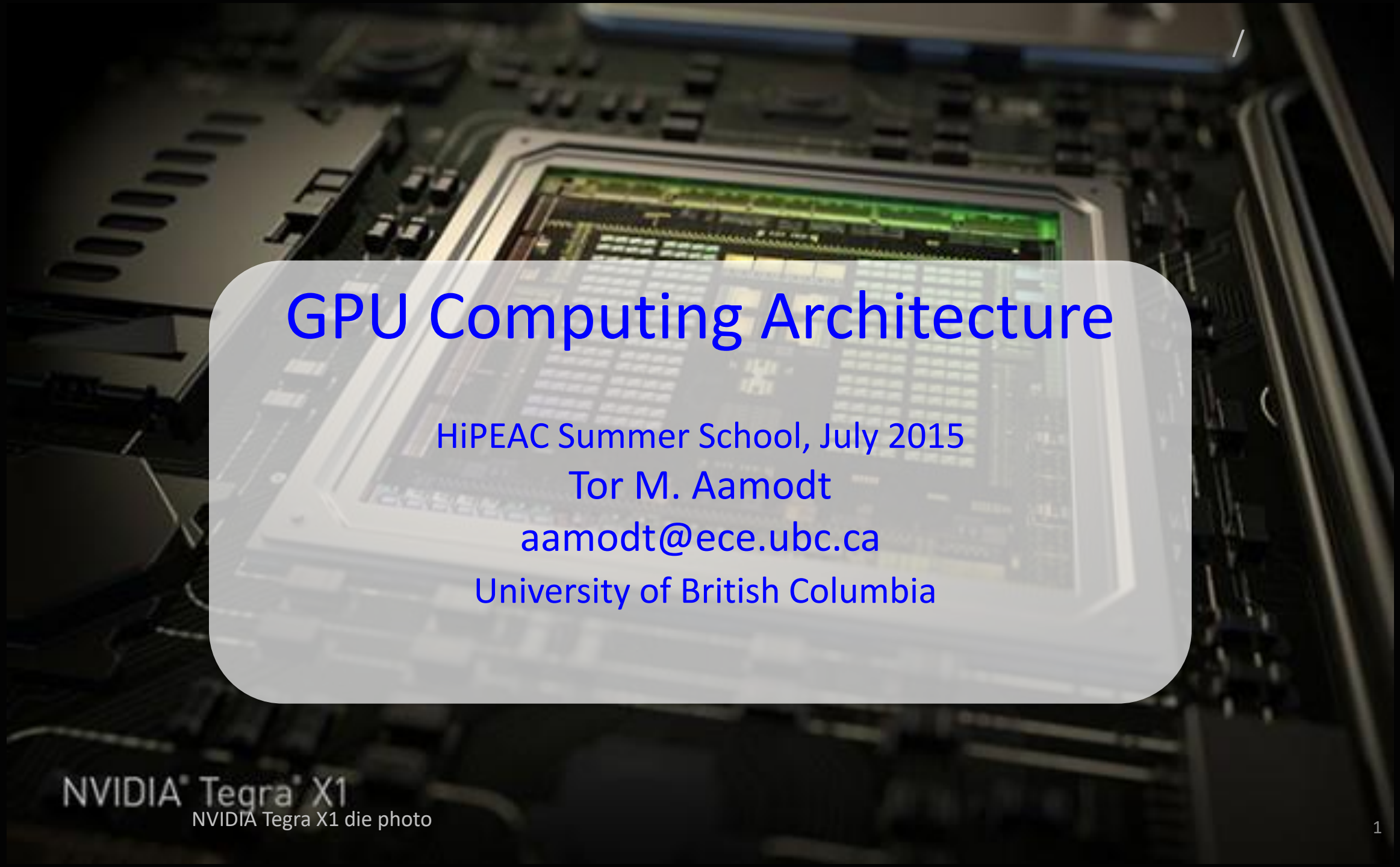
GPU Computing Architecture

HiPEAC Summer School, July 2015
Tor M. Aamodt
aamodt@ece.ubc.ca
University of British Columbia



NVIDIA Tegra X1 die photo

General-Purpose Graphics Processor Architectures

Synthesis Lectures on Computer Architecture

May 2018, 140 pages, (https://doi.org/10.2200/S00848ED1V01Y201804CAC044)

Tor M. Aamodt
University of British Columbia

Wilson Wai Lun Fung
Samsung Electronics

Timothy G. Rogers
Purdue University

Abstract

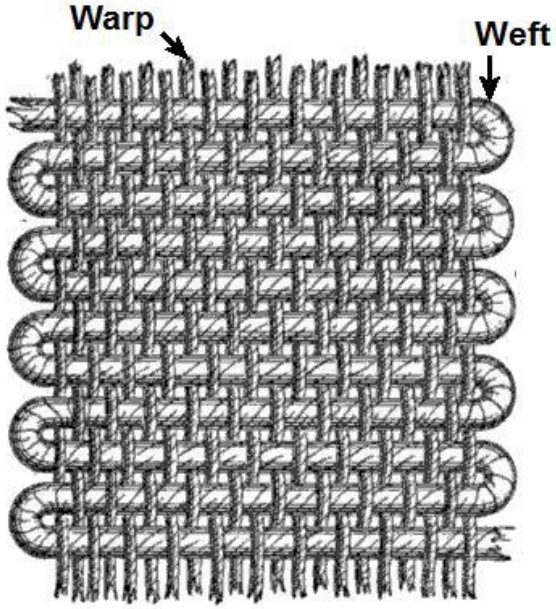
Originally developed to support video games, graphics processor units (GPUs) are now increasingly used for general-purpose (non-graphics) applications ranging from machine learning to mining of cryptographic currencies. GPUs can achieve improved performance and efficiency versus central processing units (CPUs) by dedicating a larger fraction of hardware resources to computation. In addition, their general-purpose programmability makes contemporary GPUs appealing to software developers in comparison to domain-specific accelerators. This book provides an introduction to those interested in studying the architecture of GPUs that support general-purpose computing. It collects together information currently only found among a wide range of disparate sources. The authors led development of the GPGPU-Sim simulator widely used in academic research on GPU architectures.

The first chapter of this book describes the basic hardware structure of GPUs and provides a brief overview of their history. Chapter 2 provides a summary of GPU programming models relevant to the rest of the book. Chapter 3 explores the architecture of GPU compute cores. Chapter 4 explores the architecture of the GPU memory system. After describing the architecture of existing systems, Chapters \ref{ch03} and \ref{ch04} provide an overview of related research. Chapter 5 summarizes cross-cutting research impacting both the compute core and memory system.

This book should provide a valuable resource for those wishing to understand the architecture of graphics processor units (GPUs) used for acceleration of general-purpose applications and to those who want to obtain an introduction to the rapidly growing body of research exploring how to improve the architecture of these GPUs.

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].

