A Script-Based Autotuning Compiler System to Generate High-Performance CUDA Code

MALIK KHAN, University of Southern California/ISI, NUST
PROTONU BASU, GABE RUDY, MARY HALL, and CHUN CHEN, University of Utah
JACQUELINE CHAME, USC/Information Sciences Institute

This article presents a novel compiler framework for CUDA code generation. The compiler structure is designed to support *autotuning*, which employs empirical techniques to evaluate a set of alternative mappings of computation kernels and select the mapping that obtains the best performance. This article introduces a *Transformation Strategy Generator*, a meta-optimizer that generates a set of *transformation recipes*, which are descriptions of the mapping of the sequential code to parallel CUDA code. These recipes comprise a search space of possible implementations. This system achieves performance comparable and sometimes better than manually tuned libraries and exceeds the performance of a state-of-the-art GPU compiler.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—Code generation, Compilers, Optimization; D.1.2 [Programming Techniques]: Automatic Programming; D.1.3 [Programming Techniques]: Concurrent Programming—Parallel Programming; I.3.1 [Computer Graphics]: Hardware Architecture—Graphic Processors

General Terms: Algorithms, Experimentation, Languages, Performance

ACM Reference Format:

Khan, M., Basu, P., Rudy, G., Hall, M., Chen, C., and Chame, J. 2013. A script-based autotuning compiler system to generate high-performance cuda code. *ACM Trans. Architect. Code Optim.* 9, 4, Article 31 (January 2013), 25 pages.

DOI = 10.1145/2400682.2400690 <http://doi.acm.org/10.1145/2400682.2400690>

1. INTRODUCTION

Peak performance of graphics processors (GPUs) is now in the TeraFlop range for a single device, and general-purpose code on GPUs has demonstrated up to two orders of magnitude performance gains as compared to conventional CPUs [Volkov and Demmel 2008; Datta et al. 2008; Bell and Garland 2009]. In spite of this enormous potential, it is very difficult to develop GPU applications that make efficient use of all the architectural features and achieve high performance, particularly given the significant architectural changes across GPU generations. The programmer must explicitly manage the heterogeneous memory hierarchy and two levels of parallelism. Further, subtle differences in how the software is mapped to GPUs have enormous impact on

This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. This research was funded in part by DARPA contract HR0011-10-9-0008, and National Science award CCF-1018881. Partial support for this work was provided through Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy Office of Advanced Scientific Computing Research under award number DE-SC0006947.

Authors' addresses: M. Khan (corresponding author), USC/ISI, Marina del Rey, CA and National University of Science and Technology, ISB, Pakistan; email: mmurtaza@usc.edu; P. Basu, G. Rudy, M. Hall, and C. Chen, University of Utah, Salt Lake City, UT; J. Chame, USC/Information Sciences Institute, Marina del Rey, CA. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/01-ART31 \$15.00

DOI 10.1145/2400682.2400690 <http://doi.acm.org/10.1145/2400682.2400690>

performance. Tools to simplify the programming and performance tuning process have the potential to increase the accessibility of this important technology and improve performance of the resulting code.

To this end, this article¹ describes a novel compiler framework that eases a programmer's burden in developing highly tuned implementations for a GPU. The compiler is designed to support *autotuning*, which employs empirical techniques to evaluate a set of alternative mappings of computation kernels and select the mapping that obtains the best performance. Autotuning software has achieved high performance for self-tuning libraries [Bilmes et al. 1997; Frigo 1999; Vuduc et al. 2005a; Whaley and Petitet 2005; Püschel et al. 2005], composing tuned applications from programmer-specified variants and parameters [Chung and Hollingsworth 2004; Ren et al. 2008; Ansel et al. 2009], and in compiler-directed autotuning [Chen et al. 2005a; Baskaran et al. 2008b; Pouchet et al. 2007, 2008] where compiler algorithms select which variants and parameters to evaluate. Autotuning is particularly valuable when targeting GPUs given that the best solution is greatly affected by subtle changes in the problem description or the architecture; the ability to explore empirically different data placement and thread/block mapping strategies, along with other code generation decisions, facilitates finding a high-performance solution. The goal of building an autotuning compiler that can evaluate multiple code variants and systematically generate code motivated us to structure our compiler in a completely different way from traditional compilers. In this article, we describe this novel script-based compiler framework as applied to the problem of automatic parallelization from sequential loop nest computations expressed in C to CUDA programs. Loop nest computations can be translated to *data-parallel* constructs in Nvidia's CUDA programming extensions, which are well supported by the architecture [Kirk and Hwu 2010]. The system employs autotuning on the automatically generated CUDA programs to compare performance and select the best implementation for a candidate architecture.

Other programming systems for GPUs are largely designed to achieve performance gains while increasing productivity of average programmers (for example, Han and Abdelrahman [2009], Lee et al. [2009], and Catanzaro et al. [2011]); our system in contrast is designed to increase productivity of developers seeking very high levels of performance and willing to expend extra programming effort and/or compile time to achieve it. Some other compiler-based solutions optimize explicitly parallel CUDA code [Ueng et al. 2008; Yang et al. 2010; Unkule et al. 2012], but require that the initial CUDA kernel be properly parallelized to facilitate optimizations. Other automatic parallelization systems for GPUs use a fixed optimization strategy [Baskaran et al. 2010; Leung et al. 2010; Amini et al. 2011], while our autotuning compiler can generate multiple code variants that can be evaluated empirically, leading to better performance. As compared to two prior GPU compiler efforts that make use of autotuning [Baskaran et al. 2008b; Cui et al. 2011], our compiler explores a richer optimization search space that encompasses the best ideas in manual tuning [Volkov and Demmel 2008; Nath et al. 2010], and can therefore find better optimized solutions.

The compiler framework is organized into several layers of abstraction. By design, a user of the system can operate at any of the layers, although in this article we focus on completely automatic generation of CUDA code from sequential C programs. We introduce in this article the highest layer, a *Transformation Strategy Generator (TSG)*, which generates a set of *transformation recipes* [[Hall et al. 2009], Donadio et al.]. Each compact recipe represents a different optimization strategy and is parameterized so that optimization parameters can be adjusted during autotuning. The set of recipes comprises a search space of possible implementations that can be automatically

¹New Paper, Not an Extension of a Conference Paper.

generated. A thin middle layer called CUDA-CHiLL takes as input these GPU transformation recipes (as Lua scripts) and generates the commands for the next layer [Rudy et al. 2011]. At the lowest layer, a polyhedral loop transformation and code generation framework called CHiLL performs standard loop transformations (not GPU-specific) [Chen 2007; Chen et al. 2008; Tiwari et al. 2009], and is accessed using a lower-level script.

The unique organization of our compiler facilitates achieving the high levels of performance we have demonstrated throughout our group's six-year history in building autotuning compiler technology, starting with a model-guided autotuning compiler algorithm targeting memory hierarchies [Chen et al. 2005a, 2005b], multimedia extensions [Chen et al. 2007b], and Chen's dissertation that introduced the CHiLL polyhedral transformation and code generation framework which underlies this work [Chen 2007]. More recently, we have worked closely with application developers to demonstrate performance improvements of production codes running on supercomputers [Chen et al. 2007a; Bailey et al. 2008; Shin et al. 2009, 2010; Chame et al. 2011] and have integrated our compiler with Active Harmony, which employs parallel search to explore very large optimization search spaces during autotuning [Tiwari et al. 2009, 2011]. The article extends this work significantly, and makes several contributions.

- We describe a TSG algorithm, a meta-optimizer that generates a collection of transformation recipes, which are used to transform the sequential code into parallel CUDA code. TSG's optimization process is uniquely decomposed into phases (computation decomposition, data placement, code variant selection, and optimization parameter selection), exploring a rich but manageable optimization search space of high-performance solutions.
- We demonstrate that our system organization facilitates *performance portability* across different execution contexts, yielding high performance on two very different GPU architecture generations, and across problem sizes and data types (e.g., single vs. double precision) by identifying different optimization parameters and sometimes different code variants.
- We show that the overall compiler approach is powerful, yielding performance results comparable to and sometimes exceeding manually tuned libraries that are widely used as standard reference points for performance comparisons. The system obtains higher performance compared to a previous GPU compiler based on PLUTO, on average 1.48X and 1.33x faster for the C2050 and GTX-280, respectively, for the set of benchmarks from Baskaran et al. [2010].

2. TRANSFORMATIONS AND CODE GENERATION

In this section we present the details of our compiler system organization and the GPU architectures. We describe how commonly used compiler transformations are adapted to generate CUDA code to direct both parallel computation partitioning and data partitioning, similar to other GPU automatic parallelization systems.

2.1. System Overview

The layers of the compiler framework are illustrated by Figure 1. The system takes as input sequential C code representing a single function comprised of loop nests. The TSG, the focus of this article, analyzes this code to understand its data dependences and reuse, and emits a set of *transformation recipes*, which express an optimization strategy as a sequence of composable transformations [Donadio et al. 2005; Hall et al. 2009]. The TSG expresses these recipes at a high level using the commands in Table I. A compiler abstraction and code generation layer CUDA-CHiLL takes these high-level transformation specifications tailored to CUDA code generation and generates the specific set of compiler transformations that must be composed to produce the CUDA

Fig. 1. System structure for automatic CUDA code generation.

