CSCI-GA.3033-004
Graphics Processing Units (GPUs): Architecture and Programming

Lecture 13: Putting It All Together

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com
This Lecture

We will pick a problem, **analyze** it, and see how it can be **written** and **optimized** for GPU.

- Minimize memory access
- Minimize thread divergence
- Fully parallelized
An N-body simulation numerically approximates the evolution of a system of bodies in which each body continuously interacts with every other body.
Frames from an Interactive 3D Rendering of a 16,384-Body System
N-Body Problem

- Manifests itself in many domains: physics, astronomy, electromagnetics, molecules, etc.
- N points
- The answer at each point depends on data at all the other points
- \(O(n^2)\)
- To reduce complexity: compress data of groups of nearby points
  - A well-known algorithm to do this: Barnes Hut
Challenges of CUDA Implementation of Barnes Hut

- Repeatedly builds and traverse an irregular tree-based data structure.
- Performs a lot of pointer-chasing memory operations.
- Typically expressed recursively.
- Results in thread divergence
- Many slow uncoalesced accesses
- Must use iterations
Barnes Hut n-Body Algorithm

Divided into 3 steps

1. Building the tree - $O(n \times \log n)$
2. Computing cell centers of mass - $O(n)$
3. Computing Forces - $O(n \times \log n)$
Barnes Hut n-Body Algorithm

0. Read input data and transfer to GPU
for each timestep do {
   1. Compute bounding box around all bodies
   2. Build hierarchical decomposition by inserting each body into octree
   3. Summarize body information in each internal octree node
   4. Approximately sort the bodies by spatial distance
   5. Compute forces acting on each body with help of octree
   6. Update body positions and velocities
}
7. Transfer result to CPU and output
Barnes Hut n-Body Algorithm

0. Read input data and transfer to GPU
for each timestep do {
   1. Compute bounding box around all bodies
   2. Build hierarchical decomposition by inserting each body into octree
   3. Summarize body information in each internal octree node
   4. Approximately sort the bodies by spatial distance
   5. Compute forces acting on each body with help of octree
   6. Update body positions and velocities
}
7. Transfer result to CPU and output
Barnes Hut n-Body Algorithm

0. Read input data and transfer to GPU
for each timestep do {
    1. Compute bounding box around all bodies
    2. Build hierarchical decomposition by inserting each body into octree
    3. Summarize body information in each internal octree node
    4. Approximately sort the bodies by spatial distance
    5. Compute forces acting on each body with help of octree
    6. Update body positions and velocities
}
7. Transfer result to CPU and output

cells: Internal tree nodes
Bodies: leaves
Barnes Hut n-Body Algorithm

0. Read input data and transfer to GPU
for each timestep do {
  1. Compute bounding box around all bodies
  2. Build hierarchical decomposition by inserting each body into octree
  3. Summarize body information in each internal octree node
  4. Approximately sort the bodies by spatial distance
  5. Compute forces acting on each body with help of octree
  6. Update body positions and velocities
}
7. Transfer result to CPU and output

Calculate for each cell:
  • Center of gravity
  • Cumulative mass
0. Read input data and transfer to GPU
for each timestep do {
  1. Compute bounding box around all bodies
  2. Build hierarchical decomposition by inserting each body into octree
  3. Summarize body information in each internal octree node
  4. Approximately sort the bodies by spatial distance
  5. Compute forces acting on each body with help of octree
  6. Update body positions and velocities
}
7. Transfer result to CPU and output

Kernel 4 is not needed for correctness but for optimization.
• It is done by in-order traversal of the tree.
• Typically places spatially close bodies close together.
First Step: Data Structure

• Dynamic data structures like trees are usually built using heap objects.

• Is it the best way to go?

