3D Finite Difference Computation on GPUs with CUDA

Paulius Micikevicius, NVIDIA
Outline

3D stencil computation
Access redundancy
  Metric for predicting relative performance of implementations
  (could be applied to any user-managed cache architecture)
3 Implementations:
  Naïve
  2-pass
  1-pass
Performance results
  Single GPU
  Multi-GPU
3D Finite Difference

- 25-point stencil (8th order in space)
- Isotropic: 5 distinct coefficients
- For each output element’s stencil we need:
  - 29 flops
  - 25 input values

Some applications:
- FD of the wave equation
  (oil & gas exploration)
General Approach

Tile a 2D slice with 2D threadblocks
- Slice in the two fastest dimensions: x and y

Each thread iterates along the slowest dimension (z)
- Each thread is responsible for one element in every slice
- Only one kernel launch
- Also helps data reuse
Naive Implementation

- One thread per output element
- Fetch all data for every output element
  - Redundant: input is read 25 times
  - Required bandwidth = 25 reads, 1 write (26x)

**Access Redundancy:**
- Proposed metric for evaluating implementations
- Number of memory accesses per output element
  - Appropriate for user-managed-cache architectures

**Optimization: share data among threads**
- Use shared memory for data needed by many threads
- Use registers for data not shared among threads
3D Subdomain in Shared Memory
Using Shared Memory: First Take

Read a 3D subdomain from gmem into smem
   Compute from smem

Limited by amount of smem (16KB)
   Need 4-element halos in each direction:
      (dimx+8)×(dimy+8)×(dimz+8) storage for dimx×dimy×dimz subdomain
      dimx should be multiple of 16 for max bandwidth (coalescing)
   What would fit (4-byte elements):
      24x14x12 storage (16x6x4 subdomain)
      Only 9.5% of storage is not halo (could be improved to 20%)

Requires bandwidth for 5.8x data size
   4.83x read, 1 write
   Better than 26x but still redundant
Using Shared Memory: Second Take

3D FD done with 2 passes:
- 2D-pass (2DFD)
- 1D-pass (1DFD and output of 2D-pass)

SMEM is sufficient for 2D subdomains
- Square tiles require the smallest halos
- Up to 64x64 storage (56x56 subdomain)
  - 76.5% of storage is not halo

Volume accesses:
- Read/write for both passes
  - 2D-pass reads original, halo, and 1D-pass output
  - 16x16 subdomain tiles: 6.00 times
  - 32x32 subdomain tiles: 5.50 times
  - 56x56 subdomain tiles: 5.29 times
Using Shared Memory: Third Take

Combine the 2D and 1D passes
- 1D pass needs no SMEM: keep data in registers
Input Reuse within a 2x2 Threadblock

Store the xy-slice in SMEM
Each thread keeps its 8 z-elements in registers
  4 “infront”, 4 “behind”
Using Shared Memory: Third Take

Combine the 2D and 1D passes
- 1D pass needs no SMEM: keep data in registers

16x16 2D subdomains in shared memory
- 16x16 threadblocks
- 24x24 SMEM storage (2.25KB) per threadblock
  - 44% of storage is not halo
  - Volume is accessed 3 times (2 reads, 1 write)
    - 2 reads due to halo
Using Shared Memory: Third Take

Combine the 2D and 1D passes
- 1D pass needs no SMEM: keep data in registers

16x16 2D subdomains
- 16x16 threadblocks
- 24x24 SMEM storage (2.25KB) per threadblock
  - 44% of storage is not halo
  - Volume is accessed 3 times (2 reads, 1 write)

32x32 2D subdomains
- 32x16 threadblocks
- 40x40 SMEM storage (6.25KB) per threadblock
  - 64% of storage is not halo
  - Volume is accessed 2.5 times (1.5 reads, 1 write)
Inner Loop of 16x16-tile stencil kernel

// ------- advance the slice (move the thread-front) -------------------
behind4 = behind3;
behind3 = behind2;
behind2 = behind1;
behind1 = current;
current = infront1;
infront1 = infront2;
infront2 = infront3;
infront3 = infront4;
infront4 = g_input[in_idx];
in_idx += stride;
out_idx += stride;
__syncthreads();

// ------- update the data slice in smem ----------------------------------
if( threadIdx.y<radius )
  // top and bottom halo
  {
  s_data[threadIdx.y][tx] = g_input[out_idx – radius * dimx];
  s_data[threadIdx.y+16+radius][tx] = g_input[out_idx + 16 * dimx];
  }
if( threadIdx.x<radius )
  // left and right halo
  {
  s_data[ty][threadIdx.x] = g_input[out_idx – radius];
  s_data[ty][threadIdx.x+16+radius] = g_input[out_idx + 16];
  }
  s_data[ty][tx] = current;
  // 16x16 “internal” data
  __syncthreads();

