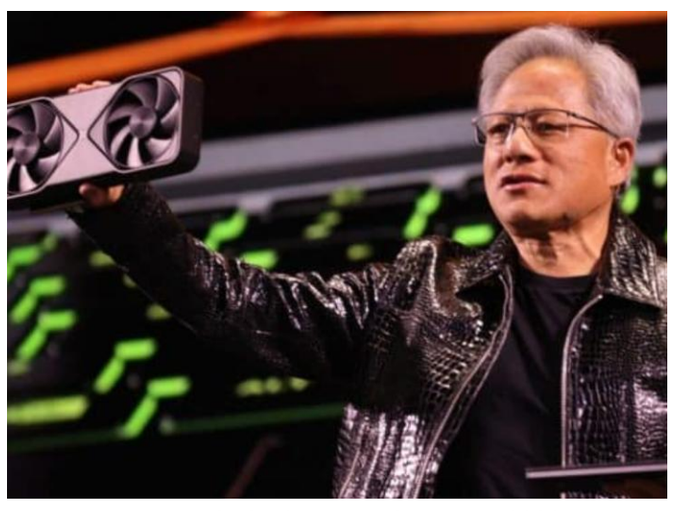
Imperial College London

Advanced Computer Architecture
Chapter 9

Data-Level Parallel Architectures: GPUs



Lisa Su, CEO of AMD at CES 2025 (image: Lisa Su)


Jensen Huang, CEO of NVIDIA, Computex 2026 keynote

+Intel, ARM, Apple, Huawei, Qualcomm, Moore Machines, etc

November 2026
Paul Kelly

These lecture notes are partly based on:

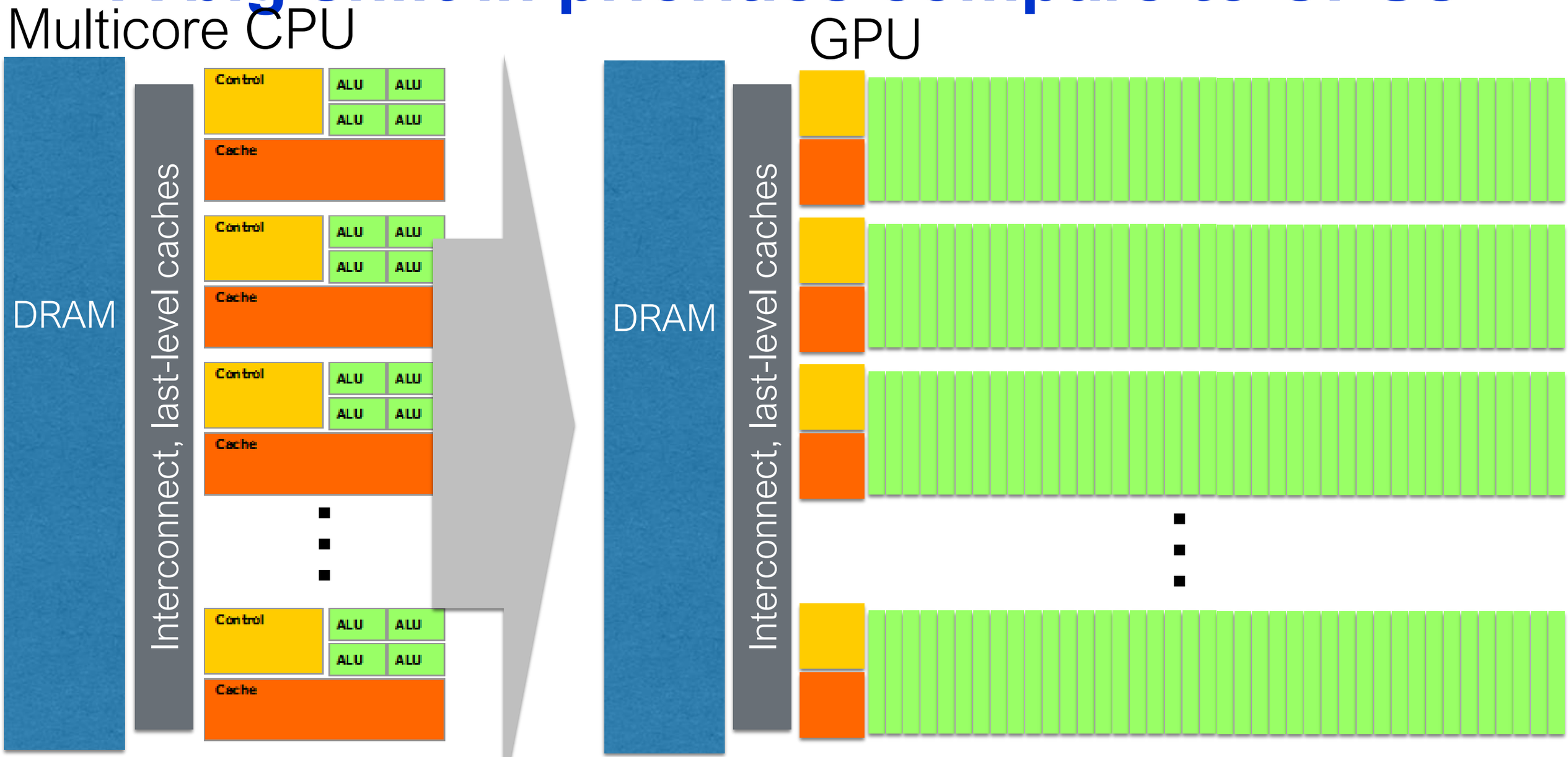
- **Contributions to the lecture slides from Luigi Nardi (postdoc at Imperial and Stanford, now academic at Lund, Sweden), Fabio Luporini (Imperial PhD, postdoc, now CTO, DevitoCodes), and Nicolai Stawinoga (Imperial PhD, postdoc, now at NVIDIA)**
- **the course text, Hennessy and Patterson's Computer Architecture (7th 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:
  - **Use lots of threads** (FGMT or SMT) to hide main memory latency
  - ***Never* speculate**: there is always another thread waiting with work you *know* you have to do
    - No speculative branch execution, no branch prediction
  - Control is at a premium: **Turing tax avoidance**:
    - *Wide* **SIMD**
    - How to launch hundreds or thousands of threads?
    - What if they **branch in different directions**?
    - What if they access different data? **memory bandwidth**?
      - (what if they all access different addresses?)
      - (what if they all access the same bank?)

- **This is the "manycore" world**
- **Initially driven by the gaming market, then computational science**
- **And now overwhelmingly dominant in AI**

A big shift in priorities compare to CPUs



- Similar number of cores per chip, but simpler (no need for OoO)
- But with wide SIMD (eg 32) – and often support for matrices as well as vectors
- And massively multi-threading (eg 48 threads per core)
- But less cache per core – the multithreading hides the memory latency

The plan for this chapter:

1

- Start with architecture of early, simple GPUs
  - Which were designed for graphics
  - But used also for general-purpose computing and then AI
- **We focus on understanding GPUs as a part of the *CPU* architectural design space**
- We have covered everything you need for this
  - **Fine-grained Multithreading (FGMT)**
  - **Vector instructions (SIMD)**
  - **Caches, pipelines etc**
- We won't dive into graphics-specific features

2

- We will look at the programming model (CUDA) – because it helps understand the architecture. But this is not a GPU programming course
- We will use NVIDIA's terminology – but it *is* confusing:
  - **CPU terms**: core, thread, SIMD, lane
  - **NVIDIA terms**: SM, warp, SIMT, thread

3

- Once we have covered the key GPU ideas, we will look at:
  - Some of the implementation concepts
  - Some more recent extensions, notably for matrix multiply

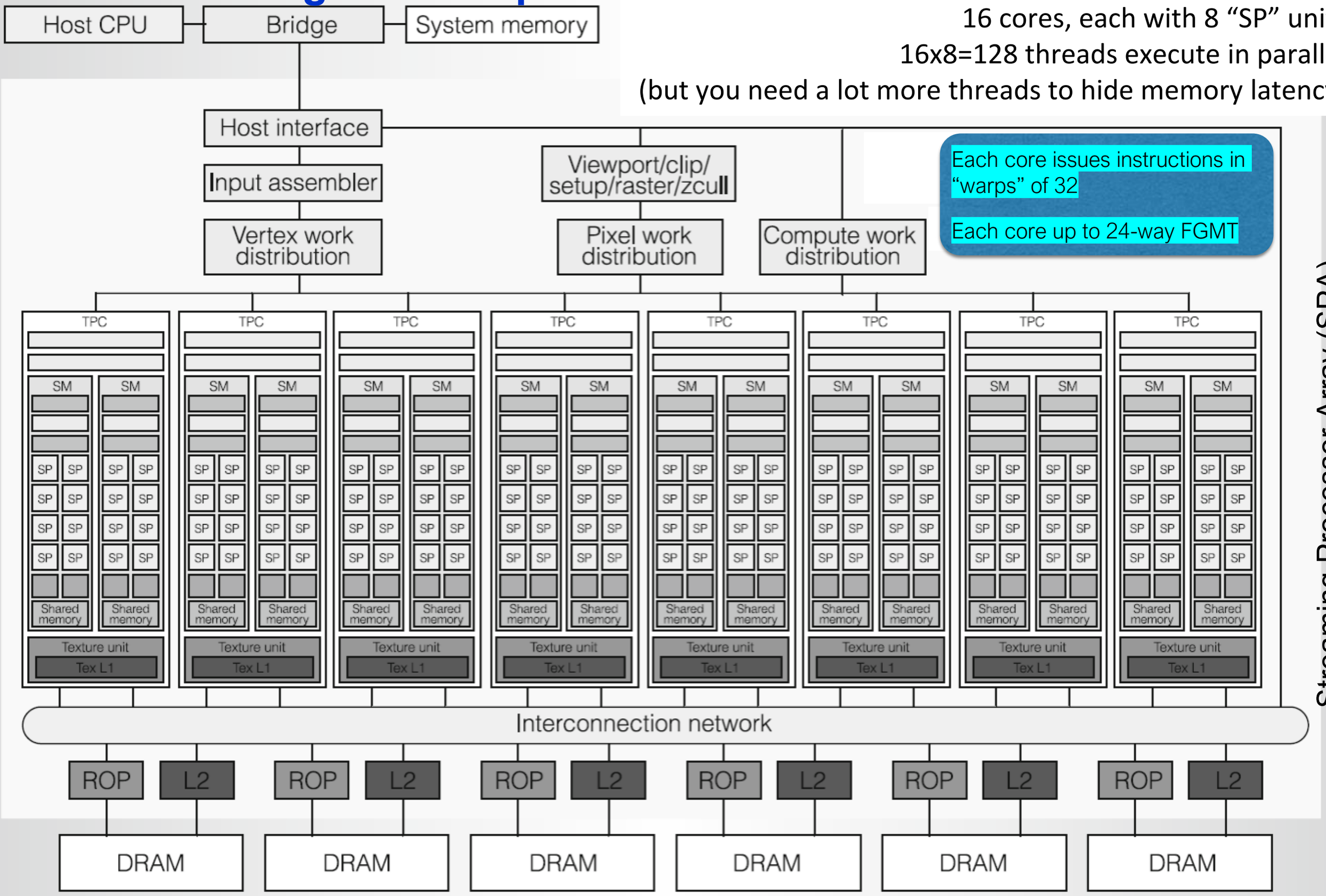
A landmark design that maps out the idea:

NVIDIA G80 (2006)

16 cores, each with 8 “SP” units

16x8=128 threads execute in parallel

(but you need a lot more threads to hide memory latency)



