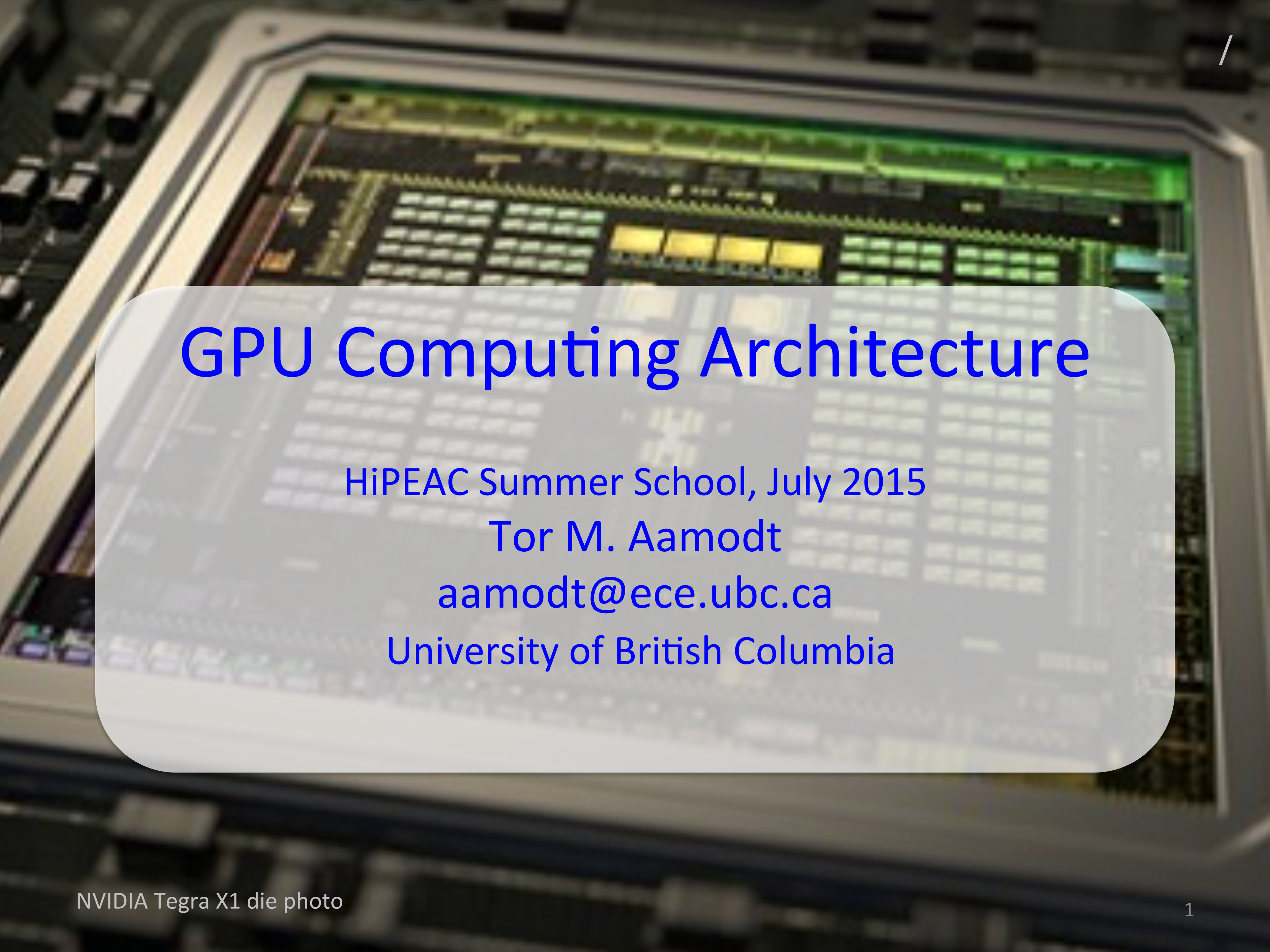
GPU Computing Architecture

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

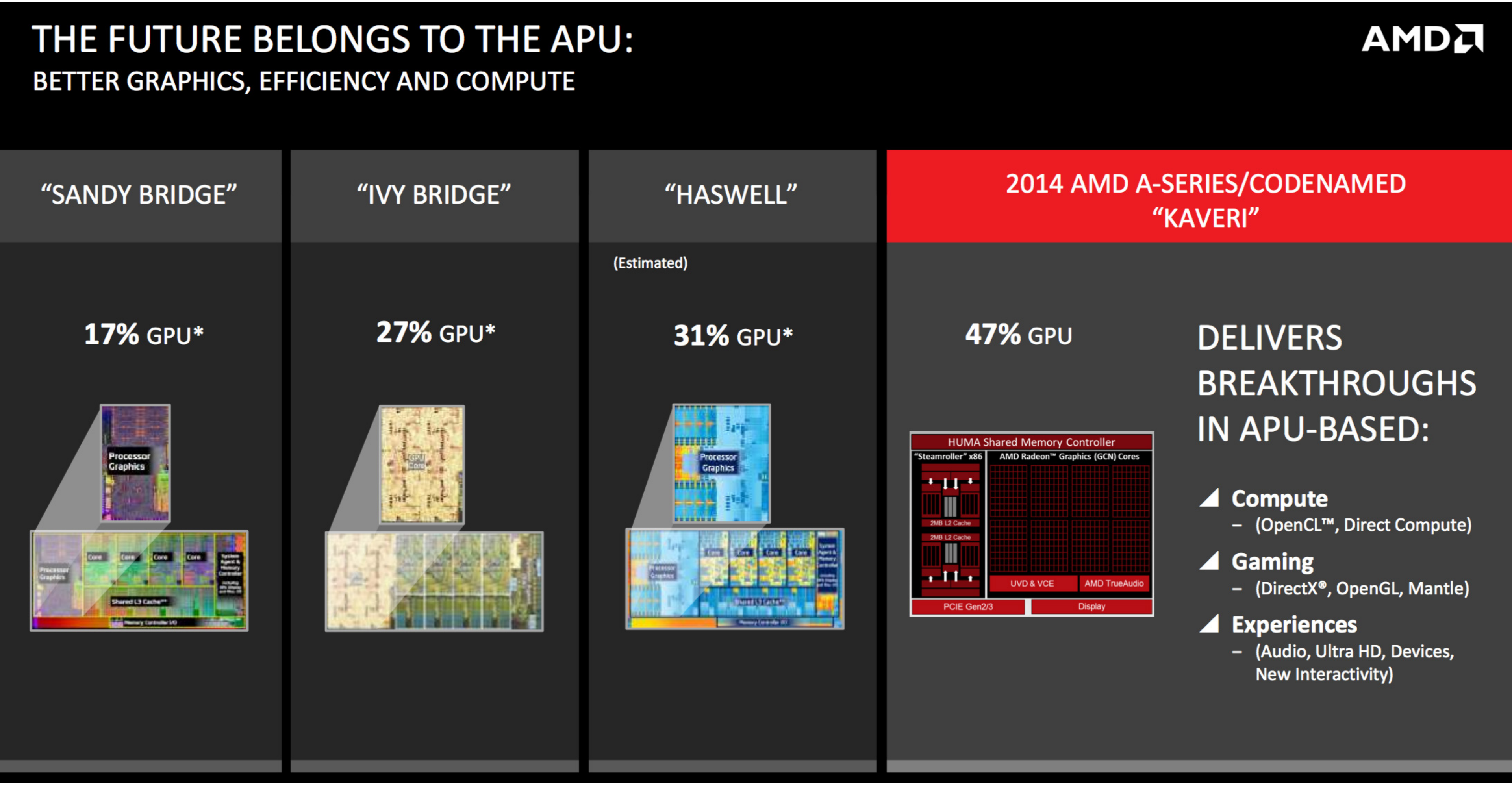


NVIDIA Tegra X1 die photo

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"
17% GPU*	27% GPU*	(Estimated) 31% GPU*	47% GPU

DELIVERS BREAKTHROUGHS IN APU-BASED:

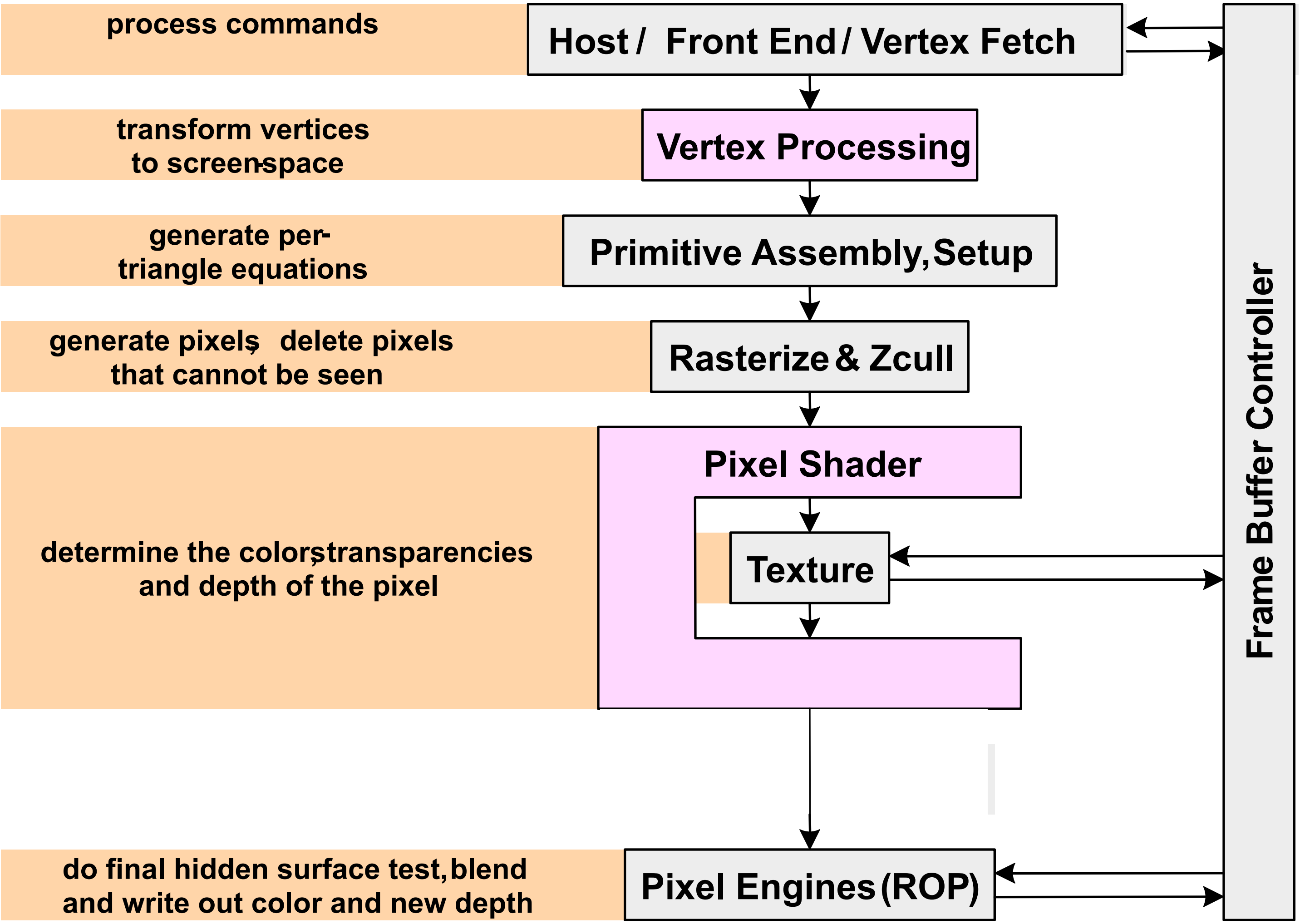
- **Compute**
  - (OpenCL™, Direct Compute)
- **Gaming**
  - (DirectX®, OpenGL, Mantle)
- **Experiences**
  - (Audio, Ultra HD, Devices, New Interactivity)

[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



process commands

Host / Front End / Vertex Fetch

transform vertices
to screen-space

Vertex Processing

generate per-
triangle equations

Primitive Assembly, Setup

generate pixels, delete pixels
that cannot be seen

Rasterize & Zcull

determine the colors, transparencies
and depth of the pixel

Pixel Shader

Texture

do final hidden surface test, blend
and write out color and new depth

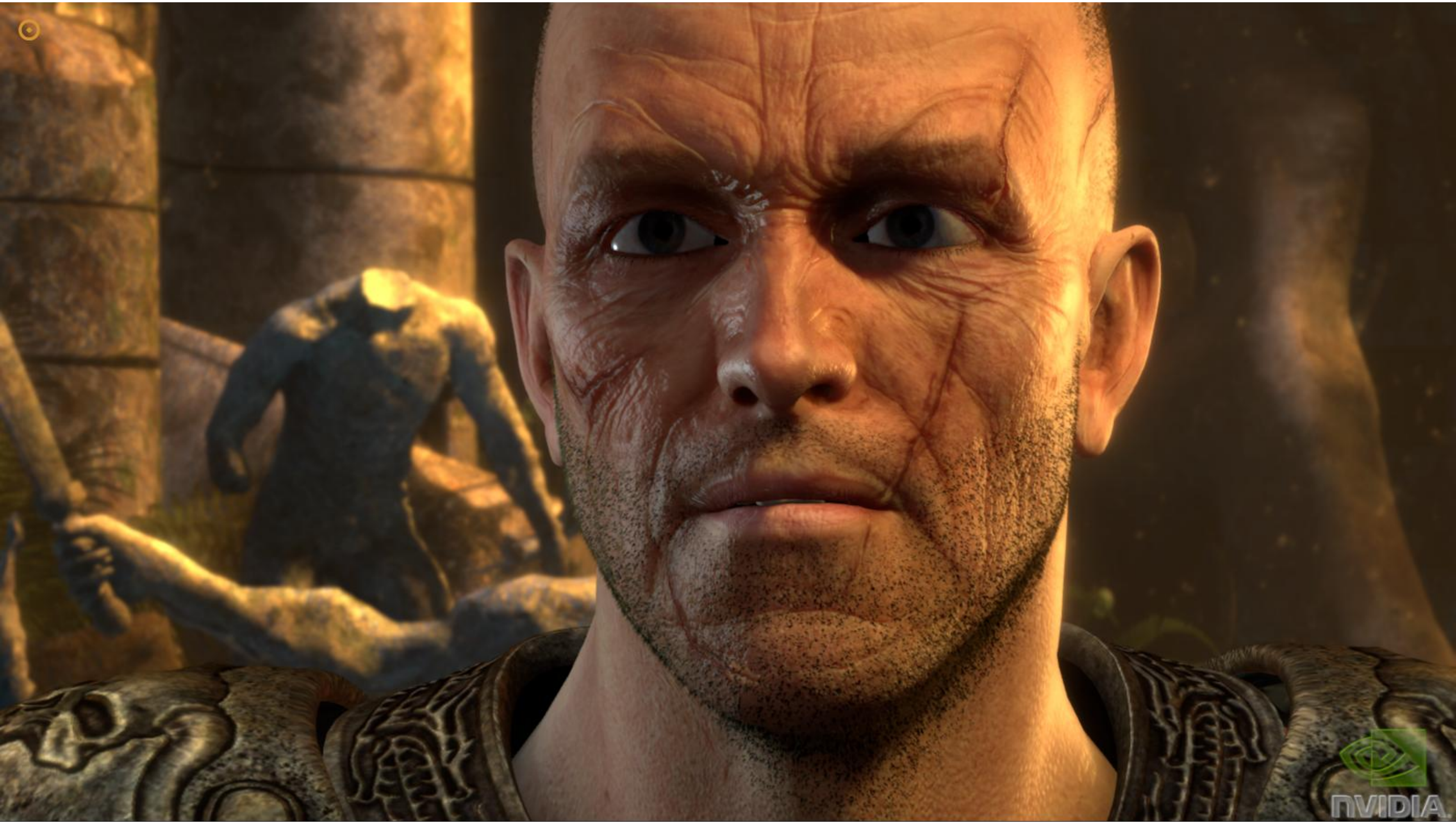
Pixel Engines (ROP)

Frame Buffer Controller

[David Kirk / Wen-mei Hwu]

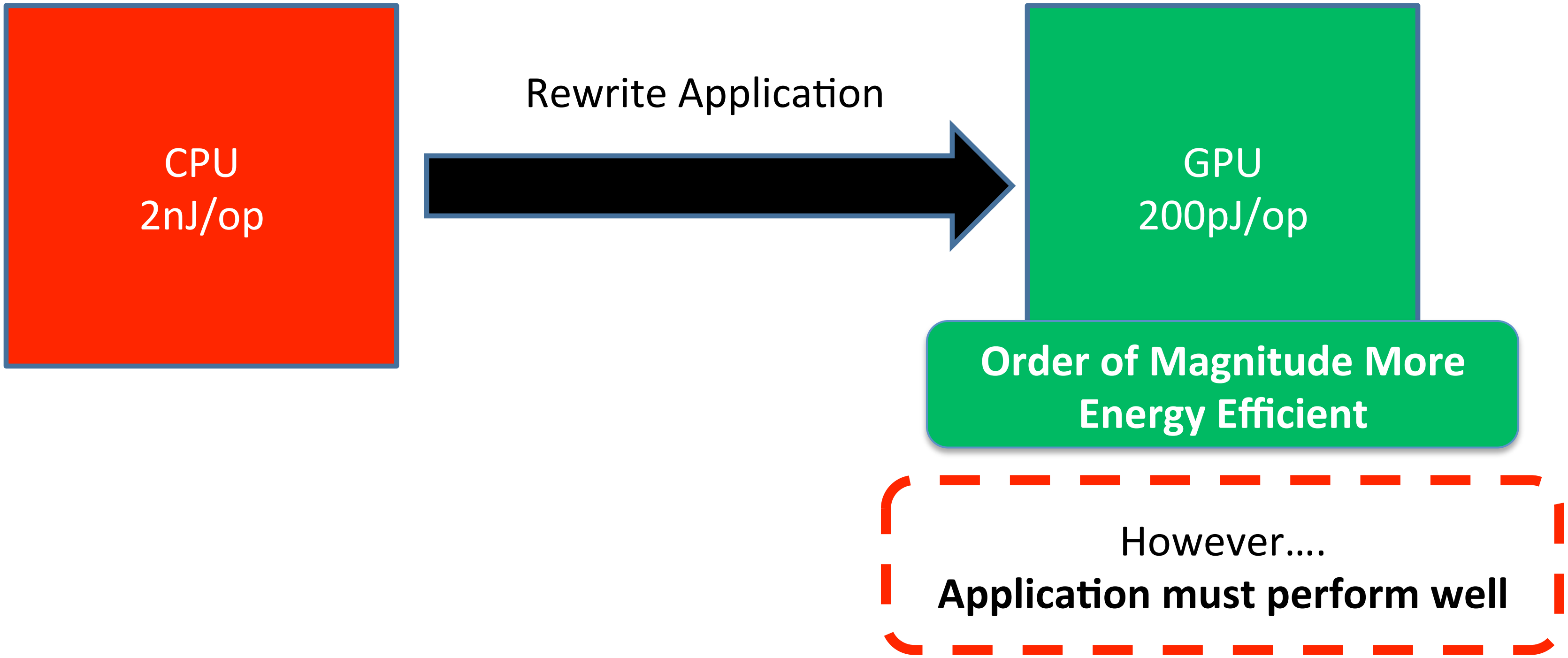
pixel color result of running “shader” program

+



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.



CPU
2nJ/op

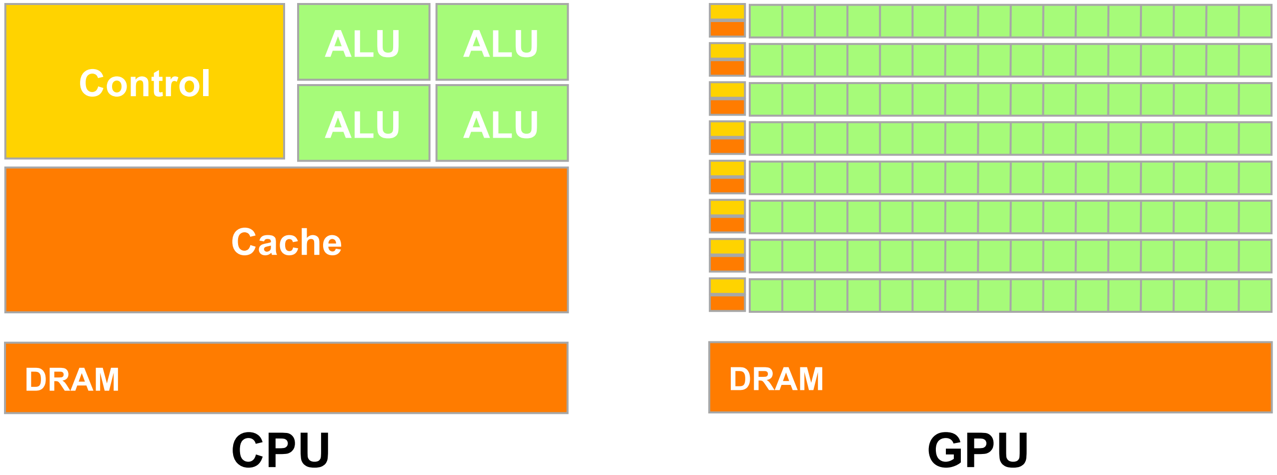
Rewrite Application

GPU
200pJ/op

Order of Magnitude More Energy Efficient

However….
Application must perform well

GPU uses larger fraction of silicon for computation than CPU?



[NVIDIA]

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

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.

Course Learning Objectives

After course you should be able to:

1. Explain motivation for investigating novel GPU-like computing architectures
2. Understand basic CUDA / PTX programs
3. Describe features of a generic GPU architecture representative of contemporary GPGPUs
4. Describe selected research on improving GPU computing programming models and hardware efficiency

Further Reading?

The following title is under development:

Tor M. Aamodt, Wilson W. L. Fung, Tim G. Rogers,
General Purpose Graphics Processor Architectures ,
Morgan and Claypool (late 2015 or early 2016)

Other resources (primarily research papers) will be mentioned throughout the lectures.

Course Outline

- Part 1: Introduction to GPGPU Programming Model
- Part 2: Generic GPGPU Architecture
- Part 3: Research Directions
  - Mitigating SIMT Control Divergence
  - Mitigating High GPGPU Memory Bandwidth Demands
  - Coherent Memory for Accelerators
  - Easier Programming with Synchronization

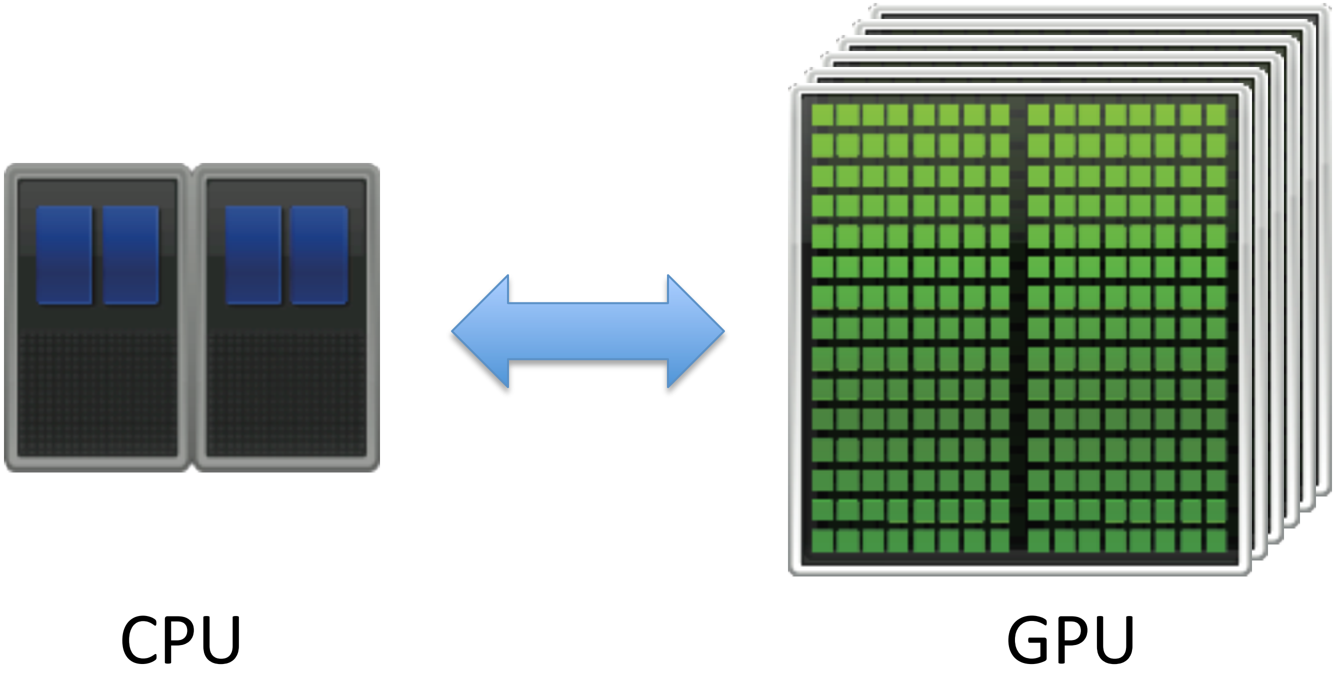
Part 1: Introduction to GPGPU Programming Model

GPGPU Programming Resources

- 9 week MOOC covering CUDA, OpenCL, C++AMP and OpenACC https://www.coursera.org/course/hetero

- Kirk and Hwu, Programming Massively Parallel Processors, Morgan Kaufmann, 2nd edition, 2014 (NOTE: 2nd edition includes coverage of OpenCL, C++AMP, and OpenACC)

GPU Compute Programming Model



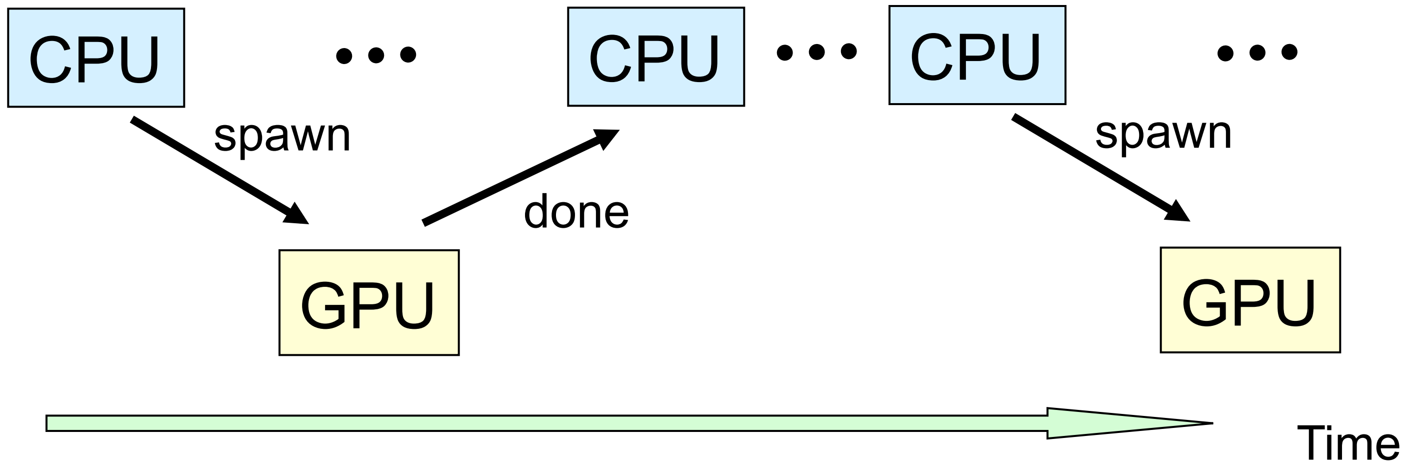
CPU

GPU

How is this system programmed (today)?

GPGPU Programming Model

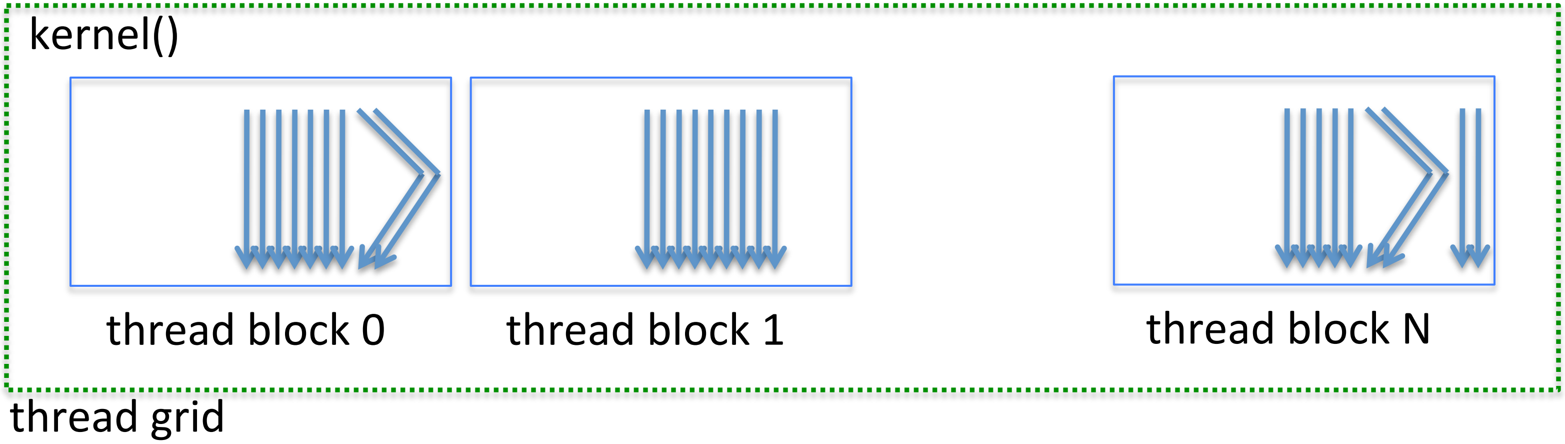
- CPU “Off-load” parallel kernels to GPU



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

CUDA/OpenCL Threading Model

CPU spawns fork-join style "grid" of parallel threads



kernel()

thread block 0

thread block 1

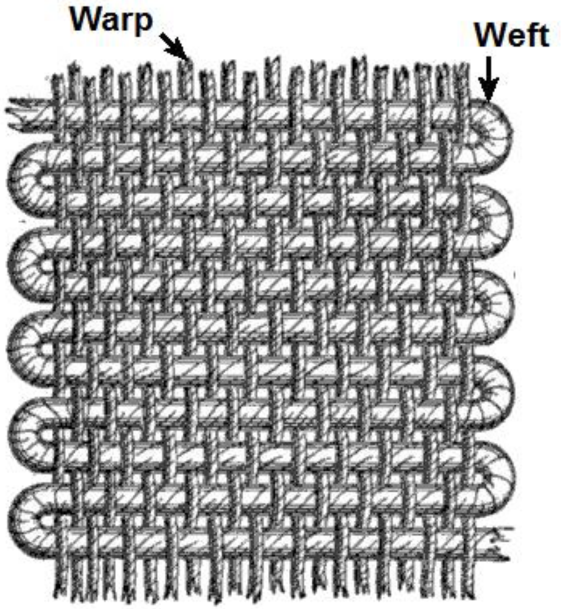
thread block N

thread grid

- 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.

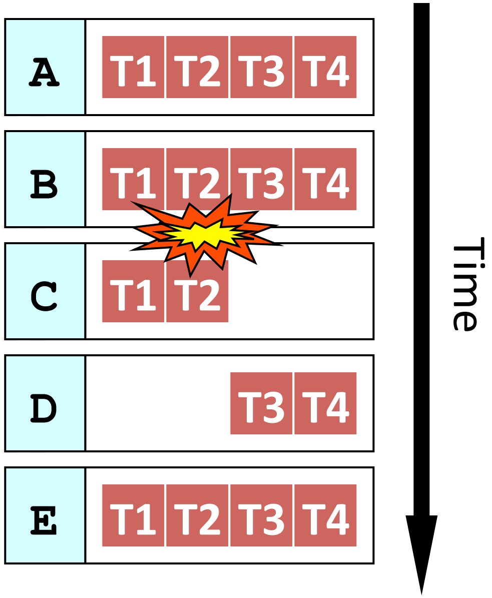


[https://en.wikipedia.org/wiki/Warp_and_woof]

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;
```



CUDA Syntax Extensions

- Declaration specifiers

```
__global__ void foo(...);  // kernel entry point (runs on GPU)
__device__ void bar(...); // function callable from a GPU thread
```

- Syntax for kernel launch

```
foo<<<500, 128>>>(...); // 500 thread blocks, 128 threads each
```

- Built in variables for thread identification

```
dim3 threadIdx; dim3 blockIdx; dim3 blockDim;
```

Example: Original C Code

```c
void saxpy_serial(int n, float a, float *x, float *y)
{
   for (int i = 0; i < n; ++i)
     y[i] = a*x[i] + y[i];
}

int main() {
  // omitted: allocate and initialize memory
  saxpy_serial(n, 2.0, x, y); // Invoke serial SAXPY kernel
  // omitted: using result
}
```

CUDA Code

```
__global__ void saxpy(int n, float a, float *x, float *y) {
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if(i<n) y[i]=a*x[i]+y[i];
}
```

Runs on GPU

```
int main() {
  // omitted: allocate and initialize memory
  int nblocks = (n + 255) / 256;

  cudaMalloc((void**) &d_x, n);
  cudaMalloc((void**) &d_y, n);
  cudaMemcpy(d_x,h_x,n*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_y,h_y,n*sizeof(float),cudaMemcpyHostToDevice);
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  cudaMemcpy(h_y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
  // omitted: using result
}
```

OpenCL Code

```
__kernel void saxpy(int n, float a, __global float *x, __global float *y) {
   int i = get_global_id(0);
   if(i<n) y[i]=a*x[i]+y[i];
}
```

Runs on GPU

```
int main() {
  // omitted: allocate and initialize memory on host, variable declarations

  int nblocks = (n + 255) / 256;
  int blocksize = 256;

  clGetPlatformIDs(1, &cpPlatform, NULL);
  clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &cdDevice, NULL);
  cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL, NULL, &ciErr1);
  cqCommandQueue = clCreateCommandQueue(cxGPUContext, cdDevice, 0, &ciErr1);
  dx = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, sizeof(cl_float) * n, NULL, &ciErr1);
  dy = clCreateBuffer(cxGPUContext, CL_MEM_READ_WRITE, sizeof(cl_float) * n, NULL, &ciErr1);

  // omitted: loading program into char string cSourceCL
  cpProgram = clCreateProgramWithSource(cxGPUContext, 1, (const char **)&cSourceCL, &szKernelLength,
    &ciErr1);
  clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL);
  ckKernel = clCreateKernel(cpProgram, “saxpy_serial”, &ciErr1);

  clSetKernelArg(ckKernel, 0, sizeof(cl_int), (void*)&n);
  clSetKernelArg(ckKernel, 1, sizeof(cl_float), (void*)&a);
  clSetKernelArg(ckKernel, 2, sizeof(cl_mem), (void*)&dx);
  clSetKernelArg(ckKernel, 3, sizeof(cl_mem), (void*)&dy);

  clEnqueueWriteBuffer(cqCommandQueue, dx, CL_FALSE, 0, sizeof(cl_float) * n, x, 0, NULL, NULL);
  clEnqueueWriteBuffer(cqCommandQueue, dy, CL_FALSE, 0, sizeof(cl_float) * n, y, 0, NULL, NULL);
  clEnqueueNDRangeKernel(cqCommandQueue, ckKernel, 1, NULL, &nblocks, & blocksize, 0, NULL, NULL);
  clEnqueueReadBuffer(cqCommandQueue, dy, CL_TRUE, 0, sizeof(cl_float) * n, y, 0, NULL, NULL);

  // omitted: using result
}
```

C++AMP Example Code

```
#include <amp.h>
using namespace concurrency;

int main() {
  // omitted: allocation and initialization of y and x
  array_view<int> xv(n, x);
  array_view<int> yv(n, y);
  parallel_for_each(yv.get_extent(), [=](index<1> i) restrict(amp) {
    yv[i] = a * xv[i] + yv[i];
  });
  yv.synchronize();
  // omitted: using result
}
```

Runs on GPU

OpenACC Example Code

```
void saxpy_serial(int n, float a, float *x, float *y)
{
   #pragma acc kernels
   for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
```

Runs on GPU

Review: Memory

- E.g., use to save state between steps in a computation.
- Each memory location has an associated *address* which identifies the location. The location contains a value:

Example: Memory with 4 one byte locations. Location with address 1 contains value 0x42.

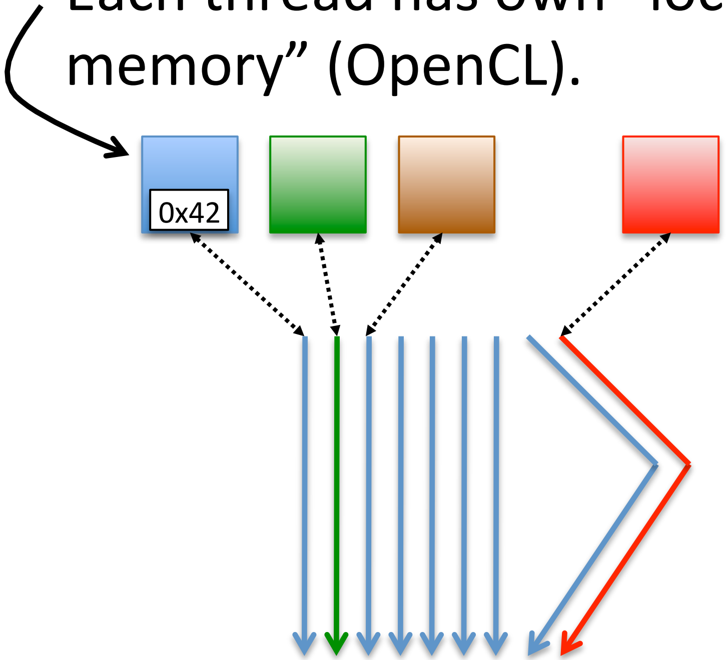
address	value
0	0xFF
1	0x42
2	0x00
3	0x01

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

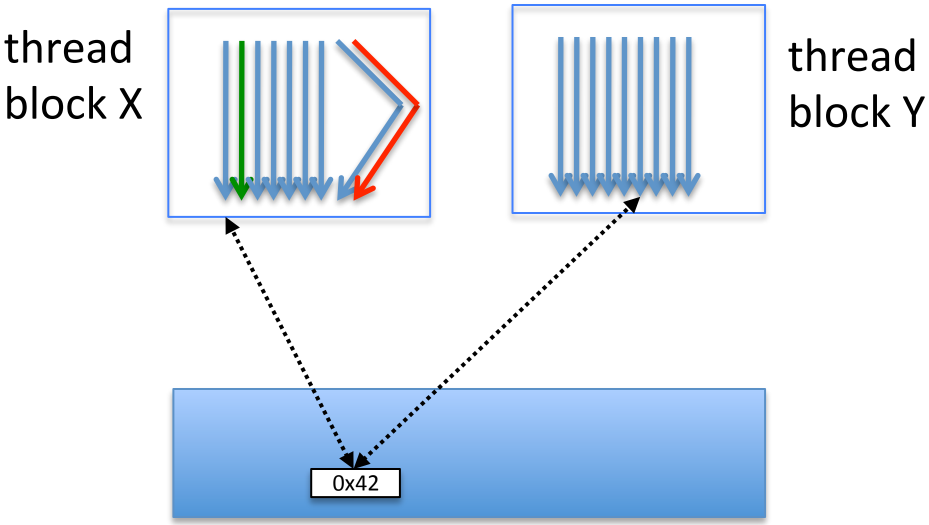
Each thread has own “local memory” (CUDA) “private memory” (OpenCL).



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.

History of “global memory”

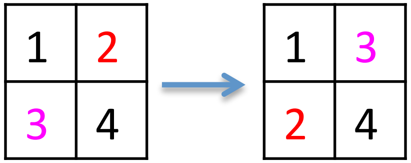
- Prior to NVIDIA GeForce 8800 and CUDA 1.0, access to memory was through texture reads and raster operations for writing.

- Problem: Address of memory access was highly constrained function of thread ID.

- CUDA 1.0 enabled access to arbitrary memory location in a flat memory space called “global”

Example: Transpose (CUDA SDK)

