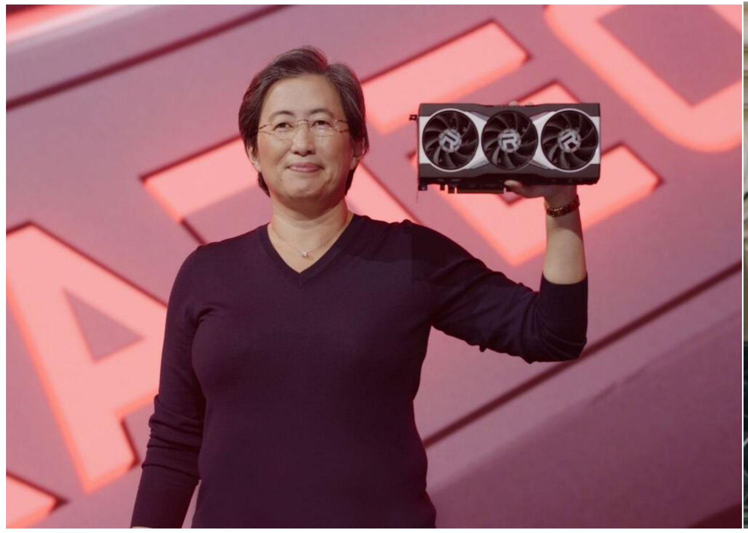
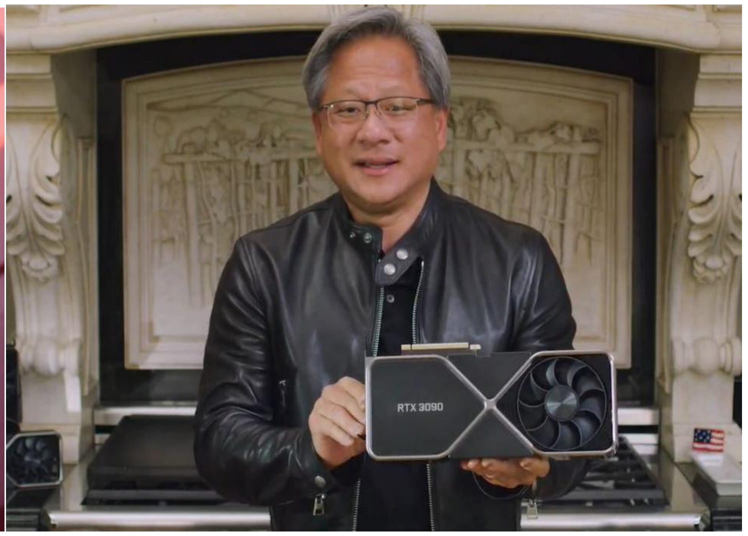
Imperial College London

Advanced Computer Architecture Chapter 9

Data-Level Parallel Architectures: GPUs


Lisa Su, CEO of AMD, launching the rx6000 series



Jensen Huang, CEO of NVIDIA, launching the RTX 30 Series GPUs

November 2025
Paul Kelly

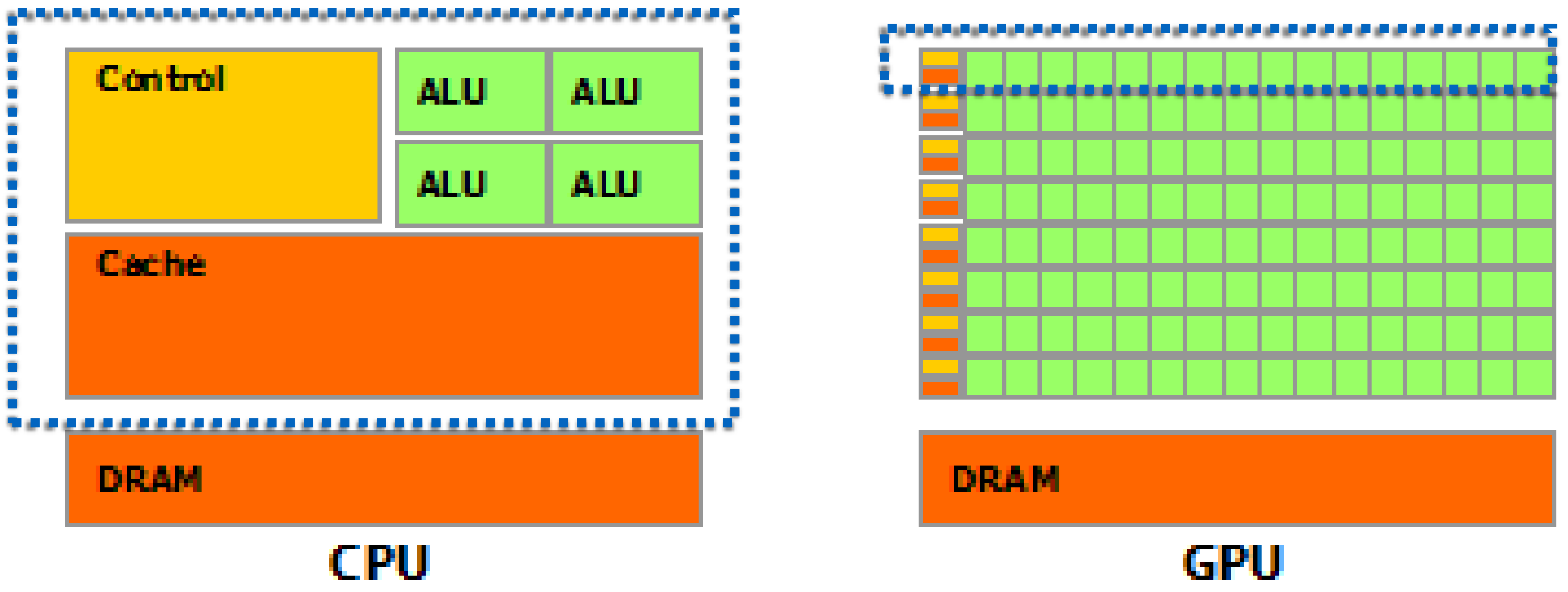
These lecture notes are partly based on:

- **Contributions to the lecture slides from Luigi Nardi (postdoc at Imperial and Stanford, now academic at Lund, Sweden), Fabio Luporini (Imperial PhD, postdoc, now CTO, DevitoCodes), and Nicolai Stawinoga (Imperial PhD, postdoc, now researcher at TU Berlin)**
- **the course text, Hennessy and Patterson's Computer Architecture (5th ed.)**

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, perhaps even no branch prediction
  - Can use FGMT or SMT to hide cache access latency, and maybe even main memory latency
  - Control is at a premium (Turing tax avoidance):
    - 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
- Initially driven by the gaming market – but with many other applications

A first comparison with CPUs



Source: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

- "Simpler" cores
- Many functional units (FUs) (implementing the SIMD model)
- Much less cache per core; just thousands of threads and super-fast context switch
- Drop sophisticated branch prediction mechanisms

NVIDIA G80 (2006)

16 cores, each with 8 "SP" units

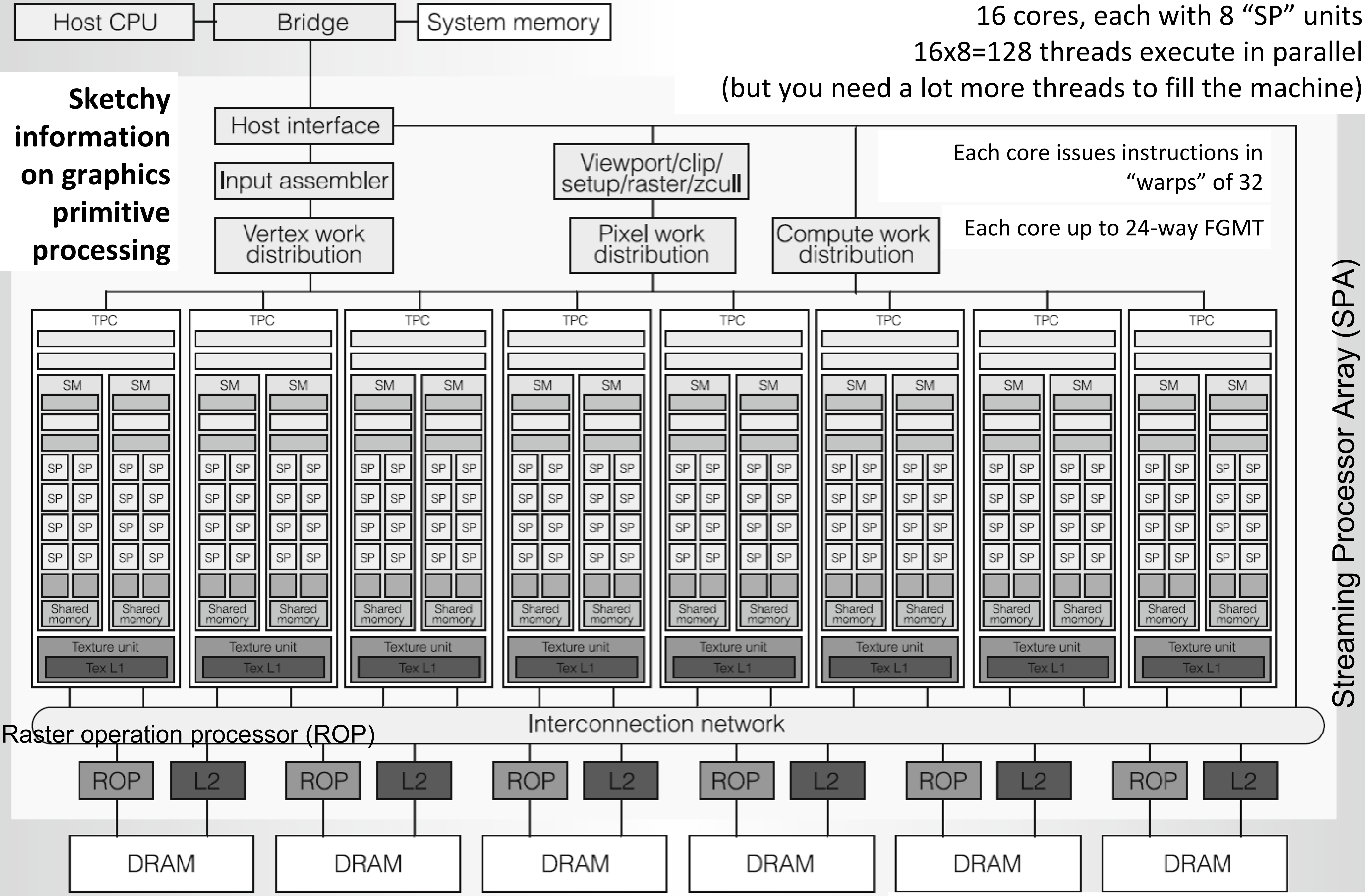
16x8=128 threads execute in parallel

(but you need a lot more threads to fill the machine)

Sketchy information on graphics primitive processing

Each core issues instructions in "warps" of 32

Each core up to 24-way FGMT

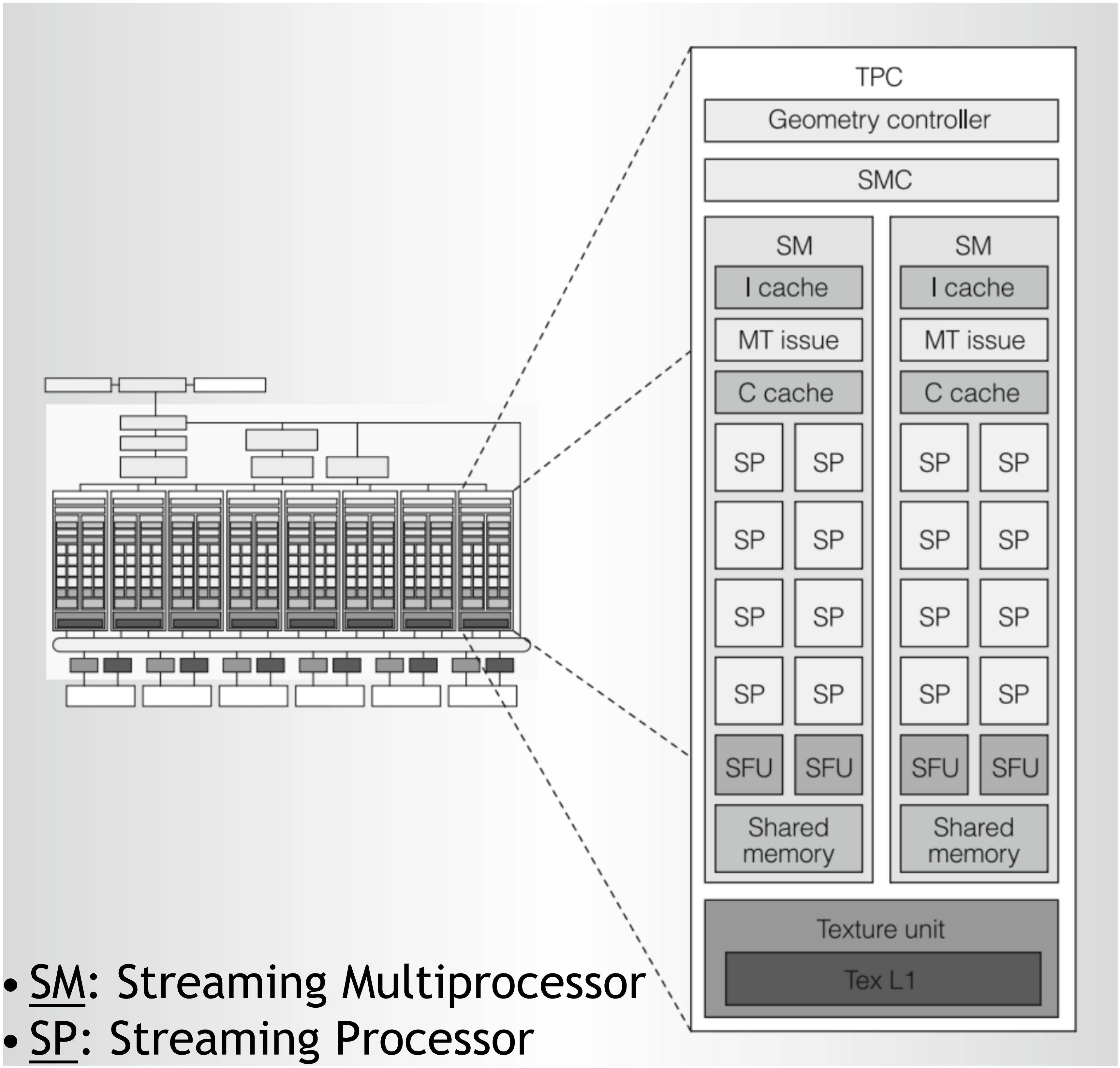


No L2 cache coherency problem, data can be in only one cache. Caches are small

ROP performs colour and depth frame buffer operations directly on memory

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

Texture/Processor Cluster (TPC)

