M02: High Performance Computing with CUDA

Optimizing CUDA

Paulius Micikevicius
Outline

- Memory Coalescing
- Staging Coefficients
- Streams and Asynchronous API
- Sample: 3D Finite Difference
MEMORY COALESCING
Memory Performance

To maximize global memory bandwidth:
- Minimize the number of bus transactions
- Coalesce memory accesses

Coalescing
- Memory transactions are per half-warp (16 threads)
- In best cases, one transaction will be issued for a half-warp
- Latest hardware relaxes coalescing requirements
  - Compute capability 1.2 and later
Coalescing: Compute Capability < 1.2

A coordinated read by a half-warp (16 threads)

A contiguous region of global memory:
- 64 bytes - each thread reads a word: int, float, ...
- 128 bytes - each thread reads a double-word: int2, float2
- 256 bytes – each thread reads a quad-word: int4, float4, ...

Additional restrictions:
- Starting address must be a multiple of region size
- The $k^{th}$ thread in a half-warp must access the $k^{th}$ element in a block being read

Exception: not all threads must be participating
- Predicated access, divergence within a halfwarp
Coalesced Access: Reading floats

All threads participate

Some Threads Do Not Participate
Uncoalesced Access: Reading floats

Permuted Access by Threads

Misaligned Starting Address (not a multiple of 64)
Coalescing: Timing Results

Experiment:
- Kernel: read a float, increment, write back
- 3M floats (12MB)
- Times averaged over 10K runs

12K blocks x 256 threads:
- 356μs – coalesced
- 357μs – coalesced, some threads don’t participate
- 3,494μs – permuted/misaligned thread access
Possible bus transaction sizes:
- 32B, 64B, or 128B
- Memory segment must be aligned
  - First address = multiple of segment size

Hardware coalescing for each half-warp:
- Carry out the smallest possible number of transactions
- Reduce transaction size when possible
Thread 0
Address 0
Address 4
Address 8
Address 12
Address 16
Address 20
Address 24
Address 28

Thread 1
Address 1
Address 5
Address 9
Address 13
Address 17
Address 21
Address 25
Address 29

Thread 2
Address 2
Address 6
Address 10
Address 14
Address 18
Address 22
Address 26
Address 30

Thread 3
Address 3
Address 7
Address 11
Address 15
Address 19
Address 23
Address 27
Address 31

Thread 4
Address 4
Address 8
Address 12
Address 16
Address 20
Address 24
Address 28
Address 32

Thread 5
Address 5
Address 9
Address 13
Address 17
Address 21
Address 25
Address 29
Address 33

Thread 6
Address 6
Address 10
Address 14
Address 18
Address 22
Address 26
Address 30
Address 34

Thread 7
Address 7
Address 11
Address 15
Address 19
Address 23
Address 27
Address 31
Address 35

Thread 8
Address 8
Address 12
Address 16
Address 20
Address 24
Address 28
Address 32
Address 36

Thread 9
Address 9
Address 13
Address 17
Address 21
Address 25
Address 29
Address 33
Address 37

Thread 10
Address 10
Address 14
Address 18
Address 22
Address 26
Address 30
Address 34
Address 38

Thread 11
Address 11
Address 15
Address 19
Address 23
Address 27
Address 31
Address 35
Address 39

Thread 12
Address 12
Address 16
Address 20
Address 24
Address 28
Address 32
Address 36
Address 40

Thread 13
Address 13
Address 17
Address 21
Address 25
Address 29
Address 33
Address 37
Address 41

Thread 14
Address 14
Address 18
Address 22
Address 26
Address 30
Address 34
Address 38
Address 42

Thread 15
Address 15
Address 19
Address 23
Address 27
Address 31
Address 35
Address 39
Address 43

