GPUs: The Hype, The Reality, and The Future

David Black-Schaffer
Assistant Professor, Department of Information Technology
Uppsala University

Today

1. The hype
2. What makes a GPU a GPU?
3. Why are GPUs scaling so well?
4. What are the problems?
5. What’s the Future?

THE HYPE

How Good are GPUs?

Real World Software

- Press release 10 Nov 2011:
  - “NVIDIA today announced that four leading applications... have added support for multiple GPU acceleration, enabling them to cut simulation times from days to hours.”
- GROMACS
  - 2-3x overall
  - implicit solvers 10x, PME simulations 1x
- LAMPS
  - 2-8x for double precision
  - Up to 15x for mixed
- QMCPACK
  - 3x

2x is AWESOME! Most research claims 5-10%.

GPUs for Linear Algebra
GPUs by the Numbers (Peak and TDP)

<table>
<thead>
<tr>
<th>Watts</th>
<th>GFLOP</th>
<th>Bandwidth</th>
<th>GFLOP/W</th>
<th>Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel 3960X</td>
<td>Nvidia GTX 580</td>
<td>AMD 6970</td>
<td>Intel 32nm vs. 45nm</td>
<td>36% smaller per transistor</td>
</tr>
</tbody>
</table>

Efficiency (DP, peak)

- GPUs are enormously more efficient designs for doing FLOPs

Efficiency in Perspective

- Nvidia says that ARM + Laptop GPU → 7.5GFLOPS/W
- LINPACK: Green 500 #2 Blue Gene/Q
- LINPACK: Green 500 #6 GPU/CPU

Intel’s Response

- Larabee
  - Manycore x86-"light" to compete with Nvidia/AMD in graphics and compute
  - Didn’t work out so well (despite huge fab advantages — graphics is hard)
- Repositioned it as an HPC co-processor
  - Knight’s Corner
    - 1TF double precision in a huge expensive) single 22nm chip
    - At 300W this would beat Nvidia’s peak efficiency today (40nm)

Show Me the Money

- AMD Q3
- Nvidia Q3
- Intel Q3

WHAT MAKES A GPU A GPU?
**GPU Characteristics**

- **Architecture**
  - Data parallel processing
  - Hardware thread scheduling
  - High memory bandwidth
  - Graphics rasterization units
  - Limited caches (with texture filtering hardware)

- **Programming/Interface**
  - Data parallel kernels
  - Throughput-focused
  - Limited synchronization
  - Limited OS interaction

**GPU Innovations**

- **SMT (for latency hiding)**
  - Massive numbers of threads
  - Programming SMT is far easier than SIMD

- **SIMT (thread groups)**
  - Amortize scheduling/control/data access
  - Warps, wavefronts, work-groups, gangs

- **Memory systems optimized for graphics**
  - Special storage formats
  - Texture filtering in the memory system
  - Bank optimization across threads

- **Limited synchronization**
  - Improves hardware scalability

(They didn’t invent any of these, but they made them successful.)

---

**Lots of Room to Grow the Hardware**

- **Simple cores**
  - Shared registers
  - No branch prediction
  - In-order

- **Simple memory system**
  - Limited caches
  - Noncoherence
  - Split address space

- **Simple control**
  - Limited context switching
  - Limited processes

- **But, they do have a lot...**
  - Unlimited thread scheduling

**Why are GPUs Scaling So Well?**

- **Room in the hardware design**
- **Scalable software**

*They’re not burdened with 30 years of cruft and legacy code...*  

...lucky them.

---

**“Nice” Programming Model**

- All GPU programs have:
  - Explicit parallelism
  - Hierarchical structure
  - Restricted synchronization
  - Data locality
    - Inherent in graphics
    - Enforced in compute by performance
  - Latency tolerance

- Easy to scale!
WHERE ARE THE PROBLEMS?

- Moving data to the GPU is slow...
- Moving data from the GPU is slow...
- Moving data to/from the GPU is really slow.
- Limited data storage
- Limited interaction with OS

Amdahl’s Law

- Always have serial code
  - GPU single threaded performance is terrible
- Solution: Heterogeneity
  - A few fat latency-optimized cores
  - Many thin throughput-optimized cores
  - Plus hard-coded accelerators

WHERE ARE THE PROBLEMS?