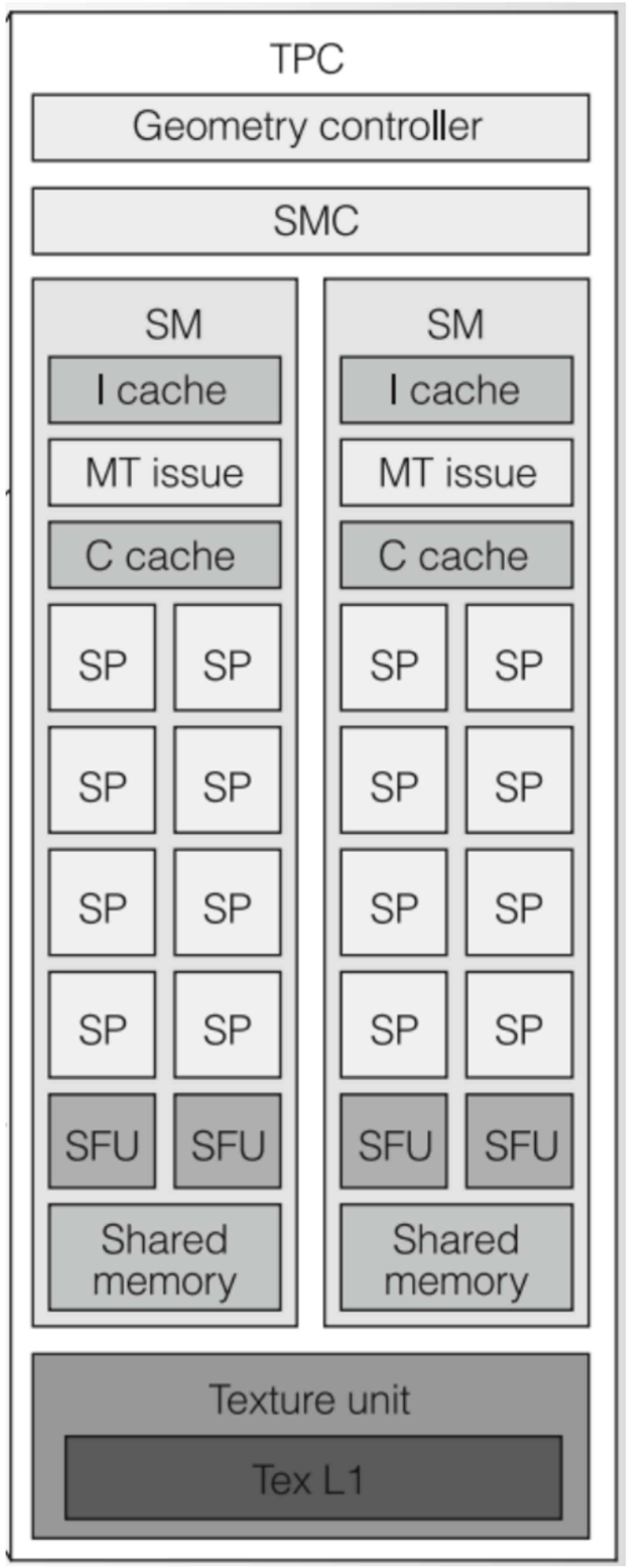


- SM: Streaming Multiprocessor
- SP: Streaming Processor

- SMC: Streaming Multiprocessor controller
- MT issue: multithreaded instruction fetch and issue unit
- C cache: constant read-only cache
- I cache: instruction cache
- Geometry controller: directs all primitive and vertex attribute and topology flow in the TPC
- SFU: Special-Function Unit, compute trascendental functions (sin, cos, log x, 1/x)
- Shared memory: scratchpad memory, i.e. user managed cache
- Texture cache does interpolation

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

NVIDIA's Tesla micro-architecture



Combines many of the ideas we have learned about:

- Many fetch-execute processor devices (16 "SMs")
- Each one uses fine-grain multithreading (FGMT) to run 32 "**warps**" per SM
  - NVIDIA is confusing about terminology!
  - Warps on a GPU are like threads on a CPU
  - Threads on a GPU are like lanes on a SIMD CPU

- MT issue selects which "warp" to issue from in each cycle (FGMT)

- Each warp's instructions are actually 32-wide SIMD instructions
- Executed in four steps, using 8 SPs ("vector pipelining", Ch08)
- With lanewise predication (Ch08)

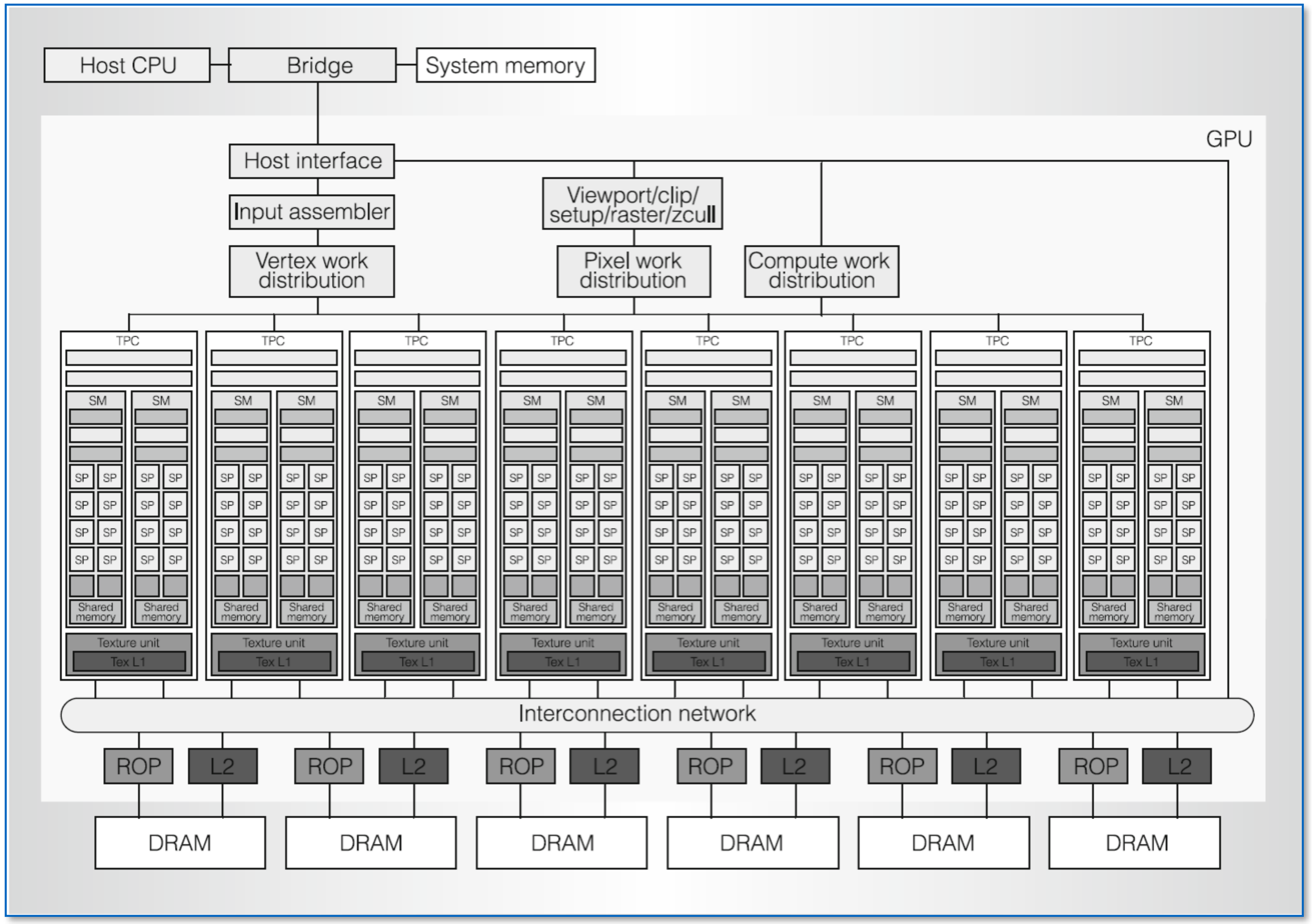
- Each SM has local, explicitly-programmed scratchpad memory
- Different warps on the same SM can share data in this "shared memory"

- SM's also have an L1 data cache (but no cache-coherency protocol)

- The chip has multiple DRAM channels, each of which includes an L2 cache (but each data value can only be in one L2 location, so there's no cache coherency issue at the L2 level)

- There are also graphics-specific mechanisms, which we will not discuss here (eg a special L1 "texture cache" that can interpolate a texture value)

Tesla memory, interconnect, control

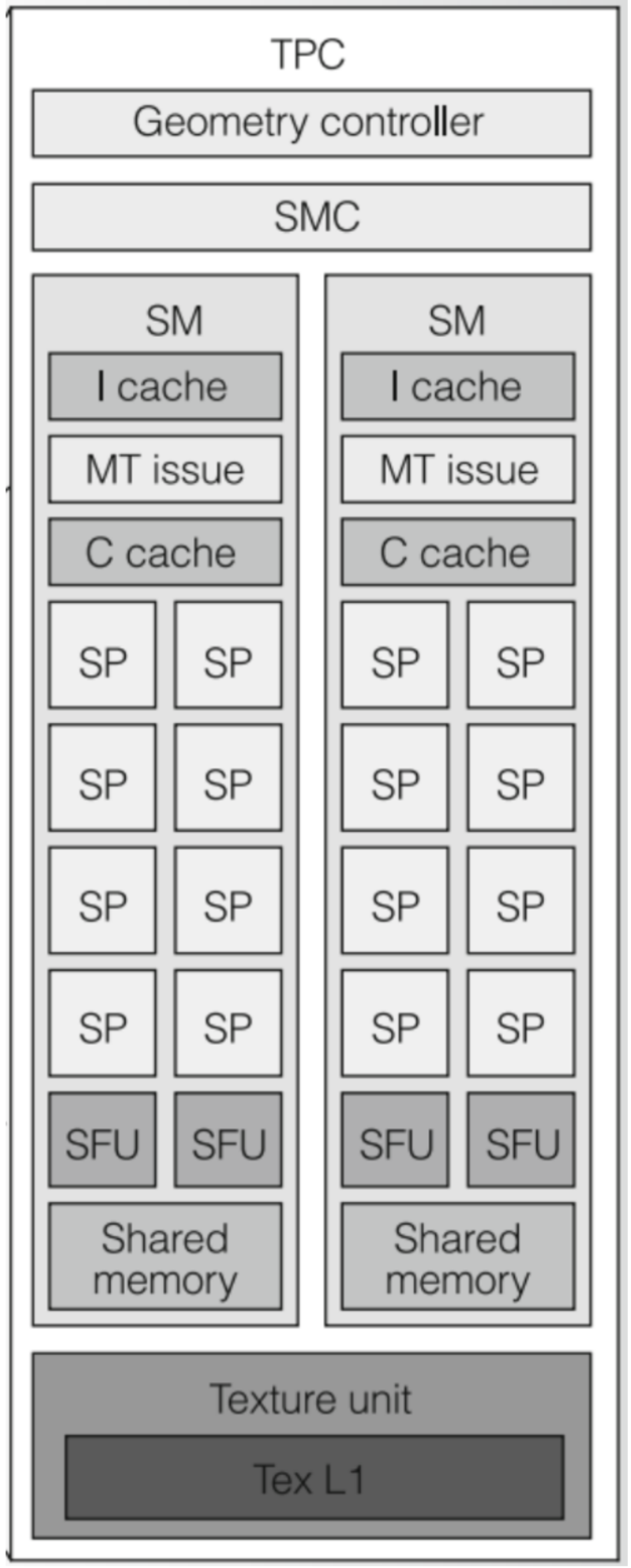


- SM's also have an L1 data cache (but no cache-coherency protocol – flushed on kernel launch)

- The chip has multiple DRAM channels, each of which includes an L2 cache
- but each data value can only be in one L2 location, so there's no cache coherency issue at the L2 level

- Tesla has more features specific to graphics, which are not our focus here:
  - Work distribution, load distribution
  - Texture cache, pixel interpolation
  - Z-buffering and alpha-blending (the ROP units, see diagram)

CUDA: using NVIDIA GPUs for general computation



- Designed to do rendering
- Evolved to do general-purpose computing (GPGPU)
  - But to manage thousands of threads, a new programming model is needed, called CUDA (Compute Unified Device Architecture)
  - CUDA is proprietary, but 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

We focus initially on NVIDIA architecture and terminology. AMD GPUs are quite similar, and the OpenCL programming model is similar to CUDA. Mobile GPUs are somewhat different

