Why are Graphics Systems so Fast?

Pat Hanrahan
Pervasive Parallelism Laboratory
Stanford University

PACT Keynote
September 14, 2009

Answer 1

Simulating virtual worlds requires high performance
Modern Graphics Pipeline

Application
Command
Geometry
Rasterization
Texture
Fragment
Display

NVIDIA Historicals

<table>
<thead>
<tr>
<th>Year</th>
<th>Product</th>
<th>Tri rate</th>
<th>CAGR</th>
<th>Tex rate</th>
<th>CAGR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1998</td>
<td>Riva ZX</td>
<td>3m</td>
<td>-</td>
<td>100m</td>
<td>-</td>
</tr>
<tr>
<td>1999</td>
<td>Riva TNT2</td>
<td>9m</td>
<td>3.0</td>
<td>350m</td>
<td>3.5</td>
</tr>
<tr>
<td>2000</td>
<td>GeForce2 GTS</td>
<td>25m</td>
<td>2.8</td>
<td>664m</td>
<td>1.9</td>
</tr>
<tr>
<td>2001</td>
<td>GeForce3</td>
<td>30m</td>
<td>1.2</td>
<td>800m</td>
<td>1.2</td>
</tr>
<tr>
<td>2002</td>
<td>GeForce Ti 4600</td>
<td>60m</td>
<td>2.0</td>
<td>1200m</td>
<td>1.5</td>
</tr>
<tr>
<td>2003</td>
<td>GeForce FX</td>
<td>167m</td>
<td>2.8</td>
<td>2000m</td>
<td>1.7</td>
</tr>
<tr>
<td>2004</td>
<td>GeForce 6800 Ultra</td>
<td>170m</td>
<td>1.0</td>
<td>6800m</td>
<td>2.7</td>
</tr>
<tr>
<td>2005</td>
<td>GeForce 7800 GTX</td>
<td>940m</td>
<td>3.9</td>
<td>10300m</td>
<td>2.0</td>
</tr>
<tr>
<td>2006</td>
<td>GeForce 7900 GTX</td>
<td>1400m</td>
<td>1.5</td>
<td>15600m</td>
<td>1.4</td>
</tr>
<tr>
<td>2007</td>
<td>GeForce 8800 GTX</td>
<td>1800m</td>
<td>1.3</td>
<td>36800m</td>
<td>2.3</td>
</tr>
<tr>
<td>2008</td>
<td>GeForce GTX 280</td>
<td>48160m</td>
<td>1.7</td>
<td>48160m</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Yearly Growth well above 1.5 (Moore’s Law)
Answer 2

Cinematic games and media drive large GPU market

Afford the cost of SOA designs

ATI Radeon 4870

- 55 nm process
- 958 million transistors
- 1016-wide 5-op cores
- 750 Mhz clock
- 256-bit GDDR memory
  GDDR3/4/5 @ 3.6 Ghz = 115.2 GB/s
- 334 Watts
NVIDIA GTX 280

- 65 nm TSMC process
- 1.4 billion transistors
- 575 mm^2
- 240 scalar processors
  - 1.3 Ghz clock rate
- 512-bit GDDR memory
  - GDDR @ 1.1 Ghz = 141.7 GB/s
- 236 Watts

Answer 3

GPUs efficiently use semiconductor technology
Scaling Laws

Moore’s Law

- Number of transistors doubles every 18 months
- Number of transistors increases by \(~50\%\) / yr
- Feature size decreases by \(~25\%\) / yr
- Gate delay decreases with feature size – by \(~25\%\) / yr

Semiconductor capability =

Number of transistors / Switching speed

- 50\% (number) + 25\% (speed)

The Capability Gap

Graph courtesy of Bill Dally
Answer 4

GPUs cleverly employ many forms of parallelism in innovative ways

3 Axes of Parallelism

- Multi-Core
- SIMD Vector
- Multi-Thread
GPU Architectures

A Closer Look at GPUs
Kayvon Fatahalian and Mike Houston
Communications of the ACM. Vol. 51, No. 10 (October 2008)

From Shader Code to a Teraflop: How Shader Cores Work
Kayvon Fatahalian
Beyond Programming Shading, SIGGRAPH 2009 Course Notes

