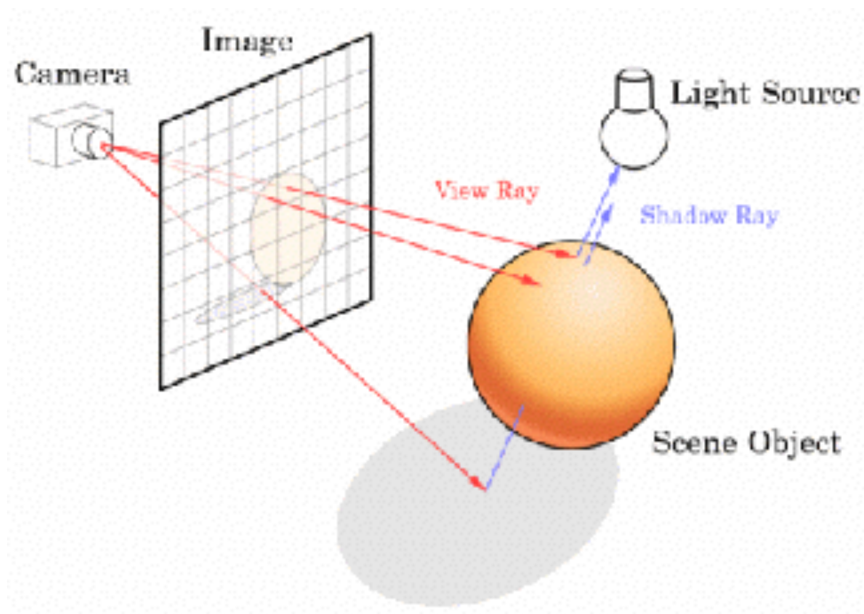
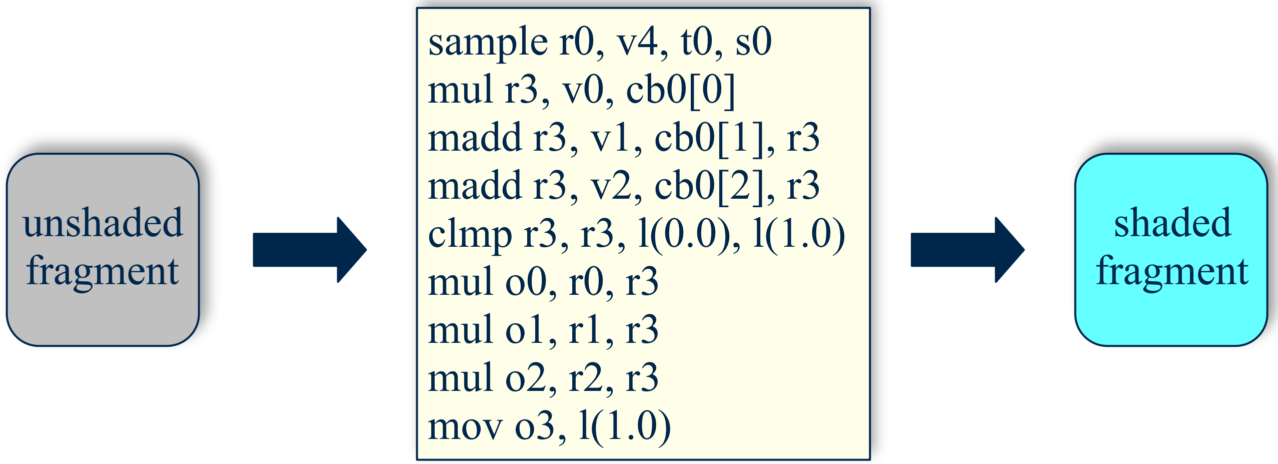
GPUs

Why GPUs?

In order to render a scene, we must determine the color assigned to each pixel (usually based on light transport)



