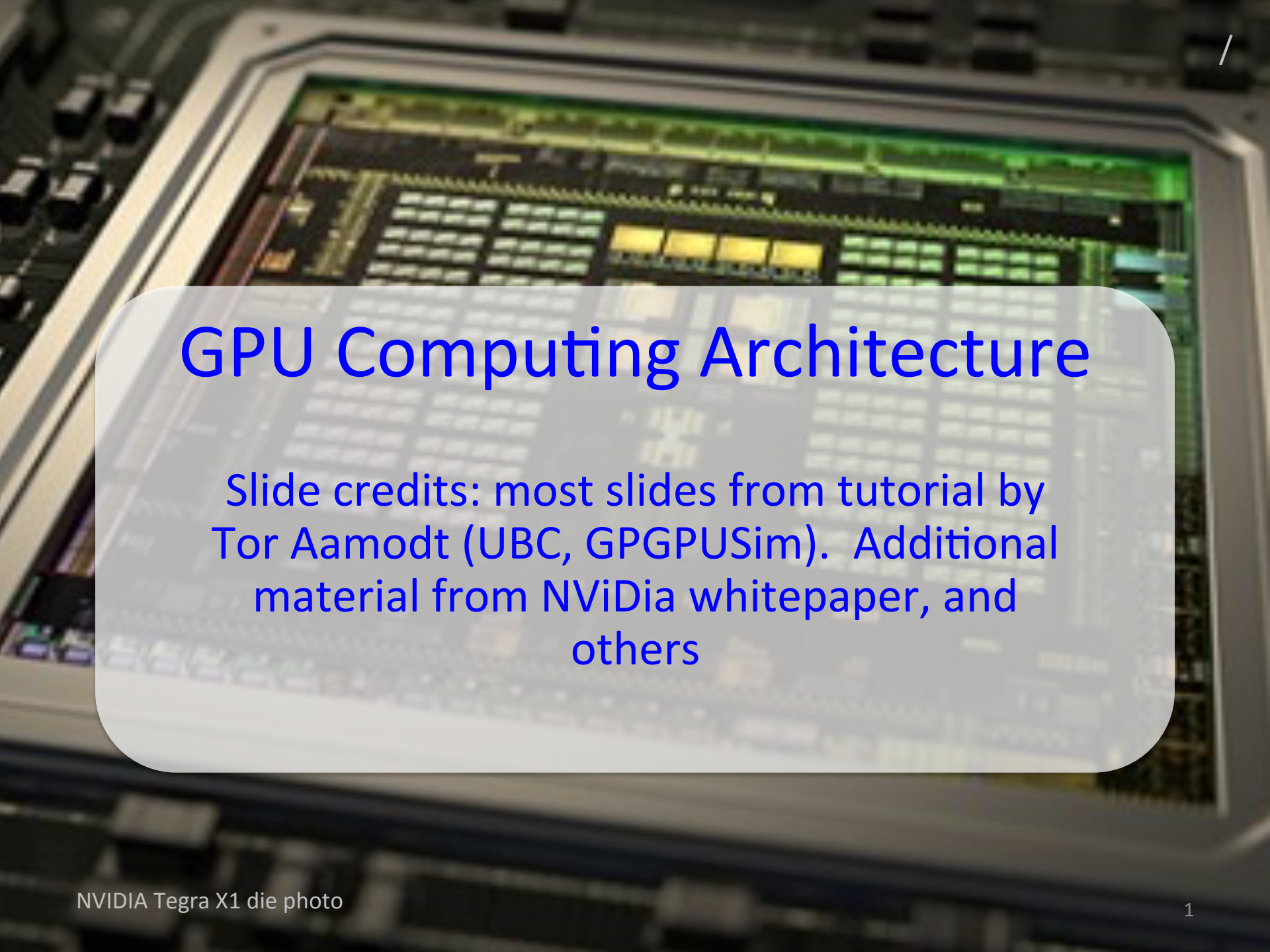


GPU Computing Architecture

Slide credits: most slides from tutorial by Tor Aamodt (UBC, GPGPUSim). Additional material from NViDia whitepaper, and others

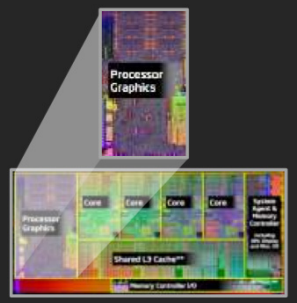
NVIDIA Tegra X1 die photo

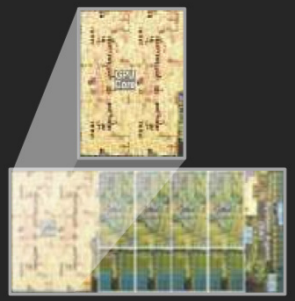
The GPU is Ubiquitous

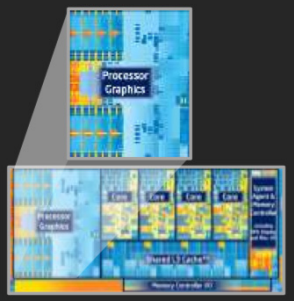
THE FUTURE BELONGS TO THE APU:
BETTER GRAPHICS, EFFICIENCY AND COMPUTE

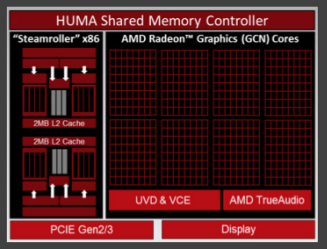
AMD

"SANDY BRIDGE"	"IVY BRIDGE"	"HASWELL"	2014 AMD A-SERIES/CODENAMED "KAVERI"
		(Estimated)	
17% GPU*	**27%** GPU*	**31%** GPU*	**47%** GPU









DELIVERS BREAKTHROUGHS IN APU-BASED:

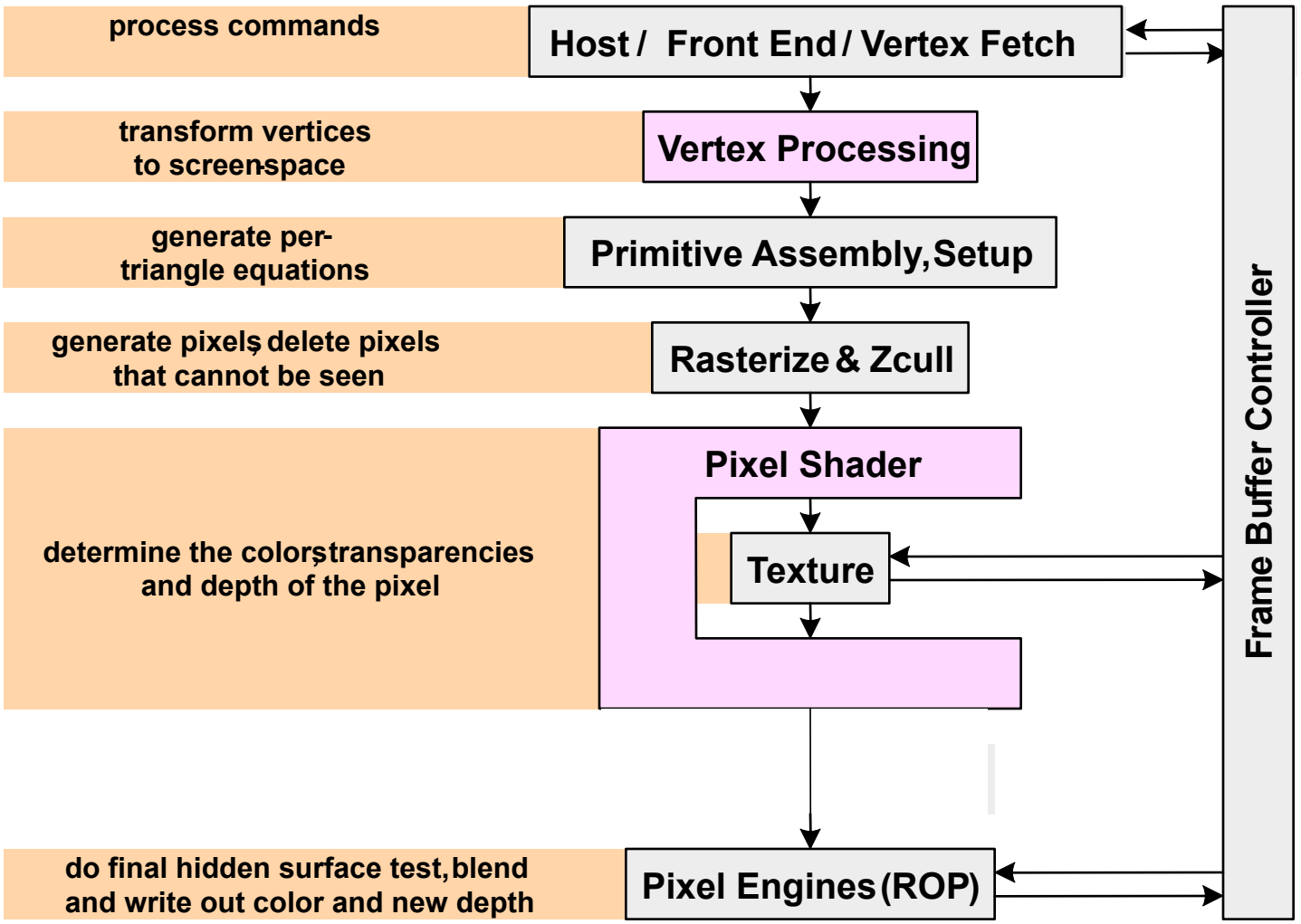
- **Compute**
  - (OpenCL™, Direct Compute)
- **Gaming**
  - (DirectX®, OpenGL, Mantle)
- **Experiences**
  - (Audio, Ultra HD, Devices, New Interactivity)

[APU13 keynote]

“Early” GPU History

- 1981: IBM PC Monochrome Display Adapter (2D)
- 1996: 3D graphics (e.g., 3dfx Voodoo)
- 1999: register combiner (NVIDIA GeForce 256)
- 2001: programmable shaders (NVIDIA GeForce 3)
- 2002: floating-point (ATI Radeon 9700)
- 2005: unified shaders (ATI R520 in Xbox 360)
- 2006: compute (NVIDIA GeForce 8800)

GPU: The Life of a Triangle

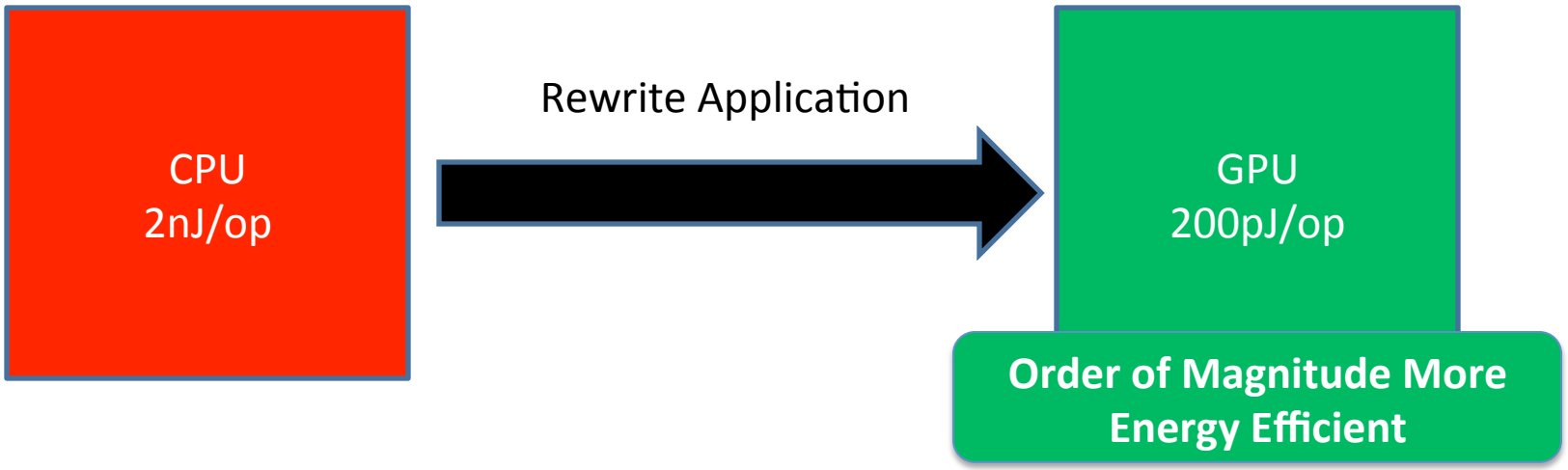


[David Kirk / Wen-mei Hwu]

pixel color result of running “shader” program

Why use a GPU for computing?

- GPU uses larger fraction of silicon for computation than CPU.
- At peak performance GPU uses order of magnitude less energy per operation than CPU.



e.g., paper in ISCA 2010 shows average speedup for throughput bound applications over core i7 is 2.5 x

However….
Application must perform well

Growing Interest in GPGPU

- Supercomputing – Green500.org Nov 2014

  "the top three slots of the Green500 were powered by three different accelerators with number one, L-CSC, being powered by AMD FirePro™ S9150 GPUs; number two, Suiren, powered by PEZY-SC many-core accelerators; and number three, TSUBAME-KFC, powered by NVIDIA K20x GPUs. Beyond these top three, the next 20 supercomputers were also accelerator-based."

