Scientific Computing on Graphics Processing Units

Nicholas Frontiere
ANL/University of Chicago
ATPESC
Overview

- GPU vs. CPU
- CUDA vs. OpenCL (Briefly)
- OpenCL execution and memory framework
- GPU Hardware
- GPU Coding Obstacles and Solutions
  - Lock in Step execution (divergent if’s)
  - Memory Latency
  - Coalesced Memory
  - Bank Conflicts
- N-Body Example
- Conclusion
GPU vs. CPU

*D. Kirk & W. Hwu 2010*
CPU

- Follows the “multicore” design of a microprocessor
  - Attempt to increase the speed of sequential programs.
  - Example Intel i7 processor.

- Optimized to handle out of order execution

- Retains multilevel cache for quick memory access

- Implements sophisticated branch prediction

- Multiple cores allow for increased multi-tasking as well as threading
GPU

- Follows the “many-core” design of a microprocessor
  - Maximize throughput of parallel algorithms.
- Typically the number of cores doubles with each new generation
  - Same is true for CPUs, yet GPU’s have many many more cores.
- Throughput of Single Precision has increased dramatically

*http://docs.nvidia.com*
Question

Would you rather outsource to a
Grad Student
Capable Contractor

High Latency
Low Throughput

Low Latency
Good Throughput
Question

BUT what about MORE grad students?
BUT what about MORE grad students?

Do NOT forget Amdahl’s law

Ok Latency
High Throughput

Low Latency
Good Throughput
Amdahl’s Law

- $n$ – Number of Threads
- $T$ – Execution Time
- $F$ – Serial Fraction

Speedup = \[
\frac{T(1)}{T(n)} = \frac{1}{F + \frac{(1-F)}{n}}
\]
Take Away

Completely limited by the Serial Fraction!

- Examples of GPU accelerated code:
  - Matrix multiplication, Graphics, Tabular applications, Visual Reduction, etc.
OpenCL vs. Cuda

- Both languages capable of executing GPU kernels.
- CUDA is vendor dependent (Nvidia GPUs)
- OpenCL can run on many different heterogeneous platforms (CPU, GPU, DSP, etc)
- CUDA is more mature and as a result has highly optimized libraries
- OpenCL would be considered a “lower level” language and thus harder to code.

Which to choose?
- Depends on what you want to do, what platforms you want to use, and the targeted users.
OpenCL Platform Model

- Host code (CPU)
  - Device Queries and Platform Setups (allows one to use multiple devices)
  - Push/pull memory to/from device (GPU)
  - Compile and Launch Kernel(s)
  - Typically performs the branched logic of the application

- Kernel Code (GPU)

```c
__kernel void addTwoArrays(__global float * arr, __global float* arr2, __global float * return)
{
    return[get_global_id(0)]=arr[get_global_id(0)] + arr2[get_global_id(0)];
}
```
// Create an OpenCL context on first available platform
context = CreateContext();

// Create a command-queue on the first device available on the created context
commandQueue = CreateCommandQueue(context, &device);

// Create OpenCL program from HelloWorld.cl kernel source
program = CreateProgram(context, device, "HelloWorld.cl");
// Create OpenCL kernel
kernel = clCreateKernel(program, "hello_kernel", NULL);

// Set the kernel arguments (result, a, b)
errNum = clSetKernelArg(kernel, 0, sizeof(cl_mem), &memObjects[0]);
errNum |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &memObjects[1]);
errNum |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &memObjects[2]);
size_t globalWorkSize[1] = { ARRAY_SIZE };
size_t localWorkSize[1] = { 1 };

// Queue the kernel up for execution across the array
errNum = clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
                                 globalWorkSize, localWorkSize,
                                 0, NULL, NULL);

// Read the output buffer back to the Host
errNum = clEnqueueReadBuffer(commandQueue, memObjects[2], CL_TRUE,
                              0, ARRAY_SIZE * sizeof(float), result,
                              0, NULL, NULL);
OpenCL Kernel Execution

- Kernels get executed by threads or “work items.” Each item is assigned a “global index (id)”

- These work items are collected as “work groups,” and assigned a “group id” and “local id”

- These id’s allow the kernel code to perform thread, group, or global specific tasks.

