Introduction to GPUs and CUDA

Mark Greenstreet

CpSc 418 – Nov. 22, 2011

Lecture Outline

- GPUs
  - Early geometry engines.
  - Adding functionality and programmability.
  - GPGPUs
- CUDA
  - Execution Model
  - Memory Model
  - Code Snippets

Before the first GPU

Early 1980's: bit-blit hardware for simple 2D graphics.

- Draw lines, simple curves, and text.
- Fill rectangles and triangles.
- Color used a "color map" to save memory:
  - bit-wise logical operations on color map indices!

1989: The SGI Geometry Engine

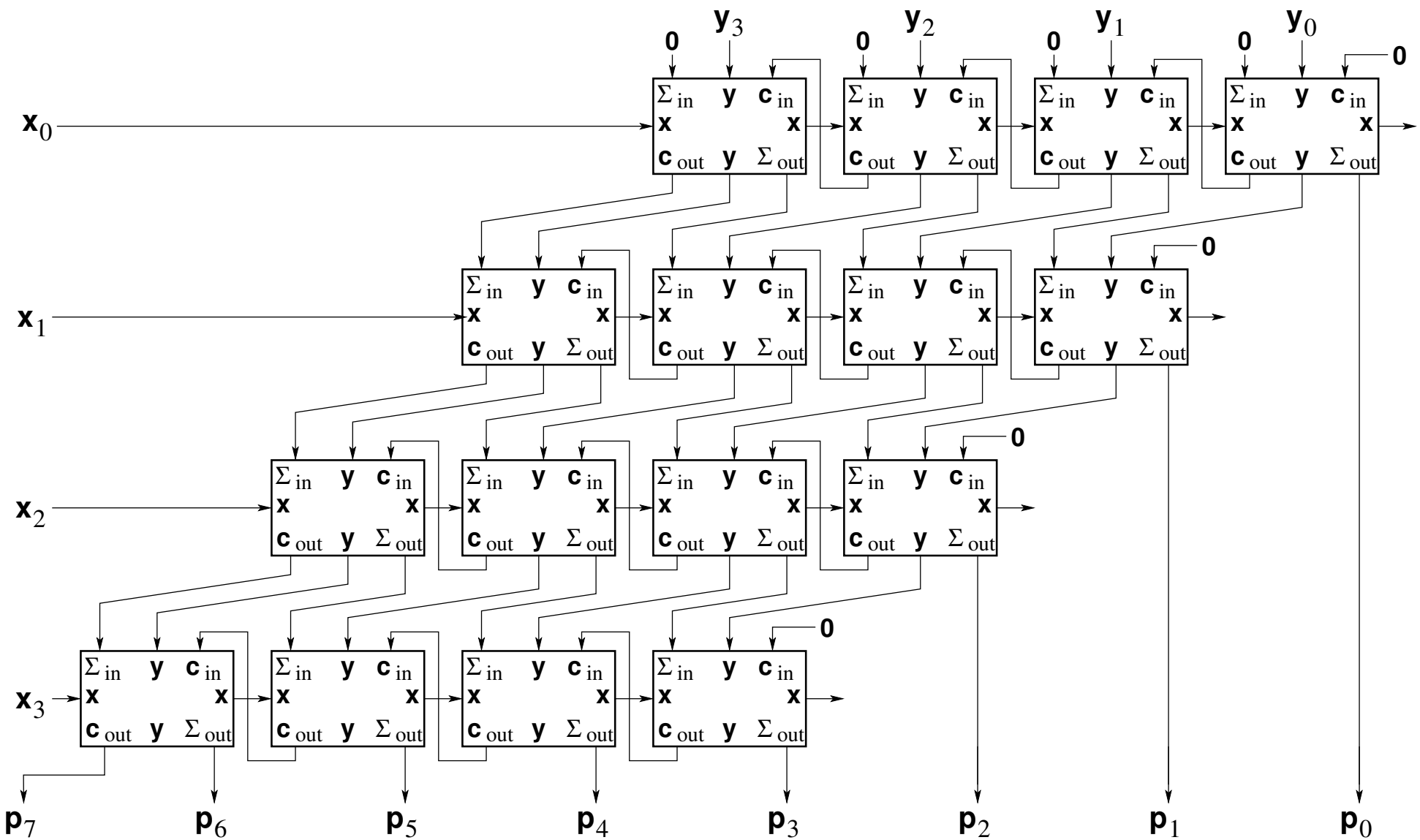
- Basic rendering: coordinate transformation.
  - Represent a 3D point with a 4-element vector.
  - The fourth element is 1, and allows translations.
  - Multiply vector by matrix to perform coordinate transformation.
- Dedicated hardware is much more efficient that a general purpose CPU for matrix-vector multiplication.
  - For example, a 32×32 multiplier can be built with $32^2 = 1024$ one-bit multiplier cells.
    - ★ A one-bit multiplier cell is about 50 transistors.
    - ★ That's about 50K transistors for a very simple design.
      30K is quite feasible using better architectures.

