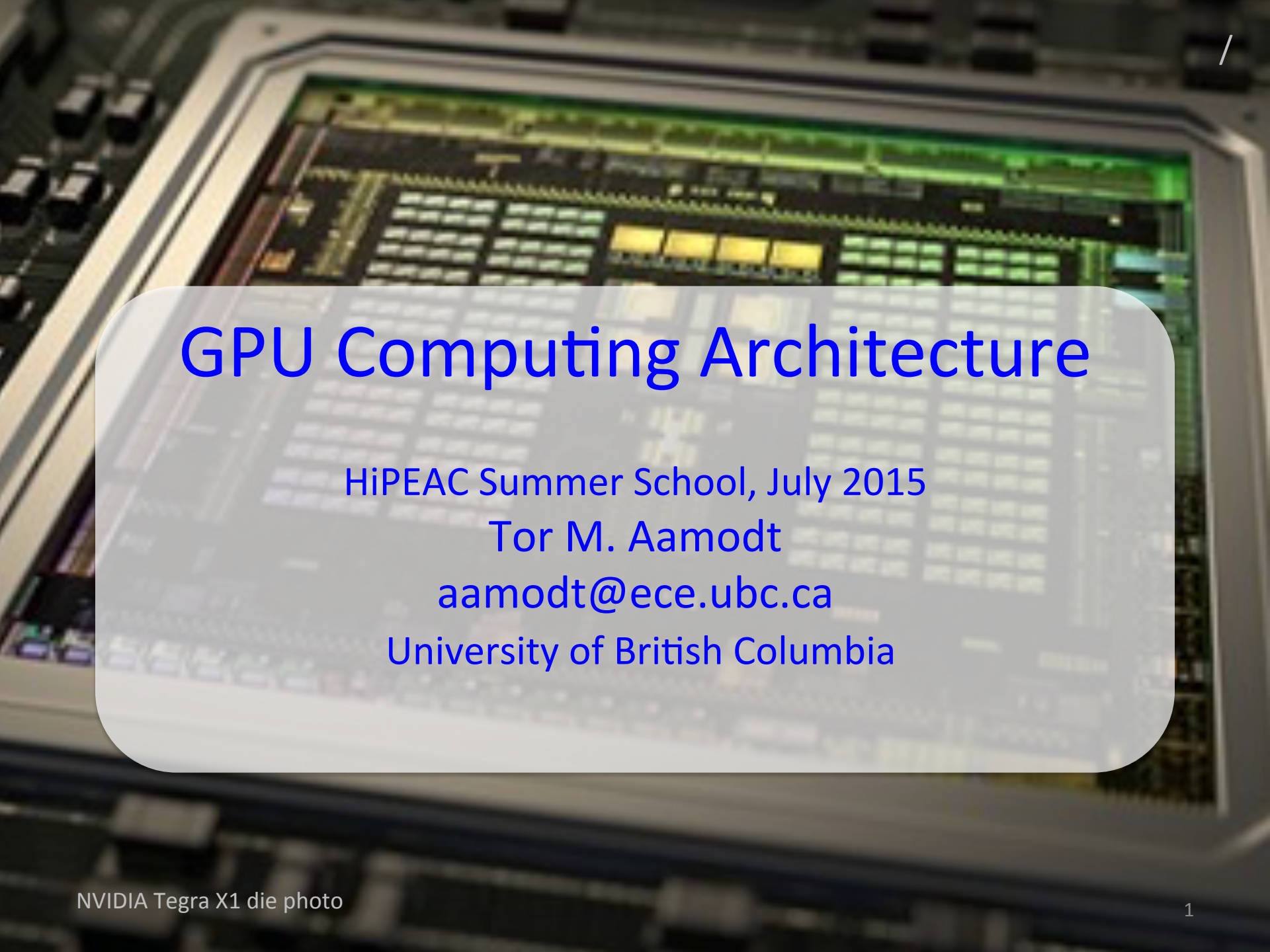


GPU Computing Architecture

HiPEAC Summer School, July 2015

Tor M. Aamodt
aamodt@ece.ubc.ca
University of British Columbia

NVIDIA Tegra X1 die photo

What is a GPU?

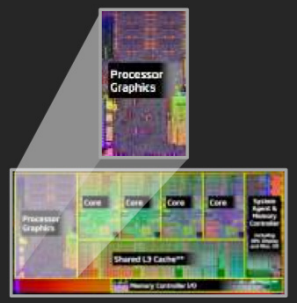
- GPU = Graphics Processing Unit
  - Accelerator for raster based graphics (OpenGL, DirectX)
  - Highly programmable (Turing complete)
  - Commodity hardware
  - 100's of ALUs; 10's of 1000s of concurrent threads

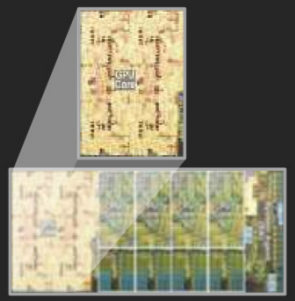
The GPU is Ubiquitous

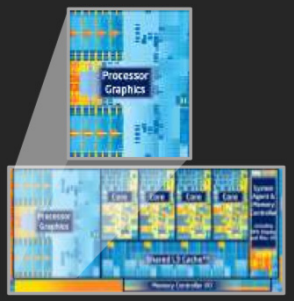
THE FUTURE BELONGS TO THE APU:
BETTER GRAPHICS, EFFICIENCY AND COMPUTE

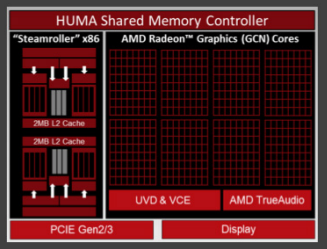
AMD

"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:

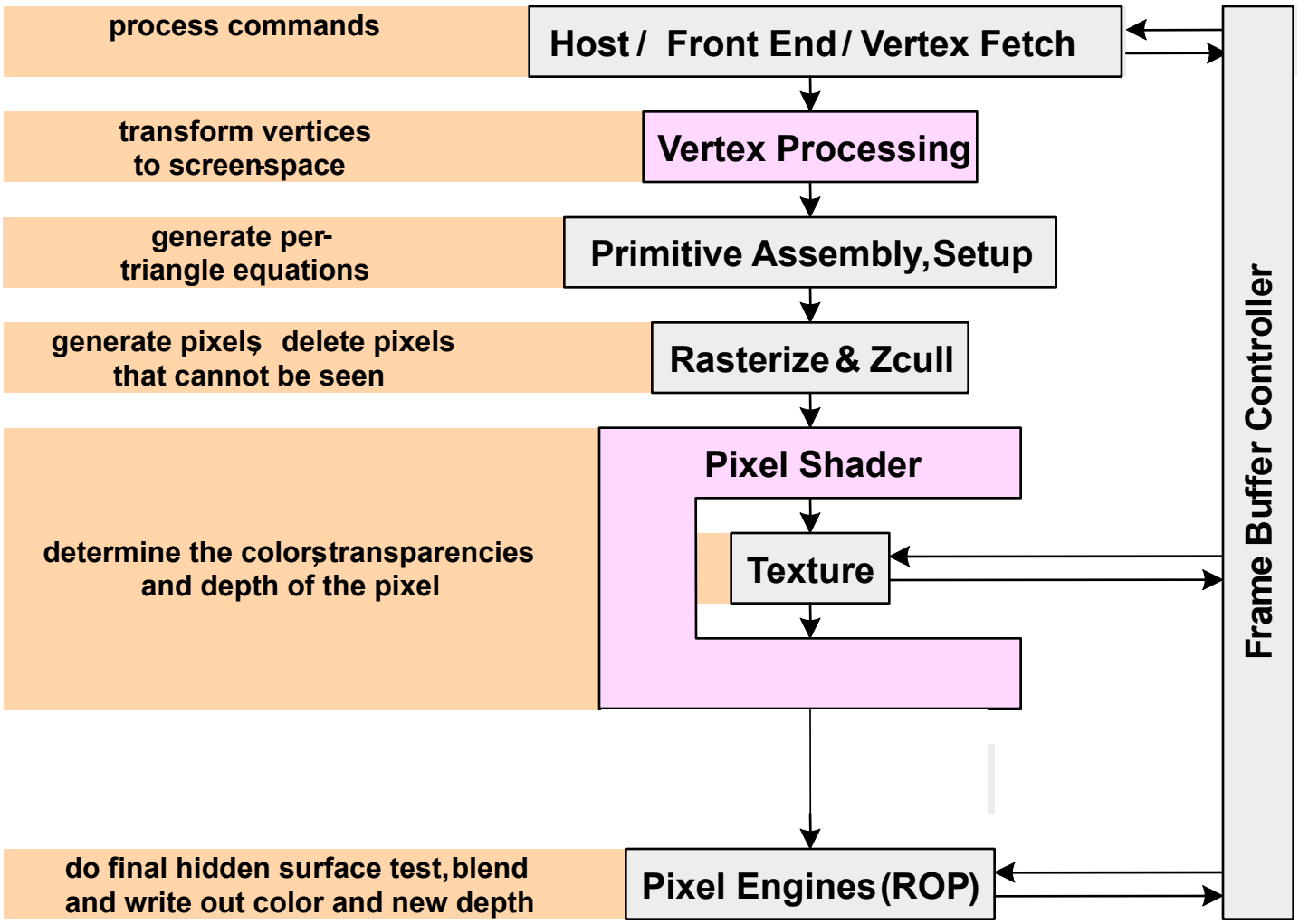
- **Compute**
  - (OpenCL™, Direct Compute)
- **Gaming**
  - (DirectX®, OpenGL, Mantle)
- **Experiences**
  - (Audio, Ultra HD, Devices, New Interactivity)

[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)

