A Cross-Input Adaptive Framework for GPU Program Optimizations

Yixun Liu Eddy Z. Zhang Xipeng Shen
Computer Science Department
College of William and Mary
{enjoywm,eddy,xshen}@cs.wm.edu

***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, due 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.

***Index Terms*—GPU, Program Optimizations, Empirical Search, CUDA, G-ADAPT, Cross-input Adaptation**

I. 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 computation beyond the traditional uses in graphics rendering [3], [7], [18], [20]–[22].

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) [17] 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 development of GPU programs.

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 make manual optimizations time consuming and difficult to attain the optimal, and at the same time, form great hurdles to automatic optimizations as well.

On the other hand, optimizations are 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], [20], [21].

Several recent studies have tried to tackle the problem through empirical search-based approaches. Ryoo and his colleagues [21] have defined *efficiency* and *utilization* models for GPU programs 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 then, tailor the decisions for each program input.

More specifically, this work makes three major contributions. *First*, we develop a source-to-source compiler-based framework, G-ADAPT, for empirically searching for the best optimizations for GPU applications. The framework is distinctive in that it conducts program transformations and optimization-space search in a fully automatic fashion, and meanwhile, offers a set of pragmas for programmers to easily incorporate their knowledge into the empirical search process. *Second*, this work examines 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 do not matter to the optimizations as long as the sub-tasks remain similar. Our experiments show that, although many GPU kernels conform that intuition, some GPU programs exhibit strong input-sensitivity due to their computation patterns and the interplay with optimization parameters. *Finally*, based on the exposed input sensitivity, we construct a cross-input predictor by employing statistical learning (Regression Trees in particular) to make G-ADAPT automatically tailor 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 most of the 7 GPU applications with over 93% accuracy. The adaptive optimization improves the program performance by as much as 2.8 times in comparison with manually optimized versions.

We organize the paper as follows. Section II provides some background on GPU and its programming model. Section III discusses the challenges in GPU program optimizations. Section IV describes G-ADAPT as our solution to those challenges. Section V reports evaluation of the framework. Section VI discusses the training overhead and some other complexities of G-ADAPT. After an overview of some related work in Section VII, we conclude the paper with a brief summary.

II. 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 and 512 MB 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 1.51 GHz. 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 16 KB shared memory. Unlike cache, the shared memory has to be managed explicitly in each GPU application.

The off-chip memory includes a 512 MB 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 a set of abstractions. The programming interface of CUDA is ANSI C with certain extensions. A GPU application written in CUDA is composed of CPU code and GPU kernels. CUDA abstracts the execution of a GPU kernel as multithreaded SIMD computation. The threads are grouped into many warps with 32 threads in each. Those warps are organized into a number 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 an 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. Threads in a block may communicate through shared memory and be synchronized by a *__syncthreads* primitive. But communications between threads that belong to different thread blocks have to use off-chip global memory; the communications are hence slow and inflexible.

III. Challenges in the Optimization of GPU Programs

Although CUDA simplifies GPU programming, it reduces little if any difficulty in optimizing GPU applications; to some degree, the added abstractions even complicate the optimization as they make performance prediction still harder.

a) 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 help keep the SM busy when the threads in one block are stalled for synchronization. Example 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 (e.g., texture memory), the reduction of bank conflicts in shared memory, and coalesced memory references (i.e., when threads in a warp reference a sequence of contiguous memory addresses at the same time.)

Optimizations to reduce the number of dynamic instructions include 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], [21].

b) 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 [21]. The undisclosed details of the CUDA compiler and other abstractions add further unpredictability. *Second*, different optimizations often affect each other. Loop unrolling, for example, removes some dynamic instructions and exposes certain opportunities for the instruction scheduler to exploit; but it also increases register pressure for 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. In GeForce 8800 GT, for instance, 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 16 KB 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 strategy for determining the best optimizations is through empirical search, whereby the optimizer searches for the best optimization parameters by running the GPU application many times, each time with different optimizations applied. Three obstacles must be removed before this solution becomes practical. First, a compiler is needed for abstracting out the optimization space and transforming the program accordingly. Second, effective space prunes are necessary for the search efficiency, especially when the optimization space is large. Finally, the optimizer must be able to handle the influence of program inputs. Our study (Section V) 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 is desirable to detect the input-sensitivity and make the optimization cross-input adaptive.

