OpenMPC: Extended OpenMP for Efficient Programming and Tuning on GPUs

Seyong Lee*

Computer Science and Mathematics Division,
Oak Ridge National Laboratory,
Oak Ridge, TN 37831, USA
Fax: 1-865-241-2650
E-mail: lees2@ornl.gov
*Corresponding author

Rudolf Eigenmann

School of Electrical and Computer Engineering,
Purdue University,
West Lafayette IN 47907, USA
Fax: 1-765-494-6440
E-mail: eigenman@purdue.edu

Abstract: General-Purpose Graphics Processing Units (GPGPUs) provide inexpensive, high performance platforms for compute-intensive applications. However, their programming complexity poses a significant challenge to developers. Even though the CUDA (Compute Unified Device Architecture) programming model offers better abstraction, developing efficient GPGPU code is still complex and error-prone. This paper proposes a directive-based, high-level programming model, called OpenMPC, which addresses both programmability and tunability issues on GPGPUs. We have developed a fully automatic compilation and user-assisted tuning system supporting OpenMPC. In addition to a range of compiler transformations and optimizations, the system includes tuning capabilities for generating, pruning, and navigating the search space of compilation variants. Evaluation using fourteen applications shows that our system achieves 75% of the performance of the hand-coded CUDA programs (92% if excluding one exceptional case).

Keywords: OpenMP; GPU; CUDA; OpenMPC; programming model; directives; automatic translation; compiler transformation; performance tuning; code generation; optimizations.

Reference to this paper should be made as follows: Seyong Lee and Rudolf Eigenmann. (2012) 'OpenMPC: Extended OpenMP for Efficient Programming and Tuning on GPUs', *Int. J. Computational Science and Engineering*, Vol. 7, No. 1, pp. 1–16.

Biographical notes: Seyong Lee is a computer scientist in the Computer Science and Mathematics Division at Oak Ridge National Laboratory. His research interests include parallel programming and performance optimization in heterogeneous computing environments, program analysis, and optimizing compilers. Lee received a PhD in electrical and computer engineering from Purdue University, USA.

Rudolf Eigenmann is a professor in the School of Electrical and Computer Engineering at Purdue University. His research interests include optimizing compilers, programming models for parallel computing, and cyberinfrastructures. Eigenmann received a PhD in electrical engineering/computer science from ETH Zurich, Switzerland.

1 Introduction

Graphics Processing Units (GPUs) have been widely adapted as alternative platforms for inexpensive, high performance computing. Their increased popularity results not only from their unique combination of power, performance, and cost but also from the great improvement in GPU programmability. CUDA (Compute Unified Device Architecture) is one such approach to lower the barriers to the complex GPU programming. The CUDA model abstracts a GPU as a general-purpose multi-threaded SIMD (Single Instruction, Multiple Data) architecture, known as a general purpose GPU (or GPGPU). Even though the CUDA programming model offers a more user-friendly interface, programming GPGPUs is still complex and error-prone, due to the exposure of its unique memory and execution models to programmers.

OpenMP [OpenMP] is an API (Application Programming Interface) for multi-platform shared memory programming, which consists of compiler directives, library routines, and environment variables. In our prior work [Lee et al., 2009], we have proposed an automatic OpenMP-to-CUDA translation framework, which extends the ease of creating parallel applications with OpenMP to GPGPU architectures such as CUDA. It includes various optimizations to deal with the performance gap caused by architectural differences between traditional shared memory systems, served by OpenMP, and stream architectures adopted by most GPUs.

However, developing efficient CUDA programs still remains difficult; complex interactions among hardware resources and the multi-layered software stack used for CUDA compilation and execution make it difficult for the compiler to statically predict the overall performance effect [Ryoo et al., 2008b, Liu et al., 2009]. Moreover, the abstraction provided by the OpenMP-to-CUDA translation framework does not allow enough control over fine-grained tuning. To achieve optimal performance, additional, manual modifications of the generated CUDA code may be required, which can be tedious and error-prone [Liu et al., 2009, Han and Abdelrahman, 2011, Baskaran et al., 2010].

There exists extensive work to optimize the performance of CUDA-based GPGPU programs. Studies on general CUDA optimization strategies reveal that the performance gap between well-optimized GPU applications and poorly optimized ones can be orders of magnitude [Ryoo et al., 2008b,a, Baskaran et al., 2008]. The studies also show that porting applications onto GPGPUs may require very high effort and expertise to obtain optimal performance. Even though there have been several efforts to automatically optimize and tune the performance of GPGPU programs, most of them are either application-specific [Nukada and Matsuoka, 2009, Datta et al., 2008, Volkov and Demmel, 2008], restricted to certain types of applications [Baskaran et al., 2008], or applied to only a small subset of optimization parameters [Liu et al., 2009]. Therefore, achieving maximum performance for general GPGPU applications is still a challenge and often involves manual work.

To address this challenge, we propose *OpenMPC* – OpenMP extended for CUDA, which comprises a standard OpenMP API plus a new set of directives and environment variables to control important CUDA-related parameters and optimizations.

This paper makes the following contributions:

- We propose OpenMPC, an extended OpenMP API for improved CUDA programming, which offers a high-level programming and tuning environment where users can generate CUDA programs in many optimization variants without detailed knowledge of the underlying CUDA programming and memory models.

- We have developed a fully automated and parameterized compilation system by extending the framework proposed in our previous work [Lee et al., 2009]; the OpenMPC directives and environment variables allow users to have fine-grained control over the overall OpenMP-to-CUDA translation and optimizations.

- In addition to various transformation and optimization techniques for seamless performance porting, we have developed several tools that assist users in performance tuning; the *search space pruner* analyzes a given input OpenMP program, plus optional user settings, and suggests optimizations that are applicable to the given program. This automatic suggestion has the effect of pruning the search space that a tuning system should navigate to find the best performance. For the search space suggested by the pruner, another tool called *configuration generator* defines all corresponding compilation variants, such that they can be created automatically by the OpenMP-to-CUDA translator.

- We have evaluated the effectiveness of OpenMPC using the reference compilation system and tuning tools. The experiments on fourteen OpenMP programs (two kernel benchmarks, three NAS Parallel Benchmarks, and nine Rodinia Benchmarks) demonstrate that tuning using OpenMPC with optional manual modification on the input OpenMP programs, which may be automated by an advanced compiler, can improve the performance up to 7.71 times (1.24 times on average) over un-tuned versions, which is 75% of the performance of hand-written CUDA versions. If we exclude one exceptional case (Rodinia Benchmark *LUD*), the average speedup will be 92% of that of the manual CUDA versions.

This paper is a revised and expanded version of our previous paper [Lee and Eigenmann, 2010]. It is organized as follows: Section 2 gives an overview of the CUDA model, and Section 3 describes the baseline OpenMP-to-CUDA translation scheme and new optimizations added in this work. Section 4 introduces OpenMPC, and Section 5 presents a reference compilation system and a prototype tuning framework supporting OpenMPC. Experimental results are shown in Section 6, and related work and conclusion are presented in Section 7 and Section 8, respectively.

2 Background on the CUDA Model

CUDA-compatible GPGPUs consist of multiprocessors called *streaming multiprocessors (SMs)*, each of which contains a fixed number of SIMD processing units called *streaming processors (SPs)*. Each SM has a set of registers, which are logically partitioned among threads running on the SM, a fast on-chip *shared memory*, which is shared by SPs in the same SM, and special read-only caches (*constant cache* and *texture cache*), which are shared by SPs. A slow off-chip *global memory* is used for communication among different SMs.

A CUDA program consists of parallel and serial code regions; code regions with rich parallelism are implemented as a set of *kernel functions*, which are executed on the GPU by a number of threads in an SIMD fashion. Serial regions, which are outside of kernel functions, are executed by a host CPU.

In the CUDA model, threads are grouped as a grid of thread blocks, each of which is mapped to an SM on the GPU device. The number of thread blocks and the number of threads per thread block, called *thread batching*, are specified through language extensions at each kernel invocation.

In the CUDA memory model, *global memory*, *constant memory*, and *texture memory* are accessible by all threads, *shared memory* is shared only by threads in the same thread block, and *registers* and *local memory* are private to each thread. The *shared memory* and *registers* in an SM are dynamically partitioned among the active thread blocks running on the SM. Therefore, register and shared memory usages per thread block can be a limiting factor preventing full utilization of execution resources.

The CUDA model assumes that the host CPU and the GPU device have separate address spaces. For communication between CPU and GPU, the CUDA model provides an API for explicit GPU memory management, including functions to transfer data between CPU and GPU.

