GPU Architecture and Programming

Erik Sintorn
Postdoc, Computer Graphics Group

Outline

Part 1: Motivation

If you're not into hardware, why should you care?

Part 2: History lesson

Where did GPUs come from, and what were they meant to do?

Part 3 GPU Architecture

What is a modern GPU, and how does it work?

Part 4: General Purpose GPU Programming

A little bit about non-gfx GPU programming

Outline

Part 1: Motivation

If you're not into hardware, why should you care?

Part 2: History lesson

Where did GPUs come from, and what were they meant to do?

Part 3 GPU Architecture

What is a modern GPU, and how does it work?

Part 4: General Purpose GPU Programming

A little bit about non-gfx GPU programming

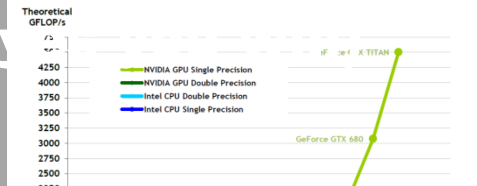
Why care about GPU hardware?

- Really, why? Aren't OpenGL and DX there so we don't have to care?
  - Graphics applications are never "fast enough"
    - Unlike, say, the rendering of a web page.
    - If you can do something twice as fast, you can make something else look better.
  - Knowing a little about how the HW works can lead to *very* much faster programs.
    - Unlike on a CPU, it is easy to write very slow code.
  - Conventional wisdom does not apply
    - As we will see, the GPU is a very different kind of beast.

Then I won't program GPUs!

(y

- Compute Power
- Memory Ban
- In every poc & PC…
- And superco
  - 5 / Top 10
  - More on t



Theoretical GFLOP/s

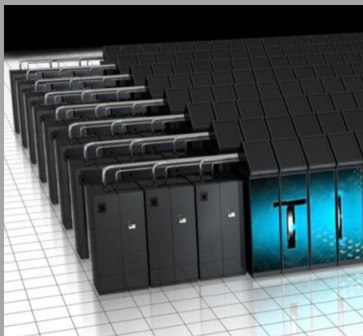
4750 4500 4250 4000 3750 3500 3250 3000 2750 2500 2250 2000 1750 1500 1250 1000 750 500 250 0

300 270 240 210 180 150 120 90 60 30 0

CPU
GeForce GPU
Tesla GPU

GeForce GTX TITAN, Tesla K20X, GeForce GTX 680, GeForce GTX 480, Tesla M2090, GeForce GTX 280, Tesla C2050, GeForce 8800 GTX, Tesla C1060, GeForce 7800 GTX, Sandy Bridge, GeForce 6800 GT, Bloomfield, Woodcrest, Prescott, Westmere, Harpertown, Northwood

2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013



Outline

Part 1: Motivation

If you're not into hardware, why should you care?

Part 2: History lesson

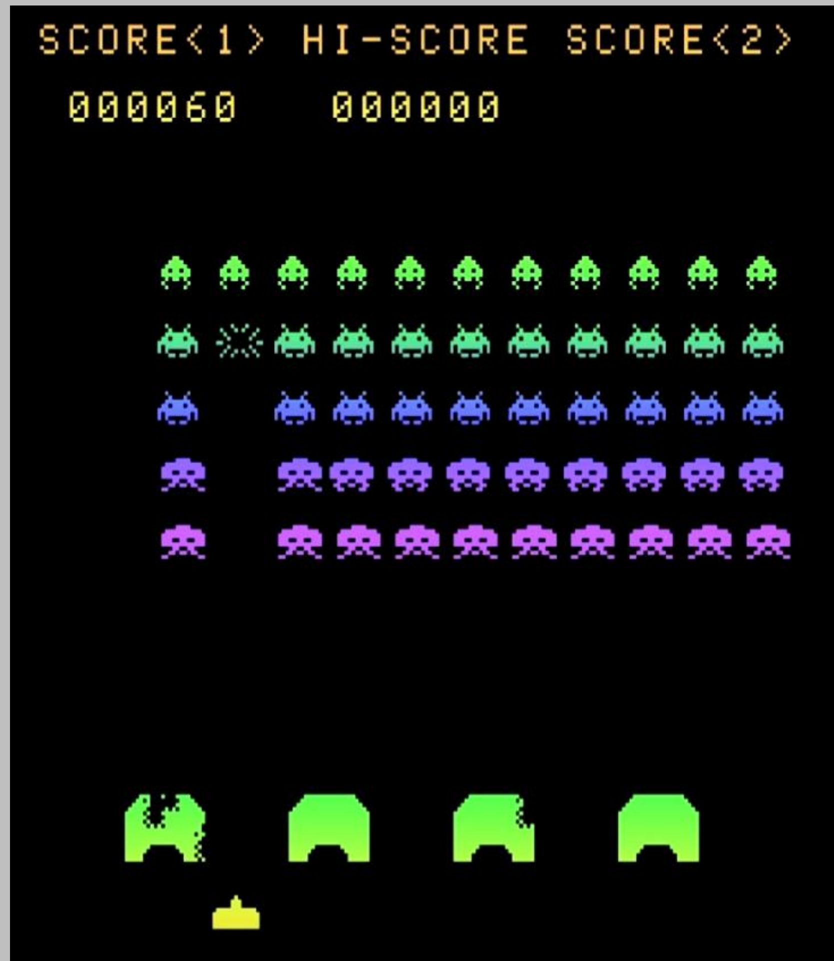
Where did GPUs come from, and what were they meant to do?