```
__global__ void transposeNaive(float *odata, float* idata, int width, int height)
{
  int xIndex = (blockIdx.x * TILE_DIM) + threadIdx.x;  // TILE_DIM = 16
  int yIndex = (blockIdx.y * TILE_DIM) + threadIdx.y;

  int index_in  = xIndex + (width * yIndex);
  int index_out = yIndex + (height * xIndex);
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) { // BLOCK_ROWS = 16
    odata[index_out+i] = idata[index_in+(i*width)];
  }
}
```

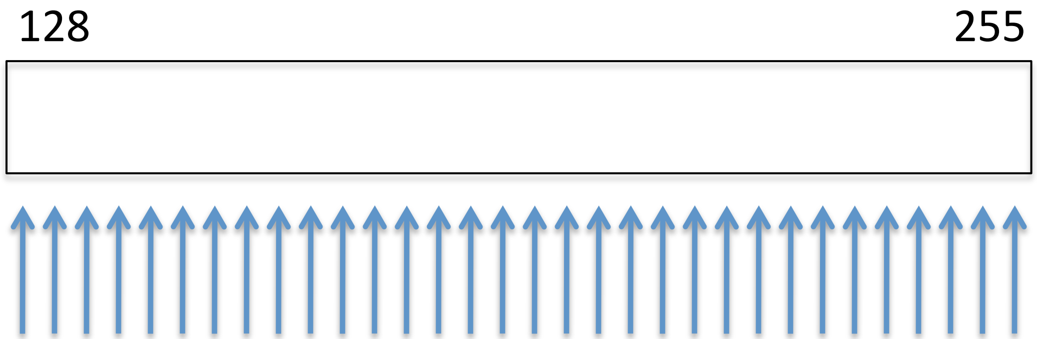


NOTE: “xIndex”, “yIndex”, “index_in”, “index_out”, and “i” are in <u>local memory</u>
(local variables are register allocated but stack lives in local memory)

“odata” and “idata” are pointers to <u>global memory</u>
(both allocated using calls to cudaMalloc -- not shown above)

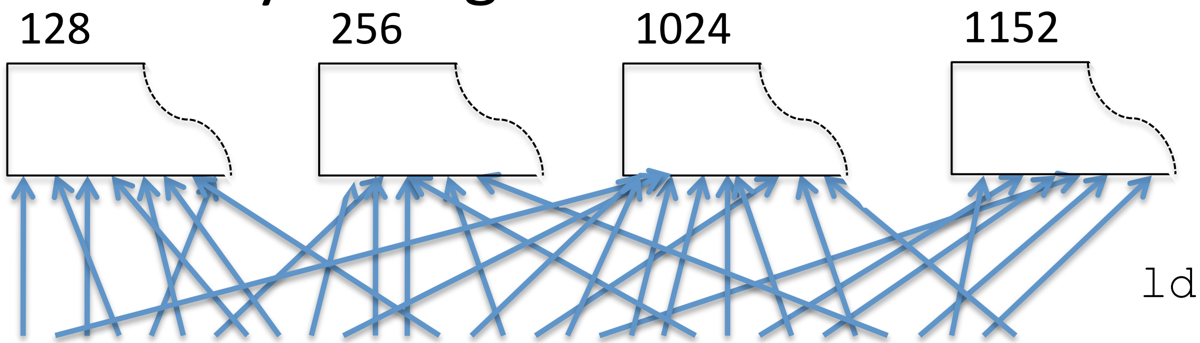
“Coalescing” global accesses

- 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)`

Example: Transpose (CUDA SDK)

```
__global__ void transposeNaive(float *odata, float* idata, int width, int height)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = xIndex + width * yIndex;
  int index_out = yIndex + height * xIndex;
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    odata[index_out+i] = idata[index_in+i*width];
  }
}
```

Assume height=16 and consider i=0:

Thread x=0,y=0 has xIndex=0, yIndex=0 so accesses odata[0]
Thread x=1,y=0 has xIndex=1, yIndex=0 so accesses odata[16]

Write to global memory highlighted above is not “coalesced”.

Redundant Global Memory Accesses

```
__global__ void matrixMul (float *C, float *A, float *B, int N)
{
  int xIndex = blockIdx.x * BLOCK_SIZE + threadIdx.x;
  int yIndex = blockIdx.y * BLOCK_SIZE + threadIdx.y;

  float sum = 0;

  for (int k=0; k<N; i++)
    sum += A[yIndex][k] * B[k][xIndex];

  C[yIndex][xIndex] = sum;
}
```

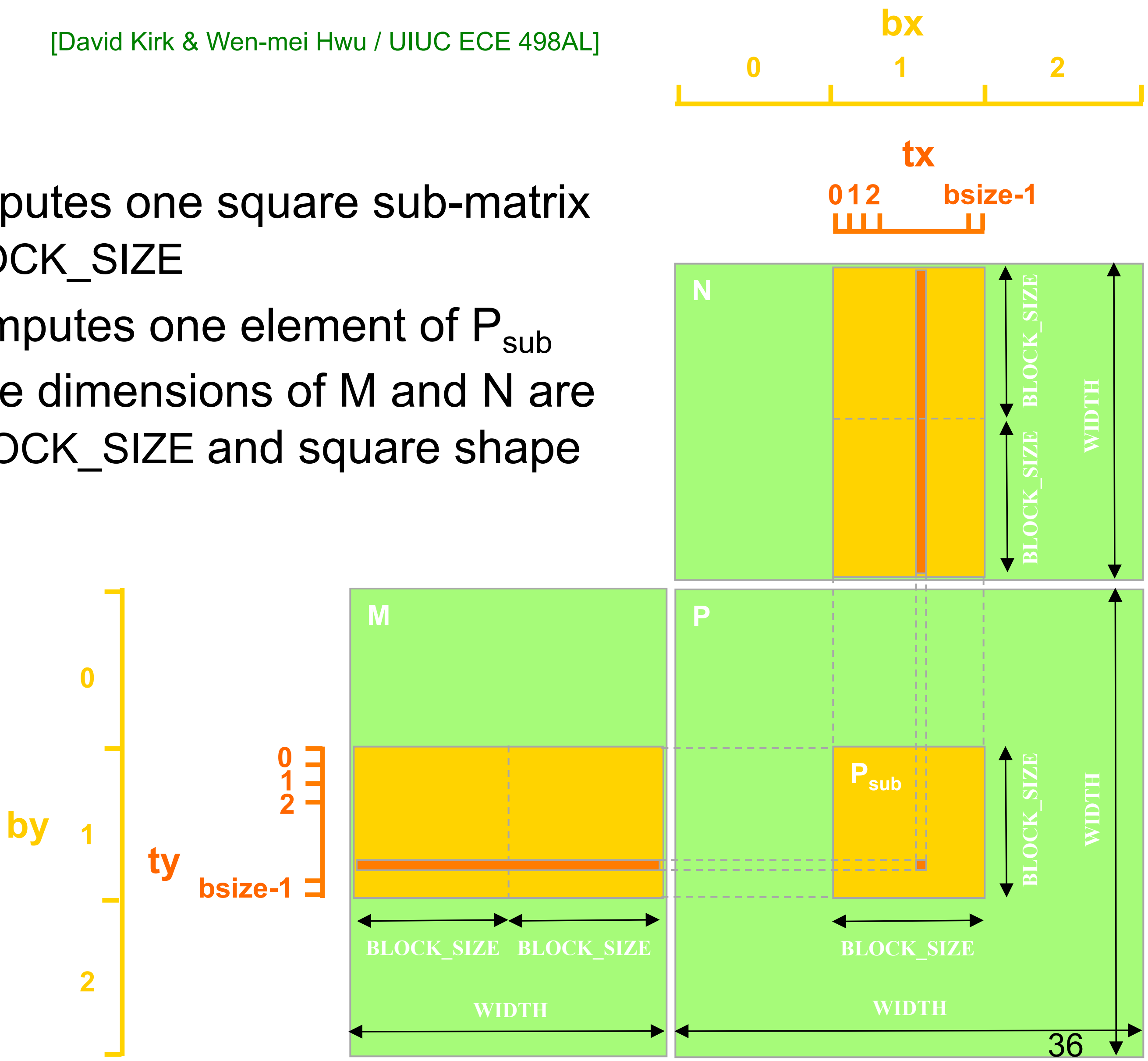
E.g., both thread x=0,y=0 and thread x=32, y=0 access A[0][0] potentially causing two accesses to off-chip DRAM. In general, each element of A and B is redundantly fetched O(N) times.

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

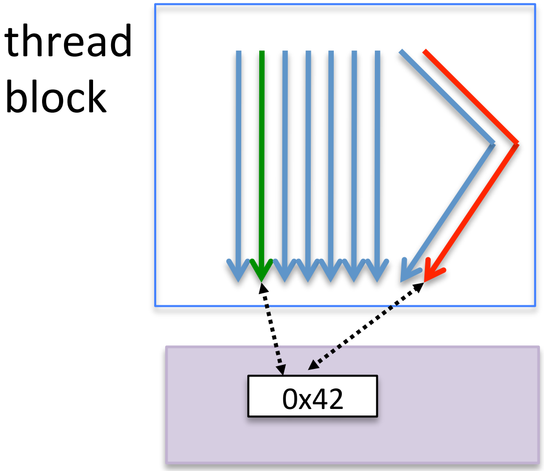


History of “shared memory”

- 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).

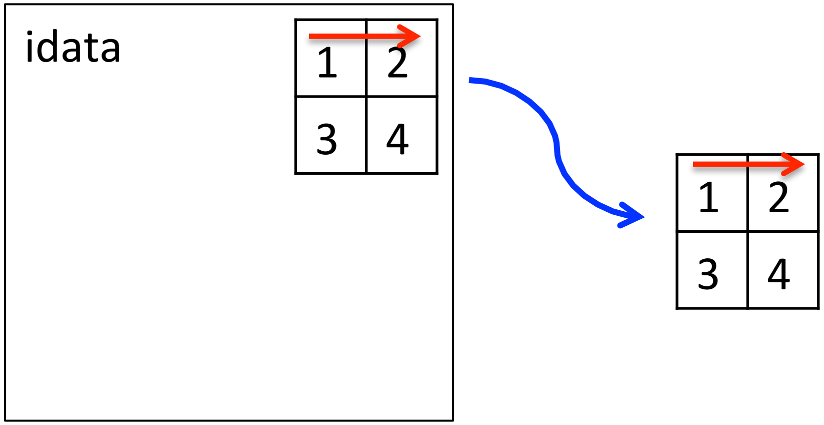
Shared memory address space is limited in size (16 to 48 KB).

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

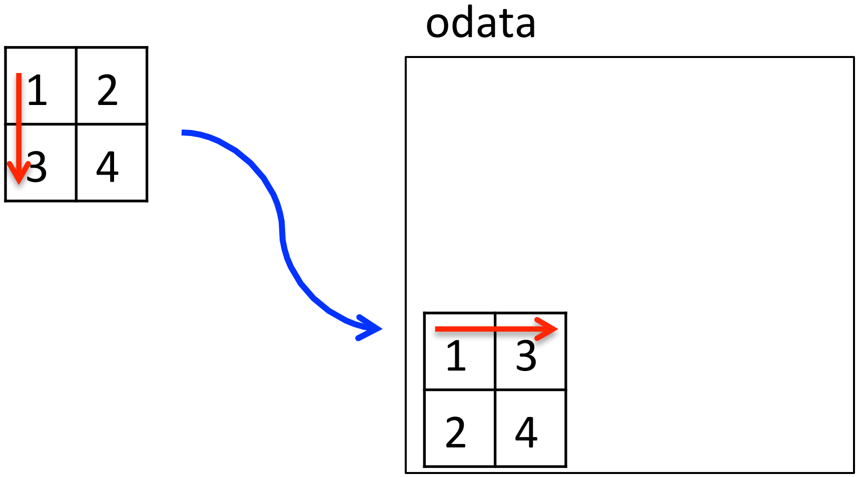
Synchronize threads in a thread block using __syncthreads();

Optimizing Transpose for Coalescing

Step 1: Read block of data into shared memory



Step 2: Copy from shared memory into global memory using coalesce write



Optimizing Transpose for Coalescing

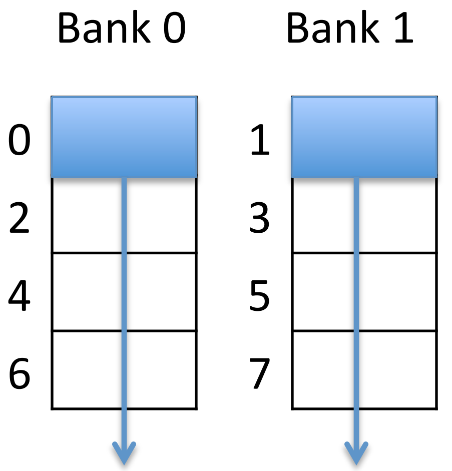
```
__global__ void transposeCoalesced(float *odata, float *idata, int width, int height)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = (blockIdx.x * TILE_DIM) + threadIdx.x;
  int yIndex = (blockIdx.y * TILE_DIM) + threadIdx.y;
  int index_in = xIndex + (width * yIndex);

  xIndex = (blockIdx.y * TILE_DIM) + threadIdx.x;
  yIndex = (blockIdx.x * TILE_DIM) + threadIdx.y;
  int index_out = xIndex + (yIndex*height);

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    tile[threadIdx.y+i][threadIdx.x] = idata[index_in+(i*width)];
  }

  __syncthreads();  // wait for all threads in block to finish above for loop

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
  }
}
```

GOOD: Coalesced write

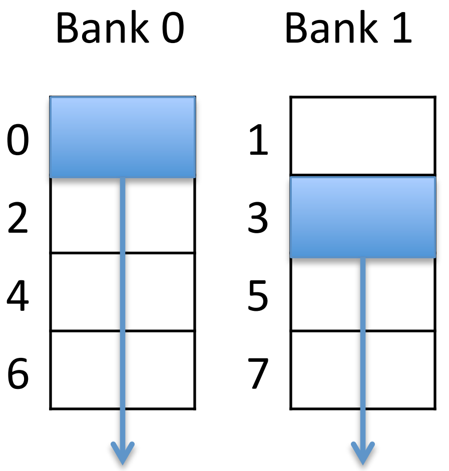
BAD: Shared memory bank conflicts

Review: Bank Conflicts

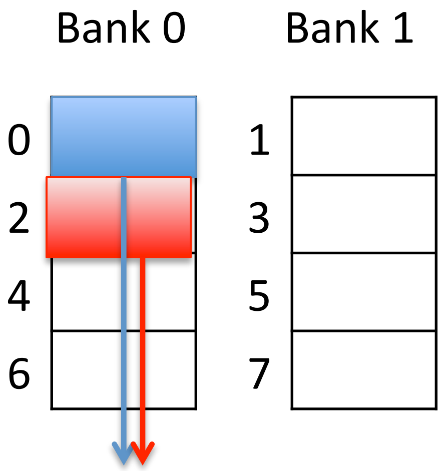
- To increase bandwidth common to organize memory into multiple banks.
- Independent accesses to different banks can proceed in parallel



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



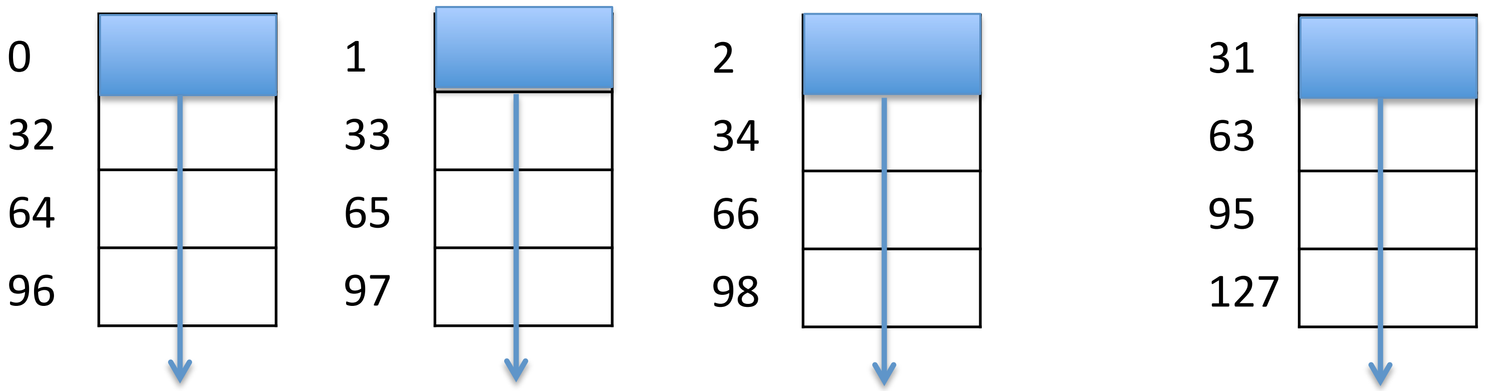
Example 2: Read 0, Read 3
(can proceed in parallel)



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

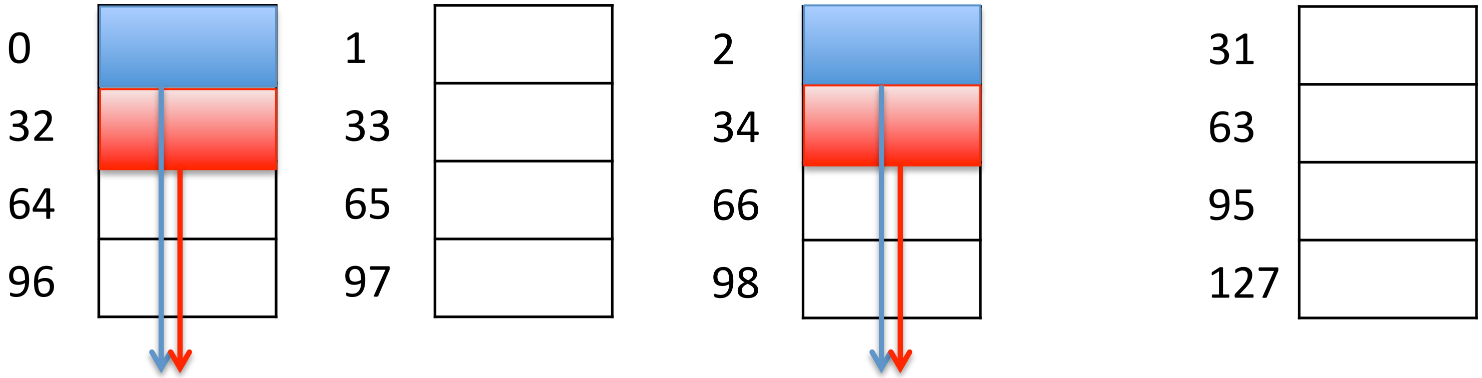
Shared Memory Bank Conflicts

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



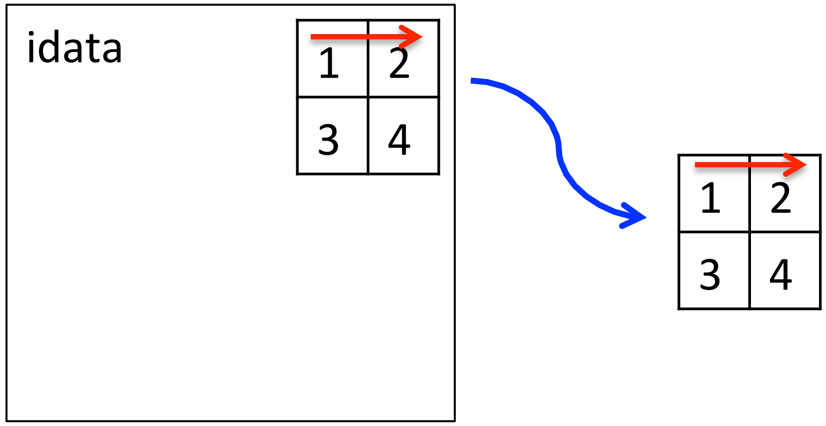
Shared Memory Bank Conflicts

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

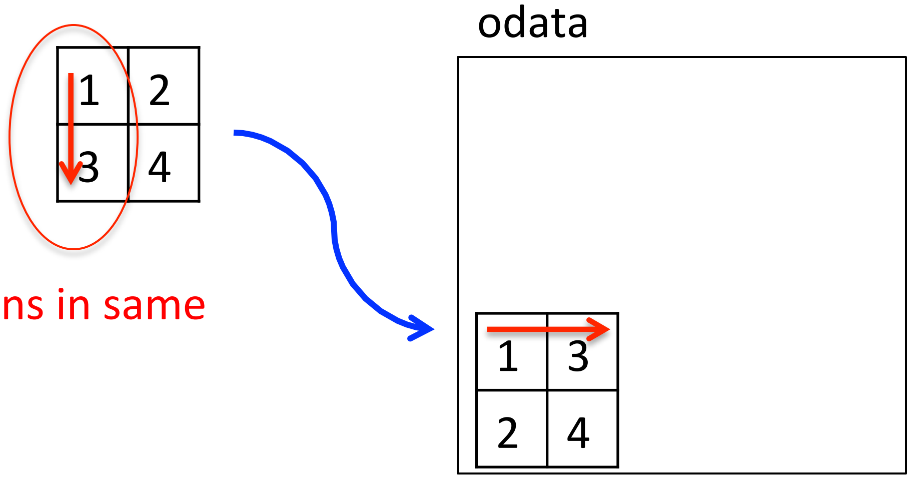


Optimizing Transpose for Coalescing

Step 1: Read block of data into shared memory



Step 2: Copy from shared memory into global memory using coalesce write



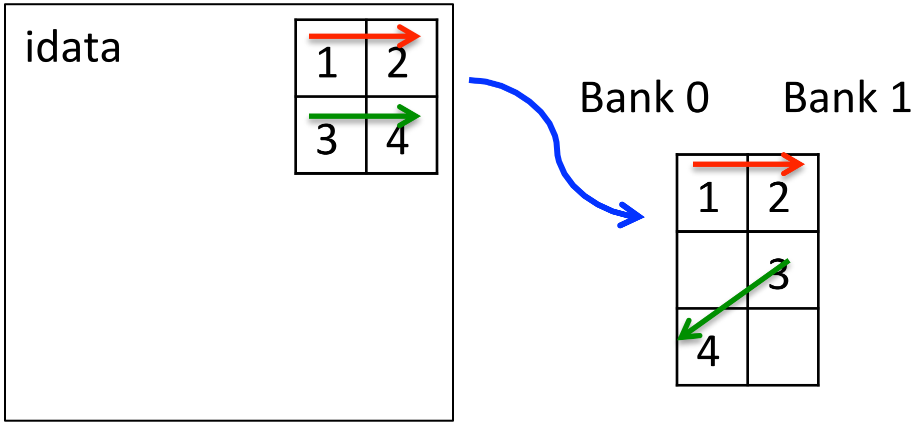
Problem: Access two locations in same shared memory bank.

+ Eliminate Bank Conflicts

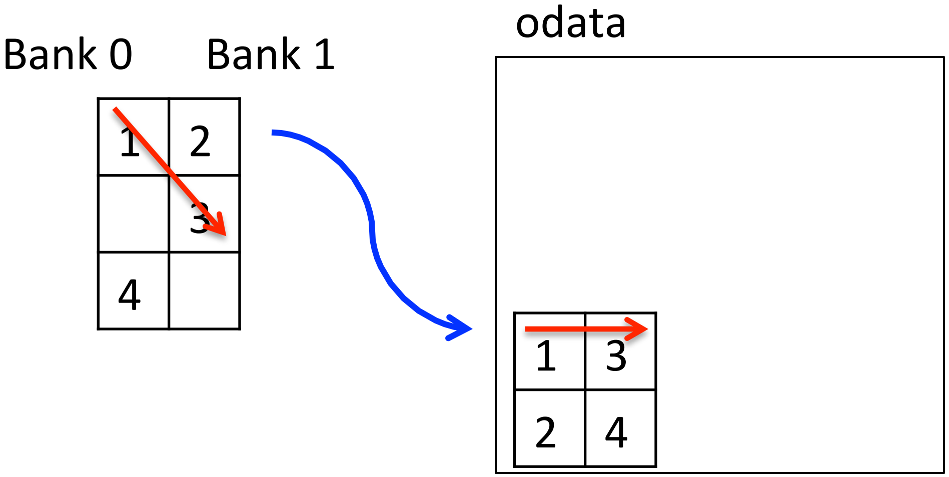
```
__global__ void transposeNoBankConflicts (float *odata, float *idata, int width, int height)
{
  __shared__ float tile[TILE_DIM][TILE_DIM+1];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
  }

  __syncthreads();

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
  }
}
```

Optimizing Transpose for Coalescing

Step 1: Read block of data into shared memory



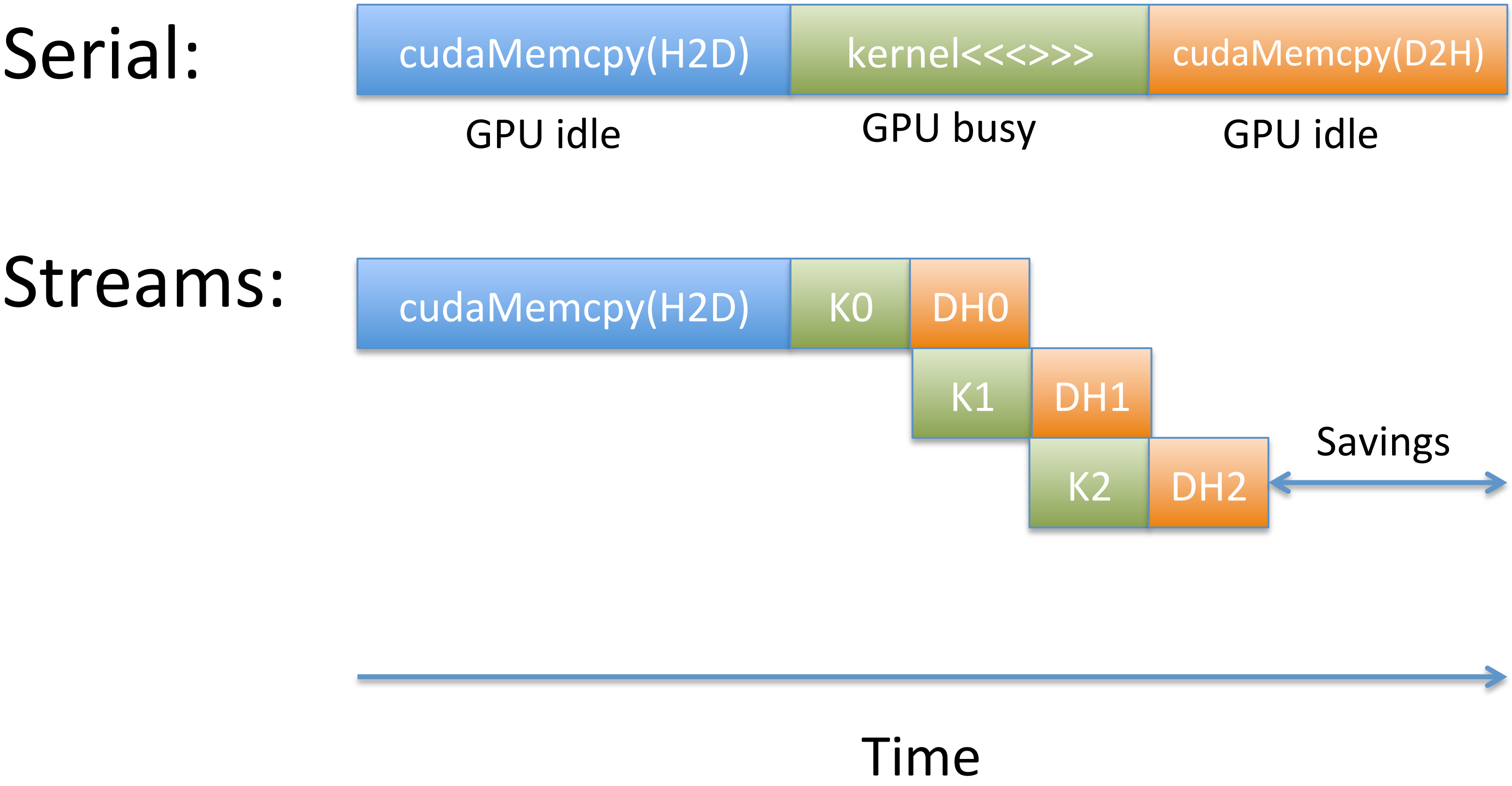
Step 2: Copy from shared memory into global memory using coalesce write



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?



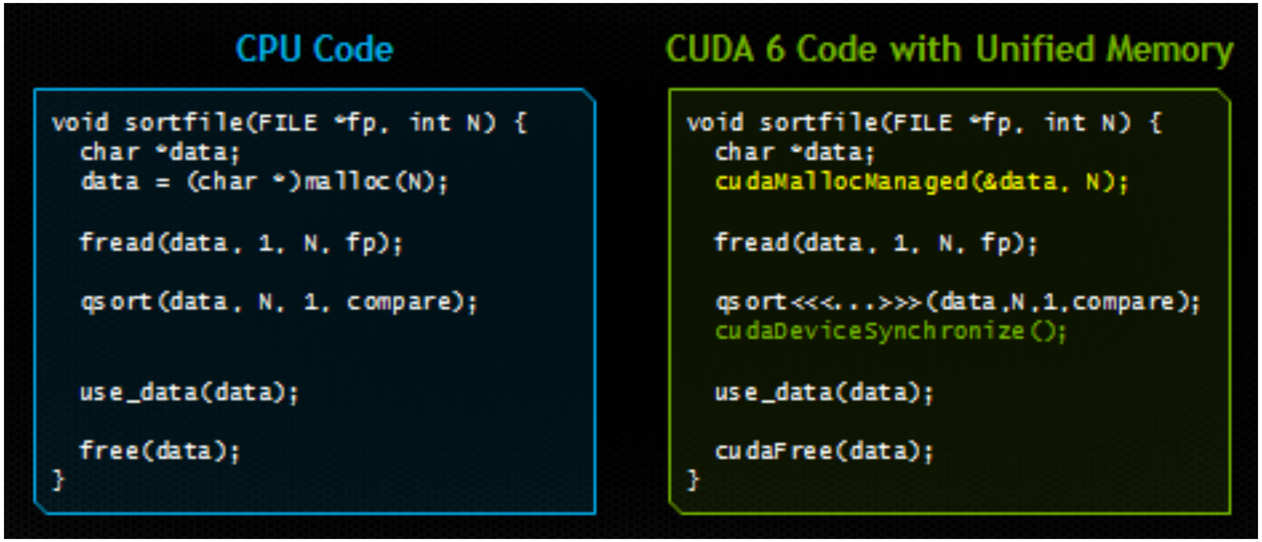
http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf

CUDA Streams

```
cudaStream_t streams[3];
for(i=0; i<3; i++)
  cudaStreamCreate(&streams[i]);  // initialize streams

for(i=0; i<3; i++) {
  cudaMemcpyAsync(pD+i*size,pH+i*size,size,
    cudaMemcpyHostToDevice,stream[i]);             // H2D
  MyKernel<<<grid,block,0,stream[i]>>>(pD+i,size); // compute
  cudaMemcpyAsync(pD+i*size,pH+i*size,size,
    cudaMemcpyDeviceToHost,stream[i]);             // D2H
}
```

Recent Features in CUDA

- Dynamic Parallelism (CUDA 5): Launch kernels from within a kernel. Reduce work for e.g., adaptive mesh refinement.
- Unified Memory (CUDA 6): Avoid need for explicit memory copies between CPU and GPU



CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

See also, Gelado, et al. ASPLOS 2010.

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
```

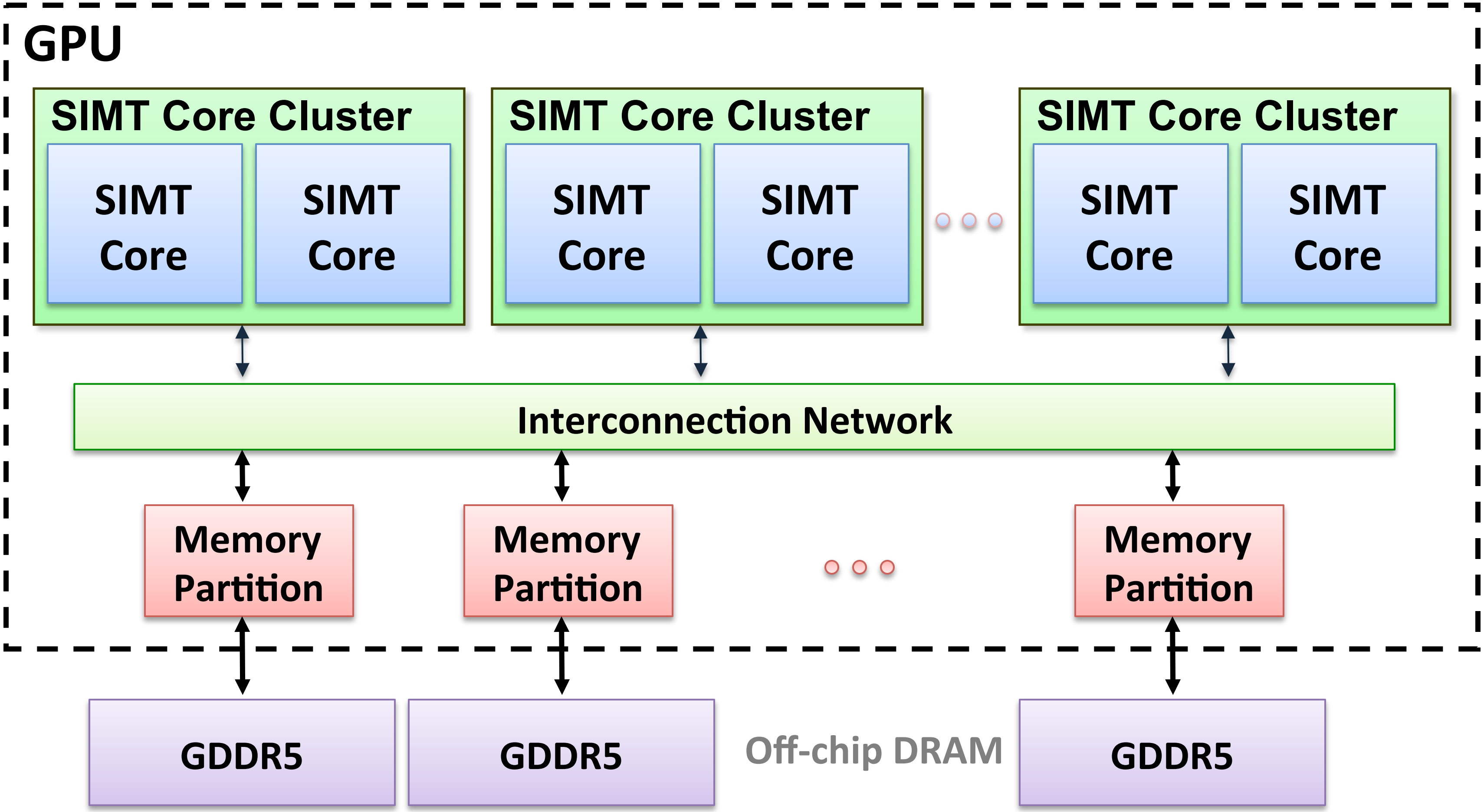
Part 2: Generic GPGPU Architecture

Extra resources

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

GPU Microarchitecture Overview

Single-Instruction, Multiple-Threads

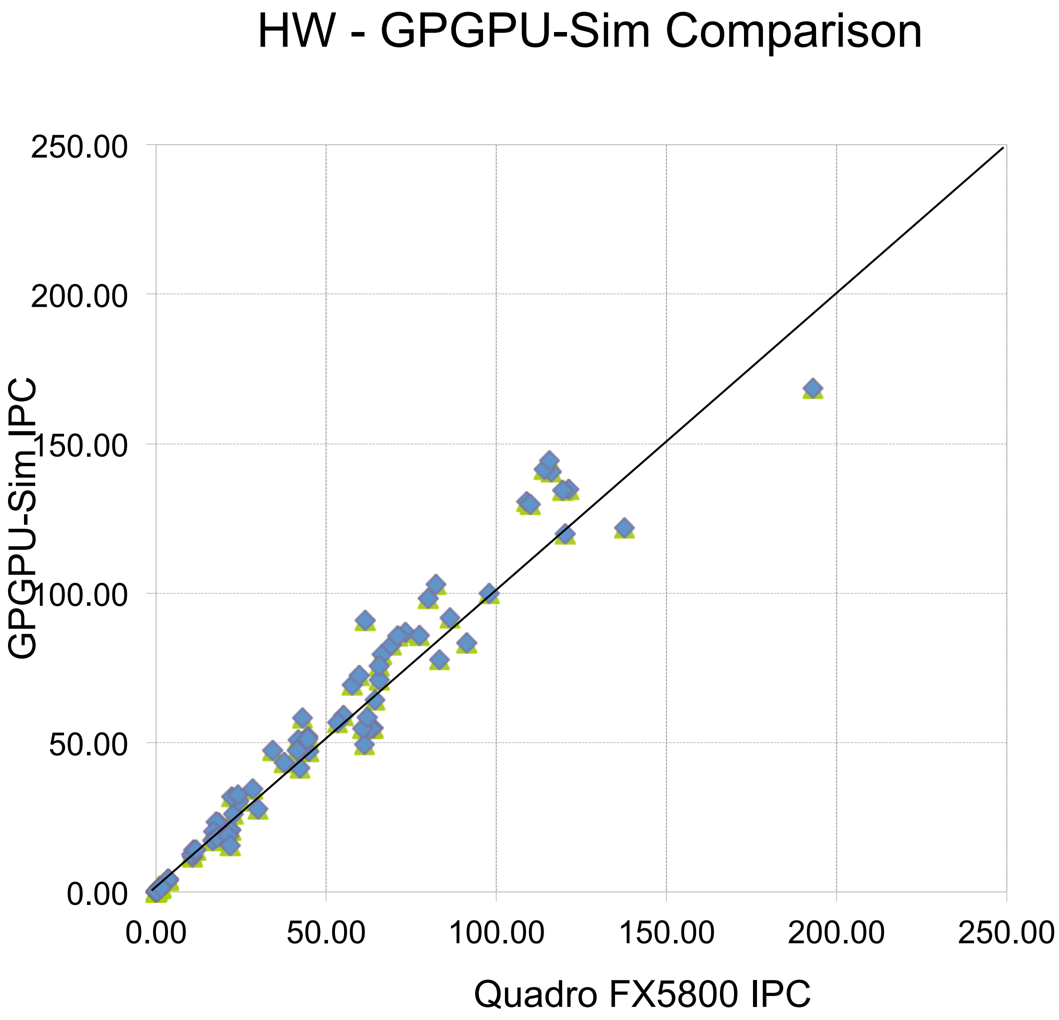


GPU Microarchitecture

- 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

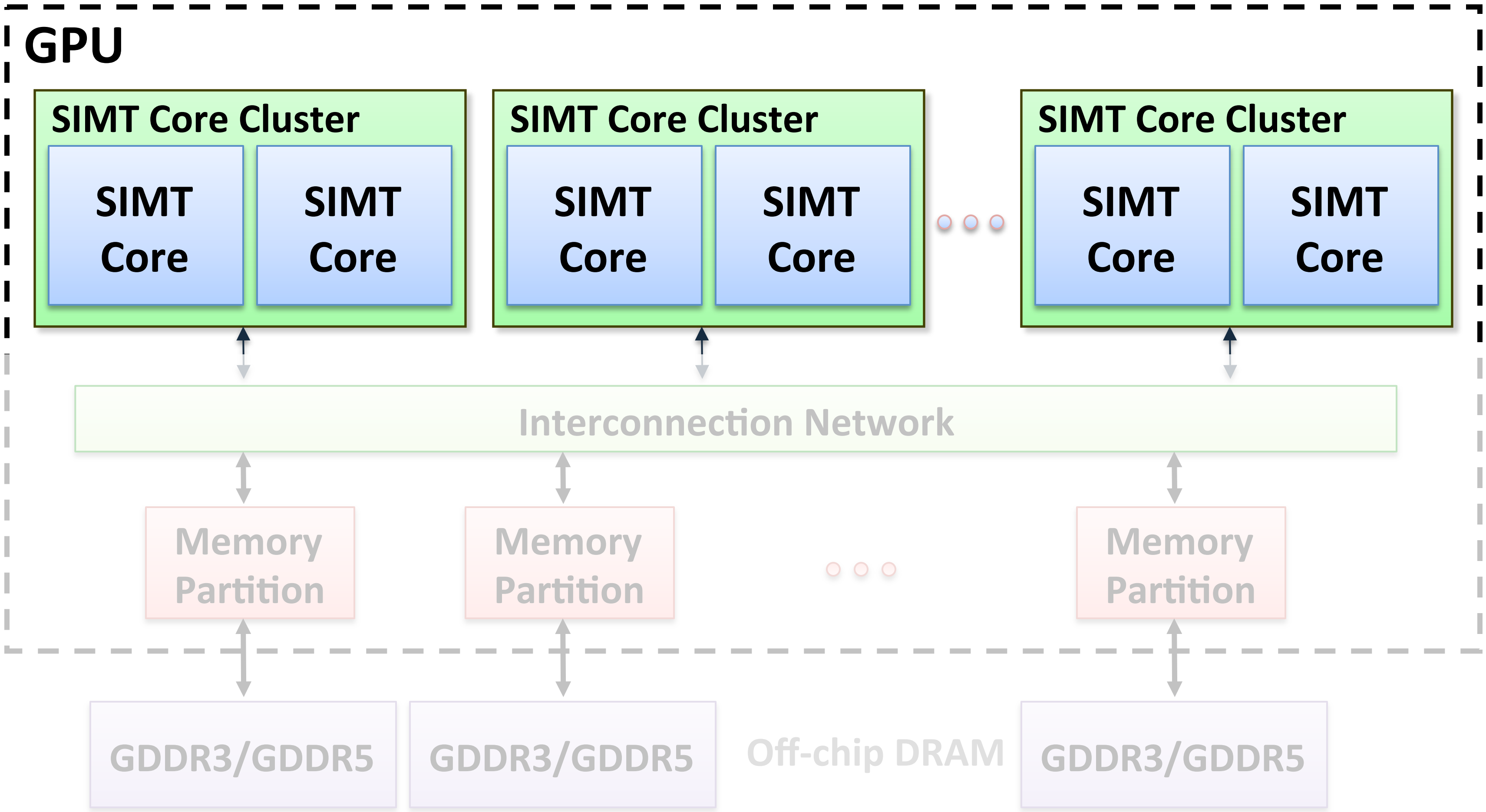
- Model described next, embodied in GPGPU-Sim, developed from: white papers, programming manuals, IEEE Micro articles, patents.