One limitation of the CUDA model is the lack of efficient global synchronization mechanisms. Synchronization within a thread block can be enforced by using the *__syncthreads()* runtime primitive. However, synchronization across thread blocks can be accomplished only by returning from a kernel call, after which *global memory* data modified by threads in different thread blocks are guaranteed to be globally visible.

3 Overview of the Automatic OpenMP-to-CUDA Translation System

This section describes the OpenMP-to-CUDA translation system, which automatically converts a standard OpenMP program into a CUDA program and applies various optimizations to achieve high performance. This system has been built on top of the Cetus compiler infrastructure [Dave et al., 2009].

3.1 Baseline Translation of OpenMP to CUDA

The baseline translation comprises two steps: (1) interpreting OpenMP semantics under the CUDA model and identifying *kernel regions* (code sections to be executed on a GPU) and (2) transforming eligible kernel regions into CUDA kernel functions and inserting codes for memory transfers between CPU and GPU.

3.1.1 Interpretation of OpenMP Semantics under the CUDA Programming Model

OpenMP directives can be classified into four categories:

(a) *Parallel* construct (*omp parallel*) – this construct specifies parallel regions. Parallel regions may be further split into sub-regions. The translator identifies eligible *kernel regions* among the (sub-)regions and transforms them into GPU kernel functions.

(b) Work-sharing constructs (*omp for*, *omp sections*, etc.) – these constructs contain the true parallel codes in OpenMP. Other sub-regions, within an *omp parallel* region but outside of work-sharing constructs, are executed by one thread, serialized among threads, or executed redundantly among participating threads. The translator interprets these constructs to partition work among threads on the GPU device.

(c) Synchronization constructs (*omp barrier*, *omp critical*, *omp flush*, etc.) – these constructs contain explicit or implicit synchronization points. A parallel region must be split into two sub-regions at each of these constructs to preserve a global synchronization in the CUDA programming model, as explained in Section 2.

(d) Data-property constructs (*omp shared*, *omp threadprivate*, *omp private*, etc.) – the translator interprets these constructs to map data into GPU memory spaces. OpenMP *shared* data are shared by all threads, and OpenMP *private* data are accessed by a single thread. In the CUDA memory model, shared data can be mapped to *global memory*, and private data can be mapped to *registers* or *local memory* assigned for each thread. OpenMP *threadprivate* data are private to each thread, but they have global lifetimes. The semantics of *threadprivate* data can be implemented using data expansion, which allocates copies of the *threadprivate* data on *global memory* for each thread.

3.1.2 Transformation of Kernel Regions into Kernel Functions

The translator considers OpenMP parallel regions as potential *kernel regions*. At each synchronization construct, these parallel regions must be split. Among the resulting sub-regions, the ones containing at least one work-sharing construct become kernel regions. To reduce expensive memory transfers between CPU and GPU, a top-down splitting algorithm proposed in our previous work [Lee et al., 2009] is used.

The translator outlines each eligible kernel region into a CUDA kernel function and replaces the original region with a call to the function. The kernel-region transformation involves two important steps: *work partitioning* and *data mapping*. For work partitioning, each iteration of *omp for* loops and each section of *omp sections* are assigned to a thread, and remaining code sections in a kernel region are executed redundantly by all participating threads. To decide the *thread batching* for a kernel function, the translator calculates the maximum partition size among parallel work contained in the kernel region. By default, the maximum partition size becomes the total number of threads executing the kernel function. If a user explicitly sets the thread batching using directives or environment variables, the translator performs necessary tiling transformations to fit the work partition into the specified *thread batching*.

For data mapping, the translator exploits the information in the data-property constructs. For the data that are referenced in a kernel region, but not in a construct, the translator can determine their sharing attributes using OpenMP data sharing rules. Default data mapping follows the rule explained in Section 3.1.1. The CUDA memory model offers several specialized memory spaces; read-only shared data can be assigned to either *constant memory* or *texture memory* to exploit temporal locality through dedicated caches, and shared data with temporal locality can use fast memory spaces, such as *registers* and *shared memory*, as a cache. Even with no locality, passing read-only shared scalar variables as kernel arguments can reduce global memory traffic, since they will be placed on shared memory without involving global memory.

Because the CUDA memory model requires explicit memory transfers for threads executing a kernel function to access data on the CPU, the translator must insert necessary memory transfer calls for the *shared* and *threadprivate* data accessed by each kernel function. A basic strategy is to move all the shared data accessed by a kernel function from the CPU to the GPU and copy modified shared data back to the CPU after the kernel finishes. The data movement strategy for threadprivate data is decided by OpenMP semantics. However, the basic strategy may be inefficient in that the CPU may not use all shared data modified by GPU kernels, and the data in the GPU *global memory* may be persistent across kernel calls. To deal with these issues, we have developed several compiler optimizations, described in Section 3.3.

3.2 Transformation Techniques Supporting OpenMP-to-CUDA Translation

This section explains transformation techniques that are used to address various issues arising during the OpenMP-to-CUDA translation.

3.2.1 Upward-Exposed Private Variable Removal

As shown in Section 3.1.2, the baseline translation may split original parallel regions at each synchronization point to preserve a global synchronization under the CUDA programming model. However, this splitting may create an *upward exposed private variable problem (UEP problem)*. A variable is upward exposed if the variable is used before it is written (*defined*). In the OpenMP memory model, the value of a private variable is not defined upon entry to a parallel region unless the variable is in a *firstprivate* clause, and the value is also not defined upon exit from the region if the variable does not appear in a *lastprivate* clause. Therefore, private variables without firstprivate clauses should be defined first in a parallel region before they are used. When a parallel region is split into two sub-regions, a private variable becomes upward exposed if the variable is defined in the first sub-region but used in the second sub-region. To resolve this problem, the definition statement of the private variable should be moved out of the enclosing sub-region, and the private variable should be changed to a firstprivate variable in both sub-regions.

A transformation algorithm is shown in Figure 1. To find UEP variables, a traditional live analysis is performed; live-in private variables upon entry of a kernel region are the UEP variables for the kernel region. However, some private variables may become upward exposed after the kernel region is transformed into a kernel function, if they are used in the control part (initial statement, condition expression, or step expression) of any *omp-for* loop in the kernel region. The translator decides the *thread batching* of a kernel region by calculating the maximum partition size among parallel work contained the kernel region, and the resulting code for the thread batching is inserted somewhere before the kernel function call site. (Actual insertion point can vary depending on related optimizations.) Therefore, private variables that are used in the control part of any omp-for loop will be included in the thread batching calculation code, and depending on the insertion location of the code, the private variables may become upward exposed. The current algorithm conservatively includes these variables as potential UEP variables (lines from 8 to 10 in Figure 1). The next step is to find statements where the UEP variables are defined (*DEF statements*) and move them before their enclosing kernel regions, so that the DEF statements are reachable to kernel regions with UEP problems. However, we may not be able to move all DEF statements; all variables used in a DEF statement should be visible in the lexically enclosing block, the value of the UEP variable should be thread-independent, and moving the DEF statement out of the enclosing region should not change the program semantics. To check these conditions, the algorithm performs three tests:

- Check whether a kernel region containing a DEF statement has a UEP problem related to the DEF statement (lines 14 and 15 in Figure 1). If so,

```
Algorithm to remove upward−exposed private variables
Input: OpenMP program where kernel splitting algorithm is applied
Output: OpenMP program where upward−exposed private variables are removed
1  for each procedure p containing kernel regions
2      perform live analysis on p
3      for each kernel region KR in p
4          for each private variable s, which is live upon entry to KR
5              if ( s is not a reduction variable ) add S to UEPSymSet
6          for each omp−for loop L in KR
7              for each private variable s used in L
8                  if ( s is not a index variable of L )
9                      if ( s is used in initial statement, condition expression, or step expression of L )
10                         add s to UEPSymSet
11     for each private variable s in UEPSymSet
12         convType = 0
13         for each kernel region KR in p
14             if ( s is written in KR )
15                 if ( s is upward−exposed in KR ) convType = −3; break
16                 else if ( s is written in any loop in KR )
17                     if( (s is not used as index variable) and (s is not used as reduction variable) )
18                         convType = −1; break
19                 else if ( s is written in a simple statement SS in KR ) add SS into removeStmtSet
20                 else convType = −2; break
21         if ( convType < 0 ) continue
22         for each statement SS in removeStmtSet, move SS before the enclosing kernel region
23         for each kernel region KR where s is upward−exposed
24             remove s from OpenMP private clause; add s to OpenMP firstprivate clause
```

Figure 1 Transformation algorithm to remove upward exposed private variable problems

moving the DEF statement out of the kernel region may change the program semantics.