Table I. Description of High-Level Commands Emitted by TSG

Command	Example Parameter	Description
tile_by_index	{ <i>"i"</i> , <i>"j"</i> }	The index variables of the loops that will be tiled.
	{ <i>TI</i> , <i>TJ</i> }	Variables representing the respective tile sizes for each index variable.
	{ <i>l1_ctrl</i> = <i>"ii"</i> , <i>l2_ctrl</i> = <i>"jj"</i> }	Specifies control loop variable names and optionally renames tile loop index variables.
	{ <i>"ii"</i> , <i>"jj"</i> , <i>"i"</i> , <i>"j"</i> }	Final order of nested loops with updated loop index names.
cudaize	<i>"mm.GPU"</i>	The name of the kernel function
	{ <i>a</i> = <i>N</i> , <i>b</i> = <i>N</i> , <i>c</i> = <i>N*N</i> }	The data sizes of the arrays if not statically determinable.
	{ <i>block</i> = <i>"ii"</i> , <i>thread</i> = <i>"jj"</i> }	Specifies block and thread indices. The bounds for these loops are used to derive grid dimensions.
copy_to_registers	<i>"kk"</i>	The loop level, given as an index variable, to place the copy.
	<i>"c"</i>	The array variable to be copied.
copy_to_shared	<i>"tx"</i>	The loop level, given as an index variable, to place the copy.
	<i>"b"</i>	The array variable to be copied.
	-16	Ensure the last dimension of the temporary array are coprime with 16 to pad.
copy_to_texture	<i>"b"</i>	The array variable to be accessed from texture memory.
copy_to_constant	<i>"b"</i>	The array variable to be accessed from constant memory.
unroll_to_level	1	Unrolls all statements up to specific level from innermost loops outwards. Stops unrolling if it encounters a CUDA thread index.

code [Rudy et al. 2011]. The key commands in a CUDA-CHiLL recipe are summarized in Table I, with example scripts in Figures 2(b) and 2(d).

CUDA-CHiLL is built upon an underlying polyhedral framework CHiLL [Chen 2007; Tiwari et al. 2009]. CHiLL supports composition of code transformations targeting complex loop nests. CHiLL operates most effectively on sequential code restricted to an affine domain, where loop bounds and subscript expressions are linear functions of the loop index variables. Compile-time unknown loop bounds are supported. Nonaffine references must be read-only. The CHiLL transformation algorithms will generate correct code when dependence conditions are satisfied. Such dependence information is kept updated during each step of the transformation process.

The output of the system is a set of code variants, with unbound parameter values that are executed in an autotuning phase to select the best-performing solution.

```

for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[j][i] = c[j][i] + a[k][i] * b[j][k];

```

(a) Matrix multiply source code (BLAS)

```

1 tile_by_index({"i","j"},{TI,TJ},{l1_control="ii",l2_control="jj"},
  {"ii","jj","ii","jj"})
2 tile_by_index({"k"},{TK},{l1_control="kk"},
  {"ii","jj","kk","ii","jj","k"},strided)
3 tile_by_index({"i"},{TJ},{l1_control="tt",l1_tile="tt"},
  {"ii","jj","kk","tt","tt","ii","k"})
4 cudaize("mm_GPU",{a=N*N,b=N*N,c=N*N},
  {block={"ii","jj"},thread={"tt","tt"}})
5 copy_to_shared("tx","b",-16)
6 copy_to_texture("b")
7 copy_to_registers("kk","c")
8 unroll_to_depth(2)

```

(b) CUDA-CHiLL script for SGEMM on GTX-280.

```

float P1[16];
__shared__ float P2[16][17];
bx = blockIdx.x, by = blockIdx.y;
tx = threadIdx.x, ty = threadIdx.y;
P1[0:15] = c[16*by:16*by+15][tx+64*bx+16*ty];
for (t6 = 0; t10 <= 1008; t6+=16) {
  P2[tx][4*ty:4*ty+3] = tex1Dfetch(texbRef,1024*
    ((16*by+4*ty:16*by+4*ty+3) + (tx+t6)));
  __syncthreads();
  P1[0:15] += a[t6][64*bx+16*ty+tx]*P2[0][0:15]
  P1[0:15] += a[t6+1][64*bx+16*ty+tx]*P2[1][0:15]
  ...
  P1[0:15] += a[t6+15][64*bx+16*ty+tx]*P2[15][0:15]
  __syncthreads();
}
c[16*by:16*by+15][tx+64*bx+16*ty] = P1[0:15];

```

(c) Generated CUDA kernel code from script in (b) (N=1024,TI=64,TJ=TK=16)

```

1 tile_by_index({"i","j"},{TI,TJ},{l1_control="ii",l2_control="jj"},
  {"ii","jj","ii","jj"})
2 tile_by_index({"k"},{TK},{l1_control="kk"},
  {"ii","jj","kk","ii","jj","k"},strided)
3 tile_by_index({"i"},{TJ},{l1_control="tt",l1_tile="tt"},
  {"ii","jj","kk","tt","tt","ii","k"})
4 tile_by_index({"j"},{TK},{l1_control="ss",l1_tile="ss"},
  {"ii","jj","kk","tt","tt","ss","s","k"})
5 cudaize("mm_GPU",{a=N*N,b=N*N,c=N*N},
  {block={"ii","jj"},thread={"tt","ss"}})
6 copy_to_shared("tx","b",-16)
7 copy_to_shared("tx","a",-16)
8 copy_to_texture("b")
9 copy_to_texture("a")
10 copy_to_registers("kk","c")
11 unroll_to_depth(2)

```

(d) CUDA-CHiLL script for DGEMM on Tesla C2050 Fermi

```

double P3[4][4];
__shared__ double P2[64][17];
__shared__ double P1[16][65];
bx = blockIdx.x, by = blockIdx.y;
tx = threadIdx.x, ty = threadIdx.y;
for (t=0; t<=48; t+=16)
  for (s=0; s<=48; s+=16)
    P3[t/16][s/16] = c[64*by+ty+t][64*bx+tx+s];

for (kk=0; kk<=3952; kk+=16) {
  for (tmp=0; tmp<=3; tmp++) {
    P1[tx][16*tmp+ty] = tex1Dfetch(texbRef,3968*
      ((16*tmp+64*by+ty) + (kk+ty)));
  }
  __syncthreads();
  for (tmp=0; tmp<=3; tmp++) {
    P2[16*tmp+tx][ty] = tex1Dfetch(texaRef,3968*
      ((16*tmp+64*bx+tx) + (kk+ty)));
  }
  __syncthreads();
  for (k=0; k<=15; k++) {
    P3[0][0] += P2[tx][k]*P1[k][ty]
    P3[1][0] += P2[tx][k]*P1[k][16+ty]
    ...
    P3[0][1] += P2[16+tx][k]*P1[k][ty]
    ...
    P3[3][3] += P2[48+tx][tk]*P1[k][48+ty]
  }
  __syncthreads();
}
for (t=0; t<=48; t+=16)
  for (s=0; s<=48; s+=16)
    c[64*by+ty+t][64*bx+tx+s] = P1[t/16][s/16];

```

(e) Generated CUDA kernel code from script in (d) (N=3968,TI=TJ=64,TK=16)

Fig. 2. Compiler-generated transformation recipe and automatically generated code for MM.

Table II. Nvidia GPU Target Architectures

	# SMs	Cores per SM	Total cores	Peak (SP)	Peak (DP)	Glob. Mem.	B/W	Reg. File per SM	Tex. Mem.	Const. Mem.
GTX-280	30	8	240	933 GF	97 GF	1 GB	142 GB/s	64KB	yes	64KB
C2050	14	32	448	1.03 TF	515 GF	3 GB	144 GB/s	128KB	yes	64KB

Additionally, this layered system exposes the optimization strategies of the compiler and facilitates savvy developers to intervene in optimization when the compiler-generated code falls short of performance goals. For example, a programmer can write their own CUDA-CHiLL scripts or insert lower-level CHiLL commands within the CUDA-CHiLL scripts if desired, or even add their own commands to CUDA-CHiLL.

2.2. GPU Architecture Features and CUDA

Table II describes key features of our two target architectures, the previous-generation Nvidia GeForce GTX-280, and the newer Nvidia Tesla C2050 Fermi. Both architectures are organized in a two-level parallelism hierarchy, with a number of streaming multiprocessors (SMs), each of which has a SIMD unit comprised of several cores. Reflecting this two-level hierarchy, a CUDA program describes a computation decomposition into a one- or two-dimensional space of thread blocks called a *grid*, where a block is mapped to one of the SMs. Each thread block defines a one- to three-dimensional space. A *kernel* thread program is executed for each point in the grid. Contiguous threads within a block run concurrently on the same SM in batches, called *warps*. Synchronization between threads in an SM is supported by a barrier; synchronization between threads mapped to different SMs can be costly and typically relies on atomic operations or a return to the host processor.

The GPUs have a heterogeneous, and mostly software-controlled, memory hierarchy. GTX-280 and C2050 both have a global memory that can be accessed by all cores on the device. Each SM has a large register file, larger in C2050, and a shared memory that can be freely accessed by all threads in a block mapped to that SM. Shared memory and registers have comparable latencies. Read-only constant and texture data in global memory are cached for low-latency average access time. A data cache for global memory accesses is available on the C2050. On the GTX-280, the bulk of global memory accesses are typically not cached, with latencies of hundreds of cycles.

The fundamental execution model, memory structures, and processor organization are similar for these two GPU devices, there are a few key differences that affect optimization. In addition to the presence of a data cache in the C2050, the balance of the architectures is different. SMs in a C2050 have four times more cores than in the GTX-280, which suggests that the number of threads per block should perhaps be larger. The GTX-280, in spite of its fewer overall number of cores, has a larger number of SMs, suggesting more blocks would be desirable. The configurable shared memory on the C2050 can be up to 3 times larger than the shared memory on the GTX-280, which is consistent with being able to support more threads per block. However, if the C2050 employs more threads per block, then the number of registers used per thread may need to be reduced. These architectural differences suggest that the optimization trade-offs may lead to very different implementations for the same code targeting the two platforms.

2.3. Optimizations for GPUs

We now describe how loop nest computations are mapped to the GPU architectures. We focus on computation partitioning and memory hierarchy optimizations, using a set of optimizations commonly found in GPU compilers [Leung et al. 2010; Baskaran et al. 2008a, 2010; Cui et al. 2011]. Memory hierarchy optimizations include both *improving data locality* through careful data placement in the memory hierarchy and *maximizing memory bandwidth*. Data placement considers registers, shared memory, texture and constant memory, or leaving the data in global memory. The bandwidth optimizations involve selecting computation partitions that maximize parallelism of memory accesses.

