High Performance Computing on GPUs using NVIDIA CUDA

Slides include some material from GPGPU tutorial at SIGGRAPH2007: http://www.gpgpu.org/s2007
Outline

• Motivation
• Stream programming
  – Simplified HW and SW model
  – Simple GPU programming example
• Increasing stream granularity
  – Using shared memory
  – Matrix multiplication
• Improving performance
• Some real life example
Disclaimer

This lecture will discuss GPUs from the Parallel Computing perspective since I am NOT an expert in graphics hardware.
Motivation: Computational Power

• GPUs are fast...
  – 3.0 GHz Intel Core2 Quad (QX6850):
    • Computation: 96 GFLOPS peak
    • Memory bandwidth: 21 GB/s peak
    • Price: $1100 (chip)
  – NVIDIA GeForce 8800 GTX:
    • Computation: 330 GFLOPS observed
    • Memory bandwidth: 55.2 GB/s observed
    • Price: $550 (board)

• GPUs are getting faster, faster
  – CPUs: 1.4× annual growth
  – GPUs: 1.7×(pixels) to 2.3× (vertices) annual growth

Courtesy Kurt Akeley, Stanford GPUBench project
Why GPUs-II

![Graph showing performance over time for AMD (GPU), NVIDIA (GPU), and Intel (CPU). The x-axis represents years from 2000 to 2008, and the y-axis represents GFLOPS. The graph includes markers for dual-core and quad-core.]
Is it a miracle? NO!

- Architectural solution prefers parallelism over single thread performance!
- Example problem – I have 100 apples to eat
  1) “high performance computing” objective: optimize the time of eating one apple
  2) “high throughput computing” objective: optimize the time of eating all apples
- The 1st option has been exhausted!!!
- Performance = parallel hardware + scalable parallel program!
Why not in CPUs?

- Not applicable to general purpose computing
- Complex programming model
- Still immature
  - Platform is a moving target
    - Vendor-dependent architectures
    - Incompatible architectural changes from generation to generation
  - Programming model is vendor dependent
    - NVIDIA – CUDA
    - AMD(ATI) – Close To Metal (CTM)
    - INTEL (LARRABEE) – nobody knows

Mark Silberstein, Technion
Simple stream programming model
Generic GPU hardware/software model

- Massively parallel processor: many concurrently running threads (thousands)
- Threads access global GPU memory
- Each thread has limited number of private registers
- Caching: two options
  - Not cached (latency hidden through time-slicing)
  - Cached with unknown cache organization, but optimized for 2D spatial locality
- Single Program Multiple Data (SPMD) model
  - The same program, called kernel, is executed on the different data
How we design an algorithm

- Problem: compute product of two vectors \( A[10000] \) and \( B[10000] \) and store it in \( C[10000] \)

- Think data-parallel: **same set** of operations (*kernel*) applied to multiple data chunks
  - apply fine grain parallelization (**caution here! - see in a few slides**)
    - Thread creation is cheap
    - The more threads the better

- Idea: one thread multiplies 2 numbers
How we implement an algorithm

- **CPU**
  1. Allocate three arrays in GPU memory
  2. Copy data CPU -> GPU
  3. Invoke kernel with 10000 threads, pass ptrs to the arrays from the step 1.
  4. Wait until complete and copy data GPU->CPU

- **GPU**
  - Get my threadID
  - \( C[\text{threadId}] = A[\text{threadId}] \times B[\text{threadId}] \)
Any performance estimates?

- Performance criterion - GFLOP/s

- Key issue: memory or CPU bound?
  - We can fully utilize GPUs only if the data can be made available in the ALUs on time!!!
  - Otherwise – at most the number of operations which can be performed on the available data.

- **Arithmetic intensity**: number of FLOPs per memory access
  - Performance = min[MemBW*A, GPU HW]

- For example: A=1/3, GPU HW=345GFLOP/s, MemBW=22GFloat/s: Performance = ~7GFLOP/s ~2% utilization!!!
Enhanced model
Generic model - limitations

• Best used for streaming-like workloads
  • Embarrassingly parallel: running algorithm on multiple data
  • Low data reuse
    – High number of operations per memory access (arithmetic intensity) to allow latency hiding
• Low speedups otherwise
  – Memory bound applications benefit from higher memory bandwidth, but result in low GPU utilization
NVIDIA CUDA extension: Fast on-chip memory

Without shared memory

With shared memory

Adopted from CUDA programming guide

Mark Silberstein, Technion
Changed programming model

- Low latency/high bandwidth memory shared between threads in one **thread block** (up to 512 threads).

