Imperial College London

332
Advanced Computer Architecture
Chapter 7

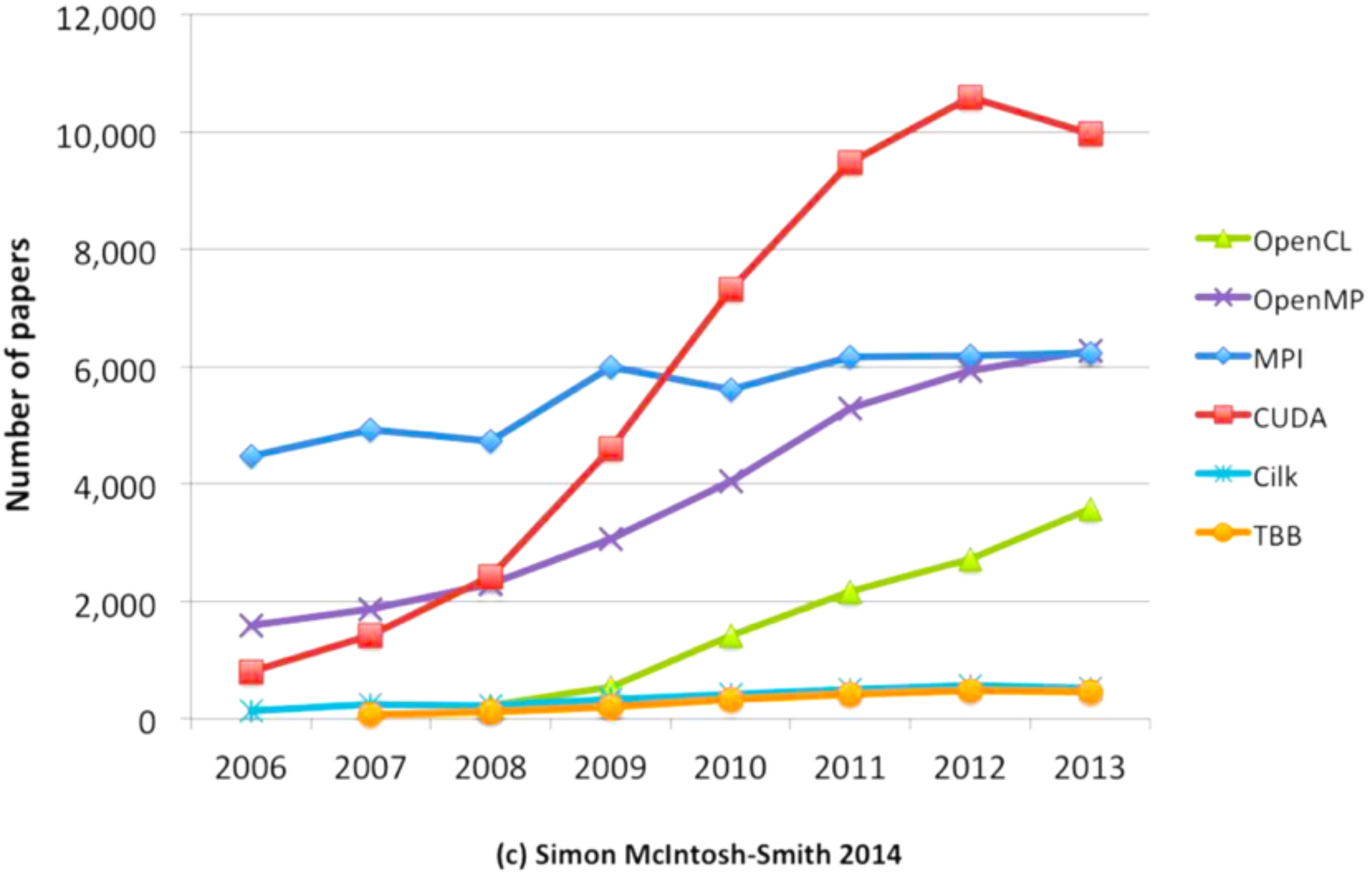
Data-Level Parallelism Architectures and Programs

March 2017
Luigi Nardi

These lecture notes are partly based on:

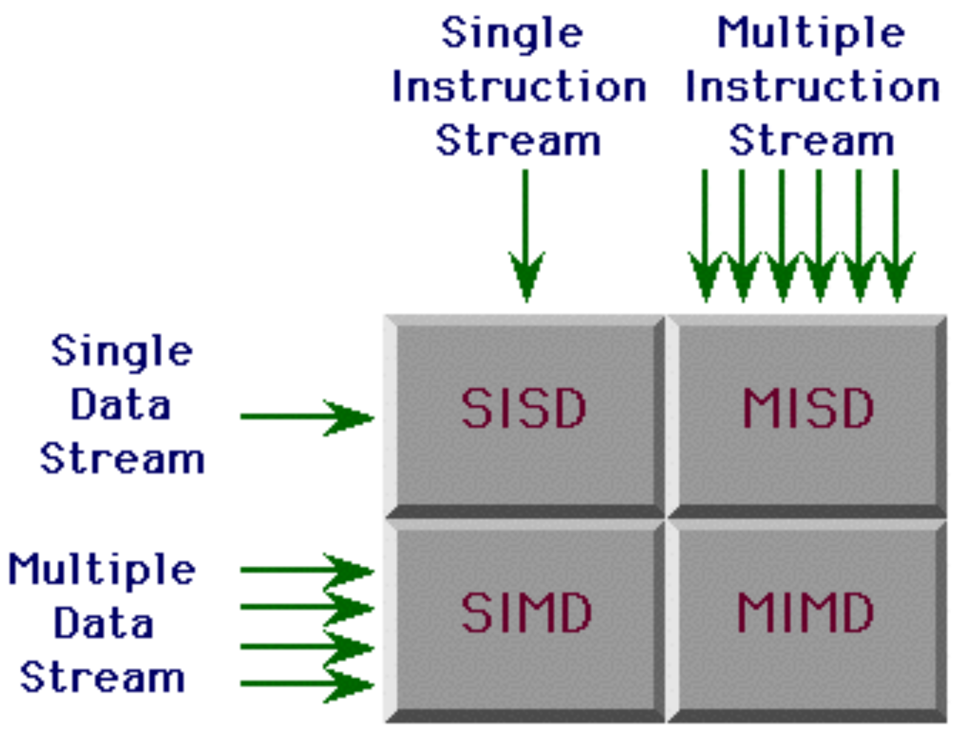
- lecture slides from Paul H. J. Kelly (CO332/2013-2014)
- lecture slides from Fabio Luporini (CO332/2014-2015)
- the course text, Hennessy and Patterson's Computer Architecture (5th ed.)

Papers on parallel programming languages. Data according to Google Scholar (Feb. 2014)



Flynn's Taxonomy (1966): classification of computer architectures

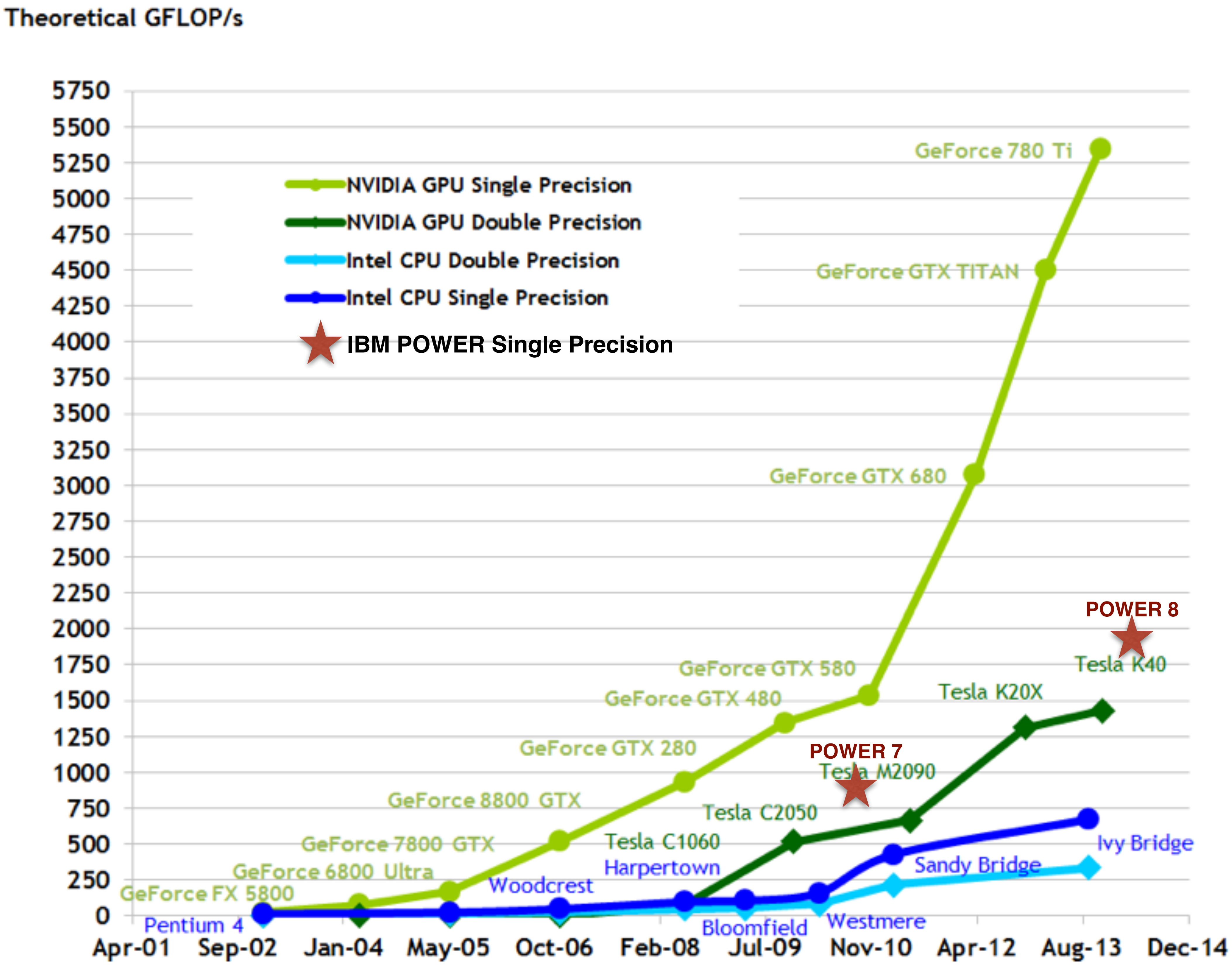
- **SISD**: single-instruction, single-data (single-core CPU)
- **MIMD**: multiple-instruction, multiple-data (multi-core CPU, clusters)
- **SIMD**: single-instruction, multiple-data (data-based parallelism)
- **MISD**: multiple-instruction, single-data (fault-tolerant computers)



Images source: http://www.cems.uwe.ac.uk/teaching/notes/PARALLEL/ARCHITEC

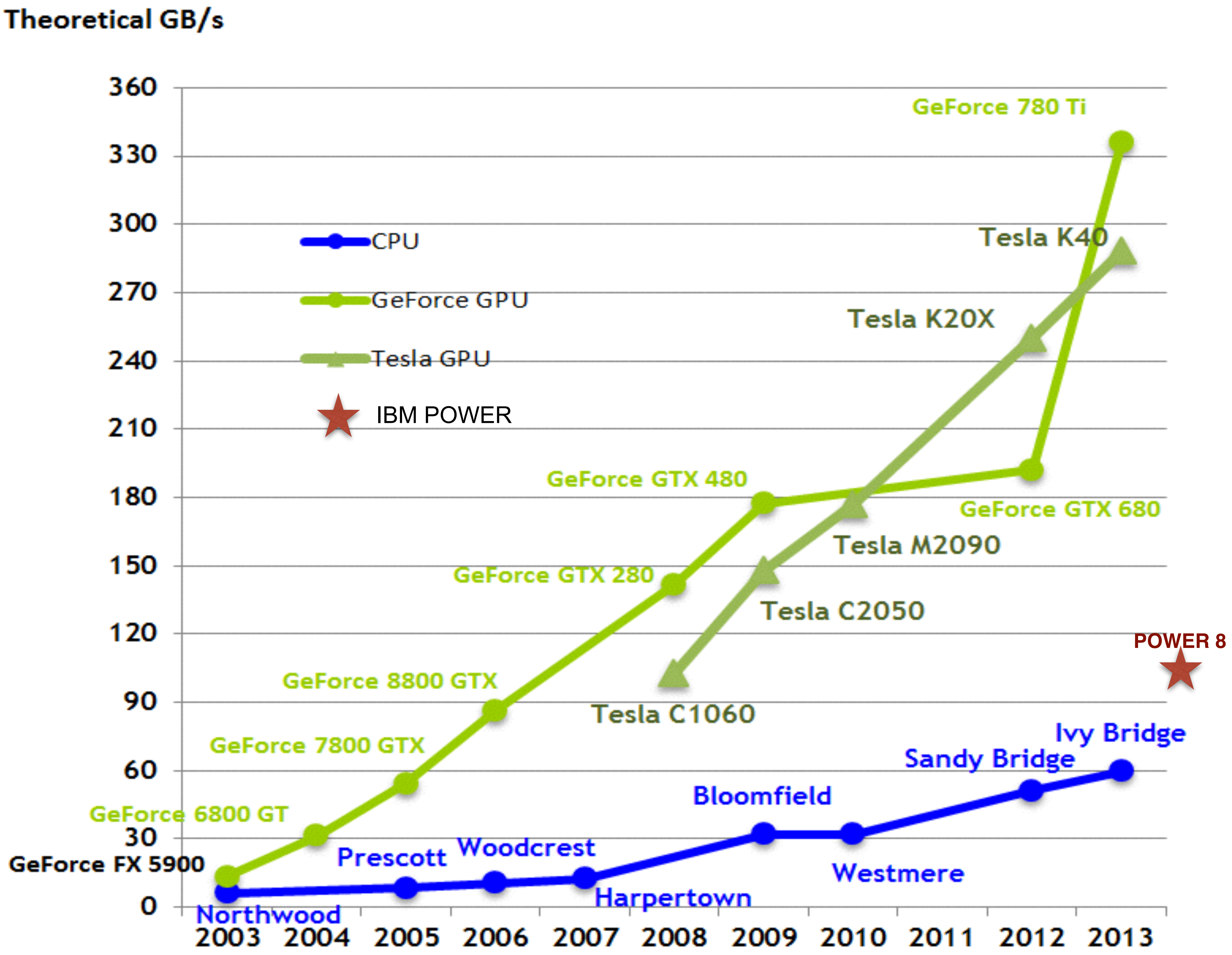
Prof Michael J Flynn (Stanford) was Design Manager on a prototype of the IBM 7090, and later the IBM 360/91 (the first implementation of Tomasulo's Algorithm). He is also Chairman of Maxeler Technologies, who are hiring…