GPGPU-Sim v3.x w/ SASS

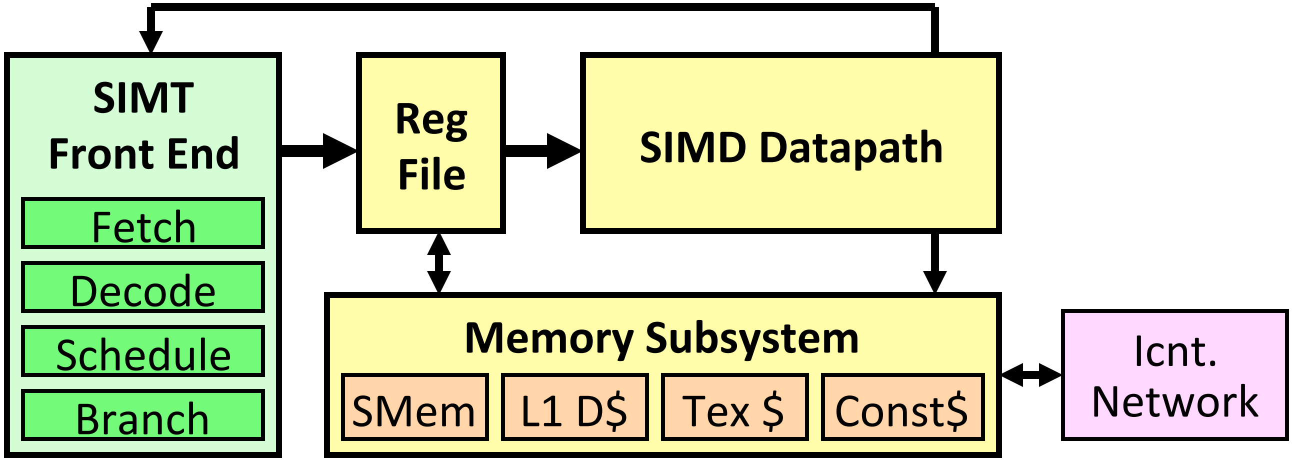


Correlation
~0.976

GPU Microarchitecture Overview

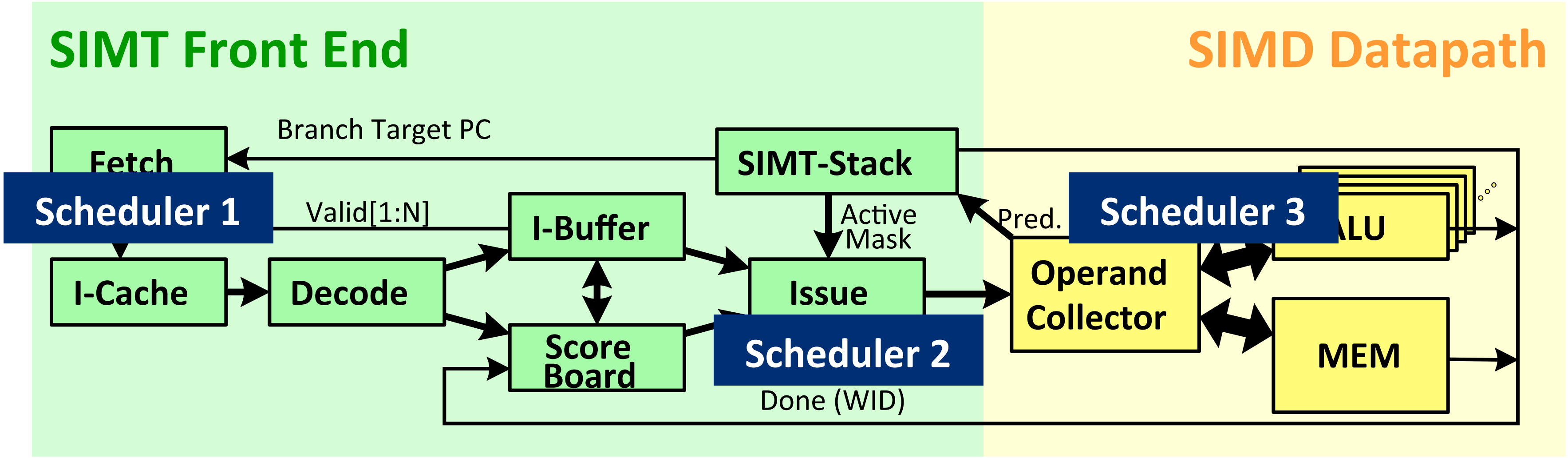


Inside a SIMT Core



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

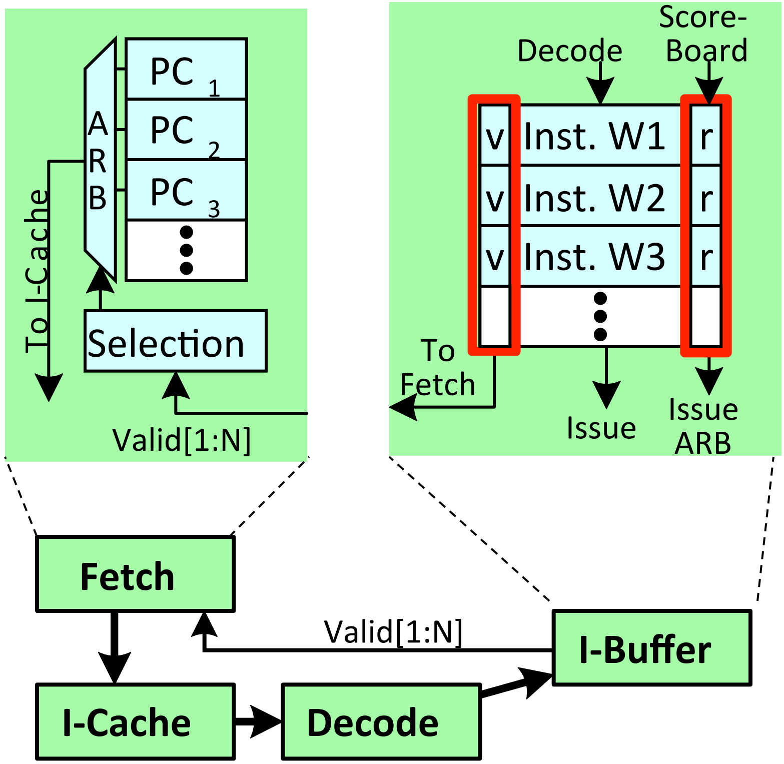
Inside an “NVIDIA-style” SIMT Core

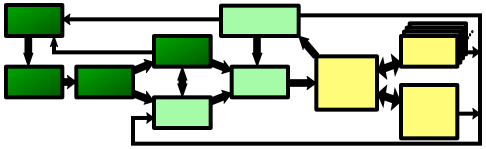


- Three decoupled warp schedulers
- Scoreboard
- Large register file
- Multiple SIMD functional units

Fetch + Decode

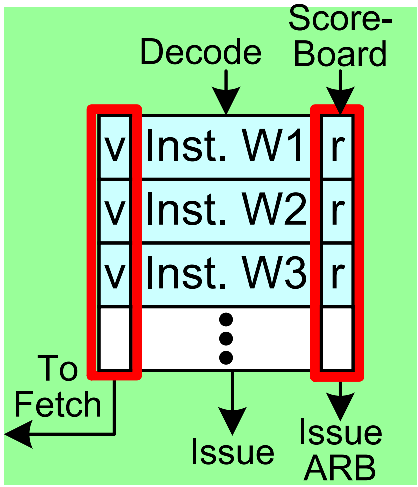
- Arbitrate the I-cache among warps
  - Cache miss handled by fetching again later
- Fetched instruction is decoded and then stored in the I-Buffer
  - 1 or more entries / warp
  - Only warp with vacant entries are considered in fetch

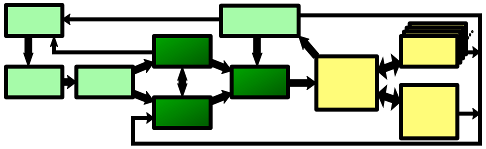




Instruction Issue

- 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)

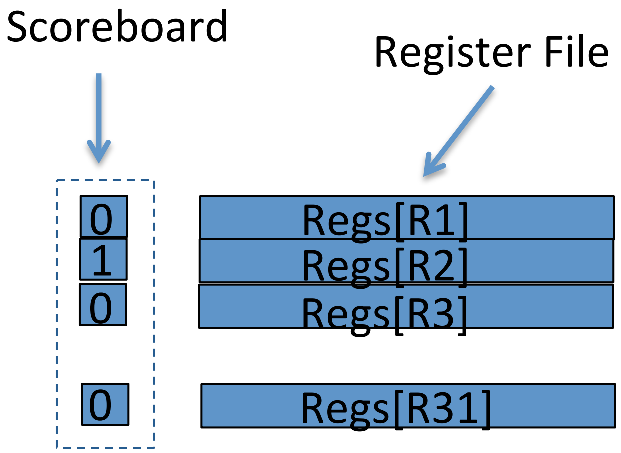




Review: In-order Scoreboard

- 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

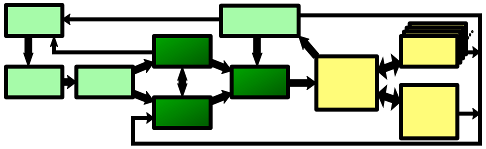
    i.e., some outstanding instruction is going to change it
- Issue in-order: RD ← Fn (RS, RT)
  - If SB[RS] or SB[RT] is set → RAW, stall
  - If SB[RD] is set → WAW, stall
  - Else, dispatch to FU (Fn) and set SB[RD]
- Complete out-of-order
  - Update GPR[RD], clear SB[RD]



[Gabriel Loh]

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

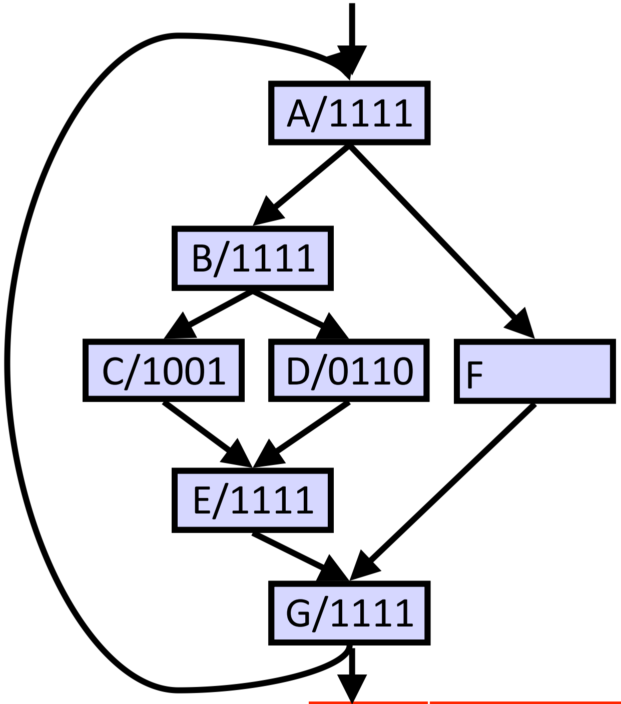
		i0	i1	i2	i3
Warp 0					
	add r8, r6, r7	0	0	0	0
Warp 1					

⋮

SIMT Using a Hardware Stack

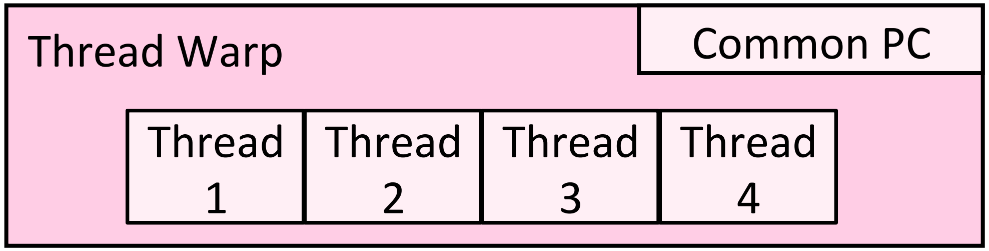
Stack approach invented at Lucafilm, Ltd in early 1980's

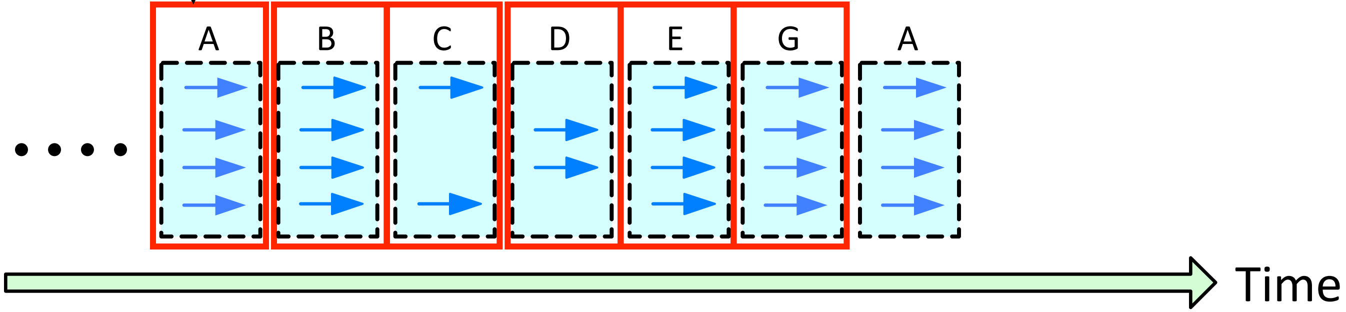
Version here from [Fung et al., MICRO 2007]



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

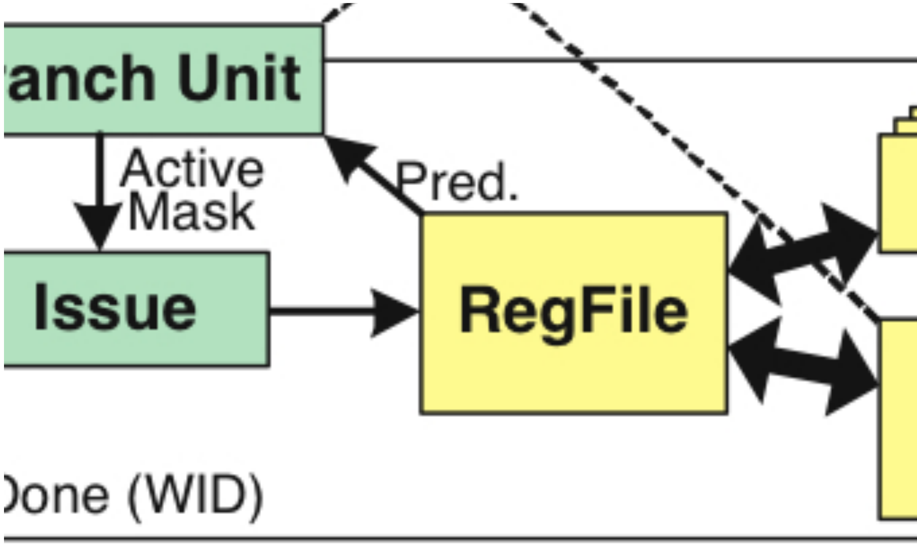
- Execution mask stack implemented with special instructions to push/pop. Descriptions can be found in AMD ISA manual and NVIDIA patents.
- In practice augment stack with predication (lower overhead).

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

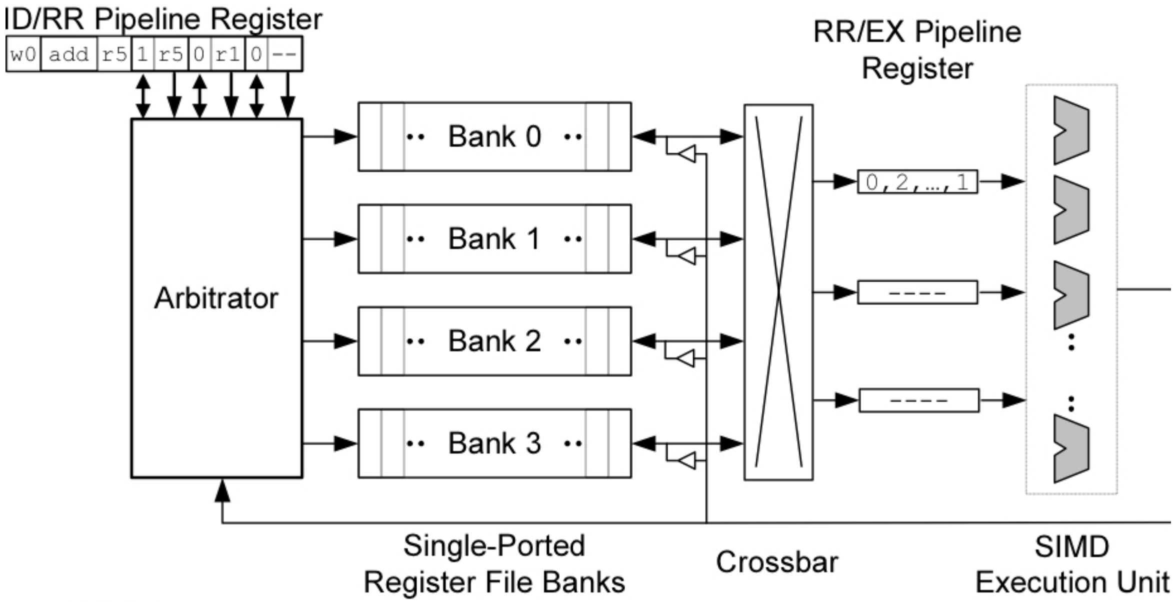
Register File

- 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

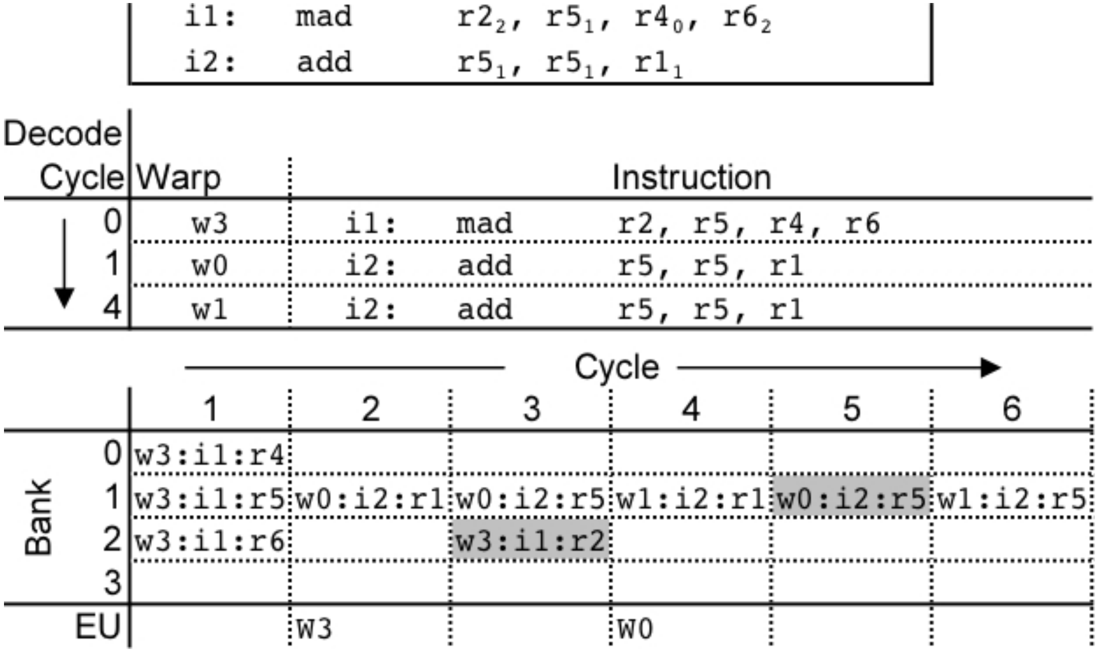
Strawman microarchitecture:



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

Register Bank Conflicts



i1:	mad	$r2_2$, $r5_1$, $r4_0$, $r6_2$
i2:	add	$r5_1$, $r5_1$, $r1_1$

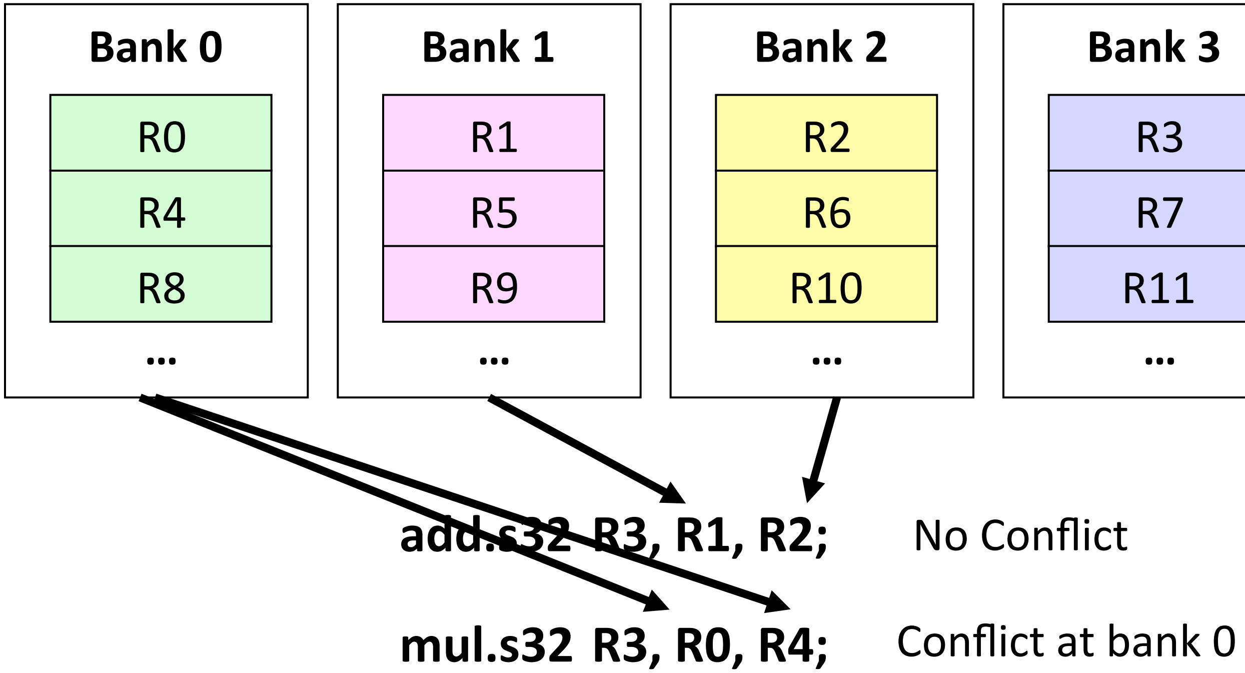
Decode Cycle	Warp	Instruction		
0	w3	i1:	mad	r2, r5, r4, r6
1	w0	i2:	add	r5, r5, r1
4	w1	i2:	add	r5, r5, r1

Cycle →

Bank	1	2	3	4	5	6
0	w3:i1:r4					
1	w3:i1:r5	w0:i2:r1	w0:i2:r5	w1:i2:r1	w0:i2:r5	w1:i2:r5
2	w3:i1:r6		w3:i1:r2			
3						
EU		W3		W0		

- 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

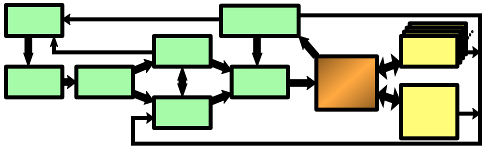


Bank 0	Bank 1	Bank 2	Bank 3
R0	R1	R2	R3
R4	R5	R6	R7
R8	R9	R10	R11
…	…	…	…

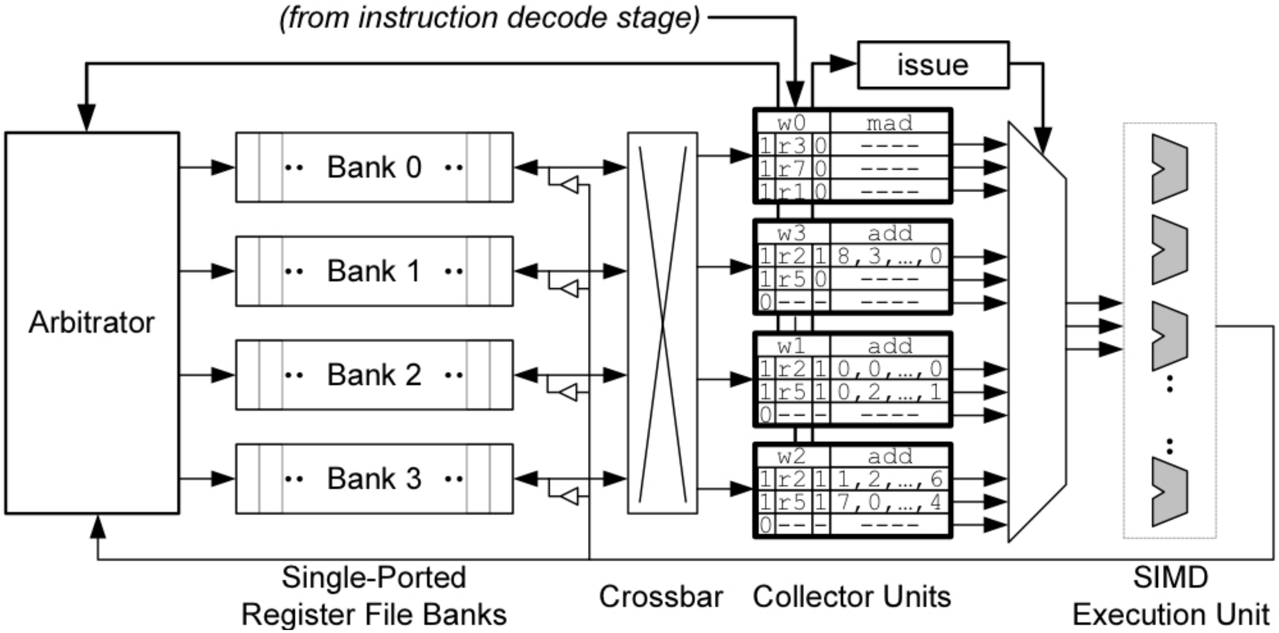
add.s32 R3, R1, R2; No Conflict

mul.s32 R3, R0, R4; Conflict at bank 0

- 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



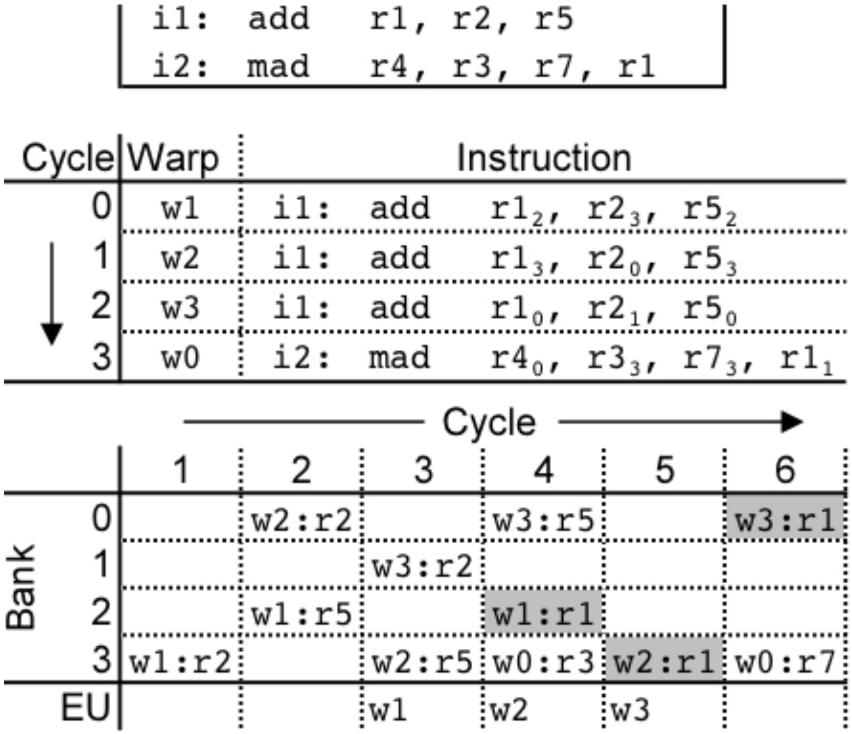
Operand Collector (1)



- 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)

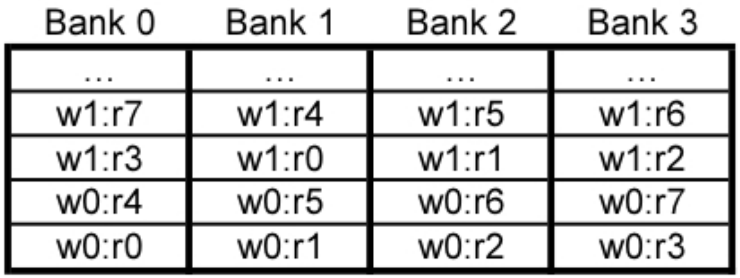
- Combining swizzling and access scheduling can give up to ~ 2x improvement in throughput



```
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 $r4_0$, $r3_3$, $r7_3$, $r1_1$

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

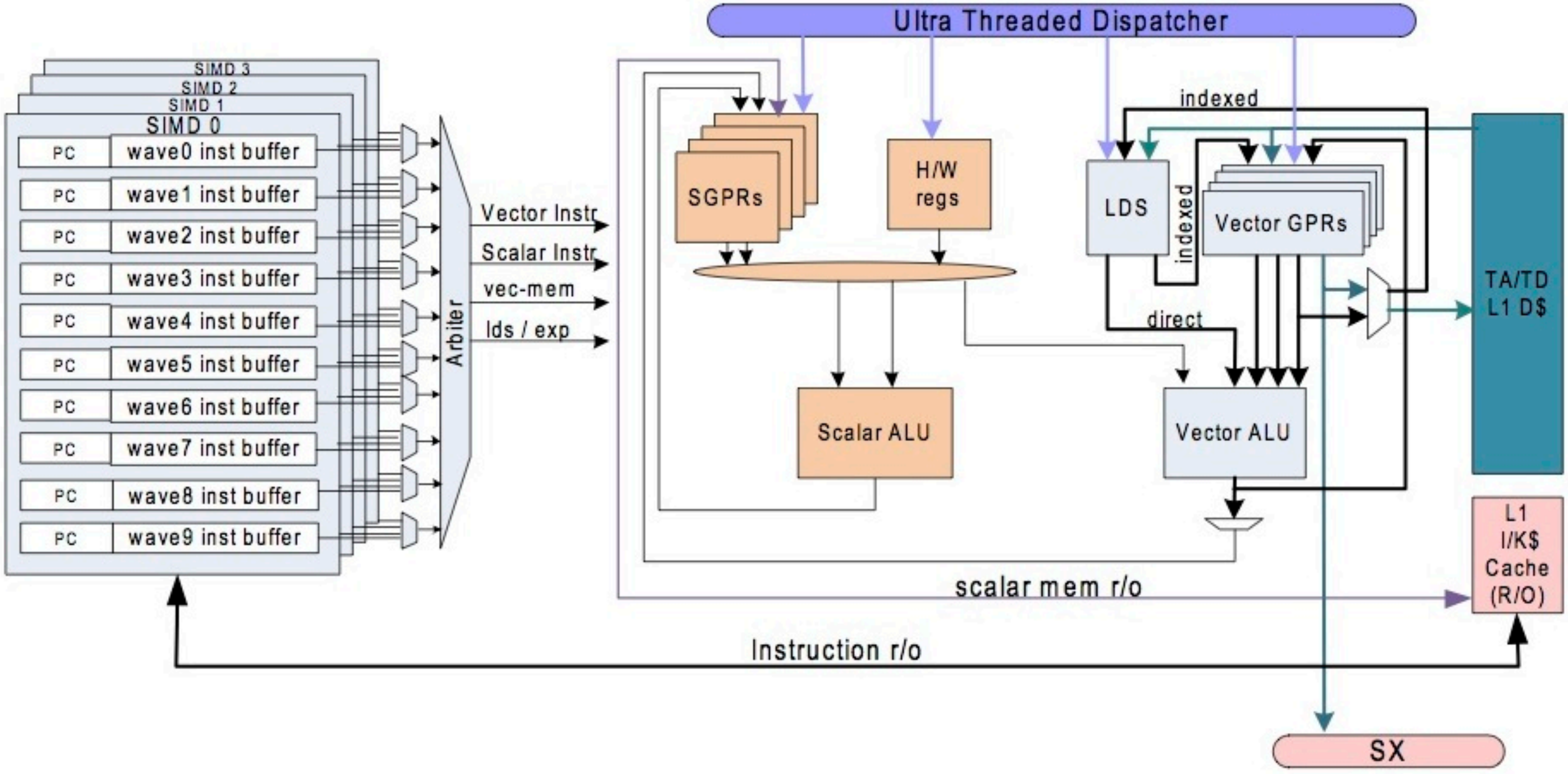
AMD Southern Islands

- 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
```

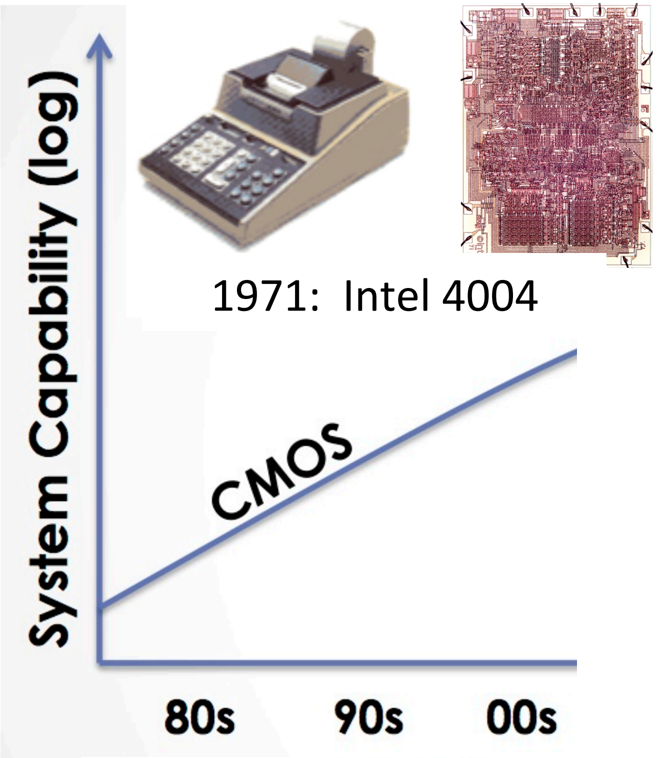
[Southern Islands Series Instruction Set Architecture, Aug. 2012]

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

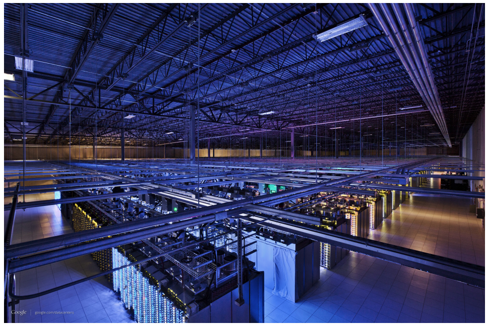
Decreasing cost per unit computation



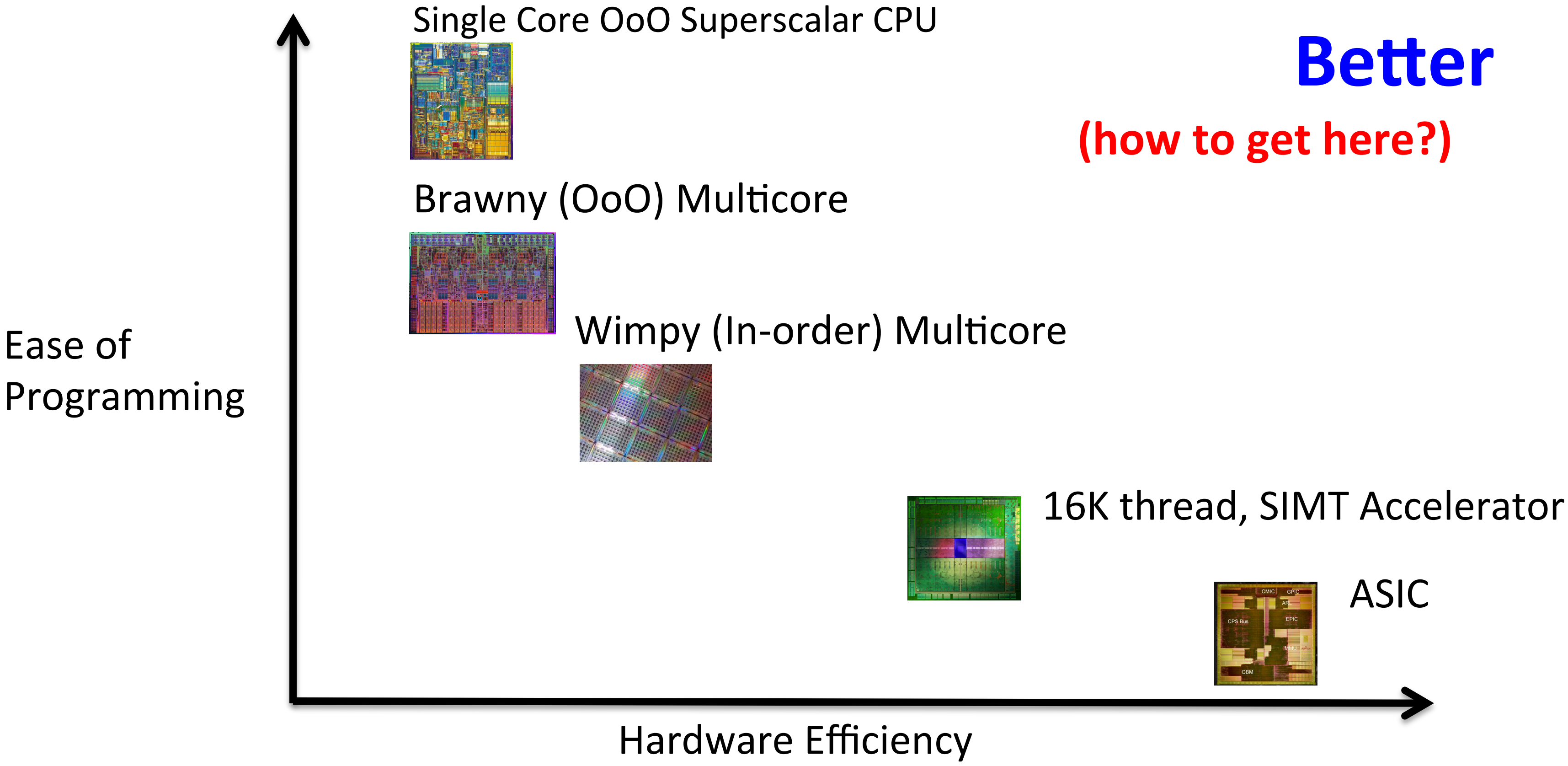
1971: Intel 4004

1981: IBM 5150

2007: iPhone



2012: Googleatacenter



Single Core OoO Superscalar CPU

Brawny (OoO) Multicore

Wimpy (In-order) Multicore

16K thread, SIMT Accelerator

ASIC

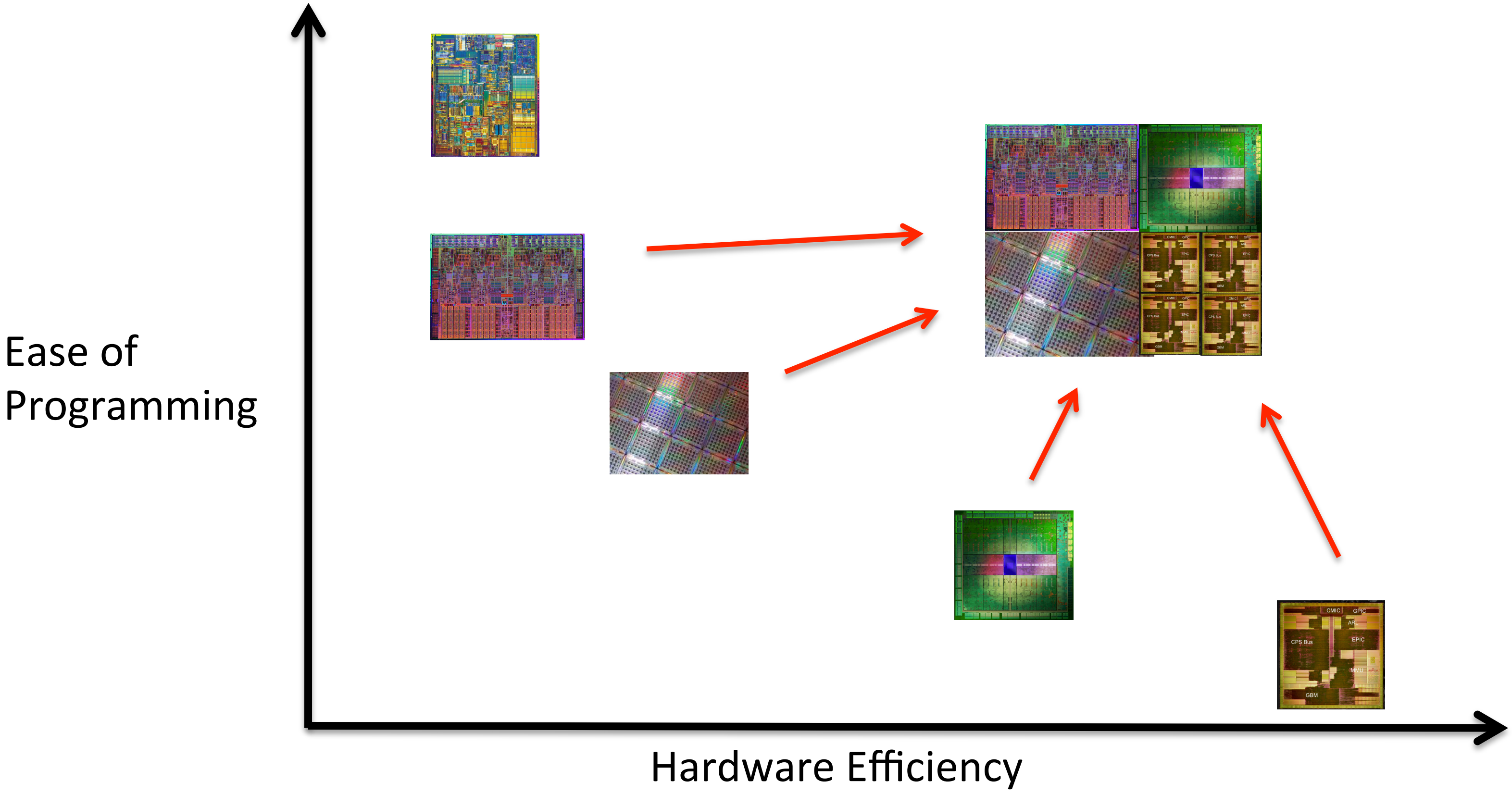
Ease of Programming

Hardware Efficiency

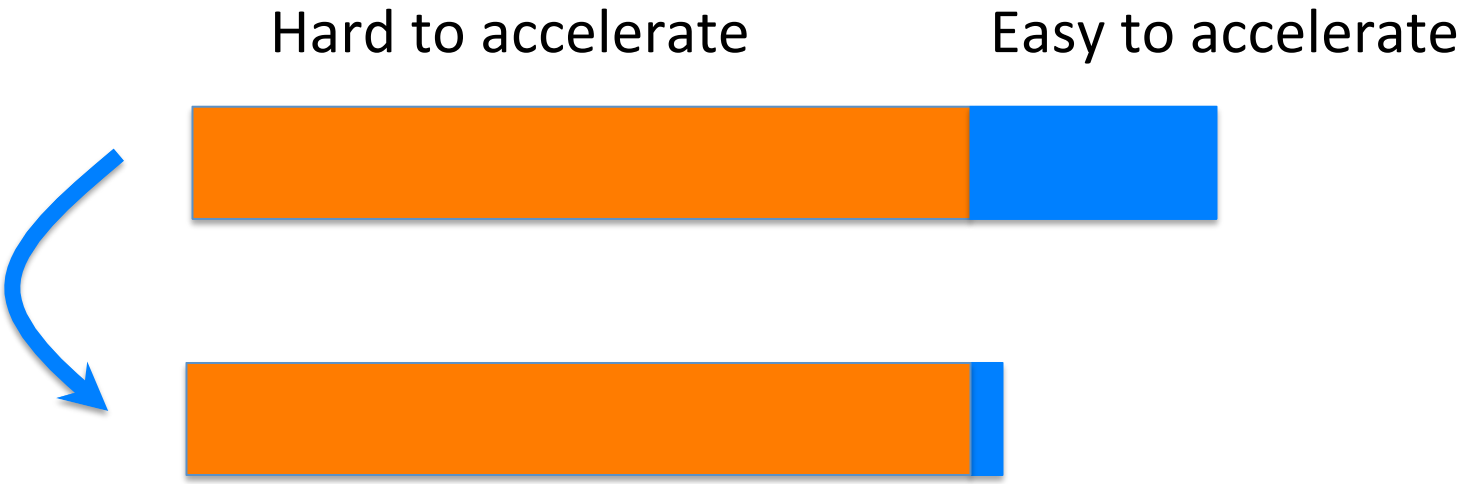
Better

(how to get here?)