1989: The SGI Geometry Engine

- Basic rendering: coordinate transformation.
- Dedicated hardware is much more efficient that a general purpose CPU for matrix-vector multiplication.
  - For example, a 32×32 multiplier can be built with $32^2 = 1024$ one-bit multiplier cells.
    - ★ A one-bit multiplier cell is about 50 transistors.
    - ★ That's about 50K transistors for a very simple design.
      30K is quite feasible using better architectures.
  - The 80486DX was also born in 1989.
    - ★ The 80486DX was 1.2M transistors, 16MHz, 13MIPs.
    - ★ That's equal to 24 dedicated multipliers.
    - ★ 16 multiply-and-accumulate units running at 50MHz (easy in the same 1μ process) produce 1.6GFlops!

Why is dedicated hardware so much faster?

Consider a multiiplier:



Building a better multiplier

- Simple multiplier takes time $O(N^2)$.
- Use carry-lookahead adders (compute carries with a scan)
  - time is $O(N \log N)$
  - but the hardware is more complicated.
- Use carry-save adders and one carry-lookahead at the end
  - time is $O(N)$
  - and the hardware is more simpler (than the all CLA design)
- Add pipeline registers between rows
  - throughput is one multiply per cycle.
  - but the latency is $O(N)$.
  - Graphics and many numerical computations are very tolerant of latency.

The fundamental challenge of graphics

Human vision isn't getting any better.

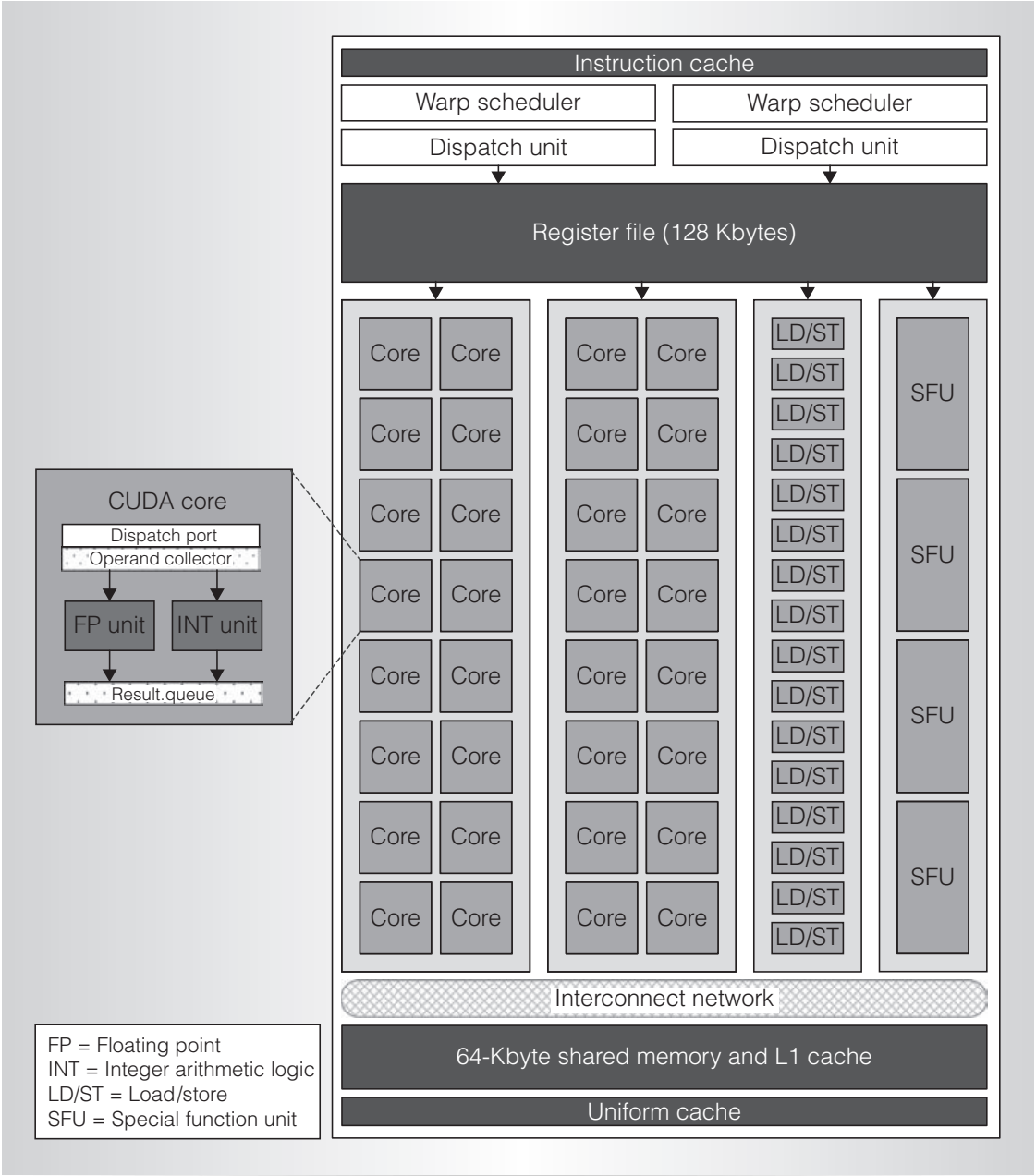
- Once you can perform a graphics task at the limits of human perception (or the limits of consumer budget for monitors), then there's no point in doing it any better.
- Rapid advances in chip technology meant that coordinate transformations (the specialty of the SGI Geometry Engine) were soon as fast as anyone needed.
- Graphics processors have evolved to include more functions. For example,
  - Shading
  - Texture mapping
- This led to a change from hardwired architectures, to programmable ones.

