Introduction to GPUs

EE379K - Architectures for Big Data Sciences
Dr. Sriram Vishwanath
Department of Electrical and Computer Engineering
The University of Texas at Austin
Spring 2016

GPUs

- **G**raphics **P**rocessing **U**nits
- Multi-processors that are tuned for graphics
- GPUs excel at **data-parallel** applications
  - Graphics
  - Computer vision
  - Many machine learning algorithms (k-means, deep learning, etc)

(A Short) History of GPUs

The Early Days (1976 - 1990s)

- The first “GPUs” were just display controllers. They took in image data from the CPU and converted it to the proper display format
- They also handled display syncing (VSYNC)

IBM Monochrome Display Adapter (1981)

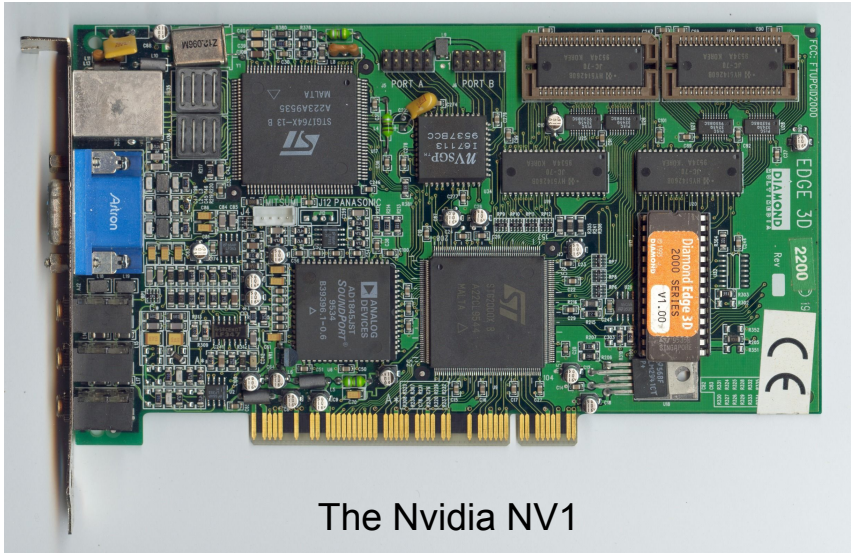
The Early Days (1980s)

- Intel's iSBX 275 card introduced multi-color support (8 colors) and a 256x256 resolution
- 1985 - ATI was formed and they launch their Color Emulation Card, which helps them break $10million in first year sales

ATI EGA800: One of the first VGA cards

The Early Days (1990s)

- 1993 - Nvidia is formed
- 1995 - Nvidia launches the NV1, which is capable of 3D rendering, video acceleration, and GUI acceleration



The Nvidia NV1

The Early Days (1990s)

- Voodoo Graphics develops a GPU, the 3Dfx, that outperforms all other competitors
- Tile-based rendering technology developed
- The graphics space becomes very crowded. However, Silicon Graphics (SGI), ATI, 3DFx, and Nvidia stand out.

The NVIDIA and ATI Era (2000 - 2008)

- 3Dfx bought out by Nvidia
- 2005 - Nvidia launches the 6800Ultra, introducing SLI for multi-GPU configurations
- 2006 - ATI bought by AMD
- 2006 - Nvidia launches its first dual-GPU card, the 7900 GX2

The Nvidia 7900 GX2

The General Purpose GPU Era (2008+)

- 2007 - AMD and NVIDIA begin pushing general purpose computing on GPUs. NVIDIA launches CUDA, AMD begins work on OpenCL
- 2012 - AMD launches the 7000 series GPUs, which eventually become extremely popular for Bitcoin mining.
- 2014+ - GPGPU becomes largely mainstream with GPUs being used for many non-graphics problems.

GPUs used today for Machine Learning

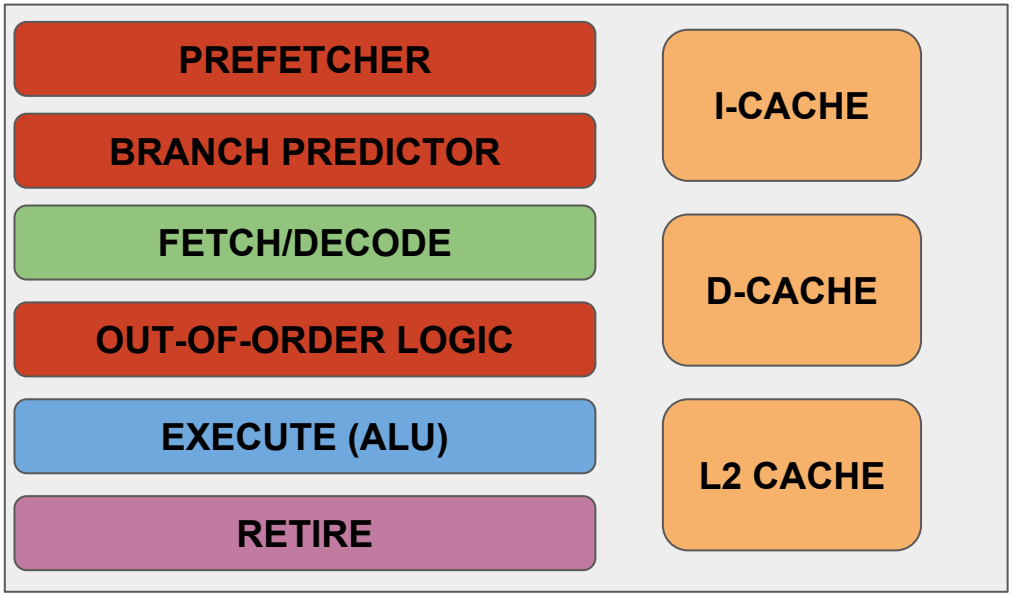
- NVIDIA Tesla
- 8 Gb RAM and 3400 cores
- Used for medium-sized problems:
  - recursive neural networks
  - deep learning
- NVIDIA's software platform, CUDA, enables you to leverage your own GPU