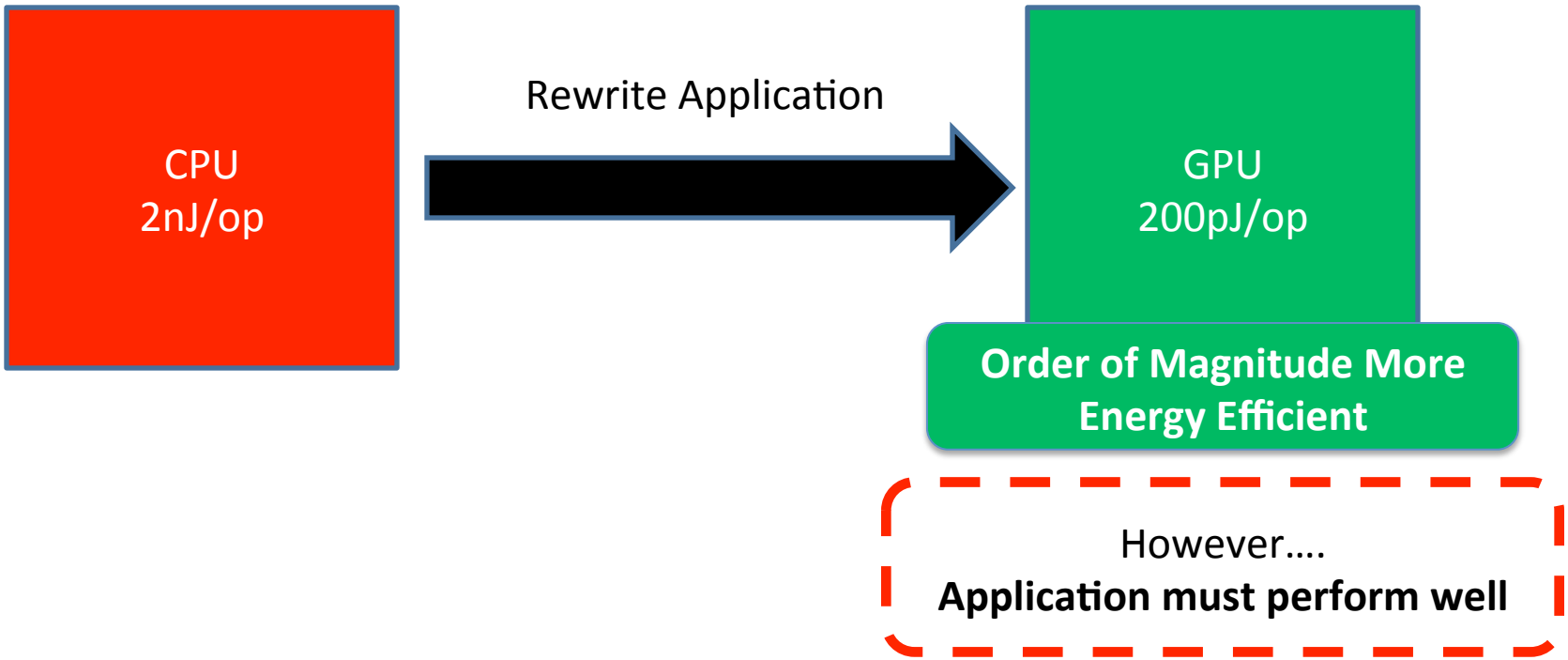
GPU: The Life of a Triangle



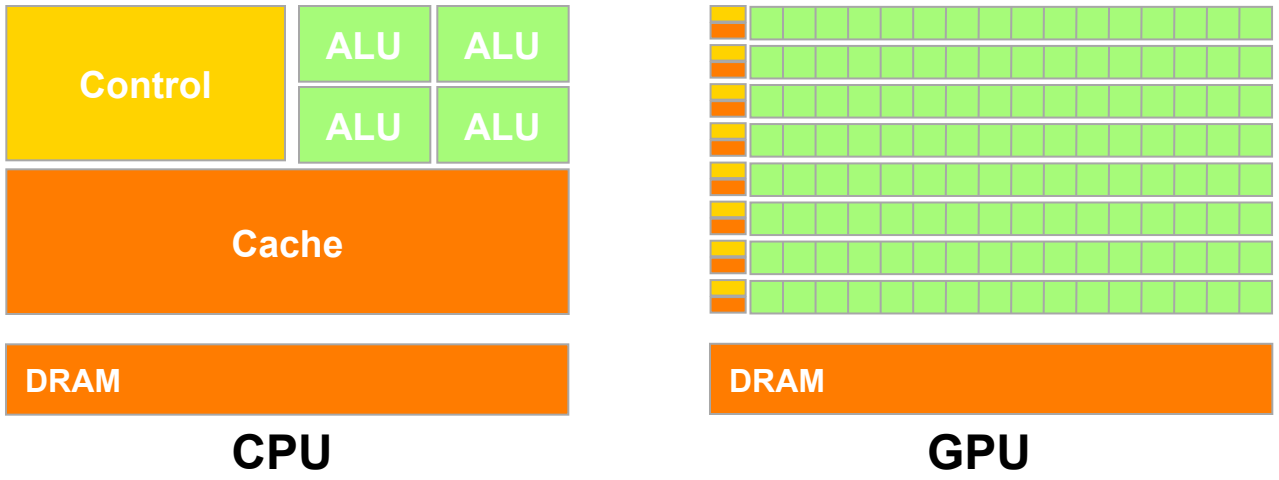
[David Kirk / Wen-mei Hwu]

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.



GPU uses larger fraction of silicon for computation than CPU?



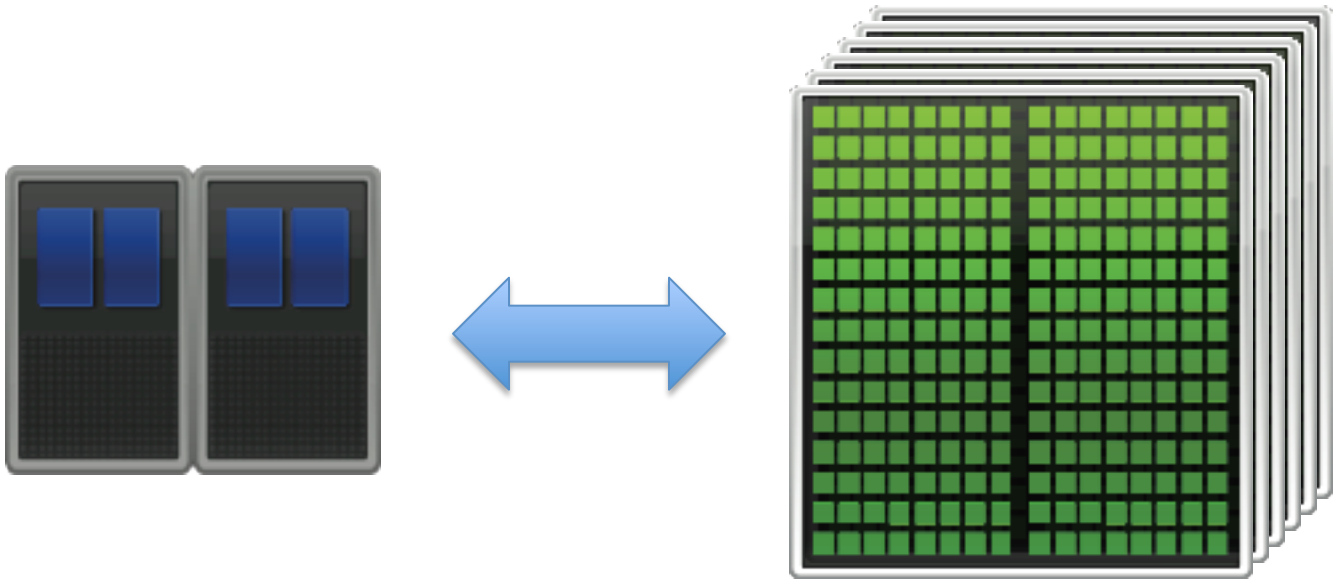
[NVIDIA]

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 Compute Programming Model

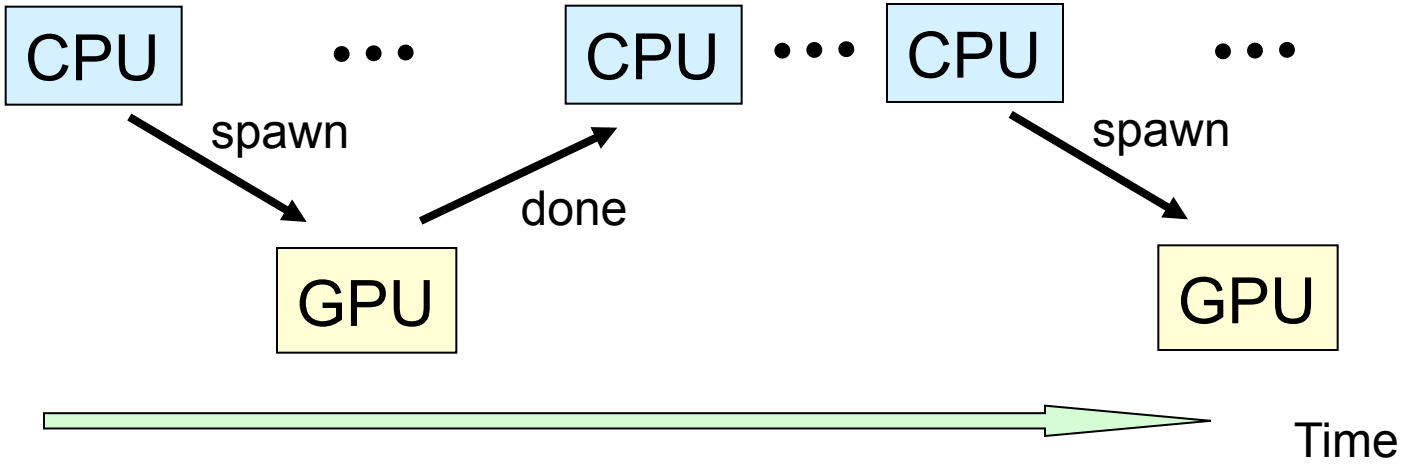


CPU GPU

How is this system programmed (today)?