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

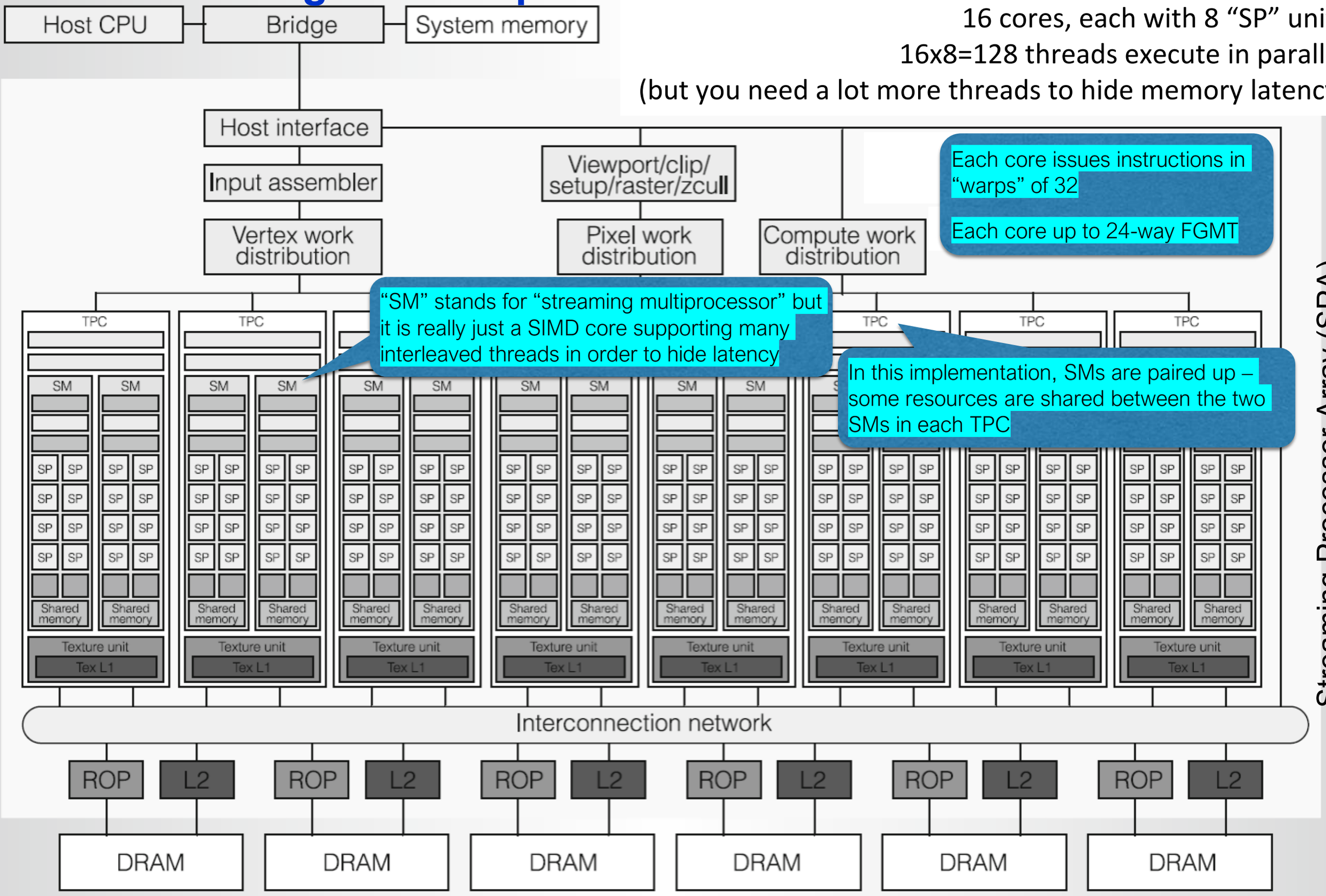
A landmark design that maps out the idea:

NVIDIA G80 (2006)

16 cores, each with 8 “SP” units

16x8=128 threads execute in parallel

(but you need a lot more threads to hide memory latency)



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

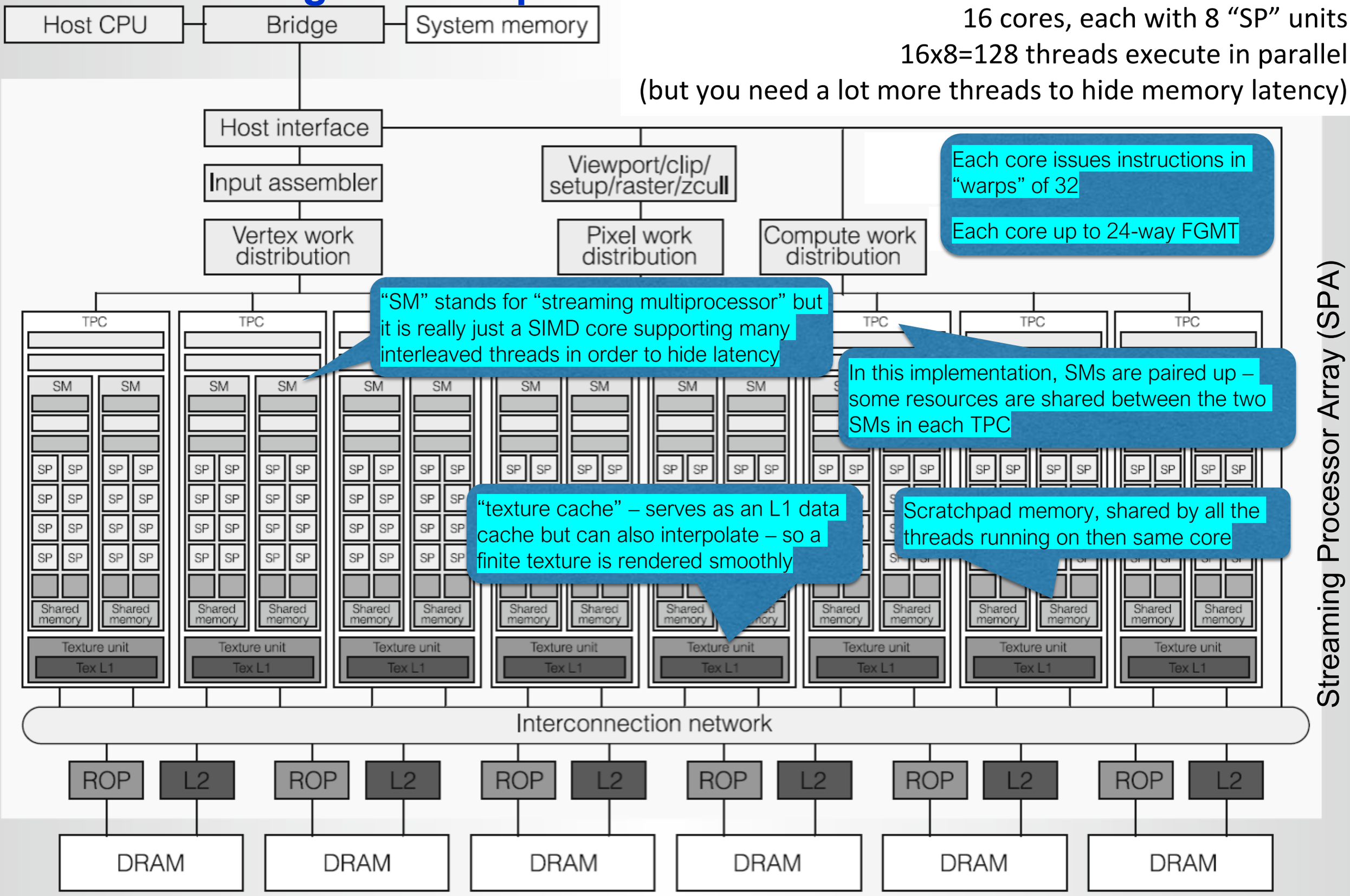
A landmark design that maps out the idea:

NVIDIA G80 (2006)

16 cores, each with 8 “SP” units

16x8=128 threads execute in parallel

(but you need a lot more threads to hide memory latency)



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

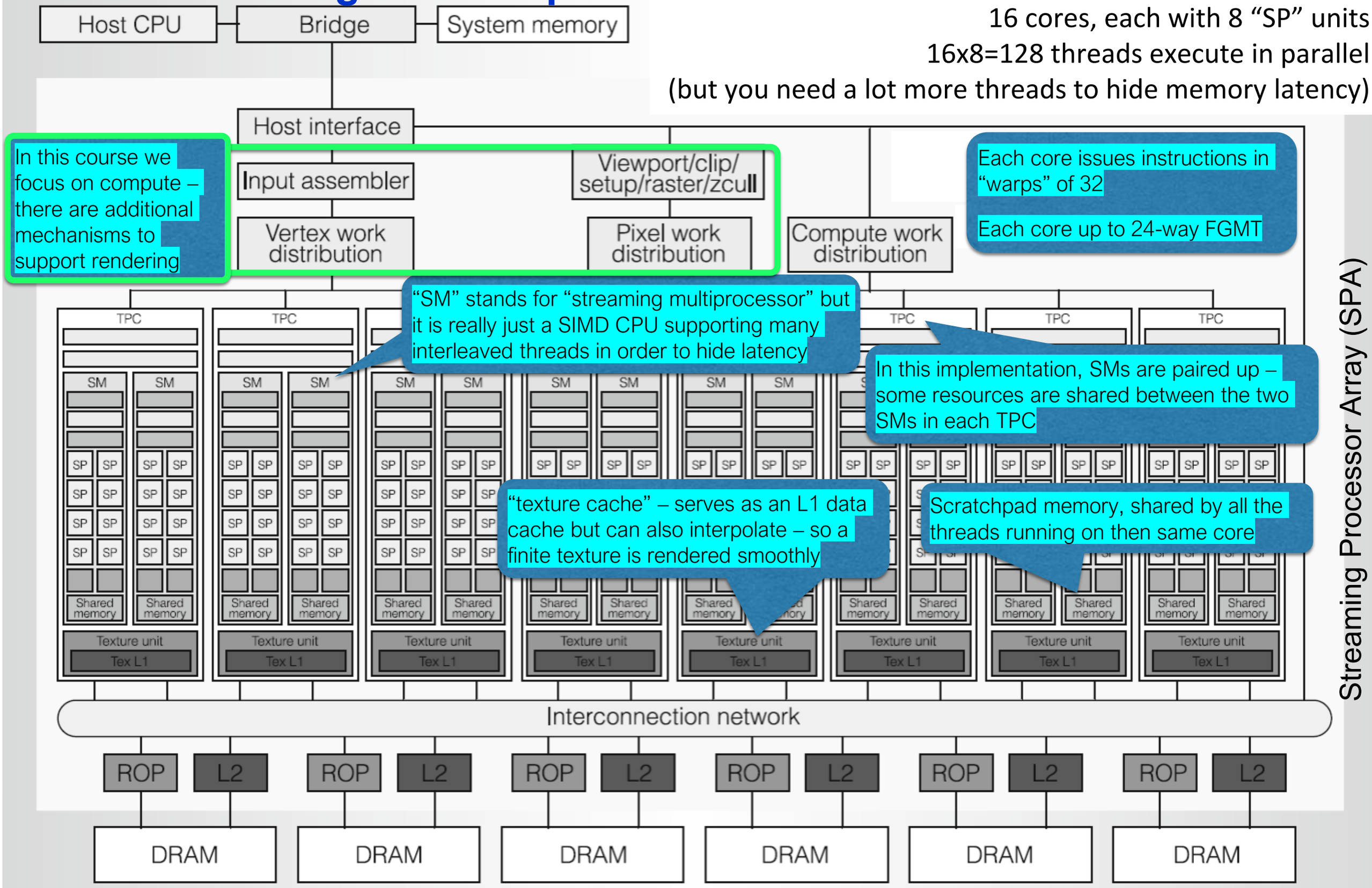
A landmark design that maps out the idea:

NVIDIA G80 (2006)

16 cores, each with 8 “SP” units

16x8=128 threads execute in parallel

(but you need a lot more threads to hide memory latency)



In this course we focus on compute – there are additional mechanisms to support rendering

Each core issues instructions in “warps” of 32

Each core up to 24-way FGMT

“SM” stands for “streaming multiprocessor” but it is really just a SIMD CPU supporting many interleaved threads in order to hide latency

In this implementation, SMs are paired up – some resources are shared between the two SMs in each TPC

“texture cache” – serves as an L1 data cache but can also interpolate – so a finite texture is rendered smoothly