GPU Architecture

Key Concepts

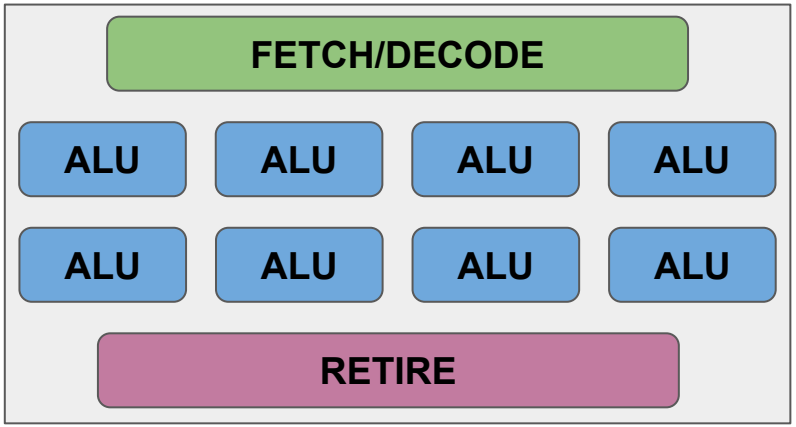
1. Use **many** low-performance cores instead of a few high-performance cores (latency vs bandwidth)
2. **Share** instruction streams across many cores. Pack cores with many ALUs for high compute performance.
3. Interleave execution of many threads to hide latency stalls

Traditional CPU Core



- Designed to minimize instruction **latency**
- Lots of logic structures (highlighted in red) need to make this happen
- Extremely power-hungry

Traditional GPU Core



- Designed to maximize instruction **throughput**
- Only uses the bare minimum logic
- Lots of ALUs
- Power efficient, so we can have many of these cores in parallel

Tradeoffs

- CPUs are designed to execute a single program very fast
    a. Minimize clock cycles / instruction
- GPUs are designed to execute **many parallel** instruction streams
    a. Maximize instructions retired / cycle
- **Tradeoff:** instruction latency vs instruction throughput
- However, to achieve high throughput you need many threads

Graphics

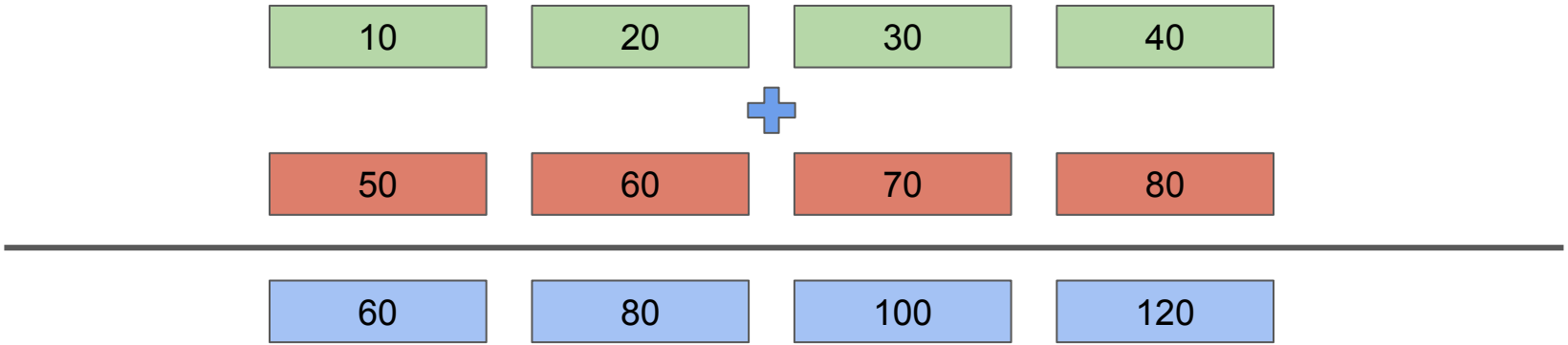
- The goal of graphics is to assign a color to each pixel
- However, pixels are **independent** of each other
- Thus, we can create a thread **per-pixel** and run all of them in parallel (2 million threads for a 1920x1080 display)
- Graphics is a natural use case for these processors (hence, GPU)

Types of Parallelism

- You can split up parallelism into two key components:
- How many instructions are you running?
- And, how much data is each instruction operating on?

