Effects of Compiler Optimizations in OpenMP to CUDA Translation

Amit Sabne, Putt Sakdhnagool, and Rudolf Eigenmann

Purdue University, West Lafayette IN 47907, USA

Abstract. One thrust of the OpenMP standard development focuses on support for accelerators. An important question is whether or not OpenMP extensions are needed, and how much performance difference they would make. The same question is relevant for related efforts in support of accelerators, such as OpenACC. The present paper pursues this question. We analyze the effects of individual optimization techniques in a previously developed system that translates OpenMP programs into GPU codes, called OpenMPC. We also propose a new tuning strategy, called *Modified IE (MIE)*, which overcomes some inefficiencies of the original OpenMPC tuning scheme. Furthermore, MIE addresses the challenge of tuning in the presence of runtime variations, owing to the memory transfers between the CPU and GPU. MIE, on average, performs 11% better than the previous tuning system while restricting the tuning system time complexity to a polynomial function.

Keywords : GPU, CUDA, Tuning System, Compiler Optimizations

1 Introduction

OpenMP has established itself as a standard in parallel programming and is of particular interest for today's and future multicores. There is a large and growing code base, the standard is well understood and documented, and there exists a multitude of compilers and supporting tools. These features are of paramount importance to the programmer. They help significantly reduce the difficulty and the cost of developing parallel software.

The number of new parallel languages that have been proposed in even just the past two decades is massive. The question of cost versus benefit arises with every such proposal. Unfortunately, few quantitative analyses are available that would allow one to find out if the same objective could have been achieved with an existing language standard and what are costs and benefits of new versus old, in terms of performance and productivity. Obviously, any new language will start from zero in building a code base, compilers, tools, and programming experience.

* This work was supported, in part, by the National Science Foundation under grants No. CNS-0720471, 0707931-CNS, 0833115-CCF, and 0916817-CCF.

A new language development has emerged in the context of new graphics processing units, or accelerators. These devices offer promising avenues towards low-energy, highly parallel computation for a class of applications. Among the proposed programming languages are CUDA and OpenCL, both of which allow the programmer to access architecture-specific features. These architecture-specific interfaces, however, significantly depart from the parallel programming semantics offered by standards, such as OpenMP. The cost/benefit question arises anew.

In previous work, we have addressed this cost/benefit question. We have provided quantitative comparisons of hand-written CUDA programs versus equivalent programs written in OpenMP and translated to CUDA [1]. Using an automatic translator and tuning system, called OpenMPC, we were able to achieve performance results that came close to hand-coded CUDA on a large set of benchmarks. The contribution of the present paper is to address three open issues of that work.

- The previous work provided overall performance numbers. The breakdown into individual techniques was not yet available. In the current paper, we quantify the contributions of each individual technique. Of particular interest in this analysis is also the importance of CUDA-specific OpenMP extensions, which are generated automatically in the OpenMPC system.
- A key component of the OpenMPC is its tuning system, which empirically searches through a large space of optimization variants and tries to find the best. The initial OpenMPC system used an inefficient exhaustive search mechanism. In this work, we use an improved navigation algorithm, significantly reducing tuning time.
- A problem faced by all empirical tuning systems is the variability of execution times, even for the same program executed repeatedly on the same platform in single-user mode. This effect makes it difficult to correctly measure the impact of an optimization technique. A common method is to average over multiple runs, increasing tuning time. We have developed a new method that identifies optimizations that are vulnerable to runtime variation and uses increased measuring time only for those.

The remainder of the paper is organized as follows. Section 2 describes OpenMPC and its available optimization options. It also identifies opportunities for improvement in the present OpenMPC tuning system. Section 3 explains our tuning mechanism for finding the best tuning options. Individual performance analysis is shown in Section 4. Section 5 makes concluding remarks and mentions ongoing work.

2 Overview of OpenMPC System

OpenMPC [1] is a programming framework that generates CUDA programs from OpenMP programs. The framework includes an extended OpenMP programming interface, a source-to-source translator, and an automatic tuning system.

The programming interface extends OpenMP with a new set of directives and environment variables (henceforth referred to as CUDA extensions[1]) for controlling CUDA-related parameters and optimizations. OpenMP translates standard OpenMP programs by applying a set of program transformations and by inserting CUDA extensions. OpenMPC includes an empirical tuning system that automatically generates, prunes, and searches the optimization space and determines the best combination of optimizations. Fig. 1 shows the workflow of the OpenMPC translator. Fig. 2 displays a small example of the OpenMPC translated CUDA code for Jacobi benchmark.