IV. Adaptive Optimization Framework

G-ADAPT is our solution to the challenges in GPU program optimization. It is a cross-input adaptive framework, unifying source-to-source compilation, performance modeling, and pattern recognition. This section first gives an overview of the framework, and then elaborates on every component in the framework.

A. Overview

Figure 1 shows the structure of *G-ADAPT*. Its two parts separated by the dot vertical line correspond to two stages of the optimization. The task of the first stage, shown as the left part 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 input features, 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 corresponding suitable optimization decisions. G-ADAPT then transforms the original GPU code into a program that is able to automatically adapt to an arbitrary input.

The first part uses empirical search to overcome the difficulty in modeling GPU program performance; the second part addresses the input-sensitivity issue by recognizing the influence of inputs and making GPU program adaptive.

B. Stage 1: Heuristic-Based Empirical Search and Data Collection

The first stage is an iterative process. The inputs to the process include a given GPU application (with some pragmas inserted) with a set of typical inputs.

In the iterative process, the adaptive framework, for each of the given inputs to the GPU application, automatically searches for the best values of optimization parameters that can maximize the performance of the application. The process results in a performance database, consisting of a set of $<$*input, best parameter values*$>$ tuples.

Three components are involved in this iterative process. For a given input to the GPU program, in each iteration, a compiler produces a new version of the application, a calibrator then measures the performance of the program on the given input, and the measured 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 values into the performance database, and starts the iterations for another input.

Several issues need to be addressed to make the empirical search efficient and widely applicable. 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. In the following, we describe how the 3 components in the first stage of G-ADAPT work together to address these issues.

1) Optimization Pragmas and G-ADAPT Compiler: We classify the optimization parameters in GPU applications into three categories, corresponding to three different optimization levels. In the first category are execution configurations 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, such as loop unrolling levels and size of loop tiles. The third category includes other implementation-level or algorithmic decisions, such as the selection of different algorithms for implementing a function. These parameters together constitute the space for the empirical search.

