CSE 262
Lecture 11

GPU Implementation of stencil methods (II)
Announcements

• Final presentations
  ☐ Friday March 13th, 10:30 AM to 1:00PM
  ☐ Room 3217, CSE Building (EBU3B)
Today’s lecture

• GPU Implementation of Stencil methods
  ♦ 2D
  ♦ 3D
Data Dependencies – Aliev Panfalov Model

- **ODE solver:**
  - No data dependency, trivially parallelizable
  - Requires a lot of registers to hold temporary variables

- **PDE solver:**
  - Jacobi update for the 5-point Laplacian operator.
  - Sweeps over a uniformly spaced mesh
  - Updates voltage to weighted contributions from the 4 nearest neighbors

```c
for (j=1; j<=m+1; j++) {
    _DOUBLE_ *RR = &R[j][1], *EE = &E[j][1];
    for (i=1; i<=n+1; i++, EE++, RR++) {
        // PDE SOLVER
        EE[0] = E_p[j][i]+α*(E_p[j][i+1]+E_p[j][i-1]-4*E_p[j][i]+E_p[j+1][i]+E_p[j-1][i]);
        // ODE Solver
        EE[0] += -dt*(kk*EE[0]*(EE[0]-a)*(EE[0]-1)+EE[0]*RR[0]);
        RR[0] += dt*(ε+M1* RR[0]/( EE[0]+M2))*(-RR[0]-kk*EE[0]*(EE[0]-b-1));
    }
}
```
Naïve CUDA Implementation

- All array references go through device memory
- ./apf -n 6144 -t 0.04, 16x16 thread blocks
  - C1060 (1.3)
  - SP, DP: 22, 13GFlops

```c
#define E_T-1[i,j] E_prev[(j+1)*(n+3) + (i+1)]
I = blockIdx.y*blockDim.y + threadIdx.y;
J = blockIdx.x*blockDim.x + threadIdx.x;
if ((I <= n) && (J <= n) )
```

```c
for (j=1; j<= n+1; j++)
  for (i=1; i<= n+1; i++)
```
Using Shared Memory (Kepler and cap. 1.3)

PDE Part
\[ E_t[i,j] = E_{t-1}[i,j] + \alpha (E_{t-1}[i+1,j] + E_{t-1}[i-1,j] + E_{t-1}[i,j+1] + E_{t-1}[i,j-1] - 4E_{t-1}[i,j]) \]
CUDA Code

__shared__ float block[DIM_Y + 2][DIM_X + 2];
int idx = threadIdx.x, idy = threadIdx.y;  //local indices
//global indices
int x = blockIdx.x * (DIM_X) + idx;
int y = blockIdx.y * (DIM_Y) + idy;
idy++; idx++;
unsigned int index = y * N + x;

//interior points
float center = E_prev[index];
block[idy][idx] = center;

__syncthreads();
Copying the ghost cells

if (idy == 1 && y > 0 ) // Most threads are idle
    block[0][idx] = E_prev[index - N];
else if (idy == DIM_Y && y < N-1)
    block[DIM_Y+1][idx] = E_prev[index + N];
if ( idx==1 && x > 0 )
    block[idy][0] = E_prev[index - 1];
else if (idx== DIM_X && x < N-1 )
    block[idy][DIM_X +1] = E_prev[index + 1];
__syncthreads();
The stencil computation and ODE

float r = R[index];

float e = center + \alpha \cdot (block[idy][idx-1] + block[idy][idx+1] + block[idy-1][idx] + block[idy+1][idx] - 4*center);

e = e - dt*(kk * e * (e - a) * (e - 1) + e * r);

E[index] = e;

R[index] = r + dt*(\varepsilon + M1 * r / (e + M2) \cdot r + kk * e * (e - b - 1));
Results on C1060 (Tesla)

GFlop/s rates for Nehalem and C1060 implementations

<table>
<thead>
<tr>
<th>N=4K</th>
<th>1 CPU-only</th>
<th>1 GPU</th>
<th>Speedup over CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>2.36</td>
<td>124.48</td>
<td>52.84</td>
</tr>
<tr>
<td>Double</td>
<td>2.01</td>
<td>25.69</td>
<td>12.81</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>N=16K</th>
<th>4 CPUs-only</th>
<th>4 GPUs</th>
<th>Speedup over CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>7.79</td>
<td>454.47</td>
<td>58.32</td>
</tr>
<tr>
<td>Double</td>
<td>3.88</td>
<td>102.48</td>
<td>26.41</td>
</tr>
</tbody>
</table>

- Single Precision *(22 Gflops w/o optimizations)*
  - Nearly saturates the off-chip memory bandwidth
  - Utilizing 98% of the sustainable bandwidth for the Tesla C1060.
  - Achieves 13.3% of the single precision peak performance
    - Single precision performance is bandwidth limited.

