OpenMPC: Extended OpenMP Programming and Tuning for GPUs

Seyong Lee and Rudolf Eigenmann
School of ECE, Purdue University
West Lafayette, IN 47907, USA
Email: {lee222,eigenman}@purdue.edu

***Abstract*—General-Purpose Graphics Processing Units (GPGPUs) are promising parallel platforms for high performance computing. The CUDA (Compute Unified Device Architecture) programming model provides improved programmability for general computing on GPGPUs. However, its unique execution model and memory model still pose significant challenges for developers of efficient GPGPU code. This paper proposes a new programming interface, called OpenMPC, which builds on OpenMP to provide an abstraction of the complex CUDA programming model and offers high-level controls of the involved parameters and optimizations. 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. Our results demonstrate that OpenMPC offers both programmability and tunability. Our system achieves 88% of the performance of the hand-coded CUDA programs.**

I. Introduction

General-Purpose Graphics Processing Units (GPGPUs) have emerged as promising building blocks for high-performance computing. While a GPGPU provides an inexpensive parallel computing system with higher throughput and performance than traditional CPUs, its programming complexity poses a significant challenge for developers. To improve the programmability of GPUs for general purpose computing, the CUDA (Compute Unified Device Architecture) programming model has been introduced, which abstracts a GPU as a general-purpose multi-threaded SIMD (Single Instruction, Multiple Data) architecture. Even though the CUDA programming model offers a more user-friendly interface, programming GPGPUs is still complex and error-prone, as CUDA exposes its unique memory model and execution model to programmers.

The OpenMP [1] API (Application Programming Interface) is a specification of compiler directives, library routines, and environment variables that provides an easy parallel programming model portable across shared memory architectures. To extend the ease of creating parallel applications with OpenMP to GPGPU architectures such as CUDA, we have previously developed an automatic OpenMP-to-CUDA translation framework [2]. It includes several optimizations that deal with the 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 execution stack used for CUDA compilation and execution limit the compiler's ability to predict the performance effect of its optimizations [3], [4]. In the OpenMP-to-CUDA translation framework, the CUDA programming model and memory model are transparent to users. However, this transparency comes at the cost of reduced control over fine-grained tuning. Achieving optimal performance with the generated programs may require additional, manual changes to the output CUDA code, which can be tedious and error-prone [4], [5], [6].

There has been extensive work on optimizing the performance of CUDA-based GPGPU programs. Studies on general optimization strategies found that the performance difference between well optimized GPU applications and poorly optimized ones can be orders of magnitude [3], [7], [8]. The studies also show that the level of effort and expertise required to obtain optimal application performance on GPGPUs can be very high. Even though there are several efforts to automatically optimize and tune the performance of GPGPU programs, most of them are either application-specific [10], [11], [12], restricted to certain types of applications [8], or applied to only a small subset of optimization parameters [4]. Therefore, achieving maximum performance for general GPGPU applications is still a challenge and usually involves manual work.

To overcome this challenge, we propose *OpenMPC* – OpenMP extended for CUDA. OpenMPC consists of 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 an API for improved CUDA programming, called OpenMPC, providing programmers with a high-level abstraction of the CUDA programming model. OpenMPC also provides a tuning environment that assists users in generating CUDA programs in many optimization variants without detailed knowledge of the programming and memory model.
- We have developed a reference compilation system to support OpenMPC by extending the framework proposed in our previous work [2]. The new framework is fully automated and parameterized; with OpenMPC directives and environment variables, users can gain fine-grained

SC10 November 2010, New Orleans, Louisiana, USA 978-1-4244-7558-2/10/$26.00

control over the OpenMP-to-CUDA translation and optimization.
- We have also 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 applicable optimization parameters to prune the optimization space that a tuning system should navigate in search of the best performance. Because this static analysis tool suggests applicable tuning parameters, programmers can tune a target program without deep knowledge of the program. 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 proposed compilation system and tuning tools. We have created GPU codes with various tuning configurations for a set of applications (NAS OpenMP Benchmarks EP and CG) and kernels (JACOBI and SPMUL). Our results show that the performance significantly depends on a program's input data. For best, user-assisted tuning, the OpenMPC codes improve the performance up to 102% (14% on average) over un-tuned versions, which is 88% of the performance of hand-written CUDA versions. Moreover, the search space pruner eliminates on average 98% of the optimization space for the tested programs.

The rest of this paper is organized as follows: Section II provides an overview of the GPGPU architecture and the CUDA programming model. Section III describes the baseline OpenMP-to-CUDA translation scheme and new optimizations added in this work. Section IV introduces OpenMPC, and Section V presents a reference compilation system and a prototype tuning framework supporting OpenMPC. Experimental results are shown in Section VI, and related work and conclusion are presented in Section VII and Section VIII, respectively.