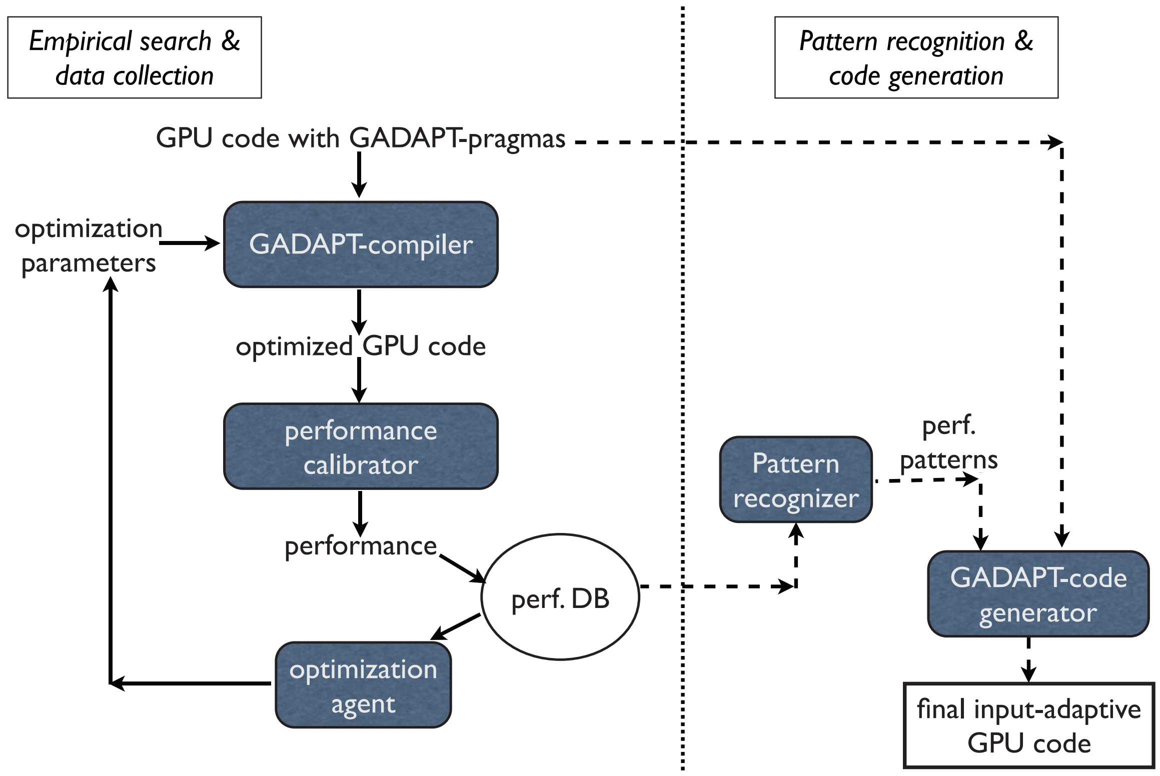


Fig. 1. G-ADAPT: An adaptive optimization framework for GPU programs.

Different applications have different parameters to optimize; some parameters may be implicit in a program, and the ranges of some parameters may be difficult to be automatically determined because of aliases, pointers, and the entanglement among program data.

So even though compilers may automatically recognize some parameters in the first two categories, for automatic search to work generally, 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 with base 2.

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.

2) Performance Calibrator and Optimization 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 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 $\prod_{i=1}^{K} D_i$. It 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. [21]. 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.

C. Stage 2: Pattern Recognition and Cross-Input Adaptation

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. It divides the input space into local regions with each region having a regular pattern. In the resulting tree, every non-leaf node contains a question on the input features, and every leaf node corresponds to a region in the input space. The question contained in a non-leaf node is automatically selected in the light of entropy reduction, defined as the increase of the purity of the data set after the data are split by that question. We then apply Least Mean Squares (LMS) to the data that fall into each leaf node to produce the final predictive models.

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.

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

V. Evaluation

We use seven benchmarks to test the effectiveness of the optimization framework, as listed in Table I. Most of the programs are from NVIDIA SDK [1]. The program, *mvMul*, is a matrix vector multiplication program from Fujimoto [9]. It is an efficient implementation, outperforming the NVIDIA CUBLAS [1] version significantly, thanks to its adoption of a new algorithm along with an effective use of texture memory [9].

We emphasize that the programs we use have all been manually tuned by the developers. The *reduction* program, for instance, has gone through seven optimizations, respectively on the algorithm, locality, branch divergence, loop unrolling and so on. NVIDIA has used it as a typical example to demonstrate manual optimizations on GPU programs. The sequence of optimizations have accelerated the program by as much as a factor of 30 [10].

The third column of the table shows the number of different inputs we have used for each benchmark. We create those inputs based on our understanding to the applications, with an attempt to cover a wide range of the input space.

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

Before presenting the detailed results on each benchmark, we briefly summarize the results. 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 (except 80% for one program) for the best optimization decisions. The prediction yields several times of speedup compared the running times of the original programs. In the following subsections, we present the results of the input-sensitive programs first, followed by the results of other programs.

TABLE I
BENCHMARKS

Benchmark	Description	Num of Inputs	Prediction acc	Training iterations	Training time (s)
convolution	convolution filter of a 2D signal	10	100%	200	2825
matrixMul	dense matrix multiplication	9	100%	196	2539
mvMul	dense matrix-vector multiplication	15	93.3%	124	124
reduction	sum of array	15	80%	75	29
scalarProd	scalar products of vector pairs	7	100%	93	237
transpose	matrix transpose	18	100%	54	1639
transpose-co	matrix transpose with coalescing memory references	18	100%	54	631

A. Matrix-Vector Multiplication

The program, *mvMul*, computes the product between a dense matrix and a vector. 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 negligible effect of loop unrolling is that there is little room for adjustment: the innermost loop can have iterations of at most a quarter of the width of a thread block.

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 and thus results in low occupancy on GPU processors. This benchmark demonstrates that the shape of the input matrix is critical for the optimization decisions.

B. 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. Given that many optimizations (e.g., loop unrolling) have been manually applied in the development of the original program, our experiment concentrates on a single parameter, the number of threads per block.