GeForce G80 Series GPU
 Shader Model 4.0 Architecture

Basic Vertex Program

DP4  o[HPOS].x, c[0], v[OPOS];  # Transform pos.
DP4  o[HPOS].y, c[1], v[OPOS];
DP4  o[HPOS].z, c[2], v[OPOS];
DP4  o[HPOS].w, c[3], v[OPOS];
DP3  R0.x, c[4], v[NRML];       # Transform normal.
DP3  R0.y, c[5], v[NRML];
DP3  R0.z, c[6], v[NRML];
DP3  R1.x, c[32], R0;           # R1.x = L DOT N'
DP3  R1.y, c[33], R0;           # R1.y = H DOT N'
MOV  R1.w, c[38].x;            # R1.w = specular
LIT  R2, R1;                   # Compute lighting
MAD  R3, c[35].x, R2.y, c[35].y; # diffuse + ambient
MAD  o[COL0].xyz, c[36], R2.z, R3; # + specular
END
G80 “core” / CUDA Architecture

Each core
- Thread block
  - SIMD (SIMT)
  - 8 functional units
  - MADD + MUL
  - 16/32 “warp”
- 32 thread blocks per core
- 1024 “threads” total

Each unit
- 2-3 cores
- 16 KB shared memory

Critical Inner Loop for Graphics

```plaintext
ps_2_0
DCL   t0.xy  # Interpolate t0.xy
DCL   v0.xyzw # Interpolate v0.xyzw
DCL_2D s0 # Declaration – no code
TEXID r0, t0, s0 # TEXTURE LOAD!
MUL   r1, r0, v0 # Multiply
MOV   oC0, r1 # Store to framebuffer
```

The program must run at 100% efficiency

- Short inner loop
- Very little state (few registers)
- Random memory (texture) access
GPU Multi-threading

Change thread after texture fetch/stall

NVIDIA GeForce GTX 285 “core”
Typical Chip

16 cores
8 mul-add ALUs per core (128 total)
= 256 GFLOPs (@ 1GHz)

“Enthusiast” Chip!

32 cores x 8 SIMD functional units x 3 flops/cycle x 1.3 Ghz = 933 Gflops
NVIDIA GeForce GTX 285

- NVIDIA-speak:
  - 240 stream processors
  - “SIMT execution”

- Generic speak:
  - 30 cores
  - 8 SIMD functional units per core

AMD Radeon HD 4890 “core”

- SIMD VLIW functional unit, control shared across 16 units
- instruction stream decode
- multiply-add
- execution context storage
AMD Radeon HD 4890 “core”

- Groups of 64 [fragments/vertices/etc.] share instruction stream
  (AMD doesn’t have a fancy name like “WARP”)
  - One fragment processed by each of the 16 SIMD units
  - Repeat for four clocks
AMD Radeon HD 4890

- **AMD-speak:**
  - 800 stream processors
  - HW-managed instruction stream sharing (like “SIMT”)

- **Generic speak:**
  - 10 cores
  - 16 SIMD functional units per core
  - 5 ALUs per VLIW unit per SIMD lane

---

**Larrabee**

Larrabee: A many-core x86 architecture for visual computing,
(IEEE Micro 2009, Top Pick)
**Larrabee Core**

- Separate scalar and vector units
- Separate register files
- In-order IA scalar core
- Vector unit: 16 32-bit ops/clock
- Short execution pipelines
- Fast access from L1 cache
- Direct connection to L2 cache
- Prefetch to manage L1/L2 caches

**Vector Processing Unit**

- Vector instructions support
  - Fast, wide read from L1 cache
  - Numeric type conversion and data
  - Rearrange the lanes on register read
  - Fused multiply add (three arguments)
  - Int32, Float32 and Float64 data

- Augmented vector instruction set
  - Scatter/gather for vector load/store
  - Mask registers select lanes
Example LRBni Vector Instructions

- **Multiply-add:**
  - `vmadd132ps v1, v2, v3`

- **Mask the writing of the elements:**
  - `vmadd132ps v1 {k1}, v2, v3`

- **Source from memory:**
  - `vmadd132ps v1 {k1}, v2, [rbx+rcx*4]`

- **Memory source undergoes format conversion:**
  - `vmadd132ps v1 {k1}, v2, [rbx+rcx*4]{float16}`

---

Different Notions of “SIMD”

- **Option 1: Scalar and vector instructions**
  - Small number of threads
  - Intel/AMD x86 SSE, Intel Larrabee

- **Option 2: Scalar instructions ⇒ implicit vectors**
  - Only scalar instructions; hardware merges instruction streams
  - Each instruction is executed a small number of times
  - NVIDIA GeForce (“SIMT” warps), AMD Radeon architectures
Various Notions of Threads

**Cores**: Each runs multiple threads

**Thread**: HW-managed context (hide short unpredictable latencies)

... More Threads (up to 4 per core, share memory via L1 & L2 caches)