2.3.1. Parallel Computation Partitioning Using Tiling. A GPU is a *tiled* architecture, with each streaming multiprocessor (SM) representing a separate tile with its own private memory. Parallel code should be partitioned across SMs so that each thread operates on mostly independent, localized data. Subdividing the iteration space of a loop into blocks or *tiles* with a fixed maximum size has been widely used when constructing parallel computations [Irigoin and Triolet 1988; Wolfe 1989; Kennedy and McKinley

1992]. The shape and size of the tile can be chosen to take advantage of the target parallel hardware and memory architecture, maximizing reuse while maintaining a data footprint that meets memory capacity constraints and other optimization criteria. *Tiling* involves deconstructing an iteration space into a control loop and tile loop. Given an iteration space of size N and a tile size of TX the tile loop will iterate over a maximum space defined by the tile size (TX), while the control loop then has N/TX steps (when TX divides N evenly).

After tiling, the *cudaize* command specifies the name of the generated kernel function, the mapping of one or two loop levels to block indices in the grid dimension, and the mapping of up to three loops to thread indices. The effect of the *cudaize* call on the generated code is to allocate storage and marshall inputs for the CUDA kernel, and select the loops from the transformed loop nest that will be mapped to grid and thread dimensions. These tile controlling loops are effectively removed from the code. Example code generated by this process can be found in Figure 2.

2.3.2. Memory Hierarchy Optimization Using Data Copy. In addition to its role in parallel code generation, tiling also reorganizes computation within thread programs so that its data footprint fits within a specific level of the memory hierarchy. Data destined for shared memory must be explicitly copied to/from the device global memory. Similarly, data locally declared within a thread program will usually be placed in registers by the *nvcc* backend compiler, but such data may require explicit copies to/from global or shared memory. Therefore, we couple tiling with explicit copying of data and associated synchronization to perform the *data staging* into shared memory or registers.

When managing registers, we must also explicitly unroll the tiled loops so that accesses to local array data will have constant indices that can be mapped to specific registers. CHiLL supports loop unrolling through a combination of iteration space transformations and statement rewriting in the polyhedral representation.

To support CUDA code generation, we augment the data copy algorithm based on polyhedral scanning [Chen 2007] with a new capability that supports parallel code generation and GPU-specific memory structures. When copying into shared memory, the copy must be properly synchronized to avoid race conditions with other threads. Further, a copy into registers copies a data footprint that is private to individual threads. These *datacopy* transformations ensure correctness by understanding the data dependencies between loop statements, deriving conservative data footprints automatically.

2.3.3. Memory Bandwidth Optimizations. As will be discussed in the next two sections, the compiler primarily optimizes bandwidth by selecting computation partitions that: (1) enable the memory controller to *coalesce* multiple memory references across threads into a single transfer, by creating spatial reuse across threads; and (2) parallelize shared memory accesses so that each consecutive thread accesses a different bank in shared memory, accompanied by data padding as needed to avoid or reduce bank conflicts. In addition, the compiler: (3) uses texture memory accesses in parallel with other global memory accesses. This third bandwidth optimization is not included in other GPU compilers.

3. UNDERSTANDING THE OPTIMIZATION SEARCH SPACE

The compiler's structure facilitates exploring a rich search space that subsumes that of previous GPU compilers [Baskaran et al. 2008a, 2010; Leung et al. 2010; Cui et al. 2011]. To create a manageable search space, we employ compiler heuristics to focus on candidate implementations that are likely to yield high performance. This section articulates the optimization search space, and can also serve as a roadmap for the next section where the algorithms are presented in detail. We explain the search space in the context of a specific highly tuned library function for matrix multiply,

SGEMM, and DGEMM, for single or double precision, respectively. Because the compiler can explore multiple optimization strategies, insights from the experiences of manually tuning this code were integrated into our compiler implementation [Volkov and Demmel 2008; Nath et al. 2010].

3.1. Defining the Search Space

To manage complexity, we decompose the optimization decisions into three categories, each of which is evaluated in separate phases of the algorithm in the next section.

- (1) *Computation Partition*: Identify structure of blocks and threads, realized using permute and tile transformations (Section 4.1 and Figure 4).
 - (a) Candidate loops to select for threads and blocks (based on dependences, global memory coalescing, reuse).
 - (b) Candidate thread dimensions, created from a single loop or from multiple loops (determines data partitioning across threads).
 - (c) Data distributions per thread, cyclic vs. block distributions in all dimensions (affects global memory coalescing and memory system).
- (2) *Data Placement*: Identify where in memory hierarchy data should reside (Sections 4.2–4.3 and Figures 6–7), interrelated with computation partitioning.
 - (a) Read-write data structure with reuse within a thread: use registers up to capacity limits (may require additional tiling to match register footprint or use existing tile).
 - (b) Read-write data structure with reuse across threads: try shared memory (may require additional tiling to match footprint to parallel threads).
 - (c) Read-only data structure with reuse across threads: try constant memory or shared memory.
 - (d) Read-only data already mapped to shared memory: try texture memory.
- (3) *Optimization Parameters*: Select parameter values that govern number of threads and blocks, and data footprint (Section 4.4 and Figure 8).
 - (a) Number of threads for each dimension in a block.
 - (b) Granularity of computation within a thread.
 - (c) Number of elements per thread to place in shared memory or registers at a time (may require additional tiling).

3.2. Solution Comparison across Execution Contexts: Matrix Multiply

To illustrate important differences in the search space explored by our compiler, we use the examples in Figure 2, the code generated by our tool for matrix multiply for the two architectures, starting from the sequential code in Figure 2(a). We first consider the automatically generated transformation recipe and the generated code in Figures 2(b) and (d) for the GTX-280 and for C2050, respectively.

3.2.1. Computation Partition. In both examples (Figures 2(b) and (d)), loops i and j are selected for block loops, as they carry no dependences, and having two block dimensions matches the 2-dimensional array accesses in the computation. Matching the dimensionality of the largest-dimensioned data structure to the dimensionality of threads and blocks is a heuristic in our system which prunes many alternative strategies, and is consistent with other compilers and manually written code. The computation partition for the GTX-280 versions results from tiling the tile controlling loop for i further to create two dimensions of threads, while maintaining a per-thread, partial column (block) data partitioning for output c , and a column and row partition for inputs a and b , respectively. In contrast, for the C2050, at lines 3 and 4 of the script we tile both i and j loops to create threads (and also impact data footprint as described below). This results in a per-thread data layout for c that is two dimensional. To achieve global memory

coalescing on accesses to this two-dimensional object, we permute the tile controlling and tile loops for j (see line 4 Figure 2(d)). This gives us a two-dimensional cyclic partition for c . The 2D cyclic partition of c subtly impacts the order in which c is accessed from the global memory, which in turn may affect bank conflicts in global memory [Yang et al. 2010] and smooths out the performance for different problem sizes [Nath et al. 2010].

3.2.2. Data Placement. In both versions at line 2, the script tiles the k loop to be placed outside the potential thread loops, both for exploiting reuse of c in registers at thread level and b in shared memory inside the block. The subsequent copy at lines 7 and 10 of Figure 2(b) needs to identify the footprint of c to be placed in registers, and this corresponds to the data accessed within the thread program which will arise from loop i . Array b is targeted for shared memory in both versions because it carries reuse and requires shared memory to support coalesced access. Also, one or two of the inputs (depending on problem size) is assigned to the texture memory to further accelerate loading data into shared memory.

3.2.3. Optimization Parameters and Comparison with Manually Tuned Versions. The “cudaize” command in both scripts triggers CUDA code generation; it appears at this point in the script because prior statements set up the computation partitioning and loop structure, and subsequent statements refer to transformations that occur inside the kernel program (unrolling). The result of Figure 2(b) closely matches the SGEMM code in Volkov and Demmel [2008] except for a different parallel mapping scheme for the shared memory copy code, and the use of texture memory loads for b . The C2050 version varies only slightly. In addition to the differences in data distribution, the chosen recipe puts both inputs into shared memory, given the larger shared memory size available on the C2050, favoring the larger SM over L1 cache. The resulting code in Figure 2(e) is similar to that of the MAGMA library [Nath et al. 2010], but we do not perform a “prefetch” of input data into dummy variables to hide the latency of memory accesses, and for larger matrix sizes ($> 1k$), we only access b through texture memory. The SP (SGEMM) code, not shown in Figure 2, uses much larger tiles on the C2050 than the GTX-280 (TX=TY=96 on C2050 and TX=64, TY=16 on GTX-280); by doing so, we can control the data footprint as well as the computational decomposition suitable for the architecture. Note that similar scripts generate significant differences in code.

4. TRANSFORMATION STRATEGY GENERATOR

The Transformation Strategy Generator (TSG), an automated meta-optimization algorithm, is depicted in Figure 3. The TSG algorithm proceeds in four phases to derive the collection of transformation recipes associated with each code: (I) an analysis of computation partitioning strategies, which considers dependences, memory coalescing, and data reuse; (II) an analysis of data placement, which considers data reuse, sharing patterns, and memory coalescing; (III) recipe generation with symbolic optimization parameters by combining parallelization decisions and data placement decisions from I and II; and (IV) autotuning for optimization parameters and code variant selection. Throughout this section we refer to a *master recipe* that concisely represents the search space of possible code variants.

4.1. Phase I: Computation Partitioning Analysis

The algorithm in Figure 4 uncovers the possible parallelizations for a given computation, which may be further constrained once memory hierarchy optimizations are applied in phase II. The output of this first phase, as described in Figure 4, is a set of computation partitioning variants $v \in V$, each containing candidate loops for GPU blocks (v.BX, v.BY) and threads (v.TX, v.TY and v.TZ), and a loop order (v.Order) that moves these parallel loops to outermost levels such that block loops precede thread

Fig. 3. Transformation Strategy Generator (TSG) to automatically generate transformation recipes and autotune optimization variants and parameters.

loops and according to the x,y or x,y,z dimension sequencing. Supporting algorithms in Figure 5 collect the tile and permute transformations required to produce variants that represent different block and thread decompositions. Using constant memory may impact computation partitioning and loop order by requiring an additional level of tiling in the host code, so we also consider variants that place data into constant memory during this phase.

