CUDA-free programming of 3D stencil methods with Mint

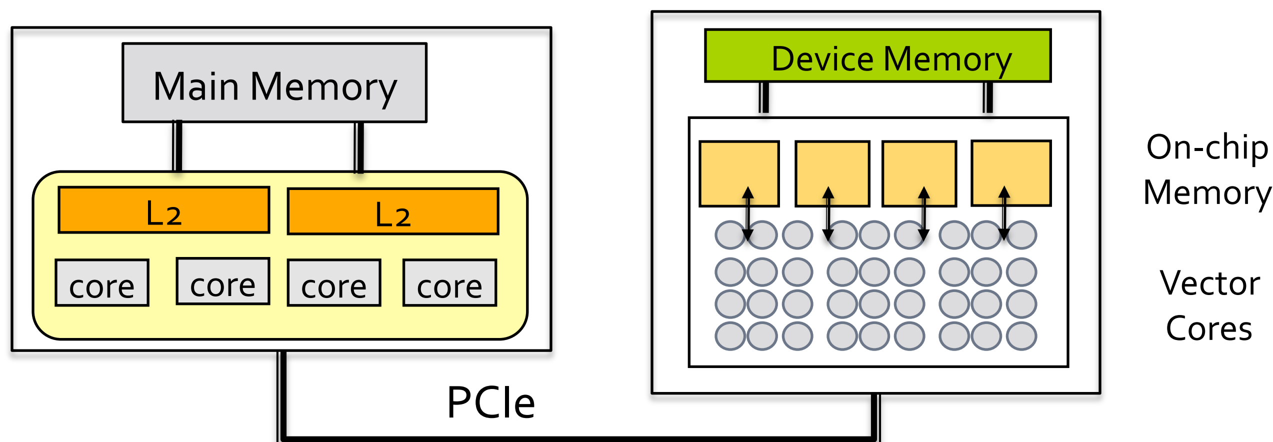
Didem Unat[1], Xing Cai[2], Scott B. Baden[1]

[1]Computer Science & Eng., Univ of California, San Diego
[2]Simula 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

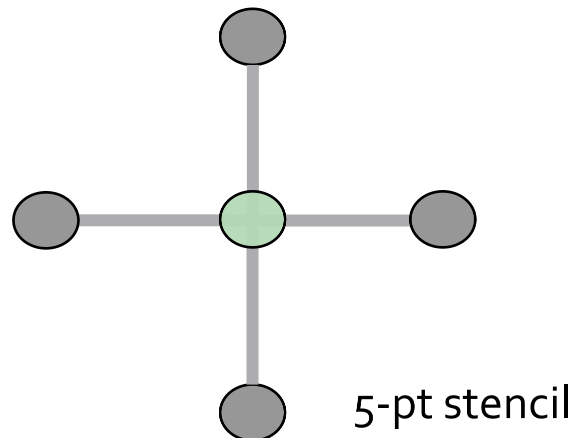
Outline

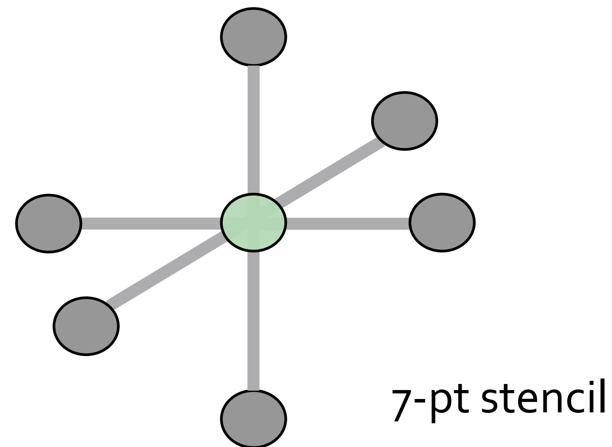
- Motivation and Background
  - Stencil Computation
- Mint Programming Model
  - Mint Pragmas
- Mint-to CUDA Translator
  - Translation Engine
  - Mint Optimizer
- Performance Results
- Related Work

Stencil Computations

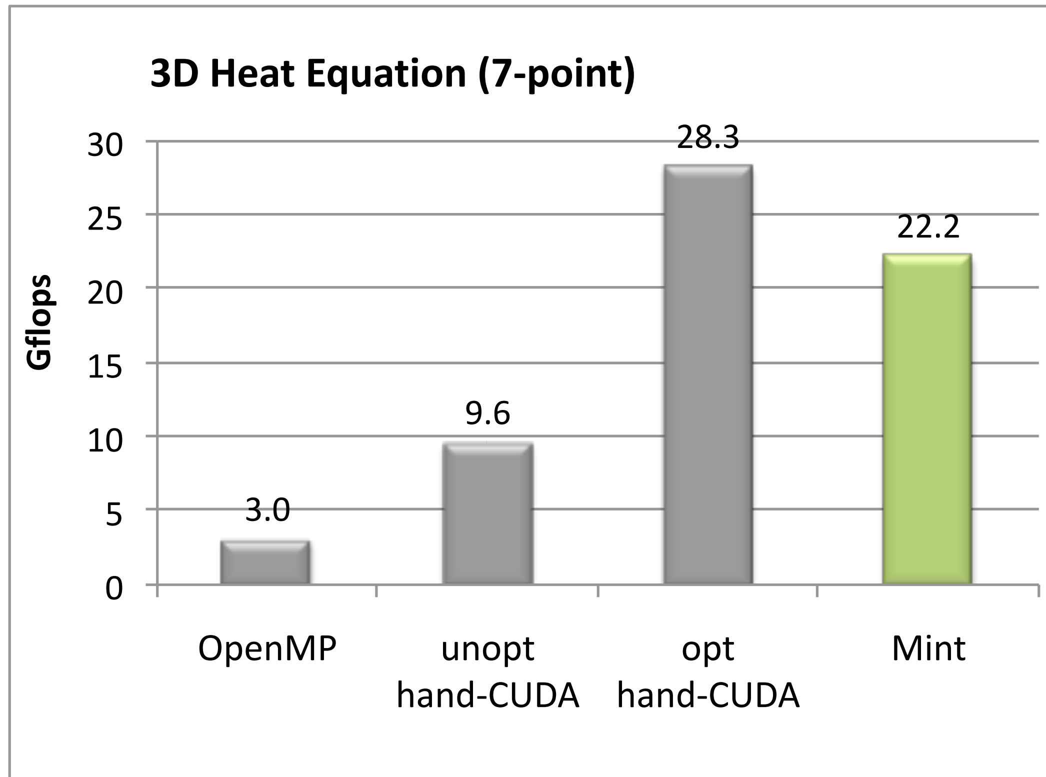
- Arise in some important classes of applications
  - Finite difference discretization of PDEs and image processing
  - Highly data parallel and typically implemented as nested for-loops
  - Updates each point of the mesh with weighted contributions

  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





A flavor of Mint



CUDA results are obtained on Tesla C1060.

OpenMP ran on Intel Nehalem with 4 threads.

Vasily Volkov provided the optimized CUDA implementation.

