Input Sensitivity of GPU Program Optimizations

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

***Abstract*—Graphic Processing Units (GPU) have become increasingly adopted for the enhancement of computing throughput. 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. Many recent efforts have been employing empirical search-based auto-tuners to tackle the problem, but few of them have concentrated on the influence of program inputs on the optimizations. In this paper, based on a set of CUDA and OpenCL kernels, we report some evidences on the importance for auto-tuners to adapt to program input changes, and present a framework, 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. G-ADAPT+ is based on source-to-source compilers, specifically, Cetus and ROSE. It supports the optimizations of both CUDA and OpenCL programs.**

I. Introduction

Graphic Processing Units (GPU) have drawn increasing adoptions for high performance computing. The development of programming models (e.g., CUDA and OpenCL) have simplified the creation of a GPU program, but not much the optimizations.

Developing an *efficient* GPU program remains a challenge to programmers, as well as compilers and performance auto-tuners. 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. But optimizing GPU programs are facing three-fold special difficulties. The *first* is the complexity in GPU architecture. On an NVIDIA GeForce 8800 GT, for example, there are hundreds of cores, several 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. *Second*, an optimization often has multiple effects, and the optimizations on different parameters often strongly affect each other. The *final* and also the least-understood challenge is that some GPU applications are strongly sensitive to their inputs (including command-line arguments, input file content, and other data coming outside the program). The best optimization decisions for 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.

Some prior studies have tried to tackle the problem through performance auto-tuners (or empirical search) [1], [3], [7]. But few of them have systematically considered the influence of program inputs on the optimization decisions.

In this paper, based on our prior study [6] and some recent experiments, we report some evidences on the important impact of inputs on GPU program optimizations. The measurements are on both CUDA and OpenCL programs. We present a framework, called G-ADAPT+, that efficiently discovers near-optimal decisions for GPU program optimizations, and then, tailors the decisions for each program input. G-ADAPT+ is based on source-to-source compilers: Cetus for CUDA and ROSE for OpenCL. 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. 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 customize optimizations to program inputs. Experiments demonstrate the promise of such adaptive optimizations frameworks in overcoming the input sensitivity of GPU program optimizations, leading to speedups of a factor of integers.

II. Challenges in the Optimization of GPU Programs

For consistency, we use CUDA terminology throughout the entire paper, even in the discussions of OpenCL programs. A GPU usually contains a number of Streaming Multiprocessors (SM), containing hundreds of cores for computing. When a GPU kernel is launched, it is often executed by thousands of GPU threads. These threads are organized into a number of *thread blocks*. The size and dimensions of a thread block often affect the degree in which the massively parallel GPU hardware computing units are utilized. A GPU memory hierarchy is complex. There are three types of off-chip memory (global memory, constant memory, texture memory), and at least three types of on-chip storage (shared memory, cache, registers). A good usage of the computing units and memory systems is essential for exerting the full power of GPU.

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 GPU compilers have many of these techniques included, researchers have seen great potential to adjust some of those optimizations, such as the levels of loop unrolling.

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

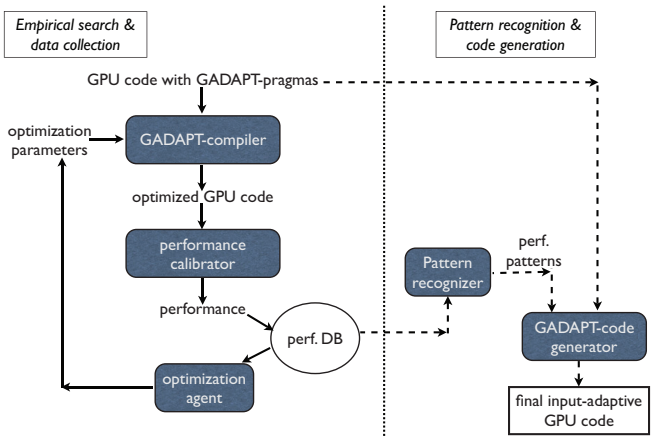


Fig. 1. G-ADAPT: An adaptive optimization framework for GPU programs.

on the performance of the GPU application. The effects are often non-linear as what Ryoo et al. have shown [7]. 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 IV) 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.

