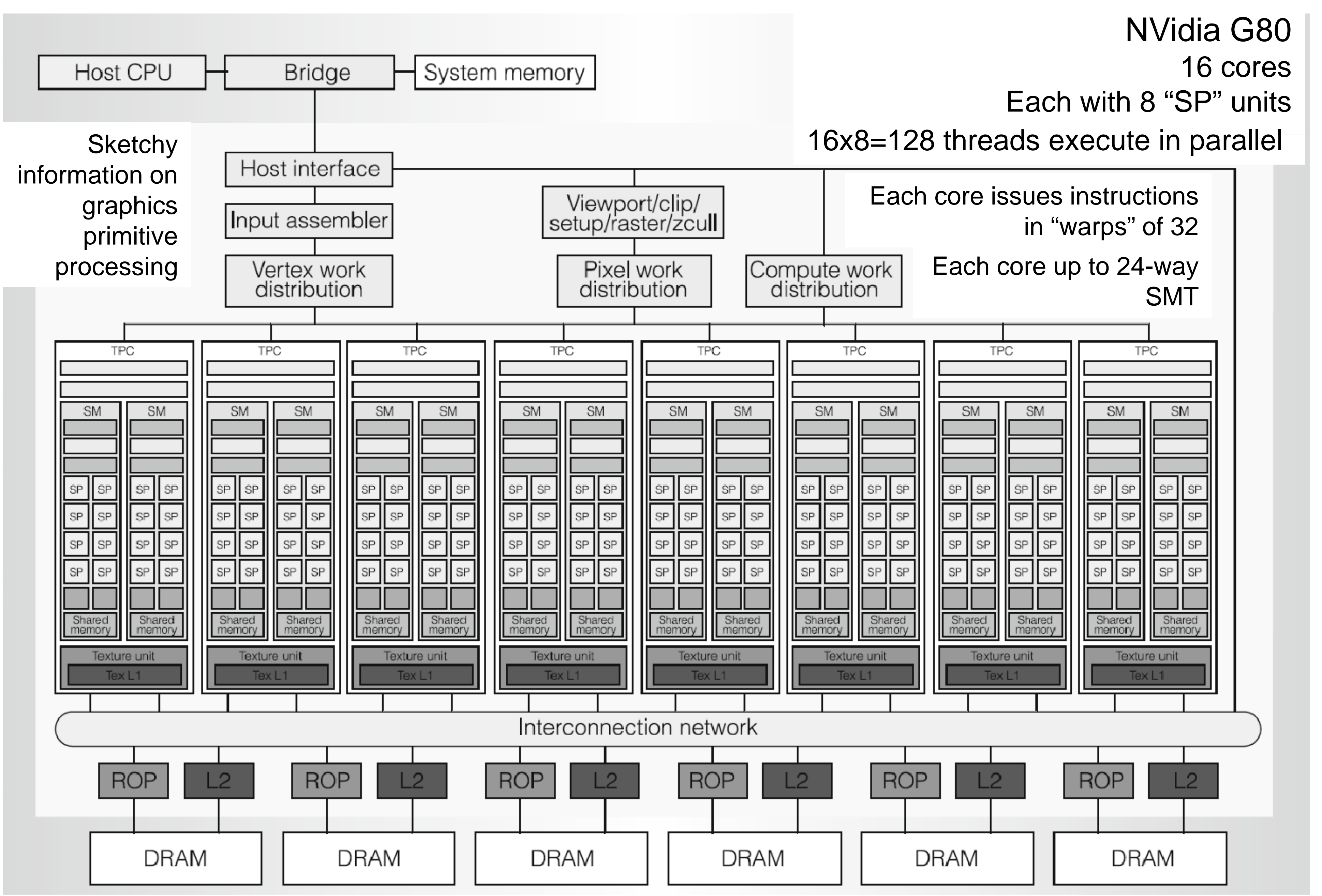
Graphics and manycore architectures

Advanced Computer Architecture, 332, Chapter 6 (March 2010)

Lecture notes by Anton Lokhmotov (ex-Imperial, now with ARM), Paul H J Kelly (Imperial), with slides from Lee Howes (AMD) and others

Graphics Processors (GPUs)

- Much of our attention so far has been devoted to making a single core run a single thread faster
- If your workload consists of thousands of threads, *everything* looks different:
  - *Never* speculate: there is always another thread waiting with work you *know* you have to do
  - No speculative branch execution, no branch prediction
  - Caches are for bandwidth, not latency: you can use SMT to hide cache access latency
  - Control is at a premium:
    - How to launch >10,000 threads?
    - What if they branch in different directions?
    - What if they access random memory blocks/banks?
- This is the "manycore" world
- Driven by the gaming market – but with many other applications



No L2 cache coherency problem, since data can be in only one cache

All caches are small

Texture cache does interpolation

ROP does Z-buffering & alpha blending

NVIDIA TESLA: A UNIFIED GRAPHICS AND COMPUTING ARCHITECTURE; Erik Lindholm John Nickolls, Stuart Oberman, John Montrym (IEEE Micro, March-April 2008)

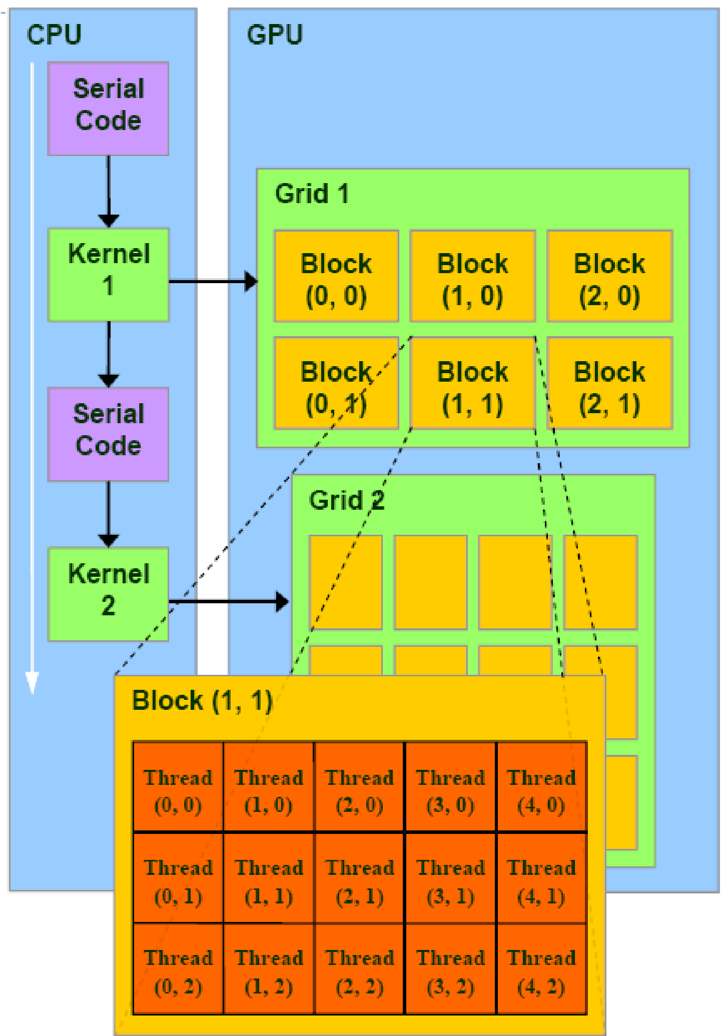
NVidia's TESLA microarchitecture

- Designed to do rendering
- Designed to do general-purpose computing
  - But to manage thousands of threads, a new programming model is needed, called CUDA
  - CUDA is proprietary, but essentially the same model lies behind OpenCL, an open standard with implementations for multiple vendors' GPUs

- GPU evolved from hardware designed specifically around the OpenGL/DirectX rendering pipeline, with separate vertex- and pixel-shader stages
- "Unified" architecture arose from increased sophistication of shader programs
- TESLA still has some features specific to graphics:
  - Work distribution, load distribution
  - Texture cache, pixel interpolation
  - Z-buffering and alpha-blending (the ROP units, see overleaf)

CUDA Execution Model

- CUDA is a C extension
  - Serial CPU code
  - Parallel GPU code (kernels)
- GPU kernel is a C function
  - Each *thread* executes kernel code
  - A group of threads forms a *thread block* (1D, 2D or 3D)
  - Thread blocks are organised into a *grid* (1D or 2D)
  - Threads within the same thread block can synchronise execution, and share access to local scratchpad memory