The Mint-annotated version

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

Mint Program for the 3D Heat Eqn

```
1  #pragma mint copy(dU,U,toDevice,(n+2),(m+2),(k+2))
2  #pragma mint copy(dUnew,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
22   }//end of while
23 }//end of parallel region
24
25 #pragma mint copy(U,dU,fromDevice,(n+2),(m+2),(k+2))
```

Mint Program for the 3D Heat Eqn

```
1  #pragma mint copy(dU,U,toDevice,(n+2),(m+2),(k+2))
2  #pragma mint copy(dUnew,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
22  }//end of while
23 }//end of parallel region
24
25 #pragma mint copy(U,dU,fromDevice,(n+2),(m+2),(k+2))
```

Accelerated Region

Mint Program for the 3D Heat Eqn

```
1  #pragma mint copy(dU,U,toDevice,(n+2),(m+2),(k+2))
2  #pragma mint copy(dUnew,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
22   }//end of while
23 }//end of parallel region
24
25 #pragma mint copy(U,dU,fromDevice,(n+2),(m+2),(k+2))
```

Data Transfers

Mint Program for the 3D Heat Eqn

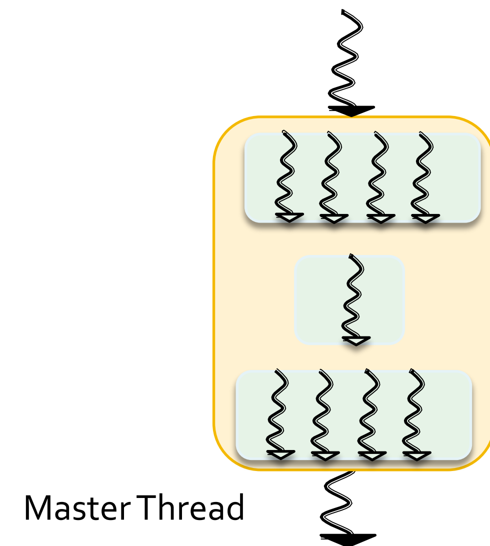
```
1  #pragma mint copy(dU,U,toDevice,(n+2),(m+2),(k+2))
2  #pragma mint copy(dUnew,Unew,toDevice,(n+2),(m+2),(k+2))
3
4  #pragma mint parallel default(shared)
5  {
6    int t=0;
7    while( t++ < T ){
8
9  #pragma mint for nest(all) tile(16,16,1)
10      for (int z=1; z<= k; z++)
11       for (int 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
22   }//end of while
23 }//end of parallel region
24
25 #pragma mint copy(U,dU,fromDevice,(n+2),(m+2),(k+2))
```

Nested-for

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*



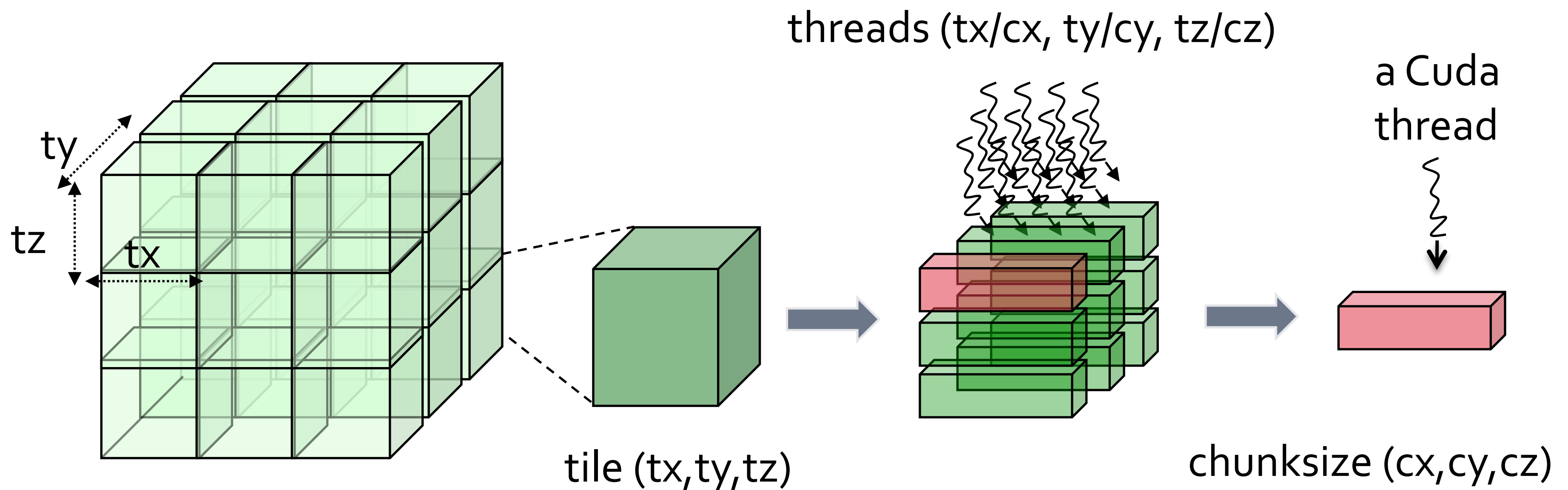
- *mint copy*
  - *Data transfers between the host and device*
  - *`#pragma mint copy`(dst, src , direction, [N_x , N_y , N_z])*
    - `Direction: (toDevice | fromDevice)`

#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

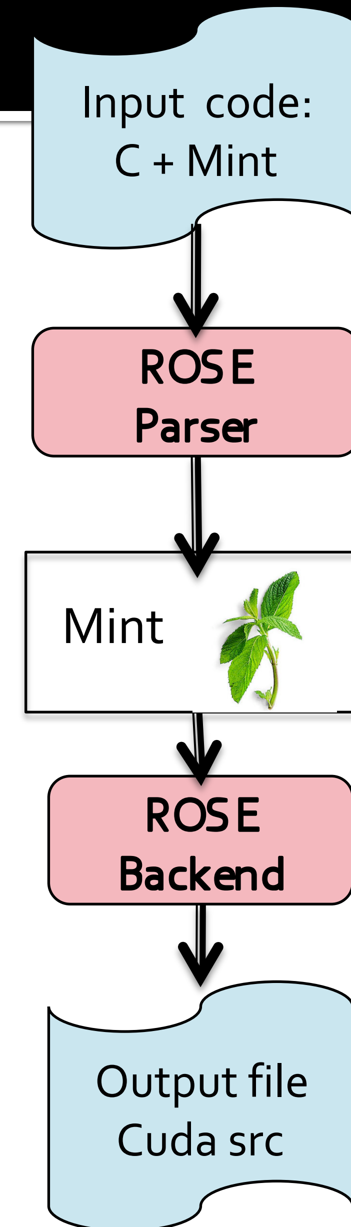
- A 3D grid is broken into 3D tiles base on the tile clause
- Elements in a tile are divided among a CUDA thread block based on chunksize clause.
- An example of data decomposition

