Mint: Realizing CUDA performance in 3D Stencil Methods with Annotated C

Didem Unat
Dept. of Computer Science and Engineering
Univ. of California, San Diego
La Jolla, CA, USA
dunat@cs.ucsd.edu

Xing Cai
Simula Research Laboratory, Department of Informatics
University of Oslo
Norway
xingca@simula.no

Scott B. Baden
Dept. of Computer Science and Engineering
Univ. of California, San Diego
La Jolla, CA, USA
baden@cs.ucsd.edu

ABSTRACT

We present Mint, a programming model that enables the non-expert to enjoy the performance benefits of hand coded CUDA without becoming entangled in the details. Mint targets stencil methods, which are an important class of scientific applications. We have implemented the Mint programming model with a source-to-source translator that generates optimized CUDA C from traditional C source. The translator relies on annotations to guide translation at a high level. The set of pragmas is small, and the model is compact and simple. Yet, Mint is able to deliver performance competitive with painstakingly hand-optimized CUDA. We show that, for a set of widely used stencil kernels, Mint realized 80% of the performance obtained from aggressively optimized CUDA on the 200 series NVIDIA GPUs. Our optimizations target three dimensional kernels, which present a daunting array of optimizations.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Optimization, Compilers, Code generation*

General Terms

Algorithms, Design, Languages, Performance

Keywords

Automatic Translation and Optimization, CUDA, Parallel Programming Model, Stencil Computation

1. INTRODUCTION

GPUs are an effective means of accelerating certain types of applications, but an outstanding problem is how to manage the expansion of detail entailed in a highly tuned implementation. The details are well known; managing on-chip locality is perhaps the most challenging, and when managed effectively it offers considerable rewards. Finding a way around this stumbling block is crucial, not only at the desktop but also on high-end mainframes. Many top-500 systems [1] are populated with GPUs and their number is growing.

In order to make GPU technology more accessible to the user, we have developed Mint: a programming model based on programmer annotations (pragmas) and a source-to-source translator that implements the model. The Mint translator takes annotated C source code, and produces legal CUDA code that is subsequently compiled by `nvcc`, the CUDA C compiler [2]. Mint targets stencil methods, an important problem domain with a wide range of applications, with an emphasis on optimizations for 3-dimensional stencils.

A general-purpose translator is the philosopher's stone of high performance computing. One of the most successful compilers was CFT, the Cray vectorizing compiler that came into its own in the early to mid 1980s. This success story was a consequence of the fact that the Cray architecture possessed a reasonable "cartoon" for how to obtain high performance, a simple set of guidelines the programmer could grasp intuitively[1]. We contend that no such cartoon (yet) exists for GPUs; a general-purpose compiler would have to wade through a sea of possible optimizations, specific to each problem class [3, 4, 5, 6, 7].

We advocate a "middle ground" approach. We restrict the application space so that we can incorporate semantic content into the translation process, but choose an application space that is important enough to make the restriction reasonable. Mint, a domain-specific approach for stencil methods, embodies this approach. Like OpenMP [8], Mint employs pragmas to guide translation. Some Mint pragmas are inherited from OpenMP, but interpreted differently in order to address the requirements of GPU hardware.

The benefit of our approach is improved productivity. The Mint pragmas enable the programmer to control the hardware at a much higher level than that of CUDA C. In exchange for a restricted application space, we reap the benefits of specialization–namely a reduced optimization space–leading to improved performance compared to a general-purpose compiler. As a result, application developers use their valuable time to focus on the application, rather than on the idiosyncratic features of the hardware.

The contribution of this paper is as follows. We introduce a CUDA-free interface to implement stencil methods on GPU hardware, which is based on modest amounts of program annotation. We identify a set of pragmas with an

[1]Phil Colella, private communications

ICS'11, May 31–June 4, 2011, Tuscon, Arizona, USA.

OpenMP-like syntax that address system requirements not met by OpenMP, for example, in managing locality on the device, with an emphasis on supporting three-dimensional problems. We provide a source-to-source translator that incorporates domain specific knowledge to generate highly efficient CUDA C code, delivering performance that is competitive with hand-optimized CUDA. For a set of widely used stencil kernels, the Mint translator realized 80% of the performance of highly optimized hand-written CUDA on the Tesla C1060. The corresponding result on Fermi is 76%.

The paper is organized as follows. §2 provides background on the characteristics of stencil methods, and explains the motivation behind our work. §3 introduces the Mint programming model, §4 describes the translator and optimizer. We present performance results and evaluation in §5. §6 discusses related work. We conclude by evaluating the limitations of the model and translator.

2. STENCIL COMPUTATIONS

Stencil computations arise in some important classes of applications, notably finite difference discretization of partial differential equations [9] and in image processing [10]. They are good candidates for acceleration because they are highly data parallel and are typically implemented as nested for-loops. Many stencil methods are iterative; they sweep the mesh repeatedly, updating the mesh (or meshes) over a series of iterations. Each sweep will update each point of the mesh as a function of a surrounding neighborhood of points in space and time. We refer to a *stencil* as the neighborhood in the spatial domain. The most well known examples are the 5-point stencil approximation of the 2D Laplacian operator and the corresponding 7-point stencil in 3D, both shown in Fig. 1.

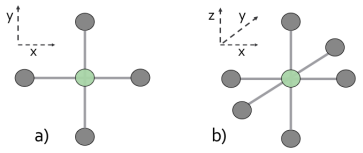

Figure 1: a) 5-point Stencil b) 7-point Stencil

As a motivating example, we consider the 3D heat equation $\partial u/\partial t = \kappa \nabla^2 u$, where ∇^2 is the Laplacian operator, and we assume a constant heat conduction coefficient κ and no heat sources. We use the following explicit finite difference scheme to solve the problem on a uniform mesh of points.

$$\frac{u^{n+1}_{i,j,k} - u^n_{i,j,k}}{\Delta t} = \frac{\kappa}{\Delta x^2}\big(u^n_{i,j,k-1} + u^n_{i,j-1,k} + u^n_{i-1,j,k} - 6u^n_{i,j,k} + u^n_{i+1,j,k} + u^n_{i,j+1,k} + u^n_{i,j,k+1}\big).$$

The superscript n denotes the discrete time step number (an iteration), the triple-subscript i, j, k denotes the spatial index. The quantity Δt is the temporal discretization (the timestep) and the mesh spacing Δx is equal in all directions. Note that the above formula is a 7-point computational stencil applicable only to inner grid points, and for simplicity we have omitted the treatment of boundary points.