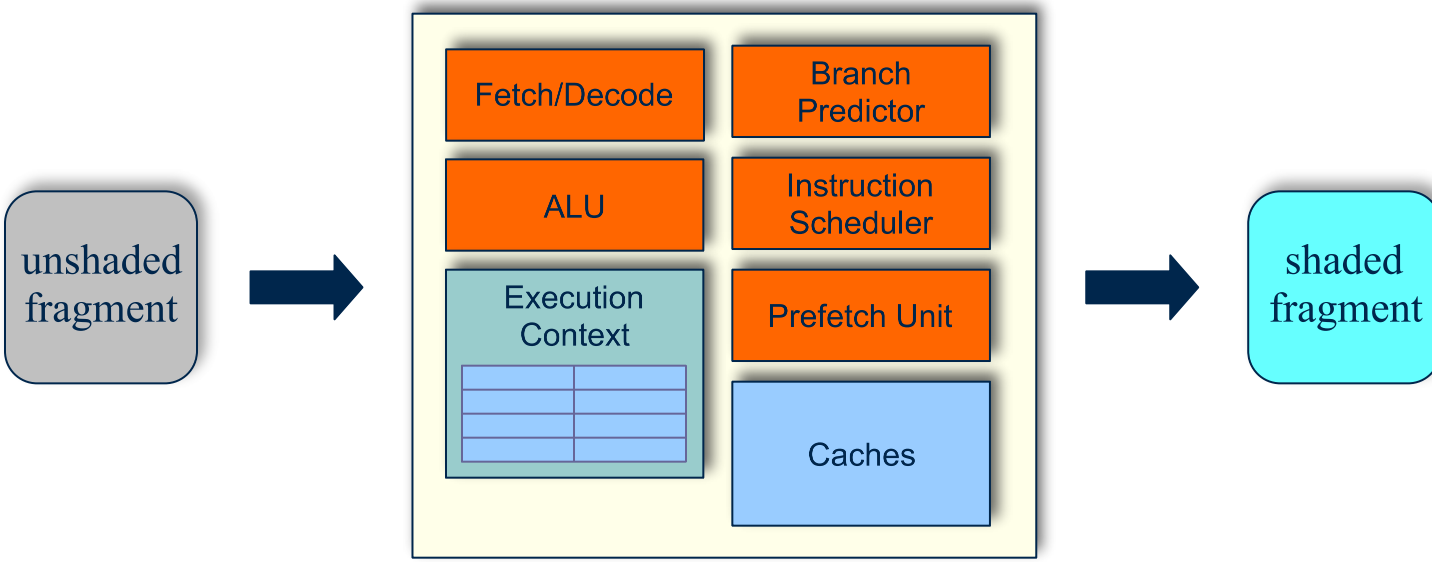
Work Per Fragment



Fixed work per fragment

Ideally process several hundred thousands of these at 60Hz

Working on the CPU



CPU is big and complex but fast on a single thread

…But even a really fast thread isn't sufficient for shader execution…

