COMP 633 - Parallel Computing

Lecture 14
October 25, 2018

Cuda examples
Example 1: all-pairs n-body computation (3D)
Force calculation

- Recall simple force calculation

\[ F_i = \sum_{1 \leq j \leq N, j \neq i} f_{ij} = Gm_i \cdot \frac{\sum_{1 \leq j \leq N, j \neq i} m_j r_{ij}}{||r_{ij}||^3} \]

- Softening factor \( \varepsilon^2 > 0 \) to limit forces

\[ F_i \approx Gm_i \cdot \sum_{1 \leq j \leq N} \frac{m_j r_{ij}}{\left(||r_{ij}||^2 + \varepsilon^2\right)^{3/2}} \]

\[ A_i = \frac{F_i}{m_i} \approx G \cdot \sum_{1 \leq j \leq N} \frac{m_j r_{ij}}{\left(||r_{ij}||^2 + \varepsilon^2\right)^{3/2}} \]
Computational Tile

Figure 31-2. A Schematic Figure of a Computational Tile
Left: Evaluation order. Right: Inputs needed and outputs produced for the $p^2$ interactions calculated in the tile.
Body-body interaction

Listing 31-1. Updating Acceleration of One Body as a Result of Its Interaction with Another Body

```c
__device__ float3
body_body_interaction(float4 bi, float4 bj, float3 ai)
{
    float3 r;

    // r_ij [3 FLOPS]
    r.x = bj.x - bi.x;
    r.y = bj.y - bi.y;
    r.z = bj.z - bi.z;

    // distSqr = dot(r_ij, r_ij) + EPS^2 [6 FLOPS]
    float distSqr = r.x * r.x + r.y * r.y + r.z * r.z + EPS2;

    // invDistCube = 1/distSqr^(3/2) [4 FLOPS (2 mul, 1 sqrt, 1 inv)]
    float distSixth = distSqr * distSqr * distSqr;
    float invDistCube = 1.0f/rsqrtf(distSixth);

    // s = m_j * invDistCube [1 FLOP]
    float s = bj.w * invDistCube;

    // a_i = a_i + s * r_ij [6 FLOPS]
    ai.x += r.x * s;
    ai.y += r.y * s;
    ai.z += r.z * s;

    return ai;
}
```

use reciprocal square root rsqrt()

20 FLOPS per interaction
Evaluation of a single tile

```c
__device__ float3
tile_calculation(float4 myPosition, float3 accel)
{
    int i;
    extern __shared__ float4[] shPosition;
    for (i = 0; i < p; i++) {
        accel = body_body_interaction(myPosition,
                                      shPosition[i], accel);
    }
    return accel;
}
```

Load shared memory and synchronize at these points

Parallelism

Time
Evaluation of all tiles in a thread block

Listing 31-3. The CUDA Kernel Executed by a Thread Block with $p$ Threads to Compute the Gravitational Acceleration for $p$ Bodies as a Result of All $N$ Interactions

```c
__global__ void calculate_forces(void *devX, void *devA)
{
    extern __shared__ float4 [] shPosition;

    float4 *globalX = (float4 *)devX;
    float4 *globalA = (float4 *)devA;
    float4 myPosition;
    int i, tile;
    float3 acc = {0.0f, 0.0f, 0.0f};
    int gtid = blockIdx.x * blockDim.x + threadIdx.x;

    myPosition = globalX[gtid];

    for (i = 0, tile = 0; i < N; i += p, tile++) {
        int idx = tile * blockDim.x + threadIdx.x;
        shPosition[threadIdx.x] = globalX[idx];
        __syncthreads();
        acc = tile_calculation(myPosition, acc);
        __syncthreads();
    }
    // Save the result in global memory for the integration step.
    float4 acc4 = {acc.x, acc.y, acc.z, 0.0f};
    globalA[gtid] = acc4;
}
```

These $p$ float4 values occupy consecutive locations in device memory. The $p$ loads are coalesced and transfer at full memory bandwidth.
Execution configuration

// N bodies, N threads
int p = 256;
dim3 DimBlock(p, 1, 1); // p threads per block
dim3 DimGrid(N/p, 1); // N/p thread blocks

// p bodies in shared memory per tile evaluation
size_t SharedMemBytes = p * sizeof(Float4);

CalculateForces <<< DimGrid, DimBlock, SharedMemBytes >>>
(Posns, Accels);
Performance \((p = 256)\) GTX 8800 (2007 !)