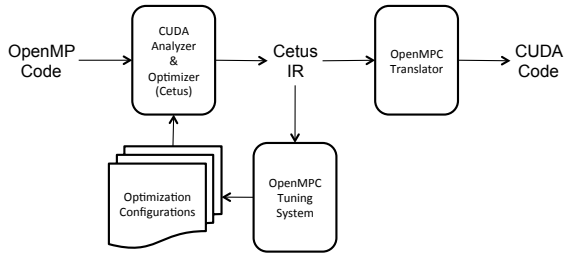

Fig. 1. OpenMPC workflow

```
#pragma omp parallel for private(i, j)
for (i = 1; i <= SIZE; i++){
  for (j = 1; j <= SIZE; j++){
    a[i][j]=(b[i-1][j]+b[i+1][j]+b[i][j-1]+b[i][j+1])/4.0f;
}}
```

(a)

```
__global__ void kernel(...){

  int _bid = (blockIdx.x+(blockIdx.y*gridDim.x));
  int _gtid = (threadIdx.x+(_bid*blockDim.x));
  tid=(_gtid+1);

  if (tid<=SIZE){
    for (i=1; i<=SIZE; i ++ ){
      a[i][j]=(b[i-1][tid]+b[i+1][tid]+b[i][tid-1]+b[i][tid+1])/4.0F;
}}}
```

(b)

Fig. 2. OpenMPC translation example. (a) source code in OpenMP (b) result CUDA kernel from OpenMPC translation

[1] Our CUDA extensions are not meant to be a proposal for extending the OpenMP standard. They represent a research framework for exploring questions such as those addressed in this paper.

2.1 Optimization options

There are 18 optimization options available in OpenMPC, grouped into 4 categories: (1) Program environment configuration, (2) Data caching strategy, (3) Data offloading optimizations, and (4) Code transformation. Table 1 shows all optimization options in OpenMPC that are considered for individual optimization analysis. The first three groups are supported by our CUDA extensions. The fourth group is applied through source-to-source transformation in the OpenMPC compiler.

2.2 Improving the OpenMPC Tuning System

To analyze the effects of individual tuning options, we make use of the OpenMPC system, which allows us to implement the method in [2]: Using the highest-optimized program variant as a baseline, this method iteratively switches off one optimization at a time, to measure its effect in terms of the slowdown incurred. To this end, we have identified a number of open issues in OpenMPC, which we address in the present work.

Advanced Optimization Space Navigation: The goal of an empirical tuning system is to generate a set of optimizations that yield best performance. In OpenMPC, 18 optimizations are available as compiler flags. Finding the best combination from these flags is non-trivial, because each optimization can improve or worsen the performance of a program, depending upon its characteristic and depending on other present optimizations.

The initial OpenMPC system uses simple *exhaustive search* to navigate the space of optimization variants. This space can be very large (for n on-off optimization options, the size is 2^n). OpenMPC reduces this space using aggressive tuning heuristics, which we refer to as *pruned exponential search (PE)*. PE does the program analysis to prune the tuning space by removing the inapplicable or non-beneficial tuning options for the particular program. It then runs exhaustive search over the remaining tuning options. However, two issues remain: The resulting search space can still be large (which was acceptable for obtaining the original research results [1], but can be too long for end users). In addition, sometimes the aggressive pruning heuristics may eliminate the best optimization combination.

Runtime Variations – A Key Problem of Auto-Tuning Systems: In computer systems, unpredictable system variations during program execution are usual. They arise due to OS overheads, other running processes, or underlying hardware operations. Although these variations do not affect the correctness of the program, they can impact its execution time. We define this type of variation as *runtime variation*. Although runtime variation does not disrupt program execution, in auto-tuning system, runtime variation can be problematic. Since the auto-tuning systems improves the program based on execution time,

Table 1. Optimization options in OpenMPC

Program Environment Configuration

Compiler Flags	Description
cudaThreadBlockSize=N	Set the default CUDA thread block size
assumeNonZeroTripLoops	Assume that all loops have non-zero iterations