CUDA Execution Model

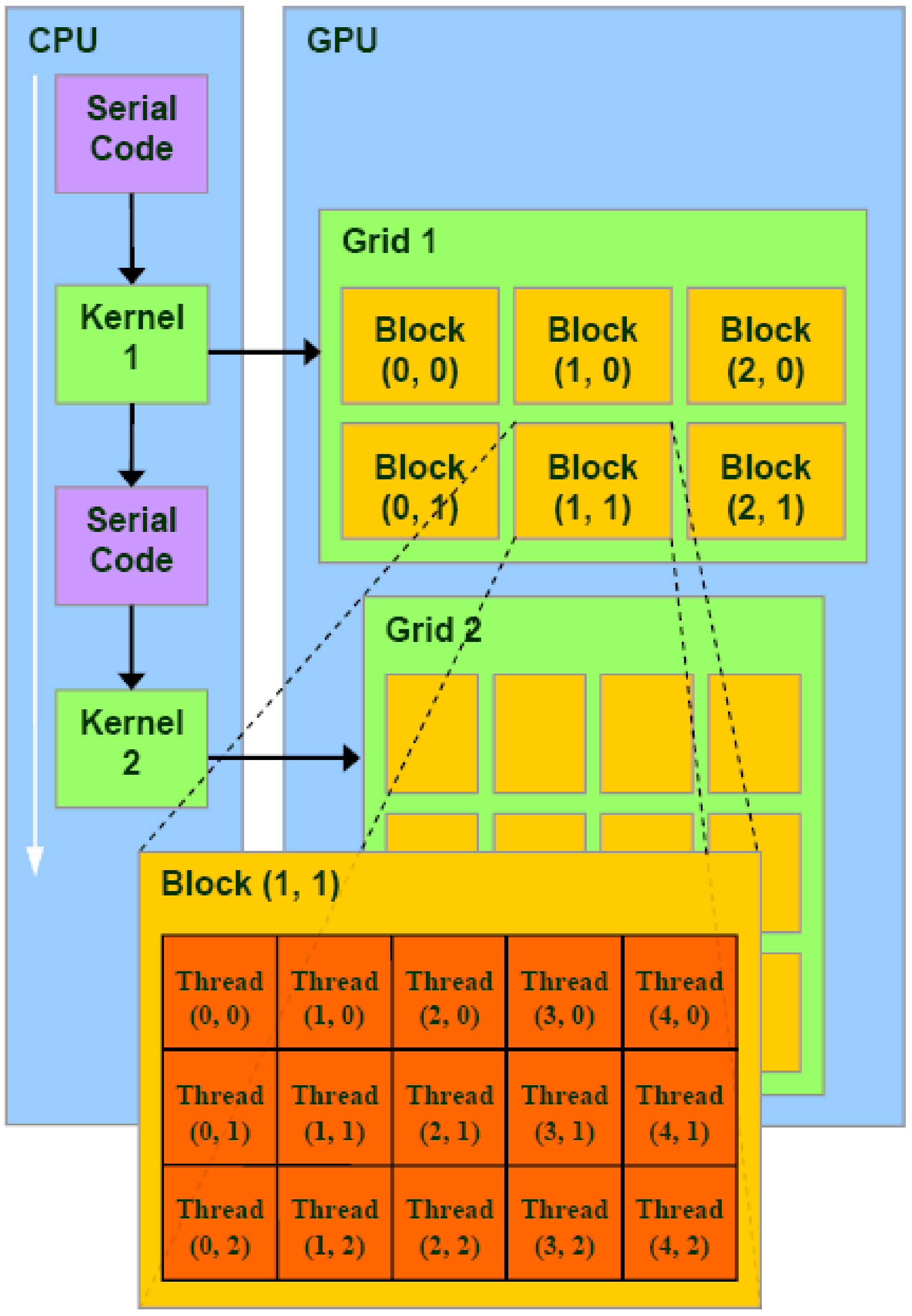
- CUDA is a C extension
  - Serial CPU code
  - Parallel GPU code (*kernels*)
- GPU kernel is a C function
  - Each *thread* executes kernel code
  - A group of threads form a *thread block* (1D, 2D or 3D)
  - Thread blocks are organised into a *grid* (1D, 2D or 3D)
  - Threads within the same thread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

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

Thread blocks are allocated (dynamically) to SMs, and run to completion

Threads (warps) within a block **run on the same SM,** so can share data and synchronise

Different blocks in a grid can't interact with each other



Source: CUDA programming guide

CUDA example: DAXPY

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

CUDA kernel

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n,
           double a,
           double* x,
           double* y) {
    for(int i=0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

C version for comparison

fully parallel loop

CPU code to launch kernel on GPU

```
int main(){
  // Kernel setup
  int N = 1024;
  int blockDim = 256;          // These are the threads per block
  int gridDim = N / blockDim;  // These are the number of blocks
  daxpy<<<gridDim, blockDim>>>(N, 2.0, x, y); // Kernel invocation
}
```

- **Kernel invocation ("<<<...>>>") corresponds to enclosing loop nest, managed by hardware**
- **Explicitly split into 2-level hierarchy:**
  blocks (256 threads that can share "shared" memory), and grid (N/256 blocks)
- **Kernel commonly consists of just one iteration but could be a loop**
- **Multiple tuning parameters trade off register pressure, shared-memory capacity and parallelism**

PTX Example (SAXPY code)

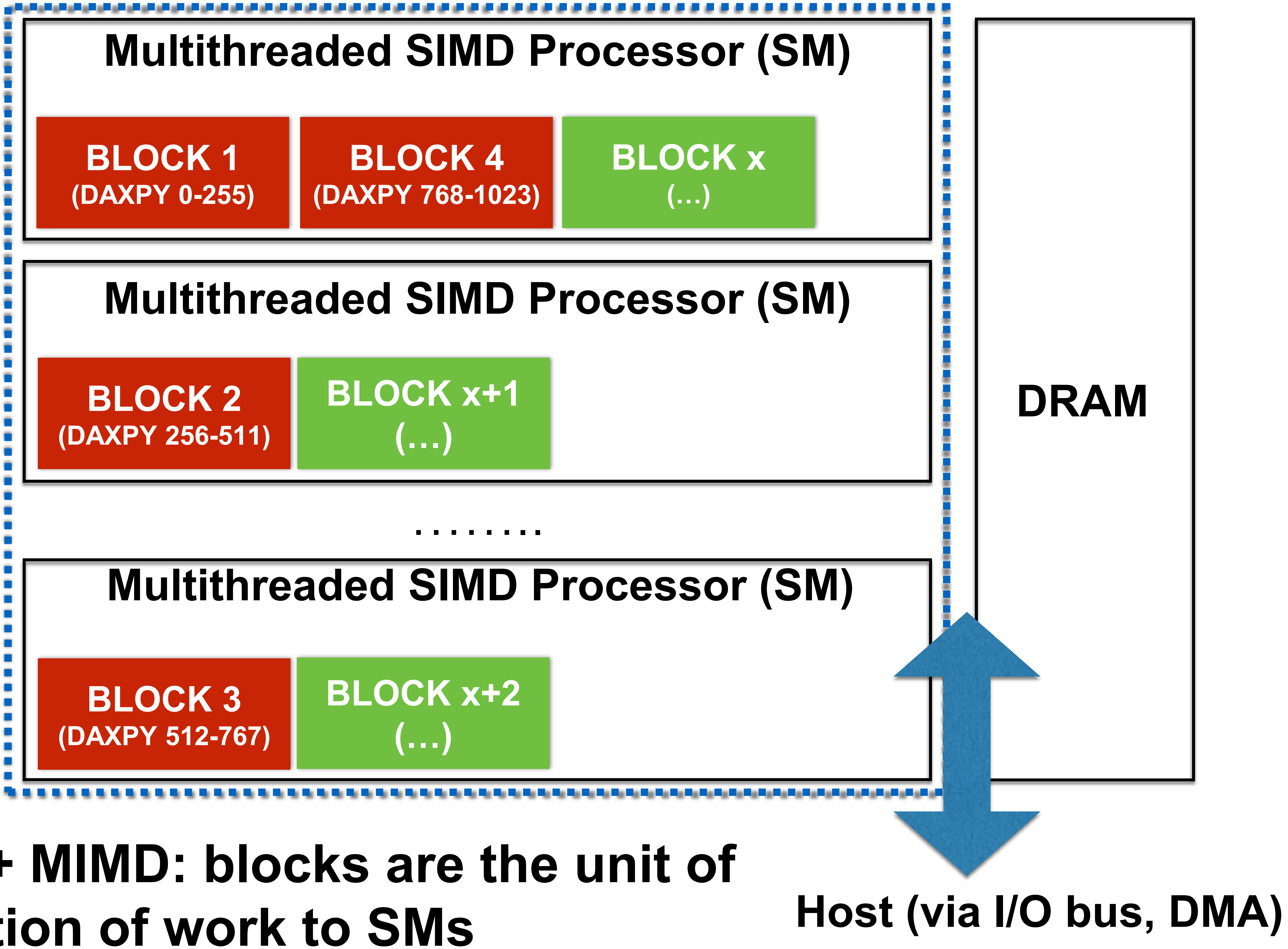
```
cvt.u32.u16     $blockid, %ctaid.x;     // Calculate i from thread/block IDs
cvt.u32.u16     $blocksize, %ntid.x;
cvt.u32.u16     $tid, %tid.x;
mad24.lo.u32    $i, $blockid, $blocksize, $tid;
ld.param.u32    $n, [N];                // Nothing to do if n ≤ i
setp.le.u32     $p1, $n, $i;
@$p1 bra        $L_finish;

mul.lo.u32      $offset, $i, 4;         // Load y[i]
ld.param.u32    $yaddr, [Y];
add.u32         $yaddr, $yaddr, $offset;
ld.global.f32   $y_i, [$yaddr+0];
ld.param.u32    $xaddr, [X];            // Load x[i]
add.u32         $xaddr, $xaddr, $offset;
ld.global.f32   $x_i, [$xaddr+0];

ld.param.f32    $alpha, [ALPHA];        // Compute and store alpha*x[i] + y[i]
mad.f32         $y_i, $alpha, $x_i, $y_i;
st.global.f32   [$yaddr+0], $y_i;

$L_finish:      exit;
```

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

- This is PTX: a pseudo-assembly code that is translated to proprietary ISA
- Threads are scheduled in hardware
- Each thread is provided with its position in the Grid through registers %ctaid, %ntid, %tid
- p1 is a predicate register to determine the outcome of the "if"
- The conditional branch "@$p1 bra $L_finish" may be (probably is) translated to predication in the target ISA

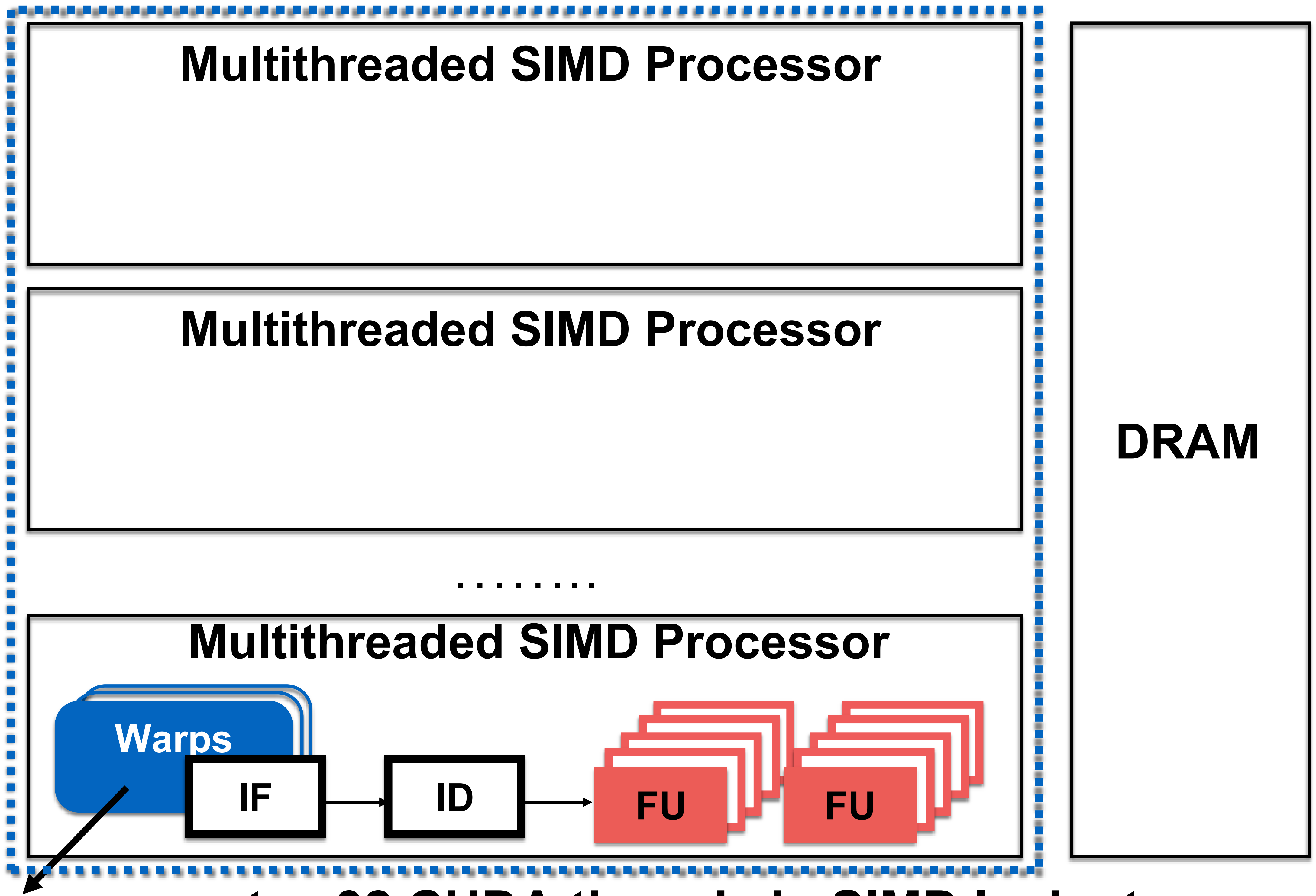
(Joy Lee and others, NVIDIA)

Running DAXPY (N=1024) on a GPU



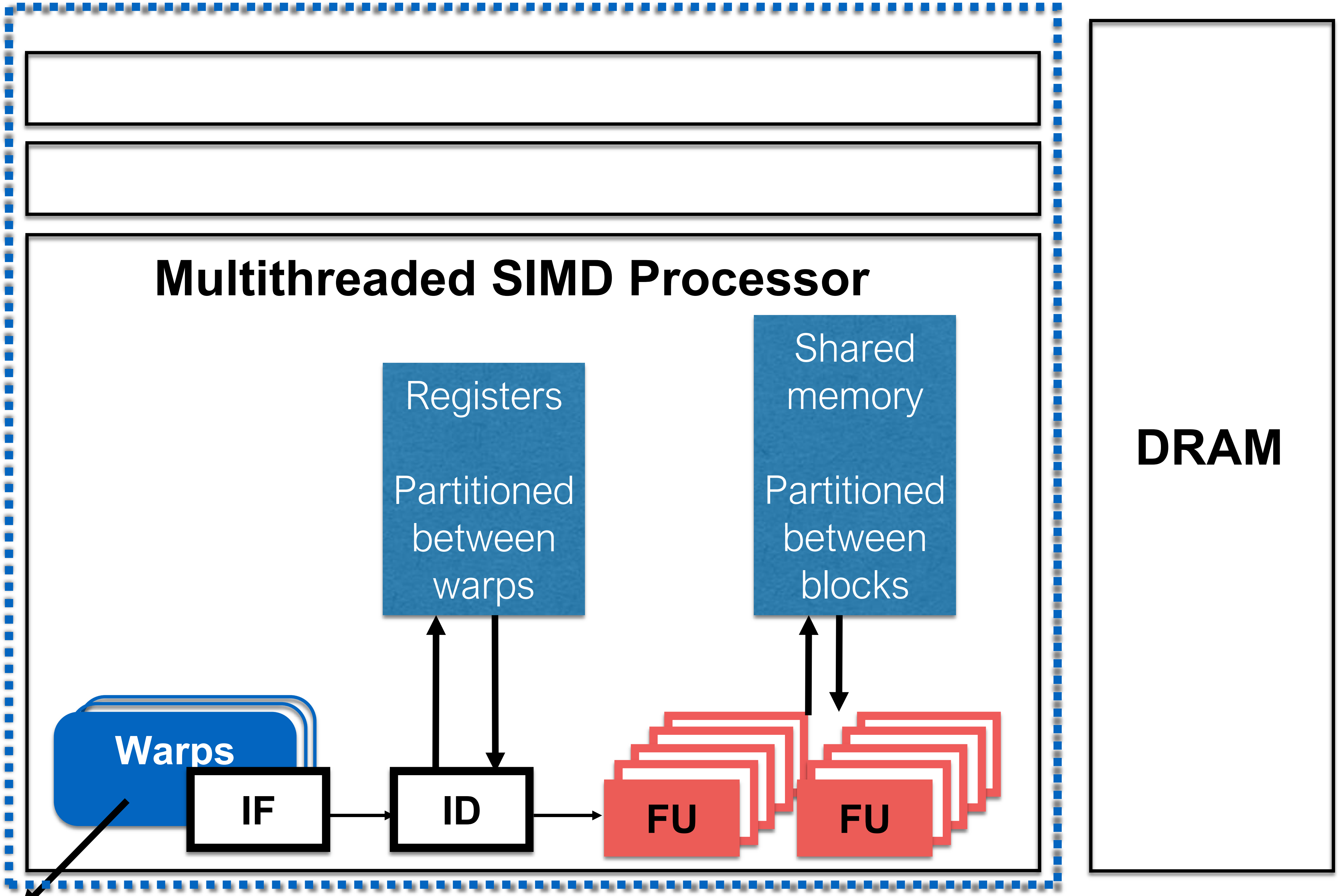
SIMD + MIMD: blocks are the unit of allocation of work to SMs