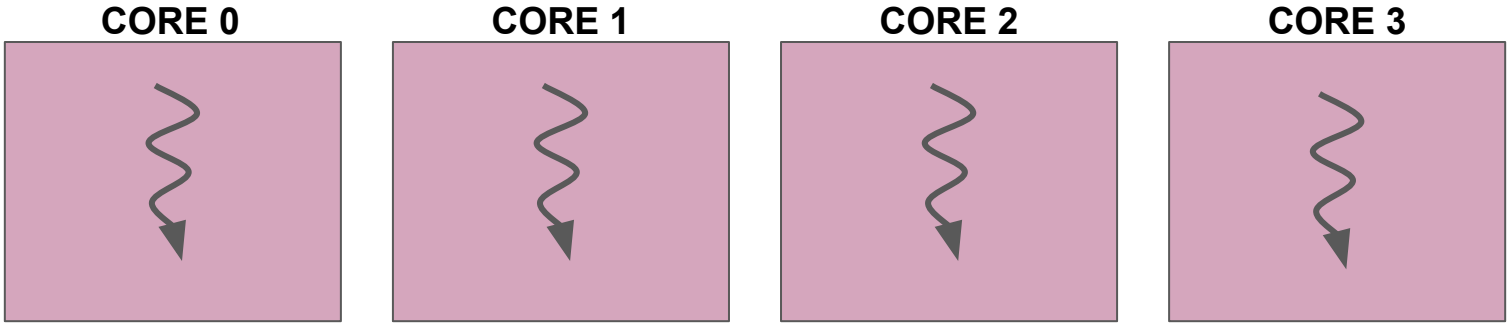
Types of Parallelism: Flynn's Taxonomy

- Single Instruction Single Data (SISD)
  a. A typical processor core. One thread of execution. Single set of operands.
- Single Instruction Multiple Data (SIMD)



Types of Parallelism: Flynn's Taxonomy

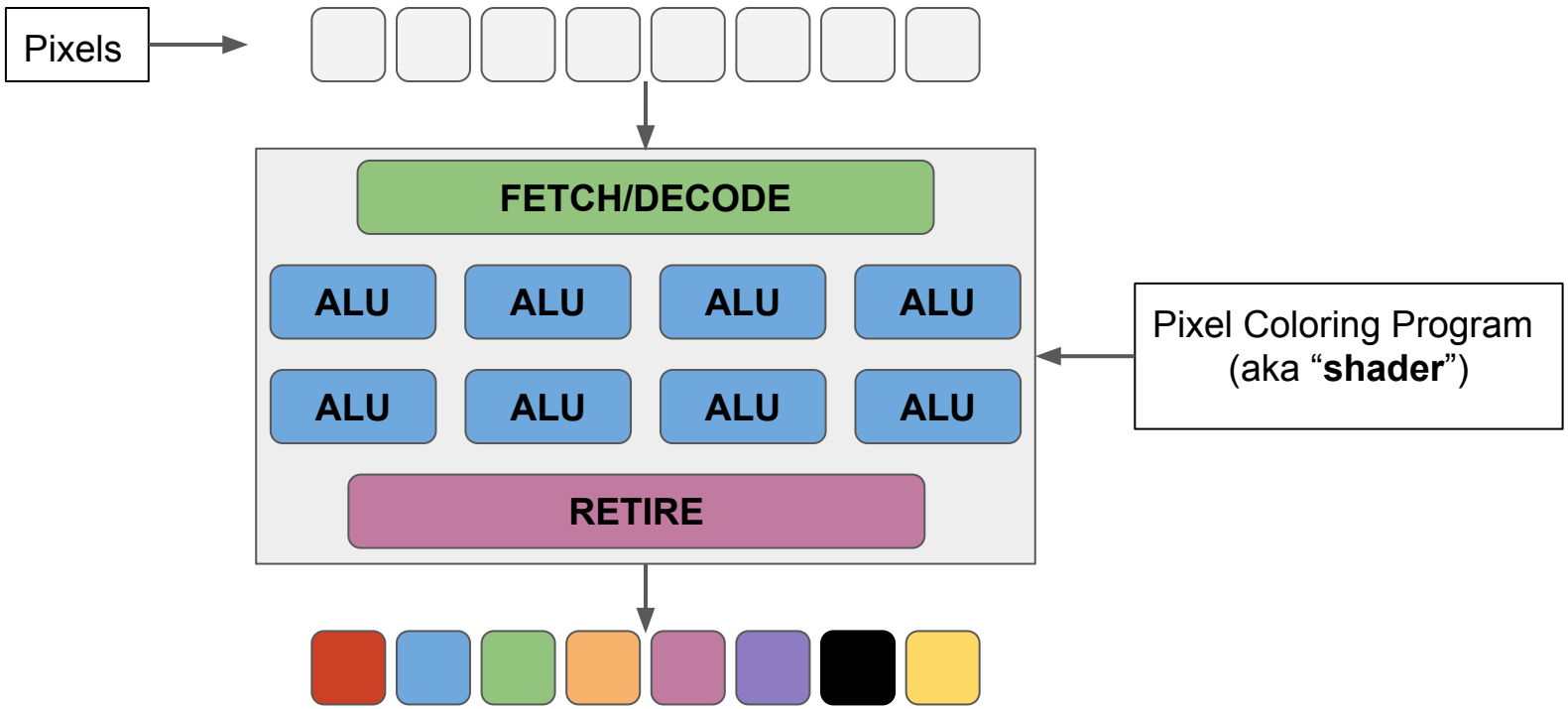
- Multiple Instruction Single Data (MISD)
  a. Multiple instructions operating on the same data. Not really used.
- Multiple Instruction Multiple Data (MIMD)
  a. Multi-core processor. Each core runs a completely separate program.