- Deep Belief Networks map *very* well to GPUs (e.g., Google keynote at 2015 GPU Tech Conf.)

  http://blogs.nvidia.com/blog/2015/03/18/google-gpu/

  http://www.ustream.tv/recorded/60071572

Part 1: Preliminaries and Instruction Set Architecture

GPGPUs vs. Vector Processors

- Similarities at hardware level between GPU and vector processors.

- (I like to argue) SIMT programming model moves hardest parallelism detection problem from compiler to programmer.

GPU Instruction Set Architecture (ISA)

- NVIDIA defines a virtual ISA, called "PTX" (Parallel Thread eXecution)
- More recently, Heterogeneous System Architecture (HSA) Foundation (AMD, ARM, Imagination, Mediatek, Samsung, Qualcomm, TI) defined the HSAIL virtual ISA.
- PTX is Reduced Instruction Set Architecture (e.g., load/store architecture)
- Virtual: infinite set of registers (much like a compiler intermediate representation)
- PTX translated to hardware ISA by backend compiler ("ptxas"). Either at compile time (nvcc) or at runtime (GPU driver).

Some Example PTX Syntax

- Registers declared with a type:

```
.reg .pred  p, q, r;
.reg .u16   r1, r2;
.reg .f64   f1, f2;
```

- ALU operations

```
add.u32 x, y, z;         // x = y + z
mad.lo.s32 d, a, b, c; // d = a*b + c
```

- Memory operations:

```
ld.global.f32 f, [a];
ld.shared.u32 g, [b];
st.local.f64  [c], h
```

- Compare and branch operations:

```
    setp.eq.f32 p, y, 0;  // is y equal to zero?
@p bra L1  // branch to L1 if y equal to zero
```

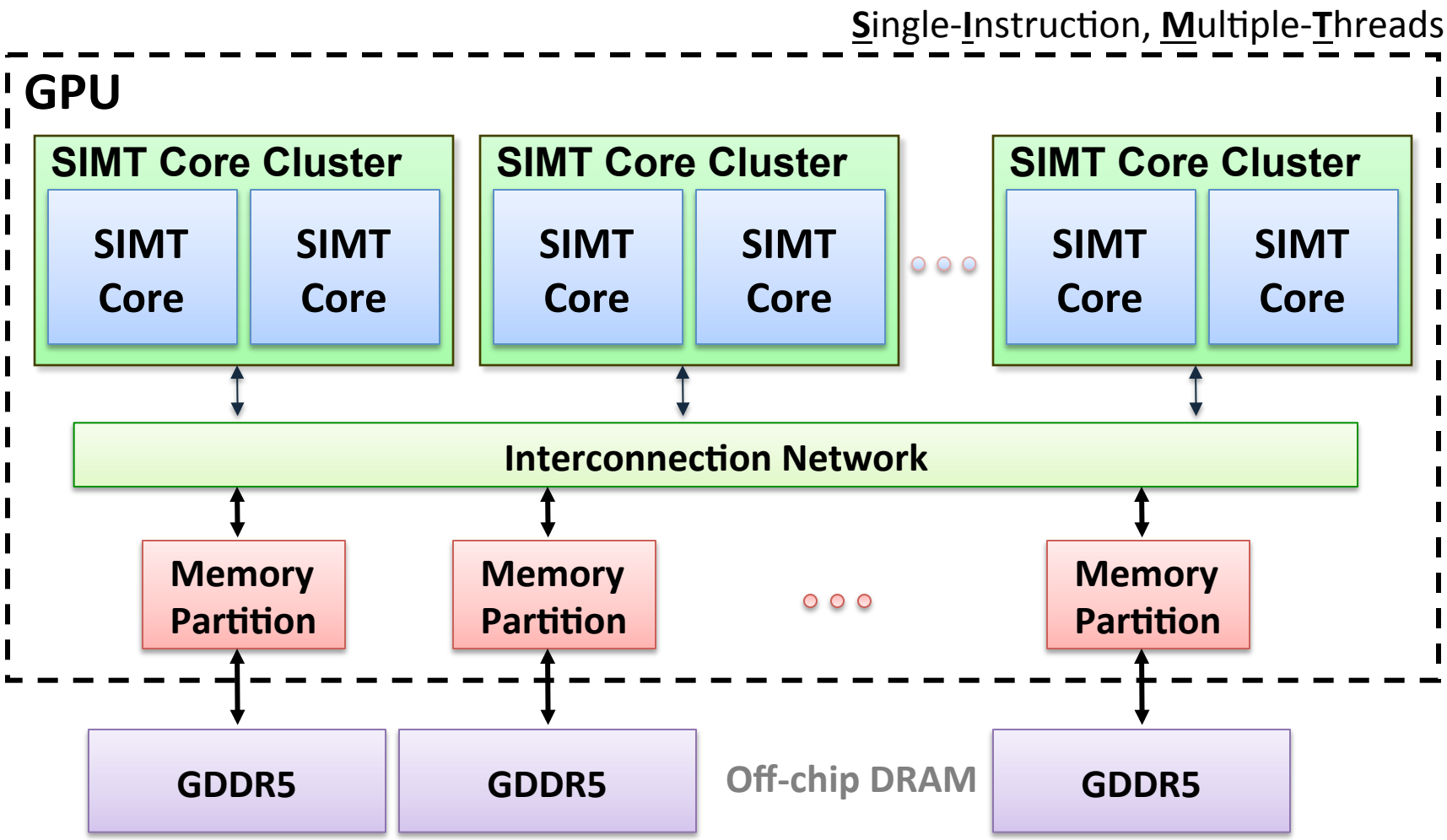
Part 2: Generic GPGPU Architecture

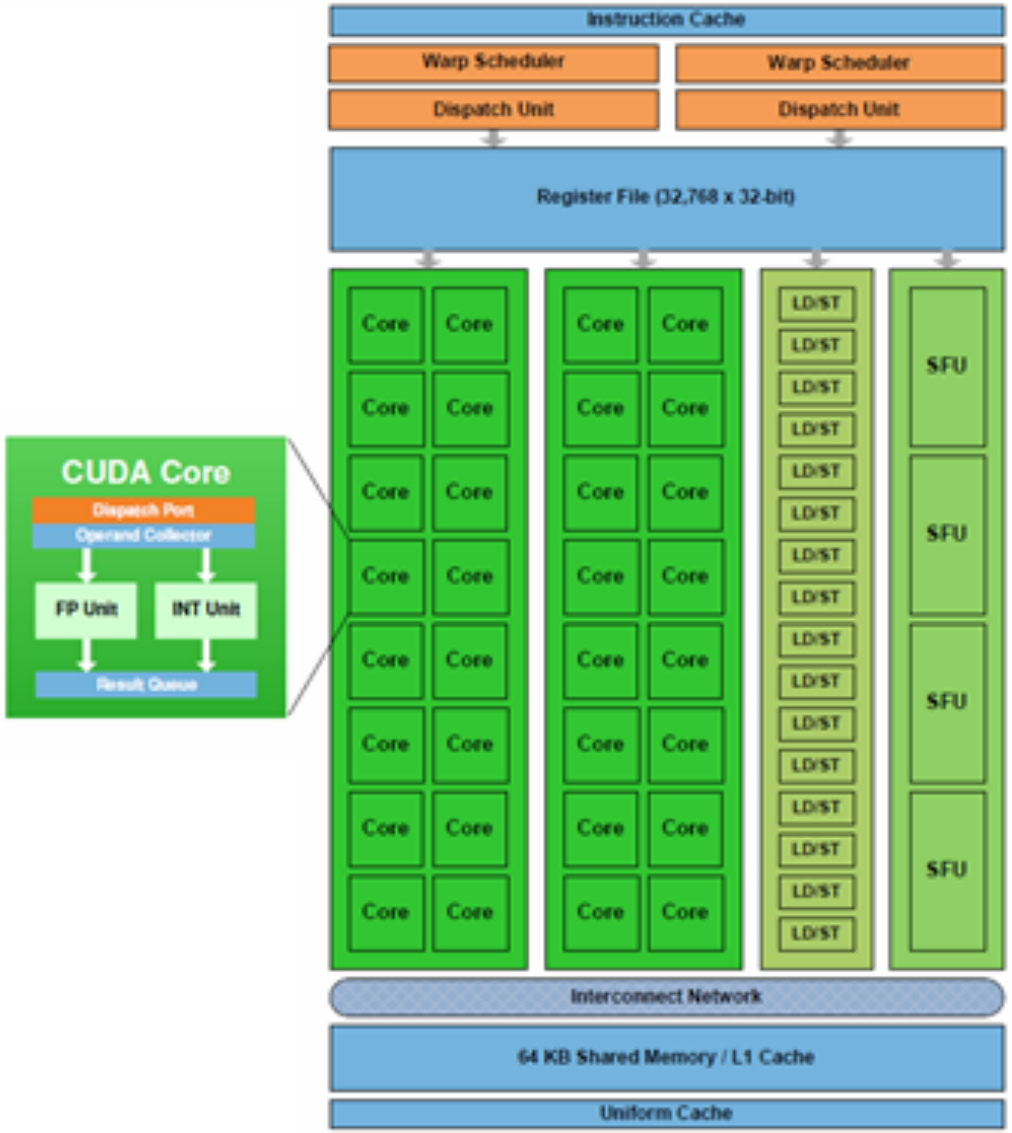
Extra resources

GPGPU-Sim 3.x Manual

http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual

GPU Microarchitecture Overview



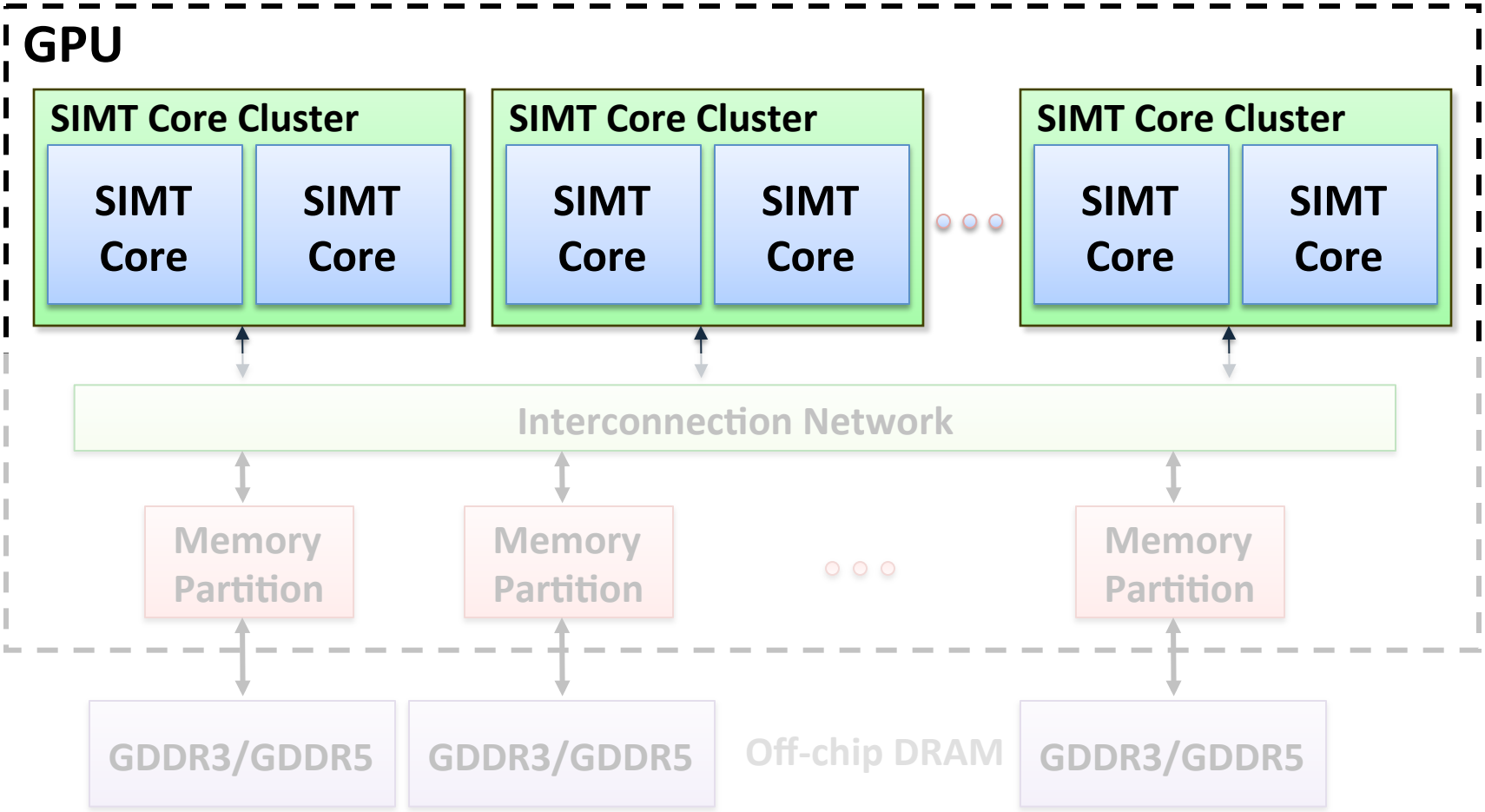


Fermi Streaming Multiprocessor (SM)