Figure 31-6. Performance Increase as \(N\) Grows

This is about 10B interactions/sec (single precision)
Performance (p=256) V100 (2018)

- 10 timesteps

<table>
<thead>
<tr>
<th>GPU</th>
<th>n</th>
<th>SP inter/s</th>
<th>SP GFLOPS</th>
<th>DP inter/s</th>
<th>DP GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 8800</td>
<td>16384</td>
<td>10B</td>
<td>200</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>V100</td>
<td>16384</td>
<td>314B</td>
<td>6300</td>
<td>100B</td>
<td>3000</td>
</tr>
<tr>
<td>V100</td>
<td>65536</td>
<td>370B</td>
<td>7400</td>
<td>135B</td>
<td>3900</td>
</tr>
<tr>
<td>V100</td>
<td>1,048,576</td>
<td>463B</td>
<td>9300</td>
<td>161B</td>
<td>4840</td>
</tr>
</tbody>
</table>
Two more CUDA examples

• Reduction
  – Illustrating control divergence

• Matrix multiplication
  – Illustrating shmem reuse
A Parallel Sum Reduction Example
Parallel Sum Reduction

- Parallel implementation
  - halve # of active threads in each step, add two values per thread in each step
  - Takes log(n) steps for n elements, requires n/2 threads

- In-place reduction using shared memory within a block
  - The original vector of floats is in device memory
  - The shared memory is used to hold a partial sum vector
  - Each step brings the partial sum vector closer to the sum
  - The final sum will be in element 0 of the partial sum vector
  - Reduces global memory traffic due to partial sum values
A Naive Thread to Data Mapping

- Each thread is responsible for an even-index location of the partial sum vector (location of responsibility)
- After each step, half of the threads are no longer needed
- One of the inputs is always from the location of responsibility
- In each step, one of the inputs comes from an increasing distance away
A Simple Thread Block Design

- Each thread block takes 2*BlockDim.x input elements
- Each thread loads 2 elements into shared memory

```c
__shared__ float partialSum[2*BLOCK_SIZE];

unsigned int t = threadIdx.x;
unsigned int start = 2*blockIdx.x*blockDim.x;

// load first half of block data into shmem
partialSum[t] = input[start + t];

// load second half of block data into shmem
partialSum[blockDim+t] = input[start + blockDim.x+t];
```
The Reduction Steps

for (unsigned int stride = 1;
     stride <= blockDim.x; stride *= 2)
{
    __syncthreads();
    if (t % stride == 0)
        partialSum[2*t] += partialSum[2*t+stride];
}

Why do we need __syncthreads()?
Back to the Global Picture

- At the end of the kernel, Thread 0 in each thread block writes the sum of the thread block in partialSum[0] into a vector indexed by the blockIdx.x

- There can be a large number of such sums if the original vector is very large
  - The host code may iterate and launch another kernel

- If there are only a small number of sums, the host can simply transfer the data back and add them together
Some Observations on the naïve reduction kernel

- In each iteration, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition still consume execution resources

- Half or fewer of threads will be executing after the first step
  - All odd-index threads are disabled after first step
  - After the 5th step, entire warps in each block will fail the if test, poor resource utilization but no divergence
    - This can go on for a while, up to 6 more steps (stride = 32, 64, 128, 256, 512, 1024), where each active warp only has one productive thread until all warps in a block retire
Thread Index Usage Matters

- In some algorithms, one can shift the index usage to improve the divergence behavior
  - Commutative and associative operators

- Always compact the partial sums to the front locations in the partialSum[ ] array

- Keep the active threads consecutive
An Example of 4 threads

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25</td>
</tr>
</tbody>
</table>
A Better Reduction Kernel

for (unsigned int stride = blockDim.x; 
     stride > 0; stride /= 2) 
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
A Quick Analysis

– For a 1024 thread block
  – No divergence in the first 5 steps
    – 1024, 512, 256, 128, 64, 32 consecutive threads are active in each step
    – All threads in each warp either all active or all inactive
  – The final 5 steps will still have divergence
Second CUDA example

- Matrix multiplication
  - Illustratingshmem use
Matrix Multiplication

- Data access pattern
  - Each thread - a row of M and a column of N
  - Each thread block – a strip of M and a strip of N
Tiled Matrix Multiplication

- Break up the execution of each thread into phases
- so that the data accesses by the thread block in each phase are focused on one tile of M and one tile of N
- The tile is of BLOCK_SIZE elements in each dimension
Loading a Tile

- All threads in a block participate
  - Each thread loads one M element and one N element in tiled code
Phase 0 Load for Block (0,0)
Phase 0 Use for Block (0,0) (iteration 0)
Phase 0 Use for Block (0,0) (iteration 1)
Phase 1 Load for Block (0,0)
Phase 1 Use for Block (0,0) (iteration 0)
### Phase 1 Use for Block (0,0) (iteration 1)