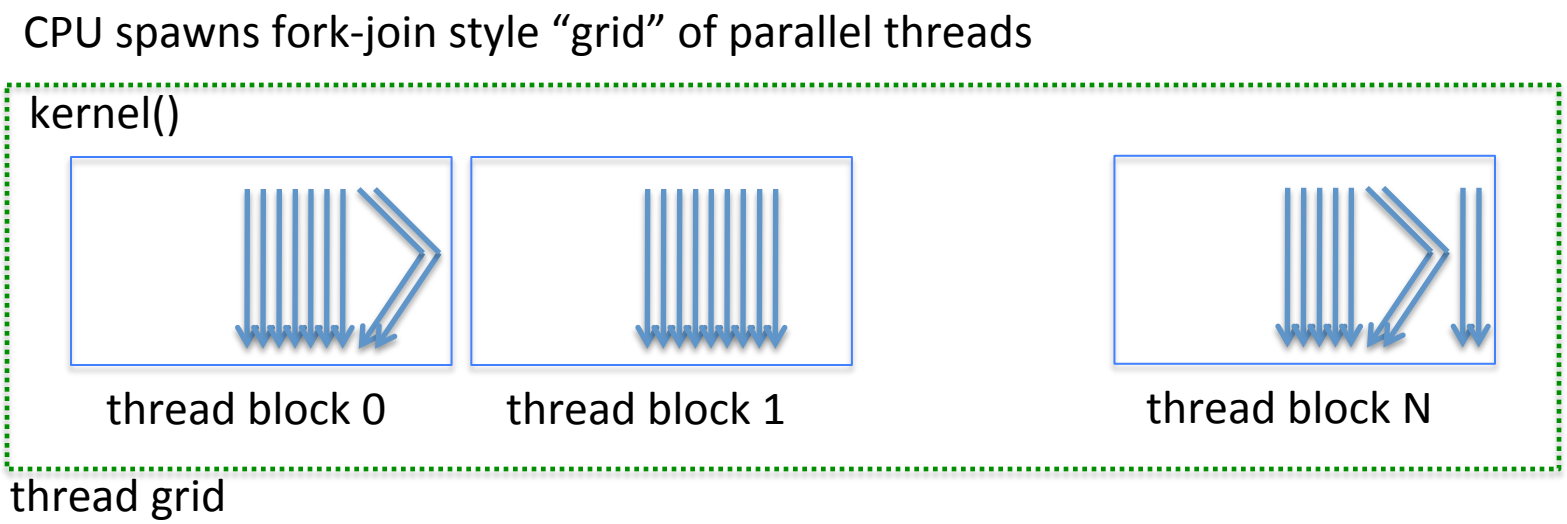
GPGPU Programming Model

- CPU "Off-load" parallel kernels to GPU



  - Transfer data to GPU memory
  - GPU HW spawns threads
  - Need to transfer result data back to CPU main memory

CUDA/OpenCL Threading Model

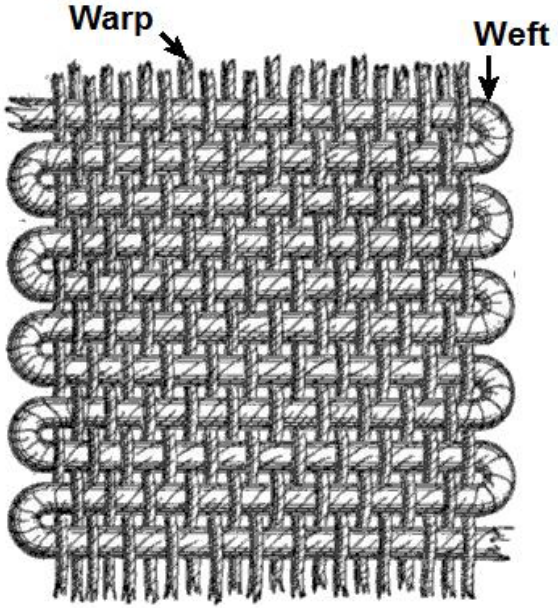


- Spawns more threads than GPU can run (some may wait)
- Organize threads into “blocks” (up to 1024 threads per block)
- Threads can communicate/synchronize with other threads in block
- Threads/Blocks have an identifier (can be 1, 2 or 3 dimensional)
- Each kernel spawns a “grid” containing 1 or more thread blocks.
- Motivation: Write parallel software once and run on future hardware

SIMT Execution Model

- Programmers sees MIMD threads (scalar)
- GPU bundles threads into warps (wavefronts) and runs them in lockstep on SIMD hardware
- An NVIDIA warp groups 32 consecutive threads together (AMD wavefronts group 64 threads together)

- Aside: Why "Warp"? In the textile industry, the term "warp" refers to "the threads stretched lengthwise in a loom to be crossed by the weft" [Oxford Dictionary].
- Jacquard Loom => Babbage's Analytical Engine => … => GPU.

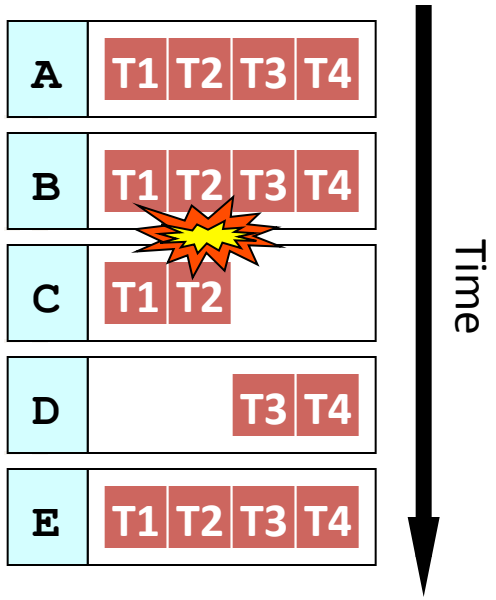


[https://en.wikipedia.org/wiki/Warp_and_woof]

SIMT Execution Model

- Challenge: How to handle branch operations when different threads in a warp follow a different path through program?
- Solution: Serialize different paths.

```
foo[] = {4,8,12,16};

A: v = foo[threadIdx.x];

B: if (v < 10)

C:     v = 0;

   else

D:     v = 10;

E: w = bar[threadIdx.x]+v;
```

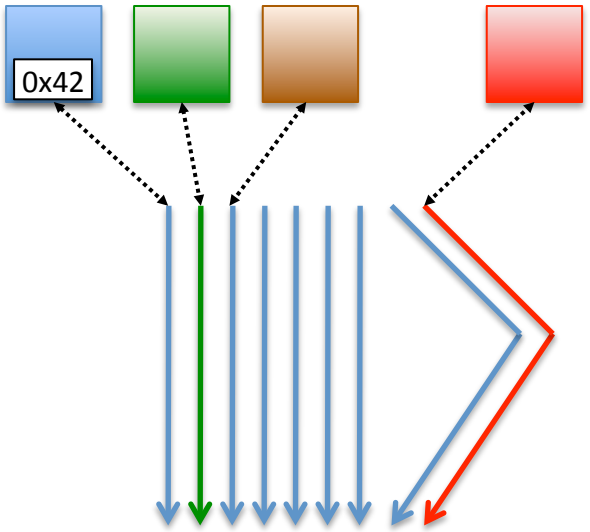


GPU Memory Address Spaces

- GPU has three *address spaces* to support increasing visibility of data between threads: local, shared, global
- In addition two more (read-only) address spaces: Constant and texture.

Local (Private) Address Space

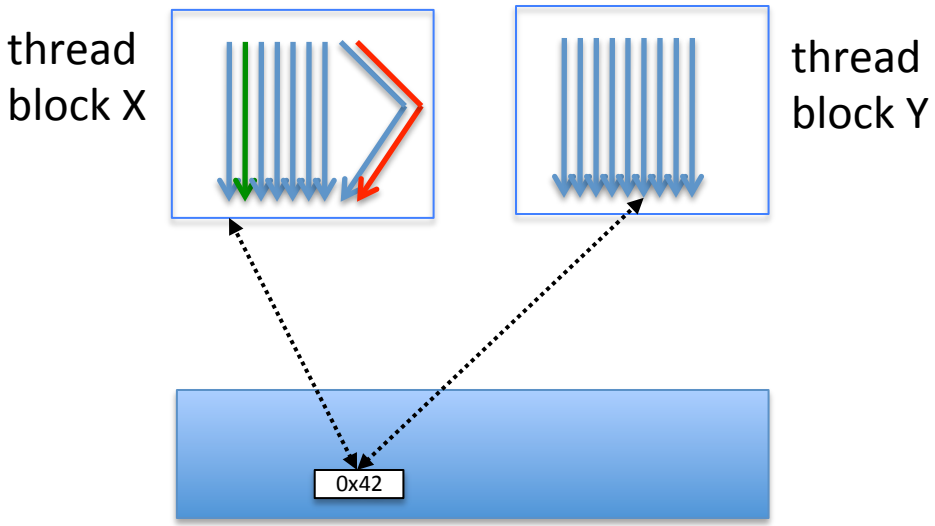
Each thread has own “local memory” (CUDA) “private memory” (OpenCL).



Note: Location at address 100 for thread 0 is different from location at address 100 for thread 1.

Contains local variables private to a thread.

Global Address Spaces

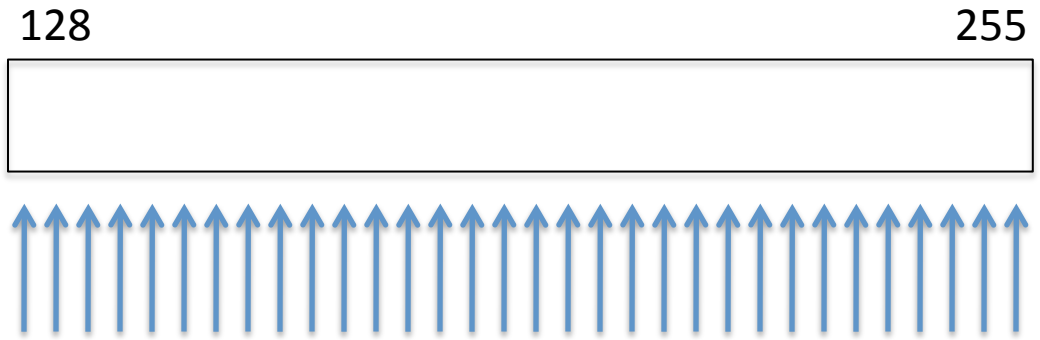


Each thread in the different thread blocks (even from different kernels) can access a region called “global memory” (CUDA/OpenCL).