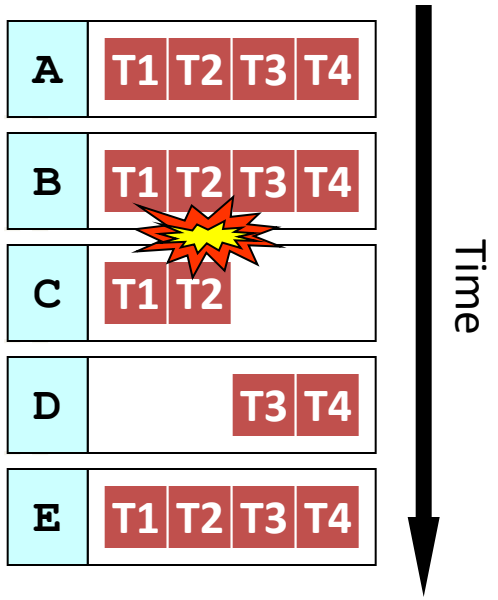


[https://en.wikipedia.org/wiki/Warp_and_woof]

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 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:

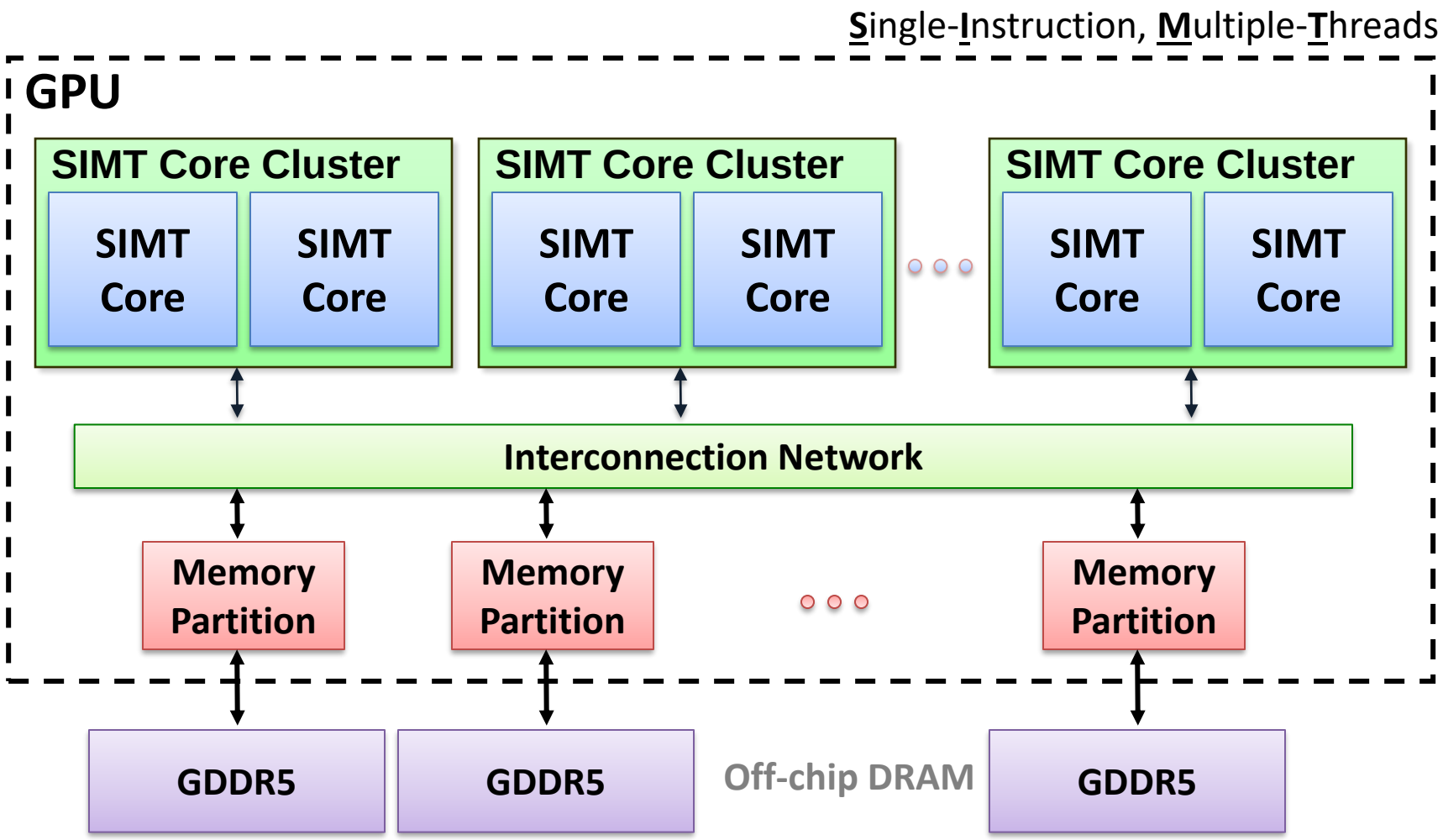
```
    setp.eq.f32 p, y, 0;  // is y equal to zero?
@p bra L1  // branch to L1 if y equal to zero
```

Part 2: Generic GPGPU Architecture

Extra resources

GPGPU-Sim 3.x Manualhttp://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual

GPU Microarchitecture Overview

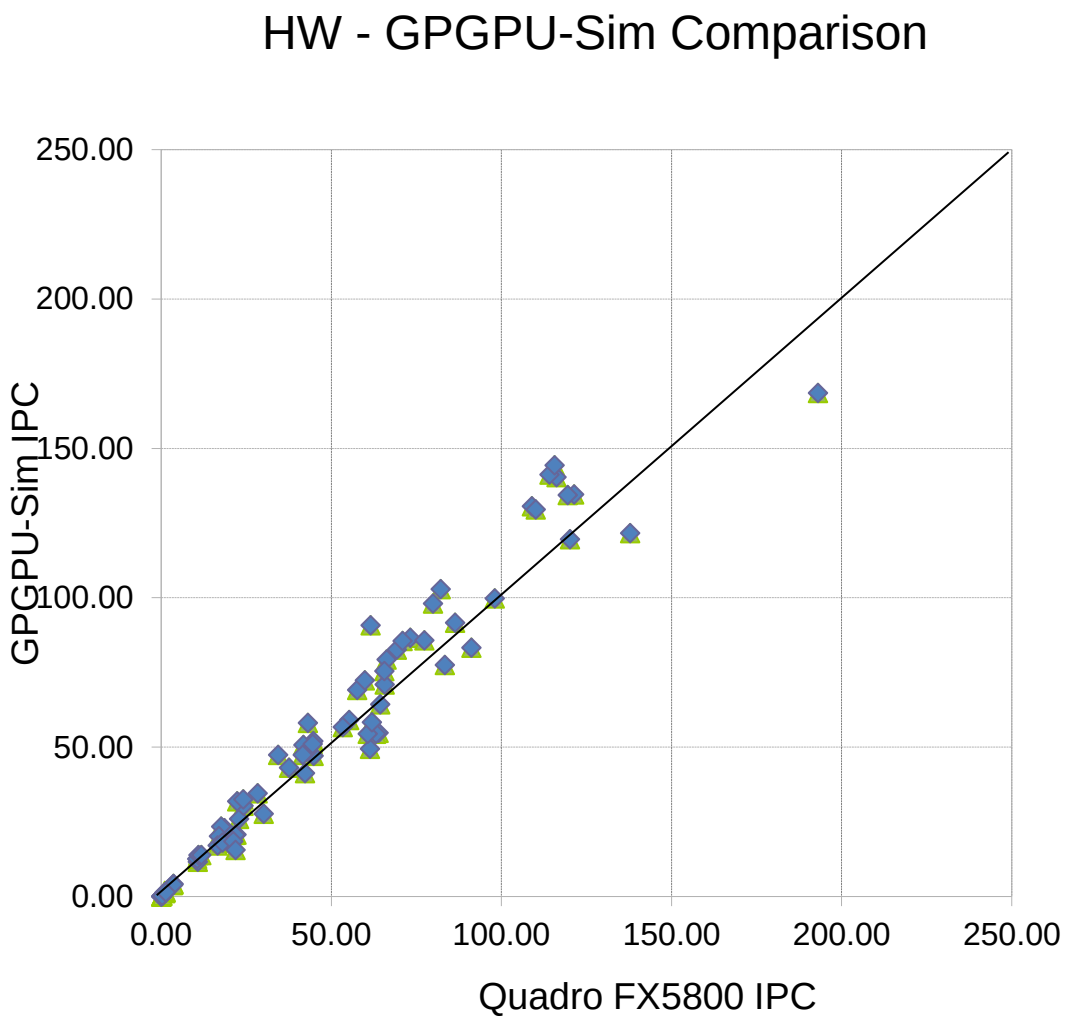


GPU Microarchitecture

- 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

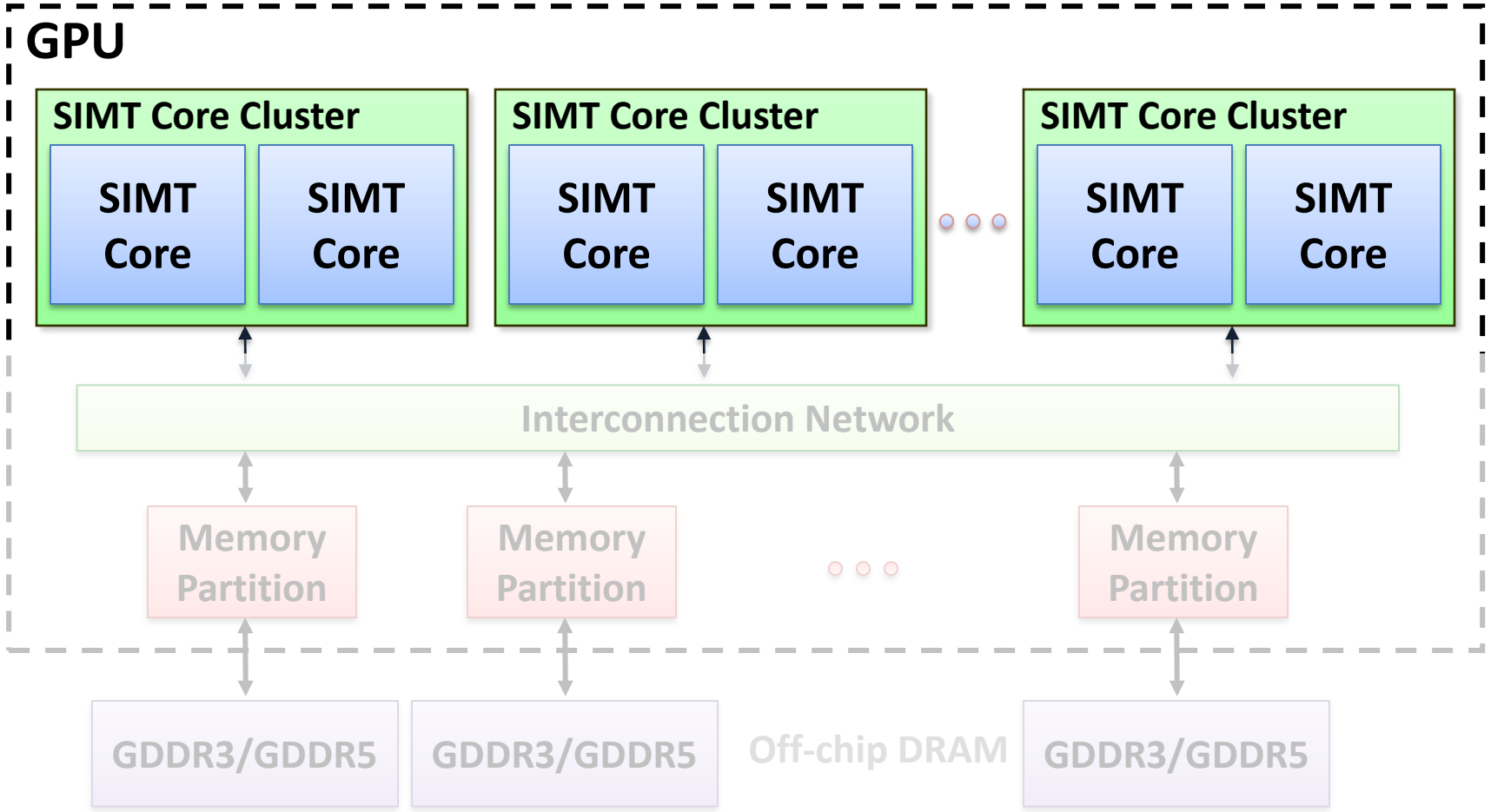
- Model described next, embodied in GPGPU-Sim, developed from: white papers, programming manuals, IEEE Micro articles, patents.