A glimpse of the performance (GFLOPS/s)



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

A glimpse of the performance (GB/s)

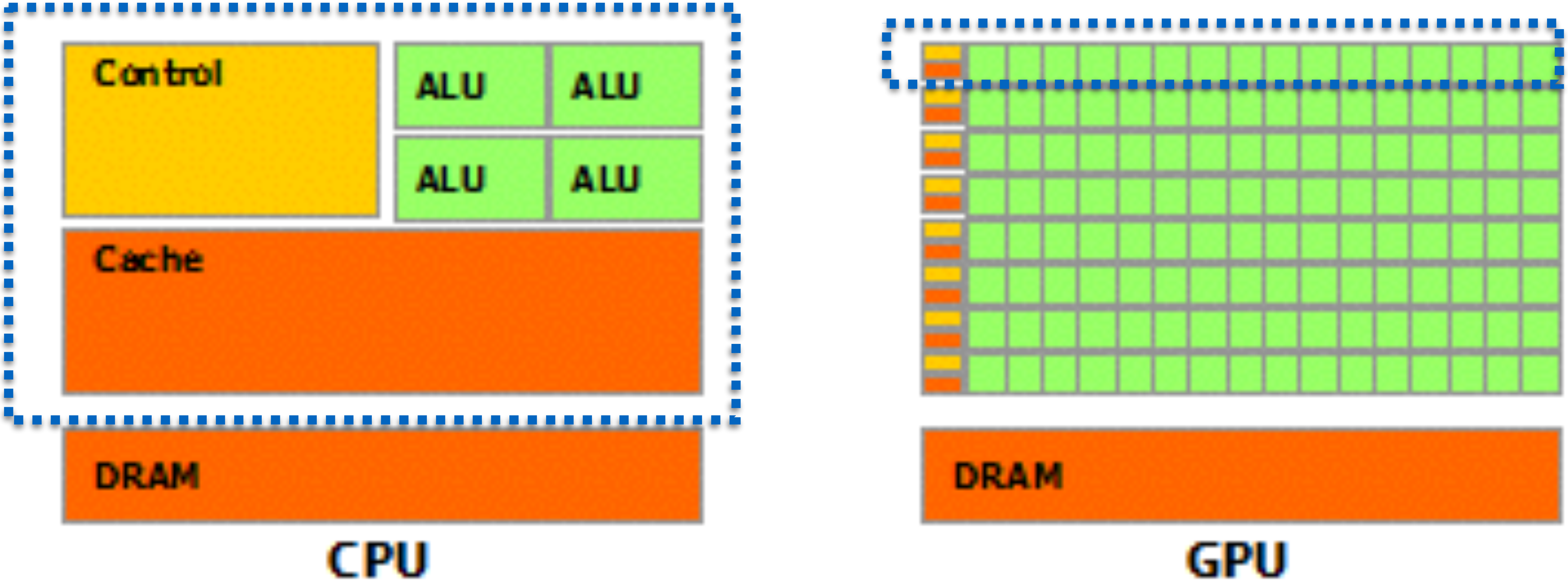


Sources:
http://docs.nvidia.com/cuda/cuda-c-programming-guide/
http://www.anandtech.com/show/10435/assessing-ibms-power8-part-1/7

Graphics Processors (GPUs)

- Much of our attention so far: single-core that runs a single thread faster
- If workload consists of thousands of threads, things look 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 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 (GPUs but also certain CPUs)
- Driven by the gaming market - but with many other applications

A first comparison with CPUs



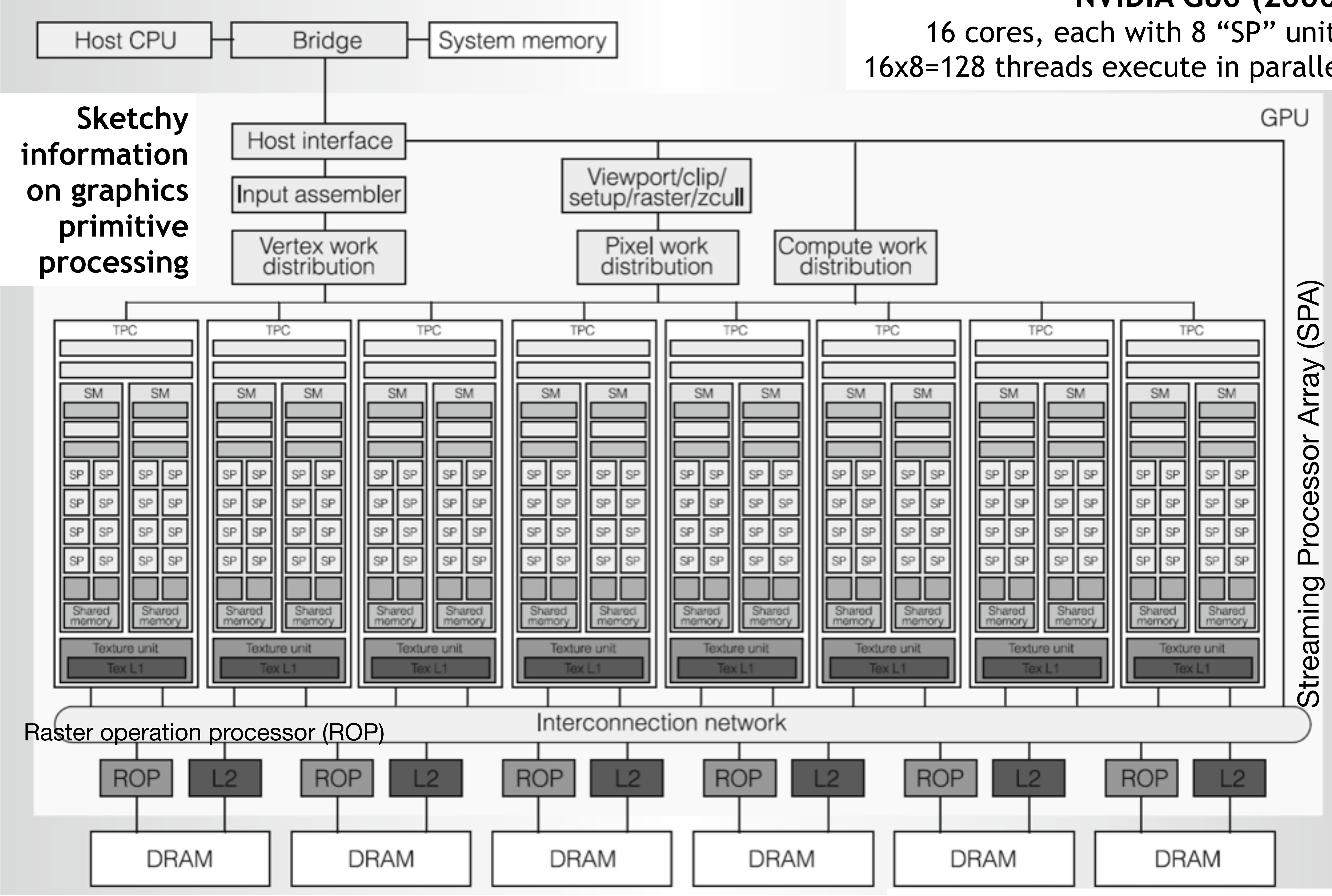
Source: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

- “Simpler” cores
- Many functional units (FUs) (implementing the SIMD model)
- No (or limited) caching; 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

Sketchy information on graphics primitive processing



Streaming Processor Array (SPA)

Raster operation processor (ROP)

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

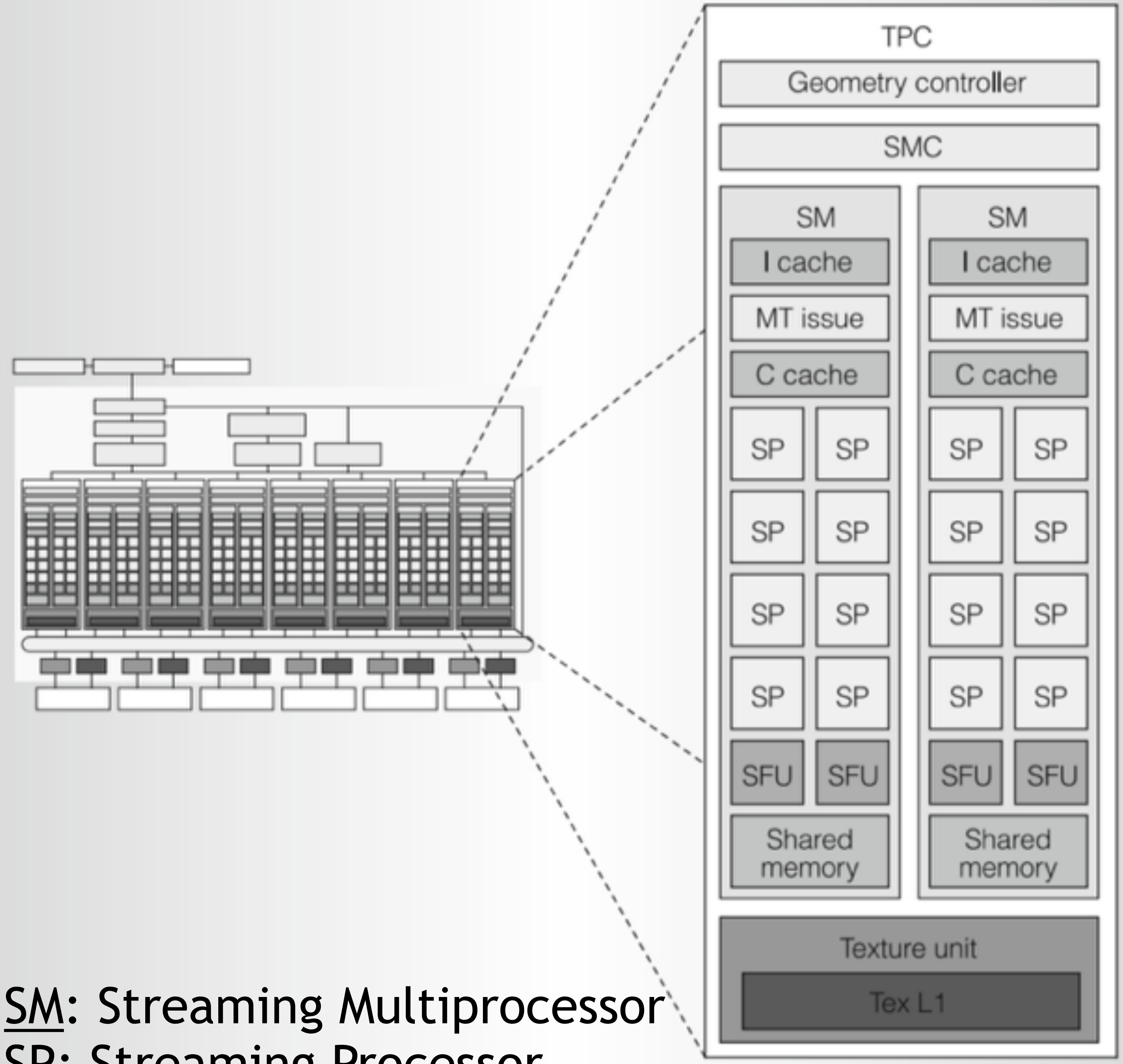
ROP performs colour and depth frame buffer operations directly on memory

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)

NVIDIA G80 (2006)

16 cores, each with 8 SP units



- SM: Streaming Multiprocessor
- SP: Streaming Processor

- Geometry controller: directs all primitive and vertex attribute and topology flow in the TPC
- SMC: Streaming Multiprocessor controller
- I cache: instruction cache
- MT issue: multithreaded instruction fetch and issue unit
- C cache: constant read-only cache
- SFU: Special-Function Unit, compute transcendental functions (sin, log, 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

- 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
- 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 diagram)
- Tesla is also the NVIDIA brand codename for server GPUs:
  - NVIDIA micro-architectures: Tesla, Fermi, Kepler, Maxwell, Pascal, Volta
  - NVIDIA brands: Tegra, Quadro, GeForce, Tesla

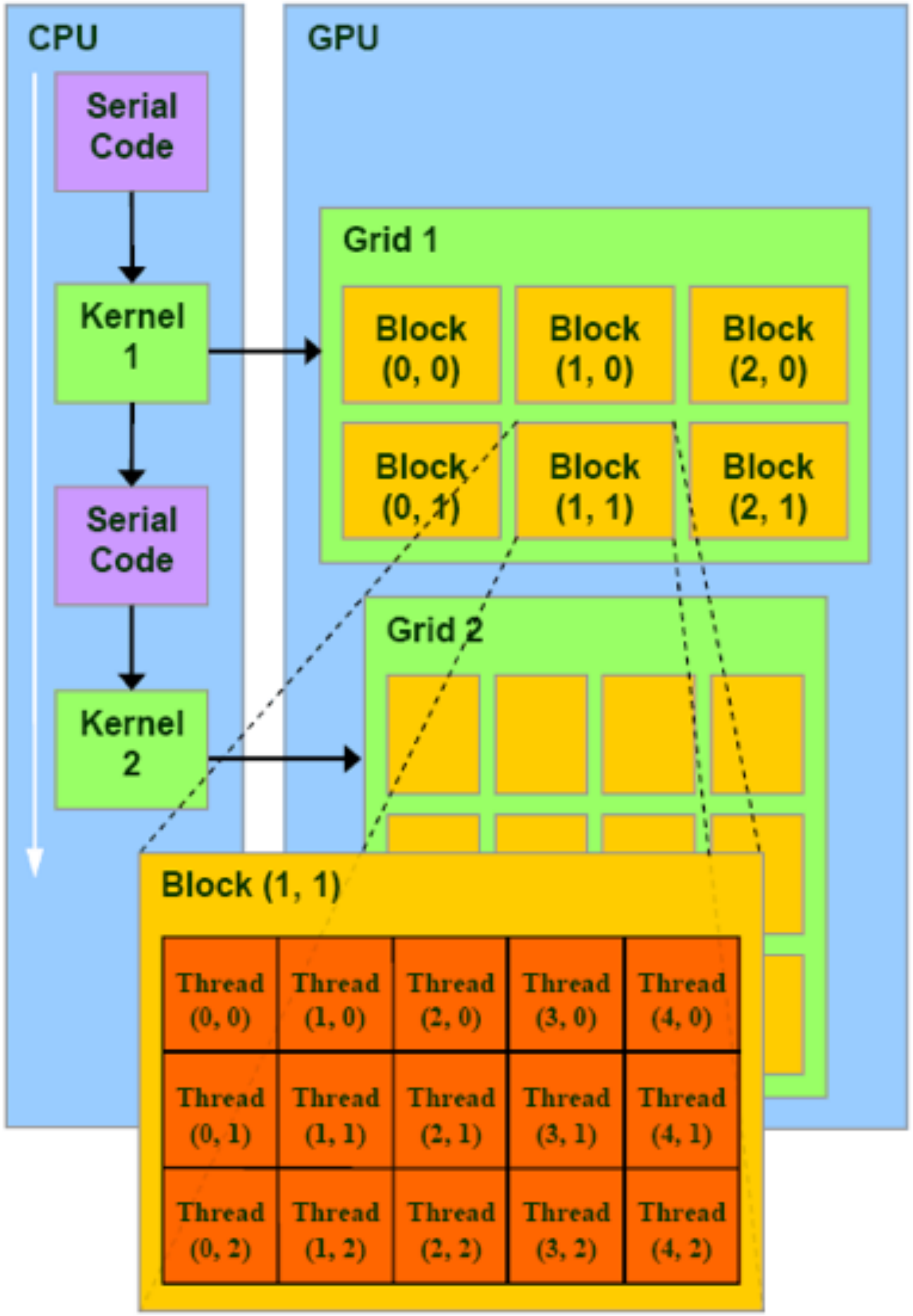
CUDA Execution Model

- 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, 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

