AMath 483/583 — Lecture 24 — May 20, 2011

Today:

- The Graphical Processing Unit (GPU)
- GPU Programming

Today's lecture developed and presented by Grady Lemoine

References:

Andreas Kloeckner's High Performance Scientific Computing course at NYU:
http://cs.nyu.edu/courses/fall10/G22.2945-001/lectures.html

The Khronos Group's OpenCL page:
http://www.khronos.org/opencl/

Notes:

What's a GPU?

- GPU stands for Graphics Processing Unit (a.k.a. graphics card)
- Many models, not all suited to scientific computing
- Performance improvements driven by PC gaming market
- GPGPUs (General-Purpose GPUs) developed only in the past few years
- GPUs are not suited for every task, but what they can do, they do *very* well
    - Sometimes 10x speedup over CPU, sometimes more

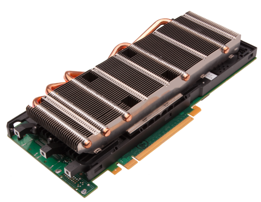


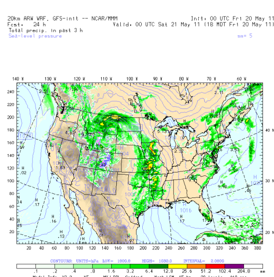
Photo credit: nVidia

Notes:

Some GPU application areas

GPUs are currently being used in:

- Fluid dynamics
- Atmospheric science
- Petroleum exploration
- Computational finance
- Medical imaging
- X-ray diffraction analysis
- Molecular dynamics
- ... and many other fields



NCAR WRF model, partially GPU-accelerated

Credit: NCAR

Notes:

Where is a GPU?

- So far in this class we've just talked about CPU and RAM
- GPU is (usually) a separate entity
- Two broad types of GPU:
  - Integrated: part of CPU or supporting chipset, uses same RAM pool as CPU
  - Discrete: separate chip or card, connected to chipset by I/O bus, often has own RAM
- Integrated GPUs are generally less powerful
- GPUs for HPC are usually extremely powerful discrete models

Notes:

GPU pros and cons

- Why should I use a GPU?
  - Very high aggregate computation rate
  - Low power consumption relative to work done (good performance-per-watt)
    - High-end GPUs use more power than high-end CPUs, but perform *much* more computation
- Why should I *not* use a GPU?
  - Massively parallel hardware – no good for inherently serial computations
  - More complex to program

Notes:

Differences between CPUs and GPUs

CPUs:

- Make a few threads run fast individually
- Have a few powerful cores
- Reduce the need for the programmer to micromanage

GPUs:

- Make many threads run fast in aggregate
- Have many weak "cores"
- Give the programmer greater control

Notes:

Differences between CPUs and GPUs

- How to evolve from a CPU to a GPU:
  1. Remove CPU parts used to improve single-thread performance (caches, instruction reordering, branch predictor, etc.)
  2. Add more cores in the space freed up
  3. Assume many cores using same instruction stream, so share instruction decoding across multiple ALUs (Arithmetic-Logical Units)
     - Results in Single Instruction, Multiple Data (SIMD) model – a bit different from SPMD model of OpenMP and MPI
  4. Add more cores in the space freed up
  5. Reduce clock speed, to reduce power consumption and allow *even more* cores
- May end up with dozens of instruction streams, each acting on 8+ data items at once

Notes:

Branches with SIMD

- Problem: What happens when an instruction stream has a conditional that goes different ways for different data?
  - Each group of ALUs must all execute the same instructions, but those instructions might be wrong for some ALUs
- Solution: Cores for which the condition is true and those for which it's false execute separately
- Warning: Can reduce performance – some ALUs idle while waiting for the other part of the branch

Code	ALU 1	ALU 2	ALU 3	ALU 4	ALU 5	ALU 6
`if (x >= 0) then`	T	T	F	T	T	T
`x2 = x`	⇓	⇓	⊗	⇓	⇓	⇓
`else`						
`x2 = -x`	⊗	⊗	⇓	⊗	⊗	⊗
`end if`						

Notes:

Memory latency (yet again)

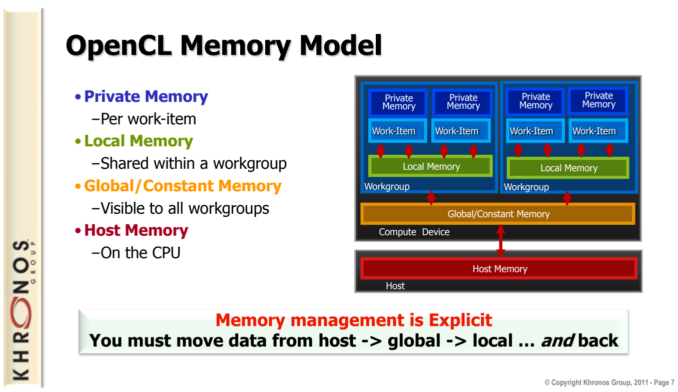
- Problem: Memory still has a long latency, and we've just removed the cache hardware that helped us fight that...
- Solution: Hide latency by queueing many more threads than we can run
- When thread 1 stalls for a memory request, thread 2 can execute while it waits
- When thread 2 stalls, thread 3 can execute while it waits
- When thread 3 stalls...
- Eventually thread 1's request finishes, and it can run again once the current thread stalls
- Requires extra context-switching hardware, but cheaper than the cache it replaced