II. Overview of GPGPU Architecture and CUDA Programming Model

GPGPUs supporting CUDA consist of a set of multiprocessors called *streaming multiprocessors (SMs)*, each of which contains a set of SIMD processing units called *streaming processors (SPs)*. Each SM has a fast on-chip *shared memory*, which is shared by SPs in the same SM, a fixed number of registers, which are logically partitioned among threads running on the 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 communications among different SMs.

The CUDA programming model is a general-purpose multithreaded SIMD model for GPGPU programming. In the CUDA programming model, code regions with rich data parallelism are implemented as a set of *kernel functions*, which are executed on the GPU by a number of threads in an SIMD fashion. Other code regions 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, which constitute a *thread batching*, are specified through language extensions at each kernel invocation.

In the CUDA memory model, *global memory*, *texture memory*, and *constant 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.

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

III. OpenMP-to-CUDA Translation and Optimization

This section gives an overview of the OpenMP-to-CUDA translation system, which performs a source-to-source conversion of a standard OpenMP program to a CUDA program and applies various optimizations to achieve high performance. This system has been built on top of the Cetus compiler infrastructure [13].

A. Baseline Translation of OpenMP-to-CUDA

The baseline translation consists of two steps: (1) interpreting OpenMP semantics under the CUDA programming 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 necessary memory transfer code to move data between CPU and GPU.

1) Interpretation of OpenMP Semantics under the CUDA Programming Model: OpenMP directives can be classified into four categories:

(a) *Parallel* construct (*omp parallel*) – this is the construct that 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*) – these constructs contain the only 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 flush*, *omp critical*, etc.) – these constructs contain explicit/implicit synchronization points. A parallel region must be split into two sub-regions at each of these constructs. The split is required to enforce a global synchronization in the CUDA programming model, as explained in Section II.

(d) Directives specifying data properties (*omp shared*, *omp private*, *omp threadprivate*, 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.

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, as explained above. Among the resulting sub-regions, the ones containing at least one work-sharing construct become kernel regions.

Once eligible kernel regions are identified, the translator outlines the regions into CUDA kernel functions and replaces the original regions with calls to these functions. The kernel-region transformation includes 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. Because the number of thread blocks and the thread block size determine the mapping of threads onto SMs (*thread batching*), these two parameters can be set through command line options or user directives. In this case, the translator performs necessary tiling transformations to fit the work partition into the specified *thread batching*.

For data mapping, the translator uses the information specified by OpenMP 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 III-A1 (d). Because the CUDA memory model allows several specialized memory spaces, certain data can take advantage of the specialized memory resources. Read-only shared data can be assigned to either *constant memory* or *texture memory* to exploit temporal locality through dedicated caches, and frequently reused shared data can use fast memory spaces, such as *registers* and *shared memory*, as a cache. Even though no locality exists, putting read-only shared scalar variables in shared memory can be beneficial, since it can reduce global memory traffic; passing read-only shared scalar variables as kernel arguments puts the data 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 that are accessed by kernel functions from the CPU to the GPU, and copy back the shared data that are modified by kernel functions. 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* are persistent across kernel calls. To deal with these issues, we have developed several compiler optimization techniques. They will be described in the following section.

B. Compiler Optimizations

Our translation system includes several optimizations of GPU memory accesses:

- Techniques to optimize data movement between CPU and GPU
- Techniques to optimize GPU global memory accesses
- 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 [2], 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 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 know 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 overall algorithm is shown in Figure 1. The 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 (*gResidentGVars*). The rationale is that the translator implements reduction operations using a two-level tree reduction algorithm [14], where the final reduction is performed on the CPU; after the reduction operation finishes,

```
Resident GPU Variable Analysis
Input: OpenMP program where Kernel Splitting Algorithm is applied
Output: OpenMP program annotated with OpenMPC clauses (noc2gmemtr)
gResidentGVars_in(program entry node) = {}
for( node m : predecessor nodes of a node n )
    gResidentGVars_in(n) ^= gResidentGVars_out(m) // ^ is an intersection operation
gResidentGVars_out(n) = gResidentGVars_in(n) + GEN(n) − KILL(n)
  where,
    GEN(n) = a set of shared variables whose GPU variables are globally allocated
                // If n is an exit node from a kernel region
             {} // Otherwise
    KILL(n) = a set of reduction variables in a kernel region // If n is an exit node from the kernel region
              a set of shared variables modified // If n represents a node in a CPU region
              a set of R/O shared scalar variables in a kernel region
                // If the variables do not exist in gResidentGVars_in set,
                // and if optimization to cache shared scalar variable on shared memory is on,
                // and if n is an exit node from the kernel region
              {} // Otherwise
```

Fig. 1. Interprocedural Analysis to Identify Resident GPU Variables, which does not include a function-call-handling part and an OpenMPC-clause (*noc2gmemtr*) generation part

```
Live CPU Variable Analysis
Input: OpenMP program where Kernel Splitting Algorithm is applied
Output: OpenMP program annotated with OpenMPC clauses (nog2cmemtr)
gLiveCPUVars_out(program exit node) = {}
for( node m : successor nodes of a node n )
    gLiveCPUVars_out(n) = += gLiveCPUVars_in(m) //+ is a union operation
gLiveCPUVars_in(n) = gLiveCPUVars_out(n) − KILL(n) + GEN(n)
    KILL(n) = a set of modified shared variables
    GEN(n) = a set of shared variables used in a node n // If n represents a node in a CPU region
```

Fig. 2. Interprocedural Analysis to Identify Live CPU Variables, which does not include a function-call-handling part and an OpenMPC-clause (*nog2cmemtr*) generation part

only the CPU has the final reduction output. Moreover, if a read-only shared scalar variable is cached on the GPU shared memory for the current kernel execution, the variable is directly copied to the shared memory through kernel-argument passing, which does not use 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. The overall algorithm of the new, modified live analysis is shown in Figure 2.