- Check whether a DEF statement resides in any of loops in the kernel region, and the corresponding UEP variable is neither used as index variable nor used as reduction variable (lines from 16 to 18 in Figure 1). If so, the UEP variable is likely to have a thread-dependent value, not suitable for DEF statement movement.
- Check whether a DEF statement is a simple assignment statement without conditions (lines 19 in Figure 1). If not, moving the DEF statement may alter the program semantics.

If all of these tests are passed, the algorithm will perform the code transformation; move DEF statements out of the enclosing parallel regions and change UEP variables from OpenMP *private* variables to *firstprivate* variables (lines from 22 to 24 in Figure 1). However, the safety tests conducted in the algorithm are not conservative enough to always guarantee correctness. Therefore, the compiler provides multiple options so that a user can choose and approve an appropriate transformation level.

3.2.2 Selective Procedure Cloning

This transformation selectively clones procedures that contain kernel regions or calls to procedures that contain kernel regions (possibly indirectly). This transformation is performed as a preprocessing step for context-sensitive, interprocedural analyses, such as the ones described in Section 3.3. This preprocessing ensures that every kernel function is called in a unique calling context, which supports context-sensitive analysis and translation.

The core algorithm of the selective procedure cloning transformation is shown in Figure 2; first, the algorithm traverses the *call graph* in post order and creates a list of procedures, where callee procedures come before their callers. Second, the algorithm finds the procedures that contain kernel regions and their caller procedures including indirect callers. These procedures are candidates for cloning; we include caller procedures, because if they are called multiple times, kernel regions in the callee procedures will be called in different contexts. Third, the algorithm clones procedures in the candidate list if they are called more than once. In this step, cloning should be performed first on the procedures that are higher in the call graph, since cloning of a caller procedure will result in multiple invocations of its callees.

An alternative to this selective cloning is to use a selective inlining method. However, inlining is more complex than cloning and may lose information at subroutine boundaries, leading to more conservative aliasing. Therefore, our translator uses cloning instead of inlining.

3.3 Compiler Optimizations

Compiler optimizations related to GPU memory accesses can be classified as follows: (1) techniques to optimize data movement between CPU and GPU, (2) techniques to optimize GPU global memory accesses, and (3) techniques to exploit GPU on-chip memories. In simple kernel programs, the first category may not be an issue; most previous work has focused on the last two categories. In our prior work [Lee et al., 2009], we have also identified several key transformation techniques to enable efficient GPU global memory access and exploit GPU on-chip memories. However, in experiments with

```
Selective procedure cloning algorithm
Input: OpenMP program where kernel splitting algorithm is applied
Output: OpenMP program where procedures are cloned so that each procedure
        containing kernel regions is called in a unique context
1  procedureList = a list of procedures in ascending order of call graph
2  for each procedure p in procedureList
3      if ( p contains kernel regions ) add p into callingProcList
4      while ( callingProcList is not empty )
5          c_p = callingProcList.removeFirst()
6          for each procedure p_p which calls c_p
7              if ( (p_p is not a main procedure) and (p_p is not in visitedProcList) )
8                  add p_p into callingProcList; add p_p into visitedProcList
9          if ((c_p is called more than once) and (c_p is not in clonedProcList))
10             add c_p into clonedProcList in the order of procedureList
11         if ((c_p is called only once) and (c_p is not in mayClonedProcList))
12             add c_p into mayClonedProcList in the order of procedureList
13 while ( (clonedProcList is not empty) or (mayClonedProcList is not empty) )
14     if ( clonedProcList is not empty ) c_p = clonedProcList.removeLast()
15     else if ( mayClonedProcList is not empty ) c_p = mayClonedProcList.removeLast()
16     if ( c_p is called more than once )
17         for each function call fcall to c_p, clone c_p and replace fcall with a new call to the cloned procedure
```

Figure 2 Procedure cloning algorithm which clones procedures containing kernel regions interprocedurally

larger applications, which typically contain several kernel functions called in different procedures, we have found that data movements between the CPU and the GPU can be costly. We have developed several compile-time techniques to reduce this cost, described next.

Techniques to Optimize Data Movement between the CPU and the GPU: The basic data movement strategy is to transfer data accessed by a kernel function from the CPU to the GPU before the kernel function is called, and transfer back modified data from the GPU to the CPU after the kernel function returns. However, if a compiler can determine that GPU global memory already has up-to-date data, they do not have to be copied again from the CPU. For this, we have developed an interprocedural data flow analysis that identifies *resident GPU variables*, which are the variables that reside in the GPU global memory and contain the same contents as the corresponding OpenMP *shared* variables in the CPU. The analysis algorithm recognizes if an OpenMP *shared* variable is used as a reduction variable in a kernel region and removes the variable from the resident GPU variable set. The rationale is that the translator implements reduction operations using a two-level tree reduction algorithm [CUDA_reduction], where the final reduction is performed on the CPU; after the reduction operation finishes, only the CPU has the final reduction output. Moreover, if a read-only shared scalar variable is passed as a kernel argument for the current kernel execution, the variable is directly copied to the shared memory without using global memory. In this case, the variable is not added to the resident GPU variable set, since global memory may contain stale data.

We have developed another interprocedural data flow analysis to identify redundant memory transfers from the GPU to the CPU. We define a *live CPU variable* as the variable that resides in the CPU and may be potentially read before its next write. Even though a shared variable is modified by a kernel function, if it is not a live CPU variable at the exit of the kernel function, it does not have to be copied from the GPU to the CPU, since it will not be used by the CPU before it is modified again. We can not blindly apply a traditional live analysis, because the CUDA memory model has two separate address spaces, while a traditional live analysis assumes only one address space. More details on this analysis can be found in our previous work [Lee and Eigenmann, 2010].

The information obtained from these analyses is passed to the actual translator in the form of annotations, and the translator will perform necessary transformations depending on the passed information.

4 OpenMPC: OpenMP Extended for CUDA

OpenMPC extends the baseline programming system described in Section 3.1 by adding new directives and environment variables that allow users or automatic tuning systems to control the overall translation and apply various optimizations and related parameters in an abstract way. The directives are also internally used by the translation system to pass information among compiler passes.

4.1 Directive Extension

OpenMPC directives are used to annotate OpenMP parallel regions using the syntax common in OpenMP. The syntax of the OpenMPC directives is shown in Table 1.

Table 1 OpenMPC directive format

#pragma cuda gpurun [clause [,] clause]...]
#pragma cuda cpurun [clause [,] clause]...]
#pragma cuda nogpurun
#pragma cuda ainfo procname(pName) kernelid(kID)

Table 2 Brief description of OpenMPC clauses, which control kernel-specific thread batchings and optimizations

Clause	Description
maxnumofblocks(N)	Set Maximum number of CUDA thread blocks for a kernel
threadblocksize(N)	Set CUDA thread block size for a kernel
noloopcollapse	Do not apply Loop Collapsing optimization
noploopswap	Do not apply Parallel Loop-Swap optimization
noreductionunroll	Do not apply loop unrolling for in-block reduction

Table 3 Brief description of OpenMPC clauses, which control kernel-specific data caching strategies

Clause	Description
registerRO(list)	Cache R/O variables in the list on GPU registers
registerRW(list)	Cache R/W variables in the list on GPU registers
noregister(list)	Set the list of variables not to be cached on GPU registers
sharedRO(list)	Cache R/O variables in the list on GPU shared memory
sharedRW(list)	Cache R/W variables in the list on GPU shared memory
noshared(list)	Set the list of variables not to be cached on GPU shared memory
texture(list)	Cache variables in the list on GPU texture memory
notexture(list)	Set the list of variables not to be cached on GPU texture memory
constant(list)	Cache variables in the list on GPU constant memory
noconstant(list)	Set the list of variables not to be cached on GPU constant memory

Table 4 Brief description of OpenMPC clauses, which control data mapping or movement between CPU and GPU. These clauses are used either internally by a compiler framework or externally by a manual tuner.

Clause	Description
c2gmemtr(list)	Set the list of variables to be transferred from a CPU to a GPU
noc2gmemtr(list)	Set the list of variables not to be transferred from a CPU to a GPU
g2cmemtr(list)	Set the list of variables to be transferred from a GPU to a CPU
nog2cmemtr(list)	Set the list of variables not to be transferred from a GPU to a CPU
nocudamalloc(list)	Set the list of variables not to be CUDA-mallocated
nocudafree(list)	Set the list of variables not to be CUDA-freed

Table 5 Brief description of OpenMPC environment variables, which control program-level behaviors of various optimizations, thread batchings, and translation configurations.