<table>
<thead>
<tr>
<th>Local Id</th>
<th>Global Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0)</td>
<td>(0,0)</td>
</tr>
<tr>
<td>(0,1)</td>
<td>(0,1)</td>
</tr>
<tr>
<td>(0,0)</td>
<td>(0,2)</td>
</tr>
<tr>
<td>(0,1)</td>
<td>(0,3)</td>
</tr>
<tr>
<td>(0,0)</td>
<td>(0,4)</td>
</tr>
<tr>
<td>(0,1)</td>
<td>(0,5)</td>
</tr>
<tr>
<td>(0,0)</td>
<td>(0,6)</td>
</tr>
<tr>
<td>(0,1)</td>
<td>(0,7)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Group Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
</tr>
<tr>
<td>4 5 6 7</td>
</tr>
</tbody>
</table>
OpenCL Memory Hierarchy

- **Global Memory**: Available to all work items; high latency fetching (hundreds of cycles).
- **Constant Memory**: Read only memory, low latency.
- **Local Memory**: Available to a group of threads, fast fetching.
- **Private Memory**: Available to each thread.
- **Host (CPU) Global Memory**: Not accessible to device threads. Needs to be pushed by host code to device. High latency.

---

**Diagram**:

- **Host**: Contains Global Memory, not accessible to device.
- **Device**: Includes Local Memory, Global Memory/Constant Memory Cache, Private Memory for Work Items 1 to N within Work Groups 1 to M.

---

**Diagram Elements**:

- Private memory
- Work Item 1
- Work Item N
- Work Group 1
- Work Group M
- Local Memory
- Global Memory/Constant Memory Cache
Each Work-Item (thread) is executed on a Stream Processor (SP)

SPs are located on one Stream Multiprocessor (SM or SMX)
  - Work-Groups are executed on SM’s, where local memory is provided.

Warp Schedulers execute threads of Work-groups on the SM’s; a common optimization goal is to execute as many “warps” on each SM as possible.
GPU Execution Model (SIMT)

- Work Groups are executed via 32 thread launches (aka Warps)
- Warps follow Single Instruction Multiple Threading (i.e. all threads in a warp perform the SAME instruction); Optimization implications, will come back to this.
- Multiple warps can be executed concurrently on the same SM, aka “waves.” Keplar GPU’s can schedule 4 warps concurrently. However the maximum number of warps will not always launch; Depends on memory, number of threads per warp, number of threads per group, etc.
- Tails: If groups are not divisible by warp size, can affect performance.
## GPU Hardware

<table>
<thead>
<tr>
<th>Feature</th>
<th>FERMI GF100</th>
<th>FERMI GF104</th>
<th>KEPLER GK104</th>
<th>KEPLER GK110</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
</tr>
<tr>
<td>Threads / Warp</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max Warps / Multiprocessor</td>
<td>48</td>
<td>48</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Max Threads / Multiprocessor</td>
<td>1536</td>
<td>1536</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Max Thread Blocks / Multiprocessor</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>32-bit Registers / Multiprocessor</td>
<td>32768</td>
<td>32768</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers / Thread</td>
<td>63</td>
<td>63</td>
<td>63</td>
<td>255</td>
</tr>
<tr>
<td>Max Threads / Thread Block</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Shared Memory Size Configurations (bytes)</td>
<td>16K</td>
<td>16K</td>
<td>16K</td>
<td>16K</td>
</tr>
<tr>
<td></td>
<td>48K</td>
<td>48K</td>
<td>32K</td>
<td>32K</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>48K</td>
<td>48K</td>
</tr>
<tr>
<td>Max X Grid Dimension</td>
<td>$2^{16-1}$</td>
<td>$2^{16-1}$</td>
<td>$2^{32-1}$</td>
<td>$2^{32-1}$</td>
</tr>
<tr>
<td>Hyper-Q</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic Parallelism</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
GPU obstacle: Shortage of Memory

- Copying memory from the host CPU to GPU is a necessary step in all GPU kernel applications.

- Unfortunately, GPUs can only store a couple GBs of memory in total; even state-of-the-art Kepler can only hold around 8 GBs. Many applications require more, and as a result employ continuous reading and writing to the GPU. This can typically result in transfer latency performance hits.

- Possible solution: Simultaneously copy memory to the GPU while performing calculations on the previous memory transfer.
Lock-Step Execution

- Simple example is a divergent IF statement:

```c
if(get_local_id(0) < 4) {
    Do something
} else {
    Do something else
}
```

- SIMT ensures that when a warp of threads is launched for a work-group and encounters the above statement, both branches are executed (BAD).