Commonly in GPGPU workloads threads write their own portion of global memory. Avoids need for synchronization—slow; also unpredictable thread block scheduling.

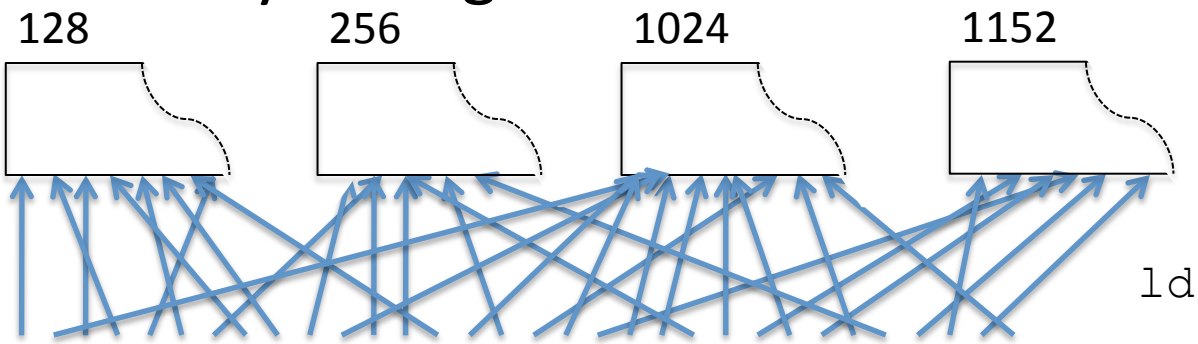
“Coalescing” global accesses

- Not same as CPU write combining/buffering:
- Aligned accesses request single 128B cache blk




```
ld.global r1,0(r2)
```

- Memory Divergence:

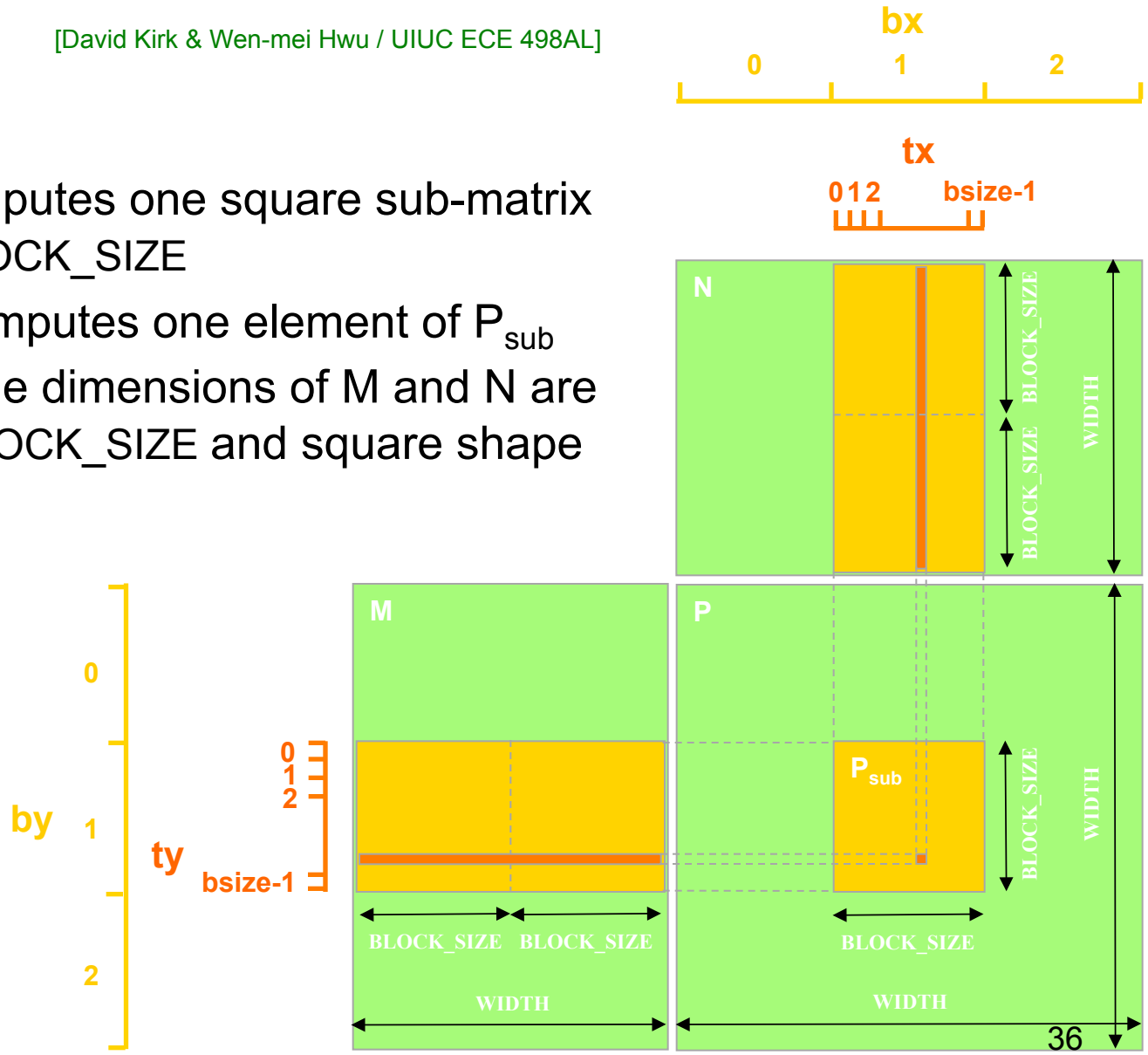



```
ld.global r1,0(r2)
```

Tiled Multiply Using Thread Blocks

[David Kirk & Wen-mei Hwu / UIUC ECE 498AL]

- One block computes one square sub-matrix P_{sub} of size BLOCK_SIZE
- One thread computes one element of P_{sub}
- Assume that the dimensions of M and N are multiples of BLOCK_SIZE and square shape

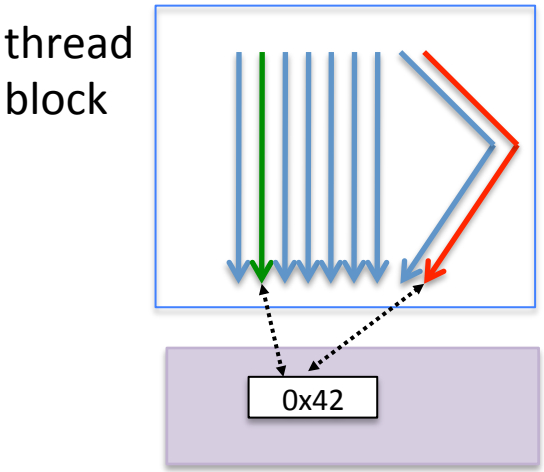


History of “shared memory”

- Prior to NVIDIA GeForce 8800 and CUDA 1.0, threads could not communicate with each other through on-chip memory.

- “Solution”: small (16-48KB) programmer managed scratchpad memory shared between threads within a thread block.

Shared (Local) Address Space



Each thread in the same thread block (work group) can access a memory region called “shared memory” (CUDA) “local memory” (OpenCL).

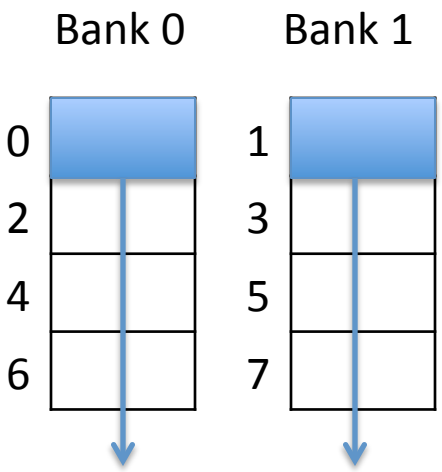
Shared memory address space is limited in size (16 to 48 KB).

Used as a software managed “cache” to avoid off-chip memory accesses.

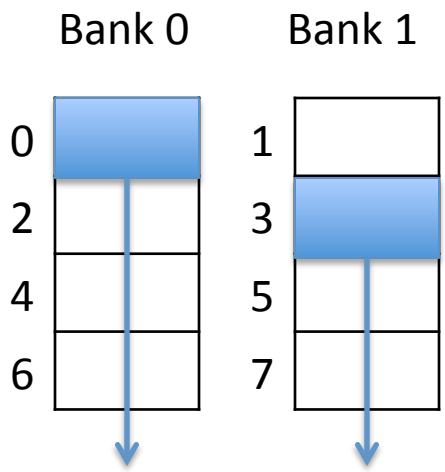
Synchronize threads in a thread block using __syncthreads();

Review: Bank Conflicts

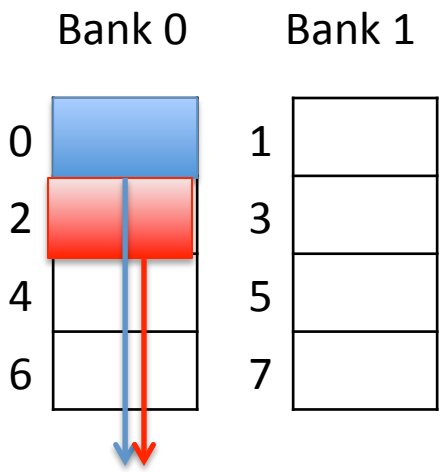
- To increase bandwidth common to organize memory into multiple banks.
- Independent accesses to different banks can proceed in parallel



Example 1: Read 0, Read 1
(can proceed in parallel)