Address 0 to 43
Coalescing Algorithm

- Find the memory segment that contains the address requested by the lowest numbered active thread:
  - 32B segment for 8-bit data
  - 64B segment for 16-bit data
  - 128B segment for 32, 64 and 128-bit data.
- Find all other active threads whose requested address lies in the same segment
- Reduce the transaction size, if possible:
  - If size == 128B and only the lower or upper half is used, reduce transaction to 64B
  - If size == 64B and only the lower or upper half is used, reduce transaction to 32B
- Carry out the transaction, mark threads as inactive
- Repeat until all threads in the half-warp are serviced
Comparing Compute Capabilities

Compute capability < 1.2
- Requires threads in a half-warp to:
  - Access a single aligned 64B, 128B, or 256B segment
  - Threads must issue addresses in sequence
- If requirements are not satisfied:
  - Separate 32B transaction for each thread

Compute capability ≥ 1.2
- Does not require sequential addressing by threads
- Perf degrades gracefully when a half-warp addresses multiple segments
STAGING COEFFICIENTS
General Use Case

Kernel contains a loop:
- For a given iteration, all threads read the same value
- Different values for different iterations

Implementation choices:
- Each thread reads in every iteration:
  - From global memory
  - From constant memory (cached)
  - From 1D texture (cached)
- Threads *stage* reads through shared memory:
  - Threads collectively place coefficients into smem
  - Each thread reads from smem in every iteration
Experiment

Threads iterate through a 3D volume along z

- For a given z-iteration, threads write the same coefficient
- Different coefficients for every z

```c
__global__ void gmem_bcast( float *g_data, float *g_coeff, int dimx, int dimy, int dimz )
{
    int ix  = blockIdx.x*blockDim.x + threadIdx.x;
    int iy  = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;
    int stride = dimx*dimy;

    for(int iz=0; iz<dimz; iz++)
    {
        g_data[idx] = g_coeff[iz];
        idx += stride;
    }
}
```
Kernel with Staged Coefficients:

Number of coefficients ≤ threads per block

```c
__global__ void gmem_staged( float *g_data, float *g_coeff, int dimx, int dimy, int dimz )
{
    int ix  = blockIdx.x*blockDim.x + threadIdx.x;
    int iy  = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;
    int stride = dimx*dimy;

    __shared__ float s_coeff[NUM_COEFF];
    int thread_id = threadIdx.y*blockDim.x + threadIdx.x;
    if( thread_id < NUM_COEFF )
        s_coeff[thread_id] = g_coeff[thread_id];
    __syncthreads();

    for(int iz=0; iz<dimz; iz++)
    {
        g_data[idx] = s_coeff[iz];
        idx += stride;
    }
}
```
Kernel with Staged Coefficients

Any number of coefficients that fits into shared mem

```c
__global__ void gmem_staged( float *g_data, float *g_coeff, int dimx, int dimy, int dimz )
{
    int ix  = blockIdx.x*blockDim.x + threadIdx.x;
    int iy  = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;
    int stride = dimx*dimy;

    __shared__ float s_coeff[NUM_COEFF];
    int thread_id = threadIdx.y*blockDim.x + threadIdx.x;
    int num_threads = blockDim.x*blockDim.y;
    for(int i=thread_id; i<NUM_COEFF; i+=num_threads)
    {
        s_coeff[i] = g_coeff[i];
    }

    __syncthreads();

    for(int iz=0; iz<dimz; iz++)
    {
        g_data[idx] = s_coeff[iz];
        idx += stride;
    }
}
```
### 800x800x400 data

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gmem bcast</td>
<td>19</td>
</tr>
<tr>
<td>cmem bcast</td>
<td>9</td>
</tr>
<tr>
<td>texture bcast</td>
<td>9</td>
</tr>
<tr>
<td>gmem staged</td>
<td>9</td>
</tr>
</tbody>
</table>
STREAMS AND ASYNC API
Streams and Async API

Default API:
- Kernel launches are asynchronous with CPU
- Memcopies (D2H, H2D) block CPU thread
- CUDA calls block on GPU
- Serialized by the driver