The default setting is 128 threads per block. That setting turns out to be the best for most inputs, except two inputs whose array sizes are 2^{19} and 2^{20}, in which case, the best block size is 64. Even on these two inputs, the default setting works virtually similar to the optimal, with only 3% performance difference.

C. 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.

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.

D. 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 these 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 fourth column in Table I. For input-insensitive programs, the prediction is simple. For the input-sensitive programs, the prediction accuracy is 80% for *reduction*, 93.3% for *mvMul*, and 100% for *transpose*.

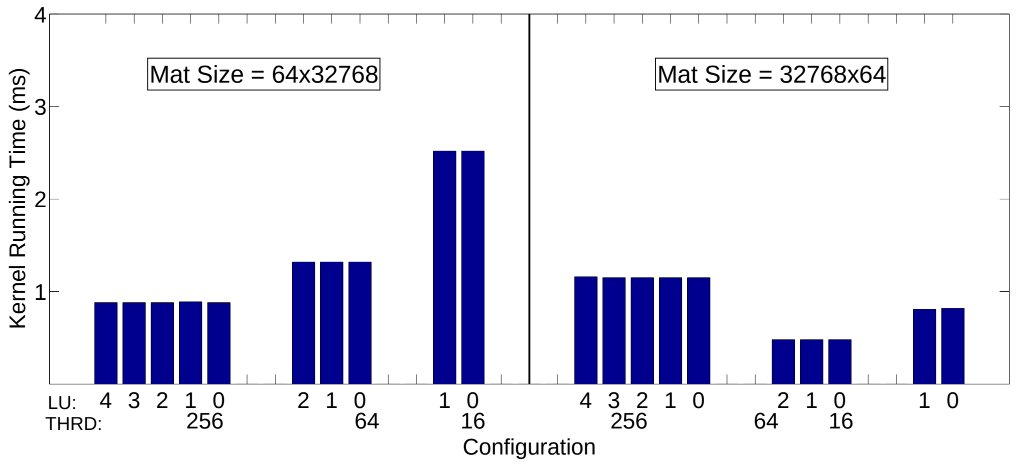
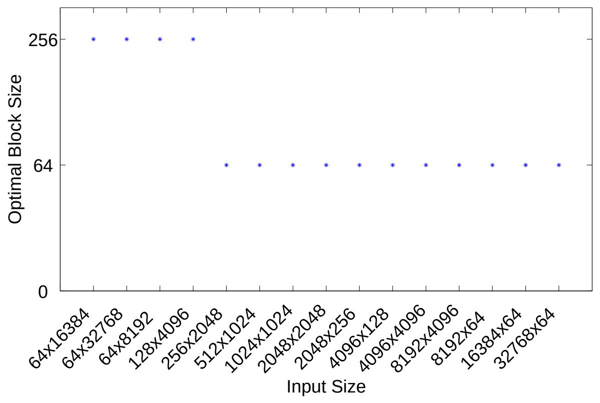
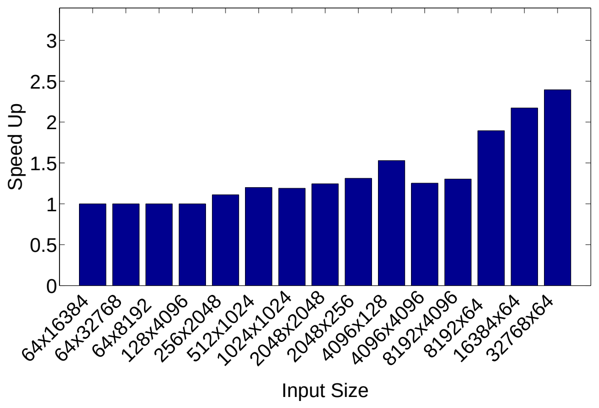


Fig. 2. Performance of matrix-vector multiplication on two inputs when different optimization decisions are used. LU: loop unrolling levels; THRD: number of threads per block. The maximum unrolling level can only be $\sqrt{THRD}/4$.

