. The optimization agent uses hill climbing to accelerate the search. Let K be the number of parameters. The search starts with all the parameters having their minimum values. In each of the next K iterations, it increases one parameter by a step and keeps the others unchanged. After iteration $(K+1)$, it finds the best of the K parameter vectors that are just tried, and use it as the base for the next K iterations. This process continues. When one parameter reaches the maximum, it stops increasing. When all parameters reach their maximum values, the search stops.

This hill climbing search differs from the model-based prune proposed by Ryoo et al. [18]. Their approach is applicable when the program performance is not bounded by memory bandwidth; the method has shown more significant prune rate than our approach does. On the other hand, the hill climbing search is more generally applicable, making no assumptions on the GPU application.

4.3 Stage 2: Performance Pattern Recognition and Input-Adaptive Code Generation

After the first stage, the performance database contains a number of <*input, best parameter values*> tuples, from which, the pattern recognizer learns the relation between program inputs and the optimization parameters. A number of statistical learning techniques can be used in the learning process. In this work, we select Regression Trees [11] for its simplicity and good interpretability. Regression Trees is a divide-and-conquer learning approach, which divides the input space into local regions with each region having a regular pattern. In the resulting tree, each non-leaf node asks a question on the input features, and each leaf node represents a local region. The question asked in a non-leaf node is automatically selected in the light of entropy reduction—that is, the increase of class label purity of the data set after the data are split on their answers to the question in that node. For each leaf node, we apply Least Mean Squares (LMS) to the data that fall into the node to produce a linear model that fits the data.

To capitalize on the learned patterns, we need to integrate them into the GPU application. If there were just-in-time compiler (JIT) support, the integration could happen during runtime implicitly: The JIT compiles the program functions using the parameters predicted as the best for the program input. Without JIT, the integration can occur either through a linker, which links the appropriate versions of object files into an executable before every execution of the application, or an execution wrapper, which every time selects the appropriate version of executables to run. In our experiments, we use the wrapper solution because it has no linking overhead, and the programs in our experiments need only few versions of executables. The G-ADAPT compiler, along with the CUDA compiler, produces one executable for each parameter vector that is considered as the best for some training inputs in the performance database. When the application is launched with an arbitrary input, the version selector in the wrapper uses the constructed regression trees to quickly determine the right executable based on the input and then runs the program.

5 Evaluation

We use seven benchmarks to test the effectiveness of the optimization framework, as listed in Table 1. The third column of the table shows the number of different inputs we used for each benchmark. Most of

Table 1. Benchmarks

Benchmark	Description	Inputs#	Pred Acc
convolution	convolution filter of a 2D signal	10	100%
matrixMul	dense matrix multiplication	9	100%
mvMul	dense matrix-vector multiplication	15	93.3%
reduction	sum of array	15	93.3%
scalarProd	scalar products of vector pairs	7	100%
transpose	matrix transpose	18	100%
transpose-co	matrix transpose with coalescing memory references	18	100%

the programs are from NVIDIA [1]; the program, *mvMul*, is a matrix vector multiplication program from Fujimoto [9], which has shown significant performance improvement over the implementation in NVIDIA CUBLAS [1].

The type of GPU we use is NVIDIA GeForce 8800 GT. It contains 512M global memory, 14 multiprocessors, 112 cores, with clock rates set at 1.51 GHz. Each multiprocessor has 16KB shared memory and 8192 registers. Every GPU co-runs with 2 Intel Xeon processors (3.6GHz) on a machine with SUSE Linux 2.6.22 installed.

Before presenting the detailed results on each benchmark, we first briefly summarize the results as follows. The best configurations of three out of the seven programs change with their inputs. For all the programs, the G-ADAPT is able to learn the relation between inputs and optimization parameters, producing over 93% prediction accuracy for the best optimization decisions. The prediction yields several times of speedup compared the running times of the original programs. In the following, we present the results of the input-sensitive programs first, followed by the results of other programs.

5.1 Matrix-Vector Multiplication

The program, *mvMul*, computes the product between a dense matrix and a vector. It outperforms previous implementation because of the adoption of a new algorithm along with an effective use of texture memory.