III. 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. It extends the capability of its predecessor, G-ADAPT [6], by adding support to OpenCL programs (in addition to CUDA support).

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.

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 employ source-to-source compilers to construct and explore the optimization space. The G-ADAPT+ compiler has two modes. One is for CUDA program optimizations. It is based on Cetus [5], a C compiler developed by the group led by Eigenmann and Midkiff. With some extensions added to Cetus, the 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 second mode of the G-ADAPT+ compiler is for OpenCL program optimizations. It is based on ROSE [8], a C++ compiler developed by a group at Lawrence Livermore National Laboratory, led by Quinlan. The reason for using two compilers is historical. We started with Cetus, but had to switch to ROSE for OpenCL programs because the lack of support to C++ language features in Cetus.

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 native GPU compiler 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 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 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.

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 [4] 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.

IV. Evaluation

We use a number of kernels, written in either CUDA or OpenCL or both, to measure the influence of program inputs on GPU optimizations and evaluate the effectiveness of G-ADAPT+ in attacking the optimization difficulties. They mainly come from NVIDIA SDK. The program, *mvMul*, is a matrix vector multiplication program implemented efficiently by Fujimoto [2]. To study the influence of inputs, we collect or create 7–18 different inputs for each of the programs. We use two types of GPU devices, GeForce 8800 GT for CUDA experiments and Telsa 1060 for OpenCL experiments. That helps examine whether the input sensitivity persist across devices.

Figure 2 shows the performance of *mvMul* 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

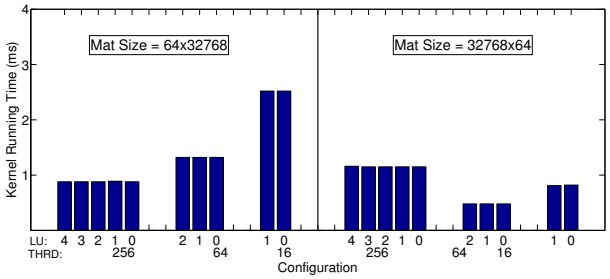


Fig. 2. Performance of matrix-vector multiplication on two inputs when different optimization decisions are used. LU: loop unrolling levels; THRD: number of threads per block. The maximum unrolling level can only be $\sqrt{THRD}/4$.

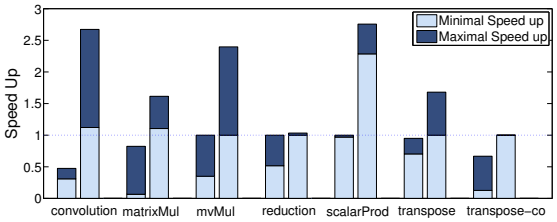


Fig. 3. The ranges of speedup (on CUDA programs) brought by different optimization decisions. For each program, the left bar shows the range of speedup (less than 1 means slowdown) if the worst decision is taken. The right bar shows the range of speedup when the G-ADAPT+ prediction is used.

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.

Input sensitivity shows up on some other kernels as well. For lack of space, we give only a brief summary of the experimental results, and leave the detailed results in the appendix (Figures 5, 6, 7).

We apply the predictions of G-ADAPT+ to measure the effectiveness in performance improvement. The prediction is based on leave-one-out cross validation [4], which is a typical practice in statistical learning to estimate the error of a predictive model in real uses by separating testing and training data apart. On average, the scheme achieves 80% to 100% accuracy for predicting the best optimization parameters for a kernel, demonstrating 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 3 summarizes the ranges of speedup brought by G-ADAPT+ on the 7 CUDA 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.

Figure 4 summarizes the ranges of speedup on a set of OpenCL kernels. The speedup is even more significant than the CUDA results.

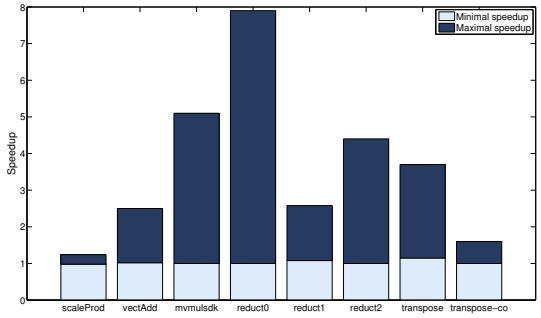


Fig. 4. The ranges of speedup (on OpenCL programs) brought by G-ADAPT+ optimization decisions.