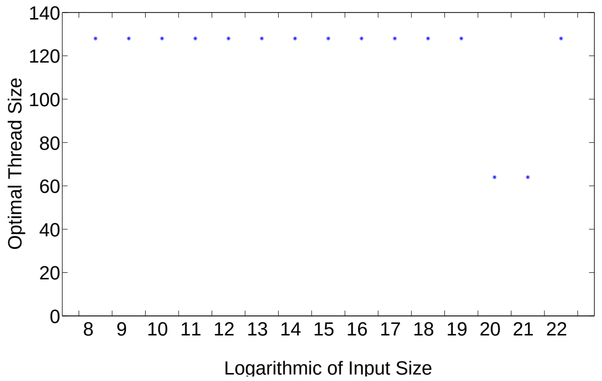


(a) Best thread-block size

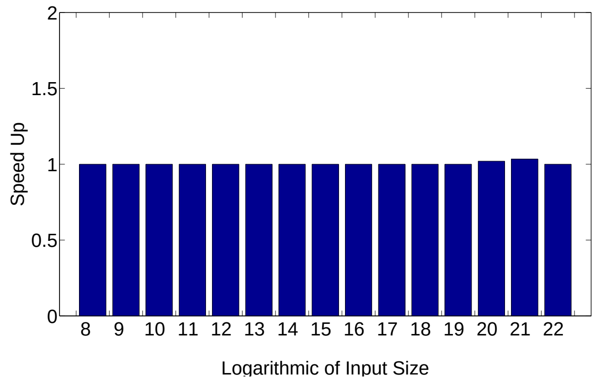


(c) Speedup brought by G-ADAPT

Fig. 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.

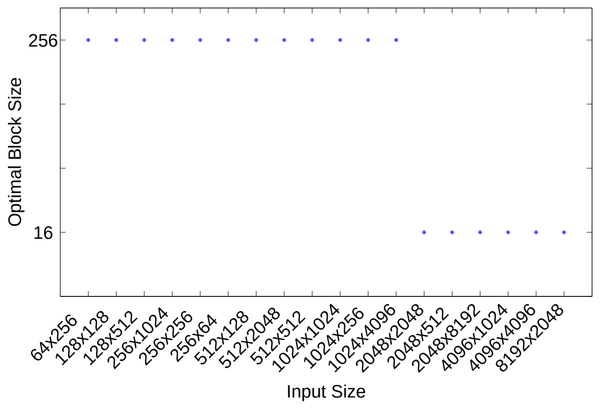


(a) Best thread-block size

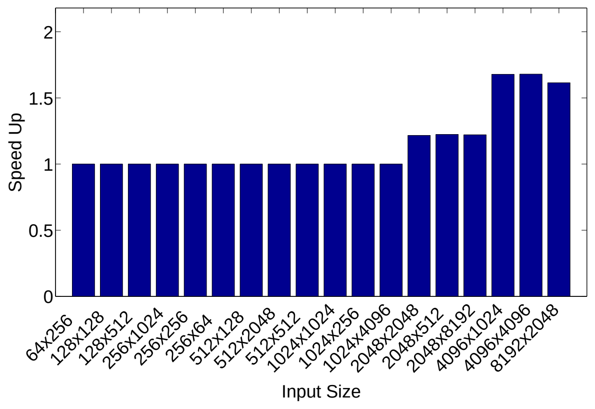


(c) Speedup brought by G-ADAPT

Fig. 4. Experimental results on *reduction*.



(a) Best thread-block size



(b) Speedup brought by G-ADAPT

Fig. 5. Experimental results on *transpose*.

These results demonstrate the effectiveness of the Regression Trees method in modeling the relation between inputs and optimization decisions.

On different inputs to an application, the G-ADAPT yields different speedups. Figure 6 summarizes the ranges of speedup brought by G-ADAPT on the 7 GPU programs. The baseline is the running times of the original GPU programs. For each program, the left 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 right bar shows the effectiveness of G-ADAPT. Among all programs, only the default settings in *transpose-co* and *reduction* happen to be (almost) the same as the one G-ADAPT finds. The 1.5 to 2.8 times of speedup on other programs demonstrate the effectiveness of input-adaptive optimizations enabled by G-ADAPT.

VI. Discussions

In this section, we first present the training overhead of G-ADAPT and then discuss some complexities in applying G-ADAPT for large applications.