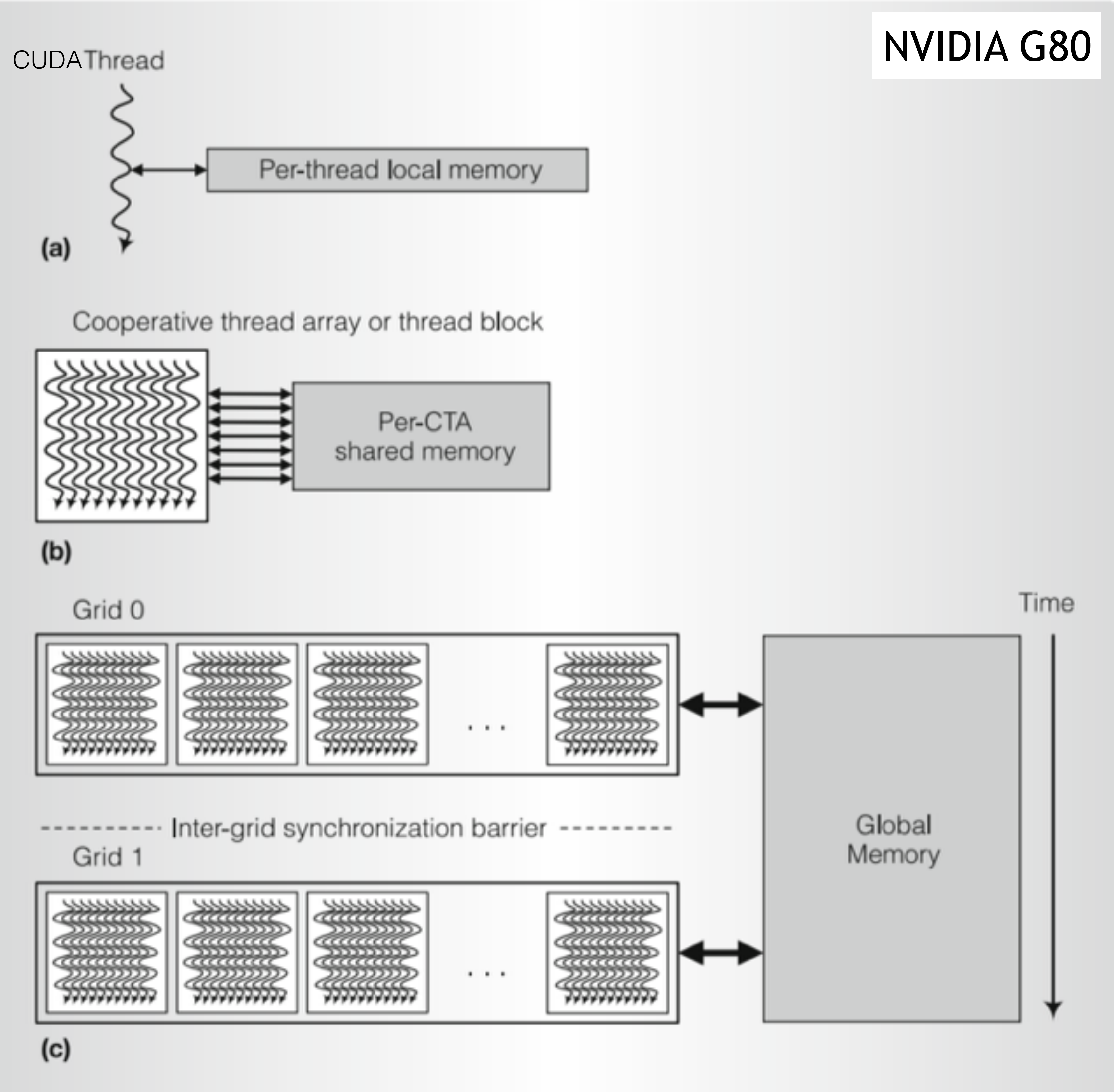
Blocks are allocated (dynamically) to SMs. Threads within a block run on the same SM

Blocks in a grid can't interact with each other



Source: CUDA programming guide

Nested granularity levels



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

Cooperative Thread Array (CTA) = thread block

Different levels have corresponding memory-sharing levels:

- (a) thread
- (b) thread block
- (c) grid

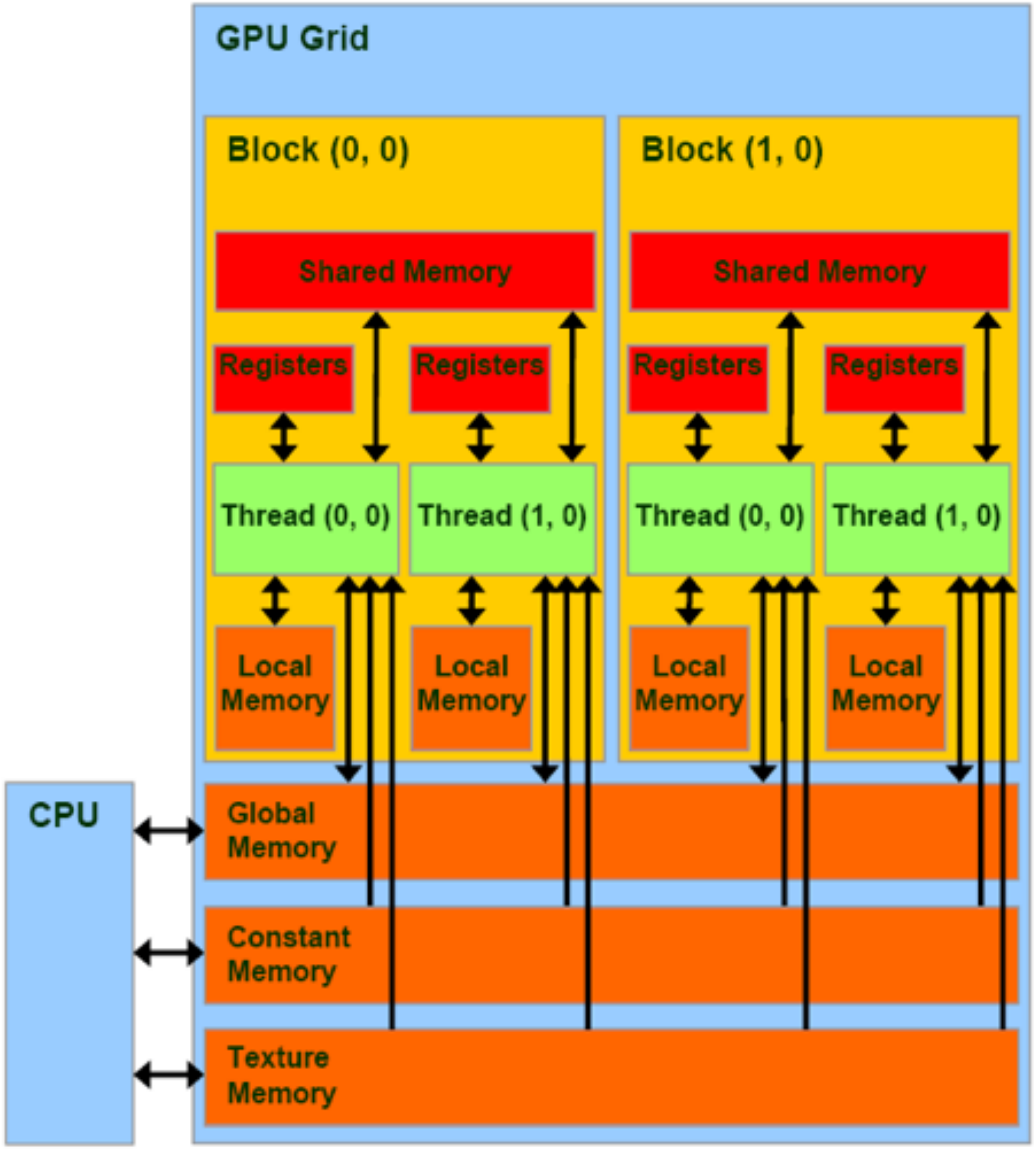
Note:

CUDA thread is just a vertical cut of a thread of SIMD instructions, corresponding to one element executed by on SIMD lane.

CUDA threads are very different from POSIX threads; you can't make arbitrary system calls from a CUDA thread

CUDA Memory Model

- 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 but in the GPU card)
- Constant memory (small, read-only)
- Texture memory (read-only; cached, stored in Global memory)
- Memory instructions load-global, store-global, load-shared, store-shared, load-local, and store-local



Source: CUDA programming guide

- 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

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];
 }
```

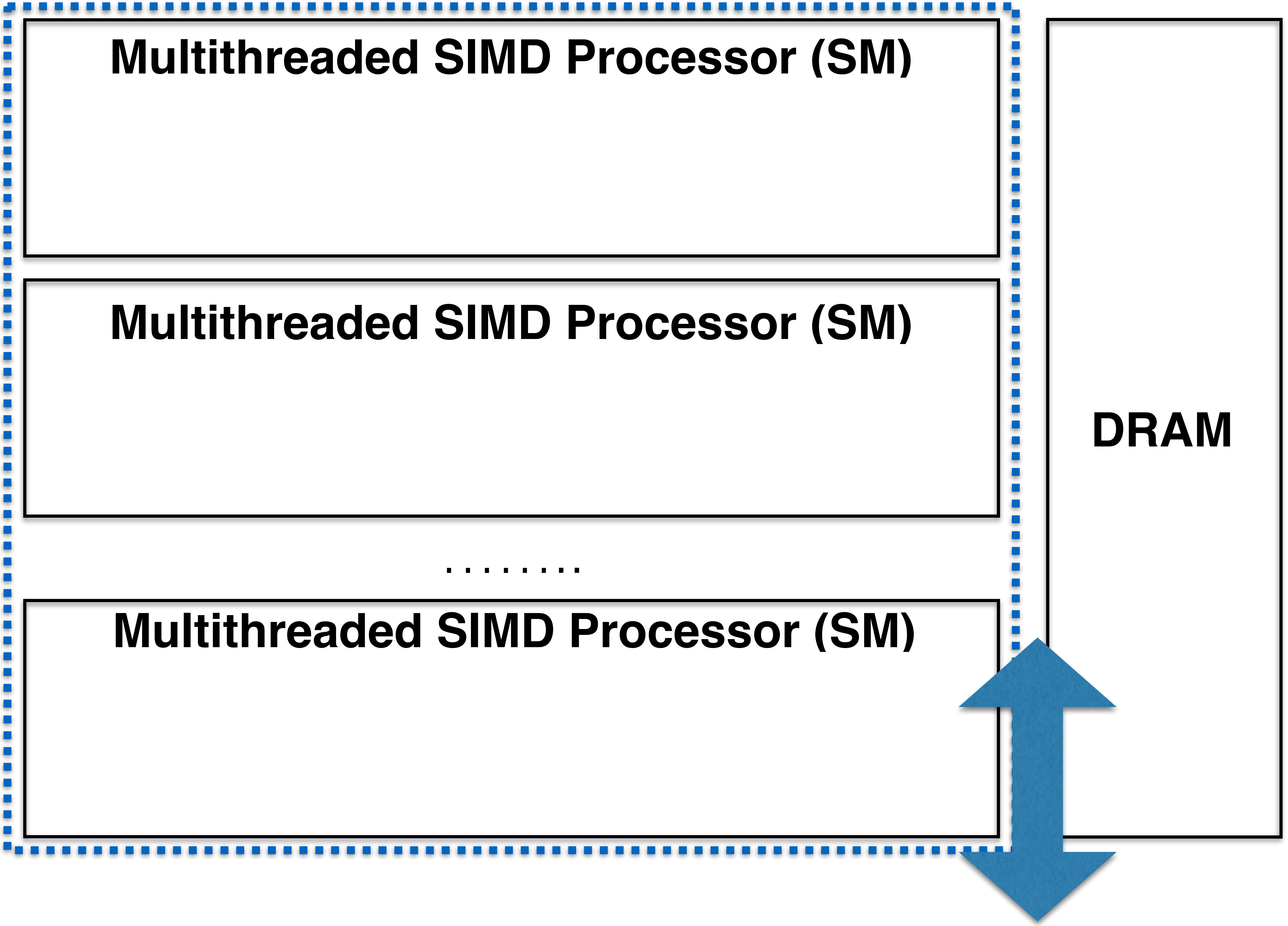
```
// Invoke DAXPY
daxpy(n, 2.0, x, y);
// DAXPY in C
void daxpy(int n,
           double a,
           double* x,
            double* y) {
    for(int i=0; i < n; ++i)
       y[i] = a*x[i] + y[i];
}
```

fully parallel loop

```
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 (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**