Scratchpad memory, shared by all the threads running on then same core

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

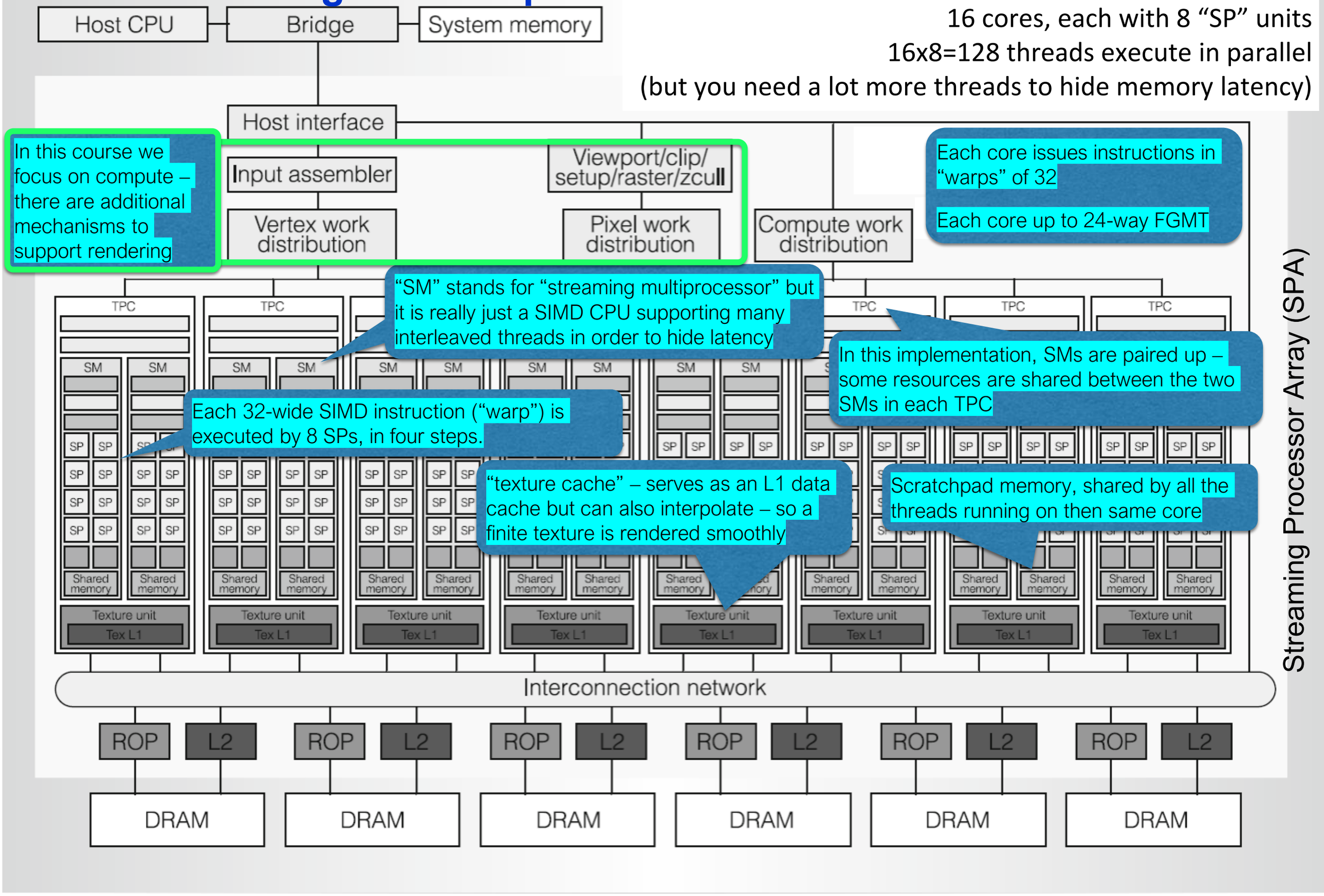
A landmark design that maps out the idea:

NVIDIA G80 (2006)

16 cores, each with 8 “SP” units

16x8=128 threads execute in parallel

(but you need a lot more threads to hide memory latency)



In this course we focus on compute – there are additional mechanisms to support rendering

Each core issues instructions in “warps” of 32

Each core up to 24-way FGMT

“SM” stands for “streaming multiprocessor” but it is really just a SIMD CPU supporting many interleaved threads in order to hide latency

In this implementation, SMs are paired up – some resources are shared between the two SMs in each TPC

Each 32-wide SIMD instruction (“warp”) is executed by 8 SPs, in four steps.

“texture cache” – serves as an L1 data cache but can also interpolate – so a finite texture is rendered smoothly

Scratchpad memory, shared by all the threads running on then same core

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

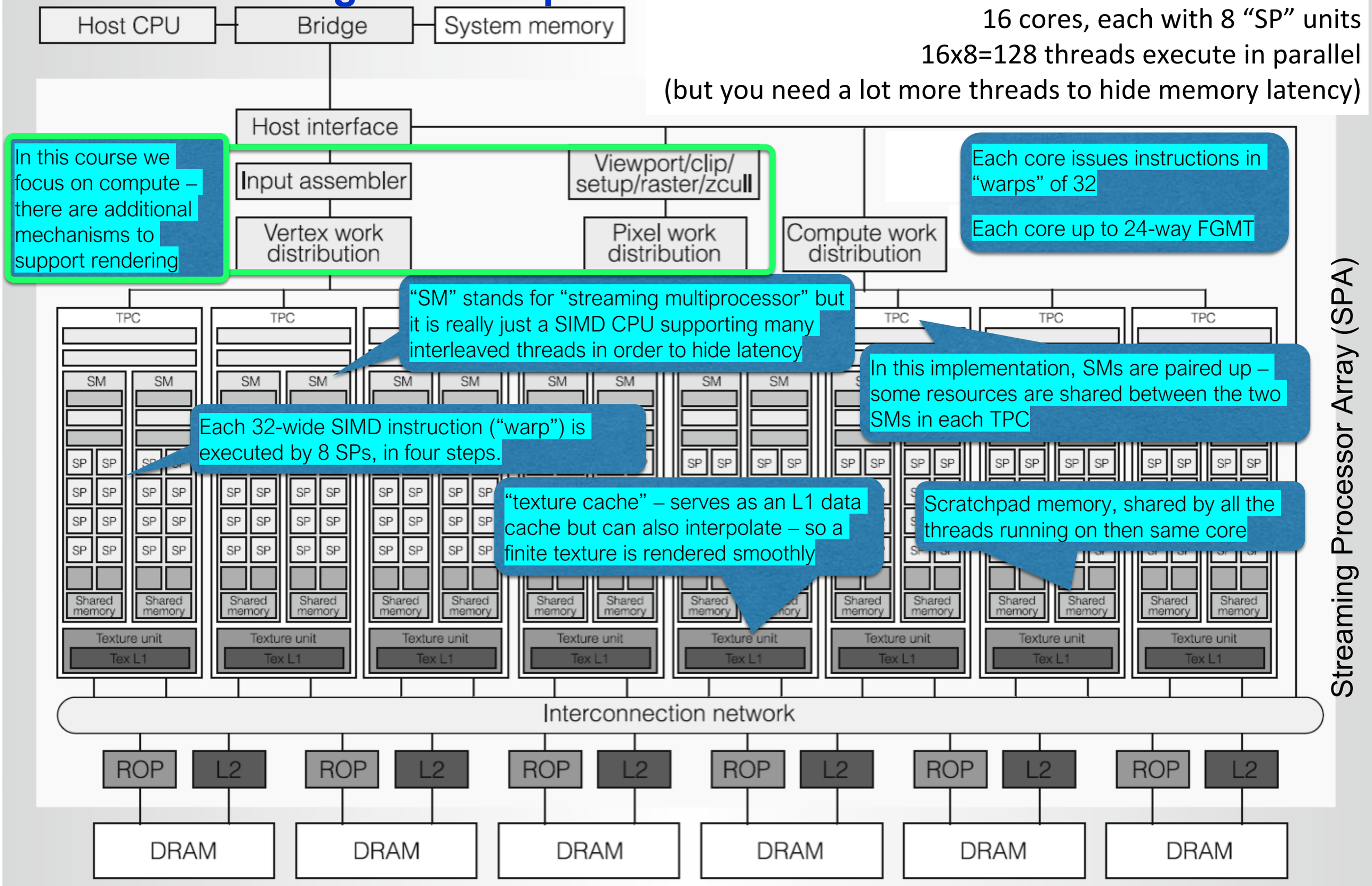
A landmark design that maps out the idea:

NVIDIA G80 (2006)

16 cores, each with 8 “SP” units

16x8=128 threads execute in parallel

(but you need a lot more threads to hide memory latency)



In this course we focus on compute – there are additional mechanisms to support rendering

Each core issues instructions in “warps” of 32

Each core up to 24-way FGMT

“SM” stands for “streaming multiprocessor” but it is really just a SIMD CPU supporting many interleaved threads in order to hide latency

In this implementation, SMs are paired up – some resources are shared between the two SMs in each TPC

Each 32-wide SIMD instruction (“warp”) is executed by 8 SPs, in four steps.

“texture cache” – serves as an L1 data cache but can also interpolate – so a finite texture is rendered smoothly

Scratchpad memory, shared by all the threads running on then same core

Six DRAM channels, each with an L2 data cache. No L2 cache coherency problem, data can be in only one cache. Caches are small

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

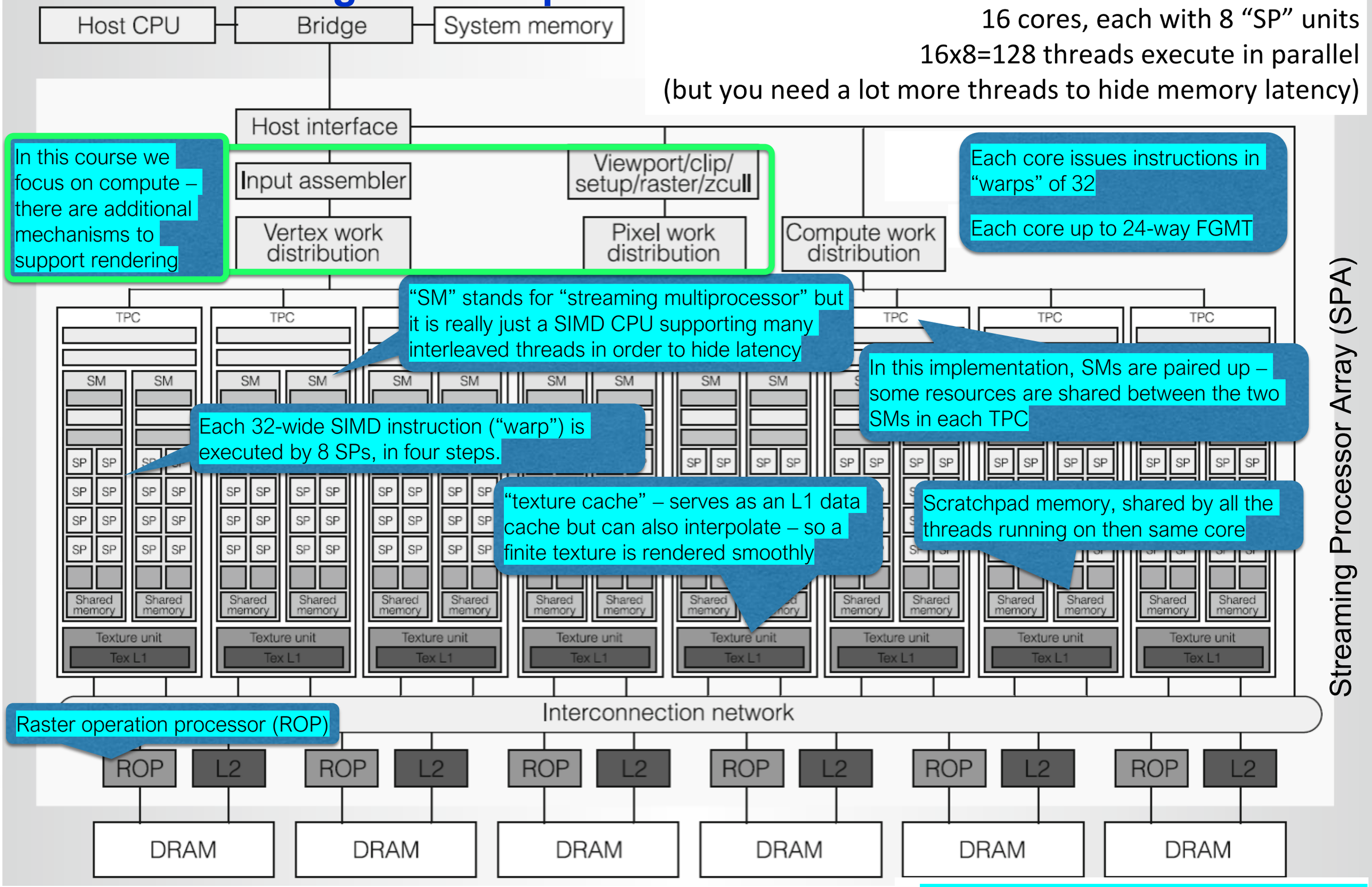
A landmark design that maps out the idea:

NVIDIA G80 (2006)

16 cores, each with 8 “SP” units

16x8=128 threads execute in parallel

(but you need a lot more threads to hide memory latency)



In this course we focus on compute – there are additional mechanisms to support rendering

Each core issues instructions in “warps” of 32

Each core up to 24-way FGMT

“SM” stands for “streaming multiprocessor” but it is really just a SIMD CPU supporting many interleaved threads in order to hide latency

In this implementation, SMs are paired up – some resources are shared between the two SMs in each TPC

Each 32-wide SIMD instruction (“warp”) is executed by 8 SPs, in four steps.

