CUDA-free programming of 3D stencil methods with Mint

Didem Unat¹, Xing Cai², Scott B. Baden¹

¹Computer Science & Eng., Univ of California, San Diego
²Simula Research Lab, Norway

ICCS Workshop 2011
Programming Accelerators

- GPUs are effective means of accelerating data parallel applications
- Unique memory hierarchy
- Low level programming to tune for best performance

We need software tools to facilitate CUDA programming.
Mint Programming Model

- Enables the non-expert to enjoy the performance benefits of hand-coded CUDA without becoming entangled in the details
- Based on programmer annotations (pragmas)
- Source-to-source translator
  - Translates from annotated C source to optimized CUDA C
- Targets 3D stencil methods
  - An important class of scientific applications
  - Optimizations benefit from the domain-specific knowledge
- Mint realized 76% of the performance obtained from aggressively optimized CUDA on the 400-series of GPUs
Outline

- Motivation and Background
  - Stencil Computation

- Mint Programming Model
  - Mint Pragmas

- Mint-to CUDA Translator
  - Translation Engine
  - Mint Optimizer

- Performance Results

- Related Work
**Stencil Computations**

- Arise in some important classes of applications
  - Finite difference discretization of PDEs and image processing
  - Highly data parallel and typically implemented as nested for-loops
  - Updates each point of the mesh with weighted contributions from its neighbors in both in time and space
  - Ex: 5-pt stencil approximation of 2D Laplacian operator and the 7-pt stencil in 3D

---

5-pt stencil

7-pt stencil
The Mint-annotated version

- Achieves 78% of the performance by the heroically optimized CUDA
- Requires a modest programming effort

CUDA results are obtained on Tesla C1060.

OpenMP ran on Intel Nehalem with 4 threads.

Vasily Volkov provided the optimized CUDA implementation.
Mint Program for the 3D Heat Eqn

```mint
#pragma mint copy(dU,U,toDevice,(n+2),(m+2),(k+2))
#pragma mint copy(dUnew,Unew,toDevice,(n+2),(m+2),(k+2))

#pragma mint parallel default(shared)
{
    int t=0;
    while( t++ < T ){

#pragma mint for nest(all) tile(16,16,1)
        for (int z=1; z<= k; z++)
            for (int y=1; y<= m; y++)
                for (int x=1; x<= n; x++)
                    Unew[z][y][x] = c0 * U[z][y][x] +
                                    c1 * (U[z][y][x-1] + U[z][y][x+1] +
                                          U[z][y-1][x] + U[z][y+1][x] +
                                          U[z-1][y][x] + U[z+1][y][x]);

#pragma mint single{
    double*** tmp;
    tmp = U; U = Unew; Unew = tmp;
} //end of single

} //end of while
} //end of parallel region

#pragma mint copy(U,dU,fromDevice,(n+2),(m+2),(k+2))
```
Mint Program for the 3D Heat Eqn

```mint
#pragma mint copy(dU,U,toDevice,(n+2),(m+2),(k+2))
#pragma mint copy(dUnew,Unew,toDevice,(n+2),(m+2),(k+2))

#pragma mint parallel default(shared)
{
  int t=0;
  while( t++ < T ){

#pragma mint for nest(all) tile(16,16,1)
    for (int z=1; z<= k; z++)
      for (int y=1; y<= m; y++)
        for (int x=1; x<= n; x++)
          Unew[z][y][x] = c0 * U[z][y][x] +
                          c1 * (U[z][y][x-1] + U[z][y][x+1] +
                          U[z][y-1][x] + U[z][y+1][x] +
                          U[z-1][y][x] + U[z+1][y][x]);

#pragma mint single{
  double*** tmp;
  tmp = U; U = Unew; Unew = tmp;
} //end of single

} //end of while
} //end of parallel region

#pragma mint copy(U,dU,fromDevice,(n+2),(m+2),(k+2))
```
#pragma mint copy(dU,U,toDevice,(n+2),(m+2),(k+2))
#pragma mint copy(dUnew,Unew,toDevice,(n+2),(m+2),(k+2))

#pragma mint parallel default(shared)
{
    int t=0;
    while( t++ < T ){

#pragma mint for nest(all) tile(16,16,1)
    for (int z=1; z<= k; z++)
        for (int y=1; y<= m; y++)
            for (int x=1; x<= n; x++)
                Unew[z][y][x] = c0 * U[z][y][x] +
                    c1 * (U[z][y][x-1] + U[z][y][x+1] +
                    U[z][y-1][x] + U[z][y+1][x] +
                    U[z-1][y][x] + U[z+1][y][x]);

#pragma mint single(
    double*** tmp;
    tmp = U; U = Unew; Unew = tmp;
} // end of single