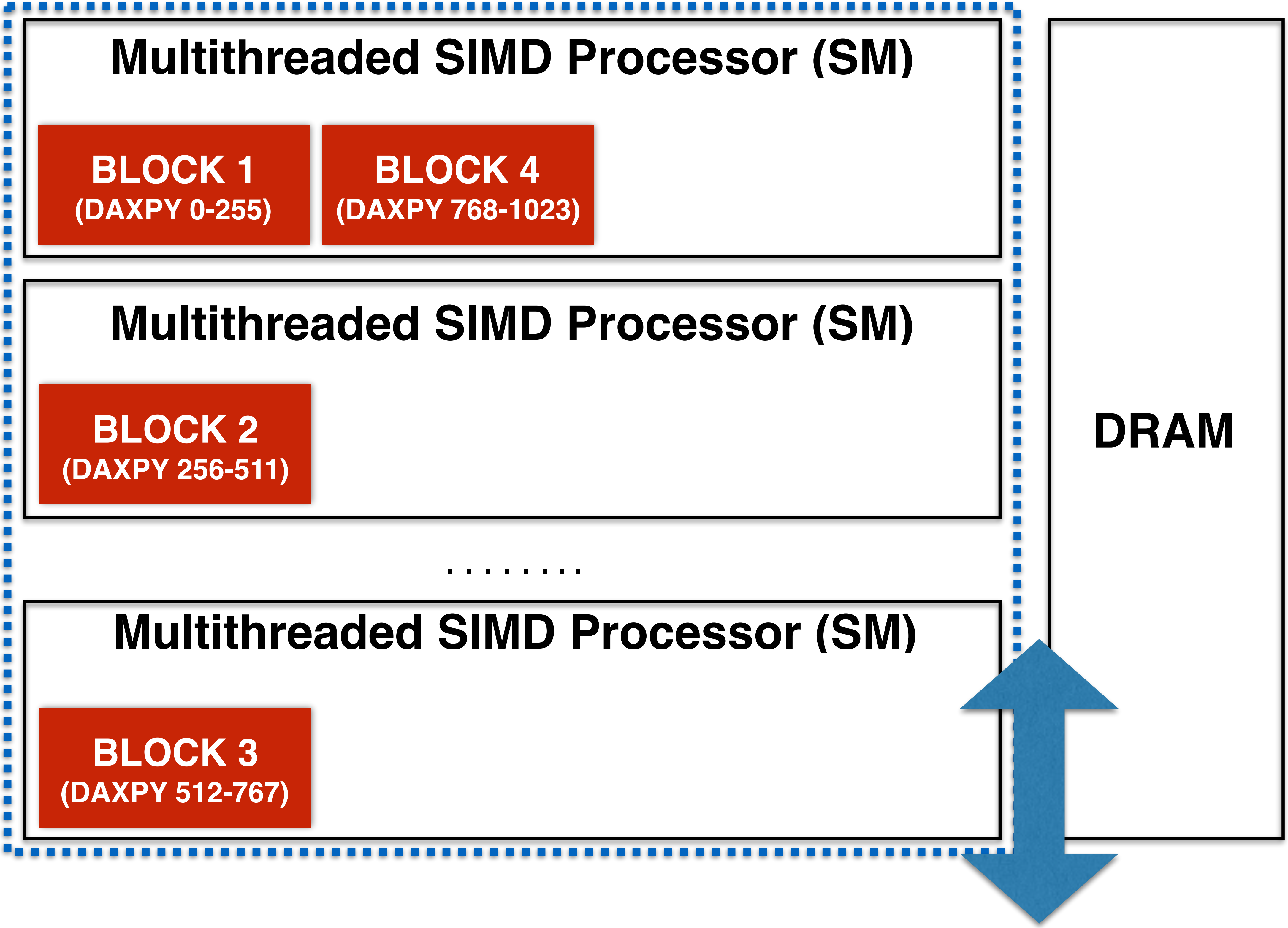
Running DAXPY (N=1024) on a GPU


Multithreaded SIMD Processor (SM)
Multithreaded SIMD Processor (SM)
……..
Multithreaded SIMD Processor (SM)
DRAM

Note that: SIMD + MIMD

Host (via I/O bus, DMA)

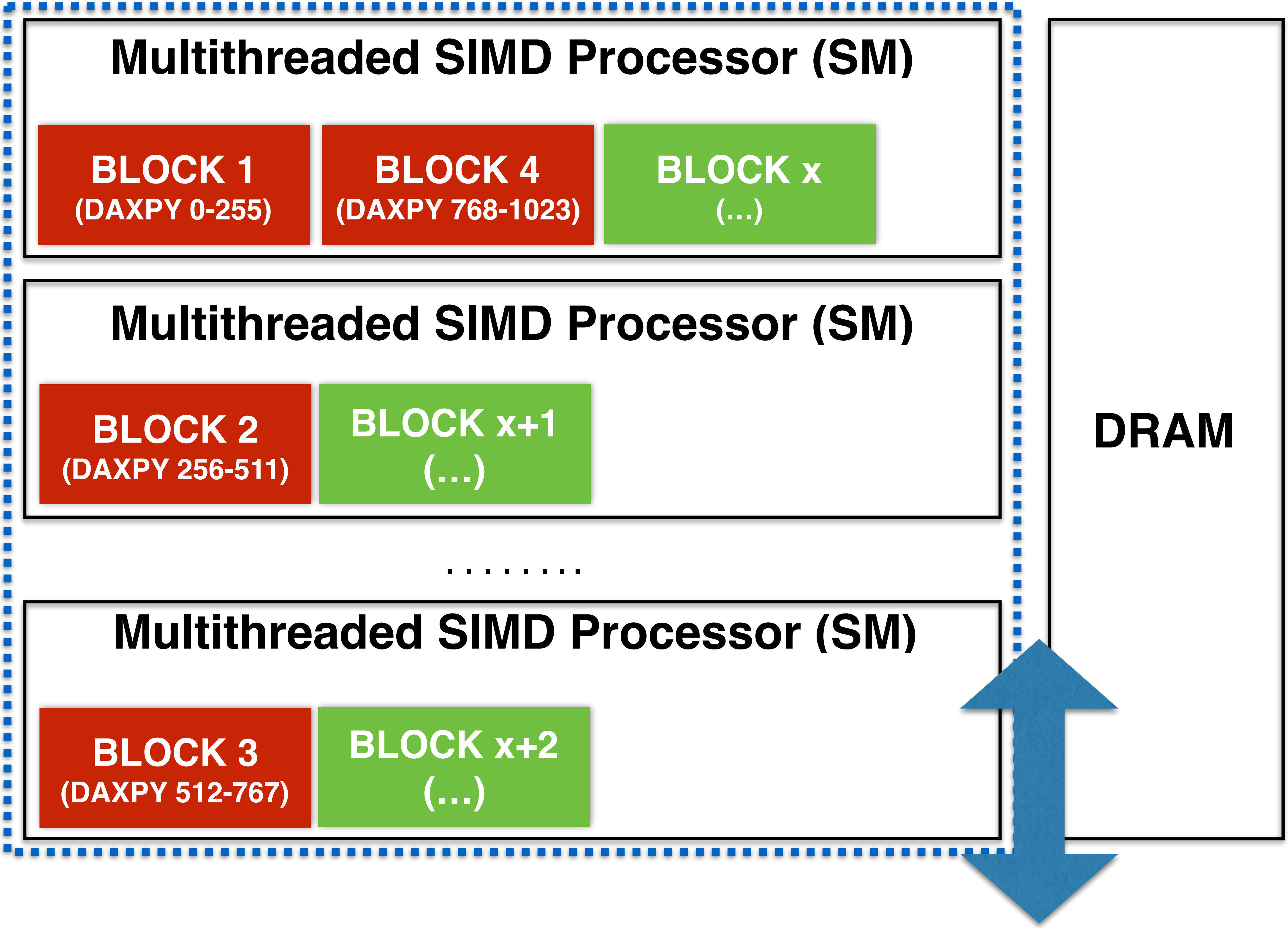
Running DAXPY (N=1024) on a GPU



Host (via I/O bus, DMA)

Note that: SIMD + MIMD

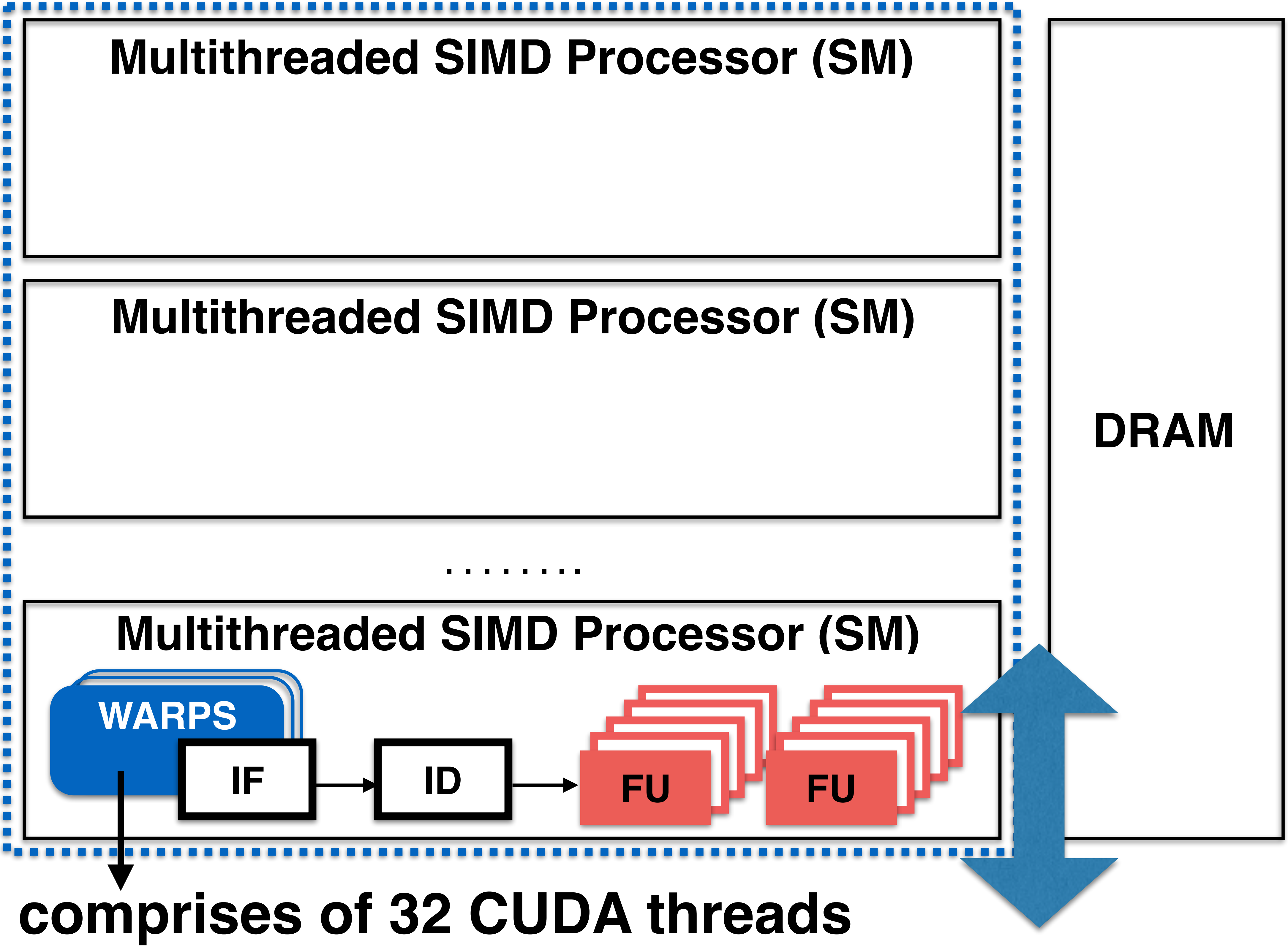
Running DAXPY (N=1024) on a GPU



Note that: SIMD + MIMD

Host (via I/O bus, DMA)

Running DAXPY on a GPU


Multithreaded SIMD Processor (SM)
Multithreaded SIMD Processor (SM)
....
Multithreaded SIMD Processor (SM)
WARPS
IF
ID
FU
FU
DRAM

A warp comprises of 32 CUDA threads

Host (via I/O bus, DMA)

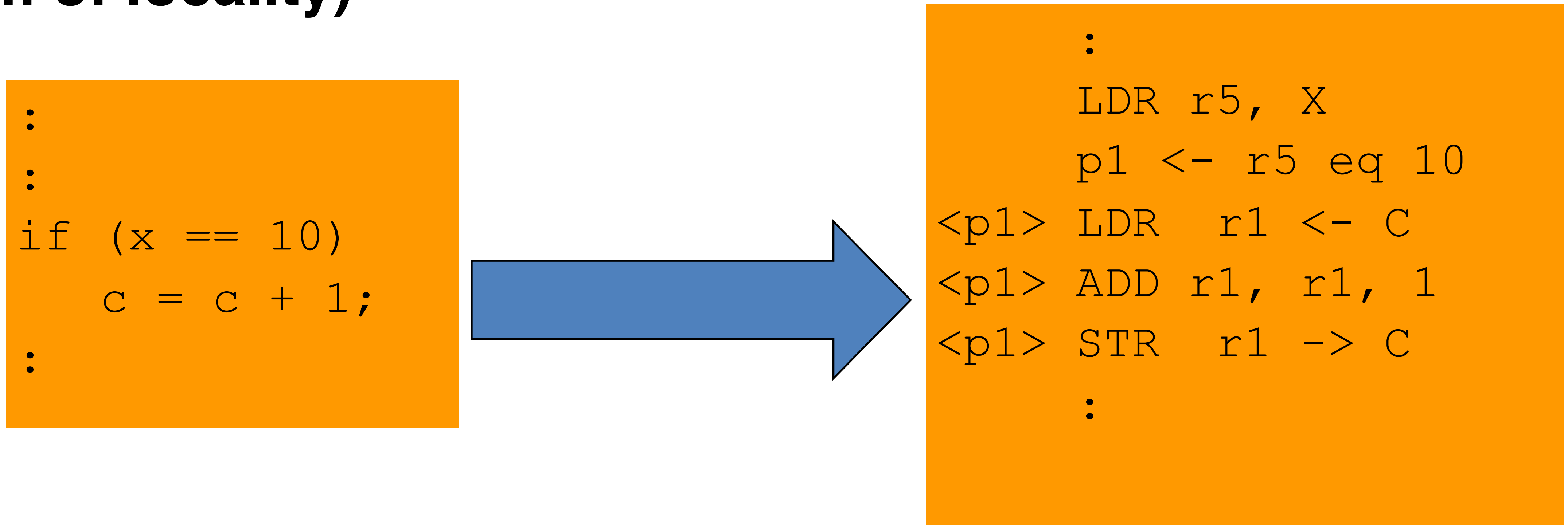
Mapping from CUDA to TESLA

- Array of streaming multiprocessors (SMs)
  - (we might call them “cores”, when comparing to conventional multi-core; 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, hyper-threading)
- A warp is like a traditional thread (32 CUDA threads executed as 32 SIMD operations)