Data Caching Strategy

Compiler Flags	Description
shrdSclrCachingOnReg	Cache shared scalar variables onto GPU register
shrdArryElmtCachingOnReg	Cache shared array elements onto GPU register
shrdSclrCachingOnSM	Cache shared scalar variables onto GPU shared memory
prvtArryCachingOnSM	Cache private array variables onto GPU shared memory
shrdArryCachingOnTM	Cache 1-dimensional, R/O shared array variables onto GPU texture memory
shrdSclrCachingOnConst	Cache R/O shared scalar variables onto GPU constant memory
shrdArryCachingOnConst	Cache R/O shared array variables onto GPU constant memory

Data Offloading Optimization

Compiler Flags	Description
useMallocPitch	Use cudaMallocPitch() for 2-dimensional arrays
useGlobalGMalloc	Allocate GPU variables as global variables which provides more scope for reducing memory transfers
globalGMallocOpt	Apply CUDA malloc optimization for globally allocated GPU variables
cudaMallocOptLevel=N	Set CUDA malloc optimization level for locally allocated GPU variables
cudaMemTrOptLevel=N	Set CUDA CPU-GPU memory transfer optimization level

Code Transformation

Compiler Flags	Description
localRedVarConf=N	Configure how local reduction variables are generated for array-type variables
useMatrixTranspose	Apply Matrix Transpose optimization
useParallelLoopSwap	Apply Parallel Loop Swap optimization
useUnrollingOnReduction	Apply Loop Unrolling for in-block reduction

the variation can cause some beneficial optimizations to be removed from the tuning result and vice versa.

One of the significant observations made during our study was the fact that most of the variations on GPU programs are due to the variations in memory transfer times. Since GPU and CPU do not share a common address space, memory transfers form an essential part of GPU programs. GPUs are generally

connected to the CPU using a PCIe bus, thereby leading to a variability in the memory transfer times. Table 2 compares the relative standard deviation in computation time and the memory transfer time. Relative standard deviation is a percentage of the ratio of standard deviation to the mean of the sample. It acts as an indicator as of how the variations relate to the average. From Table 2, we can see that the relative standard deviations in memory transfer can be as much as 7000 times the relative standard deviations in computation time.

Table 2. Variations on GPU Programs

Benchmark	Relative Standard Deviation for Memory Transfer Time (A)	Relative Standard Deviation for Computation Time (B)	Ratio (A/B)
NW (8192)	0.2395	0.0128	18.71
Jacobi (12288)	0.7394	0.0001	7394
CG (W)	0.2562	0.0706	3.63
FT (W)	0.1521	0.0112	13.58

To alleviate runtime variations, one can average execution times across multiple runs. However, multiple executions can increase the tuning time significantly. The PE algorithm does not take runtime variations into consideration, and therefore is more prone to erroneous final option combinations on GPU programs.

Objectives of this Work: Our goal is to determine the impact of individual optimization techniques in the OpenMP to CUDA translator. To this end, we use the improved OpenMPC translation and tuning system, which can find the best combination of optimization techniques for each program. In doing so, it also reports the performance difference made by individual optimizations. We proceed as follows.

- We modify a previously described *Iterative Elimination (IE)* [3] tuning algorithm to make it applicable to GPU programs.
- We describe a generic tuning methodology to deal with memory transfer time based variations of GPU applications.
- With the best tuning option combination generated by the above tuning system, we analyze the impact of each tuning option or compiler flag.

The next section presents the new tuning algorithm. Section 4 presents results obtained using this methodology.

3 Modified IE (MIE) Algorithm for OpenMPC

To address the issues presented in Section 2.2, we propose a *Modified IE (MIE)* algorithm, which is a tuning algorithm based on *Iterative Elimination (IE)* [3]. In this section, we briefly describe IE and then present our MIE algorithm.

3.1 Iterative Elimination

The IE algorithm is shown in Algorithm 1. IE begins by switching on all optimization options, and then iteratively measures their effect by switching off one tuning option at a time. Next, it removes the one with the most negative effect. The process repeats until all remaining optimizations show non-negative effects. The complexity of IE is $O(n^2)$, compared to $O(2^n)$ of the *PE* algorithm.