V. 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.

References

[1] 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.

[2] 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.

[3] Rudy G., M. Khan, M. Hall, C. Chen, and C. Jacqueline. A programming language interface to describe transformations and code generation. In *Proceedings of LCPC. Lecture Notes in Computer Science*, 2010.

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

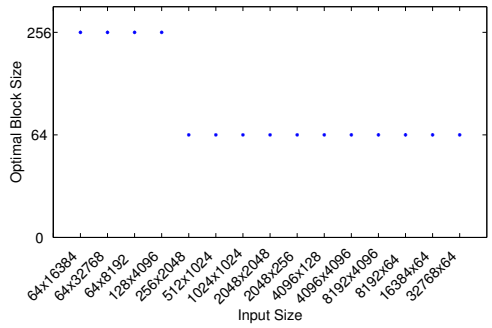
[5] S. Lee, T. Johnson, and R. Eigenmann. Cetus - an extensible compiler infrastructure for source-to-source transformation. In *In Proceedings of the 16th Annual Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 539–553, 2003.

[6] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for gpu programs optimization. In *Proceedings of International Parallel and Distribute Processing Symposium (IPDPS)*, pages 1–10, 2009.

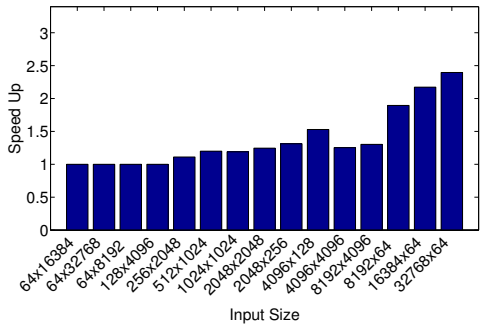
[7] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *CGO'08: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 195–204, 2008.

[8] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *Proceedings of the Joint Modular Languages Conference held in conjunction with EuroPar'03*, 2003.

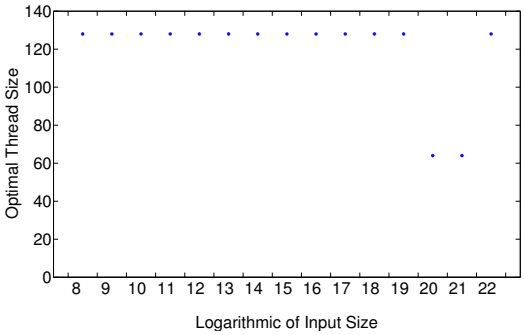
Appendix A
Detailed Experimental Results



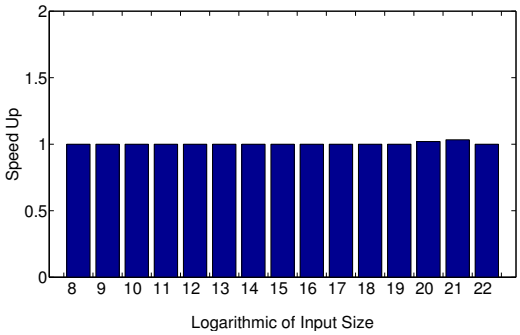
(a) Best thread block size of *mvmul*



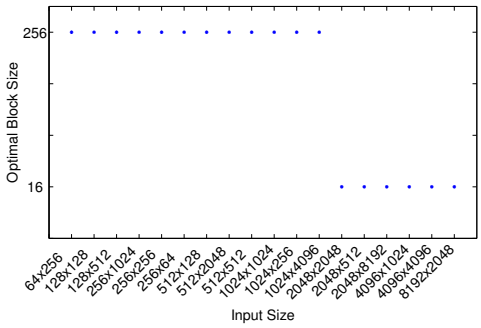
(b) Speedup of *mvmul*



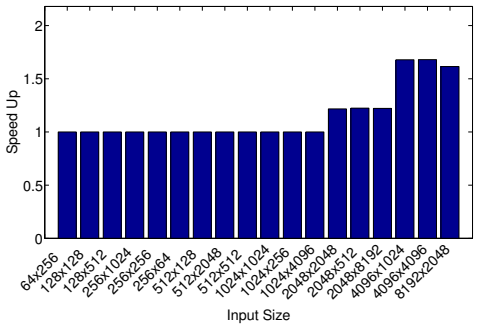
(c) Best thread block size of *reduction*