The parameters of this program include the size of a thread block, and the loop unrolling factors in the kernel function. Figure 2 shows the performance of the program on two example inputs when different configurations are used. The different parameter values cause up to 2.5 times performance difference. The block size has more significant influence than the unrolling levels. Moreover, the results clearly show the influence of program inputs on the optimal parameter values. The best block size for the first input turns out to be the worst for the second input, causing 2.4 times slowdown than its best run. One of the reasons for the less effect of loop unrolling is because the innermost loop can have iterations of at most a quarter of the width of a thread block. So, there is little room for adjustment.

Figure 3 (a) reports the best block size for each of the 15 inputs. The block size used in the original program is 256, which works the best for the 4 inputs on the left. For the other inputs, the best block size is 64. Figure 3 (b) plots the speedups of the program when it uses the optimizations predicted by the G-ADAPT framework. The baseline is the running times of the original program. The trend is that as the height of the input matrix becomes larger than its width, the speedup becomes larger. The reason why large blocks work poorly for thin matrices is that each time, a block is in charge of a group of rows, and in thin matrices, each thread has little work to do. This benchmark demonstrates that the shape of the input matrix is critical for the optimization decisions.

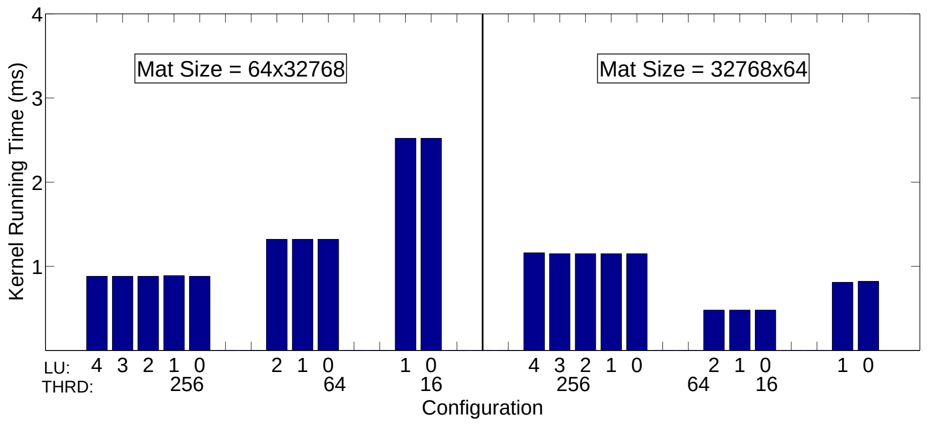
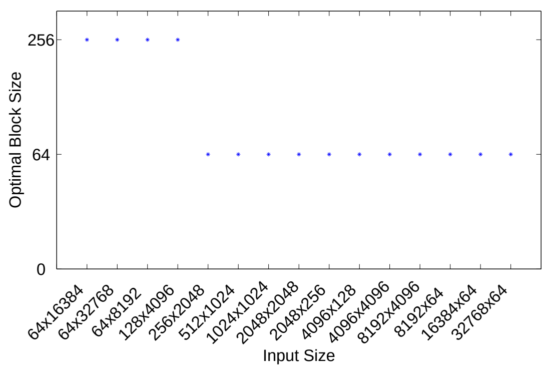
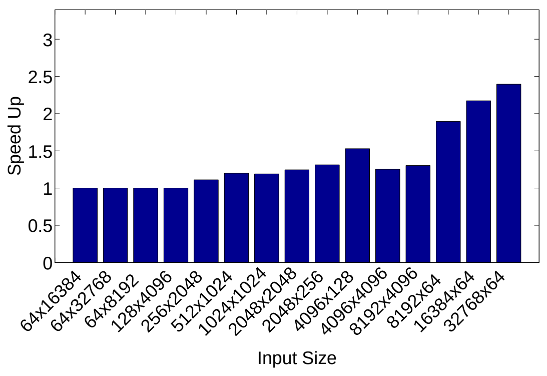


Figure 2. Performance of matrix-vector multiplication on two inputs when different optimization decisions are used. LU: loop unrolling levels; THRD: number of threads per block. The maximum unrolling level can only be $\sqrt{THRD}/4$.