Running DAXPY on a GPU


Multithreaded SIMD Processor
Multithreaded SIMD Processor
....
Multithreaded SIMD Processor
Warps
IF
ID
FU
FU
DRAM

- Each warp executes 32 CUDA threads in SIMD lock-step
- Each CUDA thread executes one instance of the kernel
- Each SM is shared by many warps (possibly from the same or different blocks)

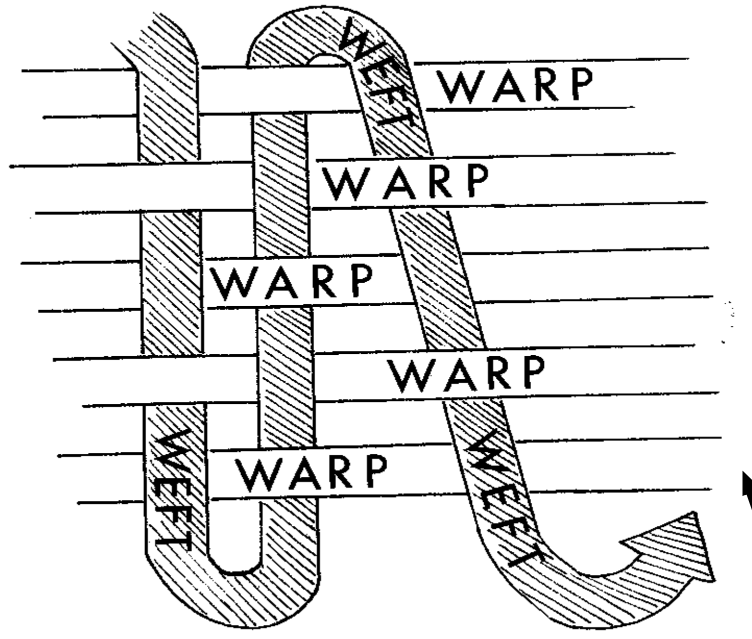
Running DAXPY on a GPU

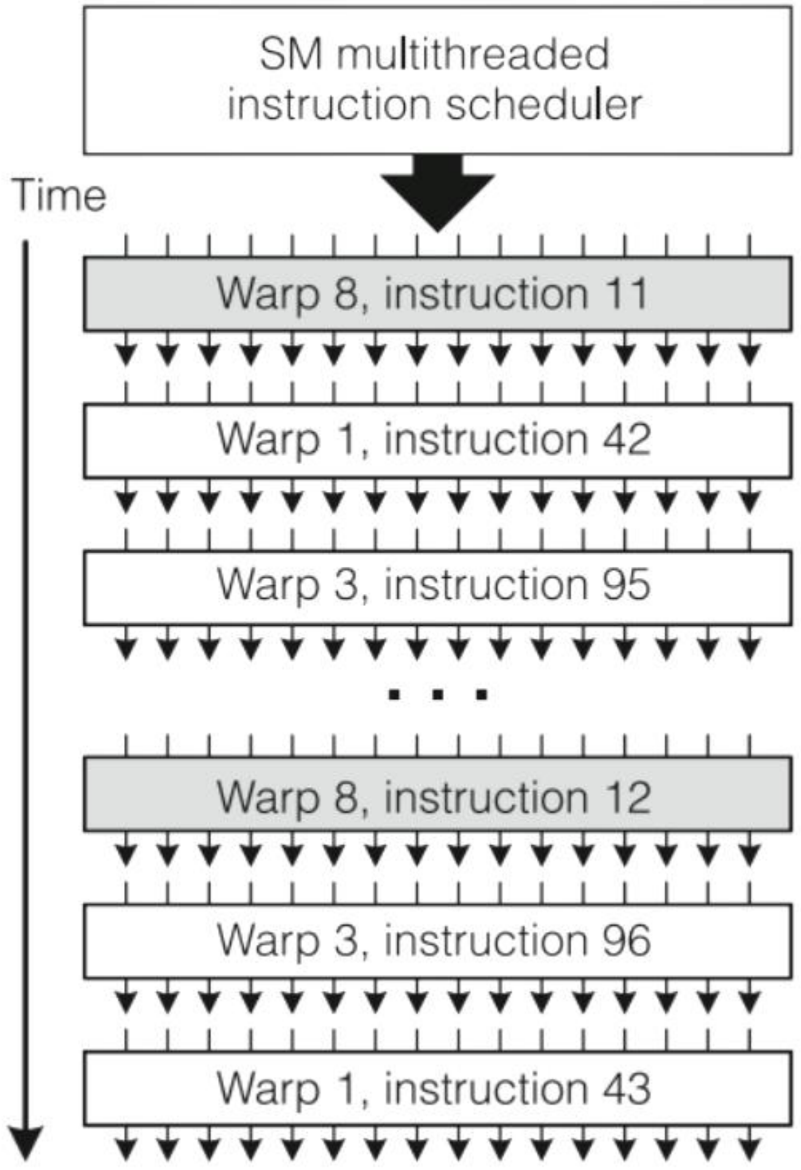


Multithreaded SIMD Processor
Registers
Partitioned between warps
Shared memory
Partitioned between blocks
Warps
IF
ID
FU
FU
DRAM

- Each warp executes 32 CUDA threads in SIMD lock-step
- Each CUDA thread executes one instance of the kernel
- Each SM is shared by many warps (possibly from the same or different blocks)

Single-instruction, multiple-thread (SIMT)

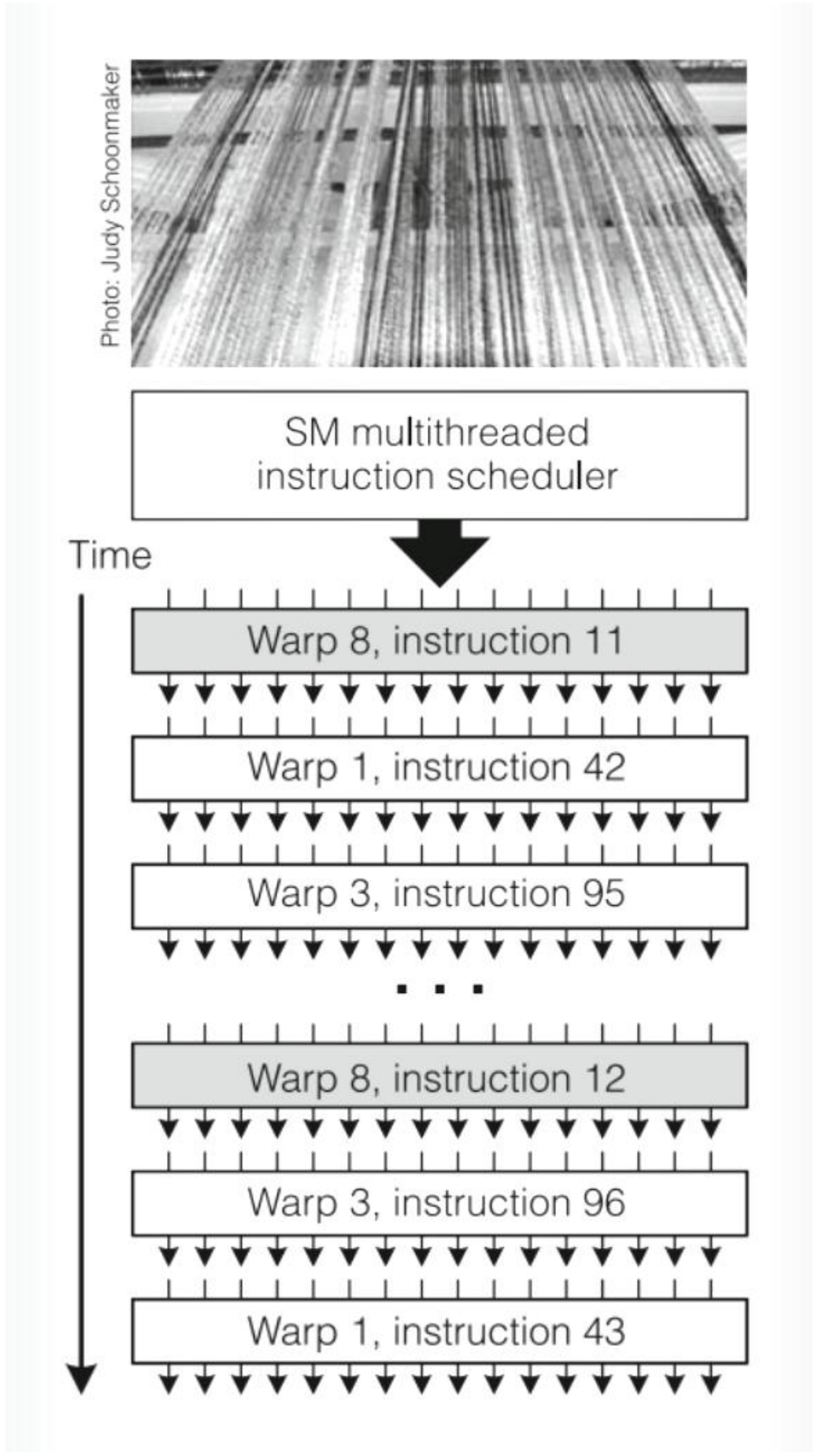




- A new parallel programming model: SIMT
- The SM's SIMT multithreaded instruction unit creates, manages, schedules, and executes threads in groups of warps
- The term warp originates from weaving
- Each SM manages a pool of 24 warps, 24 ways FGMT (more on later devices)
- Individual threads composing a SIMT warp start together at the same program address, but they are otherwise free to branch and execute independently
- At instruction issue time, select ready-to-run warp and issue the next instruction to that warp's active threads

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

Reflecting on SIMT



- SIMT architecture is similar to SIMD design, which applies one instruction to multiple data lanes
- The difference: SIMT applies one instruction to multiple independent threads in parallel, not just multiple data lanes. A SIMT instruction controls the execution and branching behaviour of one thread
- For program correctness, programmers can ignore SIMT executions; but, they can achieve performance improvements if threads in a warp don't diverge
- Correctness/performance analogous to the role of cache lines in traditional architectures
- The SIMT design shares the SM instruction fetch and issue unit efficiently across 32 threads but requires a full warp of active threads for full performance efficiency

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

Branch divergence

- In a warp, threads all take the same path (good!) or diverge!
  - A warp serially executes each path, disabling some of the threads
  - When all paths complete, the threads reconverge
- Divergence only occurs within a warp - different warps execute independently
- Control-flow coherence: when all the threads in a warp goes the same way we get good utilisation (a form of locality – spatial branch locality)

```
:
:
if (x == 10)
   c = c + 1;
:
```

Predicate bits: enable/disable each lane

```
      :
      LDR r5, X
      p1 <- r5 eq 10
<p1> LDR  r1 <- C
<p1> ADD r1, r1, 1
<p1> STR  r1 -> C
      :
```

SIMT vs SIMD – GPUs without the hype

- GPUs combine many architectural techniques:
  - Multicore
  - Simultaneous multithreading (SMT)
  - Vector instructions
  - Predication

- So basically a GPU core is a lot like the processor architectures we have studied!
- But the SIMT programming model makes it look different

- **Overloading the same architectural concept doesn't help GPU beginners**
- **GPU learning curve is steep in part because of using terms such as "Streaming Multiprocessor" for the SIMD Processor, "Thread Processor" for the SIMD Lane, and "Shared Memory" for Local Memory - especially since Local Memory is not shared between SIMD Processors**

SIMT vs SIMD – GPUs without the hype

SIMT:

- One thread per lane
- Adjacent threads ("warp"/"wavefront") execute in lockstep
- SMT: multiple "warps" run on the same core, to hide memory latency

SIMD:

- Each thread may include SIMD vector instructions
- SMT: a small number of threads run on the same core to hide memory latency

Which one is easier for the programmer?

SIMT vs SIMD – spatial locality & coalescing

SIMT:

- Spatial locality = adjacent threads access adjacent data
- A load instruction can result in a completely different address being accessed by each lane
- "Coalesced" loads, where accesses are (almost) adjacent, run *much* faster

SIMD:

- Spatial locality = adjacent loop iterations access adjacent data
- A SIMD vector load usually *has* to access adjacent locations
- Some recent processors have "gather" instructions which can fetch from a different address per lane
- But performance is often serialised

SIMT vs SIMD – spatial locality & coalescing

SIMD (on CPU):

```
void add (float *c, float *a, float *b)
{
  for (int i=0; i <= N; i++)
    #pragma omp simd
    for (int j=0; j <= N; j++)
       c[i][j]=a[i][j]+b[i][j];
}
```

Using OpenMP

This example has good spatial locality because it traverses the data in layout order:

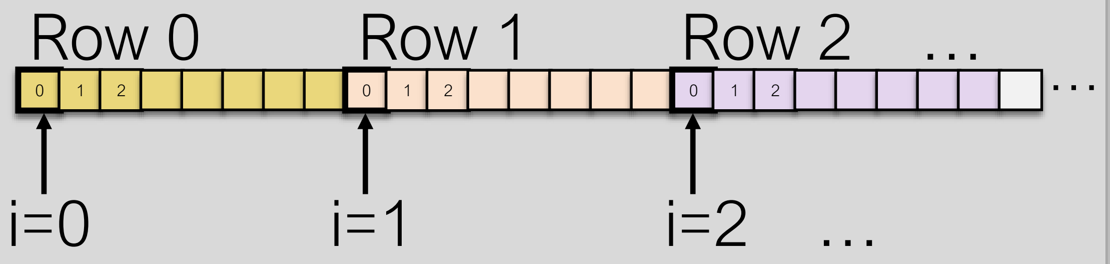
```
void add (float *c, float *a, float *b)
{
  for (int i=0; i <= N; i++) {
    __m128* pa = (__m128*) &a[i][0];
    __m128* pb = (__m128*) &b[i][0];
    __m128* pc = (__m128*) &c[i][0];
    for (int i=0; i <= N/4; i++)
      *pc++ = _mm_add_ps(*pa++,*pb++);
  }
}
```

Using intrinsics

SIMT (on GPU):

```
__global__ void add(int N,
                      double* a,
                      double* b,
                      double* c) {
    int i = blockIdx.x *
            blockDim.x +
            threadIdx.x;
 for (int j=0; j <= N; j++)
   c[i][j] = a[i][j] + b[i][j];
}
```

This example has terrible spatial locality because adjacent threads access different rows



Threads with adjacent thread ids access data in different cache lines

SIMT vs SIMD – spatial *control* locality

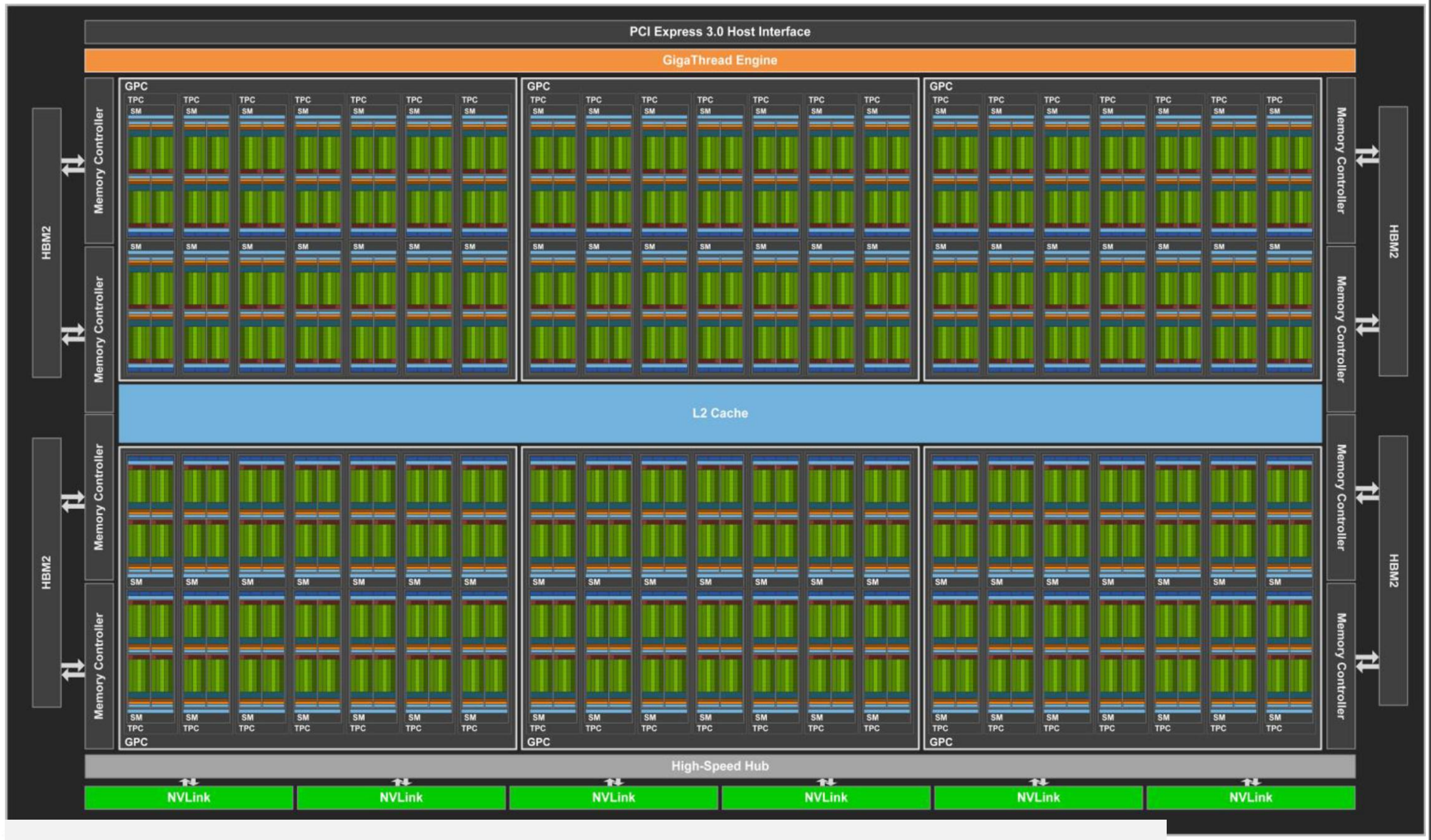
SIMT:

- Branch coherence = adjacent threads in a warp all usually branch the same way (*spatial* locality for branches, across threads)

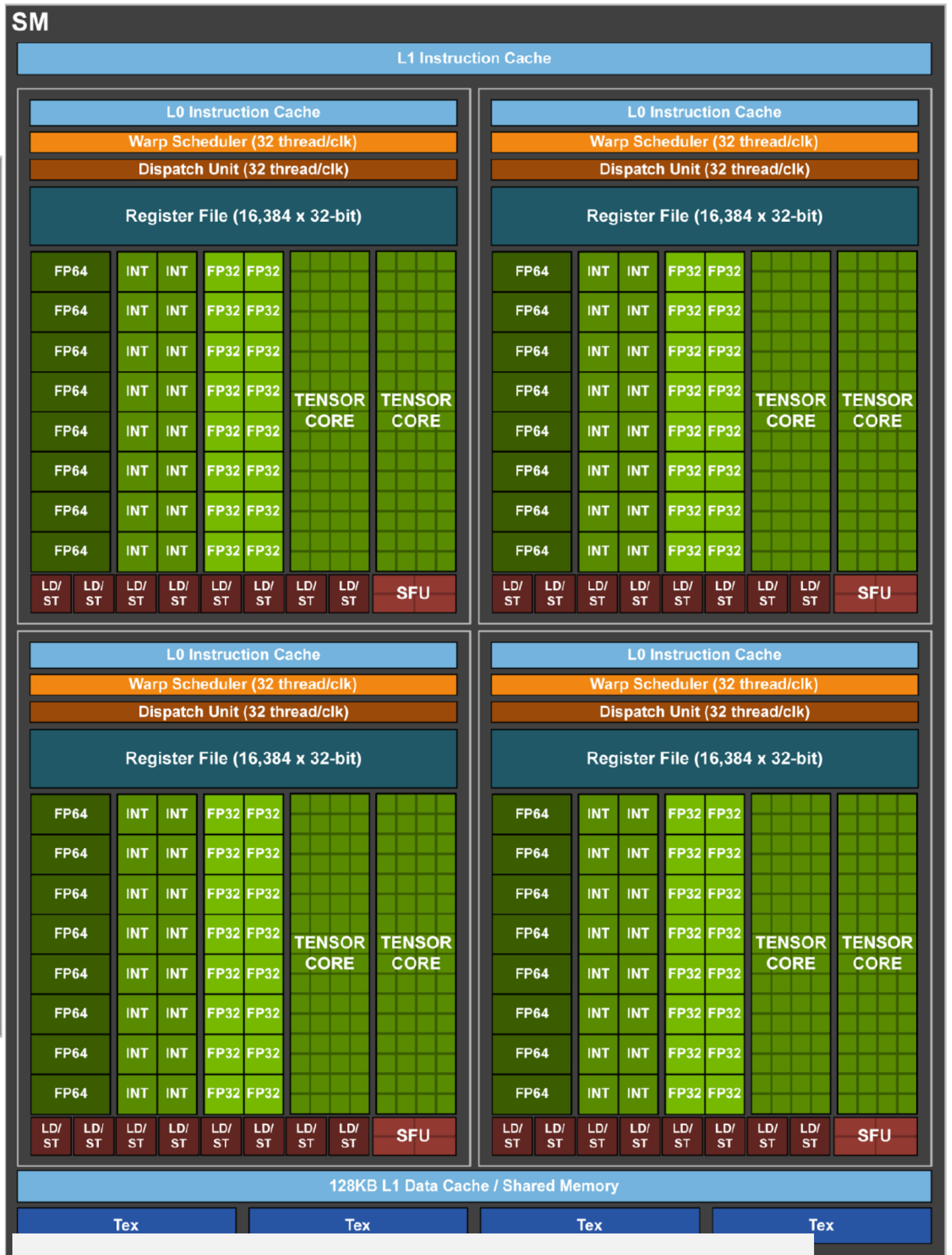
SIMD:

- Branch predictability = each *individual* branch is mostly taken or not-taken (or is well-predicted by global history)

NVIDIA Volta GPU (2017)



GV100 with 84 SMs



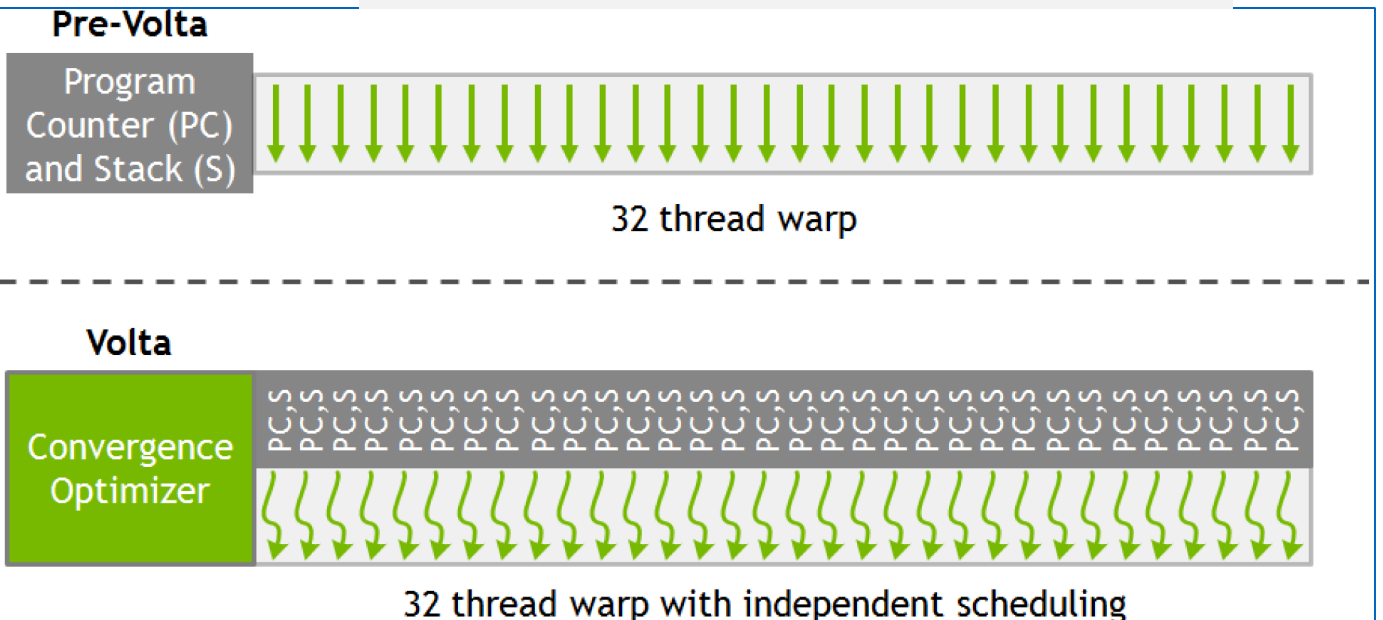
SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

GV100's SM includes 8 tensor cores

$$
D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}
$$

FP16 or FP32 FP16 FP16 FP16 or FP32

Tensor core computes matrix-matrix multiply on FP16s with FP32 accumulation



Pre-Volta
Program Counter (PC) and Stack (S)
32 thread warp
Volta
Convergence Optimizer
PC,S
32 thread warp with independent scheduling

Each CUDA thread has its *own* PC and stack, enabling dynamic scheduling in hardware to heuristically enhance branch convergence

NVIDIA TESLA V100 GPU ARCHITECTURE (Aug 2017) https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