GPGPU-Sim v3.x w/ SASS

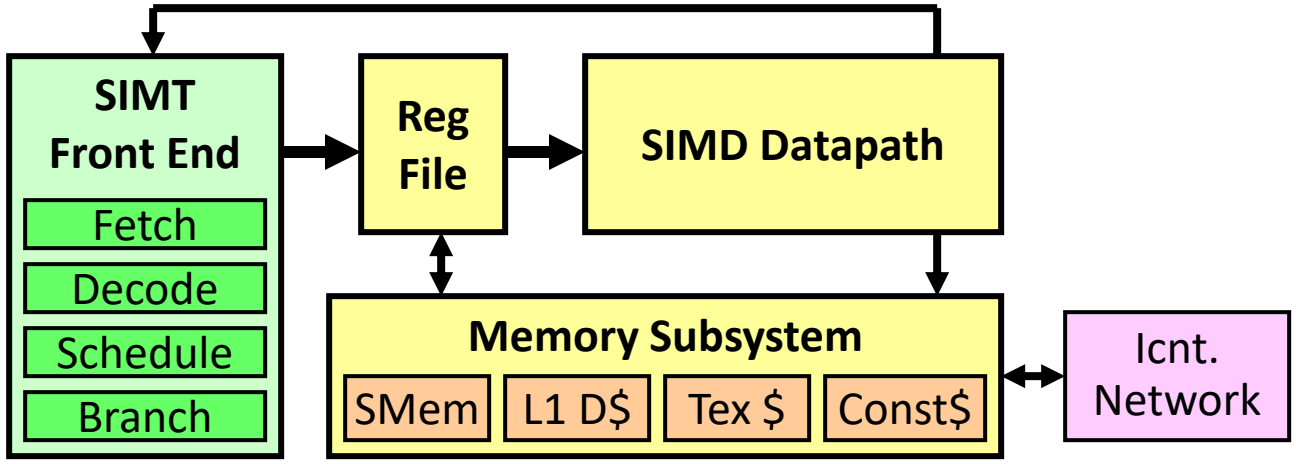


Correlation
~0.976

GPU Microarchitecture Overview

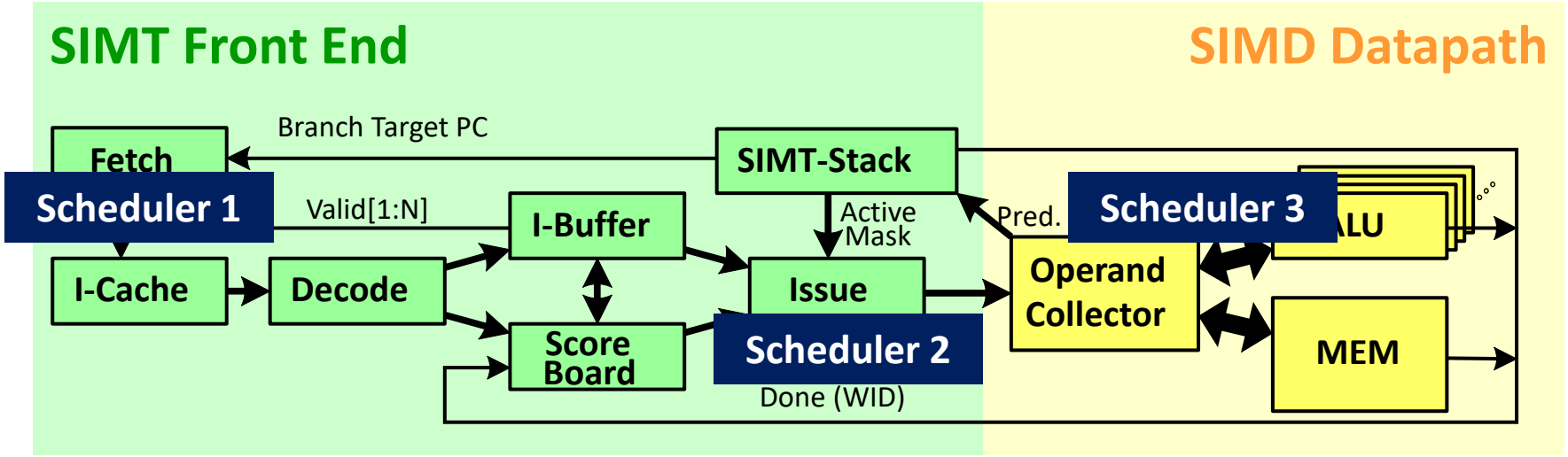


Inside a SIMT Core



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

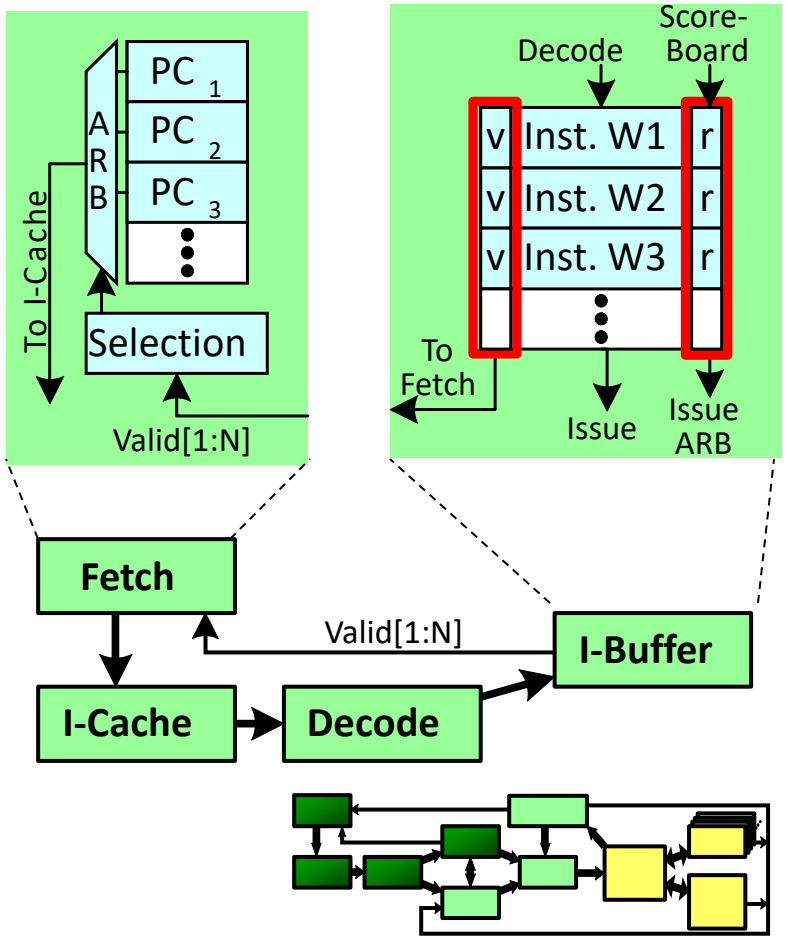
Inside an “NVIDIA-style” SIMT Core



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

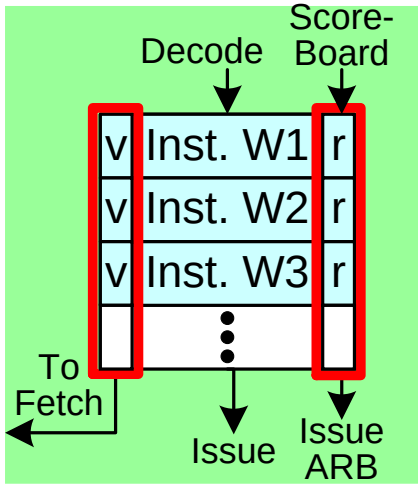
Fetch + Decode

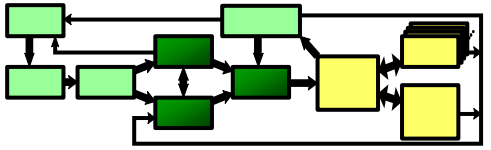
- Arbitrate the I-cache among warps
  - Cache miss handled by fetching again later
- Fetched instruction is decoded and then stored in the I-Buffer
  - 1 or more entries / warp
  - Only warp with vacant entries are considered in fetch



Instruction Issue

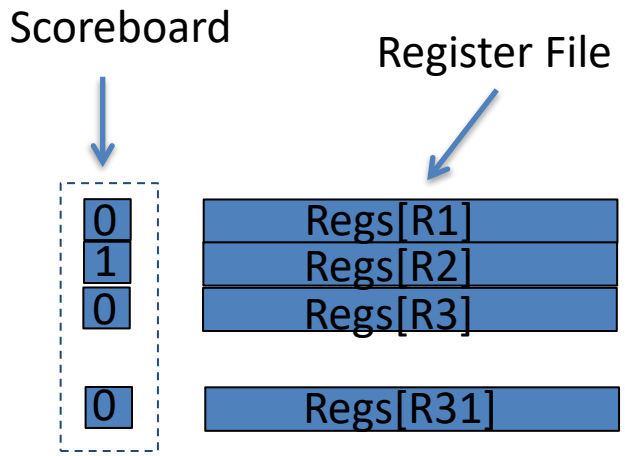
- Select a warp and issue an instruction from its I-Buffer for execution
  - Scheduling: Greedy-Then-Oldest (GTO)
  - GT200/later Fermi/Kepler: Allow dual issue (superscalar)
  - Fermi: Odd/Even scheduler
  - To avoid stalling pipeline might keep instruction in I-buffer until know it can complete (replay)





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]