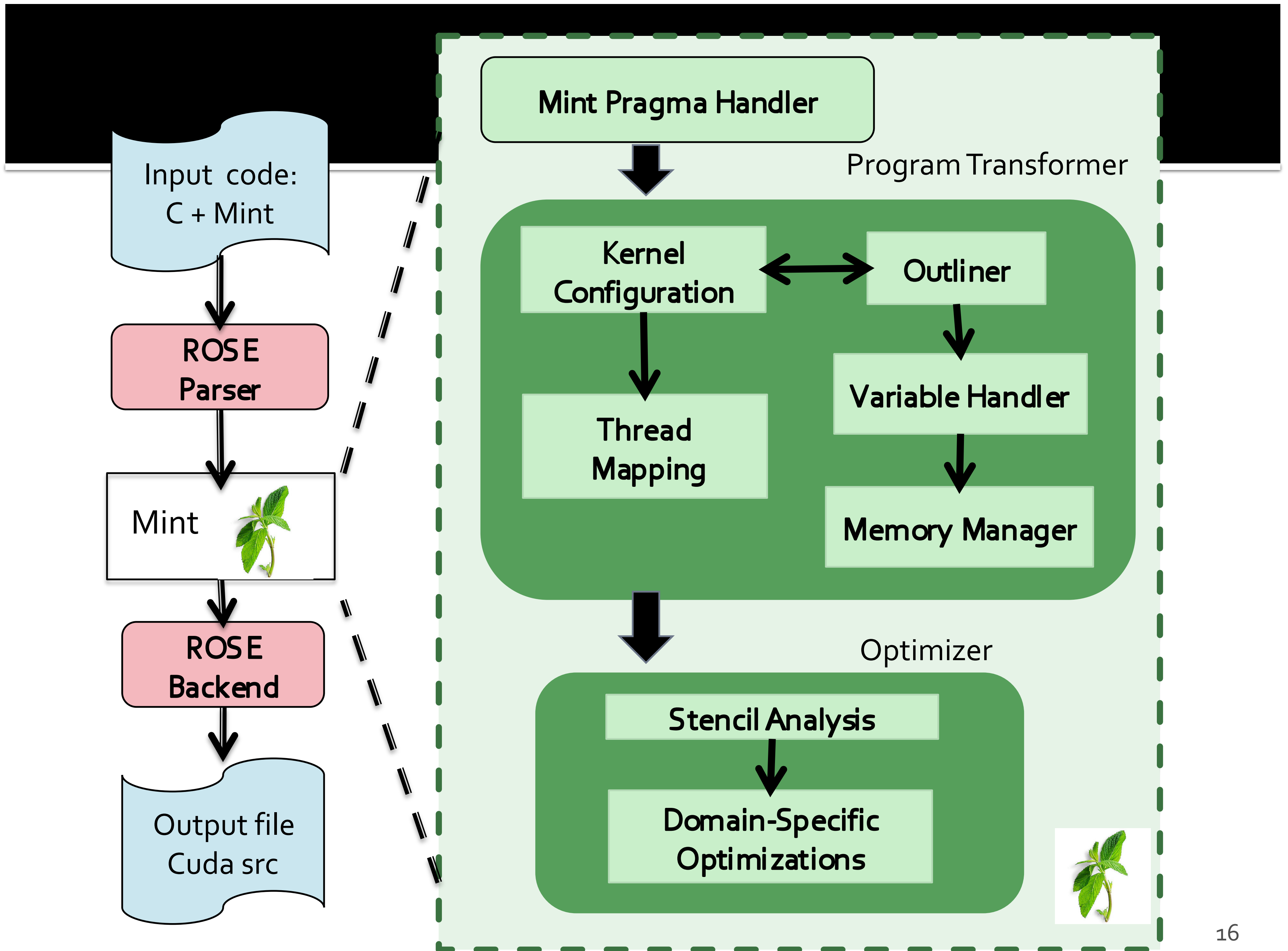


C to CUDA Translation

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
ROSE
Parser
Mint
ROSE
Backend
Output file
Cuda src
Mint Pragma Handler
Program Transformer
Kernel
Configuration
Outliner
Variable Handler
Thread
Mapping
Memory Manager
Optimizer
Stencil Analysis
Domain-Specific
Optimizations

Outlining

```
1  #pragma mint copy(dU,U,toDevice,(n+2),(m+2),(k+2))
2  #pragma mint copy(dUnew,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
22   }//end of while
23 }//end of parallel region
24
25 #pragma mint copy(U,dU,fromDevice,(n+
```

```
{
   while ( t < T){
     t += dt ;

     . . .

     mint_1_1517<<<param>>>(…);


     . . .
  }
 …
}
```

```
/* Outlined Kernel */

__global__ void mint_1_1517( . . .)
{ . . . }
```

Mint-generated Kernel Code

```
__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

Mint-generated Kernel Code

```
__global__ void mint_1_1517(
         cudaPitchedPtr ptr_dU ...)
{
  do
  in
  in
  ..

  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

Each CUDA thread computes a single data point.

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.

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

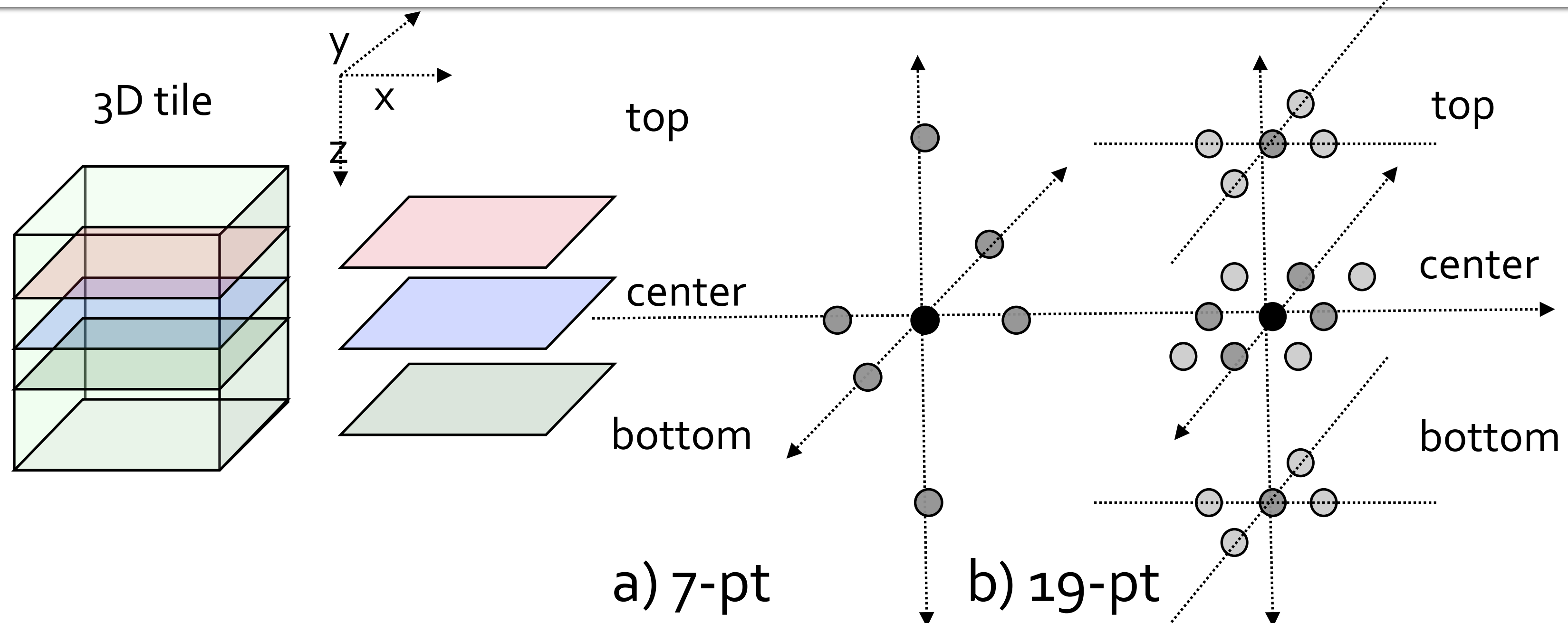
- “Fat” threads
- 3D partial blocking optimization with on-chip memory optimizations

