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

Didem Unat, Xing Cai, Scott Baden
Accelerator-based Computing

- GPUs are effective means of accelerating data parallel applications
- However, porting algorithms on a GPU still remains a challenge.
- A directive-based programming model is a promising approach.
Mint Programming Model

- Simple, compact and comes with a source-to-source translator
- Based on programmer annotations
  - e.g. `#pragma mint for`
- Source-to-source translator
  - Translates from annotated C source to optimized CUDA C
  - Benefits from the domain-specific knowledge
  - Allows different levels of optimizations for stencil methods
- Targets non-expert CUDA programmers who
  - Do not want to invest in significant programming effort
  - Seek reasonable performance (70-80% of hand-coded)
Outline

- Motivation and Background
  - Stencil Computation

- Mint Programming Model
  - Mint Pragmas

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

- Performance Results

- Related Work
Stencil Computation

- 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
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.     `#pragma mint for` nest(all) tile(16,16,1)
9.     for (int z=1; z<= k; z++)
10.    for (int y=1; y<= m; y++)
11.       for (int x=1; x<= n; x++)
12.         Unew[z][y][x] = c0 * U[z][y][x] +
13.            c1 * (U[z][y][x-1] + U[z][y][x+1] +
14.                U[z][y-1][x] + U[z][y+1][x] +
15.                U[z-1][y][x] + U[z+1][y][x]);
16.     `#pragma mint` single{
17.       double*** tmp;
18.       tmp = U; U = Unew; Unew = tmp;
19.     }//end of single
20.   }
21. }//end of while
22.}//end of parallel region
23. 
24.`#pragma mint copy`(U,fromDevice,(n+2),(m+2),(k+2))
Mint Program for the 3D Heat Eqn

1. #pragma mint copy(U, toDevice, (n+2), (m+2), (k+2))
2. #pragma mint copy(Unew, toDevice, (n+2), (m+2), (k+2))

4. #pragma mint parallel default(shared)