- Double Precision
  - 41.5% of the sustainable bandwidth
  - 1/3 of the peak double precision performance
  - Performance hurt by the division operation that appears in ODE
Sliding row optimization

- Create 1D thread block to process 2D data block
- Iterate over rows in y dim
- While first and last threads read ghost cells, others are idle

Compared to 2D thread blocking, 1D thread blocks improve performance by 12% improvement in double precision and 64% in single precision
Sliding rows

Top Row in Registers

Curr Row in Shared memory

Bottom Row in Registers

Top row ← Curr row,
Curr row ← Bottom row
Bottom row ← Read next row from global memory

Sliding row algorithm

Read new row from global memory
Limits to performance

• Recall that maximum sustainable performance is constrained by arithmetic intensity $q$, and the hardware’s capabilities

• Roofline: Running time = Max(Data motion, computation)

\[
\text{Attainable Performance}_{ij} = \min \left\{ \frac{\text{FLOP/s with Optimizations}_{1-i}}{q \times \text{Bandwidth with Optimizations}_{1-j}} \right\}
\]

• Division is slow on 1.3 capability devices: running time of Aliev-Panfilov kernel is not bandwidth bound!

<table>
<thead>
<tr>
<th>Double Precision, N=4K</th>
<th>Using Div</th>
<th>Replace(div, add)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak BW (GB/s)</td>
<td>102</td>
<td>102</td>
</tr>
<tr>
<td>Peak DP GFlop/s</td>
<td>78</td>
<td>78</td>
</tr>
<tr>
<td>Sustained BW (GB/s)</td>
<td>73</td>
<td>73</td>
</tr>
<tr>
<td>Achieved Bandwidth (GB/s)</td>
<td>30.3</td>
<td>51.3</td>
</tr>
<tr>
<td>Achieved GFlop/s</td>
<td>25.7</td>
<td>43.6</td>
</tr>
<tr>
<td>% of Sustained Bandwidth</td>
<td>41.5 %</td>
<td>70.3 %</td>
</tr>
<tr>
<td>% of Peak GFlop/s</td>
<td>32.9 %</td>
<td>55.8 %</td>
</tr>
</tbody>
</table>
Instruction Throughput (1.3 capability)

<table>
<thead>
<tr>
<th></th>
<th>Stencil</th>
<th>ODE 1</th>
<th>ODE 2</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>Multiply</td>
<td>-</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Madd</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>14*</td>
</tr>
<tr>
<td>Division</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>7*</td>
<td>9*</td>
<td>12*</td>
<td>28</td>
</tr>
</tbody>
</table>

- Not all the operations are multiply-and-add instructions
  - Add or multiply run at the half speed of MADD
- Register-to-register instructions achieve highest throughput
- Shared memory instructions only a fraction of the peak
  (66% in single, 84% in double precision)
Memory Accesses

Total Memory Accesses = \(4N^2 + \text{ghost cells}\)

Number of blocks = \(\frac{N}{d_x} \times \frac{N}{d_y}\)

Total Memory Accesses = \(4N^2 + \frac{N^2}{d_x d_y} \times 2(d_x + d_y)\)

Estimated Kernel Time = Total Mem. Acces (bytes) / Empirical Device Bandwidth
Today’s lecture

• Stencil methods on the GPU
  ♦ 2D
  ♦ 3D
  ♦ Source to source translation
3D Stencils

- More demanding
  - Large strides
  - Curse of dimensionality
Memory strides

For each \(i, j, k\)

\[
E_t(i, j, k) = c_0 \ast (E_{t-1}(i, j, k) + c_1 \ast (E_{t-1}(i+1, j, k) + E_{t-1}(i-1, j, k) + E_{t-1}(i, j+1, k) + E_{t-1}(i, j-1, k) + E_{t-1}(i, j, k-1) + E_{t-1}(i, j, k+1)))
\]