- No ability to launch kernels from within kernels
- No effective support for task-based parallelism
- Very expensive to synchronize with the CPU (data transfer)
- No effective support for task-based parallelism
- No ability to launch kernels from within kernels

Code that Doesn’t Look Like Graphics

- If it’s not painfully data-parallel you have to redesign your algorithm
  - Example: scan-based techniques for zero counting in JPEG
    - Why? It’s only 64 entries!
  - Single-threaded performance is terrible. Need to parallelize.
    - Overhead of transferring data to GPU is too high.
- If it’s not accessing memory well you have to re-order your algorithm
  - Example: DCT in JPEG
    - Need to make sure your access to local memory has no bank conflicts across threads.
- Libraries starting to help
  - Lack of composability
    - Example: Combining linear algebra operations to keep data on the device
- Most code is not purely data-parallel
  - Very expensive to synchronize with the CPU (data transfer)
  - No effective support for task-based parallelism
  - No ability to launch kernels from within kernels

Legacy Code

- Code lives forever
  - Amdahl: Even optimizing the 90% of hot code limits speedup to 10x
  - Many won’t invest in proprietary technology
- Programming models are immature
  - CUDA mature low-level Nvidia, PGI
  - OpenCL immature low-level Nvidia, AMD, Intel, ARM, Altera, Apple
  - OpenMP mature
  - OpenACC immature high-level CAPS, Fatigue
- Amdahl’s Law

Corollary: What are GPUs Good For?

- Data parallel code
  - Lots of threads
  - Easy to express parallelism (no SIMD nastiness)
- High arithmetic intensity and simple control flow
  -Lots of FPs
  - No branch predictors
- High reuse of limited data sets
  - Very high bandwidth to memory (1-6GB)
  - Extremely high bandwidth to local memory (16-64kB)
- Code with predictable access patterns
  - Small (or no) caches
  - User-controlled local memories
WHAT’S THE FUTURE OF GPUs?

The Future is Not in Accelerators

- Memory
  - Unified memory address space
  - Low performance coherency
  - High performance scratchpads
- OS interaction between all cores

Focus on Data Locality

- Not just on-chip/off-chip but within a chip
- Software controllable memories
  - Configure for cache/scratch pad
  - Enable/disable coherency
  - Programmable DMA/prefetch engines
- Program must expose data movement/locality
  - Explicit information to the runtime/compiler
  - Auto-tuning, data-flow, optimization
- But we will have global coherency to get code correct

(See the Micro paper “GPUs and the Future of Parallel Computing” from Nvidia about their Echelon project and design.)

Heterogeneity for Efficiency

- No way around it in sight
- Specialize to get better efficiency
  - This is why GPUs are more efficient today
- Heterogeneous mix
  - Throughput-oriented “thin” cores
  - Latency-focused “fat” cores
- Fixed-function accelerators
  - Video, audio, network, etc.
  - Already in OpenCL 1.2
- Dark silicon
  - OS/runtime/app will have to adapt
  - Energy will be a shared resource

Focus on Data Locality

- Requires 2x bandwidth and storage

Graphics will (still) be a Priority

- It’s where the money is
- Fixed-function graphics units
- Memory-system hardware for texture interpolation will live forever...
- Half-precision floating point will live forever...
  (And others else might actually use it. Hint hint.)
CONCLUSIONS

Breaking Through The Hype

- Real efficiency advantage
  - Intel is pushing hard to minimize it
  - Much larger for single precision
- Real performance advantage
  - About 2-5x
  - But you have to re-write your code
- The market for GPUs is tiny compared to CPUs
- Everyone believes that specialization is necessary to tackle energy efficiency

The Good, The Bad, and The Future

- The Good:
  - Limited domain allows more efficient implementation
  - Good choice of domain allows good scaling
- The Bad:
  - Limited domain focus makes some algorithms hard
  - They are not x86/linux (legacy code)
- The Future:
  - Throughput-cores + latency-cores + fixed accelerators
  - Code that runs well on GPUs today will port well
  - We may share hardware with the graphics subsystem, but we won’t “program GPUs”