The Mint program for solving this equation appears in Listing 1. (This program is also legal C, since a standard compiler will simply ignore the pragmas). Fig. 2 compares the double precision performance of three implementations

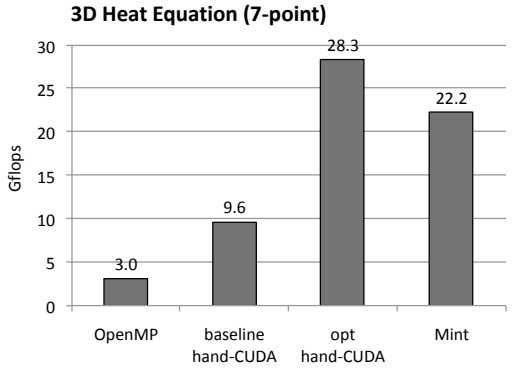

Figure 2: Comparing performance of OpenMP with 4 threads, hand-written unoptimized CUDA (unopt hand-CUDA), manually optimized CUDA (opt hand-CUDA), and Mint.

of the 3D heat equation solver running on an NVIDIA Tesla C1060 GPU. Our performance basis is an OpenMP implementation running on 4 cores of an Intel Nehalem processor. (§5 describes the testbeds used to obtain these results). The three CUDA versions are: baseline CUDA, aggressively hand-optimized CUDA, and Mint-generated CUDA. The baseline CUDA variant resolves all references to solution arrays through global (device) memory. It does not take advantage of fast on-chip shared memory. The heroically hand-optimized CUDA version improves performance by a factor of 3 over the baseline version. Running at 28.3 Gflops on the Tesla C1060 performance is comparable to previously published results [6] for this kernel[2], and is 9.4 times faster than the OpenMP variant. The programming effort required to realize this performance milestone is substantial; not only it manages threads and host-device transfers as in the baseline version, but also it tiles the iteration space to utilize fast on-chip device memory.

```
1  #pragma mint copy(U,toDevice,(n+2),(m+2),(k+2))
2  #pragma mint copy(Unew,toDevice,(n+2),(m+2),(k+2))
3
4  #pragma mint parallel default(shared)
5  {
6    int t=0;
7    while( t++ < T ){
8
9  #pragma mint for nest(all) tile(16,16,1)
10     for (int z=1; z<= k; z++)
11      for (int y=1; y<= m; y++)
12       for (int x=1; x<= n; x++)
13        Unew[z][y][x] = c0 *  U[z][y][x] +
14                c1 * (U[z][y][x-1] + U[z][y][x+1] +
15                      U[z][y-1][x] + U[z][y+1][x] +
16                      U[z-1][y][x] + U[z+1][y][x]);
17 #pragma mint single{
18    double*** tmp;
19    tmp = U; U = Unew; Unew = tmp;
20    }//end of single
21  }//end of while
22 }//end of parallel region
23
24 #pragma mint copy(U,fromDevice,(n+2),(m+2),(k+2))
```

Listing 1: Mint program for the 3D heat equation. (`Unew` corresponds to u^{n+1} and `U` to u^n and c_0=$1 - 6\kappa\Delta t/\Delta x^2$ and c_1=$\kappa\Delta t/\Delta x^2$.)

[2]Vasily Volkov kindly provided the 7-point stencil used in [6] which is equivalent to our 7-pt 3D heat equation kernel.

By comparison, the Mint-annotated version shown in Listing 1 came within 78% of the performance achieved by the heroically optimized CUDA, and required a much more modest programming effort. Only 6 pragmas were introduced and no existing source code was modified. The Mint `for` directive appearing at line 9 enables the translator to parallelize the `for` loop nest on lines (10-16). The `nest(all)` clause specifies that all loops should be parallelized. It is important to clarify that the Mint keyword `nest` specifies that a loop nest is to be parallelized using multi-dimensional thread structures. This is different from the nested parallelism in OpenMP, which specifies trees of threads. We need to specify higher dimensional threading structures in order to effectively utilize the GPU hardware, especially for three dimensional kernels.

3. MINT PROGRAMMING MODEL

3.1 System Assumptions

The Mint programming model assumes a system design depicted in Fig. 3, comprising a host processor and an accelerator[3]. Since accelerator technology is in a state of flux, our model abstracts away some aspects of how the system functions. For example, in an NVIDIA-based system, the host and device have physically distinct memories and the host controls all data motion between the two. However, Mint is neutral about how the data motion is brought about. Future systems may treat data motion differently, for example, the device may be able to initiate data transfers.

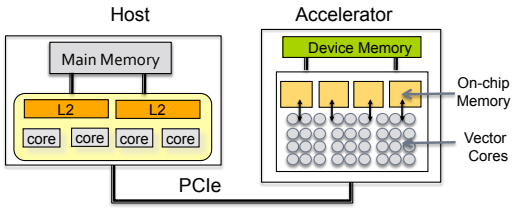

Figure 3: System Design

The accelerator contains several vector units that see a global device memory address space. The host invokes multithreaded kernels on the accelerator, which execute as a sequence of long vector operations that are partitioned into pieces by the accelerator and assigned to vector units. The vector elements are computed independently in an undefined order.

Each vector unit has a small local storage hierarchy that delivers much higher bandwidth (and a lower access time) than device memory. In NVIDIA 200 series devices, each vector processor has a private, software-managed on-chip memory. Fermi's on-chip memory is partitioned into a first level of cache and shared memory, and there is a second level of cache to back up L1. Mint does not assume a specific memory hierarchy, other than that there is fast and slow memory. We believe that Mint will not require special clauses to handle the Fermi architecture, except perhaps a flag to set the relative amounts of shared memory and L1 cache. Currently, Mint uses default configurations for Fermi: 48KB shared memory and 16KB L1.

[3]Extensions are required to handle multi-GPU platforms in Mint.

3.2 The Model

With the non-expert GPU user in mind, simplicity is our principal design goal for the Mint programming model. The principal effort should be to (1) identify and effectively parallelize time consuming-loop nests that can benefit from acceleration and (2) move data between host and accelerator.

A pragma-based programmer interface is a natural way of meeting our requirements. It allows us to optimize code for GPU execution incrementally, while maintaining a code base that can always run on a conventional CPU. Mint parallelizes a loop nest by associating one logical thread with some number of points in the iteration space of the nest. It then partitions and maps the logical threads onto physical ones, guided by any clauses that the programmer employs to tune the pragmas. The details come at a high level. For example, the programmers need not concern themselves with "flattening" a multidimensional array, which is common in GPU implementations. The translator takes care of the details.

Mint employs just five different directives, four of which appear in Listing 1. The `for` directive is the most important, as it identifies a parallel `for` loop nest and helps guide optimization. This construct resembles the familiar OpenMP directive, but is interpreted differently to meet the device capabilities. Mint creates a multi-dimensional array of threads to parallelize the specified loop nest. This capability of Mint is crucial; it enables the user to employ higher dimensional CUDA thread blocks, which are required to use the device effectively. Lastly, Mint helps the user manage the separate host and device memory spaces. The Mint programmer specifies transfers at a high level through the Mint `copy` directive, avoiding storage management and setup.

3.3 The Mint Pragmas

We next describe the 5 pragmas of Mint.

- **mint parallel [clauses]** indicates the start of a parallel region containing parallel work. These regions will be accelerated. Before control enters the parallel region, any data used in the region must have previously been transferred using the `copy` directive. Mint provides the `shared` clause as in OpenMP to indicate the data sharing rule between threads. Mint maps a shared array onto device memory and it is visible to all threads employed by the launched kernel. Variables not declared as shared are thread-private, and typically reside in registers.