Mint Optimizer

- The translated (un-optimized) code performs all the memory references through global memory
  - Can still benefit from multi-dim kernels

Focus on on-chip memory optimizations not device occupancy

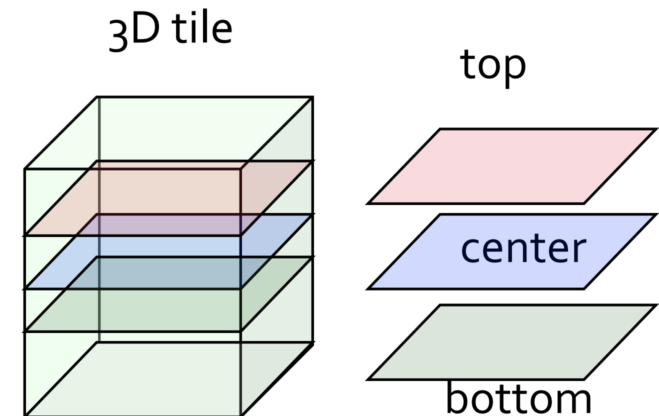
Stencil Analyzer



- 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

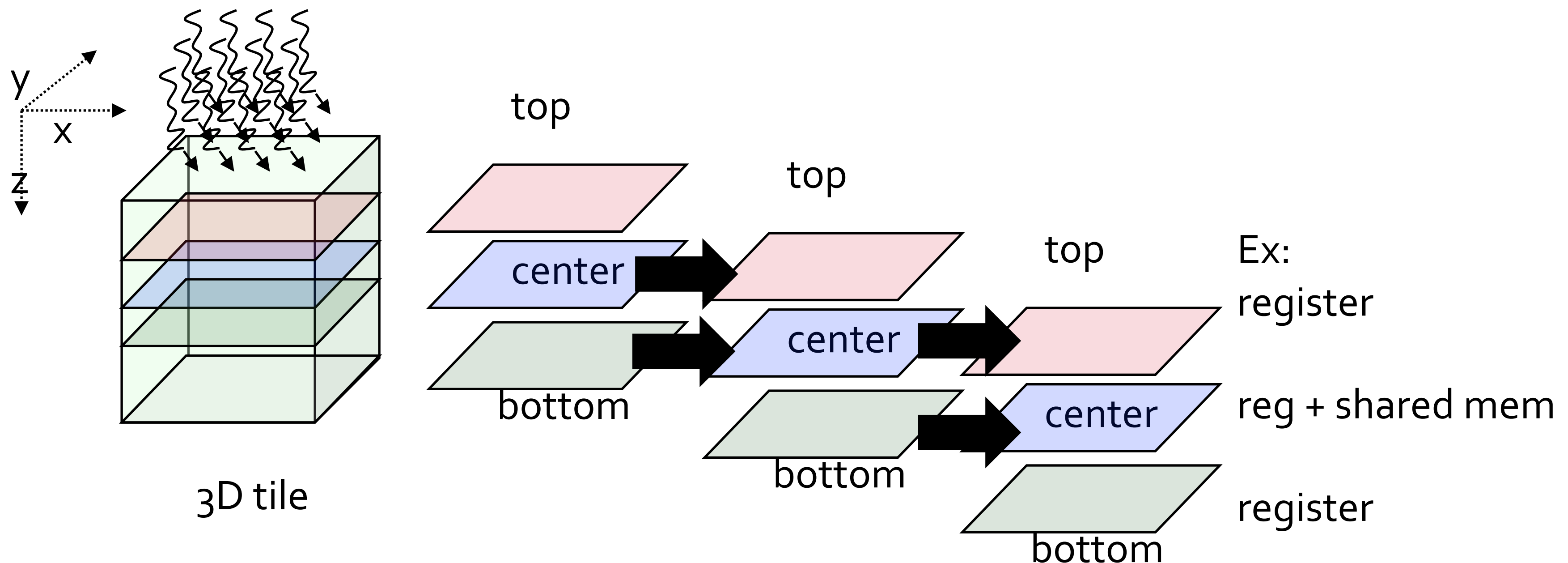


- Register optimization
  - Alleviates pressure on shared memory
  - Increases device occupancy and instruction throughput

- 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.

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



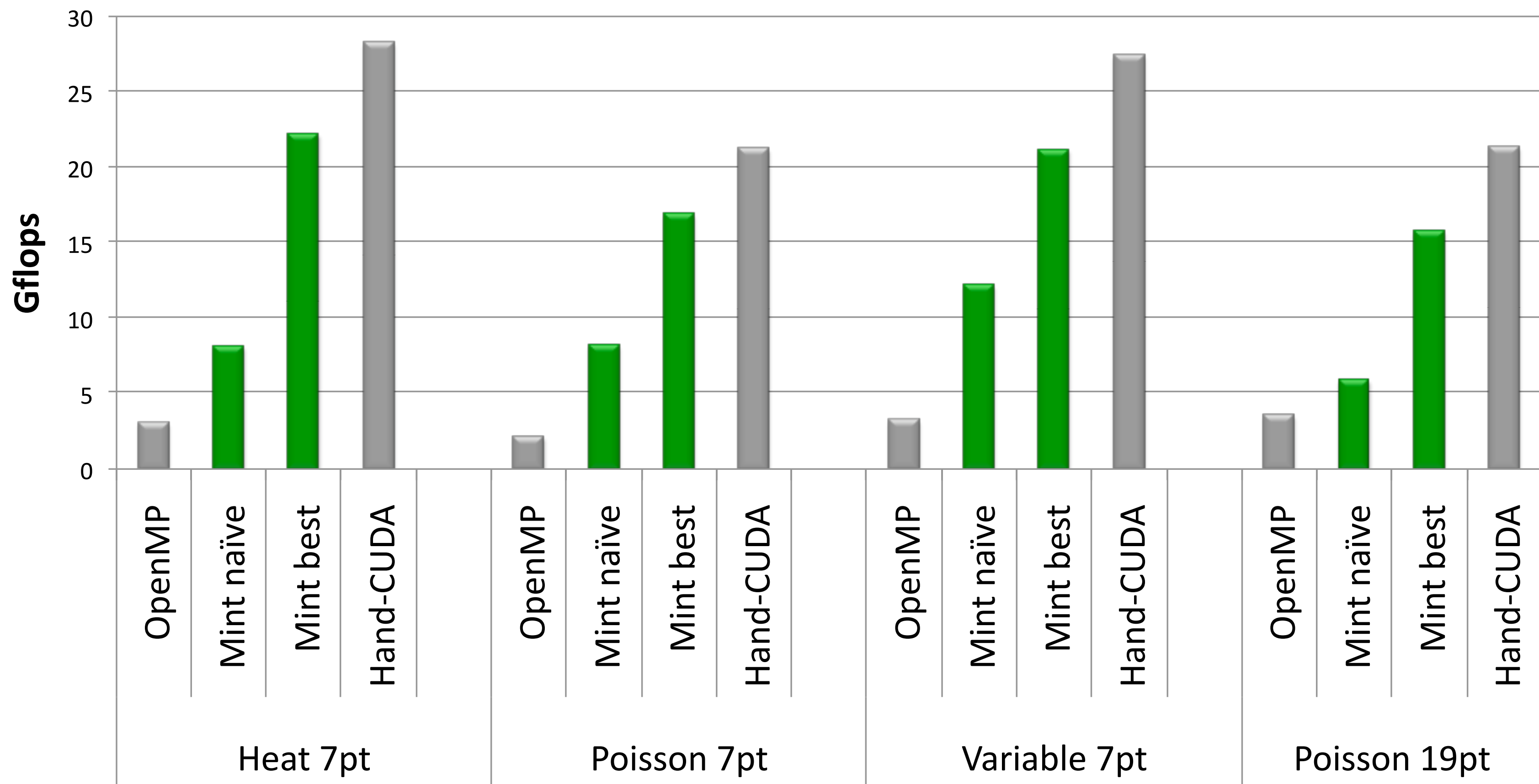
Performance Results

- We present results for Mint, hand-written CUDA and OpenMP
  - Mint was not used to generate the hand-written CUDA versions