4.2. Phase II: Data Placement Analysis

The second phase examines each data structure in the computation and identifies candidates for mapping to different levels of the memory hierarchy (other than constant memory). The algorithm shown in Figure 6 takes as input the master recipe from phase I with parallelization decisions. Data placement decisions are made for each possible parallelization strategy. These candidates are determined based on the heuristics from Section 3.1, and the capacity limitations from Table II. Shared memory supports data locality across threads, while registers support data locality within threads; tiling may be required to fit within limited capacity of each. Texture memory with its dedicated access channel to global memory is suited for accessing read-only data structures, increasing the available bandwidth for accessing global memory. From the previous subsection, constant memory is most beneficial for read-only data structures less than 64K in size or else additional tiling in the host is required. The algorithm marks which data structures can be destined for which level of the memory hierarchy and marks loops for tiling, but defers any associated tiling or permutation until the code generation phase once all decisions can be composed into a transformation recipe.

4.3. Phase III: Generating Autotuning Variants

The algorithm in Figure 7 generates CUDA-CHiLL recipes from all unique combinations of decisions prescribed in the master recipe. The algorithm emits the commands in Table I.

If all possible variants were generated, and evaluated with all possible parameter values, the search space would still be prohibitively large, as will be discussed in the next section. Therefore, at this step, once the set of variants has been generated, we perform a preliminary pruning to rule out some of the data placement decisions. This pruning can be thought of as treating the data placement decisions and optimization parameter values as independent tuning phases. Using default optimization parameter

Algorithm ComputationPartitioningCandidates

Inputs: a loop nest computation L and a single computation variant v initialized for L

$L = \langle \text{Loops}, \text{Statements}, \text{Refs}, \text{Dependences} \rangle$

Here, $\text{Loops} = \{l_1, l_2, \dots, l_n\}$ represents a (possibly imperfect) loop nest in its nesting order

```

 $v = \{$ 
     $T = \emptyset$                                 // set of tile commands
     $CM = \emptyset$                             // constant memory candidates
     $BX = BY = \emptyset$                       // assignment of block loops
     $TX = TY = TZ = \emptyset$                 // assignment of thread loops
     $\text{Order} = \{l_1, l_2, \dots, l_n\}$         // assignment of loop order
 $\}$ 

```

Output: (Partial) Master Recipe V , a set of computation variants (v)

- (1) *Preprocessing.* Master recipe $V = v$. Permute $v.\text{Order}$ such that any loop carrying a dependence is moved inward so that the dependence is carried within a thread program, or outward so that it is carried by the sequential host program. If needed, loop skewing can be performed so that the dependences are carried by the outermost loop and synchronized on the host. (As an alternative to synchronizing on the host, Baskaran et al. use a single-writer, multiple-reader block synchronization on the GPU [Baskaran et al. 2010], which has a limitation of only being able to execute a single block at a time to avoid deadlock). Variant v is added to the master recipe V . If multiple options exist, each can result in a new element v_i being added to V with different orders in $v_i.\text{Order}$.
- (2) *Constant memory candidates.* Read-only variables in $L.\text{Refs}$ with suitable reuse patterns are candidates for constant memory. Tiling may be required if the data footprint of the variable in the loop nest is larger than the capacity limit of constant memory. For each candidate we create $\tilde{v}$, a copy of variant v (already in V). We set $\tilde{v}.CM$ as the candidate, $\tilde{v}.T$ as the new tiling command and $\tilde{v}.\text{Order}$ as the modified order after tiling. We add $\tilde{v}$ to the master recipe V .

The next three steps are similar, each selecting loops for tiling and candidates for GPU blocks or threads. They call the functions **VariantsBlock/ VariantsThreadX/ VariantsThreadYZ**, respectively, to create and add variants to the Master Recipe.

- (3) *Block loop candidates.* This step selects the candidates for block loops, which must meet the following criteria: (1) it does not carry a dependence; and, (2) it is safe to permute the candidate to the outermost position (or next-to-outermost position if the outer loop carries the dependences). We use 2D blocks for computations with multi-dimensional data structures, and only a single dimension otherwise. This heuristic improves the surface to volume ratio of partitioning. This step calls the function **VariantsBlock** with the current Master Recipe V , the set of sets of loops to be tiled and the associated candidates for block loops as arguments.
- (4) *Candidate thread loops in the x dimension and memory coalescing.* A candidate loop for the x thread dimension must meet the following criteria: (1) it does not carry a dependence; (2) permutation of the loop to inside the block loops is safe; and, (3) The last dimension of at least one array is accessed by the corresponding loop index. The first is needed for correctness and the third criterion is designed so that the x dimension linearly accesses the bulk of its data in global memory, resulting in coalesced accesses. When there are no loops that meet these criteria, the last one is relaxed. This step then calls **VariantsThreadX** with loops to be tiled and associated candidates for threads.
- (5) *Additional thread dimensions.* A remaining loop mapped to the y or z thread dimensions must meet just two criteria: (1) it does not carry a dependence; and, (2) it is safe to permute the loop to just within the x thread dimension. As with blocks, we match thread dimensionality to data dimensionality. We may obtain additional thread dimensions by further tiling $v.TX$ (as the example in Figure 2(b)), or by selecting a different loop for $v.TY$ as the example in Figure 2(d)). This step calls **VariantsThreadXY** with loops to be tiled and associated candidates for threads.

Fig. 4. Phase I algorithm: computation partitioning analysis.

Algorithm VariantsBlock**Algorithm VariantsThreadX****Algorithm VariantsThreadYZ**

// The three algorithms are largely identical, except where preceded by comments

Inputs: *the current Master Recipe V and block or thread candidates along with requisite tiling*

V // (Partial) Master Recipe, which is a set of variants v

C // Set of tile commands and associated (block or thread) candidate
// loops for each variant

// Here, in C for the three functions, respectively, c represents

// tile commands and associated (block or thread) candidate loop(s):

// $C = \{ c : \langle t, BX, BY \rangle \}$

// $C = \{ c : \langle t, TX \rangle \}$

// $C = \{ c : \langle t, TY, TZ \rangle \}$

Output: Modified Partial Master Recipe V

for each $v \in V$ {

$S = \emptyset$ // S is a temporary set of variants

for each $c \in C$ {

$\tilde{v} = COPY\ v$

$\tilde{v}.T = \tilde{v}.T \cup c.t$

$\tilde{v}.Order =$ order associated with tiling $c.t$

// The candidates below may include loops created by tiling

// The next two statements are executed only for VariantsBlock

$\tilde{v}.BX =$ candidate BX value

$\tilde{v}.BY =$ candidate BY value

// The following statement is executed only for VariantsThreadX

$\tilde{v}.TX =$ candidate TX value

// The next two statements are executed only for VariantsThreadYZ

$\tilde{v}.TY =$ candidate TY value

$\tilde{v}.TZ =$ candidate TZ value

$S = S \cup \tilde{v}$

}

$V = V - v$

$V = V \cup S$

}

Fig. 5. Phase I: Supporting algorithms to tile to create block and thread decomposition variants.

values, we evaluate the set of variants. For each possible parallelization strategy, we use empirical measurement of the associated variants to determine the best-performing data placement strategy and rule out other data placement strategies.

We validated this pruning strategy of holding the optimization parameters fixed for data placement decisions by doing an exhaustive search of parameters and data placement for five of the benchmarks in the next section: MM, MT, TMV, CP, MPEG, and NBODY, and achieved the same final result as compared to exhaustive search in all cases. For remaining benchmarks, this comparison is not feasible because the search space size or compilation time would be prohibitive (MRIQ and MRIFH). We found that, while the performance varies with different parameter values, the relative differences between implementations leads us to the right data placement decisions.

4.4. Phase IV: Autotuning Optimization Parameter Values

Figure 8 describes the final autotuning step, which binds the optimization parameter values in the recipes, specialized to problem size. These unbound optimization parameters represent thread and block sizes and tile sizes associated with data

Algorithm DataPlacementCandidates**Inputs:**

$L = \langle \text{Loops}, \text{Statements}, \text{Refs}, \text{Dependences} \rangle$ // a loop nest computation

V = master recipe from phase I with variants v representing different parallelization candidates

Output: V , master recipe with variants v representing parallelization and data placement candidates

The next three steps process each of the variants v in the master recipe V to add data placement decisions.

- (1) *Shared memory candidates:* Candidates for shared memory $v.SM$ are not in $v.CM$. They have reuse across one of the thread loops $v.TX$, $v.TY$ or $v.TZ$, or their accesses are not coalesced in $v.TX$ and coalescing is possible for other loops $l \in \text{Loops} - \{v.BX, v.BY, v.TX\}$. For each candidate data structure d , the algorithm creates a shared memory candidate object $sc = \langle d, l, t, ct, o \rangle$ that describes the data structure, a loop level l for which the copy should be placed, and, optionally, a specification of how to tile internal loops within each thread. This specification includes a set of internal loops t to be tiled, corresponding tile controlling loops ct , and a modified loop order o after tiling. These latter three optional fields are only necessary if d 's footprint within l is an array that is larger than the number of threads, as determined by polyhedral scanning. For example, the loop order o is defined as $\{v.BX, v.BY, ct, v.TX, v.TY, v.TZ, t\}$ if t represents a single loop. This order captures the right footprint for the copy across threads. (The footprint of the copied array will be bound when tile sizes are fixed to constant values by the Phase IV algorithm.) We add each sc to $v.SM$.
- (2) *Register candidates:* Candidates for register $v.R$ are not in $v.CM$ or $v.SM$, and have reuse carried within the innermost loop or across some other loop inside the thread program. For each candidate data structure d , the algorithm creates a register candidate object $rc = \langle d, l, t, ct, o \rangle$ that describes the data structure, a loop l to place the copy, and an optional tiling specification as for shared memory tiling. For registers, the loop order o when tiling for a single loop t within the thread would be $\{v.BX, v.BY, v.TX, v.TY, v.TZ, ct, t\}$. We add each rc to $v.R$.
- (3) *Texture memory candidates:* Candidates for texture memory are read-only data that are also in $v.SM$. We have found through experimentation that texture memory has only demonstrated performance gains in such cases (and not for data in registers or global memory).