- **mint for [clauses]** marks the succeeding `for` loop (or nested loops) for GPU acceleration and manages data decomposition and work assignment. Each such parallel `for` loop becomes a CUDA kernel. Mint may merge kernels if there is `no-wait` clause attached to a for loop directive. Other optional `for` clauses are followings:

  nest(# | all) indicates the depth of `for`-loop parallelization within a loop nest, which can be an integer, or the keyword `all` to indicate that all the loops are independent, and hence parallelizable. This clause supports multi-dimensional thread geometries. If the `nest` clause is not specified, Mint assumes that only the outermost `for` loop is parallelizable.

tile(t_x, t_y, t_z) specifies how the iteration space of a loop nest is to be subdivided into *tiles*. A data tile is assigned to a group of threads and the sizes are passed as parameters to the clause. In the CUDA context, a tile corresponds to the number of data points computed by a thread block.

chunksize(c_x, c_y, c_z) aggregates logical threads into a single CUDA thread. Each CUDA thread serially executes these logical threads via a C `for` loop. This clause is similar to the OpenMP *schedule* clause, though OpenMP confines chunking to a single dimension. Together with the `tile` clause, the `chunksize` clause establishes the number of CUDA threads that execute a tile. Specifically, the size of a CUDA thread block is $threads(t_x/c_x, t_y/c_y, t_z/c_z)$, as depicted in Fig. 4. In the absence of `tile` and `chunksize` clauses, the compiler will choose default values[4].

- **mint barrier** synchronizes all the threads.
- **mint single** indicates serial regions. Depending on the requirements, either a host or a single device thread executes the region.
- **mint copy(src | dst, toDevice | fromDevice,** [N_x, N_y, N_z, ...]) expresses data transfers between the host and device. Mint binds a host array to the corresponding device array using this directive. Mint handles the declaration, allocation/deallocation, and data transfers (with padding for alignment) on the device. The parameters *toDevice* and *fromDevice* indicate the direction of the copy. The remaining parameters are optional, and specify the array dimensions. This directive in fact is the only hint to a reader of Mint code about the use of an accelerator. We choose to get help from the programmer for the sake of better performance in terms of *when* the copy should occur, and between *which* arrays.

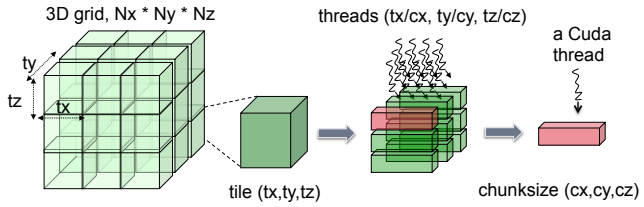

Figure 4: A 3D grid is broken into 3D tiles based on the tile clause. Elements in a tile are divided among a thread block based on chunksize clause.

To summarize, Mint will attempt to migrate a parallel region that contains at least one for-loop to the accelerator. The `barrier` directive will be translated into costly global synchronization among CUDA threads. The `single` directive will cause the indicated code segment to run on the host or as a single device thread.

Those familiar with OpenMP will recognize the `parallel`, `for`, `single` and `barrier` pragmas. A legal Mint program can be converted to a legal OpenMP program using string substitution, or, it could be compiled as it is by a standard C compiler, which would ignore the Mint pragmas. Thus, the code can always be run on conventional hardware.

[4]Mint currently chooses 16x16x1 tiles with a chunksize of 1 in all dimensions, but the default is configurable.

4. C TO CUDA TRANSLATION

We have developed a fully automated translation and optimization system for the Mint programming model. To construct our source-to-source translation and analysis tools, we used the ROSE compiler framework[11, 12], open source software developed and maintained at Lawrence Livermore National Laboratory. ROSE is a convenient tool for developing our infrastructure, because it provides an API for generating and manipulating in memory representations of Abstract Syntax Trees (ASTs).

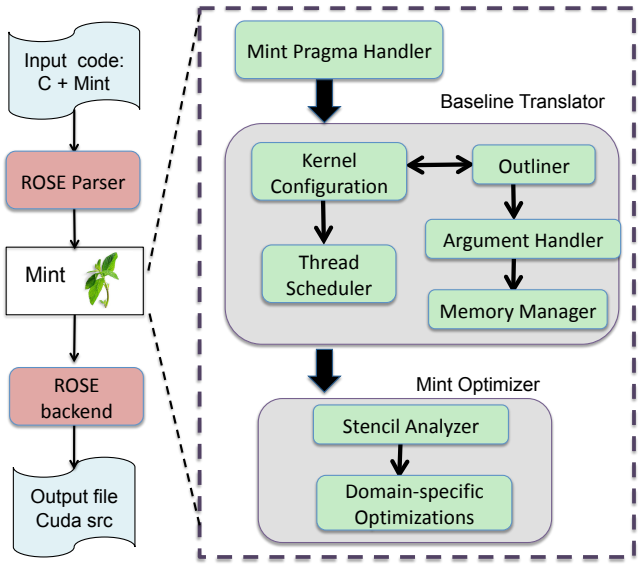

Figure 5: Modular design of Mint Translator and translation flow.

4.1 Mint Translator

Fig. 5 shows the modular design of the Mint translator and the translation work flow. The input to the compiler is C source code annotated with Mint pragmas. The *Pragma Handler* parses the Mint directives and clauses. Once the translator has constructed the AST, it queries the parallel regions containing data parallel for-loops. Directives in a candidate parallel region go through several transformation steps inside the Baseline Translator: *Outliner*, *Kernel Configuration*, *Argument Handler*, *Memory Manager*, and *Thread Scheduler*.

The *Baseline Translator* generates both device kernel code and host code. With the help of the `mint for` directive and attendant clauses, the transformer determines the kernel configuration parameters, i.e. CUDA thread block and grid sizes. *Outliner* outlines the candidate parallel for-loop into a function: a CUDA kernel. It moves the body of the loop into a newly-created `__global__` function and replaces the statement it vacates (including the original `for` directive) with a launch of the newly-created kernel. *Argument handler* works with *Outliner* and determines which variables are local to the function and which need to be passed as arguments. Naturally, all the parameters in the function argument become kernel call parameters. Depending on whether these parameters are vectors or scalars, they may require data transfers.

Unless already residing on the device (say from a previous loop body), any vector arguments need to be transferred to device memory. The *Memory Manager* checks whether or not the programmer requested the transfer of vector variables via the Mint `copy` pragma. If not, then the compiler tries to infer the information necessary to generate the code for the transfer. In the case of statically declared arrays, it will carry out the translation, but it cannot determine this information for dynamically allocated arrays. In such cases it will issue a message and abort.

The *Thread Scheduler* inserts code into the generated kernel body to compute global thread IDs. It also rewrites references (i.e array subscripts) to original `for` loop indices to use these global thread IDs instead. When `chunksize` is set to 1, the loop iteration space is mapped one-to-one onto physical threads. Otherwise, the compiler inserts a serial loop into the kernel so that the thread can compute the multiple points assigned to it.