Parameter	Description
maxNumOfCudaThreadBlocks=N	Set the maximum number of CUDA thread blocks
cudaThreadBlockSize=N	Set the default CUDA thread block size
useMatrixTranspose	Apply Matrix Transpose optimization
useLoopCollapse	Apply Loop Collapsing optimization
useParallelLoopSwap	Apply Parallel Loop-Swap optimization
useUnrollingOnReduction	Apply loop unrolling for in-block reduction
useMallocPitch	Use cudaMallocPitch() for 2-dimensional arrays
useGlobalGMalloc	Allocate GPU variables as global variables
globalGMallocOpt	Apply CUDA malloc optimization for globally allocated GPU variables
cudaMallocOptLevel=N	Set CUDA malloc optimization level for GPU variables
cudaMemTrOptLevel=N	Set CUDA CPU-GPU memory transfer optimization level
assumeNonZeroTripLoops	Assume that all loops have non-zero iterations
convStatic2Global	Convert static variables in procedures except for main into global variables
disableCritical2ReductionConv	Disable critical-to-reduction conversion pass
UEPRemovalOptLevel=N	Set the level of upward exposed private variable removal optimization
MemTrOptOnLoops	Apply memory transfer promotion optimization
localRedVarConf=N	Configure how local reduction variables are generated for array-type variables
forceSyncKernelCall	Enforce explicit synchronization for each kernel call
addCudaErrorCheckingCode	Add CUDA-error-checking code right after each kernel call
addSafetyCheckingCode	Add GPU-memory-usage checking code just before each kernel call

The *gpurun* and *cpurun* directives are inserted either automatically by the compiler or manually by the user, while the *nogpurun* directive is inserted mostly by the user, and the *ainfo* directive is always inserted by the compiler. The *gpurun* directive specifies that the attached parallel region is eligible for kernel-region transformation. Clauses in Table 2, Table 3, and Table 4 may be used for this directive. The *gpurun* directive controls the translation of each kernel region. The *cpurun* directive says that the associated parallel region will be executed by the CPU. For this directive, the following four clauses from Table 4 can be used: *c2gmemtr*, *noc2gmemtr*, *g2cmemtr*, and *nog2cmemtr*. The *nogpurun* directive prevents the translator from transforming the attached kernel region. In our system, the *gpurun* directive is usually added by the automatic translator; it can be overridden by the *nogpurun* directive inserted by a user or a tuning system. The translator uses the

Table 6 Brief description of OpenMPC environment variables, which control program-level behaviors of data caching strategies.

Parameter	Description
shrdSclrCachingOnReg	Cache shared scalar variables onto GPU registers
shrdArryElmtCachingOnReg	Cache shared array elements onto GPU registers
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

Table 7 Brief description of OpenMPC environment variables, which control tuning-related configurations.

Parameter	Description
cudaUserDirectiveFile=filename	Set the name of file containing user directives
extractTuningParameters=filename	Extract tuning parameters applicable to a given input program
genTuningConfFiles=directory	Generate sets of tuning configuration files and user-directive files
defaultTuningConfFile=filename	Set the name of file containing default CUDA tuning configurations
tuningLevel=N	Set tuning level (0: Program-level tuning 1: Kernel-level tuning)

ainfo directive to assign unique IDs to each kernel region. This allows programmers and tuning systems to provide additional directives via a separate *user directive file*, rather than annotating the input OpenMP code. Directives provided in a user directive file have a similar syntax as that in Table 1, but are prefixed by the procedure name and kernel ID they refer to.

4.2 Environment Variable Extension

OpenMPC environment variables control the program-level behavior of various optimizations or execution configurations for an output CUDA program. The supported environment variables are summarized in Table 5, Table 6, and Table 7; variables in Table 5 control default behaviors of various optimizations and translations and set default thread batching for an input program. Those in Table 6 control program-level behaviors of data caching strategies, which may be overridden if each kernel region is annotated with data-caching clauses in Table 3. The ones in Table 7 are special variables used to set various tuning configurations. Because directives have priority over environment variables, users or tuning systems can alter the program-level optimizations and configurations for each kernel region.

5 Reference OpenMPC Compilation and Tuning System

We have developed a reference compilation and tuning system supporting OpenMPC by extending our previous compiler framework [Lee et al., 2009]. The new reference compiler has several distinct features compared to the previous framework: (1) all the optimizations that were applied manually have been automated, (2) new transformations and optimizations, including those in Section 3.2 and Section 3.3, are implemented, (3) an OpenMPC directive handler is added, (4) there is a clear separation between optimization passes and translation passes, and compiler passes communicate each other using the new directives, and (5) new transformation passes are implemented to perform the necessary code changes for each OpenMPC directive or environment variable.

The new compiler also includes capabilities for efficient tuning: a *search space pruner* and a *tuning configuration generator*. Using the compilation system, we also created a prototype tuning system, which builds a pruned optimization search space by analyzing the program and optional user settings. It then creates a path through the space and generates output CUDA code for each point in the search space.

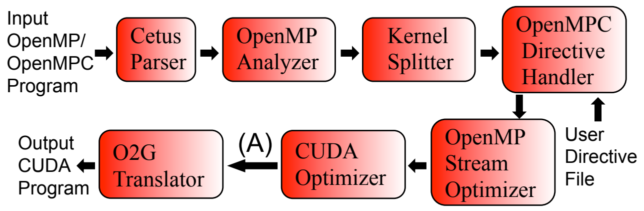


Figure 3 Overall compilation flow. For automatic tuning, additional passes are invoked between *CUDA Optimizer* and *O2G Translator*, marked as **(A)** in the figure (See Figure 4).

5.1 Overall Compilation Flow

Figure 3 illustrates the overall compilation flow. The *Cetus Parser* converts the input OpenMP/OpenMPC program into an internal representation (Cetus IR). The *OpenMP Analyzer* interprets standard OpenMP directives and analyzes the program to find all explicit/implicit OpenMP *shared*, *threadprivate*, *private*, and *reduction* variables. The analyzer also identifies implicit barriers defined by OpenMP semantics and replaces implicit synchronization points with explicit barrier statements. The *Kernel Splitter* splits parallel regions at each synchronization point to enforce synchronization semantics under the CUDA programming model. The *OpenMPC-directive Handler* processes OpenMPC directives present in the input program and annotates each *kernel region* with *ainfo* directives to assign a unique ID. The handler also parses a user directive file, if present. The *OpenMP Stream Optimizer* transforms CPU-oriented OpenMP programs into GPU-oriented ones, and the *CUDA Optimizer* performs CUDA-specific optimizations. Both optimization passes store their results in the form of OpenMPC directives in the Cetus IR. In the last pass, the *O2G Translator* performs the actual code transformations according to the directives inserted either by a user or by the optimization passes.

5.2 Compiler Support for Efficient Tuning

The OpenMPC directives and environment variables control various transformations and optimizations. Therefore, this set constitutes the basis of a tuning system. The compiler provides two important building blocks for efficient tuning systems: *search space pruning* and *tuning configuration generation*.

5.2.1 Search Space Pruning

A complete optimization search space may consist of all possible combinations of values of the OpenMPC directives and environment variables. For automatic tuning, however, only the clauses in Table 2 and Table 3 and the environment variables in Table 5 and Table 6 are used as tuning parameters. Clauses in Table 4 have a predictable effect – they are used either by a user or by the translator internally, and variables in Table 7 are used to control various tuning-related configurations. Even with the subset of directives and environment variables used for tuning, the overall optimization space can be too large to be searched. The *automatic search space pruning* function attempts to reduce this optimization space to a feasible size.

First, the *search space pruner* analyzes conditions necessary for applying each optimization and checks whether a given program has code sections satisfying the conditions. If no eligible code section is found, the optimization is removed from the optimization space. Second, the *pruner* suggests applicable caching methods for each variable that exhibits locality, based on the caching strategies described in our previous work [Lee et al., 2009].

The *search space pruner* may not be able to analyze the applicability of all parameters since the analysis may be too complex or sensitive to runtime inputs (i.e., unsafe). The pruner reports these parameters. In response, a user may decide and express the validity of these parameters in the *optimization-space-setup*, described next.

5.2.2 Tuning Configuration Generation

Once the search space pruner defines a pruned optimization space, the *configuration generator* creates tuning configuration files for each point in the search space. The configuration files are fed to the O2G translator, one at a time, generating output CUDA code. By default, the configuration generator builds tuning configurations for *program-level tuning*, but a user can choose more exhaustive *kernel-level tuning*, using an environment variable (*tuningLevel* in Table 7).