“texture cache” – serves as an L1 data cache but can also interpolate – so a finite texture is rendered smoothly

Scratchpad memory, shared by all the threads running on then same core

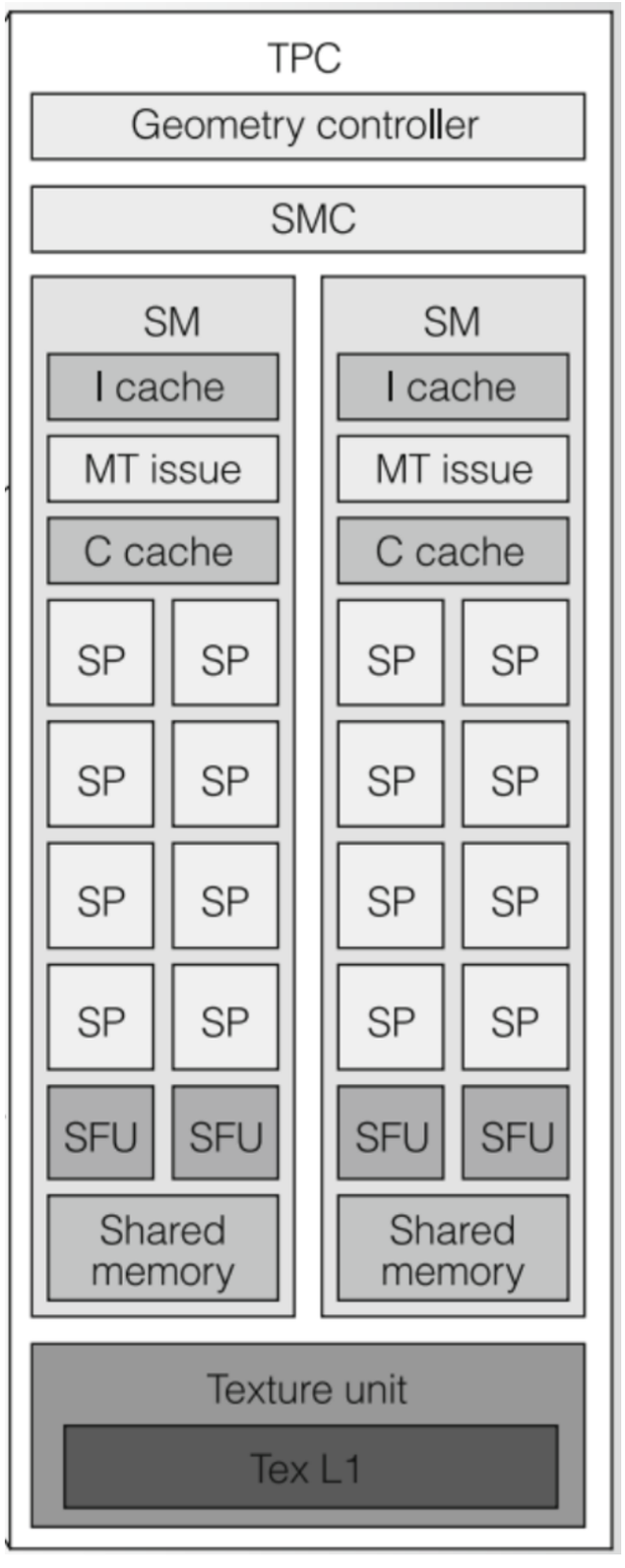
Raster operation processor (ROP)

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 (alpha blending and z-buffering)

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

- A hardware scheduler "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)

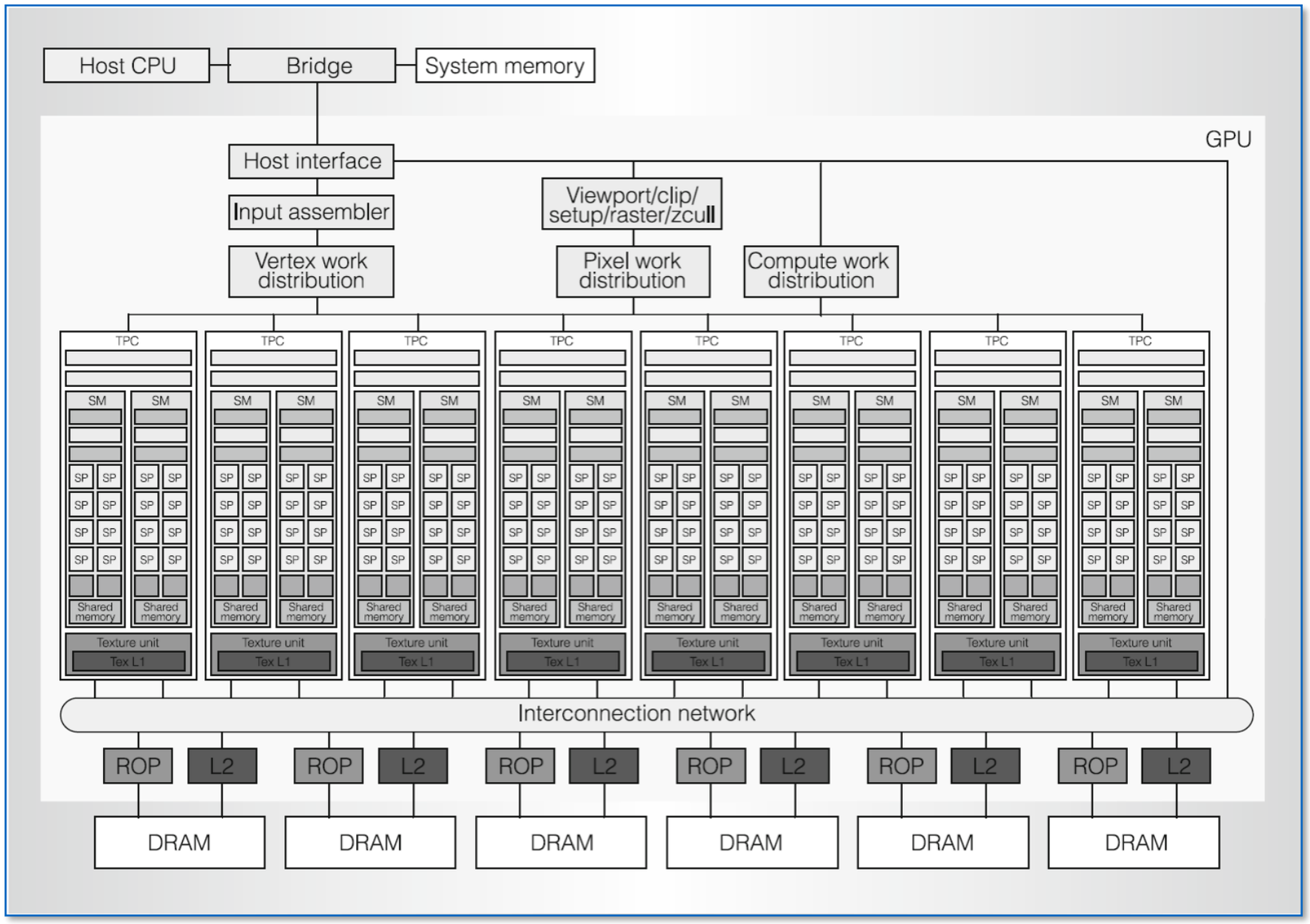
- Each SM has local, explicitly-programmed scratchpad memory
- Different warps on the same SM can share data in this "shared memory"

- 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. L1 caches are invalidated whenever a new kernel is launched)

- 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, z-culling
  - Texture cache, pixel interpolation
  - Z-buffering and alpha-blending (the ROP units, see diagram)

Big idea: multiple levels of parallelism

- **The G80 has 16 cores** (NVIDIA calls them “SMs”)

- **Each core can run up to 32 threads** (NVIDIA: “warps”)
  - The threads are managed by a hardware scheduler
  - The register file is big enough to hold the state of all the threads
  - So there is zero context-switching overhead

- **Each thread** (NVIDIA: “warp”) **executes 32-wide vector(SIMD) instructions**
  - (32-wide SIMD, but on G80 each 32-wide instruction is actually executed in four steps using 8 “SP”s)

- At the bottom of the hierarchy we have the SIMD lanes.
  - **Each lane of the SIMD machine executes a serial “kernel program”** operating on that lane’s scalar data.
  - Confusingly, NVIDIA calls these “SIMT threads”

CPU term	NVIDIA term	How much parallel-ism in G80?
Core	SM	16
thread	warp	24
SIMD lanes	SIMT threads	32
Total parallelism: 16 x 24 x 32		12,288

To exploit all the the GPU’s parallelism, you need *lots* of application parallelism: at least 12K parallel kernel instances

Terminology confusion

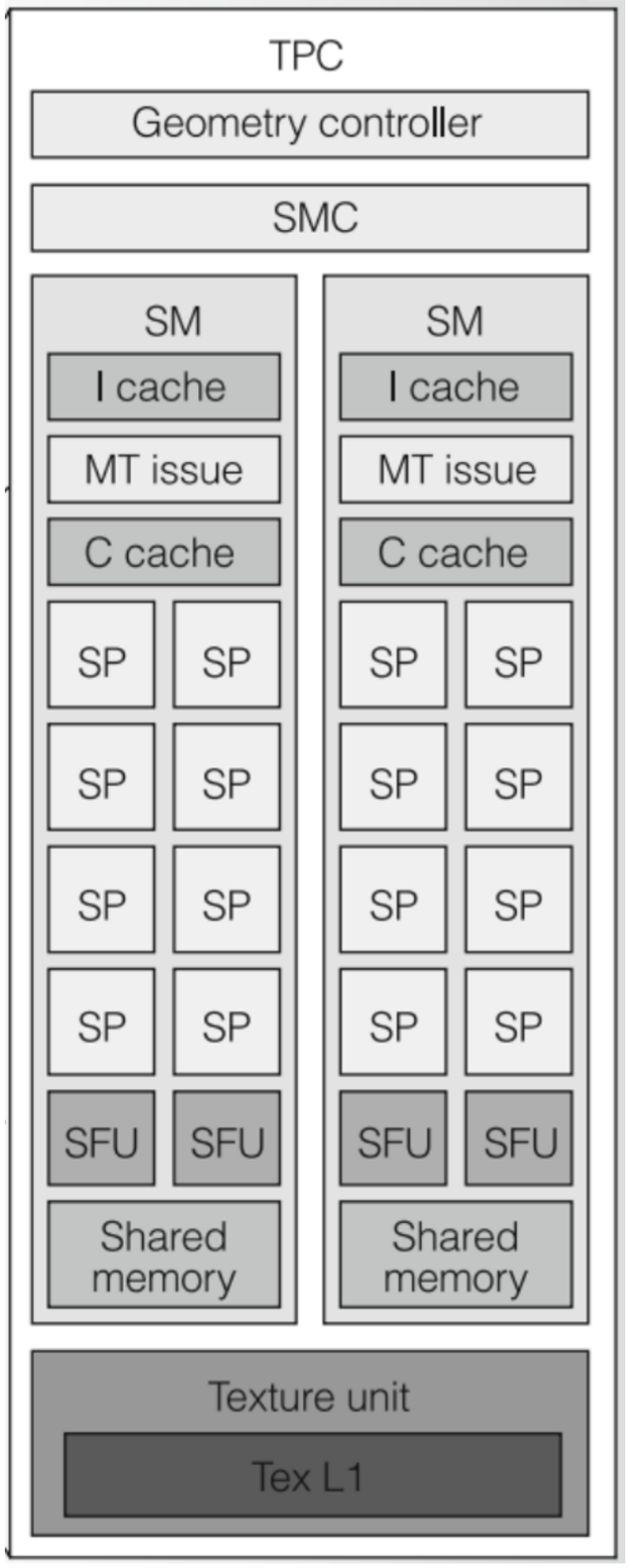
- **NVIDIA originated the general architectural concepts for modern, programmable GPUs**

- **So the dominant terminology is theirs, both the architecture and the programming model, CUDA**

- **OpenCL is an attempt to define a programming model and terminology that is independent of NVIDIA – and hopefully less confusing**

CPU term	NVIDIA term	OpenCL term
Core	SM	Compute Unit
threads	warps	Sub-group, or Wavefront
SIMD lane	SIMT thread	Workitem
Additional terms:		
SIMD ALU lane	CUDA core/SP	Processing element
Group of threads executing the same kernel and locked to the same core	Thread block	Workgroup
Group of threads executing the same kernel	Grid	NDRange
Per-thread storage/context	Registers/ Local memory	Private memory
Scratchpad SRAM	Shared memory	Local memory
	Constant memory	Constant memory
	Texture memory	Image objects