- **Corollary: enough warps are needed to avoid stalls (i.e., enough threads per block). Also called GPU occupancy in CUDA**
- **But: high occupancy is not always a good solution to achieve good performance, i.e., memory bound applications may need a less busy bus to perform well. Reduce the number of in-flight loads/stores by reducing the number of blocks on a SM and improve cache trashing. How?**
  - **If you are a ninja: use dynamic shared memory to reduce occupancy**
  - **Or increase the number of registers in your kernel**

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
- This model of execution is called lockstep instructions are serialised on branch divergence
- Control-flow coherence: every thread goes the same way (a form of locality)

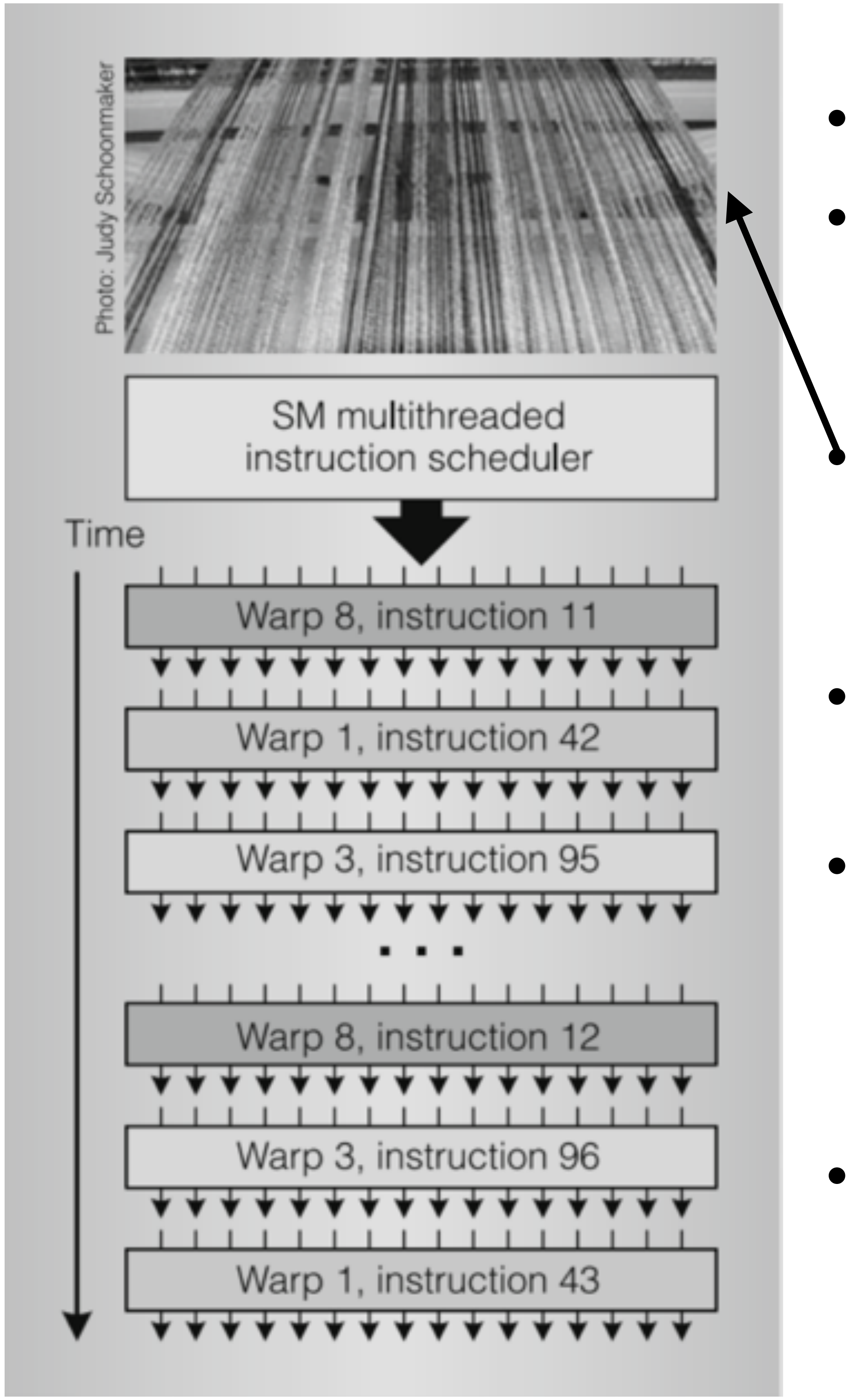


Predicate bits: enable/disable each lane

```
:
:
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
      :
```

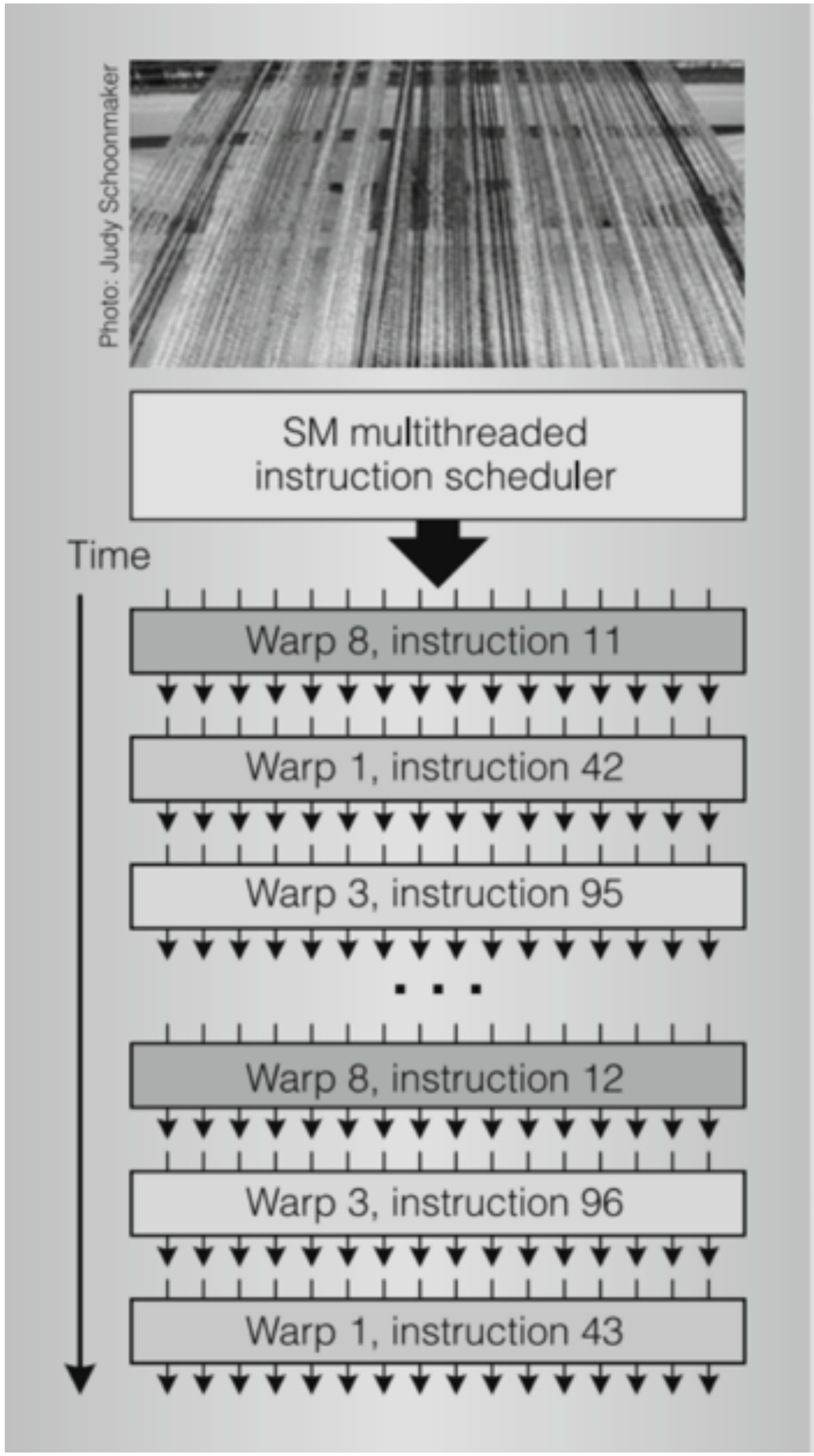
Single-instruction, multiple-thread (SIMT)



- 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: refers to the group of threads being woven together into fabric
- Each SM manages a pool of 24 warps, 24 ways SMT
- 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)

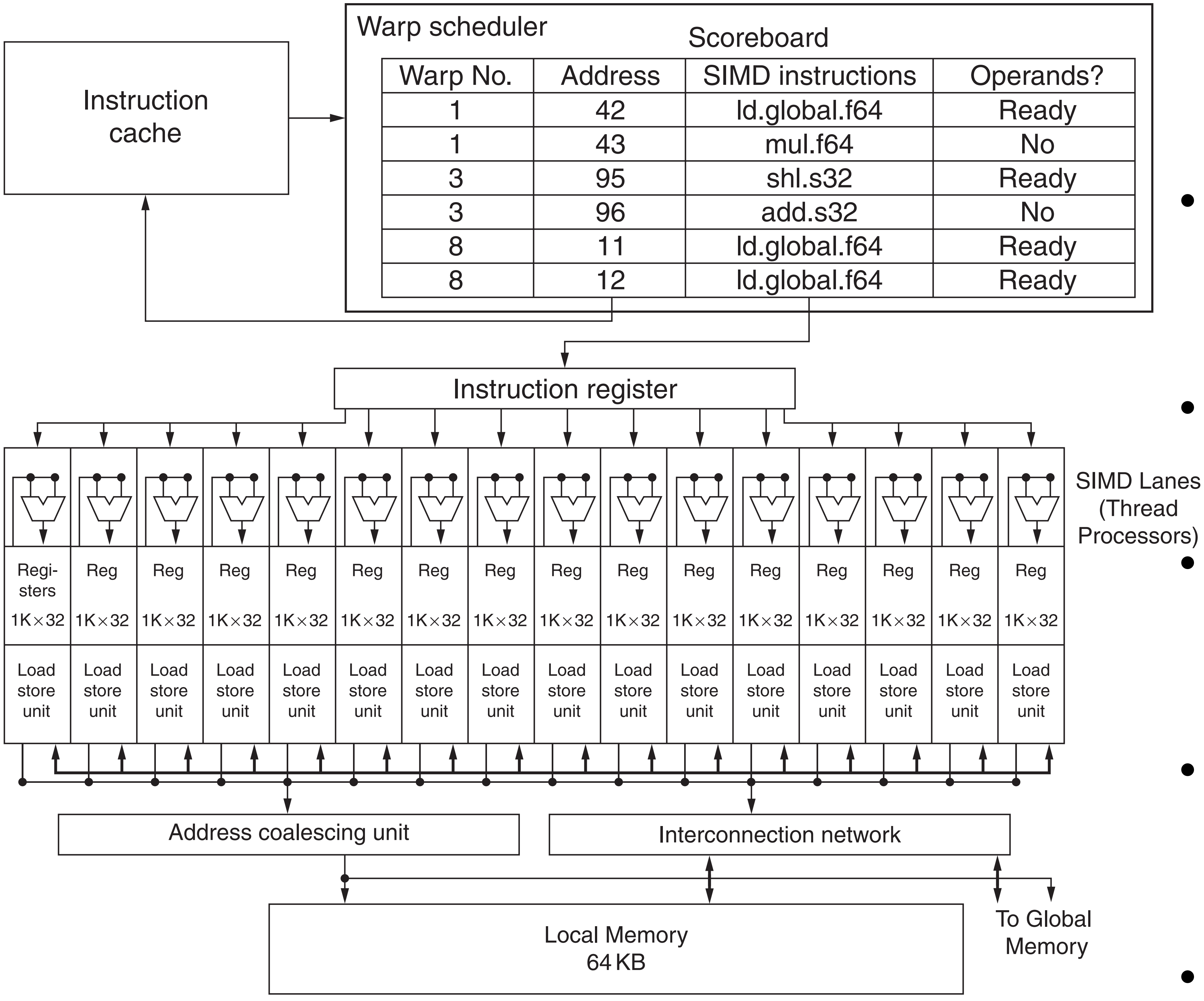
More on SIMT



- 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)

GPU SM or multithreaded SIMD processor



Warp No.	Address	SIMD instructions	Operands?
1	42	ld.global.f64	Ready
1	43	mul.f64	No
3	95	shl.s32	Ready
3	96	add.s32	No
8	11	ld.global.f64	Ready
8	12	ld.global.f64	Ready

Figure 4.14 Hennessy and Patterson's Computer Architecture (5th ed.)