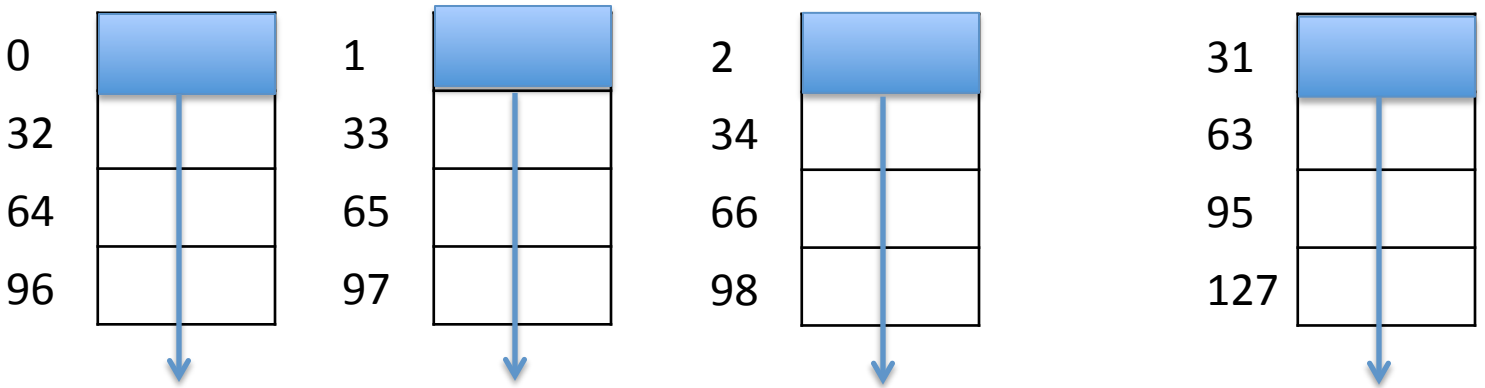
Example 2: Read 0, Read 3
(can proceed in parallel)



Example 3: Read 0, Read 2
(bank conflict)

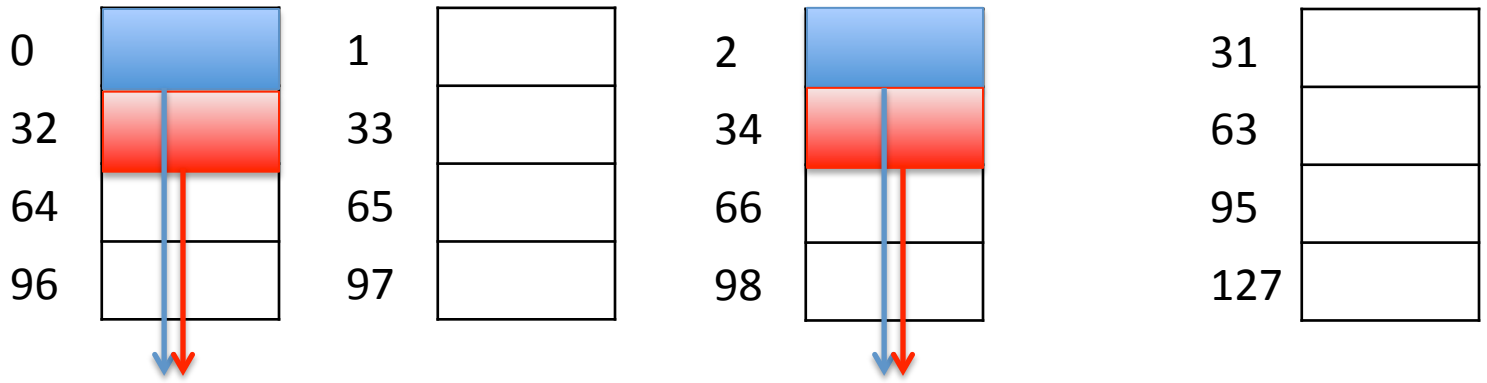
Shared Memory Bank Conflicts

```
__shared__ int A[BSIZE];
…
A[threadIdx.x] = … // no conflicts
```



Shared Memory Bank Conflicts

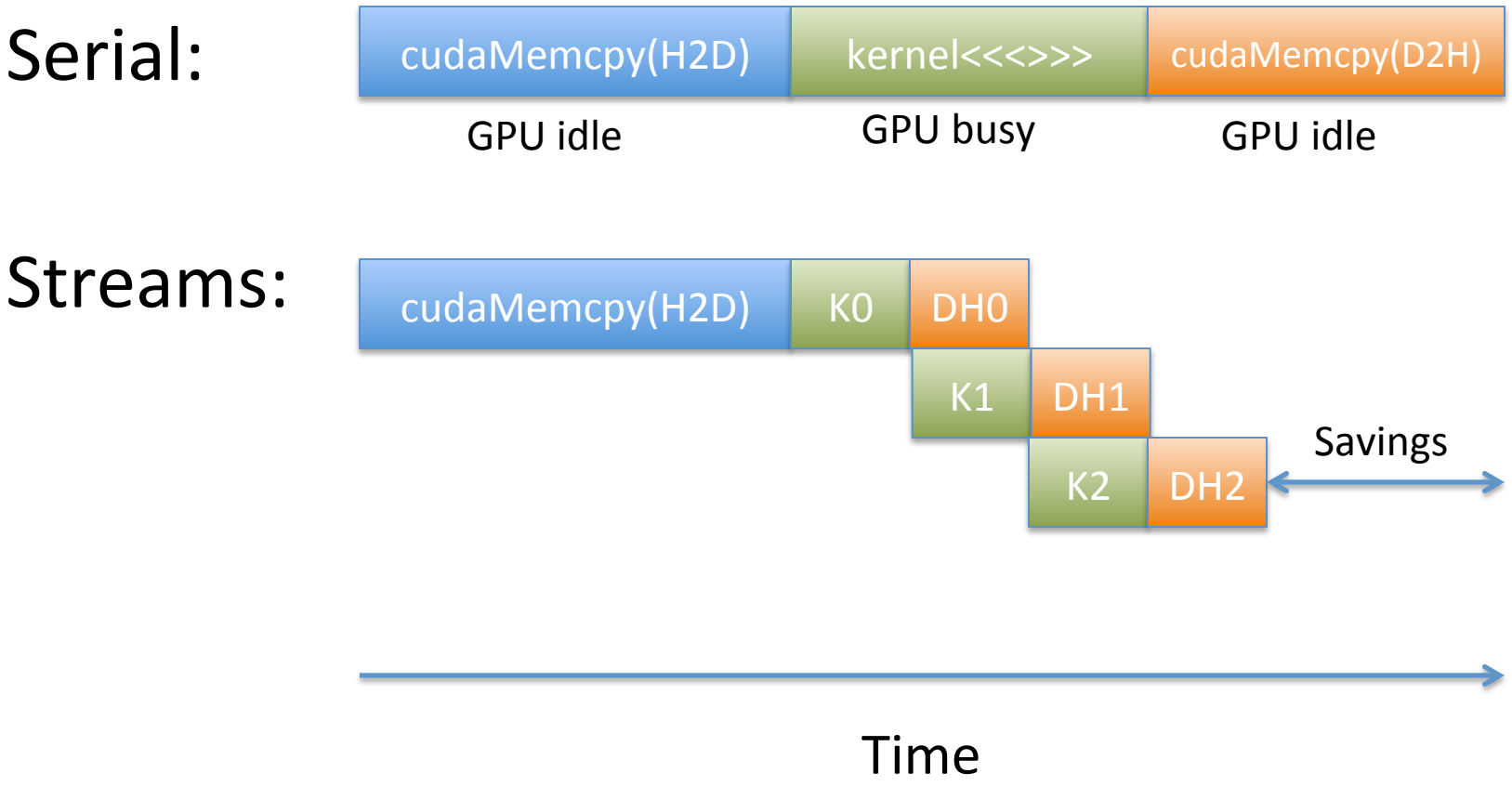
```
__shared__ int A[BSIZE];
…
A[2*threadIdx.x] = // 2-way conflict
```



CUDA Streams

- CUDA (and OpenCL) provide the capability to overlap computation on GPU with memory transfers using “Streams” (Command Queues)
- A Stream orders a sequence of kernels and memory copy “operations”.
- Operations in one stream can overlap with operations in a different stream.

How Can Streams Help?