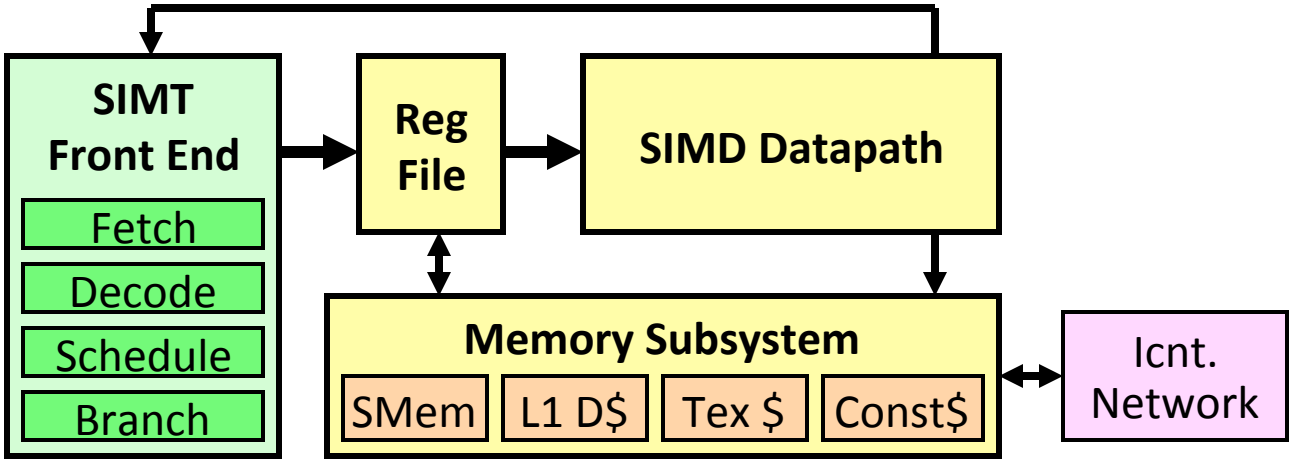
GPU Microarchitecture

- Companies tight lipped about details of GPU microarchitecture.
- Several reasons:
  - Competitive advantage
  - Fear of being sued by "non-practicing entities"
  - The people that know the details too busy building the next chip
- Model described next, embodied in GPGPU-Sim, developed from: white papers, programming manuals, IEEE Micro articles, patents.

GPU Microarchitecture Overview

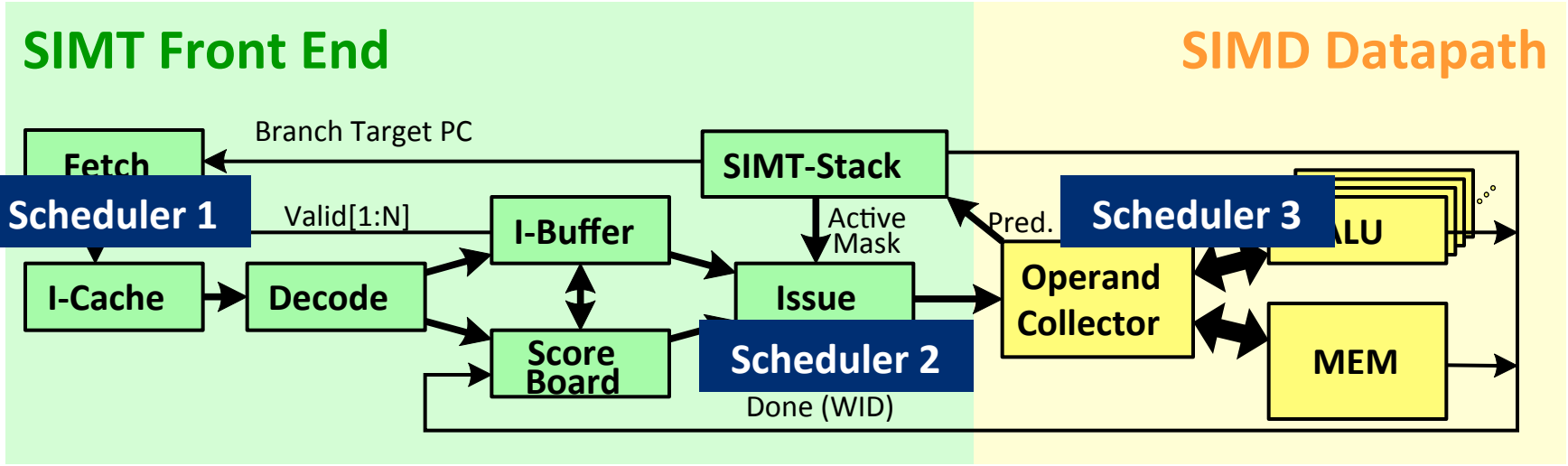


Inside a SIMT Core



- SIMT front end / SIMD backend
- Fine-grained multithreading
  - Interleave warp execution to hide latency
  - Register values of all threads stays in core

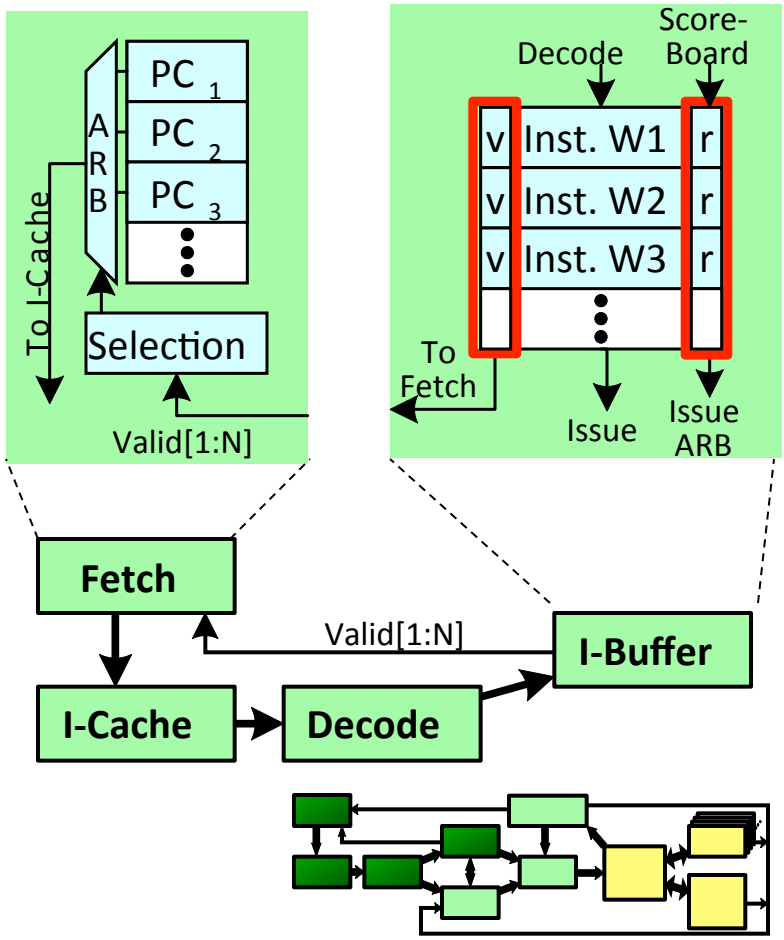
Inside an "NVIDIA-style" SIMT Core



- Three decoupled warp schedulers
- Scoreboard
- Large register file
- Multiple SIMD functional units

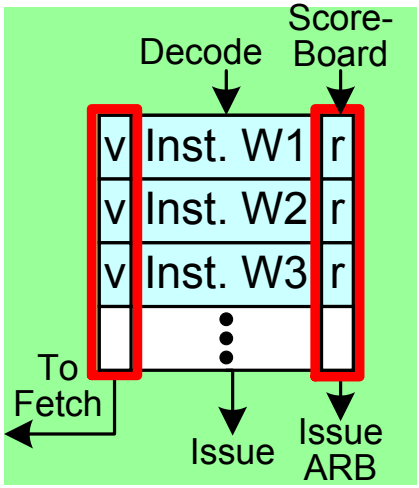
Fetch + Decode

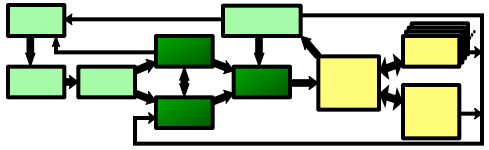
- Arbitrate the I-cache among warps
  - Cache miss handled by fetching again later
- Fetched instruction is decoded and then stored in the I-Buffer
  - 1 or more entries / warp
  - Only warp with vacant entries are considered in fetch



Instruction Issue

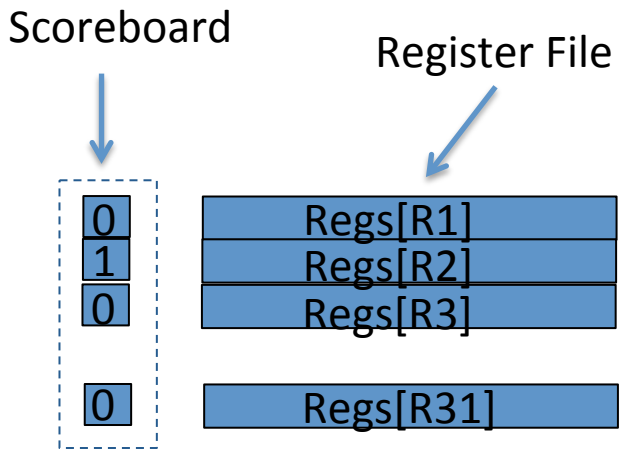
- Select a warp and issue an instruction from its I-Buffer for execution
  - Scheduling: Greedy-Then-Oldest (GTO)
  - GT200/later Fermi/Kepler: Allow dual issue (superscalar)
  - Fermi: Odd/Even scheduler
  - To avoid stalling pipeline might keep instruction in I-buffer until know it can complete (replay)





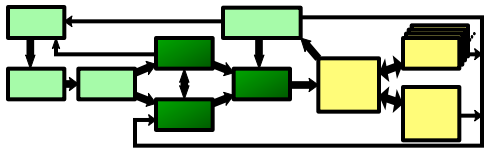
Review: In-order Scoreboard

- Scoreboard: a bit-array, 1-bit for each register
  - If the bit is *not* set: the register has valid data
  - If the bit is set: the register has stale data
    i.e., some outstanding instruction is going to change it
- Issue in-order: RD ← Fn (RS, RT)
  - If SB[RS] or SB[RT] is set → RAW, stall
  - If SB[RD] is set → WAW, stall
  - Else, dispatch to FU (Fn) and set SB[RD]
- Complete out-of-order
  - Update GPR[RD], clear SB[RD]



In-Order Scoreboard for GPUs?

- Problem 1: 32 warps, each with up to 128 (vector) registers per warp means scoreboard is 4096 bits.
- Problem 2: Warps waiting in I-buffer needs to have dependency updated every cycle.
- Solution?
  - Flag instructions with hazards as *not ready* in I-Buffer so not considered by scheduler
  - Track up to 6 registers per warp (out of 128)
  - I-buffer 6-entry bitvector: 1b per register dependency
  - Lookup source operands, set bitvector in I-buffer. As results written per warp, clear corresponding bit



Example

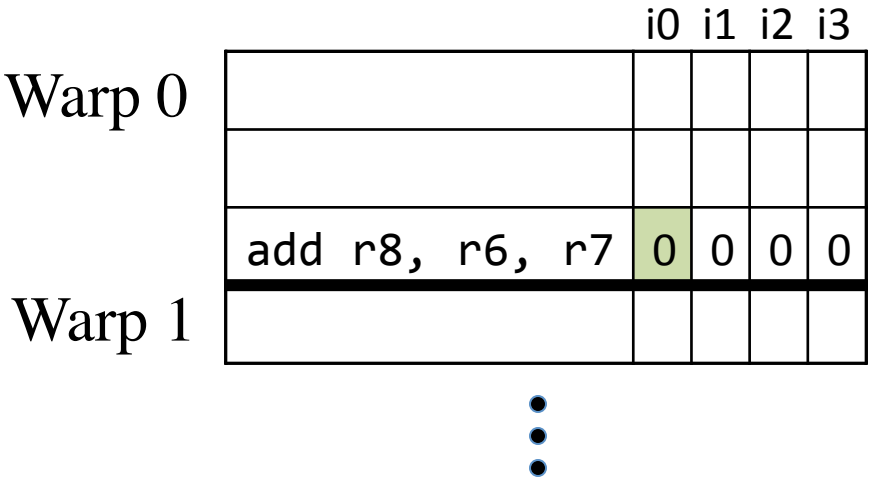
Code

```
ld  r7, [r0]
mul r6, r2, r5
add r8, r6, r7
```