- Two Solutions:
  - A) DON’T DO IT!
  - B) Make the branched logic modulo warp size.
  - Regardless should play around with group size
Memory Latency

- Fetching global memory requires many latency cycles (~ hundreds), and a result is one of the biggest performance hits.
- Local Memory on the other hand has much less latency cycles (~ tens) but can have bank conflicts (described later).
- Solutions:
  - A) Hide latency with arithmetic calculation; while threads are waiting for a memory transfer other warps can be launched to do calculations. Depends on algorithm.
  - B) Do one copy from global to local memory and use the local memory speed to distribute the data. Can make use of Coalesced memory. NOTE: Global memory has GBs of data, whereas the local memory per SM has KBs. Very important to proceed this way if original fetch was not coalesced.

```c
__kernel void addTwoArrays(__global float * arr, __global float* arr2,
                          __global float * return)
{
    return[get_global_id(0)] = arr[get_global_id(0)]
                 + arr2[get_global_id(0)];
}
```
If memory is accessed non-contiguously, memory fetches will be performed sequentially (BAD if from global memory).

If desired memory fetches are coalesced, the GPU can perform them all at once (modulo half warp size).

Newly Allowed
Bank Conflicts

- To avoid multiple global memory latencies, one can copy data to Local Memory for quick access. However, Local Memory is fetched with banks.
- Banks contain 4 bytes (Fermi) or 8 bytes (Kepler) of memory.
- GPUs typically contain 32 banks per SM
- If threads access different memory elements, then all fetches occur at maximum speed (GOOD). Otherwise, fetches are sequential (BAD). Exception: Broadcast to all threads is fast, can be very powerful.
NVIDIA OpenCL Visual Profiler

- Can profile kernel execution time, as well as host data transfer time.

- Can analyze memory bandwidth and instruction issue rate.

- Can report number of coalesced loads/stores

- Occupancy
  - Ratio of active warps per SM to maximum allowed.
  - Very informative measure of performance.
Exemplar: Short Range Force Solver

- Our N-body PM solver can resolve forces to ~ 3 grid units. We then require a short range solver to increase the resolution.

- A simple approach is to perform a brute force $O(N^2)$ nearest neighbor calculation (within radius of 3 cells) utilizing an accelerator such as a GPU (The $P^3M$ Method).

- One could also use a tree method to reduce computation. We currently have employed such an algorithm, but it is not currently accelerated.

- The Brute Force method is a simple algorithm which combined with the GPU performance enhancement techniques discussed has proven to be a factor of 4-5 faster than the CPU tree code.

- NOTE: GPU code runs at approximately the same speed most redshift.
imagine a cube of data

divide it into slabs
imagine a GPU
Keep Repeating
Keep Repeating
Optimization checklist:

- **Memory Shortage:** ✓
  - Algorithm only requires slabs of data, not the entire cube.

- **Lock-step execution:** ✓
  - As all particles in a group (chaining mesh) perform the same instruction -- calculate force from neighboring meshes -- there is no divergent logic within a work group.

- **Global Memory Latency:** ✓
  - Each thread caches one particle from neighboring mesh into local memory; (thus only one fetch per particle per group not per thread!)

- **Coalesced Memory Fetching?** ✓
  - Yes. As particle order does not matter for each mesh bin calculation, each thread can do a local cache sequentially, further reducing latency.

- **Bank Conflicts:** ✓
  - Use broadcast from local memory as group of threads need to fetch the same particles.
Conclusion

- GPU acceleration gains are completely determined by serial fraction of the algorithm.
- OpenCL allows one to use any heterogeneous platform vs. CUDA which is a more mature but vendor specific language.
- **Multiple GPU specific considerations**
  - Memory Storage
  - Lock-step execution
  - Global memory latency
  - Coalesced memory fetching
  - Bank Conflicts
- Profilers exist to aid you in determining performance
- Question? nfrontiere@gmail.com
OpenACC

- Similar to OpenMP, utilized directives
- Perhaps a good first step toward attempting acceleration.
- With every higher level language, one loses sophistication
- Example: Matrix Multiplication

```c
!$acc kernels
do k = 1,n1
  do i = 1,n3
    c(i,k) = 0.0
  do j = 1,n2
    c(i,k) = c(i,k) + a(i,j) * b(j,k)
  enddo
enddo
enddo
!$acc end kernels
```