<table>
<thead>
<tr>
<th>N₀,₀</th>
<th>N₀,₁</th>
<th>N₀,₂</th>
<th>N₀,₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>N₁,₀</td>
<td>N₁,₁</td>
<td>N₁,₂</td>
<td>N₁,₃</td>
</tr>
<tr>
<td>N₂,₀</td>
<td>N₂,₁</td>
<td>N₂,₂</td>
<td>N₂,₃</td>
</tr>
<tr>
<td>N₃,₀</td>
<td>N₃,₁</td>
<td>N₃,₂</td>
<td>N₃,₃</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>M₀,₀</th>
<th>M₀,₁</th>
<th>M₀,₂</th>
<th>M₀,₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>M₁,₀</td>
<td>M₁,₁</td>
<td>M₁,₂</td>
<td>M₁,₃</td>
</tr>
<tr>
<td>M₂,₀</td>
<td>M₂,₁</td>
<td>M₂,₂</td>
<td>M₂,₃</td>
</tr>
<tr>
<td>M₃,₀</td>
<td>M₃,₁</td>
<td>M₃,₂</td>
<td>M₃,₃</td>
</tr>
</tbody>
</table>

Shared Memory

<table>
<thead>
<tr>
<th>N₂,₀</th>
<th>N₂,₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>N₃,₀</td>
<td>N₃,₁</td>
</tr>
</tbody>
</table>

Shared Memory

<table>
<thead>
<tr>
<th>M₀,₂</th>
<th>M₀,₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>M₁,₂</td>
<td>M₁,₃</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P₀,₀</th>
<th>P₀,₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁,₀</td>
<td>P₁,₁</td>
</tr>
<tr>
<td>P₂,₀</td>
<td>P₂,₁</td>
</tr>
<tr>
<td>P₃,₀</td>
<td>P₃,₁</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P₀,₂</th>
<th>P₀,₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁,₂</td>
<td>P₁,₃</td>
</tr>
<tr>
<td>P₂,₂</td>
<td>P₂,₃</td>
</tr>
<tr>
<td>P₃,₂</td>
<td>P₃,₃</td>
</tr>
</tbody>
</table>
## Execution Phases of Toy Example

<table>
<thead>
<tr>
<th>Thread</th>
<th>Phase 0</th>
<th>Phase 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M_{0,0} )</td>
<td>( N_{0,0} )</td>
<td>( N_{0,0} )</td>
</tr>
<tr>
<td>( \downarrow ) Mds(_{0,0} )</td>
<td>( \downarrow ) Nds(_{0,0} )</td>
<td>( \downarrow ) Mds(_{0,0} )</td>
</tr>
<tr>
<td>( \text{PValue}<em>{0,0} += \text{Mds}</em>{0,0} \times \text{Nds}_{0,0} )</td>
<td>( \text{Mds}<em>{0,1} \times \text{Nds}</em>{1,0} )</td>
<td>( \text{PValue}<em>{0,0} += \text{Mds}</em>{0,0} \times \text{Nds}_{0,0} )</td>
</tr>
<tr>
<td>( \text{M}_{0,1} )</td>
<td>( \text{N}_{0,1} )</td>
<td>( \text{M}_{0,2} )</td>
</tr>
<tr>
<td>( \downarrow ) Mds(_{0,1} )</td>
<td>( \downarrow ) Nds(_{1,0} )</td>
<td>( \downarrow ) Mds(_{0,1} )</td>
</tr>
<tr>
<td>( \text{PValue}<em>{0,1} += \text{Mds}</em>{0,0} \times \text{Nds}<em>{0,1} + \text{Mds}</em>{0,1} \times \text{Nds}_{1,1} )</td>
<td>( \text{M}_{0,3} )</td>
<td>( \text{N}_{2,1} )</td>
</tr>
<tr>
<td>( \downarrow ) Mds(_{0,1} )</td>
<td>( \downarrow ) Nds(_{0,0} )</td>
<td>( \downarrow ) Mds(_{0,1} )</td>
</tr>
<tr>
<td>( \text{PValue}<em>{1,0} += \text{Mds}</em>{1,0} \times \text{Nds}<em>{0,0} + \text{Mds}</em>{1,1} \times \text{Nds}_{1,0} )</td>
<td>( \text{M}_{1,2} )</td>
<td>( \text{N}_{3,0} )</td>
</tr>
<tr>
<td>( \downarrow ) Mds(_{1,0} )</td>
<td>( \downarrow ) Nds(_{1,0} )</td>
<td>( \downarrow ) Mds(_{1,0} )</td>
</tr>
<tr>
<td>( \text{PValue}<em>{1,1} += \text{Mds}</em>{1,0} \times \text{Nds}<em>{0,1} + \text{Mds}</em>{1,1} \times \text{Nds}_{1,1} )</td>
<td>( \text{M}_{1,3} )</td>
<td>( \text{N}_{3,1} )</td>
</tr>
<tr>
<td>( \downarrow ) Mds(_{1,1} )</td>
<td>( \downarrow ) Nds(_{1,1} )</td>
<td>( \downarrow ) Mds(_{1,1} )</td>
</tr>
</tbody>
</table>