(d) Speedup of *reduction*

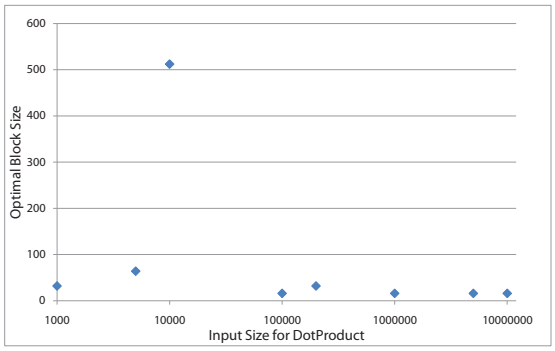


(e) Best thread block size of *transpose*

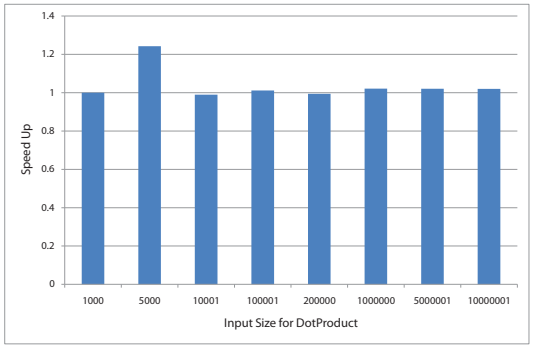


(f) Speedup of *transpose*

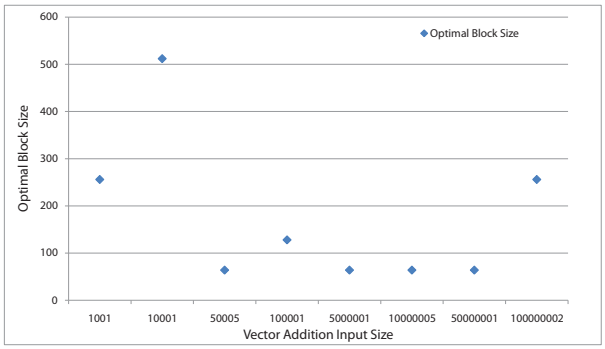
Fig. 5. The best values of optimization parameters and speedups of CUDA programs. The other four benchmarks, *matMulGPU, convolution, scalarProd, transpose-co*, do not show input sensitivity.