Example

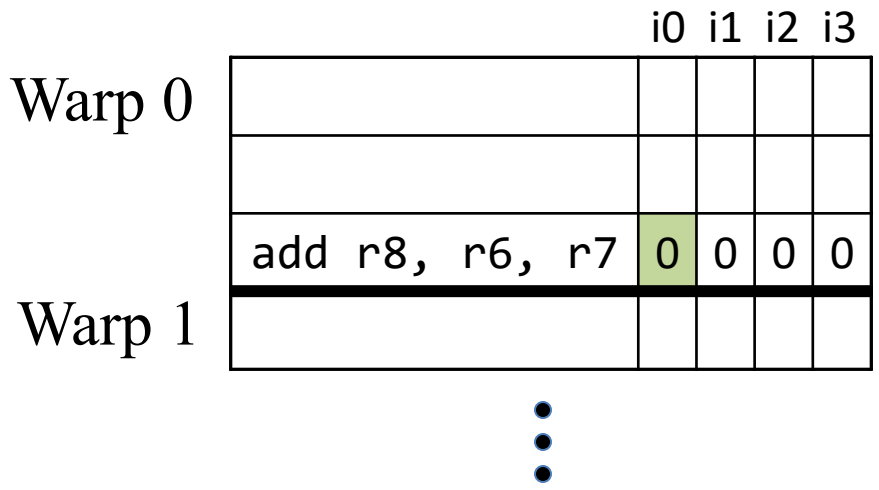
Code

```
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	-	-	-	-

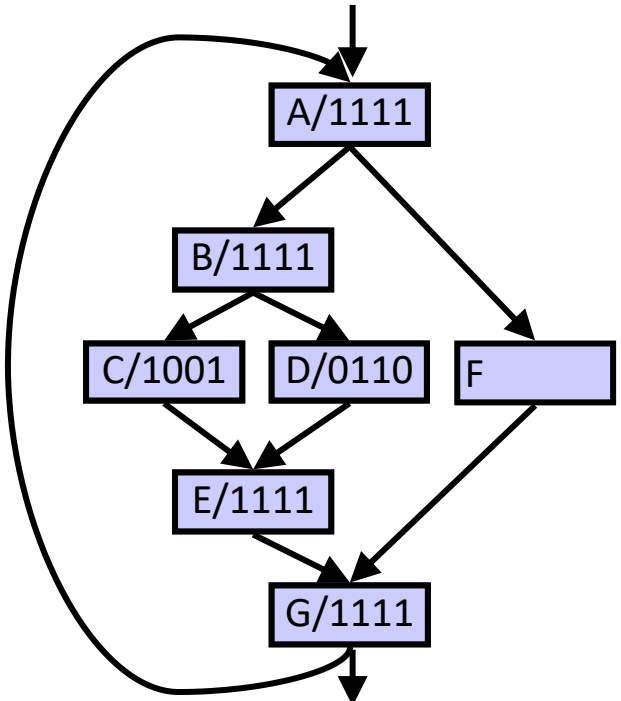
Instruction Buffer



SIMT Using a Hardware Stack

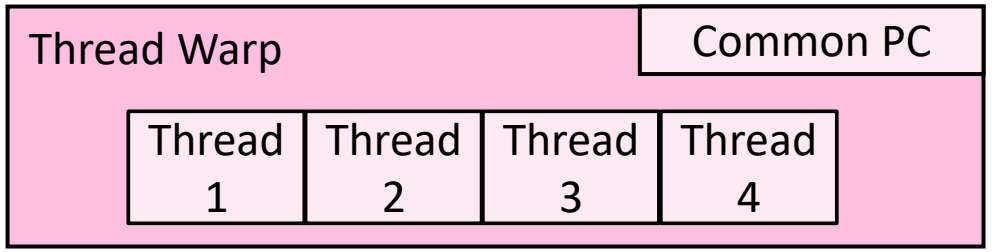
Stack approach invented at Lucasfilm, Ltd in early 1980's

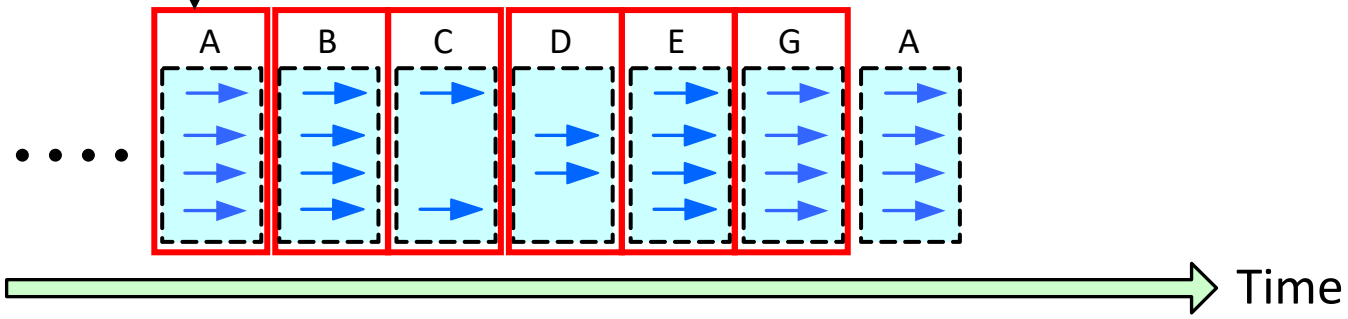
Version here from [Fung et al., MICRO 2007]



Stack

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





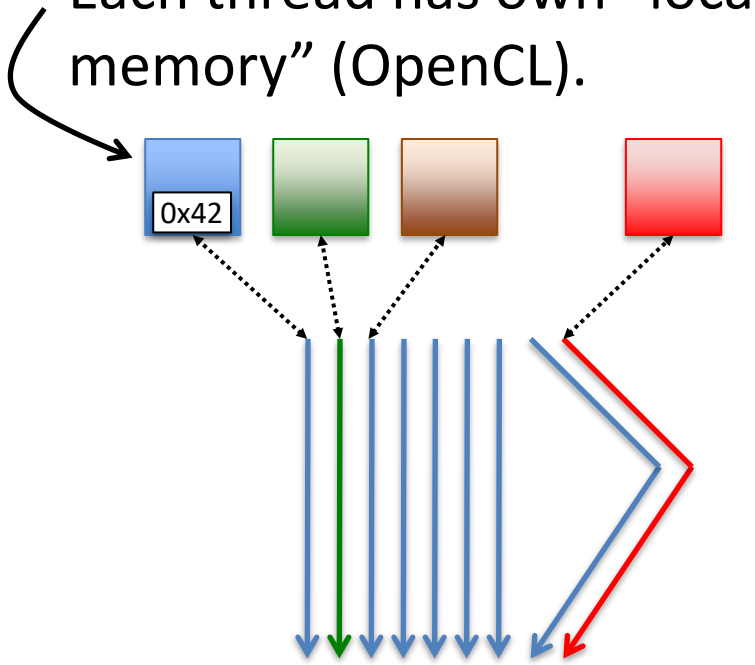
SIMT = SIMD Execution of Scalar Threads

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

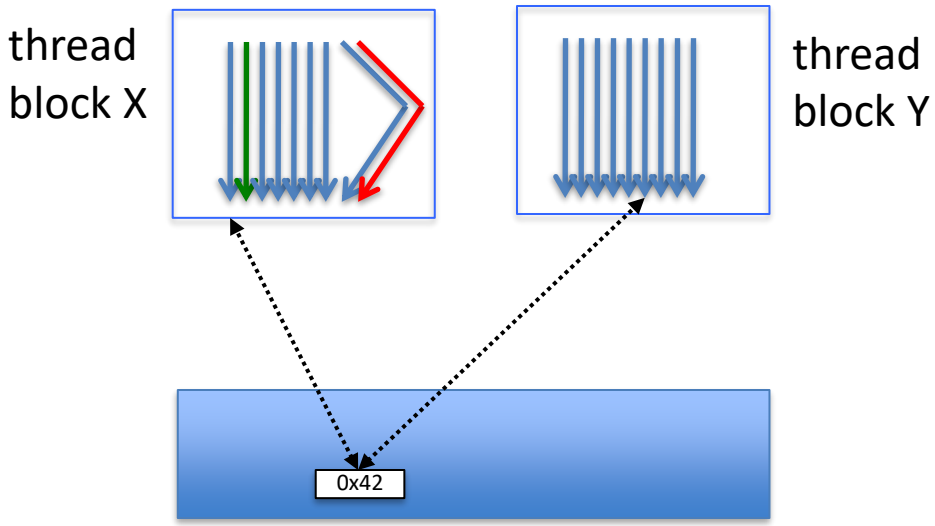
Each thread has own “local memory” (CUDA) “private memory” (OpenCL).



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.

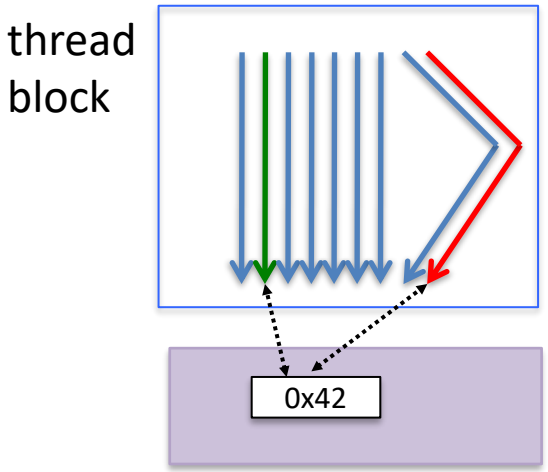
Global Address Spaces



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.

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).

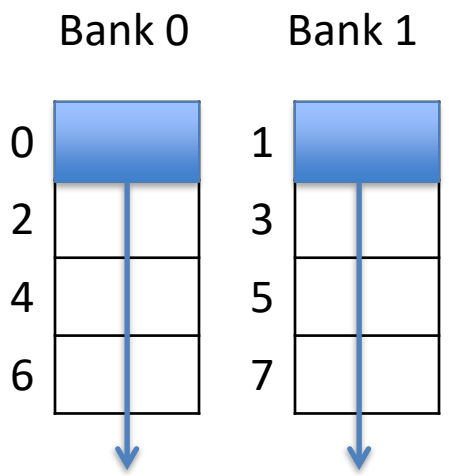