Key idea: hierarchy of parallelism, to handle *thousands* of threads

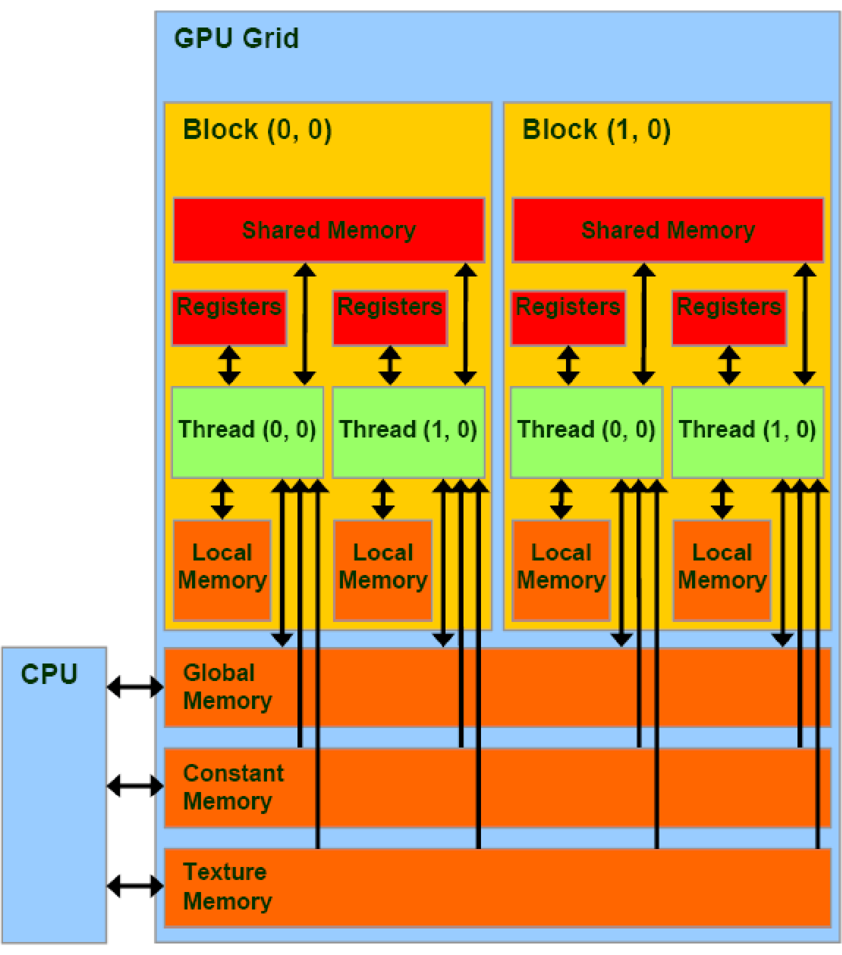
Example: Matrix addition

```
__global__ void matAdd(float A[N][N], float B[N][N], float C[N][N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   if (i < N && j < N)
      C[i][j] = A[i][j] + B[i][j];
}

int main()
{
   // Kernel setup
   dim3 blockDim(16, 16);
   dim3 gridDim((N + blockDim.x – 1) / blockDim.x,
                (N + blockDim.y – 1) / blockDim.y);
   // Kernel invocation
   matAdd<<<gridDim, blockDim>>>(A, B, C);
```

- **Kernel invocation ("<<<…>>>") corresponds to enclosing loop nest, managed by hardware**
- **Explicitly split into 2-level hierarchy: blocks (which share "shared" memory), and grid**
- **Kernel commonly consists of just one iteration but could be a loop**
- **Multiple tuning parameters trade off register pressure, shared-memory capacity and parallelism**

- Local memory – private to each thread (slow if off-chip, fast if register allocated)
- Shared memory – shared between threads in a thread block (fast on-chip)
- Global memory – shared between thread blocks in a grid (off-chip DRAM)
- Constant memory (small, read-only)
- Texture memory (read-only; cached, stored in Global memory)



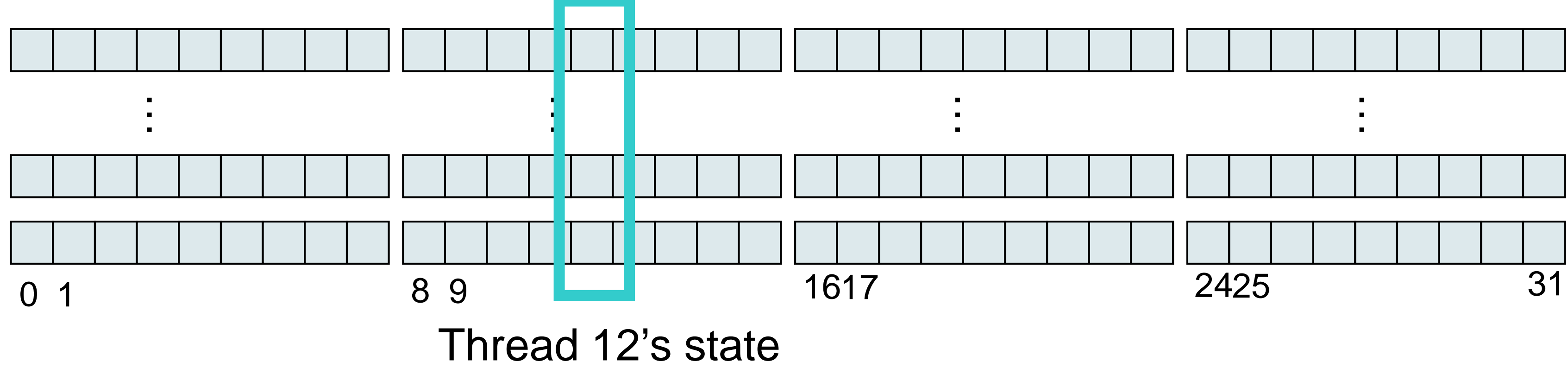
- This diagram is misleading: logical association but not hardware locality
- "Local memory" is non-cached (in Tesla), stored in global DRAM
- Critical thing is that "shared" memory is shared among all threads in a block, since they all run on the same SM

Mapping from CUDA to TESLA

- Array of streaming multiprocessors (SMs)
  - (we might call them "cores", when comparing to conventional multicore; each SM is an instruction-fetch-execution engine)
- CUDA thread blocks get mapped to SMs
- SMs have thread processors, private registers, shared memory, etc.
- Each SM executes a pool of "warps", with a separate instruction pointer for each warp. Instructions are issued from each ready-to-run warp in turn (SMT, hyperthreading)

SIMT: Single-Instruction, Multiple-Thread

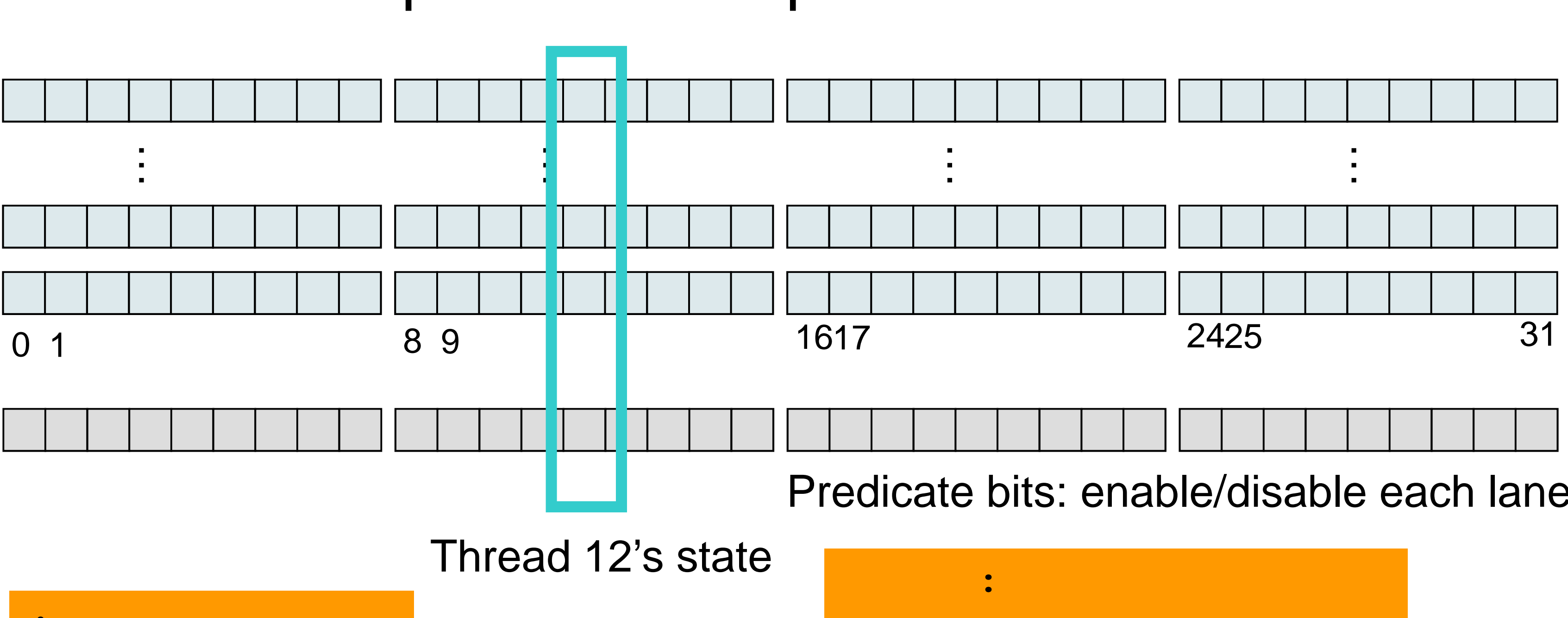
- SIMT: each warp instruction operates on 32 data items:

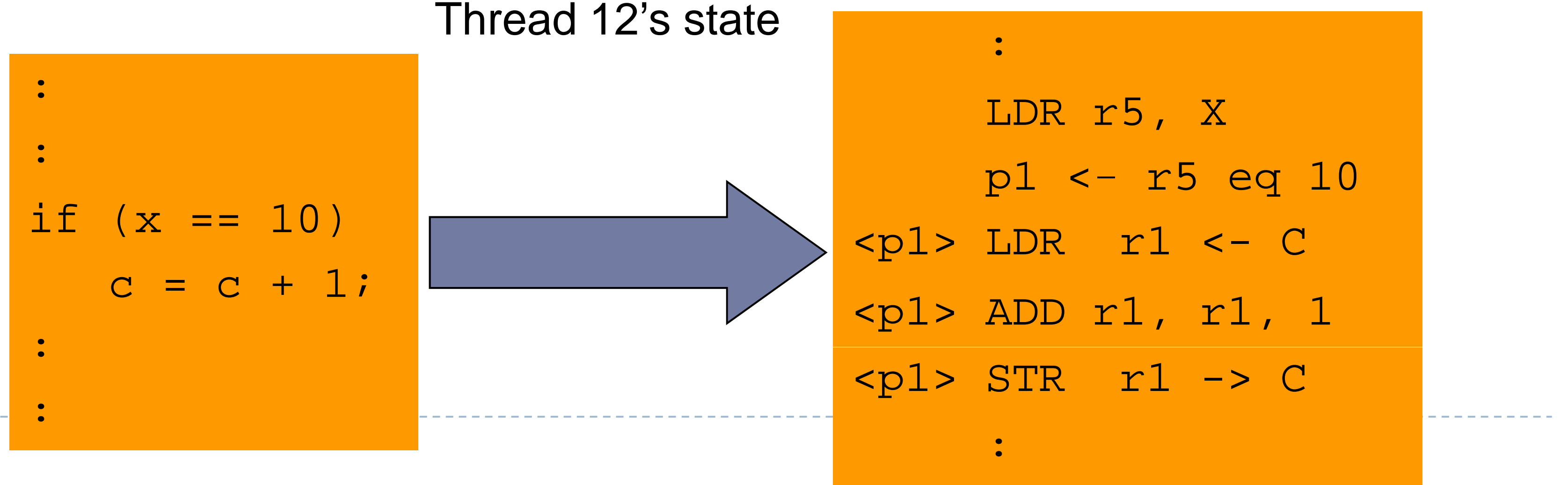


- Each thread's state is represented by a "lane" of the SIMD register set
- ((Each 32-wide warp instruction is actually executed in 4 cycles using the 8 SPs in the SM))
- CUDA programmers write code for each thread as if it runs independently – but actually, instructions are broadcast to 32 threads in parallel

SIMT: Single-Instruction, Multiple-Thread

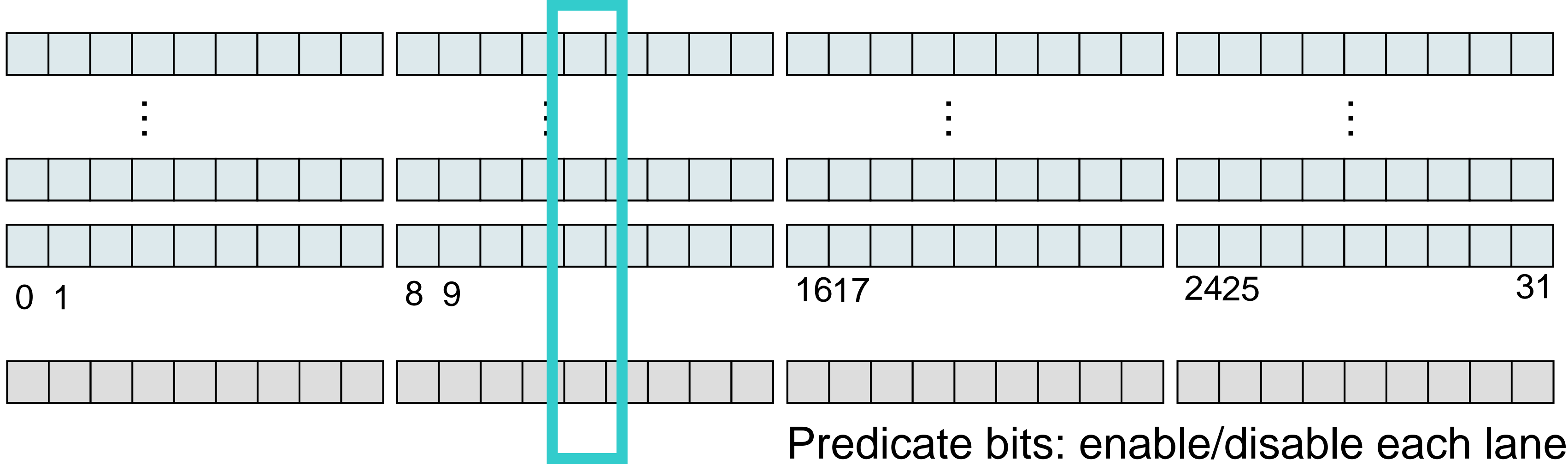
- SIMT: each warp instruction operates on 32 data items:



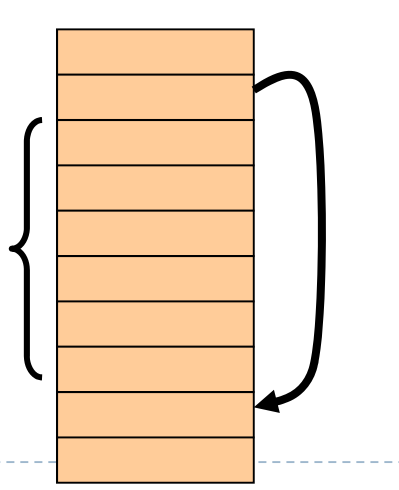


SIMT: Single-Instruction, Multiple-Thread

- SIMT: each warp instruction operates on 32 data items:

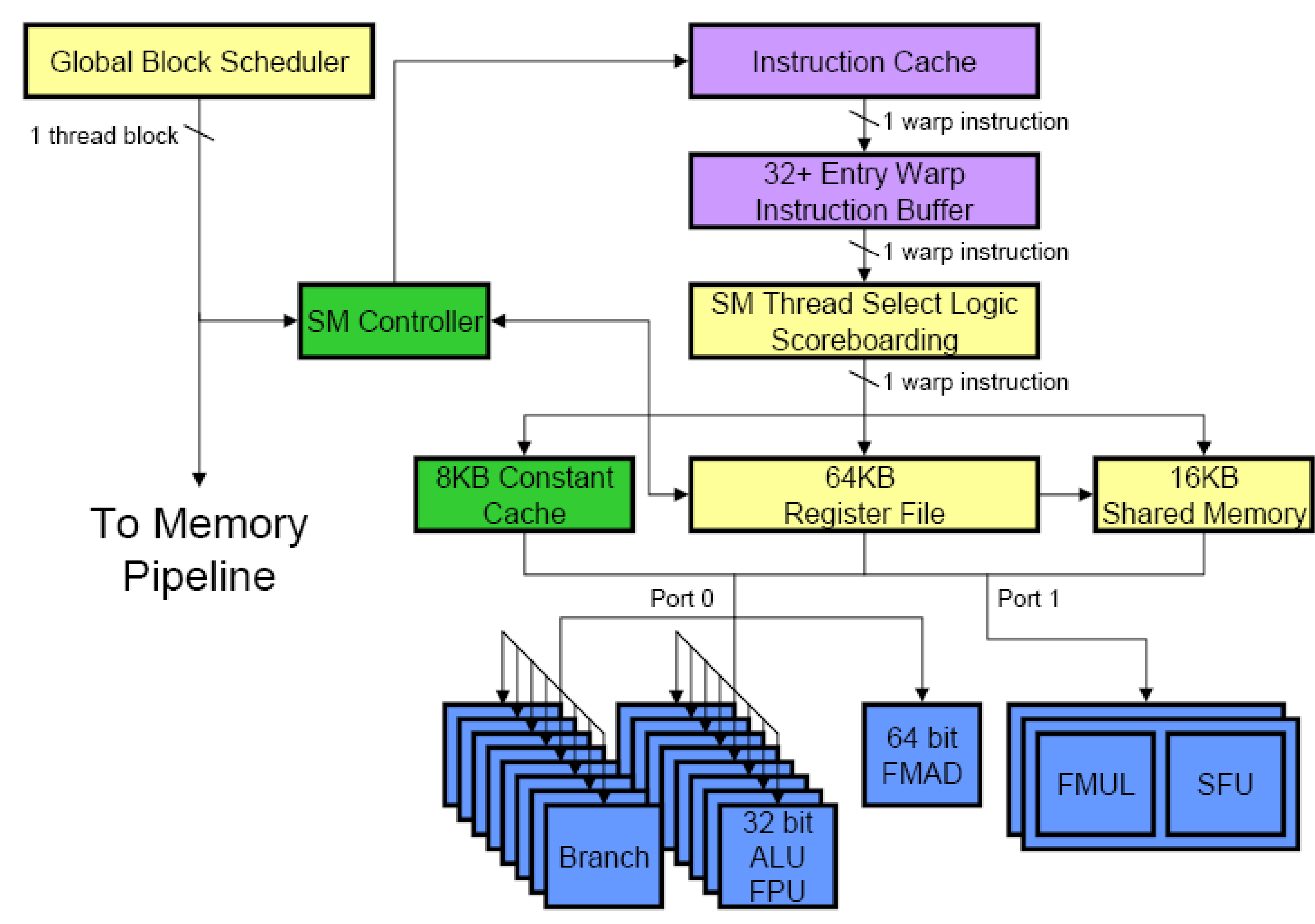


- We can jump
  - If the whole warp's predicate bits are the same
  - and the branch skips enough (7?) instructions



- ***Control-flow coherence***: every thread goes the same way (a form of locality)

GT200 streaming multiprocessor



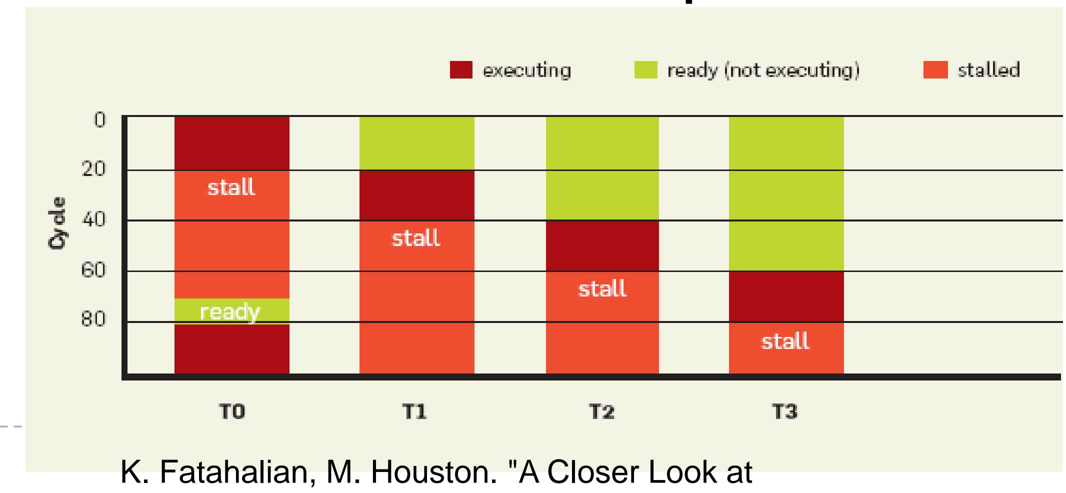
http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242

GT200 streaming multiprocessor

- Multithreaded instruction fetch and issue unit
- 8 streaming processors (SP)
  - 32-bit ALU and FPU (e.g. bitwise, min, max; FP add, mul, mad)
  - Branch unit (e.g. cmp)
- 2 special-function units (SFU)
  - Transcendental functions (e.g. reciprocal, sine), interpolation
  - Interpolation hardware can be used as 4 FP multipliers
- 16KB shared memory (as fast as registers, if no conflicts)
- 64-bit FP and integer unit (not in G80)
- 16K 32-bit registers (only 8K in G80)

SM multithreaded instruction unit

- Instruction unit creates, manages, schedules and executes threads in groups of 32 threads called *warps*
- Warp instructions are fetched into an instruction buffer
- Scoreboard qualifies warp instructions for issue
- Scheduler prioritises ready to issue warp instructions based on their type and "fairness"
- Scheduler issues a warp instruction with highest priority

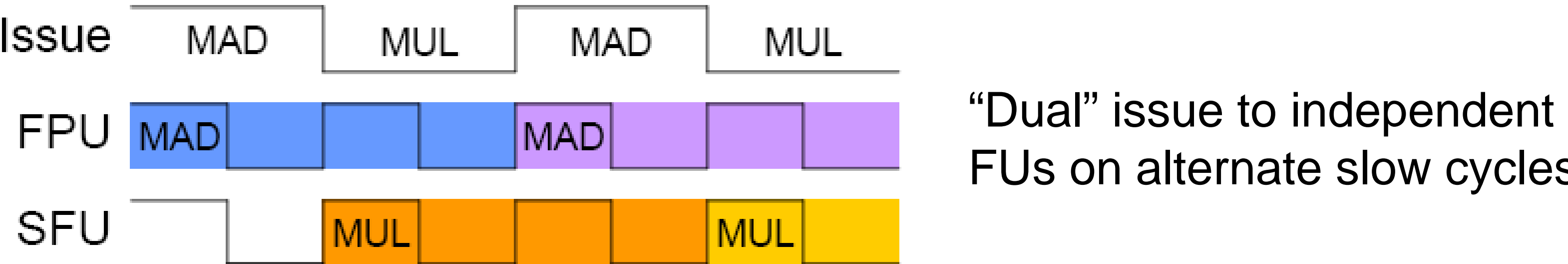


K. Fatahalian, M. Houston. "A Closer Look at GPUs," *Commun. ACM*, 51:10, pp. 50-57.

A hypothetical GPU:
- Four warp contexts T0–T3
- 20 cycle compute
- 50 cycle memory access

SM instruction issue

- To optimise for power and performance, GPU units are in multiple clock domains
  - Core clock (e.g. 600 MHz) – "slow" cycles
  - FUs clock (e.g. 1300 MHz) – "fast" cycles
  - Memory clock (e.g. 1100 MHz)
- Issue logic can issue every slow cycle (every 2 fast cycles)
- FUs can typically be issued instructions every 4 fast cycles (1 cycle per 8 threads from a warp)