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.

IV. OpenMPC: Extended OpenMP for CUDA

OpenMPC extends the programming system described in the previous section by adding directives and environment variables that enable users and automatic tuning systems to apply CUDA-specific optimizations. The OpenMPC optimization system uses these directives to pass information generated by various analysis passes to the actual OpenMP-to-CUDA translator.

A. Directive Extension

The format of the OpenMPC directives is shown in Table I.

TABLE I
OpenMPC Directive Format

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

The directives in the table are used to annotate OpenMP parallel regions using the syntax common in OpenMP. The *gpurun* directive specifies that the attached parallel region is eligible for kernel-region transformation. Clauses that may be used for this directive are shown in Table II and Table III.

TABLE II
BRIEF DESCRIPTION OF OPENMPC CLAUSES, WHICH CONTROL KERNEL-SPECIFIC THREAD BATCHINGS, DATA MAPPING STRATEGIES, AND OPTIMIZATIONS

Clause	Description	Category
maxnumofblocks(N)	Set Maximum number of CUDA thread blocks for a kernel	CUDA Thread Batching
threadblocksize(N)	Set CUDA thread block size for a kernel	CUDA Thread Batching
registerRO(list)	Cache R/O variables in the list onto GPU registers	OpenMP-to-CUDA Data Mapping
registerRW(list)	Cache R/W variables in the list onto GPU registers	OpenMP-to-CUDA Data Mapping
sharedRO(list)	Cache R/O variables in the list onto GPU shared memory	OpenMP-to-CUDA Data Mapping
sharedRW(list)	Cache R/W variables in the list onto GPU shared memory	OpenMP-to-CUDA Data Mapping
texture(list)	Cache variables in the list onto GPU texture memory	OpenMP-to-CUDA Data Mapping
constant(list)	Cache variables in the list onto GPU constant memory	OpenMP-to-CUDA Data Mapping
noloopcollapse	Do not apply Loop Collapse optimization	OpenMP Stream Optimization
noploopswap	Do not apply Parallel Loop-Swap optimization	OpenMP Stream Optimization
noreductionunroll	Do not apply loop unrolling for in-block reduction	CUDA Optimization
nogpurun	Do not run the kernel region on a GPU	Execution Configuration

TABLE III
BRIEF DESCRIPTION OF ADDITIONAL OPENMPC CLAUSES, WHICH ARE USED EITHER INTERNALLY BY A COMPILER FRAMEWORK OR EXTERNALLY BY A MANUAL TUNER.

Clause	Description	Category
c2gmemtr(list)	Set the list of variables to be transferred from a CPU to a GPU	Data Movement between CPU and GPU
noc2gmemtr(list)	Set the list of variables not to be transferred from a CPU to a GPU	Data Movement between CPU and GPU
g2cmemtr(list)	Set the list of variables to be transferred from a GPU to a CPU	Data Movement between CPU and GPU
nog2cmemtr(list)	Set the list of variables not to be transferred from a GPU to a CPU	Data Movement between CPU and GPU
noregister(list)	Set the list of variables not to be cached on GPU registers	OpenMP-to-CUDA Data Mapping
noshared(list)	Set the list of variables not to be cached on GPU shared memory	OpenMP-to-CUDA Data Mapping
notexture(list)	Set the list of variables not to be cached on GPU texture memory	OpenMP-to-CUDA Data Mapping
noconstant(list)	Set the list of variables not to be cached on GPU constant memory	OpenMP-to-CUDA Data Mapping
nocudamalloc(list)	Set the list of variables not to be CUDA-mallocated	OpenMP-to-CUDA Data Mapping
nocudafree(list)	Set the list of variables not to be CUDA-freed	OpenMP-to-CUDA Data Mapping

TABLE IV
BRIEF DESCRIPTION OF OPENMPC ENVIRONMENT VARIABLES, WHICH CONTROL PROGRAM-LEVEL BEHAVIORS OF VARIOUS OPTIMIZATIONS, THREAD BATCHINGS, DATA MAPPING STRATEGIES, AND TUNING LEVEL.