Fig. 6. Phase II algorithm: data placement analysis.

placement decisions. The parameter values are pruned based on the following heuristics.

Number of threads per block in x dimension. The scheduling unit on a GPU is a *warp* of 32 threads. Of particular importance are the number of threads in the x dimension for global memory coalescing. Thus the algorithm to govern the number of threads per block should be the warp size and its multiples. Also, the number of threads in a block has an upper bound defined by the architecture.

Number of blocks. The algorithm takes into account a dataset size (or a range) to determine whether a particular tile size, representing blocks, leads to a computation partition where a significant fraction of the resources are idle. This algorithm was derived from empirical analysis using a much larger set of tile size candidates, and prunes away sizes that lead to idle resources.

Tile sizes used for data staging. When tiling is used to stage data into shared memory or registers, the maximum tile size is limited by shared memory or register capacity constraints, respectively. The footprint of data accessed by a thread block cannot exceed the size of shared memory, which is quite small as shown in Table II.

Algorithm GenerateVariants**Input:** V = master recipe from phase II**Output:** S = set of parameterized CUDA-CHILL scriptsInitially, $S = \emptyset$ Process each variant $v \in V$.Initialize a script s to represent v . Add s to S . Emit commands to s as follows:

- (1) *Constant memory*: If this variant uses constant memory, the algorithm emits the following for all data structures d in $v.CM$: *emit*($s, copy_to_constant(d)$)
- (2) *Reorder and tile for computation partitioning*: Emit the command to tile and order block and thread loops. If this variant requires tiling in the host for constant memory, the algorithm also emits a tile command to accomplish this. Prior to generating the command, the algorithm creates parameters representing tile sizes $T_1, \dots, T_n$ and identifiers representing tile controlling loops $t_1^c, \dots, t_n^c$. This step generates the ordering for the block and thread loops, and applies the necessary tiling to adjust parallelism as needed: *emit*($s, tile_by_index(v.T, \{T_1, \dots, T_n\}, \{l_{control} = l_{t_i^c}, \dots\}, v.Order)$)
- (3) *Copy to registers, tiling if needed*: The algorithm creates a copy $\tilde{v}$ of variant v for each subset $\tilde{v}.R \subset v.R$, and then a corresponding copy $\tilde{s}$ of s to represent $\tilde{v}$ and adds it to S . This allows TSG to evaluate which subset yields the best utilization of the register file, as follows:
 - (a) Scan the set $\tilde{v}.R$ to identify whether additional tiling is needed from the individual $rc = \langle d, l, t, ct, o \rangle$ objects in $\tilde{v}.R$.
 - (b) For example, assume only a single loop t must be tiled, the tile size T is created, and the remainder is specified in the register candidate objects.
 - (c) Generate tile commands as needed: *emit*($\tilde{s}, tile_by_index(t, \{T\}, \{ct\}, o, strided)$)
 - (d) for each $rc \in \tilde{v}.R$, *emit*($\tilde{s}, copy_to_registers(rc.d, rc.l)$)
- (4) *Copy to shared memory, tiling if needed*: The algorithm creates a copy $\tilde{v}$ of variant v for each subset $\tilde{v}.SM \subset v.SM$, and then a corresponding copy $\tilde{s}$ of s to represent $\tilde{v}$ and adds it to S . This allows TSG to evaluate which subset yields the best utilization of the shared memory, as follows:
 - (a) Scan the set $\tilde{v}.SM$ to identify whether additional tiling is needed from the individual $sc = \langle d, l, t, ct, o \rangle$ objects in $\tilde{v}.SM$.
 - (b) For example, assume only a single loop t must be tiled, the tile size T is created, and the remainder is specified in the shared memory candidate objects.
 - (c) Generate tile commands as needed: *emit*($\tilde{s}, tile_by_index(t, \{T\}, \{ct\}, o, strided)$)
 - (d) for each $sc \in \tilde{v}.SM$, *emit*($\tilde{s}, copy_to_shared(sc.d, sc.l)$)
- (5) *Fetch using texture memory*: The algorithm creates a copy $\tilde{v}$ of variant v for each subset $\tilde{v}.TM \subset v.TM$, and then a corresponding copy $\tilde{s}$ of s to represent $\tilde{v}$ and adds it to S . This allows TSG to evaluate which subset yields the best utilization of texture memory, as follows:
 - (a) for each $tc \in \tilde{v}.TM$, *emit*($\tilde{s}, copy_to_texture(tc.d)$)
- (6) *CUDA blocks, threads and kernel*: Generate the CUDA command for variant v with corresponding script s :
emit($s, cudaize(\langle kernel \rangle, \{ \langle params \rangle \}, \{ block = \{v.BX, v.BY\}, thread = \{v.TX, v.TY, v.TZ\} \})$)
- (7) *Loop Unrolling and Unroll-and-Jam*:
 As a final step, the algorithm creates up to three copies of s corresponding to variant v for different levels of unrolling:
emit($s_1, unroll_to_level(0)$), *emit*($s_2, unroll_to_level(1)$), *emit*($s_3, unroll_to_level(2)$)

Fig. 7. Phase III algorithm: code variant generation.

5. PERFORMANCE ANALYSIS

This section presents a detailed performance analysis resulting from applying the previously described compilation system to the a set of benchmarks in Table III. These results show the effectiveness of our system, along with the size of the optimization search space and the contributions of individual optimizations. As our goal is to closely approach the performance of manually tuned code, the first set of benchmarks in

Algorithm SelectBlockTileSizes**Input:** p = problem size**Output:** BTS //set of tile sizes to consider during autotuning

```

 $X = MAX\_THREADS$ 
 $BTS = \emptyset$ 
 $M$  = number of Streaming Multiprocessors
// Prune block sizes that would leave more than half of SMs idle in last compute cycle
for ( $t = scheduling\_unit$ ;  $t \leq X$ ;  $t = t + scheduling\_unit$ )
   $B = \lceil p / t \rceil$  // number of blocks
   $R = (B \% M)$  // SMs participating in last compute cycle
  if ( $R == 0 \parallel (M-1)/2 < R < M$ )
     $BTS = BTS \cup t$ 

```

Algorithm AutotuningParameterizedVariants**Input:**

```

 $PROB = \{\text{set of problem sizes}\}$ 
 $PARAM = \{\text{set of parameters}\}$ 
 $s$  = selected CUDA-CHILL script from phase III

```

Output: $Code$ = optimized code

```

 $time\_val = MAX\_TIME$ 
 $Tl = \{Tl_{param_i} \mid \text{where } Tl_{param_i} \text{ is the set of tile sizes for parameter } i\}$ 
for each ( $prob \in PROB$ )
   $Tl = \emptyset$ 
  for each ( $param \in PARAM$ )
     $Tl_{param_i} = \text{SelectBlockTileSizes}(prob)$ 
     $Tl = Tl \cup Tl_{param_i}$ 
  // Generate set of combination of selected tile sizes
   $Tl_{comb} = \{Tl_{param_1} \times Tl_{param_2} \times \dots \times Tl_{param_n}\}$ 
  for each ( $t \in Tl_{comb}$ )
    // Bind a unique parameters combination, generate code and time variant
     $tmpcode \leftarrow \text{CodeGen}(\text{BindParms}(s, t))$ 
     $tmp = \text{evaluate}(tmpcode)$ 
    if  $tmp < timeval$ 
       $Code = tmpcode$ 
       $timeval = tmp$ 

```

Fig. 8. Phase IV algorithm: prune and search optimization parameter values.

Table III are manually tuned dense linear algebra functions from the CUBLAS library. Another important demonstration is that the compiler organization achieves performance results beyond state-of-the-art compiler technology using a fixed optimization strategy. For this purpose, the second set of benchmarks in Table III are from Baskaran et al. [2010] and allow us to directly compare with a GPU compiler based on PLUTO, which previously had performed the most extensive study of automatic parallelization for GPUs on general-purpose codes to date. As is common in the GPU community, our measurements show GFlops rates, which are an absolute measure of raw performance.

To show the performance portability of our optimization strategy, we generate highly-tuned code for the very different GTX-280 and C2050, described in Table II. For all measurements, each version of the generated code and the CUBLAS library was run 10 times, and we used the mean of those run times, recorded using CUDA events². Performance results were reasonably stable, with a standard deviation of less than 0.1msec for execution times in the tens of milliseconds.

²Benchmarks from [Baskaran et al. 2010] use timers packaged with the benchmarks.

Table III. Benchmarks Set Evaluated by CUDA-CHILL

Benchmarks	Domain	Benchmark Source/ Performance Compared to
Matrix-Matrix Multiplication (MM)	Linear Algebra	CUBLAS
Matrix-Vector Multiplication (MV)		CUBLAS
Transpose Matrix-Vector Multiplication (TMV)		CUBLAS
N-Body (Nbody)	Scientific Computation, Multimedia and Imaging	PLUTO
Coulombic Potential (CP)		PLUTO
MPEG4		PLUTO
MRI-FH		PLUTO
MRIQ		PLUTO/PARBOIL

Table IV. Search Space Pruning Results for Computation Partitioning Phase

	Computational Partitioning Phase						Autotuning Phase				
	In	Pruned by		Out	Decomp	Tot	Pot	Evaluation		Versions	Total
		Data	Mem		map	Var	Var			Eval	% Pruned
Bench		Dep	Coal					Opt	Param		
MM	6	4	1	1	4	5	7936	124	64	188	97.63
MV	2	1	0	1	1	2	560	35	16	51	90.80
TMV	2	1	0	1	1	2	560	35	16	51	90.80
MRIQ	2	1	0	1	1	2	176	11	16	27	84.65
MRI-FH	2	1	0	1	1	2	5104	319	16	335	93.43
MPEG4	24	22	1	1	4	5	9984	156	64	220	97.79
CP	6	4	1	1	4	5	2816	44	64	108	96.16
Nbody	2	1	0	1	1	2	304	19	16	35	88.48

5.1. Size of Optimization Search Space

Table IV shows the effectiveness of compiler heuristics on pruning a potentially huge space of solutions. The first column shows the number of possible input loop orders that must be considered during computation partitioning. The next two columns show that dependences and memory coalescing heuristics help prune these input loop orders according to Phase I of the algorithm, shown in Figure 4; the result is the remaining loop orders to be considered, shown in the fourth column. These variations along with loop mapping options for block and thread decompositions (shown in the column labeled *Decomp.map*), limit the search space to no more than five different computation partitioning variants, shown in the last column. Similarly, the data placement decisions are made for all the data structures of the computations, explained in Phase II of the algorithm, shown in Figure 6.