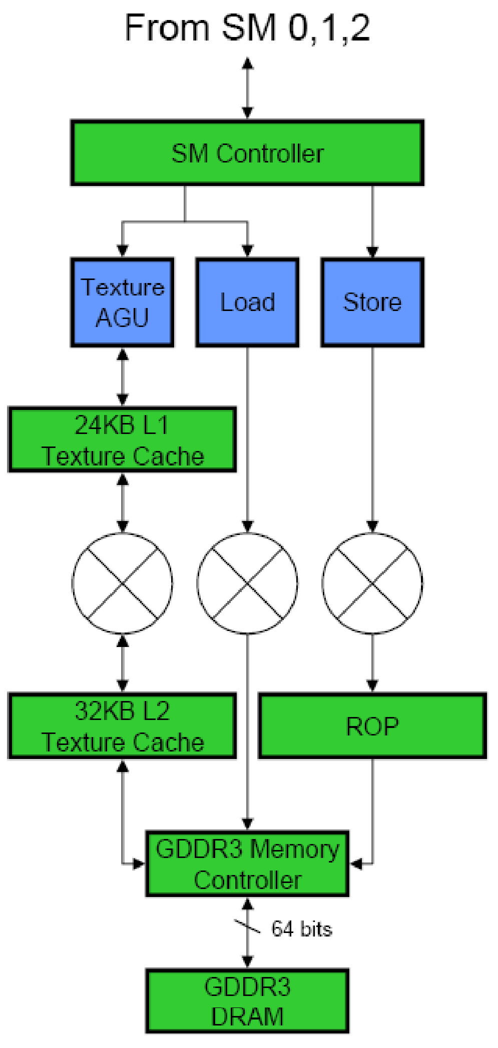
"Dual" issue to independent FUs on alternate slow cycles

SM branch management

- Threads in a warp start executing from the same address
- A warp that executes a branch instruction waits until the target address for every thread in the warp is computed
- Threads in a warp can take the same branch path ☺
- Threads in in warp can diverge to different paths
  - the warp serially executes each path, disabling threads that are not on that path;
  - when all paths complete, the threads reconverge
- Divergence only occurs within a warp – different warps execute independently
- Minimising divergence is important for performance, but not correctness (cf. cache lines)

TPC memory pipeline

- Load and store instructions are generated in the SMs
  - Address calculation (register + offset)
  - Virtual to physical address translation
- Issued a warp at a time – executed in half-warp groups (i.e. 16 accesses at a time)
- Memory coalescing and alignment
  - Threads with adjacent indices should access adjacent memory locations (i.e. thread K should access Kth data word)
  - Accesses should be aligned for half-words



http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242

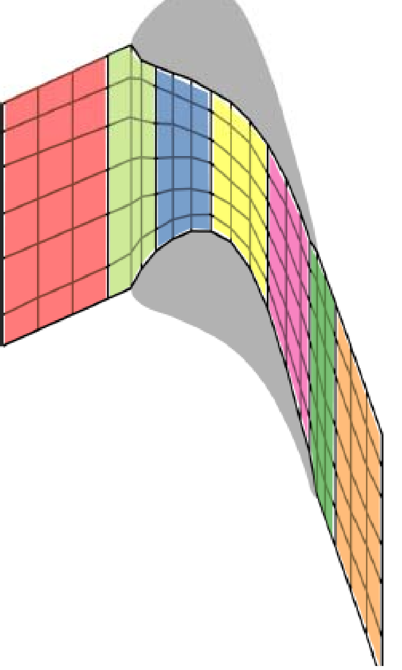
CUDA example

- Tobias Brandvik and Graham Pullan's automatic program generator for jet-engine fluid dynamics

- http://www.industrialmath.net/CUDA09_talks/pullan.pdf

Structured grid strategy

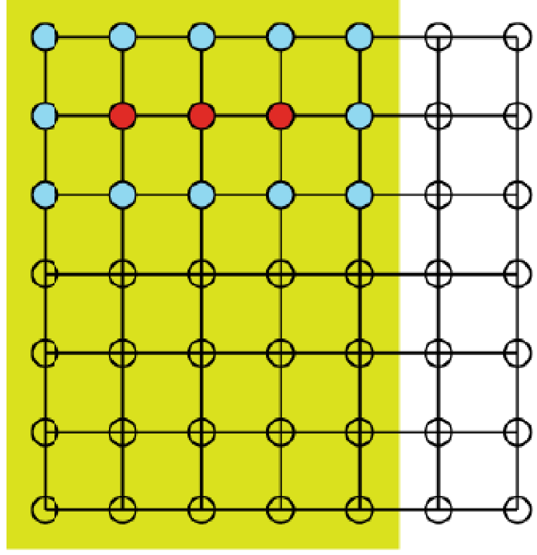
- Divide up domain
  - each sub-domain to a thread block
  - update nodes in sub-domain with most efficient stencil operation we can come up with (make effective use of shared mem)



- For each block, start a plane of threads (an i-k plane)
- Load three planes into shared memory
  - Compute one plane
- Load next plane into shared memory (swap out first plane)
  - Compute next plane
- Repeat, moving along j direction

- Compute on structured 3D mesh
- Update each element using data from its neighbours
- Use CUDA shared memory to buffer neighbour data

CUDA strategy



Zone solved by Block 1

1. Load 2 rows into shared mem
2. To compute points, load next row into shared mem
3. Move up domain, row by row (load new row into shared mem, drop lowest row out of shared mem)

- Allocate (small!) 3D buffer in shared memory

CUDA code (nearest neighbour stencil)

```
__global__ void stencil_kernel(float sf, float *a_data, float *b_data){
__shared__ float a[16][3][5];            declare shared memory array
i = (int) threadIdx.x;
k = (int) threadIdx.y;
a[i][0][k] = a_data[i0m10];
a[i][1][k] = a_data[i000];
/* begin loop in j-direction */
a[i][2][k] = a_data[i0p10];
__syncthreads();
/* compute */
b_data[i000] =
   sf1*a[i][1][k] + sfd6*(a[im1][1][k] +
   a[ip1][1][k] + a[i][0][k] +
   a[i][2][k] + a[i][1][km1] + a[i][1][kp1])
/* repeat: load j-plane, syncthreads, compute...*/
```

- Find out which iteration this thread is doing

CUDA code (nearest neighbour stencil)

```
__global__ void stencil_kernel(float sf, float *a_data, float *b_data){
__shared__ float a[16][3][5];
i = (int) threadIdx.x;
k = (int) threadIdx.y;
a[i][0][k] = a_data[i0m10];
a[i][1][k] = a_data[i000];
/* begin loop in j-direction */
a[i][2][k] = a_data[i0p10];
__syncthreads();
/* compute */
b_data[i000] =
   sf1*a[i][1][k] + sfd6*(a[im1][1][k] +
   a[ip1][1][k] + a[i][0][k] +
   a[i][2][k] + a[i][1][km1] + a[i][1][kp1])
/* repeat: load j-plane, syncthreads, compute...*/
```

get i,k thread indices

- This thread loads its share of the block into shared memory

CUDA code (nearest neighbour stencil)

```
__global__ void stencil_kernel(float sf, float *a_data, float *b_data){
__shared__ float a[16][3][5];
i = (int) threadIdx.x;
k = (int) threadIdx.y;
a[i][0][k] = a_data[i0m10];
a[i][1][k] = a_data[i000];
/* begin loop in j-direction */
a[i][2][k] = a_data[i0p10];
__syncthreads();
/* compute */
b_data[i000] =
   sf1*a[i][1][k] + sfd6*(a[im1][1][k] +
   a[ip1][1][k] + a[i][0][k] +
   a[i][2][k] + a[i][1][km1] + a[i][1][kp1])
/* repeat: load j-plane, syncthreads, compute...*/
```

load initial 2 planes

- Each thread executes a loop
- Two phases:
  - Load next plane
  - Compute
- "syncthreads()" is needed to ensure all threads in this block have done phase, before moving on

CUDA code (nearest neighbour stencil)

```
__global__ void stencil_kernel(float sf, float *a_data, float *b_data){
__shared__ float a[16][3][5];
i = (int) threadIdx.x;
k = (int) threadIdx.y;
a[i][0][k] = a_data[i0m10];
a[i][1][k] = a_data[i000];
/* begin loop in j-direction */
a[i][2][k] = a_data[i0p10];
__syncthreads();
/* compute */
b_data[i000] =
   sf1*a[i][1][k] + sfd6*(a[im1][1][k] +
   a[ip1][1][k] + a[i][0][k] +
   a[i][2][k] + a[i][1][km1] + a[i][1][kp1])
/* repeat: load j-plane, syncthreads, compute...*/
```

main loop:
load next plane
syncthreads (whole plane loaded)

compute result (if not a halo node)

UNIVERSITY OF CAMBRIDGE

- Common pattern:
  - Threads cooperate to load data into shared memory in parallel
  - Syncthreads()
  - Use it, and repeat

GT200

- 10 Thread Processing Clusters (TPCs)
- 3 Streaming Multiprocessors (SMs) per TPC
- 8 32-bit FPU per SM
- 1 64-bit FPU per SM
- 16K 32-bit registers per SM
- Up to 1024 threads / 32 warps per SM
- 16KB Shared memory per SM
- 8 64-bit memory controllers (512-bit wide memory interface)