The GPGPU

General Purpose Graphics Processing Unit

- The volume market is for graphics, and the highest profit is GPUs for high-end gamers.
  - Most of the computation is floating point.
  - Latency doesn't matter.
  - Abundant parallelism.
- Make the architecture fit the problem:
  - SIMD – single instruction, multiple (parallel) data streams.
    - ★ Amortize control overhead over a large number of functional units.
    - ★ They call it SIMT (. . . , multiple threads) because they allow conditional execution.
  - High-latency operations
    - ★ Allows efficient, high-throughput, high-latency floating point units.
    - ★ Allows high latency accesses to off-chip memory.
  - This means lots of threads per processor.

The Fermi Architecture



What does a core look like?

- Picture to be drawn on the whiteboard.
- Many parallel functional units – amortize control overhead.
- Deeply pipelined for high throughput.
- No bypassing – simplifies hardware
  - But the software must be extensively multithreaded to get good performance.

Lecture Outline

- GPUs
  - been there, done that.
- CUDA – we are here!
  - Execution Model
  - Memory Model
  - Code Snippets

Execution Model

- The anatomy of a CUDA program
  - A CUDA application consists of one or more thread blocks.
  - A thread block consists of one or more warps.
  - A warp consists of one or more threads.
- Why?
  - The program structure reflects the GPGPU architecture.
  - To get good performance, the programmer needs to focus on "more" for each "or more" mentioned above.
- The next few slides describe the program structure in more detail,
  - Working up from threads to applications.

CUDA Threads

- Sequential threads of execution.
- Basically like normal C-code.

CUDA Warps

- Multiple threads that are executing the same code.
- These will map to a "streaming multiprocessor" on the GPGPU.
  - On Fermi, a streaming multiprocessor supports 32 parallel operations.
  - Thus, for optimal efficiency, a warp should have a multiple of 32 threads.
- A word about conditionals:
  - A thread can have control statements such as `if`.
  - If all threads in a warp do the same thing at a conditional, execution is efficient.
    - ⋆ Otherwise, the then threads will execute, while the else threads do nothing,
    - ⋆ Likewise, the else threads will execute while the then threads do nothing.
    - ⋆ Warning: CUDA doesn't say that it's then before else or any other particular order. Don't depend on the ordering.

CUDA Thread Blocks

- Multiple warps running on the same core.
- Switching between warps handles pipeline hazards.
- The warps in a block seem the same local memory – easy communication.
- Hardware support for barrier operations to synchronize the warps of a block.

The Fermi Memory Hierarchy

saxpy