6.5930/1
Hardware Architectures for Deep Learning

GPU Computation

March 4, 2024

Joel Emer and Vivienne Sze
Acknowledgement: Srini Devadas (MIT)

Massachusetts Institute of Technology
Electrical Engineering & Computer Science

Why are GPU interesting?

- Very successful commodity accelerator/co-processor
- GPUs combine two strategies to increase efficiency
  - Massive parallelism
  - Specialization
- Illustrates tension between performance and programmability in accelerators
- Pervasively applied in deep learning applications

Background Reading

- **GPUs**
  - ***Computer Architecture: A Quantitative Approach*, by Hennessy and Patterson**
    - Edition 6: Ch 4: p310-336
    - Edition 5: Ch 4: p288-315

All these books and their online/e-book versions are available through MIT libraries.

GPUs were originally designed for 3D rendering

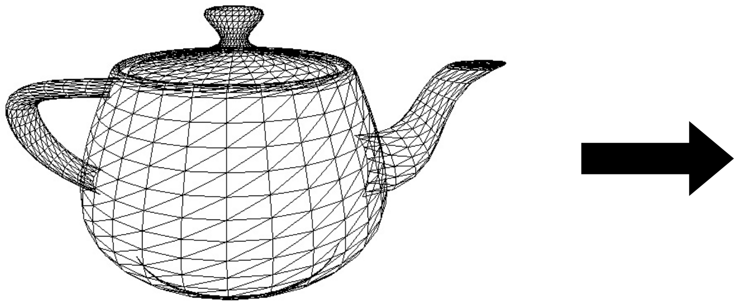


Image credit: Henrik Wann Jensen

Input: description of a scene:
3D surface geometry (e.g., triangle mesh)
surface materials, lights, camera, etc.

Output: image of the scene

Simple definition of rendering task: computing how each triangle in 3D mesh contributes to appearance of each pixel in the image?

Courtesy Kayvon Fatahalian

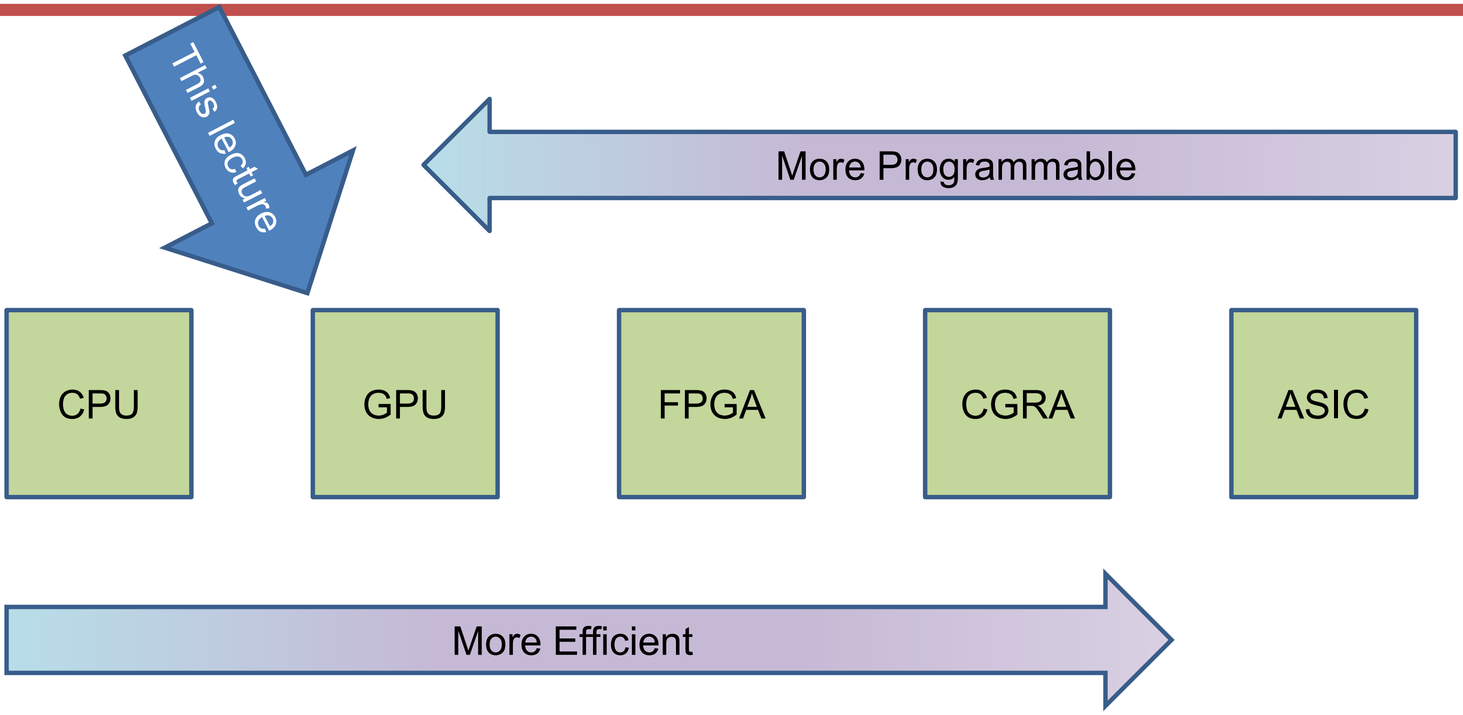
What GPUs were originally designed to do

Render high complexity 3D scenes, in real time

Graphics Processors Timeline

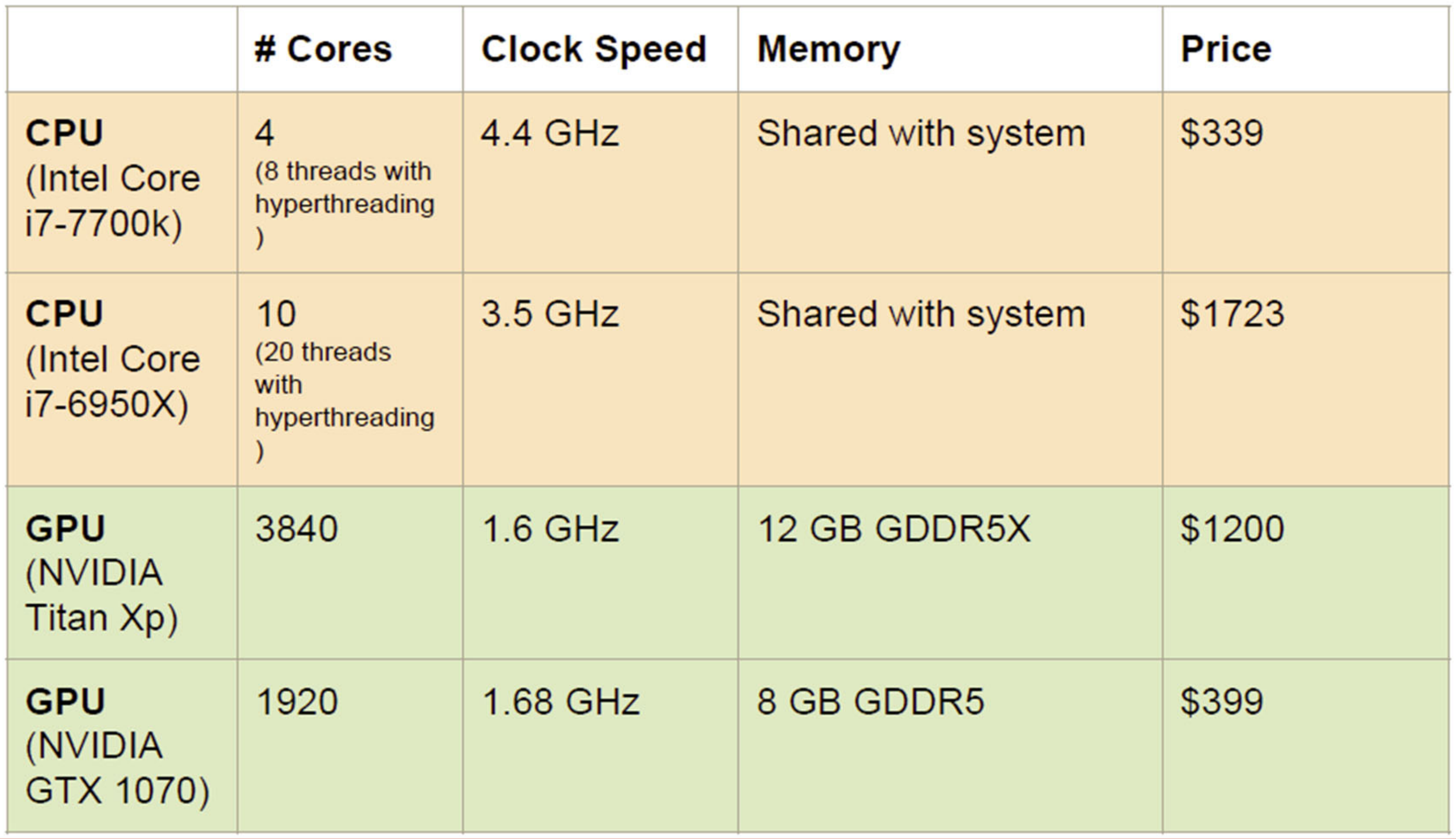
- Until mid-90s
  - Most graphics processing in CPU
  - VGA controllers used to accelerate some display functions
- Mid-90s to mid-2000s
  - Fixed-function accelerators for 2D and 3D graphics
    - triangle setup & rasterization,
    - texture mapping & shading
  - Programming:
    - OpenGL and DirectX APIs -> BrookGPU
- Modern GPUs
  - Some fixed-function hardware (texture, raster ops, ray tracing…)
  - Plus programmable data-parallel multiprocessors
  - Programming:
    - OpenGL/DirectX
    - Plus more general-purpose languages (CUDA, OpenCL, …)

Programmability vs Efficiency



FPGA	=> Field programmable gate array
CGRA	=> Coarse-grained reconfigurable array
ASIC	=> Application-specific integrated circuit

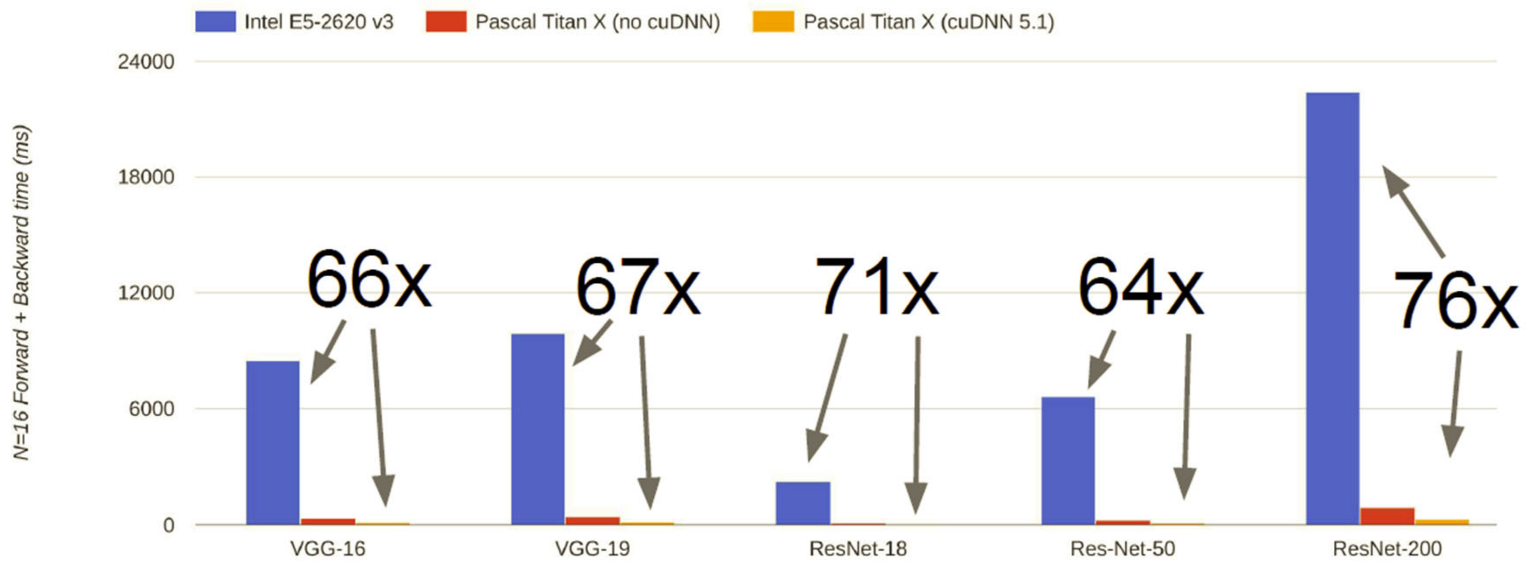
CPU vs GPU Attribute Summary



	# Cores	Clock Speed	Memory	Price
CPU (Intel Core i7-7700k)	4 (8 threads with hyperthreading)	4.4 GHz	Shared with system	$339
CPU (Intel Core i7-6950X)	10 (20 threads with hyperthreading)	3.5 GHz	Shared with system	$1723
GPU (NVIDIA Titan Xp)	3840	1.6 GHz	12 GB GDDR5X	$1200
GPU (NVIDIA GTX 1070)	1920	1.68 GHz	8 GB GDDR5	$399

Source: Stanford CS231n

CPU vs. GPU Performance

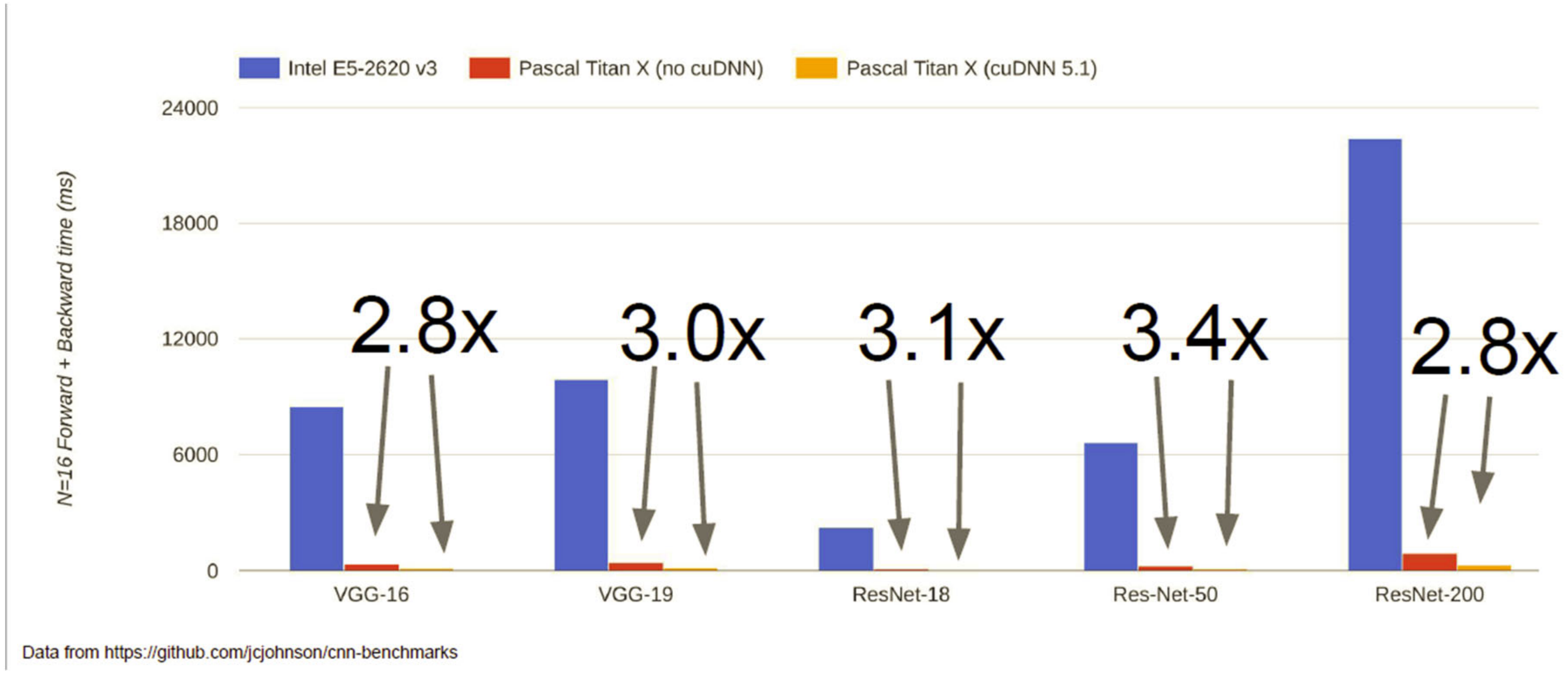


Data from https://github.com/jcjohnson/cnn-benchmarks

Ratio of (partially-optimized) CPU vs. CUDA library (cuDNN)

Source: Stanford CS231n

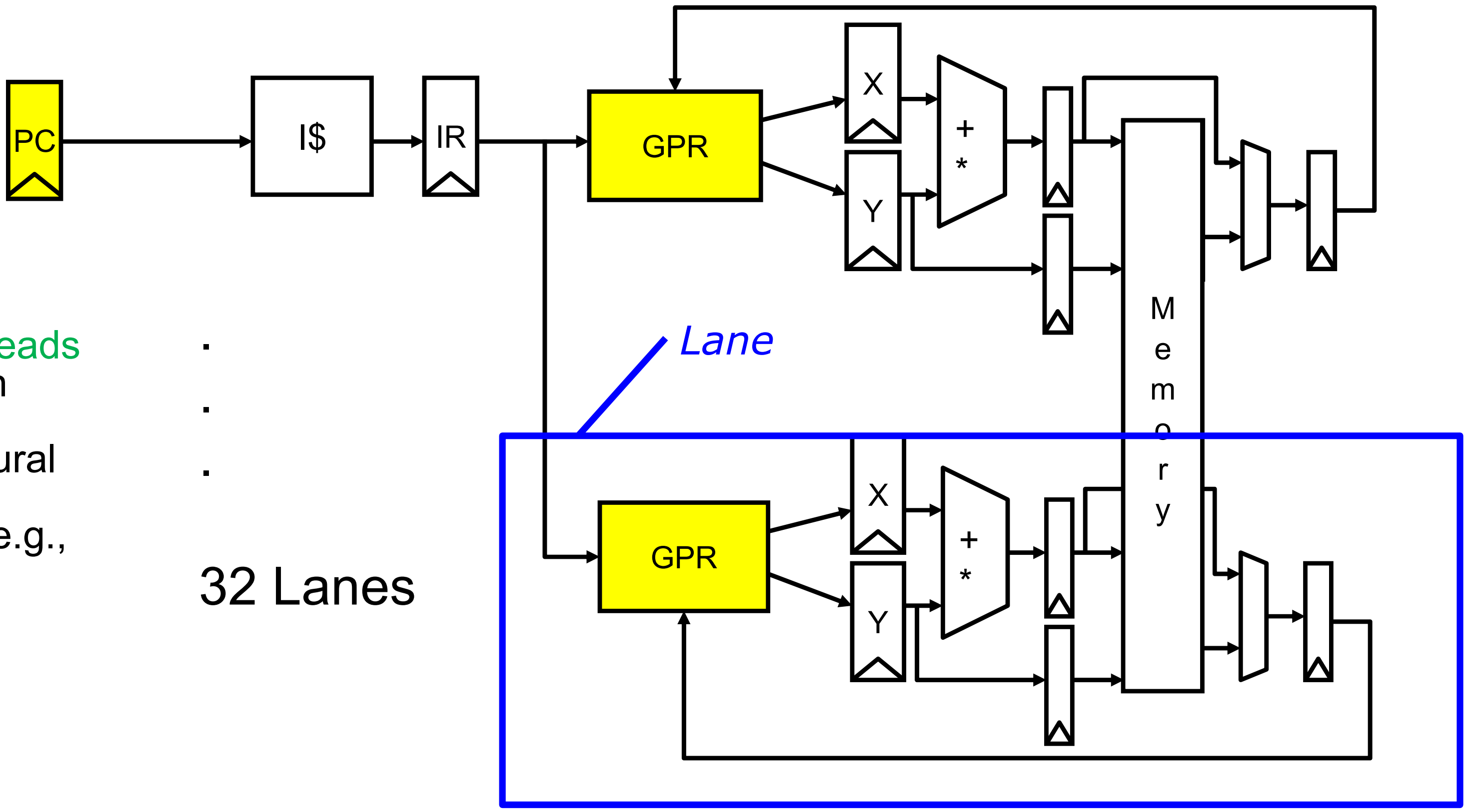
CPU vs. GPU Performance



Ratio of unoptimized CUDA vs. CUDA library (cuDNN)

Source: Stanford CS231n

Single Instruction Multiple Thread

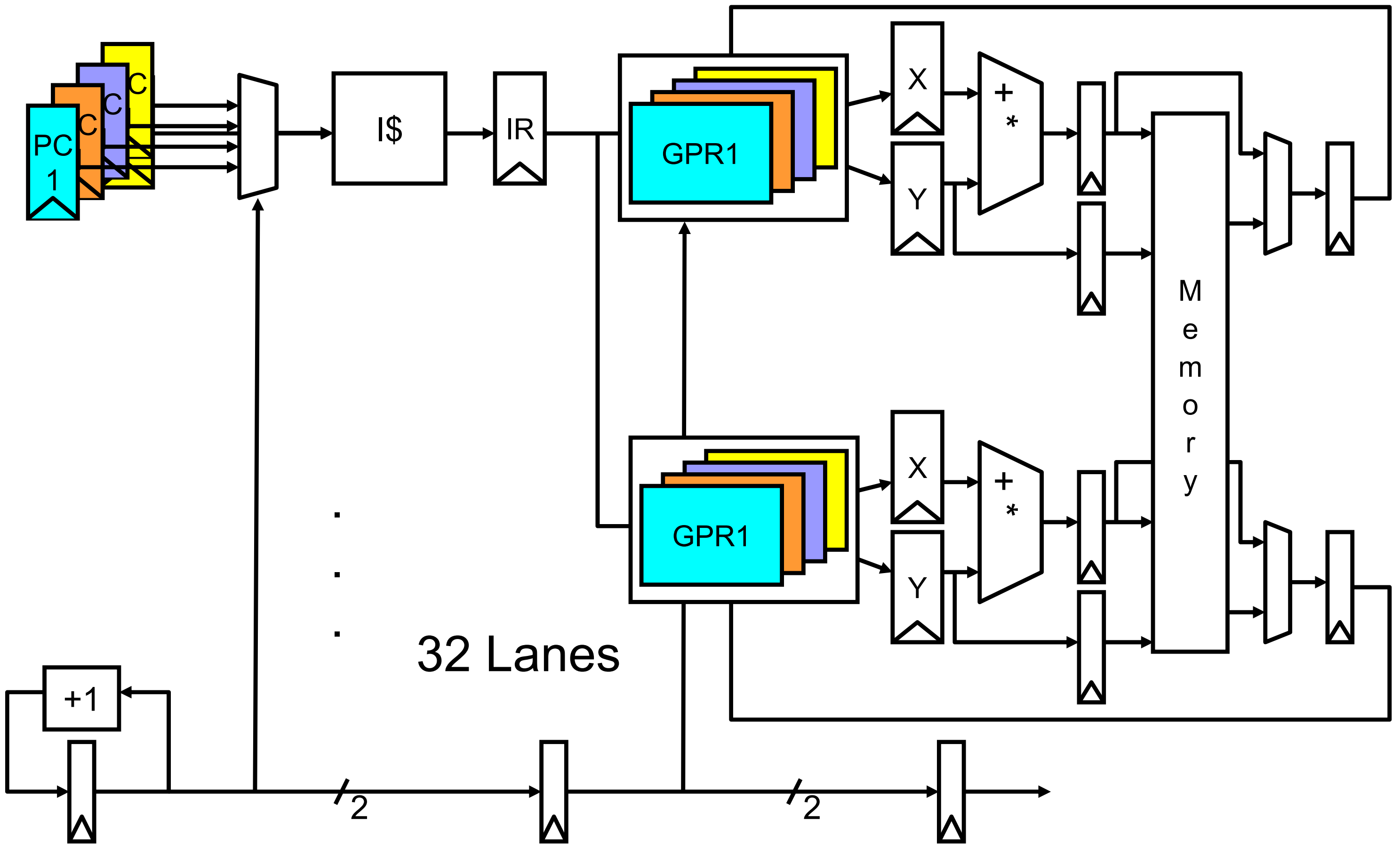


SIMT

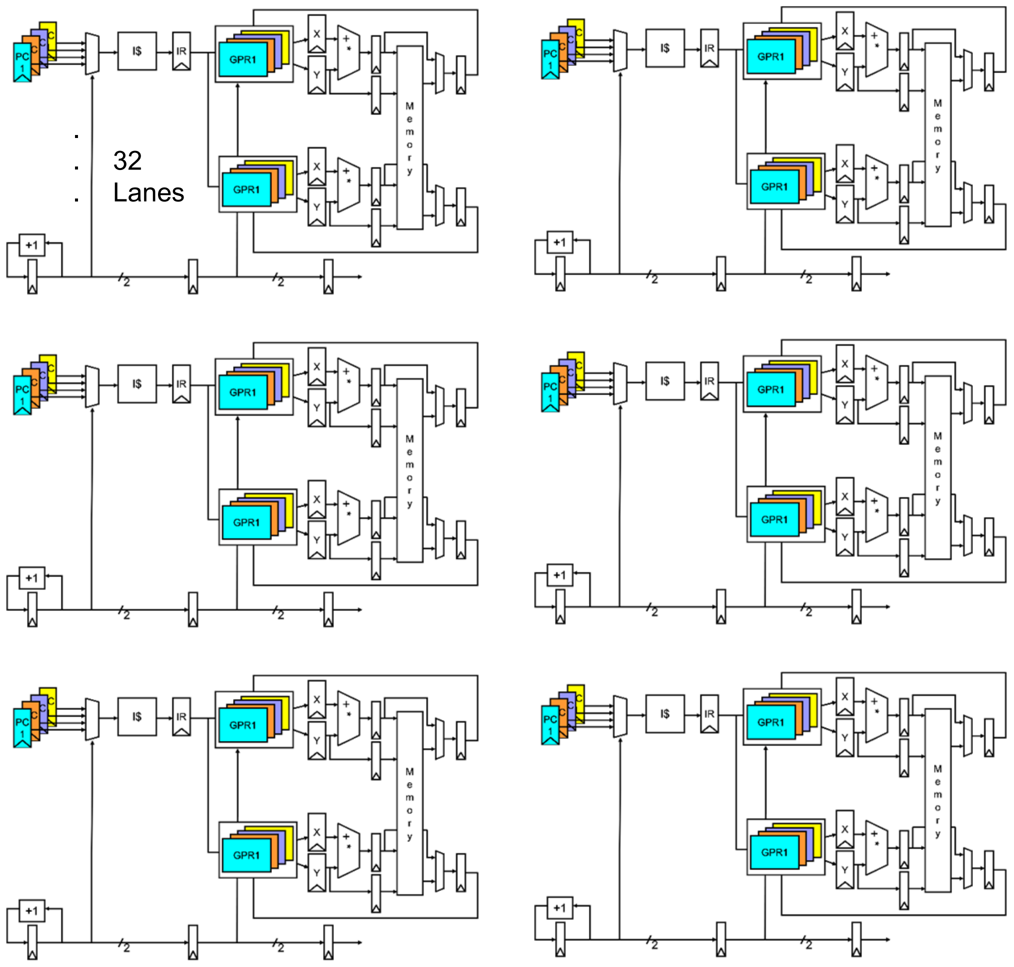
- Many threads each with private architectural state or context, e.g., registers.

green-> Nvidia terminology

Multiple Thread – Single Instruction Multiple Thread



GPU

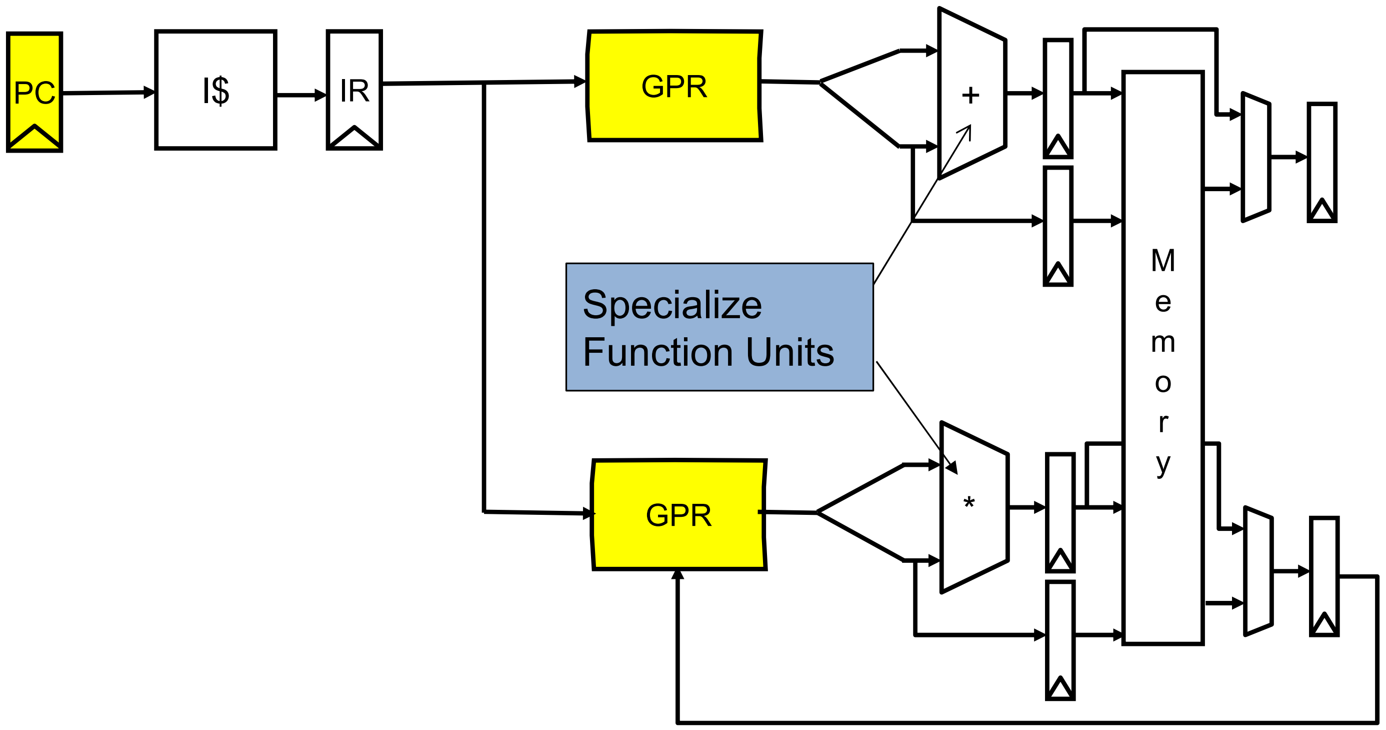


SIMT

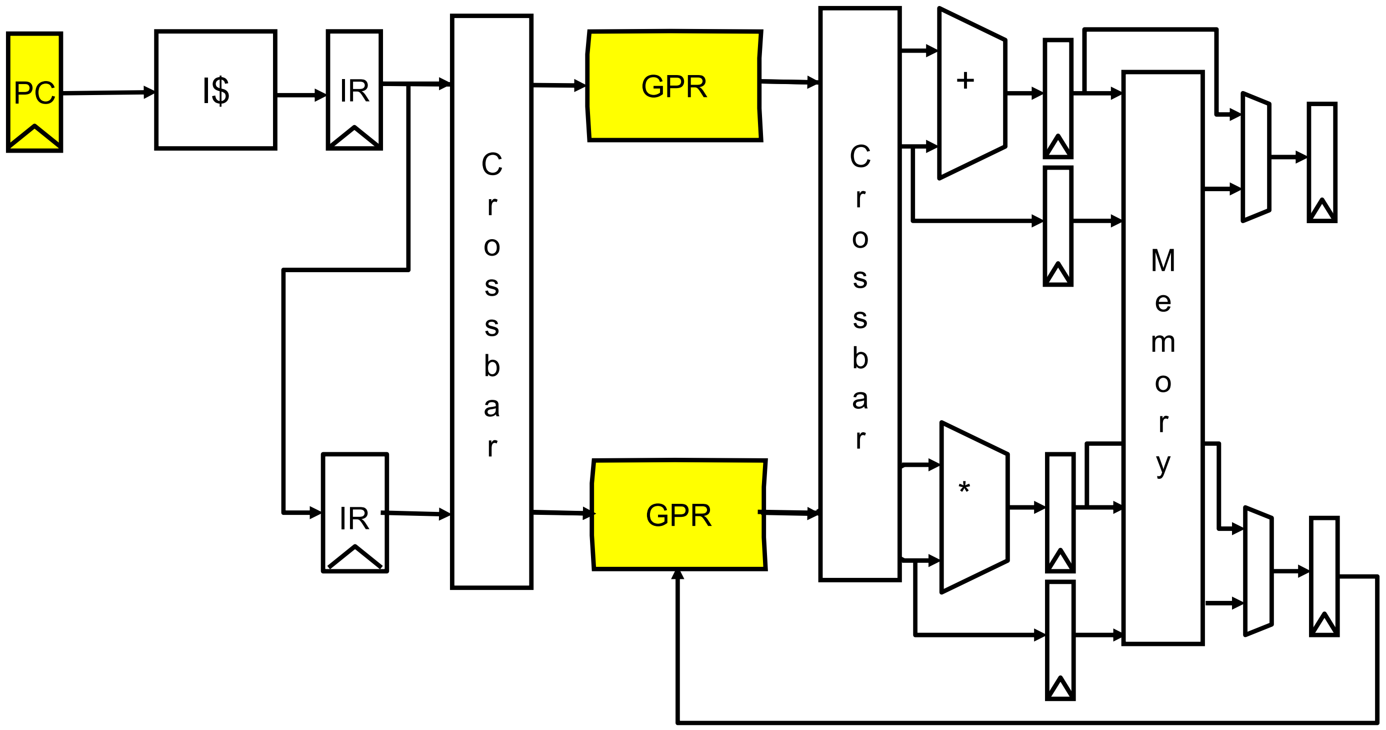
- Many threads, each with private architectural state, e.g., registers
- Group of threads that issue together (same color) called a warp (32)
- All threads that issue together execute same instruction
- Entire pipeline is an SM or streaming multiprocessor (32-48 warps)
- **Many (64-128) SMs in a GPU**

green-> Nvidia terminology

Function unit optimization

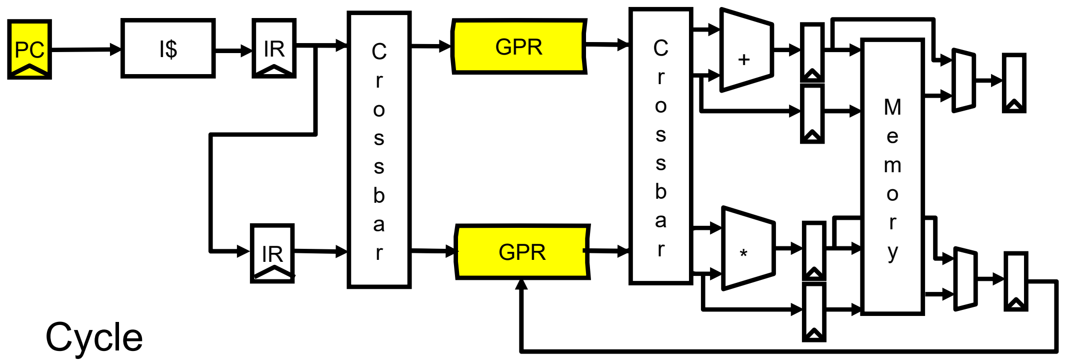


Function unit optimization



Restriction: Can’t issue same operation twice in a row

Function unit optimization



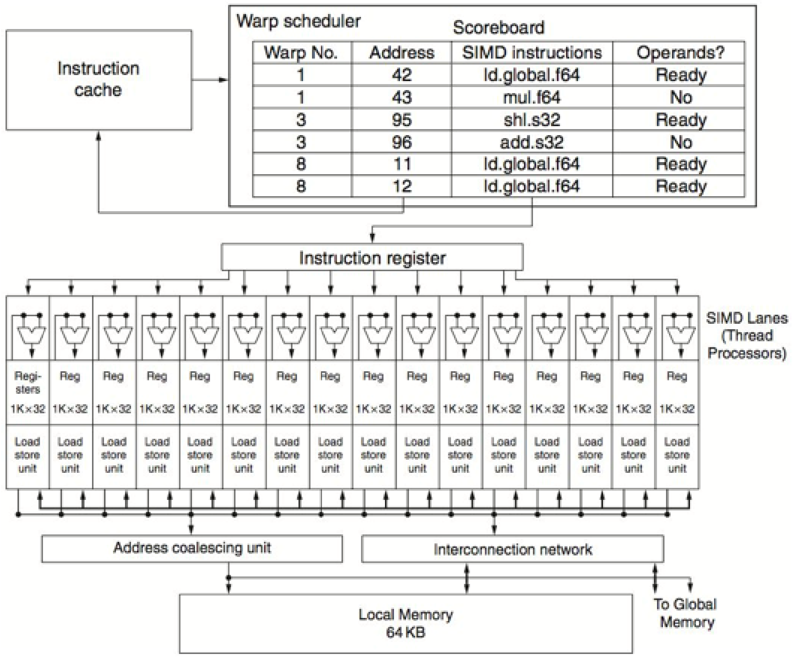
Cycle

Unit

	0	1	2	3	4	5
Fetch	$\text{Add}_{0,0\text{-}1}$	$\text{Mul}_{1,0\text{-}1}$	$\text{Add}_{2,0\text{-}1}$	$\text{Mul}_{3,0\text{-}1}$		…
GPR0		$\text{Add}_{0,0}$	$\text{Mul}_{1,0}$	$\text{Add}_{2,0}$	$\text{Mul}_{3,0}$	…
GPR1			$\text{Add}_{0,1}$	$\text{Mul}_{1,1}$	$\text{Add}_{2,1}$	…
Adder			$\text{Add}_{0,0}$	$\text{Add}_{0,1}$	$\text{Add}_{2,0}$	…
Mul				$\text{Mul}_{1,0}$	$\text{Mul}_{1,1}$	…

Key: $\text{Opcode}_{\text{inum,thread(s)}}$

Streaming Multiprocessor Overview

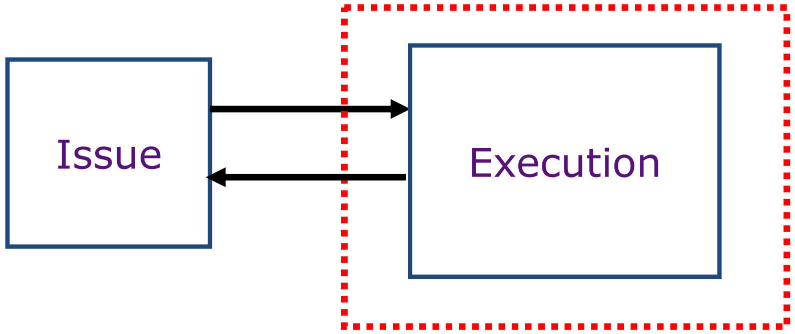


Warp No.	Address	SIMD instructions	Operands?
1	42	ld.global.f64	Ready
1	43	mul.f64	No
3	95	shl.s32	Ready
3	96	add.s32	No
8	11	ld.global.f64	Ready
8	12	ld.global.f64	Ready

- Each SM supports 10s of warps (e.g., 64 in Kepler) with 32 threads/warp
- Fetch 1 instr/cycle
- Issue 1 ready instr/cycle
  - Simple scoreboarding (or static dependency tracking): all warp elements must be ready
- Instruction broadcast to all lanes
- Multithreading is the main latency-hiding mechanism

Little's Law

Throughput ($\overline{T}$) = *Number in Flight* ($\overline{N}$) / *Latency* ($\overline{L}$)



Example:

64 warps (number of instructions in flight)
1 instruction / cycle (desired throughput)

$\Rightarrow$ *<64 cycle average instruction latency*

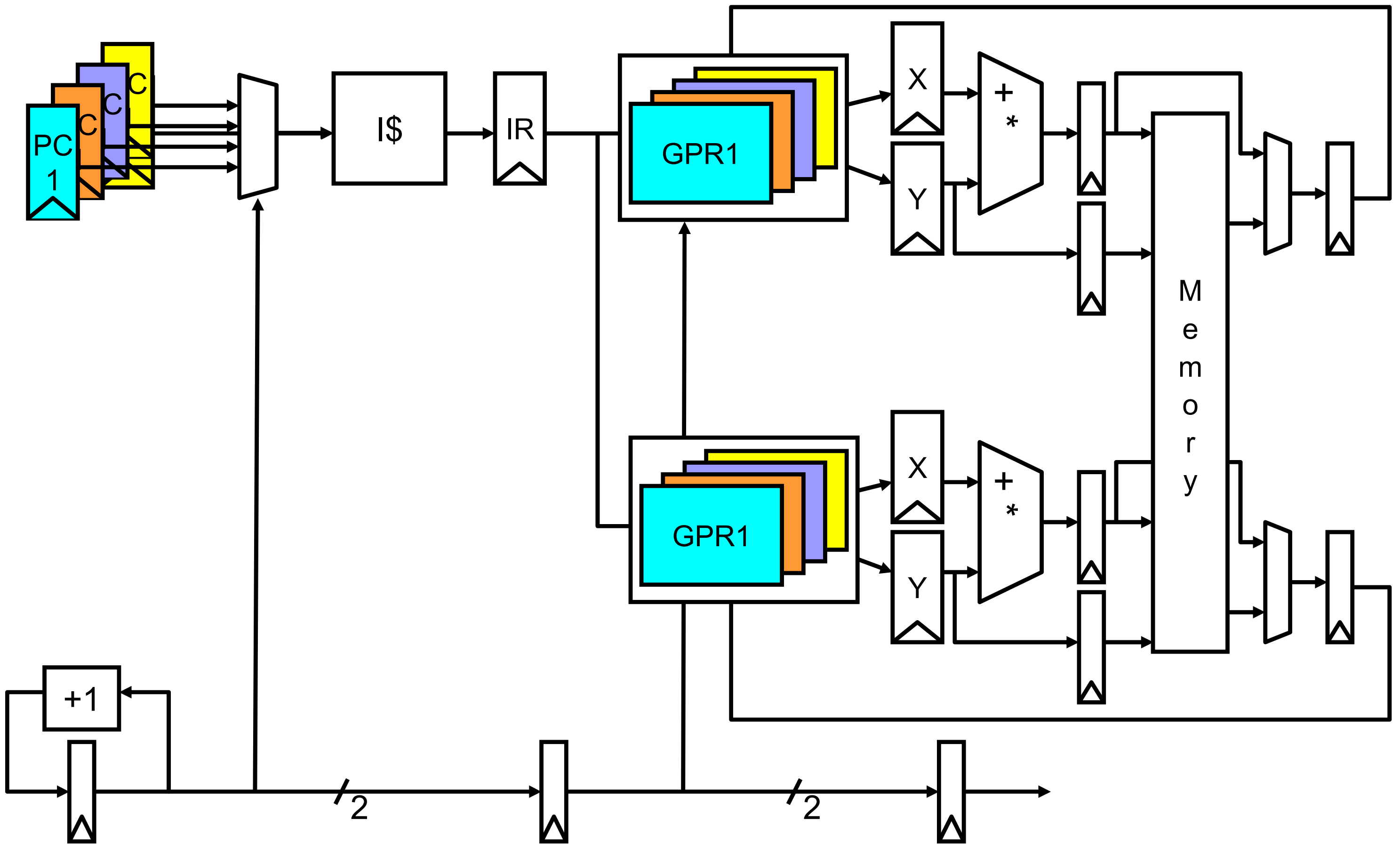
Number of Active Warps

- SMs support a variable number of active warps based on required registers (and shared memory). Fewer warps allows more registers per warp, i.e., a larger context.
  - Few large contexts → Fewer register spills
  - Many small contexts → More latency tolerance
  - Choice left to the compiler
- Example: Kepler has 2K registers/SM and supports up to 64 warps
  - Max: 64 warps @ <=32 registers/thread
  - Min: 8 warps @ 256 registers/thread

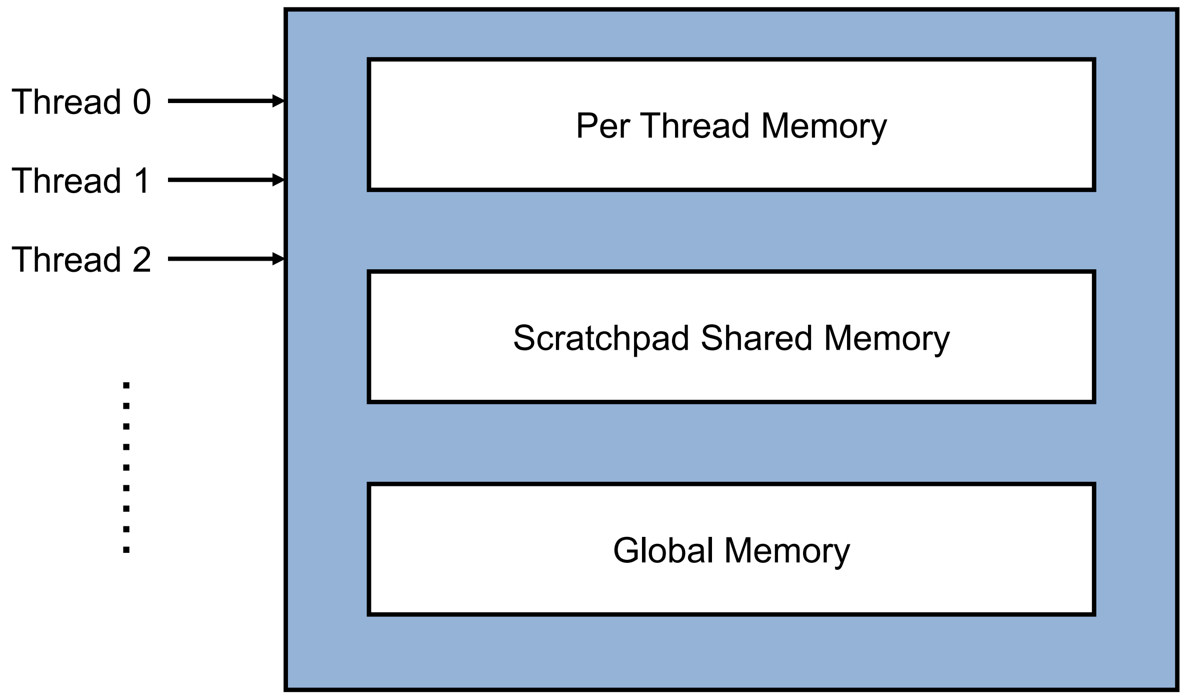
GPU ISA and Compilation

- GPU micro-architecture and instruction set change very frequently
- To achieve compatibility:
  - Compiler produces intermediate pseudo-assembler language (e.g., Nvidia PTX)
  - GPU driver JITs kernel, tailoring it to specific micro-architecture
- In practice, little performance portability
  - Code is often tuned to specific GPU architecture

Multiple Thread – Single Instruction Multiple Thread

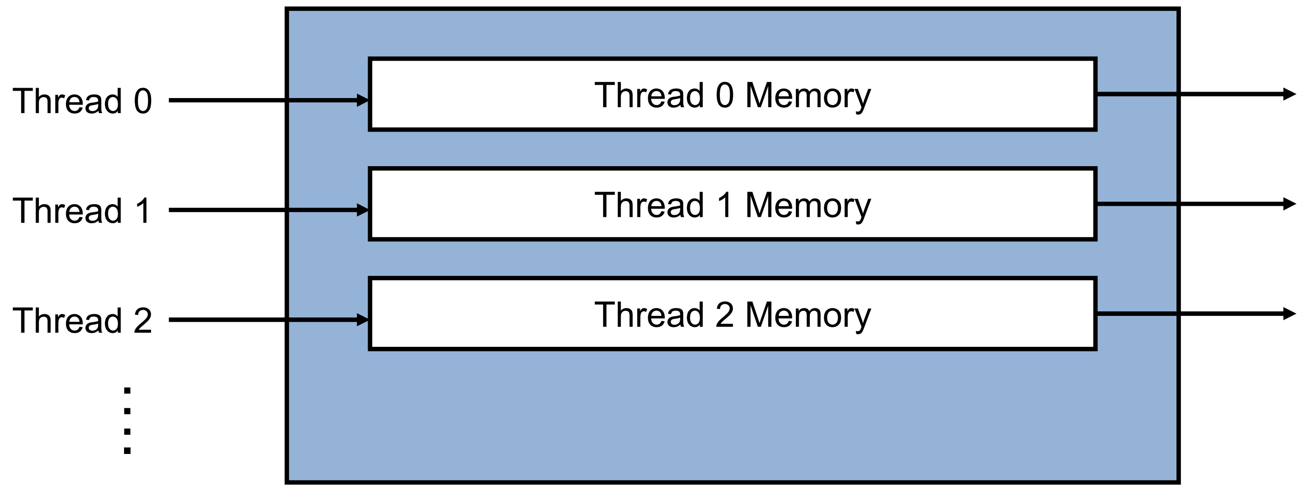


Many Memory Types



Note: Implementation is distinct from semantics.

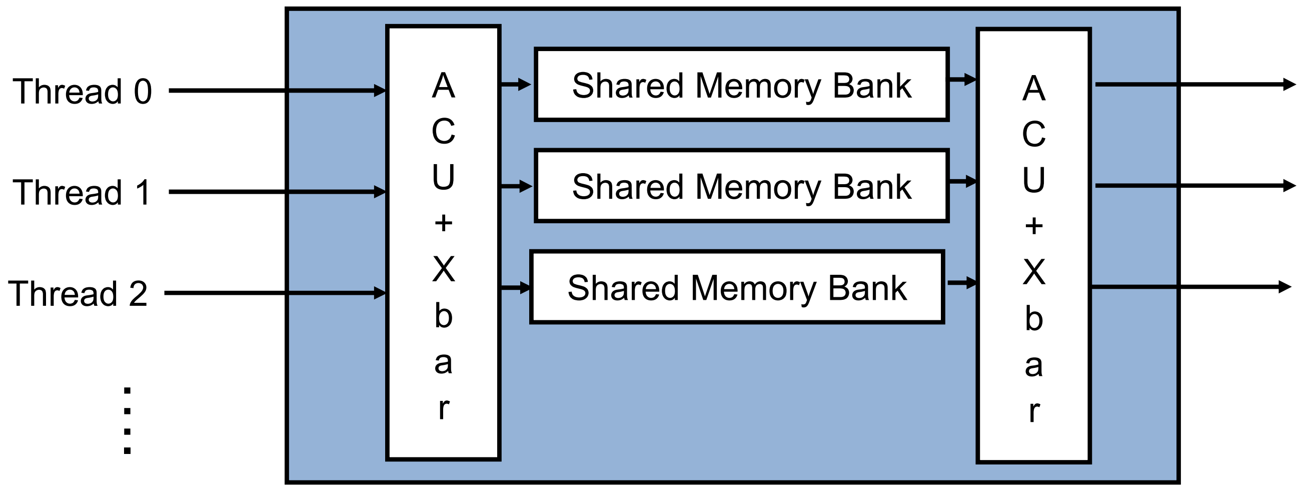
Private Per Thread Memory



Private memory

- No cross-thread sharing
- Small, fixed size memory
  – Can be used for constants
- Multi-bank implementation (can be in global memory)

Shared Scratchpad Memory



Shared scratchpad memory (threads share data)

- Small, fixed size memory (16K-64K / ‘core’)
- Banked for high bandwidth
- Fed with address coalescing unit (ACU) + crossbar
  – ACU can buffer/coalesce requests

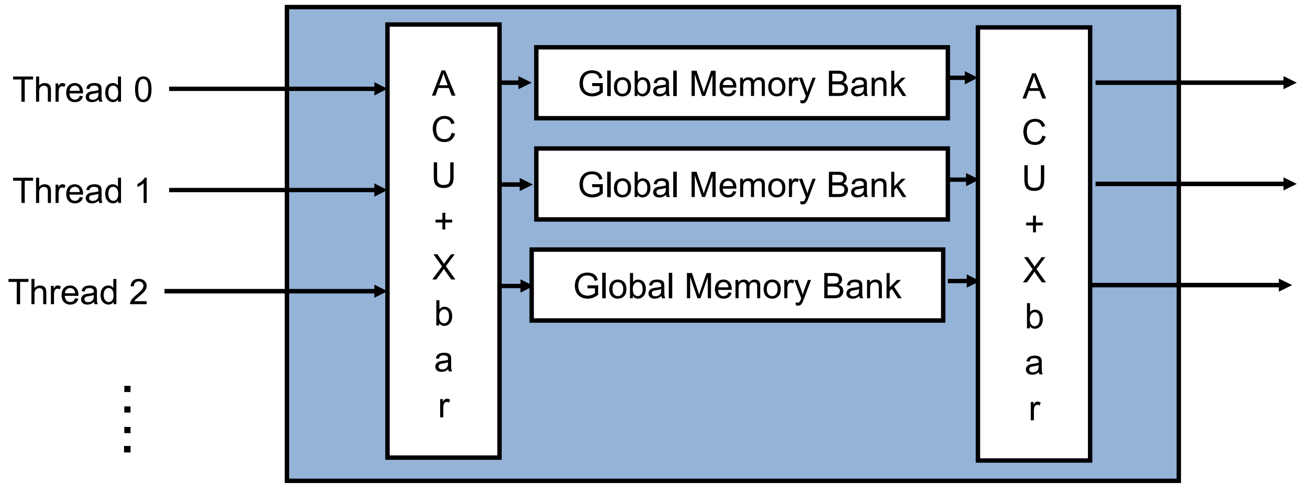
Memory Access Divergence

- All loads are gathers, all stores are scatters
- Address coalescing unit (ACU) detects sequential and strided patterns, coalesces memory requests
- Complex patterns can result in multiple lower bandwidth requests – this is called memory divergence, which hurts performance
- Writing efficient GPU code requires most accesses to not conflict, even though programming model allows arbitrary patterns!

Why isn’t address coalescing as serious problem as in vector machines?

Latency tolerance!

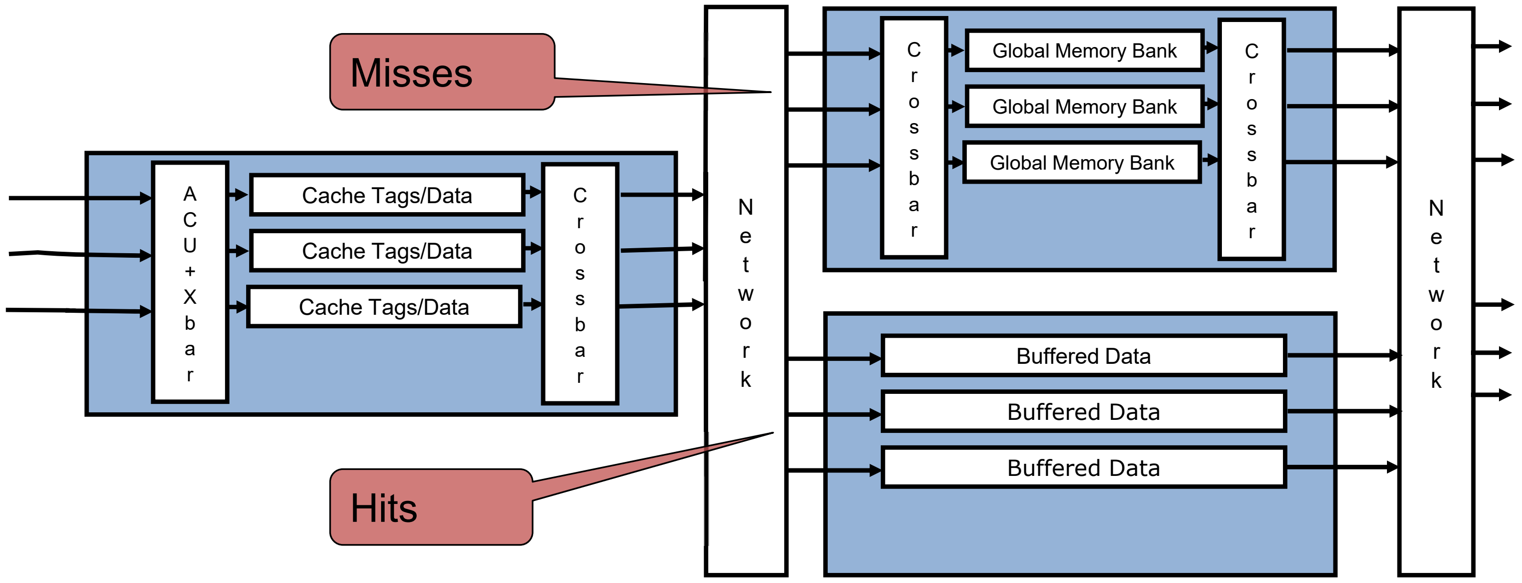
Shared Global Memory (off-chip)



Shared global memory

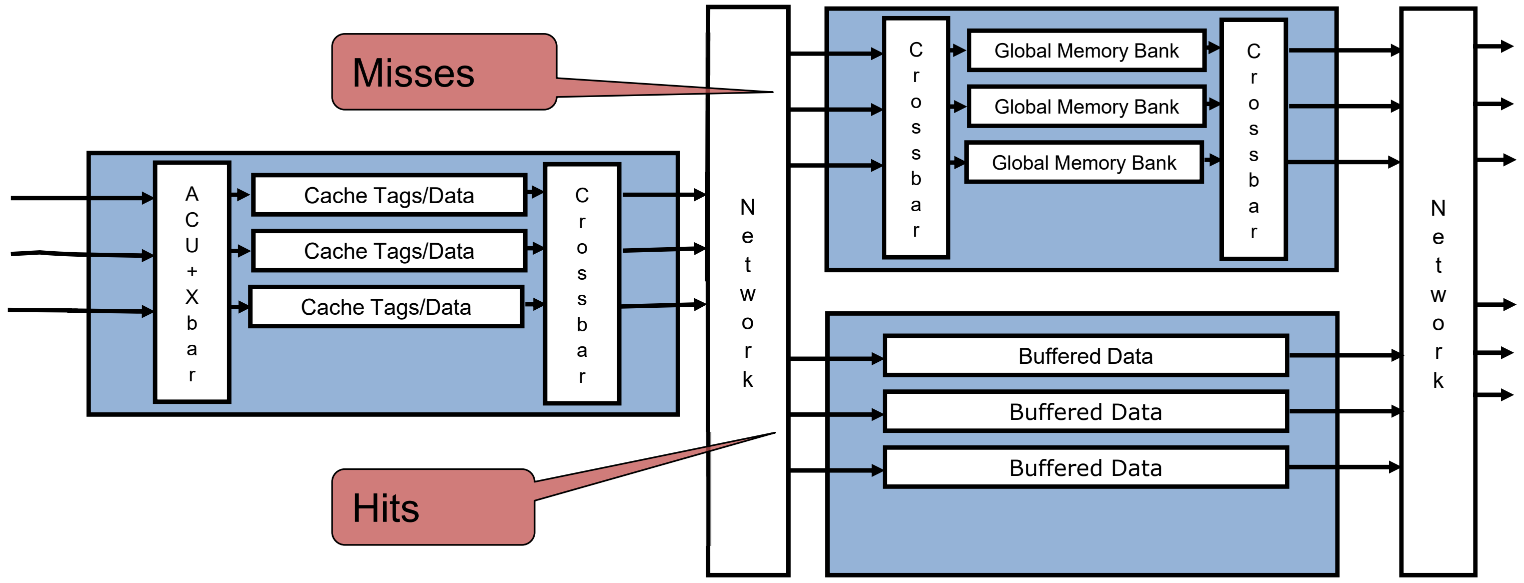
- Large shared memory
- Different requests going to different banks is good
- Will suffer also from memory divergence

Shared Global Memory (off-chip)



Need all requests to come back at the same time!
If hits take the same time as misses, what's the point of caches?

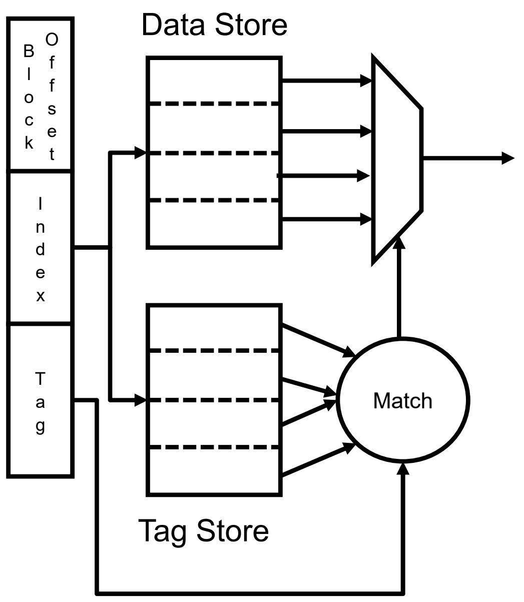
Shared Global Memory (off-chip)

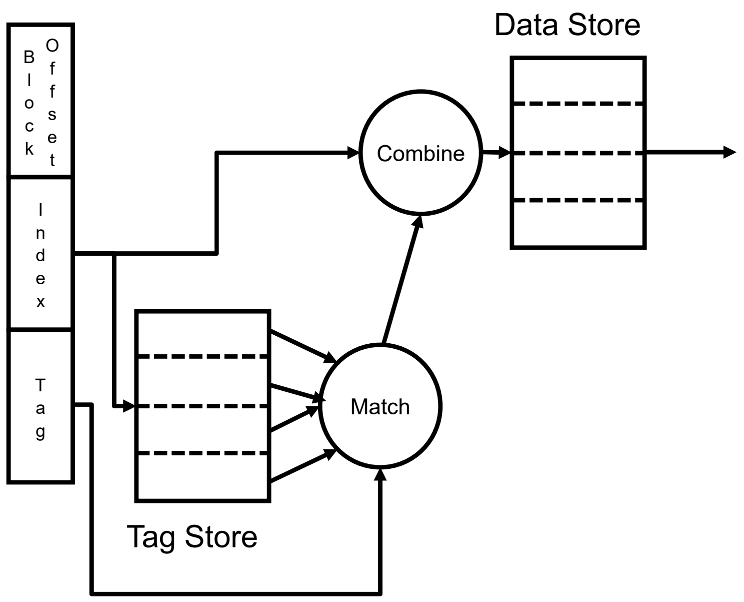


Memory hierarchy with caches

– Cache to save memory bandwidth
– Caches also enable compression/decompression of data

Serialized cache access





Trade latency for power/flexibility

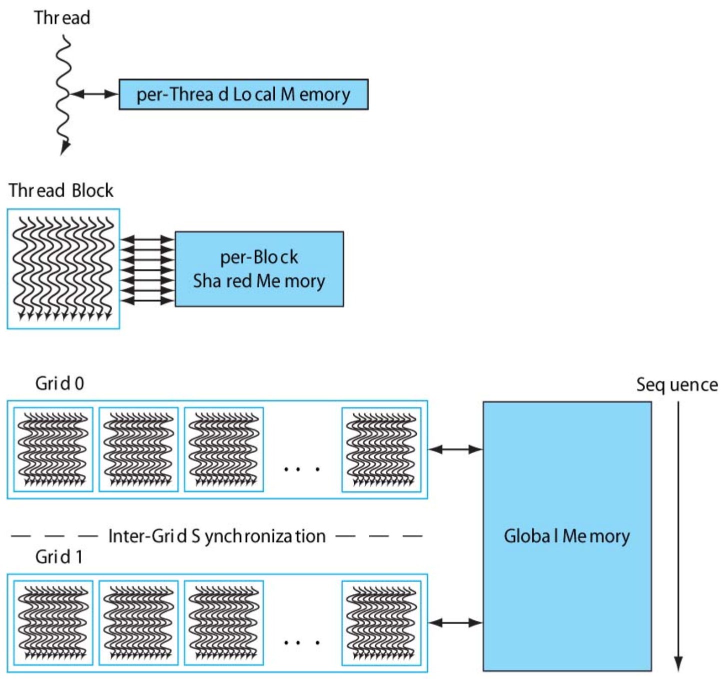
- Only access data bank that contains data
- Facilitate more sophisticated cache organizations
  - e.g., greater associatively

GPU Programming Environments

Hard to generate efficient GPU code directly from C/C++, so new languages (or language variants) have been introduced:

- CUDA (Nvidia-only)
  - C-like language that runs on GPU
  - Libraries: cuDNN, cuBLAS, cuFFT
- OpenCL (open standard)
  - C-like language that runs on GPU, CPU or FPGA
  - usually less optimized than CUDA

CUDA GPU Thread Model



Single-program multiple data (SPMD) model

Each context is a thread

- Threads have registers
- Threads have local memory

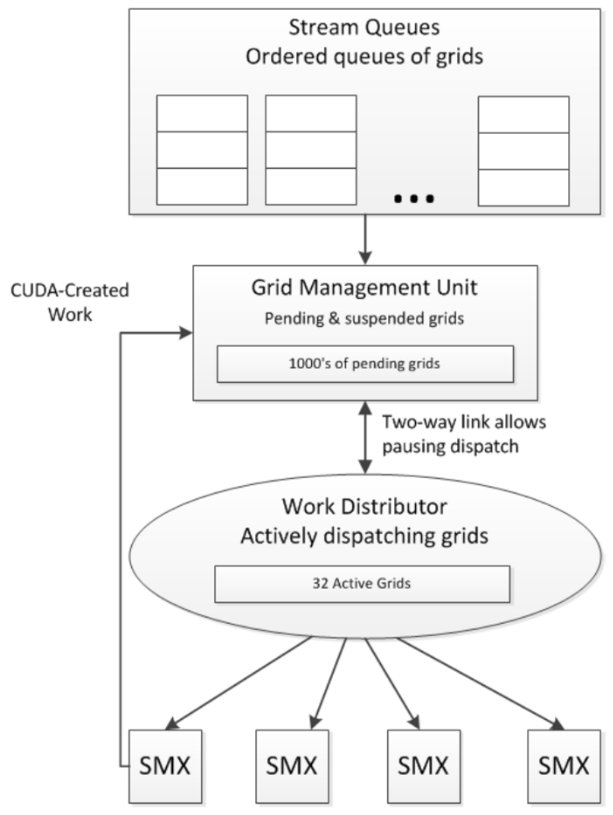
Parallel threads packed in blocks

- Blocks have shared memory
- Threads synchronize with barrier
- Blocks run to completion (or abort)

Grids include independent blocks

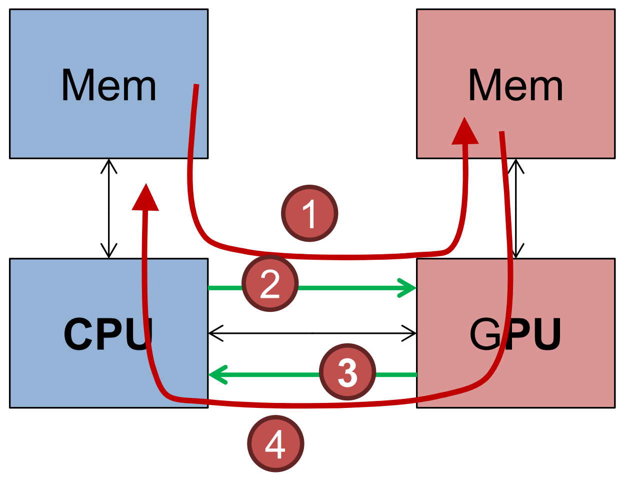
- May execute concurrently
- Share global memory, but
- Have limited inter-block synchronization

Hardware Scheduling



- Grids can be launched by CPU or GPU
  - Work from multiple CPU threads and processes
- HW unit schedules grids on SMs (labeled SMX in diagram)
  - Priority-based scheduling
- 32 active grids
  - More queued/paused

GPU Kernel Execution

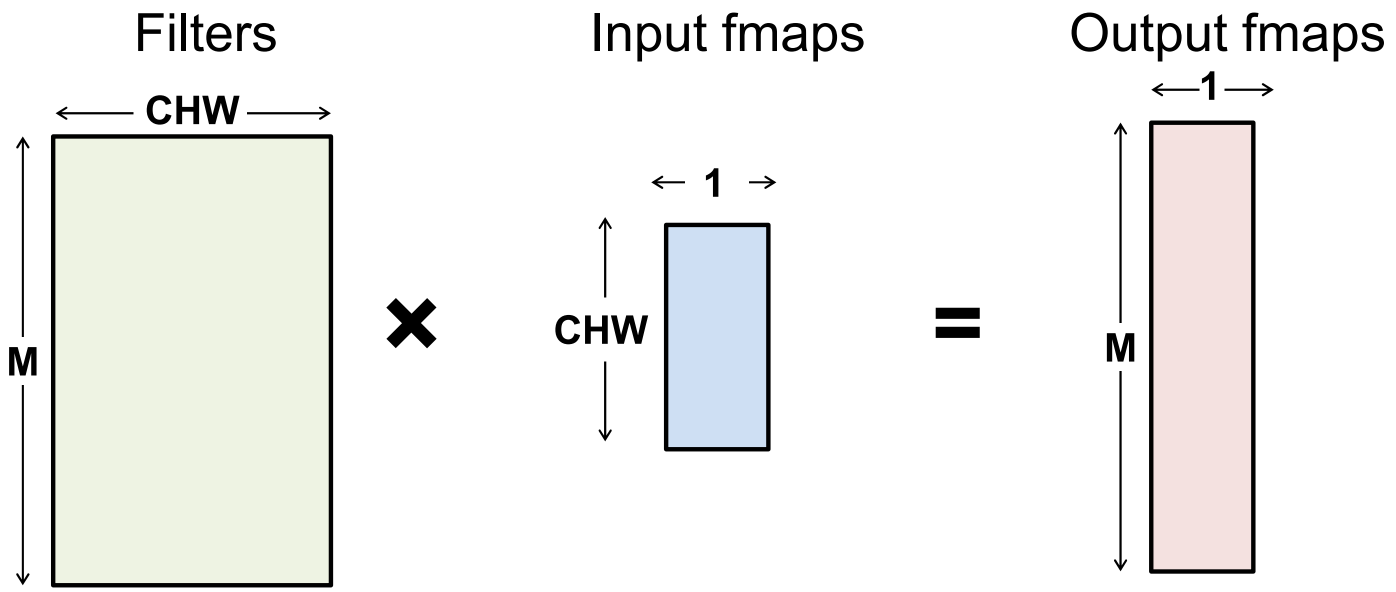


1. Transfer input data from CPU to GPU memory
2. Launch kernel (grid)
3. Wait for kernel to finish (if synchronous)
4. Transfer results to CPU memory

- Data transfers can dominate execution time
- Integrated GPUs with unified address space → no copies, but…

Fully-Connected (FC) Layer

- Matrix–Vector Multiply:
  - Multiply all inputs in all channels by a weight and sum



Fully Connected Computation

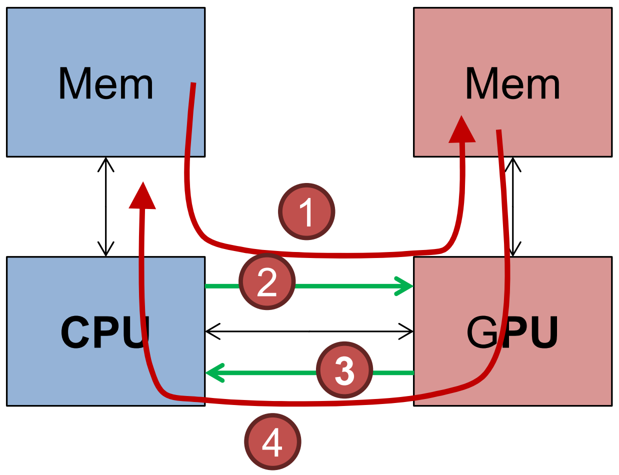
```
int i[C*H*W];     # Input activations
int f[M*C*H*W];   # Filter Weights
int o[M];         # Output activations


for (m = 0; m < M; m++) {
  o[m] = 0;
  CHWm = C*H*W*m;
  for (chw = 0; chw < C*H*W; chw++) {
     o[m] += i[chw] * f[CHWm + chw];
  }
}
```

Parallelize here

$I[C_0\ H_0\ W_0]$ $I[C_0\ H_0\ W_1]$ …
$I[C_0\ H_1\ W_0]$ $I[C_0\ H_1\ W_1]$ …
$I[C_0\ H_2\ W_0]$ $I[C_0\ H_2\ W_1]$ …
.
.
$I[C_1\ H_0\ W_0]$ $I[C_1\ H_0\ W_1]$ …
$I[C_1\ H_1\ W_0]$ $I[C_1\ H_1\ W_1]$ …
$I[C_1\ H_2\ W_0]$ $I[C_1\ H_2\ W_1]$ …
.
.
.

$F[M_0\ C_0\ H_0\ W_0]$ $F[M_0\ C_0\ H_0\ W_1]$ …
$F[M_0\ C_0\ H_1\ W_0]$ $F[M_0\ C_0\ H_1\ W_1]$ …
$F[M_0\ C_0\ H_2\ W_0]$ $F[M_0\ C_0\ H_2\ W_1]$ …
.
.
$F[M_0\ C_1\ H_0\ W_0]$ $F[M_0\ C_1\ H_0\ W_1]$ …
$F[M_0\ C_1\ H_1\ W_0]$ $F[M_0\ C_1\ H_1\ W_1]$ …
$F[M_0\ C_1\ H_2\ W_0]$ $F[M_0\ C_1\ H_2\ W_1]$ …
.
.
.
$F[M_1\ C_0\ H_0\ W_0]$ $F[M_1\ C_0\ H_0\ W_1]$ …
$F[M_1\ C_0\ H_1\ W_0]$ $F[M_1\ C_0\ H_1\ W_1]$ …
$F[M_1\ C_0\ H_2\ W_0]$ $F[M_1\ C_0\ H_2\ W_1]$ …
.
.
.

GPU Kernel Execution



1. Transfer input data from CPU to GPU memory
2. Launch kernel (grid)
3. Wait for kernel to finish (if synchronous)
4. Transfer results to CPU memory

- Data transfers can dominate execution time
- Integrated GPUs with unified address space → no copies, but…

FC - CUDA Main Program

```
int i[C*H*W],    *gpu_i;    # Input activations
int f[M*C*H*W], *gpu_f;    # Filter Weights
int o[M],        *gpu_o;    # Output activations

# Allocate space on GPU
cudaMalloc((void**) &gpu_i, sizeof(int)*C*H*W);
cudaMalloc((void**) &gpu_f, sizeof(int)*M*C*H*W);

# Copy data to GPU
cudaMemcpy(gpu_i, i, sizeof(int)*C*H*W);
cudaMemcpy(gpu_f, f, sizeof(int)*M*C*H*W);

# Run kernel
fc<<<M/256 + 1, 256>>>(gpu_i, gpu_f, gpu_o, C*H*W, M);

# Copy result back and free device memory
cudaMemcpy(o, gpu_o, sizeof(int)*M, cudaMemCpyDevicetoHost);
cudaFree(gpu_i);
...
```

References to data on GPU

Num thread blocks

Thread block size

FC – CUDA Terminology

- CUDA code launches 256 threads per block

- CUDA vs vector terminology:
  - Thread = 1 iteration of scalar loop [1 element in vector loop]
  - Block = Body of vectorized loop [VL=256 in this example]
    - Warp size = 32 [Number of vector lanes]
  - Grid = Vectorizable loop

[vector terminology]

FC - CUDA Kernel

```
__global__
void fc(int *i, int *f, int *o, const int CHW, const int M){

    int tid=threadIdx.x + blockIdx.x * blockDim.x;
    int m = tid

    int sum = 0;

    if (m < M){
        for (int chw=0; chw <CHW; chw ++) {
            sum += i[chw]*f[(m*CHW)+chw];
        }

        o[m] = sum;
    }
}
```

Code for one thread

Calculate “global” thread id (tid)

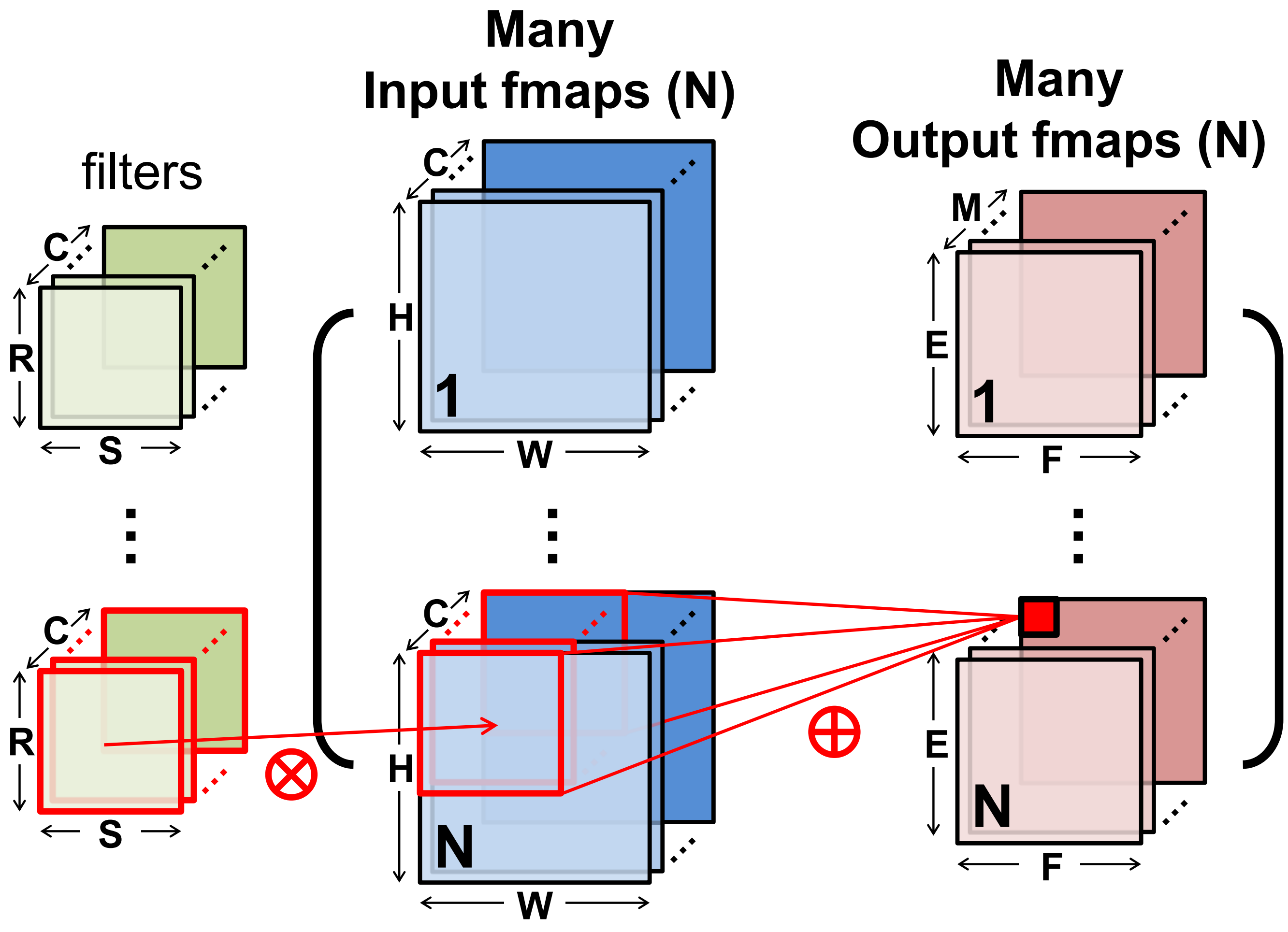
tid is “output channel” number

Can be unrolled

Any consequences of f[(m*CHW)+chw]? Yes, strided references

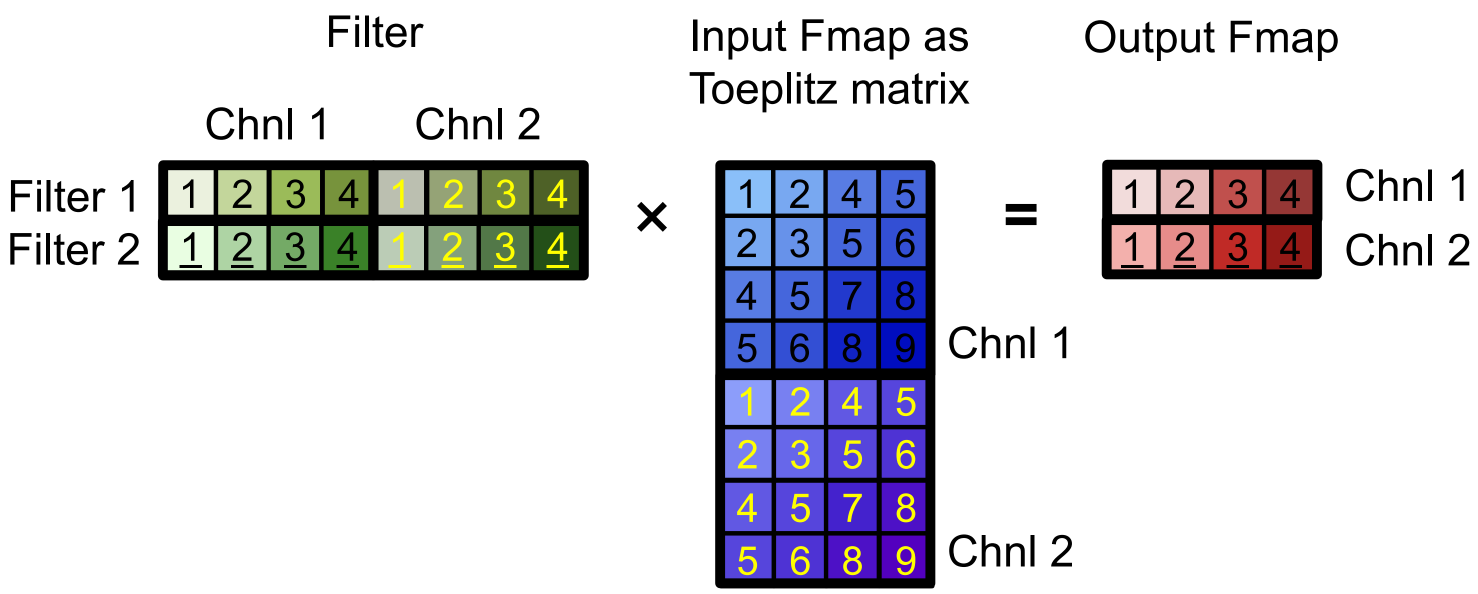
Any consequences of f[(chw*M)+m]? Yes, different data layout

Convolution (CONV) Layer



Batch Size (N)

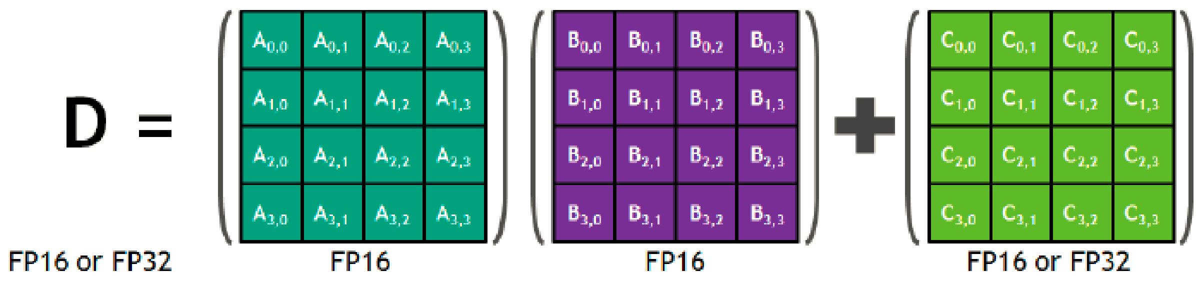
Convolution (CONV) Layer



GPU Implementation:

- Keep original input activation matrix in main memory
- Conceptually do tiled matrix-matrix multiplication
- Copy input activations into scratchpad to small Toeplitz matrix tile
- On Volta tile again to use ‘tensor core’

GV100 – “Tensor Core”



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

New opcodes – Matrix Multiply Accumulate (HMMA)

How many FP16 operands? Inputs 48 / Outputs 16

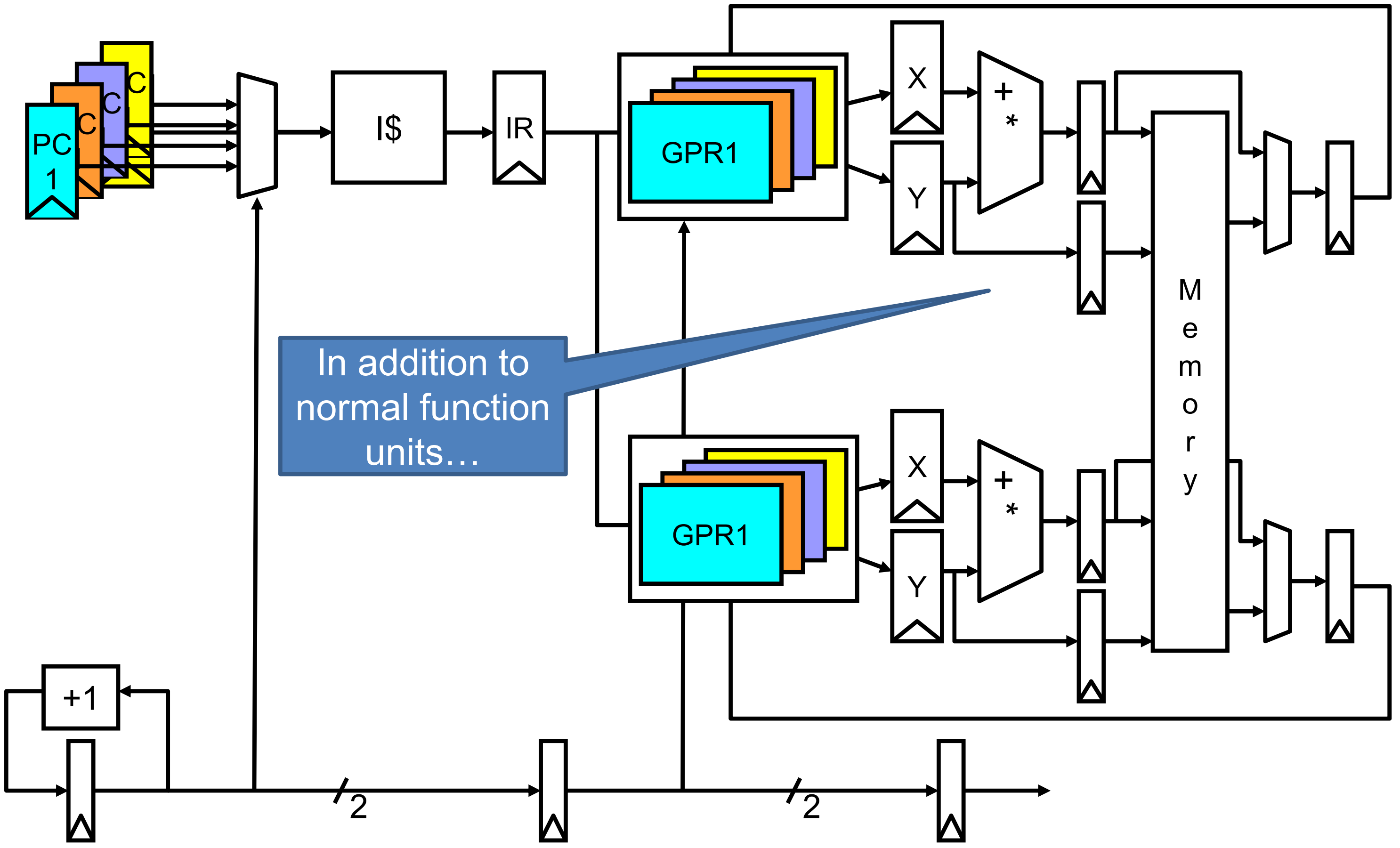
How many multiplies? 64

How many adds? 64

Volta tensor Core…. 120 TFLOPS (FP16), 400 GFLOPS/W (FP16)

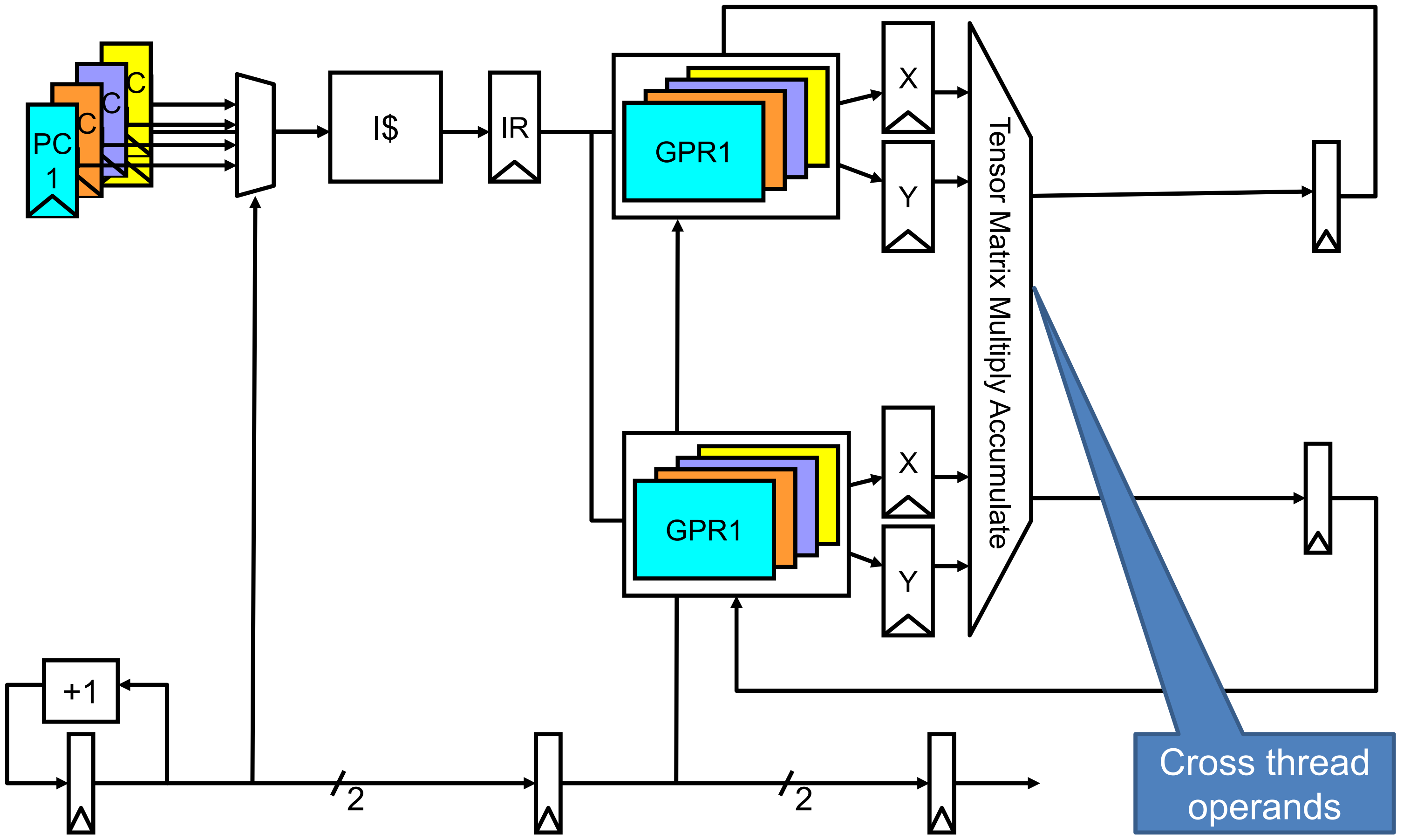
Toeplitz expansion is essential to exploit hardware

Tensor Core Integration



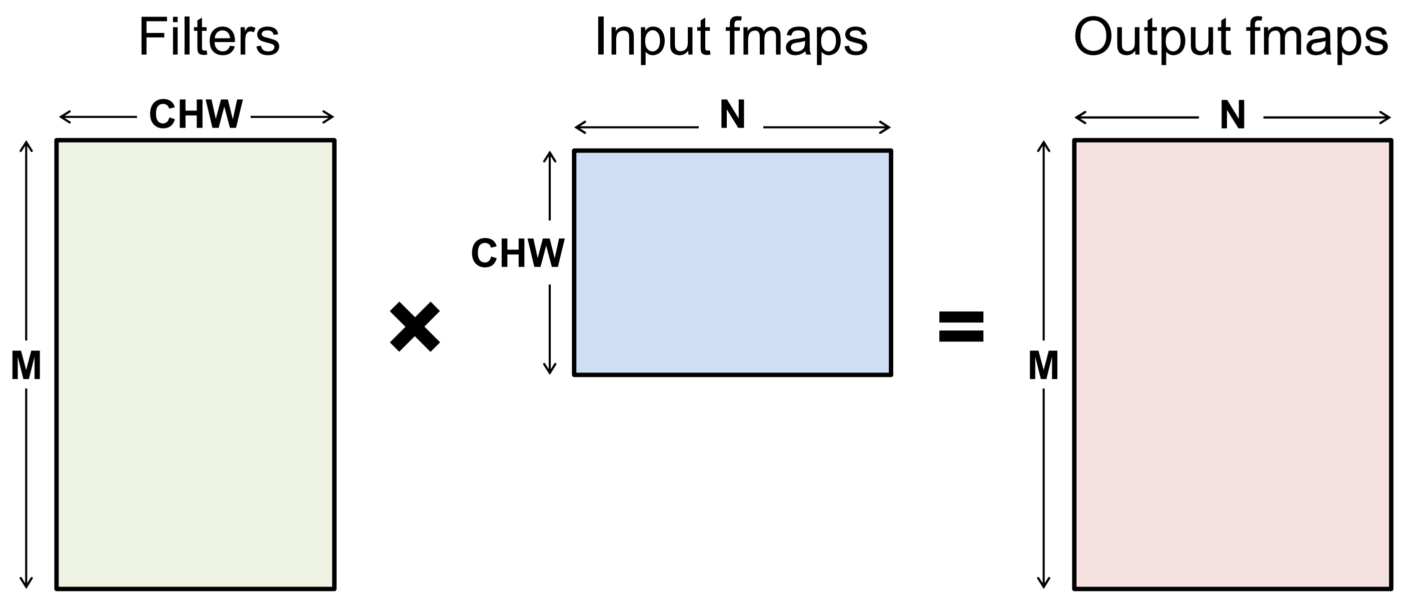
In addition to normal function units…

Tensor Core Integration



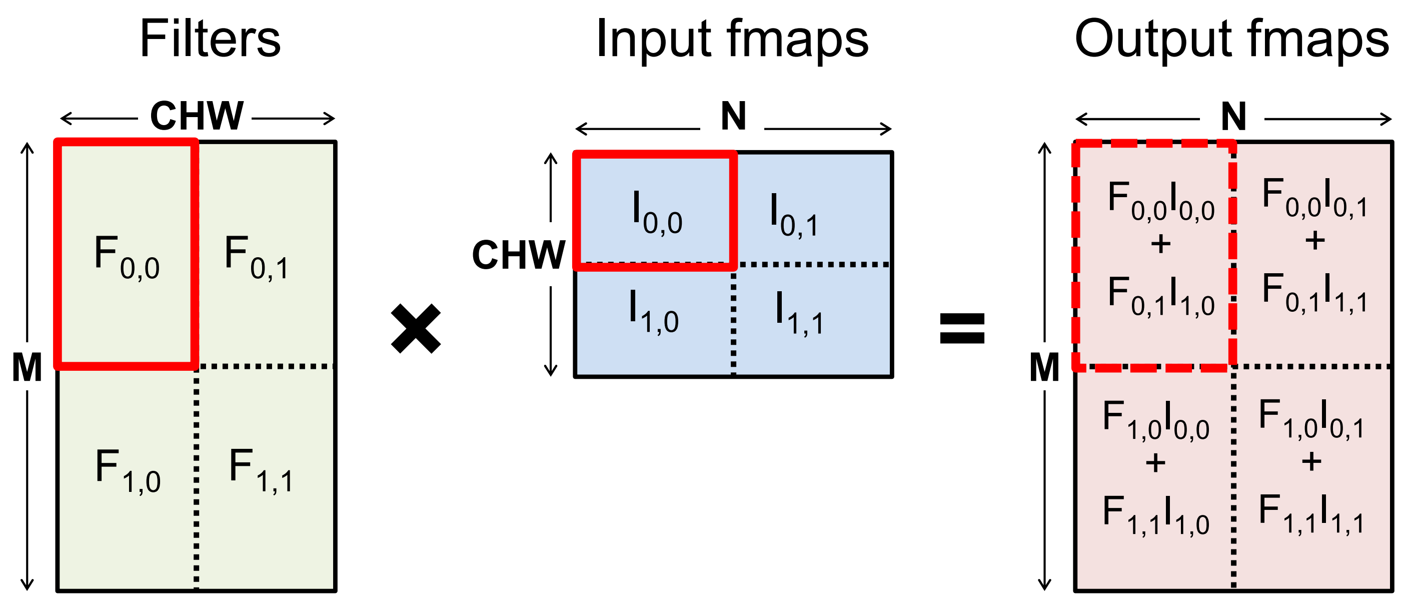
Cross thread operands

Fully-Connected (FC) Layer



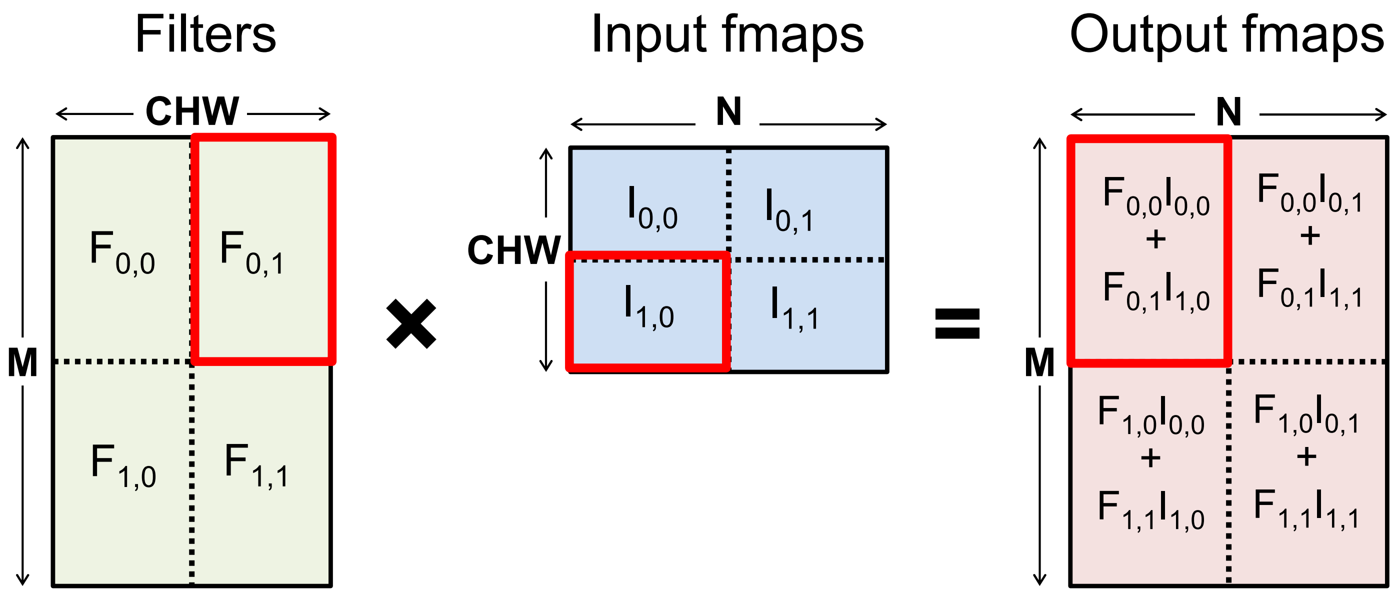
- After flattening, having a batch size of N turns the matrix-vector operation into a matrix-matrix multiply

Tiled Fully-Connected (FC) Layer


Filters
Input fmaps
Output fmaps
CHW
M
N
CHW
N
M
$F_{0,0}$
$F_{0,1}$
$F_{1,0}$
$F_{1,1}$
$I_{0,0}$
$I_{0,1}$
$I_{1,0}$
$I_{1,1}$
$F_{0,0}I_{0,0} + F_{0,1}I_{1,0}$
$F_{0,0}I_{0,1} + F_{0,1}I_{1,1}$
$F_{1,0}I_{0,0} + F_{1,1}I_{1,0}$
$F_{1,0}I_{0,1} + F_{1,1}I_{1,1}$

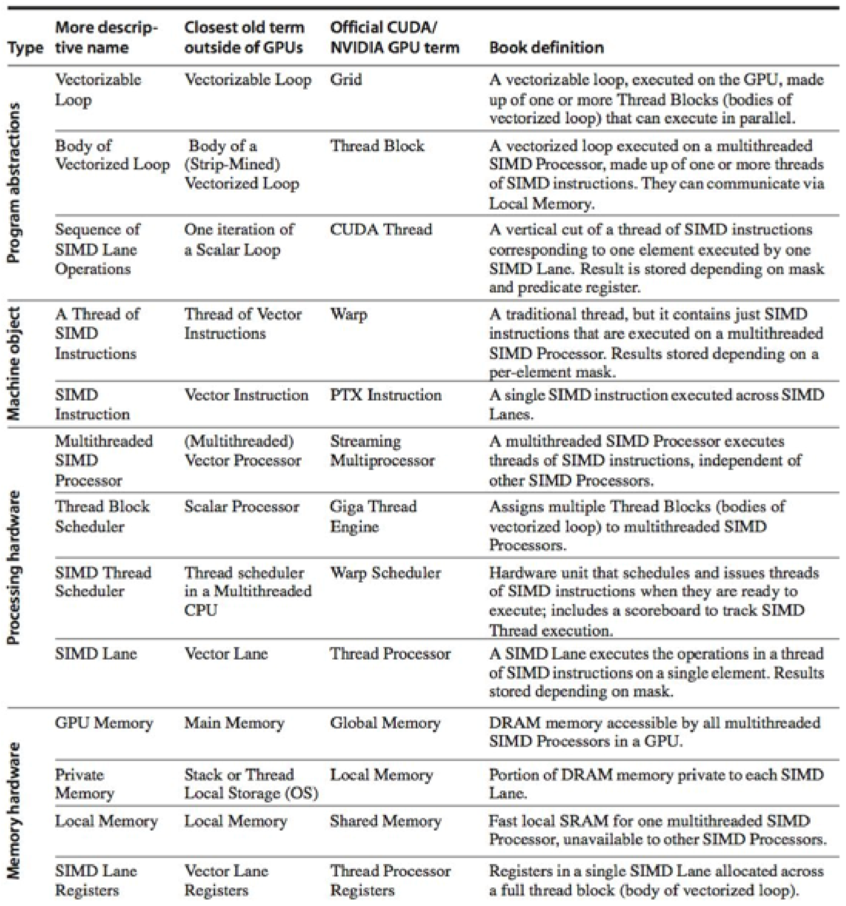
Matrix multiply tiled to fit in tensor core operation
and computation ordered to maximize reuse of data in scratchpad
and then in cache.

Tiled Fully-Connected (FC) Layer


Filters
Input fmaps
Output fmaps
CHW
M
N
$F_{0,0}$
$F_{0,1}$
$F_{1,0}$
$F_{1,1}$
$I_{0,0}$
$I_{0,1}$
$I_{1,0}$
$I_{1,1}$
$F_{0,0}I_{0,0} + F_{0,1}I_{1,0}$
$F_{0,0}I_{0,1} + F_{0,1}I_{1,1}$
$F_{1,0}I_{0,0} + F_{1,1}I_{1,0}$
$F_{1,0}I_{0,1} + F_{1,1}I_{1,1}$

Matrix multiply tiled to fit in tensor core operation
and computation ordered to maximize reuse of data in scratchpad
and then in cache.

Vector vs GPU Terminology



Type	More descriptive name	Closest old term outside of GPUs	Official CUDA/ NVIDIA GPU term	Book definition
Program abstractions	Vectorizable Loop	Vectorizable Loop	Grid	A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel.
	Body of Vectorized Loop	Body of a (Strip-Mined) Vectorized Loop	Thread Block	A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory.
	Sequence of SIMD Lane Operations	One iteration of a Scalar Loop	CUDA Thread	A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register.
Machine object	A Thread of SIMD Instructions	Thread of Vector Instructions	Warp	A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask.
	SIMD Instruction	Vector Instruction	PTX Instruction	A single SIMD instruction executed across SIMD Lanes.
Processing hardware	Multithreaded SIMD Processor	(Multithreaded) Vector Processor	Streaming Multiprocessor	A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors.
	Thread Block Scheduler	Scalar Processor	Giga Thread Engine	Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors.
	SIMD Thread Scheduler	Thread scheduler in a Multithreaded CPU	Warp Scheduler	Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution.
	SIMD Lane	Vector Lane	Thread Processor	A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask.
Memory hardware	GPU Memory	Main Memory	Global Memory	DRAM memory accessible by all multithreaded SIMD Processors in a GPU.
	Private Memory	Stack or Thread Local Storage (OS)	Local Memory	Portion of DRAM memory private to each SIMD Lane.
	Local Memory	Local Memory	Shared Memory	Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors.
	SIMD Lane Registers	Vector Lane Registers	Thread Processor Registers	Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop).

[H&P5, Fig 4.25]

Summary

- GPUs are an intermediate point in the continuum of flexibility and efficiency

- GPU architecture focuses on throughput (over latency)
  - Massive thread-level parallelism, with
    - Single instruction for many threads
  - Memory hierarchy specialized for throughput
    - Shared scratchpads with private address space
    - Caches used primarily to reduce bandwidth not latency
  - Specialized compute units, with
    - Many computes per input operand

- Little’s Law useful for system analysis
  - Shows relationship between throughput, latency and tasks in-flight

Next Lecture: Spatial Architectures

Thank you!