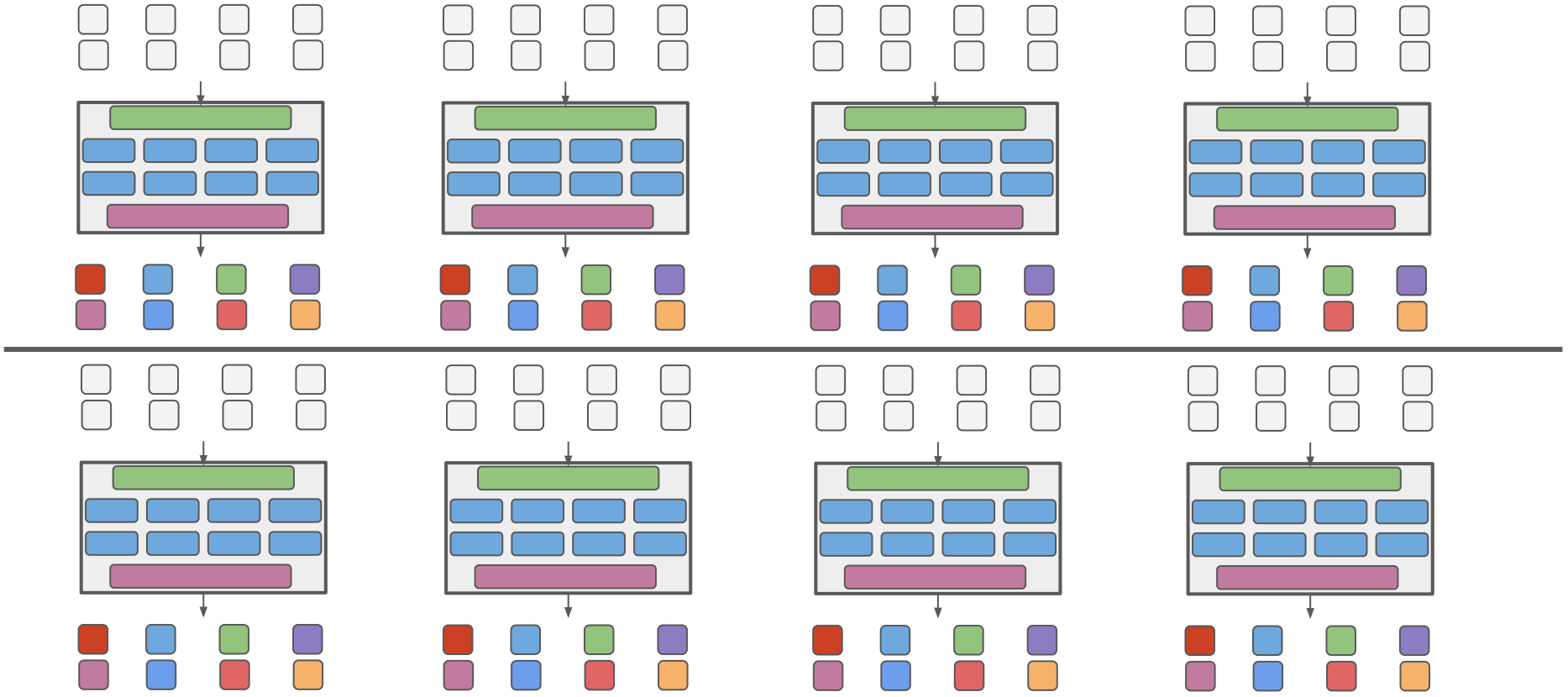
Types of Parallelism: SIMT

- GPUs are an extension of the SIMD paradigm
- Some call them “SIMT” processors
    a. Single instruction multiple threads
- In SIMD, multiple data operands share an instruction
- In SIMT, multiple **threads** share instructions
    a. Each thread has its own separate data, register file, and flow path

Example: Coloring 8 Pixels Using 1 Core



Example: Coloring 64 Pixels Using 8 Cores



Applications

To GPU or not to GPU

- The GPU is designed to address applications that are **data-parallel**
- Parallelism is an inherent factor to determine suitability of a problem for GPU applications
- In fact, applications in which enough parallelism cannot be exposed may be slower on a GPU in comparison to a single threaded CPU
- Since the same program is executed for each data element, there is **no sophisticated flow control**

Latency vs Bandwidth

- CPUs are better when you need to run a single program really fast
- GPUs are better when you have a lot of independent threads that can run in parallel
- CPU + GPU systems are called **heterogeneous systems**