Algorithm 1 Iterative Elimination Algorithm

Require: n = Number of Tuning Options (F_1, F_2, ... F_n)
Ensure: $B = \{F_1 = 1, F_2 = 1, ..., F_n = 1\}$ B is a set of combination options
  $i \leftarrow 1$; $NextB \leftarrow B$; ▷ $NextB$ stores the fastest combination in every iteration
  for $i = 1 \rightarrow n$ **do**
    for $j = 1 \rightarrow n$ **do**
      if $F_j \neq 0$ **then**
        $NextB$ = min($NextB$, B with $F_j = 0$); ▷ Compares the runtimes
      end if
    end for ▷ Termination: No F_i has changed from 1 to 0
    if $NextB = B$ **then**
      break; ▷ None of the switched on options has a negative impact
    end if
    $B \leftarrow NextB$; ▷ Start next iteration with a new baseline $NextB$
  end for ▷ Creates set of best tuning options e.g $B = \{F_1 = 1, F_2 = 0, ., F_n = 1\}$

Another tuning method, *Combined Elimination (CE)* [3] performs the option removal in a more aggressive fashion, under the assumption that some interferences between options are negligible. The tuning time of CE is known to be shorter than IE. However, since the performance of IE is known to be the best amongst the available tuning algorithms [3], we chose IE as our base algorithm. Other algorithms could be adapted in place of IE in our system [4, 5]. Unlike the work in [6], which uses optimal ordering of compiler flags, IE tries to find the best tuning options set, irrespective of the order.

3.2 Grouping of different Optimization Options

To deal with the problem of runtime variations, a direct implementation of IE would require multiple runs and averaging before eliminating an optimization option. This would lead to high tuning times, because the runtime variations of GPU programs can be large.

Comparing only the computation runtime instead of the total execution time can eliminate the effect of memory transfer variations on tuning. To achieve that, the behavior of memory transfers must be the same between two comparable candidate combinations of IE. If this invariant is maintained, the memory transfer time can be subtracted from total execution time (e.g., by obtaining these times from available hardware profilers) an optimization technique is evaluated by IE.

An intuitive strategy would be to apply techniques that affect memory transfers (i.e. data offloading optimizations shown in Table 1) in a first tuning phase,

averaging the results over multiple runs. In a second phase, the remaining optimization options are tuned, whereby transfer times are removed from execution times. In this way, most of the runtime variations in the GPU program can be filtered out; a single run suffices.

The split into the two phases is beneficial only when the data offloading optimizations do not interfere with other. That is not always the case. For example, *useMallocPitch*, which manages 2D array allocation and transfer, may or may not be beneficial depending on the stride of 2D array accesses. Since *useParallelLoopSwap* transforms the array accesses in the code, *useMallocPitch* may improve performance if *useParallelLoopSwap* is applied.

To address this problem, *MIE* uses a third phase, in which memory transfer optimizations that are affected by computation optimization options are placed. This phase also averages runtimes over multiple runs. In a fourth phase, MIE tunes separately those optimizations that do not interact with others. It uses a simple, fast tuning algorithm for this phase.

Phase 1 contains all memory transfer-based (data offloading) optimizations, except useMallocPitch. *Phase 2* contains program environment configuration and code transformation options that impact the computation. *Phase 3* contains dependent optimizations. With the currently available tuning options in OpenMPC, *Phase 3* contains only *useMallocPitch*. This technique impacts the data offloading (memory transfers), but is dependent upon computation technique *useParallelLoopSwap*. *Phase 4* contains data caching optimizations. They are independent of the techniques in the other groups.

Table 3. Grouping of OpenMPC Options for Tuning (MemTR = Memory Transfer Optimization, Comp = Computation Optimization). Options in paranthesis imply multi-values options

Phase	Type	Tuning Options
1	MemTR	useGlobalGMalloc, globalGMallocOpt, cudaMallocOptLevel=1, cudaMemTrOptLevel=2
2	Comp	useUnrollingOnReduction, useLoopCollapse, useMatrixTranspose, useParallelLoopSwap, prvtArryCachingOnSM, localRedVarConf=0, assumeNonZeroTripLoops
3	Dependent	useMallocPitch
4	Independent	ArrayCache = {shrdArryElmtCachingOnReg, shrdArryCachingOnTM, shrdArryCachingOnConst} ScalarCache = {shrdSclrCachingOnReg, shrdSclrCachingOnSM, shrdSclrCachingOnConst}

3.3 MIE Running Strategy

With the above groups of optimizations in place, we now describe the MIE run strategy.