Start by using right tool for each job…

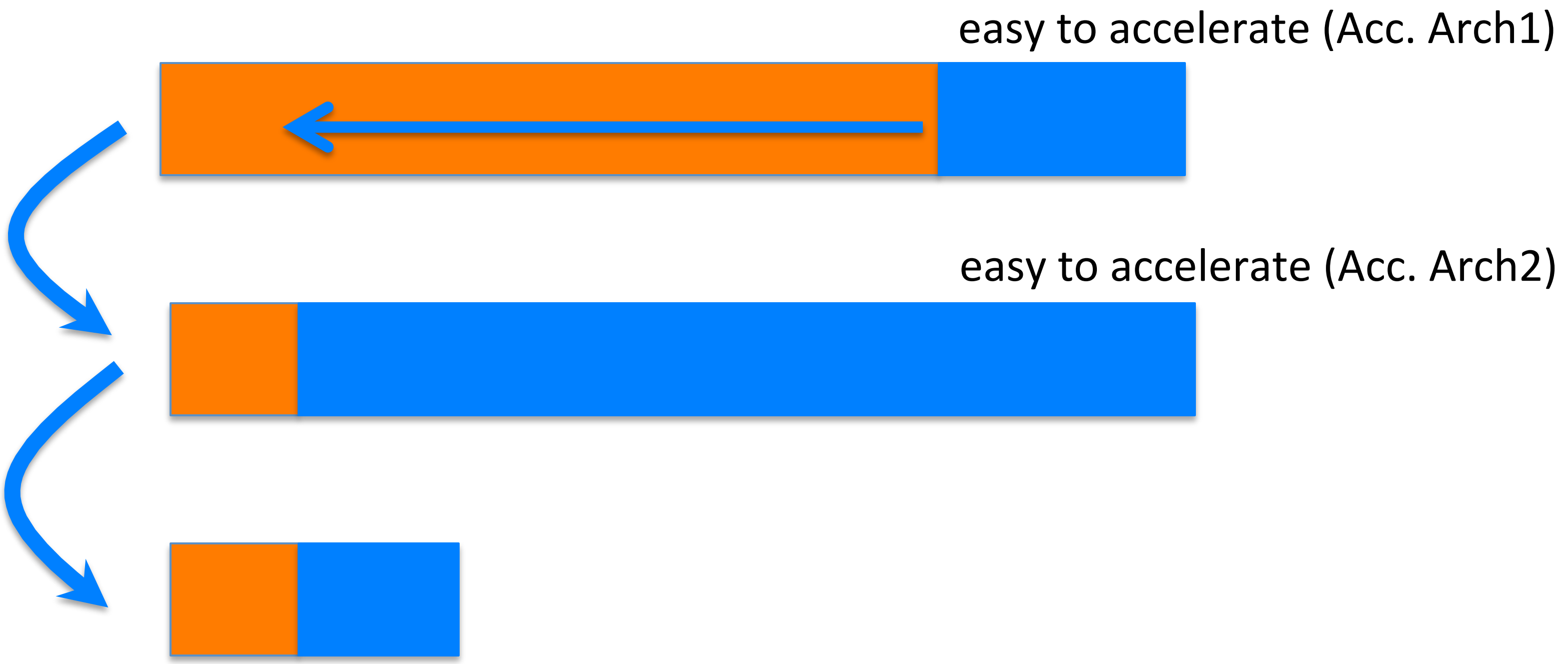


Amdahl's Law Limits this Approach



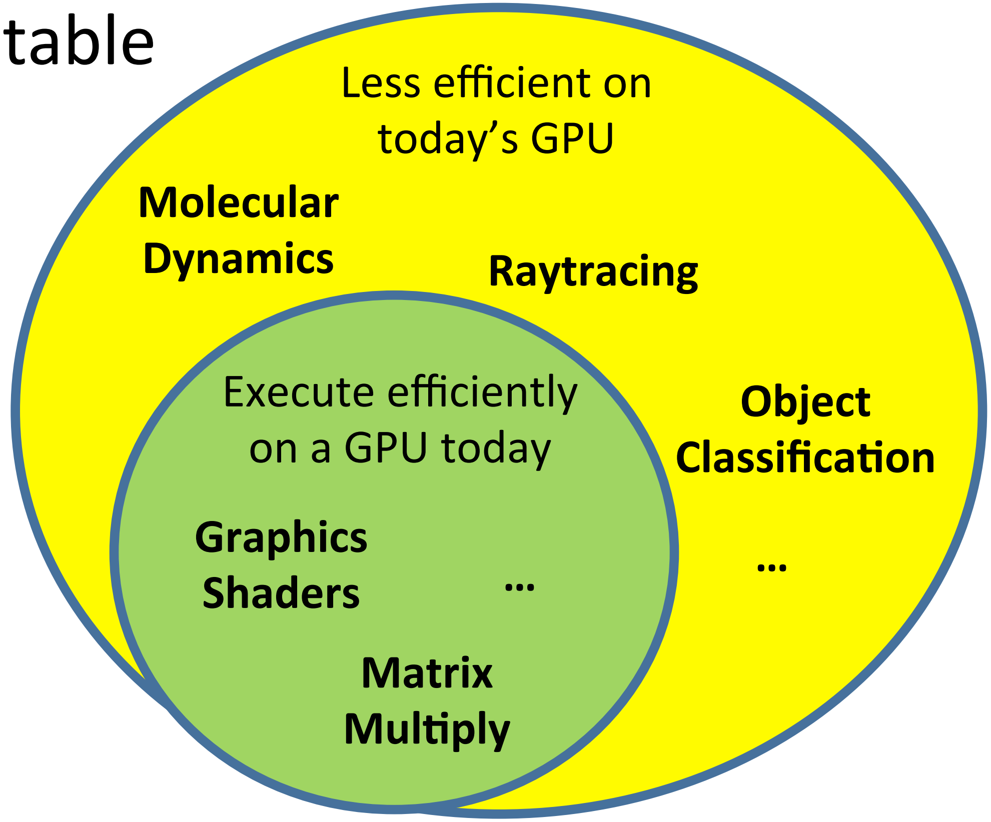
$$\text{Improvement}_{\text{overall}} = \frac{1}{\text{Fraction}_{\text{hard}} + \dfrac{1 - \text{Fraction}_{\text{hard}}}{\text{Improvement}_{\text{easy}}}}$$

Question: Can dividing line be moved?


easy to accelerate (Acc. Arch1)
easy to accelerate (Acc. Arch2)

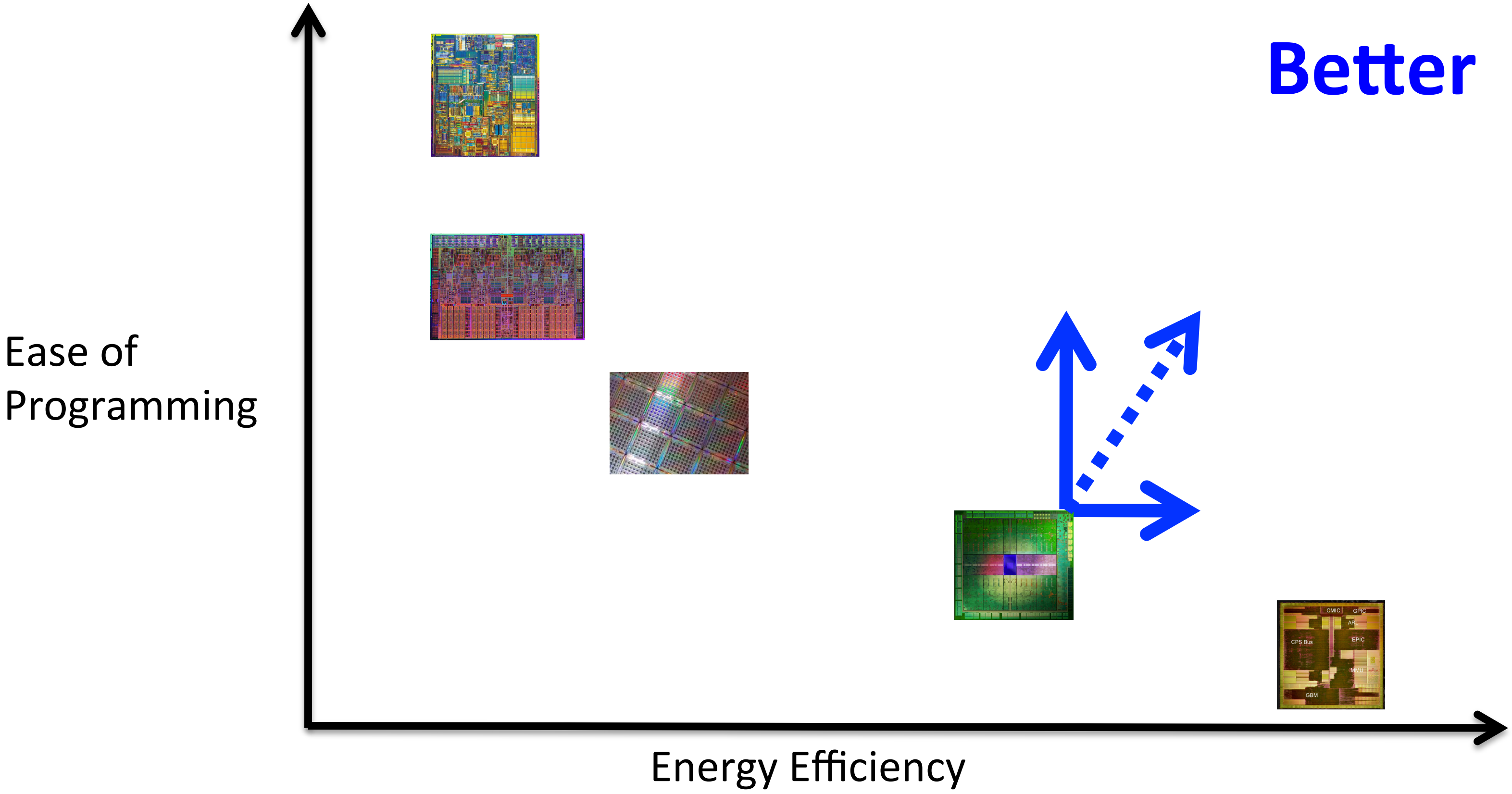
Forward-Looking GPU Software

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



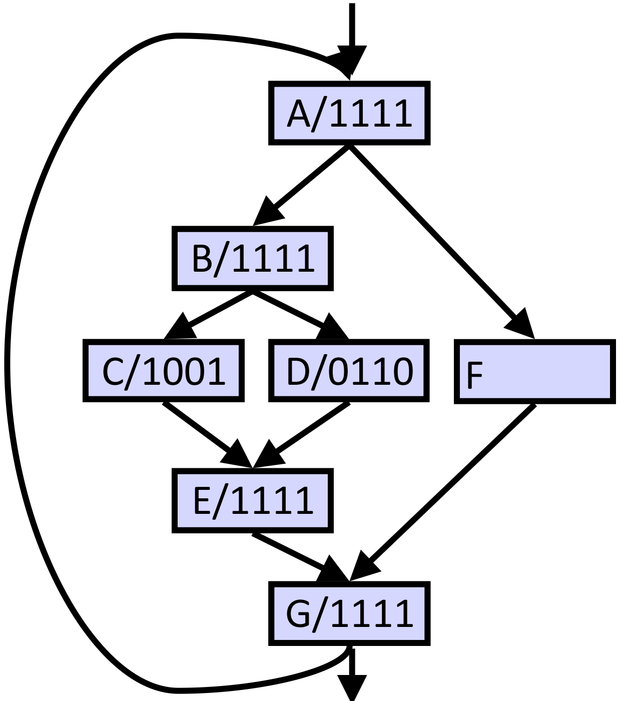
[Tim Rogers]

Two Routes to “Better”



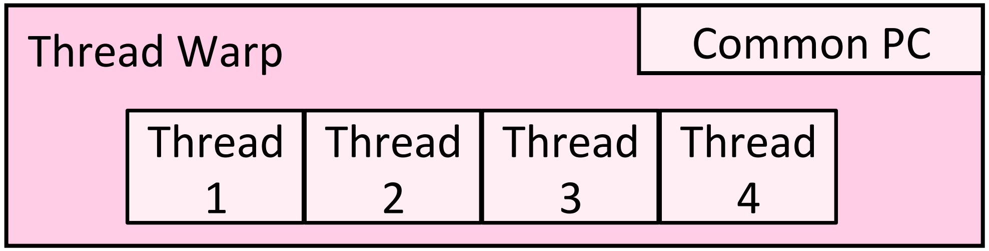
Research Direction 1: Mitigating SIMT Control Divergence

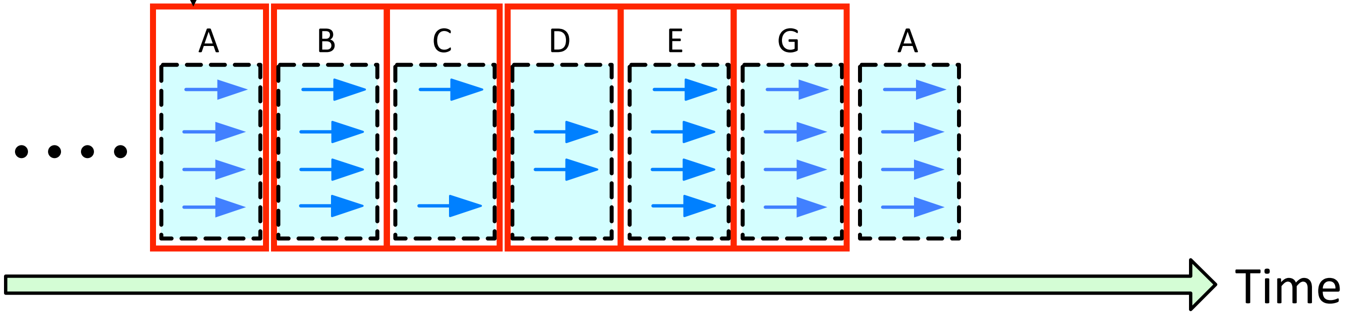
Recall: SIMT Hardware Stack



Stack

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

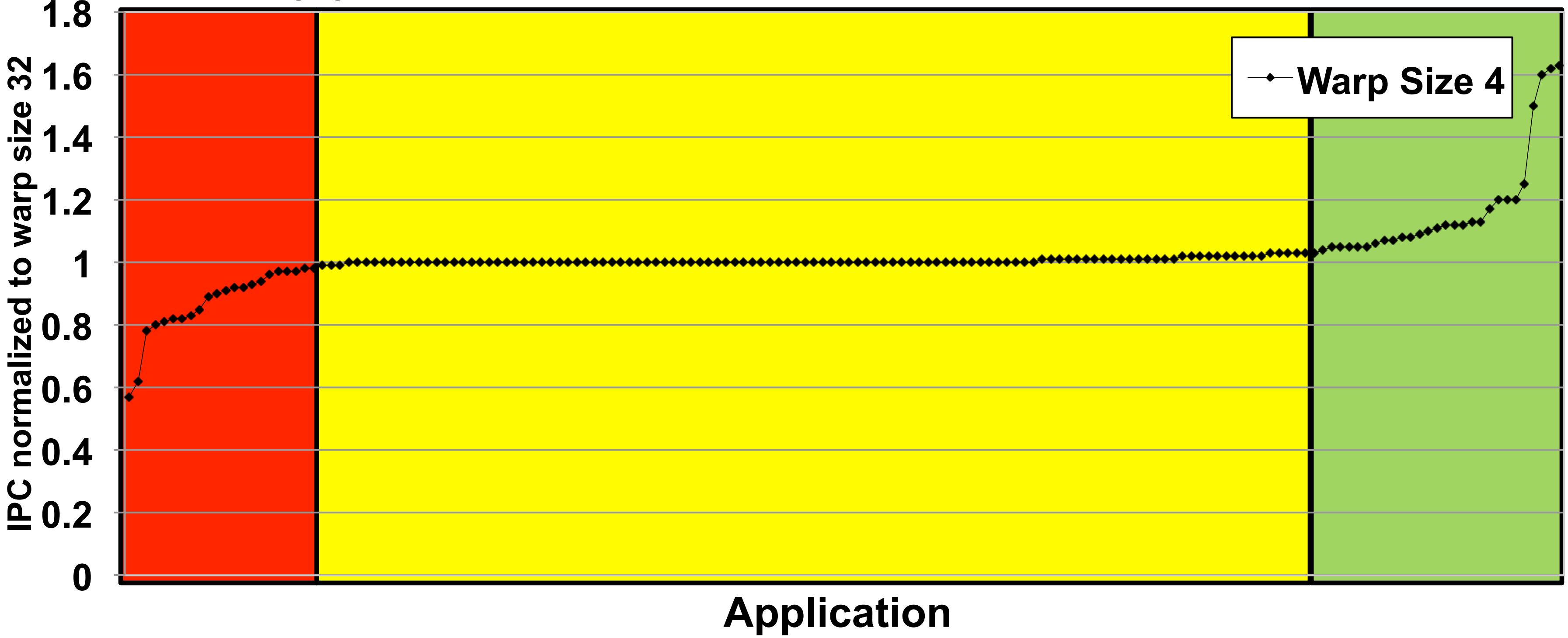




Potential for significant loss of throughput when control flow diverged!

Performance vs. Warp Size

- 165 Applications



Convergent Applications

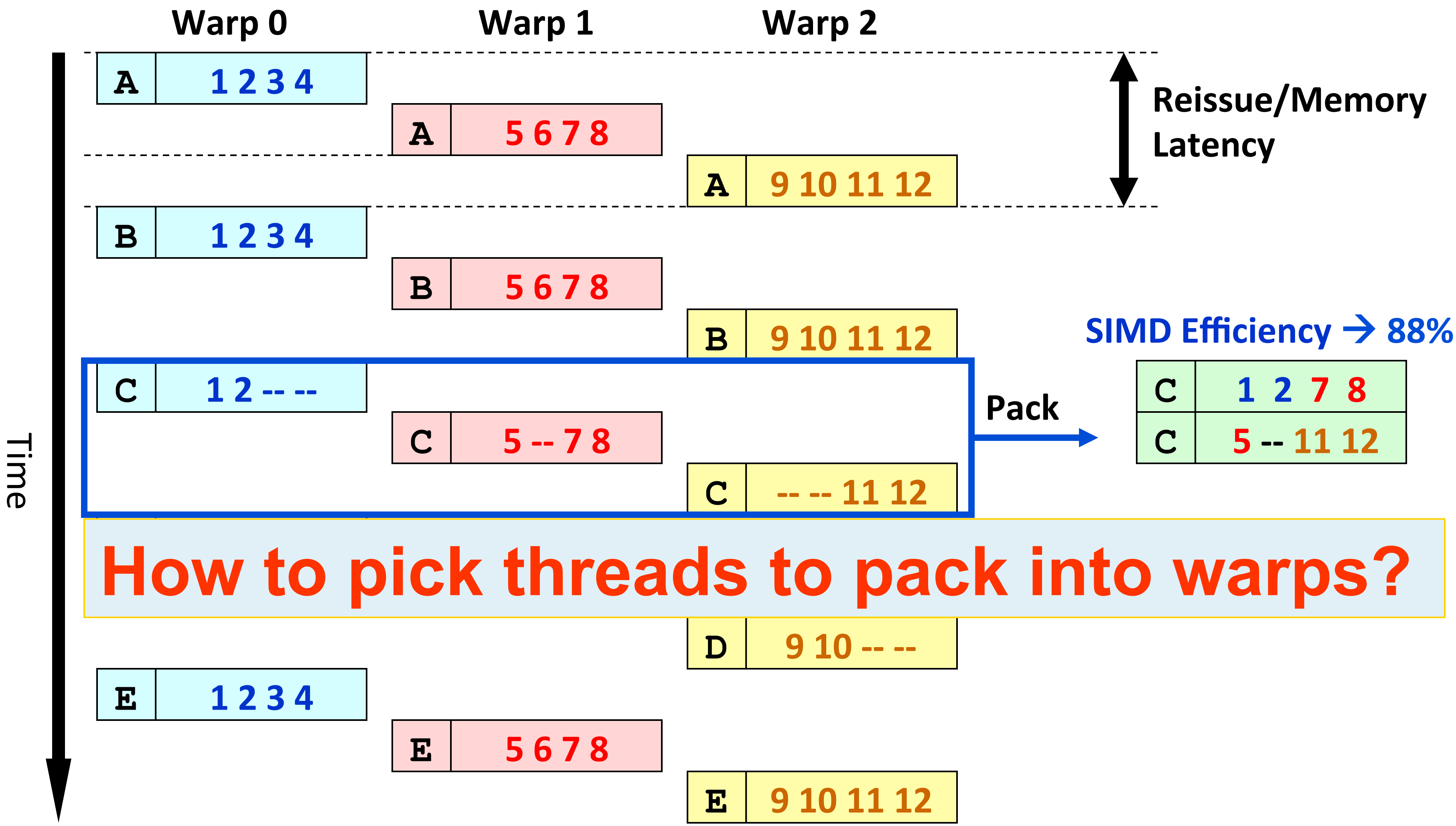
Warp-Size Insensitive Applications

Divergent Applications

Rogers et al., A Variable Warp-Size Architecture, ISCA 2015

Dynamic Warp Formation

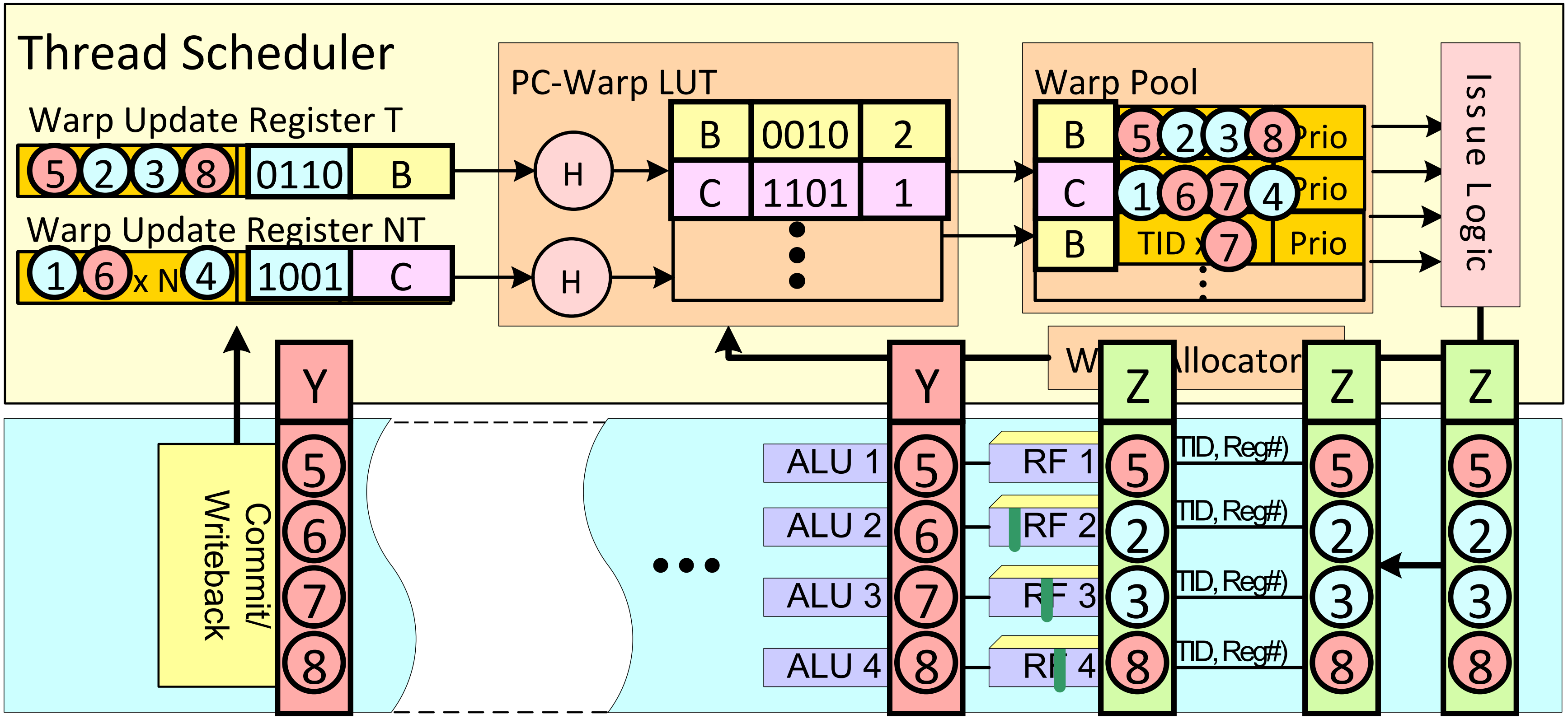
(Fung MICRO’07)



How to pick threads to pack into warps?

Dynamic Warp Formation: Hardware Implementation

A: BEQ R2, B
C: …

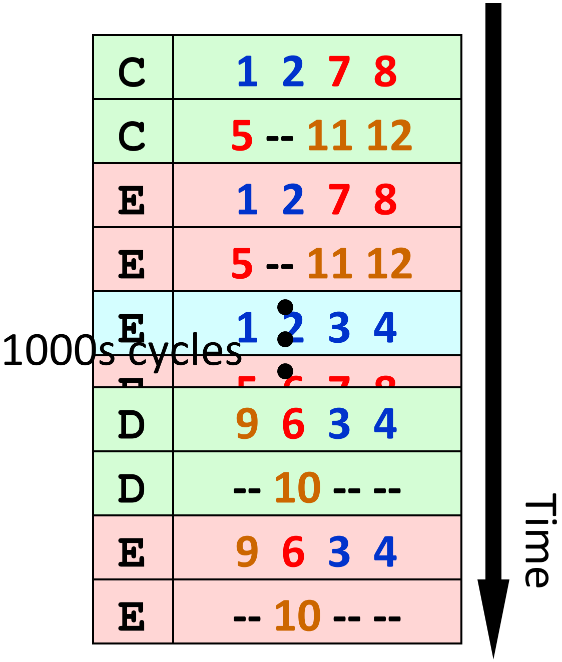


No Lane Conflict

DWF Pathologies: Starvation

```
B: if (K > 10)
C:     K = 10;
   else
D:     K = 0;
E: B = C[tid.x] + K;
```

- Majority Scheduling
  - Best Performing
  - Prioritize largest group of threads with same PC
- ***Starvation***
  - <u>LOWER</u> SIMD Efficiency!
- Other Warp Scheduler?
  - Tricky: Variable Memory Latency

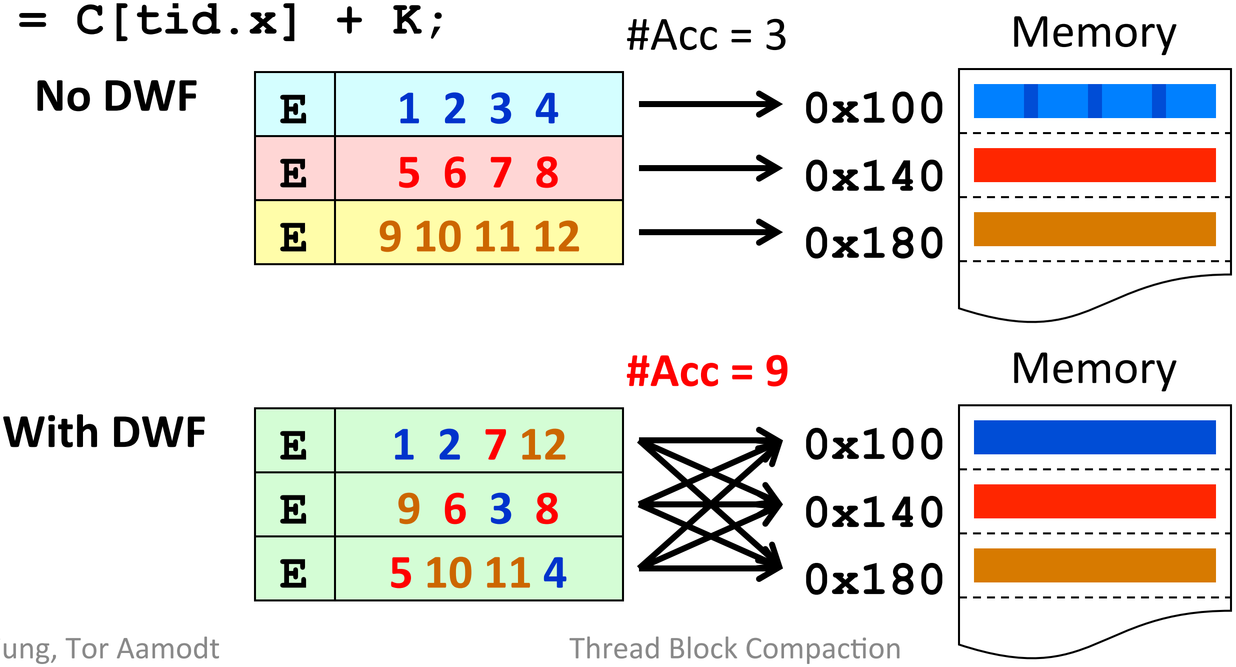


C	1 2 7 8
C	5 -- 11 12
E	1 2 7 8
E	5 -- 11 12
E	1 2 3 4
E	5 6 7 8
D	9 6 3 4
D	-- 10 -- --
E	9 6 3 4
E	-- 10 -- --

DWF Pathologies:
Extra Uncoalesced Accesses

- Coalesced Memory Access = Memory SIMD
  - 1st Order CUDA Programmer Optimization
- Not preserved by DWF

```
E: B = C[tid.x] + K;
```



#Acc = 3

No DWF

E	1 2 3 4
E	5 6 7 8
E	9 10 11 12

#Acc = 9

With DWF

E	1 2 7 12
E	9 6 3 8
E	5 10 11 4

L1 Cache Absorbs Redundant Memory Traffic

L1$ Port Conflict

DWF Pathologies: Implicit Warp Sync.

- Some CUDA applications depend on the lockstep execution of “static warps”

Warp 0	Thread 0 ... 31
Warp 1	Thread 32 ... 63
Warp 2	Thread 64 ... 95

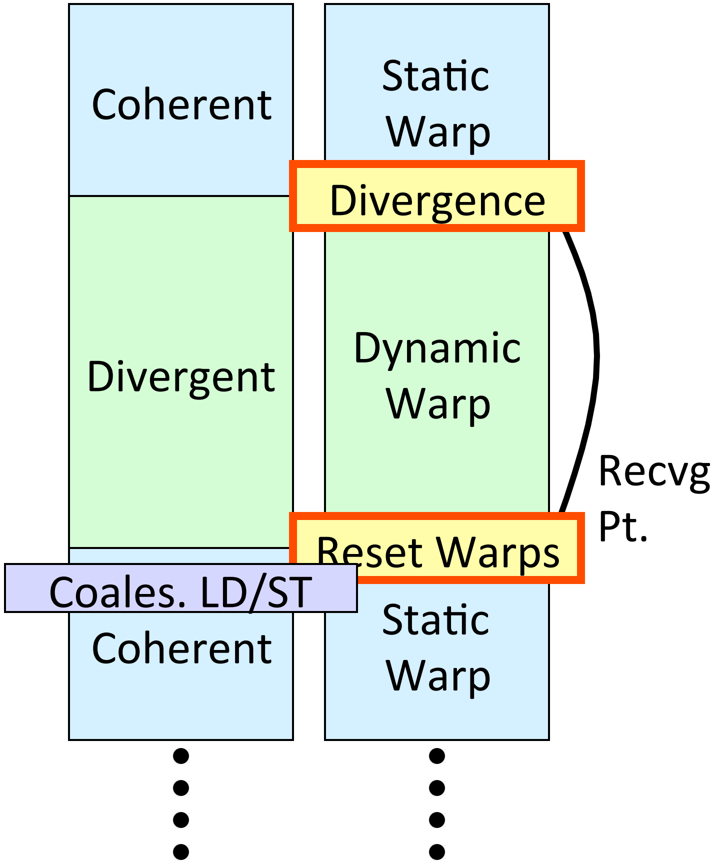
  - E.g. Task Queue in Ray Tracing

```
int wid = tid.x / 32;
if (tid.x % 32 == 0) {
  sharedTaskID[wid] = atomicAdd(g_TaskID, 32);
}
my_TaskID = sharedTaskID[wid] + tid.x % 32;
ProcessTask(my_TaskID);
```

Implicit Warp Sync.

Observation

- Compute kernels usually contain divergent and non-divergent (coherent) code segments
- Coalesced memory access usually in coherent code segments
  - DWF no benefit there



Thread Block Compaction

- Run a thread block like a warp
  - Whole block move between coherent/divergent code
  - Block-wide stack to track exec. paths reconvg.
- Barrier @ Branch/reconverge pt.
  - All avail. threads arrive at branch
  - Insensitive to warp scheduling
- Warp compaction
  - Regrouping with all avail. threads
  - If no divergence, gives static warp arrangement

✓ **Implicit Warp Sync.**

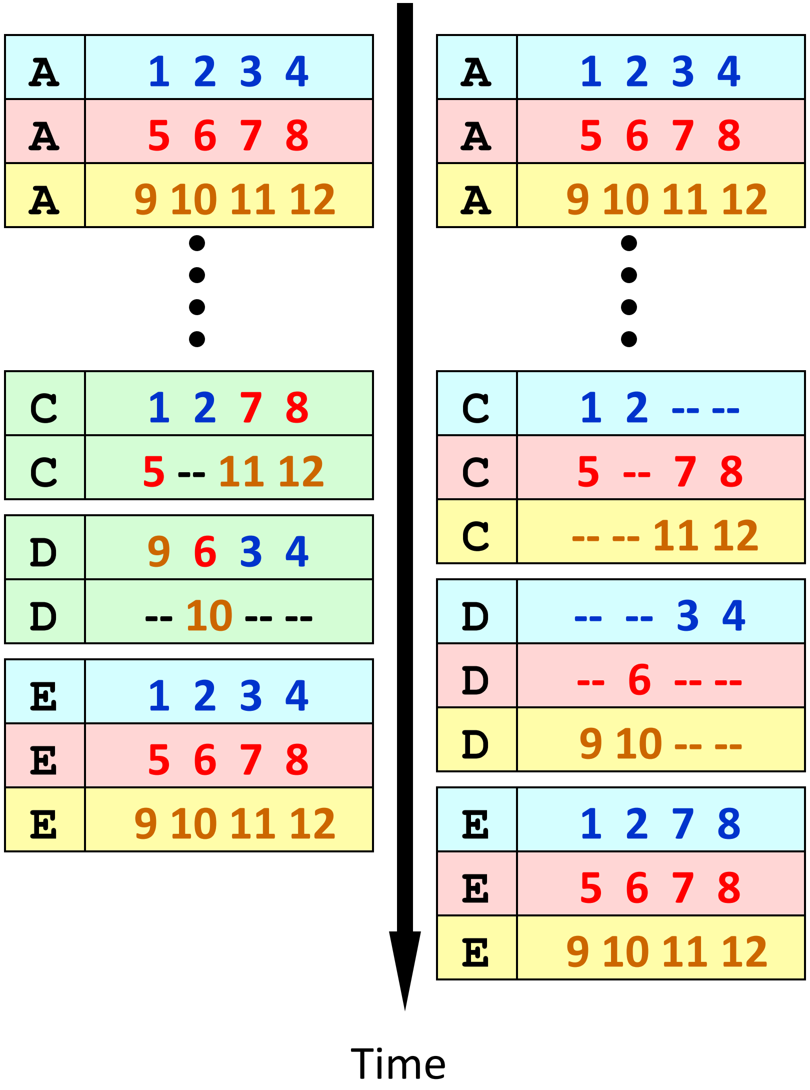
~~Starvation~~

~~Extra Uncoalesced Memory Access~~

Thread Block Compaction

PC	RPC	Active Threads											
E	-	1	2	3	4	5	6	7	8	9	10	11	12
D	E	--	--	--	--	--	--	--	--	--	--	--	--
C	E	--	--	--	--	--	--	--	--	--	--	--	--

```
A: K = A[tid.x];
B: if (K > 10)
C:     K = 10;
   else
D:     K = 0;
E: B = C[tid.x] + K;
```

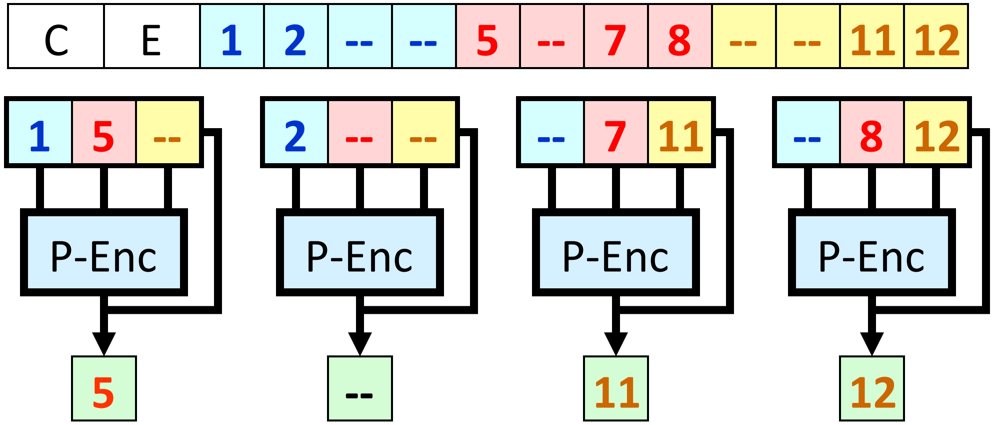


A	1 2 3 4
A	5 6 7 8
A	9 10 11 12
⋮	
C	1 2 7 8
C	5 -- 11 12
D	9 6 3 4
D	-- 10 -- --
E	1 2 3 4
E	5 6 7 8
E	9 10 11 12

A	1 2 3 4
A	5 6 7 8
A	9 10 11 12
⋮	
C	1 2 -- --
C	5 -- 7 8
C	-- -- 11 12
D	-- -- 3 4
D	-- 6 -- --
D	9 10 -- --
E	1 2 7 8
E	5 6 7 8
E	9 10 11 12

Thread Compactor

- Convert *activemask* from block-wide stack to *thread IDs* in warp buffer
- Array of Priority-Encoder

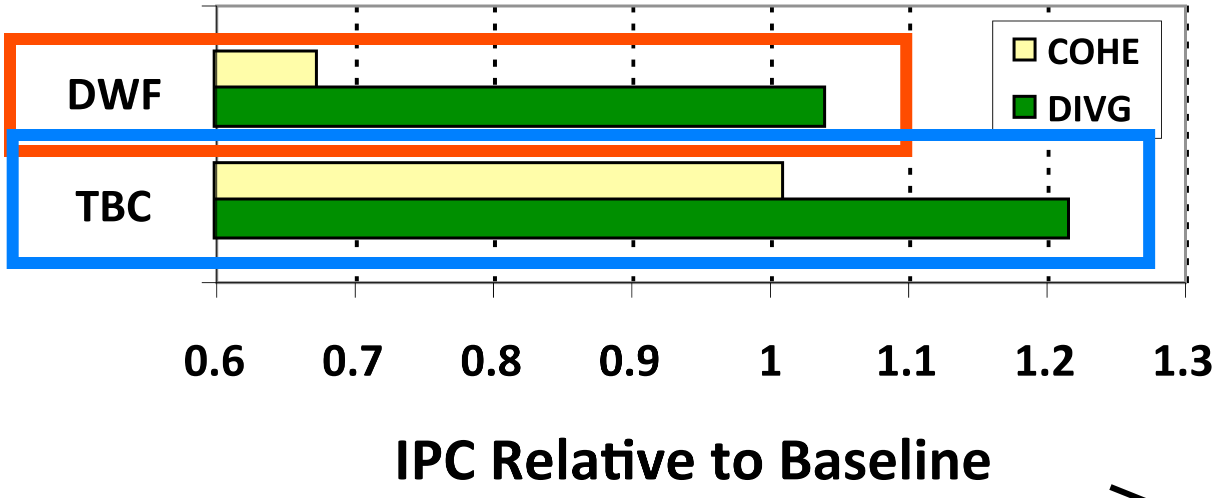


Warp Buffer

C	1 2 7 8
C	5 -- 11 12

Experimental Results

- 2 Benchmark Groups:
  - COHE = Non-Divergent CUDA applications
  - DIVG = Divergent CUDA applications



Serious Slowdown from pathologies

No Penalty for COHE

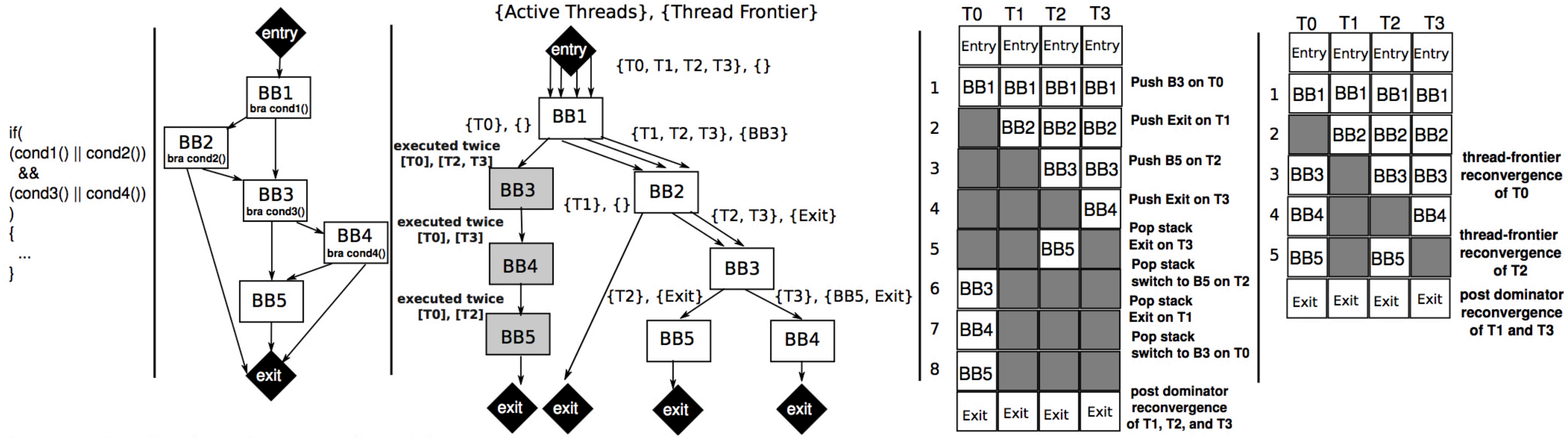
22% Speedup on DIVG

Per-Warp Stack

Recent work on warp divergence

- Intel [MICRO 2011]: Thread Frontiers – early reconvergence for unstructured control flow.
- UT-Austin/NVIDIA [MICRO 2011]: Large Warps – similar to TBC except decouple size of thread stack from thread block size.
- NVIDIA [ISCA 2012]: Simultaneous branch and warp interweaving. Enable SIMD to execute two paths at once.
- Intel [ISCA 2013]: Intra-warp compaction – extends Xeon Phi uarch to enable compaction.
- NVIDIA: Temporal SIMT [described briefly in IEEE Micro article and in more detail in CGO 2013 paper]
- NVIDIA [ISCA 2015]: Variable Warp-Size Architecture – merge small warps (4 threads) into “gangs”.