The output of the *Baseline Translator* makes all memory references through device memory. If the optimization flag is turned on, the *Mint optimizer* performs both general and stencil method-specific optimizations on the generated code. We will discuss these optimizations in depth shortly.

```
1 /* Mint: Replaced Pragma: #pragma mint copy */
2 cudaExtent ext_dU = make_cudaExtent(...);
3
4 /* Mint: Malloc on the device */
5 cudaPitchedPtr ptr_dU;
6 cudaMalloc3D(&ptr_dU,ext_dU);
7 ...
8 /* Mint: Copy host to device */
9 cudaMemcpy3DParms param_dU = {0};
10 param_dU.srcPtr = make_cudaPitchedPtr(...);
11 param_dU.dstPtr = ptr_dU;
12 param_dU.extent = ext_dU;
13 param_dU.kind = cudaMemcpyHostToDevice;
14 stat_dU = cudaMemcpy3D(&param_dU);
15 ...
16 while(t++ < T){
17
18   //Kernel configuration parameters
19   int num3block = (k-1+1)%1 == 0?(k-1+1)/1:(k-1+1)/1+1;
20   int num2block = (m-1+1)%16 == 0?(m-1+1)/16:(m-1+1)/16+1;
21   int num1block = (n-1+1)%16 == 0?(n-1+1)/16:(n-1+1)/16+1;
22
23   dim3 blockDim(16,16,1);
24   dim3 gridDim(num1block, num2block * num3block);
25
26   float invYnumblock = 1.0/num2block;
27   //kernel launch
28   mint_1_1527<<<gridDim,blockDim>>>(...);
29
30   cudaThreadSynchronize();
31   ...
32   double* tmp = (double*)ptr_dU.ptr;
33   ptr_dU.ptr = ptr_dUnew.ptr;
34   ptr_dUnew.ptr = (void*)tmp;
35
36  }//end of while
37 /* Mint: Replaced Pragma: #pragma mint copy */
38 /* Mint: Copy device to host */
```

Listing 2: Host code generated by the Mint translator for the 7-point 3D stencil input.

Listing 2 shows the host code generated by the Mint translator for the 7-point 3D stencil example provided in Listing 1. For the sake of clarity, we have omitted some of the details. Lines (1-14) perform memory allocation and data transfer for the variable U, corresponding to line 1 in Listing 1. Mint uses CUDA *pitched* pointer type and *cudaMalloc3D* to pad storage allocation on the device to ensure hardware alignment requirements are met [13]. Lines (18-24) compute the kernel configuration parameters based on values provided by the user (i.e. via pragma clauses), if there are any, else it chooses default values. Under CUDA, a grid of thread blocks can not have more than 2 dimensions. A common trick in CUDA is to emulate 3D grids (lines (24-26)) by mapping two dimensions of the original iteration space onto one dimension of the kernel. Line (28) launches the kernel and line (30) is a global barrier across all threads employed in the kernel launch. Lines (32-34) perform the pointer swap on the device pointers.

```
1 __global__ void mint_1_1527(cudaPitchedPtr ptr_dU,
2    cudaPitchedPtr ptr_dUnew,int n,int m, int k,
3    double c0,double c1,int blocksInY,float invBlocksInY)
4 {
5   double* U = (double *)(ptr_dU.ptr);
6   int widthU = ptr_dU.pitch / sizeof(double );
7   int sliceU = ptr_dU.ysize * widthU;
8   ...
9   int _idx = threadIdx.x + 1;
10  int _gidx = _idx + blockDim.x * blockIdx.x;
11  int _idy = threadIdx.y + 1;
12  int _idz = 1;
13  int blockIdxz = blockIdx.y * invBlocksInY;
14  int blockIdxy = blockIdx.y - blockIdxz * blocksInY;
15  int _gidy = _idy + blockIdxy * blockDim.y;
16  int _gidz = _idz + blockIdxz;
17  int indU = _gidx +_gidy*widthU +_gidz*sliceU;
18  int indUnew = _gidx +_gidy*widthUnew +_gidz*sliceUnew;
19
20  if (_gidz >= 1 && _gidz <= k)
21   if (_gidy >= 1 && _gidy <= m)
22    if (_gidx >= 1 && _gidx <= n)
23      Unew[indUnew] = c0 * U[indU]
24        + c1 * (U[indU - 1] + U[indU + 1]
25              + U[indU - widthU] + U[indU + widthU]
26              + U[indU - sliceU] + U[indU + sliceU]);
27 }//end of function
```

Listing 3: Unoptimized kernel generated by Mint for the 7-point 3D stencil input.

Listing 3 shows the unoptimized kernel generated by Mint. All the memory accesses pass through global memory. Lines (5-7) unpack the CUDA pitched pointer U, while lines (9-18) compute local and global indices using thread and block IDs. Lines (20-22) are `if` statements derived from the `for` statements in the original annotated source. Finally, the lines (23-26) perform the stencil mesh sweep on the flattened arrays. In CUDA, multi-dimensional indexing works correctly only if the `nvcc` compiler knows the pitch of the array at compile time. Therefore, Mint converts such indices appearing in the annotated code into their 1D equivalents.

In the generated kernel code shown in Listing 3, each CUDA thread updates a single element of `Unew`. However, there is a performance benefit to aggregating array elements so that each CUDA thread computes more than one point. Mint allows the programmer to easily manage the mapping of work to threads using the `chunksize` clause. The following code fragment shows part of the generated kernel when the programmer sets a chunking factor in the z dimension, the 3^{rd} argument of the `chunksize` clause. As an optimization, the translator moves `if` statements outside the `for` statement. It also computes the bounds of the `for`-loop.

```
1 if (_gidy >= 1 && _gidy <= m)
2   if (_gidx >= 1 && _gidx <= n)
3     for (_gidz = _gidz; _gidz <= _upper_gidz; _gidz++)
4       Unew[indUnew] = c0 * U[indU] + ...
```

Chunking affects the kernel configuration (i.e. size of the thread blocks) by rendering a smaller number of thread blocks with "fatter" threads. This clause is particularly helpful when combined with on-chip memory optimizations because it enables re-use of data. The reason will be explained in more detail in §4.2.3.

4.2 Mint Optimizer

The Mint optimizer incorporates a number of optimizations that we have found to be useful in optimizing stencil methods written in CUDA. Our insight is that occupancy should not be maximized to the exclusion of properly managed on-chip locality. Although we need a sufficient number of threads to overlap data transfers with computation, if global memory accesses are too frequent, we may not be able to hide their latency.

4.2.1 Stencil Analyzer

To optimize for on-chip memory re-use, the optimizer must analyze the structure of the stencil(s) appearing in the application. Based on this analysis, it then chooses an optimization strategy appropriate for the determined stencil pattern. The analyzer first determines if the kernel is eligible for optimization by checking to see if the pattern of array subscripts involves a central point and nearest neighbors only, that is, index expressions of the form $i \pm k$, where i is an index variable and k is a small constant.

Next, the analyzer determines the shape of the stencil. The shape affects not only the number of ghost cell loads, but also the amount of shared memory needed to optimize the kernel. Consider two 3D stencils as shown in Fig. 6. A common case is the 7-point stencil that couples the 6 nearest neighbor points (in the Manhattan directions) to the central point. The 7-point stencil requires that only one xy-plane of data be kept in shared memory at a time. By comparison, the 19-point stencil shares data in all three xy-planes because of the diagonal points, and thus has a different shared memory requirement.

Stencils may also differ in the required number of ghost cell loads depending on whether they are symmetric or cover a large neighborhood. For instance, a thread block processing a 3D tile (t_x, t_y, t_z) for a non-compact, 4th order, 13-point stencil would load a tile with size $(t_x + 2k, t_y + 2k, t_z + 2k)$ including ghost cells, where k is 2. The analyzer passes this information to the optimizer.