Part 3 GPU Architecture

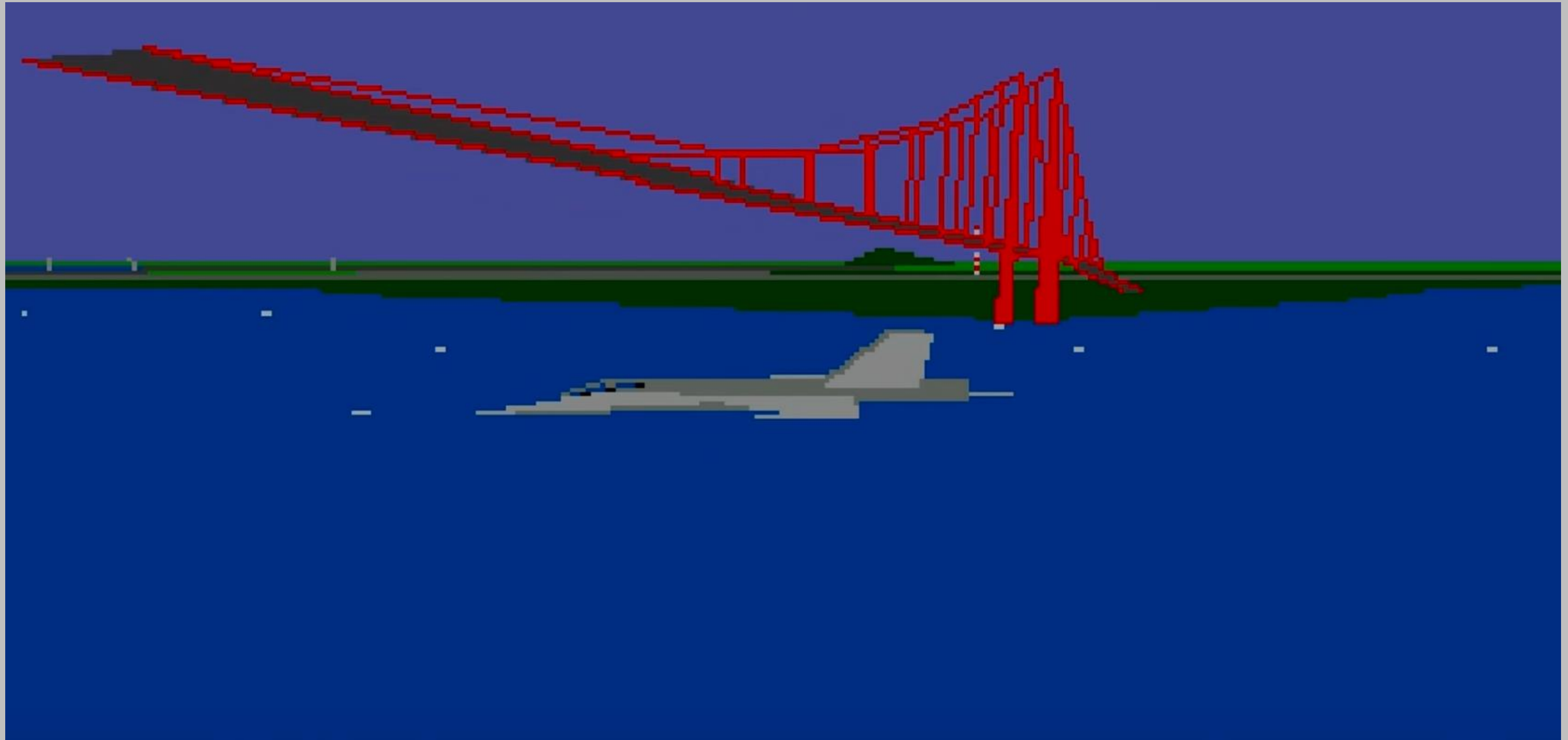
What is a modern GPU, and how does it work?

Part 4: General Purpose GPU Programming

A little bit about non-gfx GPU programming



Space Invaders - 1978

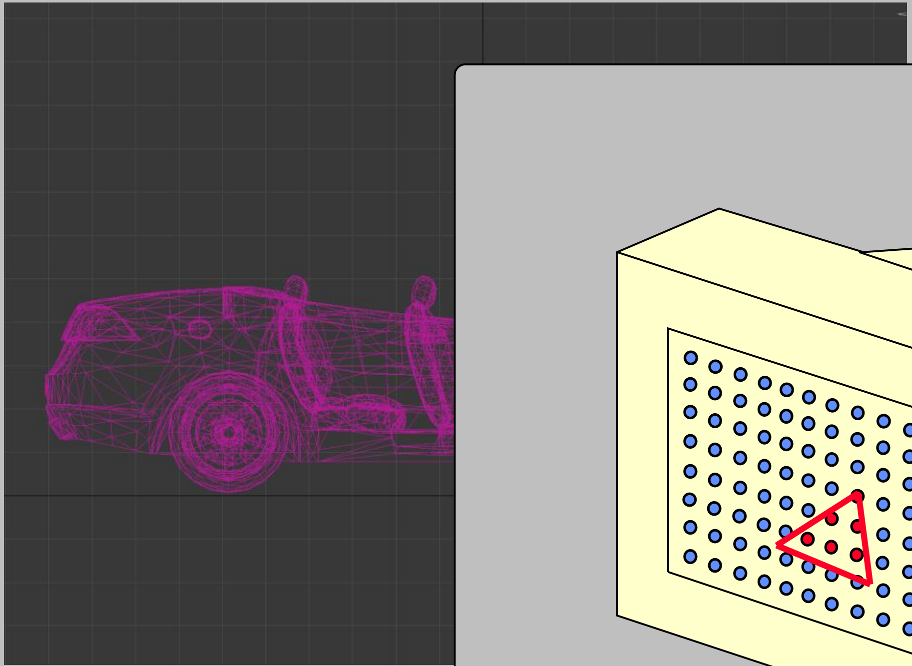


F/A-18 Interceptor - 1988

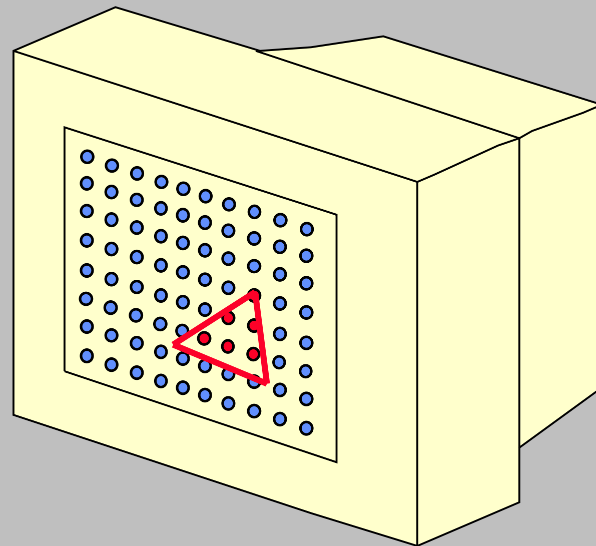
DOOM 1993

Quake 1996

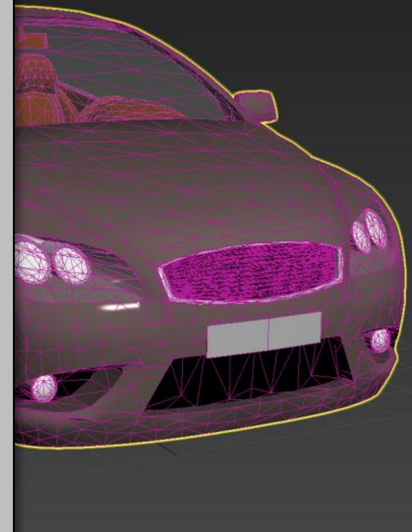
Vertex Transformation



Scene: Triangles



Rasterization



red Image

Enter the GPU!

3dFX Voodoo 1 (1996) – Rasterize Triangle

NVIDIA GeForce 256 (1999)
Triangle transformation *and*
Lighting calculations in HW!

Programmable shading

GeForce 3 (2001)

DOOM 3 (2004)

Block Diagram of 2006 GPU

Input Assembly

Primitive Assembly

Triangle Setup

Rasterizer

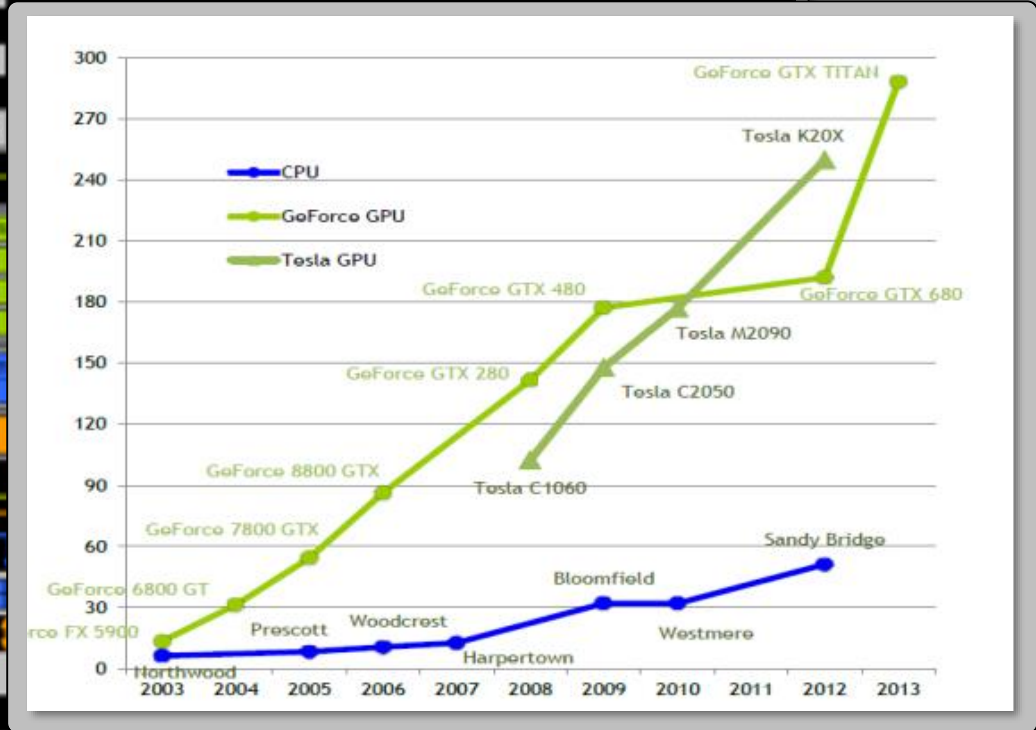
Z-culling

Blending

G80 and Unified Shader Architecture

GeForce 8800 Architecture

Host
Data Assembler
Vtx Thread Issue
L1
L1
L1
L2
L2
FB
FB



GeForce 8800 GT (2007)

Key Observations

- Many small independent and identical jobs
  - Abundant data parallelism.
- Lots of ALU operations
  - Small matrix linear algebra
  - Many MADs
- High/Streaming bandwidth requirements
  - Lots of data touched just once.

Outline

Part 1: Motivation

If you're not into hardware, why should you care?

Part 2: History lesson

Where did GPUs come from, and what were they meant to do?