Scoreboard

	Index 0	Index 1	Index 2	Index 3
Warp 0	-	-	r8	-
Warp 1	-	-	-	-

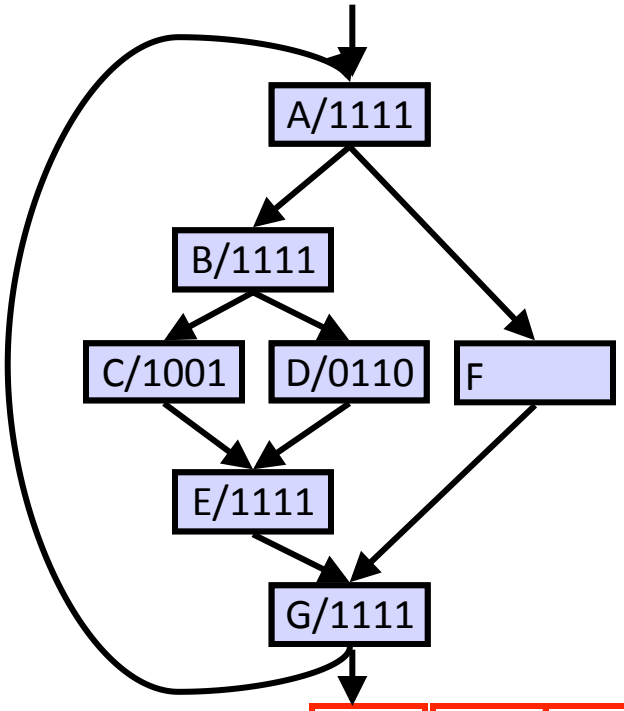
Instruction Buffer



SIMT Using a Hardware Stack

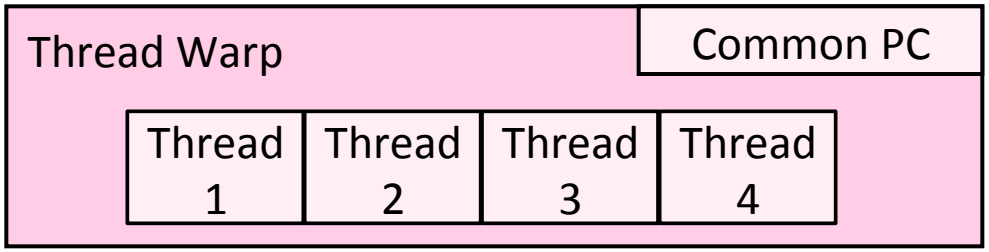
Stack approach invented at Lucafilm, Ltd in early 1980's

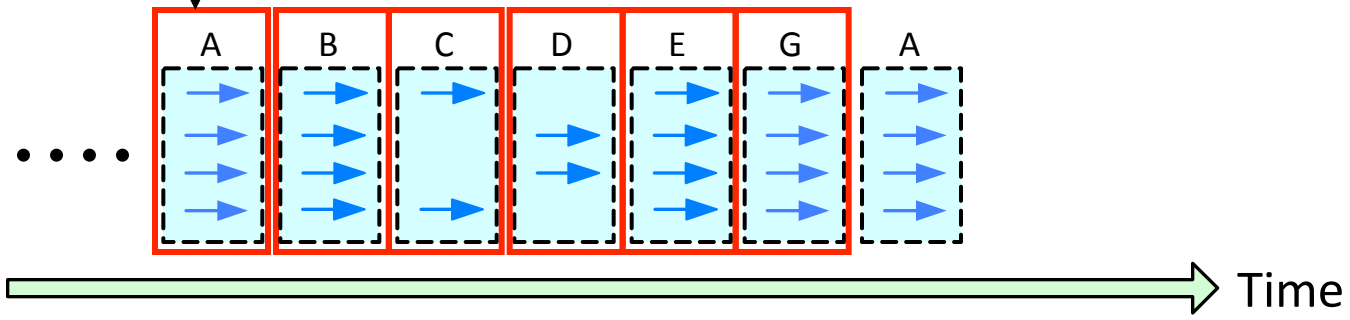
Version here from [Fung et al., MICRO 2007]



Stack

	Reconv. PC	Next PC	Active Mask
TOS →	-	E	1111
TOS →	E	D	0110
TOS →	E	E	1001





SIMT = SIMD Execution of Scalar Threads

SIMT Notes

- Execution mask stack implemented with special instructions to push/pop. Descriptions can be found in AMD ISA manual and NVIDIA patents.
- In practice augment stack with predication (lower overhead).

How is this done?

- Consider the following code:

```
if(X[i] != 0)
    X[i] = X[i] – Y[i];
else X[i] = Z[i];
```

- What does this compile to?

Example from Patterson and Hennesy's Computer Architecture: A Quantitative approach, fifth edition page 302

Implementation through predication

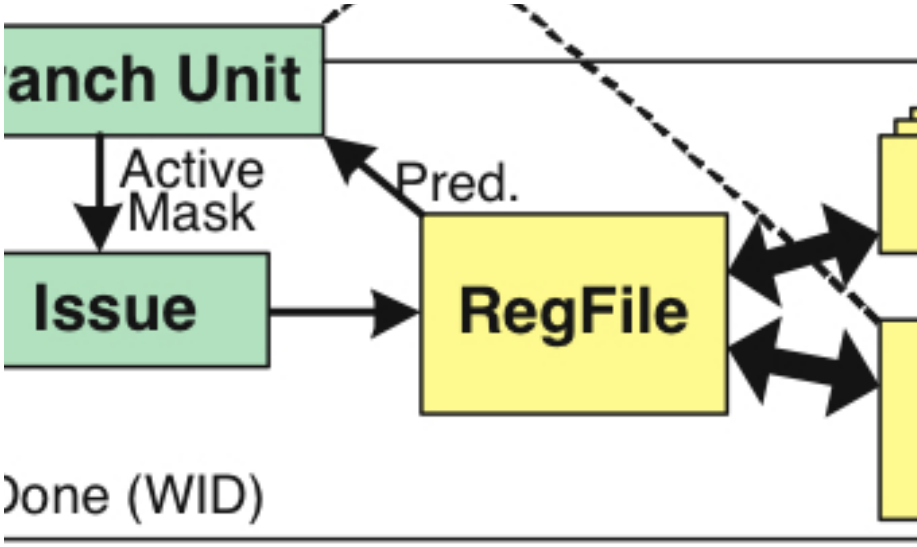
```
ld.global.f64     RD0, [X+R8]      ; RD0 = X[i]
setp.neq.s32    P1, RD0, #0        ; P1 is predicate register 1
@!P1, bra ELSE1, *Push             ; Push old mask, set new mask bits
                                   ; if P1 false, go to ELSE1
ld.global.f64     RD2, [Y+R8]      ; RD2 = Y[i]
sub.f64      RD0, RD0, RD2         ; RD0 = X[i] – Y[i]
st.global.f64    [X+R8], RD0       ; X[i] = RD0
@P1, bra ENDIF1, *Comp             ; complement mask bits
                                   ; if P1 true, go to ENDIF1
ELSE1:
ld.global.f64  RD0, [Z+R8]         ; RD0 = Z[i]
st.global.f64  [X+R8], RD0         ; X[i] = RD0
ENDIF1:
<next instruction>, *Pop           ; pop to restore old mask
```

SIMT outside of GPUs?

- ARM Research looking at SIMT-ized ARM ISA.
- Intel MIC implements SIMT on top of vector hardware via compiler (ISPC)
- Possibly other industry players in future

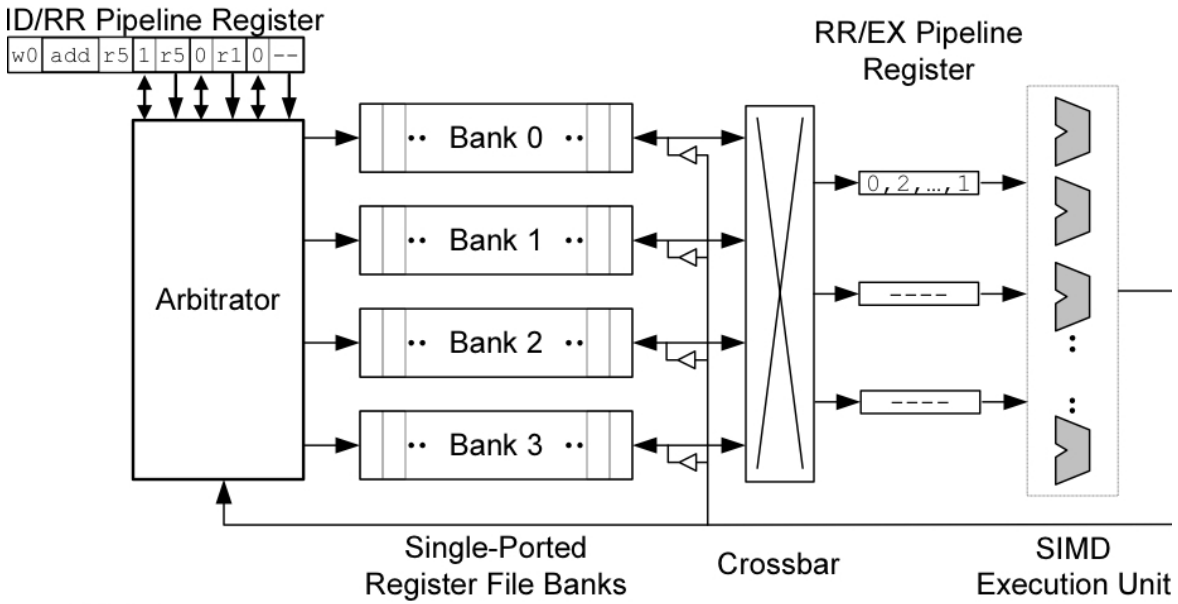
Register File

- 32 warps, 32 threads per warp, 16 x 32-bit registers per thread = 64KB register file.
- Need “4 ports” (e.g., FMA) greatly increase area.
- Alternative: banked single ported register file. How to avoid bank conflicts?



Banked Register File

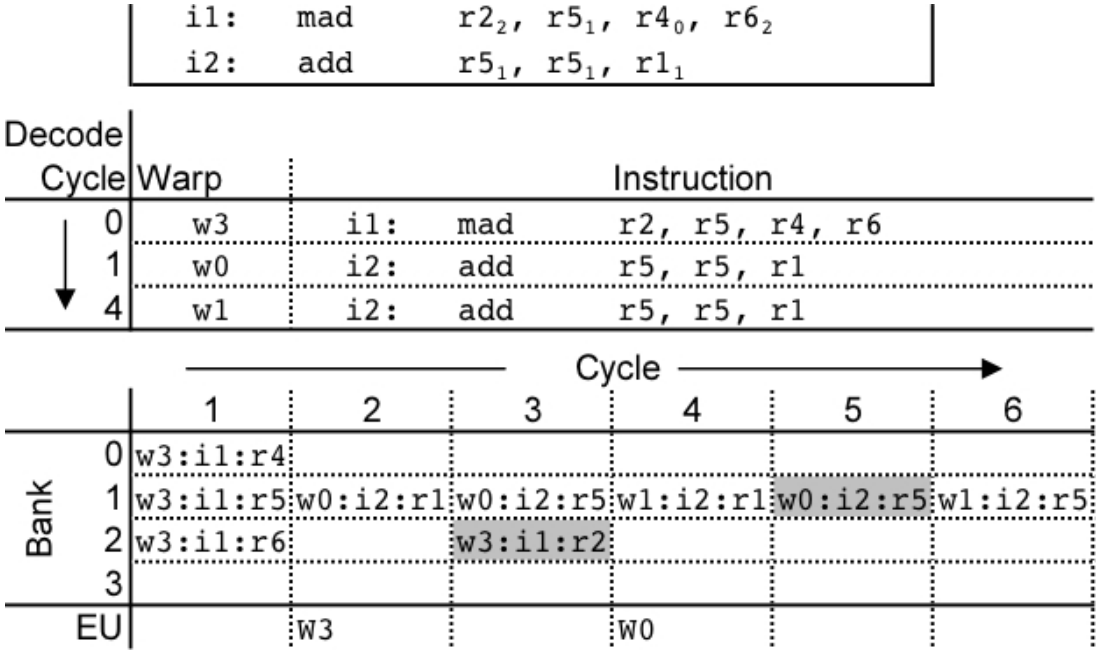
Strawman microarchitecture:



Register layout:

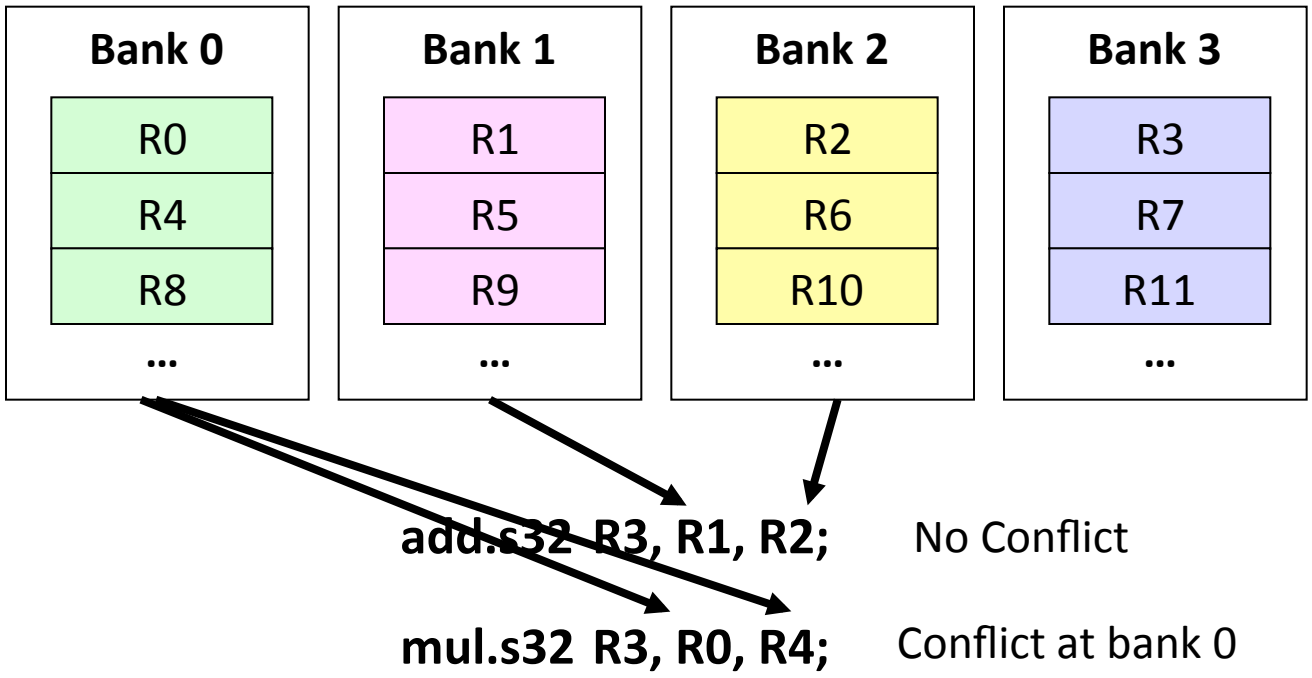
Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r4	w1:r5	w1:r6	w1:r7
w1:r0	w1:r1	w1:r2	w1:r3
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

Register Bank Conflicts

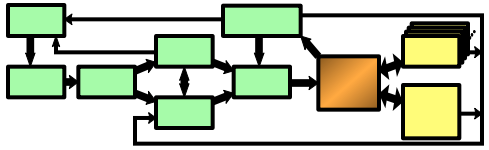


- warp 0, instruction 2 has two source operands in bank 1: takes two cycles to read.
- Also, warp 1 instruction 2 is same and is also stalled.
- Can use warp ID as part of register layout to help.