4.2.2 On-chip Memory Optimizer

When Mint optimizations are not enabled, the translated code does not utilize on-chip memory to handle array references. All array references go through global memory, and frequently referenced data is not re-used. For example, in the 7-point stencil kernel, each thread independently loads all six nearest neighbors even though each nearest neighbor is used by 6 nearby points. Owing to this high volume of potential re-use, the key optimization for stencil kernels is to buffer global memory accesses on chip, i.e. in shared memory and registers.

Shared Memory. The optimizer lets a thread block load a block of data –with respective ghost cells–into shared memory. One of the issues of using shared memory effectively is that ghost cells require special handling. We have two choices. In the first case we adjust the thread block size so that each thread is assigned a single interior point[5]. Some threads are then responsible for loading ghost cells as well as performing computation. In the second case, the thread block is large enough to cover ghost cells as well as interior points. The drawback of this approach is that we leave some border threads idle during computation. Mint implements the former approach.

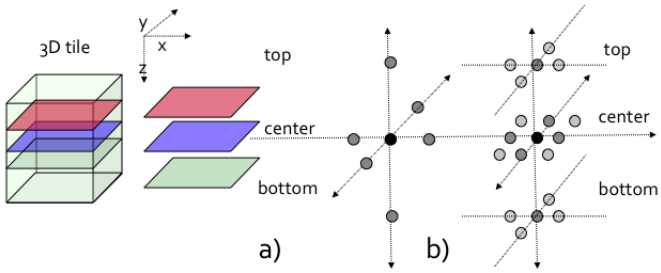

Figure 6: The black point is the central point of the stencil. a) 7-point stencil, b) 19-point stencil.

There may be various choices regarding which array(s) to place in shared memory. The optimizer chooses the most frequently referenced array(s). In case of a tie involving more than one array, the array(s) accessed along the fastest varying dimension are given priority since they require that fewer planes be kept in shared memory. However, because shared memory is a scarce resource (16KB per vector unit in the 200 series GPUs) there are obvious limitations to how many arrays we can buffer in shared memory, especially for stencils with many points. For example, owing to diagonal points, a 19-point stencil would require 3 times the space needed for a 7-point stencil. If we needed an extra mesh, say, for a variable coefficient problem, and we choose to keep it in shared memory as well, then we further increase the shared memory requirement: we double it.

Registers. We use registers to alleviate pressure on shared memory, enabling us to increase device occupancy by increasing the number of thread blocks. We improve performance still further, since an instruction with operands in shared memory runs at a lower rate (about 66% of the peak) compared to when its operands are in registers [3]. The optimizer stores, in registers, stencils that use a center point. As a result, a thread will read values from registers if available instead of reading from shared memory.

The optimizer conducts a def-use analysis to retrieve information about when to transfer data between on-chip and device memory. For example, read-only arrays are not written back to device memory and write-only arrays are not read from memory.

4.2.3 Loop Aggregation

As described previously, we can improve re-use still further by employing the `chunksize` clause in tandem with on-chip memory optimizations. The effect is to assign each CUDA thread more than one point in the iteration space of the loop nest, enabling values stored in shared memory to be shared in updating adjacent points. This optimization is particularly helpful for 3D stencils as it allows the reuse of data already in shared memory, by chunking along

[5]Without any loss of generality, we assume a chunk size of one.

Stencil kernel	Mathematical description	In,Out arrays	Read,Write per point	Operations per point
2D Heat 5-point	$u^{n+1}_{i,j} = c_0 u^n_{i,j} + c_1 \left(u^n_{i\pm1,j} + u^n_{i,j\pm1} \right)$	1,1	5,1	2(∗),4(+)
3D Heat 7-point	$u^{n+1}_{i,j,k} = c_0 u^n_{i,j,k} + c_1 \left(u^n_{i\pm1,j,k} + u^n_{i,j\pm1,k} + u^n_{i,j,k\pm1} \right)$	1,1	7,1	2(∗),6(+)
3D Poisson 7-point	$u^{n+1}_{i,j,k} = c_0 b_{i,j,k} + c_1 \left(u^n_{i\pm1,j,k} + u^n_{i,j\pm1,k} + u^n_{i,j,k\pm1} \right)$	2,1	7,1	2(∗),6(+)
3D Heat 7-point variable coefficient	$u^{n+1}_{i,j,k} = u^n_{i,j,k} + b_{i,j,k}$ $+c \Big[\kappa_{i+\frac{1}{2},j,k} \left(u^n_{i+1,j,k} - u^n_{i,j,k} \right) - \kappa_{i-\frac{1}{2},j,k} \left(u^n_{i,j,k} - u^n_{i-1,j,k} \right)$ $+\kappa_{i,j+\frac{1}{2},k} \left(u^n_{i,j+1,k} - u^n_{i,j,k} \right) - \kappa_{i,j-\frac{1}{2},k} \left(u^n_{i,j,k} - u^n_{i,j-1,k} \right)$ $+\kappa_{i,j,k+\frac{1}{2}} \left(u^n_{i,j,k+1} - u^n_{i,j,k} \right) - \kappa_{i,j,k-\frac{1}{2}} \left(u^n_{i,j,k} - u^n_{i,j,k-1} \right) \Big]$	3,1	15,1	7(∗),13(+),6(−)
3D Poisson 19-point	$u^{n+1}_{i,j,k} = c_0 \Big[b_{i,j,k} + c_1 \left(u^n_{i\pm1,j,k} + u^n_{i,j\pm1,k} + u^n_{i,j,k\pm1} \right)$ $+u^n_{i\pm1,j\pm1,k} + u^n_{i\pm1,j,k\pm1} + u^n_{i,j\pm1,k\pm1} \Big]$	2,1	19,1	2(∗),18(+)

Table 1: A summary of stencil kernels used in this paper. The ± notation is short hand to save space, $u^n_{i\pm1,j} = u^n_{i-1,j} + u^n_{i+1,j}$. 19-pt stencil Gflop/s rate is calculated based on the reduced flop counts which is 14.

the z-dimension. The programmer can explicitly trigger this optimization by setting a chunking factor in the z-dimension (the 3^{rd} argument of `chunksize` clause). Mint implements the optimization with the help of registers and shared memory. As shown in Fig. 4, a 3D input grid is subdivided into 3D tiles and then each 3D tile is further divided into a series of 2D planes. In this scheme, we use a buffer with three rotating planes. A plane that has been read from global memory starts as the bottom plane, continues as the center plane and then migrates to the top.

This optimization is referred to as partial 3D blocking in the literature. Rivera and Tseng [14] proposed the method for stencil-based computation on traditional processors. The technique is shown to be highly effective [15] on software-managed memory architectures such as the STI Cell Broadband Engine [16]. We incorporated this optimization into our compiler through a simple clause, `chunksize`, saving a good deal of programming overhead in managing data decomposition and index computation.

Since we let a thread compute multiple elements in the slowest varying dimension, we can reduce some of the index calculations by assigning a thread more than one row in the y-dimension. We do this by setting the y-dimension of the `chunksize` clause to a value greater than one. However, there is a potential drawback. If we allow a thread to compute multiple elements we increase the number of registers used by the thread, constraining the range of usable chunk sizes in the y-dimension. It is disadvantageous to apply loop aggregation to the x-dimension because it disrupts the temporal locality across threads, assigning successive elements to a thread in the fastest varying dimension. Such locality is needed to ensure coalesced accesses to global memory in the NVIDIA GPU [13].