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

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. `#pragma mint parallel default(shared)`
4. ```
   { 
   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
```
5. `#pragma mint copy(U, fromDevice, (n+2), (m+2), (k+2))`

---

**Data Transfers**
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;

8. #pragma mint for nest(all) tile(16,16,1)
9.   for (int z=1; z<= k; z++)
10.      for (int y=1; y<= m; y++)
11.         for (int x=1; x<= n; x++)
12.             Unew[z][y][x] = c0 * U[z][y][x] +
13.                c1 * (U[z][y][x-1] + U[z][y][x+1] +
14.                   U[z][y-1][x] + U[z][y+1][x] +
15.                      U[z-1][y][x] + U[z+1][y][x]);
16. #pragma mint single{
17.   double*** tmp;
18.   tmp = U; U = Unew; Unew = tmp;
19. }//end of single
20. 
21. }//end of while
22.}//end of parallel region
23. 
24. #pragma mint copy(U, dU, fromDevice, (n+2), (m+2), (k+2))
Mint Pragmas

#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 based on the tile clause.
- Elements in a tile are divided among a CUDA thread block based on chunksize clause.
- An example of data decomposition.
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.
Translation Flow

Baseline Translator

Input code: C + Mint

Mint Pragma Handler

Kernel Configuration

Outliner

Thread Scheduler

Argument Handler

Mint Optimizer

ROSE Parser

Memory Manager

Mint

ROSE backend

Stencil Analyzer

Output file

Domain-specific Optimizations

Cuda src

Mint Optimizer

Cuda src
```c
#pragma mint copy(dU,U,toDevice)
#pragma mint copy(dUnew,Unew)
#pragma mint parallel default
{
    int t=0;
    while(t++ < T){
        mint_1_1517<<<grid, thread>>>(. . .);
    }
}
#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, . . . . . . . . . . . .)
/* Outlined Kernel */
__global__ void mint_1_1517( . . . )
{
    . . .
}
```
__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
__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)

}//end of kernel

Each CUDA thread computes a single data point.

Unpack CUDA

Compute local and global indices using thread and block IDs

If-statements are derived from for-statements
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
Mint Optimizer

- The translated (un-optimized) code performs all the memory references through global memory
  - Can still benefit from multi-dim kernels
- Optimizer focuses on reduction in global memory references
Analyzer determines
- the pattern of array accesses involving a central point and nearest neighbors
- 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
Select candidates

- Compute access frequencies to the center and top-bottom planes separately
  - Either have 1 plane or 3 planes per array
  - Sort the candidates in terms of max reduction in the global memory references
  - To be eligible, there should be at least one off-mid point ref.

Ex.

- Top saves 3 refs
- Center saves 5 refs
- Bottom saves 3 refs

Shared memory

1 2 3
4 5 6
7 8 9
Loop Aggregation Optimization

- 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
Performance Results

- We present results for Mint, hand-written CUDA and OpenMP
  - Mint was not used to generate the hand-written CUDA versions
  - OpenMP results are obtained on Intel Xeon E5504, quad-core, 16 GB memory, gcc 4.4.3, -o3 –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

<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>
On Tesla C1060, Mint achieves 79% of the hand-optimized CUDA

Mint is competitive with hand-coding
Mint achieves on average 76% on the 400-series and 79% on the 200-series GPUs of the hand-CUDA.
### Performance Comparison

<table>
<thead>
<tr>
<th>Gflops</th>
<th>Mint</th>
<th>OpenMPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-pt Heat (2D)</td>
<td>14.4</td>
<td>7.3</td>
</tr>
<tr>
<td>7pt Heat Eqn.</td>
<td>22.2</td>
<td>1.06</td>
</tr>
</tbody>
</table>


Mint and OpenMPC: on the Tesla C1060 @ UCSD
### Performance Comparison

<table>
<thead>
<tr>
<th>Gflops</th>
<th>Mint</th>
<th>OpenMPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-pt Heat (2D)</td>
<td>14.4</td>
<td>7.3</td>
</tr>
<tr>
<td>7pt Heat Eqn.</td>
<td>22.2</td>
<td>1.06</td>
</tr>
</tbody>
</table>


Mint and OpenMPC: on the Tesla C1060 @ UCSD

**Mint supports multi-dim kernels (nested loops)**
### Performance Comparison

<table>
<thead>
<tr>
<th>Gflops</th>
<th>Mint</th>
<th>PGI</th>
</tr>
</thead>
<tbody>
<tr>
<td>7pt Heat Eqn.</td>
<td>22.2</td>
<td>9.0</td>
</tr>
<tr>
<td>19-pt Heat Eqn.</td>
<td>15.8</td>
<td>11.3</td>
</tr>
</tbody>
</table>

PGI Accelerator Model from Portland Group

PGI (v11.1) results:
on the Lincoln system at NCSA using Tesla C1060.

Domain-specific knowledge Mint uses on-chip memory more effectively
Demonstrate the applicability of Mint to more complex applications

Greater challenges in translation and optimization
- Real world simulations involve a high number of input grids
- Large loop body with many statements
  - Demands more resources (e.g., high number of registers)

Consider two applications
- Harris Corner Detection Algorithm
  - Collaboration with Han Suk Kim and Jurgen Schulze
  - Submitted to IEEE Visualization 2011
- Earthquake Simulation (a really real world app.)
  - Collaboration with Yifeng Cui and Jun Zhou at SDSC
Harris Corner Detection Algorithm

- Gaussian convolution
  - Computes gradient in x, y, and z directions
  - Each neighbor has a different weight (w[5][5][5])
  - Computes cornerness metric
Feature Selection

- Part of a feature selection algorithm
- Finds the best viewing angle of an image

Image courtesy Han Suk Kim
## Results

- Real-time performance: very important in visualization
- 10x speedup over OpenMP
- Inserted 5 lines of Mint code into the original code (389 lines)

<table>
<thead>
<tr>
<th>Timings</th>
<th>Size</th>
<th>Serial</th>
<th>OpenMP-4</th>
<th>Mint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine</td>
<td>256 x 256 x 256</td>
<td>15.375</td>
<td>3.927</td>
<td>0.3896</td>
</tr>
<tr>
<td>Lobster</td>
<td>301 x 324 x 56</td>
<td>5.012</td>
<td>1.466</td>
<td>0.1323</td>
</tr>
<tr>
<td>Tooth</td>
<td>94 x 103 x 161</td>
<td>1.427</td>
<td>0.426</td>
<td>0.0439</td>
</tr>
<tr>
<td>Cross</td>
<td>66 x 66 x 66</td>
<td>0.264</td>
<td>0.072</td>
<td>0.0106</td>
</tr>
</tbody>
</table>

Results are on Tesla C1060. Convolution 5x5x5. Time in second.
Earthquake Simulation

- 3D seismic earthquake wave simulator (AWP)
- Uses finite different solvers
- Yifeng and Jun at SDSC

- MPI version was a candidate for Gordon Bell Prize at SC’10
  - a full dynamic simulation of a magnitude-8 earthquake on the southern California covering a 800x400 km area
## Initial Results

<table>
<thead>
<tr>
<th>128^3, iter = 1000</th>
<th>Quad-core Nehalem</th>
<th>Tesla C1060</th>
<th>Tesla C2050</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gflops</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Serial</td>
<td>0.90</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Mint baseline</td>
<td>-</td>
<td>12.38</td>
<td>33.31</td>
</tr>
<tr>
<td>Mint register</td>
<td>-</td>
<td>16.59</td>
<td>41.95</td>
</tr>
</tbody>
</table>

- Tuning our shared memory optimizer
- Mint generates ~500 lines of code just to perform data transfers between host and device
  - The programmer writes 30 lines similar to this one:
    ```
    #pragma mint copy( d1, toDevice, nzt+4, nyt+4, nxt +4)
    ```
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.
- Project Website:
  https://sites.google.com/site/mintmodel/
- Have a stencil application? Want to use Mint?
Acknowledgments
Current Limitations 😞

- Single GPU translation
- Branches in time iteration loop
  - We manually inline the functions into the parallel region so that all the data parallel loops are in the region
- Arrays should be allocated contiguously
- Array sizes
  - We force arrays to have the same size in each dimension
  - This allows us to use a single index variable for all arrays