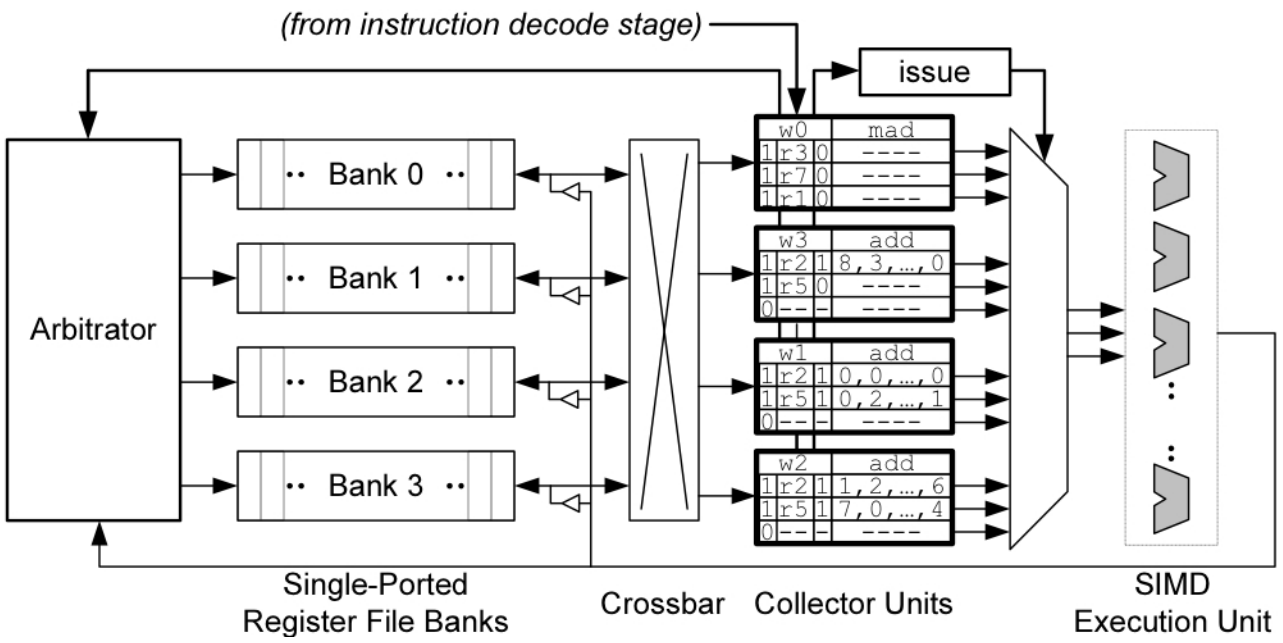
Operand Collector



- Term "Operand Collector" appears in figure in NVIDIA Fermi Whitepaper
- Operand Collector Architecture (US Patent: 7834881)
  - Interleave operand fetch from different threads to achieve full utilization



Operand Collector (1)



- Issue instruction to collector unit.
- Collector unit similar to reservation station in tomasulo's algorithm.
- Stores source register identifiers.
- Arbiter selects operand accesses that do not conflict on a given cycle.
- Arbiter needs to also consider writeback (or need read+write port)

Operand Collector (2)

- Combining swizzling and access scheduling can give up to ~ 2x improvement in throughput

```
i1:  add   r1, r2, r5
i2:  mad   r4, r3, r7, r1
```

Cycle	Warp	Instruction
0	w1	i1: add $r1_2$, $r2_3$, $r5_2$
1	w2	i1: add $r1_3$, $r2_0$, $r5_3$
2	w3	i1: add $r1_0$, $r2_1$, $r5_0$
3	w0	i2: mad $r4_0$, $r3_3$, $r7_3$, $r1_1$

Cycle →

Bank	1	2	3	4	5	6
0		w2:r2		w3:r5		w3:r1
1			w3:r2			
2		w1:r5		w1:r1		
3	w1:r2		w2:r5	w0:r3	w2:r1	w0:r7
EU			w1	w2	w3	

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r7	w1:r4	w1:r5	w1:r6
w1:r3	w1:r0	w1:r1	w1:r2
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

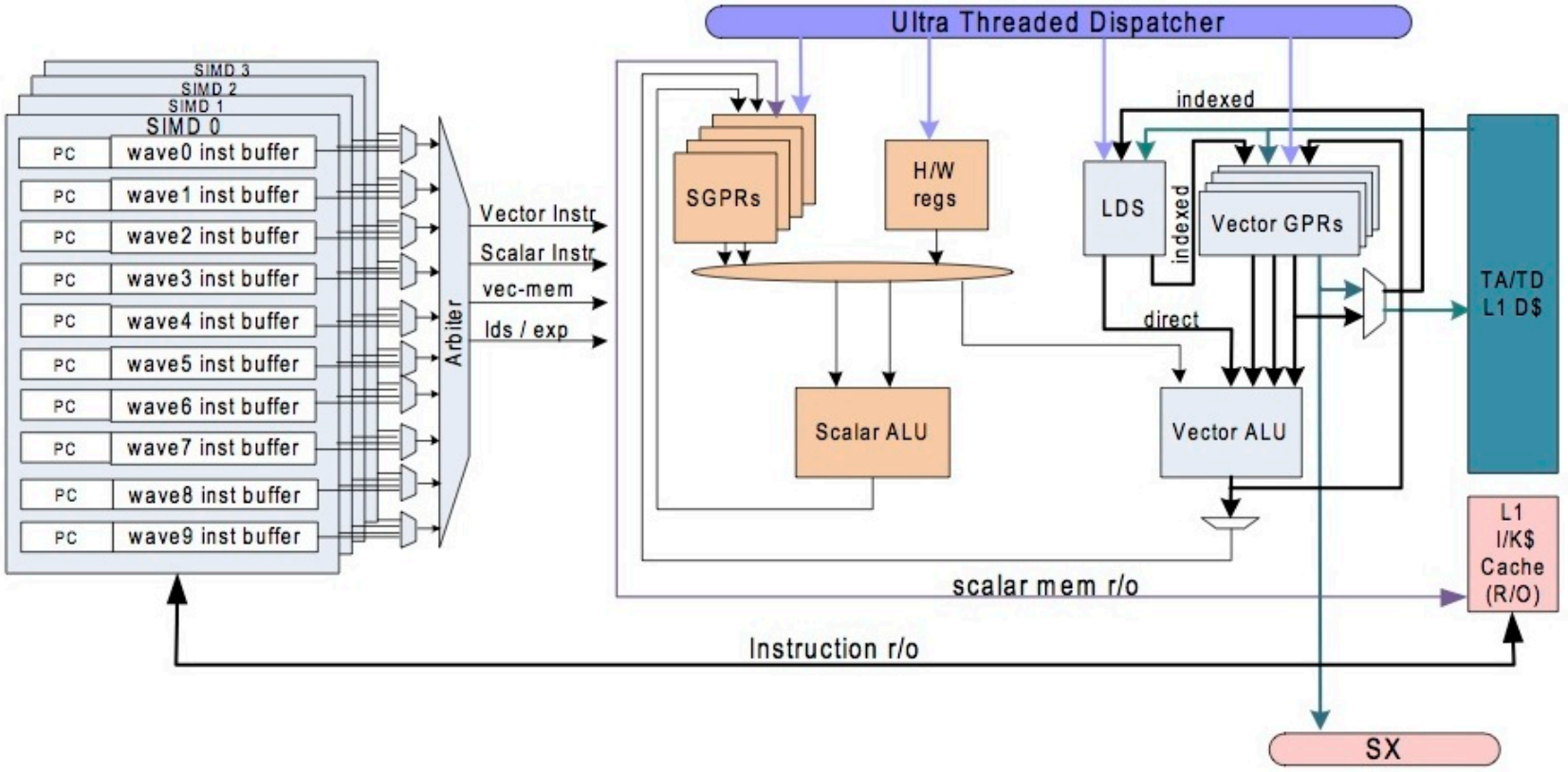
AMD Southern Islands

- SIMT processing often includes redundant computation across threads.

```
thread 0…31:
for( i=0; i < runtime_constant_N; i++ {
    /* do something with “i” */
}
```

AMD Southern Islands SIMT-Core

ISA visible scalar unit executes computation identical across SIMT threads in a wavefront



Example

```
float fn0(float a,float b)
{
   if(a>b)
      return (a * a – b);
   else
      return (b * b – a);
}
```

```
// Registers r0 contains "a", r1 contains "b"
// Value is returned in r2
    v_cmp_gt_f32 r0, r1 // a>b
    s_mov_b64 s0, exec  // Save current exec mask
    s_and_b64 exec, vcc, exec // Do "if"
    s_cbranch_vccz label0 // Branch if all lanes fail
    v_mul_f32 r2, r0, r0 // result = a * a
    v_sub_f32 r2, r2, r1 // result = result - b
label0:
    s_not_b64 exec, exec // Do "else"
    s_and_b64 exec, s0, exec // Do "else"
    s_cbranch_execz label1 // Branch if all lanes fail
    v_mul_f32 r2, r1, r1 // result = b * b
    v_sub_f32 r2, r2, r0 // result = result - a
label1:
    s_mov_b64 exec, s0   // Restore exec mask
```

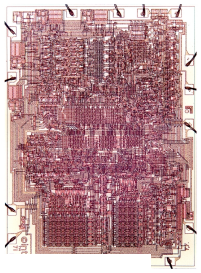
[Southern Islands Series Instruction Set Architecture, Aug. 2012]

Southern Islands SIMT Stack?

- Instructions: `S_CBRANCH_*_FORK; S_CBRANCH_JOIN`
- Use for arbitrary (e.g., irreducible) control flow
- 3-bit control stack pointer
- Six 128-bit stack entries; stored in scalar general purpose registers holding {exec[63:0], PC[47:2]}
- `S_CBRANCH_*_FORK` executes path with fewer active threads first