Parameter	Description	Category
maxNumOfCudaThreadBlocks=N	Set the maximum number of CUDA thread blocks	CUDA Thread Batching
cudaThreadBlockSize=N	Set the default CUDA thread block size	CUDA Thread Batching
shrdSclrCachingOnReg	Cache shared scalar variables onto GPU registers	OpenMP-to-CUDA Data Mapping
shrdArryElmtCachingOnReg	Cache shared array elements onto GPU registers	OpenMP-to-CUDA Data Mapping
shrdSclrCachingOnSM	Cache shared scalar variables onto GPU shared memory	OpenMP-to-CUDA Data Mapping
prvtArryCachingOnSM	Cache private array variables onto GPU shared memory	OpenMP-to-CUDA Data Mapping
shrdArryCachingOnTM	Cache 1-dimensional, R/O shared array variables onto GPU texture memory	OpenMP-to-CUDA Data Mapping
shrdCachingOnConst	Cache R/O shared variables onto GPU constant memory	OpenMP-to-CUDA Data Mapping
useMatrixTranspose	Apply Matrix Transpose optimization	OpenMP Stream Optimization
useLoopCollapse	Apply LoopCollapse optimization	OpenMP Stream Optimization
useParallelLoopSwap	Apply Parallel Loop-Swap optimization	OpenMP Stream Optimization
useUnrollingOnReduction	Apply loop unrolling for in-block reduction	CUDA Optimization
useMallocPitch	Use cudaMallocPitch() for 2-dimensional arrays	CUDA Optimization
useGlobalGMalloc	Allocate GPU variables as global variables	CUDA Optimization
globalGMallocOpt	Apply CUDA malloc optimization for globally allocated GPU variables	CUDA Optimization
cudaMallocOptLevel=N	Set CUDA malloc optimization level for locally allocated GPU variables	CUDA Optimization
cudaMemTrOptLevel=N	Set CUDA CPU-GPU memory transfer optimization level	CUDA Optimization
assumeNonZeroTripLoops	Assume that all loops have non-zero iterations	Optimization Configuration
tuningLevel=N	Set tuning level (0: Program-level tuning 1: Kernel-level tuning)	Tuning Configuration

The *gpurun* directive can control 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 III can be used: *c2gmemtr*, *noc2gmemtr*, *g2cmemtr*, and *nog2cmemtr*. The third directive (*nogpurun*) 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 a *nogpurun* directive inserted by a user or tuning system. The translator uses the *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 in Table I, but are prefixed by the procedure name and kernel ID they refer to.

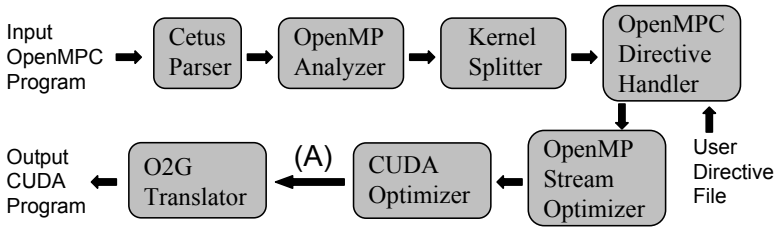


Fig. 3. Overall Compilation Flow. When the compilation system is used for automatic tuning, additional passes are invoked between *CUDA Optimizer* and *O2G Translator*, marked as **(A)** in the figure (See Figure 4)

B. Environment Variable Extension

The OpenMPC provides a rich set of environment variables, which control the program-level behavior of various optimizations or execution configurations for an output CUDA program. Table IV shows the supported environment variables. Because directives have priority over environment variables, users or tuning systems can alter the program-level optimizations and configurations for each kernel region.

V. Compilation and Tuning System for OpenMPC

This section presents a reference compilation and tuning system supporting OpenMPC. To realize the compilation system, we have modified the compiler developed in [2] by (1) separating the OpenMP-to-CUDA translator from the CUDA optimizer, (2) adding an OpenMPC directive handler, (3) implementing all the optimizations that had been applied manually, (4) adding new optimizations, including the ones described in Section III-B, (5) implementing new transformation passes to perform the necessary code changes for each OpenMPC directive or environment variable, and (6) modifying existing optimization passes and the translator so they communicate with each other using the new directives.

The compiler also includes capabilities for tuning systems such as the one described in Section V-C. These capabilities include a *search space pruner* and a *tuning configuration generator*.

Using the compilation system, we also created a prototype tuning system, which builds an optimization search space with applicable optimizations 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. For our experiments, we have chosen a simple approach that visits each point in the space; that is, it exhaustively searches the space.

A. Overall Compilation Flow

Figure 3 shows the overall flow of the compilation. The *Cetus Parser* reads the input OpenMPC program and generates an internal representation (Cetus IR). The *OpenMP Analyzer* recognizes standard OpenMP directives and analyzes the program to find all OpenMP *shared*, *threadprivate*, *private*, and *reduction* variables that are explicitly and implicitly used in each parallel region. The analyzer also identifies implicit barriers by OpenMP semantics and adds explicit barrier statements at each implicit synchronization point. The *Kernel Splitter* divides parallel regions at each synchronization point to enforce synchronization semantics under the CUDA programming model. The *OpenMPC-directive Handler* annotates each *kernel region* with an *ainfo* directive to assign a unique ID and parses a user directive file, if present. The handler also processes possible OpenMPC directives present in the input program. The *OpenMP Stream Optimizer* transforms traditional CPU-oriented OpenMP programs into OpenMP programs optimized for GPGPUs, and the *CUDA Optimizer* performs CUDA-specific optimizations. Both optimization passes express 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 provided either by a user or by the optimization passes.