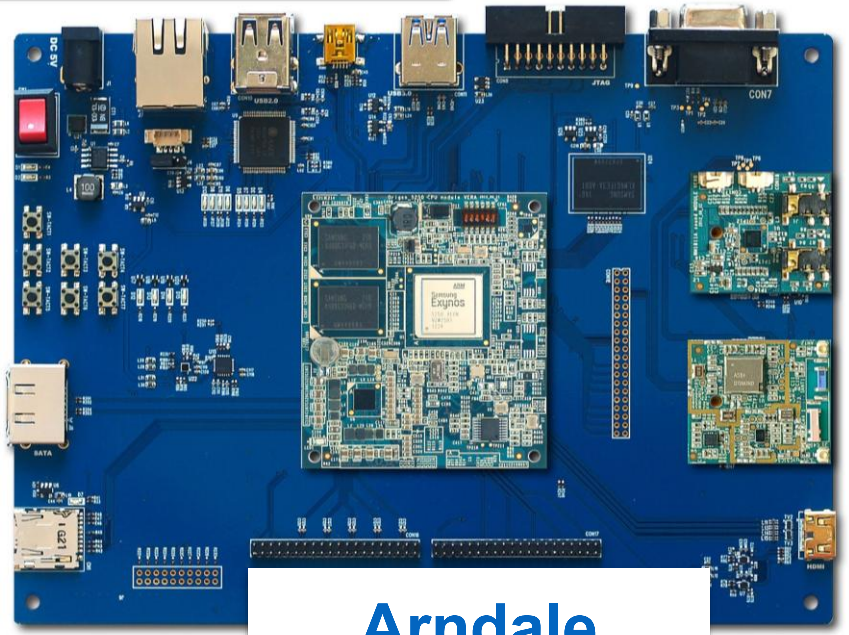
It is a heterogeneous world

TITAN
4998 GFLOPS
< 400 W

GTX 870M
2827 GFLOPS
< 100 W

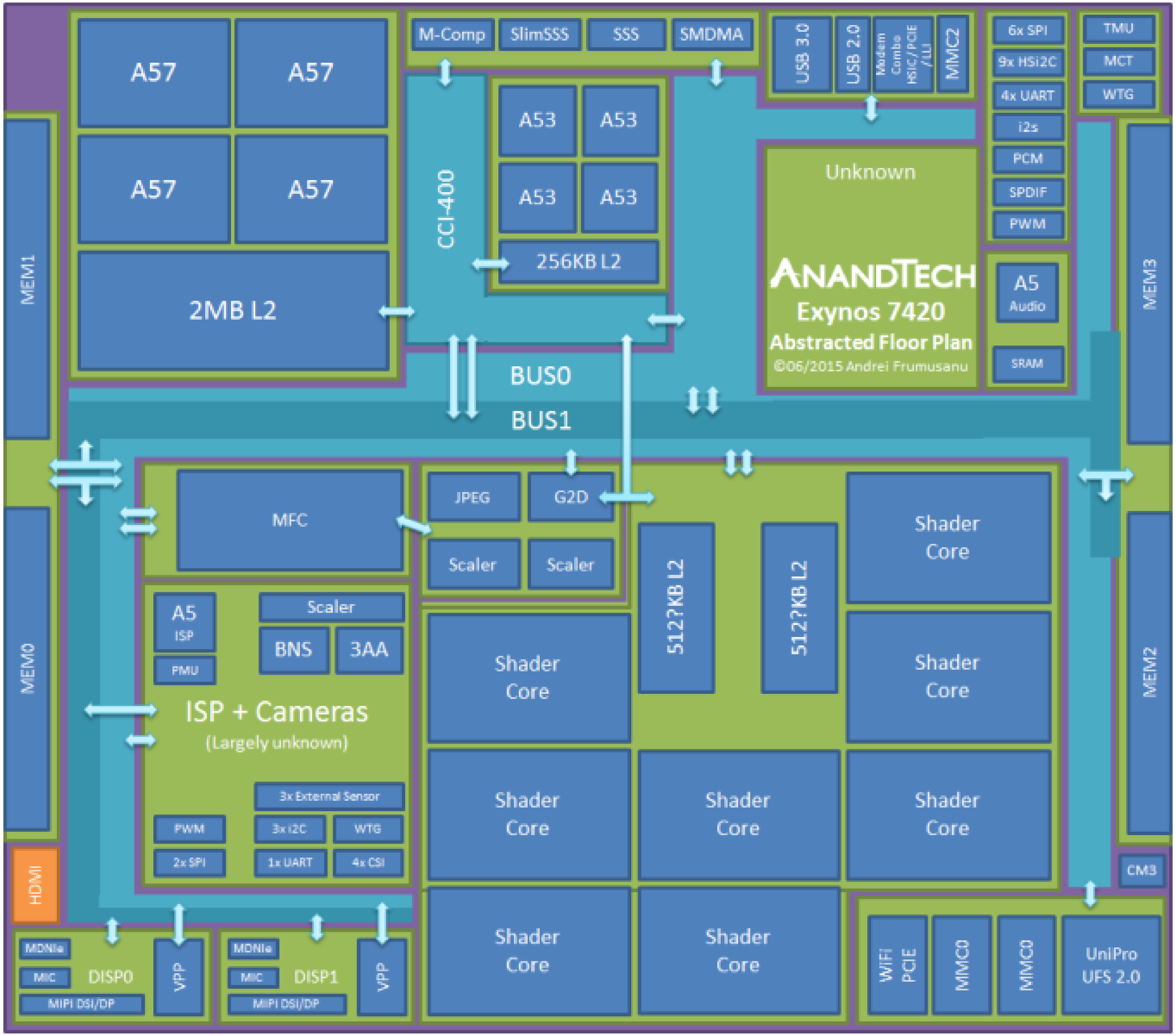
TK1
404 GFLOPS
< 20 W

ODROID
170 GFLOPS
< 10 W



Arndale
87 GFLOPS
< 5 W

ARM-based Samsung Exynos 7420 SoC Reverse engineered

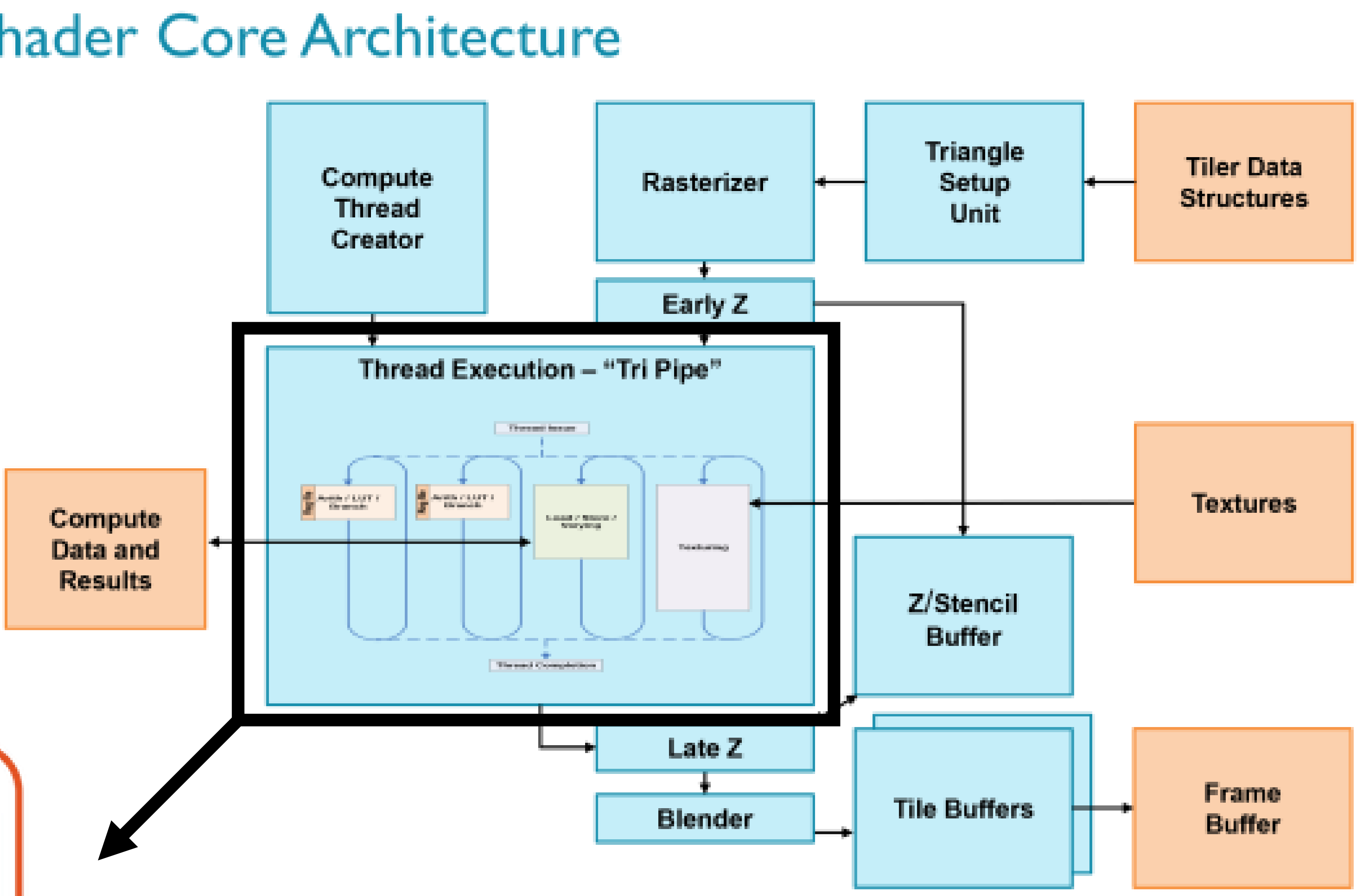


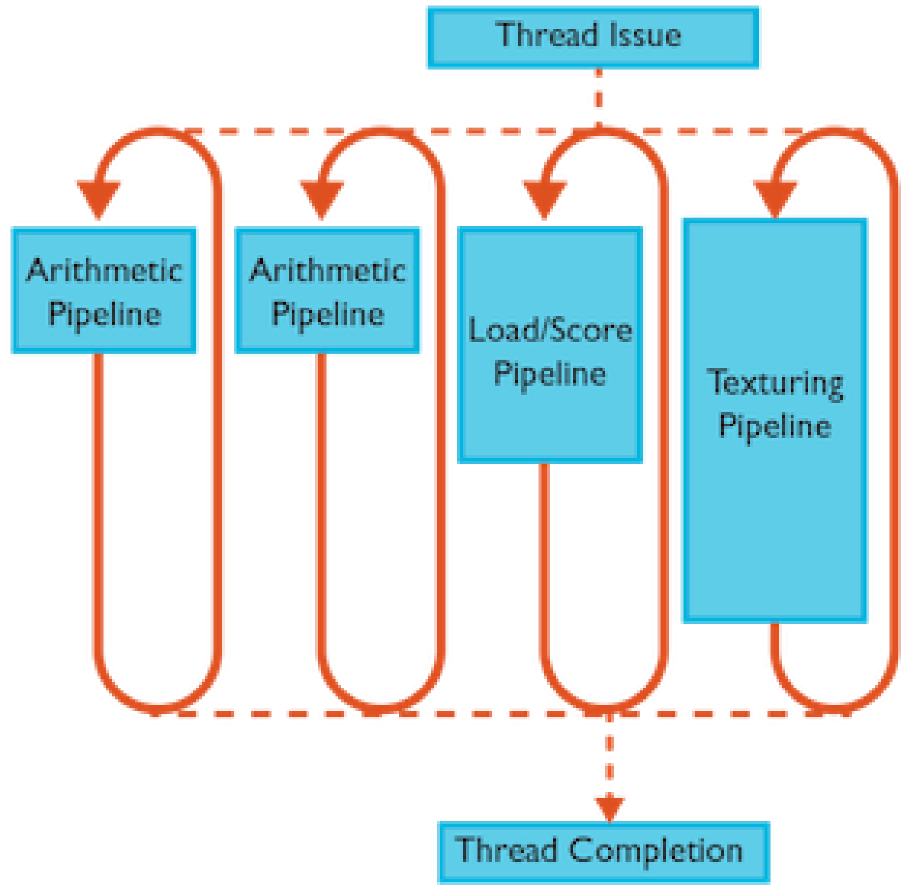
Source: http://www.anandtech.com/show/9330/exynos-7420-deep-dive/2

Used in Samsung S6 mobile phone

spare slides for interest

ARM MALI GPU: Midgard microarchitecture



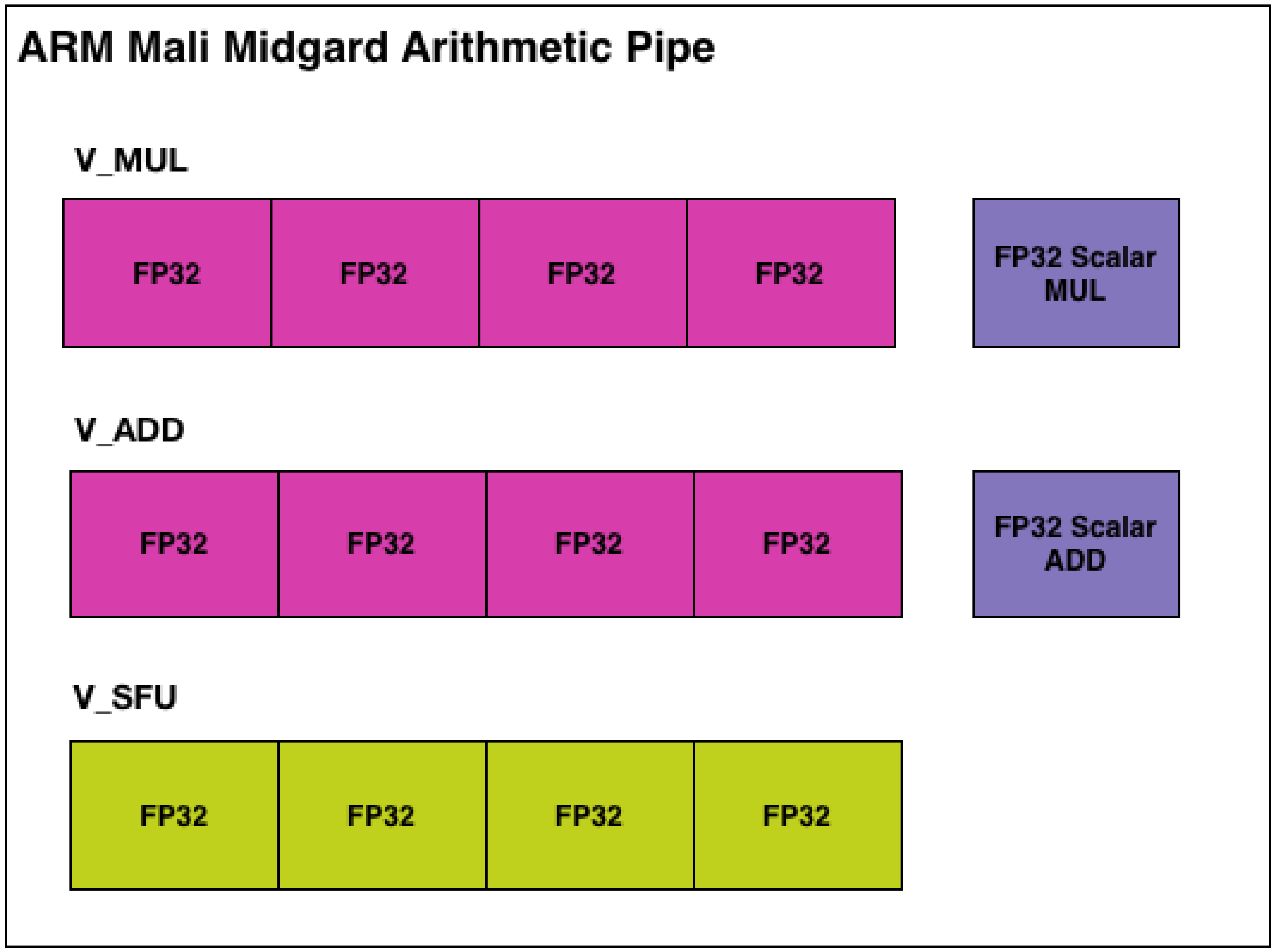


- Variable number of Arithmetic Pipelines (uncommon feature with respect to other GPUs)
- Fixed number of Load/Store and Texturing Pipelines
- In-order scheduling
- This diagram shows only the Shader Core, there is much more supporting hardware to make a complete GPU, i.e. tiling unit, memory management unit, L2 cache, etc.