The right-most two columns in Table I reveal the training overhead of G-ADAPT on the seven benchmarks. The total numbers of iterations range from 54 to 200, and the total training time spans from 29 seconds to 47 minutes. The time is determined by the number of training inputs, the dimensions of the search space, and the size of the inputs. The program, *convolution*, happens to run for a long time on some of its training inputs, resulting in the longest training time.

It is worth noting that one complexity, input characterization, happens to be simple in our experiments. Input characterization is to determine the important features of program inputs. In our experiments, the inputs to the programs are just several numbers, indicating the sizes of the input signal, matrix, array, or vector, which naturally capture the important characteristics of the input data sets. However, for large complex GPU applications, the input characterization may need special treatment. One option is to develop some input characterization procedures and link them with G-ADAPT. A recent study [15] proposes an extensible input characterization language, XICL, to ease the efforts. Detailed studies remain to be our future work.

VII. Related Work

The studies closest to this work are the recent explorations by Ryoo et al. [21], 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 work well 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 may be worthwhile.

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

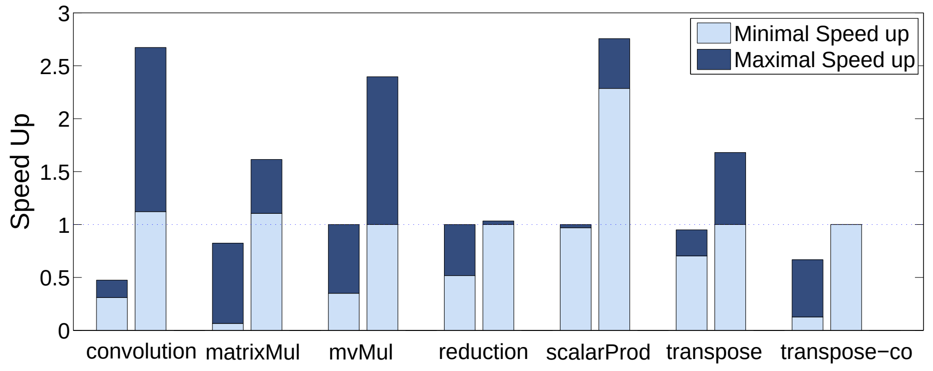


Fig. 6. The ranges of speedup brought by different optimization decisions. For each program, the left bar shows the range of speedup (less than 1 means slowdown) if the worst decision is taken. The right bar shows the range of speedup when the G-ADAPT's prediction is used.

libraries or kernels, such as ATLAS [25], PHiPAC [4], SPARSITY [12], SPIRAL [19], FFTW [8], STAPL [23]. 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 [16], the specialization of libraries in Telescoping Languages [13]. In addition, dynamic optimizations [2], [6], [15], [24] may tailor a program to their inputs by runtime code generation.

VIII. Conclusion

This paper reports our exploration of the influence of program inputs on GPU program optimizations. It shows that for some 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 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 can 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.

Acknowledgments

We thank John Owens for his suggestions to the draft of this paper. We owe a debt to the anonymous reviewers for their helpful comments. The GPU devices used in this study were donated by NVIDIA. This material is based upon work supported by the National Science Foundation under Grant No. CSR-0720499 and CCF-0811791. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

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 Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 111–129, 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*, pages 340–347, 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*, pages 71–84, 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):216–231, 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)*, pages 1–8, 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] Eun-Jin Im, Katherine Yelick, and Richard 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.

[14] S. Lee, T. Johnson, and R. Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *In Proceedings of the 16th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 539–553, 2003.

[15] F. Mao and X. Shen. Cross-input learning and discriminative prediction in evolvable virtual machine. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2009.

[16] 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*, pages 281–292, Atlanta, GA, May 1999.

[17] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, pages 40–53, March/ April 2008.

[18] 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.

[19] M. Puschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson, and N. Rizzolo. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005.

[20] 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.

[21] 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.

[22] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D. Owens. Efficient computation of sum-products on GPUs through software-managed cache. In *Proceedings of the 22nd ACM International Conference on Supercomputing*, pages 309–318, June 2008.

[23] 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 the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 277–288, 2005.

[24] M. Voss and R. Eigenmann. High-level adaptive program optimization with ADAPT. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, pages 93–102, Snowbird, Utah, June 2001.

[25] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.