B. Compiler Support for Tuning

The OpenMPC system supports a rich set of directives and environment variables controlling the automatic translation and optimization. This set can be used as the basis of a tuning system. The *search space pruning* and *tuning configuration generation* functions serve that purpose. The system allows user input for certain aggressive optimizations.

1) Search Space Pruning: Each of the new directives and environment variables that OpenMPC supports controls either an optimization or a thread batching for a kernel execution. A complete optimization search space consists of all possible combinations of values of these directives and environment variables. For automatic tuning, only the clauses in Table II and the variables in Table IV are used. Clauses in Table III have a predictable effect – they are used either by a user or by the translator internally.

Because there are many directives and environment variables, the complete optimization space cannot be feasibly searched. Non-trivial CUDA programs contain many kernel functions, each of which can be controlled with the directive set individually. 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 strategies for each variable that exhibits locality. Table V shows caching strategies for each data type.

TABLE V
CACHING STRATEGIES. *Reg* DENOTES *Registers*, *CM* MEANS *Constant Memory*, *SM* IS *Shared Memory*, AND *TM* REPRESENTS *Texture Memory*.

Variable Type	Caching Strategy
R/O shared scalar w/o locality	SM
R/O shared scalar w/ locality	SM, CM, Reg
R/W shared scalar w/ locality	Reg, SM
R/W shared array element w/ locality	Reg
R/O 1-dimensional shared array	TM
R/W private array w/ locality	SM

The *search space pruner* may not be able to analyze the applicability of all parameters (e.g., *cudaMemTrOptLevel* and *assumeNonZeroTripLoops* in Table IV) because 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.

2) Tuning Configuration Generation: Once a search space is defined by the search space pruner, 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*. Using an OpenMPC environment variable (*tuningLevel*), a user can choose the more exhaustive *kernel-level tuning*.

To further prune the search space, the user can provide an *optimization-space-setup* file containing parameters that should or should not be part of the optimization search space. These settings can direct the tuning system to choose aggressive optimizations, which otherwise might be unsafe. Additionally, the setup file may contain the value ranges of important parameters such as thread block size and the number of thread blocks.

C. Prototype Tuning System

Using the described search space functions, we have created a prototype tuning system, shown in Figure 4. The overall tuning process is as follows:

- The *search space pruner* analyzes an input OpenMPC program plus optional user settings, which exist as annotations in the input program, and suggests applicable tuning parameters.
- 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.
- For each tuning configuration, the *O2G translator* generates an output CUDA program.
- The tuning engine produces executables from the generated CUDA programs and measures the performance of the CUDA programs by running the executables.
- The tuning engine decides a direction to the next search and requests the *configuration generator* to generate new configurations.
- The last three steps are repeated, as needed.

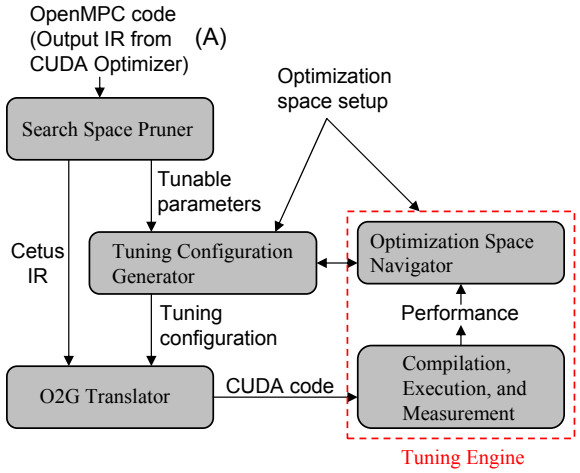


Fig. 4. Overall Tuning Framework. In the figure, input OpenMPC code is an output IR from CUDA Optimizer in the compilation system (See Figure 3)

In the example tuning framework, a programmer can replace the tuning engine with any custom engine; all the other steps from finding tunable parameters to complex code changes for each tuning configuration are automatically handled by the proposed compilation system. In our prototype, we have developed a simple tuning engine, which performs exhaustive search. Tuning with an exhaustive search algorithm is feasible for our benchmarks, because the automatic search-space pruner can effectively reduce the optimization search. Our search engine simply consists of a script that compiles CUDA codes for each configuration, runs executables, and measures their performance. Several algorithms for more efficient search space navigation exist [3], [15]; they could replace exhaustive search in our system.