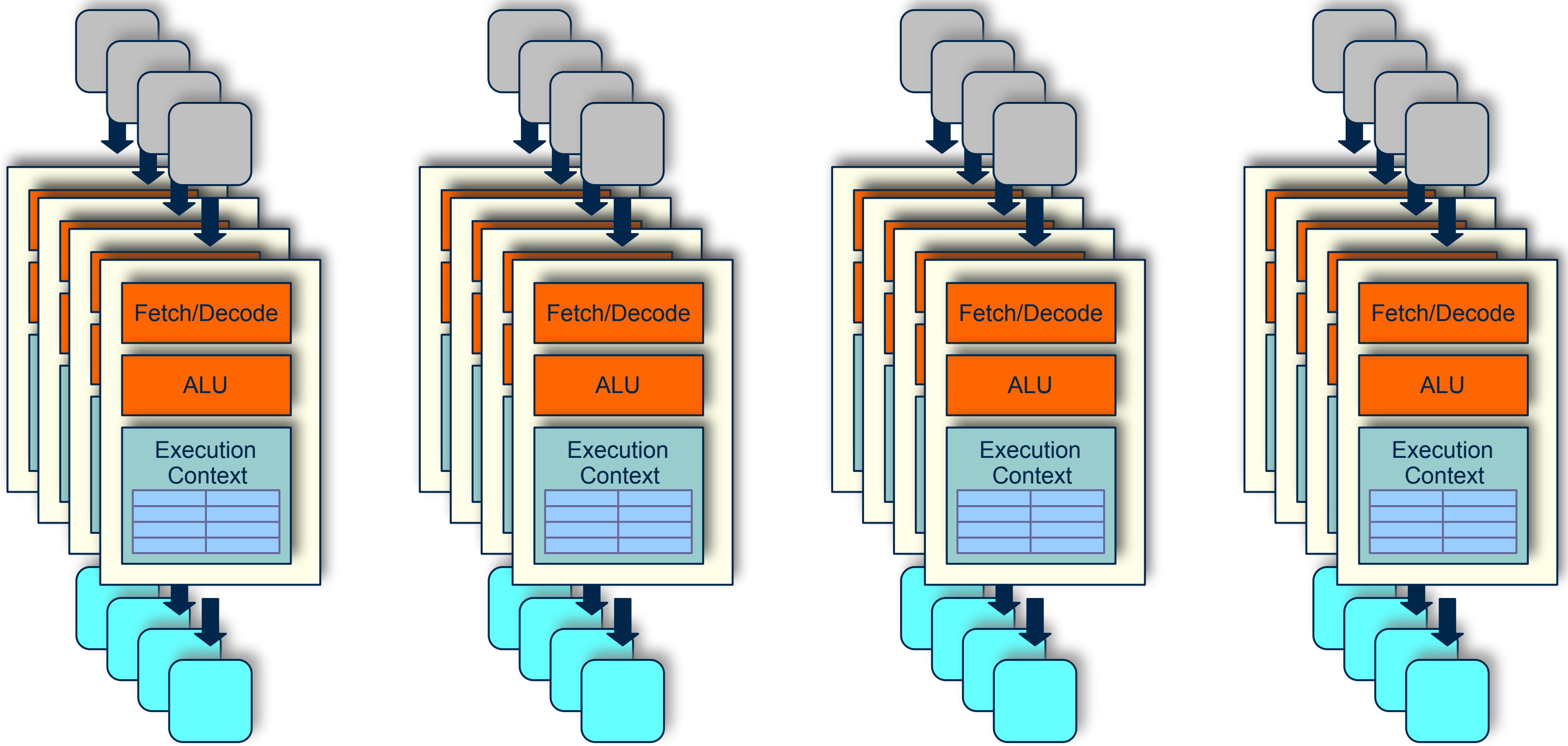
Graphics Processing Unit

Built for rendering pipeline

- Process large number of vertices
- Assumes similar, relatively simple, operations

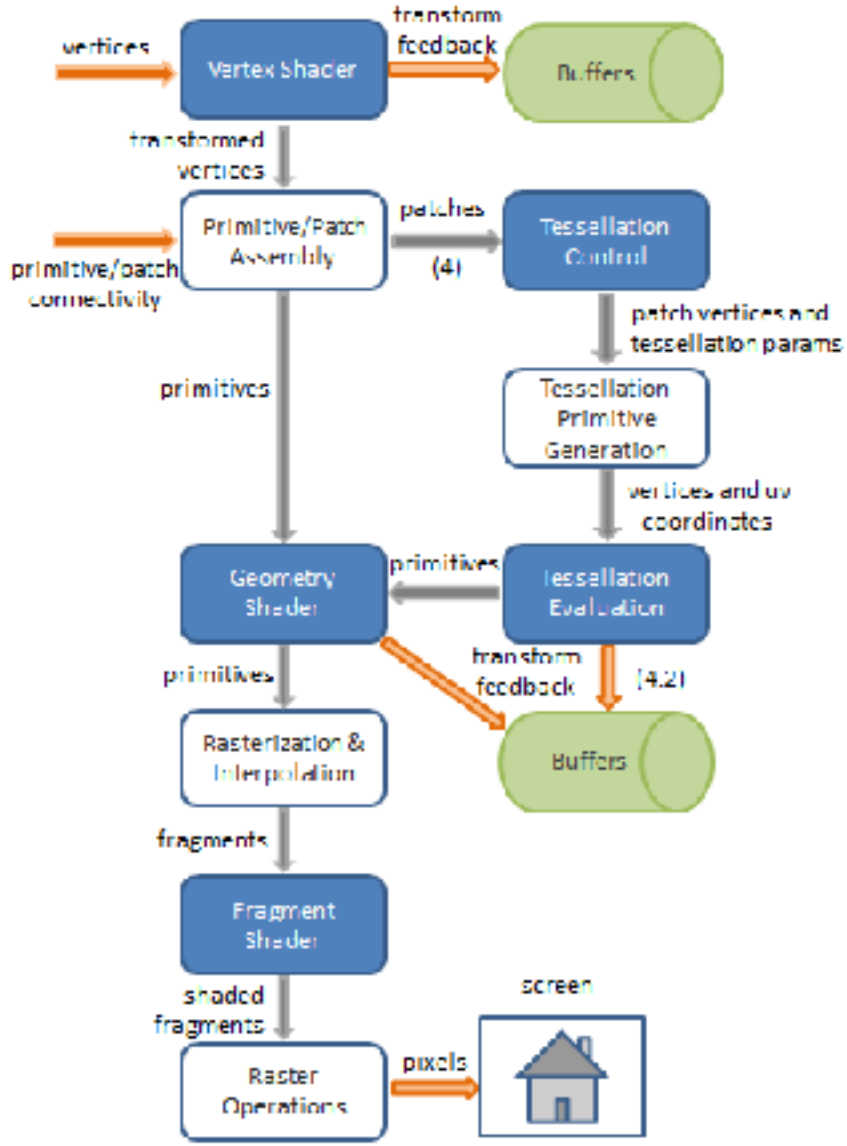
What sort of architecture facilitates this?