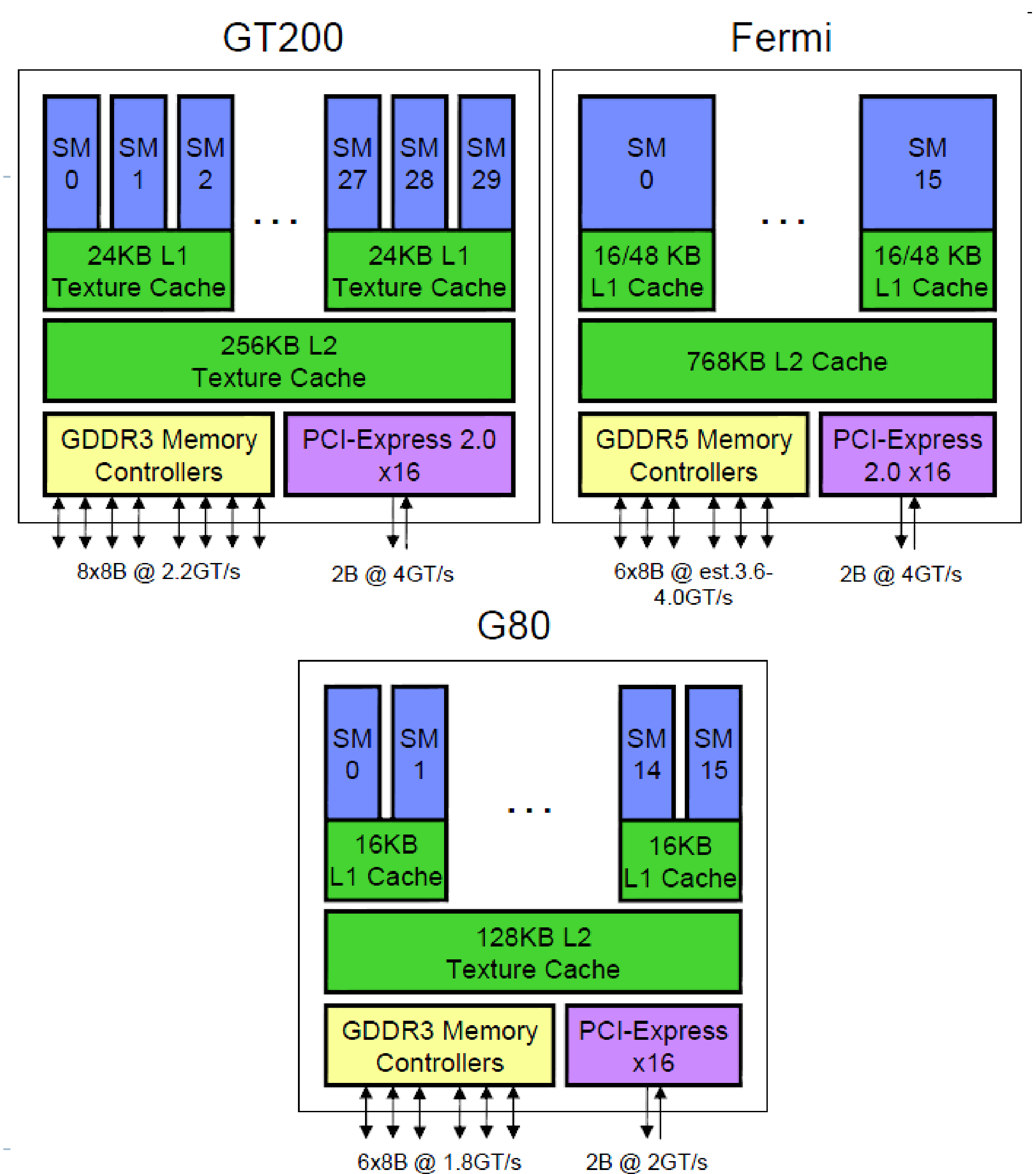
G80

- 8 Thread Processing Clusters (TPCs)
- 2 Streaming Multiprocessors (SMs) per TPC
- 8 32-bit FPU per SM
- 8K 32-bit registers per SM
- Up to 768 threads / 24 warps per SM
- 16KB Shared memory per SM
- 6 64-bit memory controllers (384-bit wide memory interface)

http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242

NVidia's Fermi

- 2010-generation GPU specifically targetted at general-purpose compute market
- Read-write L1 caches (for register spilling)
- Configurable L1/scratchpad 16K+48K
- Larger L2 cache
- No ROP units (?)



http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242

NVidia's Fermi vs Tesla/GT200

Fermi Core

Instruction Cache

Warp Instruction Queues | Warp Instruction Queues

1 warp instruction

Scoreboarded Warp Scheduler | Scoreboarded Warp Scheduler

1 warp instruction

128KB Register File

Port 0 | Port 1

32 bit ALU [16] | 32 bit FPU [16] | SFU [4] | 32 bit FPU [16] | 32 bit ALU [16]

64KB L1D, Shared Memory [16]

GT200 Core

Instruction Cache

Warp Instruction Queues

1 warp instruction

Scoreboarded Warp Scheduler

1 warp instruction

64KB Register File | 16KB Shared Memory

SM Controller

32 bit ALU FPU [8] | Branch Unit [8] | 64 bit FPU | SFU [2]