If we consider all combinations of computation partitioning and data placement decisions, we may still end up with a prohibitively large search space to explore, as shown by the numbers in the very first column for the Autotuning Phase in Table IV. By holding the optimization parameters fixed during Phase III of the algorithm (Figure 7), the variants represent the set to be autotuned for computation partitioning and data placement decisions. Once the best computation partitioning and data placement strategy is discovered, we tune the optimization parameter values. This phase constrains the possible values for optimization parameters through hardware limitations such as maximum number of threads or blocks, warpsize, and size of shared memory or register file. For the set of benchmarks evaluated in this article, the maximum number of unique parameter values considered is 64. Overall, the system evaluates a maximum of 335 transformation recipes, across all the benchmarks. Almost 97% of the data placement and optimization parameter search space never gets evaluated.

Fig. 9. Performance comparison of automatically generated CUDA-CHiLL code with CUBLAS 3.2 on GTX-280 and C2050.

5.2. Performance Comparison with Manually Tuned BLAS

We compare against the CUBLAS 3.2 library, and for the C2050 we also compare against MAGMA 0.2 for DGEMM over a range of square matrix sizes from 1K to 8K elements. Figure 9(a) shows SGEMM, which achieves the highest level of performance (up to 592 GFlops) for the compiler-generated code because it has the most computation. The compiler-generated code performs almost at the same level as the manually tuned CUBLAS 3.2, coming within 2% of CUBLAS 3.2 on the GTX-280 and within 8% on the C2050. MAGMA performance lies somewhere in between the two, but all are close. However, results from an experiment on Tesla C2050, presented in Figure 9(g), show

that the compiler-generated code consistently performs better than CUBLAS 3.2 for smaller matrices (up to 1K).

In addition to matrix multiply, we also include SP and DP results for matrix-vector multiplication (MV), and transposed matrix-vector multiplication (TMV), and compare against CUBLAS 3.2. For MV in Figures 9(c) and (d), the compiler-generated code outperforms CUBLAS 3.2 on both architectures on almost all the points, by as much as 1.84x on the C2050 and 1.22x on GTX-280. It is interesting that the raw performance for larger dataset sizes is actually better on the less powerful GTX-280. We suspect this reflects a less efficient use of the available memory bandwidth on the C2050. The compiler uses a similar computation partitioning and data placement strategy for the two architectures, but very different numbers of threads per block depending on problem size, 64–128 for the GTX-280 (64–128 range) and larger for C2050 (128–384).

Results for TMV are shown in Figure 9(e) and (f). As compared to MV, even though there is no data reuse, the compiler loads the transposed input matrix into shared memory in coalesced order, and then different threads access the data. We use constant memory for the input vector, and it improves performance for all but one problem size (8K) on the GTX-280 and for a single problem size (6K) on the C2050, in our SP experiments. DP results also follow the same trend, but time out on the GTX-280 for larger problem sizes due to its weaker double-precision support. On the GTX-280, which has no L1 cache and a smaller shared memory, SP versions always perform within 5% of CUBLAS, with a maximum speedup of 1.53x. Our DP code achieves 1.90x speedup over CUBLAS. For the C2050, the code outperforms CUBLAS significantly, up to 1.90x for both SP and DP code.

Within the limits of dataset sizes that fit on a single GPU and for sufficiently large dataset sizes to occupy the machine, this set of experiments shows that performance is robust across changes in dataset sizes and architectures; as dataset size increases the code is able to maintain its performance levels. As we move to the C2050, which has more cores, performance improves.

5.3. Performance Comparison with State-of-the-Art GPU Compiler

We applied our system to a set of optimized benchmarks generated by a state-of-the-art compiler presented in Baskaran et al. [2010]. To derive timings, compute GFlops, and compare with the benchmarks, we used the test harness provided with the benchmarks. Figure 10 shows the speedups we achieve over the Baskaran et al. [2010] benchmarks (y-axis), on the C2050 and GTX-280 GPUs; each bar represents a benchmark. (In the remainder of this section, we refer to this compiler as PLUTO, although these benchmarks were specifically produced from an extension of PLUTO for GPUs.) Performance in GFlops for each kernel is shown at the top of each bar. Overall, our system achieves an average speedup of 1.48X and 1.33X for the C2050 and GTX-280, respectively.

The CP benchmark shows the highest performance (332 GFlops) and speedup (2.03X) over PLUTO. Our system selects 32x16 blocks and 16x16 threads, and full unrolls the innermost computation loop of 256 iterations, while PLUTO uses 32x32 blocks, 16x16 threads, and less work per thread, along with an unroll factor of 16 (using an unroll pragma). A similar strategy achieves performance of over 265 GFlops on the GTX-280.

For the two MRI kernels, MRIQ and MRI-FH, we were able to find better performing solutions by using half as many threads per block as the PLUTO-generated code. In addition, in MRIQ, data placement in shared memory for the key data structure *kVals* was also different from the PLUTO-generated code. The code for MRIQ is also highlighted in Chapter 8 of Kirk and Hwu [2010]. In Kirk and Hwu [2010] and IMPACT [2007], a different strategy is used for *kVals* that tiles in the host code and places tiles of *kVals* in constant memory, as is done in the PLUTO benchmark. The PLUTO and hand-coded PARBOIL benchmarks MRIQ and MRI-FH perform comparably to each

Fig. 10. Performance comparison with state-of-the-art GPU compiler in Baskaran et al. [2010].

other; and our compiler achieves a speedup of 1.13X on the C2050 and matches their performance on the GTX-280.

In the NBody kernel, the major differences from the PLUTO strategy are fewer threads per block and unrolling of the innermost computation loop with a factor of 128 compared to PLUTO's factor of 16 unrolling. Our system achieves a 1.81X and 1.28X improvement on the C2050 and GTX-280, respectively. For MPEG4, a multimedia kernel, the computation partitioning and data placement strategy is similar across our compiler and PLUTO, but we use 256x256 blocks as compared to 8x4 blocks in PLUTO. The number of threads per block is the same, but our solution performs far less work per thread, and allows the system to co-schedule multiple blocks on the same multiprocessor. We achieve a 1.26x speedup on C2050 and 1.77x on GTX-280 as compared to PLUTO.

5.4. Impact of Memory Hierarchy Optimizations

The code targeting the two architectures shares some similarities, and uses all the various levels of the memory hierarchy. Nevertheless, the impact of the optimizations varies across architectures, applications, and dataset sizes. Specific optimizations have more impact depending on the architecture and the nature of the problem. To illustrate this point, Table V compares with the best solutions in this study for a particular problem size to isolate effects of specific optimizations. In this table, we capture the performance difference from the best-performing baseline, and one with one optimization turned off. It's worthwhile to note that combining all the optimizations may not yield the best performance in some kernels. That is why some optimization columns are displaying zero effect on the performance.

Overall, we observe that the most significant loss of performance results from either disabling unroll-and-jam or copying to registers. These optimizations work together to manage the register file and improve instruction-level parallelism. For several of the programs, small 2D arrays are accessed from registers and unrolling simplifies access expressions so that the arrays are mapped to registers by the backend compiler.

Copying data to shared memory generally improves performance on both architectures, more significantly on the GTX-280, possibly due to the absence of a data cache as compared to the C2050. TMV and MM use shared memory to copy an input matrix

Table V. Performance Percentage Loss, Resulting from Disabling Individual Optimizations

Name	Problem	Unroll		Registers		Shared Mem		Texture Mem		Constant Mem	
	Size (Sq. Mat)	GTX- 280	Tesla C2050	GTX- 280	Tesla C2050	GTX- 280	Tesla C2050	GTX- 280	Tesla C2050	GTX- 280	Tesla C2050
SGEMM	4320	92.30	97.12	92.34	84.77	72.17	97.68	1.38	4.20	0.00	0.00
DGEMM	4544	Timeout	95.39	Timeout	94.55	37.60	53.20	0.52	3.28	0.00	0.00
SGEMV	4096	67.78	63.46	67.31	70.42	54.10	0.00	0.00	0.00	0.00	1.31
DGEMV	2048	48.93	56.75	57.89	61.91	0.00	0.00	0.00	0.00	2.69	3.34
SGETMV	4096	63.06	0.51	29.71	20.75	92.59	77.32	0.00	0.00	3.84	0.00
DGETMV	2048	0.00	0.00	9.67	19.54	85.26	83.33	0.00	0.00	0.00	0.00
MRIQ (SP)	32k,3k	0.00	0.00	12.47	51.8	10.61	16.70	0.00	0.00	0.00	0.00
MRI-FH (SP)	32k,3k	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	59.09	59.15
MPEG4 (SP)	4k,16	45.61	62.04	98.87	87.75	97.69	88.20	0.00	0.00	0.00	0.00
CP (SP)	32k,3k	0.00	5.36	93.81	30.99	10.58	10.26	0.00	0.00	0.00	0.00
NBODY (SP)	4k,16	49.36	10.74	62.76	53.59	0.00	0.00	0.00	0.00	0.00	0.00

into shared memory in coalesced order that is stored in global memory in a different order, which has a huge impact on performance.

