Around GPGPU: architecture, programming, and arithmetic

Sylvain Collange, Arénaire, LIP, ENS Lyon
David Defour, DALI, ELIAUS, Université de Perpignan

November 10, 2010
Key challenges for parallel architectures

- **Scalability**
  - Moving data is more expensive than computing
  - How to minimize data movement in a many-core architecture?

- **Power efficiency**
  - Power draw/dissipation is the current bottleneck
  - Power-directed design

- **Programming model**
  - How to write portable, reusable parallel software with minimal effort?

- **Numerical accuracy**
  - Confidence in a result produced after billions of operations?
Outline

- How a GPU works
- GPU programming guidelines
- Arithmetic features
The Tesla SM uses a **new processor architecture** we call single-instruction, multiple-thread (**SIMT**).


The Streaming Multiprocessor in reality is a **highly threaded single-issue processor with SIMD**, although this is obscured by the overall complexity and marketing of the whole architecture.


- Who is right?
- Difference with parallel processors from the 80's?
First step: MIMD

- Multiple small, independent cores

- Benefit from task / data parallelism
Second step: SIMT

- Share front-end (I$, F, D) between cores
- When both threads execute the same instruction
  - Fetch and decode it once, then broadcast it

- Benefit from instruction regularity
A GPU

- More threads / core
- More cores / shared front-end
- Replicate instructions in time
- Share load-store unit, caches

- Data parallelism
- Instruction regularity
- Data locality

```
Core 1
Threads 1-2,9-10
```
```
Core 2
Threads 3-4,11-12
```
```
Core 3
Threads 5-6,13-14
```
```
Core 4
Threads 7-8,15-16
```
### SIMD vs. SIMT

<table>
<thead>
<tr>
<th></th>
<th>SIMD or vector</th>
<th>SIMT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Vectorization</strong></td>
<td>At compile-time</td>
<td>At runtime</td>
</tr>
<tr>
<td><strong>Thread divergence</strong></td>
<td>Software-managed Bit-masking, predication</td>
<td>Hardware-managed Stack, counters, multiple PCs…</td>
</tr>
<tr>
<td><strong>Memory access</strong></td>
<td>Vector load-store Gather-scatter</td>
<td>Gather-scatter with coalescing</td>
</tr>
</tbody>
</table>

And much more… [Glew09]

- SIMT architecture: run SPMD code on SIMD units
  - Both authors are right...
  - SIMD units = only one possible implementation of SIMT
GPU design space

- What can we do with SPMD threads?
  - MIMD (multi-core)
  - SIMD
  - Pipelined vectors (Cray-like)
  - Multi-threading (Hyperthreading-like)

- This is the GPU architect's problem
- Programmer's point of view: just a bunch of threads
  - Microarchitecture-specific optimizations
  - Or just focus on locality and regularity
Outline

- How a GPU works
- GPU programming guidelines
  - Bottlenecks and limitations
  - Some recipes
- Arithmetic features
Where are my transistors gone?

- Conventional wisdom
  - CPUs have huge amounts of cache
  - GPUs have almost none

- Reality check

<table>
<thead>
<tr>
<th>GPU</th>
<th>Register files + caches</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA GF100</td>
<td>3.9 MB</td>
</tr>
<tr>
<td>AMD Cypress</td>
<td>5.8 MB</td>
</tr>
</tbody>
</table>

At this rate, will catch up with CPUs by 2012…
Little's law: data = throughput × latency

<table>
<thead>
<tr>
<th>Throughput (GB/s)</th>
<th>Latency (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>180</td>
<td>50</td>
</tr>
<tr>
<td>1300</td>
<td>50</td>
</tr>
<tr>
<td>1,25</td>
<td>10</td>
</tr>
<tr>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

Intel Core i7 920

<table>
<thead>
<tr>
<th>Throughput (GB/s)</th>
<th>Latency (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>270</td>
<td>177</td>
</tr>
<tr>
<td>350</td>
<td>12</td>
</tr>
</tbody>
</table>

NVIDIA GeForce GTX 480

<table>
<thead>
<tr>
<th>Throughput (GB/s)</th>
<th>Latency (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>12</td>
</tr>
<tr>
<td>177</td>
<td>350</td>
</tr>
<tr>
<td>1300</td>
<td>12</td>
</tr>
<tr>
<td>1,25</td>
<td>10</td>
</tr>
<tr>
<td>50</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>
What about power?

- Power measurements on NVIDIA GT200 [CDT09]

<table>
<thead>
<tr>
<th></th>
<th>Energy/op (nJ)</th>
<th>Total power (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction control</td>
<td>1.8</td>
<td>18</td>
</tr>
<tr>
<td>32-way vector MAD</td>
<td>3.6</td>
<td>36</td>
</tr>
<tr>
<td>128-byte vector load</td>
<td>80</td>
<td>90</td>
</tr>
</tbody>
</table>

- Instruction overhead is under control
  - Thanks to SIMT
- FPUs are not so cheap
  - Once we put hundreds of them on a chip
- Memory is the killer
Guidelines: scheduling work

- On multicore / multiprocessor
  - Coarse-grained parallelism
  - **Decouple** tasks to reduce **conflicts** and inter-thread communication

- On GPUs
  - Fine-grained parallelism
  - **Interleave** tasks
  - Exhibit **locality**: take advantage of local memory
  - Exhibit **regularity**: take advantage of SIMT units
Packing data

- Array of Structures (AoS)
  - Alignment?
  - Partial access (only blue)?
  - Access pattern on GPU?