Out-of-order execution

- Many parallel functional units instead of a few deeply pipelined
- Thread block scheduler assigns a thread block to the SM
- Scoreboard tells which warp (or thread of SIMD instruction) is ready to run
- GPU has two levels of hardware schedulers:
  - Threads blocks
  - Warps
- Number of SIMD lanes varies across generations

SIMT vs SIMD – GPUs without the hype

- GPUs combine many architectural techniques:
  - Multi-core
  - Simultaneous multithreading (SMT)
  - Vector instructions
  - Predication
  - OoO execution

- 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 Processor**

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

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

SIMD:

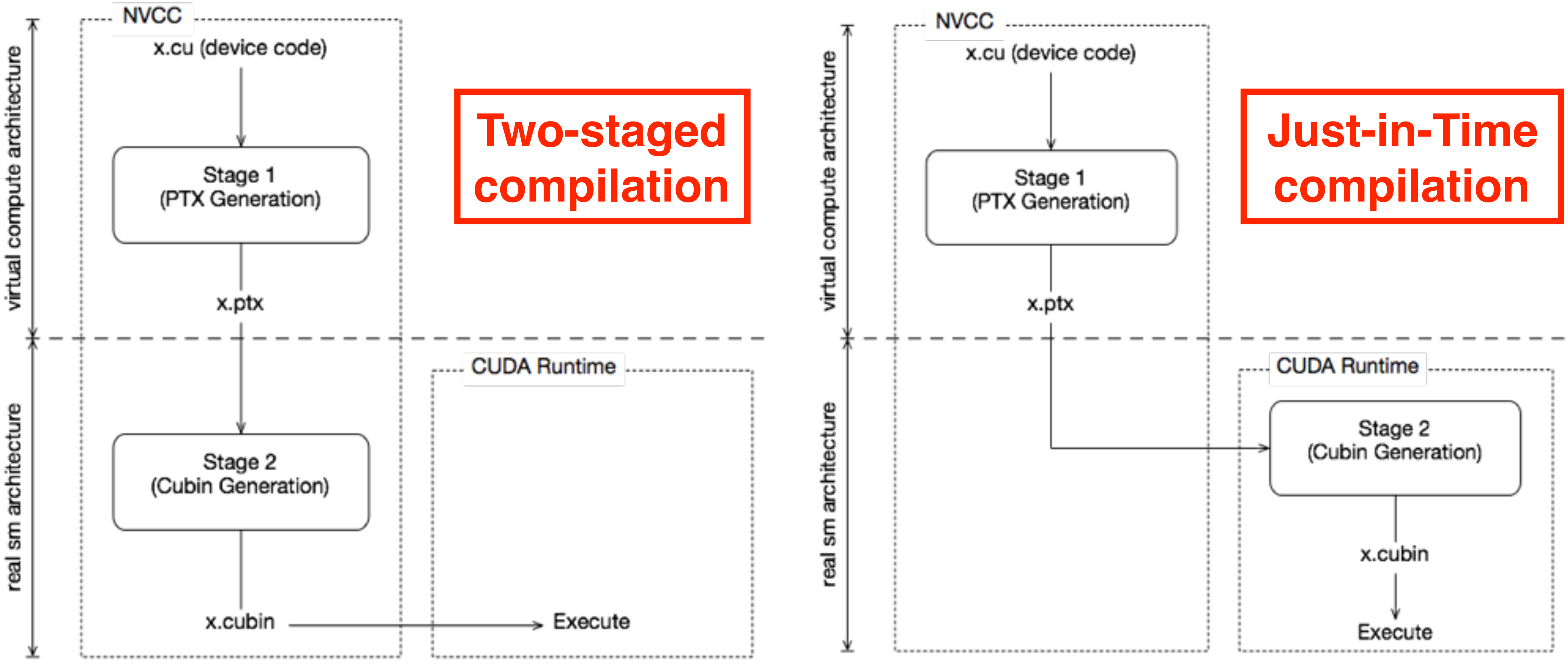
- Spatial locality = adjacent loop iterations access adjacent data
- A SIMD vector load usually *has* to access adjacent locations
- Some recent processors have "gather" instructions which can fetch from a different address per lane
- But performance is often serialised
- Branch predictability = each *individual* branch is mostly taken or not-taken (or is well-predicted by global history)

NVIDIA GPU Instruction Set Architecture

- Unlike most system processors, the instruction set target of the NVIDIA compilers is an abstraction of the hardware instruction set
- PTX (Parallel Thread Execution) assembler provides a stable instruction set for compilers as well as compatibility across generations of GPUs (PTX is an intermediate representation)
- The hardware instruction set is hidden from the programmer
- One PTX instruction can expand to many machine instructions
- Similarity with x86 microarchitecture, both translate to an internal form (microinstructions for x86). But translation happens (look at the diagram in the next slide):
  - in hardware at runtime during execution on x86
  - in software and load time on a GPU
- PTX uses virtual registers, the assignment to physical registers occurs at load time

Source code -> virtual GPU -> real GPU

- NVCC is the NVIDIA compiler
- cubin is the CUDA binary
- Runtime generation may be costly (increased load time), but it is normally cached



Source: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/

NVIDIA GPU ISA example

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

PTX instructions for one iteration of DAXPY

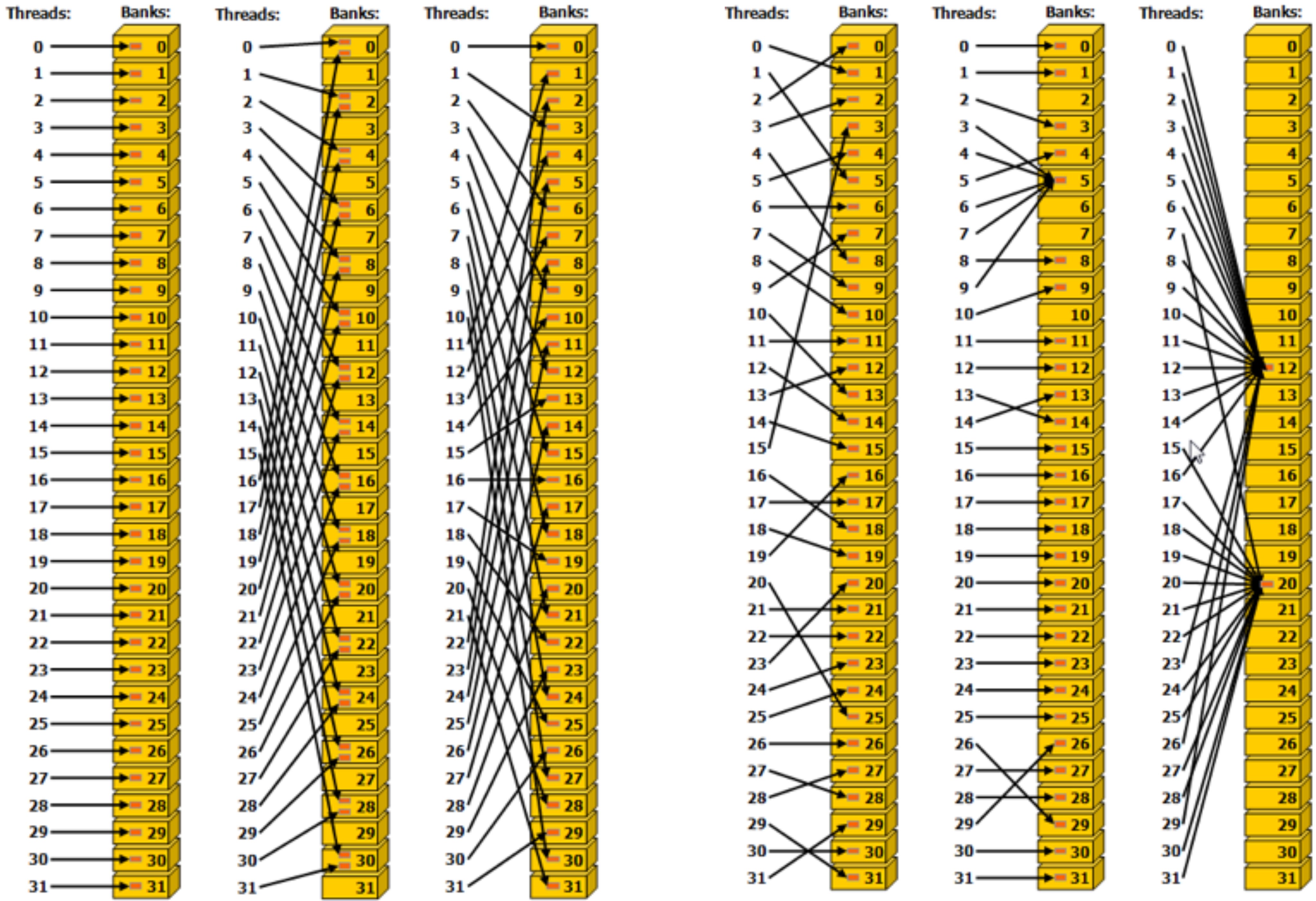
```
shl.u32 R8, blockIdx, 8   ; Thread Block ID Block size (256 or 2^8)
add.u32 R8, R8, threadIdx; R8 = i = my CUDA thread ID
shl.u32 R8, R8, 3         ; byte offset (double we shift 2^3)
ld.global.f64 RD0, [X+R8]; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]; RD0 = Y[i]
mul.f64 RD0, RD0, RD4     ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 RD0, RD0, RD2     ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0; Y[i] = sum (X[i]*a + Y[i])
```

Hennessy and Patterson’s Computer Architecture (5th ed.)

- Unlike vector architectures, GPUs don’t have separate instructions for sequential data transfers, stripped data transfers, and gather-scatter data transfers: all data transfers are gather-scatter
- Special Address Coalescing hardware recognises when the SIMD lanes within a thread of SIMD instructions are collectively issuing sequential addresses
- No loop incrementing or branching code like for the original for loop

Shared memory bank conflicts

- Shared memory has 32 banks that are organised such that successive 32-bit words map to successive banks
- Each bank has a bandwidth of 32 bits per clock cycle



Stride-1: Conflict-free

Stride-2: Conflicts

Stride-3: Conflict-free

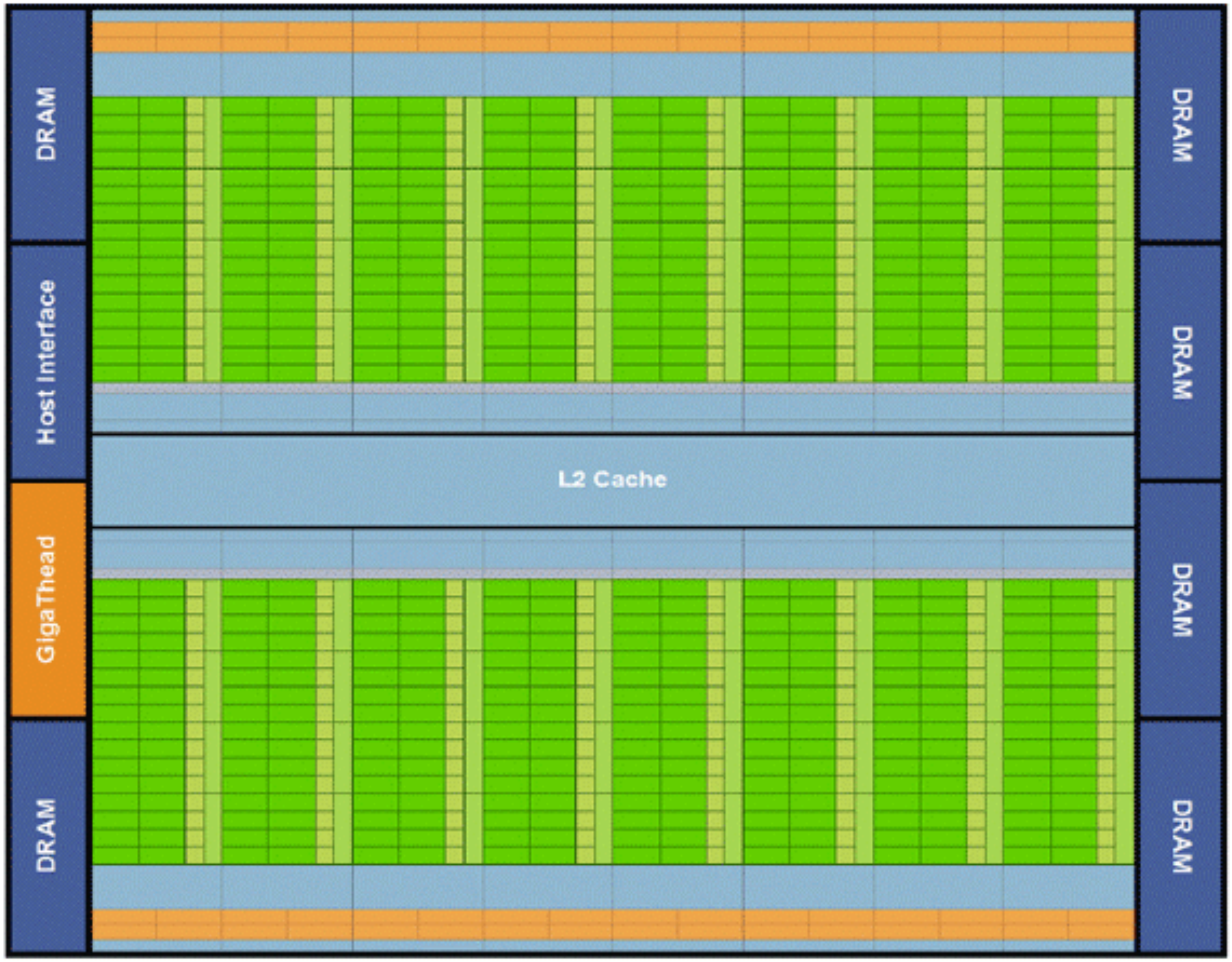
Random permutation: Conflict-free

If different threads access same *word*, no conflict

Some care is needed to avoid bank conflicts between different threads in a warp

Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability-2-x

Fermi GTX 480 (March 2010)

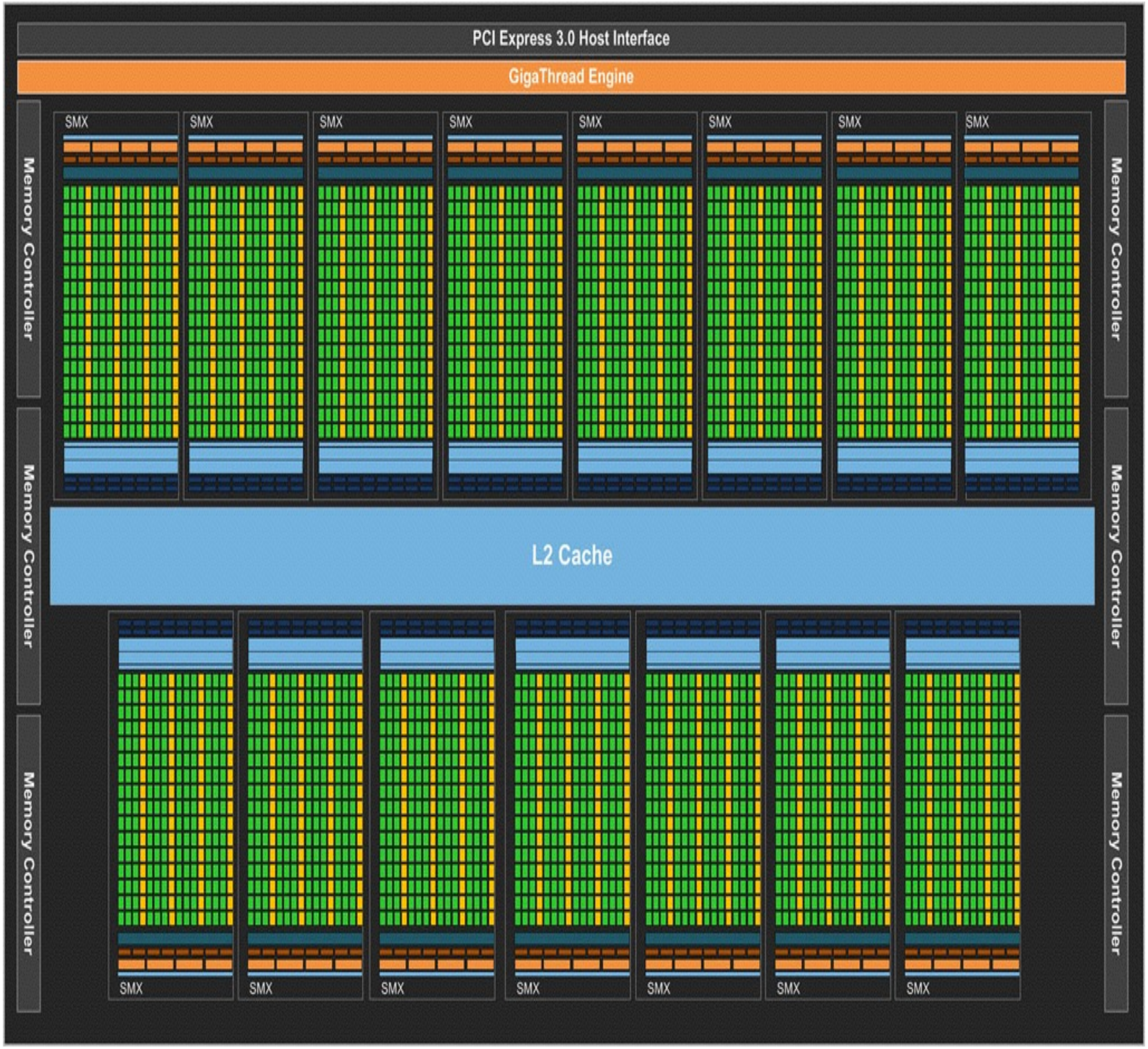


Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

Source: http://www.techspot.com/review/263-nvidia-geforce-gtx-480/page2.html

Fermi GTX 480:
32 FUs per processor
Peak (SP): 1340 GFLOPS
BW: 177 GB/s
Core clock: 700 MHz