1. **Data Offload Optimizations**: First the algorithm runs IE with the *Phase 1* optimizations as the input set. Since these options all impact memory transfers, they are vulnerable to high runtime variations. The MIE algorithm runs each IE stage multiple times and considers the average execution times for making elimination decisions.
2. **Computation Optimizations**: The configuration formed in *Phase 1* is the baseline configuration. MIE now appends *Phase 2* options to this configuration and runs IE over all new options. While making comparisons between two combinations, the memory transfer time is removed from the comparison, effectively considering only the computation time. This helps reduce the effect of variations to a large extent. This stage requires calculation of the time spent in copying the data between CPU and GPU memories. This is accomplished by using the CUDA profiler. Using this method, MIE avoids averaging over multiple runs, substantially reducing the time required.
3. **Dependent Optimizations**: In the combination formed after *Phase 2*, MIE includes *Phase 3* option i.e. *useMallocPitch* and averages the runtimes over multiple executions to see if this option is beneficial and should be included. (Should there be more tuning options added in *Phase 3*, MIE would run IE on this group, with averaging runtimes over multiple executions.)
4. **Independent Optimizations**: Since this group does not depend upon other options, MIE iteratively runs each *Phase 4* option on top of the configuration formed in *Phase 3*, and adds the best value of each multi-valued option to the final optimization configuration.

4 Performance Analysis

4.1 Setup

We ran both the PE and the MIE algorithm on NVIDIA Quadro FX 5600 GPU device, which has 16 multiprocessors (SMs) clocked at 1.35GHz and 1.5 GB of memory. Each SM consists of 8 SIMD processing units (SPs) and has 16 KB of shared memory. The host CPU is a 3-GHz AMD dual-core processor with 12 GB memory. The OpenMPC generated CUDA programs were compiled using the NVIDIA CUDA Compiler (NVCC) with option -O3.

We demonstrate the effectiveness of our tuning system on NAS OpenMP Parallel benchmarks, Rodinia OpenMP benchmarks and some scientific computation applications. As described in 3.3, we run *Phase 1* and *Phase 3* options 5 times each and use the average runtimes for IE. For other groups, we compare only the computation times for IE runs.

4.2 Performance Comparison Between Pruned Exhaustive and Modified IE Algorithms

To evaluate the performance of the MIE algorithm, we show in Figure 3 the speedup of benchmarks achieved with MIE, normalized with respect to the PE

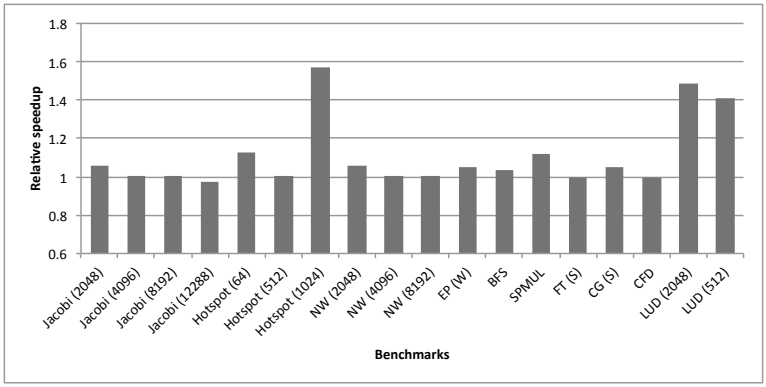

Fig. 3. Program Speedups of Modified IE relative to Pruned Exhaustive Algorithm

algorithm. MIE performs better than the PE algorithm in most of the cases, averaging to a 11% performance improvement over PE. In fact, MIE outperforms Pruned Exhaustive method substantially for the Hotspot and LUD benchmarks. This effect is due to the over-pruning occurring in the PE method, thereby missing out on the best option combination.

Another important observation is the fact that MIE performs marginally better (2 to 5 %) compared to Pruned Exhaustive method on most other programs where over-pruning does not happen. This is counter-intuitive since PE is expected to search through all possible choices. It is explained due to the excessive memory transfer based variations, wherein the best option combination produced by the Pruned Exhaustive method may not be the optimal, rather it is the one that suffered the least.

Table 4 compares the tuning time required by the Pruned Exhaustive algorithm against the tuning time required by the MIE algorithm. The advantage of IE in terms of tuning time is evident from this table.