http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf

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
```

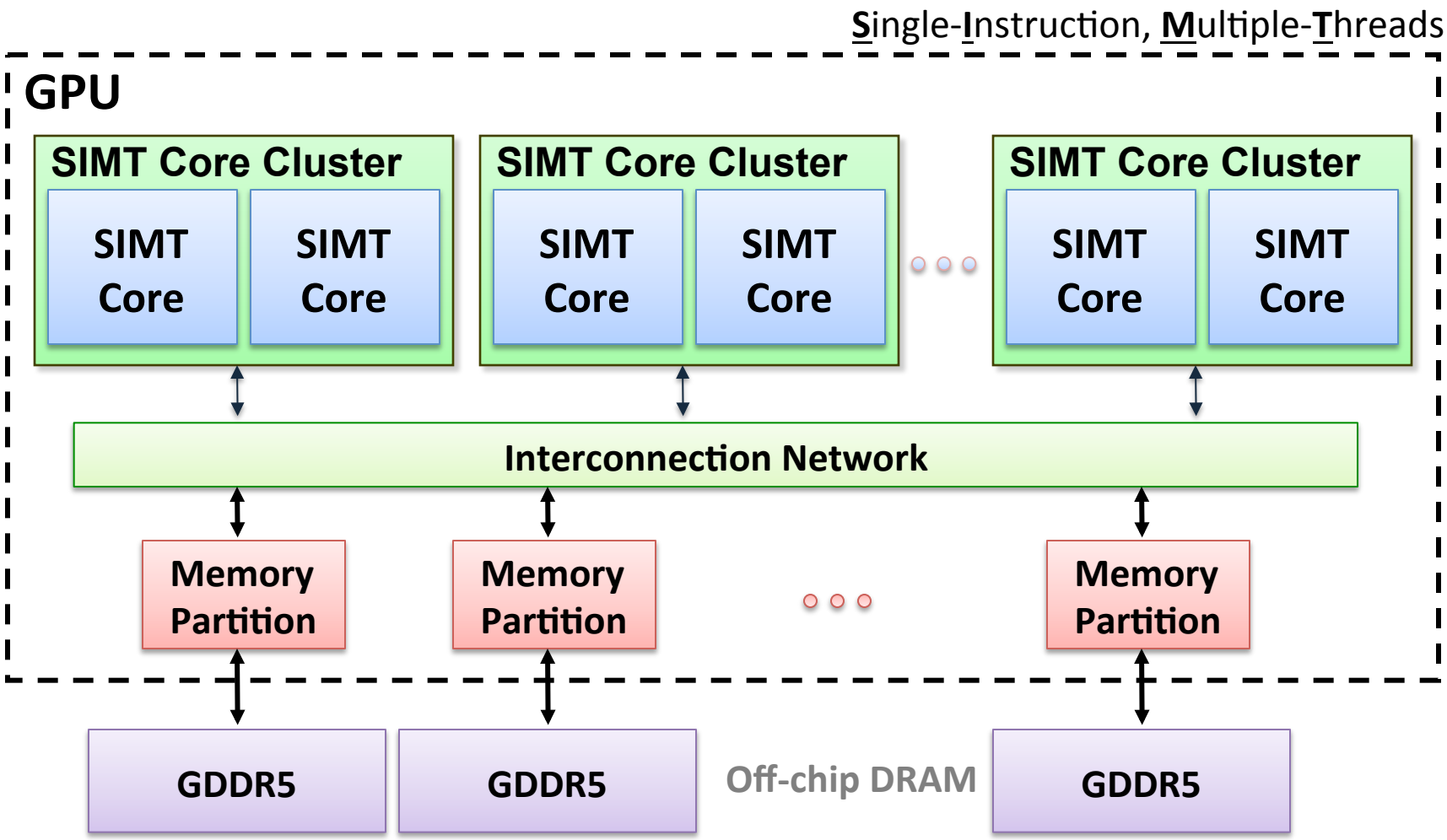
- Memory operations:

```
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
```

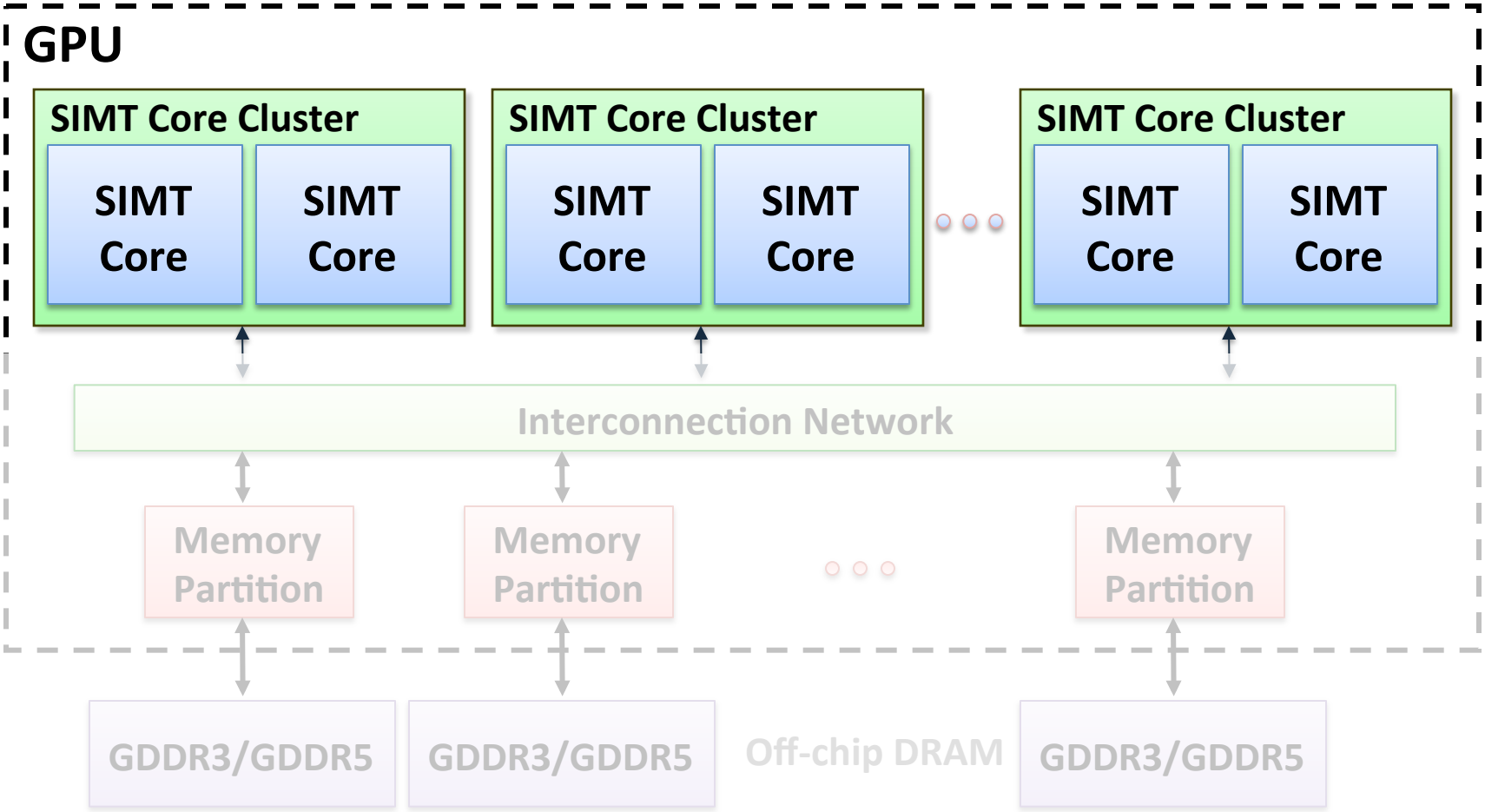
GPU Microarchitecture Overview



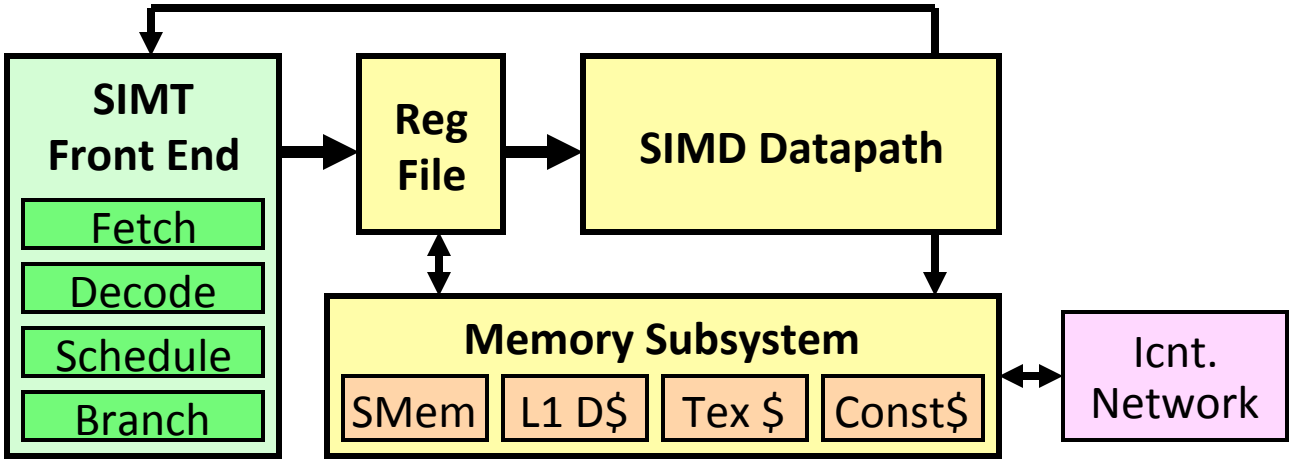
GPU Microarchitecture

- 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

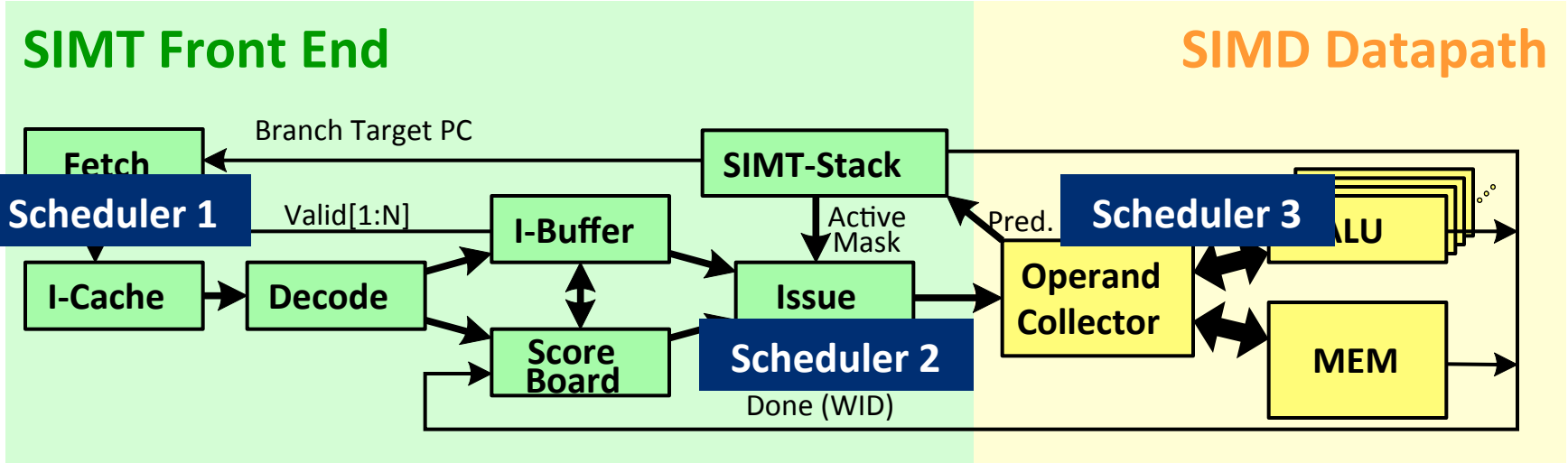


Inside a SIMT Core



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

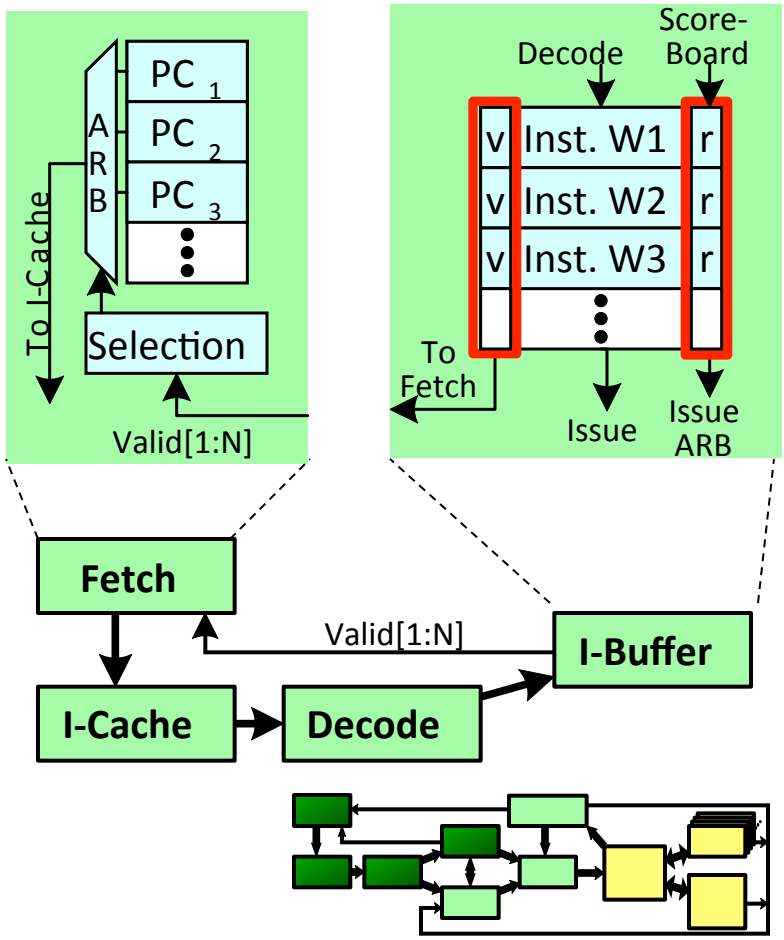
Inside an "NVIDIA-style" SIMT Core



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

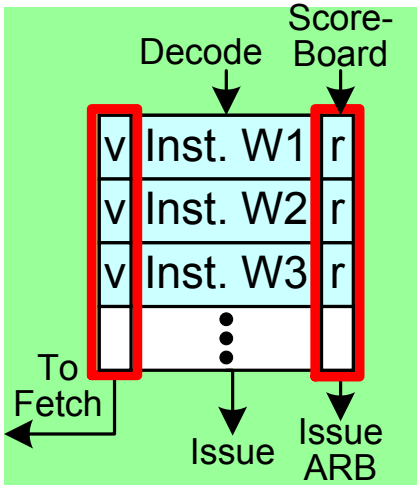
Fetch + Decode

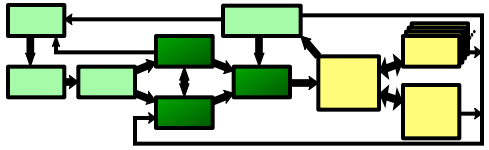
- Arbitrate the I-cache among warps
  - Cache miss handled by fetching again later
- Fetched instruction is decoded and then stored in the I-Buffer
  - 1 or more entries / warp
  - Only warp with vacant entries are considered in fetch



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)

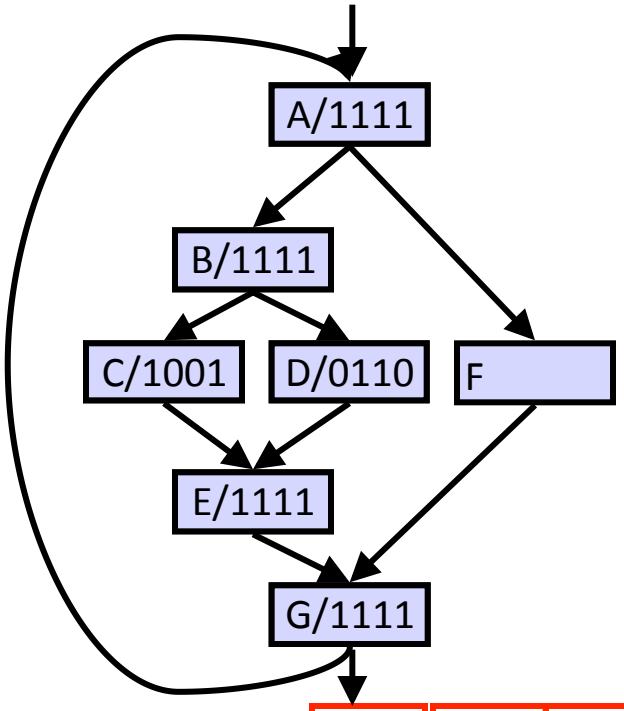




SIMT Using a Hardware Stack

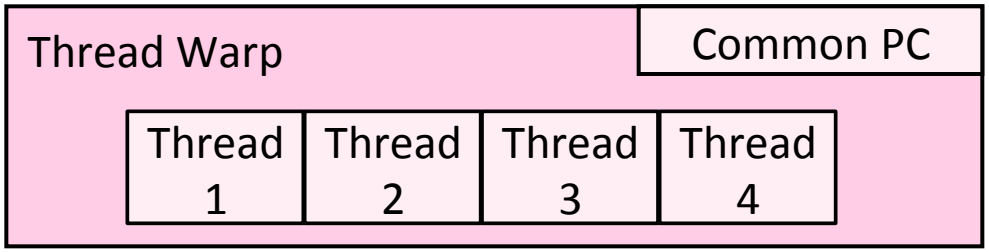