4.3 Limitations

An obvious limitation of our translator is that it is domain-specific. Mint targets stencil computations and our optimizations are specific to this problem domain. We believe that the benefit of this approach outweighs the disadvantages of the limitations. We can incorporate domain-specific optimizations into our compiler, resulting in improved performance.

Our translator can perform subscript analysis on multi-dimensional array references only. It cannot analyze "flattened" array references, for example, when determining stencil structures, and may incorrectly disqualify a computation as not expressing a stencil pattern. In addition, the compiler cannot determine the shapes of dynamically allocated arrays. In this case we require that the programmer use the `copy` pragma to express data transfers, though many of the details are hidden from view.

Our translator is currently capable of generating code that utilizes only a single CUDA device. Generating code for multi-GPU execution would require more complex analysis to manage the ghost cell communication. This remains as future work.

5. PERFORMANCE RESULTS

We next demonstrate the effectiveness of the Mint translator, using a set of widely used stencil kernels in two and three dimensions. The kernels were chosen because of their different patterns of memory access and computational intensity. Tab. 1 summarizes the characteristics of each kernel.

We compare the performance of Mint-generated CUDA with hand-written (and optimized) CUDA and with OpenMP. All GPU results were obtained from an NVIDIA Tesla C1060 with 4GB device memory. The device is 1.3 capable. It has 30 streaming multiprocessors (each with 8 cores), which we will refer to as vector units [6], each running at 1.3 GHz. Each vector unit has substantial fast on-chip memory in the form of a 16KB scratch-pad memory and 64KB of registers. Both hand-coded and translated CUDA codes were compiled with `nvcc 3.2` and `-O3` optimization. The host CPU is a server based on a 2.0 GHz quad-core Intel Nehalem-EP processor (E5504, or "Gainstown") with 4MB of L3 cache and 16GB of main memory. OpenMP programs were compiled using `gcc 4.4.3` and command line options `-O3 -fopenmp`. All computations were run in double precision.

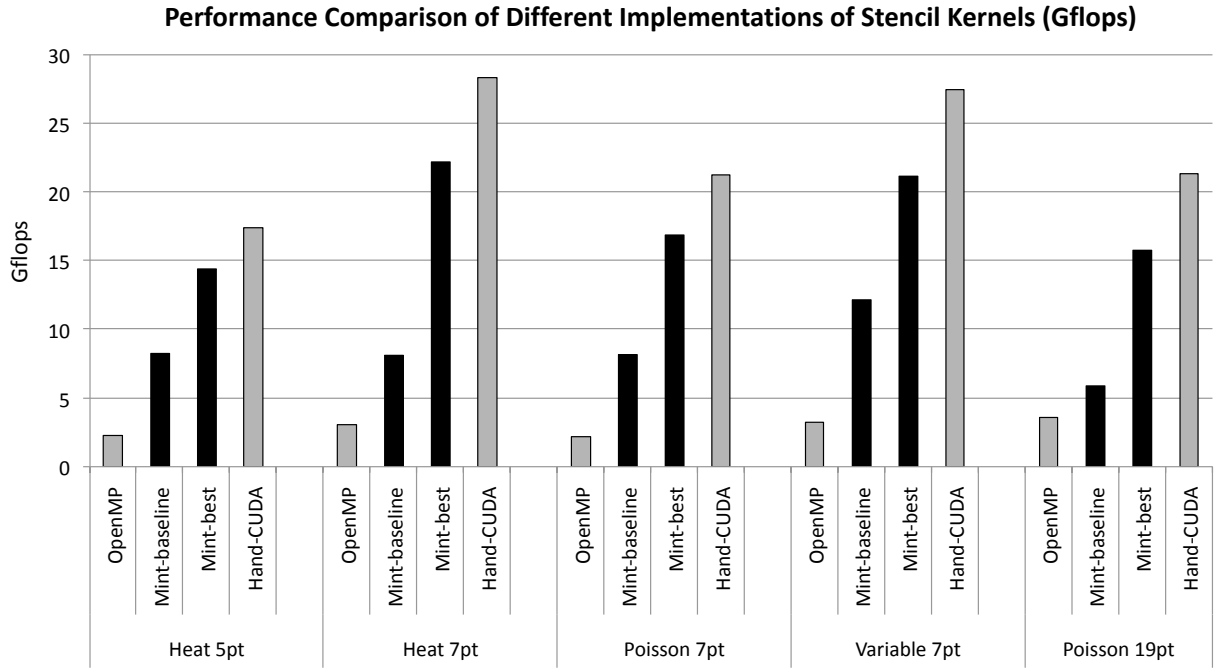

Figure 7: Performance comparison of the kernels. OpenMP ran with 4 threads on conventional hardware. Mint-baseline corresponds to the Mint baseline translation without using the Mint optimizer, Mint-best with optimizations turned on, and Hand-CUDA is hand-optimized CUDA. The Y-axis shows the measured Gflop rate. Heat 5-pt is a 2D kernel, the rest are 3D.

5.1 Performance Comparison

Fig. 7 compares the performance of the stencil kernels and their 4 different implementations. *OpenMP* results were obtained by running 4 OpenMP threads on the host. The remaining versions ran on the device. *Mint-baseline* is the result of compiling Mint-annotated C source without enabling any Mint optimizations, and *Mint-best* turns on all Mint optimizations. (Mint-generated CUDA was compiled with `nvcc` without any modification.) Lastly, *Hand-CUDA* refers to manually implemented and optimized CUDA C. Mint was **not** used to produce this code. The figure shows that even our baseline Mint code outperforms the OpenMP code running on 4 CPU cores and provides on average a 3x speedup. The Mint optimizer improves performance still further, achieving between 78% and 83% of the performance of the aggressively hand-optimized CUDA versions. The optimizer delivers 4.5 to 8 times the performance of OpenMP running with 4 threads.

We have also run the Mint-generated code on Fermi (Tesla C2050). On average, Mint achieved 76% of the performance of hand-written CUDA. Fermi results were obtained without modifying either the translator or the hand-written CUDA kernels, thus neither Mint nor our hand-coded implementations have been tuned for Fermi. Our preliminary results are therefore subject to change though they are quite promising.

5.2 Optimization Levels

The Mint optimizer provides three levels of optimization: *opt-1, opt-2 and opt-3*. The optimizations are cumulative, thus opt-3 includes all three optimizations.

- *Opt-1* turns on shared memory optimization (§4.2.2).
- *Opt-2* adds loop aggregation, which benefits shared memory (§4.2.3).
- *Opt-3* adds register optimizations (§4.2.2).

As mentioned previously, *baseline* refers to the performance of the Mint baseline translator. This variant resolves all array references through device memory. It does not buffer these references in on-chip memory and makes many redundant global memory accesses. On the other hand, it does create multi-dimensional thread blocks, if the `nest` clause is used in the mint-`for` directive.

Fig. 8 compares the performance of 4 kernels at different levels of optimization. We report performance as a speedup over Mint-generated code without any compiler optimization turned on, though we do allow source code optimization via pragmas. In all Mint code, loops are annotated with `nest(all)`. In 3D we chose `tile` sizes as follows, all resulting in 16×16 thread blocks, except for the 2D Heat kernel, which uses a (16,16) tile size.