3D Kernels	In, Out Arrays	Reads, Writes/pt	Operations/pt
Heat 7-pt	1,1	7,1	2(*),6(+)
Poisson 7-pt	2,1	7,1	2(*),6(+)
Variable 7-pt	3,1	15,1	7(*),13(+),6(-)
Poisson 19-pt	2,1	19,1	2(*),18(+)

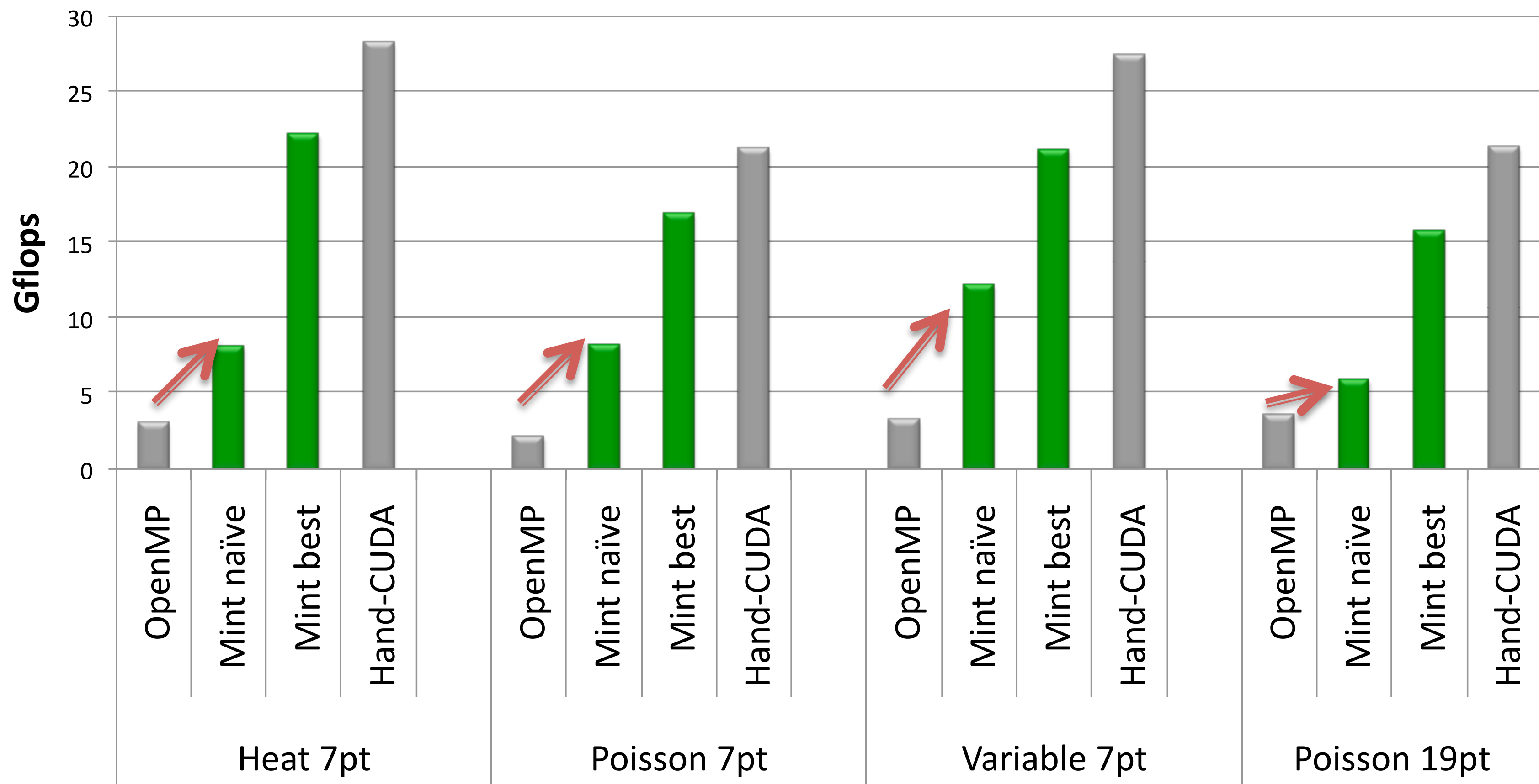
- OpenMP results are obtained on Intel Xeon E5504, quad-core, 16 GB memory, gcc 4.4.3, -03 –fopenmp using 4 threads.
- Mint and hand-CUDA results are obtained on
  - Tesla C1060, 4GB memory, nvcc 3.2 and Tesla C2050, 3GB memory, nvcc 3.2