Throughput Architecture

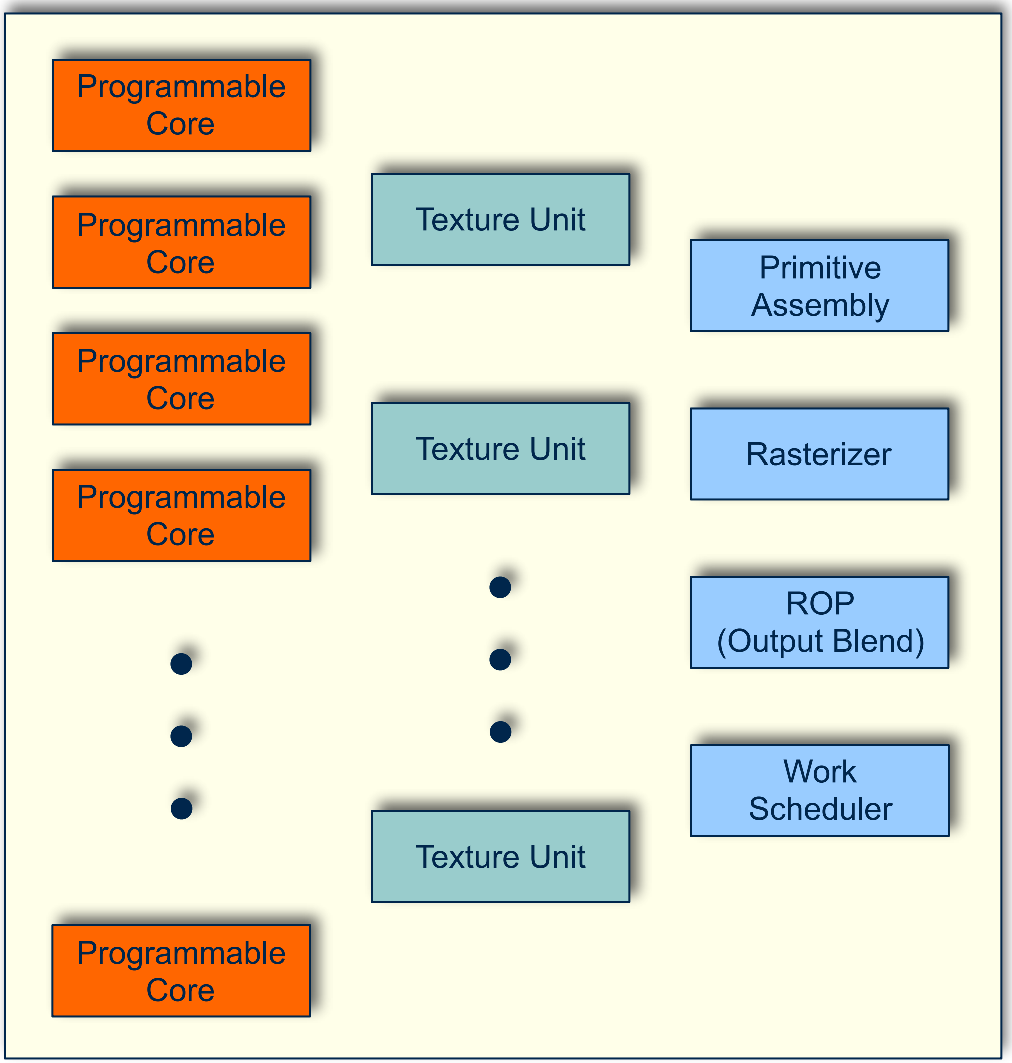


Simpler cores but lots of them in parallel!

Remember the Rendering Pipeline?