Streams and async functions provide:
- Memcopies (D2H, H2D) asynchronous with CPU
- Ability to concurrently execute a kernel and a memcopy

Stream = sequence of operations that execute in order on GPU
- Operations from different streams can be interleaved
- A kernel and memcopy from different streams can be overlapped
Overlap kernel and memory copy

Requirements:
- D2H or H2D memcopy from pinned memory
- Device with compute capability ≥ 1.1 (G84 and later)
- Kernel and memcopy in different, non-0 streams

Code:
```c
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync( dst, src, size, dir, stream1 );
kernelt <<<grid, block, 0, stream2>>>(...);
```

potentially overlapped
CUDA Events

- Events are inserted into streams of CUDA calls
  ```
  cudaEventRecord( event, stream )
  ```
- Event is *recorded* when the GPU reaches it in a stream
  ```
  Record = assign a timestamp (GPU clocktick)
  ```
- Useful for timing, querying/syncing with GPU
- Stream/event queries:
  ```
  Do not block the CPU thread
  cudaStreamQuery( stream )
  ```
  Indicates whether the stream is idle
  ```
  cudaEventQuery( event )
  ```
  Indicates whether the event has been recorded
CPU/GPU Synchronization

Three ways to synchronize CPU-thread and GPU:

- `cudaThreadSynchronize()`: Blocks until all previously issued CUDA calls complete.
- `cudaStreamSynchronize( stream )`: Blocks until all CUDA calls issued to the given stream complete.
- `cudaEventSynchronize( event )`: Blocks until the given event is recorded on GPU.

Any CUDA call to stream-0 blocks until previous calls complete.

- No CUDA calls can be overlapped with a stream-0 call.
Timing with CUDA Events

- Timer resolution ~ GPU clock period
- Remember that timing is done on GPU

Float elapsed_time_ms = 0.0f;
cudaEvent_t start, stop;
cudaEventCreate( &start );
cudaEventCreate( &stop );

cudaEventRecord( start, 0 );
some_kernel<<<....>>>(...);
cudaEventRecord( stop, 0 );

cudaEventSynchronize( stop );
cudaEventElapsedTime( &elapsed_time_ms, start, stop );
3D FINITE DIFFERENCE
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
- FD of the wave equation
  - More details on application in Scott Morton’s *Seismic Imaging* talk at 1:30
General Approach

- Tile a 2D slice with 2D threadblocks
  - Slice in the two fastest dimensions: x and y
- 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
- Optimization: share data among threads
  - Use shared memory for data needed by many threads
  - Use registers for data needed not shared among threads
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:
    - \((\text{dimx}+8)\times(\text{dimy}+8)\times(\text{dimz}+8)\) storage for \(\text{dimx}\times\text{dimy}\times\text{dimz}\) subdomain
    - \text{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 25x but still redundant
3D Subdomain in Shared Memory

- Internal data
- Halo
Using Shared Memory: Second Take

- 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

- 3D FD done with 2 passes:
  - 2D-pass
  - 1D-pass

- 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
Two-Pass Stencil 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
Using Shared Memory: Third Take

- Combine the 2D and 1D passes
  - 1D pass needs no SMEM: keep data in registers
Input Re-use by 2D Threadblocks

Store the xy-slice in SMEM
Each thread keeps its 8 z-elements in registers
  - 4 “ahead”, 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
  - 16x16 threadblocks
  - 24x24 SMEM storage (2.25KB) per threadblock
    - 44% of storage is not halo
    - Volume is accessed 3 times (2 read, 1 write)
      - 2 read 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 read, 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 read, 1 write)
Inner Loop of 16x16-tile stencil kernel

// ——— advance the slice (move the thread-front) ————
behind.w = behind.z;
behind.z = behind.y;
behind.y = behind.x;
behind.x = current;
current = infront.x;
infront.x = infront.y;
infront.y = infront.z;
infront.z = infront.w;
infront.w = g_input[in_idx];
in_idx += stride;
out_idx += stride;
__syncthreads();