(a) Best thread-block size

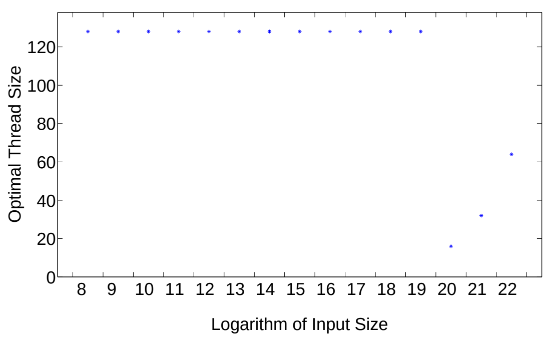


(c) Speedup brought by G-ADAPT

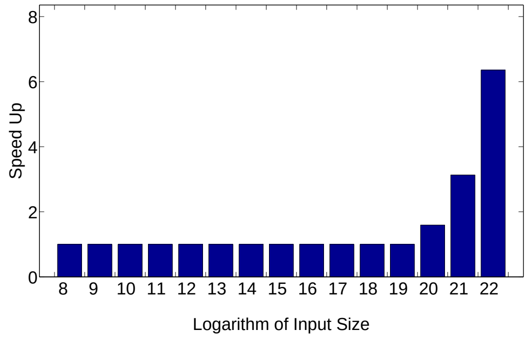
Figure 3. The best values of the optimization parameters of *mvMul* are input-sensitive. G-ADAPT addresses the influence and produces significant speedup compared to the original program.

5.2 Parallel Reduction

The program, *reduction*, performs sum operations on an array of integers. It represents one kind of common computation in parallel computing, reducing a series of values into a single value. The code we use is the one NVIDIA has used as typical example of manual optimizations on GPU programs [10]. A sequence of optimizations have been applied to the program with careful manual tuning. The loop in the kernel function has been completely unrolled using template functions. Given all the manual optimizations, our experiment concentrates on a single parameter, the number of threads per block.

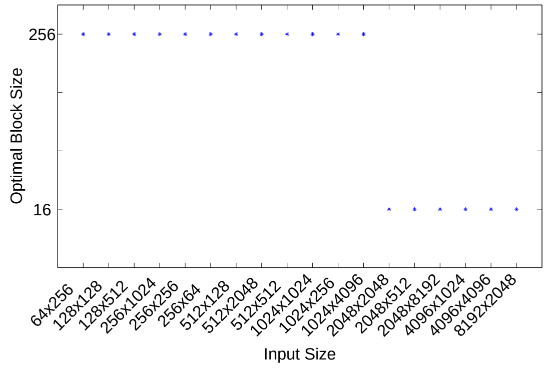


(a) Best thread-block size

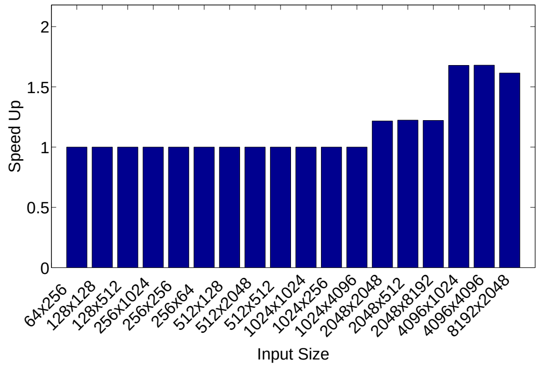


(c) Speedup brought by G-ADAPT

Figure 4. Experimental results on *reduction*.



(a) Best thread-block size



(b) Speedup brought by G-ADAPT

Figure 5. Experimental results on *transpose*.

The default setting is 128 threads per block. However, in our experiments, we find that the setting works best only for medium size arrays. For large sizes, smaller blocks work much better. Figure 4 shows the results. When the size of the array increases from 2^{19} to 2^{20}, the best block size turns from 128 to 16. Whereas, as the size of array increases more, the best block size increases linearly.

The Regressions Trees method in G-ADAPT produces a tree with two nodes corresponding to the arrays that are smaller and larger than $(2^{19} + 2^{18})$. It fits each nodes with a linear model and produces 93.3% prediction accuracy. Figure 4 (b) shows the speedup of the optimized program compared to the performance of the original program.

5.3 Matrix Transpose

There are two versions of matrix transpose in the NVIDIA SDK. One uses memory coalescing and the other one does not; we denote them as *transpose-co* and *transpose* respectively. In both versions, the kernel function contains no loops, and the key optimization parameter is the block size. Figure 5 shows the results of *transpose*. For matrices of medium sizes, the best block size is 256, the same as the default setting in the original program. Whereas, the best size becomes 16 when the matrix size increases to over 4 million elements. The speedup becomes more significant as the matrix become larger.