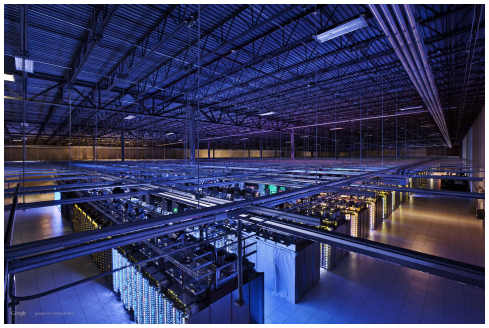
Part 3: Research Directions

Decreasing cost per unit computation



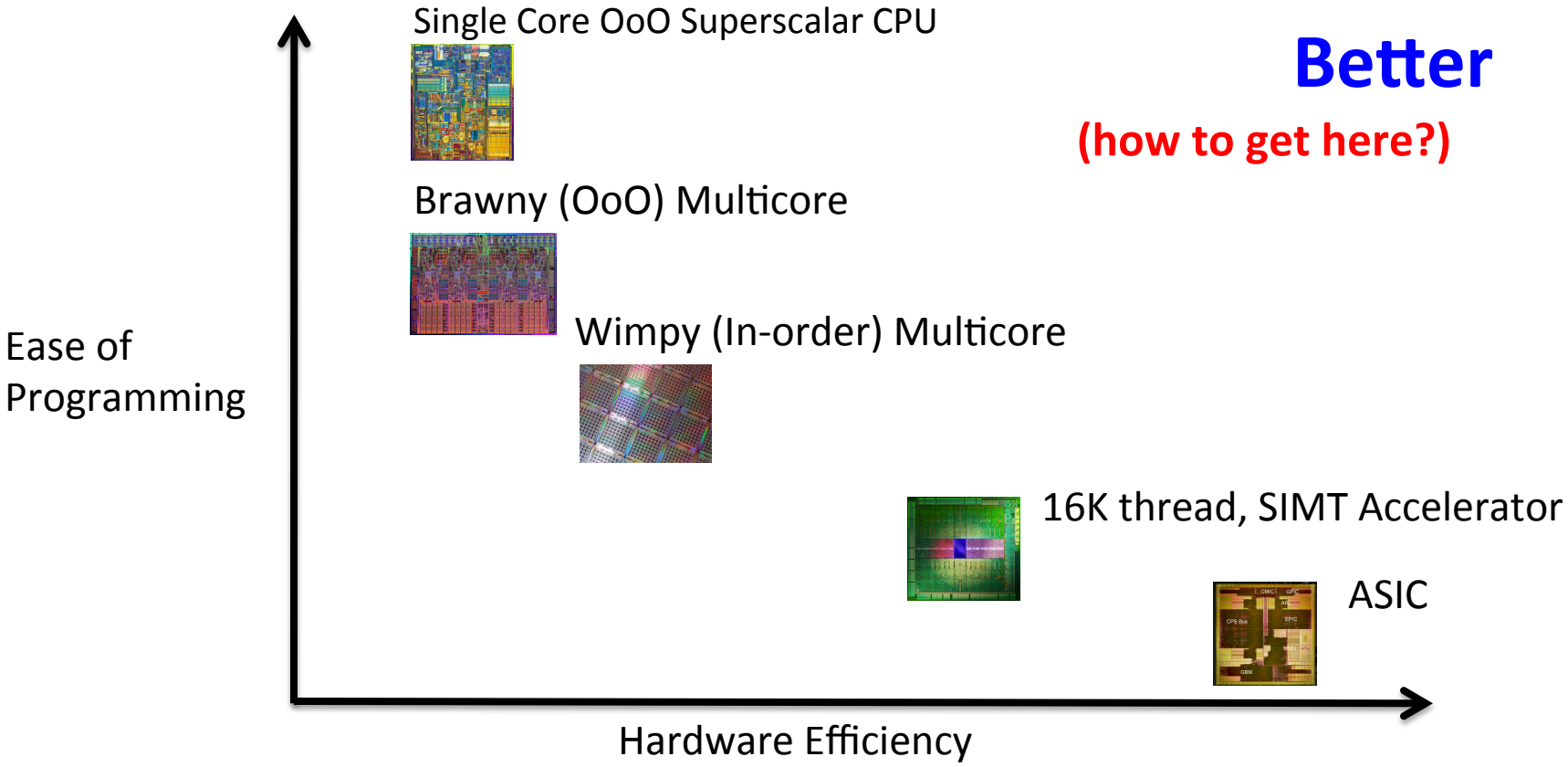
1971: Intel 4004

2007: iPhone



2012: Google datacenter

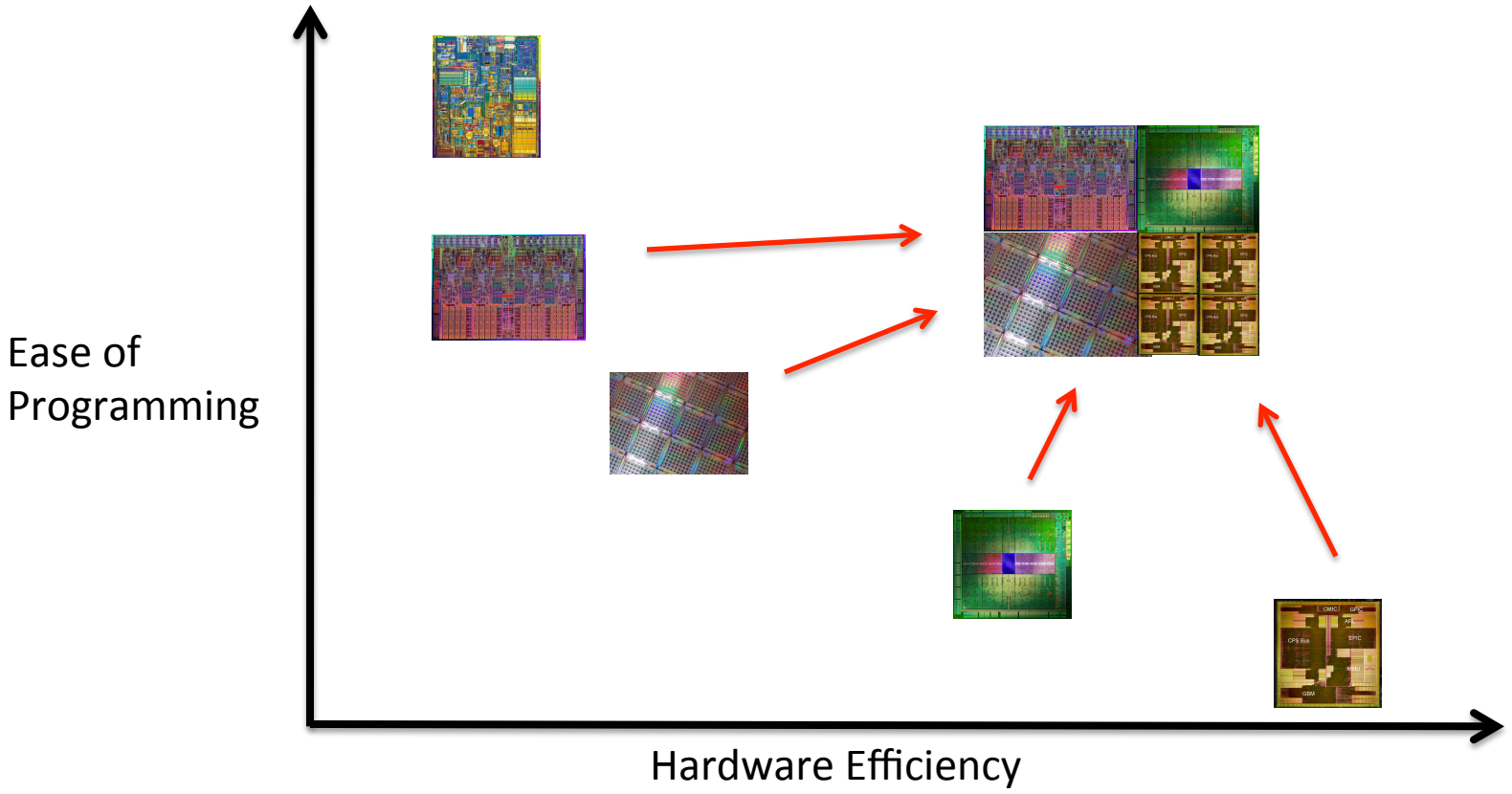
1981: IBM 5150



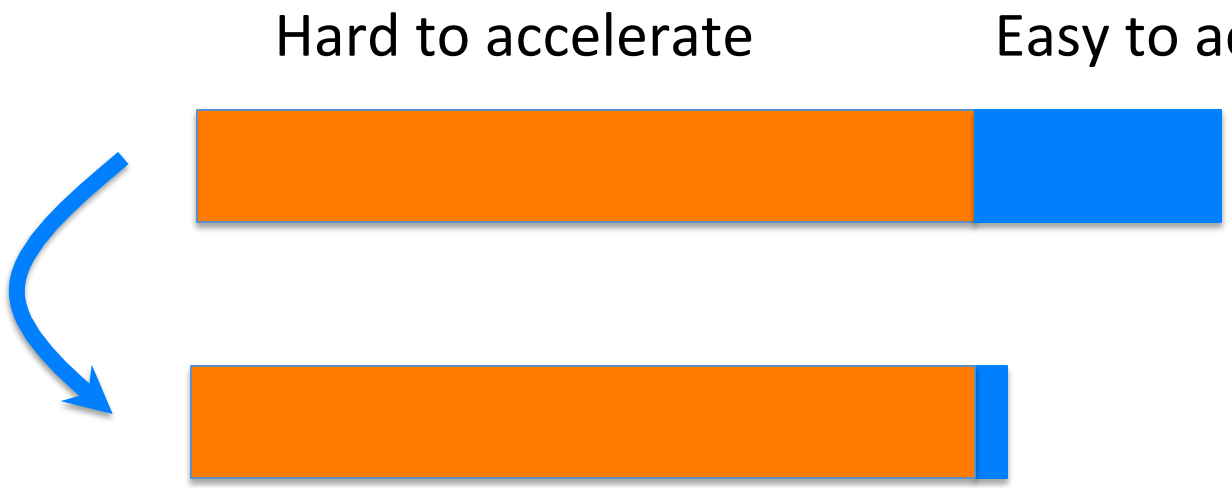
Better

(how to get here?)

Start by using right tool for each job…

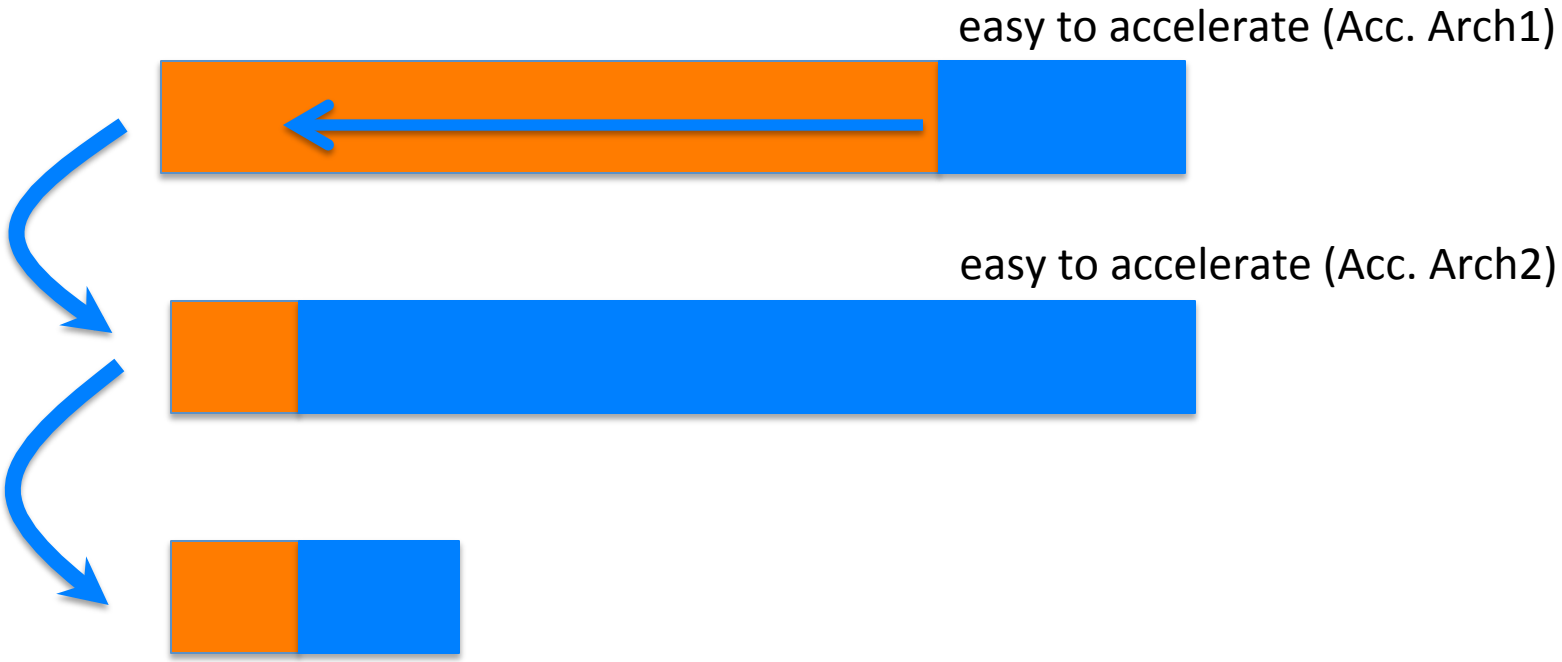


Amdahl's Law Limits this Approach



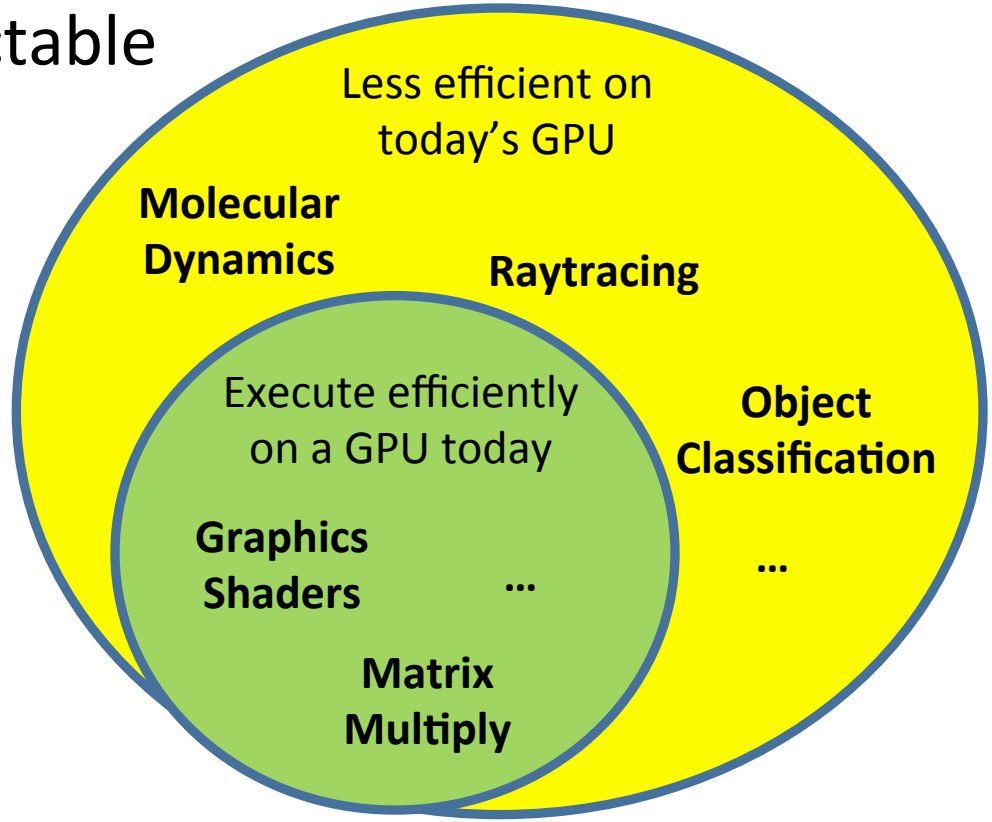
$$\text{Improvement}_{\text{overall}} = \frac{1}{\text{Fraction}_{\text{hard}} + \frac{1 - \text{Fraction}_{\text{hard}}}{\text{Improvement}_{\text{easy}}}}$$

Question: Can dividing line be moved?



