GPU Programming

Farshad Khunjush

Department of Computer Science and Engineering
Shiraz University
Fall 2025

Intoduction to GPUs
Part 1

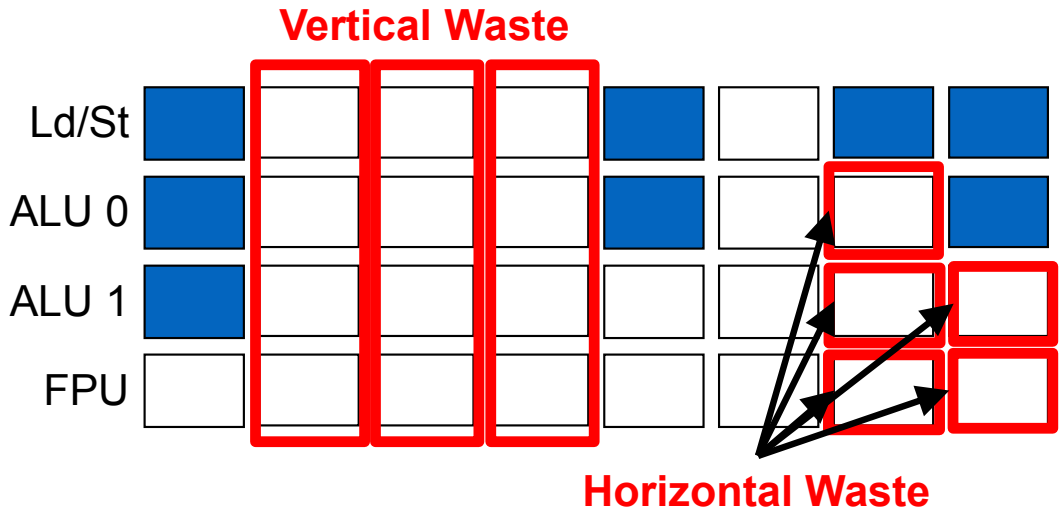
Farshad Khunjush

Slides come form

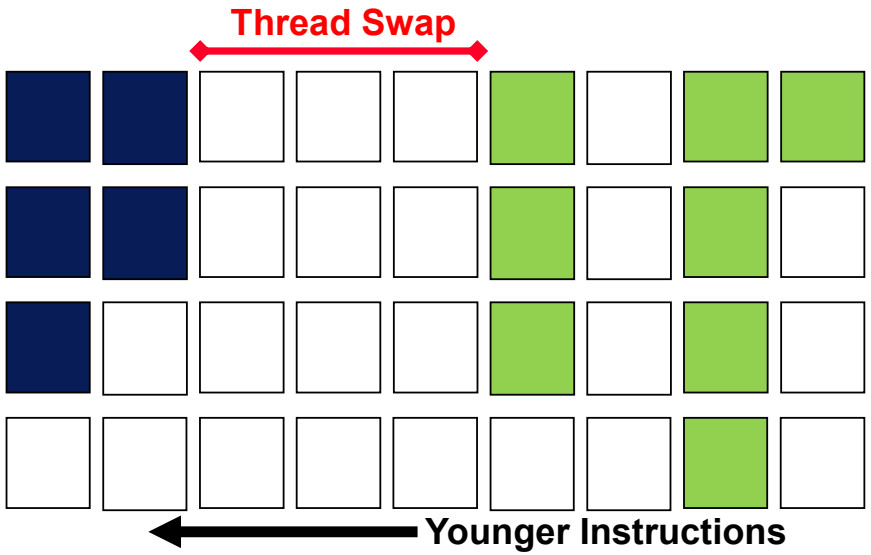
Professor Babak Falsafi @ EPFL

Multithreading (Review)

- Pipelines waste cycles in two ways
- Vertical: whole cycle empty, nothing issued
  - Most common after long latency events
- Horizontal: unable to use full issue width
  - Software not exposing enough ILP for hardware

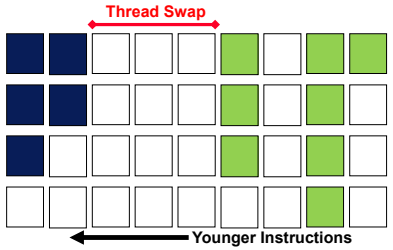


Coarse Grain Multithreading



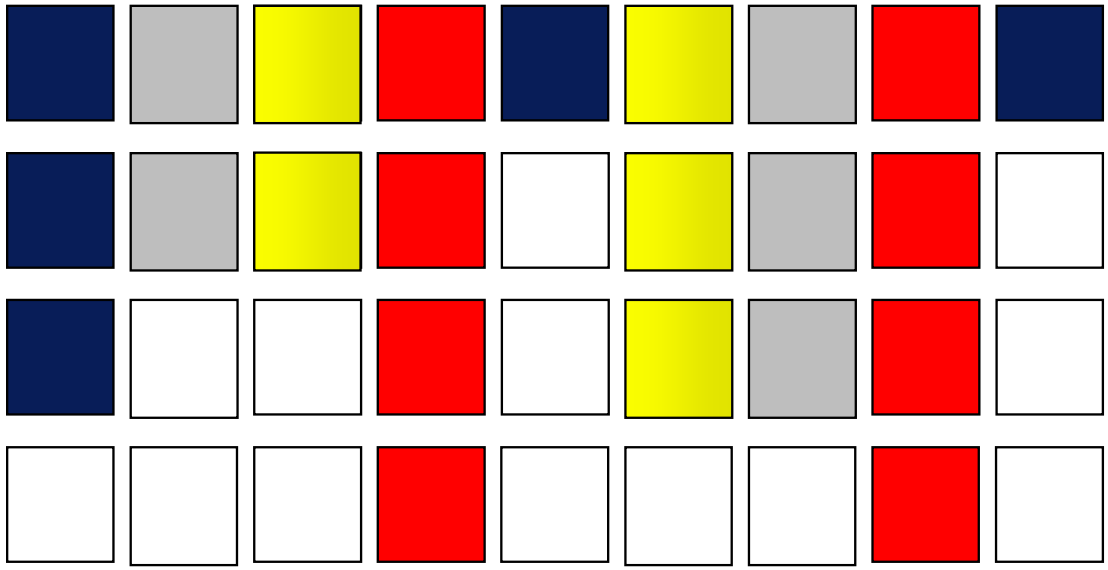
- Switch to a new thread on a long latency event
  - Pay a small cost to switch in a new context
  - Addresses purely vertical waste