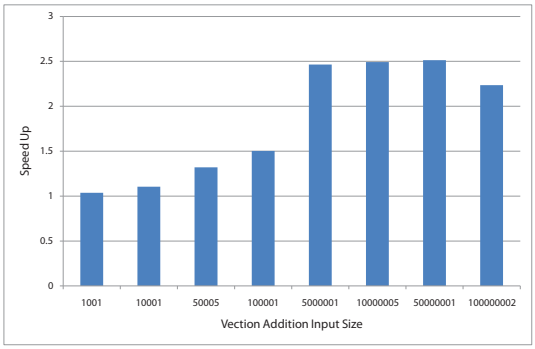
(a) Best opt. parameters of *scaleProd*



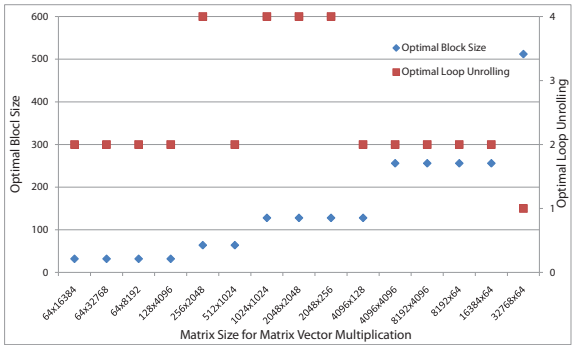
(b) Speedup of *scaleProd*



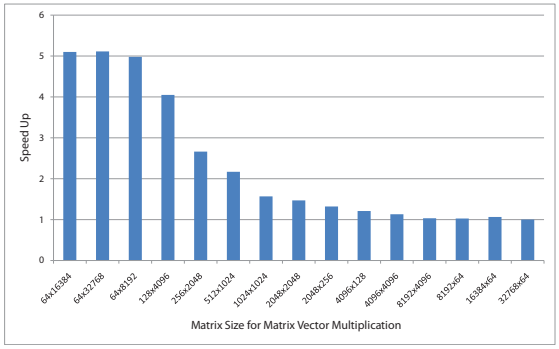
(c) Best opt. parameters of *vectAdd*



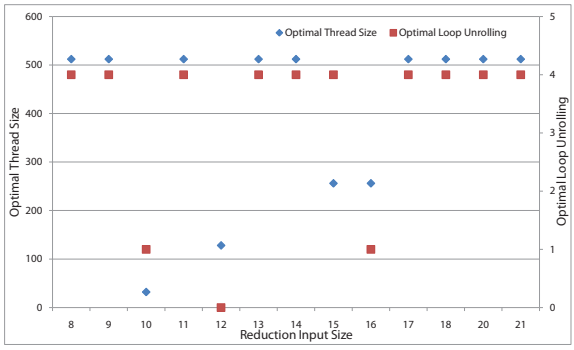
(d) Speedup of *vectAdd*



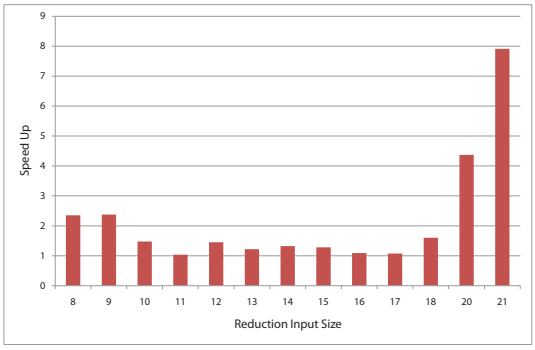
(e) Best opt. parameters of *mvmulsdk*



(f) Speedup of *mvmulsdk*

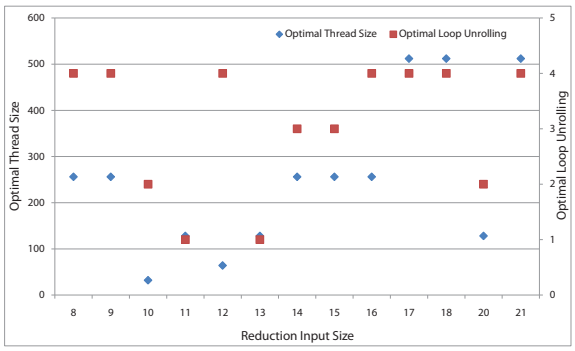


(g) Best opt. parameters of *reduct0*

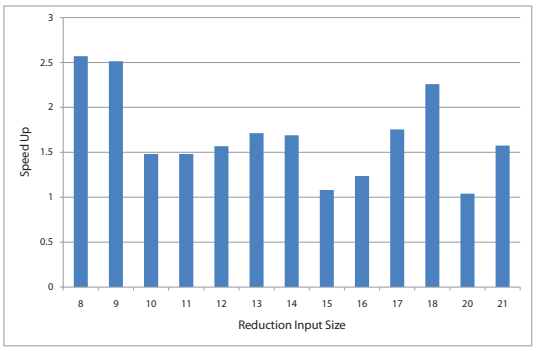


(h) Speedup of *reduct0*

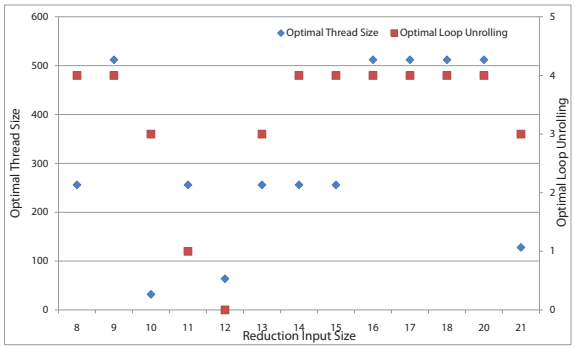
Fig. 6. The best values of optimization parameters and speedups of OpenCL programs (continued in Figure 7)