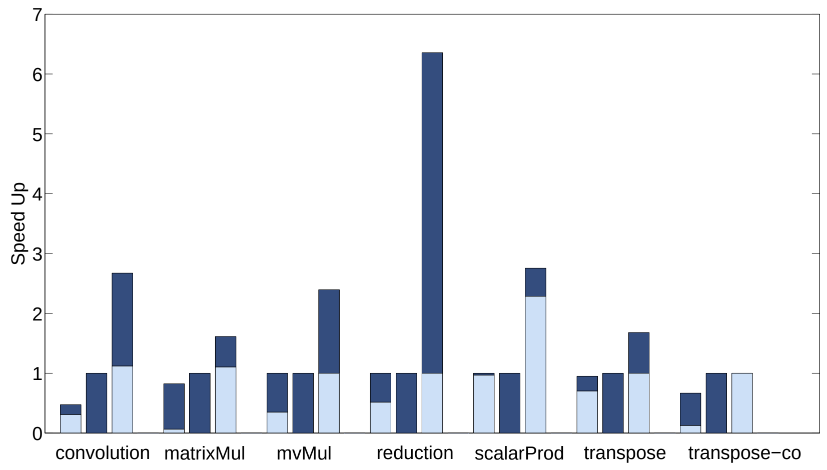


Figure 6. The ranges of speedup brought by different optimization decisions. For each program, the leftmost bar shows the range of speedup (less than 1 means slowdown) if the worst decision is taken. The rightmost bar shows the range of speedup when the G-ADAPT's prediction is used. The middle bar is the baseline.

In contrast, the coalesced version, *transpose-co*, is not input-sensitive. The best block size is always 256. This version differs from *transpose* mainly in memory accesses. In the kernel function of *mtco*, the references to the global memory are staged. The data are first brought into shared memory in a coalesced manner before the computation. Furthermore, the array is padded to reduce bank conflicts in the shared memory. The changes in memory reference patterns remove the input-sensitivity. When the block size is 16, the program achieves 100% occupancy on the multiprocessors, and thus exhibits the best performance.

5.4 Other Benchmarks and Overall Results

The best values of the parameters in the other 3 benchmarks, *matMulGPU, convolution, scalarProd*, show no sensitivity to their inputs. Besides the parameters for loop optimizations, the program *matMulGPU* has a parameter controlling the size of thread blocks, the program *convolution* has 3 parameters controlling the tile size and the number of columns, and the program *scalarProd* has 2 parameters controlling the dimensions of the grid and the dimensions of a thread block. The G-ADAPT system successfully finds the best parameter values for all the 3 programs.

We apply the predictions of G-ADAPT to the programs to measure the effectiveness in performance improvement. The prediction is based on leave-one-out cross validation [11], which is a typical practice in statistical learning to estimate the error of a predictive model in real uses. For each input, we use all the other inputs as training inputs to build regression trees, and then apply the trees to the left-out input to predict the corresponding best optimization decisions. The average prediction accuracies are shown in the right-most column in Table 1. For input-insensitive programs, the prediction is simple. For two of the input-sensitive programs, there are 6.7% prediction errors. For transpose, the prediction accuracy is 100%. These results demonstrate the effectiveness of the Regression Trees method in modeling the relation between inputs and optimization decisions.

Figure 6 summarizes the speedup ranges brought by G-ADAPT on all the 7 benchmarks. The baseline is the running times of the original programs when their default settings are used. The first bar in a benchmark corresponds to the worst configuration encountered in the explored optimization space, which reflects the risk of a careless configuration or transformation. The third bar shows the effectiveness of G-ADAPT. Among all 4 input-insensitive programs, only the default setting in *transpose-co* happens to the same as the one G-ADAPT finds. For the other three, the optimization results from G-ADAPT outperform the original program by 1.5 to 2.7. The reduction program shows the most significant speedup, as much as 6.4.

6 Related Work

The studies closest to this work are the recent explorations by Ryoo et al. [18], and Baskaran et al. [3]. Ryoo and his colleagues have defined *efficiency* and *utilization* models for GPU computing, and demonstrated the effectiveness of the models in pruning of the optimization space. Our study complements their technique in that the influence from program inputs is a dimension omitted in their work. Furthermore, the previous work conducts transformations manually, whereas, we develop a compiler framework with optimization pragmas for automatic transformations. The prune method in our tool complements the previous models in that it relaxes some assumptions made by previous work, such as the memory bandwidth is not the bottleneck on performance. On the other hand, the previous models may be very helpful in the cases when the assumptions hold.