} // end of while
} // end of parallel region

#pragma mint copy(U,dU,fromDevice,(n+2),(m+2),(k+2))
Mint Program for the 3D Heat Eqn

```
#pragma mint copy(dU,U,toDevice,(n+2),(m+2),(k+2))
#pragma mint copy(dUnew,Unew,toDevice,(n+2),(m+2),(k+2))

#pragma mint parallel default(shared)
{
    int t=0;
    while ( t ++ < T ){

#pragma mint for nest(all) tile(16,16,1)
        for (int z=1; z< k; z++)
            for (int y=1; y< m; y++)
                for (int x=1; x< n; x++)
                    Unew[z][y][x] = c0 * U[z][y][x] +
                                   c1 * (U[z][y][x-1] + U[z][y][x+1]) +
                                   U[z][y-1][x] + U[z][y+1][x] +
                                   U[z-1][y][x] + U[z+1][y][x]);

#pragma mint single{
    double*** tmp;
    tmp = U; U = Unew; Unew = tmp;
} // end of single

} // end of while
} // end of parallel region

#pragma mint copy(U,dU,fromDevice,(n+2),(m+2),(k+2))
```
#pragma mint directive-name [clauses, …]

- **mint parallel region**
  - Indicates the region to be accelerated

- **mint for**
  - Transformed into a multi-dim CUDA kernel

- **mint barrier, single**
  - Needed for synchronization and serial regions

- **mint copy**
  - Data transfers between the host and device
  - #pragma mint copy(dst, src, direction, [N_x, N_y, N_z])
    - Direction: (toDevice | fromDevice)
#pragma mint for [clauses]

- Marks the succeeding nested loops for GPU acceleration
  - Manages data decomposition and thread work-assignment.

- `nest(#,all)`
  - Indicates the depth of for-loop parallelization.
  - Unlike OpenMP, supports multi-dim thread geometries.

- `tile(t_x,t_y,t_z)`
  - Divides the iteration space into tiles
  - Data points computed by a CUDA thread block

- `chunksize(c_x,c_y,c_z)`
  - Determines the workload of a thread
  - Similar to OpenMP schedule clause
CUDA thread blocks

- A 3D grid is broken into 3D tiles base on the tile clause.
- Elements in a tile are divided among a CUDA thread block based on chunksize clause.
- An example of data decomposition.
C to CUDA Translation
We have developed fully automated translation and optimization system.

Translator is built on top of the ROSE compiler framework
- ROSE provides an API for generating and manipulating Abstract Syntax Trees.

We perform transformations on the Abstract Syntax Tree.

The Mint translator generates both host and device code.
Input code: C + Mint

Mint Pragma Handler

Program Transformer

Kernel Configuration

Outliner

Variable Handler

Memory Manager

ROSE Parser

Mint

ROSE Backend

Output file Cuda src

Thread Mapping

Stencil Analysis

Domain-Specific Optimizations
```c
#pragma mint copy(dU,U,toDevice,(n+2),(m+2),(k+2))
#pragma mint copy(dUnew,Unew,toDevice,(n+2),(m+2),(k+2))

#pragma mint parallel default(shared)
{
    int t=0;
    while( t++ < T ){
        mint_1_1517<<<param>>>(...);
        ...
    }

#pragma mint for nest(all) tile(16,16,1)
    for (int z=1; z<= k; z++)
        for (int y=1; y<= m; y++)
            for (int x=1; x<= n; x++)
                Unew[z][y][x] = c0 * U[z][y][x] +
                    c1 * (U[z][y][x-1] + U[z][y][x+1]) +
                    U[z][y-1][x] + U[z][y+1][x] +
                    U[z-1][y][x] + U[z+1][y][x];

#pragma mint single{
    double*** tmp;
    tmp = U; U = Unew; Unew = tmp;
} //end of single

} //end of while
} //end of parallel region

#pragma mint copy(U,dU,fromDevice,(n-2),(m-2),(k-2))

/** Outlined Kernel **/
__global__ void mint_1_1517(...)
{
    ...
}
```
```c
__global__ void mint_1_1517(
    cudaPitchedPtr ptr_dU ...
) {
    double* U = (double*)(ptr_dU.ptr);
    int widthU = ptr_dU.pitch / sizeof(double);
    int sliceU = ptr_dU.ysize * widthU;
    ...

    int _idx = threadIdx.x + 1;
    int _gidx = _idx + blockDim.x * blockIdx.x;
    ...

    if (_gidz >= 1 && _gidz <= k)
        if (_gidy >= 1 && _gidy <= m)
            if (_gidx >= 1 && _gidx <= n)
                Unew[indUnew] = c0 * U[indU]
                + c1 * (U[indU - 1] + U[indU + 1]
                            ... );
}
//end of kernel
```

**Unpack CUDA pitched ptrs**

**Compute local and global indices using thread and block IDs**

**If-statements are derived from for-statements**
__global__ void mint_1_1517(
    cudaPitchedPtr ptr_dU ...
) {

double* U = (double *) (ptr_dU.ptr);
int widthU = ptr_dU.pitch / sizeof(double);
int sliceU = ptr_dU.ysize * widthU;
...

int _idx = threadIdx.x + 1;
int _gidx = _idx + blockDim.x * blockIdx.x;
...

if (_gidz >= 1 && _gidz <= k)
    if (_gidy >= 1 && _gidy <= m)
        if (_gidx >= 1 && _gidx <= n)
            Unew[indUnew] = c0 * U[indU] + c1 * (U[indU - 1] + U[indU + 1] ... );

}//end of kernel

Each CUDA thread computes a single data point.