CGMT Performance



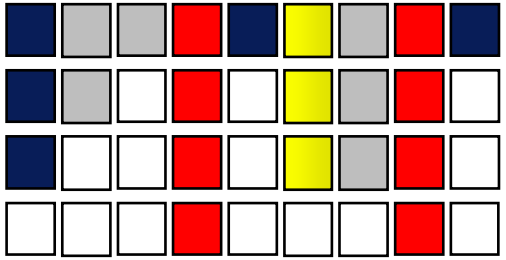
- Critical decision: when to switch threads
  - When current thread's utilization is about to drop
  - (e.g. L2 cache miss)
- Requirements for improving throughput:
  - (Thread switch) + (pipe fill time) << blocking latency
    - Need useful work to be done before other thread comes back
  - Fast thread-switch: multiple register banks
  - Fast pipe-fill: short pipe
- Advantage: small changes to existing hardware
- Drawback: single-thread performance suffers

Fine Grained Multithreading (FGMT)



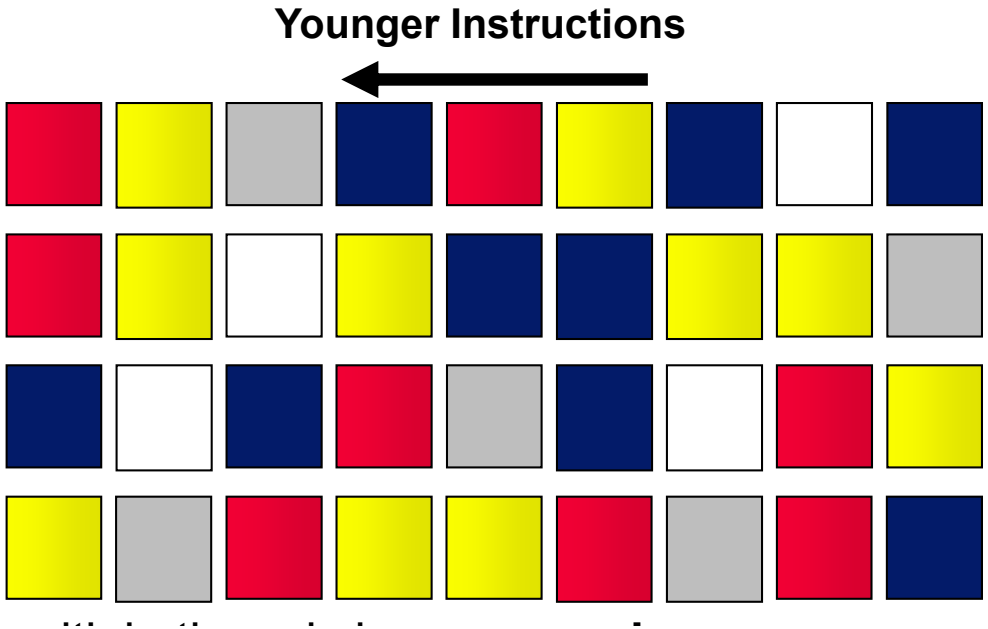
- Cycle between multiple threads periodically
  - Eliminate "switch time" by keeping all thread state hot

FGMT vs. CGMT



- Thread switch policy:
  - Most historical designs of FGMT are round robin
  - CGMT swaps on long latency events
- FGMT addresses much shorter latencies
  - Hardware costs to keep all contexts immediately ready
- For apps. with abundant ILP, FGMT could suffer
  - Can introduce "flexible interleaving" to solve
  - One thread remains scheduled for many cycles

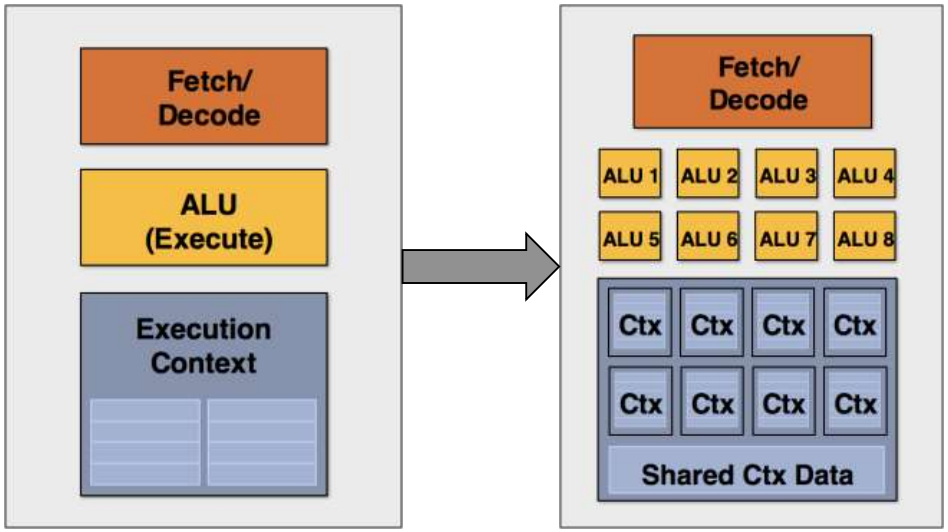
Simultaneous Multithreading (SMT)



- Instructions from multiple threads in **same cycle**
  - First design that can reduce horizontal waste

Taking Parallelism Further

- We started with a single instruction CPU...
  - Added contexts, different ways to multithread
  - Makes our processor more latency-tolerant
  - Also added ALUs to make it SIMD capable



Limitation on Arithmetic Units

- Adding ALUs has an upper limit
  - We need enough instructions to utilize all of them
  - Recall last time, pipeline width has diminishing benefit
- SMT addresses horizontal waste, by bringing in more independent instructions
  - Also has an upper limit: scaling register files and ROB!

Better Idea: One Instruction, All ALUs

- Previously, rely on programs to expose ILP
  - SMT is a pure hardware technique to exploit it
- Now, control all of the ALUs with one instruction
  - This is a "vector" processor, built ages ago

    (Cray 1 - 1976, Illinois ILLIAC IV – 1972)
- Key point: we've gone from "program threads" to "SIMD threads" running in lockstep

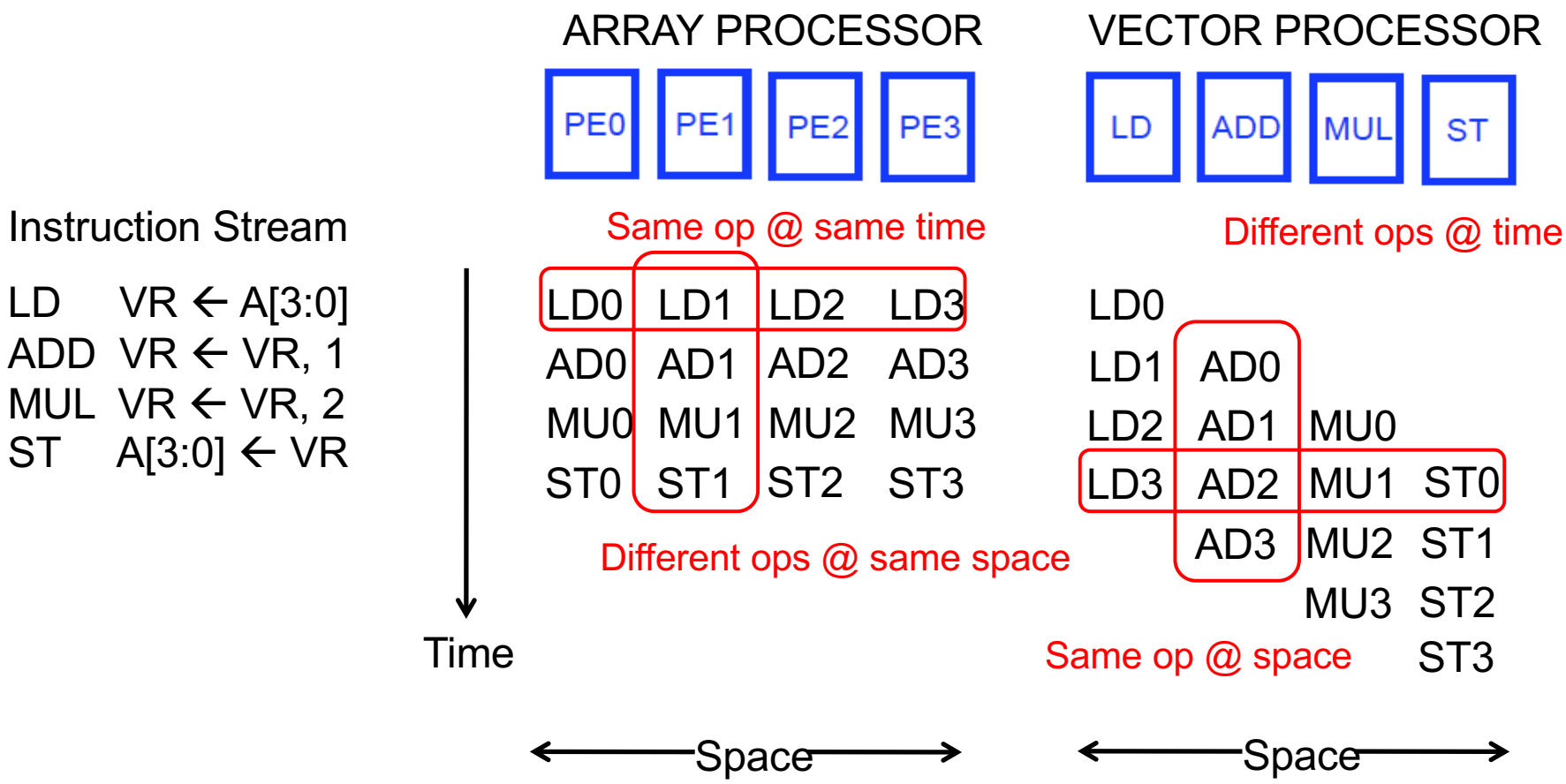
Vector vs. SIMD

- SIMD instructions have configurable width, and actually move data there
  - e.g., x86 operations call them xmms
    - Can specify 64, 128, 256, 512 bit widths
- Vector processors operate in **strides**
  - Gather elements from memory, operate in parallel on all of them
    - No need to change code for different SIMD implementations!
- GPUs are more like vector processors
  - Since instructions are executed all together, operations happen in large vectors

SIMD Processing

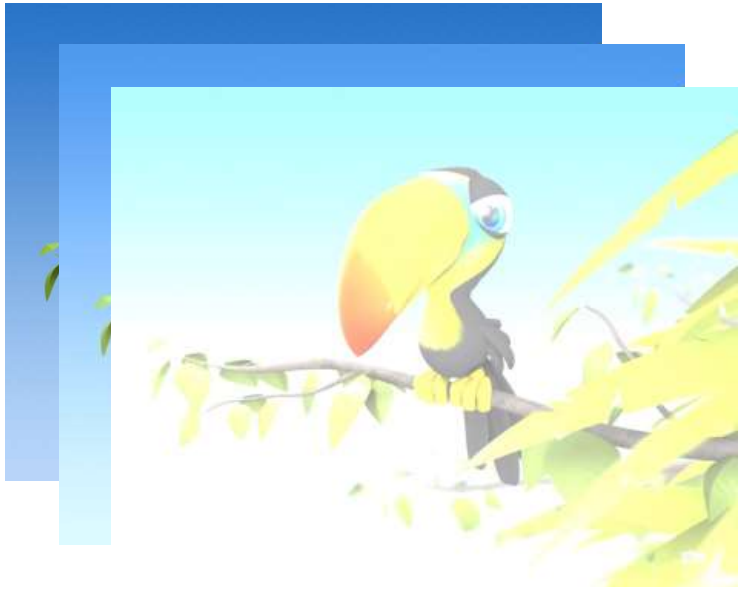
- Single instruction operates on multiple data elements
  - In time or in space
- Multiple processing elements (PEs), i.e., execution units
- Time-space duality
  - Array processor: Instruction operates on multiple data elements at the same time using different spaces (PEs)
  - Vector processor: Instruction operates on multiple data elements in consecutive time steps using the same space (PE)

Array vs. Vector Processors



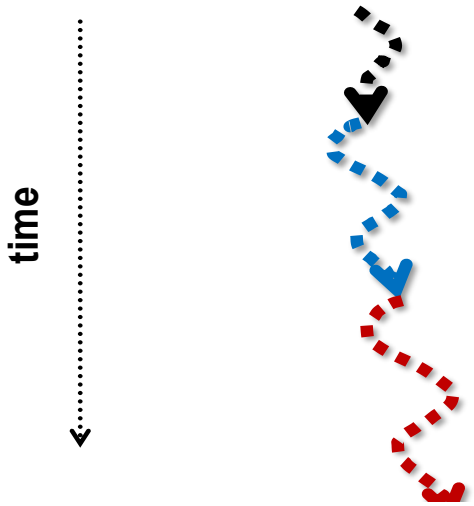
Example: Image Processing

- Simple operation to do a fade out
  - Take every pixel and multiply by a constant (e.g. 0.7)
- Single thread programming: loop over all pixels



Example: Image Processing

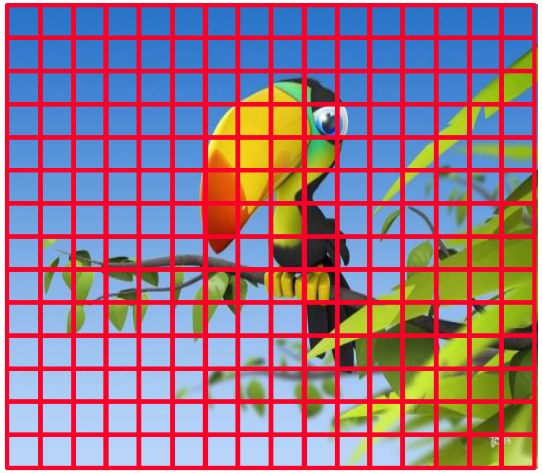
- Control flow happens every loop iteration
  - One instruction at a time
  - Optimizations are all done in the hardware
    - e.g. Loop unrolling, branch prediction, SMT

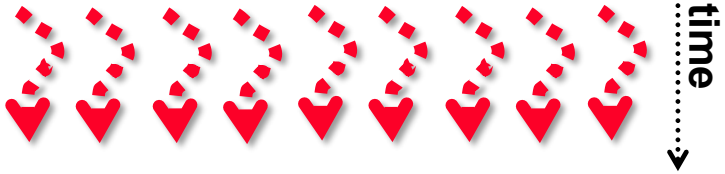



```
for(int i=0; i < a_size; i++) {
   array[i] *= fade;
}
```

Example: Image Processing

- But, given hypothetical software parallel language
  - Divide the work among a bunch of threads
  - Do everything in parallel



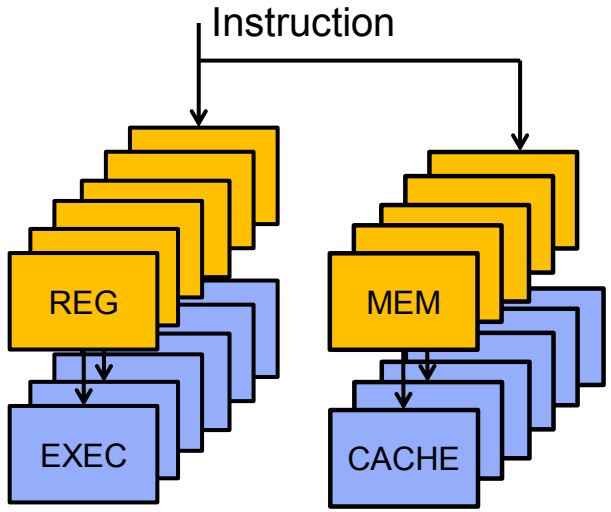



```
// hypothetically in parallel
for_all_pixels {
    array[i] *= fade;
}
```

Massively Parallel Hardware

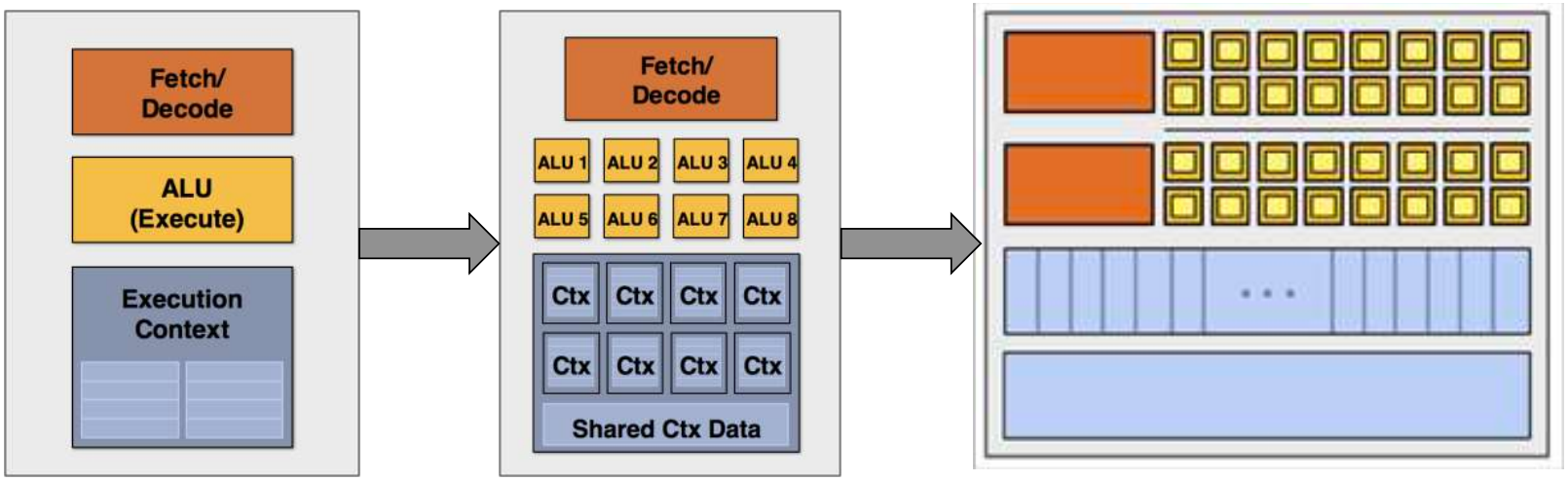
- Scale pipeline width, e.g., 1k ALUs
- Key insight: use a single instruction stream
  - Because we know all threads are doing the same thing

    (e.g. a multiplication operation and memory writes)



Massively Parallel Hardware

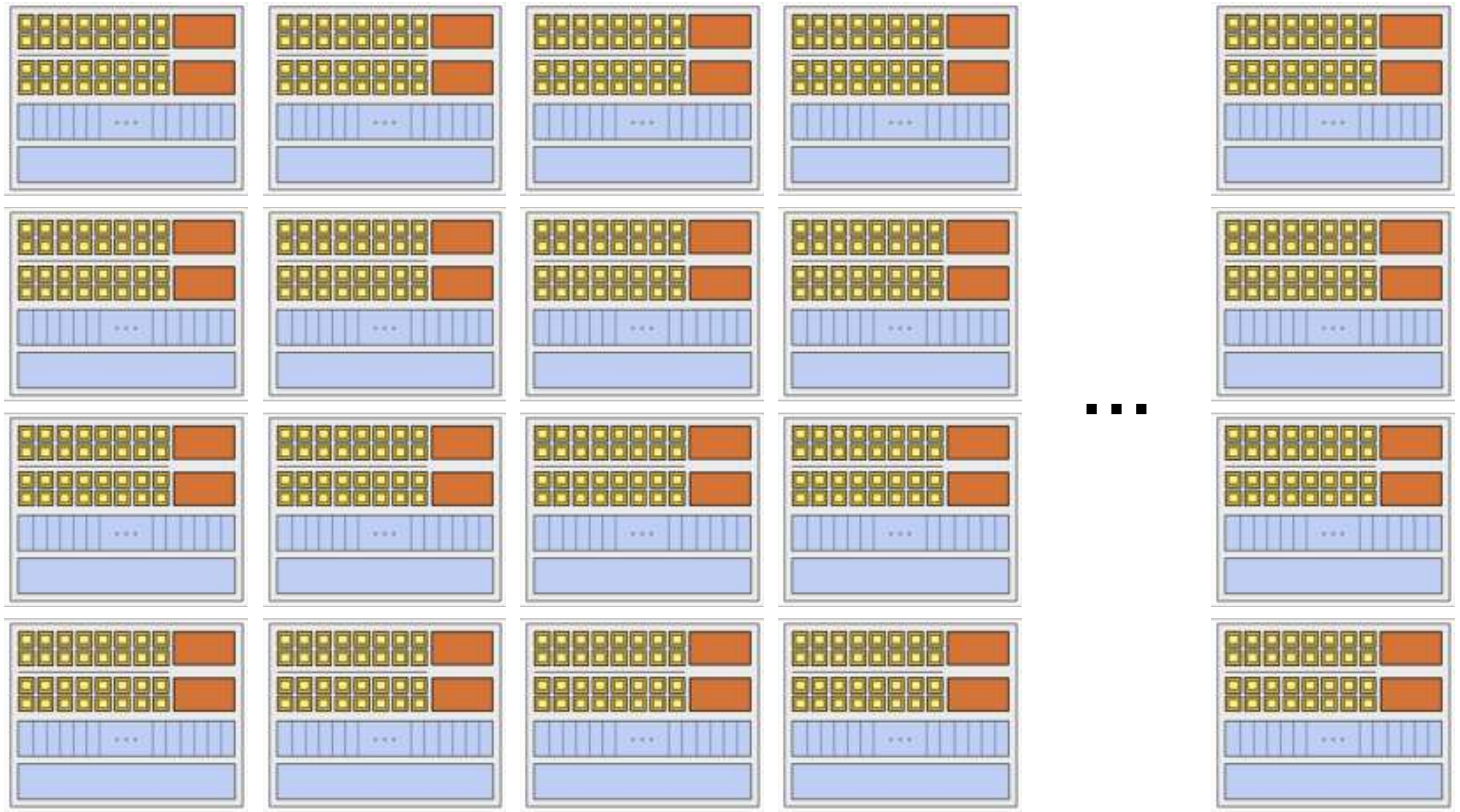
- Works great for embarrassingly parallel programs
- The problem? Memory and long latency ops.
  - Solution: Use huge degree of multithreading
    - Every clock cycle should have a thread ready to execute



~~Massively Parallel Hardware~~ GPUs

- Make cores simple
  - In-order pipelines
- Put thousands of ALUs on the die
  - NVIDIA Volta has ~5.5k integer and ~2.5k FP ALUs
  - Reminder: NVIDIA calls these "cores", but they are not similar to CPU cores
  - Parallel software favors throughput
- Share cores among thousands of threads
- Take throughput-latency trade-off to extreme
  - Trillions of integer operations per second
  - ... but, huge single thread latency

Example: NVIDIA Tesla V100



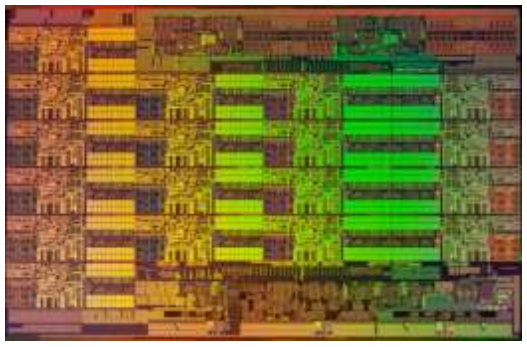
5120 CUDA cores!
(much smaller than CPU cores)

Example: NVIDIA Volta V100

Example: CPU vs. GPU

Intel Skylake-X:

- ~600 mm2
- 28 cores/56 threads
- Hierarchy: 32KB/1MB/38.5MB/1.5TB
- 205 Watts



GPU V100:

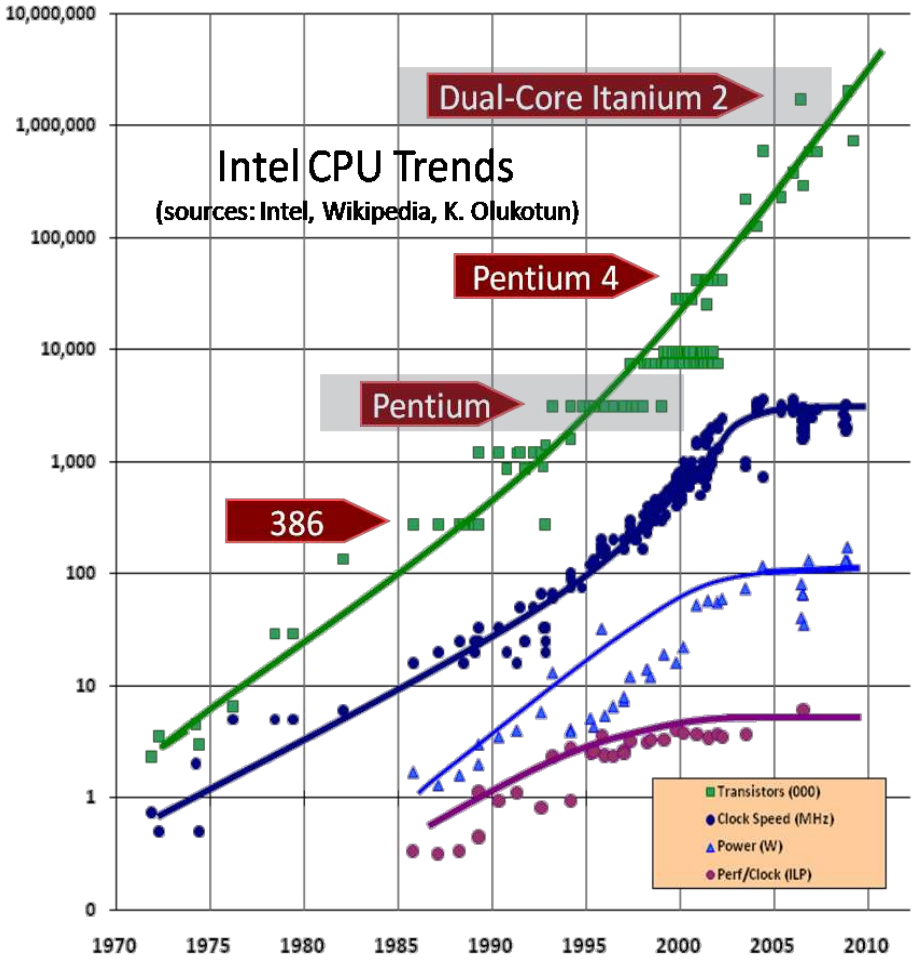
- 815 mm2
- 5120 CUDA cores
- 2MB reg file
- Hierarchy: 96KB/256KB/16GB
- 300 Watts

One Slide of GPU History

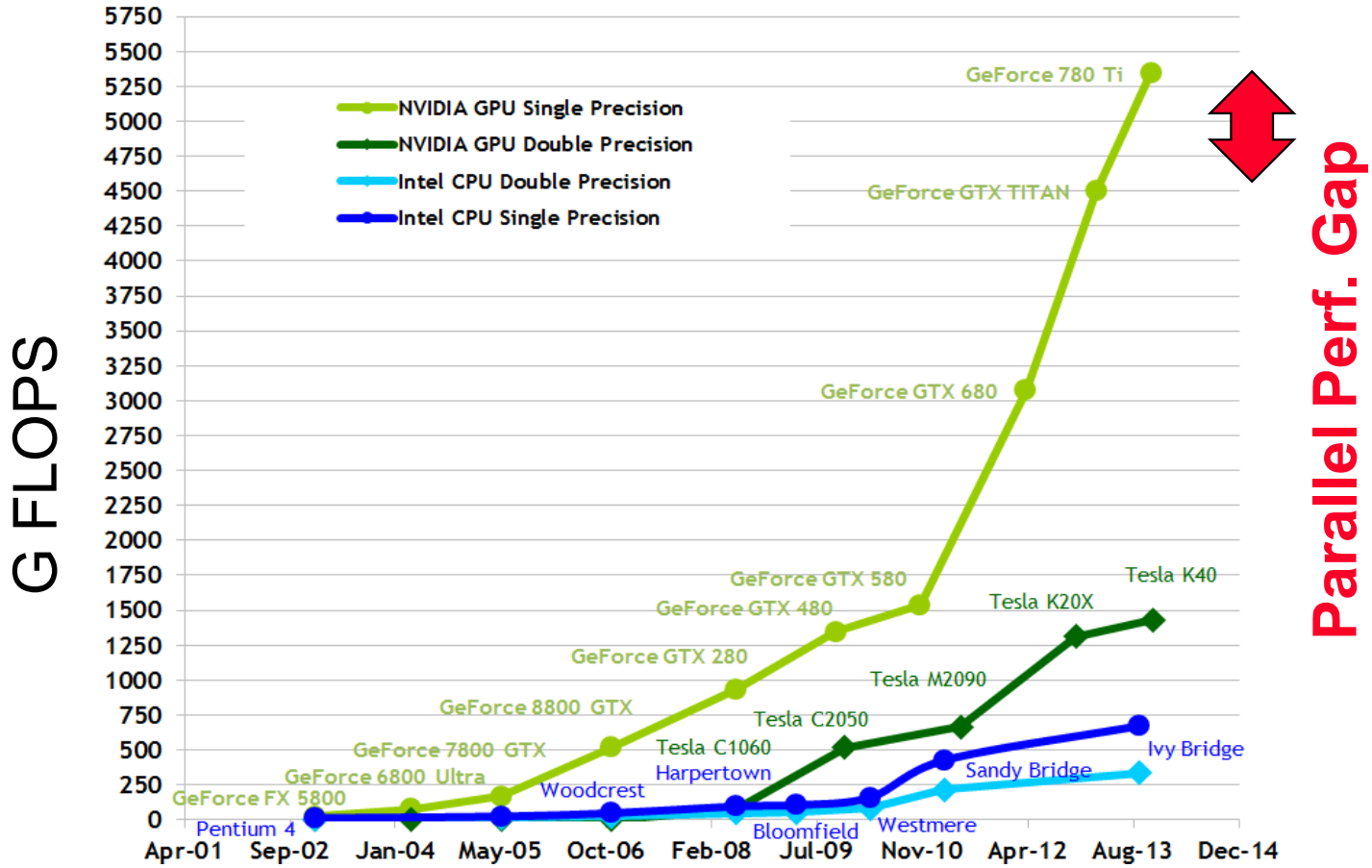
- It doesn't seem like it, but the popularity of modern GPUs sort of happened by accident
  - They were originally meant for graphics **only**
  - **No** general purpose programming language
    - Have fun with OpenGL
- Researchers started using them for computation around 2003 – called "General Purpose GPUs" (GPGPUs)
  - Plenty of data parallel computation (e.g., simulation)
  - The latest killer app is AI/ML with DNNs
  - Now, all GPUs can run regular declarative programs

Increase in GPU Popularity

- Remember this graph?
  - Free lunch is over
- Moore's Law ver. 2
  - Chip density still increases
  - Clock speed does not
- GPUs turn density to compute (parallelism)
- CPUs on single thread performance
  - But, this has saturated



GPU Parallelism Keeps Scaling



Why are GPUs Interesting?

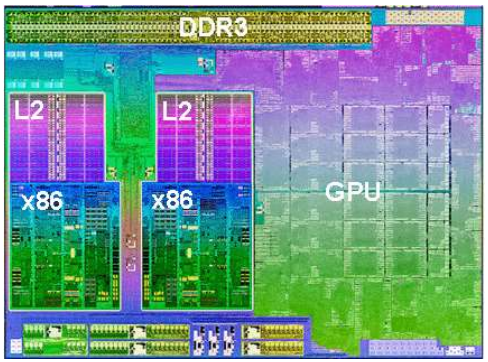
- Good platform for today's CS problems
  - If you are interested in doing ML, you will use GPUs!
  - Geoff Hinton - co-inventor of back propagation algorithm used for training multi-level neural networks
    - Quoted as saying GPUs are perfect match for deep belief networks (startup bought by Google in 2013)
  - Applications: self-driving cars, augmented reality...
- Great example of HW/SW co-design
  - New software to express parallelism
  - Hardware designers can put resources to best use

When To Use GPUs?

- Dependent on your program...
  - They aren't magic!
- Remember fundamental trade-off
  - Multithreading **always** reduces single-thread perf.
  - GPUs take that to the extreme
- GPUs run **regularly** parallel programs well
  - Very common for physics simulation, machine learning
  - Irregular parallelism with synchronization more difficult
    - System is not built for that, but improving every year

Overview of GPU Architecture

- Essentially high-performance coprocessors
- Attached to the system in a few different ways
  - Integrated onto chip with unified memory
    - Cache-coherent with CPU (e.g., AMD)
  - Discrete device, connected through I/O
    - PCI Express (PCIe) bus



Integrated

We use these!

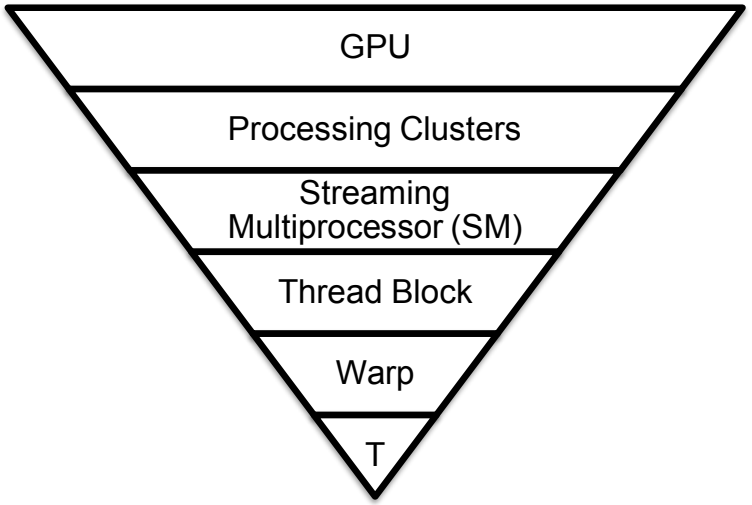
Discrete

GPU Architecture Hierarchy

- Note: we use NVIDIA terminology in this class

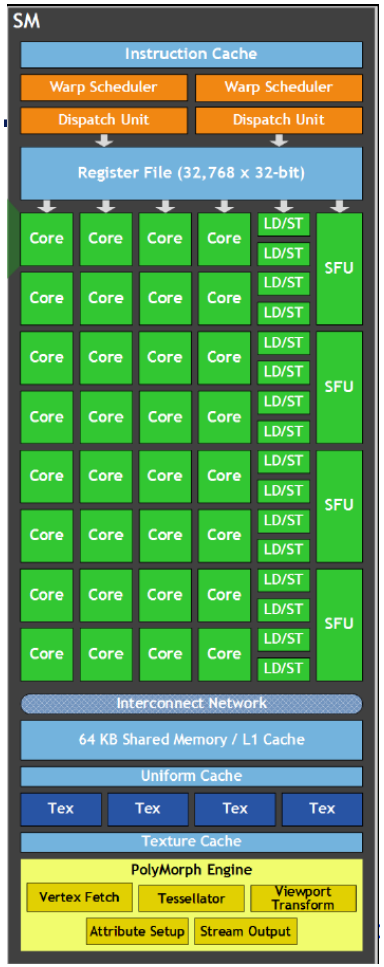
  (because those are the GPUs you will use)

- Modern GPU (V100)
  - 6 processing clusters
  - 14 SMs per cluster
  - 32 thread blocks per SM
  - 32 threads per warp



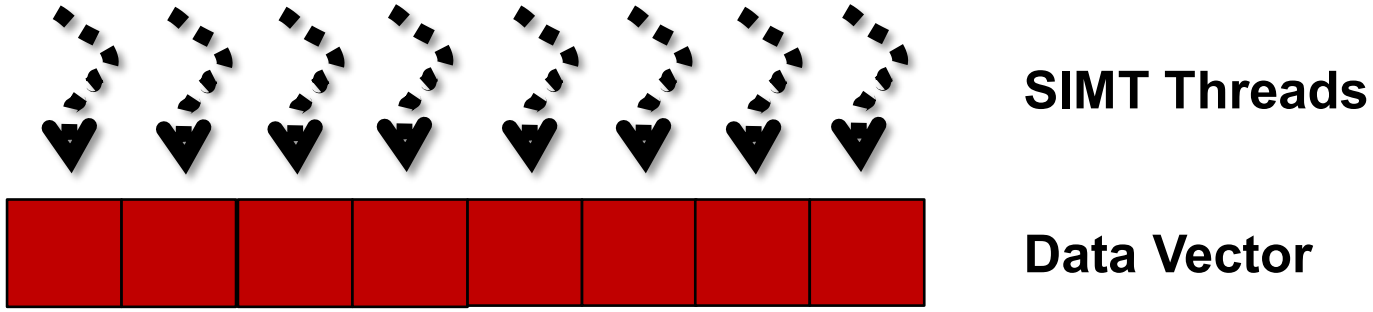
Streaming Multiprocessor (SM)

- NVIDIA's equivalent of a processor
  - Broken down into:
    - Front-end (i-cache and scheduler)
    - Registers
    - Execution units/ALUs
    - Memory
- Wide execution path, all threads execute on all cores



Parallelism Model - SIMT

- SIMT = Single Instruction Multiple Thread
  - Remember vector processors
  - GPU threads act like the ALUs inside each vector slot
- SIMT is the combination of SIMD operations, with scheduling in round robin fashion



Definition of a Thread

- What's a thread for you?
  - On a CPU, a thread has its own execution context
    - Instruction stream, stack, program counter
- On a GPU, this is **not true**
  - All threads share an execution context (including stack)
  - All threads in a program have the same instruction stream

Constructing Warps

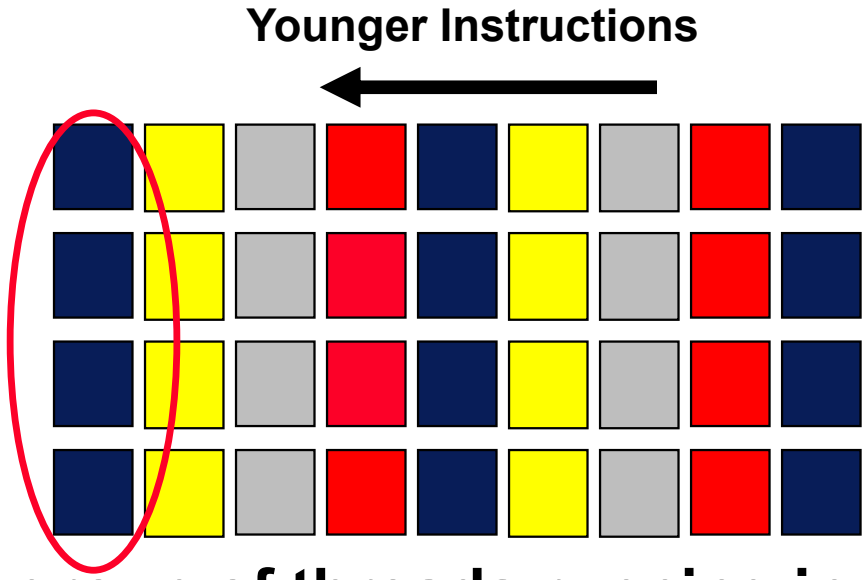
- A warp is simple: 32 threads running together
  - Natural choice because the pipeline is 32-wide SIMT
- Choice made to simplify hardware scheduling
  - Otherwise, SMs track thousands of threads
    - Current SMs allow 48 warps per SM, so 1536 CUDA threads
  - Remember what we learned about SMT
    - Tracking memory/register readiness for 1k threads is hard

Important Corollary of Using Warps

- Since all threads of a warp execute together
  - They all have to be ready, or can't execute!!!
- If one thread out of 32 is waiting... (e.g., memory)
  - SM hardware will not schedule the warp
  - **Huge vertical pipeline waste**
- H/W solves it with more warps to hide latency
  - But, better S/W solution, cover next lectures...
  - Also, talk about thread divergence (similar problem)