**Time**
### Execution Phases of Toy Example (cont.)

<table>
<thead>
<tr>
<th>Thread</th>
<th>Phase 0</th>
<th>Phase 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread(_{0,0})</td>
<td>(M_{0,0} \downarrow M_{d0,0} \downarrow N_{0,0})</td>
<td>(P_{Value_{0,0}} += M_{d0,0} * N_{d0,0} + M_{d0,1} * N_{d1,0})</td>
</tr>
<tr>
<td>(M_{d0,0})</td>
<td>(M_{d0,1})</td>
<td>(M_{d0,2})</td>
</tr>
<tr>
<td>thread(_{0,1})</td>
<td>(M_{0,1} \downarrow M_{d1,0} \downarrow N_{0,1})</td>
<td>(P_{Value_{0,1}} += M_{d0,0} * N_{d0,1} + M_{d0,1} * N_{d1,1})</td>
</tr>
<tr>
<td>thread(_{1,0})</td>
<td>(M_{1,0} \downarrow M_{d1,0} \downarrow N_{1,0})</td>
<td>(P_{Value_{1,0}} += M_{d1,0} * N_{d0,0} + M_{d1,1} * N_{d1,0})</td>
</tr>
<tr>
<td>thread(_{1,1})</td>
<td>(M_{1,1} \downarrow M_{d1,1} \downarrow N_{1,1})</td>
<td>(P_{Value_{1,1}} += M_{d1,0} * N_{d0,1} + M_{d1,1} * N_{d1,1})</td>
</tr>
</tbody>
</table>

**Shared memory allows each value to be accessed by multiple threads**

---

35
Barrier Synchronization

- Synchronize all threads in a block
  - __syncthreads()

- All threads in the same block must reach the __syncthreads() before any of the them can move on

- Best used to coordinate the phased execution tiled algorithms
  - To ensure that all elements of a tile are loaded at the beginning of a phase
  - To ensure that all elements of a tile are consumed at the end of a phase
Miscellaneous CUDA points

- Pointers
- Device runtime support
- Host runtime support
Variable Type Restrictions

- Pointers can only point to memory allocated or declared in global memory:
  - Allocated in the host and passed to the kernel:
    ```c
    __global__ void KernelFunc(float* ptr)
    ```
  - Obtained as the address of a global variable: `float* ptr = &GlobalVar;`

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1
Common Runtime Component

• Provides:
  – Built-in vector types
    • [u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4]
      – Structures accessed with $x, y, z, w$ fields:
        ```c
        uint4 param;
        int y = param.y;
        ```
    • dim3
      – Based on uint3
      – Used to specify dimensions

– A subset of the C runtime library supported in both host and device codes

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1
Runtime Component: Mathematical Functions

- `pow, sqrt, cbrt, hypot`
- `exp, exp2, expm1`
- `log, log2, log10, log1p`
- `sin, cos, tan, asin, acos, atan, atan2`
- `sinh, cosh, tanh, asinh, acosh, atanh`
- `ceil, floor, trunc, round`
- (more)
  - When executed on the host, a given function uses the C runtime implementation if available
  - These functions are only supported for scalar types, not vector types

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1
Host Runtime Component

• Provides functions to deal with:
  – **Device** management (including multi-device systems)
    • Initializes the first time a runtime function is called
    • A host thread can invoke device code on only one device
      – Multiple host threads required to run on multiple devices
  
  – **Memory** management
    • Device memory allocation
      – cudaMalloc(), cudaFree()
    • Memory copy from host to device, device to host, device to device
      – cudaMemcpy(), cudaMemcpy2D(), cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()
    • Memory addressing
      – cudaGetSymbolAddress()
  
  – **Error** handling

adapted from: David Kirk/NVIDIA and Wen-mei W. Hwu, Fall 2007 ECE 498AL1