Table 4. Tuning Time Comparison of Pruned Exhaustive Vs. Modified IE Algorithm

Benchmark	Tuning Time (mins)	
	Pruned Exhaustive Tuning	Modified IE Tuning
SRAD	538	23
FT (S)	2345	23
CG (S)	1108	17
CFD (97k)	1083	210
FT (A)	3680	97
Jacobi (12288)	98	55

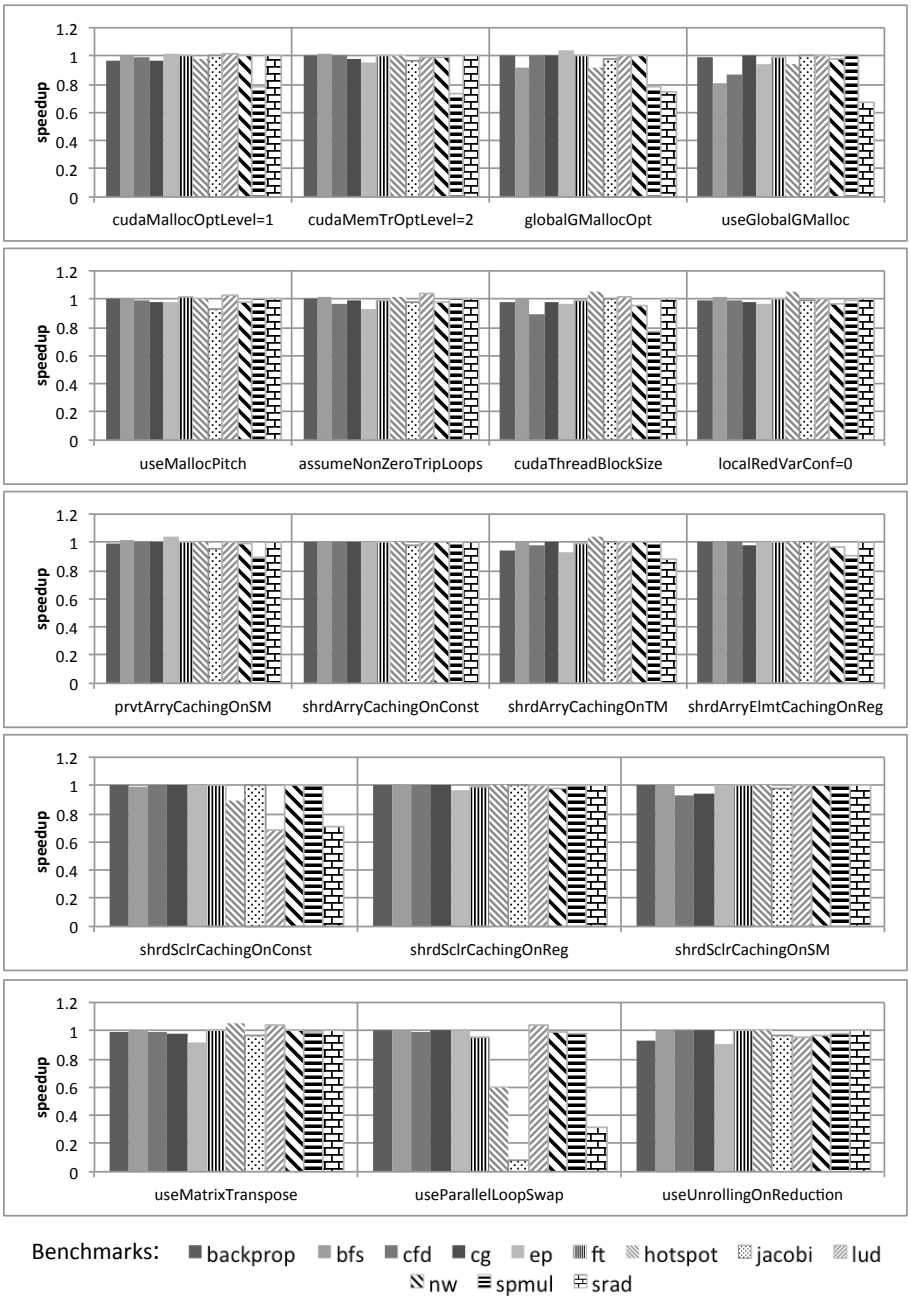

Fig. 4. Individual impacts of the 18 optimizations. Bars show normalized performance of the benchmarks after disabling the selected optimization. A large drop in performance indicates high impact.