VI. EVALUATION

To demonstrate the effectiveness of OpenMPC, we have conducted two types of performance tuning experiments, using the prototype tuning framework: *profile-based tuning* (*Profiled Tuning*) and *user-assisted tuning* (*U. Assisted Tuning*). In profile-based tuning (*Profiled Tuning*), 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 program with the actual data sets of interest (referred to as *production* data). The profile-based tuning is fully automatic.

User-assisted tuning (*U. Assisted Tuning*) 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, two regular OpenMP programs (NAS OpenMP Parallel Benchmark *EP* and *JACOBI* kernel) and two irregular OpenMP programs (NAS OpenMP Parallel Benchmark *CG* and *SPMUL* kernel) were automatically translated

and tuned. Our system is able to handle a larger class of programs, with some limitations. The translator produces appropriate warnings for unsupported program patterns.

For comparison, three types of code variants of the tested programs were also evaluated: *Baseline*, *All Opts*, and *Manual* versions. *Baseline* means CUDA programs translated by the proposed system without any optimization, *All Opts* refers to the code variants where all safe optimizations are applied, and *Manual* represents manually optimized versions. In creating the manual versions of the tested programs, 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. Creating these hand-coded reference code versions consumed substantial time.

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*.

The following sections present our results in detail. Overall, we found that: (1) user-assisted tuning using the described system increases the performance up to 102% (14% on average) over the un-tuned versions (*All Opts*), and the average performance gap between hand-written versions (*Manual*) and versions generated by our tuning system (*U. Assisted Tuning*) is less than 12%, (2) the proposed search-space pruner is able to reduce the optimization search space effectively (98% on average), and (3) in some programs, profile-based tuning is highly sensitive to input data, motivating future work in runtime tuning methods.

TABLE VI
NUMBER OF PARAMETERS SUGGESTED BY THE SEARCH-SPACE PRUNER AND THE NUMBER OF KERNEL REGIONS. IN *A*/*B*/*C* FORMAT, *A* IS THE NUMBER OF TUNABLE PROGRAM-LEVEL PARAMETERS, *B* IS THE NUMBER OF PARAMETERS THAT THE PRUNER SUGGESTS TO BE ALWAYS BENEFICIAL, AND *C* IS THE NUMBER OF PARAMETERS THAT A USER'S APPROVAL IS REQUIRED.

Benchmark	Program-level Parameter	Kernel-level Parameter	# of kernel regions
JACOBI	3/4/1	1	2
SPMUL	4/3/2	4	2
EP	5/3/2	3	1
CG	8/3/2	5	19

A. Optimization Space Reduction

Table VI lists the number of applicable tuning parameters suggested by the search-space pruner, and Table VII 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. Because of the small size of JACOBI, SPMUL, and EP, kernel-level tuning would be feasible as well, despite our simple, exhaustive search engine; we have verified that the performance of both methods are nearly equal. Applying kernel-level tuning in CG would increase the search space significantly, motivating future work in advanced search space navigations [3], [15].

TABLE VII
OPTIMIZATION SEARCH SPACE REDUCTION BY THE SEARCH-SPACE PRUNER FOR PROGRAM-LEVEL TUNING

Benchmark	Number of Tuning Configurations		Search Space Reduction (%)
	W/O pruning	W/ pruning	
JACOBI	25600	100	99.61
SPMUL	16384	128	99.22
EP	21504	336	98.44
CG	6144	384	93.75

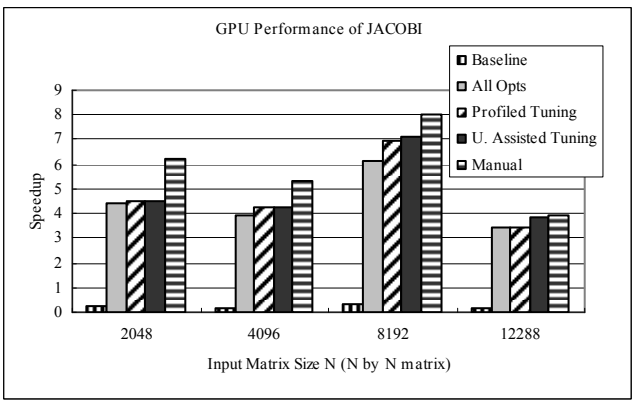
B. Performance of Regular Programs

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 uncoalesced global memory accesses (*Baseline* in Figure 5(a)). Our translator changes the access patterns to coalesced ones (*All Opts* in Figure 5(a)). The results of profile-based tuning are shown as *Profiled Tuning* in Figure 5(a). User-assisted tuning (*U. Assisted Tuning* in the figure) shows the best performance that the proposed tuning system can achieve. The manual versions (*Manual*) use tiling transformations to exploit shared memory, which is not yet supported by the current translator. We attribute the performance difference between versions generated by hand and by our tuning system primarily to this reason.