- Structure of Arrays (SoA)
  - More GPU-friendly

- Prefer SoA in memory [Mici10]
  - Library to hide layout issues: [Strz10]
How many threads?

• As many as possible?
  • Maximal data-parallelism
    ➤ Latency hiding
  • Locality
    ➤ Store private data of each thread
  • Thread management overhead
    ➤ Initialization, redundant operations

• Instruction-Level Parallelism is not dead
  • Up to 5 pending loads/thread on Tesla, more on Fermi
  • Superscalar (supervector?) execution on GF104
  • VLIW on AMD architectures
Example: SGEMM from CUBLAS 1.1


- 512 threads / CTA, 15 registers / thread
- 9 registers / 15 contain redundant data
- Only 2 registers really needed
Fewer threads, more computations

- Volkov SGEMM
  - 8 elements computed / thread
  - Unrolled loops
  - Less traffic through shared memory, more through registers

- Overhead amortized
  - 1920 registers vs. 7680 for the same amount of work
  - Works for redundant computations too

- Success story
  - +60% compared to CUBLAS 1.1
  - Adopted in CUBLAS 2.0

- More in [Volk10]
Takeaway

- Distribute work and data
  - Favor SoA
  - Favor locality and regularity
  - Use common sense (avoid extraneous copies or indirections)
- More threads ≠ higher performance
  - Saturate instruction-level parallelism first (almost free)
  - Complete with data parallelism (expensive in terms of locality)
  - Compiler optimization: thread fusion?
Outline

- How a GPU works
- GPU programming guidelines
- Arithmetic features
  - IEEE-754?
  - A bit of history
  - FP capabilities
The vector unit can perform four IEEE single-precision multiply, add, or multiply-add operations, as well as inner products, max, min, and so on.


The floating-point add and multiply operations are compatible with the IEEE 754 standard for single-precision FP numbers, including not-a-number (NaN) and infinity values.


Single precision floating point instructions now support subnormal numbers by default in hardware, as well as all four IEEE 754-2008 rounding modes (nearest, zero, positive infinity, and negative infinity).

NVIDIA's next generation CUDA compute architecture: Fermi Whitepaper, 2009

All compute devices follow the IEEE 754-2008 standard for binary floating-point arithmetic with the following deviations:

[...2-page long bullet list...]

NVIDIA CUDA C Programming Guide, 2010
A short glimpse at recent GPU history

Microsoft DirectX

<table>
<thead>
<tr>
<th>Year</th>
<th>7.x</th>
<th>8.0</th>
<th>8.1</th>
<th>9.0</th>
<th>9.0a</th>
<th>9.0b</th>
<th>9.0c</th>
<th>10.0</th>
<th>10.1</th>
<th>11</th>
</tr>
</thead>
</table>

NVIDIA

<table>
<thead>
<tr>
<th>Year</th>
<th>NV10</th>
<th>NV20</th>
<th>NV30</th>
<th>NV40</th>
<th>G70</th>
<th>G80-G90</th>
<th>GT200</th>
<th>GF100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FP 16</td>
<td>Programmable shaders</td>
<td>FP 32</td>
<td>Dynamic control flow</td>
<td>“SIMT”?</td>
<td>CUDA</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ATI/AMD

<table>
<thead>
<tr>
<th>Year</th>
<th>R100</th>
<th>R200</th>
<th>R300</th>
<th>R400</th>
<th>R500</th>
<th>R600</th>
<th>R700</th>
<th>Evergreen</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FP 24</td>
<td>CTM</td>
<td>FP 64</td>
<td>CAL</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

GPGPU traction

2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010
Arithmetic features

- 2006 (ATI R500, NVIDIA G70): “Cray-1-like” FP
  - Truncated multipliers, adders with 2 guard bits and no sticky
  - $41 / 41 \neq 1$
  - Same GPU, different units: different behavior
- 2007 (ATI R600, NVIDIA G80)
  - Correct IEEE-754 rounding to the nearest for $+$, $\times$
  - Integer arithmetic and logical ops
- 2008 (AMD R670, NVIDIA GT200)
  - Binary64
- 2010 (AMD Evergreen, NVIDIA GF100)
  - 4 mandatory IEEE rounding modes
  - FMA for both Binary32 and Binary64
  - Subnormals at full-speed
34 years later: still no complex datatypes nor correct rounding of elementary functions

But we have hardware elementary functions on GPUs

- $1/x$, $1/\sqrt{x}$, $\log_2$, $2^x$, sin, cos
- Accuracy: 22 to 23 bits

Applications: graphics, physics, finance…

We therefore conclude that

1. the entire function library should be included in the hardware if and only if COMPLEX data types and their corresponding arithmetic are formally introduced;
2. the following error/accuracy criterion should be adopted and met by the implementation: [Correct rounding].