Performance Comparison



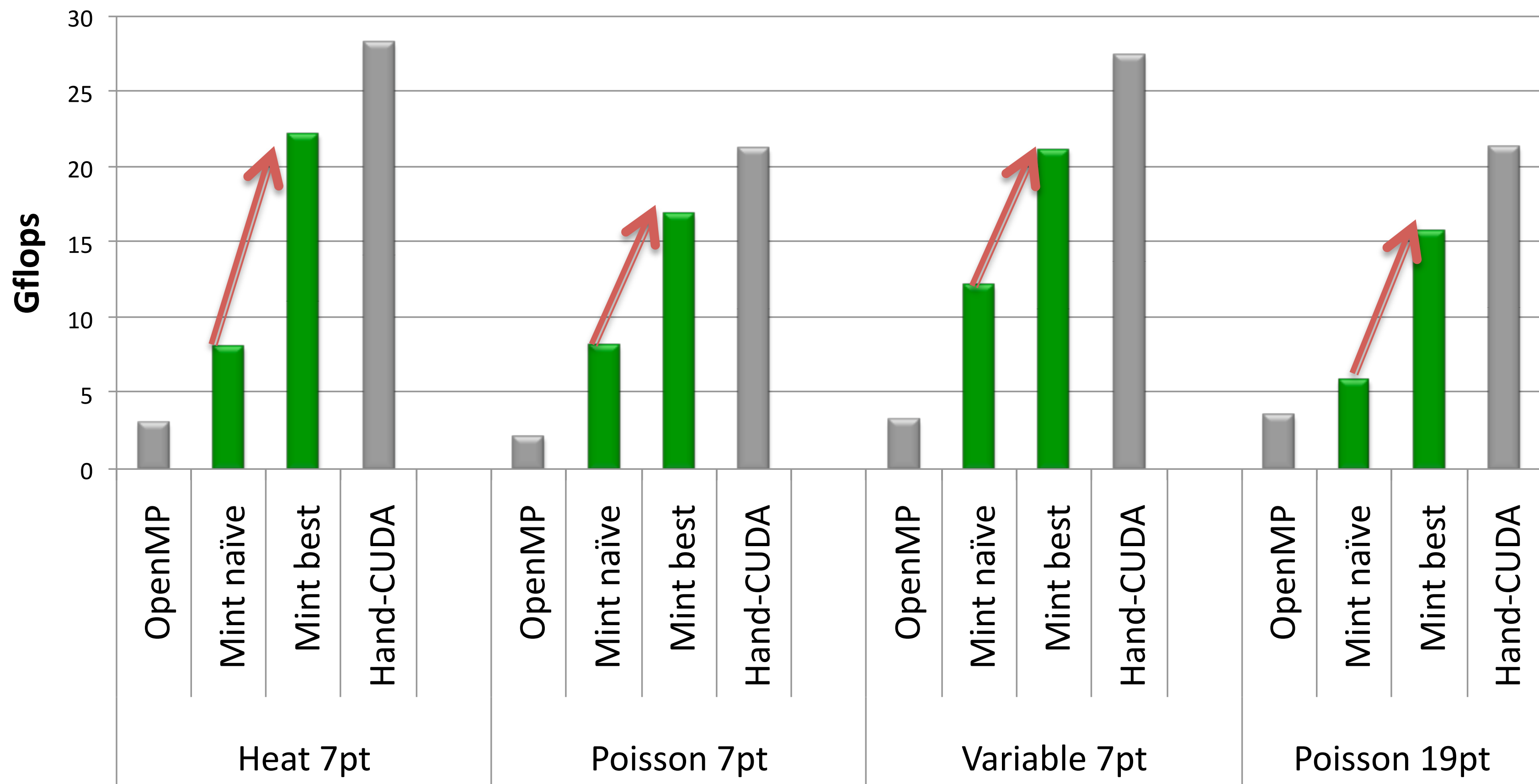
- On Tesla C1060, Mint achieves 79% of the hand-optimized CUDA

Performance Comparison



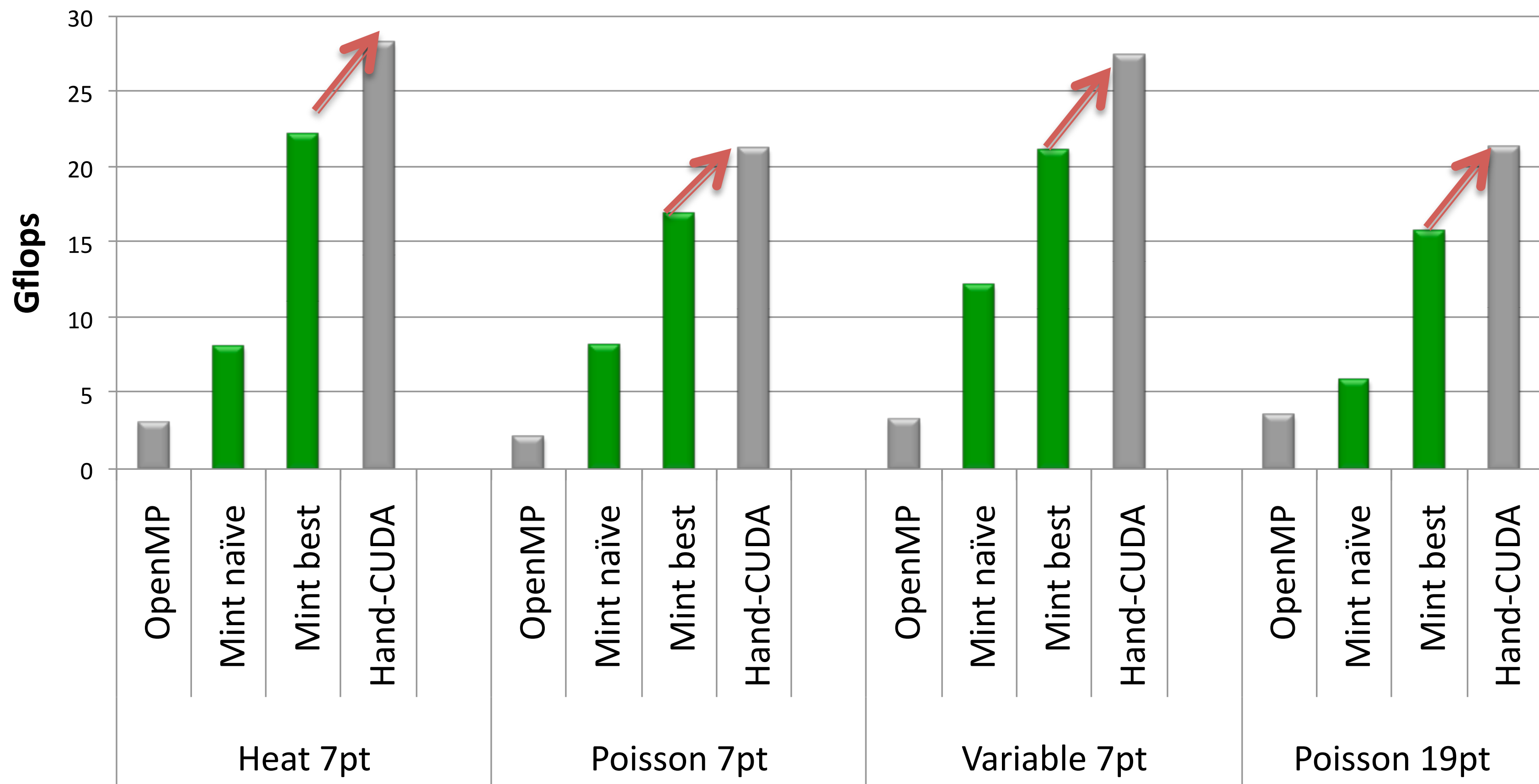
- The unoptimized Mint code outperforms the OpenMP code with an average 3x speedup.