- Programming model: stream of thread blocks

- Challenge: optimal structuring of computations to take advantage of fast memory

Mark Silberstein, Technion
Thread block

- **Scheduling of threads in a TB**
  - *Warp*: thread in one warp are executed concurrently (well... Half-warp in lock-step, half-warps are swapped)
  - Warps MAY be executed concurrently. Otherwise – according to the thread ID in the warp

- **Thread communication in a TB**
  - Shared memory
  - TB-wide synchronization (barrier)
Multiple thread blocks

- Thread blocks are completely independent
  - No scheduling guarantees
- Communication – problematic
  - Atomic memory instructions available
  - Synchronization is dangerous: may bring to deadlock if not enough hardware
- Better think of thread blocks as a STREAM
Breaking the “stream” hardware abstraction

• Processors are split into groups
  – Each group (*multiprocessor -MP*) has fast memory and set of registers shared among all processors
    • NVIDIA GTX8800: 128 6-thread processors per MP, shared memory size: 16KB, 8192 4B registers, 16 MPs per video card

• Thread block is scheduled on a SINGLE MP, why?
Thread blocks and MP

- Different thread blocks may be scheduled (via preemption) on the same MP to allow better utilization and global memory latency hiding

- **PROBLEM:** shared memory and register file should be large enough to allow preemption!

- Determining the best block size is kernel-dependent!
  - More threads per block – less blocks can be scheduled – may lead to lower throughput
  - Fewer threads per block – more blocks, but less registers/shared memory per block
Matrix multiplication example

- Product of two $N \times N$ matrices
- Streaming approach
  - Each thread computes single value of the output
  - Is it any good??? No!
    - Arithmetic Intensity $= \frac{2N-1}{2N+1} \Rightarrow$ Max performance: 22GFLOP/s (instead of 345!!!)
  - Why? $O(N)$ data reuse is NOT utilized
  - Optimally: Arithmetic intensity $= \frac{(2N-1)}{(2N/N + 1)} = O(N) \Rightarrow$ CPU bound!!!!!
Better approach (borrowed from Mark Harris slides)

**Example: Matrix Multiplication**

- Much better to block the computation
  - each block computes $M \times M$ sub-matrix
  - stage sub-matrices of A and B in shared memory
  - each element of A and B loaded $N/M$ times from global memory

- Much less bandwidth

- Much better balance of work to bandwidth
Generalized approach to shared memory

• Think of it as a distributed user-managed cache

• When regular access pattern - better to have implicit cache management
  • In matrix product we know “implicitly” that the access is sequential

• Less trivial for irregular access pattern -> implement REAL cache logic interleaved into the kernel
  • devise cache tag, handle misses, tag collisions, etc,
  • analyze it just like regular cache

• Sorry guys, self reference here: “Efficient sum-product computation on GPUs”
CUDA Tool Chain

- Standard Libraries: FFT, BLAS, ...
- Integrated CPU and GPU C Source Code
- NVIDIA C Compiler
- NVIDIA Assembly for Computing
- CPU Host Code
- CUDA Runtime & Driver
- Profiler
CUDA at glance

- Compiler
  - Handles language extensions
  - Compiles GPU code into HW-independent intermediate code (read PTX and NVCC spec to know more)

- Runtime
  - GPU memory management/transfer, CPU->GPU control, etc...

  Supports emulation mode for debugging

- NO PROFILER YET (expecting soon)

- Driver
  - JIT compilation and optimizations, mapping onto graphics pipeline, (sign NDA to know more.). Watchdog problem for kernels over 5 seconds (not on LINUX without X!!)

- HW support (only in new GPUs)
Sample code walkthrough: from NVIDIA User guide

(see http://developer.nvidia.com/object/cuda.html)
Few performance guidelines
Check SIGGRAPH tutorial for more

- **Algorithm:** data parallel + structure to use shared memory (exploit the data reuse!)
- **Estimate upper bounds!**
- **Coherent memory accesses!**
- **Use many threads**
- **Unroll loops!**
  - Use fast version of integer operations or avoid them altogether
  - Minimize synchronization where possible
- **Optimize TB size where possible.** (occupancy: # warps per MP as a possible measure) in conjunction with register and shared memory use
- **Know to use constant and texture memory**
- **Avoid divergence of a single warp**
- **Minimize CPU<->GPU memory transfers**
Real life application: genetic linkage analysis

- Used to find disease provoking genes
- Can be very demanding
- Our research: map computations onto inference in Bayesian networks
- One approach: parallelize to use thousands of computers worldwide (see “Superlink-online”)
- Another approach: parallelize to take advantage of GPUs
Method

● Parallelize sum-product computations
  – Generalization of matrix chain product
  – More challenging data access pattern
● Shared memory as a user-managed cache
  – Explicit caching mechanism is implemented
Results

- Performance comparison: NVIDIA GTX8800 vs. Single core of Intel Dual Core 2, 3GHz, 2M L2
- Speedup up to ~60 on synthetic benchmarks (57GFLOPs peak vs. ~0.9GFLOP peak)
- Speedup up to 12-15 on real Bayesian networks
- Speedup up to 700(!) if log scale used for better precision
- More on this: see my home page
Conclusion

• GPUs are great for HPC
• CUDA rocks!
  – Short learning curve
  – Easy to build proof of concepts
• GPUs seem to be the “next” many-cores architecture
  – See “The Landscape of Parallel Computing Research: A View from Berkeley”
• Go and try it!
Resources

• http://www.gpgpu.org


• CUDA forums @NVIDIA: http://forums.nvidia.com/index.php?showforum=62