Larrabee: Core, Threads

---

Various Notions of Threads

**Cores**: Each runs multiple threads

**Thread**: HW-managed context (hide short unpredictable latencies)

**Fiber**: SW-managed context (hides long predictable latencies)

16-wide vector unit

... More Fibers running (typically 2-10, depending on latency to cover)

... More Threads (up to 4 per core, share memory via L1 & L2 caches)

Larrabee: Core, Threads, Fibers, Strands
Intel Larrabee “core”

- 32 KB of L1 cache
- 256 KB of L2 cache
- Each HW context:
  - 32 vector registers
  - SIMD vector unit, control shared across 16 lanes
  - Additional scalar unit
  - Execution context storage/HW registers
  - Instruction stream decode

Intel Larrabee

- Intel speak:
  - We won’t say anything about core count or clock rate
  - Explicit 16-wide vector ISA
  - Each core interleaves four x86 instruction streams

- Generic speak:
  - That was the generic speak
Larrabee

Each Larrabee core is a complete IA core
- Context switching & pre-emptive multi-tasking
- Virtual memory and page swapping
- Fully coherent caches at all levels of the hierarchy

Efficient inter-block communication
- Ring bus for full inter-processor communication
- Low latency high bandwidth L1 and L2 caches
- Fast synchronization between cores and caches

Recap
CPU-“style” Cores

- Fetch/Decode
- ALU (Execute)
- Execution Context
- Out-of-order control logic
- Fancy branch predictor
- Memory pre-fetcher
- Data cache (A big one)

GPU-style Cores

- Fetch/Decode
- ALU 1
- ALU 2
- ALU 3
- ALU 4
- ALU 5
- ALU 6
- ALU 7
- ALU 8
- Execution Context
- Shared Local Data
Three Key Ideas

1. Simplify the core.
   - Remove high-overhead logic to control out of order execution, branch predication, etc.

2. Exploit the efficiency of SIMD processing
   - Share instructions and replicate functional units

3. Use many threads to hide memory latency
   - Smaller caches, but still need thread state
   - If you have enough thread state, never a stall

Optimizing for Throughput

*Hypothetical Core design experiment:*
*Specify a throughput-optimized processor with same area and power of a standard dual core CPU*

<table>
<thead>
<tr>
<th># CPU cores</th>
<th>2 out of order</th>
<th>10 in-order</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions per issue</td>
<td>4 per clock</td>
<td>2 per clock</td>
</tr>
<tr>
<td>VPU lanes per core</td>
<td>4-wide SSE</td>
<td>16-wide</td>
</tr>
<tr>
<td>L2 cache size</td>
<td>4 MB</td>
<td>4 MB</td>
</tr>
<tr>
<td>Single-stream</td>
<td>4 per clock</td>
<td>2 per clock</td>
</tr>
<tr>
<td><strong>Vector throughput</strong></td>
<td><strong>8 per clock</strong></td>
<td><strong>160 per clock</strong></td>
</tr>
</tbody>
</table>

20 times greater throughput for same area and power
Amdahl’s Implication – Never forget the Uni!!

Chip Assumptions:
200mm² for Cores
80W for Cores
Use as appropriate for each CMP option

<table>
<thead>
<tr>
<th>Rel. Perf.</th>
<th>P=0.25</th>
<th>P=0.5</th>
<th>P=1</th>
<th>P=1.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm²/Core</td>
<td>1.5 mm²</td>
<td>6 mm²</td>
<td>25 mm²</td>
<td>50 mm²</td>
</tr>
<tr>
<td># of Cores</td>
<td>128</td>
<td>32</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Power/Core</td>
<td>0.6 W</td>
<td>2.5 W</td>
<td>10 W</td>
<td>20 W</td>
</tr>
</tbody>
</table>

April 22, 2007

The Role of Accelerated Computing in the Multi-core Era

iPhone 3GS

SIM slot
App Processor/DDR SDRAM
Accelerometer
Transceiver
Power Amps
NAND Flash
GPS
BB-PMIC
SRAM
PMIC
BaseBand Processor
Apple/Samsung SoC (CPU, GPU, Mem)

Samsung ARM Cortex A8

S5PC100 Block Diagram

Timers
PLL x 4
DMA (32ch)
Keypad (8 x 8)
ADC & Touch Screen

Connectivity
24-bit ISA Bus 5.1
2 x ISA/AC97/PCM
2 x SPI/IF
4 x UART
I²C v1.1
1 x I²C
3 x I²C-SPI
MIPI-KS/Modern U/L
USB Host 1/0/DIO 2.0
3 x IS-MMC-SD