GPGPU

- Initially GPUs were only used for graphics
- However, people realized that GPUs are well suited for:
  a. Math intensive applications like matrix multiplies and convolutions
  b. Many parallel and multithreaded algorithms
- Thus, GPUs are now being used a lot for non-graphics purposes
- This is known as “General Purpose GPUs” or GPGPU

GPU Application: Rendering

- Each pixel is an independent thread of execution

GPU Application: Path Tracing

- Path tracing is an advanced rendering technique which simulates light rays bouncing in an environment
- Each light ray is independent of every other light ray.
- We can “trace” the path of each light ray in parallel

The Cornell Box

(Example of path tracing)

GPGPU Application: K-means

- In k-means, we need to calculate the distance from each point to every centroid
- Distances are independent of each other
- Have a thread per point which calculates closest centroid

GPGPU Application: K-means

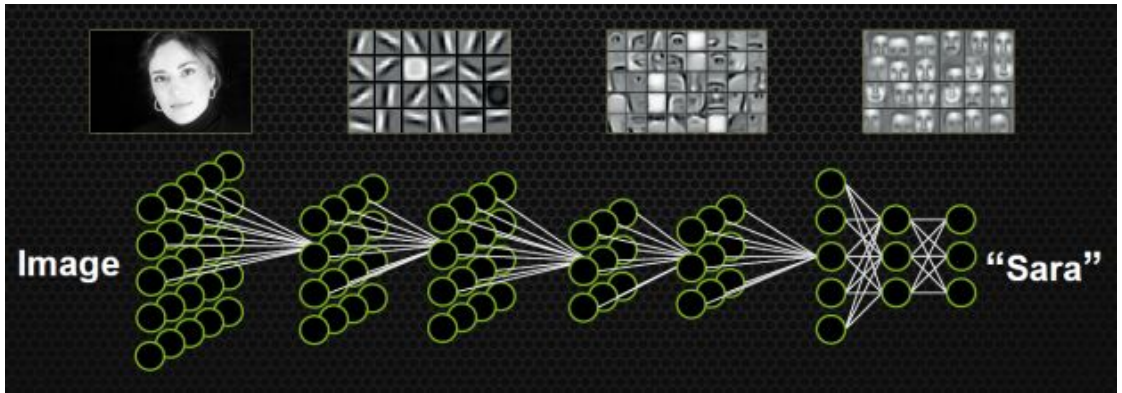
- **Cluster membership assignment** done on GPU:
- **Centroid recalculation** is then done on CPU
- Most appropriate for processing ***dense*** data

Example K-Means Performance Results [Jain et al.]

- 4 million 8-dimensional vectors
- 400 clusters
- 50 *k*-means iterations
- **9 seconds!!!**

GPGPU Application: Deep Learning

- In a deep net, each neuron independently performs similar computation for different input data
- There are also many matrix multiplies and convolutions involved
- GPUs can accelerate both of these tasks



GPU or not to GPU?

Many scenarios where GPUs are not as useful: when things are by nature serial

1. Communications & networking algorithms and applications: tend to be serial
2. Control and networked decision making
3. Biochemical pathways etc

Conclusion

- GPUs tradeoff instruction latency for throughput
- GPU architecture is all about:
  - How to run many threads efficiently
  - How to hide latency
- GPUs can accelerate machine learning algorithms:
  - Clustering
  - Deep learning