Linear array space
Data partitioning

- Split mesh into 3D tiles
- Divide elements in a tile over a thread block
On chip memory optimization

- Copy center plane into shared memory
- Store others in registers
- Move in and out of registers
Rotating planes strategy

- Copy center plane into shared memory
- Store others in registers
- Move in and out of registers
Performance Summary

- $N^3=256^3$, double precision

<table>
<thead>
<tr>
<th>GFLOPS</th>
<th>Tesla 1060</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td>8.9</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>15.5</td>
</tr>
<tr>
<td>Sliding Planes</td>
<td>20.7</td>
</tr>
<tr>
<td>Registers</td>
<td>23.6</td>
</tr>
</tbody>
</table>
Multiple Elements in Y-dim

- If we let a thread compute more than one plane, we can assign more than one row in the slowest varying dimension
- Reduces index calculations
  - But requires more registers
- May be advantageous in handling ghost cells
Contributions to Performance

• $N^3 = 256^3$, double precision

<table>
<thead>
<tr>
<th>GFLOPS</th>
<th>Tesla 1060</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve</td>
<td>8.9</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>15.5</td>
</tr>
<tr>
<td>Sliding Planes</td>
<td>20.7</td>
</tr>
<tr>
<td>Registers</td>
<td>23.6</td>
</tr>
<tr>
<td>MultipleY2</td>
<td>26.3</td>
</tr>
<tr>
<td>MultipleY4</td>
<td>26.2</td>
</tr>
</tbody>
</table>
Influence of memory traffic on performance

Flop:word (FMA)
Generational changes in implementation strategy

• On Fermi we do not see a large change in performance when we use shared memory! Cache helps!
  ♦ C1060 (1.3) cseclass05
  ♦ SP: 22, 73, 34 Gflops
  DP: 13, 45, 20 Gflops
• But on the next generation Kepler we do!
• Global memory references aren’t cached as in Fermi
• Caching used mostly to handle register spills
Today’s lecture

• Stencil methods on the GPU
  ♦ 2D
  ♦ 3D
  ♦ Source to source translation
Mint translator

• Source-to-source translator [ICS ‘11]
  ♦ Didem Unat, PhD 2012, now @ Koç Univ
  ♦ Annotated C source → optimized CUDA C
  ♦ Targets stencil methods

• For commonly used kernels Mint realized ~80% of the performance obtained from aggressively optimized CUDA on 200 and 400-series (Fermi) of GPUs [ICS ‘11]

• Real time performance for 3D image processing code [with H. Kim and J. Schülze, EAMA ’11, VDA ‘12]

• Realizes 83% of performance of hand coded earthquake modeling code AWP-ODC on Fermi [with J. Zhou, Y. Cui, CS&E 2012]
  
  185 [139 + 46] lines ~ 385 lines
Mint is competitive with hand coding

- Tesla C1060: Mint achieved 79% of hand-optimized CUDA
- OpenMP ran on Intel Nehalem with 4 threads
- Vasily Volkov’s hand optimized CUDA implementation
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. #pragma mint parallel default(shared)
4. {
5.    int t=0;
6.    while(t++ < T ){
7.        #pragma mint for nest(all) tile(16,16,1)
8.        for(int z=1; z<= k; z++)
9.            for(int y=1; y<= m; y++)
10.           for(int x=1; x<= n; x++)
11.              Unew[z][y][x] = c0 * U[z][y][x] +
12.                 c1 * (U[z][y][x-1] + U[z][y][x+1] +
13.                    U[z][y-1][x] + U[z][y+1][x] +
14.                    U[z-1][y][x] + U[z+1][y][x]);
15.       #pragma mint single{
16.           double*** tmp;
17.           tmp = U; U = Unew; Unew = tmp;
18.        } //end of single
19.    } //end of while
20. } //end of parallel region
21. #pragma mint copy(U, dU, fromDevice, (n+2), (m+2), (k+2))

Data Xfers

Accelerated Region

Nested-for

Master Thread

Data Xfer
Stencil Analyzer

- Determines pattern of array accesses involving central point and nearest neighbors
- How much shared memory do we need?
- Which ghost cells to load?