Performance Comparison



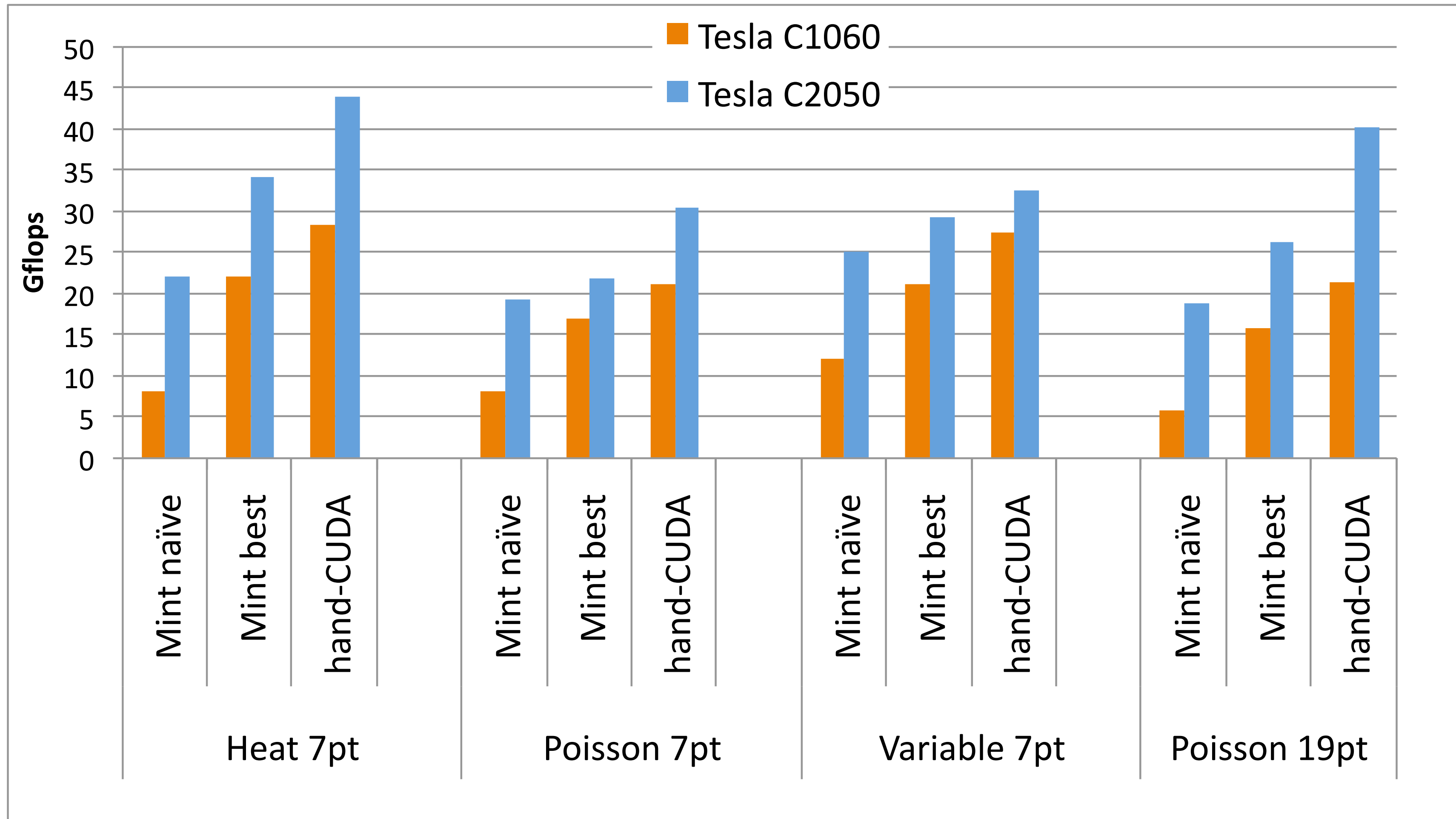
- The Mint optimizer improves the performance still further and provides a 2.6x speedup over the unoptimized translation.

Performance Comparison

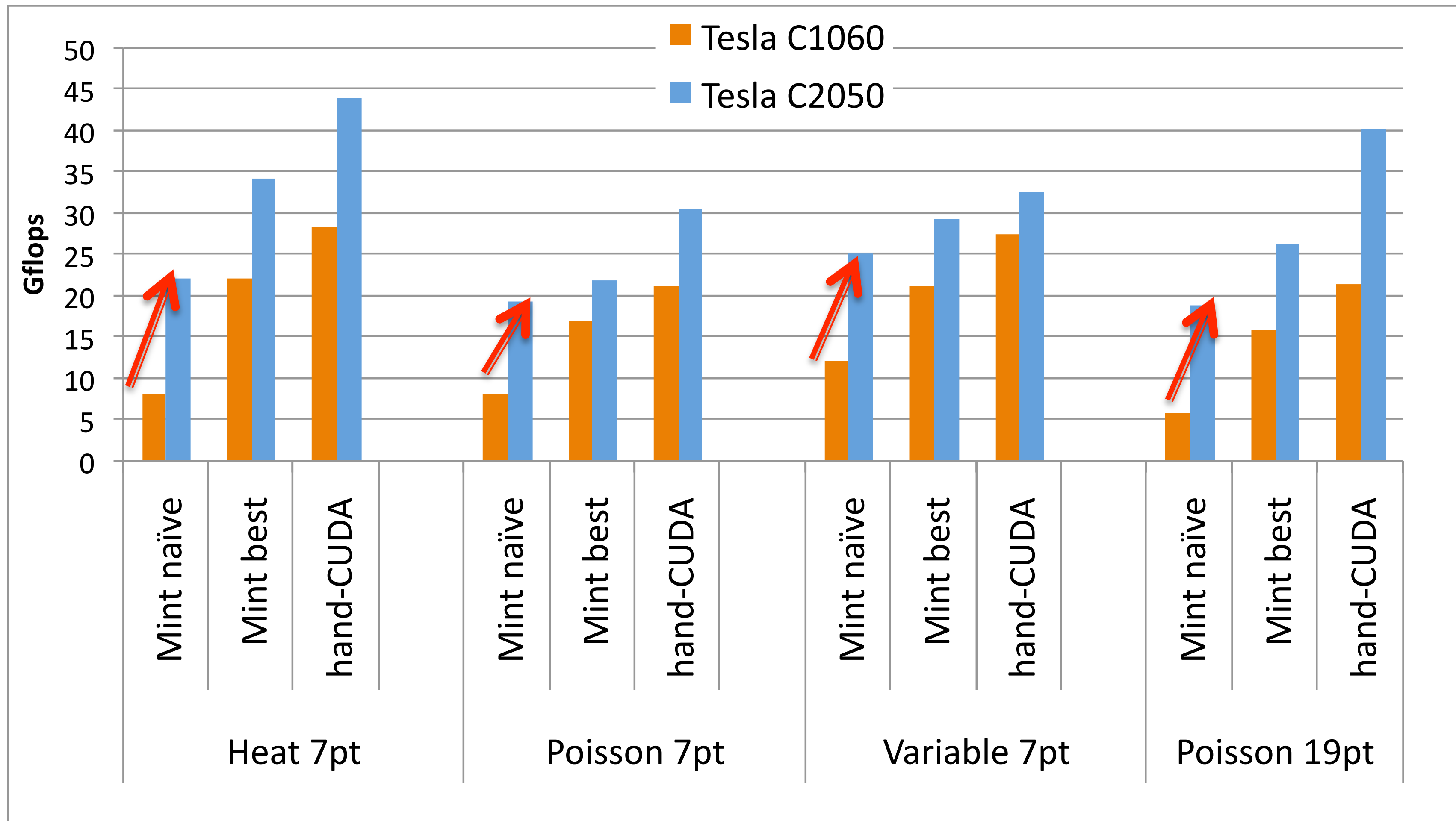


- Mint achieves on average 79% of the hand-optimized CUDA on the Tesla C1060 (200-series GPU).