- *baseline* and *opt-1* use 2D tiles: `tile(16,16,1)`. Each thread computes just one element.
- *opt-2* and *opt-3* use 3D tiles: `tile(16,16,64)`. Each thread computes 64 elements in the z-dimension.

Fig. 8 shows the performance impacts of the different optimizations. Generally, performance improves with the level of optimization, though not in all cases. Not all optimizations are relevant in two dimensions, for example, loop aggregation (*opt-2*). Shared memory optimization (*opt-1*) is always

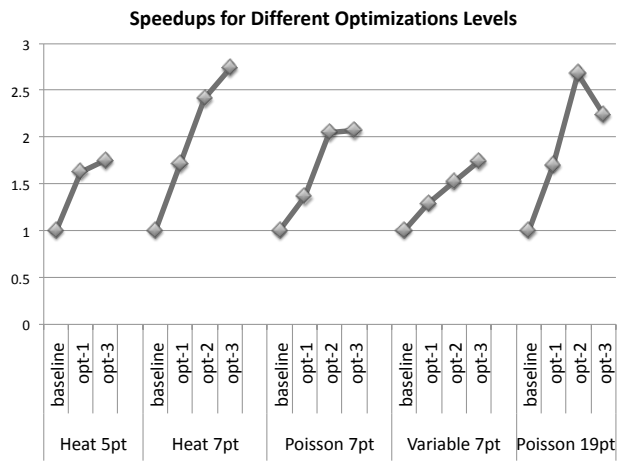

Figure 8: Effect of the Mint optimizer on the performance. The baseline resolves all the array references through device memory.

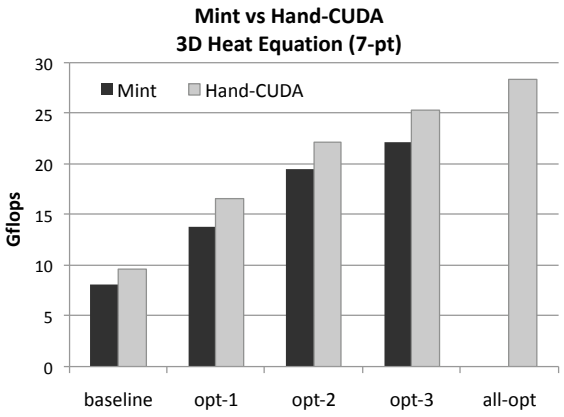

Figure 9: Comparing the performance of Mint-generated code and hand-coded CUDA. *All-opt* indicates additional optimizations on top of *opt-3*.

helpful. It takes advantage of significant opportunities for re-use in stencil kernels, reducing global memory traffic significantly. We observed performance improvements in the range of 30-70%.

The speedups attributed to tiling for shared memory differ according to the number of data streams that exhibit reuse. The Poisson 7-pt and Heat 7-pt stencils have the same flop counts but Poisson 7-pt requires an additional input grid, increasing the number of data streams from 2 to 3. As a result, its performance lags behind the Heat 7-pt because one of the input arrays (the right hand side) does not exhibit data reuse and cannot benefit from shared memory. On the other hand, the 7-pt variable coefficient kernel (Variable 7-pt) has 4 data streams, and 2 of the streams exhibit reuse. The Mint optimizer places the corresponding two meshes in shared memory, trading off occupancy for reduced global memory traffic. Since Variable 7-pt uses twice the shared memory as Heat 7-pt, occupancy is cut in half, and performance suffers. Nevertheless, shared memory still improves performance of Variable 7-pt. To see why, we experimented with hand-coded CUDA implementations. We found that if we allowed only one of the two data streams in question to reside in shared memory, then overall performance increased by 45% compared to when we used no shared memory. When we put both data streams in shared memory, the overall improvement increased to 85%.

The *opt-2* flag applies loop aggregation on top of shared memory optimization. In *opt-1*, threads can share data (reads) within the xy-plane only, whereas loop aggregation assigns each thread a column of values, allowing reuse in z-dimension as well. Thus, the thread block is responsible for 3D tile of data and can share the work across xy-planes. The generated kernels perform only the necessary loads and stores to device memory for the inner points. Once an element is read, no other thread will load the value, except for the ghost cells. This optimization has a significant impact on performance. The effect is particularly pronounced for the Poisson 19-pt stencil, owing to the high degree of sharing between threads.

The *opt-3* flag implements register optimizations on top of *opt-2*. This optimization helps in two ways: (1) an instruction executes more quickly if its operands are in registers [3] and (2) registers augment shared memory with plentiful on-chip storage (64KB compared with 16KB of shared memory). We can store more information on-chip and reduce pressure on shared memory. By reducing pressure on shared memory, the device can execute more thread blocks concurrently, i.e. with higher occupancy. Indeed, the register optimization improves performance in nearly all the kernels. The one exception is the 19-point stencil. This is an artifact of the current state of our optimizer. The hand-CUDA version of the 19-point kernel uses registers more effectively. It eliminates common expressions appearing in multiple slices of the input grid. It also uses registers to store intermediate sums along edges, and reuses the computed sums in multiple slices[6]. The effect is to reduce the number of flops performed per data point. The opportunity does not arise in the 7-pt stencils owing to limited sharing: only one value is used from the top and bottom slices. We are currently working on this optimization in Mint and expect to increase the performance of the 19-pt stencil further and hence close the performance gap with hand-coded CUDA.

5.3 Mint vs Hand-CUDA

To better understand the source of performance gap between the Mint-generated and hand-optimized CUDA code, we analyze the Heat 7-pt kernel in greater depth. Fig. 9 compares the performance of Mint-generated code for the available optimizations and compares with hand-coded CUDA (*Hand-CUDA*) that implements the same strategies. While the two variants implement the same optimization *strategies*, they may *implement* the strategies differently.

One way in which the implementations differ is in how they treat padding, which helps ensure that shared memory accesses coalesce. Mint relies on *cudaMalloc3D* to pad the storage allocation. This function aligns memory to the start of the mesh array, which includes the ghost cells. On the

[6]More information about this optimization on traditional multi-core architecture can be found in [17].

other hand, the *Hand-CUDA* implementation pre-processes the input arrays and pads them to ensure that all the global memory accesses are perfectly aligned. Memory is aligned to the inner region of the input arrays, where the solution is updated. Ghost cells are far less numerous so it pays to align to the inner region, which accounts for the lion's share of the global memory accesses.

We can achieve this effect in the Mint implementation if we pad the arrays manually prior to translation. In so doing, we observed a 10% performance improvement, on average, in the Mint-generated code at all optimization levels. Mint *opt-3* closes the gap from 86% to 90% of Hand-CUDA *opt-3* with padding.

Mint generates code that uses more registers than the hand-optimized code. This mainly stems from the fact that it maintains separate index variables to address different arrays even when the subscript expressions are shared among references to the different arrays. For example, in Listing 3, the compiler uses separate *width, surface,* and *index* variables for *U* and *Unew*. By comparison, Hand-CUDA shares the common index expressions. Combined with manual padding, reduction in index variables improved the performance by 10% and provided us with the same performance for Mint *opt-3* and the Hand-CUDA *opt-3*.

There is also an additional hand coded variant in Fig. 9, called *all-opt*, that do not appear in the compiler. This variant supports an optimization we haven't yet included in Mint, and we built the optimization on top of the *opt-3 HandCuda* variant. Currently, Mint supports `chunksize` for the z-dimension only. The *all-opt* variant implements chunking in y-dimension as well, providing a 12% improvement over Hand-CUDA *opt-3*.