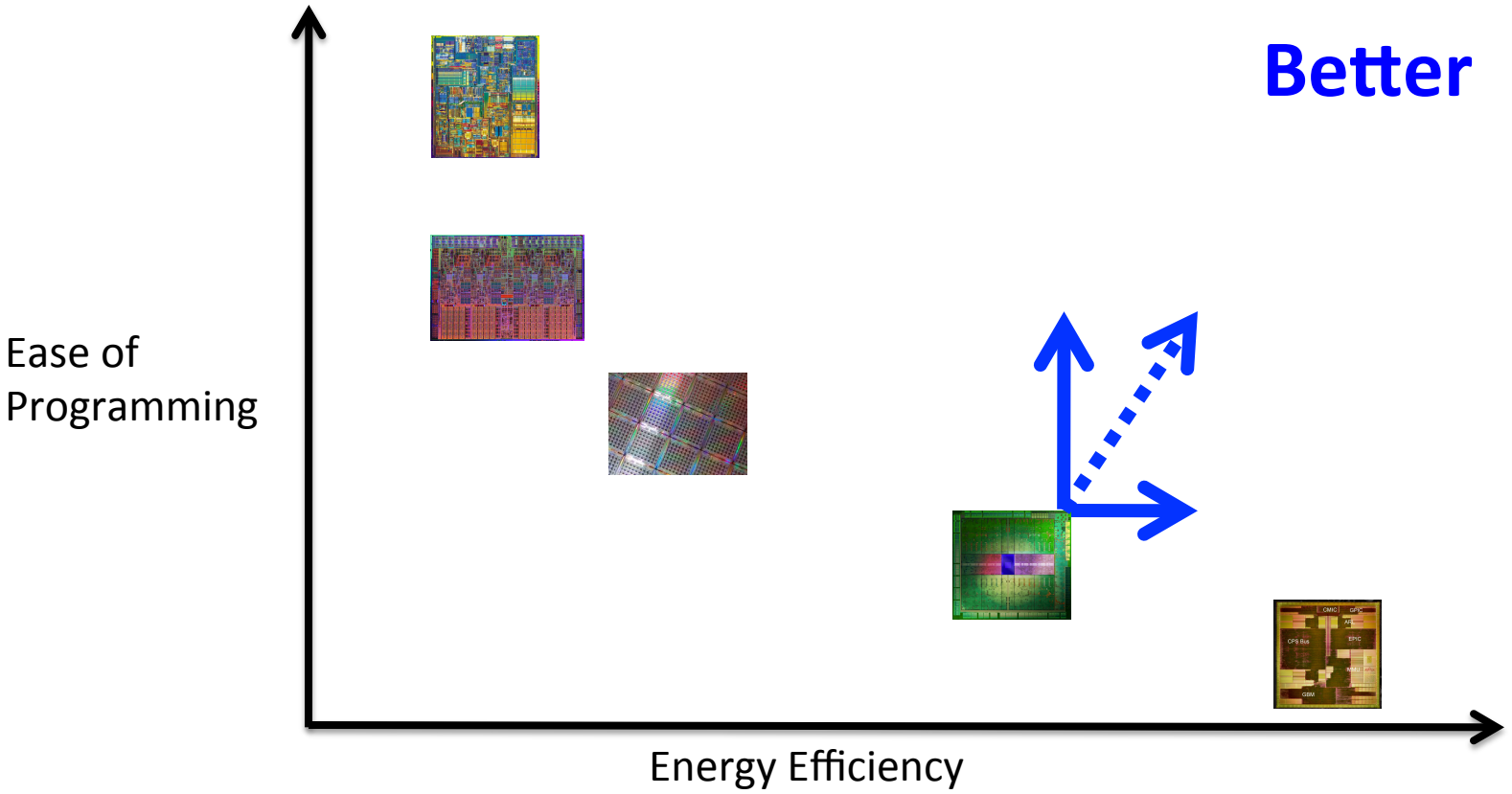
Forward-Looking GPU Software

- Still Massively Parallel
- Less Structured
  - Memory access and control flow patterns are less predictable



[Tim Rogers]

Two Routes to "Better"



Research Direction 1:
Mitigating SIMT Control Divergence

Research Direction 2: Mitigating High GPGPU Memory Bandwidth Demands

Reducing Off-Chip Access / Divergence

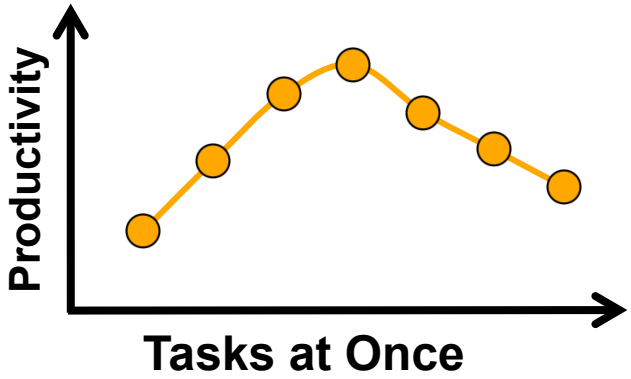
- Re-writing software to use “shared memory” and avoid uncoalesced global accesses is bane of GPU programmer existence.
- Recent GPUs introduce caches, but large number of warps/wavefronts lead to thrashing.

- NVIDIA: Register file cache (ISCA 2011, MICRO)
  - Register file burns significant energy
  - Many values read once soon after written
  - Small register file cache captures locality and saves energy but does not help performance
  - Recent follow on work from academia
- Prefetching (Kim, MICRO 2010)
- Interconnect (Bakhoda, MICRO 2010)
- Lee & Kim (HPCA 2012) CPU/GPU cache sharing

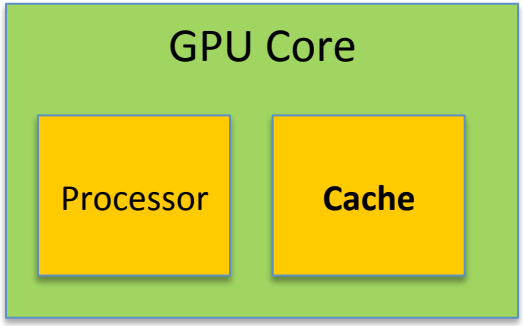
Thread Scheduling Analogy

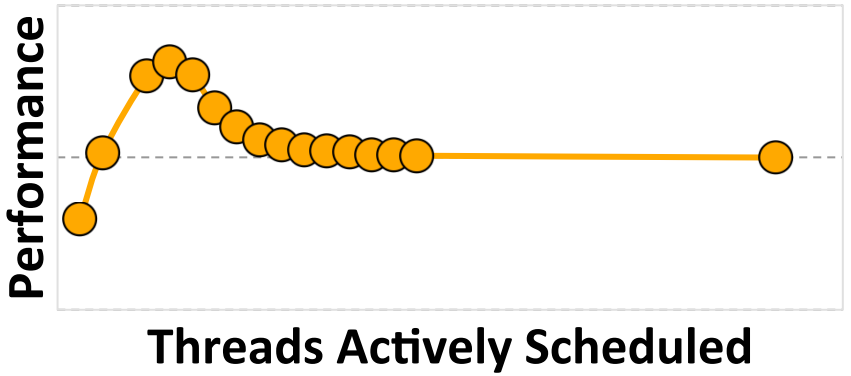
[MICRO 2012]

- Human Multitasking
  - Humans have limited **attention capacity**



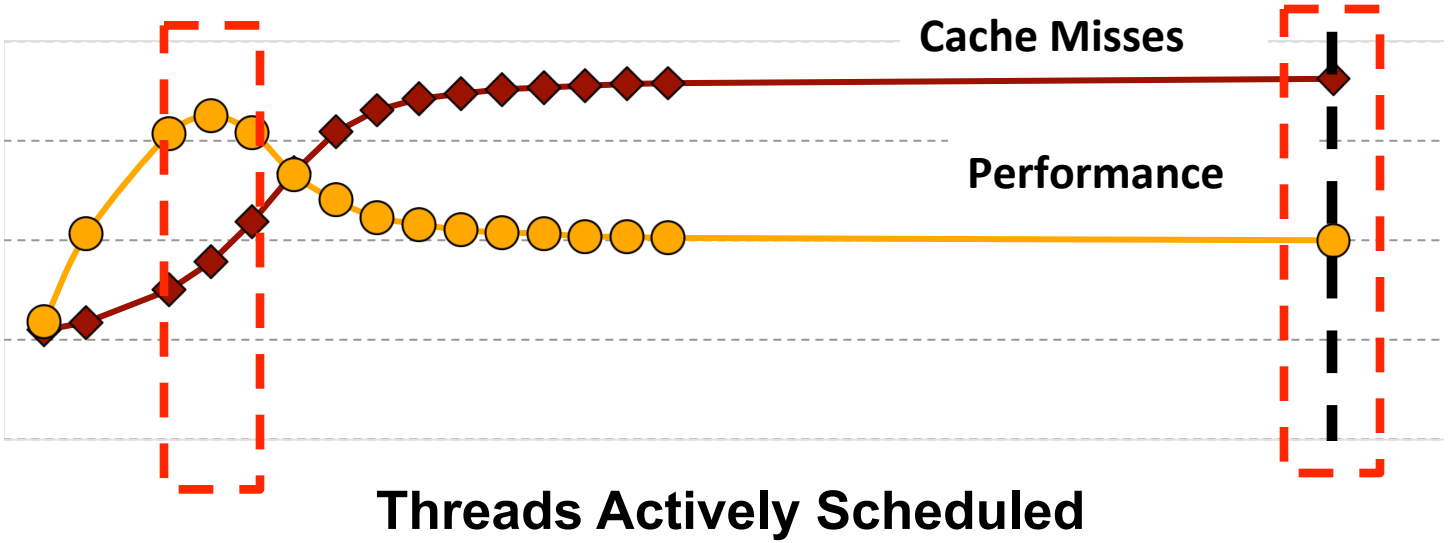
  - GPUs have limited **cache capacity**

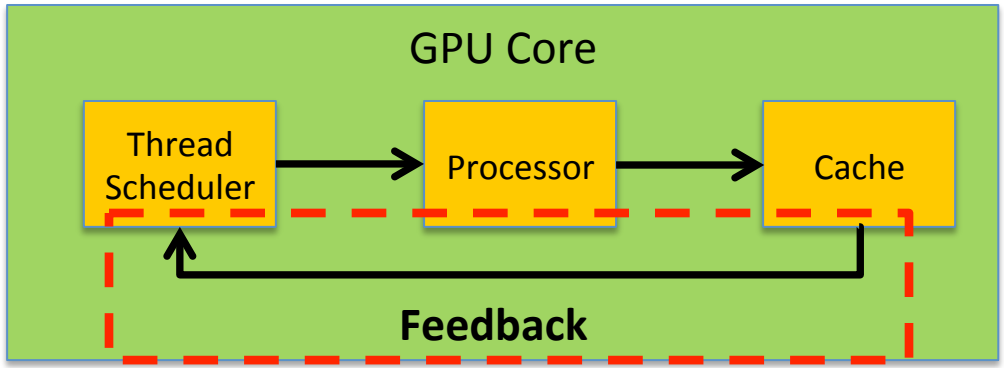




Use Memory System Feedback

[MICRO 2012]





Research Direction 3:
Coherent Memory for Accelerators

Why GPU Coding Difficult?

- Manual data movement CPU ⇔ GPU
- Lack of generic I/O , system support on GPU
- Need for performance tuning to reduce
  - off-chip accesses
  - memory divergence
  - control divergence
- For complex algorithms, synchronization
- Non-deterministic behavior for buggy code
- Lack of good performance analysis tools

Manual CPU ⇔ GPU Data Movement

- **Problem #1:** Programmer needs to identify data needed in a kernel and insert calls to move it to GPU
- **Problem #2:** Pointer on CPU does not work on GPU since different address spaces
- **Problem #3:** Bandwidth connecting CPU and GPU is order of magnitude smaller than GPU off-chip
- **Problem #4:** Latency to transfer data from CPU to GPU is order of magnitude higher than GPU off-chip
- **Problem #5:** Size of GPU DRAM memory much smaller than size of CPU main memory

Identifying data to move CPU ⇔ GPU

- CUDA/OpenCL: Job of programmer ☹
- C++AMP passes job to compiler.
- OpenACC uses pragmas to indicate loops that should be offloaded to GPU.