In the study by Baskaran et al. [3], the authors focus on the optimization of affine loops in GPU applications. They develop an approach to improving global memory accesses and use model-driven empirical search to determine optimal parameters for loop unrolling and tiling. Our work is complementary to their technique on two aspects. First, our optimizations are input adaptive, whereas, the influence of program inputs is a missing factor in the previous study. Second, our tool can be applied to not only optimization of affine loops, but also other factors that affect the performance of GPU applications, such as the size of thread block size and implementation-level decisions. On the other hand, the transformations developed in the previous work can strengthen the effectiveness of our tool. An integration of them into the tool is our on-going work.

On traditional CPU architecture, there has been many studies on empirical-search based optimizations. Many of the explorations are for the development of efficient numerical libraries or kernels, such as ATLAS [23], PHiPAC [4], SPARSITY [12], SPIRAL [21], FFTW [8], STAPL [20]. Our work is enlightened by those explorations, but focuses on a single-chip massively parallel architecture, on which, the optimizations dramatically differ from those on the previous CPU architecture. Furthermore, the targets of this work are general GPU applications, rather than a certain set of kernels. The variety in the applications further complicates input characterization and the construction of cross-input predictive models.

The adaptation to different program inputs in this work shares some common theme with code specialization, such as procedure cloning [5], the incremental run-time specialization [15], the specialization of libraries in Telescoping Languages [13]. In addition, dynamic optimizations (e.g. [2, 6, 22]) may tailor a program to their inputs by runtime code generation. However, unlike the previous work, the adaptation in our work concentrates on the whole-program level, rather than on the procedure level. We believe that a runtime optimizer may be able to better capitalize the cross-input adaptive models, but it may introduce extra runtime overhead.

7 Conclusion

This paper reports our exploration of the influence of program inputs on GPU program optimizations. It shows that for many GPU applications, their best optimizations are different for different inputs. It presents a compiler-based adaptive framework, G-ADAPT, which is able to extract optimization space from program code and a set of pragmas, and automatically search for the best optimizations for an GPU application on different inputs. With the use of Regression Trees, G-ADAPT produces cross-input predictive models from the search results. The models are able to predict the best optimizations from the input given to the GPU application, and thus enable cross-input adaptive optimizations. Experiments show significant performance improvement generated by the optimizations, demonstrating the promise of the framework as an automatic tool for resolving the productivity bottleneck in the development of efficient GPU programs.

References

[1] NVIDIA CUDA. `http://www.nvidia.com/cuda`.

[2] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2002.

[3] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for gpgpus. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234, 2008.

[4] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: A portable, high-performance, ansi c coding methodology. In *Proceedings of the ACM International Conference on Supercomputing*, 1997.

[5] K. D. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Computer Languages*, pages 96–105, 1992.

[6] P. Diniz and M. Rinard. Dynamic feedback: an effective technique for adaptive computing. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Las Vegas, May 1997.

[7] Y. Dotsenko, N. K. Govindaraju, P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 205–213, 2008.

[8] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005.

[9] N. Fujimoto. Faster matrix-vector multiplication on geforce 8800gtx. In *Proceedings of the Workshop on Large-Scale Parallel Processing (co-located with IPDPS)*, 2008.

[10] M. Harris. High performance computing with cuda. In *Tutorial in IEEE SuperComputing*, 2007.

[11] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.

[12] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004.

[13] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(2):387–408, 2005. This provides a current overview of the entire Telescoping Languages Project.

[14] S. Lee, T. Johnson, and R. Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. 2003.

[15] R. Marlet, C. Consel, and P. Boinot. Efficient incremental run-time specialization for free. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999.

[16] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka. Bandwidth intensive 3-d FFT kernel for GPUs using CUDA. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, 2008.

[17] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.

[18] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Program optimization space pruning for a multithreaded gpu. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, 2008.

[19] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on gpus through software-managed cache. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 309–318, 2008.

[20] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in stapl. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

[21] M. uschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. Spiral: code generation for dsp transforms. *Proceedings of the IEEE*, 93(2), 2005.

[22] M. Voss and R. Eigenmann. High-level adaptive program optimization with ADAPT. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, Snowbird, Utah, June 2001.

[23] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2), 2001.