Thread Frontiers [Diamos et al., MICRO 2011]



```
if(
(cond1() || cond2())
  &&
(cond3() || cond4())
)
{
  ...
}
```

	T0	T1	T2	T3	
	Entry	Entry	Entry	Entry	
1	BB1	BB1	BB1	BB1	Push B3 on T0
2	▨	BB2	BB2	BB2	Push Exit on T1
3	▨	▨	BB3	BB3	Push B5 on T2
4	▨	▨	▨	BB4	Push Exit on T3
5	▨	▨	BB5	▨	Pop stack Exit on T3
6	BB3	▨	▨	▨	Pop stack switch to B5 on T2
7	BB4	▨	▨	▨	Pop stack Exit on T1
8	BB5	▨	▨	▨	Pop stack switch to B3 on T0
	Exit	Exit	Exit	Exit	post dominator reconvergence of T1, T2, and T3

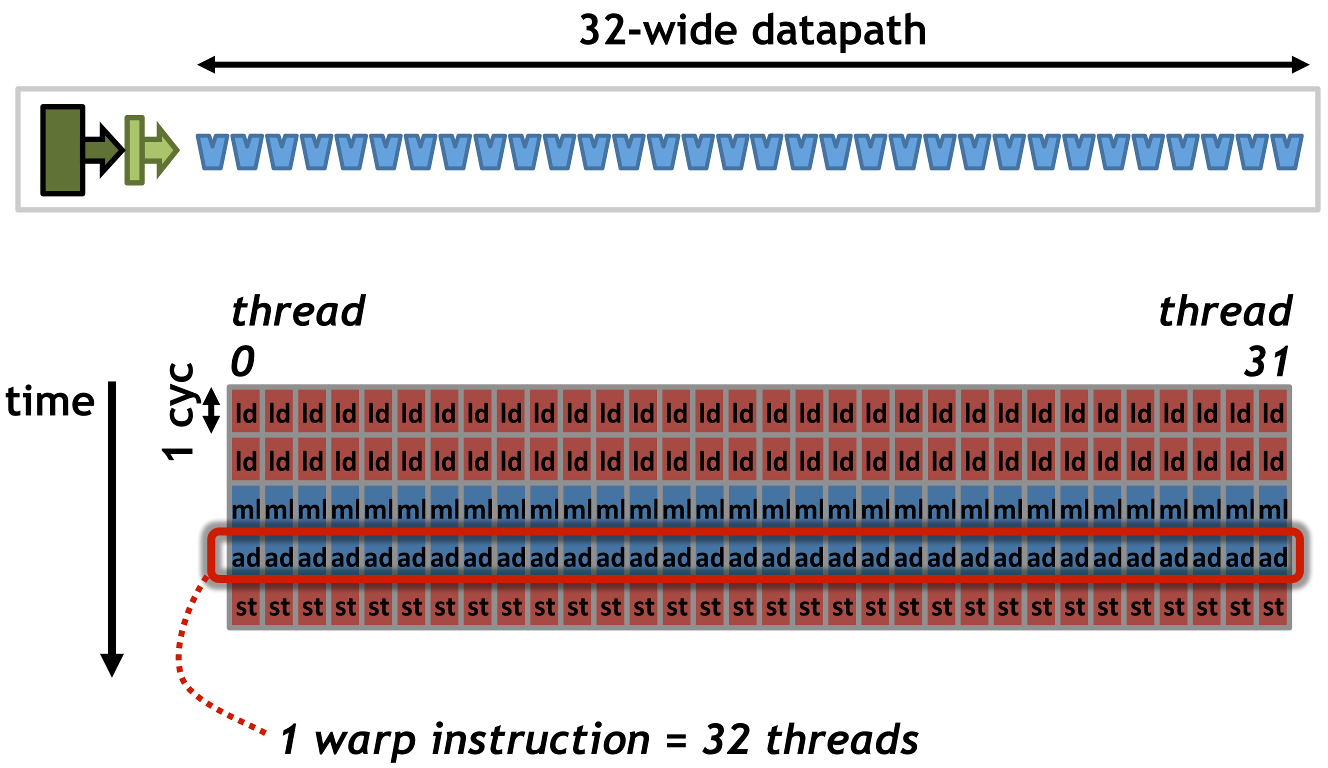
	T0	T1	T2	T3	
	Entry	Entry	Entry	Entry	
1	BB1	BB1	BB1	BB1	
2	▨	BB2	BB2	BB2	
3	BB3	▨	BB3	BB3	thread-frontier reconvergence of T0
4	BB4	▨	▨	BB4	
5	BB5	▨	BB5	▨	thread-frontier reconvergence of T2
	Exit	Exit	Exit	Exit	post dominator reconvergence of T1 and T3

a) compound conditionals b) unstructured control flow c) dynamic code expansion d) immediate post-dominator re-convergence e) re-convergence at **thread frontiers**

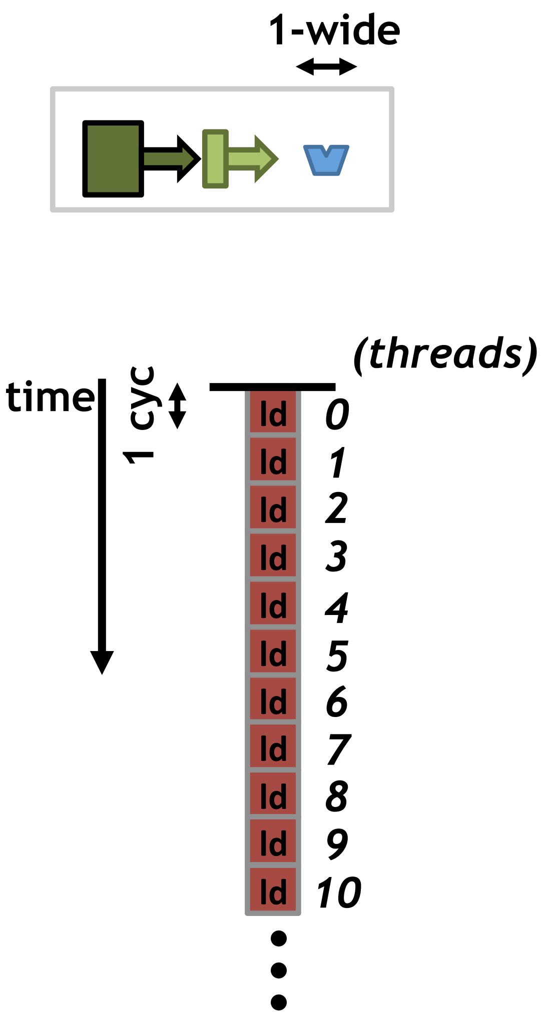
Figure 1: An example of an application with unstructured control flow leading to dynamic code expansion.

Temporal SIMT

Spatial SIMT (current GPUs)


32-wide datapath
thread 0
thread 31
time
1 cyc
ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld
ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld ld
ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml ml
ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad ad
st st st st st st st st st st st st st st st st st st st st st st st st st st st st st st st st
1 warp instruction = 32 threads

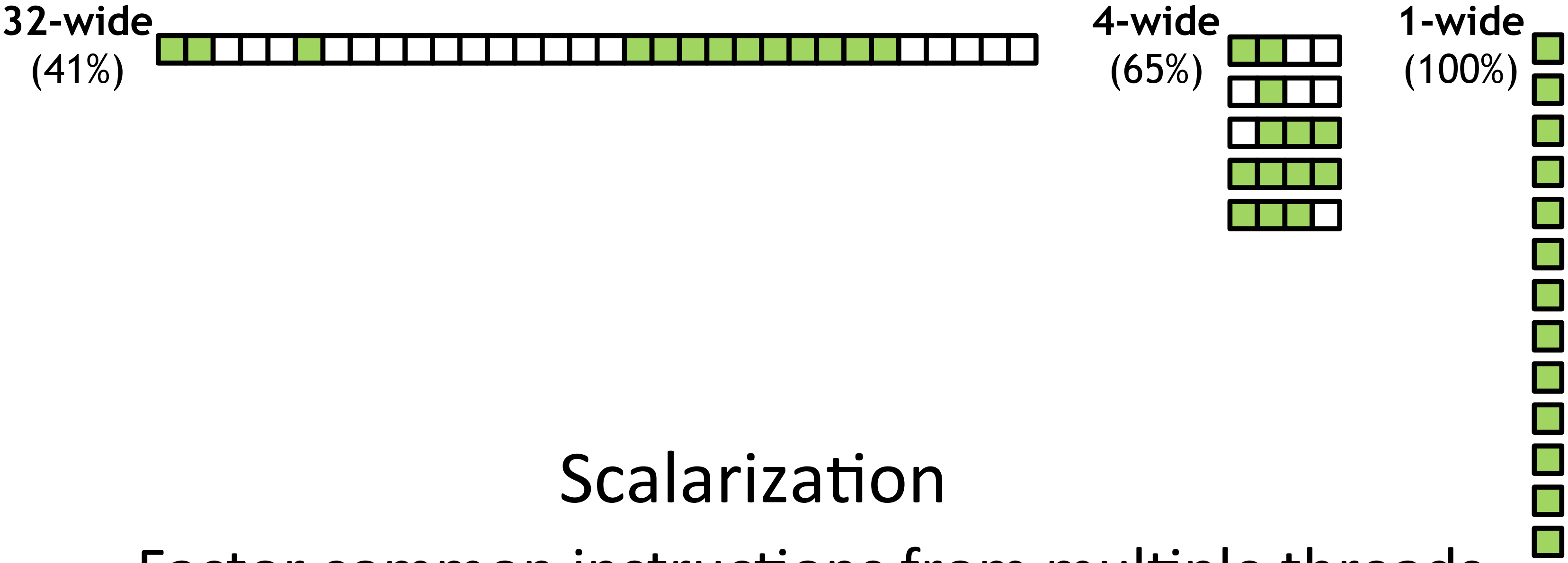
Pure Temporal SIMT



[slide courtesy of Bill Dally]

Temporal SIMT Optimizations

Control divergence — hybrid MIMD/SIMT



32-wide (41%)

4-wide (65%)

1-wide (100%)

Scalarization

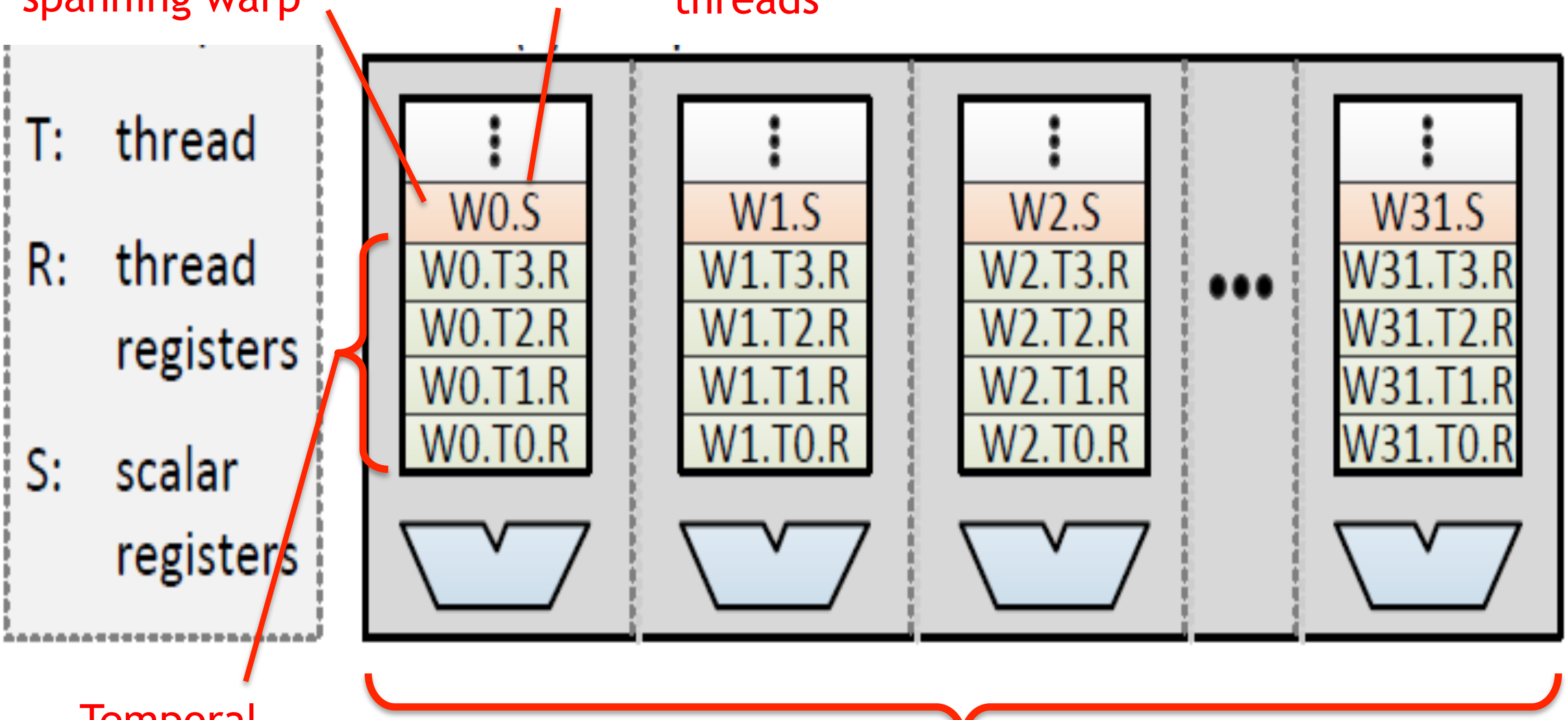
Factor common instructions from multiple threads

Execute once – place results in common registers

[See: SIMT Affine Value Structure (ISCA 2013)]

[slide courtesy of Bill Dally]

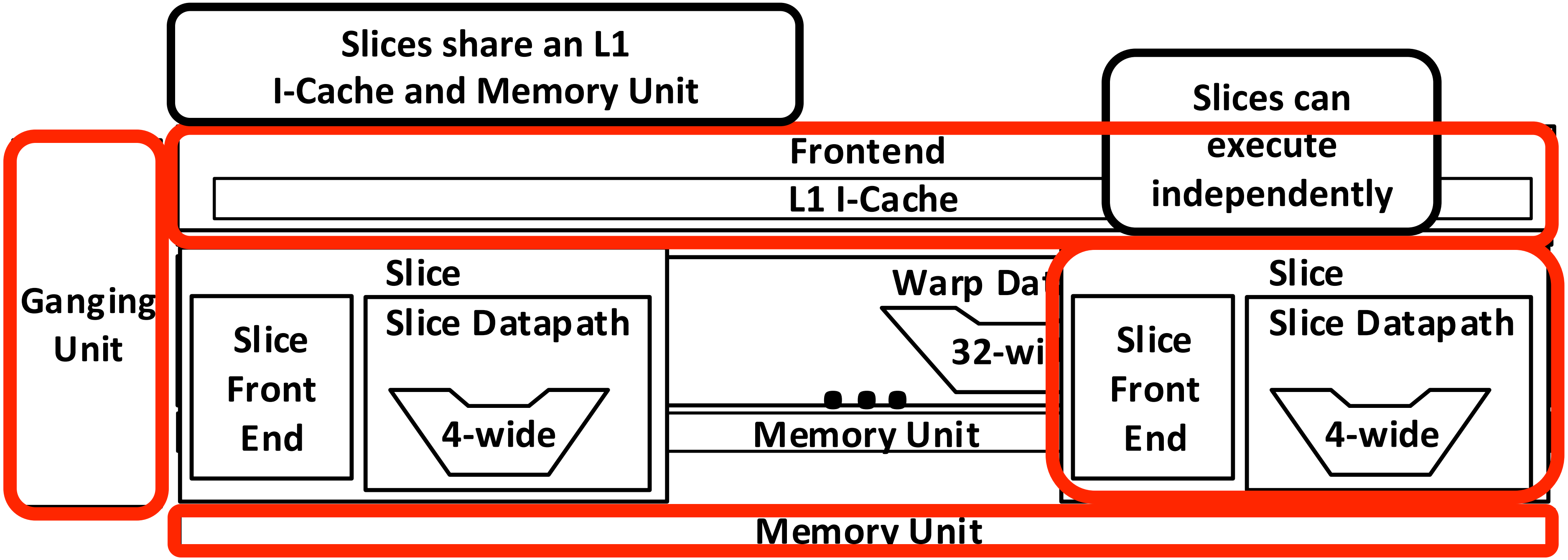
Scalar Instructions in SIMT Lanes



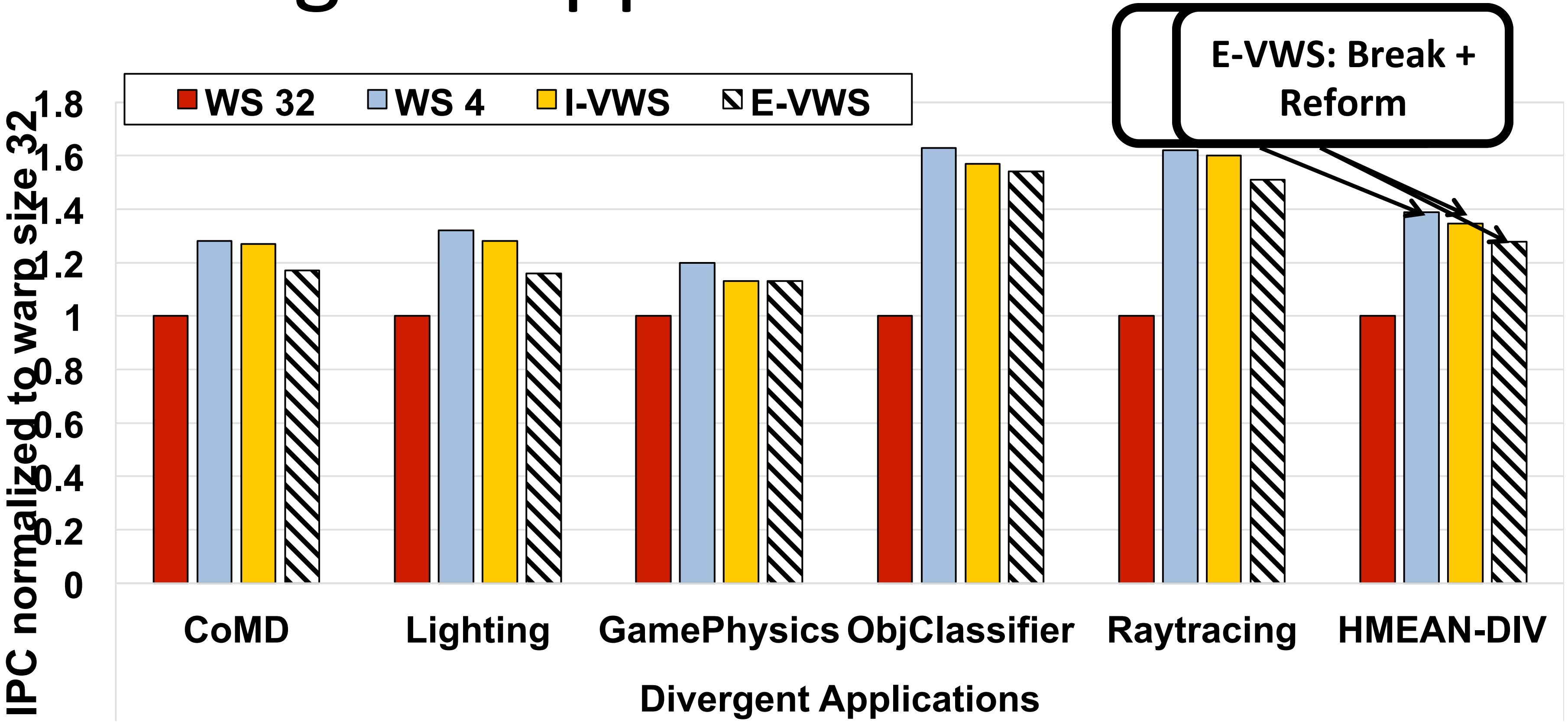
[slide courtesy of Bill Dally]

Variable Warp-Size Architecture

- Most recent work by NVIDIA [ISCA 2015]
- Split the SM datapath into narrow **slices**.
  - Extensively studied 4-thread slices
- Gang slice execution to gain efficiencies of wider warp.

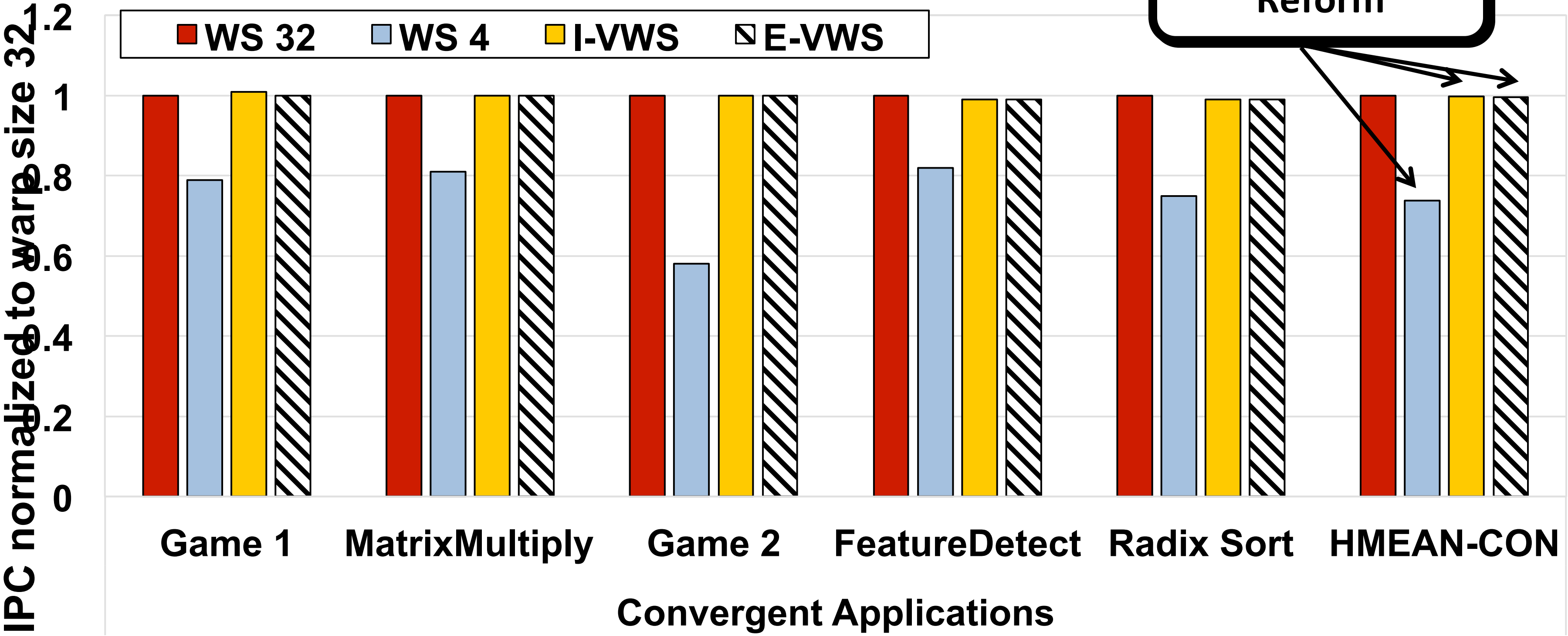


Divergent Application Performance



E-VWS: Break + Reform

Convergent Application Performance



E-VWS: Break + Reform

Warp-Size Insensitive Applications Unaffected

Research Direction 2: Mitigating High GPGPU Memory Bandwidth Demands

Reducing Off-Chip Access / Divergence

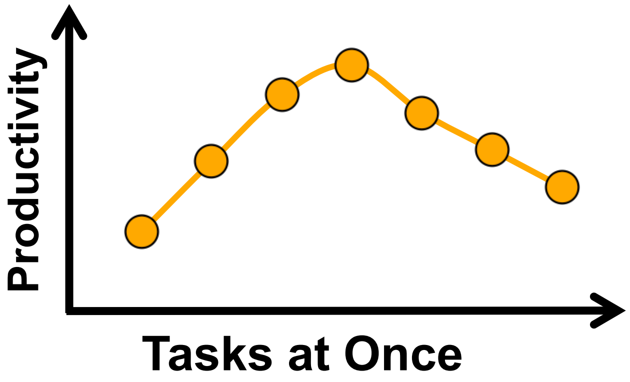
- Re-writing software to use “shared memory” and avoid uncoalesced global accesses is bane of GPU programmer existence.
- Recent GPUs introduce caches, but large number of warps/wavefronts lead to thrashing.

- 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

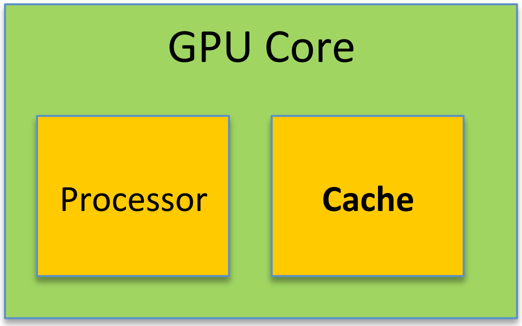
Thread Scheduling Analogy

[MICRO 2012]

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

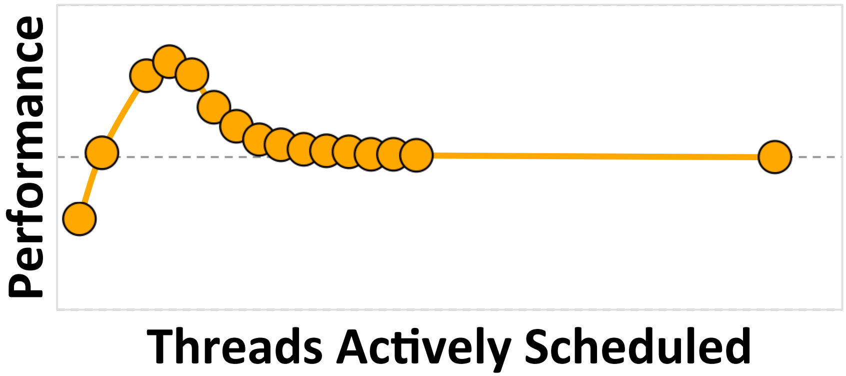


  - GPUs have limited **cache capacity**


GPU Core

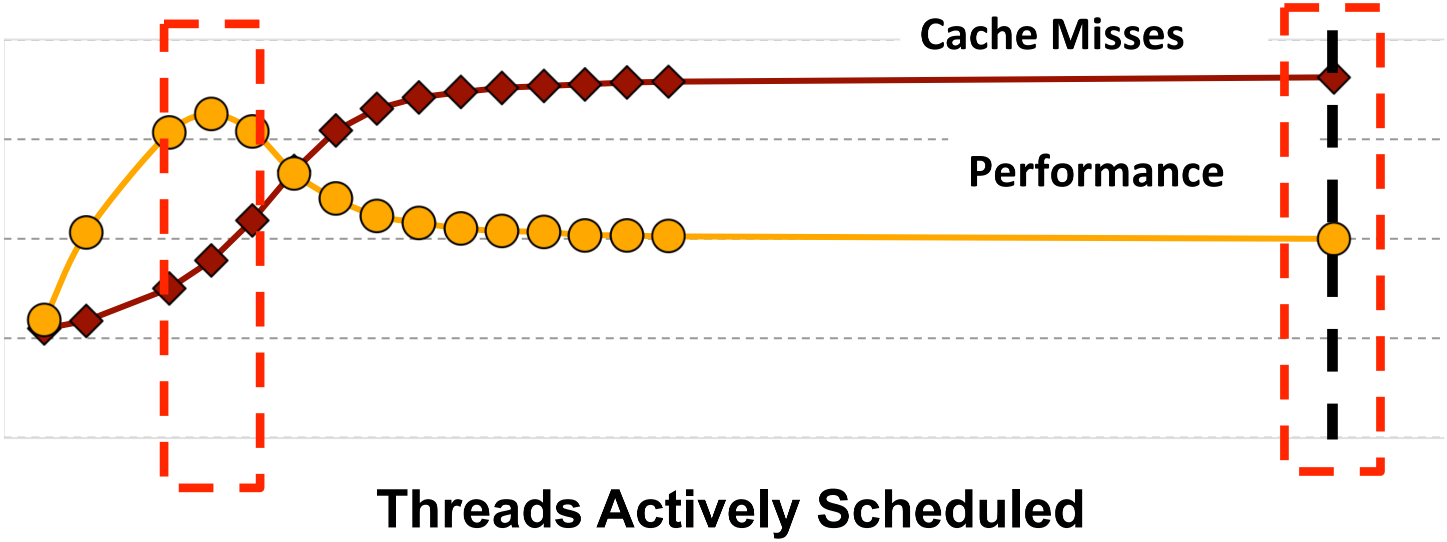
Processor

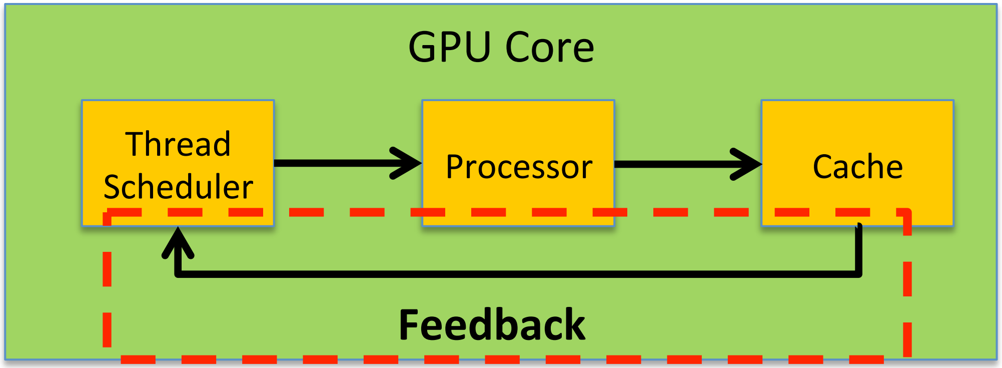
Cache



Use Memory System Feedback

[MICRO 2012]

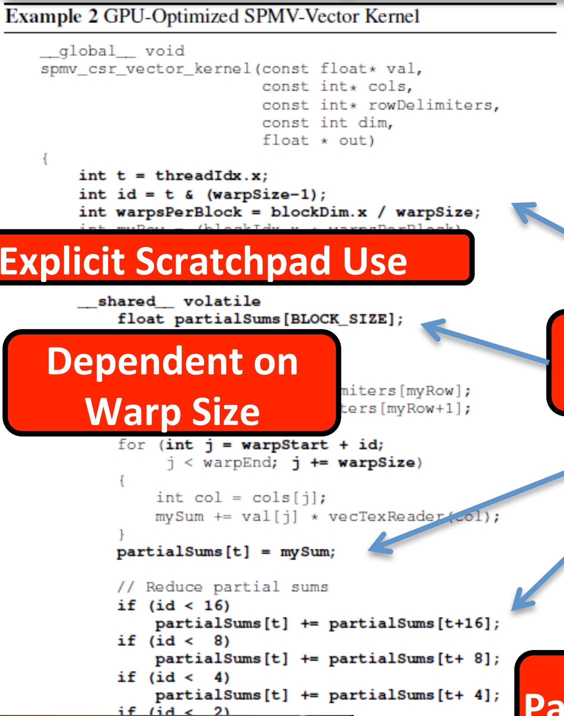




Programmability case study [MICRO 2013]

Sparse Vector-Matrix Multiply

GPU-Optimized Version
SHOC Benchmark Suite
(Oakridge National Labs)



Example 2 GPU-Optimized SPMV-Vector Kernel

```
__global__ void
spmv_csr_vector_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float * out)
{
    int t = threadIdx.x;
    int id = t & (warpSize-1);
    int warpsPerBlock = blockDim.x / warpSize;

    __shared__ volatile
        float partialSums[BLOCK_SIZE];

                              miters[myRow];
                              ters[myRow+1];

        for (int j = warpStart + id;
             j < warpEnd; j += warpSize)
        {
            int col = cols[j];
            mySum += val[j] * vecTexReader(col);
        }
        partialSums[t] = mySum;

        // Reduce partial sums
        if (id < 16)
            partialSums[t] += partialSums[t+16];
        if (id <  8)
            partialSums[t] += partialSums[t+ 8];
        if (id <  4)
            partialSums[t] += partialSums[t+ 4];
                              rtialSums[t+ 2];
                              rtialSums[t+ 1];

                              Sums[t];
```

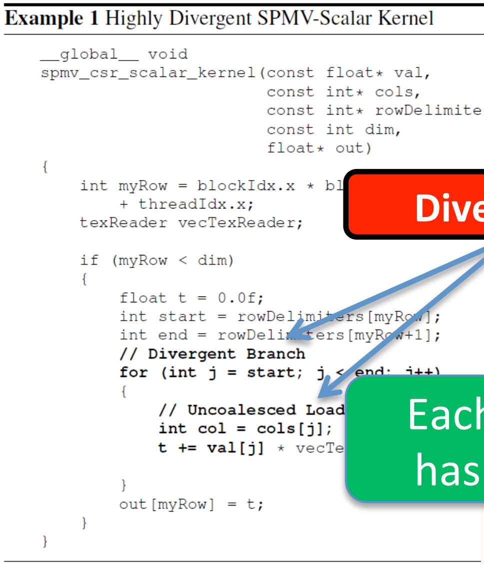
Explicit Scratchpad Use

Dependent on Warp Size

Added Complication

Parallel Reduction

Simple Version



Example 1 Highly Divergent SPMV-Scalar Kernel

```
__global__ void
spmv_csr_scalar_kernel(const float* val,
                       const int* cols,
                       const int* rowDelimiters,
                       const int dim,
                       float* out)
{
    int myRow = blockIdx.x * bl
        + threadIdx.x;
    texReader vecTexReader;

    if (myRow < dim)
    {
        float t = 0.0f;
        int start = rowDelimiters[myRow];
        int end = rowDelimiters[myRow+1];
        // Divergent Branch
        for (int j = start; j < end; j++)
        {
            // Uncoalesced Load
            int col = cols[j];
            t += val[j] * vecTe

        }
        out[myRow] = t;
    }
}
```