Shared memory address space is limited in size (16 to 48 KB).

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

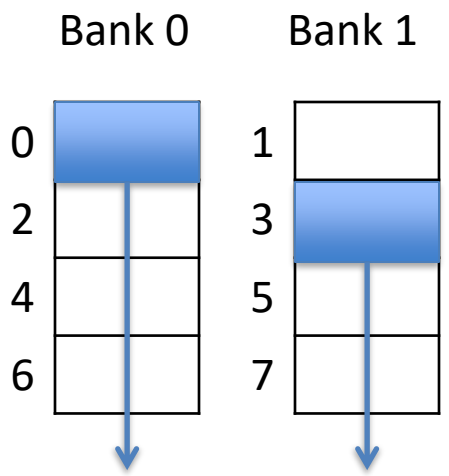
Synchronize threads in a thread block using __syncthreads();

Review: Bank Conflicts

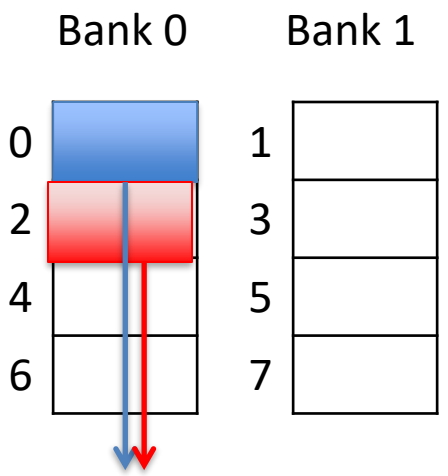
- To increase bandwidth common to organize memory into multiple banks.
- Independent accesses to different banks can proceed in parallel



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



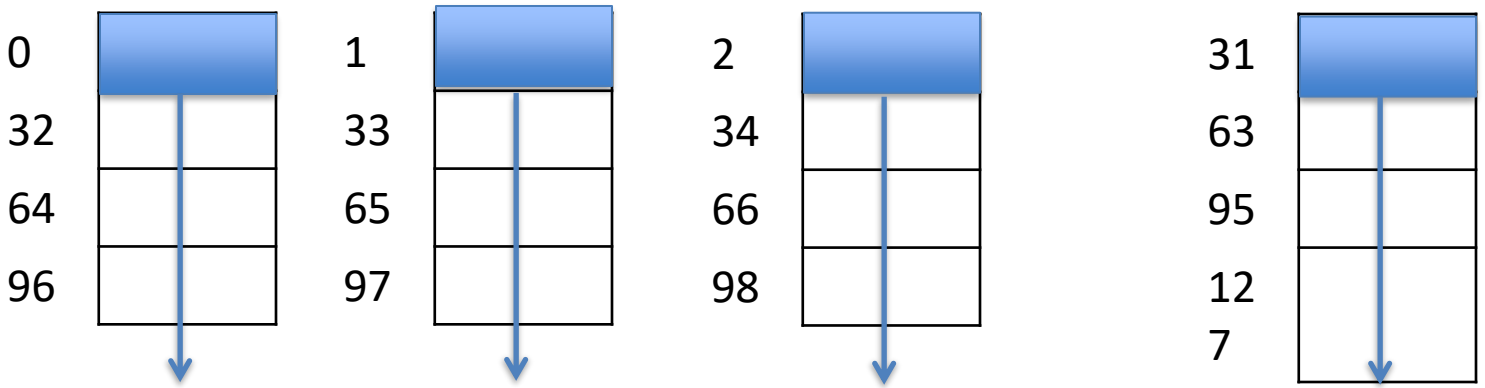
Example 2: Read 0, Read 3
(can proceed in parallel)



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

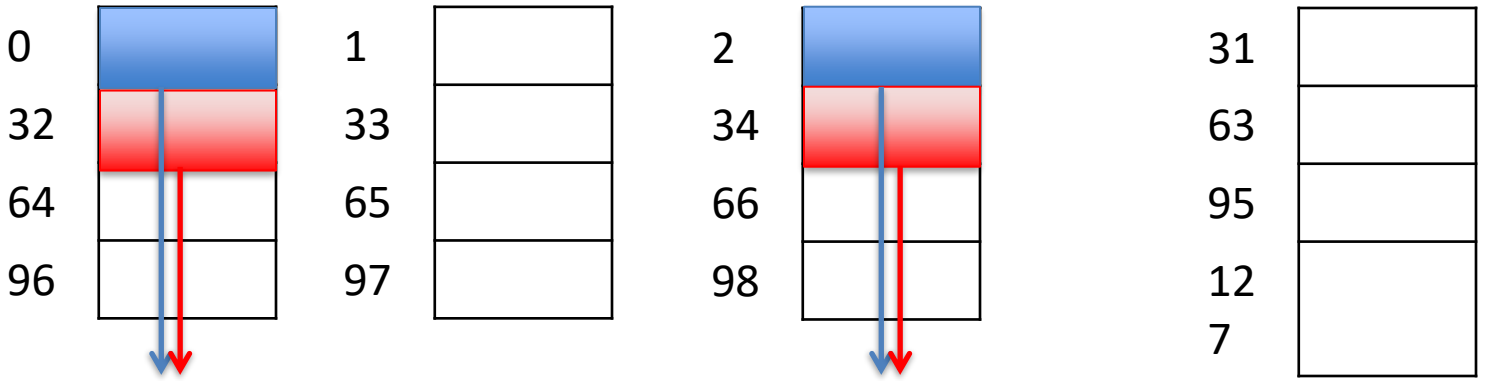
Shared Memory Bank Conflicts

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

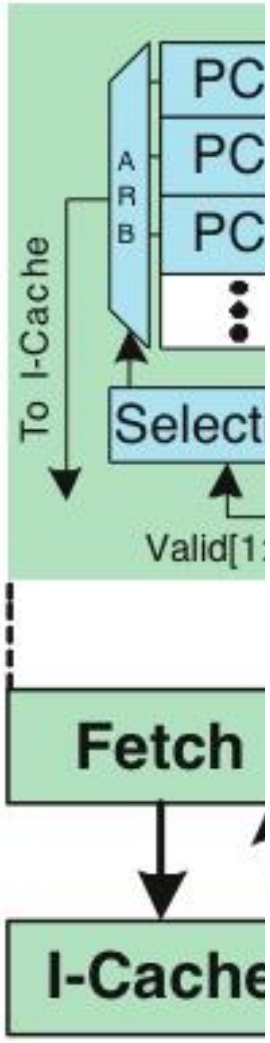


Shared Memory Bank Conflicts

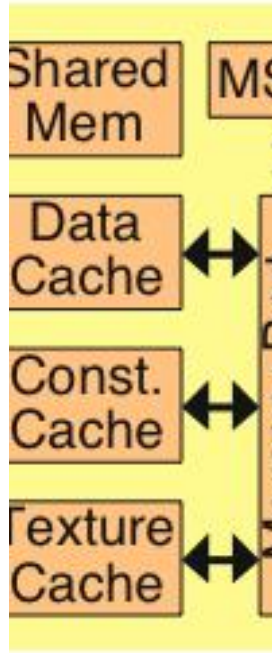
```
__shared__ int A[BSIZE];
…
A[2*threadIdx.x] = // 2-way
conflict
```

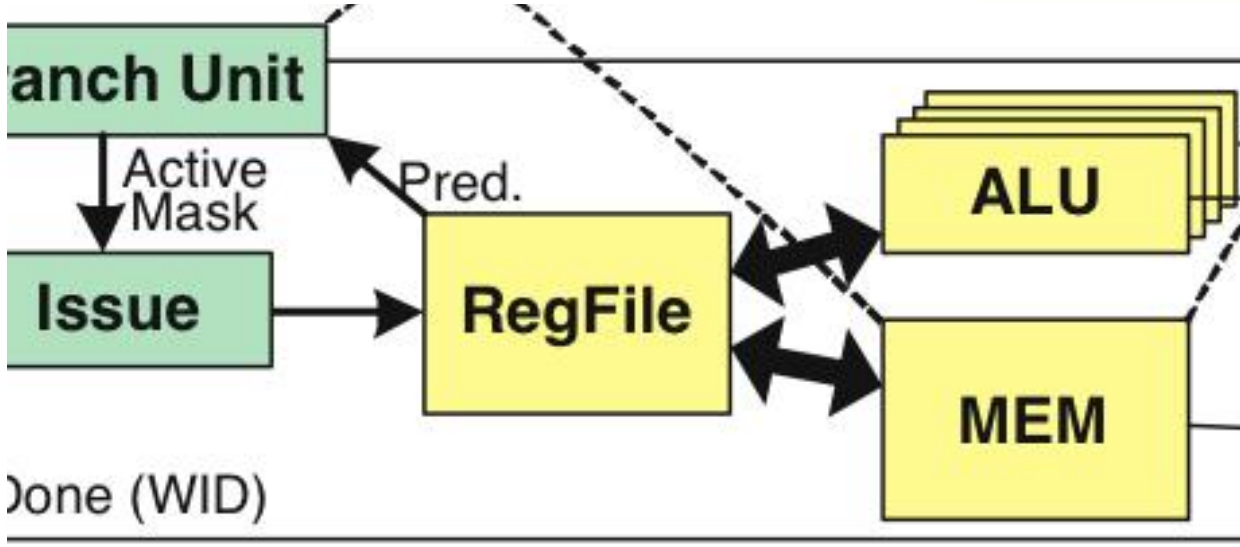


Register File