Notes:

GPU memory hierarchy

- Discrete GPUs have their own on-board RAM
  - Provides working space
  - Saves using slow I/O bus to main memory
- They also have their own fast “working memory”
  - Similar to cache on CPUs, but smaller
  - Private to each group of ALUs
- Unlike with CPUs, program manages data transfer explicitly

Notes:



(Credit: Khronos Group)

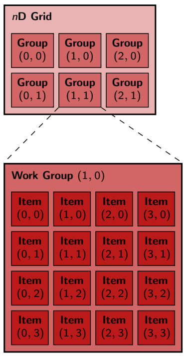
Notes:

How to program for a GPU

- GPU hardware is very different from CPU hardware
- GPU programming is pretty different too
- Current recommended language: OpenCL
  - “Open Computing Language”
  - Support from all major GPU and CPU manufacturers
  - Coordinated by the Khronos Group (non-profit industry consortium)
  - Similar to C
- OpenCL program consists of two parts:
  1. Main program (running on CPU)
  2. One or more “kernels” (running on GPU)

Notes:

OpenCL: Execution Model



- Two-tiered Parallelism
  - Grid = $N_x \times N_y \times N_z$ work groups
  - Work group = $S_x \times S_y \times S_z$ work items
  - Total: $\prod_{i \in \{x,y,z\}} S_i N_i$ work items
- Abstraction of core/SIMD lane HW concept
- Comm/Sync only within work group
- Grid/Group ≈ outer loops in an algorithm
- Device Language: `get_{global,group,local}_{id,size}(axis)`

GPU Architecture (recap) Programming GPUs

(Credit: Andreas Kloeckner, Courant Institute, NYU)

Notes:

OpenCL: Main program

- Runs on the CPU
- Handles the "administrative stuff":
  - Does various initialization chores (similar to `MPI_INIT`, `OMP_SET_NUM_THREADS`, etc.)
  - Specifies how to decompose the problem into a grid format
  - Compiles the kernel(s) (done at run time for OpenCL!)
  - Transfers data to/from the GPU
- Runs the kernel(s)
- Also does whatever can't or shouldn't be done on the GPU
  - Input/Output
  - Inherently serial computations

Notes:

OpenCL: Kernel(s)

- Run on the GPU (or CPU, for OpenCL)
- Typically simple
- Applied successively to every element of a buffer/array
  - Kernel is like the body of a Fortran `do` loop
  - Calling framework takes care of the surrounding "`do`/`end do`" equivalent
- For good performance, should pay attention to local vs. global memory (similar to CPU cache locality)
- Also best to avoid transferring data between main memory and GPU more than necessary – I/O bus is slow

Notes:

OpenCL Example Program Sketch

```
// Header files omitted
int main() {
    cl_context ctx; cl_command_queue queue; cl_int status;
    create_context_on("NVIDIA", NULL, 0, &ctx, &queue, 0);

    // Create array in main memory
    float a[10000];
    for (size_t i = 0; i < 10000; ++i) a[i] = i;

    // Allocate memory on GPU, transfer data to GPU
    cl_mem buf_a = clCreateBuffer(ctx, CL_MEM_READ_WRITE, 10000*sizeof(float), 0, &status);
    CALL_CL_GUARDED(clEnqueueWriteBuffer,
        (queue, buf_a , CL_TRUE, 0, 10000*sizeof(float), a, 0, NULL, NULL));

    // Define and compile kernel
    char* knl_text =
" __kernel void twice(__global float *a) { a[get_global_id(0)] *= 2.0; }";
    cl_kernel knl = kernel_from_string(ctx , knl_text , "twice", NULL);

    // Run on GPU
    SET_1_KERNEL_ARG(knl, buf_a);
    size_t gdim[] = { 10000 };    // Dimensions of global grid
    size_t ldim[] = { 1 };        // Dimensions of local grid
    CALL_CL_GUARDED(clEnqueueNDRangeKernel,
        (queue, knl, 1, NULL, gdim, ldim, 0, NULL, NULL));

    // Cleanup and error-checking omitted
}
```

(Adapted from Andreas Kloeckner)

Notes:

Summary

- GPUs are a major new resource in scientific computing
- They work very differently from CPUs
- Using them can be a little involved . . .
- . . . but if your problem is suitable, the results can be worth it

Notes:

References

- Andreas Kloeckner's High Performance Scientific Computing course at NYU: **http://cs.nyu.edu/courses/fall10/G22.2945-001/lectures.html**
- The Khronos Group's OpenCL page: **http://www.khronos.org/opencl/**

Notes: