Lecture 2: Overview of GPU Architecture

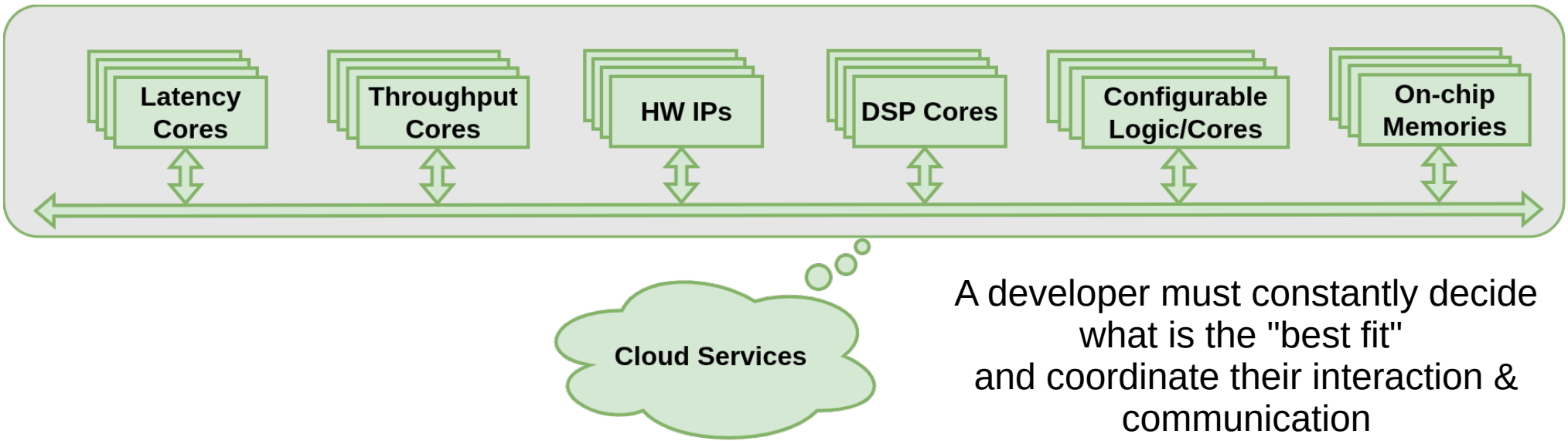
CS599: Programming Massively Parallel Multiprocessors and Heterogenous Systems (Understanding and programming the devices powering AI)

Jonathan Appavoo

Heterogeneous Computing Era

Consider Mobile SOC

"Heterogeneous Parallel Computing System: Use different types of devices that best fit particular parts of the application(s)"

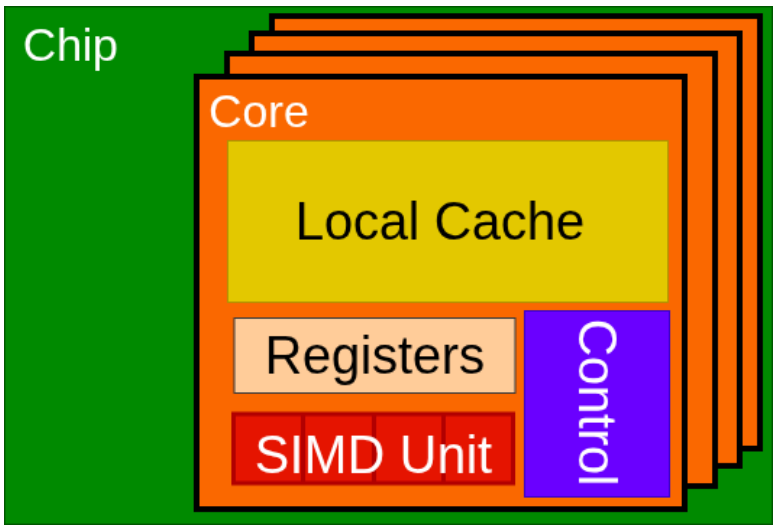


A developer must constantly decide what is the "best fit" and coordinate their interaction & communication

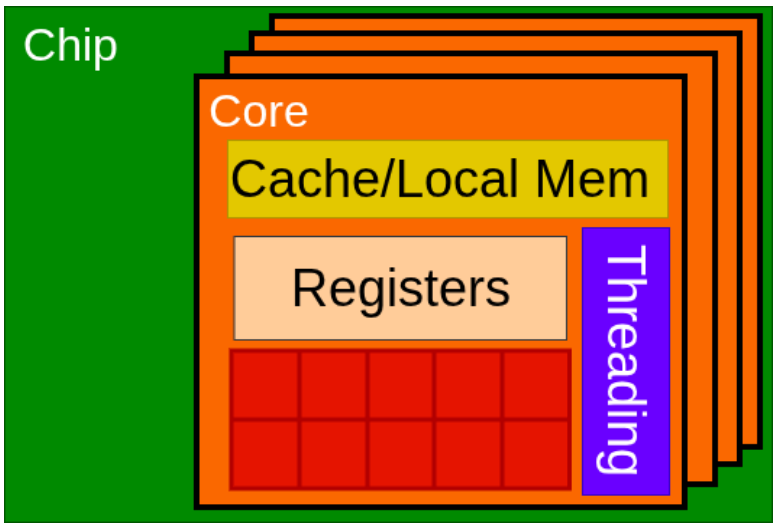
Slide inspired by the NVidia Teaching Tool Kit, Lec 1.2: http://gputeachingkit.hwu.crhc.illinois.edu/

CPU vs GPU -- Designed very differently

CPU
Latency Oriented Cores



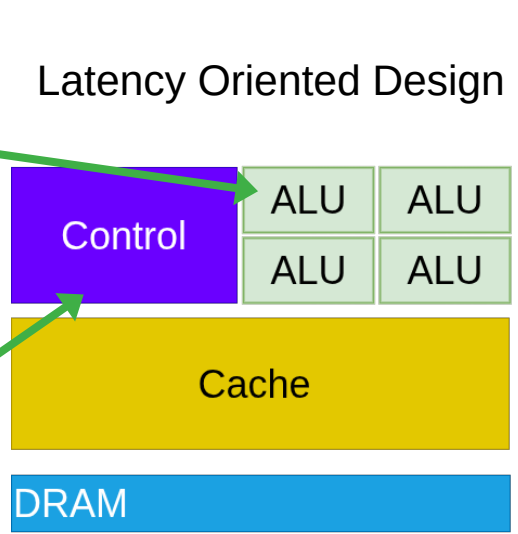
GPU
Throughput Oriented Cores

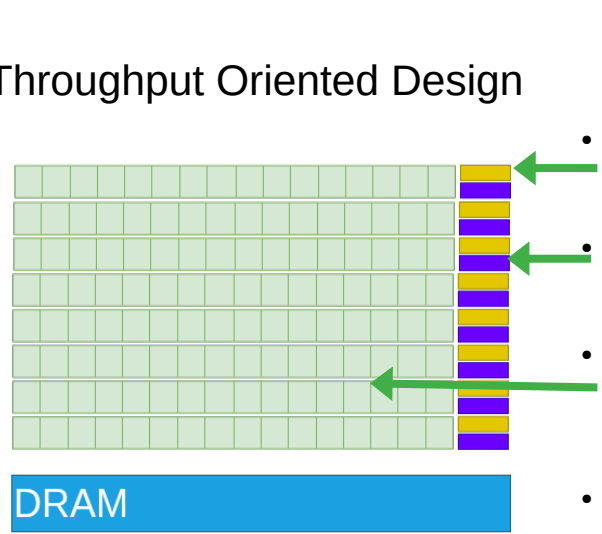


Slide inspired by the NVidia Teaching Tool Kit, Lec 1.2: http://gputeachingkit.hwu.crhc.illinois.edu/

CPU vs GPU -- Designed very differently

- Powerful ALU
  - Reduced operational latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency
  - etc





- Small caches
  - To boost memory throughput
- Simple control
  - No branch prediction
  - No data forwarding
- Energy efficient ALUs
  - Many, long latency but heavily pipelined for high throughput
- Requires massive number of threads to tolerate latencies
  - Threading logic
  - Thread state

Nice discussion of data forwarding: https://en.wikipedia.org/wiki/Operan%20d_forwardinghttps://en.wikipedia.org/wiki/Operand_forwarding
An interesting paper regarding the tradeoffs between caches and multithreading: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=159037

Slide inspired by the NVidia Teaching Tool Kit, Lec 1.2: http://gputeachingkit.hwu.crhc.illinois.edu/

"Winning" Applications use both CPU and GPU

- CPUs for sequential parts where latency matters
  - CPUs can be 10X+ faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
  - GPUs can be 10X+ faster than CPUs for parallel code

Slide inspired by the NVidia Teaching Tool Kit, Lec 1.2: http://gputeachingkit.hwu.crhc.illinois.edu/

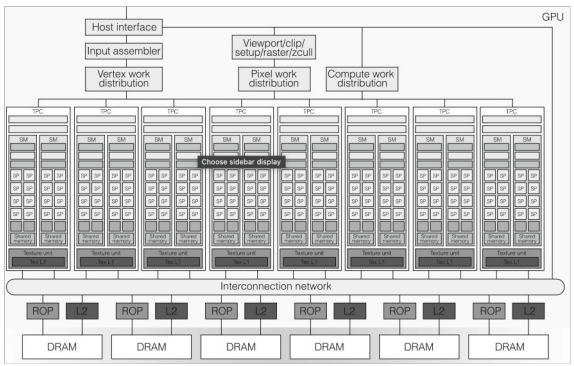
My 2 cents on the “State of the Art”

- The company line : 'This is just the way it is. We must abandon the friendly/childish homogenous view of computing and accept the realities of a complex heterogeneous model of computing. -- Humans (with perhaps the help of GenAI) have to do the work.
- Research:
  - While I accept that the realities of physics, I don't think we have to accept Complexity in programming -- abandoning the success of a simple abstract model of a machine (eg, Turing) seems like a step backwards (and a lack of creativity on our parts).
    - But I also don't believe in magic programming languages or compilers
  - Rather maybe there are fundamental relationships that we can use to convert one to the other automatically, after all don't we ourselves do this in our heads ... but that's another story ;-)

On to GPU

Build our knowledge by looking at NVidia GPU Evolution

Tesla* G80 (2008)



https://www.nvidia.co.uk/content/PDF/Geforce_8800/GeForce_8800_GPU_Architecture_Technical_Brief.pdf

Volta V100 (2017)

https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

Hopper H100 (2022)

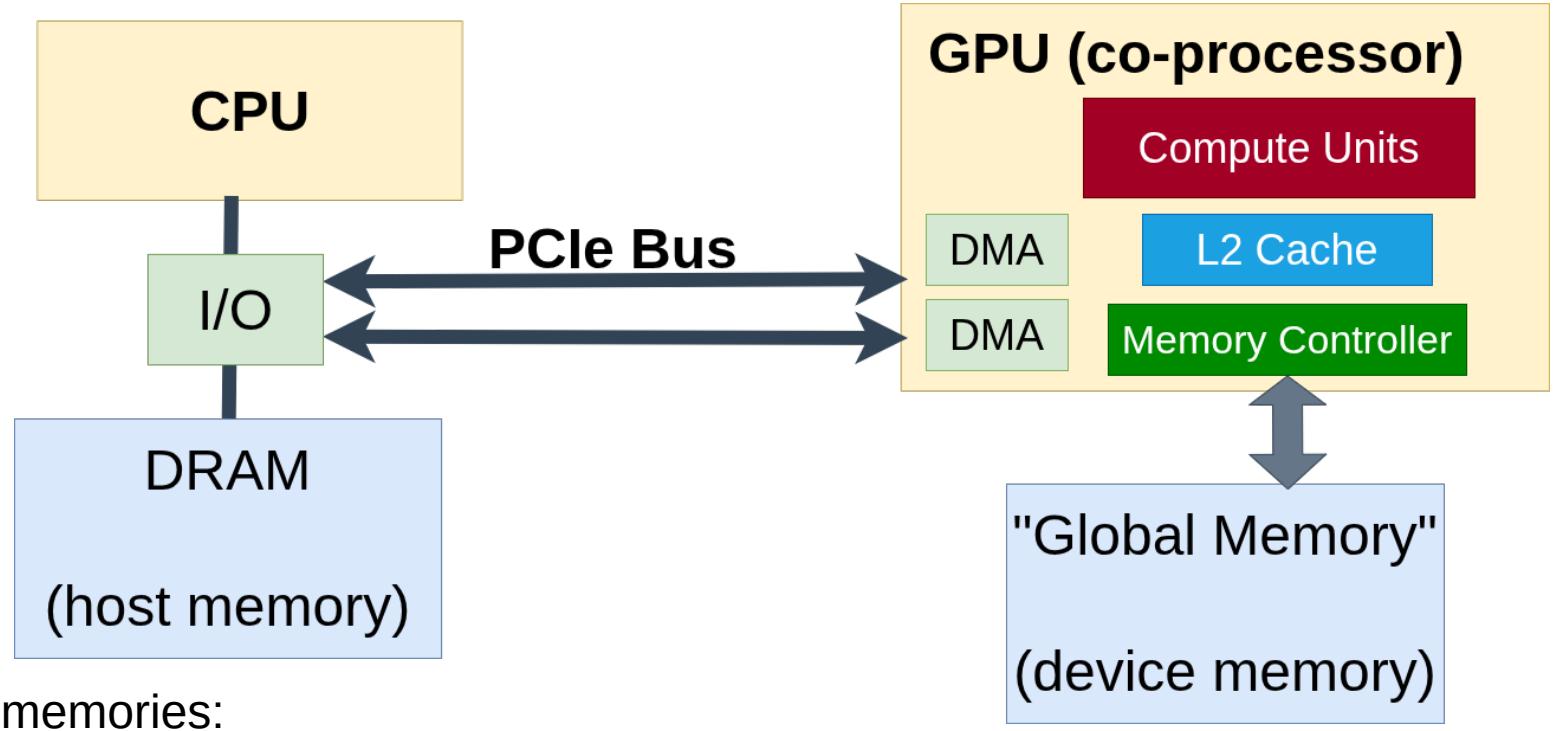
https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c

https://ieeexplore.ieee.org/document/4523358

NVidia Micro-architectures

Micro-architecture	Release	Compute Capability	GPU code name
G70	2005	-	GXX, GT2XX
Tesla	2006	1.0-1.3	GXX, GT2XX
Fermi	2010	2.0-2.1	GFXXX
Kepler	2012	3.0-3.7	GKXXX
Maxwell	2014	5.0-5.3	GMXXX
Pascal	2016	6.0-6.2	GPXXX
Volta	2017	7.0-7.2	GVXXX
Turing	2018	7.5	TUXXX
Ampere	2020	8.0-8.6	GAXXX
Lovelace	2022	8.9	ADXXX
Hopper	2022	9.0	HXXX
Blackwell	2024	9.0+ (expected)	BXXX
Rubin (Next Gen)	~2025-26	TBD	RXXX

GPU Architecture: 100,000' View



Separate memories:

- PCIe can become a bottleneck
- max 16 GB/s (v3.0), 32 GB/s (v4.0), 64 GB/s (v5.0) 128 GB/s (v6.0) assuming x16 bidirectional lanes (x16)

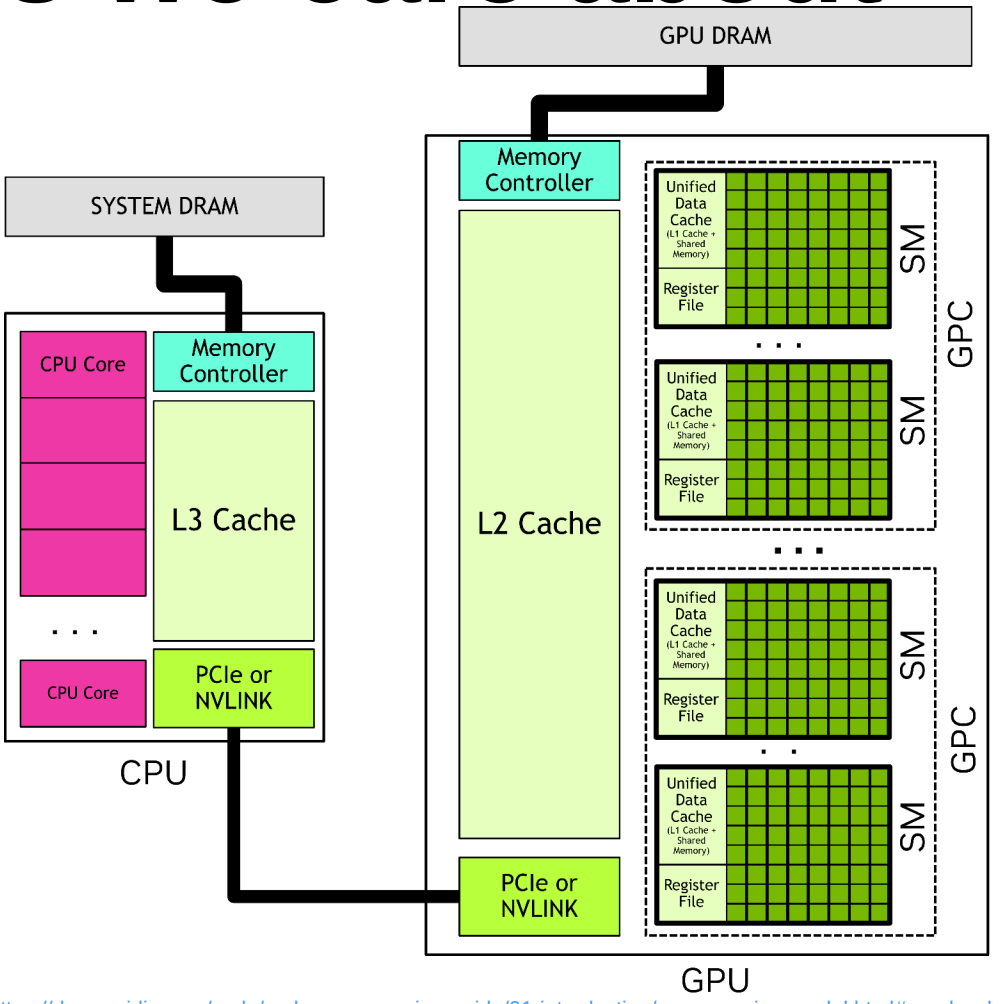
BEWARE: "unified memory" only for pinned host memory but still slow (gated by PCIe)

Three dimensions we care about

1) Compute: resources, organization, and model

2) Memory: types, organization, and speeds

3) I/O: PCIe --- speed, synchronous vs acynchronous



https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html#gpu-hardware-model

In the beginning, there was the G80 (Tesla)

Published a HotChips paper in 2008 that is a great starting point

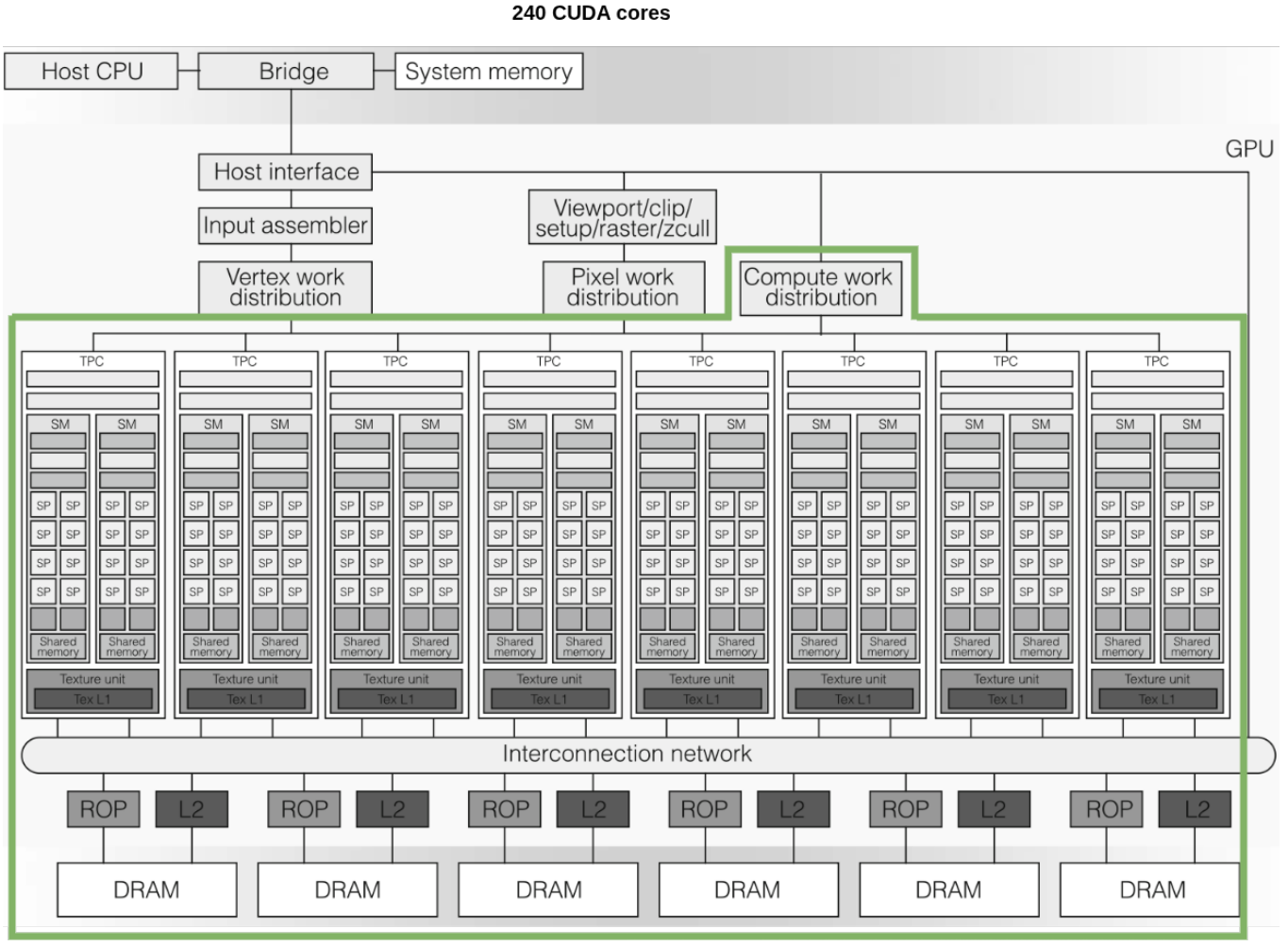
"NVIDIA TESLA: A UNIFIED GRAPHICS AND COMPUTING ARCHITECTURE", Lindholm et. al, HotChips 2008 (https://ieeexplore.ieee.org/document/4523358)

2006 Nvidia GeForce 8800 product brief -- Introduces compute into product line [p12 and on]

(https://www.nvidia.co.uk/content/PDF/Geforce_8800/GeForce_8800_GPU_Architecture_Technical_Brief.pdf)

G80

- Compute work distribution units deliver work to processors in [a] round-robin
- # SMs determines a GPU's programmable processing performance (unit of scalability)
- Various regroupings over the product lines but SM is a fundamental unit.
- Nvidia worked hard to replace special purpose cores with array of uniform (general purpose cores)
  - Lots of challenges had to be overcome to maintain performance and utilization



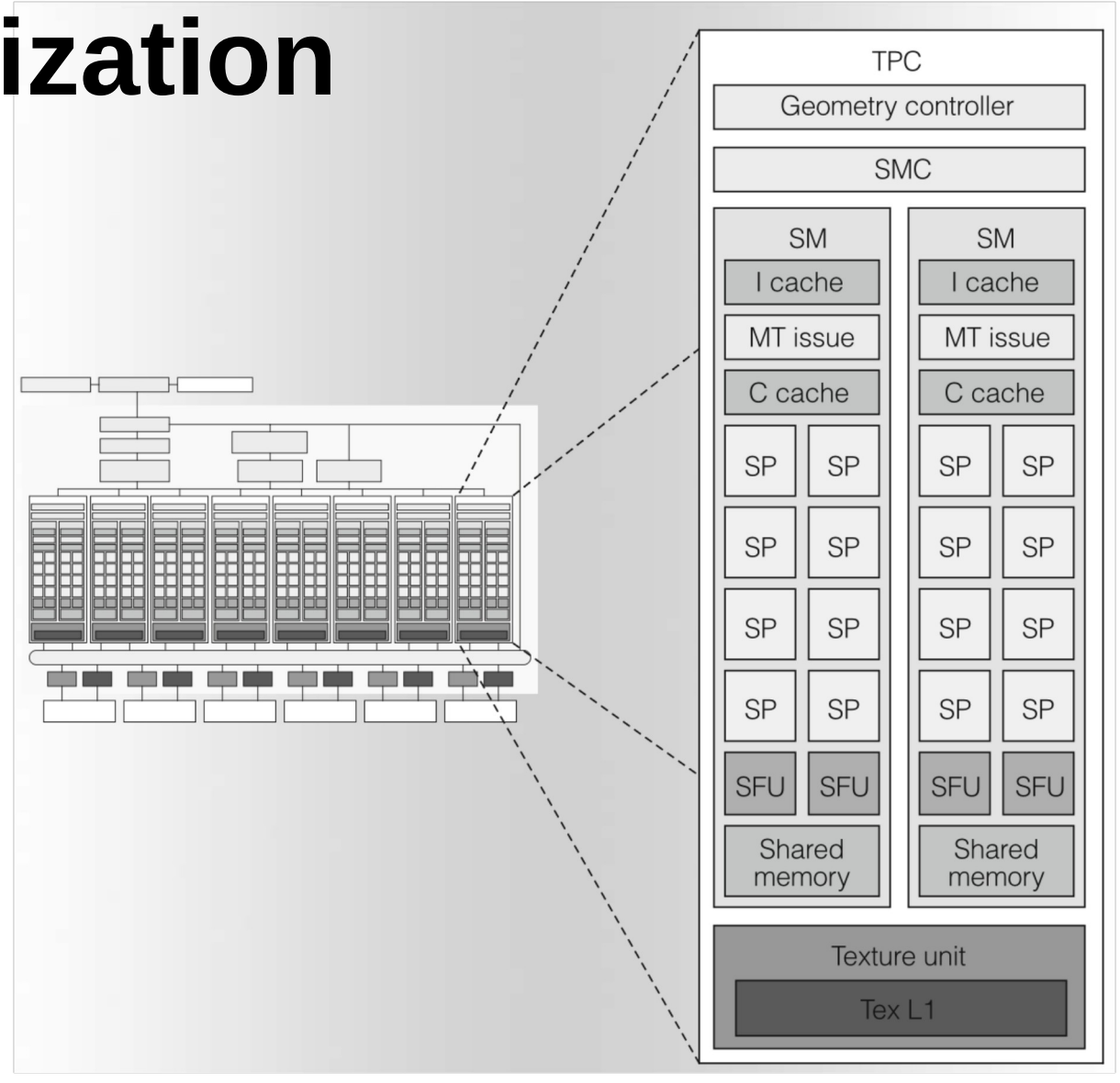
Physical Scalability == Replication

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4523358

Compute Organization

Hierarchy: array of arrays

- Names seem to change a little over the first few generations
- Physical organization dictates locality:
  - what is shared
  - what works as a unit
- Work from host is distributed evenly across units of Texture/ Processor Clusters (TPC)
  - Which then further decomposes the work



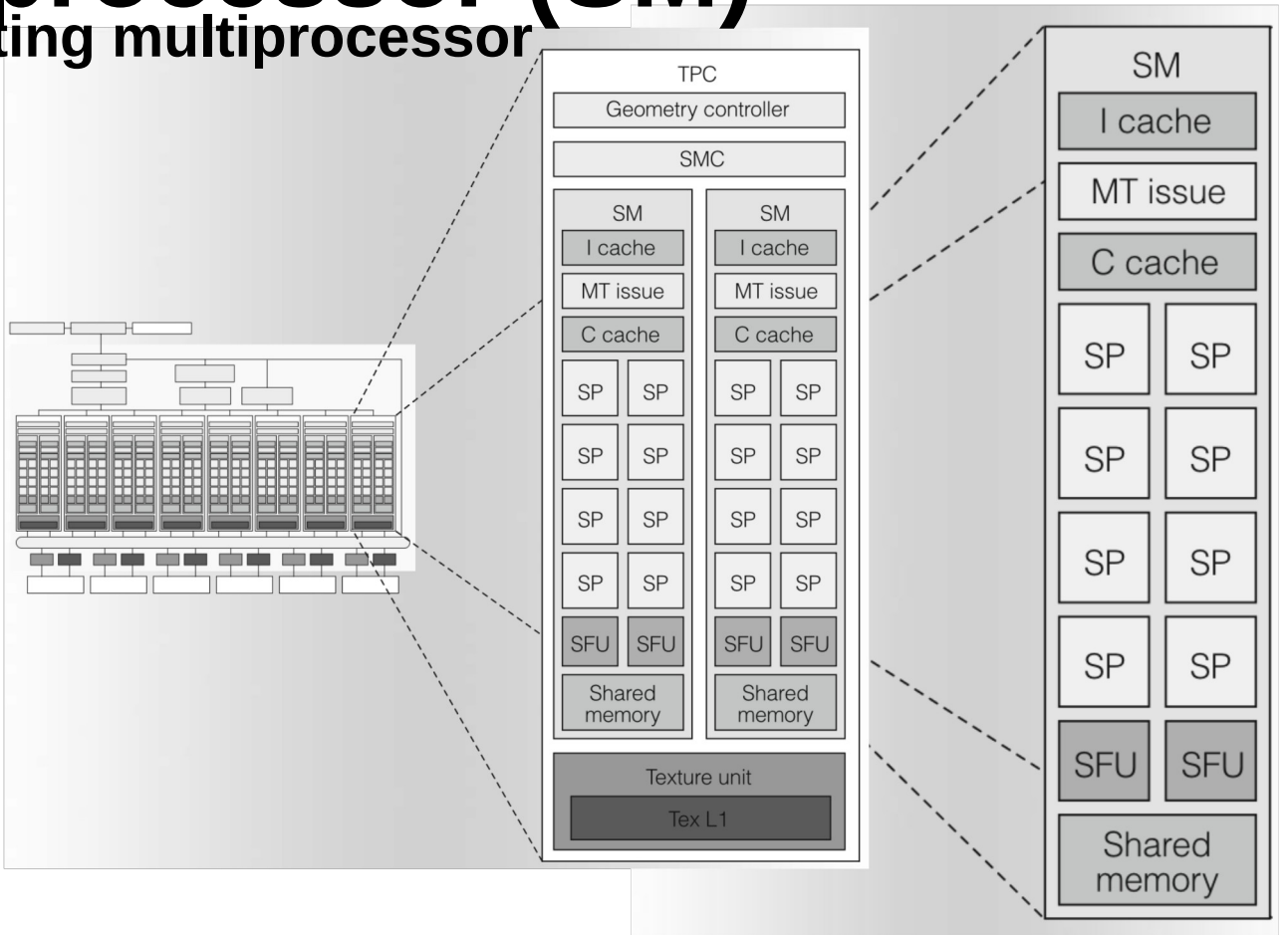
https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4523358

Streaming Mulitprocessor (SM)

Unified graphics and computing multiprocessor

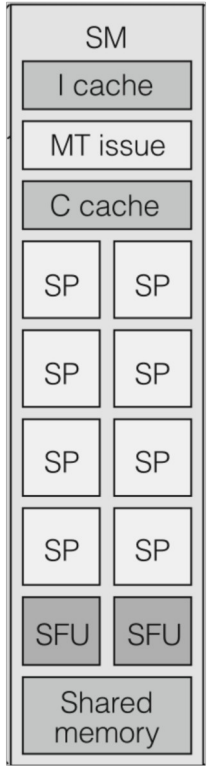
Executes "Parallel computing programs"

- Eight streaming processor (SP)
- Two special function units (SFUs)
- Multithreaded
- instruction fetch and issue unit (MT Issue)
- instruction cache (I cache)
- read-only constant cache (C cache)
- 16-Kbyte read/write shared memory

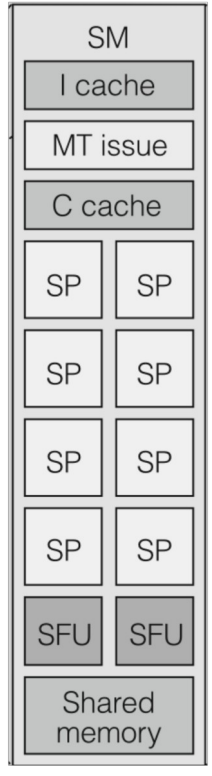


"a CUDA kernel is a C program for a single thread that describes how one thread computes a result."

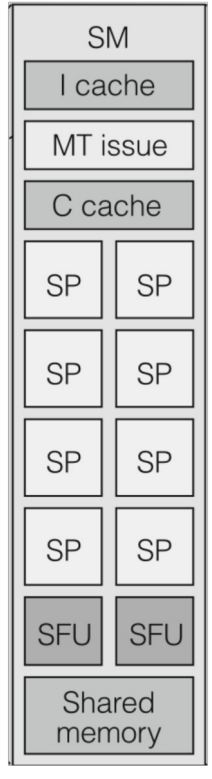
"To efficiently execute hundreds of threads in parallel while running several different programs, the SM is hardware multithreaded. It manages and executes up to 768 concurrent threads in hardware with zero scheduling overhead."



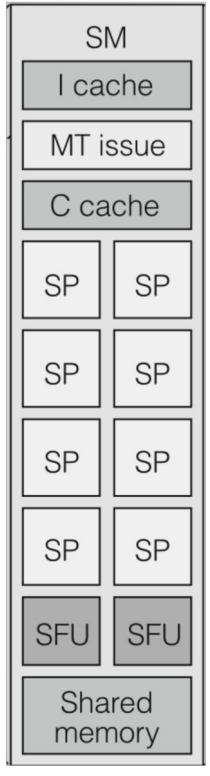
"Each SM thread has its own thread execution state and can execute an independent code path."



"Concurrent threads of computing programs can synchronize at a barrier with a single SM instruction. "

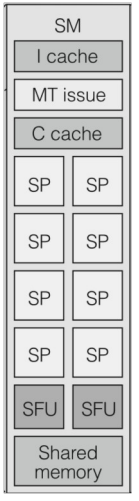


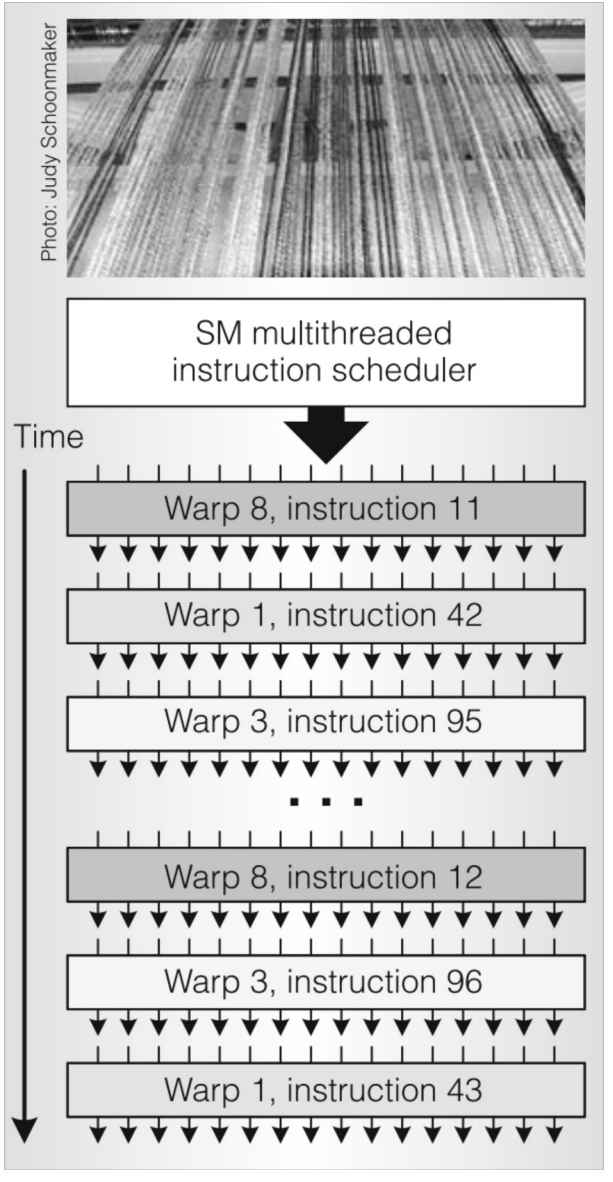
"Lightweight thread creation, zero-overhead thread scheduling, and fast barrier synchronization support very fine-grained parallelism efficiently. "



SIMT & Thread warps

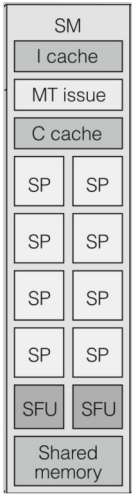
- "Tesla SM uses a new processor architecture we call single-instruction, multiple-thread (SIMT)"
- The SM's SIMT multithreaded instruction unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps
- The term warp originates from weaving, the first parallel thread technology.
- SM manages a pool of **24 warps**, with a total of **768 threads**.

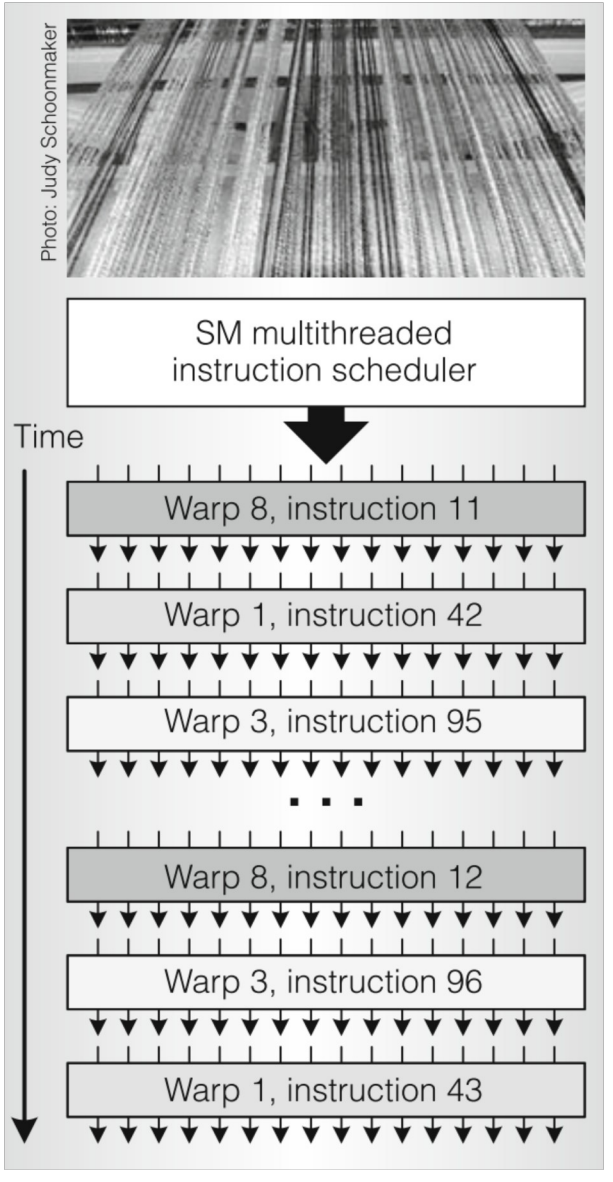




SIMT & Thread warps

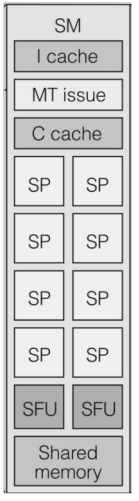
- Individual threads composing a SIMT warp are of the same type and start together at the same program address, but they are otherwise free to branch and execute independently.
- At each instruction issue time, the SIMT multithreaded instruction unit selects a warp that is ready to execute and issues the next instruction to that warp's active threads.
- **A SIMT instruction is broadcast synchronously to a warp's active parallel threads; individual threads can be inactive due to independent branching or predication.**

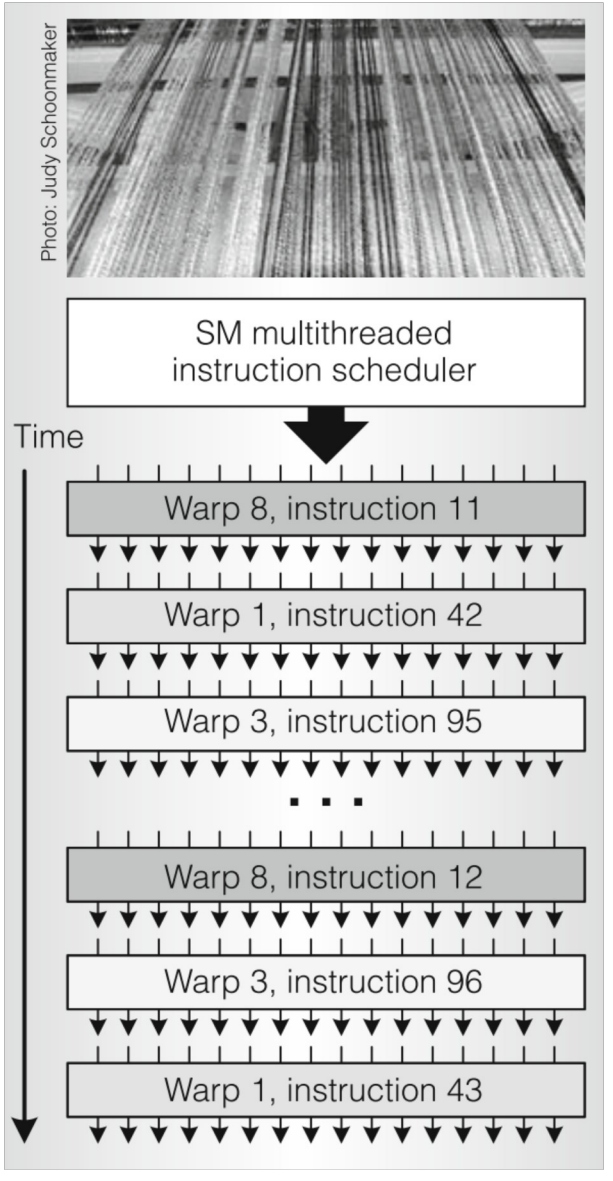




SIMT & Thread warps

1) Enables Programmers to write thread-level parallel code
2) **And** data-parallel code
3) Programmers can essentially ignore SIMT (eg warps)
4) **Substantial performance improvements if code seldom requires threads in a warp to diverge**
5) Analogous to cache line - can be ignored for correctness but consider to get peak performance accounted for





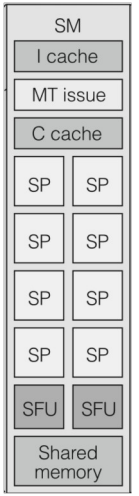
" In contrast to SIMD vector architectures, SIMT enables programmers to write thread-level parallel code for independent threads as well as data-parallel code for coordinated threads. For program correctness, programmers can essentially ignore SIMT execution attributes such as warps; however, they can achieve substantial performance improvements by writing code that seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional codes: Programmers can safely ignore cache line size when designing for correctness but must consider it in the code structure when designing for peak performance. SIMD vector architectures, on the other hand, require the software to manually coalesce loads into vectors and to manually manage divergence."

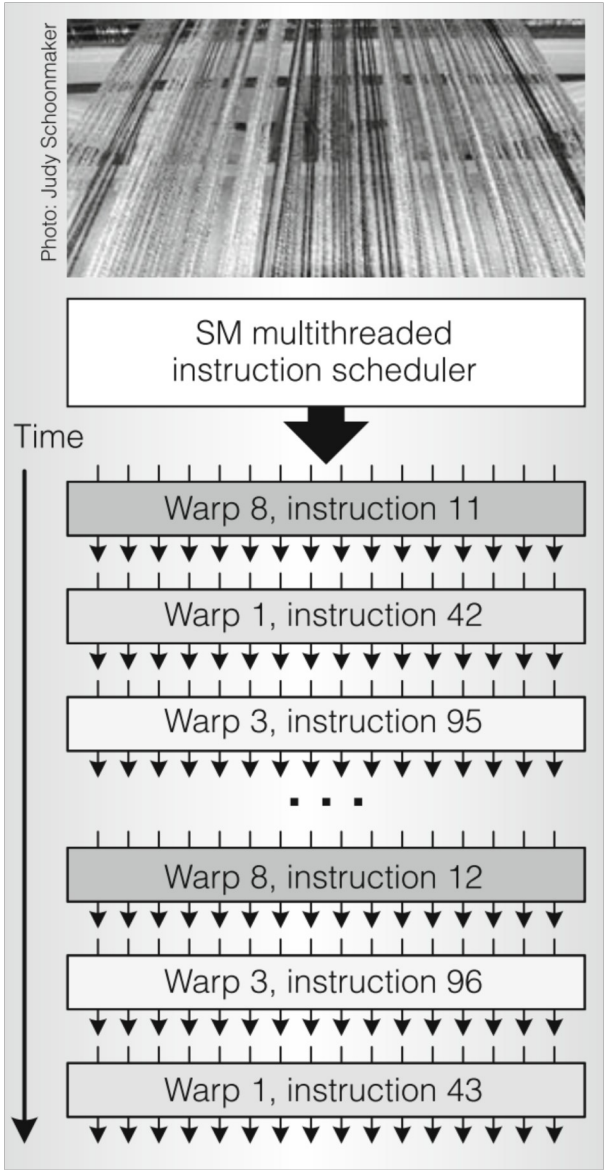
SIMT & Thread warps

1) SM scheduler packs 32 compute threads, from work assigned to this SM, into a warp

2) SIMT design allows SM to have a single fetch and decode unit across all threads of a warp

3) Requires full warp of active threads for peak performance





SIMT warp scheduling. The SIMT approach of scheduling independent warps is simpler than previous GPU architectures' complex scheduling. A warp consists of up to 32 threads of the same type—vertex, geometry, pixel, or compute. The basic unit of pixel-fragment shader processing is the 2 X 2 pixel quad. The SM controller groupsmeight pixel quads into a warp of 32 threads. It similarly groups vertices and primitives into warps and packs 32 computing threads into a warp. 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.

SM Multithreading - scheduling

Pool of 24 Warp Contexts (32*24=768 Threads)

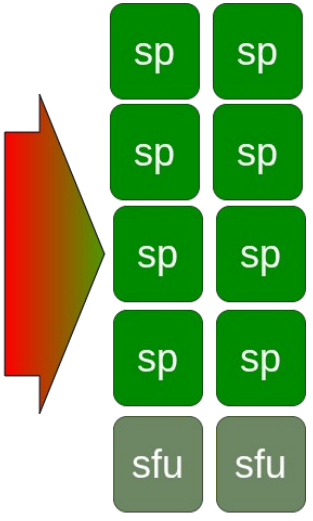
1	2	3
4	5	6
7	8	9
10	11	12
13	14	15
16	17	18
19	20	21
22	23	24

1.5-GHz processor clock rate/2 = 0.75 Ghz scheduler clock rate

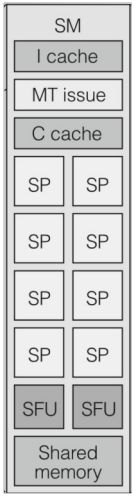
Each scheduler cycle pick a warp

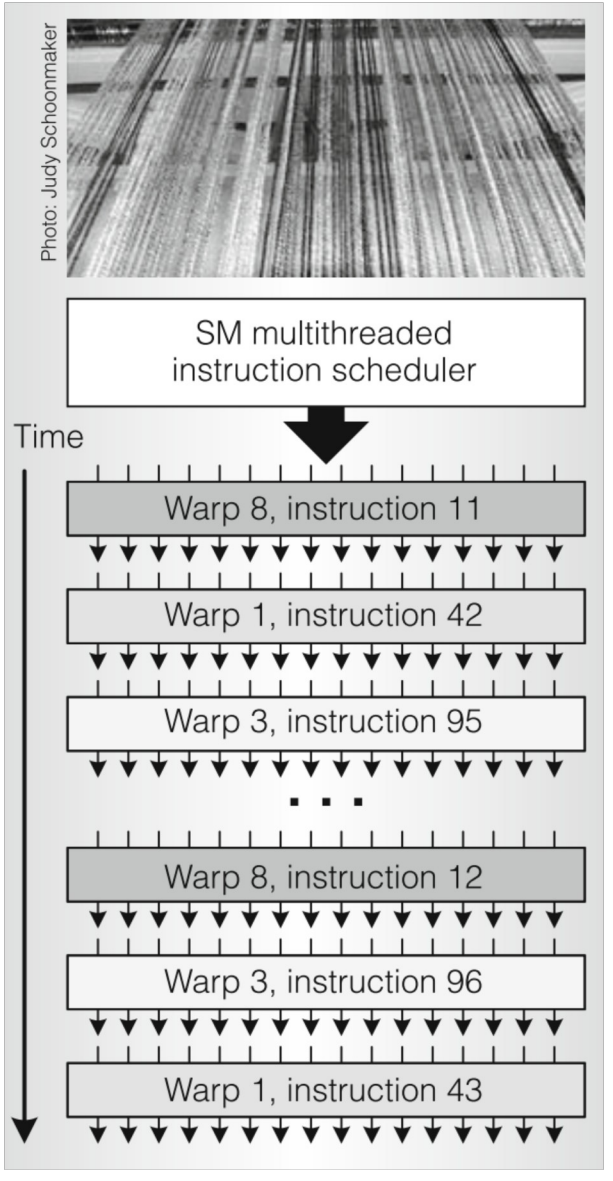
scheduler

Warp 14, Instruction 42



1. Uses a priority scheme based on instruction type, and "fairness" to pick a **ready** warp
2. Multiple processor clock cycles are used to execute the instruction on the 32 threads of the warp using the SPs and SFUs.





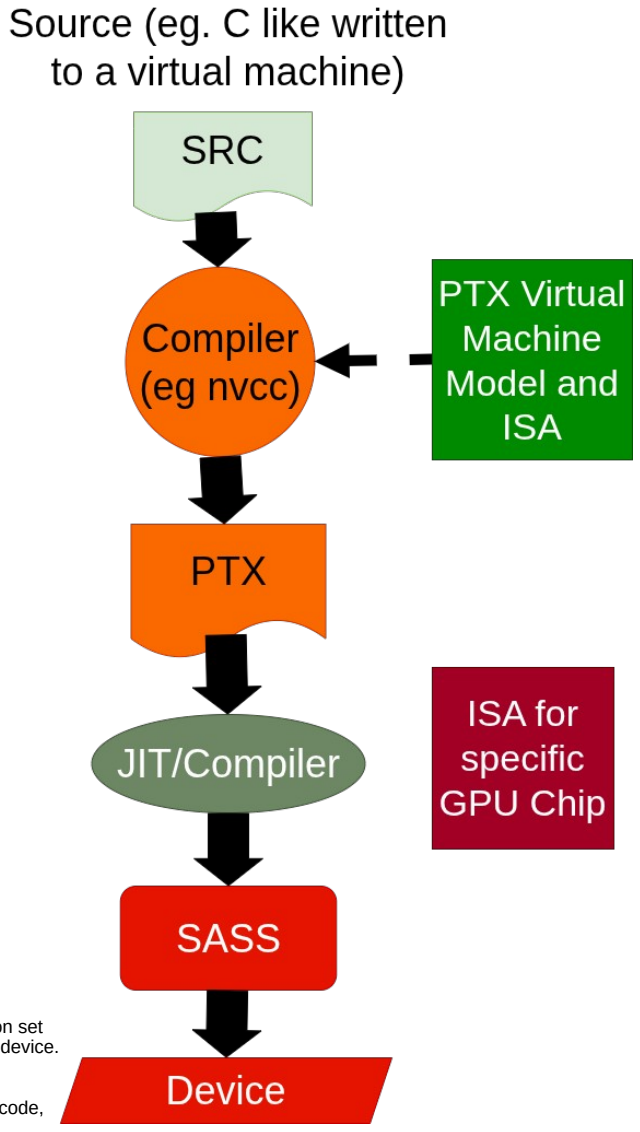
As a unified graphics processor, the SM schedules and executes multiple warp types concurrently—for example, concurrently executing vertex and pixel warps. The SM warp scheduler operates at half the 1.5-GHz processor clock rate. At each cycle, it selects one of the 24 warps to execute a SIMT warp instruction, as Figure 4 shows. An issued warp instruction executes as two sets of 16 threads over four processor cycles. The SP cores and SFU units execute instructions independently, and by issuing instructions between them on alternate cycles, the scheduler can keep both fully occupied.

Implementing zero-overhead warp scheduling for a dynamic mix of different warp programs and program types was a challenging design problem. A scoreboard qualifies each warp for issue each cycle. The instruction scheduler prioritizes all ready warps and selects the one with highest priority for issue. Prioritization considers warp type, instruction type, and "fairness" to all warps executing in the SM.

Instruction Set Architecture

Inherits JVM like model from graphics lineage

- Scaler Instructions*
- SRC -> PTX (Intermediate Representation [IR])
- Optimized PTX and then translate to Device Binary (SASS)
- PTX Stable and compatible across generations
- PTX virtual registers Optimizer
  - analyzes dependencies and maps to real registers
  - eliminate dead code
  - folds instruction together when feasible
  - optimizes thread divergence and convergence



PTX, a low-level parallel thread execution virtual machine and instruction set architecture (ISA). PTX exposes the GPU as a data-parallel computing device.

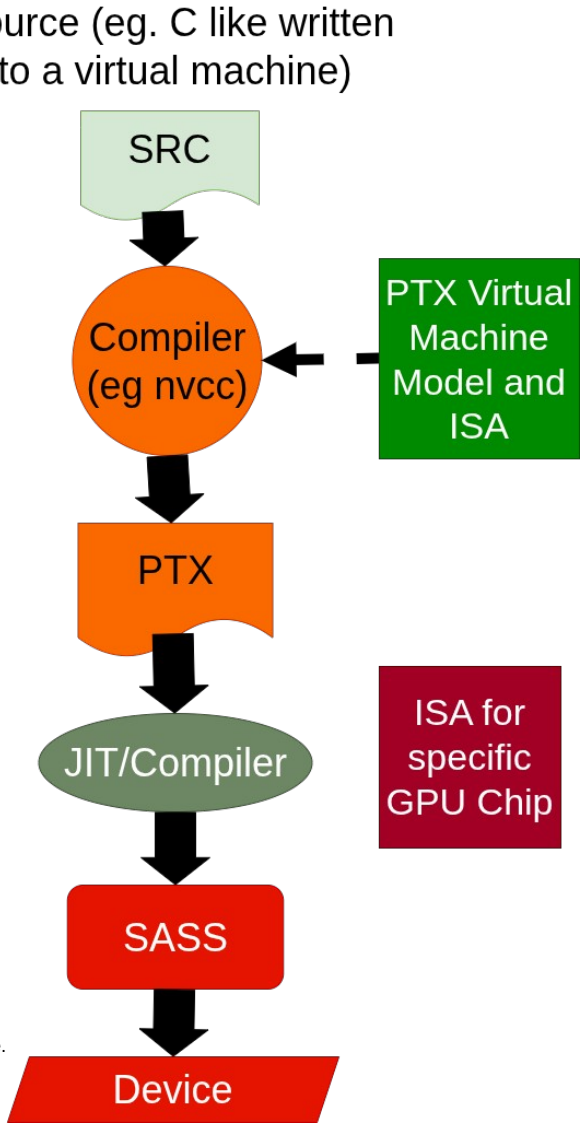
SASS is the low-level assembly language that compiles to binary microcode, which executes natively on NVIDIA GPU hardware

https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-reference

Instruction Set Architecture

Inherits JVM like model from graphics lineage

- Device SM ISA (SASS):
  - register based
  - includes:
    - floating point
    - integer
    - bit
    - conversions
    - transcendental
    - flow control
    - memory load/store
    - texture (vector like)



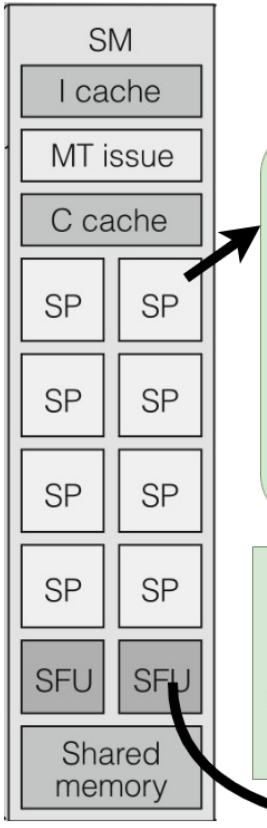
PTX, a low-level parallel thread execution virtual machine and instruction set architecture (ISA). PTX exposes the GPU as a data-parallel computing device.

SASS is the low-level assembly language that compiles to binary microcode, which executes natively on NVIDIA GPU hardware

https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-reference

Instruction Set Architecture

Inherits JVM like model from graphics lineage



Floating Point and Integer

- Add
- Multiply
- Multiply-Add
- Min and Max
- Compare
- Set predicate
- convert between integer and floating point
- FP negation and absolute value

Integer and FP instructions can set per thread status flags for zero, negative, carry, overflow, which threads can use for conditional branching

Transcendental

- Sine
- Cosine
- Binary exponential
- Binary logarithm
- Reciprocal
- Reciprocal square root

Bit

- Left shift
- Right shift
- Logical ops
- Move

Control Flow

- Branch
- Call
- Return
- Trap
- Barrier Syncrronization

Memory

- Texture memory
  - fetch
  - filter
  - write pixe frags to mem
- Load
- Store
  - integer addresses
  - register + offset

MEMORY TYPES

1. Local Memory: perthrea, private temporary data (external DRAM)
2. Shared Memory: low-latency access data shared by cooperating threads on SM
3. Global Memory: data shared by all threads of application (DRAM)

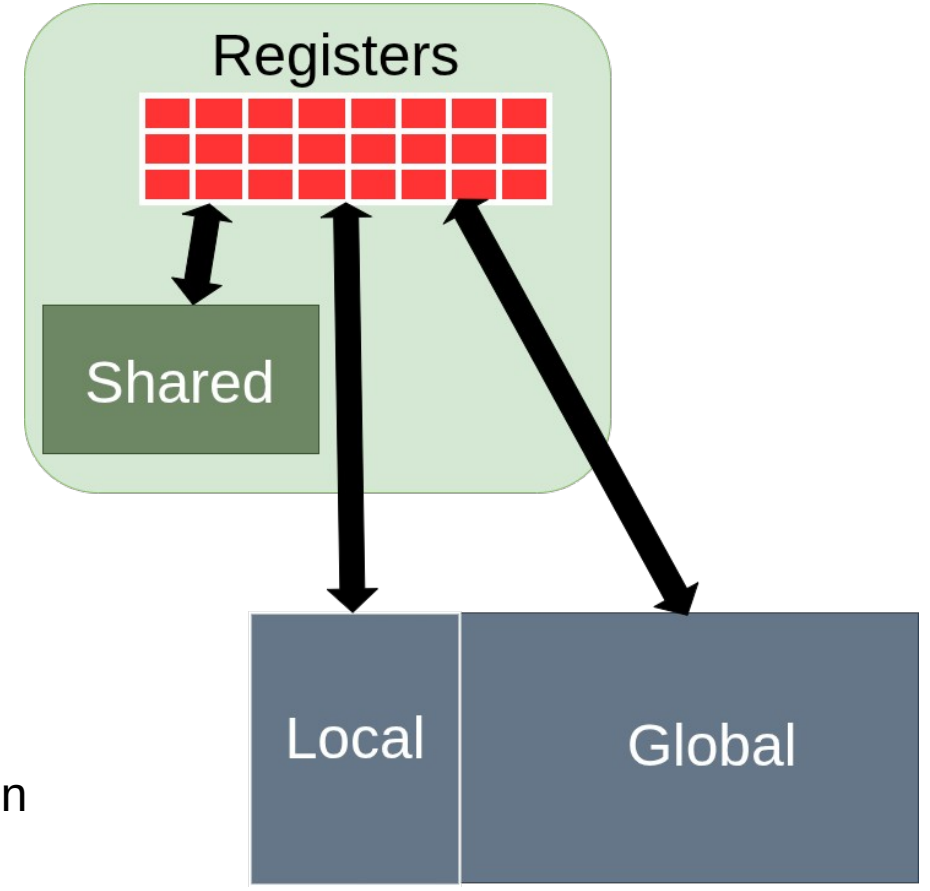
SP execution is fully pipelined

Memory

1) Local Memory (DRAM)
   - load-local, store-local
2) Shared
   - load-shared, store-shared
3) Global
   - load-global, store-global

Programmer moves data explicitly between memory types -- Various implications and constraints

NOTE: shared access needs synchronization for correctness



We will discuss this more once we look at Volta

Now we have a foundation: ~10 Years later Volta

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

First chip to add hardware to explicitly accelerate AI driven tensor operations

Generally each generation scales up SMs and adds some new features (caches, ray-tracing, atomic ops, tensor cores, etc). See extra slides for summary of Fermi, Kepler, Maxwell and Pascal

Volta

1) While lots of new features have been added in 10 years, the main compute model has not changed
2) Caches Hierarchy
   1) Added a 6 Meg L2 cache to front end all DRAM accesses (Global and Local memory) -- Synchronization (Fermi)
   2) Programmer can configure some fraction of shared memory to act as an L1 (Fermi)
3) Four Warp dual issue SM (Kepler)
4) Added a new vector unit designed for ML matrix processing -- Tensor cores
5) More external I/O (Nvlink)
6) Unified on chip address space for Global, Shared and local memory (Fermi)
7) Unified Memory -- virtual memory that works with Host (GPU page tables)
8) Eliminate external DRAM and use on-die HBM2 -- higher global and local memory bandwidth
9) Independent thread scheduling -- allows more flexible divergent code
10) Concurrent application functions (kernels)
11) Various forms of support for multiprogramming and multi-tenancy

Volta

Each Tensor Core operates on a 4x4 matrix and performs the following operation:

$D = A \times B + C$

where **A**, **B**, **C**, and **D** are 4x4 matrices (Figure 8). The matrix multiply inputs **A** and **B** are FP16 matrices, while the accumulation matrices **C** and **D** may be FP16 or FP32 matrices (see Figure 8).

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

FP16 or FP32 — FP16 — FP16 — FP16 or FP32

Today: Hopper H100

2022 NVIDIA H100 Tensor core GPU Architecture
(https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c)

AI/ML is firmly present

Hopper

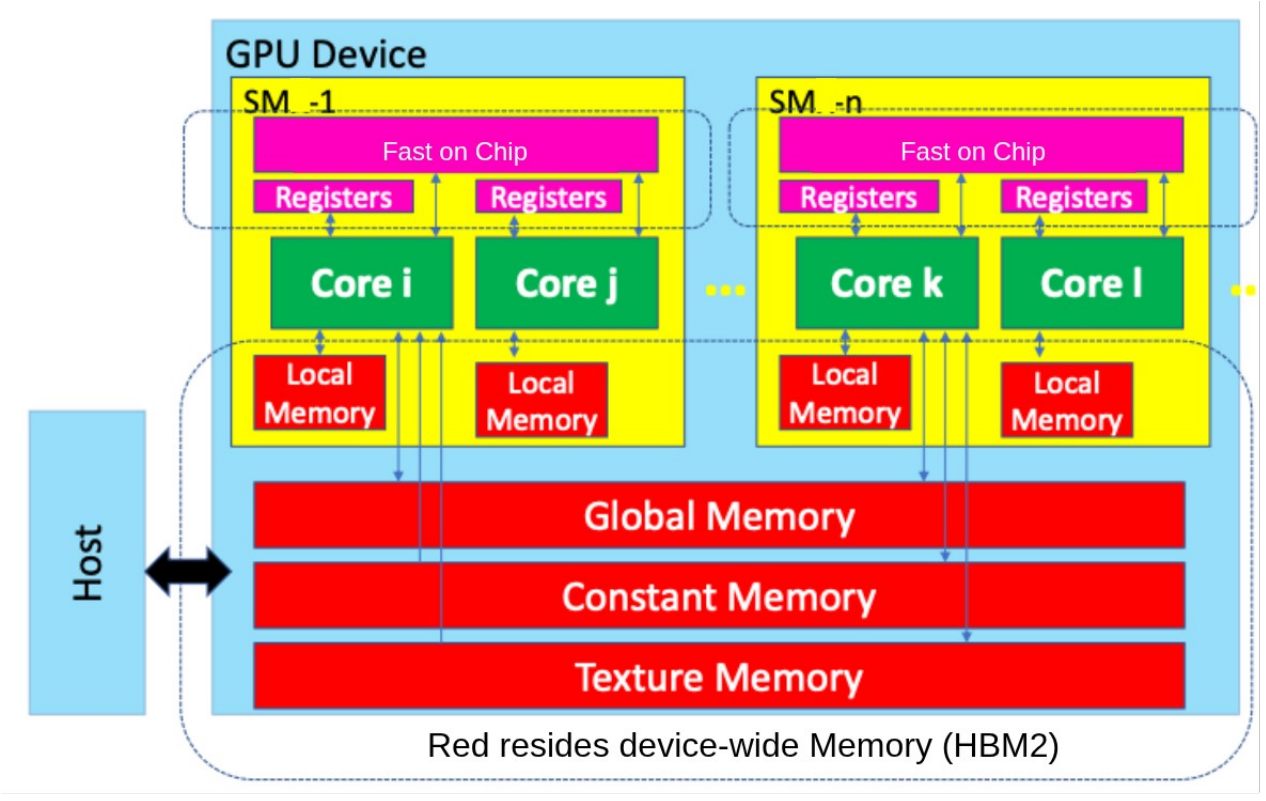
18432 CUDA cores (GH100) + 576 Tensor Units

1) Updated Tensor Cores and more of them
2) Sparse Matrix support (Ampere)
3) Super groupings Thread Block Cluster
4) Distributed Shared Memory
5) New instruction (DPX) for accelerating dynamic programming code
6) Async memory move
7) 50 MB L2
8) Transform Engine
9) Security and Isolation features (Trusted Execution Environment):
    1) On-Die Root of Trust
    2) Device Attestation
    3) AES-GCM 256 (Data transfers between CPU and H100 are encrypted and decrypted)

Let's take a closer look at accessing memory

Much of our optimization / programming effort is going to have to do with memory accesses.

Programmer’s view & sizes



SMs	80
cores/SM	64
resident warps/SM	64
threads/warp	32
resident threads/SM	2024
32 bit registers/SM	**65K**
Fast on Chip/SM	**128KB**
Memory	**6GB**

6 MB of L2 (Transparent to the Programmer)

Memory Access Latencies

Memory	Latency (cycles)	Bandwidth (GB/s)
registers	~5-10	
L1/Shared	~28	~120/~160
L2	~193	~2,048
Global/Local HBM2	~384-500	~900 (peek,ECC off)

SMs	80
cores/SM	64
resident warps/SM	64
threads/warp	32
resident threads/SM	2024
32 bit registers/SM	**65K**
Fast on Chip/SM	**128KB**
Memory	**6GB**

Approximate values for Volta (compiled with GPT4.0)

Memory Access Behaviors

Some gotchas

- Remember, the best behavior is when a warp of threads (32) executes the same instruction:
  - Eg. All do a Load or Store
- This will imply 32 memory accesses -- but you control what they access (eg addresses)
- Good behavior only if they play nicely with memory
  - Global Memory: Transfers in aligned units of a particular size (eg like a cache line). Units are: 32-, 64-, and 128-bytes
    - HW Coalesces a warp's memory accesses into one or more aligned unit transfers as needed
    - Avoid multiple by being careful
  - Shared Memory: Organized in banks
    - Accesses to the same bank are serialized
    - Avoid conflicts: ensure a warps accesses are spread across banks

See CUDA 13.0 Programming Manual Section 8.3.2 "Device Memory Accesses" for details. We will be revisiting this

Memory Access Behaviors

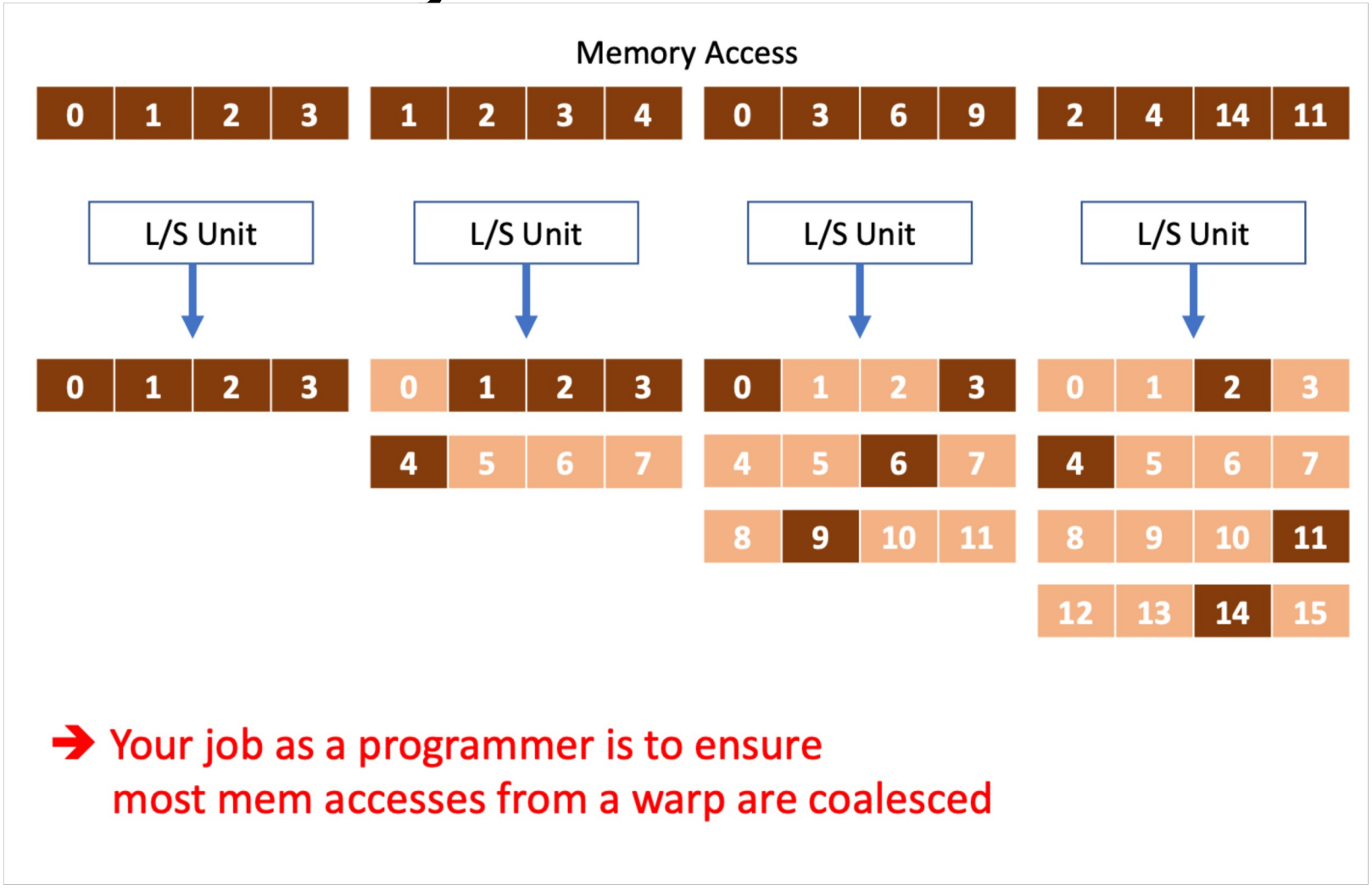
Some gotchas

- Remember, the best behavior is when a warp of threads (32) executes the same instruction:
  - Eg. All do a Load or Store
- This will imply 3 access (eg addresses)
- Good behavior
  - Global Memo eg like a cache line). Units ar
    - HW Coales aligned unit transfers as
    - Avoid multiple by being careful
  - Shared Memory: Organized in banks
    - Accesses to the same bank are serialized
    - Avoid conflicts: ensure a warps accesses are spread across banks

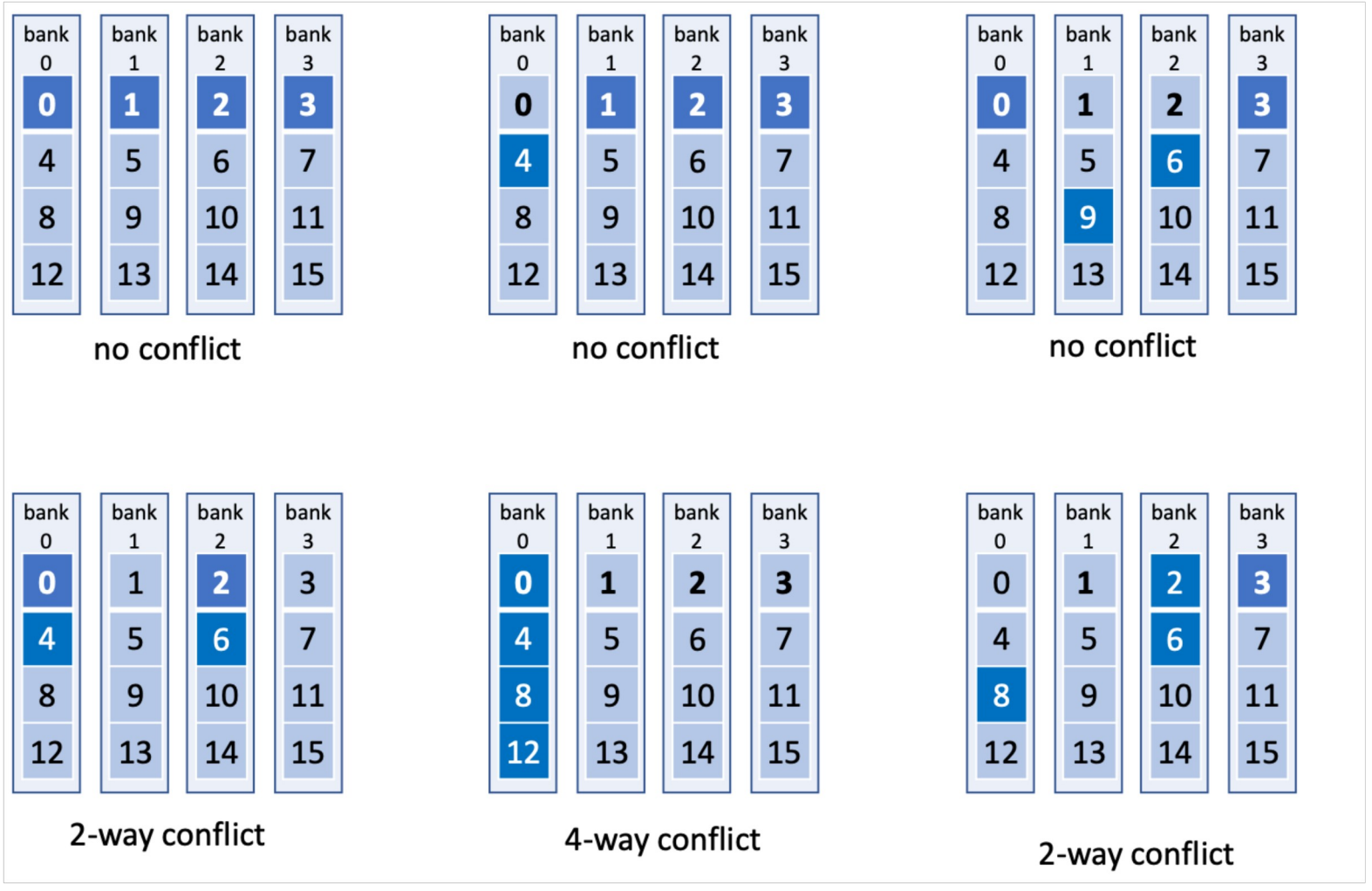
Unfortunately the exact parameters and behavior that determine optimal memory accesses are device specific!

See CUDA 13.0 Programming Manual Section 8.3.2 "Device Memory Accesses" for details. We will be revisiting this

Global Memory Coalesced Access Example



Local Memory Bank Conflict Example



General note about architecture documentation

1) Hardware Implementation of CUDA Programmers Guide
   - https://docs.nvidia.com/cuda/cuda-c-programming-guide/#hardware-implementation
2) Hardware section of profiling guide:
   - https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#hardware-model
3) PTX Machine Model:
   - https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#ptx-machine-model
4) Nvidia white papers for each architecture and associated tuning guides

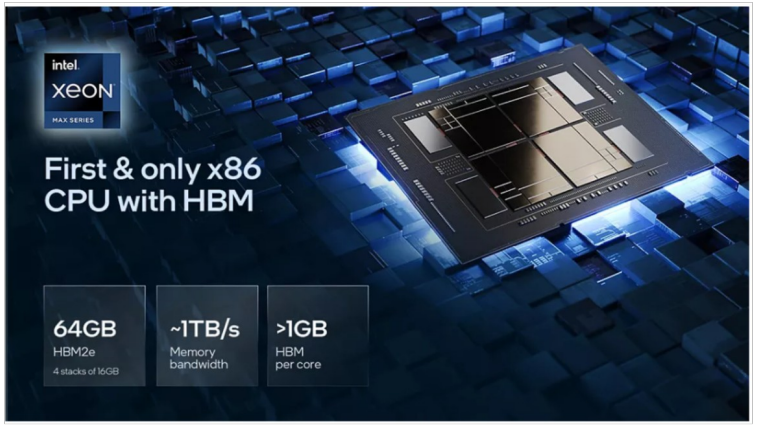
Extra Stuff

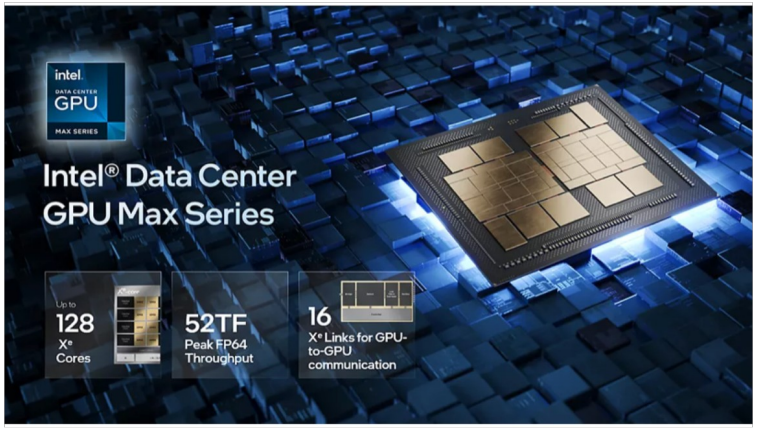
Intel’s Take: Gone Heterogeneous

Max Series CPUs and GPUs

- MAX CPU
  - More Cores
  - More integration of memory and accelerators
- MAX GPU
  - 128 Gb HBM + Larger Caches
  - Xe Matrix acceleration -- Systolic Array
- The big story -- Claims Open SW
  - OneAPI
  - A "seamless" single code base model

https://www.intel.com/content/www/us/en/products/details/processors/xeon/max-series.html



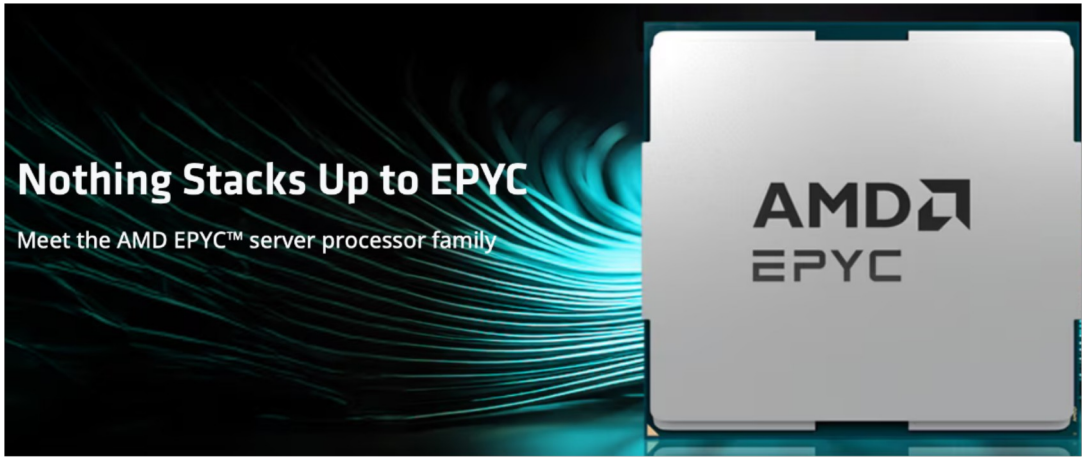


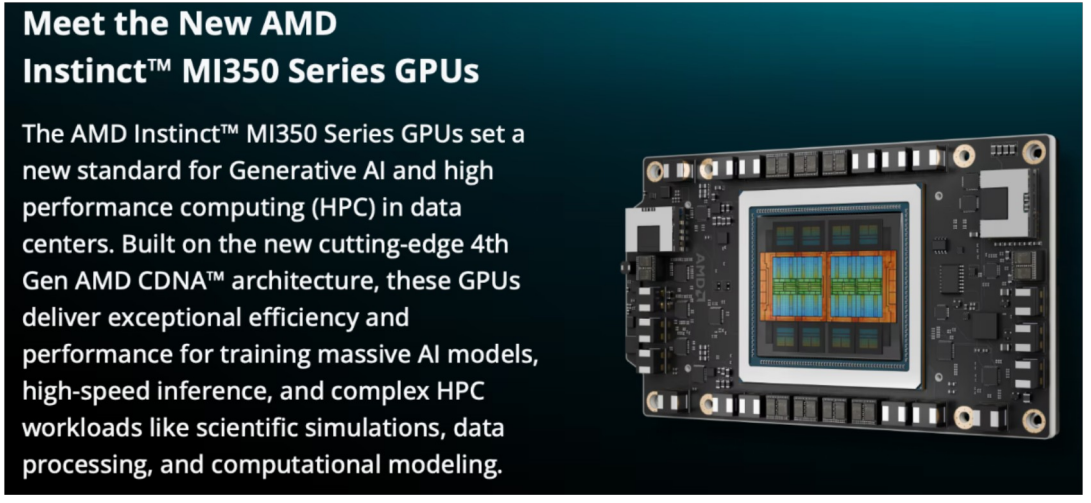
https://www.intel.com/content/www/us/en/products/details/discrete-gpus/data-center-gpu/max-series.html

https://www.intel.com/content/www/us/en/products/docs/processors/max-series/overview.html

AMD: Open

https://www.amd.com/en/products/processors/server/epyc.html





https://www.amd.com/en/products/accelerators/instinct.html

GPT 4.1 Produced GPU Feature Summary

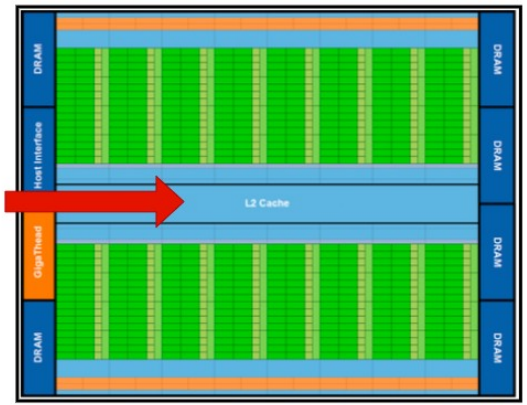
I think there are some bugs in this... but roughly correct

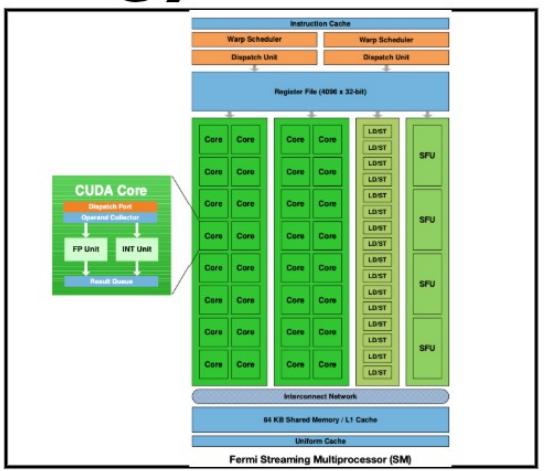
Major NVIDIA Tesla GPU Generations, Years, and Key Architectural Features

Generation	Year	Major Features
Tesla (G80/G92, GT200)	2006–2008	**Unified shader** architecture, **scalar processors**, initial **CUDA** support, **double precision** (GT200, CC 1.3), shared memory introduced (GT200), thread context switching
Fermi (GF100)	2010	**SM architecture** (Streaming Multiprocessor), **L1/L2 caches**, **ECC** memory, **concurrent kernel execution**, **atomic operations**, improved double precision (CC 2.x)
Kepler (GK10x/GK110)	2012–2013	**Greater efficiency** (more CUDA cores per SM), **dynamic parallelism**, **Hyper-Q** (better multi-process support), **GPU Boost**, improved power
Maxwell (GM10x/GM20x)	2014–2015	**Major energy efficiency**, **2nd-gen unified memory**, **improved shared memory**, **NVENC/NVDEC** hardware video codecs
Pascal (GP100/GP104)	2016	**NVLink v1**, **unified memory with page migration**, **FP16 support**, larger register file, higher clock speeds, improved scheduling, **HBM2** memory support
Volta (GV100)	2017	**Tensor Cores** (FP16 for deep learning), **NVLink v2**, **independent thread scheduling** & preemption, **fused multiply-add** for FP32/FP64, **larger shared memory**
Turing (TU102/TU104)	2018	**Advanced Tensor Cores** (FP16/INT8/INT4), **RT Cores** (ray tracing, mostly in consumer, select Tesla/Quadro models), concurrent integer/floating-point operations
Ampere (GA100)	2020	**Third-gen Tensor Cores** (TF32, BF16), **Multi-Instance GPU (MIG)**, **PCIe Gen4**, **NVLink v3**, **Sparsity acceleration**, higher energy efficiency, increased memory bandwidth
Hopper (GH100)	2022	**Transformer Engine**, **FP8 math**, **4th-gen Tensor Cores**, **NVLink v4**, **DPX instructions** (accelerating dynamic programming), expanded MIG, **larger L2**
Blackwell (GB200)	2024	**5th-gen Tensor Cores** (even higher performance, FP4/FP6 precision), **dual-GPU architecture (GB200)**, **high-bandwidth memory (HBM3e)**, **NVLink Switch System** for massive multi-GPU scaling, **advanced sparsity support**, **up to 208B transistors**, even larger memory and cache, networking enhancements for AI supercomputers

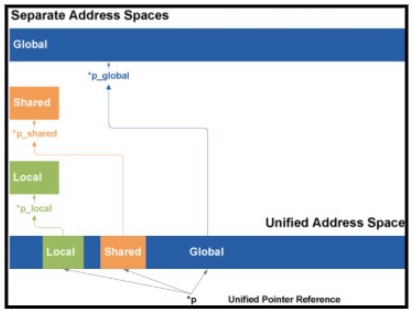
Fermi: main architectural changes (beyond scaling)

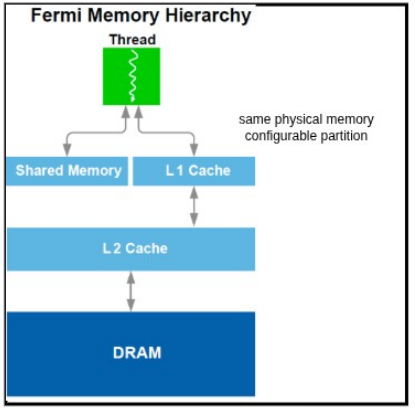
512 CUDA cores

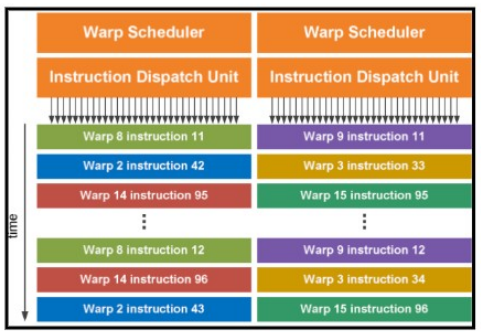


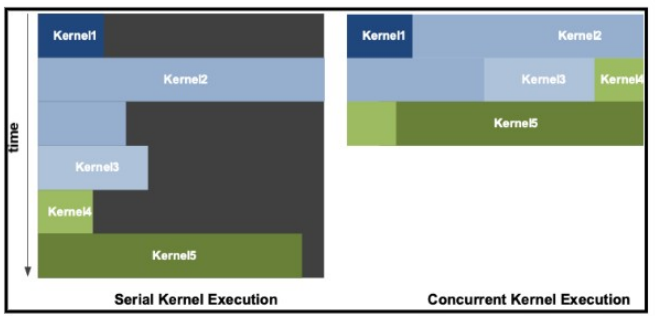


Fermi Streaming Multiprocessor (SM)





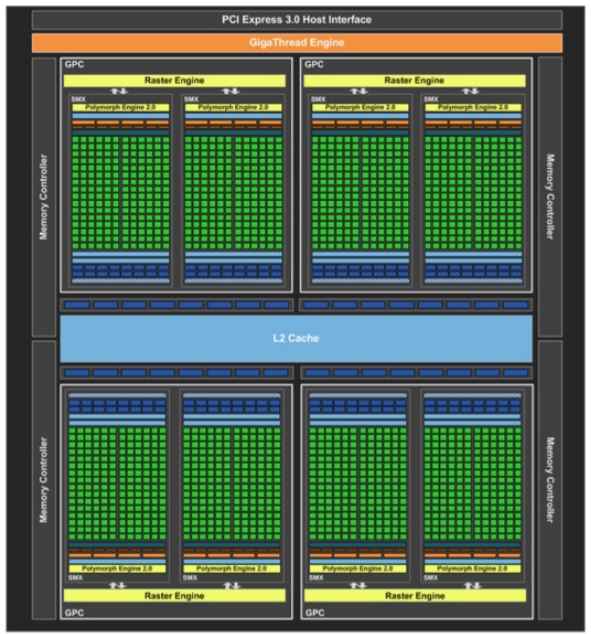




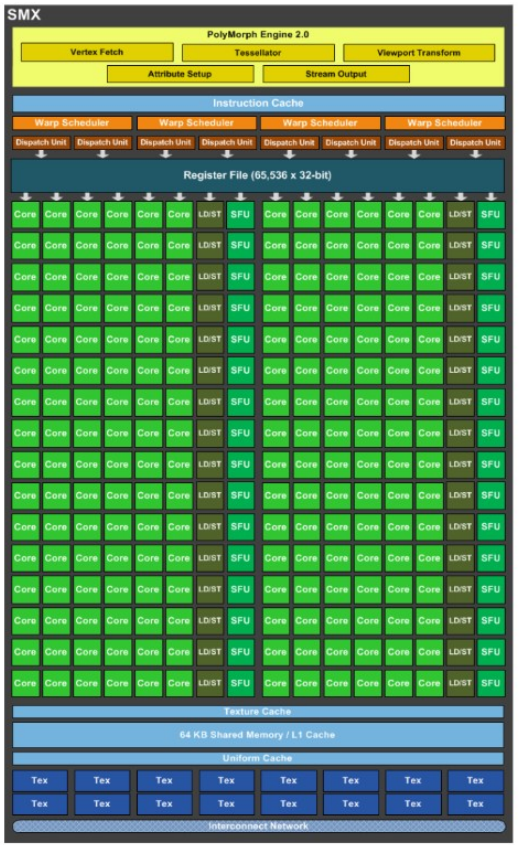
1. "True Cache hierarchy" to ease use of Shared memory
    1. Configurable L1 and an L2
2. Bigger shared memory
3. SM has a dual warp scheduler -- simultaneously schedule and dispatch instructions from two warps
4. Introduced Parallel Thread eXecution (PTX) VM and ISA
5. Introduced a unified address spaced for local, shared and global memory
    1. adds unified load and store instructions
    2. retains distinct load and store instructions
6. Concurrent Kernel Execution

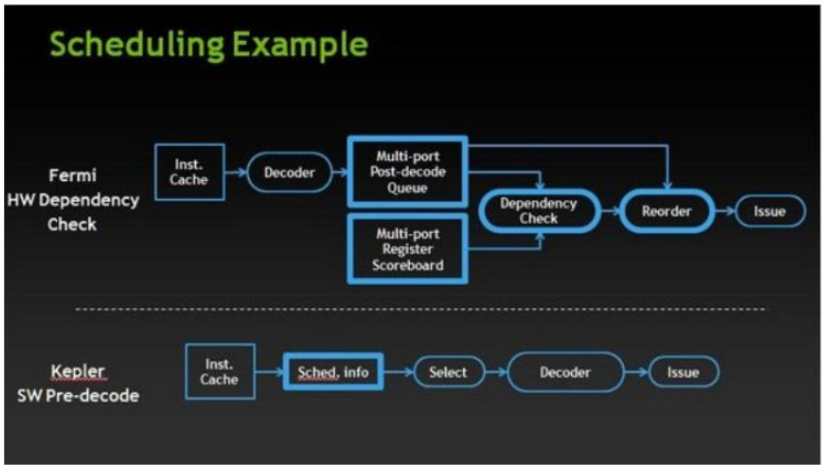
Kepler: main architectural changes (beyond scaling)

1536 CUDA cores (GK104)



2880 CUDA cores (GK110) 15 SMs

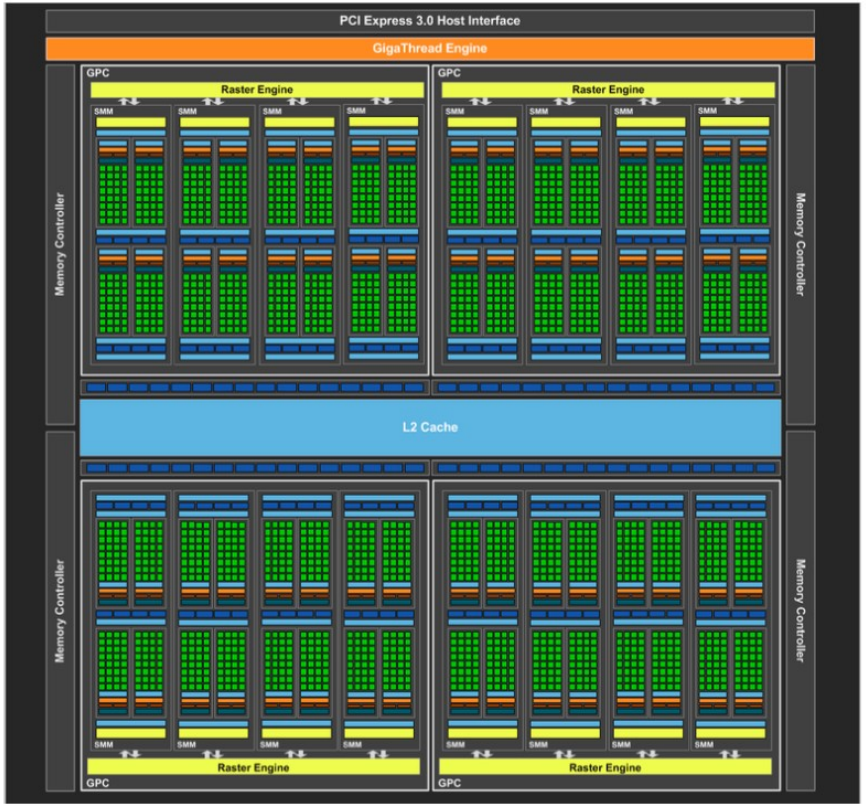




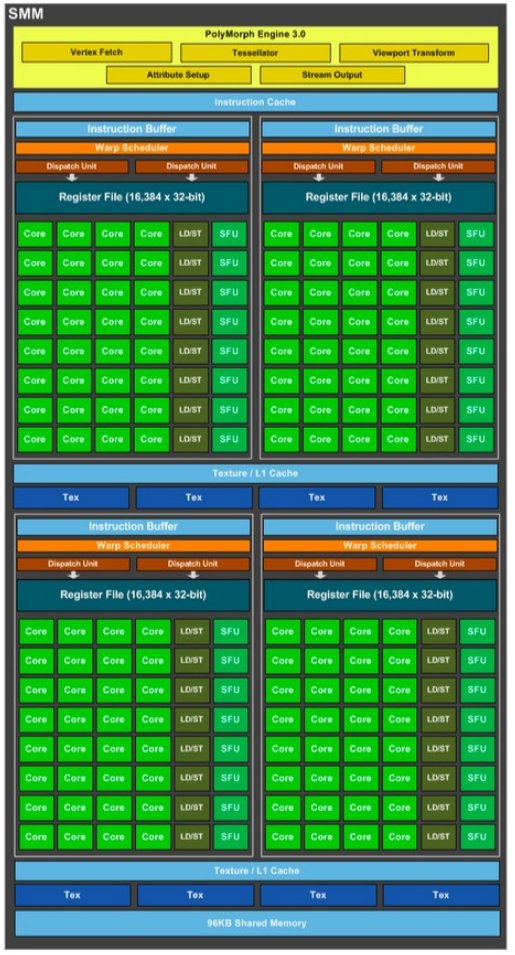
1. Not much
    1. 4 warp dual issue warp schedulers
        1. 8 instructions potentially scheduled per clock
        2. Dual issue means that two instructions (a la superscaler) can be issued if possible
    2. Remove logic that requires decode of instructions to determine dependencies of instructions (assumedly interacts with dual issue)
        1. given fixed math latencies let compiler annotate instructions with scheduling information. COMPILER dependency has crept in to get good performance!
        2. Use compiler info to determine if a warp instruction (thus warp) is eligible for dispatch
2. Documents lists HW scheduling units as:
    1. register score-boarding for long latency operations (texture and load)
    2. inter-warp scheduling decisions (eg pick the best warp to go next among eligible candidates
    3. thread block level scheduling (GigaThread engine) [decide feasibility of blocks of threads and assign them to an SM
3. Added RDMA (GPUDirect)

Maxwell: main architectural changes (beyond scaling)

2048 CUDA cores (GM204)



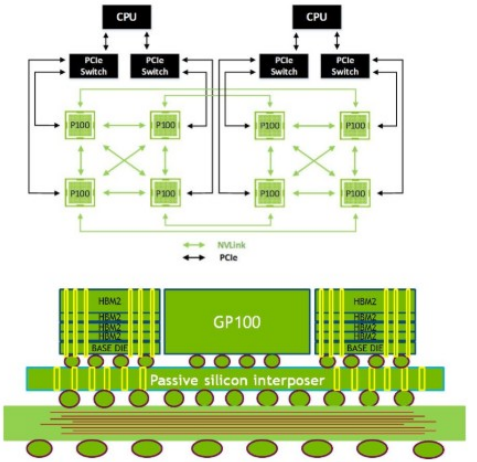
3072 CUDA cores (GK110) 24 SMs



1. Again not much --
    1. scaled, reconfigured and optimized
    2. L1 and Shared memory have been split
        1. L1 moved to be part of texture memory
    3. Larger dedicated Shared memory

Pascal: main architectural changes (beyond scaling)

3584 CUDA cores (GP100)



CPU Code

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

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

Pascal Unified Memory*

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

  fread(data, 1, N, fp);

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

  use_data(data);

  free(data);
}
```

*with operating system support

Figure 22. With Operating System Support, Pascal is Capable of Supporting Unified Memory with the Default System Allocator.
(Here, *malloc* is all that is needed to allocate memory accessible from any CPU or GPU in the system.)

1. Rebalance: reduce the number for cores per SM by half (but increase number of SMs --> 24 - 56)
2. back to two warp schedulers per SM
3. AI/ML is a focus
    1. Support reduced floating point precision (thus higher throughput vector ops on lower precisions types I assume)
    2. GPU-to-GPU Interconnect (avoid PCIe) NVLink
    3. HBM2 Memory colocated with GPU on same package (stacked memory)
        1. Higher bandwidth to global and local memory
4. Unified memory -- Ability to have a shared virtual address space with host -- page tables translation
    1. History
        1. Unified Virtual Address (UVA) 2011 uses pinned memory and zero copy but still slow due to PCIe
        2. Unified Memory (CUDA 6) : Driver does page migration prior to kernel launch -- no concurrent access
    2. Pascal Unified Memory
        1. 49 bit GPU virtual address space support (can map entire host 48 bit address space)
        2. Add GPU page-tables to allow on-demand pagefaults on the GPU
            1. Pages automatically and on-demand migrated on fault
            2. NVlink can be used instead of PCIe (GPU-GPU only)
            3. Coherent across Host and GPU (assume independent R/W access)
            4. Needs OS support
5. Compute Premption: Application (I assume kernels launched by a single host process) can be prempted and contexted switched via GPU DRAM
    1. SM contexts of an App written to DRAM
    2. new app loaded into SMs
    3. then original loaded back
    4. Supports interacive use while having long runnning apps

Extra References

- Nice history of parallel computing https://parallel.ru/history/wilson_history.html
- Systolic Array an alternative to GPU approach "Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second", 1988, https://www.eecs.harvard.edu/~htk/publication/1988-icnn-pomerleau-gusciora-touretzky-kung.pdf
- NVidia architecture & white papers
- NVidia CUDA Programming Guide (v13.0) Chapter 7 "Hardware Implementation"
- NVidia CUDA Programming Guide (v13.1) Chapter 3.2.2 “Hardware Implementation”
- Fermi white paper : https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5751939