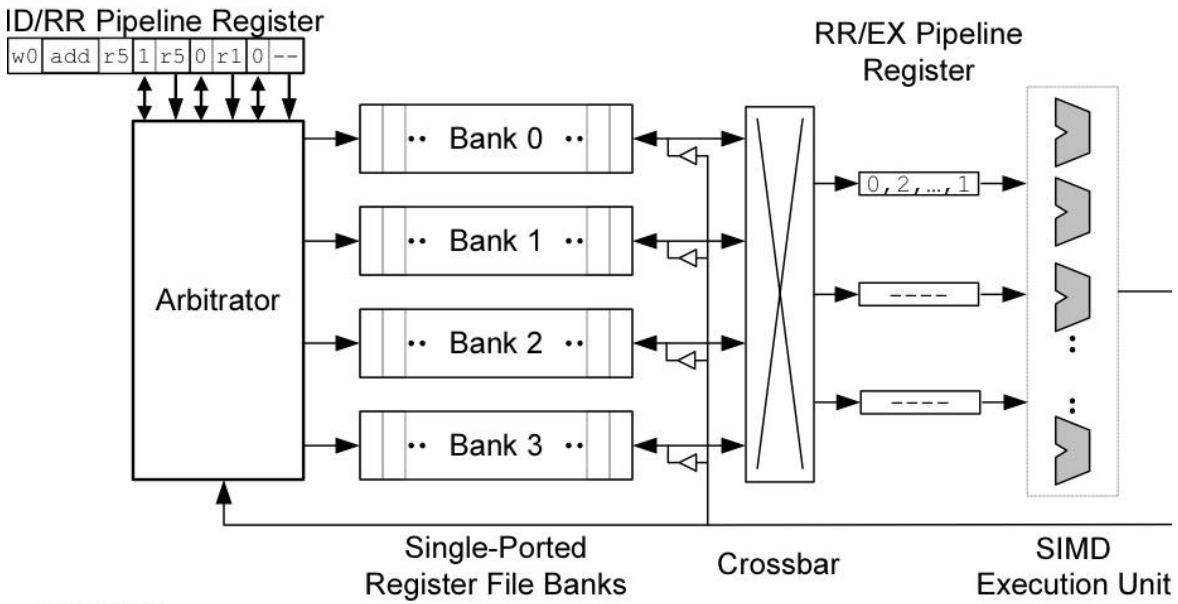
- 32 warps, 32 threads per warp, 16 x 32-bit registers per thread = 64KB register file.
- Need “4 ports” (e.g., FMA) greatly increase area.
- Alternative: banked single ported register file. How to avoid bank conflicts?





Banked Register File

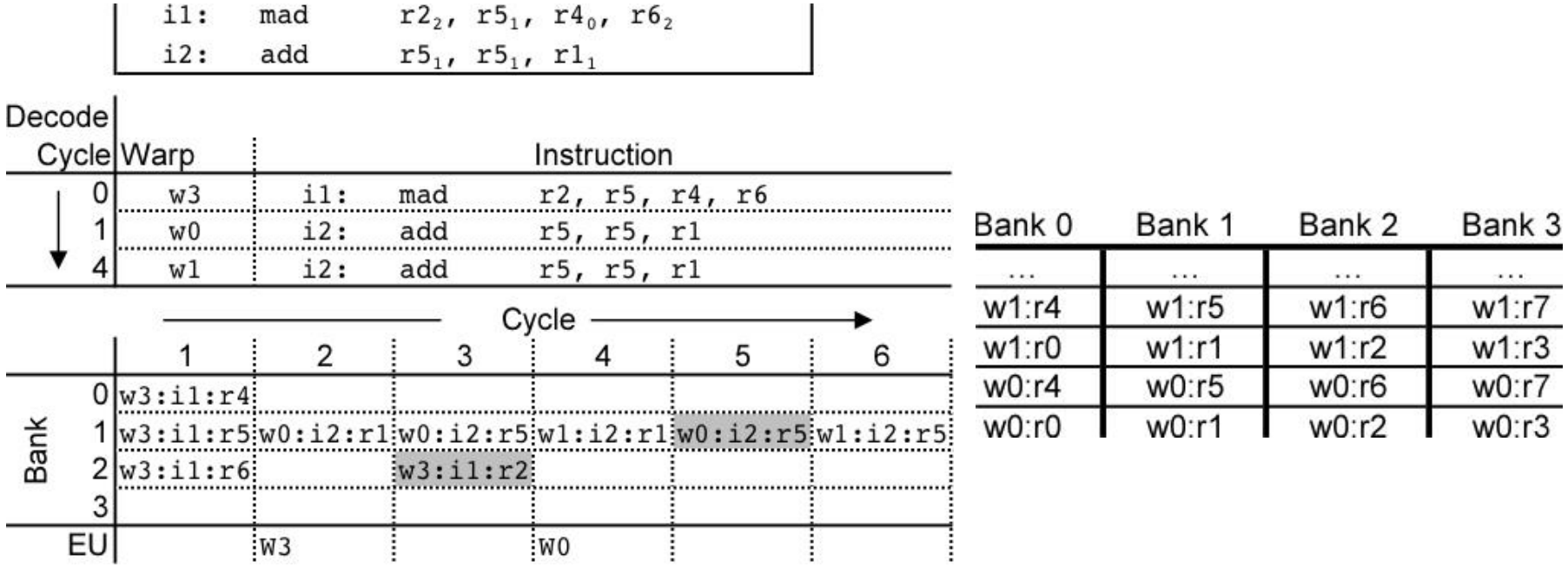
Strawman microarchitecture:



Register layout:

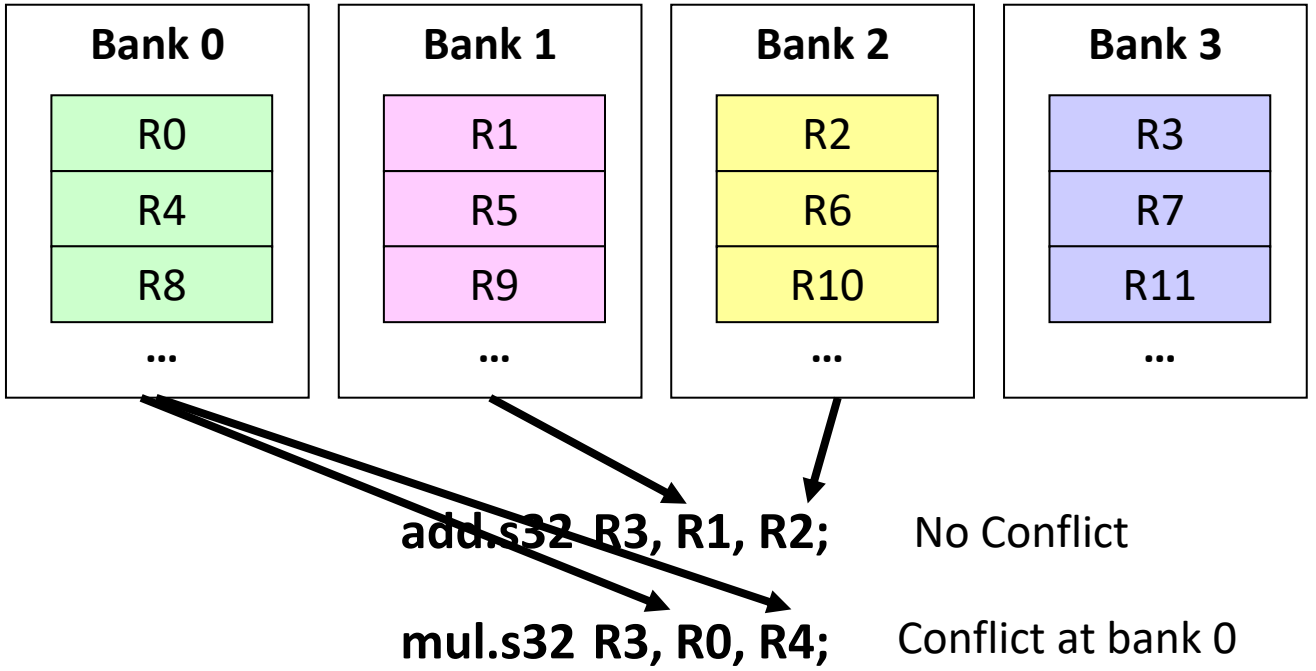
Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r4	w1:r5	w1:r6	w1:r7
w1:r0	w1:r1	w1:r2	w1:r3
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

Register Bank Conflicts

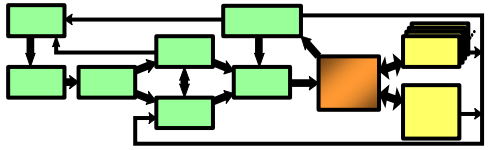


- warp 0, instruction 2 has two source operands in bank 1: takes two cycles to read.
- Also, warp 1 instruction 2 is same and is also stalled.
- Can use warp ID as part of register layout to help.

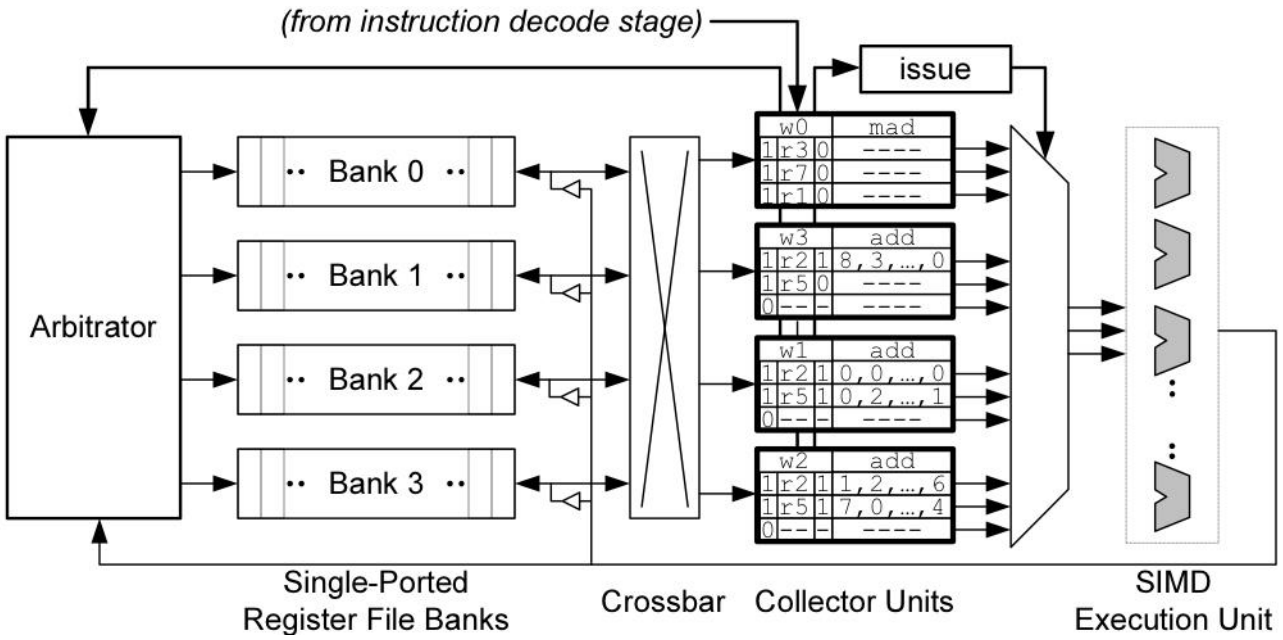
Operand Collector



- 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 →

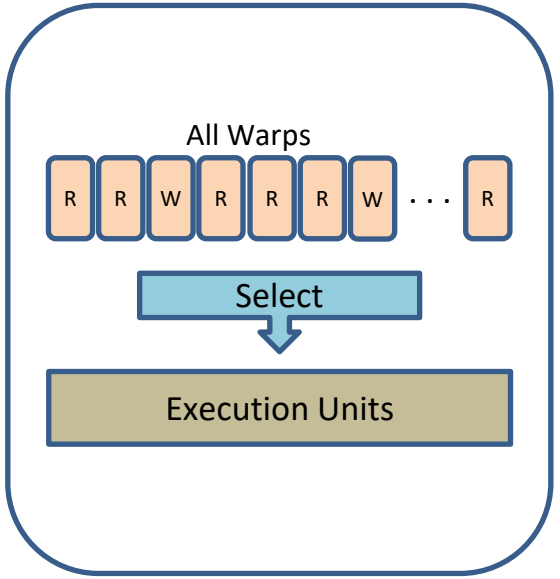