Source: http://www.anandtech.com/show/8234/arms-mali-midgard-architecture-explored/4

Midgard arithmetic Pipe

Source: http://www.anandtech.com/show/8234/arms-mali-midgard-architecture-explored/5



- Very flexible SIMD
- Simply fill the SIMD with as many (identical) operations as will fit, and the SIMD will handle it

- ARM Midgard is a VLIW design with SIMD characteristics (power efficient)
- So, at a high level ARM is feeding multiple ALUs, including SIMD units, with a single long word of instructions (ILP)
- Support a wide range of data types, integer and FP: I8, I16, I32, I64, FP16, FP32, FP64
- 17 SP GFLOPS per core at 500 MHz (if you count also the SFUs)

Optimising for MALI GPUs

How to run optimally OpenCL code on Mali GPUs means mainly to locate and remove optimisations for alternative compute devices:

- Use of local or private memory: Mali GPUs use caches instead of local memories. There is therefore no performance advantage using these memories on a Mali
- Barriers: data transfers to or from local or private memories are typically synchronised with barriers. If you remove copy operations to or from these memories, also remove the associated barriers
- Use of scalars: some GPUs work with scalars whereas Mali GPUs can also use vectors. Do vectorise your code
- Optimisations for divergent threads: threads on a Mali are independent and can diverge without any performance impact. If your code contains optimisations for divergent threads in warps, remove them
- Modifications for memory bank conflicts: some GPUs include per-warp memory banks. If the code includes optimisations to avoid conflicts in these memory banks, remove them
- No host-device copies: Mali shares the same memory with the CPU

Source: http://infocenter.arm.com/help/topic/com.arm.doc.dui0538f/DUI0538F_mali_t600_opencl_dg.pdf

Texture cache

- GPUs were built for rendering
- Critical element:
  - Mapping from a stored texture onto a triangular mesh
- To render each triangle:
  - enumerate the pixels,
  - map each pixel to the texture and interpolate
- Texture cache
  - Can be accessed with 2d float index
  - Cache includes dedicated hardware to implement bilinear interpolation
  - Can be configured to clamp, border, wrap or mirror at texture boundary
  - Hardware support to decompress compressed textures on cache miss
  - Custom hardware-specific storage layout (blocked/Morton) to exploit 2d locality
  - Triangle/pixel enumeration is tiled for locality

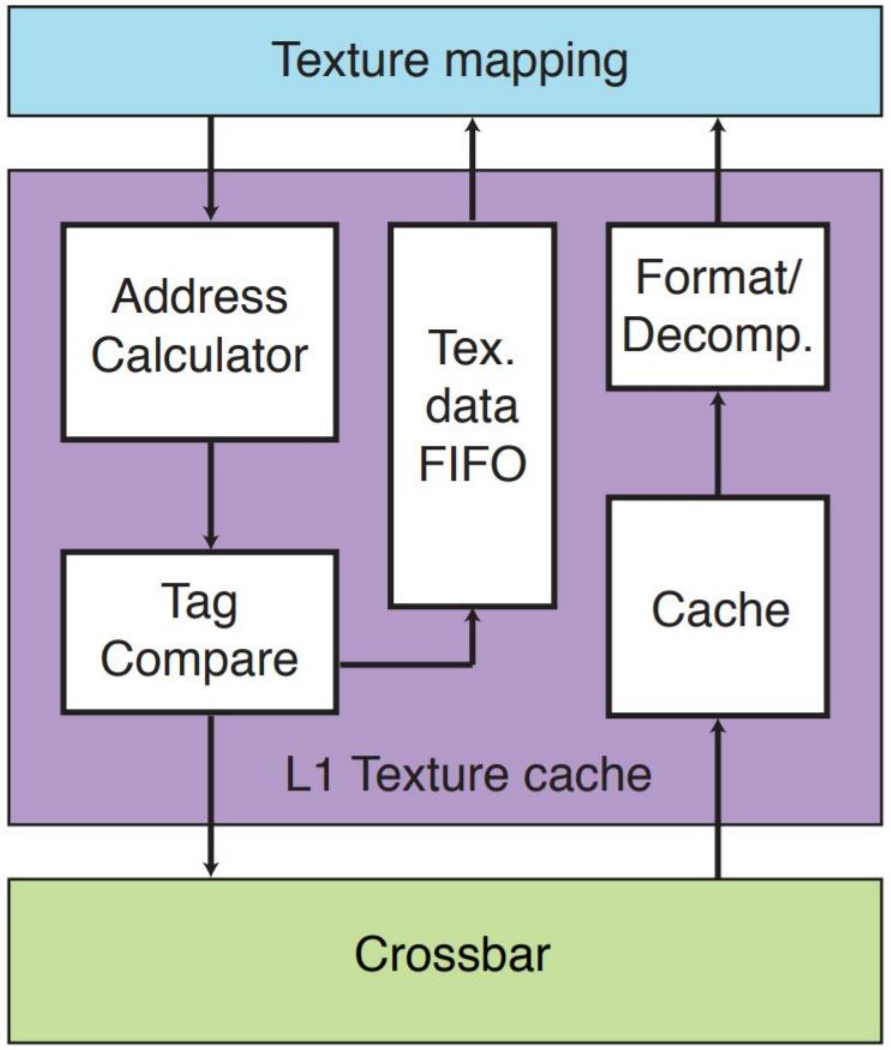
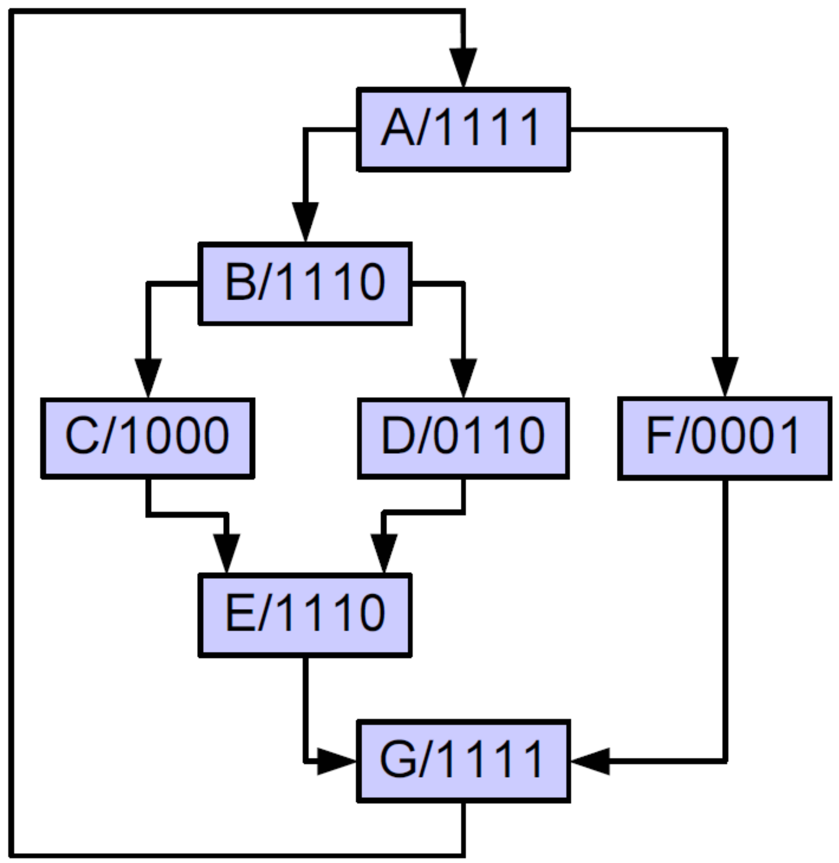


Fig. 5. An overview of a texture cache architecture. The texture mapping unit provides texture coordinates for which a memory address is calculated. The address is sent to the tag compare to determine if the data is in the cache. If the data isn't in the cache, a request is sent via the crossbar to the L2 cache. Any state associated with the original request is sent into a FIFO to return to the texture mapping unit with the texel data. Once the data arrives in the cache, or is already available in the cache, it is returned to the texture mapping unit. If the data is compressed, it is decompressed and any formatting that is required is done.

For more details see **Texture Caches,** Michael Doggett, http://fileadmin.cs.lth.se/cs/Personal/Michael_Doggett/pubs/doggett12-tc.pdf

Nested if-then-else execution



(a) Example Program

	Ret./Reconv. PC	Next PC	Active Mask
	-	G	1111
	G	F	0001
TOS →	G	B	1110

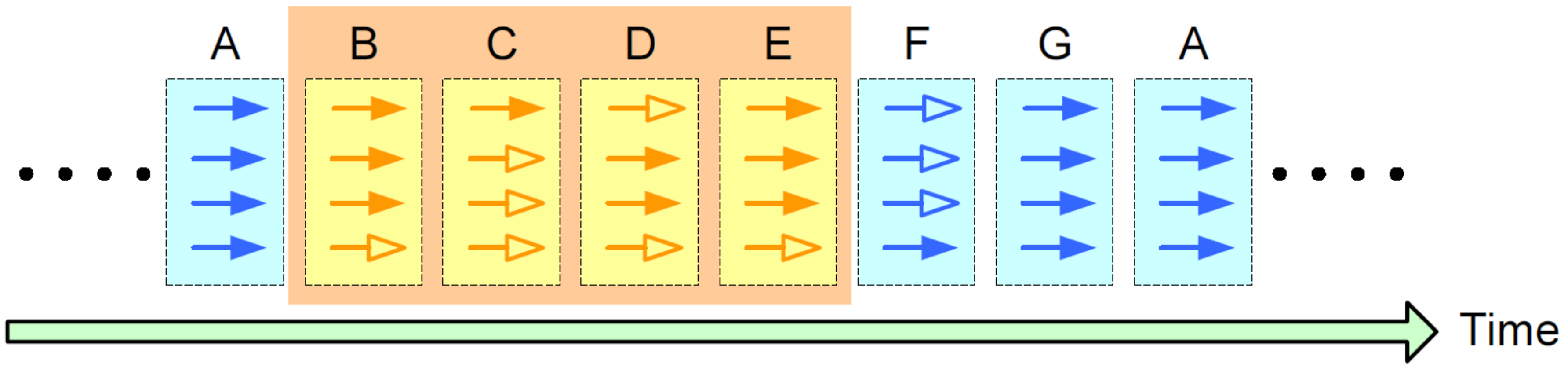
(c) Initial State

	Ret./Reconv. PC	Next PC	Active Mask	
	-	G	1111	
	G	F	0001	
	G	E	1110	(i)
	E	D	0110	(ii)
TOS →	E	C	1000	(iii)

(d) After Divergent Branch

	Ret./Reconv. PC	Next PC	Active Mask
	-	G	1111
	G	F	0001
TOS →	G	E	1110

(e) After Reconvergence



(b) Re-convergence at Immediate Post-Dominator of B

Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. ***Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow*** *(MICRO 2007)*

Student question: threads, lanes, warps, blocks

- **1 warp = many threads ; 1 thread = like a lane in CPU so handles one thing at a time ; many threads --> fit in a block (so many warps --> 1 block?) ; many blocks --> grid ; many grids --> gpu**
- **Q2: 1 warp --> has several 32 wide (does wide just mean bits?) vector instructions (so in one instruction we could add together 2 1d arrays of size 32 bits?)**
- **Q4: Also, each warp has "32 wide SIMD vector instructions" and GPUs also follow the SIMT model - so we have SIMT as the process describing what's happening with all the warps (ie FGMT threads) and SIMD describing what's happening with every instruction handled by one thread (lane like thing inside a warp) - is that correct?**

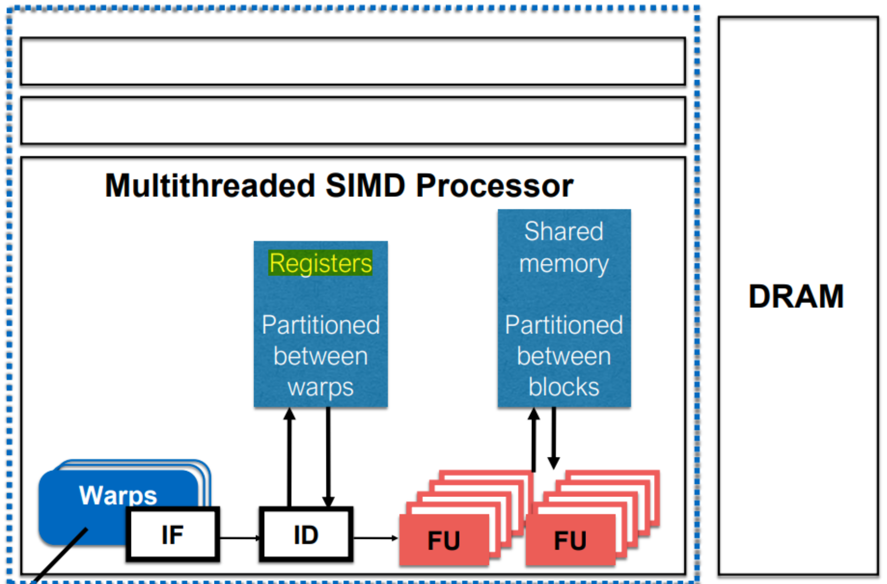
Each warp is an FGMT thread of 32-wide SIMD instructions. The source code for a CUDA kernel describes what a CUDA "thread" does; each CUDA thread is executed in one lane of the SIMD instruction sequence executed by the warp.

A block is a set of warps that are all running on the same core using FGMT (NVIDIA uses the term "SM" for core).

Because all the warps in block run on the same SM, they can actually cooperate with one another using (what NVIDIA calls) "shared memory". So you might have a phase where all the threads in a block load data into shared memory, tyhen they have a barrier, then they can all, in parallel, use that data.