Modern GPU Characteristics

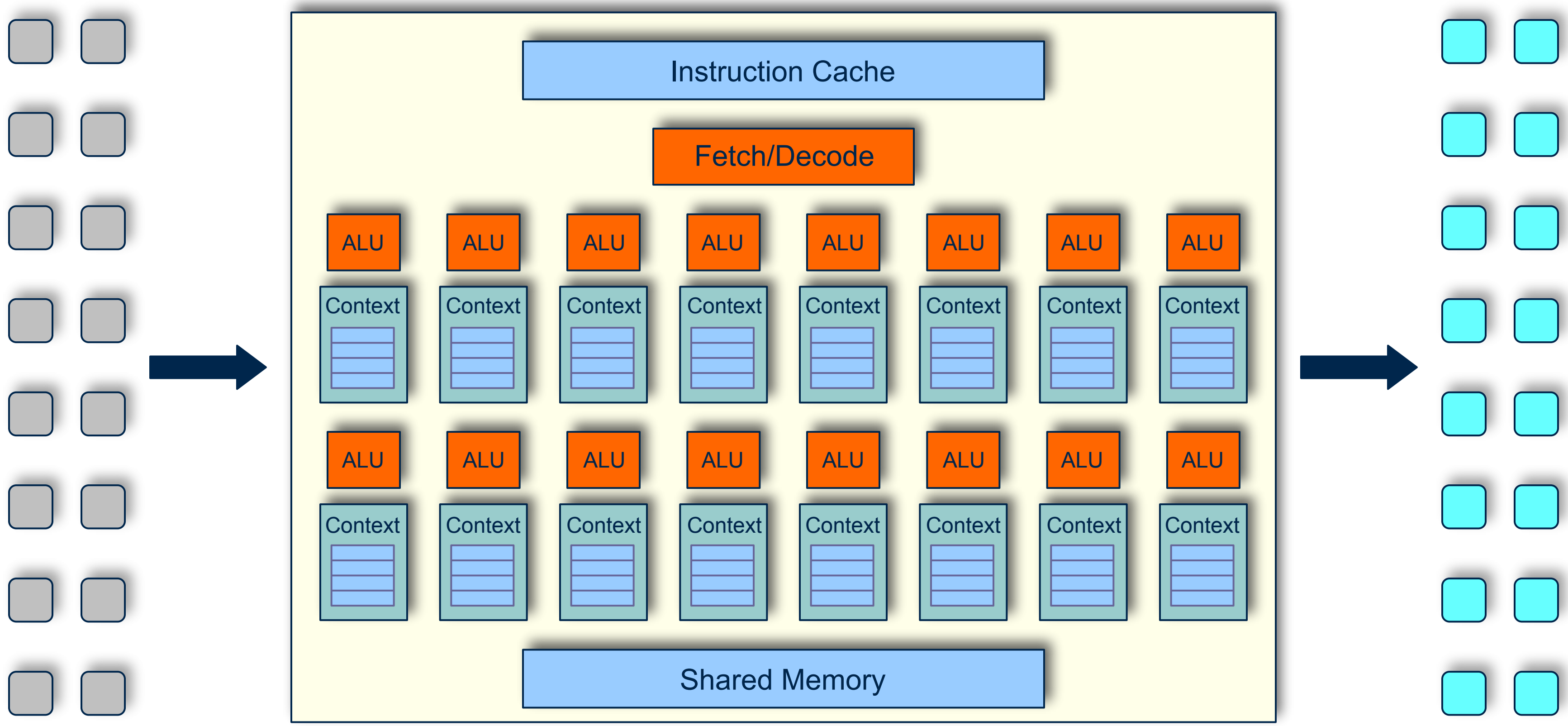


- Homogeneous programmable cores for all programmable stages
- Relatively few special purpose texture units
- Even fewer fixed function units
- Task parallel at pipeline level

SIMD

- Single instruction, multiple data
- Large vectors of data that have the same operation applied to individual elements in parallel
- Based on old super computing techniques but has regained popularity in modern architectures (both CPU and GPU)

Shared Instructions



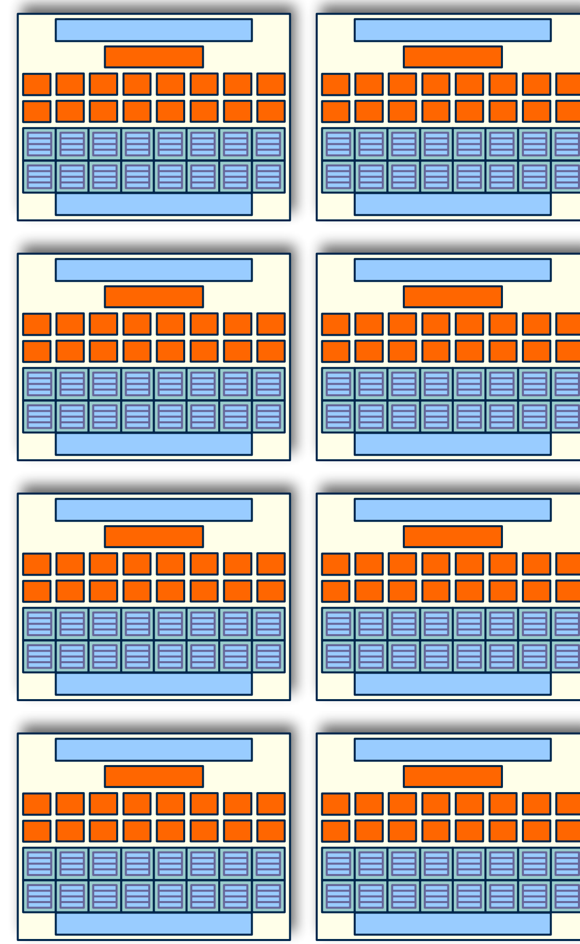
- Same thing is done in parallel for all fragments/verts/etc
- SIMD amortizes instruction handling over multiple ALUs