4.3 Impact of Individual Optimization Options

As stated earlier, to analyze the effect of individual tuning options in OpenMPC, we follow the method from [2], wherein we turn off one optimization at a time from the best tuning options set, so as to understand the effects of the individual optimization in terms of the slowdown incurred. The bigger the slowdown, the larger is the benefit of the optimization. We analyze the results in Fig 4 with respect to the techniques shown in Table 1.

Some benchmarks like SRAD, Jacobi, SPMUL depict high benefits obtained due to compiler techniques. However, some others like Backprop show relatively small effects. The effectiveness of our Modified IE tuning algorithm can be gauged from the observation that switching off an individual technique with respect to the best tuning optimization set has never improved the performance beyond 3%, which can be attributed to the computation variations.

Memory transfer optimization-based techniques show high impact on many GPU programs. Similarly, the techniques that change data access strides can be highly beneficial since they help coalesce memory accesses. *useParallelLoopSwap* and *useMatrixTranspose* are some such techniques.

Exploiting GPU specific memories for caching both the scalar and array variables can be highly beneficial. GPUs have on-chip cache and shared memories and off-chip constant and texture memories. The current OpenMPC setup tries to put all the variables (either scalar or arrays) on one of these memories, depending upon the tuning option provided. However, since these memories may not be large enough to hold the complete data sets, the compilation of such programs may fail (in which case the current tuning system ignores the option). We foresee a methodology to adaptively exploit all the GPU specific memories.

5 Conclusion and Future Work

We have analyzed the performance of GPU optimization techniques present in the OpenMPC translation and tuning system. Our main findings indicate that the compiler engineer who wishes to translate a program in a given language into a CUDA program should consider the following optimizations:

1. Memory transfer optimization-based techniques are essential for offloading-based programming models.
2. Exploiting special memories on GPUs can yield significant speedups.
3. Transformations that change the memory access strides are of great importance in GPU programs.
4. Tuning is important. With its help, *standard* OpenMP programs can be translated effectively and efficiently into CUDA/GPU code.
5. Explicit GPU programming (without tuning support) needs to make use of CUDA-extensions (above items 1, 2) for best performance. It is important for emerging standards, such as OpenMP (3.1) [7] and OpenACC [8] to support

these features. Above items 1 and 3 should be applicable to a wide range of accelerators. Item 2, however, is CUDA specific, but is necessary to obtain best performance.

We also proposed a new empirical tuning algorithm for GPU programs called Modified IE (MIE), which significantly reduces tuning time. MIE addresses and is able to tolerate runtime variations caused by memory transfer between GPU and CPU. As a result, MIE performs 11% better, on average, than the original OpenMPC tuning system [1], while maintaining polynomial tuning time.

Ongoing work: The presented analysis of different techniques has provided us with intuitions as of what kind of compiler techniques are useful on GPUs. We did not implement some of the *unsafe options* [1] in MIE, the application of which may provide larger benefits. We plan to extend the tuning system into automatically incorporating such options, with the programmer's help in understanding correctness of the output. Furthermore, best performance is achieved by inserting certain CUDA-extension directives in the OpenMP source program [1]. Our ongoing work includes the extension of the translation and tuning system to automate these modifications as well.

References

1. Lee, S., Eigenmann, R.: Openmpc: Extended openmp programming and tuning for gpus. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. SC '10, Washington, DC, USA, IEEE Computer Society (2010) 1–11
2. Blume, W., Eigenmann, R.: Performance analysis of parallelizing compilers on the perfect benchmarks programs. IEEE Transactions on Parallel and Distributed Systems **3** (1992) 643–656
3. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: Proceedings of the International Symposium on Code Generation and Optimization. CGO '06, Washington, DC, USA, IEEE Computer Society (2006) 319–332
4. Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D.I.: Compiler optimization-space exploration. In: Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. CGO '03, Washington, DC, USA, IEEE Computer Society (2003) 204–215
5. Pinkers, R.P.J., Knijnenburg, P.M.W., Haneda, M., Wijshoff, H.A.G.: Statistical selection of compiler options. In: Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems. MASCOTS '04, Washington, DC, USA, IEEE Computer Society (2004) 494–501
6. Cooper, K.D., Subramanian, D., Torczon, L.: Adaptive optimizing compilers for the 21st century. J. Supercomput. **23** (August 2002) 7–22
7. OpenMP 3.1: Openmp 3.1 released. http://openmp.org/wp/openmp-31-released/ (July 2011)
8. OpenACC. http://www.openacc-standard.org/ (November 2011)