From GT200 Core 0,1,2

SM Controller

64 bit AGU [16] | Load | Store

24KB L1 Texture Cache

On-chip Network

32KB L2 Texture Cache | ROP

GDDR3 Memory Controller

64 bits

GDDR3 DRAMs

Fermi Core 0

64 bit AGU [16] | Shared Memory Access

DTLB

64KB L1D Cache

On-chip Network

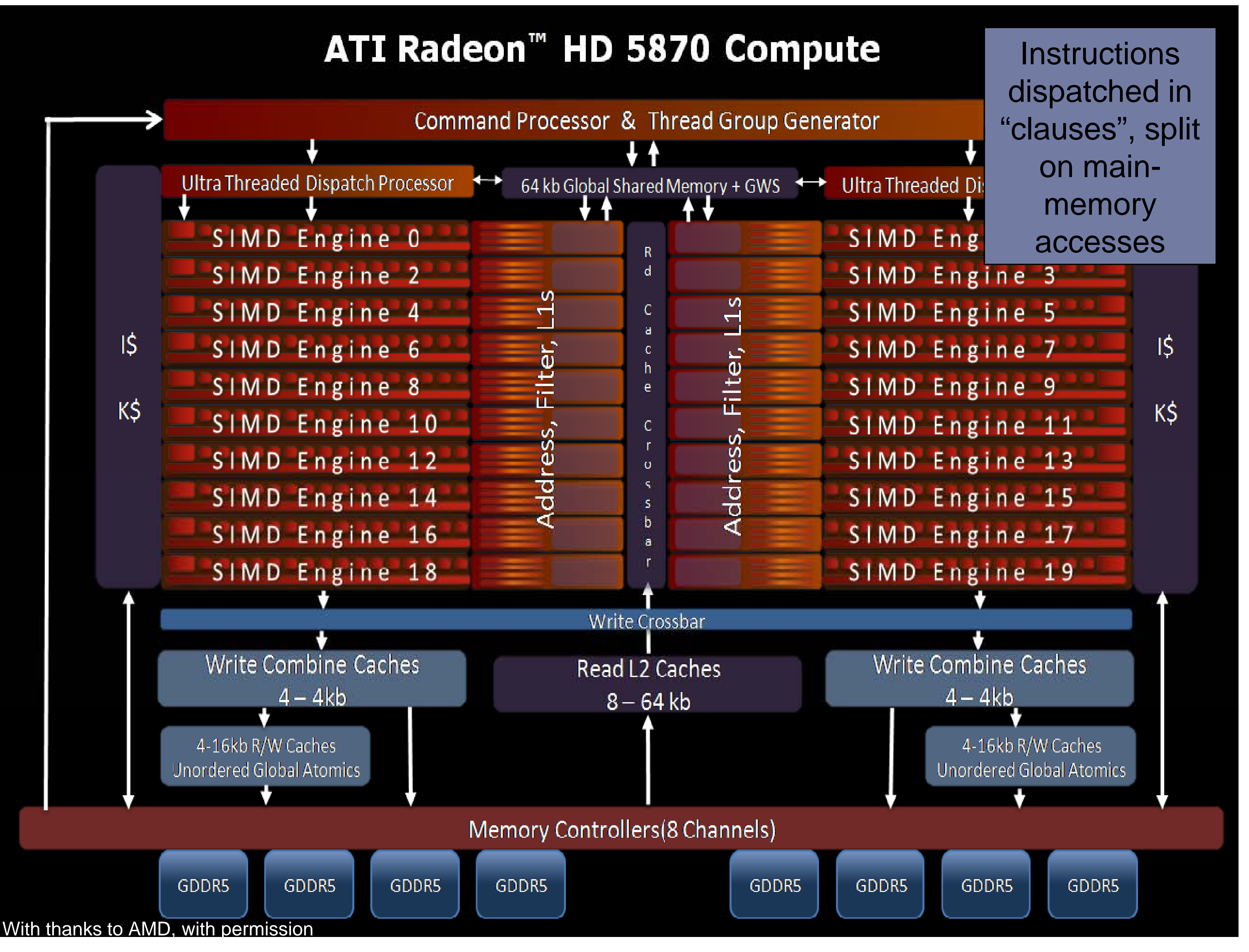
128KB L2 Cache

GDDR5 Memory Controller

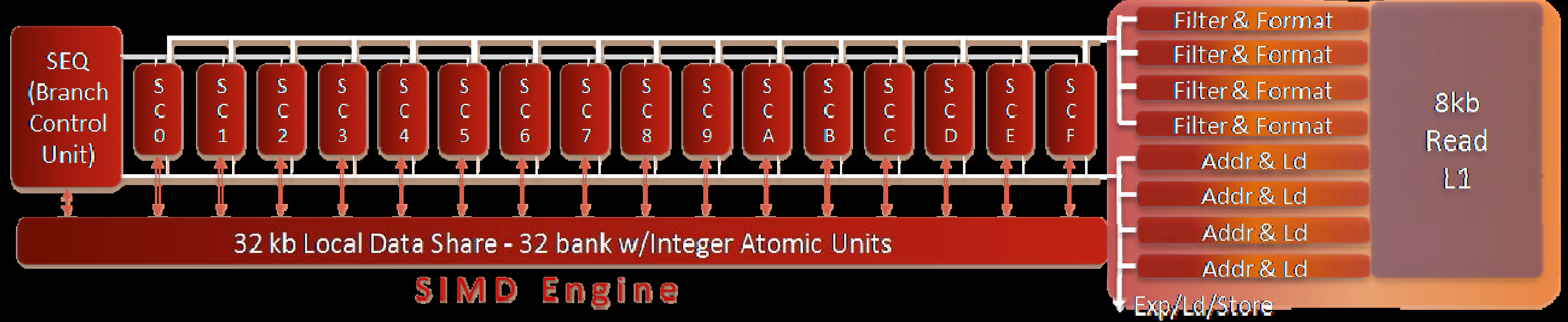
64 bits

GDDR5 DRAMs

http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242


ATI Radeon™ HD 5870 Compute
Instructions dispatched in "clauses", split on main-memory accesses
Command Processor & Thread Group Generator
Ultra Threaded Dispatch Processor
64 kb Global Shared Memory + GWS
Ultra Threaded Di
I$
K$
SIMD Engine 0
SIMD Engine 2
SIMD Engine 4
SIMD Engine 6
SIMD Engine 8
SIMD Engine 10
SIMD Engine 12
SIMD Engine 14
SIMD Engine 16
SIMD Engine 18
Address, Filter, L1s
Rd Cache Crossbar
Address, Filter, L1s
SIMD Eng
SIMD Engine 3
SIMD Engine 5
SIMD Engine 7
SIMD Engine 9
SIMD Engine 11
SIMD Engine 13
SIMD Engine 15
SIMD Engine 17
SIMD Engine 19
I$
K$
Write Crossbar
Write Combine Caches
4 – 4kb
Read L2 Caches
8 – 64 kb
Write Combine Caches
4 – 4kb
4-16kb R/W Caches
Unordered Global Atomics
4-16kb R/W Caches
Unordered Global Atomics
Memory Controllers(8 Channels)
GDDR5
GDDR5
GDDR5
GDDR5
GDDR5
GDDR5
GDDR5
GDDR5
With thanks to AMD, with permission

SIMD Engine

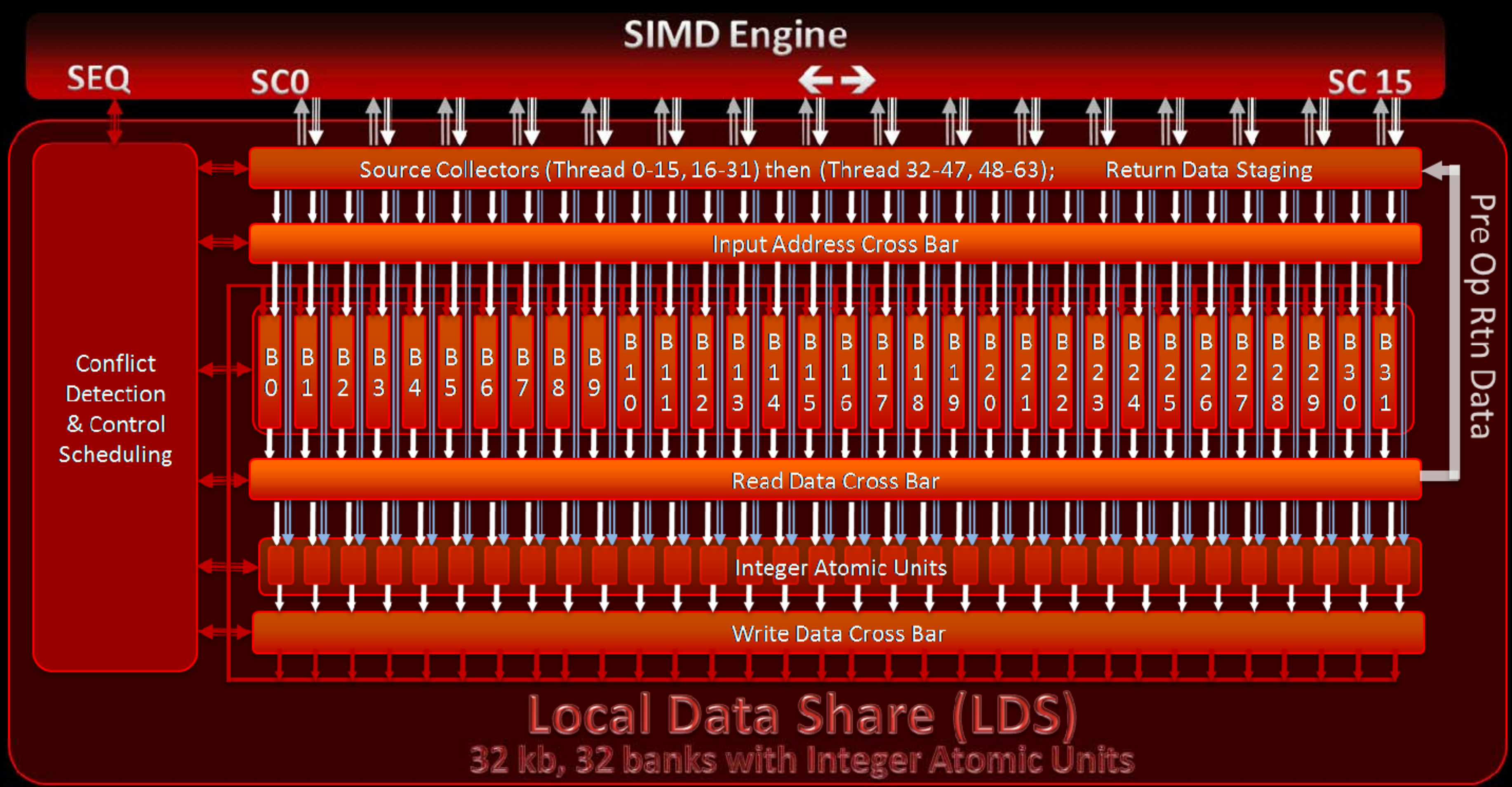


"Warps"