Multiple Types of Processing

GPUs do more than shading

- Allow execution of more than one program

Replicate SIMD processors for different SIMD computations in parallel

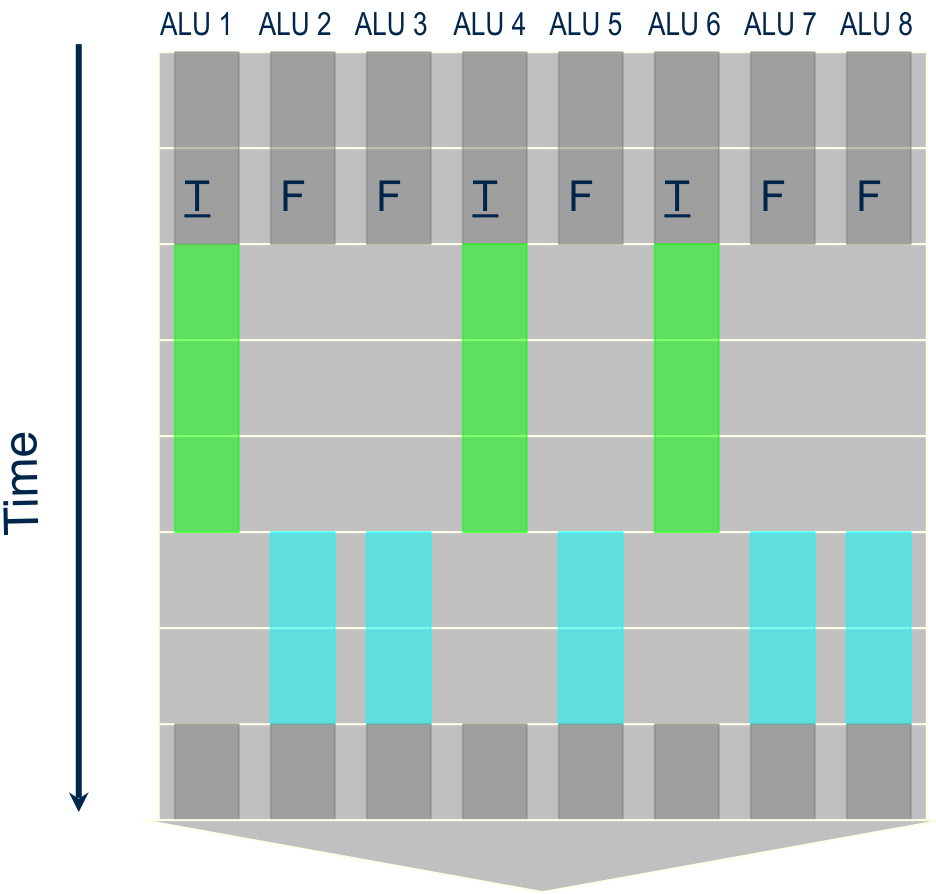


8 programs running in parallel, 128 threads in parallel

Problems?

What situations does this throughput style of architecture not handle well?