Kepler GTX Titan (February 2013)



Source: http://www.anandtech.com/show/5699/nvidia-geforce-gtx-680-review/2

Kepler GTX Titan:

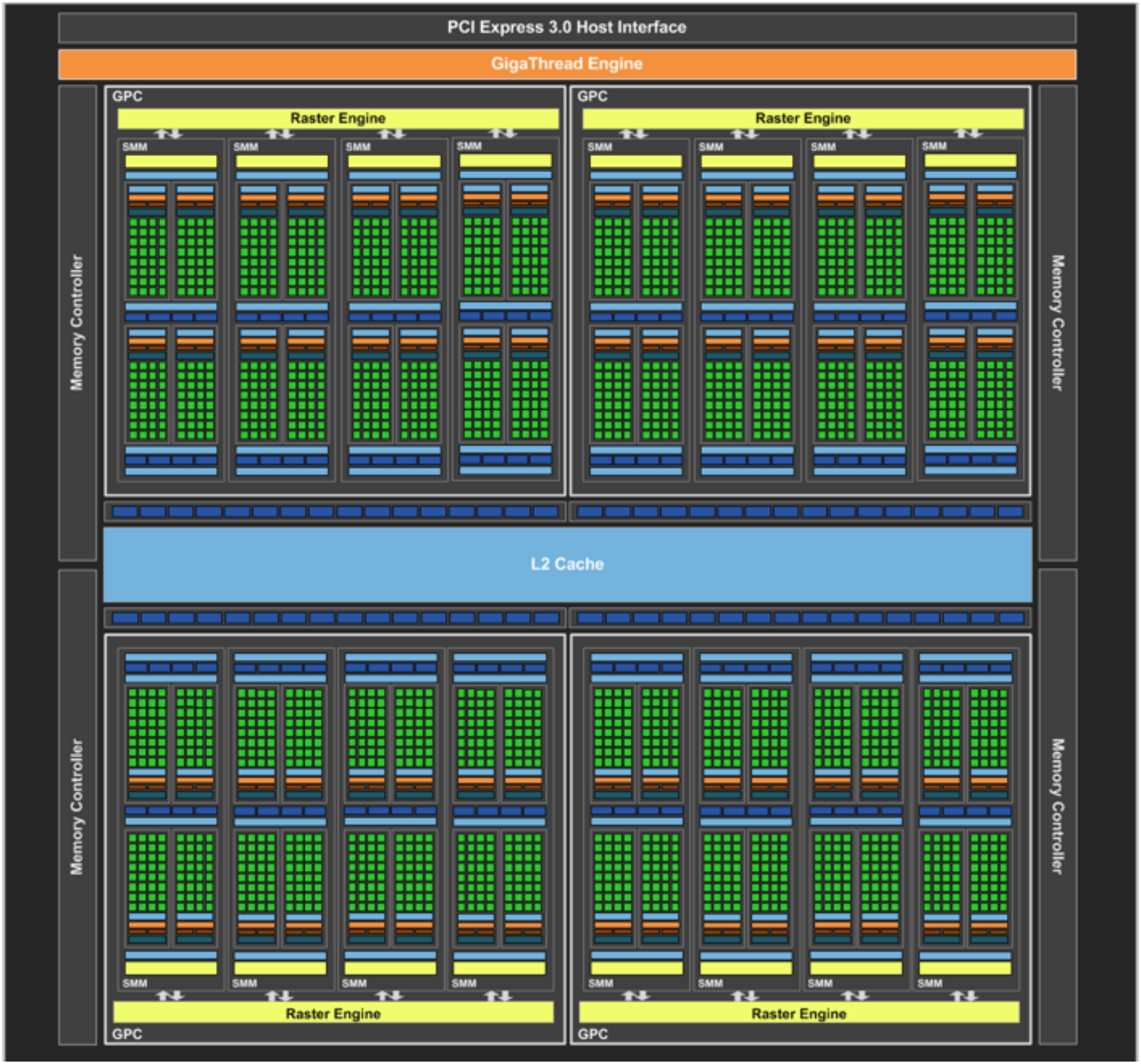
192 FUs per SMX

Peak (SP): 4.5 TFLOPS

BW: 290 GB/s

Core clock: 830 MHz

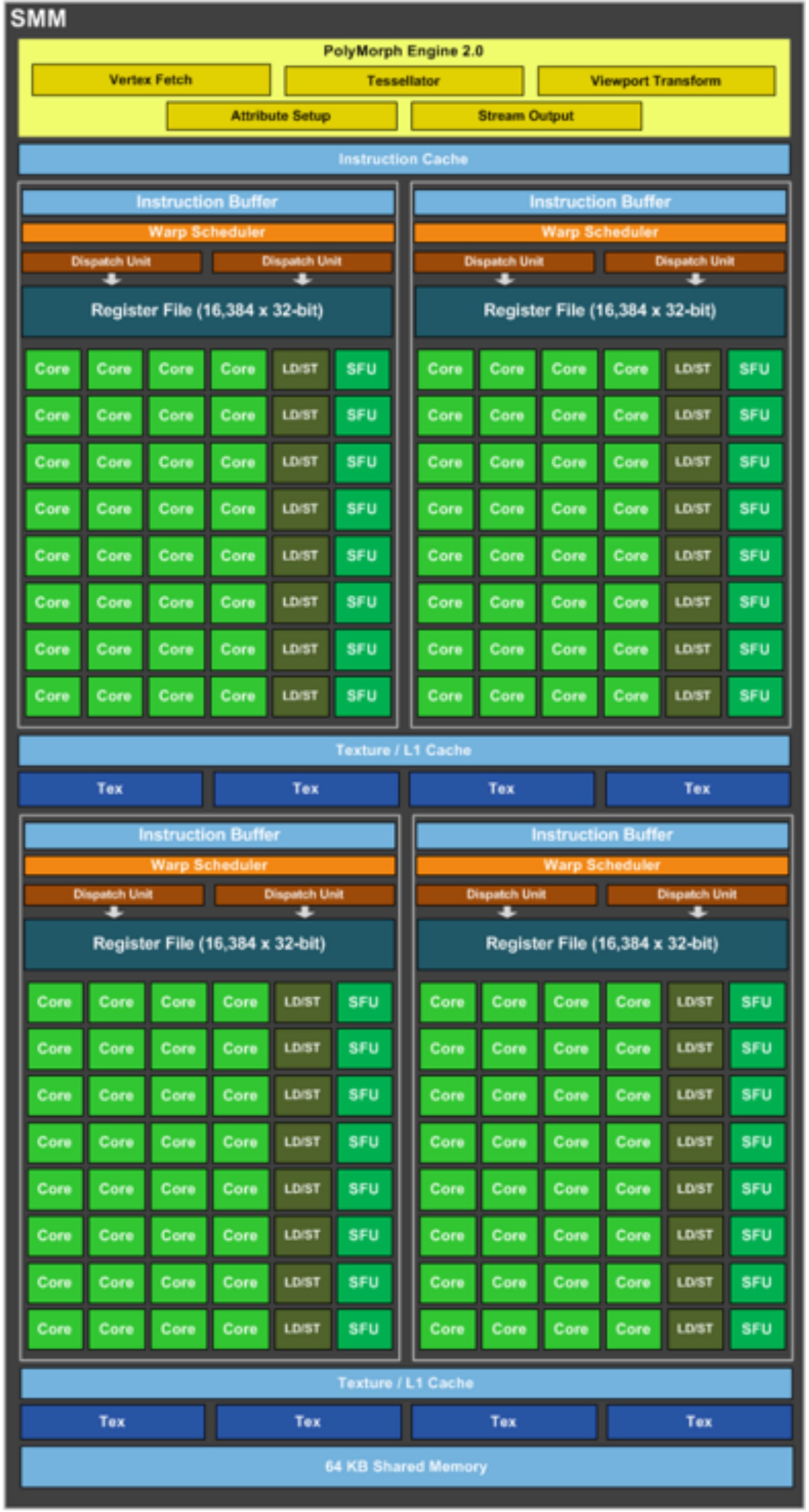
Maxwell GTX 980 Ti (June 2015)



Maxwell GTX 980 Ti:

128 FUs per SMM
Peak (SP): 5.6 TFLOPS
BW: 336 GB/s
Core clock: 1 GHz

Source: http://www.anandtech.com/show/8526/nvidia-geforce-gtx-980-review/3



Source: http://www.anandtech.com/show/8526/nvidia-geforce-gtx-980-review/2

Maxwell GTX 980 Ti:

128 FUs per SMM

Peak (SP): 5.6 TFLOPS

BW: 336 GB/s

Core clock: 1 GHz

It is a heterogeneous world

TITAN
4998 GFLOPS
< 400 W

GTX 870M
2827 GFLOPS
< 100 W

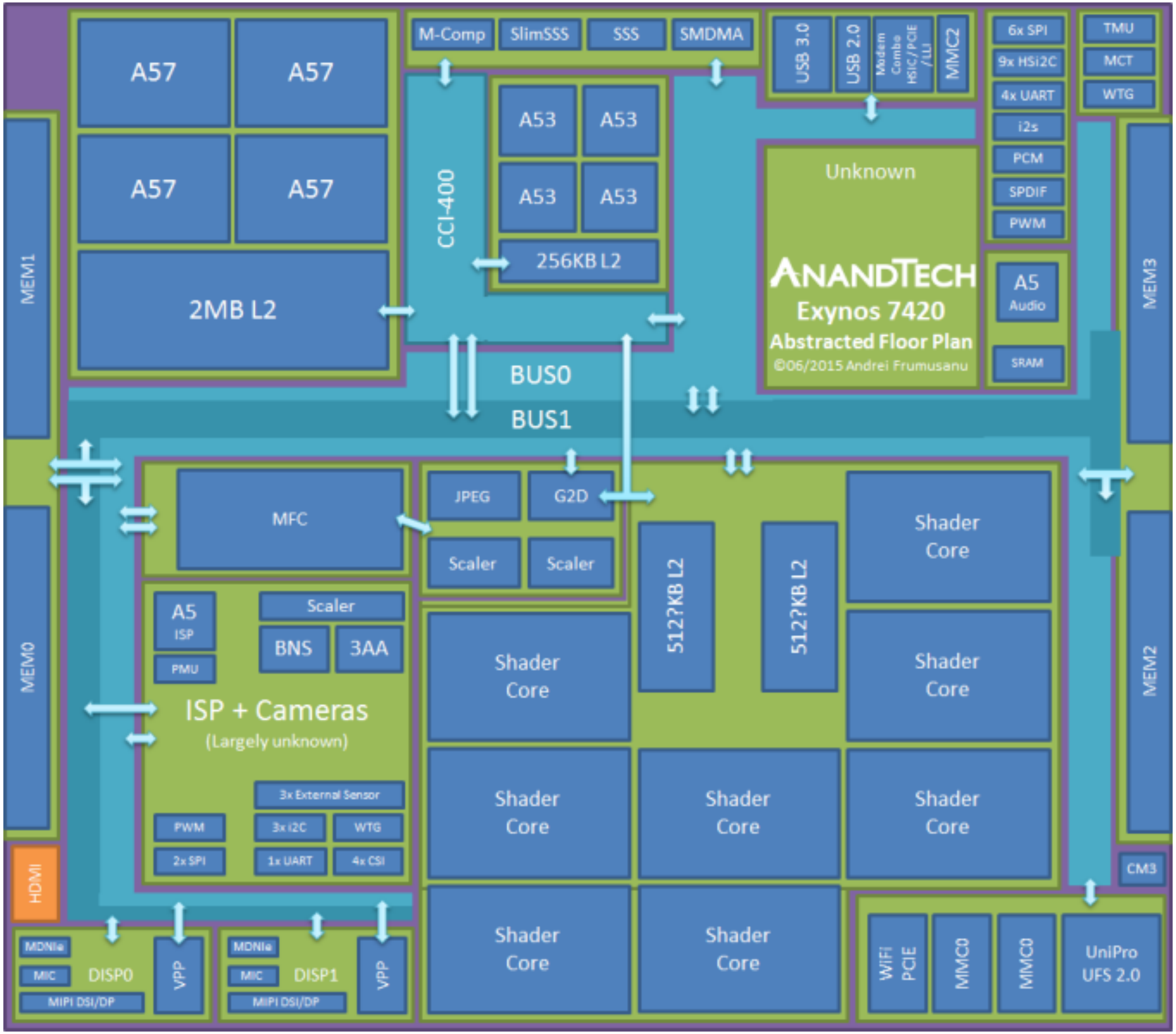
TK1
404 GFLOPS
< 20 W

ODROID-XU3
170 GFLOPS
< 10 W

Arndale
87 GFLOPS
< 5 W

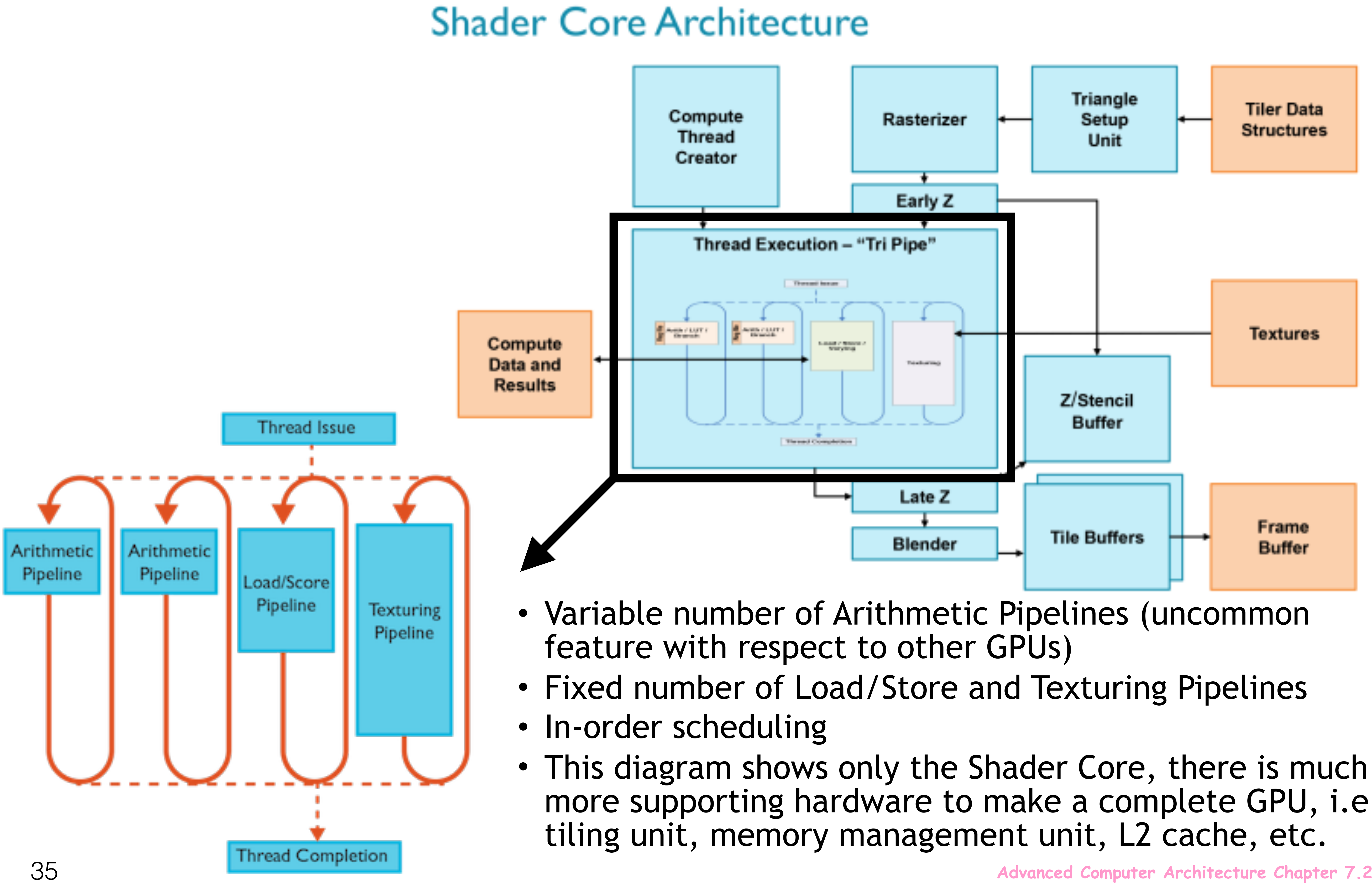
Nardi et al. "Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM", IEEE ICRA, 2015

ARM-based Samsung Exynos 7420 System on Chip (SoC) Reverse engineered



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

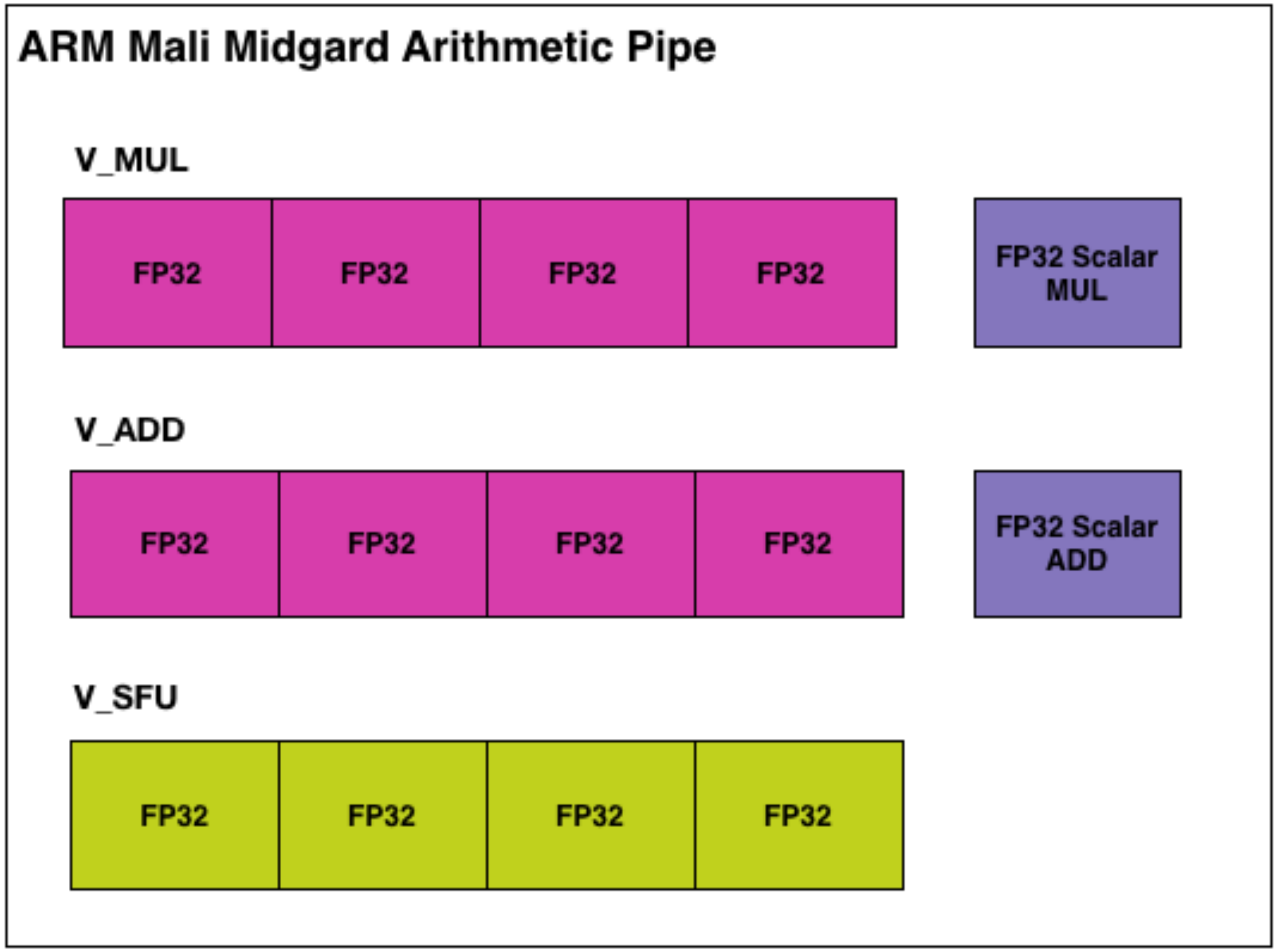
ARM MALI GPU: Midgard microarchitecture



- 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

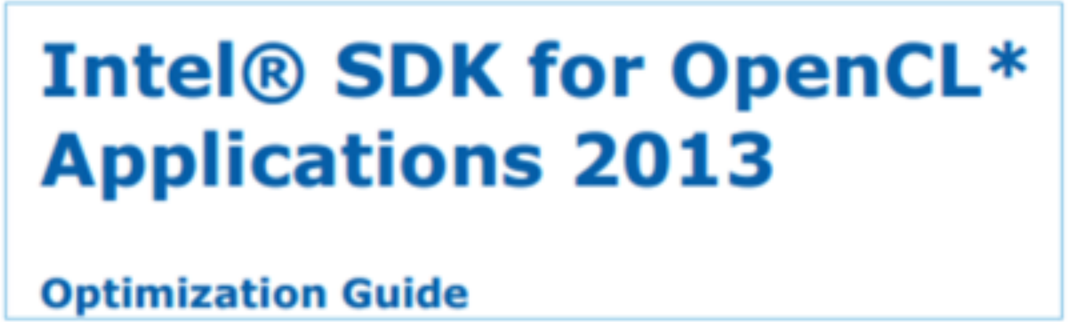
To run optimally OpenCL code on Mali GPUs means mainly to locate and remove optimisations for alternative compute devices:

- 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

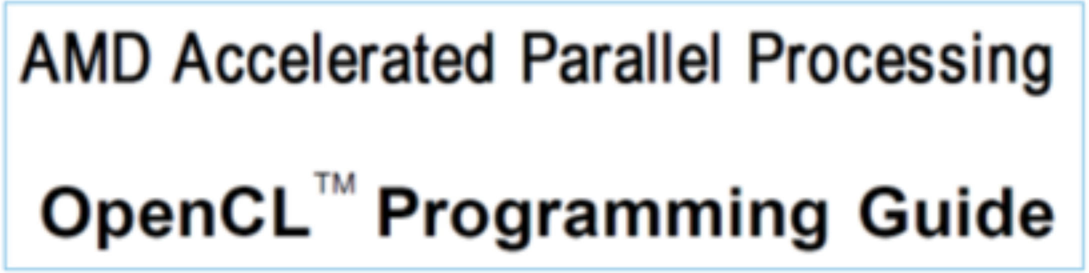
Portability: code vs performance

OpenCL performance depends on how the code is written



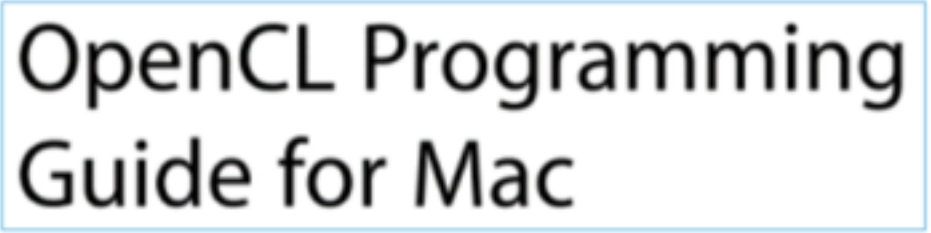
Intel® SDK for OpenCL* Applications 2013

Optimization Guide

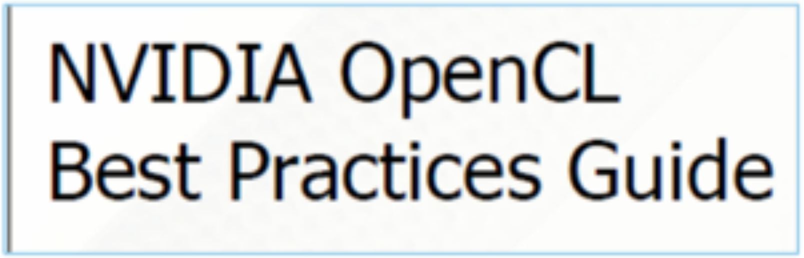


AMD Accelerated Parallel Processing

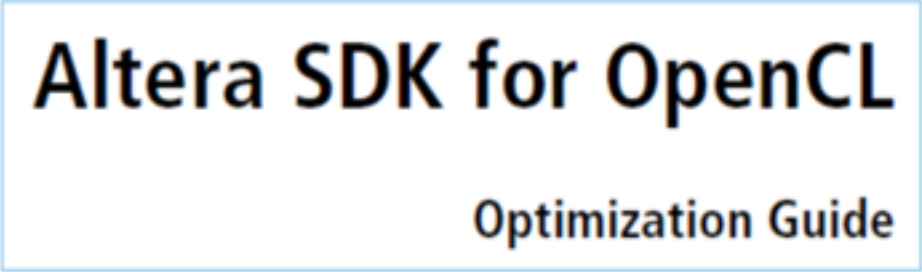
OpenCL™ Programming Guide



OpenCL Programming Guide for Mac

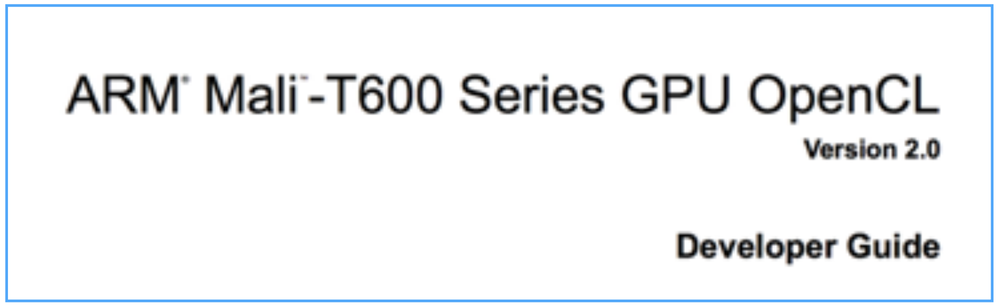


NVIDIA OpenCL Best Practices Guide



Altera SDK for OpenCL

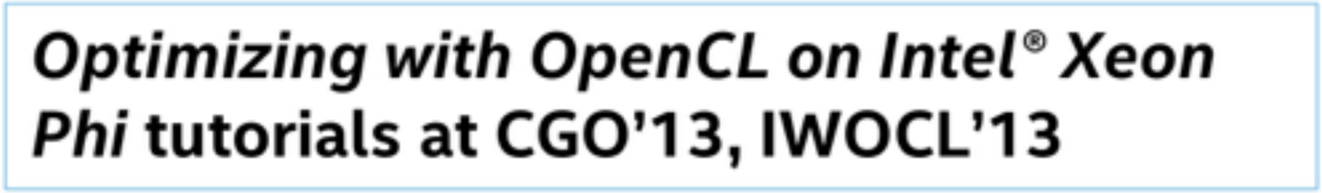
Optimization Guide



ARM® Mali™-T600 Series GPU OpenCL

Version 2.0

Developer Guide



***Optimizing with OpenCL on Intel® Xeon Phi* tutorials at CGO'13, IWOCL'13**

Source: "OpenCL heterogeneous portability – theory and practice", Ayal Zaks (Intel), PEGPUM 2014

Where to start

- CUDA programming guide: https://docs.nvidia.com/cuda/cuda-c-programming-guide/
- OpenCL http://www.nvidia.com/content/cudazone/download/opencl/nvidia_opencl_programmingguide.pdf
  http://developer.amd.com/tools-and-sdks/opencl-zone/