200- vs 400-series of Tesla

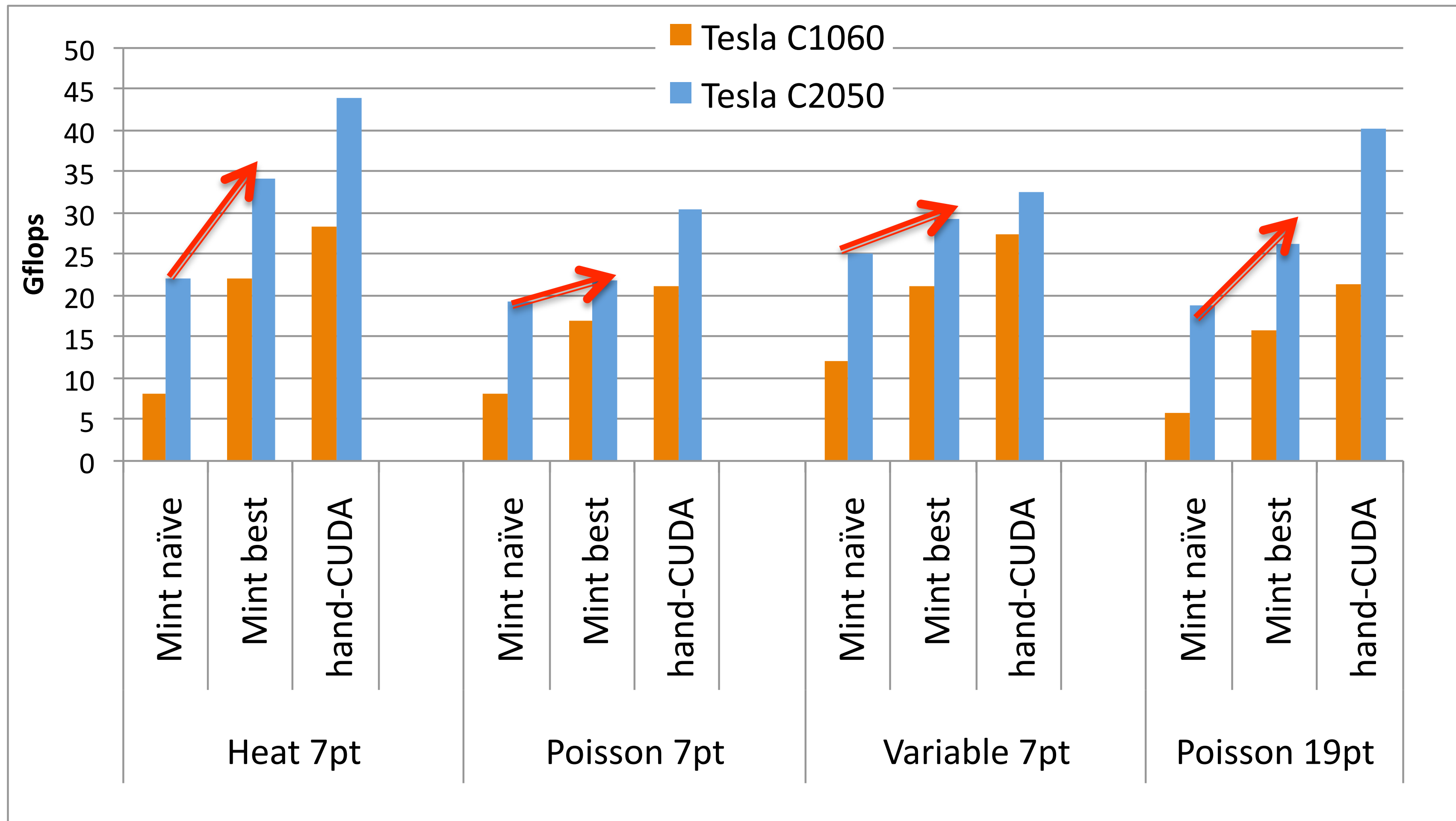


200- vs 400-series of Tesla



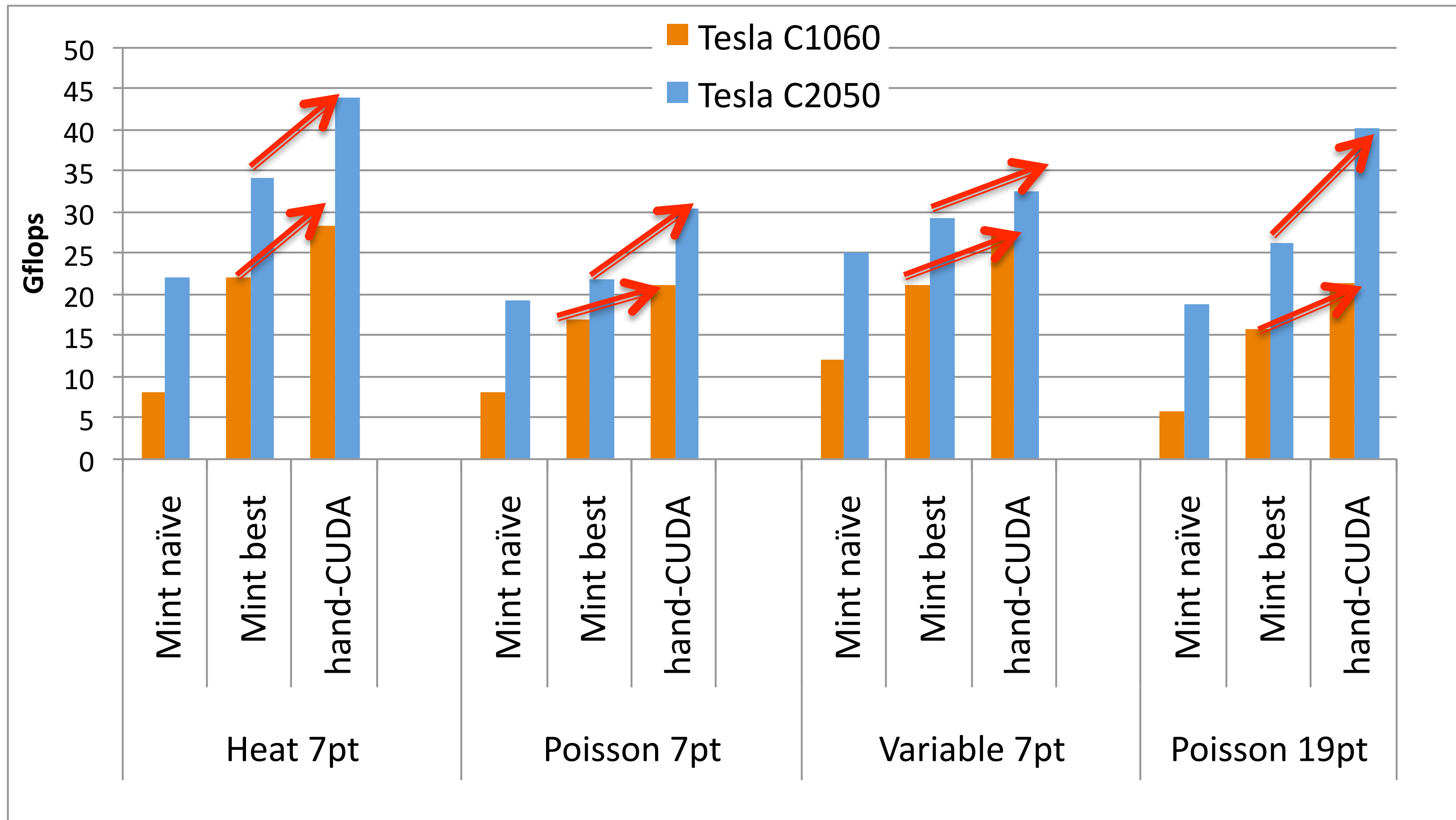
- Fermi provides a 2.5x speedup for the unoptimized Mint-generated kernels.

200- vs 400-series of Tesla



- Optimizations provide 1.3x speedup over the unoptimized kernels.

200- vs 400-series of Tesla



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]