Multithreading in GPUs

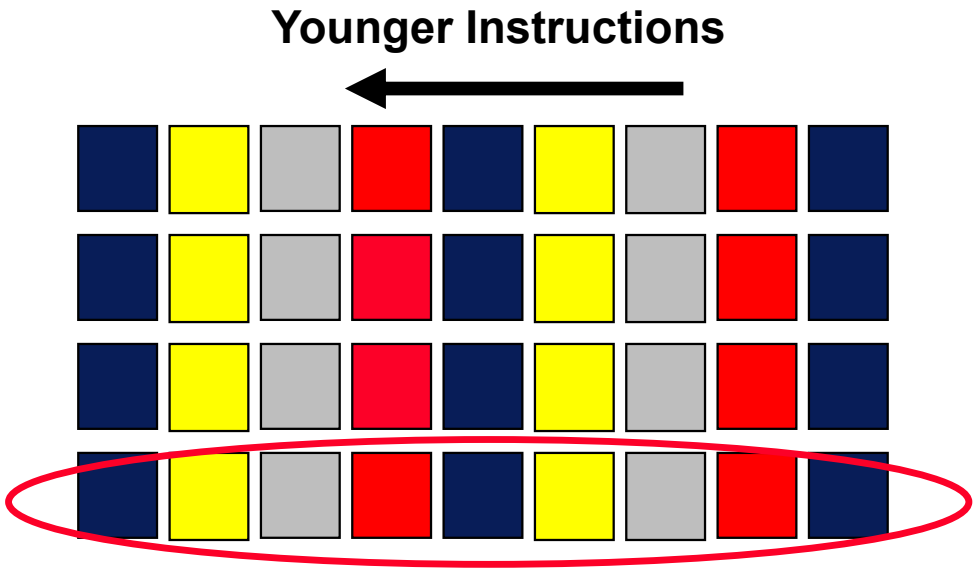
- GPUs use FGMT to hide long latency
  - Last lectures, the "thread" was a large CPU thread
  - This time, the "threads" are simply different warps



Warp: group of threads running in lockstep

Multithreading in GPUs

- GPUs use FGMT to hide long latency
  - Last lecture, the ”thread” was a large CPU thread
  - This time, the “threads” are simply different warps

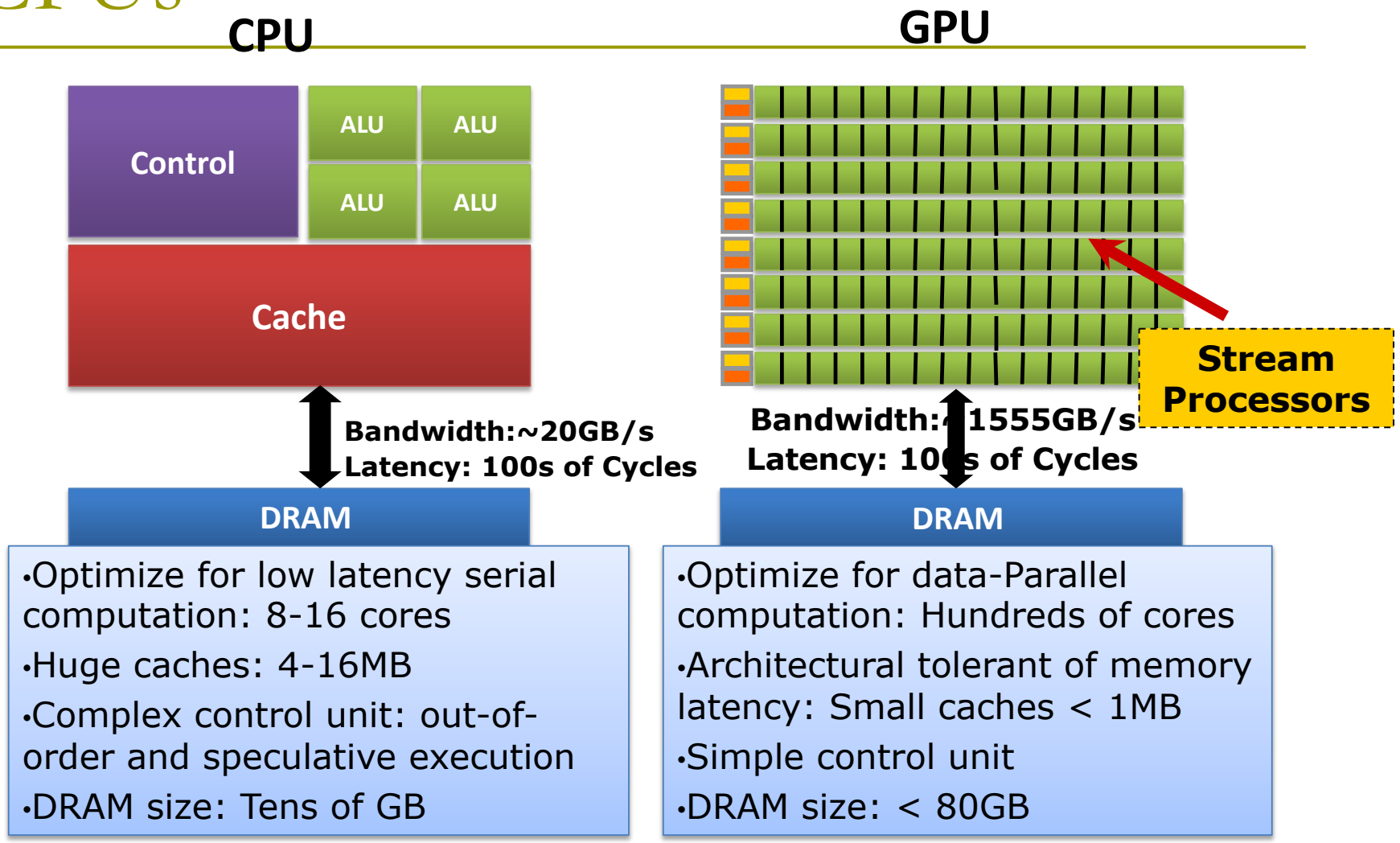


Multiple warps share pipeline depth

Allocating Warps to SMs

- Recall:
  - Modern GPU has ~80 SMs, max 48 warps each
- Two levels of resource allocation
- Global allocation maps groups of warps to cores
  - "Groups of warps" = Thread Blocks (stay tuned!)
  - Allocates registers in cores
- Instruction scheduler chooses warps to issue
  - Find warps with independent instructions

Why GPUs are so Fast: GPUs vs. CPUs



Fermi vs. Kepler GPUs

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Courtesy NVIDIA