// compute the output value -----------------------------------------------
float div = c_coeff[0] * current;
div += c_coeff[1] * ( infront1 + behind1 + s_data[ty-1][tx]+ s_data[ty+1][tx]+ s_data[ty][tx]-1+ s_data[ty][tx]+1 );
div += c_coeff[3] * ( infront3 + behind3 + s_data[ty-3][tx]+ s_data[ty+3][tx]+ s_data[ty][tx-3]+ s_data[ty][tx]+3 );
g_output[out_idx] = div;
Inner Loop of 16x16-tile FD kernel

// ------- advance the slice (move the thread-front) -------------------
behind4 = behind3;
behind3 = behind2;
behind2 = behind1;
behind1 = current;
current = infront1;
infront1 = infront2;
infront2 = infront3;
infront3 = infront4;
infront4 = g_input[in_idx];
in_idx += stride;
out_idx += stride;
__syncthreads();

// ------- update the data slice in smem --------------------------------
if( threadIdx.y<radius ) // top and bottom halo
{
    s_data[threadIdx.y][tx] = g_input[out_idx – radius * dimx];
s_data[threadIdx.y+16+radius][tx] = g_input[out_idx + 16 * dimx];
}
if( threadIdx.x<radius ) // left and right halo
{
    s_data[ty][threadIdx.x] = g_input[out_idx – radius];
s_data[ty][threadIdx.x+16+radius] = g_input[out_idx + 16];
}
s_data[ty][tx] = current; // 16x16 “internal” data
__syncthreads();

// compute the output value ---------------------------------------------
float temp = 2.f * current - g_next[out_idx];
float div = c_coeff[0] * current;
div += c_coeff[1] * (infront1 + behind1 + s_data[ty-1][tx]+ s_data[ty+1][tx]+ s_data[ty][tx-1]+ s_data[ty][tx+1]);
div += c_coeff[3] * (infront3 + behind3 + s_data[ty-3][tx]+ s_data[ty+3][tx]+ s_data[ty][tx-3]+ s_data[ty][tx+3]);
g_output[out_idx] = temp + div * g_vsq[out_idx];

2 more GMEM reads
4 more FLOPS

Per output element:
• 33 FLOPS
• 5 GMEM accesses (32bit)

Based on FD code by Scott Morton, Hess
Redundancy and Performance: Various kernels for 25-stencil

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Naïve</th>
<th>2-pass 32x32</th>
<th>1-pass 16x16</th>
<th>1-pass 32x32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Redundancy</td>
<td>26x</td>
<td>5.5x</td>
<td>3.0x</td>
<td>2.5x</td>
</tr>
<tr>
<td>480 × 480 × 400</td>
<td>614</td>
<td>3,148</td>
<td>4,774</td>
<td>4,029</td>
</tr>
<tr>
<td>544 × 544 × 400</td>
<td>597</td>
<td>3,162</td>
<td>4,731</td>
<td>4,132</td>
</tr>
<tr>
<td>640 × 640 × 400</td>
<td>565</td>
<td>3,031</td>
<td>4,802</td>
<td>3,993</td>
</tr>
<tr>
<td>800 × 800 × 400</td>
<td>492</td>
<td>3,145</td>
<td>3,611</td>
<td>4,225</td>
</tr>
</tbody>
</table>
### Redundancy and Performance: Various kernels for 25-stencil

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Naïve</th>
<th>2-pass 32x32</th>
<th>1-pass 16x16</th>
<th>1-pass* 16x16</th>
<th>1-pass 32x32</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Redundancy</strong></td>
<td>26x</td>
<td>5.5x</td>
<td>3.0x</td>
<td>3.0x</td>
<td>2.5x</td>
</tr>
<tr>
<td>480 x 480 x 400</td>
<td>614</td>
<td>3,148</td>
<td>4,774</td>
<td>3,930</td>
<td>4,029</td>
</tr>
<tr>
<td>544 x 544 x 400</td>
<td>597</td>
<td>3,162</td>
<td>4,731</td>
<td>3,822</td>
<td>4,132</td>
</tr>
<tr>
<td>640 x 640 x 400</td>
<td>565</td>
<td>3,031</td>
<td>4,802</td>
<td>3,776</td>
<td>3,993</td>
</tr>
<tr>
<td>800 x 800 x 400</td>
<td>492</td>
<td>3,145</td>
<td>3,611</td>
<td>3,603</td>
<td>4,225</td>
</tr>
</tbody>
</table>