Texture memory proves useful only for the various versions of MM. The gain comes from increased bandwidth to global memory by using the dedicated texture fetch hardware. Texture memory benefits a few other programs, but these same programs improve further by using constant memory. Constant memory on the other hand improves performance in kernels where we are able to put one or more of the appropriately sized inputs in the constant memory (MV, TMV and MRIFH) that is accessed in the same order across all the threads. Using constant memory eliminates the overhead of shared memory copy and potentially frees up shared memory for shared data in some kernels.

An interesting observation in the table is the absence of all major optimizations in the solution of MRI-FH. Given the number of data structures and their respective sizes, the best performance is achieved by using constant memory for smaller-sized data structures and accessing the larger data structures through global memory in coalesced order.

6. RELATED WORK

Several programming systems for GPUs have been introduced to achieve performance gains while increasing productivity of average programmers. Lee et al. [2009] generate CUDA code from sequential C code with extended OpenMP annotations. *hiCUDA* [Han and Abdelrahman 2009] is a pragma-based language that translates sequential C code to CUDA code. Copperhead is a high-level data-parallel language embedded in Python that generates CUDA [Catanzaro et al. 2011]. These approaches all suffer from limitations in what can be expressed, and therefore impede the compiler from having the flexibility to derive the best implementation. For example, *hiCUDA* code must be explicitly tiled by the programmer, and the pragma must describe the exact footprint for the data to be copied into shared memory.

A few compilers take CUDA programs as input, and produce optimized CUDA as output. CUDA-lite [Ueng et al. 2008] relies on annotations to instruct the compiler how to improve the coalescing of global memory accesses and bandwidth to global memory. Yang et al. [2010] can optimize a naive CUDA kernel with thread remapping and data copy optimizations. Thread remapping merges neighboring threads or thread blocks, which can be viewed as a bottom-up version of tiling. Their compiler achieves impressive performance though it still requires the input CUDA kernel to be properly parallelized. In Unkule et al. [2012], additional optimizations on CUDA code to

Table VI. A Comparison of GPU Automatic Parallelization Systems

Compilers	R-Stream	PLUTO	Cui et al.	CUDA-CHiLL
PARALLELIZATION STRATEGY				
Blocks	fixed strategy	multiple variants	appears to be fixed	multiple variants
Threads	fixed strategy	multiple variants	appears to be fixed	multiple variants
MEMORY HIERARCHY LEVELS SUPPORTED				
Registers	post-processing	post-processing	✓	✓
Shared Memory	✓	✓	✓	✓
Texture Memory	×	×	×	✓
Constant Memory	×	✓	×	✓
AUTOTUNING SUPPORT				
Optimization Parameters	×	✓	✓	✓
Computation Partitioning Variants	×	✓	✓	✓
Data Layout/Placement Variants	×	limited	✓	✓
Generates Transformation Recipe	×	×	✓	✓
Pruning Strategy/Measurements	×	×	×	✓

manipulate thread granularity and improve register utilization are described, but the effectiveness of these optimizations may also be impacted by the initial parallelization specification.

Compilers based on a polyhedral framework have been proposed to transform sequential codes into optimized CUDA codes, including our work. Table VI summarizes features of these automatic parallelization systems targeting GPUs. Our compiler is the only one that can utilize the full memory hierarchy including texture and constant memory. With multiple code variant generation and direct support for parameter tuning, we have achieved very high performance, close or in some cases faster than manual tuning. We are also the first to report results on double-precision floating-point BLAS kernels. In the following, we summarize each of the compilers in Table VI and describe the code generation algorithm it uses.

R-Stream [Leung et al. 2010] applies a fixed sequence of optimizations that are determined through a machine model. The approach of R-Stream represents a more traditional compiler where optimization is completely hidden by the compiler implementation, and autotuning is replaced with compiler models. While compilation is faster and the compiler is easier to use, the complexity of modeling modern architectures can lead to lower-performing generated code. The R-Stream compiler does not perform autotuning, nor does it utilize constant or texture memory. Some optimizations such as unrolling and privatization have to be applied after the code is lowered to the intermediate representation, whereas our system integrates these optimizations within the polyhedral framework. These limitations may partially explain why R-Stream's reported performance results fall far short of those presented in this article.

Various aspects of PLUTO to support C to CUDA transformation, are described in Baskaran et al. [2008a, 2008b, 2010]: individual algorithms such as finding parallel loops for two levels of parallelism with regard to global memory coalescing, finding data footprint size for shared memory copy and finding padding factors to avoid shared memory conflicts. In Baskaran et al. [2008b], the autotuning portion and some of the code generation steps are performed outside the polyhedral compiler, which could limit the robustness and effectiveness of the solution. Autotuning is used in a limited way to generate multiple computation partitions arising from global memory coalescing, vary data placement in shared memory, and test a small number of parameter values, but in the later PLUTO paper [Baskaran et al. 2010] parameter values and optimization

strategies are fixed. Further, it is not shown how close their performance is compared to state-of-the-art manual optimization.

The closest work to ours is a script-controlled compilation framework by Cui et al. [2011], designed specifically for generating BLAS kernels. The system takes the script, a predefined optimization scheme, and a data layout plan for matrices, as input to represent a complete optimization strategy. Complementary additional features of the compiler include transposed copy to global memory and padding of triangular matrices. The results reported in their paper are comparable to CUBLAS 3.2 (single precision only). As compared to our results, they use the identical optimization strategy for both the GTX-285 and C2050 for single-precision matrix-matrix multiply, while our experience and the experience of the MAGMA library developers [Nath et al. 2010] is that a different strategy is needed for C2050 (different computation partitioning, use of texture memory and additional optimizations). Data layout, mapping variants, and optimization strategies appear to try all valid possibilities among a (potentially large) enumerated set. The algorithm to generate optimization strategies, the number of variants generated, and strategies to prune variants are not described in detail in the paper. The compiler does not make use of constant or texture memory, and it is not clear how their approach can be expanded beyond BLAS kernels to more general application codes.

Par4All is also an automatic parallelization system that generates CUDA from sequential specification [Amini et al. 2011]. It implements a number of the same loop transformations, and relies on static analyses using abstract interpretation in polyhedral relational abstract domains. It does not employ autotuning. Rather, its focus is on automating and easing the migration of an existing code base to multi-core and many-core architectures. A complementary optimization in Amini et al. [2011] avoids copying temporary data between host and GPU.

Autotuning systems include self-tuning library generators such as ATLAS [Whaley et al. 2001; Whaley and Petitet 2005], PHiPAC [Bilmes et al. 1997], OSKI [Vuduc et al. 2005b], and SPIRAL [Püschel et al. 2005]. There are also other compiler autotuners that automatically generate and search a set of alternative implementations of a computation [Baskaran et al. 2008b; Pouchet et al. 2007, 2008; Girbal et al. 2006]. POET employs transformation scripts for autotuning, but does not use a polyhedral framework and does not target GPUs [Yi 2011]. Application-level autotuners use empirical search across parameter values and variants suggested by the application programmer [Chung and Hollingsworth 2004; Ren et al. 2008; Ansel et al. 2009].

A particular concern with autotuning is the size of the search space that must be evaluated. A significant body of research employs machine learning or heuristic search techniques in compilers to navigate complex search spaces and reduce the number of points to be evaluated (for example, Nisbet [1998], Barreteau et al. [1999], Stephenson et al. [2003], Agakov et al. [2006], Park et al. [2011], Cooper et al. [1999]). In our own prior work, we have used parallel rank order search for this purpose [Tiwari et al. 2009]. In this article, we have used a different approach in which models and heuristics prune the search space to just tens to hundreds of points. With such a small search space, our view is that it is more appropriate to search using empirical methods rather than using the complex (and sometimes fragile) machinery of machine learning, consistent with our own prior work and others [Chen et al. 2005a; Yotov et al. 2005; Qasem and Kennedy 2006]. With proper design, a priori pruning from programmer or compiler developer in combination with learning or heuristic search could combine the advantages of both techniques, an area of interest for us in future work.

7. SUMMARY AND CONCLUSION

This article described a compilation system organized around the concepts of transformation recipes and autotuning which automatically maps a sequential loop nest

computation to an equivalent high-performance CUDA implementation for a GPU. Overall this article makes a case for autotuning compiler technology as a productivity enhancement for developing high-performance CUDA code for loop nest computations, and porting code from one platform to the next generation. The results show that our system is able to achieve performance comparable and sometimes better than manually tuned code and outperforms a state-of-the-art GPU compiler.