CUDA: using NVIDIA GPUs for general computation



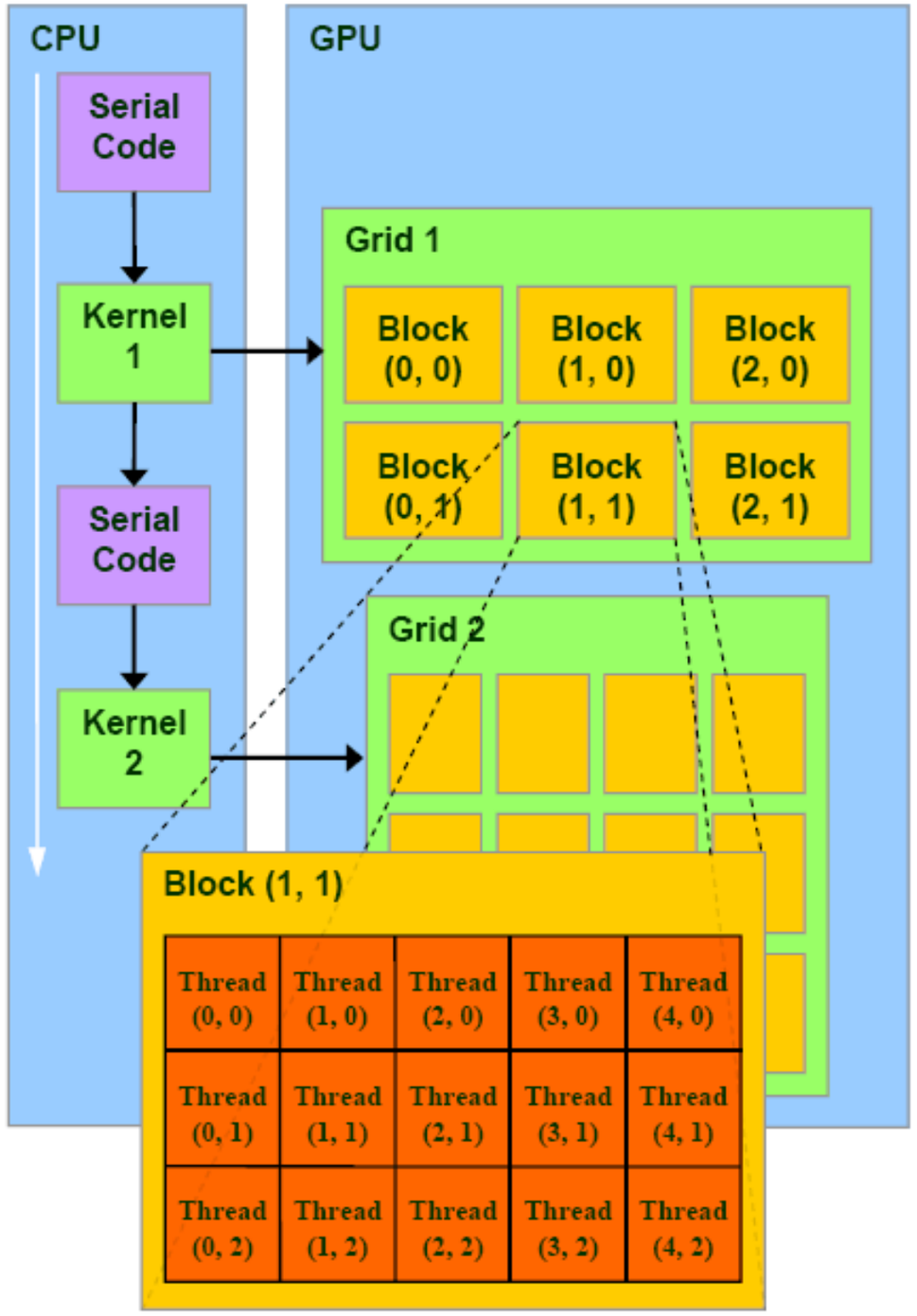
- Designed to do rendering
- Evolved to do general-purpose computing (GPGPU)
  - But to manage thousands of threads, a new programming model is needed, called CUDA (Compute Unified Device Architecture)
  - CUDA is proprietary, but the same model lies behind OpenCL, an open standard with implementations 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

Q: Is this in the exam? A: you need to understand CUDA's hierarchical programming model, but you won't be asked to write CUDA code

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



Source: CUDA programming guide

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

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;

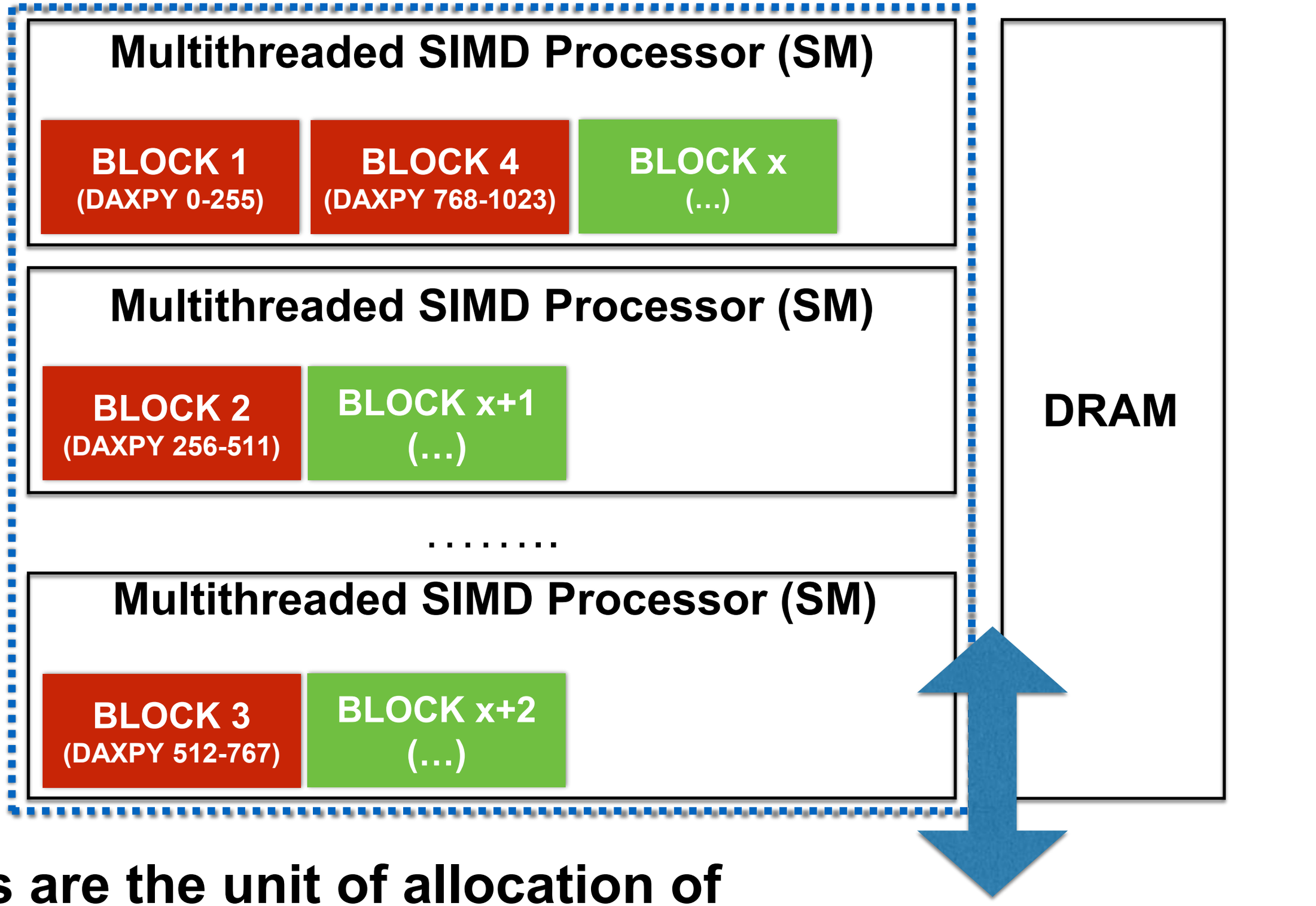
$L_finish:      exit;
```

```
__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

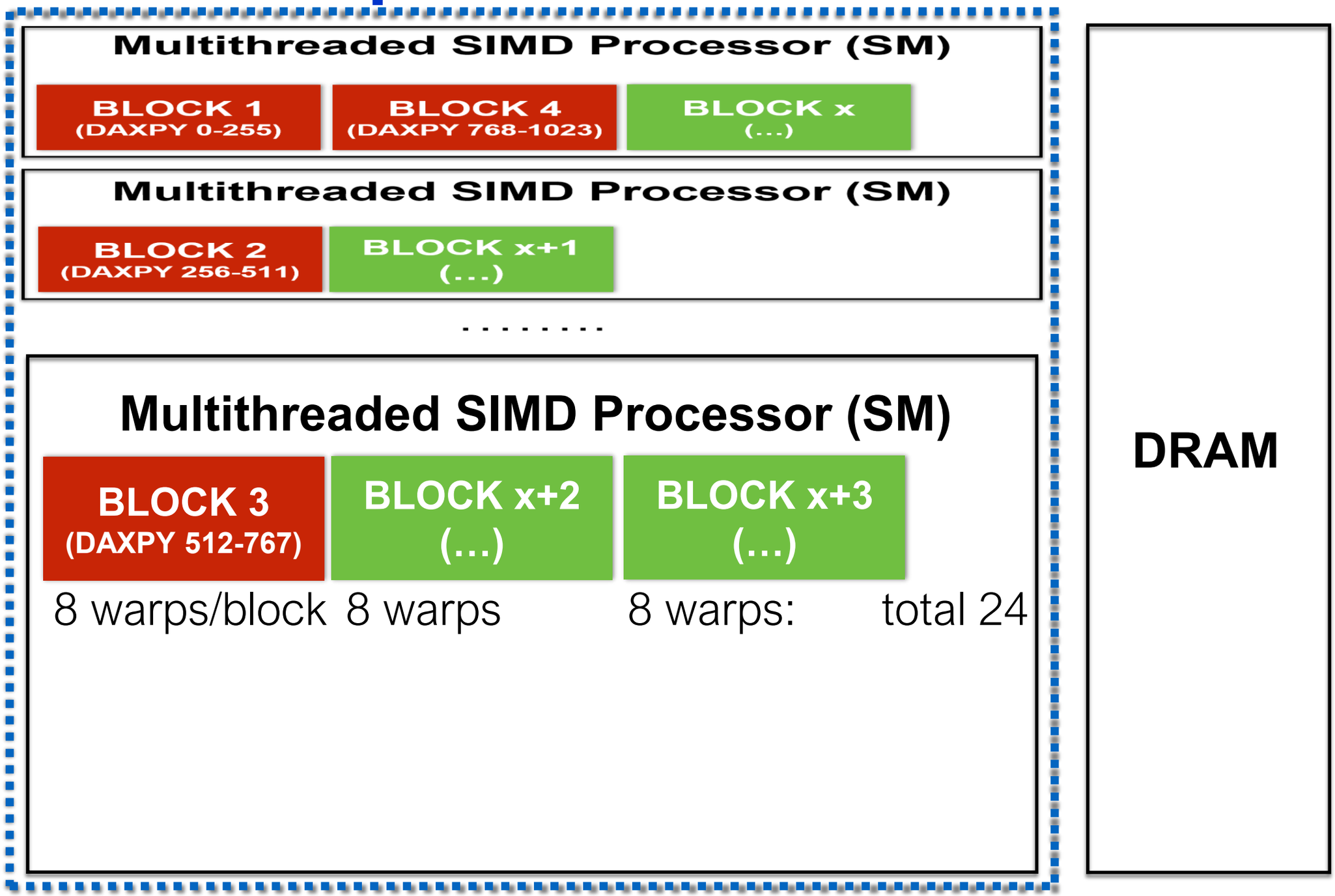
(Joy Lee and others, NVIDIA)

Running DAXPY (N=1024) on a GPU



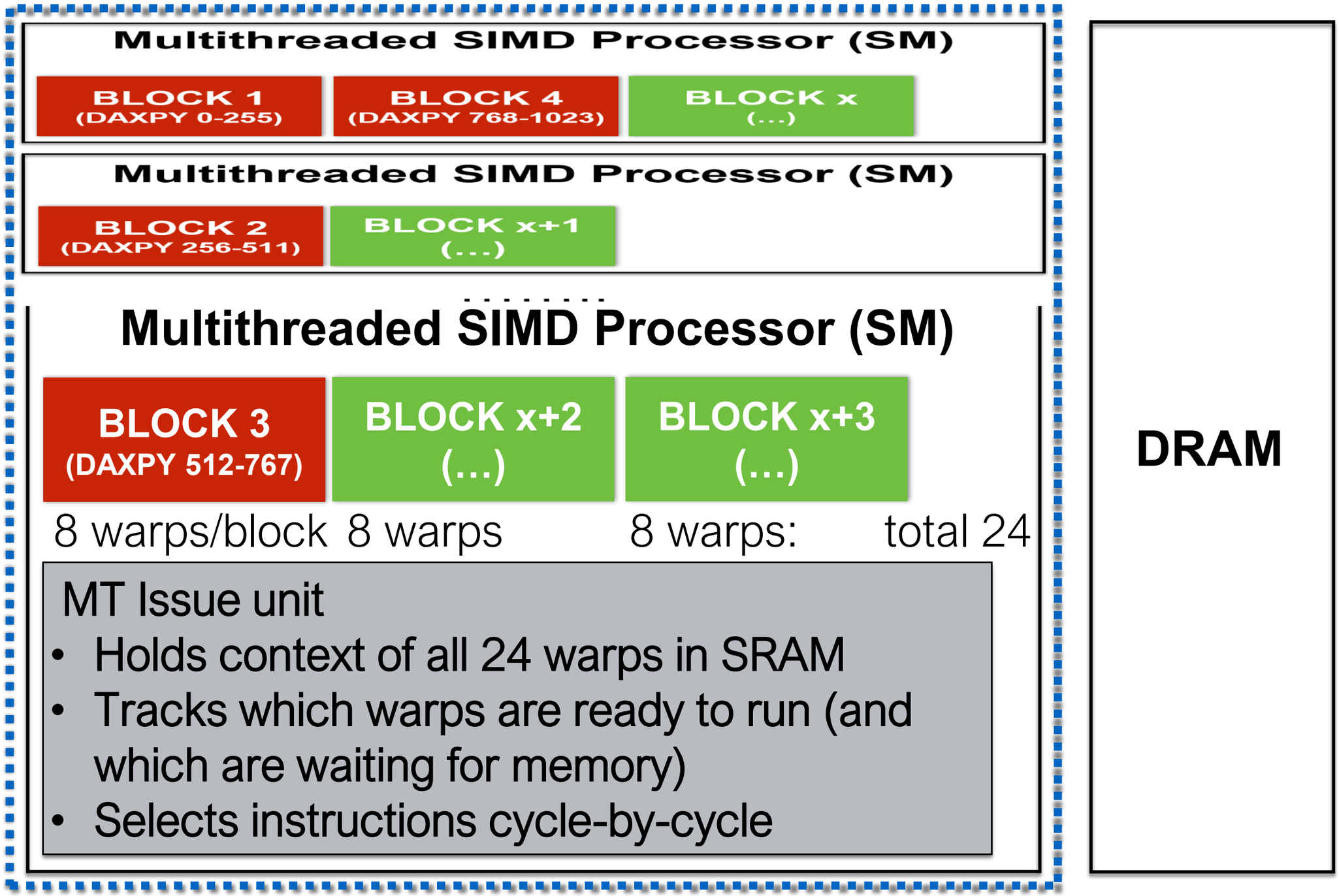
Blocks are the unit of allocation of work to SMs