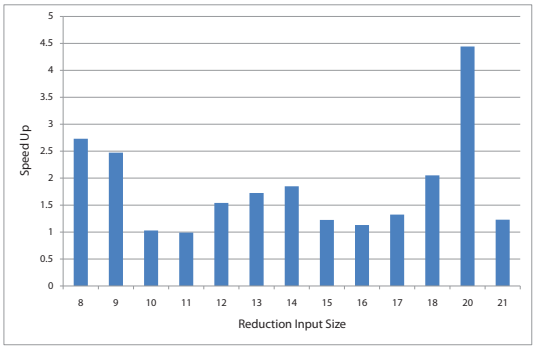
(i) Best opt. parameters of *reduct1*



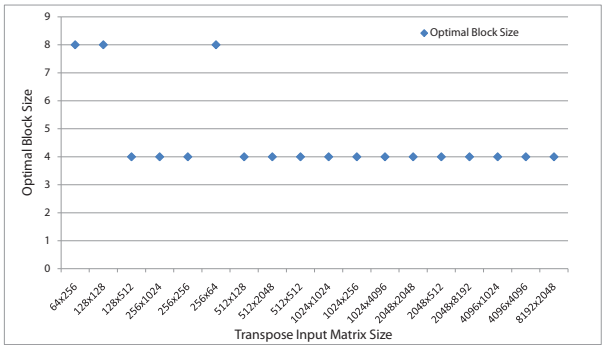
(j) Speedup of *reduct1*



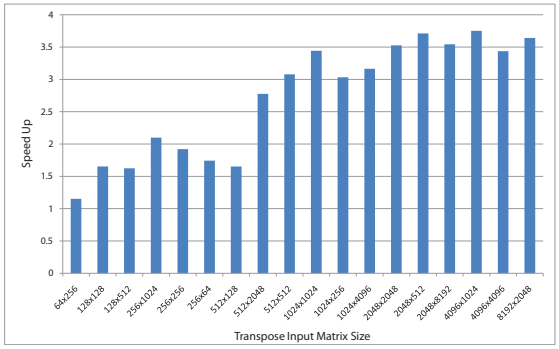
(k) Best opt. parameters of *reduct2*



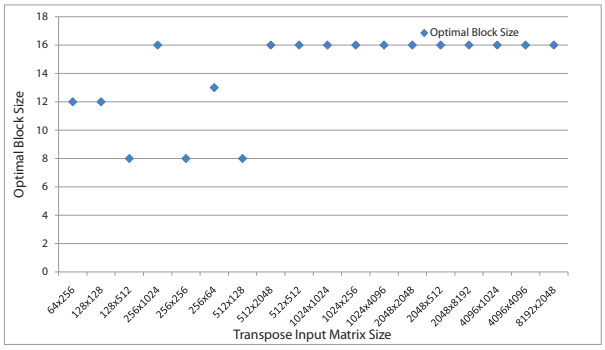
(l) Speedup of *reduct2*



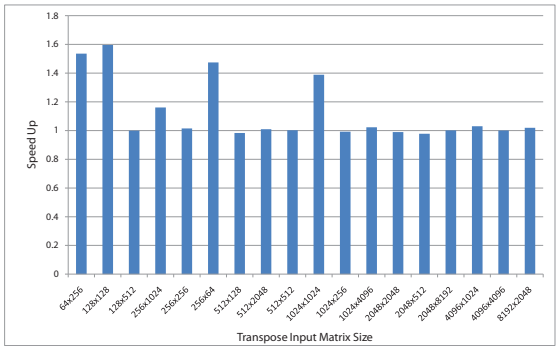
(m) Best opt. parameters of *transpose*



(n) Speedup of *transpose*



(o) Best opt. parameters of *transpose-co*



(p) Speedup of *transpose-co*

Fig. 7. The best values of optimization parameters and speedups of OpenCL programs.