- SIMD Engine can process Wavefronts from multiple kernels concurrently
- Thread divergence within a Wavefront is enabled with Lane Masking and Branching
  - Enabling each Thread in a Wavefront to traverse a unique program execution path
- Full hardware barrier support for up to 8 Work Groups per SIMD Engine (for thread data sharing)
- Each Stream Core receives up to the following per VLIW instruction issue
  - 5 unique ALU Ops - or - 4 unique ALU Ops with a LDS Op (Up to 3 operands per thread)
- LDS and Global Memory access for byte, ubyte, short, ushort reads/writes supported at 32bit dword rates
- Private Loads and read only texture reads via Read Cache
- Unordered shared consistent loads/stores/atomics via R/W Cache
- Wavefront length of 64 threads where each thread executes a 5 way VLIW Instruction each issue
  - ¼ Wavelength (16 threads) on each clock of 4 clocks (T0-15, T16-31, T32-47, T48-T63)

With thanks to AMD, with permission

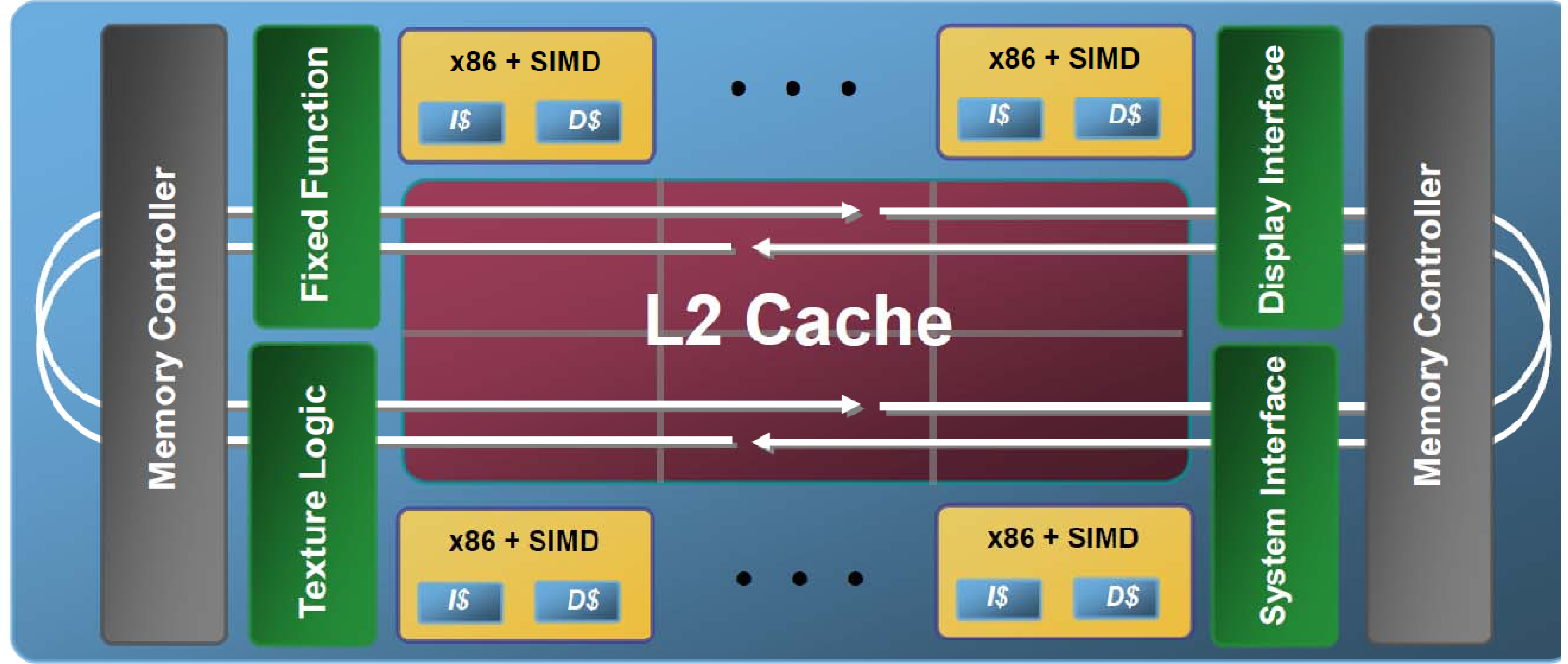
Local Data Share (LDS)



- This is AMD's implementation of the "shared" memory of the CUDA model
- 32 banks to allow concurrent access by all threads

With thanks to AMD, with permission

Intel's Larrabee



- Project to build a high-end GPU that bridges the gap to conventional multicore
- Each core is a simple in-order 4-way SMT x86
- Extended with a SIMD instruction set (16 floats wide)
- Special-purpose hardware for texture cache, it much else
- Both L1 (32KB per core) and L2 (256KB per core) data caches are coherent

Diagram: Intel

Our sources, and further information

- E. Lindholm, J. Nickolls, S. Oberman, J. Montrym. "NVIDIA Tesla: A Unified Graphics and Computing Architecture". *IEEE Micro* 28, 2 (March-April 2008), 39-55. http://dx.doi.org/10.1109/MM.2008.31
- D. Kanter. "NVIDIA's GT200: Inside a Parallel Processor". http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242
- K. Fatahalian, M. Houston. "GPUs: A Closer Look". *ACM Queue* 6, 2 (March-April 2008), 18-28. http://doi.acm.org/10.1145/1365490.1365498
- K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, K. Yelick. "Stencil Computation Optimization and Autotuning on State-of-the-Art Multicore Architectures". *Supercomputing'08.* http://doi.acm.org/10.1145/1413370.1413375
- Course on CUDA Programming on NVIDIA GPUs, July 20--24, 2009. Mike Giles, Oxford (http://people.maths.ox.ac.uk/gilesm/cuda/)
- CUDA Programming. Johan Seland, SINTEF Summer School (http://heim.ifi.uio.no/~knutm/geilo2008/seland.pdf)
- http://www.industrialmath.net/CUDA09_talks/pullan.pdf
- Larrabee: Software is the new Hardware. Tom Forsyth (Intel), talk at SIGGRAPH08 (http://s08.idav.ucdavis.edu/forsyth-larrabee-graphics-architecture.pdf)

Core Architecture	Intel Core2	AMD Barcelona	Sun Niagara2	STI Cell eDP SPE	NVIDIA GT200 SM
Type	super scalar out of order	super scalar out of order	MT dual issue†	SIMD dual issue	MT SIMD
Process	65nm	65nm	65nm	65nm	65nm
Clock (GHz)	2.66	2.30	1.16	3.20	1.3
DP GFlop/s	10.7	9.2	1.16	12.8	2.6
Local-Store	—	—	—	256KB	16KB**
L1 Data Cache	32KB	64KB	8KB	—	—
private L2 cache	—	512KB	—	—	—

System	Xeon E5355 (Clovertown)	Opteron 2356 (Barcelona)	UltraSparc T5140 T2+ (Victoria Falls)	QS22 PowerXCell 8i (Cell Blade)	GeForce GTX280
Heterogeneous	no	no	no	multicore	multichip
# Sockets	2	2	2	2	1
Cores per Socket	4	4	8	8(+1)	30 (×8)
shared L2/L3 cache	4×4MB (shared by 2)	2×2MB (shared by 4)	2×4MB (shared by 8)	—	—
DP GFlop/s	85.3	73.6	18.7	204.8	78
primary memory parallelism paradigm	HW prefetch	HW prefetch	Multithreading	DMA	Multithreading with coalescing
DRAM Bandwidth (GB/s)	21.33(read) 10.66(write)	21.33	42.66(read) 21.33(write)	51.2	141 (device) 4 (PCIe)
DP Flop:Byte Ratio	2.66	3.45	0.29	4.00	0.55
DRAM Capacity	16GB	16GB	32GB	32GB	1GB (device) 4GB (host)
System Power (Watts)§	330	350	610	270‡	450 (236)*
Chip Power (Watts)¶	2×120	2×95	2×84	2×90	165
Threading	Pthreads	Pthreads	Pthreads	libspe2.1	CUDA 2.0
Compiler	icc 10.0	gcc 4.1.2	gcc 4.0.4	xlc 8.2	nvcc 0.2.1221

Table 1. Architectural summary of evaluated platforms. †Each of the two thread groups may issue up to one instruction. **16 KB local-store shared by all concurrent *CUDA thread blocks* on the SM. ‡Cell Bladecenter power running Linpack averaged per blade. (www.green500.org) §All system power is measured with a digital power meter while under a full computational load. ¶Chip power is based on the maximum Thermal Design Power (TDP) from the manufacturer's datasheets. *GTX280 system power shown for the entire system under load (450W) and GTX280 card itself (236W).

Source: Yelick, Oliker et al, IPDPS?