Stack approach invented at Lucafilm, Ltd in early 1980's

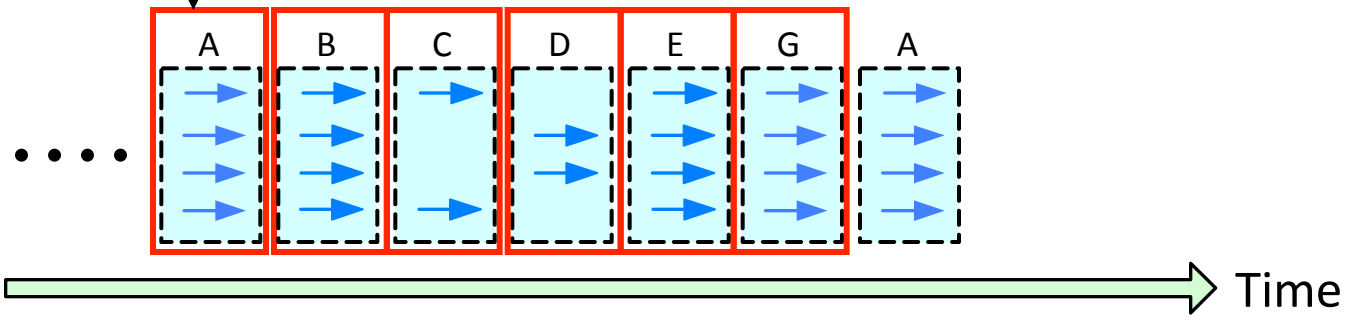
Version here from [Fung et al., MICRO 2007]



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

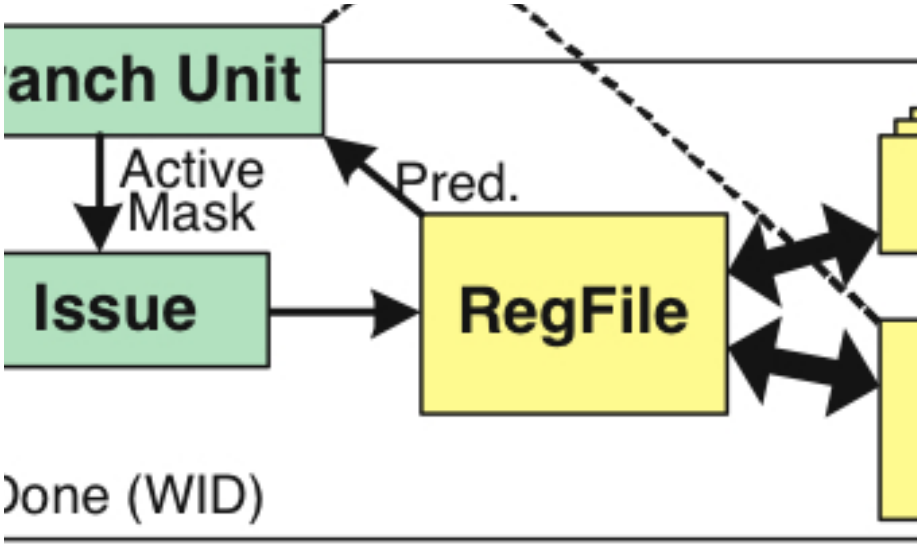
- Execution mask stack implemented with special instructions to push/pop. Descriptions can be found in AMD ISA manual and NVIDIA patents.
- In practice augment stack with predication (lower overhead).

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

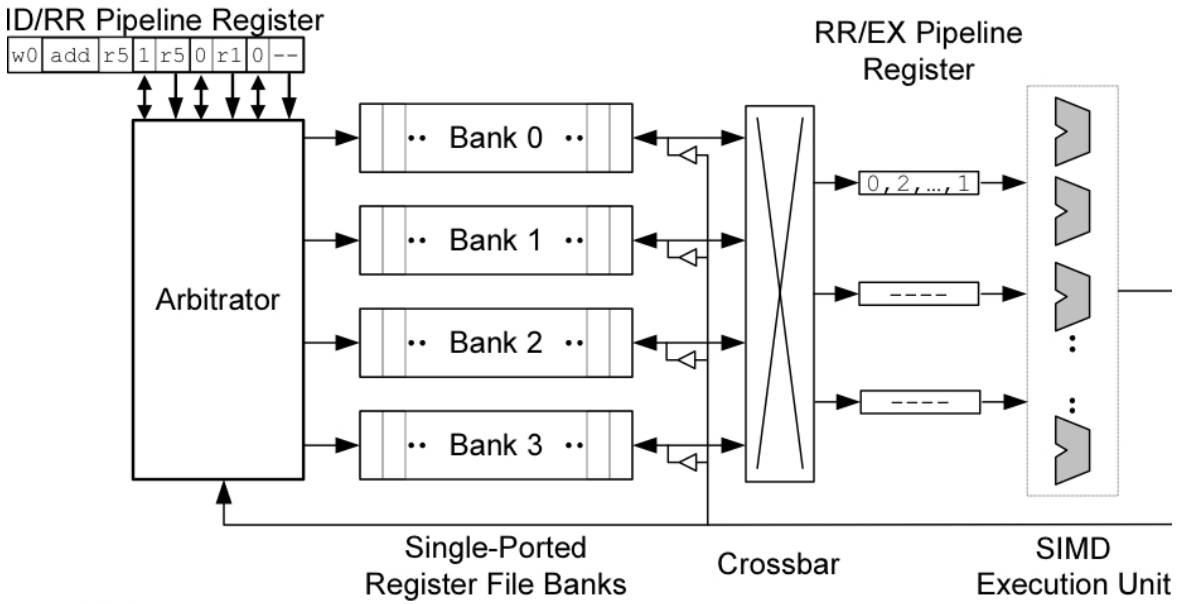
Register File

- 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

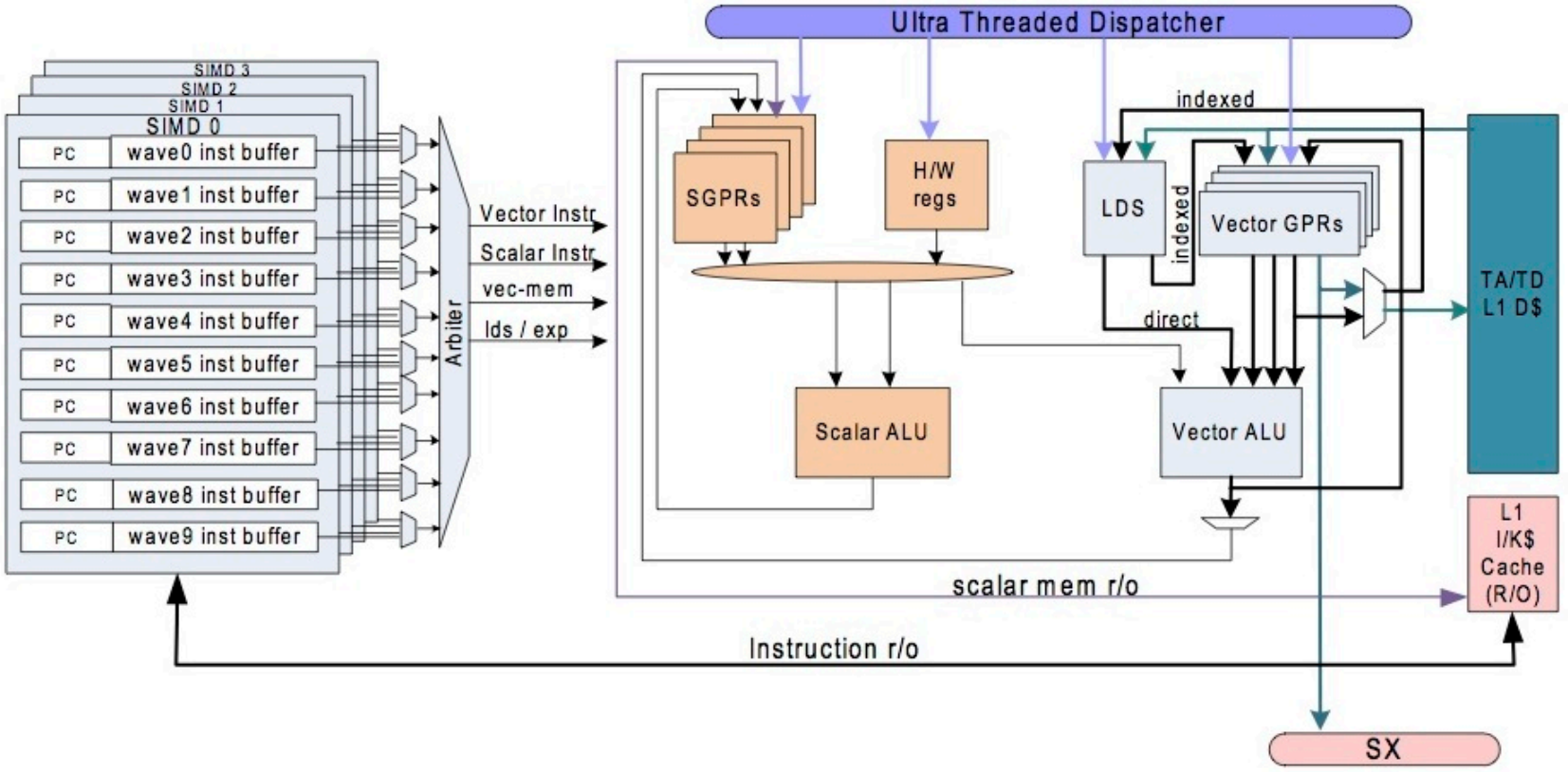
AMD Southern Islands

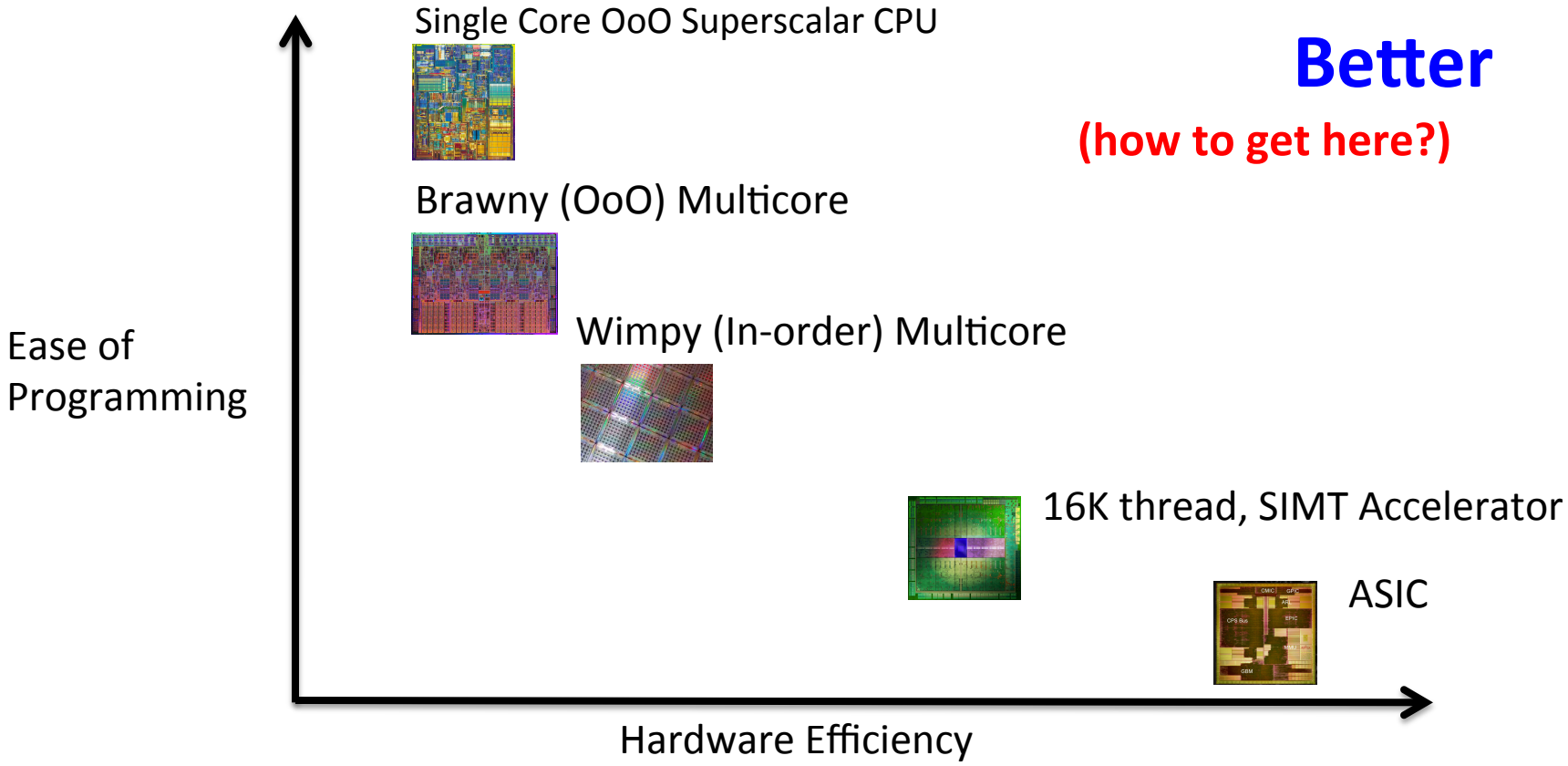
- SIMT processing often includes redundant computation across threads.

```
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




Single Core OoO Superscalar CPU
Brawny (OoO) Multicore
Wimpy (In-order) Multicore
16K thread, SIMT Accelerator
ASIC
Ease of Programming
Hardware Efficiency
Better
(how to get here?)

Start by using right tool for each job…