Memory Model

Rapid change (making programming easier)

- Late 1990's: fixed function graphics only
- 2003: programmable graphics shaders
- 2006: + global/local/shared (GeForce 8)
- 2009: + caching of global/local
- 2011: + unified virtual addressing
- 2014: + unified memory / coherence

Caching

- Scratchpad uses explicit data movement. Extra work. Beneficial when reuse pattern statically predictable.

- NVIDIA Fermi / AMD Southern Island add caches for accesses to global memory space.

CPU memory vs. GPU global memory

- Prior to CUDA: input data is texture map.
- CUDA 1.0 introduces cudaMemcpy
  - Allows copy of data between CPU memory space to global memory on GPU
- Still has problems:
  - #1: Programmer still has to think about it!
  - #2: Communicate only at kernel grid boundaries
  - #3: Different virtual address space
    - pointer on CPU not a pointer on GPU => cannot easily share complex data structures between CPU and GPU

Fusion / Integrated GPUs

- Why integrate?
  - One chip versus two (cf. Moore's Law, VLSI)
  - Latency and bandwidth of communication: shared physical address space, even if off-chip, eliminates copy: AMD Fusion. 1st iteration 2011. Same DRAM
  - Shared virtual address space? (AMD Kavari 2014)
  - Reduce latency to spawn kernel means kernel needs to do less to justify cost of launching

CPU Pointer not a GPU Pointer

- NVIDIA Unified Virtual Memory partially solves the problem but in a bad way:
  - GPU kernel reads from CPU memory space
- NVIDIA Uniform Memory (CUDA 6) improves by enabling automatic migration of data
- Limited academic work. Gelado et al. ASPLOS 2010.

CPU ⇔ GPU Bandwidth

- Shared DRAM as found in AMD Fusion (recent Core i7) enables the elimination of copies from CPU to GPU. Painful coding as of 2013.

- One question how much benefit versus good coding. Our limit study (WDDD 2008) found only ~50% gain. Lustig & Martonosi HPCA 2013.

- Algorithm design—MummerGPU++

CPU ⇔ GPU Latency

- NVIDIA's solution: CUDA Streams. Overlap GPU kernel computation with memory transfer. Stream = ordered sequence of data movement commands and kernels. Streams scheduled independently. Very painful programming.
- Academic work: Limit Study (WDDD 2008), Lustig & Martonosi HPCA 2013, Compiler data movement (August, PLDI 2011).

GPU Memory Size

- CUDA Streams

- Academic work: Treat GPU memory as cache on CPU memory (Kim et al., ScaleGPU, IEEE CAL early access).

Solution to all these sub-issues?

- Heterogeneous System Architecture: Integrated CPU and GPU with coherence memory address space.

- Need to figure out how to provide coherence between CPU and GPU.
- Really two problems: Coherence within GPU and then between CPU and GPU.

Research Direction 4: Easier Programming with Synchronization

Synchronization

- Locks are not encouraged in current GPGPU programming manuals.
- Interaction with SIMT stack can easily cause deadlocks:

```
while( atomicCAS(&lock[a[tid]],0,1) != 0 )
  ;  // deadlock here if a[i] = a[j] for any i,j = tid in warp

// critical section goes here

atomicExch (&lock[a[tid]], 0) ;
```

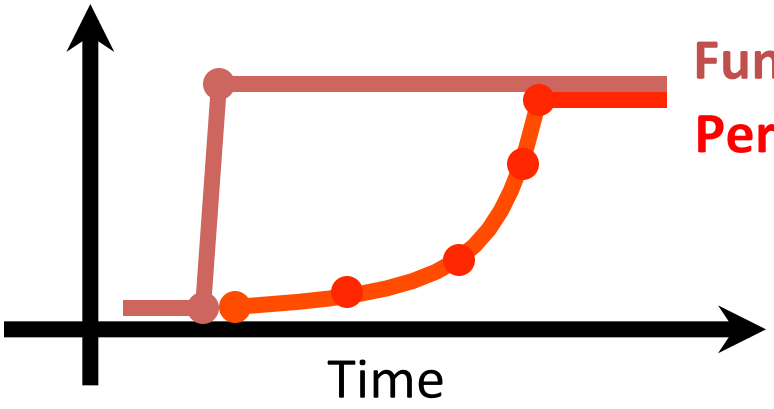
Correct way to write critical section for GPGPU:

```
done = false;
while( !done ) {
  if( atomicCAS (&lock[a[tid]], 0 , 1 )==0 ) {

    // critical section goes here

    atomicExch(&lock[a[tid]], 0 ) ;
  }
}
```

Most current GPGPU programs use barriers within thread blocks and/or lock-free data structures.

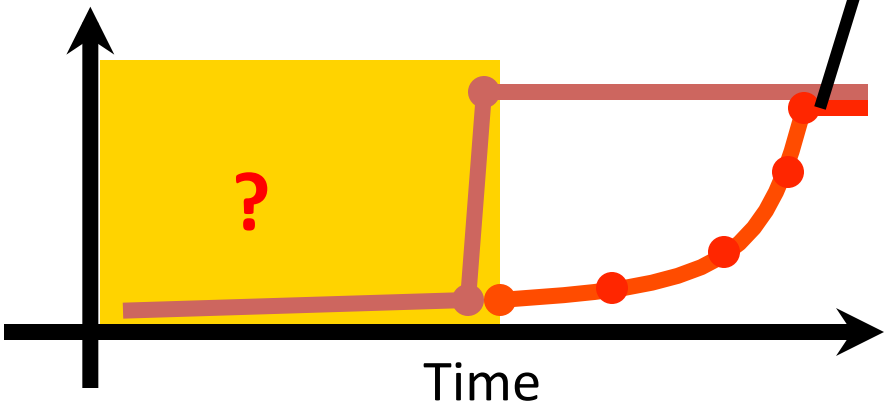
This leads to the following picture…

- Lifetime of GPU Application Development

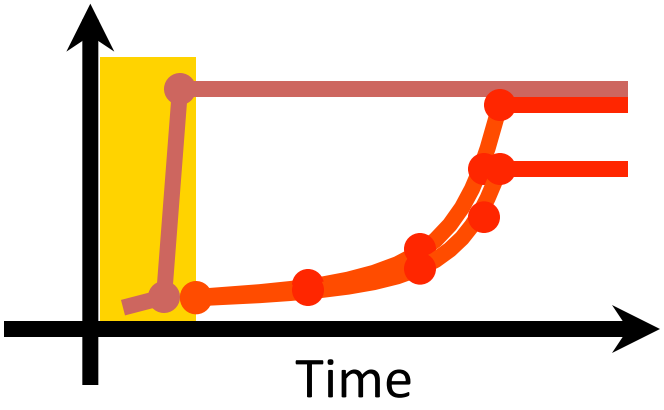


E.g. N-Body with 5M bodies
CUDA SDK: $O(n^2)$ – 1640 s (barrier)
Barnes Hut: O(nLogn) – 5.2 s (locks)

Fine-Grained Locking/Lock-Free



Transactional Memory

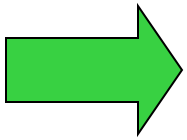


Transactional Memory

- Programmer specifies atomic code blocks called transactions [Herlihy'93]

Lock Version:

```
Lock(X[a]);
Lock(X[b]);
Lock(X[c]);
  X[c] = X[a]+X[b];
Unlock(X[c]);
Unlock(X[b]);
Unlock(X[a]);
```



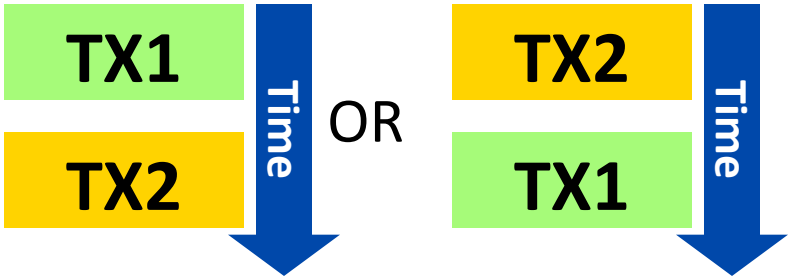
TM Version:

```
atomic {
  X[c] = X[a]+X[b];
}
```

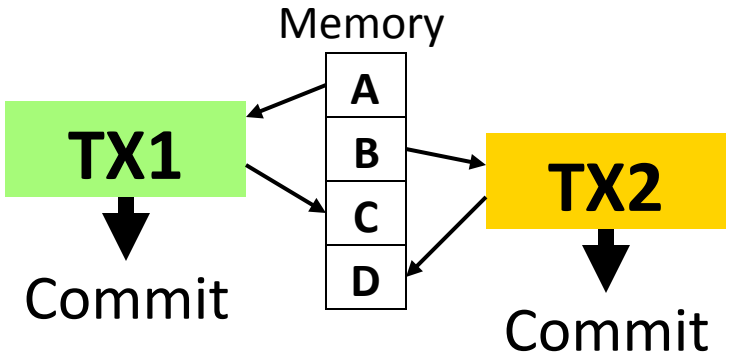
Potential Deadlock!

Transactional Memory

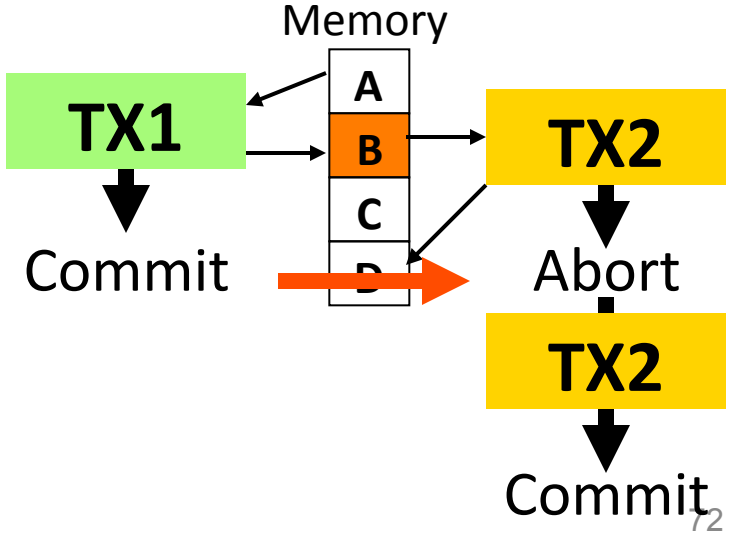
Programmers' View:



Non-conflicting transactions may run in parallel



Conflicting transactions automatically serialized



Are TM and GPUs Incompatible?

GPU uarch very different from multicore CPU…

KILO TM [MICRO'11, IEEE Micro Top Picks]

- Hardware TM for GPUs
- Half performance of fine grained locking
- Chip area overhead of 0.5%

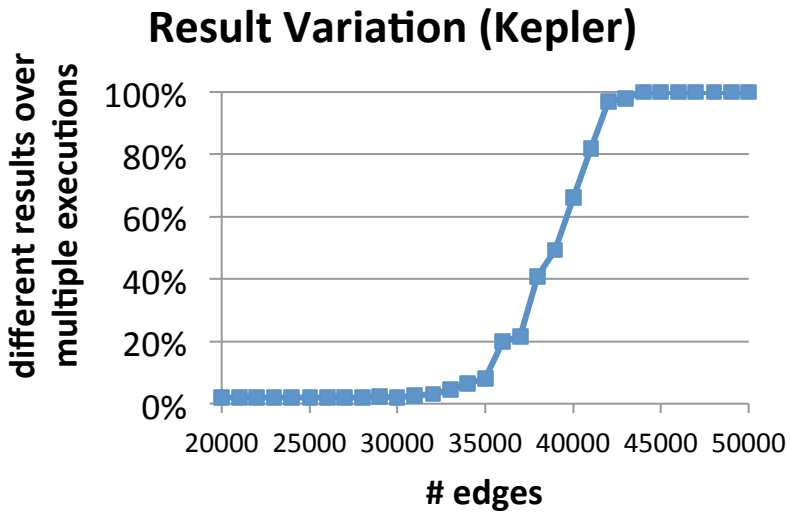
Research Direction 5:
GPU Power Efficiency

GPU power

- More efficient than CPU but
  - Consumes a lot of power
  - Much less efficient than ASIC or FPGAs
  - What can be done to reduce power consumption?
- Look at the most power hungry components
  - What can be duty cycled/power gated?
  - GPUWattch to evaluate ideas

Other Research Directions….

- Non-deterministic behavior for buggy code
  - GPUDet ASPLOS 2013



- Lack of good performance analysis tools
  - NVIDIA Profiler/Parallel NSight
  - AerialVision [ISPASS 2010]
  - GPU analytical perf/power models (Hyesoon Kim)

Lack of I/O and System Support…

- Support for printf, malloc from kernel in CUDA
- File system I/O?
- GPUfs (ASPLOS 2013):
  - POSIX-like file system API
  - One file per warp to avoid control divergence
  - Weak file system consistency model (close->open)
  - Performance API: O_GWRONCE, O_GWRONCE
  - Eliminate seek pointer
- GPUnet (OSDI 2014): Posix like API for sockets programming on GPGPU.

Conclusions

- GPU Computing is growing in importance due to energy efficiency concerns
- GPU architecture has evolved quickly and likely to continue to do so
- We discussed some of the important microarchitecture bottlenecks and recent research.
- Also discussed some directions for improving programming model