If either of these conditions is not met, then none of the elementary functions should be included in the hardware.

G. Paul, M.W. Wilson, Should the elementary function library be incorporated into computer instruction sets?, TOMS, 1976.
Graphics is bandwidth-starved too

- Lower-precision format: Binary16
  - 11-bit significand, 5-bit exponent
  - In IEEE-754:2008
- Block Floating-Point formats
  - One shared exponent, multiple significands
  - More compact storage for correlated FP data

\[
\begin{align*}
1,01100010 & = m_1 x 2^{e} \\
0,10011011 & = m_2 x 2^{e} \\
0,00110101 & = m_3 x 2^{e} \\
1,10010111 & = m_4 x 2^{e} \\
17 & = e
\end{align*}
\]

- Lossy compression of textures in memory
  - Hardware-based on-the-fly decompression
- Lossless compression of frame buffer, depth buffer…
Higher accuracy
  - One less rounding error

Error-free transformations
  - FMA(a, b, -a×b)

Different behavior than a×b+c

Loss of symmetry (dot product…)
  - a × b + c × d ≠ c × d + a × b

In CUDA
  - fmaf(), fma() C functions
  - By default, compiler turns a*b+c expressions into FMAs
  - Use __fadd_rn(), __fmul_rn(), __dadd_rn(), __dmul_rn() in place of +, * to prevent FMAzation
Static rounding attributes

- On CPUs
  - Rounding mode as a mode for each thread
  - Get/set with e.g. fegetround() and fesetround()

- On NVIDIA GPUs
  - Rounding direction: flag in the instruction word
  - C intrinsics: __fadd_ru(), __fadd_rd(), __fmul_rz, __fmaf_rn...
  - Benefit: zero-overhead mode switch

- Applications
  - Interval arithmetic
    - “Interval” CUDA SDK sample
    - 100× speedup for the same development effort
  - Stochastic arithmetic [JL10]
Conclusion

- **GPU: throughput computing monster**
  - Feed it with lots of threads (balanced ILP/DLP diet)
  - It likes: parallelism, locality, regularity (coherence)

- **Specialized in FP arithmetic**
  - From 8-bit fixed point to IEEE-754:2008 in 10 years
  - Now better FP support than on most CPUs

- **Specialized in graphics**
  - Exotic arithmetic units

- **Can HPC learn from computer graphics?**
  - Fixed-function units, memory compression?

- **Next hardware feature?**
  - Your feature?
## FP OXO

<table>
<thead>
<tr>
<th>Format</th>
<th>FMA</th>
<th>UMA</th>
<th>Rounding</th>
<th>Subnormals</th>
<th>Inf, NaN</th>
<th>Flags</th>
<th>Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel X86</td>
<td></td>
<td></td>
<td>4 Dynamic</td>
<td>Microcode</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IBM PowerPC</td>
<td>✓</td>
<td>x</td>
<td>4 Dyn.</td>
<td>Microcode</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Intel IA-64</td>
<td></td>
<td></td>
<td>4 Dyn. + Stat.</td>
<td>Microcode</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>IBM Cell SPU</td>
<td>✓</td>
<td>x</td>
<td>RZ</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NVIDIA GT200</td>
<td></td>
<td></td>
<td>2 Static</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NVIDIA GF100</td>
<td>✓</td>
<td>x</td>
<td>4 Static</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AMD RV770</td>
<td></td>
<td></td>
<td>RN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>AMD Evergreen</td>
<td>✓</td>
<td>✓</td>
<td>4 Dyn.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>OpenCL 1.1</td>
<td></td>
<td></td>
<td>RN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Direct3D 11</td>
<td></td>
<td></td>
<td>RN</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
References

Texture filtering

- Fixed-function unit (ex NVIDIA GT200)
- Interpolate color of **Pixels** from **Texels**

\[ T[i,j], T[i,j+1], ... \]

![Diagram showing texture filtering process](image)

- Applications: graphics, image processing
- Can be hijacked to evaluate piecewise polynomials
  - **Evaluate functions “for free”** [ACD10]
Handling thread divergence

- Many techniques. e.g. Fermi:
  - Generic SIMT branch instruction
    - If all threads take the same path, treat as a branch
    - If not, fall back to predication
    - Handles nested control flow with a stack
  - Predication (for very short branches)
    - Take all paths, mask out unneeded calculations
  - Predicate-or-skip (for innermost conditionals)
    - Lighter version of generic mechanism
  - Select (for selective assignment)
- Compiler: selects which one to use
- Programmer: favor nondivergent conditionals
  - Regularity at algorithmic level
State of the art in 2006

- NVIDIA G70, ATI R500
- “Cray 1-like” floating-point arithmetic
  - Truncated multipliers
  - Adders with two guard bits and no sticky
  - $41 / 41 \neq 1$
  - Different behavior for different units on the same GPU