Blocks of warps are allocated to each SM



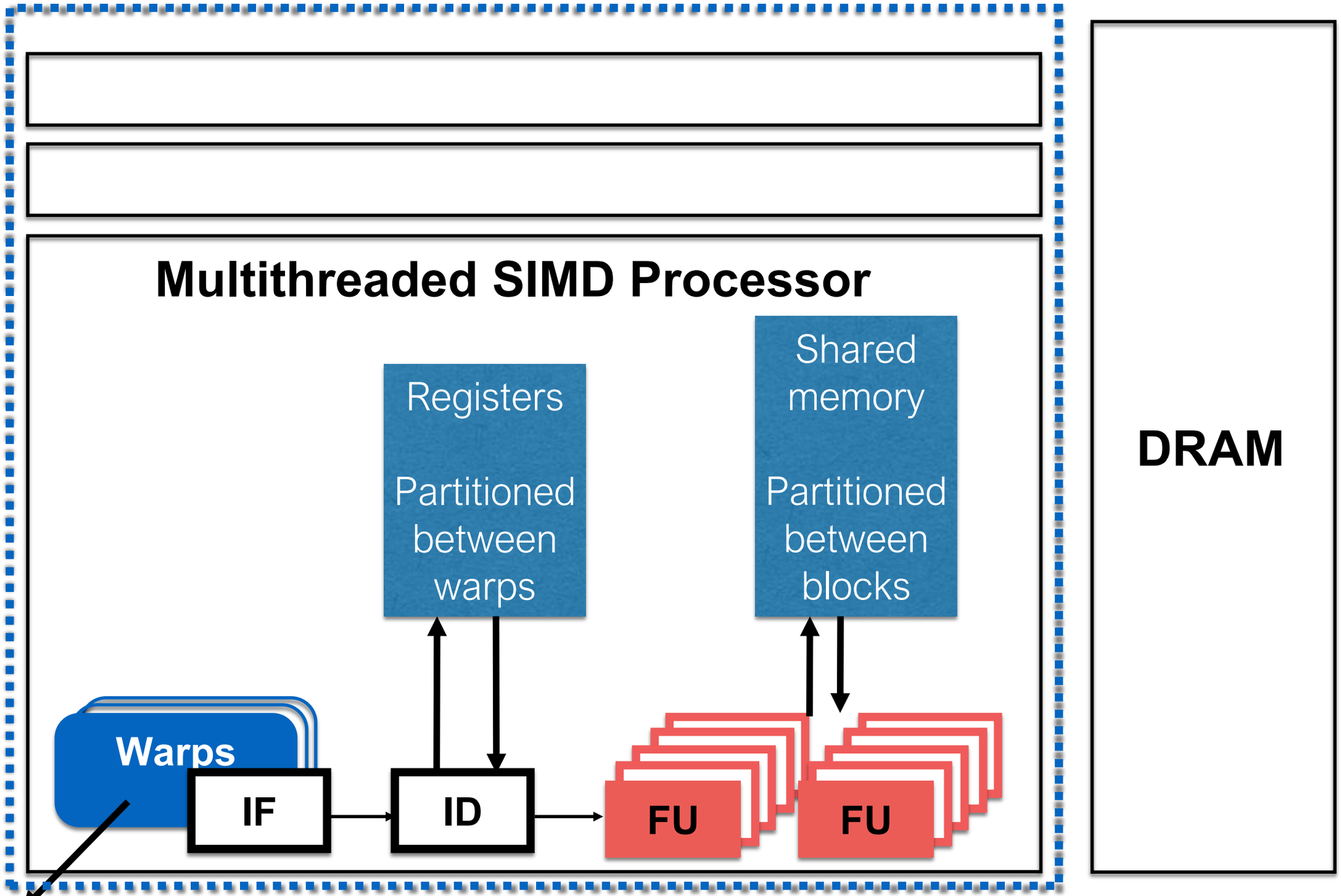
- **Each block may consist of up to 16 warps**
- **So several blocks can be scheduled at the same time**
- **So that up to 24 warps are active**

Blocks of warps are allocated



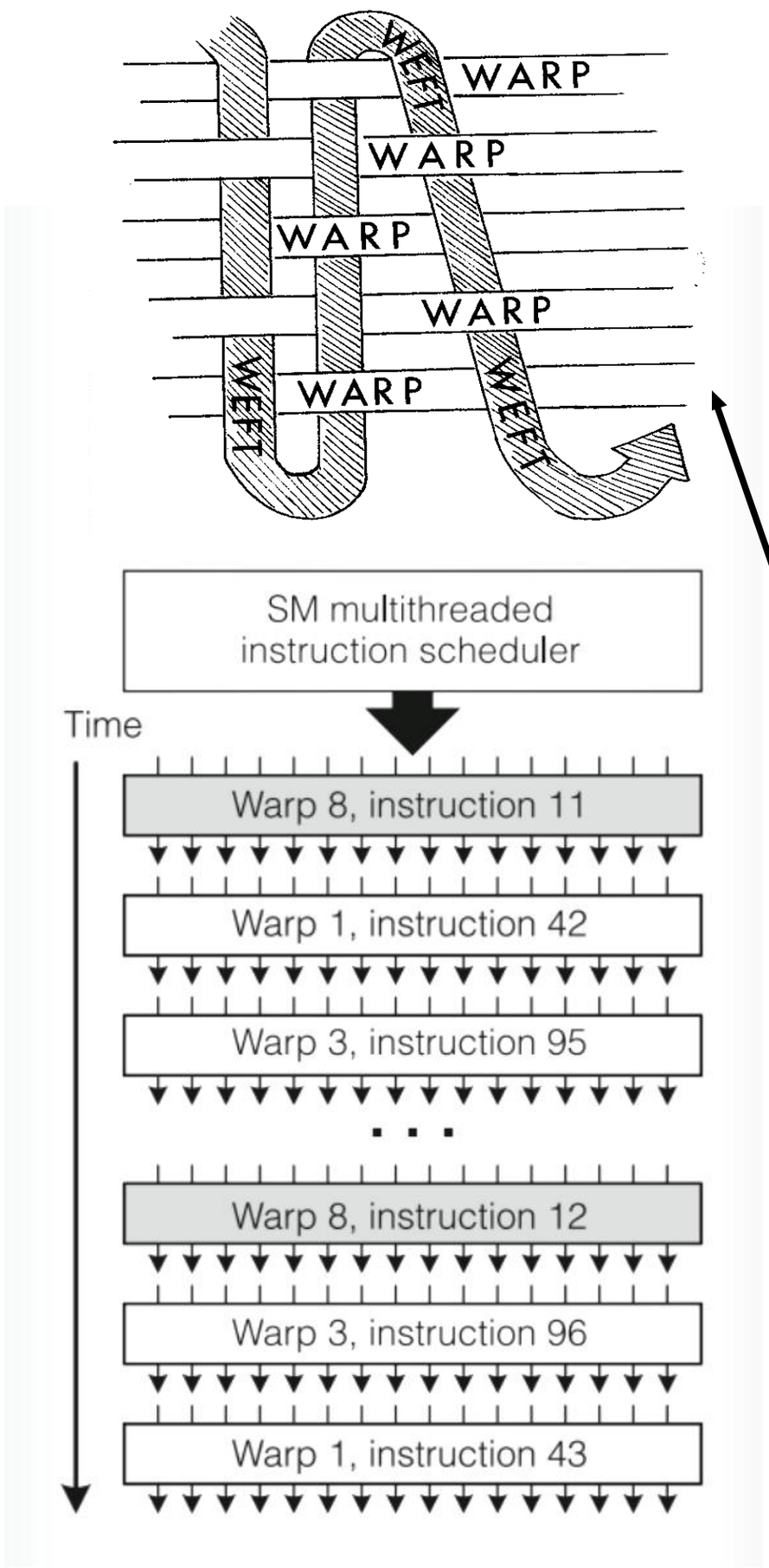
- **The warps are dynamically scheduled by the MT Issue unit**
- **in order to hide memory latency**

Running DAXPY on a GPU



- 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)

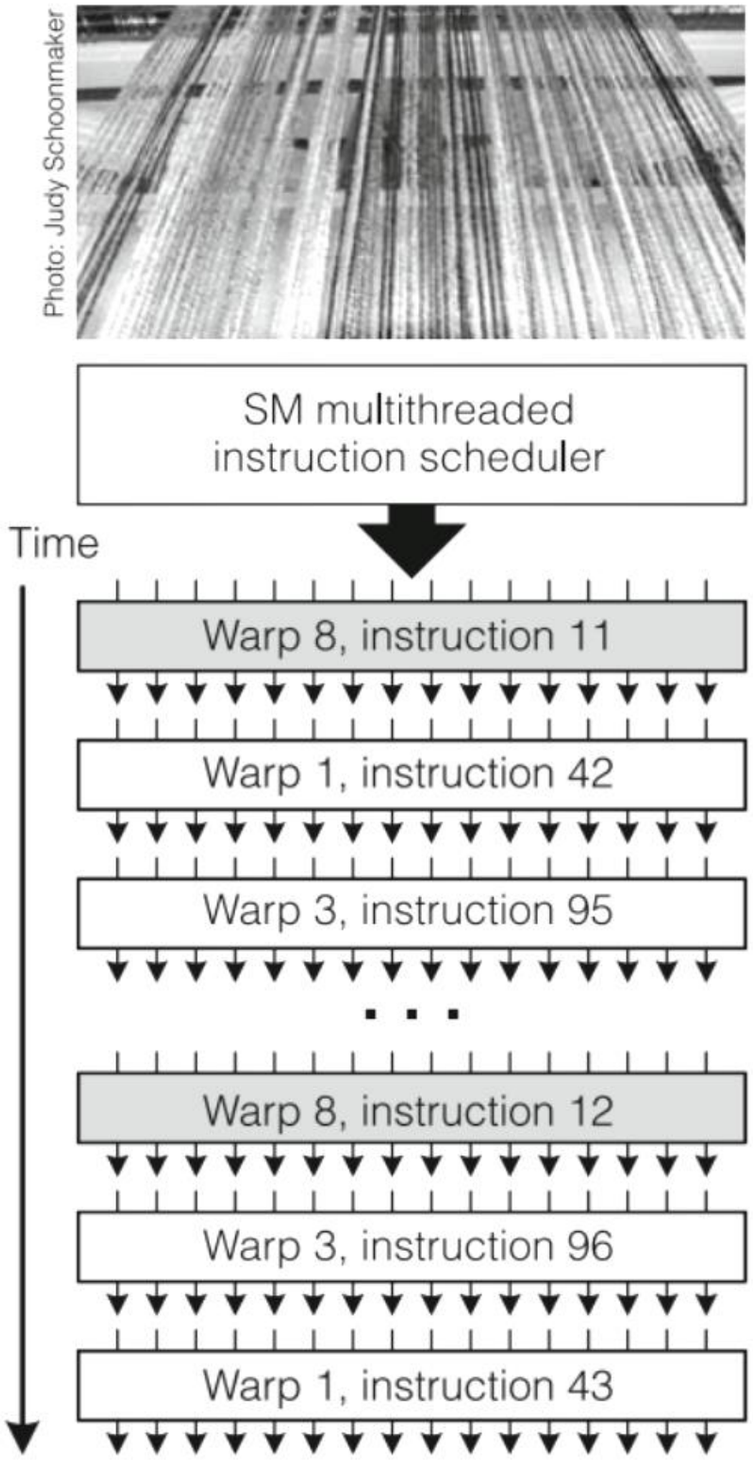
Single-instruction, multiple-thread (SIMT)