**1-pass: 16x16 vs 32x32 tile**

- 16x16 tile code runs 1.5x threads per SM than 32x32
- *1-pass* 16x16 version forced the same number of threads
Single-Pass 3D Finite Difference Performance in MPoints/s
(8\textsuperscript{th} order in space, 2\textsuperscript{nd} order in time)

<table>
<thead>
<tr>
<th>Data Dimensions</th>
<th>16x16 Tiles</th>
<th>32x32 Tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>480 x 480 x 400</td>
<td>3,077.8</td>
<td>3,081.7</td>
</tr>
<tr>
<td>544 x 544 x 544</td>
<td>2,797.9</td>
<td>3,181.2</td>
</tr>
<tr>
<td>640 x 640 x 640</td>
<td>2,558.5</td>
<td>3,106.4</td>
</tr>
<tr>
<td>800 x 800 x 400</td>
<td>2,459.0</td>
<td>3,256.9</td>
</tr>
</tbody>
</table>

Read: 25-point stencil, velocity, and previous time step value
(forward solve of the RTM, though boundary conditions are ignored)

Measured on: Tesla C1060, CUDA 2.0 driver/toolkit
Multi-GPU Approach (8th order in space)

Test with 2 GPUs:
- Split the data volume between 2 GPUs
- Split along the slowest-varying dimension
- Each GPU gets \((\text{dim}z+4)\) slices
Every Time Step

Streams and async memcopies are used to overlap computation and communication in Phase 2.
for(int i=0; i<num_time_steps; i++)
{
    launch_kernel( d_output+offset1, d_input+offset1, dimx, dimy, 12, stream1);

    launch_kernel( d_output+offset2, d_input+offset2, dimx, dimy, dimz, stream2);
    cudaMemcpyAsync( h_ghost_own, d_ghost_own, num_ghost_bytes, cudaMemcpyDeviceToHost, stream1);
    cudaStreamSynchronize( stream1);
    MPI_Sendrecv( h_ghost_own, num_ghost_elmnts, MPI_REAL, partner, i,
                  h_ghost_partner, num_ghost_elmnts, MPI_REAL, partner, i,
                  MPI_COMM_WORLD, &status);
    cudaMemcpyAsync( d_ghost_partner, h_ghost_partner, num_ghost_bytes, cudaMemcpyHostToDevice, stream1);
    cudaThreadSynchronize();
}
Performance Scaling with 2 GPUs

16×16 Tile Finite Difference Kernel

<table>
<thead>
<tr>
<th>Data Dimensions</th>
<th>Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>480 × 480 × 200</td>
<td>1.51</td>
</tr>
<tr>
<td>480 × 480 × 300</td>
<td>1.93</td>
</tr>
<tr>
<td>480 × 480 × 400</td>
<td>2.04</td>
</tr>
<tr>
<td>544 × 544 × 544</td>
<td>2.02</td>
</tr>
<tr>
<td>640 × 640 × 640</td>
<td>2.26</td>
</tr>
<tr>
<td>800 × 800 × 400</td>
<td>2.04</td>
</tr>
</tbody>
</table>

Using 2 GPUs (half of Tesla S1070)
3DFD Scaling with Multiple GPUs (8th order in space, 2nd order in space)

<table>
<thead>
<tr>
<th>Data Dimensions</th>
<th>1 GPU</th>
<th>2 GPUs</th>
<th>4 GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>480 × 480 × 800</td>
<td>1.00</td>
<td>1.99</td>
<td>3.90</td>
</tr>
<tr>
<td>480 × 480 × 1200</td>
<td>1.00</td>
<td>2.00</td>
<td>4.05</td>
</tr>
<tr>
<td>640 × 640 × 640</td>
<td>1.00</td>
<td>2.17</td>
<td>3.62</td>
</tr>
</tbody>
</table>

Each GPU communicates with 2 neighbors: twice the communication cost

Using 2 Tesla S1070s (connected to 4 CPU nodes)
Questions?
Two-Pass Stencil-Only Performance

Hardware: Tesla C1060 (4GB, 240 SPs)

2D-pass (32x32 tile):
- 544x512x200: 5,811 Mpoints/s
- 800x800x800: 5,981 Mpoints/s

1D-pass (3 gmem accesses / point):
- 544x512x200: 6,547 Mpoints/s
- 800x800x800: 6,307 Mpoints/s

Combined:
- 544x512x200: 3,075 Mpoints/s
- 800x800x800: 3,071 Mpoints/s