• Drawbacks:
  – Access to heap objects is slow
  – Very hard to coalesce objects with multiple fields

How do we deal with this?
First Step: Data Structure

• Use an array-based data structure
• To be able to coalesce:
  – use several aligned scalar arrays, one per field
• Array indices instead of pointers makes a faster code
First Step: Data Structure

- Allocate bodies at the beginning and the cells at the end of the arrays.
- Use an index of -1 as a “null pointer.”
- Advantages.
  - A simple comparison of the array index with the number of bodies determines whether the index points to a cell or a body.
  - In some code sections, we need to find out whether an index refers to a body or to null. Because -1 is also smaller than the number of bodies, a single integer comparison suffices to test both conditions.
### First Step: Data Structure

|   | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | … | t-6 | t-5 | t-4 | t-3 | t-2 | t-1 | t |
|---|----|---|---|---|---|---|---|---|---|---|---|----|-----|-----|-----|-----|-----|-----|-----|---|
| A | b0 | b1 | b2 | b3 | b4 | b5 | b6 | b7 |   |   |   |    |     |     |     |     |     |     |     |     |   |

| B |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     | c4 | c3 | c2 | c1 | c0 |

| AB | b0 | b1 | b2 | b3 | b4 | b5 | b6 | b7 |   |   |   |    |     |     |     |     |     |     |     | c4 | c3 | c2 | c1 | c0 |

**b**: body  
**c**: cell  
**t**: array length
Threads, Blocks, and Kernels

- The thread count per block is maximized and rounded down to the nearest multiple of the warp size for each kernel.
- All kernels use at least as many blocks as there are streaming multiprocessors in the GPU, which is automatically detected.
- Because all parameters passed to the kernels, such as the starting addresses of the various arrays, stay the same throughout the time step loop, we copy them once into the GPU’s constant memory.
  - This is much faster than passing them with every kernel invocation.
- Data transferred from CPU to GPU only at the beginning of the program and at the end.
- code operates on octrees in which nodes can have up to eight children.
  - It contains many loops.
  - Loop unrolling is very handy here.
Kernel 1

- computes a bounding box around all bodies
  - The root of the octree
  - has to find the minimum and maximum coordinates in the three spatial dimensions

- Implementation:
  - break up the data into equal sized chunks and assigns one chunk to each block
  - Each block then performs a reduction operation
  - The last block combine the results to generate the root node
  - reduction is performed in shared memory in a way that avoids bank conflicts and minimizes thread divergence
Kernel 2

• Implements an iterative tree-building algorithm that uses **lightweight locks**

• Bodies are assigned to the blocks and threads within a block in round-robin fashion.

• Each thread inserts its bodies one after the other by:
  – traversing the tree from the root to the desired last-level cell
  – attempting to lock the appropriate child pointer (an array index) by writing an otherwise unused value to it using an atomic operation
  – If the lock succeeds, the thread inserts the new body and release the lock
A handy group of functions to use are **atomicxxx** (must use -arch sm_11 with nvcc)

- Definition: An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.

**Synopsis of atomic function atomicOP(a,b) is typically**

```c
    t1 = *a;       // read
    t2 = t1 OP b;  // modify
    *a = t2;       // write
    return t;
```
Kernel 2

• If a body is already stored at this location, the thread:
  – creates a new cell by atomically requesting the next unused array index
  – inserts the original and the new body into this new cell
  – executes a memory fence (\texttt{threadfence}) to ensure the new subtree is visible to the rest of the cores
  – attaches the new cell to the tree
  – releases the lock.


```c
// initialize
    cell = find_insertion_point(body); // nothing is locked, cell cached for retries
    child = get_insertion_index(cell, body);
    if (child != locked) {
        if (child == atomicCAS(&cell[child], child, lock)) {
            if (child == null) {
                cell[child] = body; // insert body and release lock
            } else {
                new_cell = ...; // atomically get the next unused cell
                // insert the existing and new body into new_cell
                __threadfence(); // make sure new_cell subtree is visible
                cell[child] = new_cell; // insert new_cell and release lock
            }
            success = true; // flag indicating that insertion succeeded
        }
    }
    __syncthreads(); // wait for other warps to finish insertion
```
Kernel 3

- traverses the unbalanced octree from the bottom up to compute the center of gravity and the sum of the masses of each cell’s children
Kernel 3

- Cells are assigned to blocks and threads in a round-robin fashion.
  - Ensure load-balance
  - Start from leaves so avoid deadlocks
  - Allow some coalescing
Kernel 5

- Requires the vast majority of the runtime
- For each body, the corresponding thread traverses some prefix of the octree to compute the force acting upon this body.
Kernel 5

• Optimization: whenever a warp traverses part of the tree that some of the threads do not need, those threads are disabled due to thread divergence.
  – Make the union of the prefixes in a warp as small as possible
    • group spatially nearby bodies together → kernel 4!
• Little computation to hide memory access
  – Optimization: Allow only one thread in a warp to read the pertinent data and cache them in shared memory.
Summary of Optimizations