Bank	1	2	3	4	5	6
0		w2:r2		w3:r5		w3:r1
1			w3:r2			
2		w1:r5		w1:r1		
3	w1:r2		w2:r5	w0:r3	w2:r1	w0:r7
EU			w1	w2	w3	

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

Warp Scheduling Basics

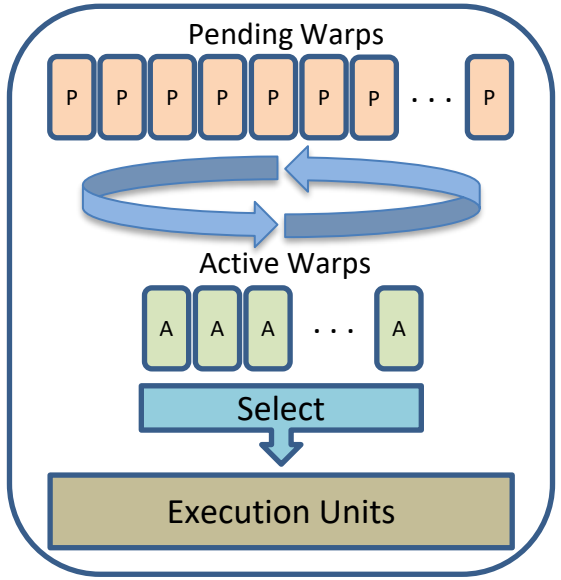
Loose Round Robin (LRR)

- Goes around to every warp and issue if ready (R)
- If warp is not ready (W), skip and issue next ready warp
- Issue: Warps all run at the same speed, potentially all reaching memory access phase together and stalling.



Two-level (TL)

- Warps are grouped into two groups:
  - Pending warps
    (potentially waiting on long latency instr.)
  - Active warps
    (Ready to execute)
  - Warps move between Pending and Active groups
- Within the Active group, issue LRR
- Goal: Overlap warps performing computation with warps performing memory access



Greedy-then-oldest (GTO)

- Schedule from a single warp until it stalls
- Then pick the oldest warp (time warp assigned to core)
- Goal: Improve cache locality for greedy warp