Unpack CUDA arguments

Compute local and global indices using thread and block IDs

If-statements are derived from for-statements
Loop Aggregation

- The programmer can manage the mapping of work to threads using the **chunksize** clause.

```c
if (_gidy >= 1 && _gidy <= m)
  if (_gidx >= 1 && _gidx <= n)
    for (_gidz=lb ; _gidz < ub ; _gidz++)
      Unew[indUnew] = c0 * U[indU] . . . ;
```

- “Fat” threads
- 3D partial blocking optimization with on-chip memory optimizations
The translated (un-optimized) code performs all the memory references through global memory
  - Can still benefit from multi-dim kernels

Focus on on-chip memory optimizations not device occupancy
Is the kernel eligible for optimization?
- If the pattern of array subscripts involves a central point and nearest neighbors only
- How much shared memory is needed?
- Which ghost cells to load?
On-chip Memory Optimizer

- **Shared memory optimizer**
  - Find candidate array(s) for shared memory
    - Most frequently referenced array(s)
  - Perform the ghost cell loads

- **Register optimization**
  - Alleviates pressure on shared memory
  - Increases device occupancy and instruction throughput

- More details about optimizing stencil computation:
  - “Tiling optimizations for 3D scientific computations” by Rivera and Tseng
  - “Stencil computation optimization and auto-tuning on state-of-the-art multi-core architectures” by Datta et al.
- Improve re-use with `chunksize` clause together with shared memory and registers
- Assign each CUDA thread more than one point in the iteration space of the loop-nest
- Possible also in Y-dim and X-dim

Ex: register
    reg + shared mem
    register
We present results for Mint, hand-written CUDA and OpenMP

- Mint was not used to generate the hand-written CUDA versions

<table>
<thead>
<tr>
<th>3D Kernels</th>
<th>In, Out Arrays</th>
<th>Reads, Writes/pt</th>
<th>Operations/pt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat 7-pt</td>
<td>1,1</td>
<td>7,1</td>
<td>2(*),6(+)</td>
</tr>
<tr>
<td>Poisson 7-pt</td>
<td>2,1</td>
<td>7,1</td>
<td>2(*),6(+)</td>
</tr>
<tr>
<td>Variable 7-pt</td>
<td>3,1</td>
<td>15,1</td>
<td>7(*),13(+),6(-)</td>
</tr>
<tr>
<td>Poisson 19-pt</td>
<td>2,1</td>
<td>19,1</td>
<td>2(*),18(+)</td>
</tr>
</tbody>
</table>

- OpenMP results are obtained on Intel Xeon E5504, quad-core, 16 GB memory, gcc 4.4.3, -03 –fopenmp using 4 threads.
- Mint and hand-CUDA results are obtained on
  - Tesla C1060, 4GB memory, nvcc 3.2 and Tesla C2050, 3GB memory, nvcc 3.2
Performance Comparison

- On Tesla C1060, Mint achieves 79% of the hand-optimized CUDA
- The unoptimized Mint code outperforms the OpenMP code with an average 3x speedup.
The Mint optimizer improves the performance still further and provides a 2.6x speedup over the unoptimized translation.
Mint achieves on average 79% of the hand-optimized CUDA on the Tesla C1060 (200-series GPU).
200- vs 400-series of Tesla

![Bar chart comparing Gflops for different methods and series on Tesla C1060 and Tesla C2050.](chart.png)

- **Methods**: Mint naive, Mint best, hand-CUDA
- **Series**: Heat 7pt, Poisson 7pt, Variable 7pt, Poisson 19pt
- **Platform**: Tesla C1060, Tesla C2050
Fermi provides a 2.5x speedup for the unoptimized Mint-generated kernels.
Optimizations provide 1.3x speedup over the unoptimized kernels.
Mint achieves on average 76% on the 400-series and 79% on the 200-series GPUs of the hand-CUDA.
Related Work

- PGI and OpenMPC
  - General purpose compilers
  - Optimizations are limited and don’t perform stencil-specific optimizations

- Domain-specific approaches
  - Libraries: Cublas, FFT
  - OpenCurrent from Nvidia

- Auto-tuning framework for stencil computations by Kamil et al.
  - Starts with a description of the stencil in a domain-specific language
Our Contribution

- A CUDA-free programming model based on just 5 pragmas.
- Manages locality and parallelizes loop-nests
  - Enables multi-dimensional CUDA kernels.
- Source-to-source translator
  - Incorporates domain specific knowledge to generate highly efficient CUDA C code.
- Mint achieves on average
  - 76% on the Tesla C2050
  - 79% on the Tesla C1060 of the hand-optimized CUDA.
- First release is coming soon!
  - http://ege.ucsd.edu

- Have a stencil application?
Acknowledgments

[ simula . research laboratory ]