To reduce the space further, the user may provide an optional *optimization-space-setup* file to exclude specific parameters from the search space. The setup file can direct the tuning system to choose aggressive optimizations, which otherwise might be unsafe. Additionally, the setup file may contain the search ranges of important tuning values, such as the number of thread blocks and thread block size.

5.3 Prototype Tuning System

We have created a prototype tuning system (shown in Figure 4) using the tuning tools described in the previous section. The overall tuning procedure is as follows:

1. The *search space pruner* analyzes an input OpenMPC program plus optional user settings, and suggests applicable tuning parameters.

2. The *tuning configuration generator* builds a search space, further prunes the space using the *optimization space setup file* if user-provided, and generates tuning configuration files for the given search space.

3. For each tuning configuration, the *O2G translator* generates an output CUDA program.

4. The tuning engine produces executables from the generated CUDA programs and measures their performance by executing them.

5. The tuning engine decides a direction to the next search and requests the *configuration generator* to generate new configurations.

6. The last three steps are repeated, as needed.

In the proposed tuning framework, a programmer can replace the tuning engine with any custom engine; all the other steps from finding applicable tuning parameters to generating code variants for each tuning configuration are automatically handled by the proposed system. In our prototype, we have developed a simple tuning engine, which performs exhaustive search. Tuning with such a simple algorithm is feasible for our benchmarks, because the automatic search-space pruner can effectively reduce the optimization search. There exist algorithms for more efficient search space navigation [Ryoo et al., 2008b, Pan and Eigenmann, 2008]; they could replace exhaustive search in our system.

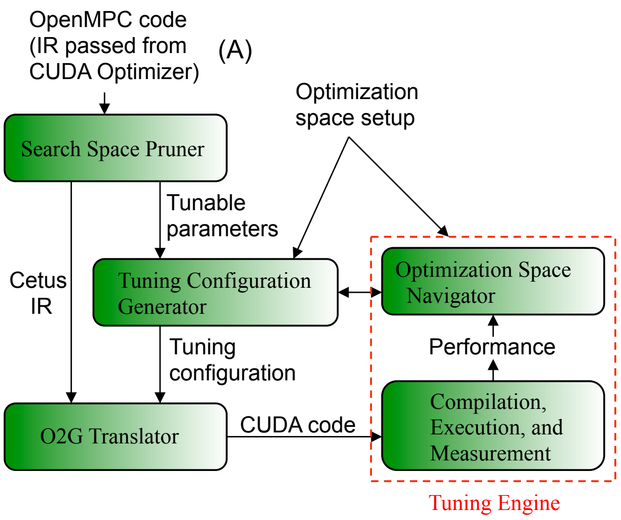

OpenMPC code (IR passed from CUDA Optimizer)
(A)
Optimization space setup
Search Space Pruner
Tunable parameters
Cetus IR
Tuning Configuration Generator
Optimization Space Navigator
Performance
Tuning configuration
O2G Translator
CUDA code
Compilation, Execution, and Measurement
Tuning Engine

Figure 4 Overall tuning framework, where input IR is passed from CUDA Optimizer in the compilation system (See Figure 3).

6 Evaluation

This section discusses the performance of the proposed OpenMPC system. Using the prototype tuning system, we have conducted two types of tuning experiments: *profile-based tuning (ProfT)* and *user-assisted tuning (UAT).* In profile-based tuning (*ProfT*), a target program is tuned with a *training* input data set – the smallest available set, in our case; the tuning system finds the best variant for the training input, and then the best variant is used to execute and measure the performance with the actual data sets of interest (referred to as *production* data). The profile-based tuning is fully automatic.

User-assisted tuning (*UAT*) is used to obtain an upper performance bound of our tuning system. The programs have been tuned for each production data set. In addition, the user assists the tuning system by confirming the applicability of aggressive optimizations. The other tuning processes are performed automatically.

In these experiments, fourteen OpenMP programs (two kernel benchmarks (*JACOBI* and *SPMUL*), three NAS OpenMP Parallel Benchmarks (*EP*, *CG*, and *FT*), and nine Rodinia Benchmarks [Che et al., 2009] (*BACKPROP*, *BFS*, *CFD*, *HEARTWALL*, *HOTSPOT*, *KMEANS*, *LUD*, *NW*, and *SRAD*)) were automatically transformed and tuned.

For comparison, three types of code variants of the tested programs were also evaluated: *Base*, *AllOpt*, and *Manual* versions. *Base* means CUDA programs translated by the proposed system without any optimization, *AllOpt* refers to the code variants where all safe optimizations are applied automatically by the compiler, and *Manual* represents hand-written CUDA versions. In creating the manual CUDA versions of the tested programs that do not have corresponding CUDA versions (*JACOBI*, *SPMUL*, *EP*, and *CG*), we have also used OpenMPC; we have first annotated each OpenMP source program using the OpenMPC directives and generated CUDA programs with our translator. We have then applied additional manual transformations to the generated CUDA programs, as possible.

The tested GPU device is an NVIDIA Quadro FX 5600 GPU, which has 16 multiprocessors (SMs) clocked at 1.35 GHz and 1.5 GB of DRAM. 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 DRAM. The translated CUDA programs were compiled using the NVIDIA CUDA Compiler (NVCC) and the serial versions of the input OpenMP programs were compiled using the GCC compiler version 4.2.2, with option *-O3*.

Table 8 summarizes the performance improvements achieved by the tested tuning systems. Overall, we found the followings:

- User-assisted tuning using the described system increases the performance up to 4.23 times (1.19 times on average) over the un-tuned versions (*AllOpt*). When the input OpenMP programs were optionally modified by hand (we believe that most of the changes can be done automatically by an advanced compiler), we could improve the performance up to 7.71 times (1.24 times on average) over the un-tuned versions; the average performance gap between hand-written versions (*Manual*) and versions generated by our tuning system (*ModUAT*) is less than 25%. If we exclude one exceptional case (Rodinia Benchmark *LUD*), the average performance gap is less than 8%.

- The proposed search-space pruner is able to reduce the optimization search space effectively (98.7% on average).

- In some programs, profile-based tuning is highly sensitive to input data, motivating future work in runtime tuning methods.

The detailed results are presented in the following sections.

Table 8 Overall tuning performance. *Input Type* refers to the types of the translation input sources; *Mod.* means that the input OpenMP code is manually modified before fed to the translator. *All-Opt* versions are the ones where all safe optimizations are automatically applied. In A(B) format, B refers to the performance when the results of *LUD*, which shows the largest performance gap between tuned and manual versions, are excluded.

Input Type	Perf. Improvement over All-Opt Versions			Relative Performance over Manual Versions		
	MIN	MAX	AVG	MIN	MAX	AVG
Orig.	1	4.23	1.19	0.02	1.92	0.5 (0.58)
Mod.	1	7.71	1.24	0.02	2.68	0.75 (0.92)

Table 9 Optimization search space reduction by the search-space pruner for program-level tuning of the original OpenMP programs

Benchmark	# of Tuning Configurations		Search Space
	W/O pruning	W/ pruning	Reduction (%)
JACOBI	25600	100	99.61
SPMUL	16384	128	99.22
EP	21504	336	98.44
CG	6144	384	93.75
FT	16384	512	96.88
BACKPROP	65536	128	99.8
BFS	49152	96	99.8
CFD	49152	384	99.22
HEARTWALL	65536	512	99.22
SRAD	65536	1024	98.44
HOTSPOT	16384	256	98.44
KMEANS	65536	256	99.61
LUD	49152	48	99.9
NW	65536	32	99.95

6.1 Optimization Space Reduction

Table 9 shows the optimization search space reduction due to pruning. Aggressive parameters are pruned, unless the user confirms their validity. In all experiments, we have used program-level tuning. For programs with small number of kernels, kernel-level tuning would be feasible as well, despite our simple, exhaustive search engine. We have verified that the performance results of both methods are nearly equal for those small programs. Applying kernel-level tuning in programs with many kernels, such as *CG* and *FT*, would increase the search space significantly, motivating future work in advanced search space navigation [Ryoo et al., 2008b, Pan and Eigenmann, 2008].