6. RELATED WORK

By and large, source-to-source translation for GPU programming has taken two approaches according to the form of the user input. In the first approach the user input is a domain specific language. Liu et al. [18] implemented a source-to-source translator to automatically optimize a kernel against program inputs, and uses statistical learning techniques to search the optimization space. Lionetti et al. [19] implemented a domain-specific translator for a cardiac simulation framework, which solved a reaction diffusion system. Included was an off-line software managed cache. The translator encapsulates expert knowledge about CUDA optimization, enabling the non-expert to remain aloof of the hardware. Another domain-specific approach is taken by Kamil et al. [20]. The authors developed a translator that takes ordinary Fortran 95 as input, and can produce CUDA source code. The code generator utilizes device memory only and does not take advantage of shared memory. As previously mentioned, shared memory plays an important role in the performance of stencil methods. Indeed, our 3D Heat 7-pt kernel performs at nearly twice the performance presented by the authors.

The second approach takes input in the form of a traditional programming language, e.g., annotated with pragmas such as OpenMP. Eichenberger et al. [21] describe a source-to-source translator for the Cell Broadband Engine [16] that takes OpenMP source code as input and distributes computation across the 9 cores of the processor. The translator implements function partitioning and an on-line software managed cache.

The PGI Accelerator model [22] and OpenMPC [23, 24] take a directive-based approach, and are closest to our work. The PGI Accelerator model is a commercial compiler and is intended to be general purpose. OpenMPC supports an extended OpenMP syntax for GPUs. The compiler generates many optimization variants, and the user guides optimization through a performing tuning system.

We compared the performance of code generated by OpenMPC and the PGI compiler with code generated by Mint. For the Heat 7-pt kernel, Mint realized 22.2 Gflops while performance dropped to 1.06 Gflops for OpenMPC. The PGI compiler delivered about half the performance of Mint: 9.0 Gflops.

OpenMPC has fundamental limitations. Notably, OpenMPC only parallelizes the outermost loop of a loop nest whereas Mint parallelizes an entire loop nest. As a result, Mint can generate multi-dimensional CUDA thread blocks. The results show the significant benefit of parallelizing all levels of a loop nest in three dimensions. Moreover, because shared memory plays an important role in performance, Mint heavily invests in shared memory optimizations. By comparison, OpenMPC uses shared memory for scalar variables only and cannot buffer arrays in shared memory.

The PGI compiler uses shared memory and multi dimensional thread blocks but not as effectively as Mint. Mint uses registers in lieu of shared memory, reducing pressure on shared memory and thereby increasing device occupancy. In addition, Mint implements loop aggregation through the `chunksize` clause, improving reuse. Mint uses domain-specific knowledge to realize this optimization and the benefit of the approach is to greatly reduce the optimization search space. The payoff is improved performance for the selected application domain.

7. CONCLUSIONS

We have introduced the Mint programming model for accelerating stencil computations on the NVIDIA GPU. The user needs only annotate a traditional C source with a few intuitive Mint directives. The accompanying source-to-source translator of Mint generates highly optimized CUDA C that is competitive with heroic hand coding. The benefit of our approach is to simplify the view of the hardware while incurring a reasonable abstraction overhead.

On-chip memory and thread aggregation optimizations are crucial to delivering high performance. For a set of widely used stencil kernels, Mint realized 78% to 83% of the performance obtained by aggressively hand-optimized CUDA. Most of the kernels were 3-dimensional, where the payoff for successful optimization is high, but so are the difficulties in optimizing CUDA code by hand. We are currently applying Mint to more complex stencil applications involving many input grids and several parallel regions. As a result, our extensions to the translator in the future will focus on more inter-kernel optimizations and data movement. Moreover, we are re-targeting to the Fermi architecture, and shall report on this work in the future. We expect that many of the same optimization strategies will apply to Fermi. More up-to-date information about the Mint source to source translator can be found on our project website: `https://sites.google.com/site/mintmodel`.

Acknowledgments

The authors would like to thank Vasily Volkov for stimulating conversations about the 7-point stencil kernel, Everett Phillips from NVIDIA for his insightful comments on CUDA-related issues, and Seyong Lee for the private communication over the OpenMPC framework. Didem Unat was supported by a Center of Excellence grant from the Norwegian Research Council to the Center for Biomedical Computing at the Simula Research Laboratory. Scott Baden dedicates his portion of this research to the memory of *Shirley D. Wallach*. He was supported by the Simula Research Laboratory and the University of California, San Diego. The work of Xing Cai was partly supported by the Research Council of Norway through Grant 200879/V11. Computations on the NVIDIA Tesla system located at UCSD were supported by NSF DMS/MRI Award 0821816.

8. REFERENCES

[1] "http://www.top500.org/."

[2] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," in *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pp. 1–14, ACM, 2008.

[3] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pp. 31:1–31:11, IEEE Press, 2008.

[4] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pp. 18:1–18:11, ACM, 2009.

[5] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 79–84, ACM, 2009.

[6] 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 *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pp. 4:1–4:12, IEEE Press, 2008.

[7] K. Moreland and E. Angel, "The FFT on a GPU," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '03, pp. 112–119, Eurographics Association, 2003.

[8] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[9] J. C. Strikwerda, *Finite Difference Schemes and Partial Differential Equations, 2nd Edition*. SIAM, 2004.

[10] R. C. Gonzalez and R. E. Woods, *Digital Image Processing, 3rd Edition*. Prentice Hall, 2008.

[11] D. J. Quinlan, B. Miller, B. Philip, and M. Schordan, "Treating a user-defined parallel library as a domain-specific language," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pp. 324–, IEEE Computer Society, 2002.

[12] "Rose." http://www.rosecompiler.org.

[13] NVIDIA, *CUDA programming guide 3.2*. 2010.

[14] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3D scientific computations," in *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, SC '00, IEEE Computer Society, 2000.

[15] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "Scientific computing kernels on the Cell processor," *Int. J. Parallel Program.*, vol. 35, pp. 263–298, June 2007.

[16] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the Cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, pp. 589–604, July 2005.

[17] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Auto-tuning the 27-point stencil for multicore," in *iWAPT, 4th International Workshop on Automatic Performance Tuning*, 2009.

[18] Y. Liu, E. Z. Zhang, and X. Shen, "A cross-input adaptive framework for GPU program optimizations," in *Int. Parallel and Distributed Processing Symp.*, pp. 1–10, 2009.

[19] F. V. Lionetti, A. D. McCulloch, and S. B. Baden, "Source-to-source optimization of CUDA C for GPU accelerated cardiac cell modeling," in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pp. 38–49, Springer-Verlag, 2010.

[20] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *Interational Conference on Parallel and Distributed Computing Systems (IPDPS)*, 2010.

[21] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo, "Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture," *IBM Syst. J.*, vol. 45, pp. 59–84, January 2006.

[22] M. Wolfe, "Implementing the PGI Accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pp. 43–50, 2010.

[23] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," *SIGPLAN Not.*, vol. 44, pp. 101–110, February 2009.

[24] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP Programming and Tuning for GPUs," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–11, IEEE Computer Society, 2010.