REFERENCES

- AGAKOV, F., BONILLA, E., CAVAZOS, J., FRANKE, B., FURSIN, G., O'BOYLE, M. F. P., THOMSON, J., TOUSSAINT, M., AND WILLIAMS, C. K. I. 2006. Using machine learning to focus iterative optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*.
- AMINI, M., COELHO, F., IRIGOIN, F., AND KERYELL, R. 2011. Static compilation analysis for host-accelerator communication optimization. In *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*. Also Tech. Rep. MINES ParisTech A/476/CRI.
- ANSEL, J., CHAN, C., WONG, Y. L., OLSZEWSKI, M., ZHAO, Q., EDELMAN, A., AND AMARASINGHE, S. 2009. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, 38–49.
- BAILEY, D. H., CHAME, J., CHEN, C., DONGARRA, J., HALL, M., HOLLINGSWORTH, J. K., HOVLAND, P., MOORE, S., SEYMOUR, K., SHIN, J., TIWARI, A., WILLIAMS, S., AND YOU, H. 2008. PERI auto-tuning. *J. Phys.* 125, 1.
- BARRETEAU, M., BODIN, F., CHAMSKI, Z., CHARLES, H.-P., EISENBEIS, C., GURD, J. R., HOOGERBRUGGE, J., HU, P., JALBY, W., KISUKI, T., KNIJNENBURG, P. M. W., VAN DER MARK, P., NISBET, A., O'BOYLE, M. F. P., ROHOU, E., SEZNEC, A., STÖHR, E., TREFFERS, M., AND WIJSHOFF, H. A. G. 1999. Oceans - optimising compilers for embedded applications. In *Proceedings of the 5th International Conference on Parallel Processing (Euro-Par'99)*. 1171–1175.
- BASKARAN, M. M., BONDHUGULA, U., KRISHNAMOORTHY, S., RAMANUJAM, J., ROUNTEV, A., AND SADAYAPPAN, P. 2008a. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.
- BASKARAN, M. M., BONDHUGULA, U., KRISHNAMOORTHY, S., RAMANUJAM, J., ROUNTEV, A., AND SADAYAPPAN, P. 2008b. A compiler framework for optimization of affine loop nests for GPGPUs. In *Proceedings of the 22nd Annual International Conference on Supercomputing*.
- BASKARAN, M. M., RAMANUJAM, J., AND SADAYAPPAN, P. 2010. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the International Conference on Compiler Construction*.
- BELL, N. AND GARLAND, M. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the ACM/IEEE Conference on Supercomputing (SC'09)*.
- BILMES, J., ASANOVIĆ, K., CHIN, C.-W., AND DEMMEL, J. 1997. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *Proceedings of the ACM International Conference on Supercomputing*.
- CATANZARO, B., GARLAND, M., AND KEUTZER, K. 2011. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*. ACM, New York, 47–56.
- CHAME, J., CHEN, C., HALL, M., HOLLINGSWORTH, J., MAHINTHAKUMAR, K., MARIN, G., RAMALINGAM, S., SRIPATHI, S., SRIPATHI, V., AND TIWARI, A. 2011. Peri autotuning of pflotran. *J. Phys.*
- CHEN, C. 2007. Model-Guided empirical optimization for memory hierarchy. Ph.D. thesis, University of Southern California.
- CHEN, C., CHAME, J., AND HALL, M. 2005a. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proceedings of the International Symposium on Code Generation and Optimization*.
- CHEN, C., CHAME, J., AND HALL, M. 2008. CHiLL: A framework for composing high-level loop transformations. Tech. rep. 08-897, University of Southern California.
- CHEN, C., CHAME, J., HALL, M., AND LERMAN, K. 2005b. A systematic approach to model-guided empirical search for memory hierarchy optimization. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*.
- CHEN, C., CHAME, J., NELSON, Y., DINIZ, P., HALL, M., AND LUCAS, R. 2007a. Compiler-Assisted performance tuning. *J. Phys.* 78.

- CHEN, C., SHIN, J., KINTALI, S., CHAME, J., AND HALL, M. 2007b. Model-guided empirical optimization for multimedia extension architectures: A case study. In *Proceedings of the Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL'07)*.
- CHUNG, I.-H. AND HOLLINGSWORTH, J. K. 2004. Using information from prior runs to improve automated tuning systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*.
- COOPER, K. D., SCHIELKE, P. J., AND SUBRAMANIAN, D. 1999. Optimizing for reduced code space using genetic algorithms. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*.
- CUI, H., WANG, L., XUE, J., YANG, Y., AND FENG, X. 2011. Automatic library generation for BLAS3 on GPUs. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*.
- DATTA, K., MURPHY, M., VOLKOV, V., WILLIAMS, S., CARTER, J., OLIVER, L., PATTERSON, D., SHALF, J., AND YELICK, K. A. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 4.
- DONADIO, S., BRODMAN, J., ROEDER, T., YOTOV, K., BARTHO, D., COHEN, A., GARZARÁN, M. J., PADUA, D., AND PINGALI, K. 2005. A language for the compact representation of multiple program versions. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*.
- FRIGO, M. 1999. A fast fourier transform compiler. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- GIRBAL, S., VASILACHE, N., BASTOUL, C., COHEN, A., PARELLO, D., SIGLER, M., AND TEMAM, O. 2006. Semi-Automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.* 34, 3, 261–317.
- HALL, M., CHAME, J., SHIN, J., CHEN, C., RUDY, G., AND KHAN, M. M. October, 2009. Loop transformation recipes for code generation and auto-tuning. In *Proceeding of the 22nd International Workshop on Languages and Compilers for Parallel Computing*.
- HAN, T. D. AND ABDELRAHMAN, T. S. 2009. *hiCUDA: A high-level directive-based language for GPU programming*. In *Proceedings of the 2nd Workshop on General-Purpose Computation on Graphics Processing Units*.
- IMPACT. 2007. The parboil benchmark suite, 2007. <http://www.crhc.uiuc.edu/IMPACT/parboil.php>.
- IRGOIN, F. AND TRIOLET, R. 1988. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '88)*. ACM, New York, 319–329.
- KENNEDY, K. AND MCKINLEY, K. 1992. Optimizing for parallelism and data locality. In *ACM International Conference on Supercomputing*.
- KIRK, D. AND HWU, W. 2010. *Programming Massively Parallel Processors: A Hands-On Approach*. Morgan Kaufmann Publishers.
- LEE, S., MIN, S.-J., AND EIGENMANN, R. 2009. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.
- LEUNG, A., VASILACHE, N., MEISTER, B., BASKARAN, M., WOHLFORD, D., BASTOUL, C., AND LETHIN, R. 2010. 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*.
- NATH, R., TOMOV, S., AND DONGARRA, J. 2010. An improved magma gemm for fermi gpus. Tech. rep. UT-CS-10-655 (also LAPACK working note 227), University of Tennessee Computer Science technical report. July.
- NISBET, A. P. 1998. GAPS: Iterative feedback directed parallelisation using genetic algorithms. In *Workshop on Profile and Feedback-Directed Compilation*.
- PARK, E., POUCHET, L.-N., CAVAZOS, J., COHEN, A., AND SADAYAPPAN, P. 2011. Predictive modeling in a polyhedral optimization space. In *Proceedings of the 9th IEEE/ACM International Symposium on Code Generation and Optimization (CGO'11)*. IEEE Computer Society Press, 119–129.
- POUCHET, L.-N., BASTOUL, C., COHEN, A., AND CAVAZOS, J. 2007. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *Proceedings of the International Symposium on Code Generation and Optimization*.
- POUCHET, L.-N., BASTOUL, C., COHEN, A., AND VASILACHE, N. 2008. Iterative optimization in the polyhedral model: Part II, multi-dimensional time. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- PÜSCHEL, M., MOURA, J. M. F., JOHNSON, J. R., PADUA, D., VELOSO, M. M., SINGER, B. W., XIONG, J., FRANCHETTI, F., GAČIĆ, A., VORONENKO, Y., CHEN, K., JOHNSON, R. W., AND RIZZOLO, N. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* 93, 2, 232–275.

- QASEM, A. AND KENNEDY, K. 2006. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the ACM International Conference on Supercomputing*.
- REN, M., PARK, J. Y., HOUSTON, M., AIKEN, A., AND DALLY, W. J. 2008. A tuning framework for software-managed memory hierarchies. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.
- RUDY, G., KHAN, M. M., HALL, M., CHEN, C., AND JACQUELINE, C. 2011. A programming language interface to describe transformations and code generation. In *Proceedings of the 23rd International Conference on Languages and Compilers for Parallel Computing (LCPC'10)*. Springer 136–150.
- SHIN, J., HALL, M., CHAME, J., CHEN, C., FISCHER, P. F., AND HOVLAND, P. D. 2010. Speeding up nek5000 with autotuning and specialization. In *Proceedings of the ACM International Conference on Supercomputing*.
- SHIN, J., HALL, M. W., CHAME, J., CHEN, C., AND HOVLAND, P. D. 2009. Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology. In *Proceedings of the 4th International Workshop on Automatic Performance Tuning*.
- STEPHENSON, M., AMARASINGHE, S., MARTIN, M., AND O'REILLY, U.-M. 2003. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- TIWARI, A., CHEN, C., CHAME, J., HALL, M., AND HOLLINGSWORTH, J. K. 2009. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the 24th International Parallel and Distributed Processing Symposium*.
- TIWARI, A., CHEN, C., LIAO, C., CHAME, J., HOLLINGSWORTH, J., HALL, M., AND QUINLAN, D. 2011. Auto-tuning full applications: A case study. *Int. J. High Perform. Comput. Appl.*
- UENG, S.-Z., LATHARA, M., BAGHSORKHI, S. S., AND MEI W. HWU, W. 2008. Cuda-lite: Reducing GPU programming complexity. In *Proceedings of the 21st Annual Workshop on Languages and Compilers for Parallel Computing LCPC*. 1–15.
- UNKULE, S., SHALTZ, C., AND QASEM, A. 2012. Automatic restructuring of gpu kernels for exploiting inter-thread data locality. In *Proceedings of the 21st International Conference on Compiler Construction*. 21–40.
- VOLKOV, V. AND DEMMEL, J. W. 2008. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the Conference on Supercomputing*.
- VUDUC, R., DEMMEL, J., AND YELICK, K. 2005a. OSKI: A library of automatically tuned sparse matrix kernels. *J. Phys.* 16, 521–530.
- VUDUC, R., DEMMEL, J. W., AND YELICK, K. A. 2005b. Oski: A library of automatically tuned sparse matrix kernels. *J. Phys.* 16, 1.
- WHALEY, R. C. AND PETITET, A. 2005. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Softw. Pract. Exper.* 35, 2, 101–121.
- WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. 2001. Automated empirical optimization of software and the ATLAS project. *Parall. Comput.* 27, 1–2, 3–35.
- WOLFE, M. 1989. More iteration space tiling. In *Proceedings of the ACM/IEEE Conference on Supercomputing*.
- YANG, Y., XIANG, P., KONG, J., AND ZHOU, H. 2010. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- YI, Q. 2011. Poet: A scripting language for applying parameterized source-to-source program transformations. *Softw. Pract. Exper.*, 42, 6.
- YOTOV, K., PINGALI, K., AND STODGHILL, P. 2005. Think globally, search locally. In *Proceedings of the ACM International Conference on Supercomputing*.

Received February 2012; revised September 2012; accepted October 2012