- 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 go the same way we get good utilisation (a form of locality – spatial branch locality)**

Predicate bits: enable/disable each lane

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



```
      :
      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
- FGMT: multiple "warps" run on the same core, to hide memory latency

SIMD:

- Each thread may include SIMD vector instructions
- FGMT: 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 (lanes) 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
- We could say that in SIMT, all loads are implemented as SIMD gathers – and they are slow unless coalesced

SIMD:

- Spatial locality = adjacent loop iterations (lanes) access adjacent data
- A normal SIMD vector load accesses adjacent locations
- To access non-adjacent locations you need a "gather" instruction which fetches from a different address per lane
- But performance is often serialised
- Unless the addresses accessed by each lane fall on the same cacheline - when coalescing occurs

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:

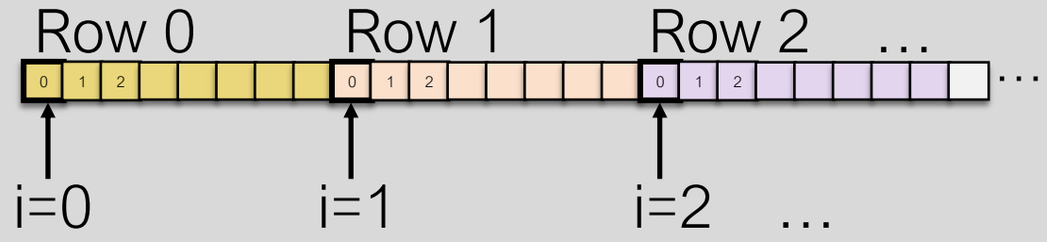
```
void add (float *c, float *a, float *b)
{
  for (int i=0; i <= N; i++) {
    __m128* pa = (__m128*) &a[i][0];
    __m128* pb = (__m128*) &b[i][0];
    __m128* pc = (__m128*) &c[i][0];
    for (int i=0; i <= N/4; i++)
      *pc++ = _mm_add_ps(*pa++,*pb++);
  }
}
```

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

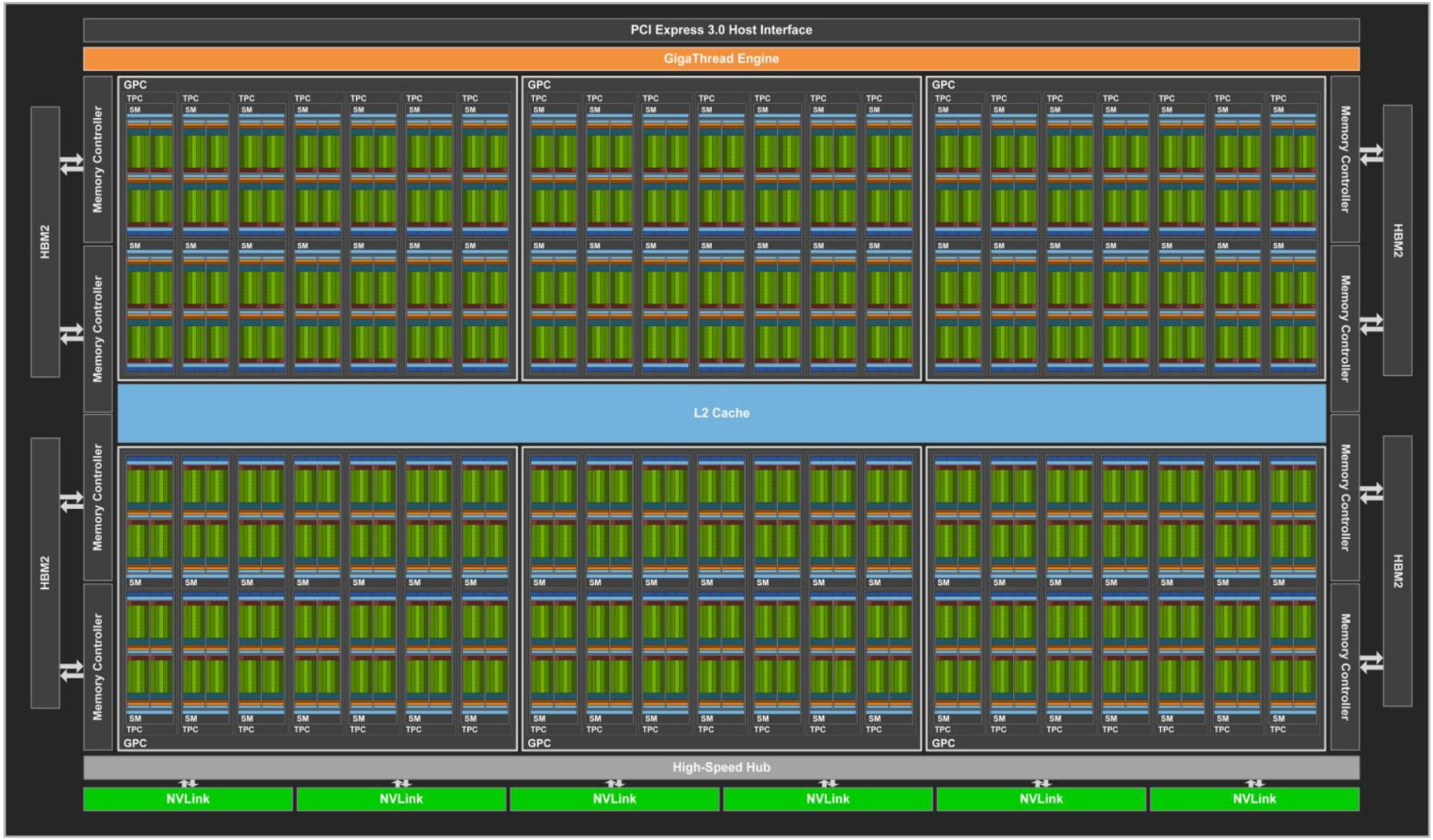
SIMT:

- 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 has 84 SMs

NVI



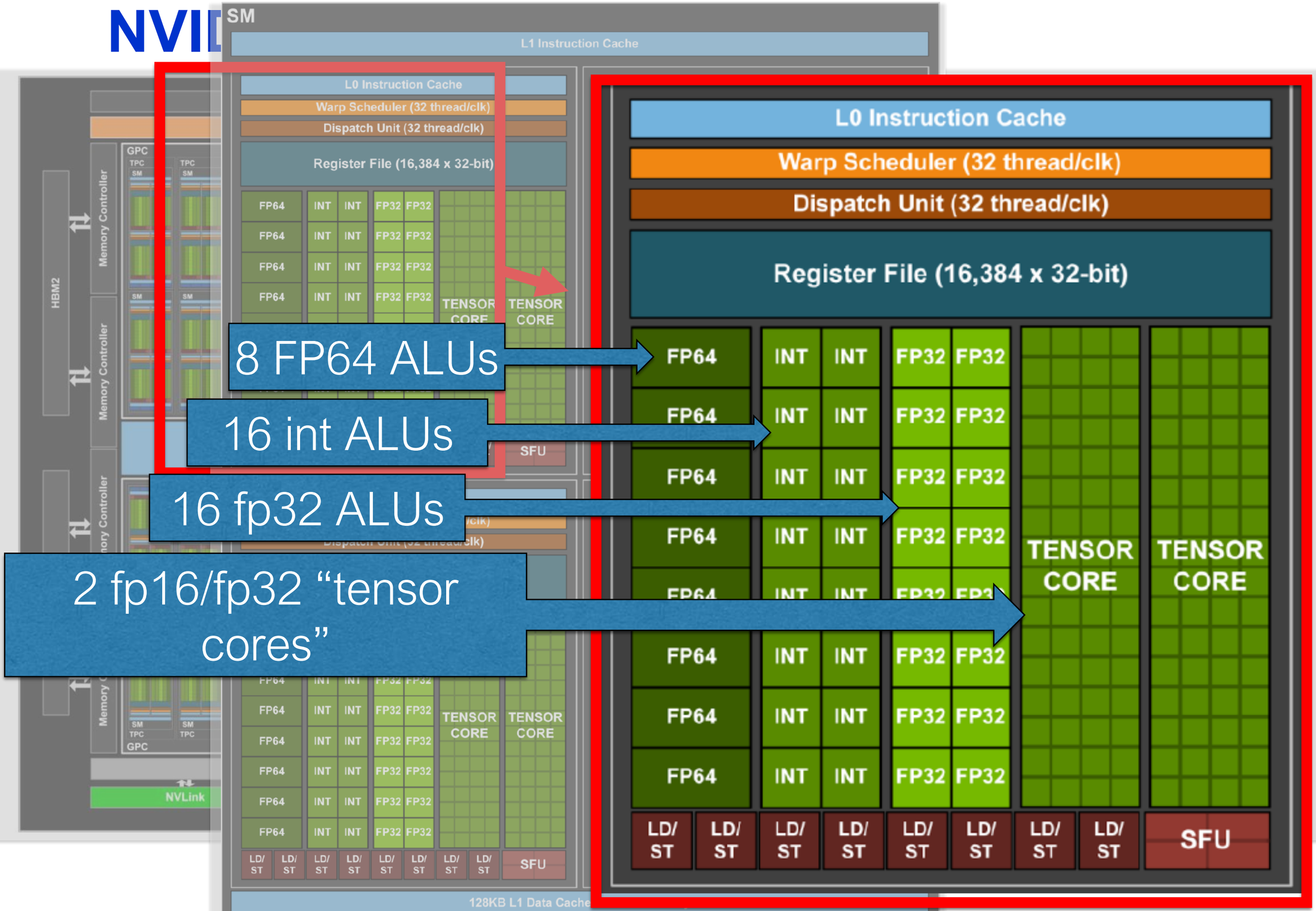
Each SM contains 4 subpartitions

NVI



Each subpartition runs different warps

NVI


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
SFU
LD/ ST
128KB L1 Data Cache
Tex
8 FP64 ALUs
16 int ALUs
16 fp32 ALUs
2 fp16/fp32 “tensor cores”
HBM2
Memory Controller
GPC
TPC
NVLink

NVI

8 FP64 ALUs

16 int ALUs

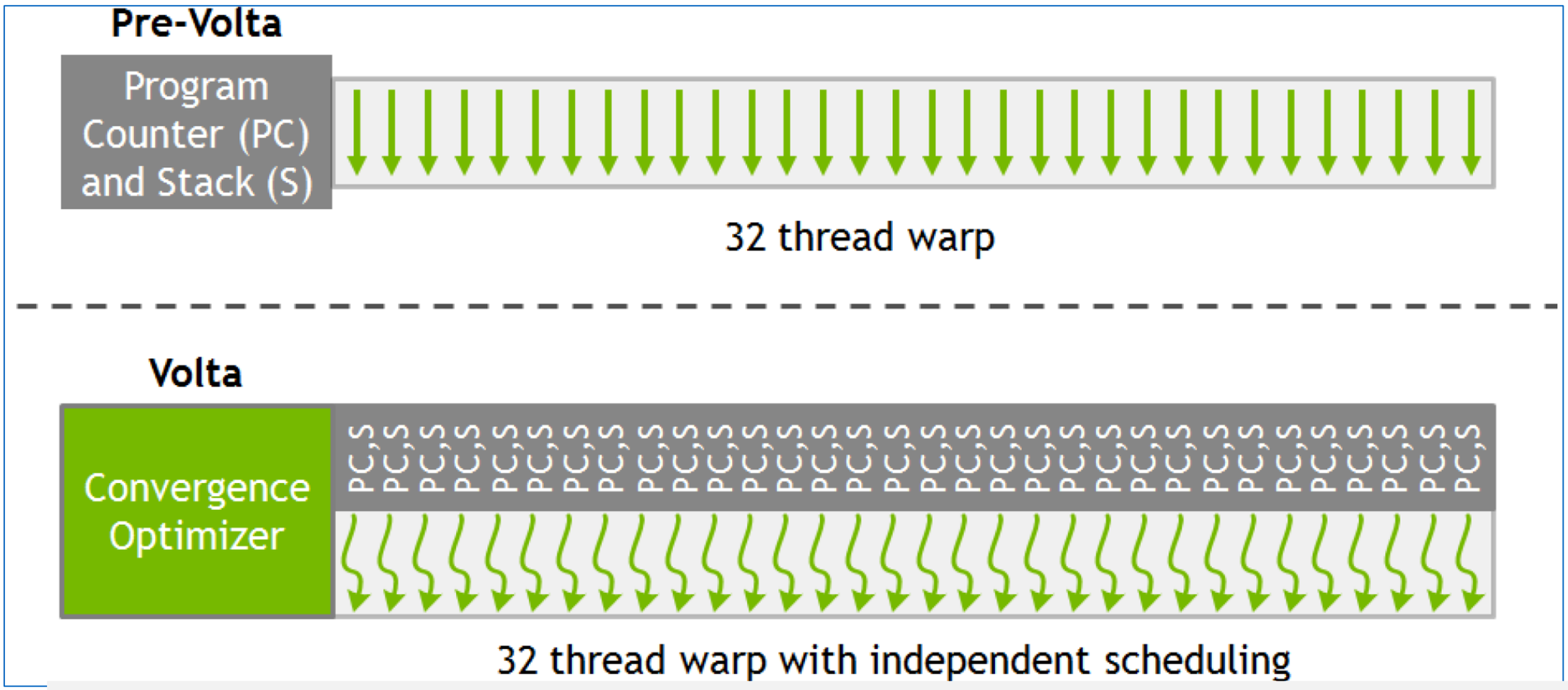
16 fp32 ALUs

2 fp16/fp32 “tensor cores”

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{matrix} \text{FP16 or FP32} \end{matrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} \begin{matrix} \text{FP16} \end{matrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix} \begin{matrix} \text{FP16 or FP32} \end{matrix}$$

Tensor core computes 4x4 matrix-matrix multiply on FP16s with FP32 accumulation

NVIDIA Volta GPU (2017)

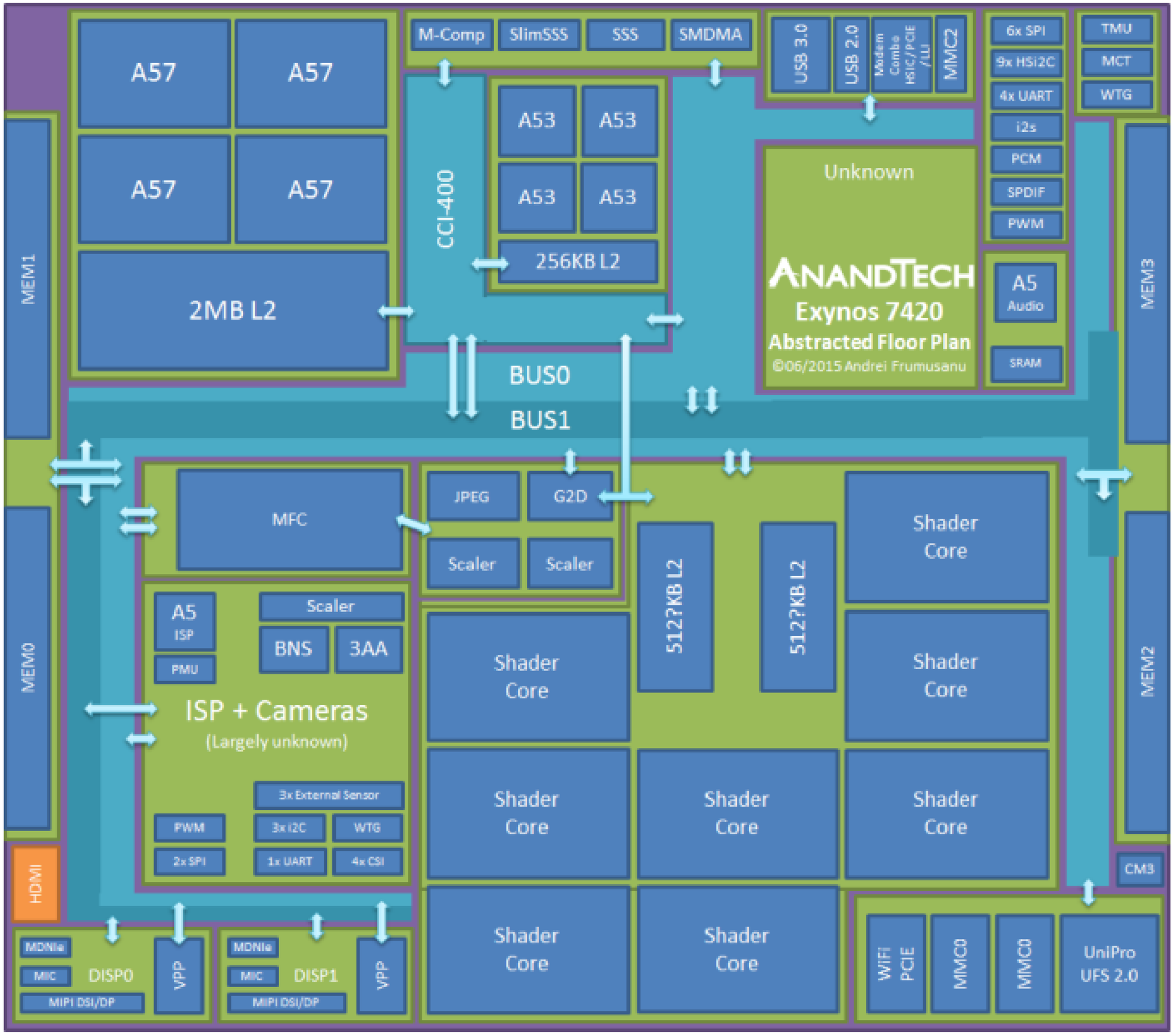


- Volta introduced a “convergence optimizer”
- This enables different threads (lanes) to follow different control-flow paths – eg different branches of if-then-else
- Each CUDA thread has its *own* PC and stack, enabling dynamic scheduling in hardware to heuristically enhance branch convergence

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

spare slides for interest

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

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

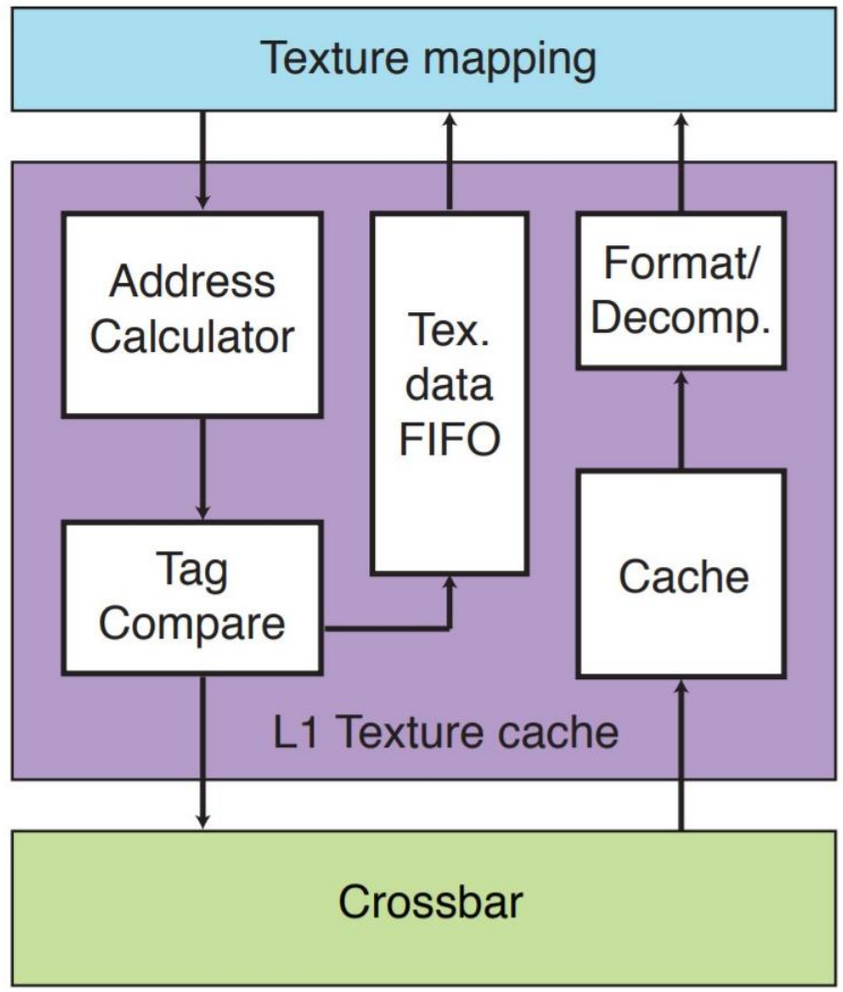


Fig. 5. An overview of a texture cache architecture. The texture mapping unit provides texture coordinates for which a memory address is calculated. The address is sent to the tag compare to determine if the data is in the cache. If the data isn't in the cache, a request is sent via the crossbar to the 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

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?)**

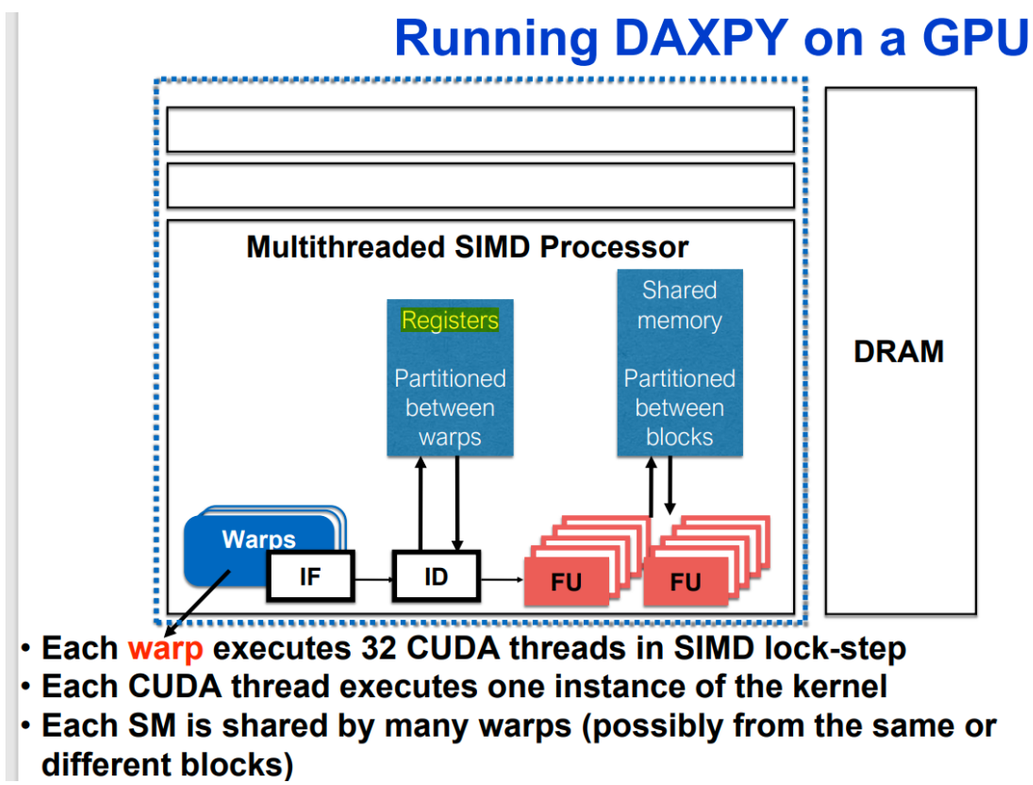
- **Q4: Also, each warp has "32 wide SIMD vector instructions" and GPUs also follow the SIMT model - so we have SIMT as the process describing what's happening with all the warps (ie FGMT threads) and SIMD describing what's happening with every instruction handled by one thread (lane like thing inside 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



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];
}
```