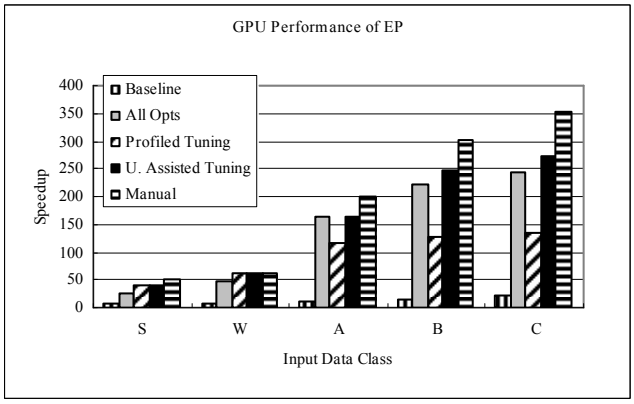
EP is a highly parallel application, which computes Gaussian deviates using pseudo-random numbers. Despite its parallelism, the base-translated version of EP performs poorly (*Baseline* in Figure 5(b)), which again is due to un-coalesced global memory accesses (details in [2]). As in JACOBI, our translator removes this limitation (*All Opts* in Figure 5(b)). 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 [4], will perform better than profile-based tuning systems.

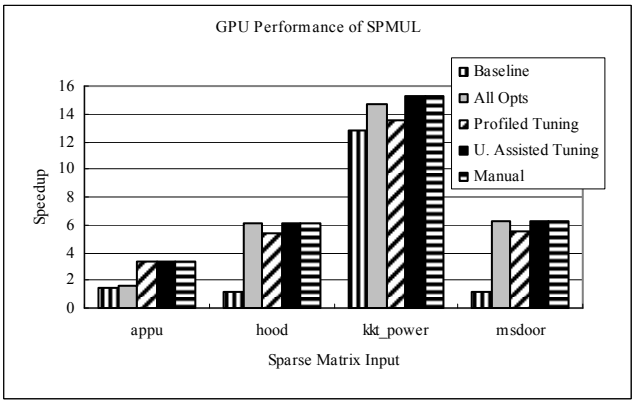
Our tuned programs (*U. Assisted Tuning* in Figure 5(b)) do not always include all cache optimizations. For example, the *private array caching* optimization allocates a private array in *shared memory* to reduce long latencies to the CUDA *local memory*. However, this optimization is implemented by expanding the private array in the *shared memory*, which puts pressure on this memory due to its small size. The performance gap between *U. Assisted Tuning* and *Manual* in Figure 5(b) is due to the difference in handling a *critical* section; *EP* uses an OpenMP *critical* construct to implement an array reduction under the OpenMP programming model. Both hand-written and system-tuned versions transform the critical section into



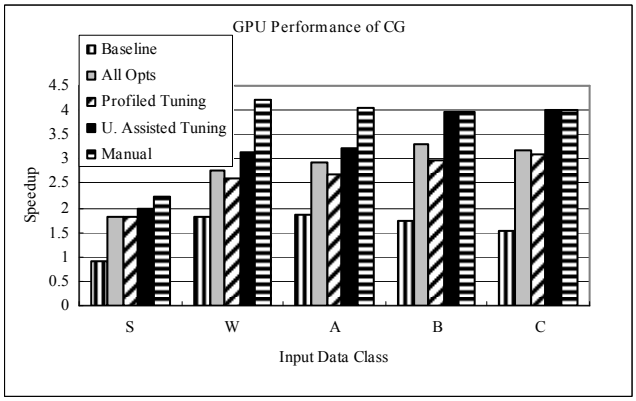
(a) JACOBI Kernel



(b) NAS Parallel Benchmark EP



(c) SPMUL Kernel



(d) NAS Parallel Benchmark CG

Fig. 5. Performance of Both Regular (*JACOBI* and *EP*) and Irregular (*SPMUL* and *CG*) Programs (Speedups are over serial on the CPU). *Baseline* is the translation without optimizations, and *All Opts* applies all safe optimizations, which do not need a user's approval. *Profiled Tuning* uses profile-based tuning, *U. Assisted Tuning* is a user-assisted tuning, which tunes the programs with production data and applies aggressive optimizations under the user's approval, and *Manual* is the manually optimized version.

array reduction code, but the manual version optimizes further by removing a redundant private array, which was used as a local reduction variable. Improved array section analysis would be able to detect this redundancy.

C. Performance of Irregular Programs

Sparse matrix computation is used in many scientific applications. *SPMUL* and *CG* are two important irregular programs performing sparse matrix computation. To test the *SPMUL* kernel, we used several real sparse matrices in the UF Sparse Matrix Collection [9]. Sparse 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 [2]), 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. Figure 5(c) shows that the version tuned by our system (*U. Assisted Tuning*) achieves the same performance as the manual version.

CG is a more challenging sparse 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 analysis presented in Section III-B plays a key role in creating efficient memory transfer patterns (*All Opts* in Figure 5(d)). In *CG*, applying aggressive optimizations increases the overall performance (*U. Assisted Tuning*), since the aggressive optimizations augment the accuracy of CUDA memory-related optimizations. (In the other tested programs, no noticeable performance improvement was achieved by applying the aggressive optimizations.) The GPU version of *CG* also shows input-sensitive performance behavior, and thus profile-based tuning was not effective (*Profiled Tuning* in Figure 5(d)). In *CG*, the manual version (*Manual*) applies more efficient GPU memory allocation and data-transfer schemes than the system-tuned version (*U. Assisted Tuning*), 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).