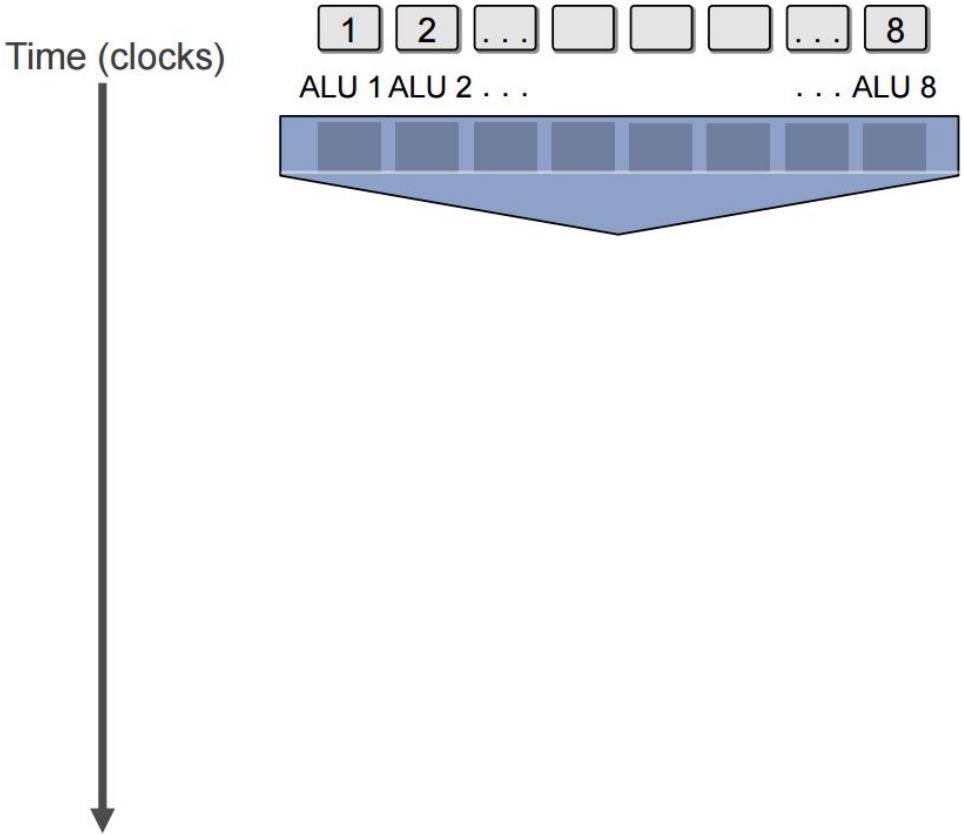
GPU Architecture Problems

Branches

- In GPU cores, a single thread is working on multiple data operands.
- All ALUs must follow the same thread
- On a branch, some ALUs will be on the right path and some will be on the wrong path.
- The results of all the wrong path ALUs must be thrown away!