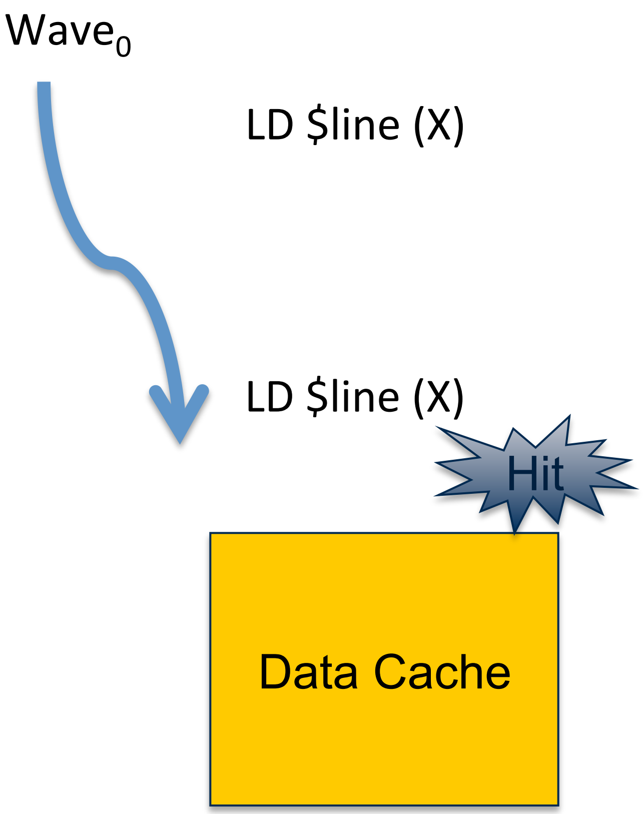
Divergence

Each thread has locality

Using DAWS scheduling within 4% of optimized with no programmer input

Sources of Locality

Intra-wavefront locality

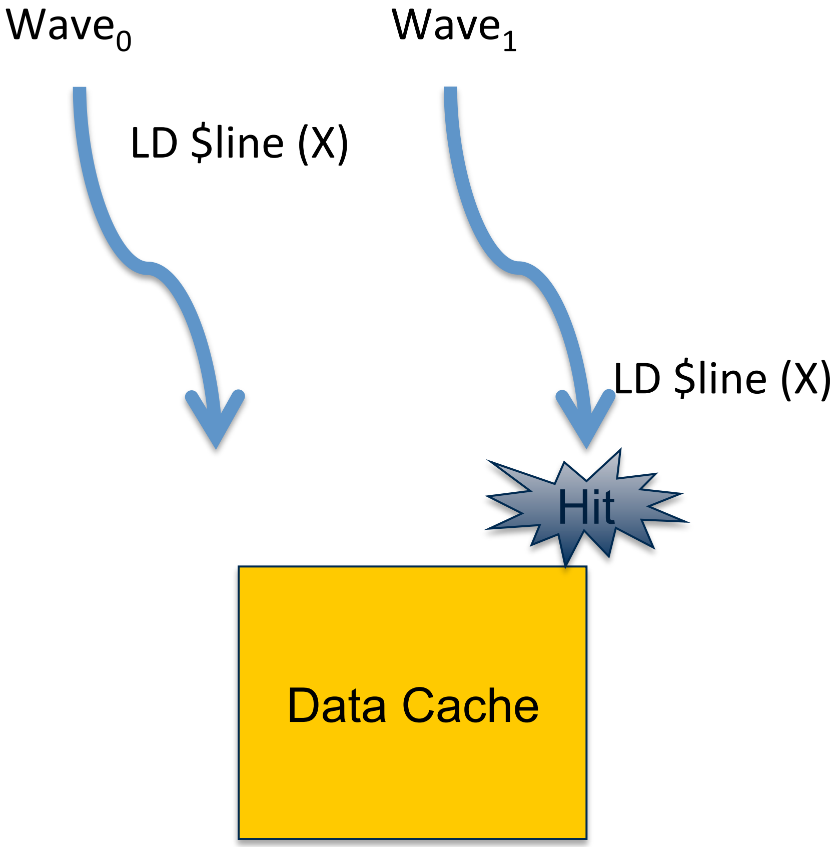


Wave$_0$

LD $line (X)

LD $line (X)

Inter-wavefront locality

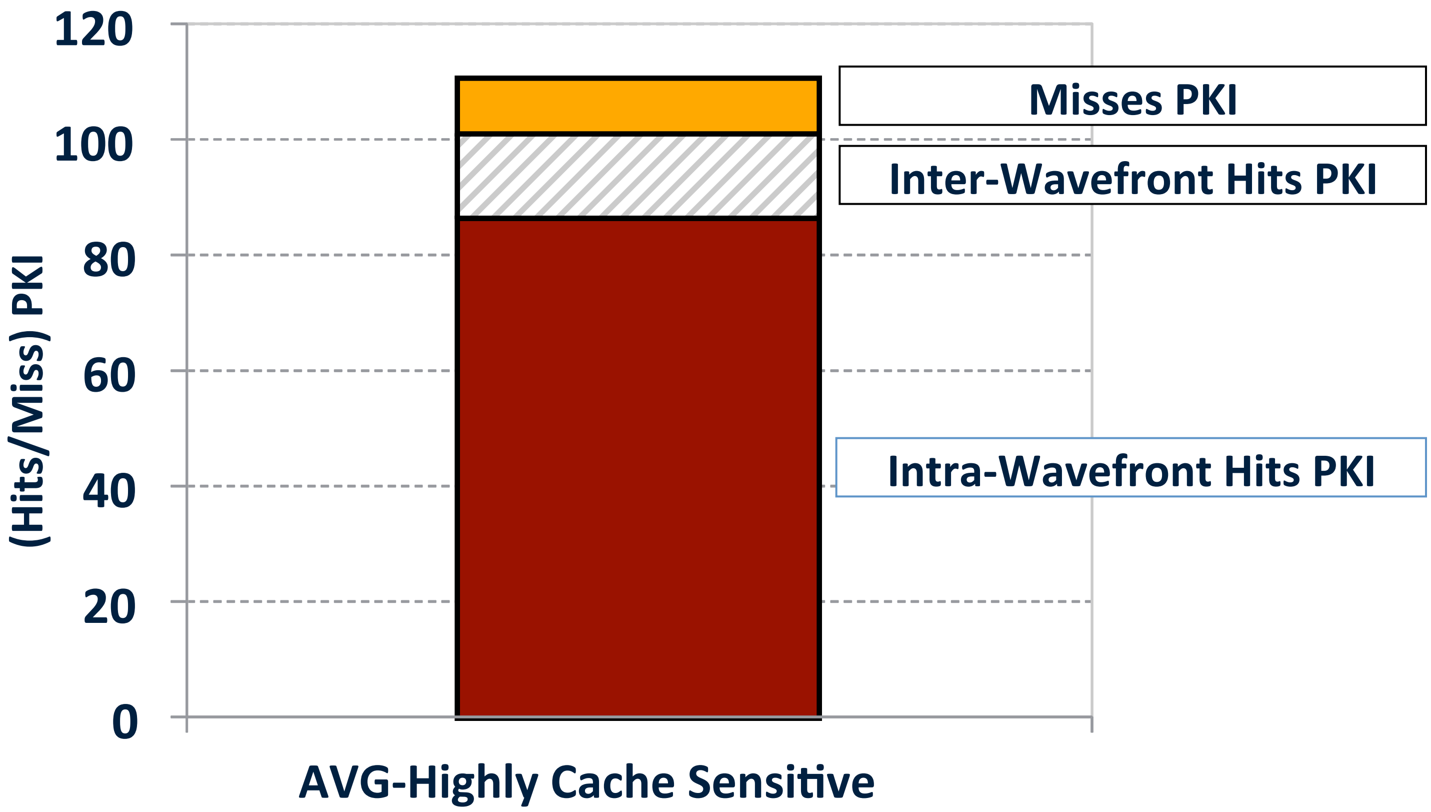


Wave$_0$

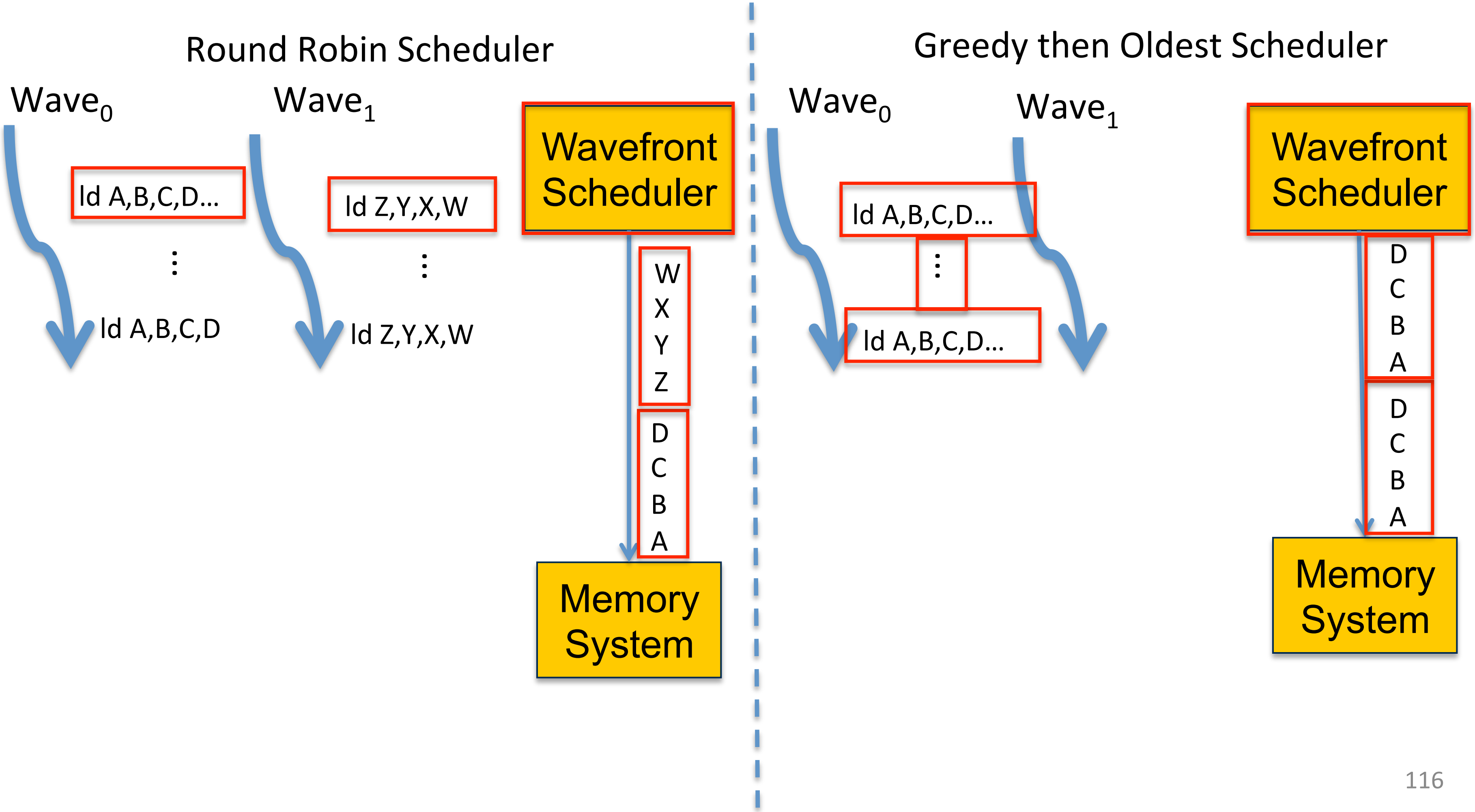
Wave$_1$

LD $line (X)

LD $line (X)


120
100
80
60
40
20
0
(Hits/Miss) PKI
Misses PKI
Inter-Wavefront Hits PKI
Intra-Wavefront Hits PKI
AVG-Highly Cache Sensitive

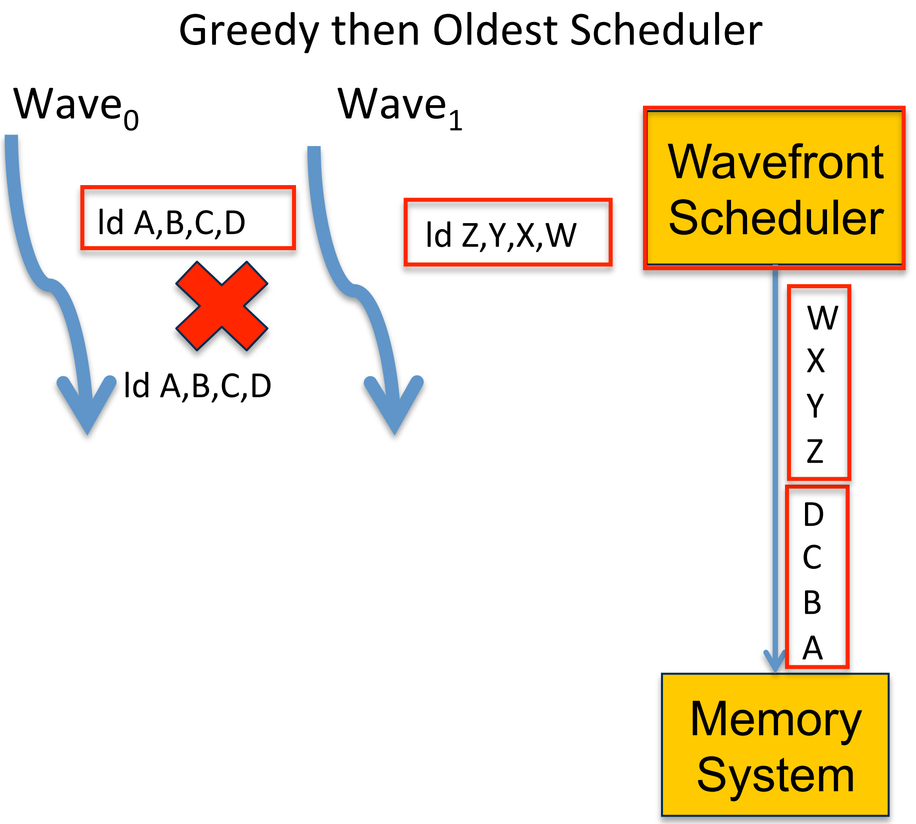
Scheduler affects access pattern



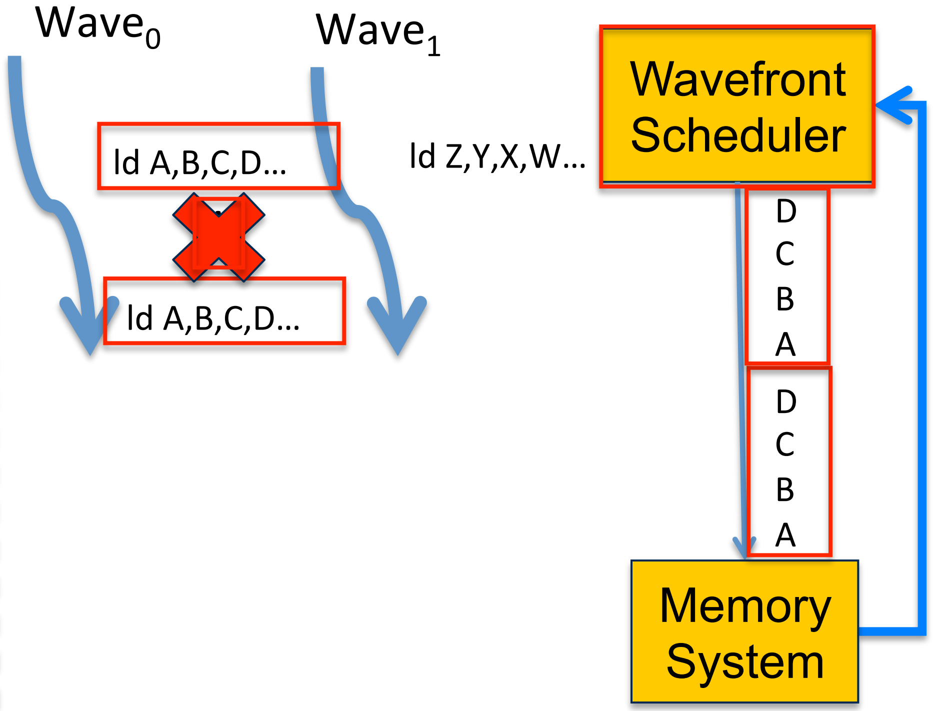
Round Robin Scheduler

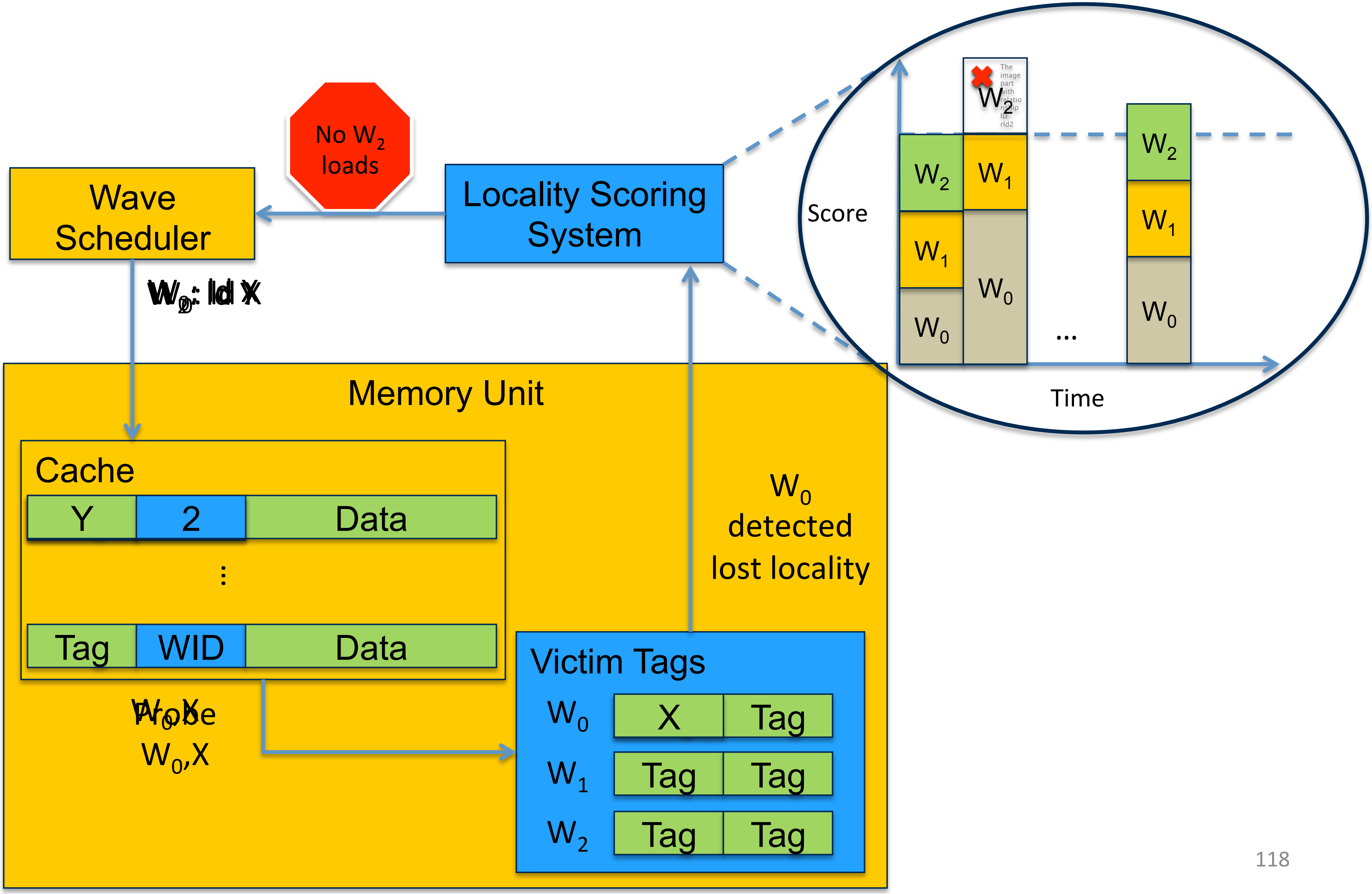
Greedy then Oldest Scheduler

Use scheduler to *shape* access pattern



Cache-Conscious Wavefront Scheduling
[MICRO 2012 best paper runner up,
Top Picks 2013, CACM Research Highlight]





Wave Scheduler

No W_2 loads

Locality Scoring System

W_0: ld X

W_2: ld X

Memory Unit

Cache

Y	2	Data
⋮		
Tag	WID	Data

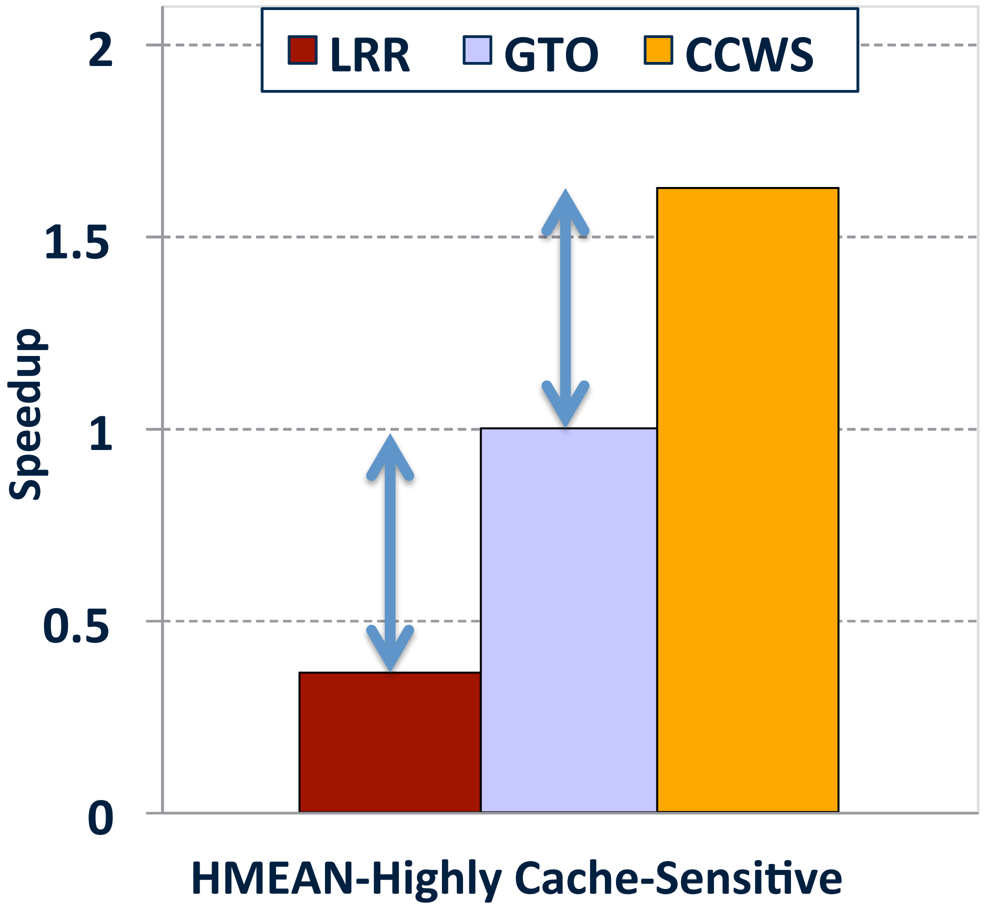
Probe W_0,X

W_0,X

W_0 detected lost locality

Victim Tags

W_0	X	Tag
W_1	Tag	Tag
W_2	Tag	Tag


LRR
GTO
CCWS
Speedup
2
1.5
1
0.5
0
HMEAN-Highly Cache-Sensitive

Static Wavefront Limiting

[Rogers et al., MICRO 2012]

- Profiling an application we can find an optimal number of wavefronts to execute
- Does a little better than CCWS.
- Limitations: Requires profiling, input dependent, does not exploit phase behavior.

Improve upon CCWS?

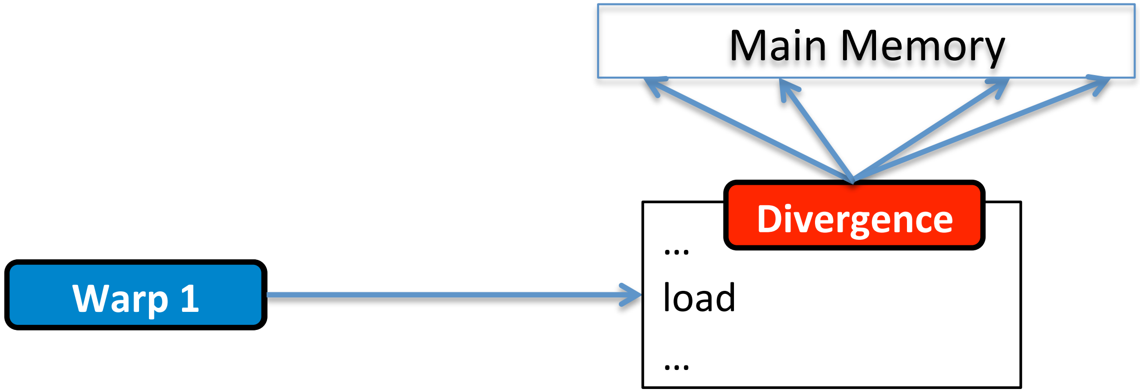
- CCWS detects bad scheduling decisions and avoids them in future.

- Would be better if we could “think ahead” / “be proactive” instead of “being reactive”

Observations

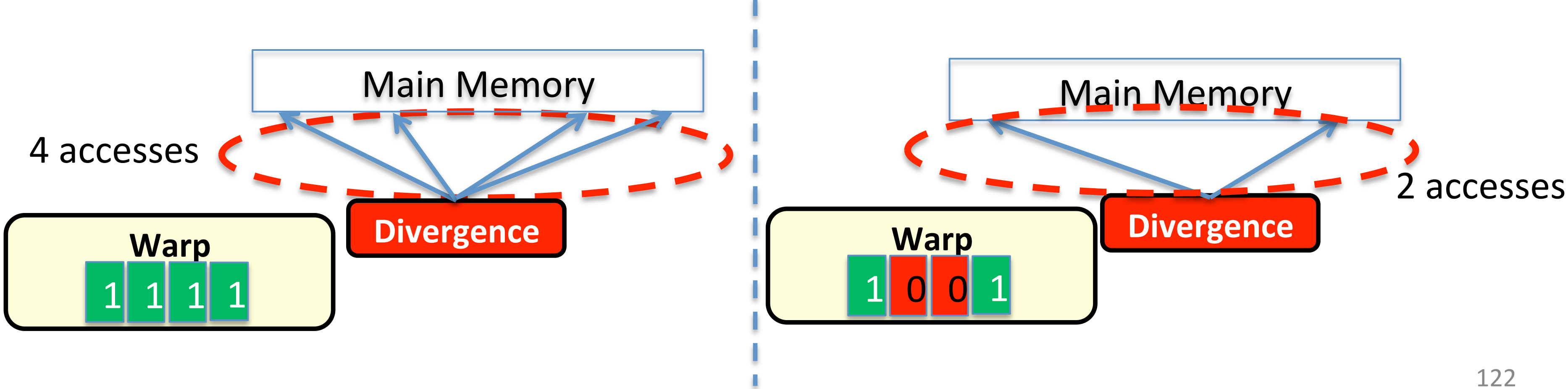
[Rogers et al., MICRO 2013]

- Memory divergence in static instructions is predictable



Both Used To Create Cache Footprint Prediction

- Data touched by divergent loads dependent on active mask



Footprint Prediction

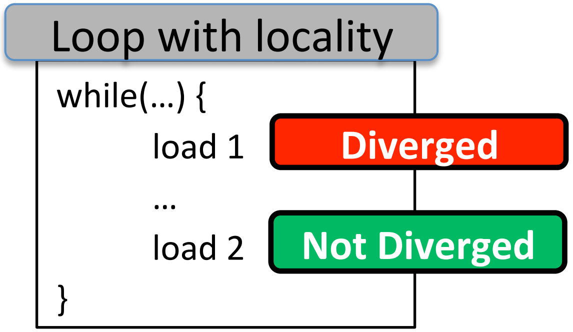
1. Detect loops with locality

Some loops have locality

Some don't

Limit multithreading here

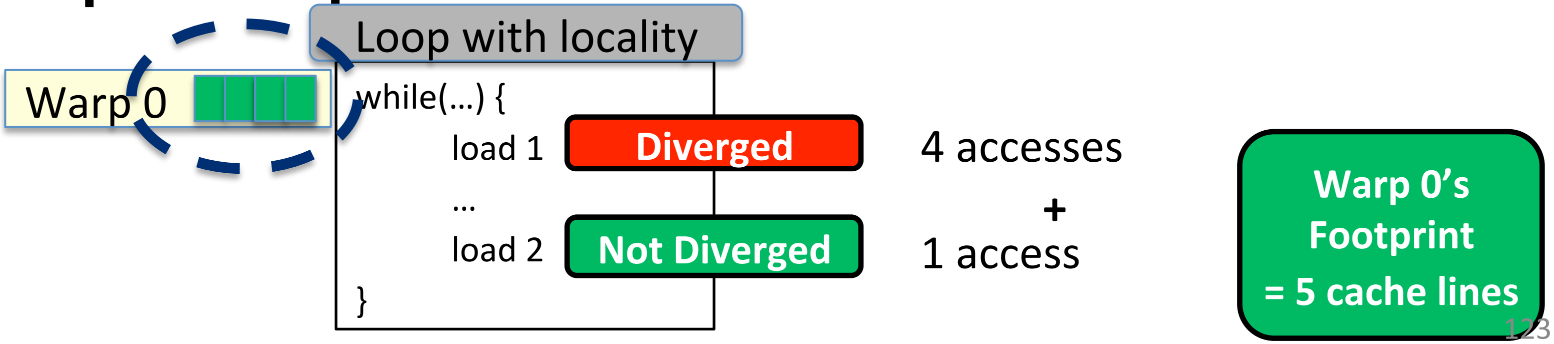
2. Classify loads in the loop



Loop with locality

```
while(…) {
        load 1    Diverged
        …
        load 2    Not Diverged
}
```

3. Compute footprint from active mask



Loop with locality

```
while(…) {
        load 1    Diverged
        …
        load 2    Not Diverged
}
```

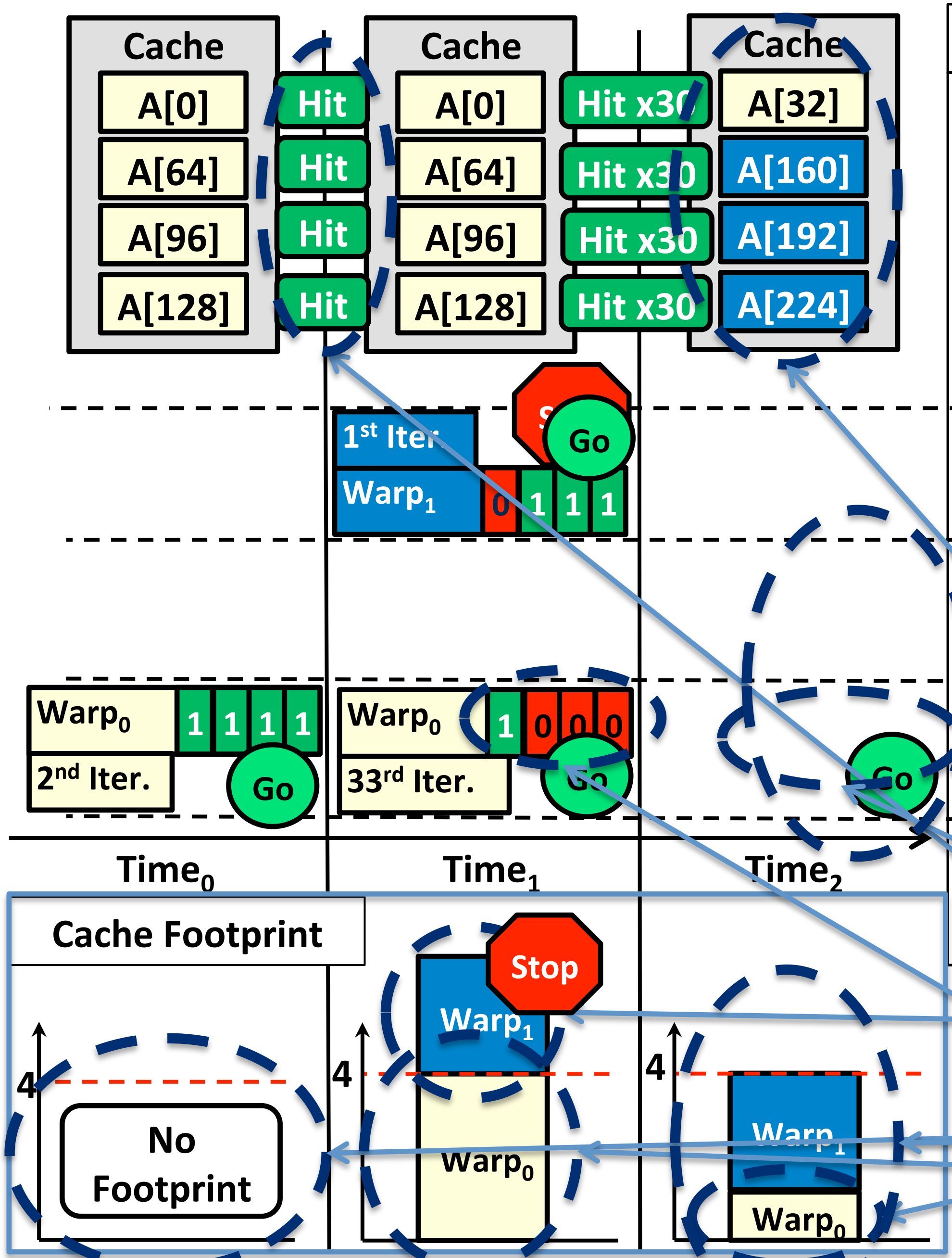
4 accesses

+

1 access

Warp 0's Footprint = 5 cache lines

DAWS Operation Example



Example Compressed Sparse Row Kernel

```
int C[]={0,64,96,128,160,160,192,224,256};
void sum_row_csr(float* A, …)  {
   float sum = 0;
   int i =C[tid];
```

Loop

```
   while(i < C[tid+1]) {

      sum  += A[ i ];

      ++i;

   }
…
```

Divergent Branch

Memory Divergence

Warp 0 has branch divergence

…arps profile for later …rps

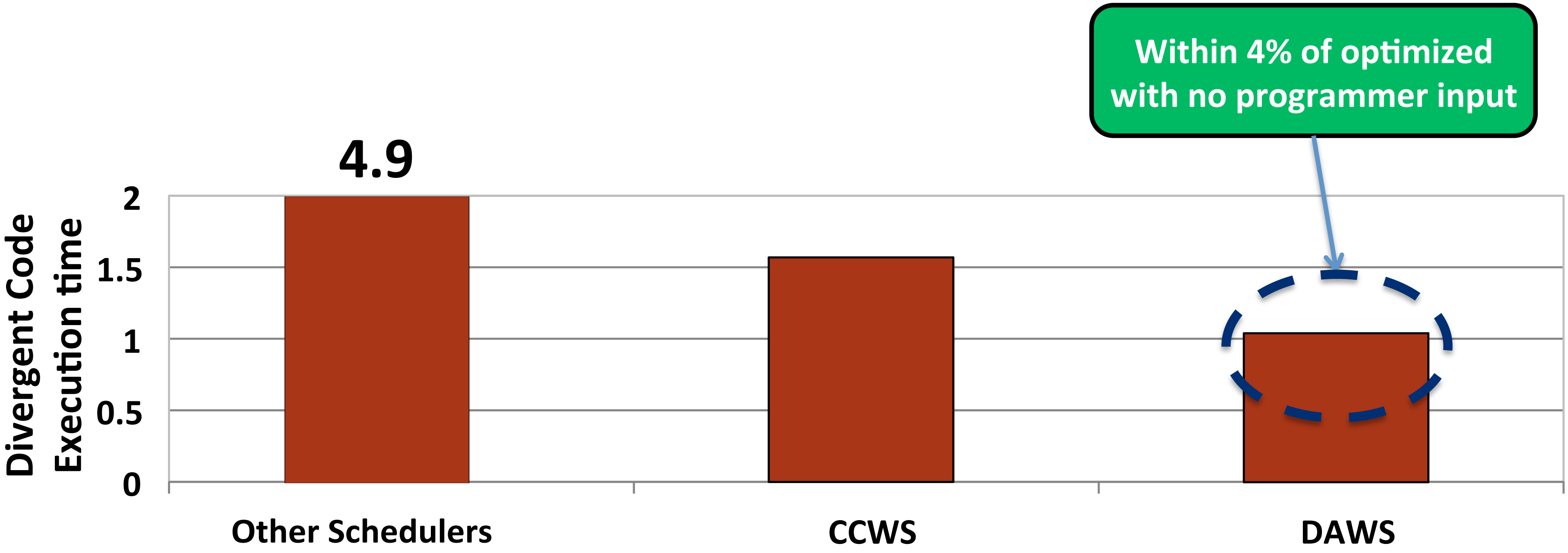
Both warps capture … together

= 4X1

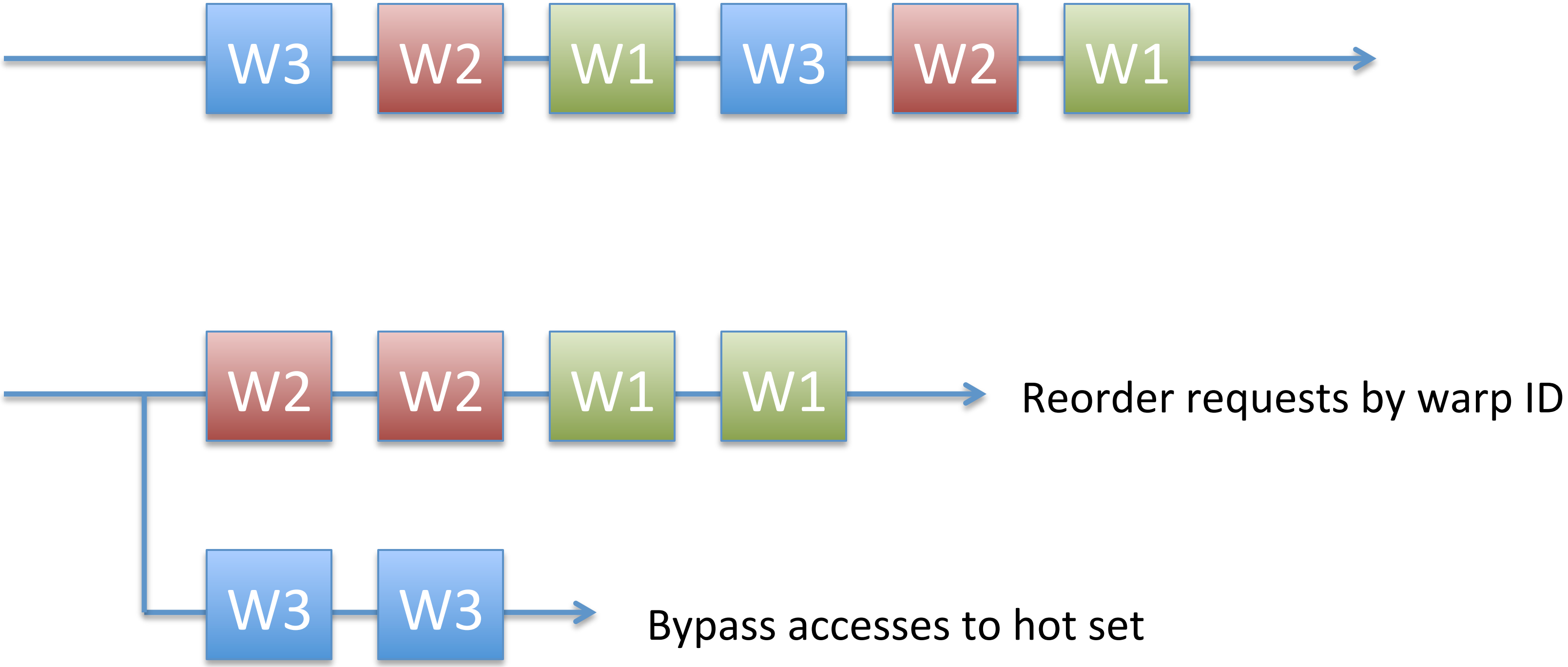
Footprint decreased

Sparse MM Case Study Results

- Performance (normalized to optimized version)



Memory Request Prioritization Buffer [Jia et al., HPCA 2014]



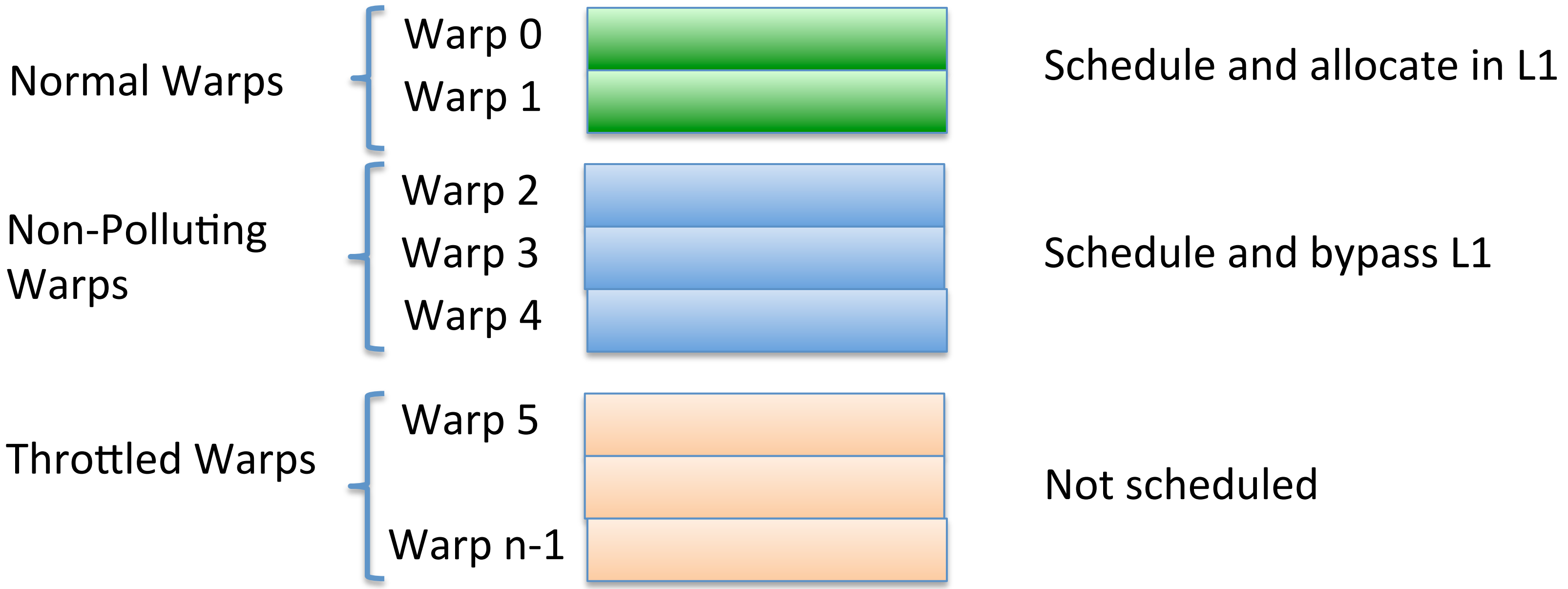
Reorder requests by warp ID

Bypass accesses to hot set

- Reorder requests by sorting by Warp ID.
- Bypass when too many accesses to same cache set.