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
}
```

CUDA example: DAXPY

- 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

16

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?

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?

Student question: spatial locality in SIMT

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.

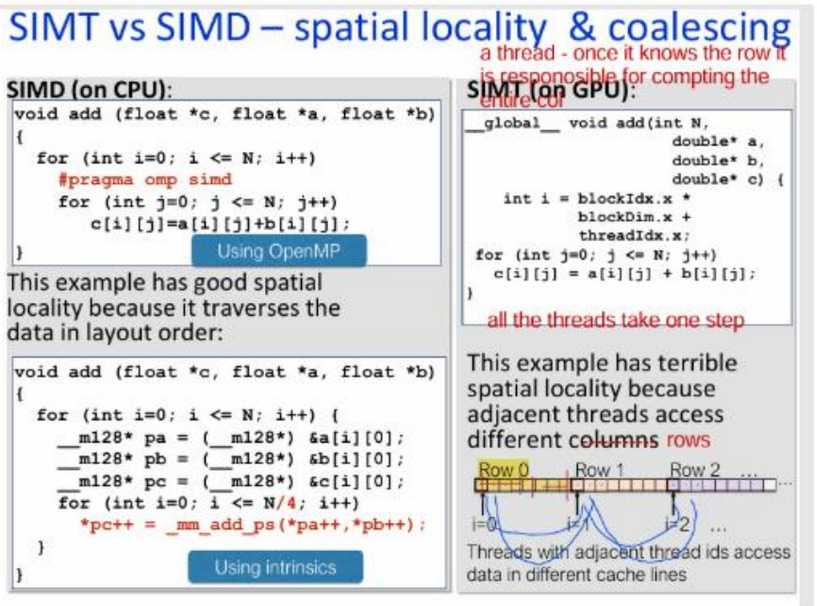
- Yes the loads in this GPU kernel benefit from spatial locality (the CUDA people call it "coalescing").

- 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



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)