6.2 Tuning Performance of Kernel Benchmarks and NAS Parallel Benchmarks

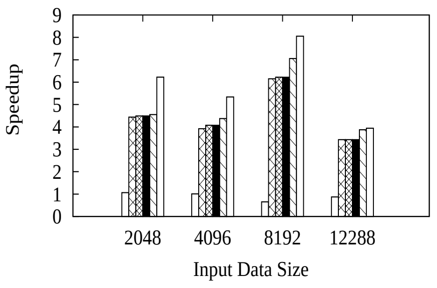
JACOBI is a stencil computation kernel used in many regular scientific applications, such as partial differential equation solvers. Even though *JACOBI* has a simple, regular access pattern, the base-translated GPU code performs poorly due to un-coalesced global memory accesses (*OrgBase* in Figure 5(a)). Our translator changes the access patterns to coalesced ones (*OrgAllOpt*). The results of profile-based tuning are shown as *OrgProfT* in the figure. User-assisted tuning (*OrgUAT*) shows the best performance that the proposed tuning system can achieve. The figure also includes the performance of production-tuning-only versions (*OrgProdT*); the performance difference between *OrgProdT* and *OrgUAT* indicates the additional gain achieved by applying unsafe, aggressive optimizations. The performance gap between the manual version (*Manual*) and the system-tuned version (*OrgUAT*) is due to a caching optimization using tiling transformation, which is not supported in the current translation system.

EP is a highly parallel application, which computes Gaussian deviates using pseudo-random numbers. Despite its parallelism, the base-translated version of EP performs poorly (*OrgBase* in Figure 5(b)), which again is due to un-coalesced global memory accesses (details in our previous work [Lee et al., 2009]). As in *JACOBI*, our translator removes this limitation (*OrgAllOpt*). In the case of *EP*, profile-based tuning is not effective. Our results indicate that the performance of some GPU applications is highly sensitive to the input data. In such cases, input-sensitive tuning systems, such as G-ADAPT [Liu et al., 2009], will perform better than profile-based tuning systems.

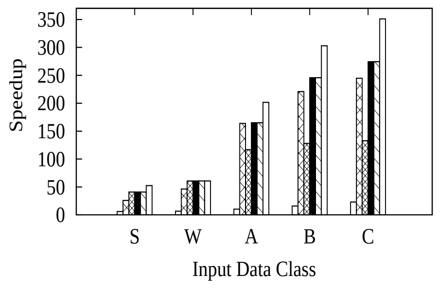
Our tuned programs (*OrgUAT*) do not always include all caching optimizations. For example, the *private array caching* optimization allocates a private array in *shared memory* to reduce long latencies to the CUDA *local memory*. However, this optimization is implemented by expanding the private array in the *shared memory*, which puts pressure on this memory due to its small size. Our compiler could perform nearly same optimizations as the manual version (*Manual*), but due to the inefficiency in handling array reduction patterns, the system-tuned versions (*OrgUAT*) show less speedups than the manual versions. An advanced array section analysis technique would be able to reduce this performance gap.

Sparse matrix computation is used in many scientific applications. *SPMUL* and *CG* are two important irregular programs performing sparse matrix computation. Sparse matrix computations tend to exhibit irregular computation and communication behavior; our results in Figure 5(c) show that profile-based tuning is not very successful. One interesting point about *SPMUL* is that none of the tuned program variants for any input had *loop collapsing* applied (details in our previous work [Lee et al., 2009]), even though this optimization was selected by most of the tuned variants of *CG*. *Loop collapsing* enables coalesced accesses to global memory by combining two nested sparse computation loops into one; additionally it caches shared data in the shared memory to reduce global memory accesses. However, the optimization increases the usage of shared memory and avoids exploiting the texture memory. Therefore, the overall benefit of the optimization is not statically predictable, making it amenable to tuning. In *SPMUL* case, the proposed translation system could perform the same optimizations as the manual version (*Manual* in the figure).

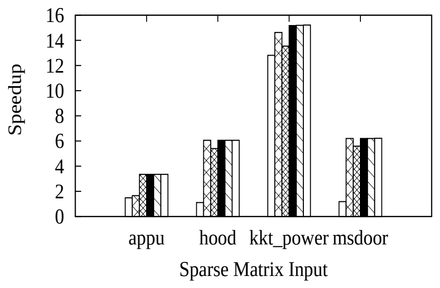
CG is a more challenging sparse matrix computation program. In *CG*, many kernel regions span across several procedures, resulting in complex memory transfer patterns between the CPU and the GPU. Interprocedural data flow analyses presented in Section 3.3 play a key role in creating efficient memory transfer patterns (*OrgAllOpt* in Figure 5(d)). In *CG*, applying aggressive optimizations increases the overall performance (*OrgUAT*), since the aggressive optimizations augment the accuracy of CUDA memory-related optimizations. The GPU version of *CG* also shows input-sensitive performance behavior, and thus profile-based tuning was not effective (*OrgProfT*). In *CG*, the manual version (*Manual*) applies more efficient GPU memory allocation and data-transfer schemes than