Priority-Based Cache Allocation in Throughput Processors [Li et al., HPCA 2015]

- CCWS leaves L2 and DRAM underutilized.
- Allow some additional warps to execute but do not allow them to allocate space in cache:



Coordinated criticality-Aware Warp Acceleration (CAWA) [Lee et al., ISCA 2015]

- Some warps execute longer than others due to lack of uniformity in underlying workload.
- Give these warps more space in cache and more scheduling slots.
- Estimate critical path by observing amount of branch divergence and memory stalls.
- Also, predict if line inserted in line will be used by a warp that is critical using modified version of SHiP cache replacement algorithm.

Other Memory System Performance Considerations

- TLB Design for GPUs.
  - Current GPUs have translation look aside buffers (makes managing multiple graphics application surfaces easier; does not support paging)
  - How does large number of threads impact TLB design?
  - E.g., Power et al., *Supporting x86-64 Address Translation for 100s of GPU Lanes*, HPCA 2014. Importance of multithreaded page table walker + page walk cache.

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

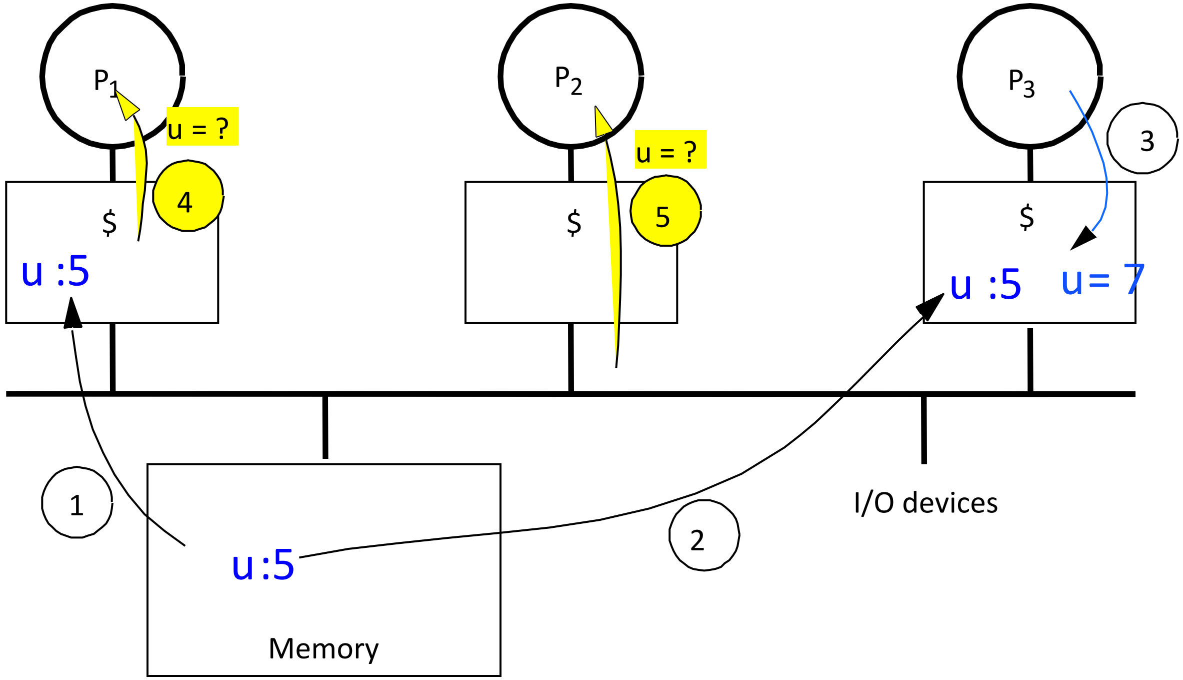
- Academic work: Treat GPU memory as cache on CPU memory (Kim et al., ScaleGPU, IEEE CAL early access).

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.

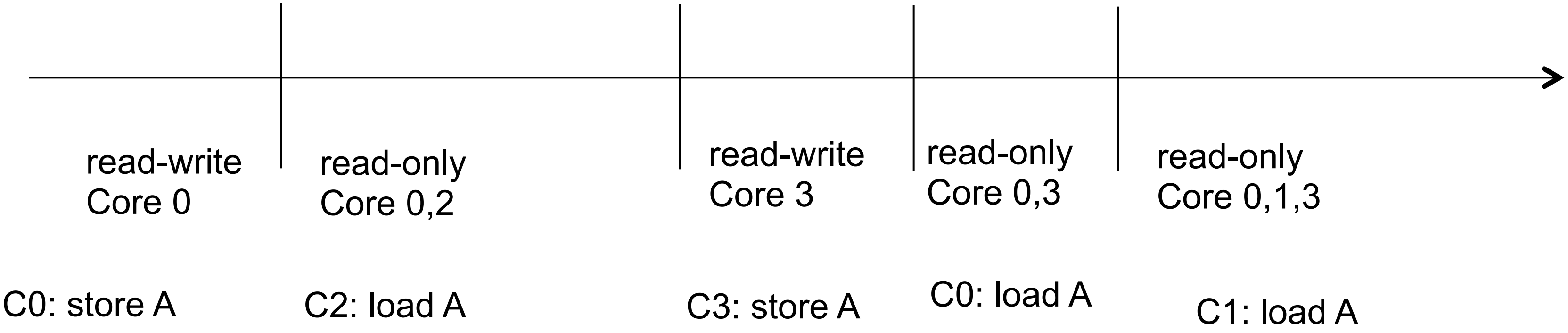
Review: Cache Coherence Problem



- Processors see different values for u after event 3
- With write back caches, value written back to memory depends on order of which cache writes back value first
- Unacceptable situation for programmers

Coherence Invariants

1. **Single-Writer, Multiple-Reader (SWMR) Invariant**



read-write Core 0

read-only Core 0,2

read-write Core 3

read-only Core 0,3

read-only Core 0,1,3

C0: store A

C2: load A

C3: store A

C0: load A

C1: load A

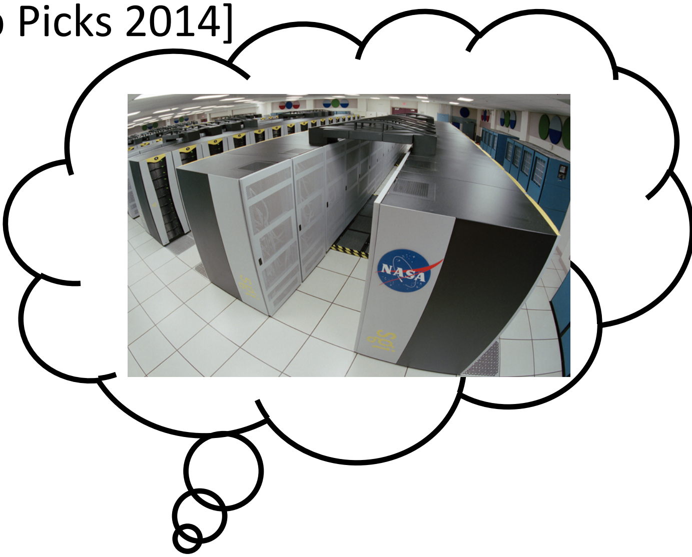
2. **Data-Value Invariant.** The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.

Coherence States

- How to design system satisfying invariants?
- Track “state” of memory block copies and ensure states changes satisfy invariants.
- Typical states: “modified”, “shared”, “invalid”.
- Mechanism for updating block state called a coherence protocol.

Intra-GPU Coherence

[Singh et al., HPCA 2013, IEEE Micro Top Picks 2014]

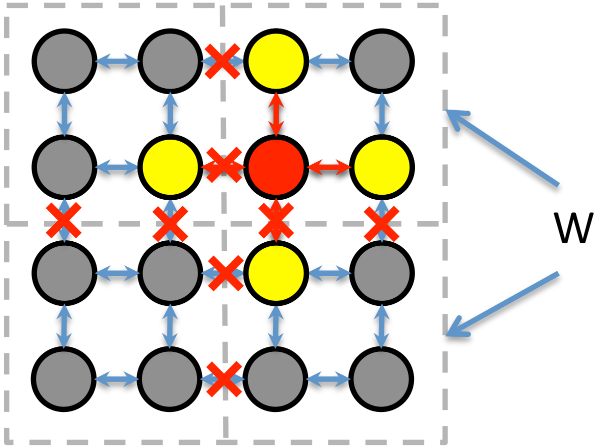


Coherent memory space

- Efficient critical sections
- Load balancing

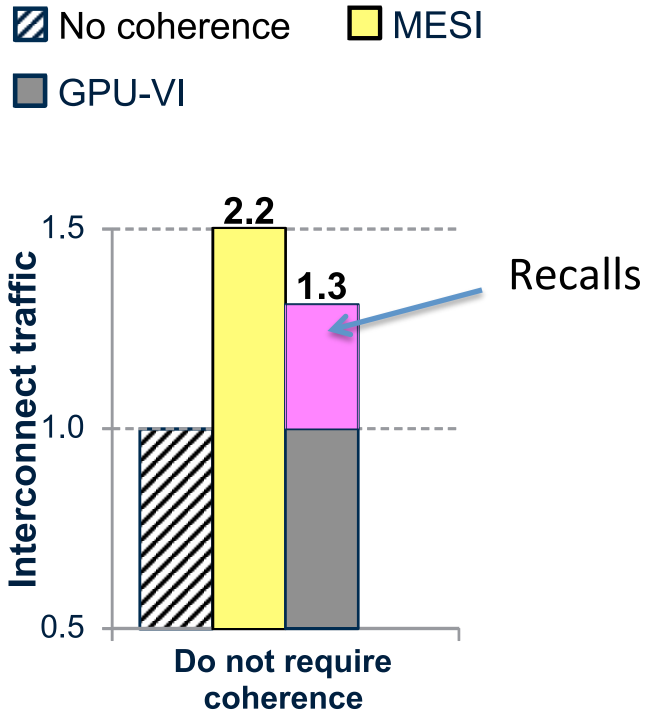
```
lock shared structure
    …
    computation
    …
unlock
```

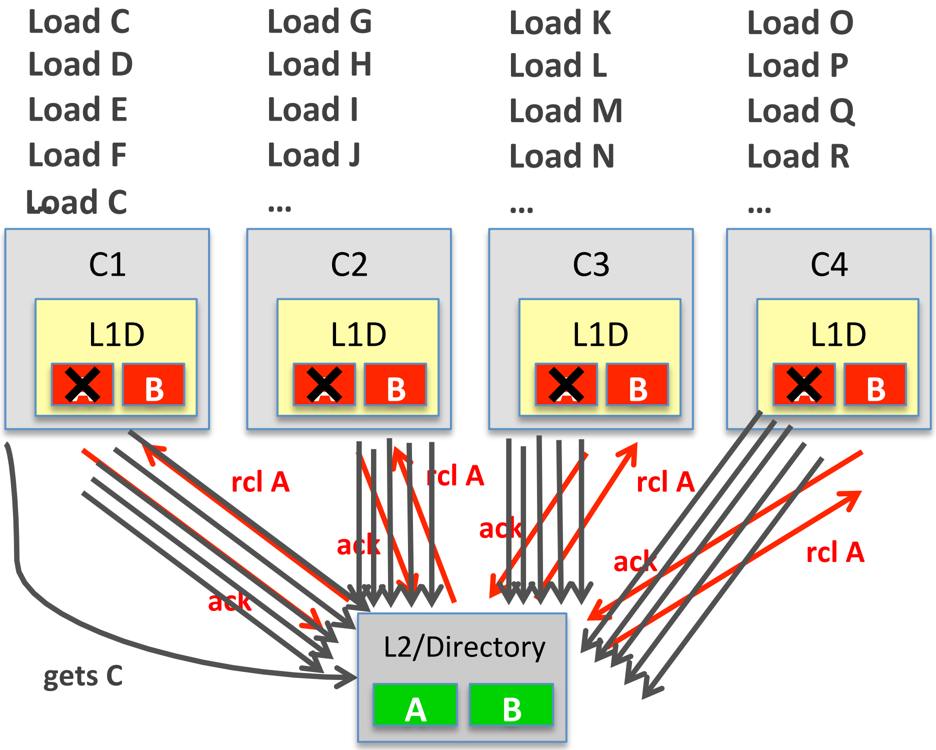
Stencil computation



GPU Coherence Challenges

- Challenge 1: Coherence traffic




Load C
Load D
Load E
Load F
Load C

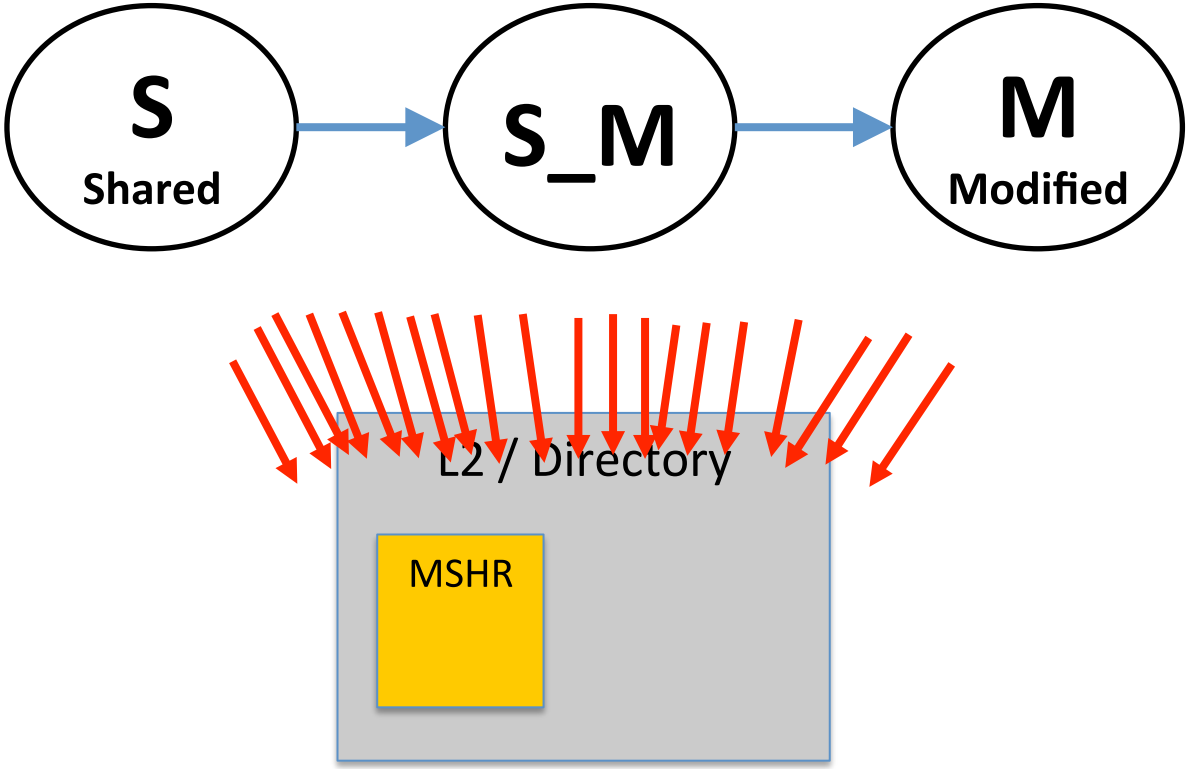
Load G
Load H
Load I
Load J
...

Load K
Load L
Load M
Load N
...

Load O
Load P
Load Q
Load R
...

GPU Coherence Challenges

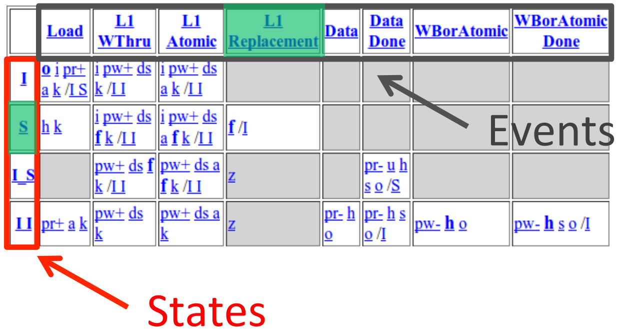
- Challenge 2: Tracking in-flight requests
  - Significant % of L2



GPU Coherence Challenges

- Challenge 3: Complexity

Non-coherent L1

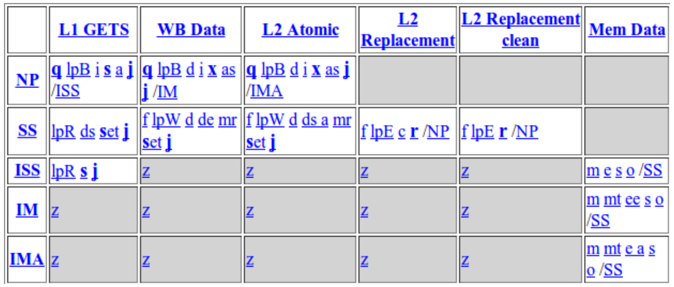


	Load	L1 WThru	L1 Atomic	L1 Replacement	Data	Data Done	WBorAtomic	WBorAtomic Done
I	o i pr+ a k /I S	i pw+ ds k /I I	i pw+ ds a k /I I					
S	h k	i pw+ ds f k /I I	i pw+ ds a f k /I I	f /I				
I_S		pw+ ds f k /I I	pw+ ds a f k /I I	z		pr- u h s o /S		
I I	pr+ a k	pw+ ds k	pw+ ds a k	z	pr- h o	pr- h s o /I	pw- h o	pw- h s o /I

Events

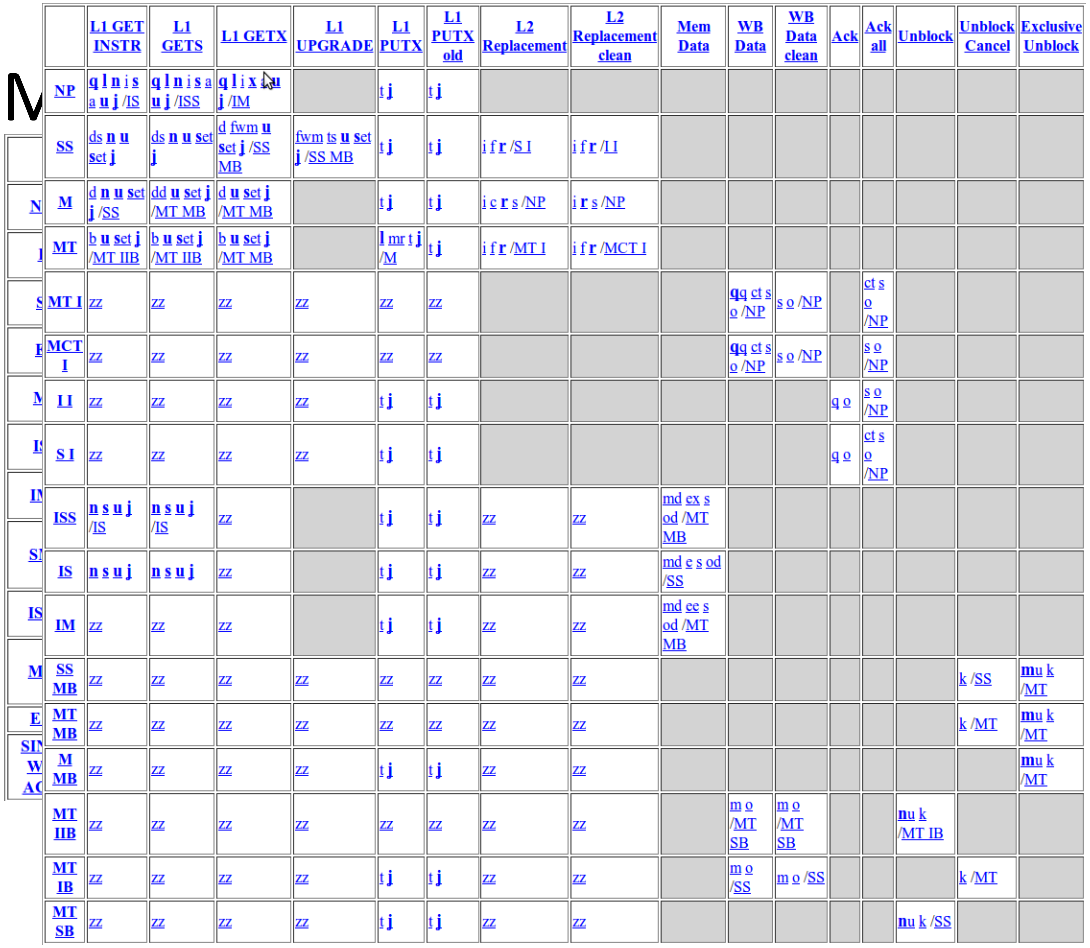
States

Non-coherent L2



	L1 GETS	WB Data	L2 Atomic	L2 Replacement	L2 Replacement clean	Mem Data
NP	q lpB i s a j /ISS	q lpB d i x as j /IM	q lpB d i x as j /IMA			
SS	lpR ds set j	f lpW d de mr set j	f lpW d ds a mr set j	f lpE c r /NP	f lpE r /NP	
ISS	lpR s j	z	z	z	z	m e s o /SS
IM	z	z	z	z	z	m mt ee s o /SS
IMA	z	z	z	z	z	m mt e a s o /SS

MESI L2 States



	L1 GET INSTR	L1 GETS	L1 GETX	L1 UPGRADE	L1 PUTX	L1 PUTX old	L2 Replacement	L2 Replacement clean	Mem Data	WB Data	WB Data clean	Ack	Ack all	Unblock	Unblock Cancel	Exclusive Unblock
NP	q l n i s a u j /IS	q l n i s a u j /ISS	q l i x a u j /IM		t j	t j										
SS	ds n u set j	ds n u set j	d fwm u set j /SS MB	fwm ts u set j /SS MB	t j	t j	i f r /S I	i f r /I I								
M	d n u set j /SS	dd u set j /MT MB	d u set j /MT MB		t j	t j	i c r s /NP	i r s /NP								
MT	b u set j /MT IIB	b u set j /MT IIB	b u set j /MT MB		l mr t j /M	t j	i f r /MT I	i f r /MCT I								
MT I	zz	zz	zz	zz	zz	zz				qq ct s o /NP	s o /NP		ct s o /NP			
MCT I	zz	zz	zz	zz	zz	zz				qq ct s o /NP	s o /NP		s o /NP			
I I	zz	zz	zz	zz	t j	t j						q o	s o /NP			
S I	zz	zz	zz	zz	t j	t j						q o	ct s o /NP			
ISS	n s u j /IS	n s u j /IS	zz		t j	t j	zz	zz	md ex s od /MT MB							
IS	n s u j	n s u j	zz		t j	t j	zz	zz	md e s od /SS							
IM	zz	zz	zz		t j	t j	zz	zz	md ee s od /MT MB							
SS MB	zz	zz	zz	zz	zz	zz	zz	zz							k /SS	mu k /MT
MT MB	zz	zz	zz	zz	zz	zz	zz	zz							k /MT	mu k /MT
M MB	zz	zz	zz	zz	t j	t j	zz	zz								mu k /MT
MT IIB	zz	zz	zz	zz	zz	zz	zz	zz		m o /MT SB	m o /MT SB			nu k /MT IB		
MT IB	zz	zz	zz	zz	t j	t j	zz	zz		m o /SS	m o /SS				k /MT	
MT SB	zz	zz	zz	zz	t j	t j	zz	zz						nu k /SS		

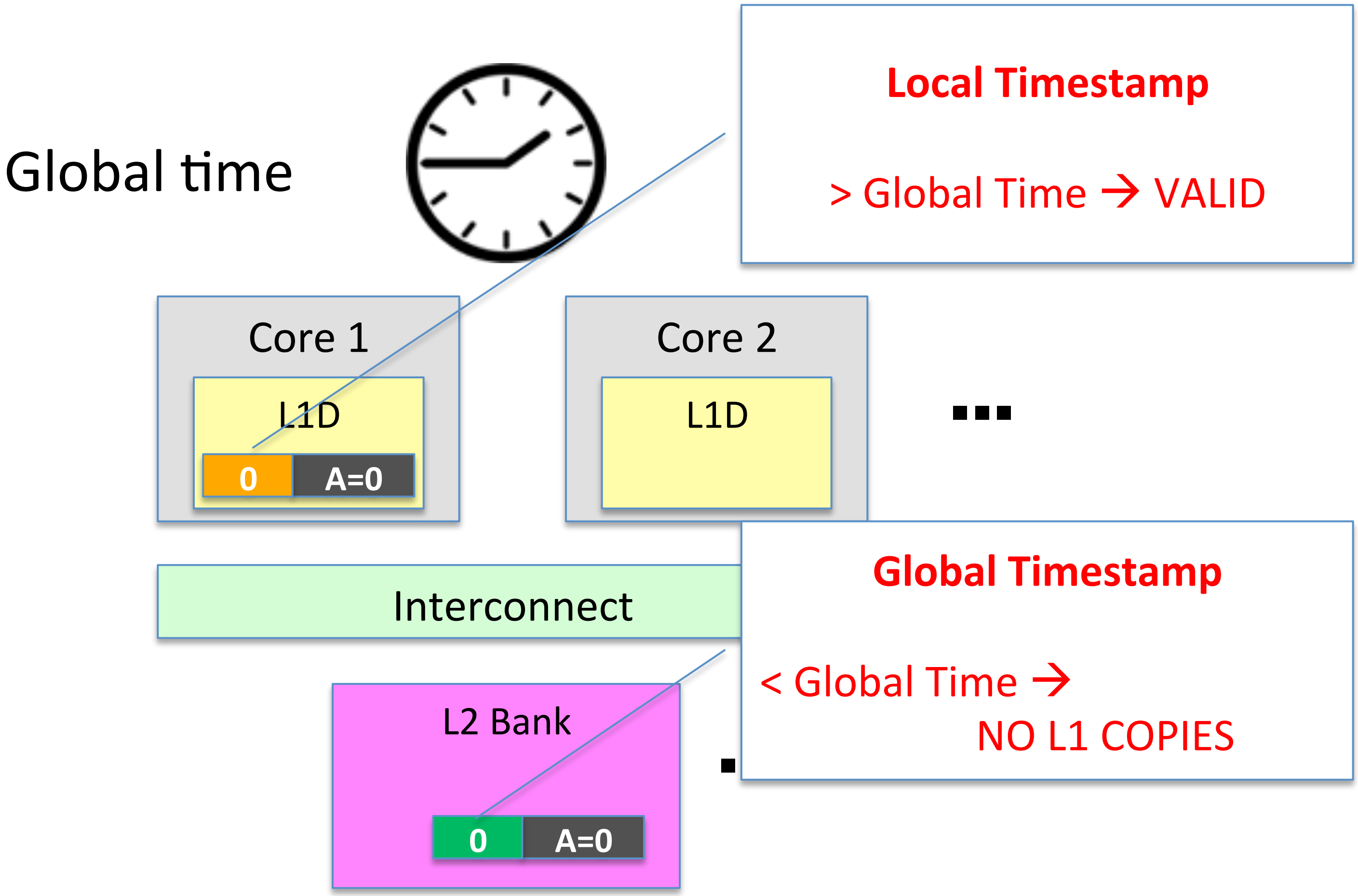
Coherence Challenges

- Challenges of introducing coherence messages on a GPU
  1. Traffic: transferring messages
  2. Storage: tracking message
  3. Complexity: managing races between messages

- GPU cache coherence without coherence messages?
  - YES – using global time

Temporal Coherence

Related: Library Cache Coherence



Global time

Local Timestamp

> Global Time → VALID

Core 1

L1D

0

A=0

Core 2

L1D

...

Interconnect

Global Timestamp

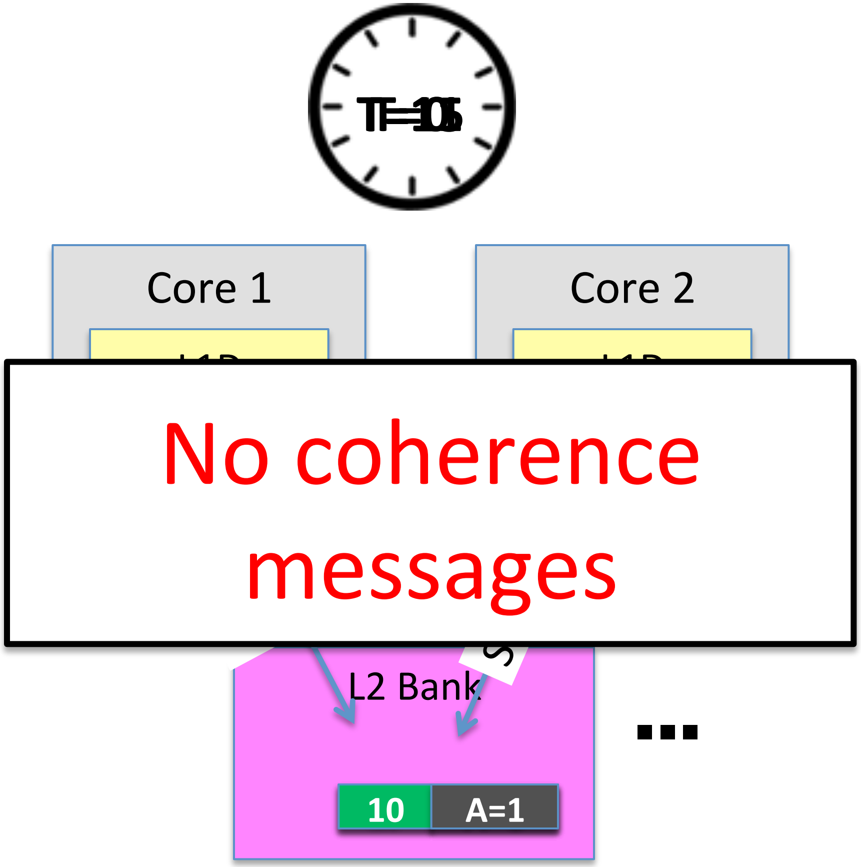
< Global Time →
NO L1 COPIES

L2 Bank

0

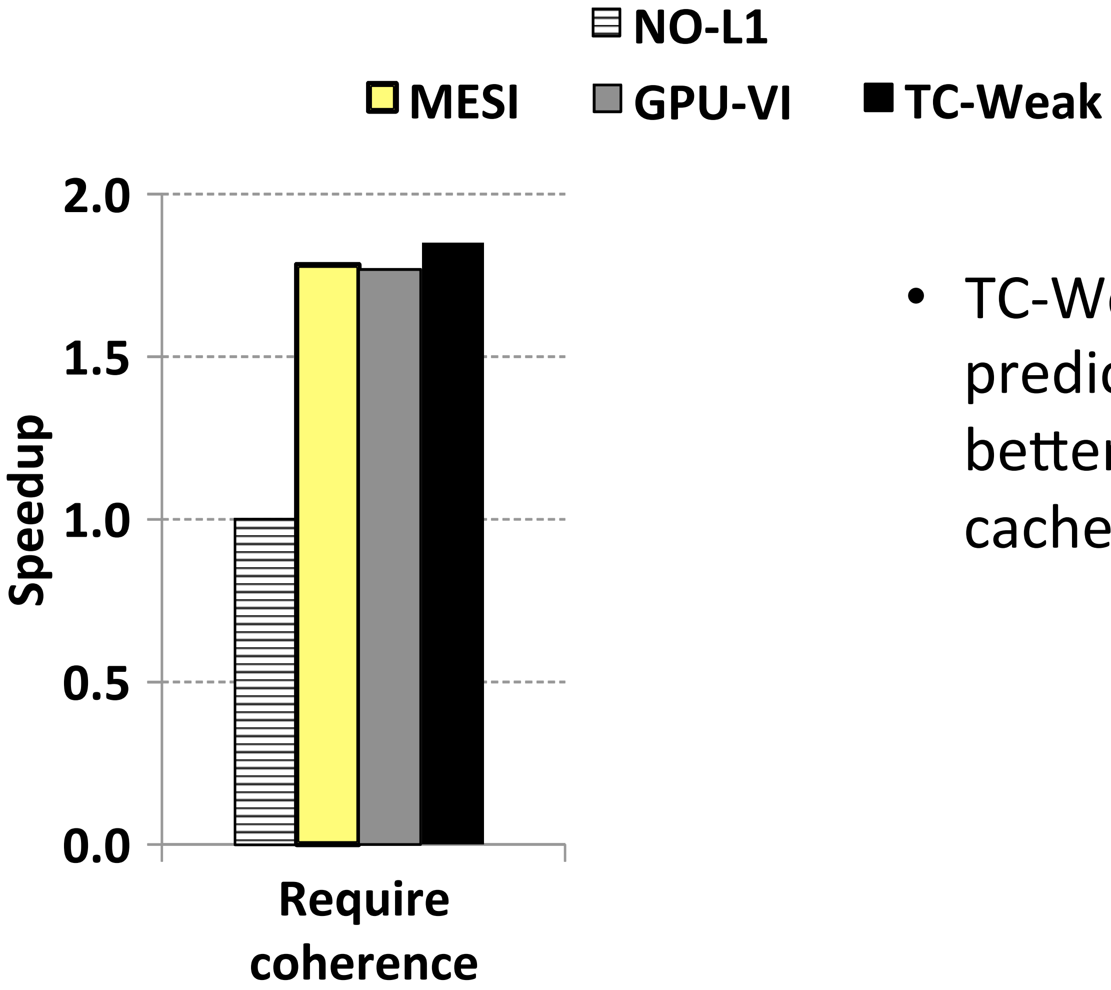
A=0

Temporal Coherence Example



No coherence messages

Performance



- TC-Weak with simple predictor performs 85% better than disabling L1 caches

CPU-GPU Coherence?

- Many vendors have introduced chips with both CPU and GPU (e.g., AMD Fusion, Intel Core i7, NVIDIA Tegra, etc…)
- What are the challenges with maintaining coherence across CPU and GPU?
- One important one: GPU has higher cache miss rate than CPU. Can place pressure on directory impacting performance.
- Power et al., *Heterogeneous System Coherence for Integrated CPU-GPU Systems*, ISCA 2013: Use "region coherence" to reduce number of GPU requests that need to access directory.

Review: Consistency Model

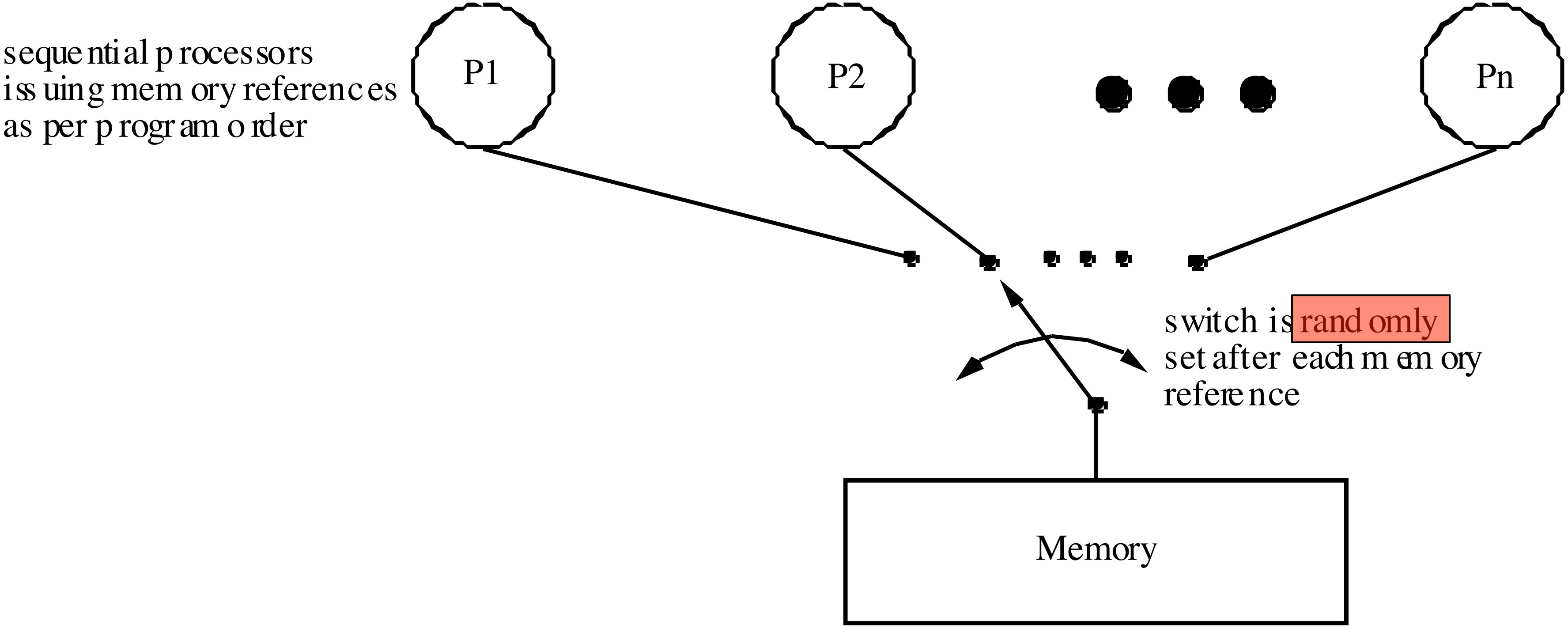
- Memory consistency model specifies **allowable** orderings of loads and stores to **different locations**

- The number of allowable execution orderings generally far greater than one.

- Ordering of operations from different processors is non-deterministic. Software must use synchronization (mutexes, semaphores, etc…) to provide determinism.

Sequential Consistency

- Sequential consistency is basically a “naïve” programmer’s intuition of allowable orderings:


sequential processors issuing memory references as per program order
P1
P2
Pn
switch is randomly set after each memory reference
Memory

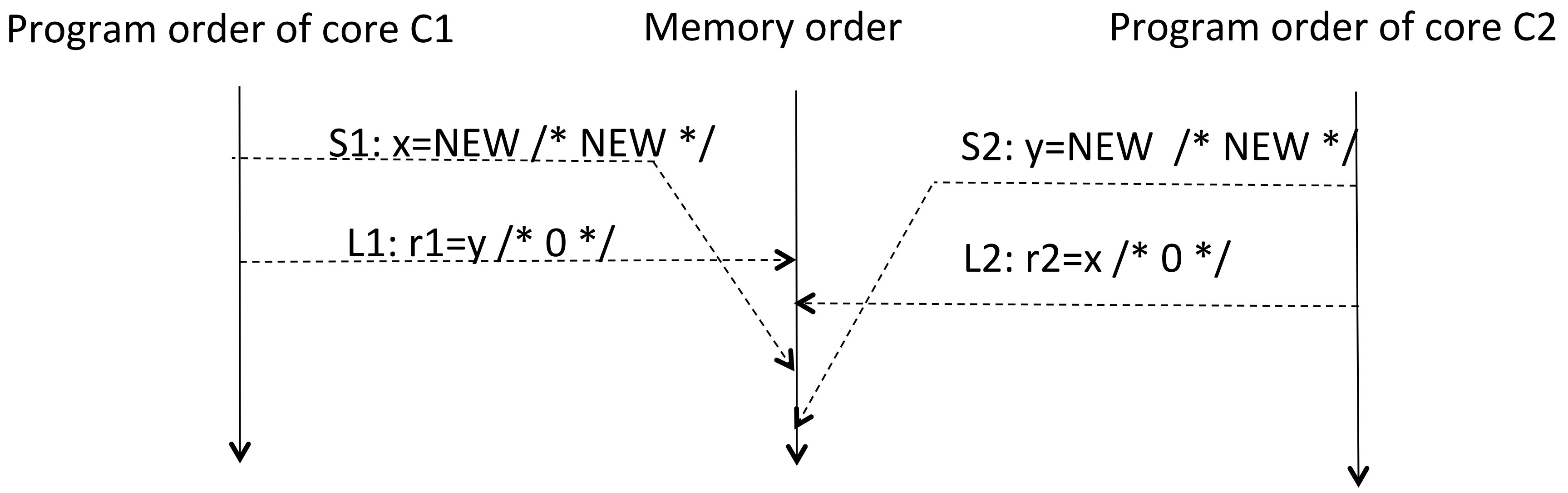
Total Store Order (TSO/x86) Memory Model

Use of write (store) buffer considered very important by Intel and AMD for x86.

Leads to total store order memory model supported by x86.

In general, memory model on multicore processors is not sequential consistency.

Example, TSO/x86 ordering



(r1, r2) = (0, 0) is legal outcome under TSO/x86 (!)

Current GPU Memory Consistency Models?

- NVIDIA Fermi: No coherence. Can have stale data in first level data cache (e.g., Barnes Hut example from GPU Gems). “Consistency”: Write from kernel N guaranteed to be visible to load from kernel N+1.

- NVIDIA Kepler restricts caching in L1D to global data compiler can prove is read only.

- See also: Alglave et al., “GPU Concurrency: Weak Behaviours and Programming Assumptions”, ASPLOS 2015.

Impact of Consistency Model on Performance of GPU Coherence?

- [Singh HPCA 2013] Assumes release consistency as do more recent AMD/Wisconsin papers on CPU-GPU coherence
- Hechtman and Sorin [ISCA 2013]: large number of threads on GPU may enable one to implement sequential consistency with same performance as more relaxed consistency models.
- One caveat: Write back caches in their study versus write through in existing GPUs.