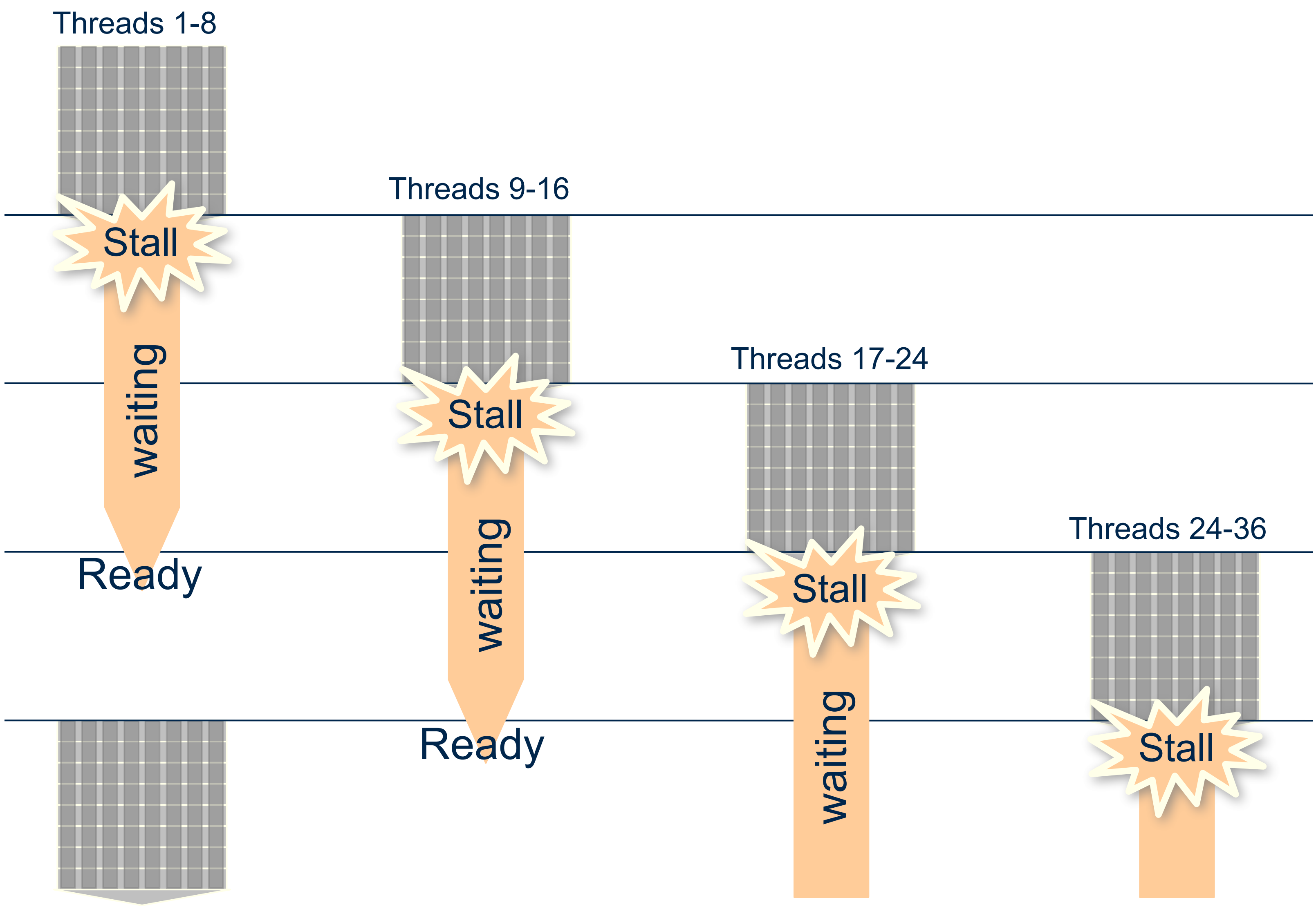
Branching and Stalling



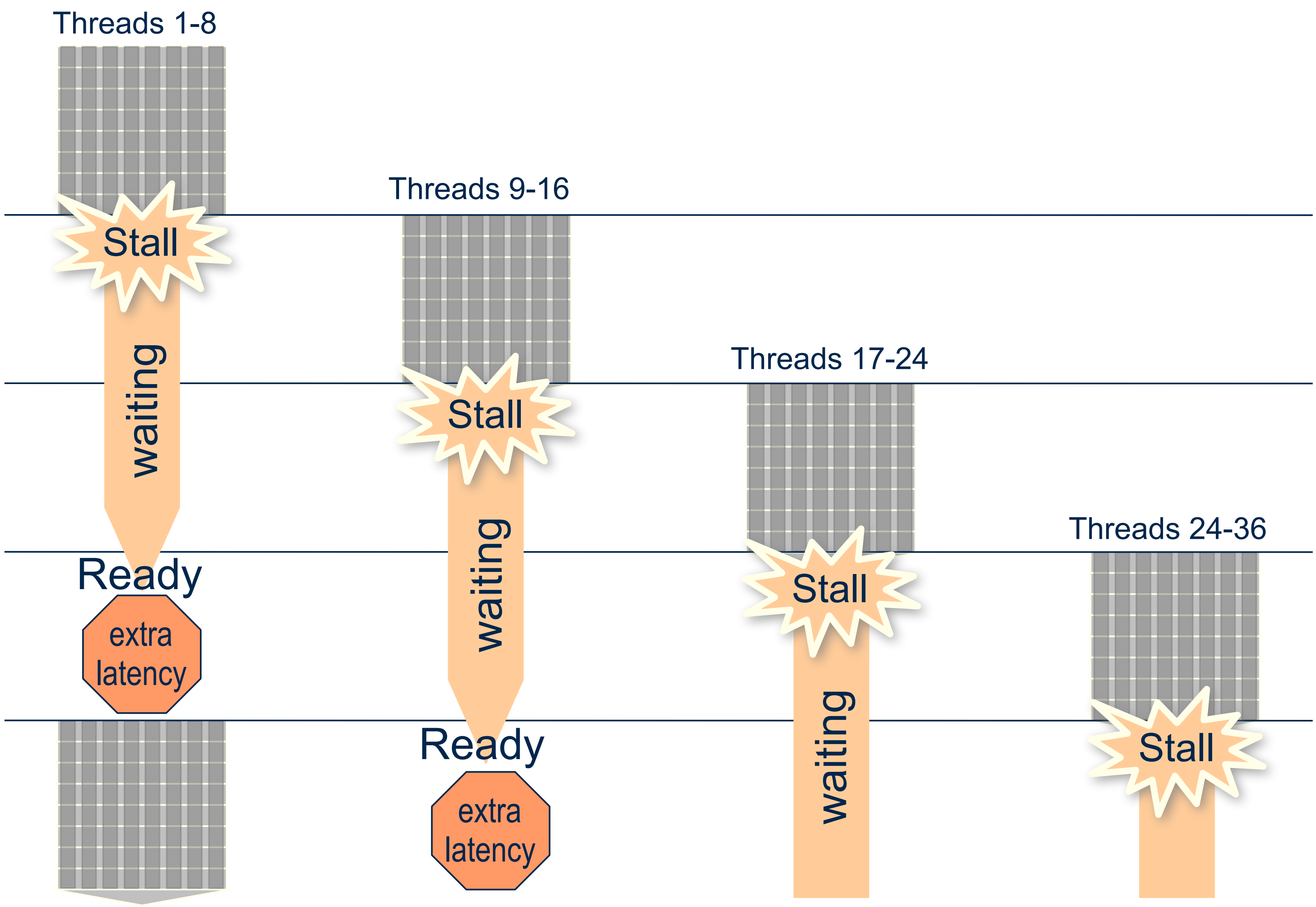
- Threads stall when next instruction depends on previous instruction's result
  - Pipeline dependencies
  - Memory latency
- How to handle these?