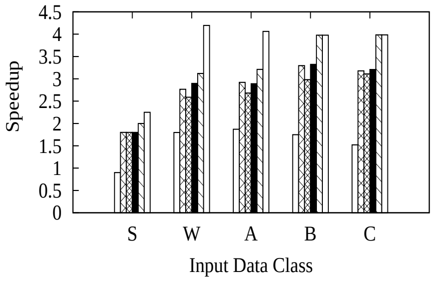
(a) JACOBI Kernel



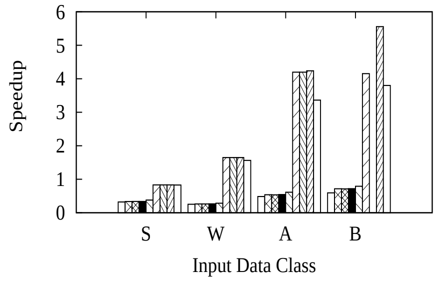
(b) NAS Parallel Benchmark EP



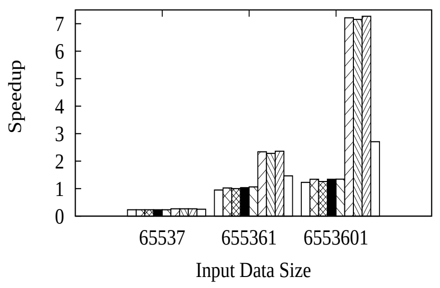
(c) SPMUL Kernel



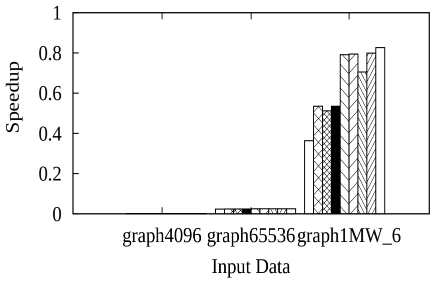
(d) NAS Parallel Benchmark CG



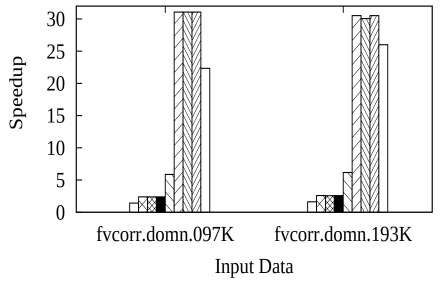
(e) NAS Parallel Benchmark FT



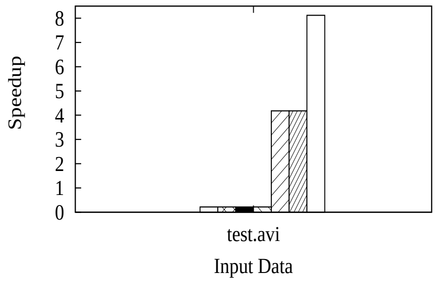
(f) Rodinia Benchmark BACKPROP



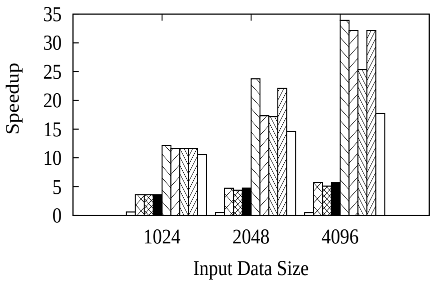
(g) Rodinia Benchmark BFS



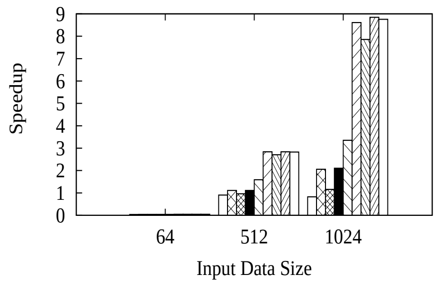
(h) Rodinia Benchmark CFD



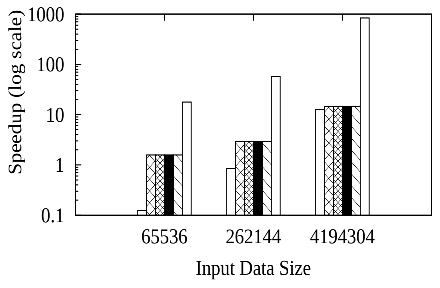
(i) Rodinia Benchmark HEARTWALL



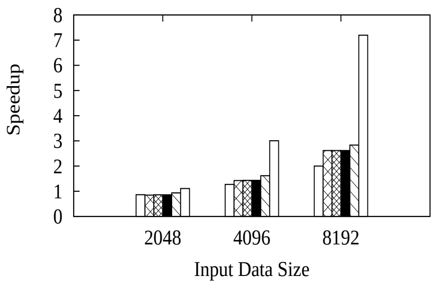
(j) Rodinia Benchmark SRAD



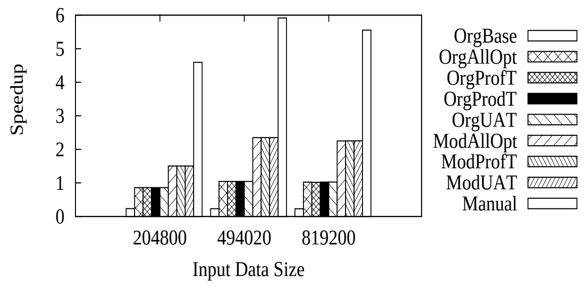
(k) Rodinia Benchmark HOTSPOT



(l) Rodinia Benchmark LUD



(m) Rodinia Benchmark NW



(n) Rodinia Benchmark KMEANS

Figure 5 Tuning performance (speedups over serial on the CPU). *OrgBase* is the translation without optimizations, *OrgAllOpt* applies all safe optimizations, which do not need user approval, and *OrgProfT* uses profile-based tuning. (Both *OrgBase* and *OrgAllOpt* represent speedups when the thread batching is pre-tuned.) Both *OrgProdT* and *OrgUAT* tune the programs with production data, but *OrgUAT* additionally applies aggressive optimizations under user approval. *ModAllOpt*, *ModProfT*, and *ModUAT* apply the same techniques as *OrgAllOpt*, *OrgProfT*, and *OrgUAT* respectively, except that in *Mod*-versions, the input OpenMP program was manually modified in a GPU-friendly way before the translation. *Manual* is the manually translated version.

the system-tuned version (*OrgUAT*), and the manual version also removes some of the implicit barriers, resulting in less kernel invocation overheads. This barrier removal is possible under the CUDA memory model, if two adjacent kernel regions are work-partitioned so that no two threads communicate with each other. The performance improvement by this manual overhead reduction is more pronounced for small input data sizes, as shown in Figure 5(d).

FT solves a 3-D partial differential equation using the Fast Fourier Transform (FFT). The original OpenMP code is heavily optimized for traditional, cache-based architectures, and thus resulting memory access patterns do not allow enough opportunity for coalesced memory accesses when translated into GPU code. Therefore, tuning the translation of the original OpenMP version does not give any noticeable performance improvement, even with various tunable parameters (*OrgUAT* in Figure 5(e)).

Mod versions (*ModAllOpt*, *ModProfT*, and *ModUAT*) refer to those where the input OpenMP program is manually modified, such that it has the same memory access patterns as the hand-written CUDA code (*Manual*). The modified versions allow several tuning opportunities for additional performance improvement; when the input data size is small, the best performance was achieved when all applicable caching optimizations were applied, even though the best thread batchings vary. As the input data size increases, however, not all caching optimizations were selected.

Another interesting finding in *FT* is that profile-based tuning (*ModProfT*) does not perform well if the input data changes drastically; when the input class is *B*, executing the best code variant selected by the profile-based tuning failed, due to excessive memory usage. This shows another example where traditional, profile-based tuning does not work well.

6.3 Tuning Performance of Rodinia Benchmarks

In *BACKPROP*, the main performance bottleneck is the uncoalesced memory access patterns caused by the layout of two-dimensional arrays. The uncoalesced access patterns may be changed by the *parallel loop-swap* technique proposed in our previous work [Lee et al., 2009], but the current compiler could not do so automatically, for lack of advanced analysis techniques. Figure 5(f) shows that improving the uncoalesced access patterns manually (Mod in the figure) gains significant performance. Even though the figure indicates that both the profile-based tuning (*ProfT*) and user-assisted tuning (*UAT*) perform no better than the versions where all applicable optimizations are blindly applied (*AllOpt*), tuning is still beneficial; different input data demand different thread batching and optimizations for the best performance, and the performance of the *AllOpt* versions also varies according to the thread batching.

BFS performs a breadth-first search, which is a commonly used graph algorithm. Even though the algorithm is simple, it has irregular access patterns, and the amount of computation is small for the given input sets. Therefore, GPU memory allocation and memory transfer times are dominant, allowing little room for performance improvement through tuning. Figure 5(g) show the tuning performance; due to its irregular, memory-intensive nature, none of the GPU versions perform better than the serial CPU versions.

CFD is an unstructured grid solver for the three-dimensional Euler equations for compressible flow. In *CFD*, applying aggressive optimizations to reduce redundant memory-transfers has a noticeable performance effect (*OrgUAT* in Figure 5(h)). Figure 5(h) shows that there are big performance gaps between the manual versions (*Manual*) and the automatic versions (*OrgUAT*). Uncoalesced memory accesses, which are difficult for the compiler to change automatically, contribute to the performance gap mostly. The uncoalesced access problem can be resolved by modifying the input OpenMP program (*Mod*), and then the compiler-translated versions could perform better than the hand-written CUDA codes, when the thread batching is properly tuned.

HEARTWALL is a program to track the movement of a mouse heart in response to a stimulus. If we parallelize the outermost loop of the main part, as done in the original OpenMP version, the translated code suffers from control flow divergences and uncoalesced memory accesses (*OrgUAT* in Figure 5(i)). The manual versions (*Manual*) use a complex thread batching scheme, where each iteration of the outermost loop processing sample points is assigned to a thread block, and iterations in the inner loops computing each pixel are mapped to threads in a thread block. We could get a similar effect by using the OpenMP *collapse* clauses, and *Mod* in the figure presents the speedups when the modified OpenMP program is translated and tuned.

The performance gap between *ModUAT* and *Manual* in the figure is largely due to the difference in handling synchronizations; in the manual version, the thread batching was applied in a way to minimize global synchronizations, and thus most synchronizations could be implemented using the *__syncthreads()* runtime library. However, in the OpenMP-to-CUDA translation system, the only way to express synchronization is to split kernels, and thus the automatically translated version (*ModUAT*) consists of many small kernels, incurring much more kernel invocation overhead than the manual version. The *HEARTWALL* case suggests that for achieving the best performance on some applications, low-level APIs, such as hiCUDA [Han and Abdelrahman, 2011], may be needed to express GPU-specific features.

SRAD is a diffusion algorithm based on partial differential equations and used to remove speckles in ultrasonic and radar imaging applications. Figure 5(j) shows the tuning performance on *SRAD*. The performance of *OrgUAT* reveals that applying unsafe, aggressive optimizations under user approval can increase the performance substantially. While the automatic versions rely on *parallel loop-swap* to enable coalesced accesses and use various caching optimizations, such as registers and texture cache for temporal locality, the manual versions (*Manual*) use tiling transformation with shared memory for both coalesced access and temporal locality. However, two-dimensional tiling and caching on the shared memory incur more control flow divergences and additional synchronizations; thus the overall performance improvement is less impressive than the automatic versions, which indicates that the optimal caching strategies depend on locality patterns. *SRAD* is a representative case showing that automatic optimizations combined with tuning (*OrgUAT*) can exceed the performance of the manually optimized versions.

HOTSPOT is a thermal simulation tool used to estimate processor temperature based on an architectural floorplan and simulated power measurements. The core part of *HOTSPOT* contains two nested loops, where the outermost loops are parallelized but the inner loops are also parallelizable. The automatically translated versions of the original OpenMP code (*Org* in Figure 5(k)) parallelize the outermost loops as OpenMP does, but the iteration spaces of the parallel loops are not big enough to create threads to hide the long global memory access latency. To increase the number of threads, the manual versions (*Manual*) apply a two-dimensional thread mapping to exploit the nested parallel loops. *Mod* in the figure shows the performance after we manually inserted the OpenMP *collapse* clauses to the nested parallel loops in the original OpenMP code; in effect, the translated kernels can be executed by a larger number of threads due to the increased iteration space. In the *HOTSPOT* case, the profile-based tuning (*OrgProfT* and *ModProfT*) does not work well, due to the different thread batching preference depending on the input size.