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) ;
```

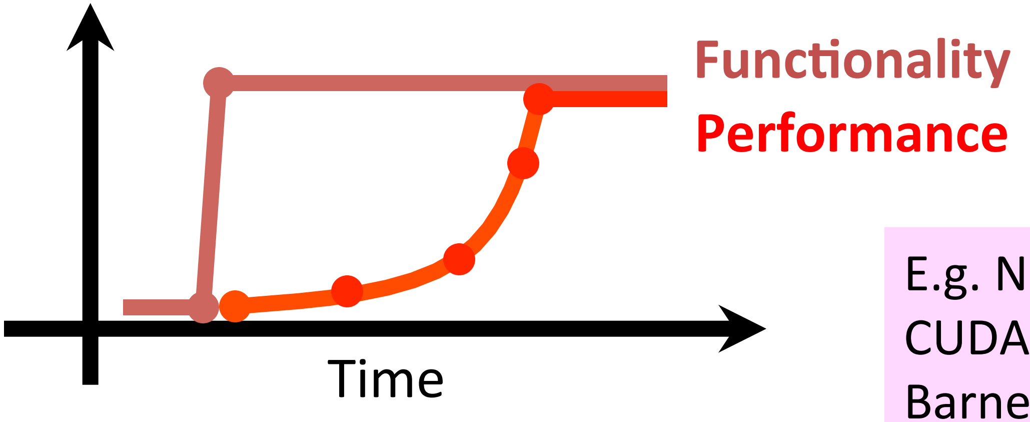
Correct way to write critical section for GPGPU:

```
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.

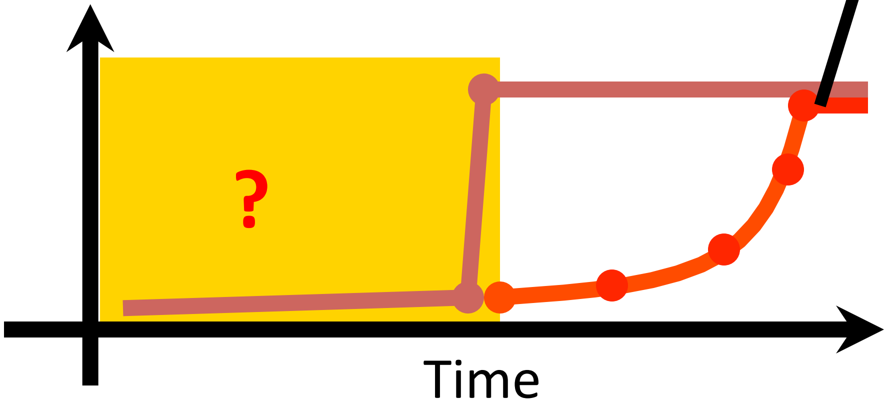
This leads to the following picture…

- Lifetime of GPU Application Development

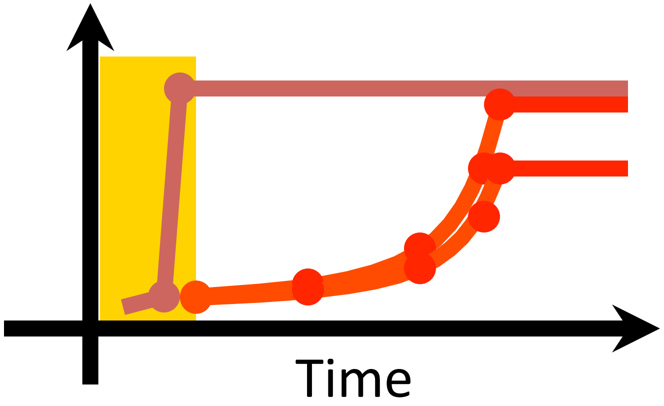


E.g. N-Body with 5M bodies
CUDA SDK: O(n^2) – 1640 s (barrier)
Barnes Hut: O(nLogn) – 5.2 s (locks)

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]);
```

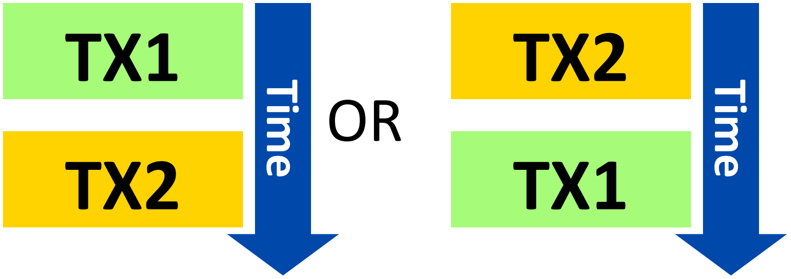
Potential Deadlock!

TM Version:

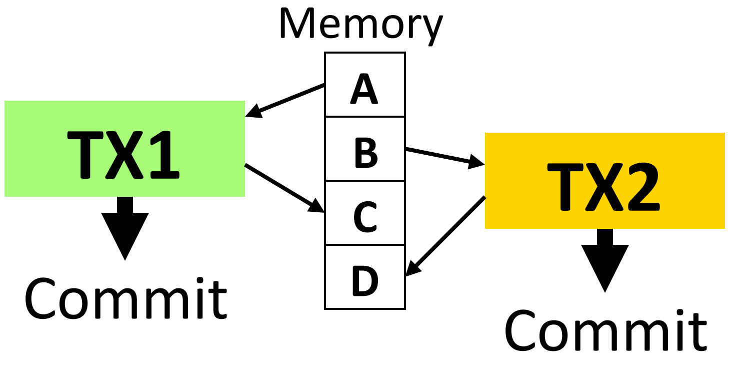
```
atomic {
  X[c] = X[a]+X[b];
}
```

Transactional Memory

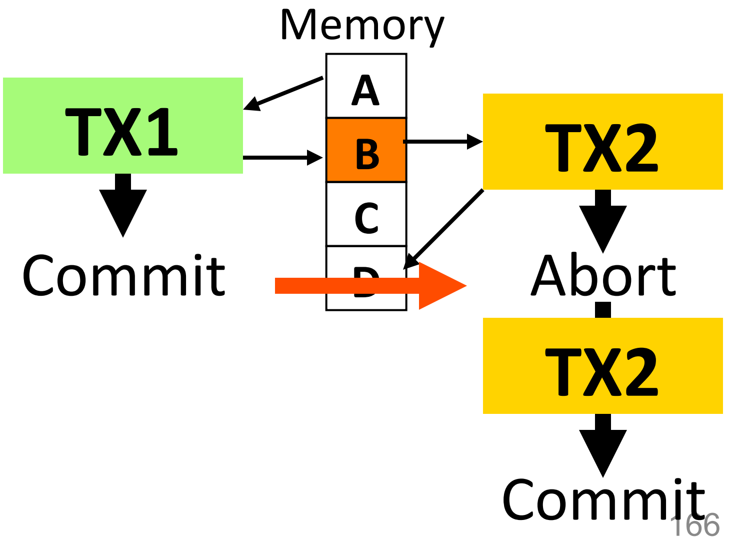
Programmers’ View:



Non-conflicting transactions may run in parallel



Conflicting transactions automatically serialized



Are TM and GPUs Incompatible?

GPU uarch very different from multicore CPU…

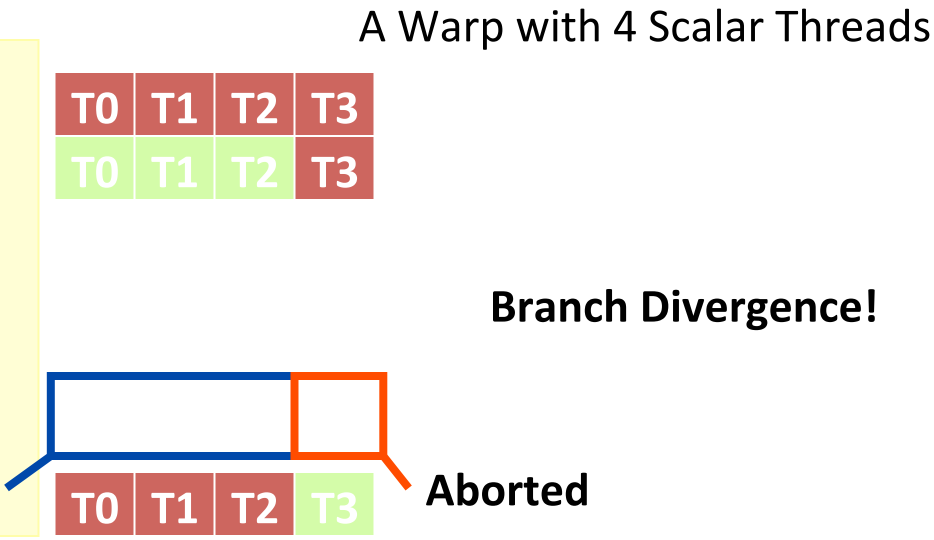
KILO TM [MICRO'11, IEEE Micro Top Picks]

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

Hardware TM for GPUs
Challenge #1: SIMD Hardware

- On GPUs, scalar threads in a warp/wavefront execute in lockstep

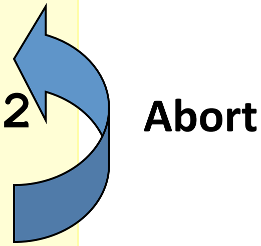
```
...
TxBegin
LD r2,[B]
ADD r2,r2,2
ST r2,[A]
TxCommit
...
```



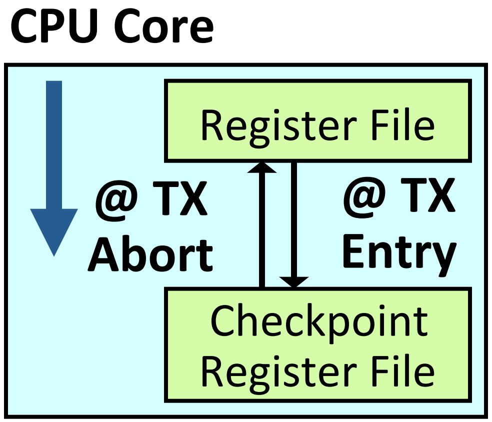
KILO TM – Solution to Challenge #1: SIMD Hardware

- Transaction Abort
  - Like a Loop
  - Extend SIMT Stack

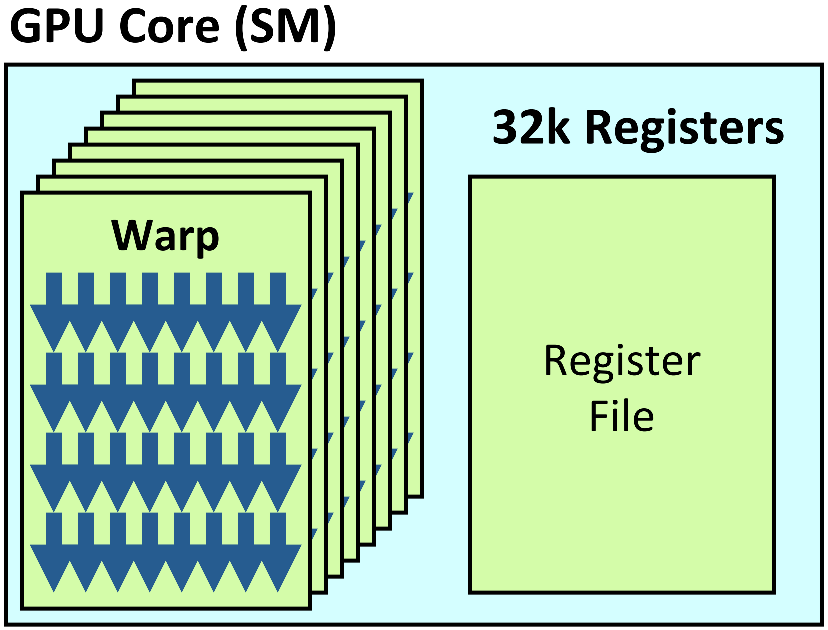
```
...
TxBegin
LD r2,[B]
ADD r2,r2,2
ST r2,[A]
TxCommit
...
```



Hardware TM for GPUs
Challenge #2: Transaction Rollback



10s of Registers

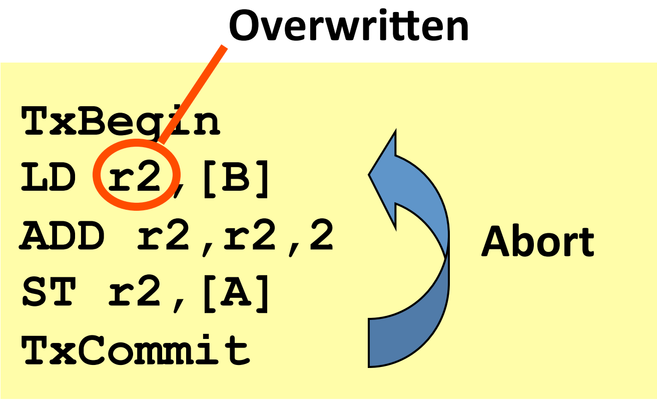


2MB Total On-Chip Storage

Checkpoint?

KILO TM – Solution to Challenge #2: Transaction Rollback

- SW Register Checkpoint
  - Most TX: Reg overwritten first appearance (idempotent)
  - TX in Barnes Hut: Checkpoint 2 registers



Overwritten

```
TxBegin
LD r2,[B]
ADD r2,r2,2
ST r2,[A]
TxCommit
```

Hardware TM for GPUs
Challenge #3: Conflict Detection

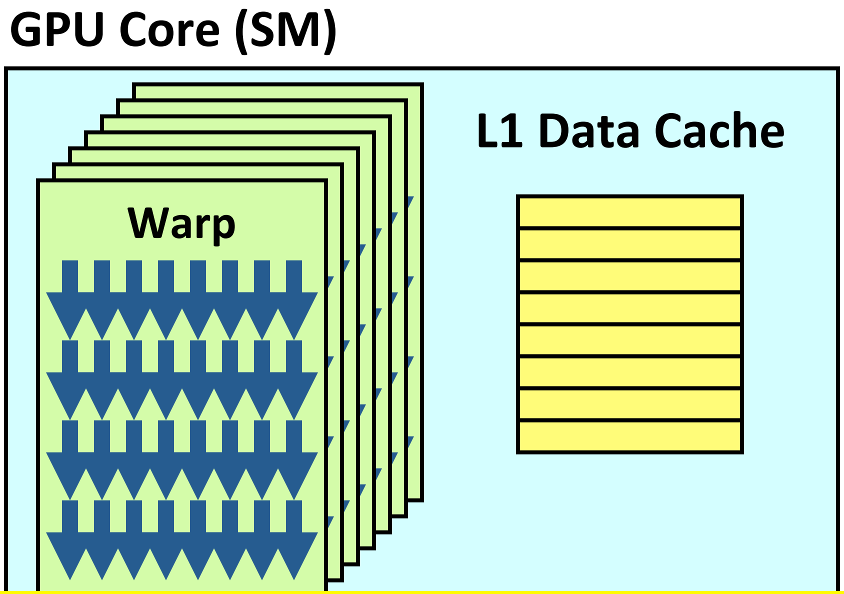
Existing HTMs use Cache Coherence Protocol

- Not Available on (current) GPUs
- No Private Data Cache per Thread

Signatures?

- 1024-bit / Thread
- **3.8MB / 30k Threads**

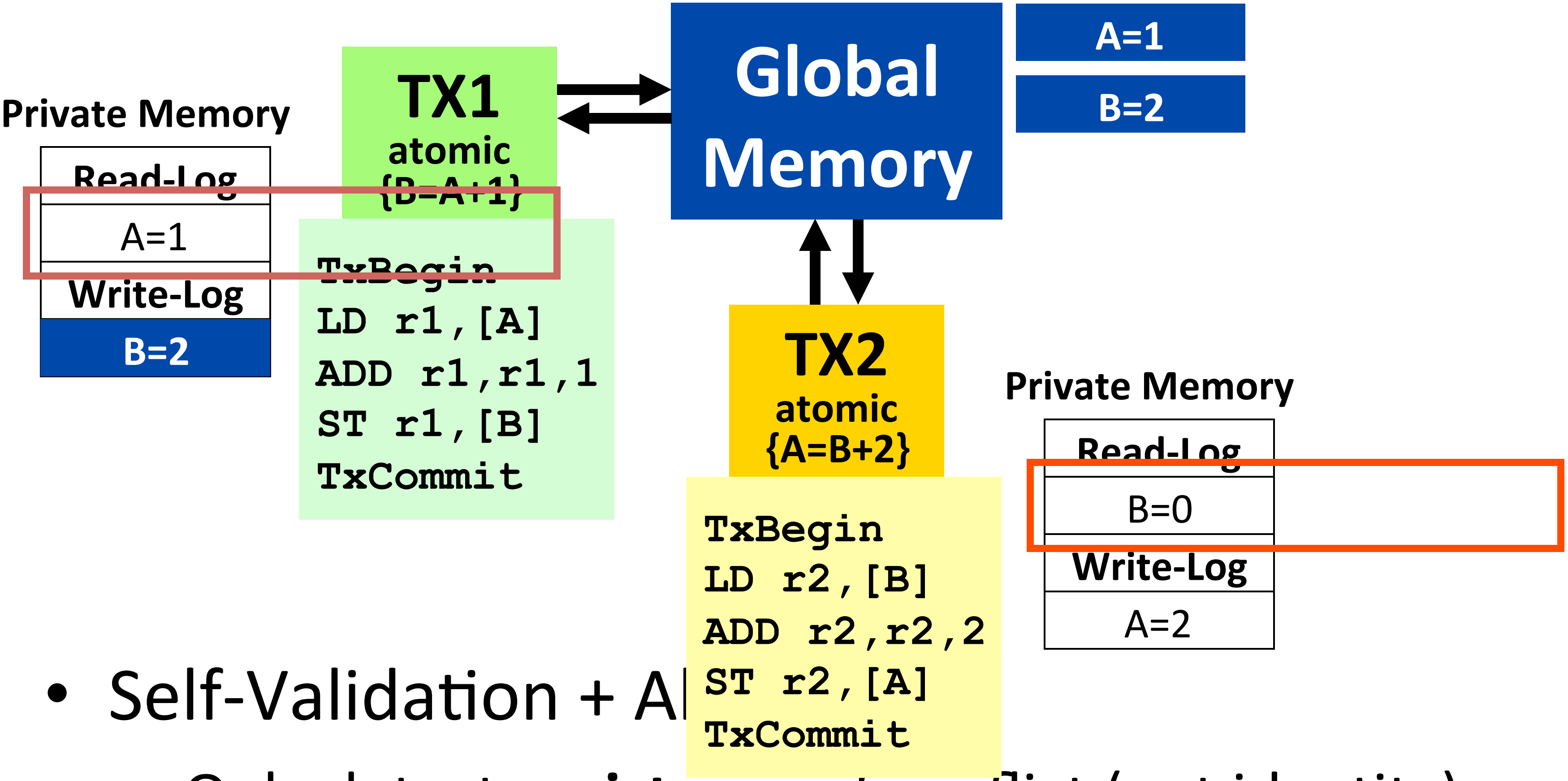
Hardware TM for GPUs
Challenge #4: Write Buffer



Problem: 384 lines / 1536 threads < 1 line per thread!

(48kB)
= 384 X 128B Lines

KILO TM:
Value-Based Conflict Detection



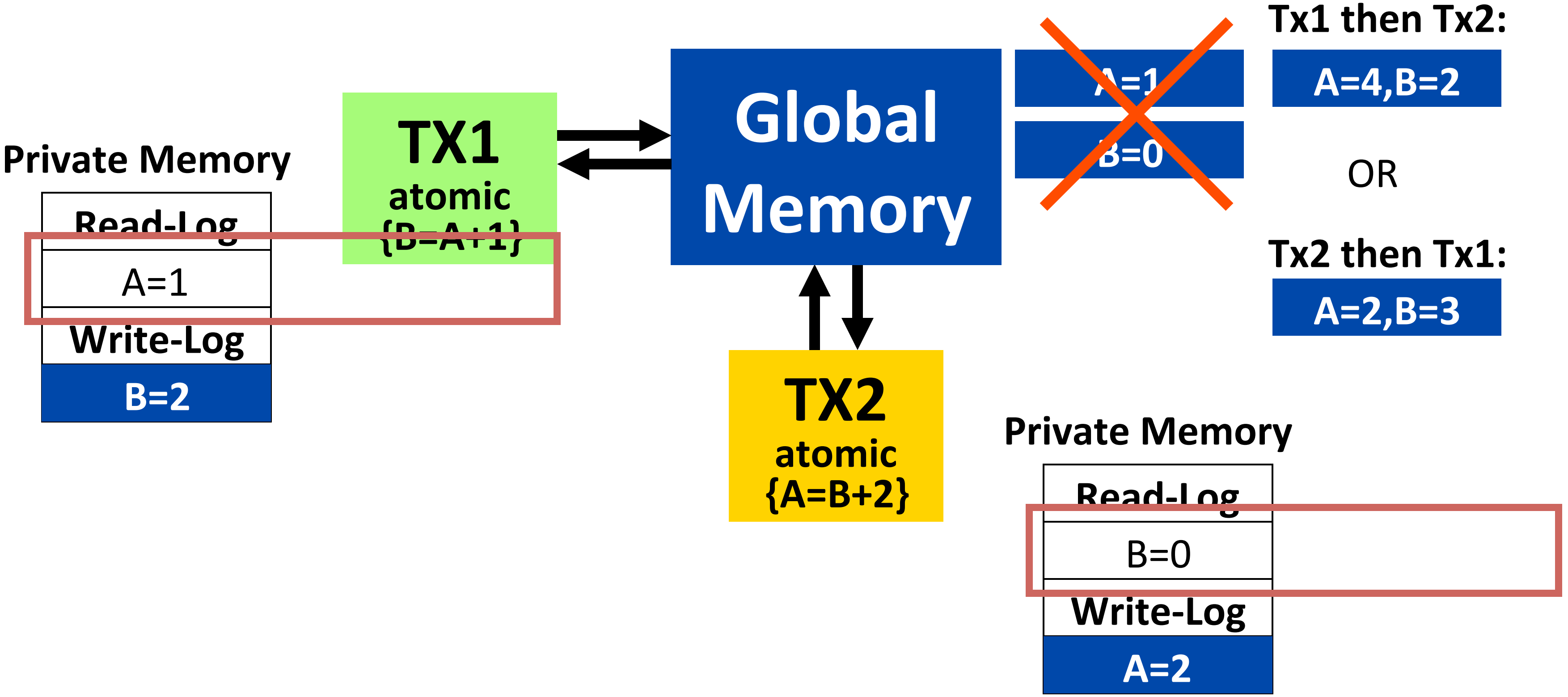
```
TxBegin
LD r1,[A]
ADD r1,r1,1
ST r1,[B]
TxCommit
```

```
TxBegin
LD r2,[B]
ADD r2,r2,2
ST r2,[A]
TxCommit
```

- Self-Validation + Al
  - Only detects **existence** of conflict (not identity)

Parallel Validation?

Data Race!?!



Private Memory

Read-Log

A=1

Write-Log

B=2

TX1

atomic {B=A+1}

Global Memory

~~A=1~~

~~B=0~~

TX2

atomic {A=B+2}

Tx1 then Tx2:

A=4,B=2

OR

Tx2 then Tx1:

A=2,B=3

Private Memory

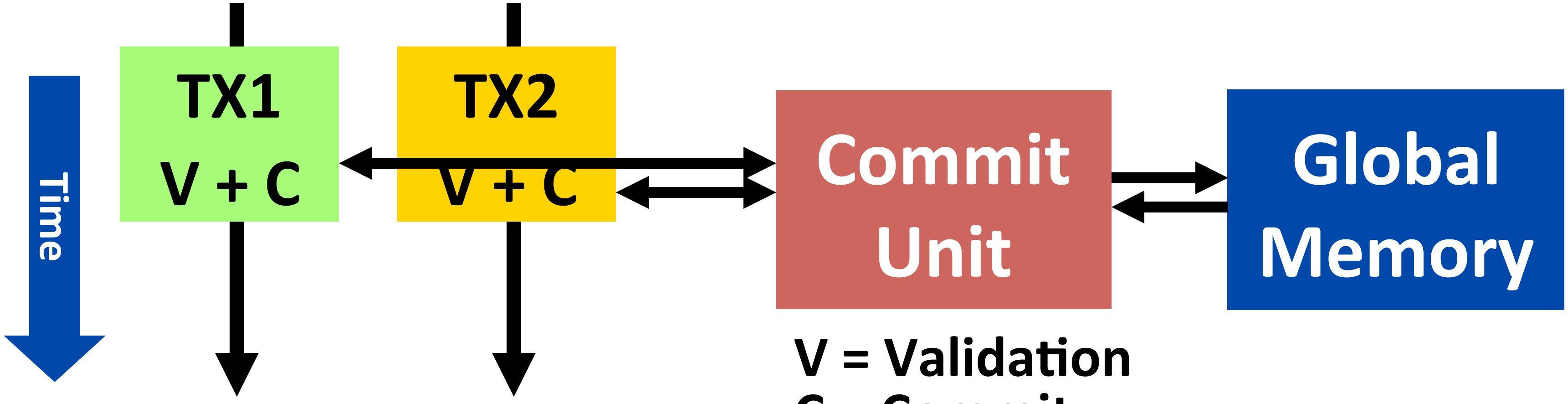
Read-Log

B=0

Write-Log

A=2

Serialize Validation?



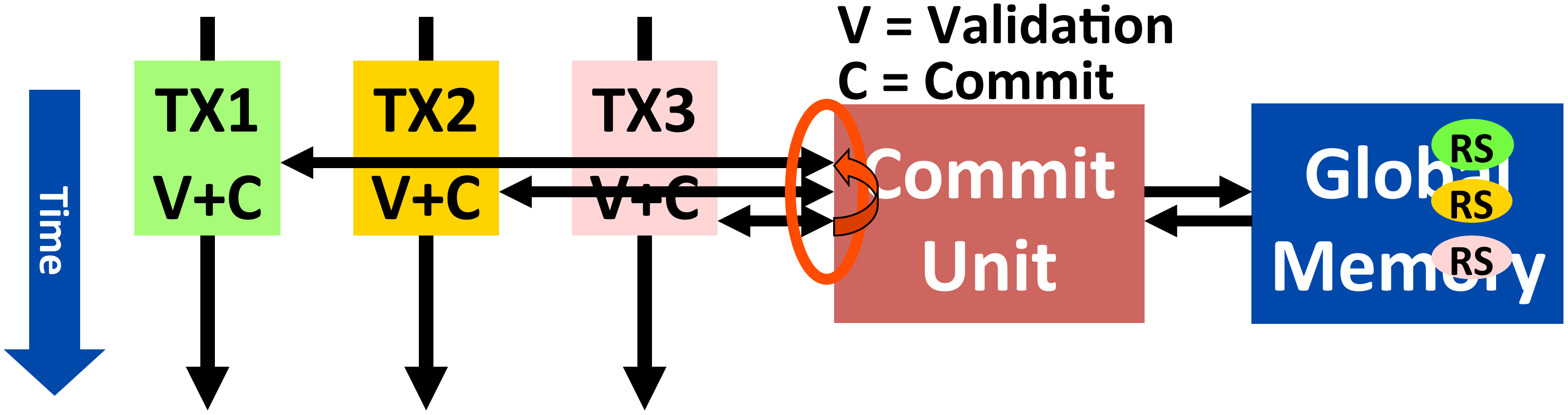
V = Validation
C = Commit

- Benefit #1: No Data Race
- Benefit #2: No Live Lock
- Drawback: Serializes **Non-Conflicting** Transactions (“collateral damage”)

Solution: Speculative Validation

Key Idea: Split Conflict Detection into two parts

1. Recently Committed TX in Parallel
2. Concurrently Committing TX in Commit Order
   - ❑ Approximate



Conflict Rare → Good Commit Parallelism

Efficiency Concerns?

128X
Speedup over CG-Locks

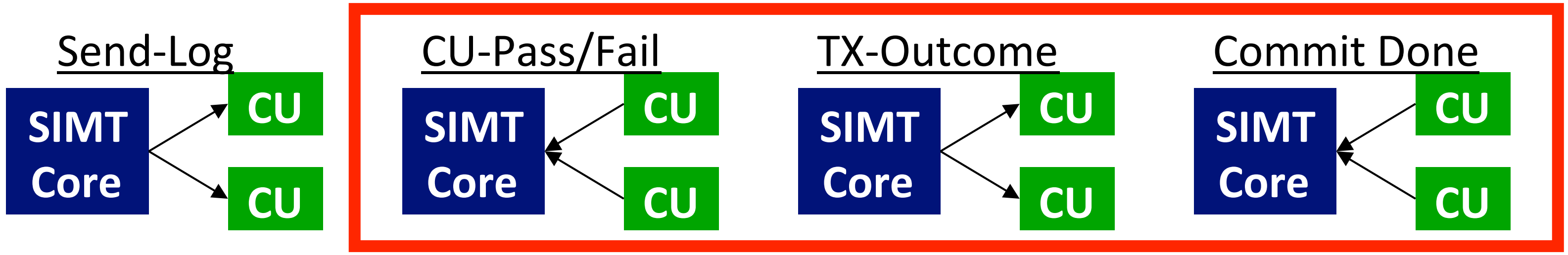
40%
FG-Locks Performance

2X
Energy Usage

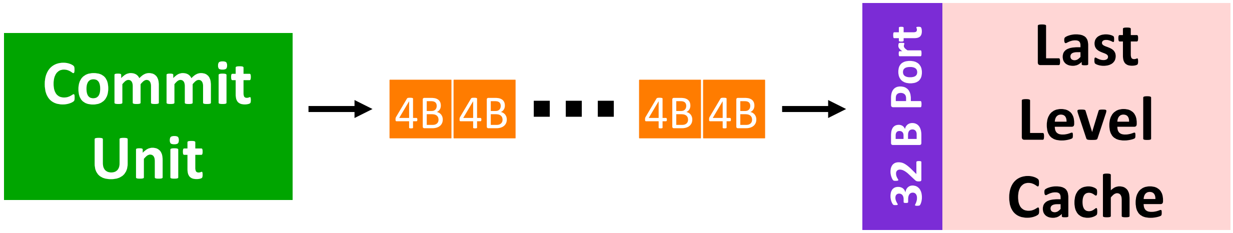
- Scalar Transaction Management
  - Scalar Transaction fits SIMT Model
  - Simple Design
  - Poor Use of SIMD Memory Subsystem
- Rereading every memory location
  - Memory access takes energy

Inefficiency from Scalar Transaction Management

- Kilo TM ignores GPU thread hierarchy
  - Excessive Control Message Traffic



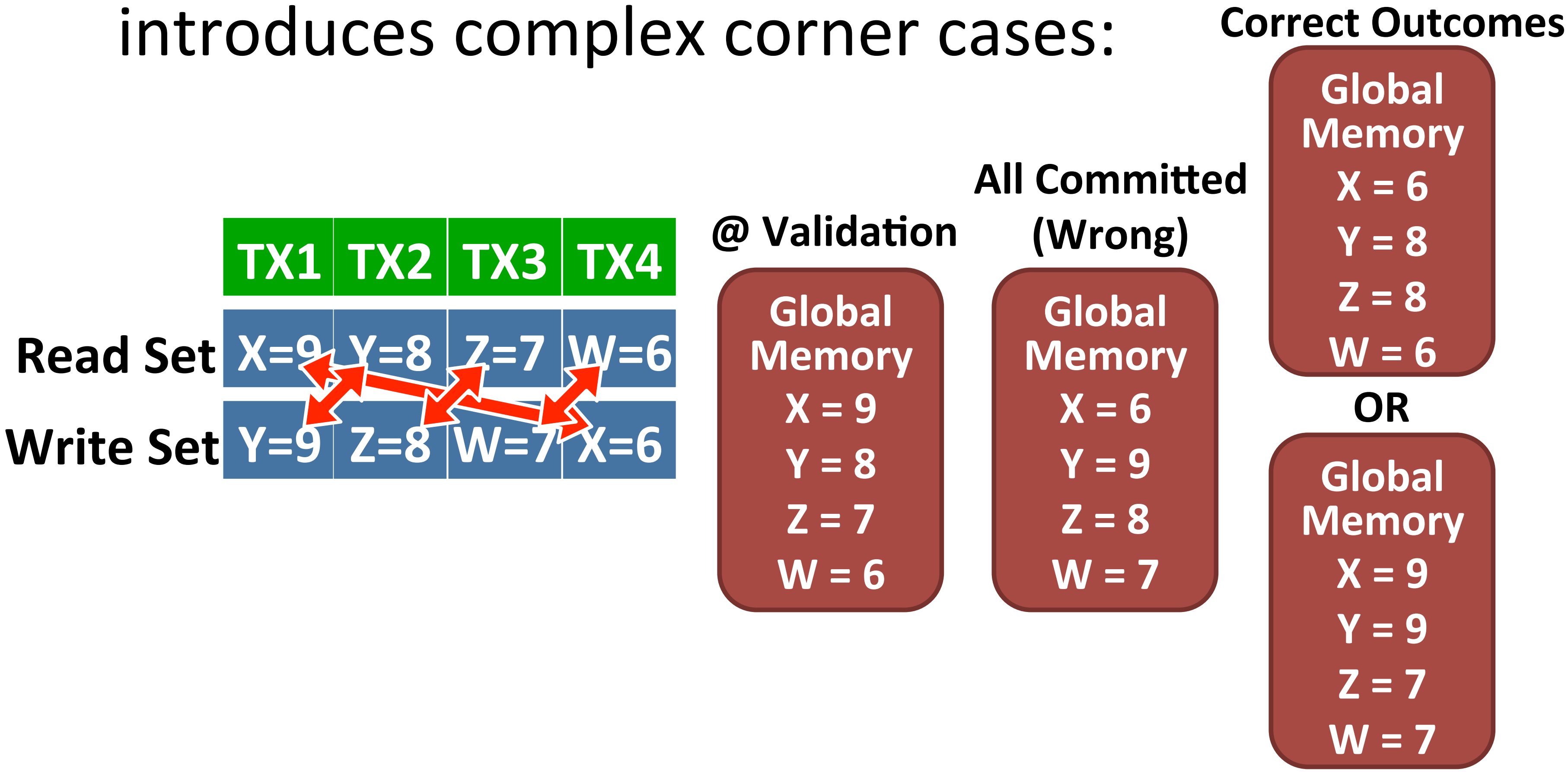
  - Scalar Validation and Commit
    → Poor L2 Bandwidth Utilization



- Simplify HW Design, but **Cost Energy**

Intra-Warp Conflict

- Potential existence of intra-warp conflict introduces complex corner cases:



	TX1	TX2	TX3	TX4
Read Set	X=9	Y=8	Z=7	W=6
Write Set	Y=9	Z=8	W=7	X=6

@ Validation

Global Memory
X = 9
Y = 8
Z = 7
W = 6

All Committed (Wrong)

Global Memory
X = 6
Y = 9
Z = 8
W = 7

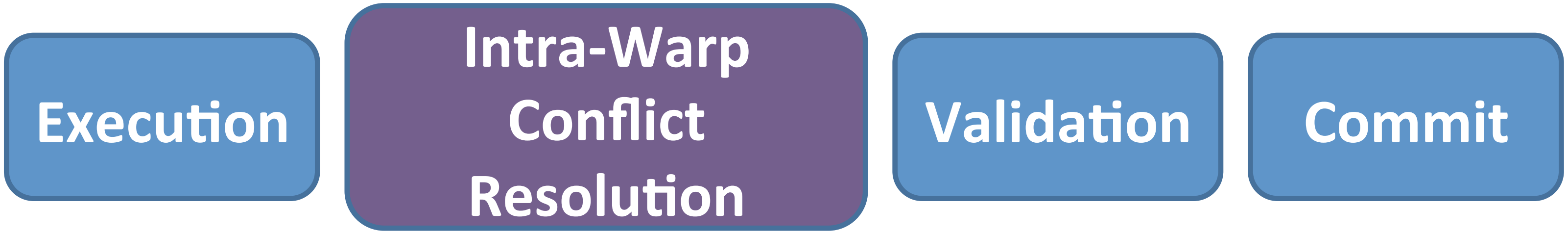
Correct Outcomes

Global Memory
X = 6
Y = 8
Z = 8
W = 6

OR

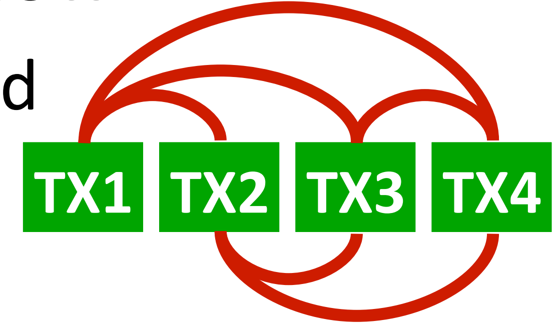
Global Memory
X = 9
Y = 9
Z = 7
W = 7

Intra-Warp Conflict Resolution



- Kilo TM stores read-set and write-set in logs
  - Compact, fits in caches
  - Inefficient for search
- Naive, pair-wise resolution too slow
  - T threads/warp, R+W words/thread
  - $O(T^2 \times (R+W)^2)$, $T \geq 32$

$O((R+W)^2)$ Comparisons Each



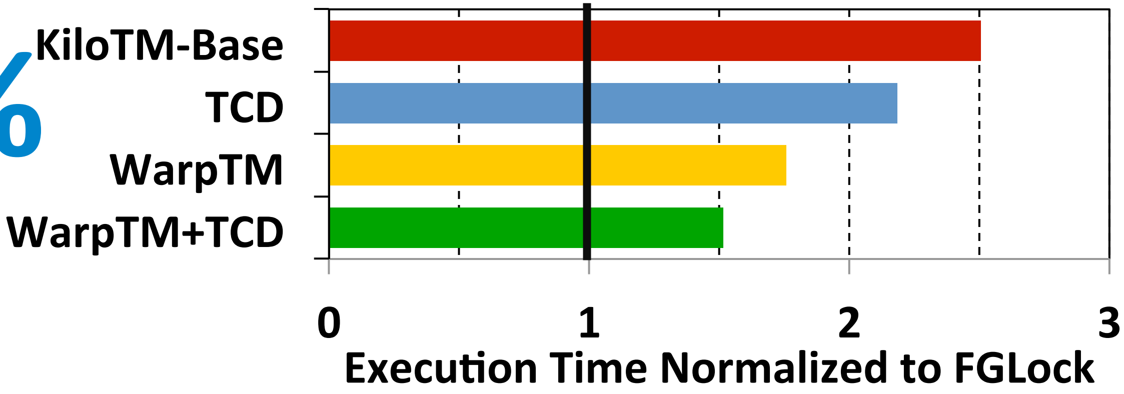
Fung, MICRO 2013
Intra-Warp Conflict Resolution:
2-Phase Parallel Conflict Resolution

- Insight: Fixed priority for conflict resolution enables parallel resolution
- O(R+W)
- Two Phases
  - Ownership Table Construction
  - Parallel Match

Results

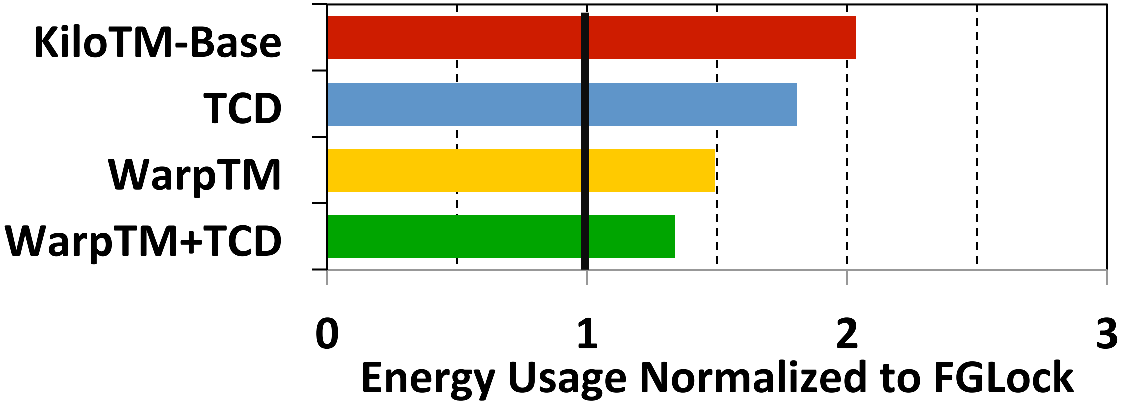
40%→66%

FG-Lock Performance



2X→1.3X

Energy Usage

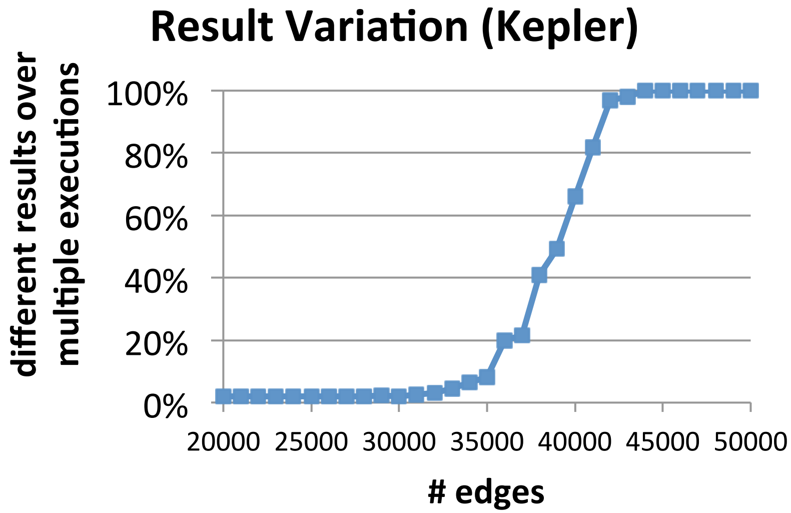


Low Contention Workload:

Kilo TM w/ SW Optimizations on par with FG Lock

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