Cortex A8
32KB/32KB I/D Cache
667/653MHz
256KB L2 Cache
NEON

Secure RAM
TFT LCD Controller w/DVI

Secure ROM
Multi-Layer AHB/AXI Bus

Multimedia Acceleration
Camera 9 w/ CSI-2
720p Video Engine
2D/3D Graphics
NTSC/PAL/HD/MI
JPG CODEC

Memory Interfaces
SRAM / ROM / NOR / 64mA/SRAM
nSDR / DDR2
OnsDRAM
LPDDR4/ LPDDR2
Mic Flash w/ 8-bit EEC
CPI (ATA)
Imagination PowerVR SGX535

SGX520 3.5M Tri/S, 125M Pix/s @ 100 Mhz

Heterogenous “Fusion” Architectures

Emergence of a hybrid processor
- 2-8 CPUs
- 16-64 GPUs
- Hardware for video compression/decompression
- ...

Plans announced by AMD and Intel
Answer 5

Graphics Systems are Programmed at a High-Level of Abstraction (Utilize Domain-Specific Languages)

Brook

Ian Buck
PhD Thesis
Stanford University

Brook for GPUs: Stream computing on graphics hardware,
I. Buck, T. Foley, D. Horn, J. Sugarman, K. Fatahalian, M. Houston,
P. Hanrahan, SIGGRAPH 2004

CUDA: Scalable parallel programming made clear,
J. Nickolls, I. Buck, K. Skadron, and M. Garland,
ACM Queue, April 2008
Brook Example

```c
kernel void foo (
    float a<>, float b<>,
    out float result<> )
{
    result = a + b;
}
```

```c
float a<100>;
float b<100>;
float c<100>;
foo(a,b,c);
```

```c
for (i=0; i<100; i++)
    c[i] = a[i]+b[i];
```

Current Statistics: September 13, 2009

<table>
<thead>
<tr>
<th>Client type</th>
<th>Current TFLOPS*</th>
<th>Active Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>215</td>
<td>225,721</td>
</tr>
<tr>
<td>Mac OS X/Intel</td>
<td>22</td>
<td>5,063</td>
</tr>
<tr>
<td>Linux</td>
<td>77</td>
<td>45,028</td>
</tr>
<tr>
<td>ATI</td>
<td>1,027</td>
<td>10,069</td>
</tr>
<tr>
<td>NVIDIA</td>
<td>1,992</td>
<td>16,736</td>
</tr>
<tr>
<td>PS/3</td>
<td>1,075</td>
<td>38,110</td>
</tr>
<tr>
<td>Total</td>
<td>4,412</td>
<td>347,825</td>
</tr>
</tbody>
</table>

*TFLOPs is actual folding flops, not peak values
Domain-Specific Languages

Graphics (GRAMPS)
Molecular dynamics (GROMACS)
Physical simulation on meshes (Liszt)
Data-parallel programming (Kore)
Statistics/machine learning and data analysis (Bern)
Computer vision and imaging
Brain simulation
Autonomous vehicles
...

Wrap-Up
Questions and Answers

Why are graphics systems so fast?

1. Simulating virtual worlds requires high performance
2. Cinematic games and media drive large GPU market
3. GPUs (more) efficiently use semiconductor resources
4. GPUs employ many forms of parallelism in innovative ways (core, thread, vector)
5. GPUs are programmed at a high-level

Why are other computer systems so slow / inefficient?

Architectural Issues

High-throughput processor design

- SIMD vs. blocked threads (SIMT)
- Software- vs. hardware-managed threads

Processor of the future likely to be a hybrid CPU/GPU

- Why? Heterogeneous workload
- Small number of traditional CPU cores running a moderate number of sequential tasks
- Large number of high-throughput GPU cores running data-parallel work
- Special hardware for tasks that need to be power efficient
Opportunities

Current hardware not optimal
- Incredible opportunity for architectural innovation

Current software environment immature
- Incredible opportunity for reinventing parallel computing software, programming environments and language

Acknowledgements

Bill Dally  Ian Buck
Alex Aiken  Kayvon Fatahalian
Eric Darve  Tim Foley
Vijay Pande  Daniel Horn
Bill Mark  Michael Houston
John Owens  Jeremy Sugerman
Kurt Akeley  Doug Carmean
Mark Horowitz  Michael Abrash

Funding: DARPA, DOE, ATI, IBM, INTEL, NVIDIA, SONY
Questions?