Student question: register file partitioning

Running DAXPY on a GPU


Multithreaded SIMD Processor
Registers
Partitioned between warps
Shared memory
Partitioned between blocks
Warps
IF
ID
FU
FU
DRAM

- Each warp executes 32 CUDA threads in SIMD lock-step
- Each CUDA thread executes one instance of the kernel
- Each SM is shared by many warps (possibly from the same or different blocks)

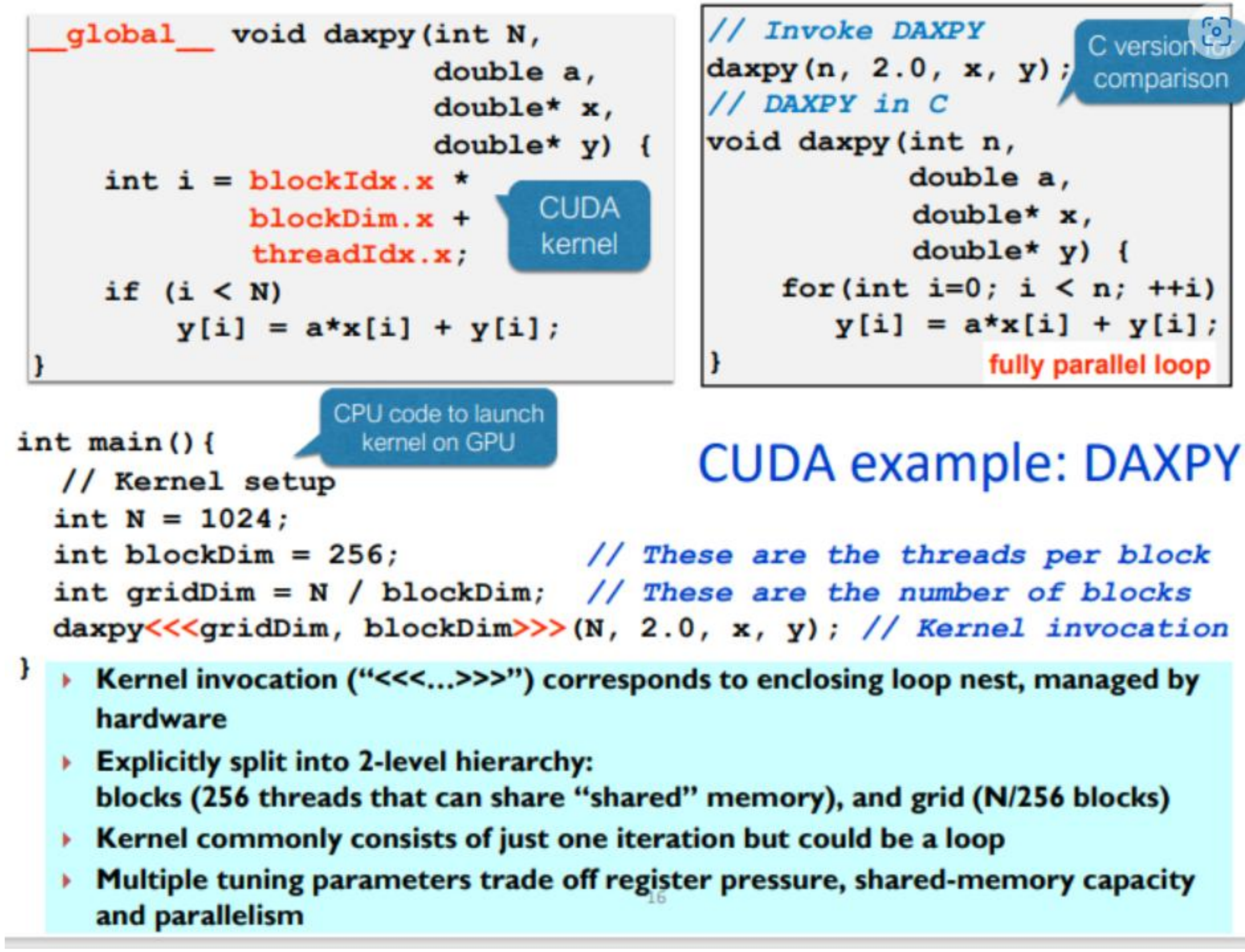
Q: Registers are partitioned by "ID" stage in the lecture slides (is that instruction decode?)

- Each thread refers to the logical register set R0, R1, R2...RN (where N depends on how the register file is partitioned - smaller N means we can have more FGMT warps sharing the SM's fixed physical register file.

- This mapping, from logical register to physical register, is done in the instruction decode (ID) stage of the pipeline.

- **At the risk of being confusing**: looking at this using CPU/SIMD terminology, each FGMT thread (aka *warp)* refers to that FGMT thread's logical register set of 32-wide vector registers R0, R1, R2...RN. Each SIMT thread (aka *thread in that warp)* refers to its own *lane* of these vector registers. So the kth CUDA thread in a 32-wide warp (0<k<31) refers to lane_k of the vector registers R0, R1 etc.

Student question: Fixing the upper bound of a loop in a CUDA kernel



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

```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n,
           double a,
           double* x,
           double* y) {
    for(int i=0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
```

CUDA example: DAXPY

```
int main(){
  // Kernel setup
  int N = 1024;
  int blockDim = 256;        // These are the threads per block
  int gridDim = N / blockDim; // These are the number of blocks
  daxpy<<<gridDim, blockDim>>>(N, 2.0, x, y); // Kernel invocation
}
```

- Kernel invocation ("<<<...>>>") corresponds to enclosing loop nest, managed by hardware
- Explicitly split into 2-level hierarchy: blocks (256 threads that can share "shared" memory), and grid (N/256 blocks)
- Kernel commonly consists of just one iteration but could be a loop
- Multiple tuning parameters trade off register pressure, shared-memory capacity and parallelism

Q: "why do we need to do the i < N check?"

A: The "i<N" check is there to handle the case when N is not a round number. It clearly is a very round number in the example - but the daxpy kernel is written to handle the case when it isn't. The warp that executes the last iteration is 32-threads wide - but not all the threads in that warp will have work to do.

Student question: predicating nested ifs

How does predication work in something like IA-64 for instructions that depend on more than one branch? Nested if-else statements for example would need 2 1-bit predicate registers

- You're right: with nested if-then, we need two predicate registers:

```
if C1 then
  S1
  if C2 then
    S2
```

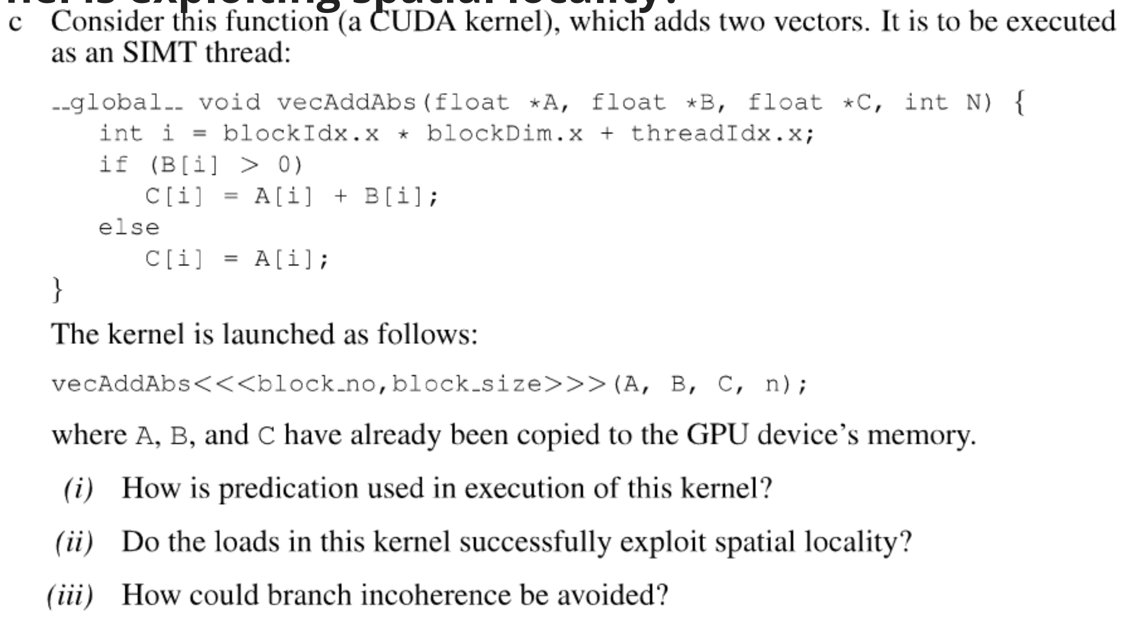
- maps to something like:

```
p1 <- C1
p1: S1
p2 <- C2 & p1
p2: S2
```

- We see this not just with Itanium but also GPUs and in SIMD/vector ISAs.
- There have been proposals for instruction set support for predication in nested control flow - in some sense a hardware predicate stack.
- In a GPU (or running SIMT on a SIMD ISA) you have the same issue that you had with predication of scalar code (as in IA64) - when you think about SIMT execution of the code above (for example if it were a CUDA kernel) you would want two predicate bits per lane.
- So you could imagine having a predicate stack per lane.
- This thinking leads to a world of really interesting stuff. For me the best introduction is the work of Caroline Collange:
- Paper: https://hal.archives-ouvertes.fr/hal-00622654/document
- Slides: https://files.inria.fr/pacap/collange/cours/ada2021_gpu_2.pdf

Hi, I am a bit confused about the question in the title and whether this GPU kernel is exploiting spatial locality?

Student question: spatial locality in SIMT



c Consider this function (a CUDA kernel), which adds two vectors. It is to be executed as an SIMT thread:

```
__global__ void vecAddAbs(float *A, float *B, float *C, int N) {
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   if (B[i] > 0)
      C[i] = A[i] + B[i];
   else
      C[i] = A[i];
}
```

The kernel is launched as follows:

```
vecAddAbs<<<block_no,block_size>>>(A, B, C, n);
```

where A, B, and C have already been copied to the GPU device's memory.

(i) How is predication used in execution of this kernel?

(ii) Do the loads in this kernel successfully exploit spatial locality?

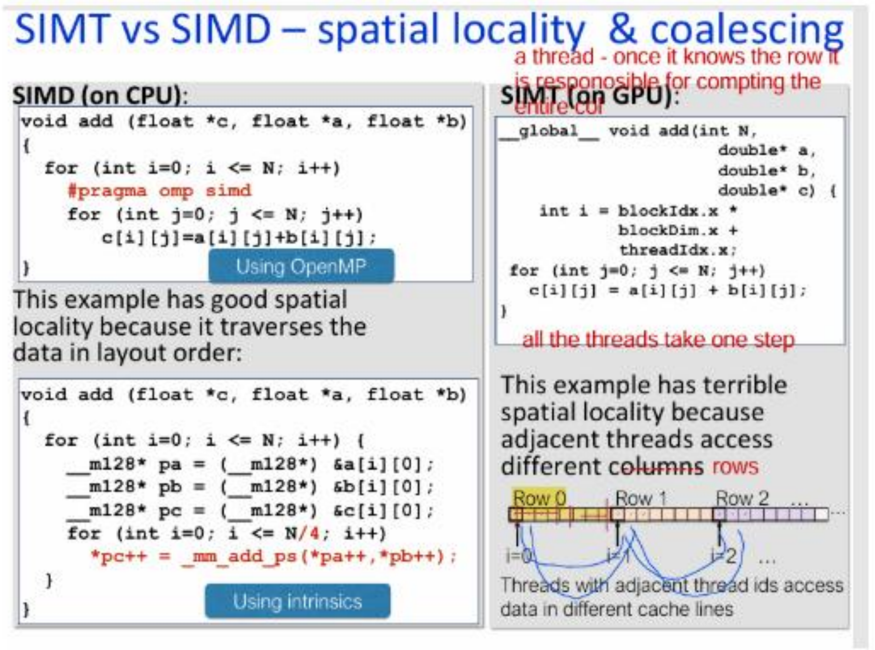
(iii) How could branch incoherence be avoided?

How does predication work in something like IA-64 for instructions that depend on more than one branch? Nested if-else statements for example would need 2 1-bit predicate registers

- This is part of the 2019-20 paper, Q4.
- Yes the loads in this GPU kernel benefit from spatial locality (the CUDA people call it "coalescing").
- In SIMT, the code of the kernel is "vector expanded" - every instruction is expanded to operate on vector registers (perhaps 32-wide). Adjacent iterations of the kernel run in adjacent "lanes" of these vector instructions. So when lane0 accesses A[i], lane1 will access A[i+1], etc. So the vector of load operations will access adjacent locations in memory - giving us spatial locality,
- The kernel launch mechanism launches execution of the kernel as grid of blocks - each block is executed by a set of warps (aka threads), each of which operates as a sequence of vector instructions, executing 32 iterations of the kernel at a time.

Student question: SIMT

1



SIMT vs SIMD – spatial locality & coalescing

a thread - once it knows the row it is responosible for compting the entire col

SIMD (on CPU):

```
void add (float *c, float *a, float *b)
{
  for (int i=0; i <= N; i++)
    #pragma omp simd
    for (int j=0; j <= N; j++)
       c[i][j]=a[i][j]+b[i][j];
}
```

Using OpenMP

This example has good spatial locality because it traverses the data in layout order:

```
void add (float *c, float *a, float *b)
{
  for (int i=0; i <= N; i++) {
    __m128* pa = (__m128*) &a[i][0];
    __m128* pb = (__m128*) &b[i][0];
    __m128* pc = (__m128*) &c[i][0];
    for (int i=0; i <= N/4; i++)
      *pc++ = _mm_add_ps(*pa++,*pb++);
  }
}
```

Using intrinsics

SIMT (on GPU):

```
__global__ void add(int N,
                    double* a,
                    double* b,
                    double* c) {
   int i = blockIdx.x *
           blockDim.x +
           threadIdx.x;
 for (int j=0; j <= N; j++)
   c[i][j] = a[i][j] + b[i][j];
}
```

all the threads take one step

This example has terrible spatial locality because adjacent threads access different ~~columns~~ rows

Threads with adjacent thread ids access data in different cache lines

considering this slide, for SIMT - each CUDA thread operates on a row of the matrix, and all the CUDA threads in a warp move one step together, do there threads move only one step at a time - adding up values in one col of A. B and storing the result in C or can they use vector insturctions to operate on multiple cols at once like a group of 4 cols and move 4 steps ahead together - like an SIMD lock step?

✓

In CUDA the word "thread" is (confusingly) used to refer to the sequence of operations performed by a *lane* of the warp. The word "warp" is used to refer to the sequence of SIMD instructions executed by the hardware*.

So the top-right code shows the source code for a CUDA thread. You can visualise how it's compiled by thinking of the assembly code for the whole function, where every instruction is expanded to be a 32-wide SIMD instruction.

The important (and confusing) thing is the way the assignment to i is handled: each lane gets a differen value of i - adjacent lanes get adjacent values of i.

(this is achieved using a special instruction that reads a magic register: threadIdx is provided in"%tid" - see CS 641 Lecture).

(* To complicate things further, the warps are dynamically scheduled using fine-grain multi-threading, but this is not relevant for this question)