LUD is a simple matrix decomposition tool, whose main computation consists of two simple parallel loops. *LUD* does not have enough tuning opportunities; the performance effect of the thread batching is small, and there exist only a few applicable optimizations. However, *LUD* is a special case, since algorithmic changes specialized for the underlying GPU memory model can achieve surprisingly high speedup (*Manual* in Figure 5(l)), while there is little room for tuning. The large performance gap reveals that for some applications like *LUD*, customized algorithms considering the underlying architectures are essential for optimal performance.

Needleman-Wunsch (*NW*) is a global optimization method for DNA sequence alignment. Due to its simple structure, *NW* cannot benefit from tuning. Our experiments show that the same tuning configuration was selected for the best one regardless of the input size. Figure 5(m) presents the overall tuning performance; the performance gap between the manual versions (*Manual*) and the system-tuned versions (*OrgUAT*) is mainly caused by the lack of automatic tiling in the translation system.

KMEANS is a well-known clustering algorithm used for various data-mining applications. In *KMEANS*, even with user-assisted tuning (*ModUAT* in Figure 5(n)), there exists a big performance gap between the manual versions (*Manual*) and the system-tuned versions. We attribute the performance gap to the differences in implementing array reductions; in the automatic versions, private copies of a reduction array are used for each thread to keep partial reduction outputs, but the local copies are too big to be cached on the shared memory. However, the manual versions use a complex subscript manipulation so that each thread can update disjoint parts of the reduction array; this allows that threads in the same thread block share the same local reduction array, which is cacheable on the shared memory. To express these changes in the OpenMP code, special extensions to explicitly express the shared memory and GPU thread IDs will be needed.

7 Related Work

Several directive-based, GPU programming models have been proposed: PGI Accelerator [PGI_Accelerator], HMPP [HMPP], OpenMP for Accelerators [Beyer et al., 2011], OpenACC [OpenACC], hiCUDA [Han and Abdelrahman, 2011], CUDA-lite [Ueng et al., 2008]. These approaches are similar to ours in that they use directives to provide abstractions of CUDA, and a compiler automatically generates CUDA code by interpreting these directives. These models provide different levels of abstraction, and programming efforts required to conform to their models and optimize the performance also vary. Moreover, most of them focus on optimizing communication between CPU and GPU; these optimizations are explicitly expressed by programmers through directives. By contrast, our system supports various automatic performance optimizations including analysis techniques to minimize memory transfers between CPU and GPU. While previous work provides little control over the diverse compiler optimizations and GPU-specific features, our system offers an abstract tuning environment to control various optimizations and involved parameters.

Other related research [Baskaran et al., 2010, Leung et al., 2010] developed automatic serial-to-CUDA translation systems for affine programs. These approaches use polyhedral compilers to find affine transforms allowing optimal access patterns and data movements between CUDA off-chip and on-chip memories. By contrast, our compiler system optimizes both regular and irregular programs and supports optimizations to minimize data movement between CPU and GPU, in addition to the ones for efficient global memory accesses.

To help programmers find the best optimizations in the large, discontinuous CUDA optimization space, Ryoo et al. proposed model-based search space pruning techniques [Ryoo et al., 2008b]. Their techniques work well unless global memory bandwidth is a performance bottleneck. By contrast, our pruning algorithm reduces the search space by checking the applicability of each optimization. Their techniques can augment our framework by providing further pruning when the assumption holds, and our framework can also complement their work by automating their manual code conversions.

There exist a large body of work to automatically tune the performance of CUDA programs, but most previous work is application-specific; Datta et al. [Datta et al., 2008] proposed an auto-tuning environment and optimization strategies for stencil computations, Nukada et al. [Nukada and Matsuoka, 2009] presented an auto-tuning algorithm for 3-D FFT, and Volkov et al. [Volkov and Demmel, 2008] conducted an extensive tuning study for dense linear algebra. Unlike the previous contributions, G-ADAPT [Liu et al., 2009] is a compiler-based, general framework for input-sensitive tuning. This work is the closest to ours; G-ADAPT performs program transformations and optimization space search automatically, and offers a set of directives for programmers to specify search criteria. However, the adaptive framework works on a small subset of the optimization space, and thus the transformations that G-ADAPT can perform are limited. Our work is complementary to this work in that our compiler framework can offer a richer set of transformations and optimizations, and also support a larger number of directives than G-ADAPT; by using our translator, G-ADAPT could extend its optimization space. G-ADAPT limitation of working only on existing GPU programs could be relaxed by adopting our OpenMPC API as a front-end programming model.

8 Conclusion

This paper proposes a new programming interface, called OpenMPC, which is based on OpenMP but extended with a new set of directives and environment variables for efficient CUDA programming. OpenMPC provides programmers with abstractions of the complex CUDA programming and memory models and high-level control over the involved parameters and optimizations. We have developed a fully automatic compilation and user-assisted tuning system, which is able to suggest applicable tuning configurations for an input OpenMP program, generate CUDA code variants for each tuning configuration, and search the best optimizations for the generated CUDA program automatically. Experiments on fourteen applications from various scientific domains demonstrate that the proposed system achieves performance improvements comparable to hand-coded CUDA for various applications. However, a large performance gap between the manual versions and the system-tuned versions in some applications also reveals that further extensions to express the CUDA-specific execution model and memory model may be necessary to fully exploit the computing power of GPGPUs.

OpenMPC is a scientific tool and API to explore research directions for programming accelerators and developing compiler technology. As this article is getting finalized, several related directive proposals for accelerators are emerging. Our ultimate goal is not to propose a competing standard, but to develop the knowledge about important directives and optimizations for use by the community at large.

Acknowledgements

This work was supported, in part, by the Office of Advanced Computing Research, U.S. Department of Energy and the National Science Foundation under grants No. CNS-0720471, 0707931-CNS, 0833115-CCF, and 0916817-CCF.

References

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. *ACM International Conference on Supercomputing (ICS)*, 2008.

Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. *International Conference on Compiler Construction (CC)*, Volume6011/2010: 244–263, March 2010.

James C. Beyer, Eric J. Stotzer, Alistair Hart, and Bronis R. de Supinski. OpenMP for Accelerators. In *IWOMP'11*, pages 108–121, 2011.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

CUDA_reduction. NVIDIA CUDA SDK - Data-Parallel Algorithms: Parallel Reduction [online]. available: http://developer.download.nvidia.com/compute/cuda/1_1/Website/Data-Parallel_Algorithms.html.

Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9.

Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *IEEE Computer*, 42(12):36–42, 2009.

Tianyi David Han and Tarek S. Abdelrahman. hicuda: High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed Systems*, 22:78–90, 2011. ISSN 1045-9219. doi: http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.62.

HMPP. HMPP Workbench, a directive-based compiler for hybrid computing [Online]. Available: www.caps-entreprise.com/hmpp.html.

Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *SC'10: Proceedings of the 2010 ACM/IEEE conference on Supercomputing*. IEEE press, 2010.

Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 101–110, New York, NY, USA, February 2009. ACM. ISBN 978-1-60558-397-6.

Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 51–61, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-935-0. doi: http://doi.acm.org/10.1145/1735688.1735698. URL `http://doi.acm.org/10.1145/1735688.1735698`.

Yixun Liu, Eddy Z. Zhang, and Xipeng Shen. A cross-input adaptive framework for GPU program optimizations. *2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, 2009.

Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, pages 1–10, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8. doi: http://doi.acm.org/10.1145/1654059.1654090.

OpenACC. OpenACC: Directives for Accelerators [Online]. Available: http://www.openacc-standard.org.

OpenMP. OpenMP [Online]. Available: http://openmp.org/wp/.

Zhelong Pan and Rudolf Eigenmann. PEAK—a fast and effective performance tuning system via compiler optimization orchestration. *ACM Trans. Program. Lang. Syst.*, 30(3):1–43, 2008.

PGI_Accelerator. The Portland Group, PGI Fortran and C Accelarator Programming Model [Online]. Available: http://www.pgroup.com/resources/accel.htm.

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. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 73–82, 2008a.

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. *International Symposium on Code Generation and Optimization (CGO)*, 2008b.

S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu. CUDA-lite: Reducing GPU programming complexity. *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2008.

Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9.