// ——— update the data slice in smem ————
if (threadIdx.y<radius) // top and bottom apron
{
    s_data[threadIdx.y][tx] = g_input[out_idx – radius * dimx];
    s_data[threadIdx.y+20][tx] = g_input[out_idx + 16 * dimx];
}  
if (threadIdx.x<radius) // left and right apron
{
    s_data[ty][threadIdx.x] = g_input[out_idx – radius];
    s_data[ty][threadIdx.x+20] = 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] * (infront.x + behind.x + s_data[ty-1][tx] + s_data[ty1][tx] + s_data[ty][tx-1] + s_data[ty][tx+1]);
div += c_coeff[2] * (infront.x + behind.x + s_data[ty-2][tx] + s_data[ty+2][tx] + s_data[ty][tx-2] + s_data[ty][tx+2]);
div += c_coeff[3] * (infront.x + behind.x + s_data[ty-3][tx] + s_data[ty+3][tx] + s_data[ty][tx-3] + s_data[ty][tx+3]);
div += c_coeff[4] * (infront.x + behind.x + s_data[ty-4][tx] + s_data[ty+4][tx] + s_data[ty][tx-4] + s_data[ty][tx+4]);
g_output[out_idx] = div;
Inner Loop of 16x16-tile FD kernel

// ------- advance the slice (move the thread-front) -------------------------
behind.w = behind.z;
behind.z = behind.y;
behind.y = behind.x;
behind.x = current;
current = infront.x;
infront.x = infront.y;
infront.y = infront.z;
infront.z = infront.w;
infront.w = g_input[in_idx];
in_idx += stride;
out_idx += stride;
__syncthreads();

// ------- update the data slice in smem -----------------------------------
if( threadIdx.y<radius ) // top and bottom apron
{
    s_data[threadIdx.y][tx] = g_input[out_idx – radius * dimx];
    s_data[threadIdx.y+20][tx] = g_input[out_idx + 16 * dimx];
}
if( threadIdx.x<radius ) // left and right apron
{
    s_data[ty][threadIdx.x] = g_input[out_idx – radius];
    s_data[ty][threadIdx.x+20] = 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] * (infront.x + behind.x + s_data[ty-1][tx]+ s_data[ty+1][tx]+ s_data[ty][tx-1]+ s_data[ty][tx+1]);
div += c_coeff[2] * (infront.x + behind.x + s_data[ty-2][tx]+ s_data[ty+2][tx]+ s_data[ty][tx-2]+ s_data[ty][tx+2]);
div += c_coeff[3] * (infront.x + behind.x + s_data[ty-3][tx]+ s_data[ty+3][tx]+ s_data[ty][tx-3]+ s_data[ty][tx+3]);
div += c_coeff[4] * (infront.x + behind.x + s_data[ty-4][tx]+ s_data[ty+4][tx]+ s_data[ty][tx-4]+ s_data[ty][tx+4]);
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
32x32 Tiles

32x32 tile is divided into upper and lower halves
- 32x16 threadblocks
- Each thread is responsible for 2 output elements

Register pressure is an issue
- Each output element requires 8 registers (z-values)
- For 32x16 threadblocks (512 threads) must use 32 or fewer registers per thread
  - Use `--maxrregcount=32` compiler flag
## Single-Pass Finite Difference Performance

Mpoints/s

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

Measured on: Tesla C1060
Multi-GPU 3D Finite Difference

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

![Diagram showing data split between two GPUs]
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);

    cudaMemcpyAsync( h_ghost_own, d_ghost_own, num_ghost_bytes, cudaMemcpyDeviceToHost, stream1 );
    launch_kernel( d_output+offset2, d_input+offset2, dimx,dimy,dimz, stream2 );
    cudaStreamSynchronize( stream1 );
    MPI_Ssendrecv( 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)