VII. Related Work

Several automatic translation techniques have been proposed with the goal of increasing the productivity of CUDA programming. In the hiCUDA directive-based language [5], a set of directives express CUDA computation and data attributes in a sequential program. This work is similar to ours in that it uses directives to provide abstractions of CUDA, and a compiler automatically generates CUDA code by interpreting these directives. However, hiCUDA uses the same programming paradigm as CUDA; even though it hides the CUDA language syntax, the complexity of the CUDA programming and memory model is directly exposed to programmers. OpenMPC is based on OpenMP, which is higher-level than hiCUDA, and thus our work provides better programmability than hiCUDA. Moreover, hiCUDA does not provide any optimization, whereas our framework supports various automatic performance optimizations. hiCUDA can complement our work, as its exposure of the CUDA model offers the potential of finer-grain control over GPUs' performance. CUDA-lite [16] is another directive-based approach, which generates code for optimal tiling of global memory data. CUDA-lite is limited in that it supports automatic code translation only on existing CUDA programs. The approach can also complement our work by providing advanced tiling transformations for optimized global memory accesses; currently, the OpenMPC compiler performs tiling optimization only for work partitioning. Another approach [6] has developed an automatic code transformation system that generates CUDA code from sequential C source code, for affine programs. This approach uses a polyhedral compiler model to find affine transforms for optimizing data movement between CUDA off-chip and on-chip memories. By contrast, our compiler framework optimizes both regular and irregular programs and supports optimizations to minimize data movement between the CPU and the GPU, as well as the ones for efficient global memory accesses.

There have been many studies on optimizing the performance of CUDA-based GPGPU programs; Ryoo et al. [7] presented an experimental study on general optimization strategies for programs on a CUDA-supported GPGPU, but code generations were performed manually. Ryoo and his colleagues presented another study on optimization space pruning techniques [3]. Their techniques use a model-based approach and work well if global memory bandwidth is not 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.

To automatically optimize the performance of CUDA programs, most previous work was application-specific; Datta et al. [11] developed a number of optimization strategies and an auto-tuning environment for stencil computations. Nukada et al. [10] presented an auto-tuning algorithm for 3-D FFT, and Volkov et al. [12] conducted an extensive study of dense linear algebra, using auto-tuning techniques to achieve the best performance. Unlike the previous contributions, G-ADAPT [4] uses a compiler-based, adaptive framework, which automatically searches the best optimizations for a general GPU program on different input data sets. 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 framework performs automatic transformation in limited ways. 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 that provide control over translation and optimization parameters 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.

VIII. Conclusion

This paper describes a new programming interface, called OpenMPC, which consists of standard OpenMP and a new set of compiler directives and environment variables, extended for CUDA. OpenMPC addresses two important issues on GPGPU programming: *programmability* and *tunability*. OpenMPC as a front-end programming model provides programmers with abstractions of the complex CUDA programming model and high-level controls over various optimizations and CUDA-related parameters. 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 both regular and irregular programs demonstrate that the proposed system achieves performance improvements comparable to hand-coded CUDA.

Acknowledgment

This work was supported, in part, by the National Science Foundation under grants No. 0751153-CNS, 0707931-CNS, 0833115-CCF, and 0916817-CCF.

References

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

[2] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A compiler framework for automatic translation and optimization," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. New York, NY, USA: ACM, Feb. 2009, pp. 101–110.

[3] 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)*, 2008.

[4] Y. Liu, E. Z. Zhang, and X. Shen, "A cross-input adaptive framework for GPU program optimizations," *2009 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–10, 2009.

[5] T. D. Han and T. S. Abdelrahman, "hiCUDA: a high-level directive-based language for GPU programming," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009, pp. 52–61.

[6] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic C-to-CUDA code generation for affine programs," *International Conference on Compiler Construction (CC)*, vol. Volume6011/2010, pp. 244–263, March 2010.

[7] 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)*, pp. 73–82, 2008.

[8] 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.

[9] T. Davis, "University of Florida Sparse Matrix Collection [Online]. Available: http://www.cise.ufl.edu/research/sparse/matrices/."

[10] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," in *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 1–10.

[11] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. 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*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[12] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[13] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *IEEE Computer*, vol. 42, no. 12, pp. 36–42, 2009.

[14] "NVIDIA CUDA SDK - Data-Parallel Algorithms: Parallel Reduction [Online]. Available: http://developer.download.nvidia.com/compute/ cuda/1_1/Website/Data-Parallel_Algorithms.html."

[15] Z. Pan and R. Eigenmann, "PEAK—a fast and effective performance tuning system via compiler optimization orchestration," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 3, pp. 1–43, 2008.

[16] 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.