Example from: http://haifux.org/lectures/267/Introduction-to-GPUs.pdf

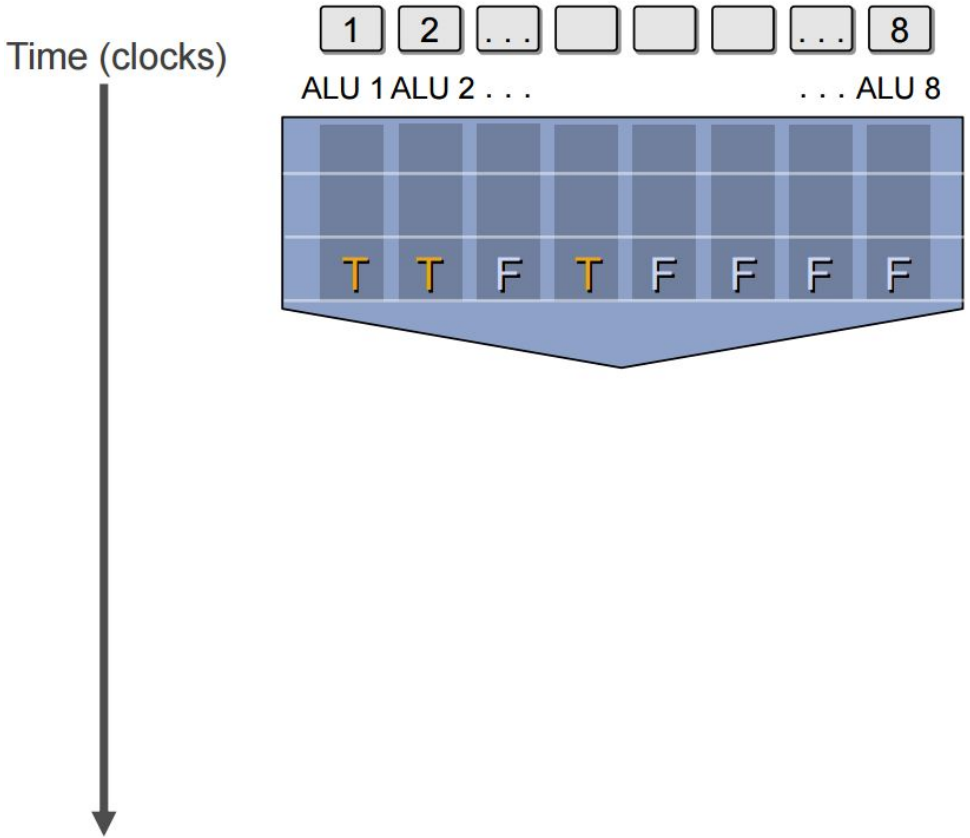
Branches




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

Example from: http://haifux.org/lectures/267/Introduction-to-GPUs.pdf

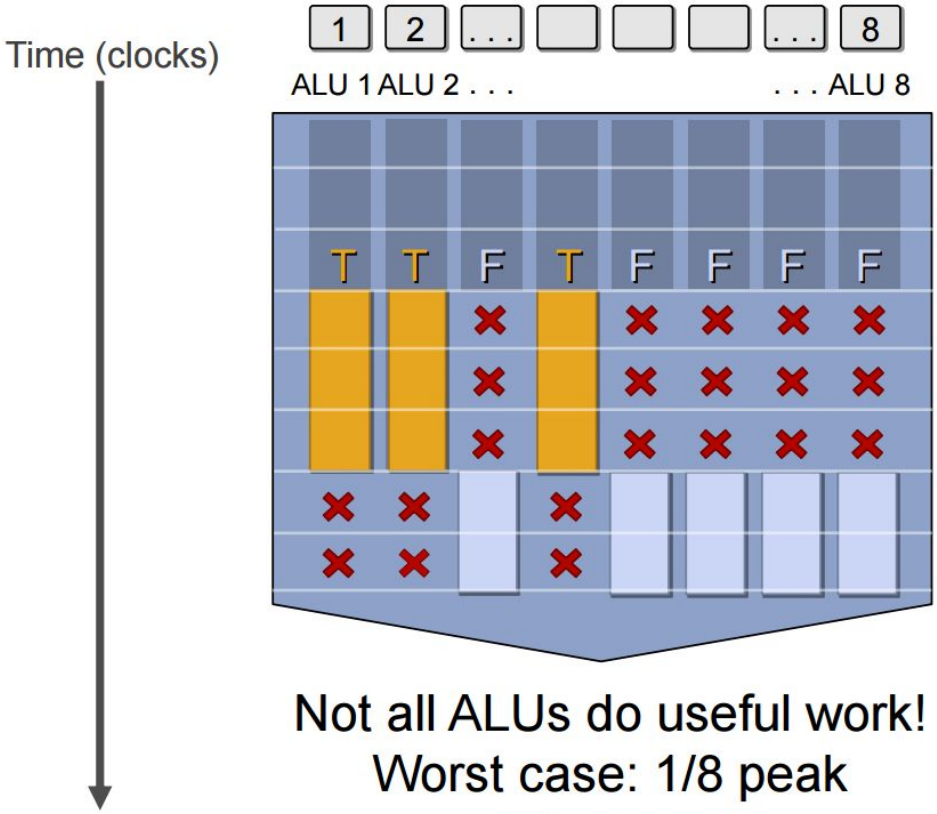
Branch Problems




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

Example from: http://haifux.org/lectures/267/Introduction-to-GPUs.pdf

Branch Problems



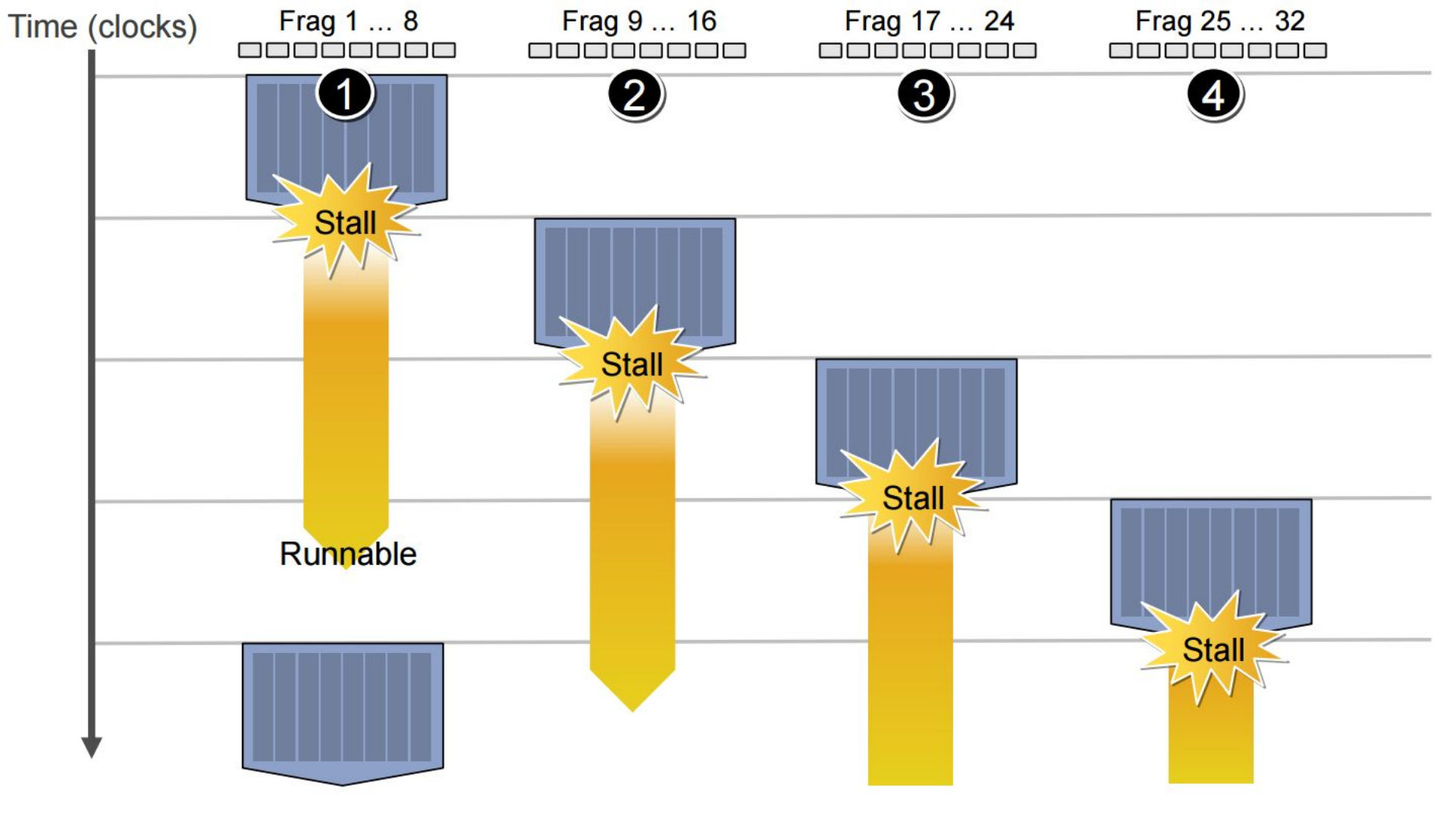

```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

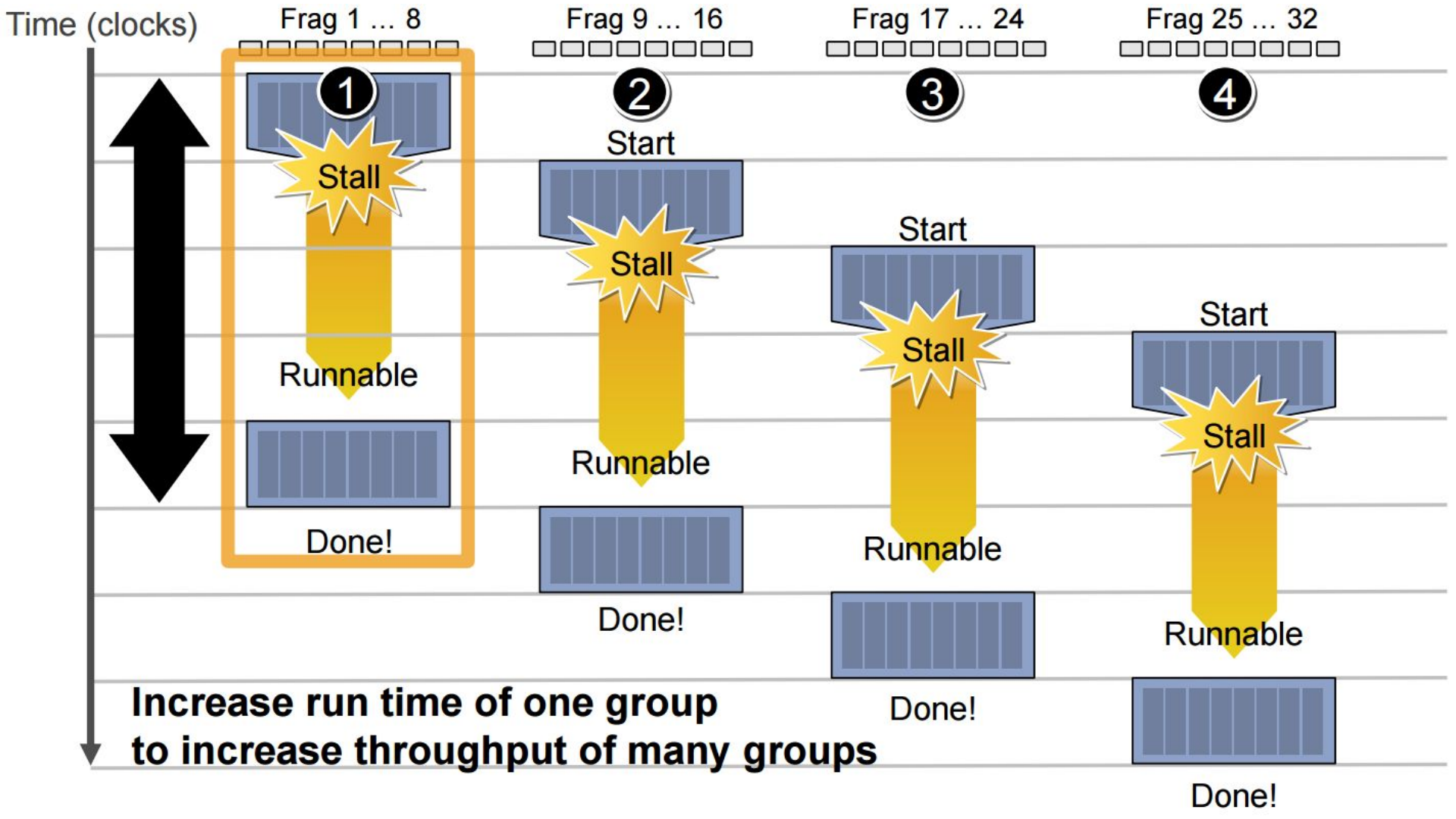
Stalls

- Stalls occur whenever a core cannot run the next instruction
- Could happen for a few reasons:
  a. Dependency on previous instruction
  b. Memory access latency (can be 100s of cycles)
- We don't have caches or out-of-order execution to help us avoid these stalls
- What can we do?

Interleaved Execution

- We have a lot of threads just waiting to be executed
- When a running thread stalls, swap in a waiting thread
- This keeps all cores active


Time (clocks)
Frag 1 … 8
Frag 9 … 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Stall
Stall
Stall
Runnable


Time (clocks)
Frag 1 … 8
Frag 9 … 16
Frag 17 … 24
Frag 25 … 32
1
2
3
4
Stall
Runnable
Done!
Start
Stall
Runnable
Done!
Start
Stall
Runnable
Done!
Start
Stall
Runnable
Done!
Increase run time of one group
to increase throughput of many groups