Multithreading

- We can assume there are more threads (scheduled computations) than processors
- Threads with similar code executed in “warps” to maintain minimal divergence
- Interleaving warp execution keeps hardware busy when an individual warp stalls



Threads 1-8
Stall
waiting
Ready
Threads 9-16
Stall
waiting
Ready
Threads 17-24
Stall
waiting
Threads 24-36
Stall


Threads 1-8
Stall
waiting
Ready
extra latency
Threads 9-16
Stall
waiting
Ready
extra latency
Threads 17-24
Stall
waiting
Threads 24-36
Stall

Working with Latency

- Latency hiding
  - Executing many warps can minimize latency (delay in processing)
- More context switching requires more storage (values in registers etc)

GPU Memory and Architecture

Designed for throughput, so bandwidth is critical

- Wide bus (150 GB/s+)
- High bandwidth DRAM organization
- Warp scheduling for latency hiding
- Small execution contexts and efficient local memory
- Limited cache hierarchy

Example: Pascal Architecture

Global and Shared Memory

Global memory scoped for entire program

- Functions like a heap
- Slowest (on device) access
- Good access patterns minimize cache touches

Shared memory located on chip

- Scoped to block
- Very fast and localized
- Good access patterns minimize bank accesses by different threads

Local Memory and Registers

Local memory is scoped to a thread

- Includes everything that does not fit onto registers
- Registers are very fast, so spilling into local memory leads to slowdowns

Programming on the GPU

The programmable shader pipeline is highly specific to rendering.

Idea: Create a language that can harness GPU throughput with more accessible programming paradigms

GPGPUs

- Solve non-graphics problems on GPUs
  - Textures act as memory
  - Compute shaders allow for small, highly parallel executions
  - Methods like map, reduce, scatter, gather, etc provided for convenience
- Languages like CUDA and OpenCL facilitate development

CUDA Example

main function runs on host (CPU)

- Allocates memory on host and device global memory

kernels that run on device (GPU) specified with __global__

- Functions treated much like standard C functions

CUDA Example: SAXPY

https://devblogs.nvidia.com/easy-introduction-cuda-c-and-c/

For every 2d vector (*x*, *y*), multiply constant *a* times *x*, then add *y*

Easily parallelized and simple algorithm

Host Code

```
int main(void) {
    /* Allocate variables here */

    //Allocate memory on host
    x = (float*)malloc(N*sizeof(float));
    y = (float*)malloc(N*sizeof(float));

    //Allocate memory on device
    cudaMalloc(&d_x, N*sizeof(float));
    cudaMalloc(&d_y, N*sizeof(float));
```

```
    /* Initialize host array here */

    //Copy host array data to device
    cudaMemcpy(d_x, x, N*sizeof(float),
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, N*sizeof(float),
        cudaMemcpyHostToDevice);

    //Launch the device kernel on N+255/256 thread blocks with 256
        threads each
    saxpy<<<(N+255)/256, 256>>>(N, 2.0, d_x, d_y);

    //Clean up host and device memory
    cudaFree(d_x);
    cudaFree(d_y);
    free(x);
    free(y);
}
```

Device Code

```
__global__
void saxpy(int n, float a, float *x, float *y) {
    //Get global index into array
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    //Run saxpy
    if (i < n) y[i] = a*x[i] + y[i];
}
```

Note: blockIdx, blockDim, threadIdx predefined in CUDA

GPGPU Challenges

- Parallelization algorithms
- Memory for throughput architecture
- Work scheduling on throughput architecture
- Hiding latency

Toward Heterogeneous Architecture

Idea: CPUs are good at some things and GPUs are good at others. Why not have them closer together to get the best of both worlds?

- Already commonly used in embedded devices (e.g. system on a chip)
- Has attractive properties for general computing as well
- Also presents numerous software and hardware challenges at all levels of programming!