MAIN MEMORY

Minimize Accesses
- Let one thread read common data and distribute data to other threads via shared memory
- When waiting for multiple data items to be computed, record which items are ready and only poll the missing items
- Cache data in registers or shared memory
- Use thread throttling

Maximize Coalescing
- Use multiple aligned arrays, one per field, instead of arrays of structs or structs on heap
- Use a good allocation order for data items in arrays

Reduce Data Size
- Share arrays or elements that are known not to be used at the same time

Minimize CPU/GPU Data Transfer
- Keep data on GPU between kernel calls
- Pass kernel parameters through constant memory
Summary of Optimizations

CONTROL FLOW

Minimize Thread Divergence
- Group similar work together in the same warp

Combine Operations
- Perform as much work as possible per traversal, i.e., fuse similar traversals

Throttle Threads
- Insert barriers to prevent threads from executing likely useless work

Minimize Control Flow
- Use compiler pragma to unroll loops

LOCKING

Minimize Locks
- Lock as little as possible (e.g., only a child pointer instead of entire node, only last node instead of entire path to node)

Use Lightweight Locks
- Use flags (barrier/store and load) where possible
- Use atomic operation to lock but barrier/store or just store to unlock

Reuse Fields
- Use existing data field instead of separate lock field
## Summary of Optimizations

### HARDWARE

**Avoid Bank Conflicts**
- Control the accesses to shared memory to avoid bank conflicts

**Use All Multiprocessors**
- Parallelize code across blocks
- Make the block count at least as large as the number of streaming multiprocessors

**Maximize Thread Count**
- Parallelize code across threads
- Limit shared memory and register usage to maximize thread count
Results

CPU: 2.53GHz Xeon E5540 CPU
GPU: 1.3 GHz Quadro FX 5800
Interesting Question

Do we need GPU for non-graphics application in mobile gadgets?
Final Note:
Error Handling in CUDA
```c
__global__ void foo(int *ptr)
{
    *ptr = 7;
}

int main(void)
{
    foo<<<1,1>>>(0);
    return 0;
}
```

What will happen when you compile and execute this piece of code?
Error Handling

• In a CUDA program, if we suspect an error has occurred during a kernel launch, then we **must explicitly check** for it after the kernel has executed.

• CUDA runtime will respond to questions ... But won’t talk without asked!
cudaError_t cudaGetLastError(void);

• Called by the host
• returns a value encoding the kind of the last error it has encountered
• check for the error only after we're sure a kernel has finished executing \(\rightarrow\) don't forget kernel calls are async!
  – What will you do?
#include <stdio.h>
#include <stdlib.h>

__global__ void foo(int *ptr)
{
    *ptr = 7;
}

int main(void)
{
    foo<<<1,1>>>(0);

    // make the host block until the device is finished with foo
    cudaThreadSynchronize();

    // check for error
    cudaError_t error = cudaGetLastError();
    if(error != cudaSuccess)
    {
        // print the CUDA error message and exit
        printf("CUDA error: %s\n", cudaGetErrorString(error));
        exit(-1);
    }

    return 0;
}
Same Technique with Synchronous Calls

cudaError_t error = cudaMemcpy((void**)&ptr, 100000000000);

if(error != cudaSuccess)
{
    // print the CUDA error message and exit
    printf("CUDA error: %s\n", cudaGetErrorString(error));
    exit(-1);
}

The output will be:
CUDA error: out of memory
Rules of Thumb

• Do not use `cudaThreadSynchronize()` a lot in your code because it has a large performance penalty.

• You can enable it during debugging and disable it otherwise.

```c
#ifndef DEBUG
    cudaThreadSynchronize();
    cudaError_t error = cudaGetLastError();
    if(error != cudaSuccess)
    {
        printf("CUDA error at %s:%i: %s\n", filename, line_number, cudaGetErrorString(error));
        exit(-1);
    }
#endif
```

If debugging, compile with:
```bash
    $ nvcc -DDEBUG mycode.cu
```
Conclusions

• When considering your problem:
  – Pick your algorithm
  – Choose the data structure
  – Try to make as many threads and blocks as possible busy
  – Know your hardware
  – Tweaks are inevitable
  – Correctness, performance, and power.