Part 3 GPU Architecture

What is a modern GPU, and how does it work?

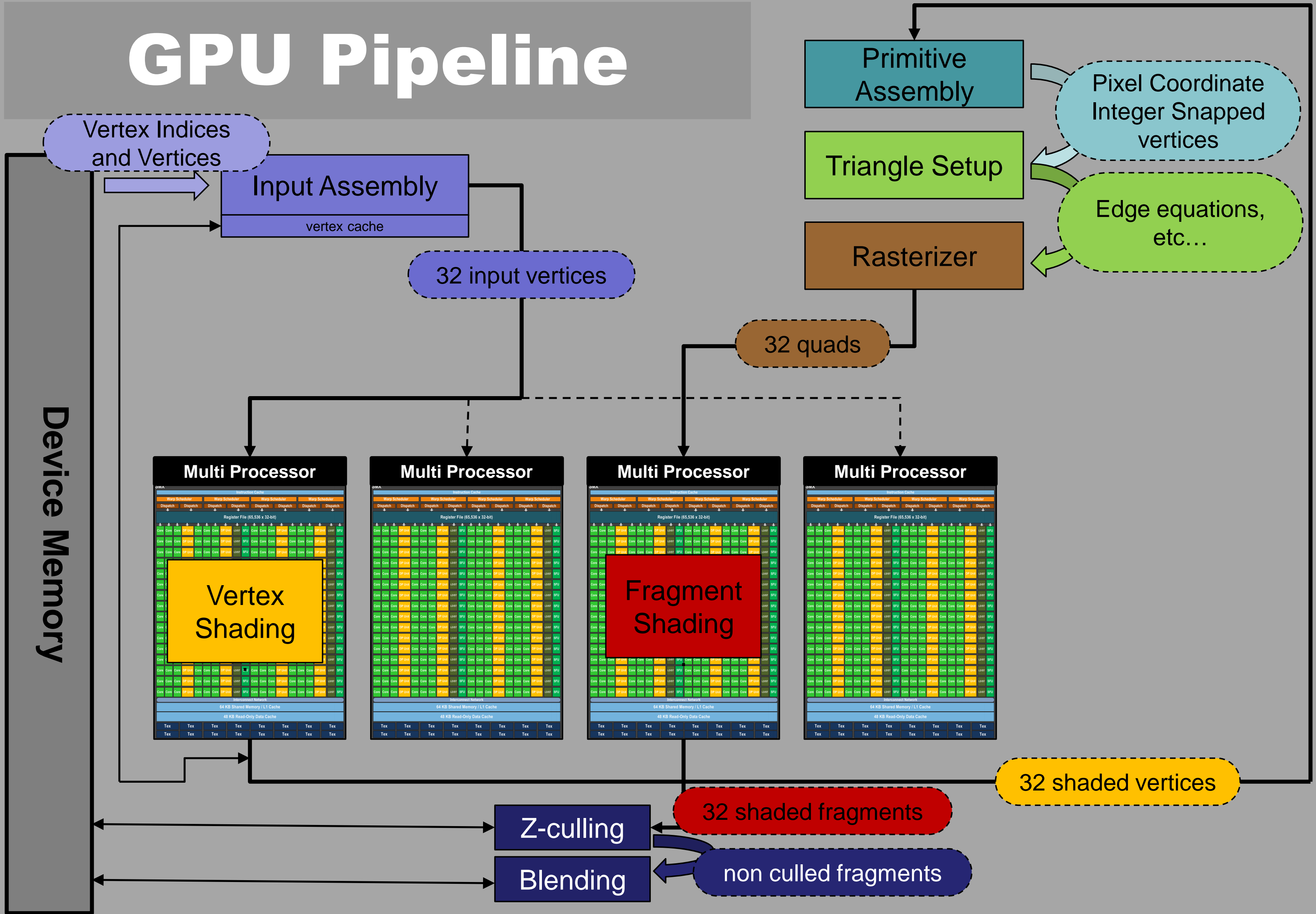
Part 4: General Purpose GPU Programming

A little bit about non-gfx GPU programming

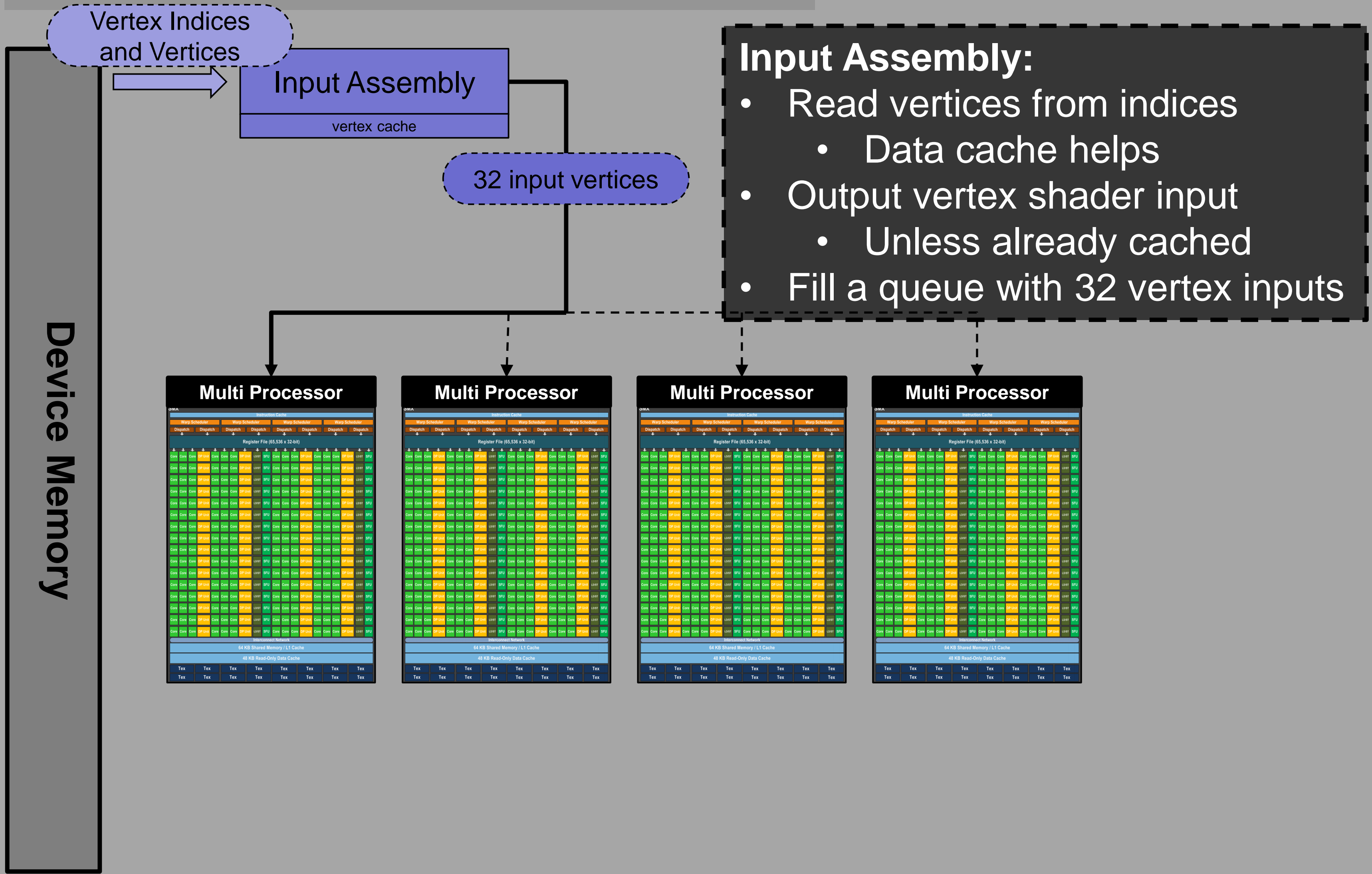
Disclaimer

- Companies are *very* secretive about hardware
  - There are a lot of things the public simply don't *know* about the details of how a GPU is build.
  - My info comes either from public presentations or from other (smarter) people who have made qualified guesses.
- These are *extremely* complex hardware systems
  - So there are many things I *know* that I don't really know how they work
  - And many things I *think* I understand how they work, but I really don't.
  - And many things I have simplified or skipped over just to make this lecture manageable.

GPU Pipeline



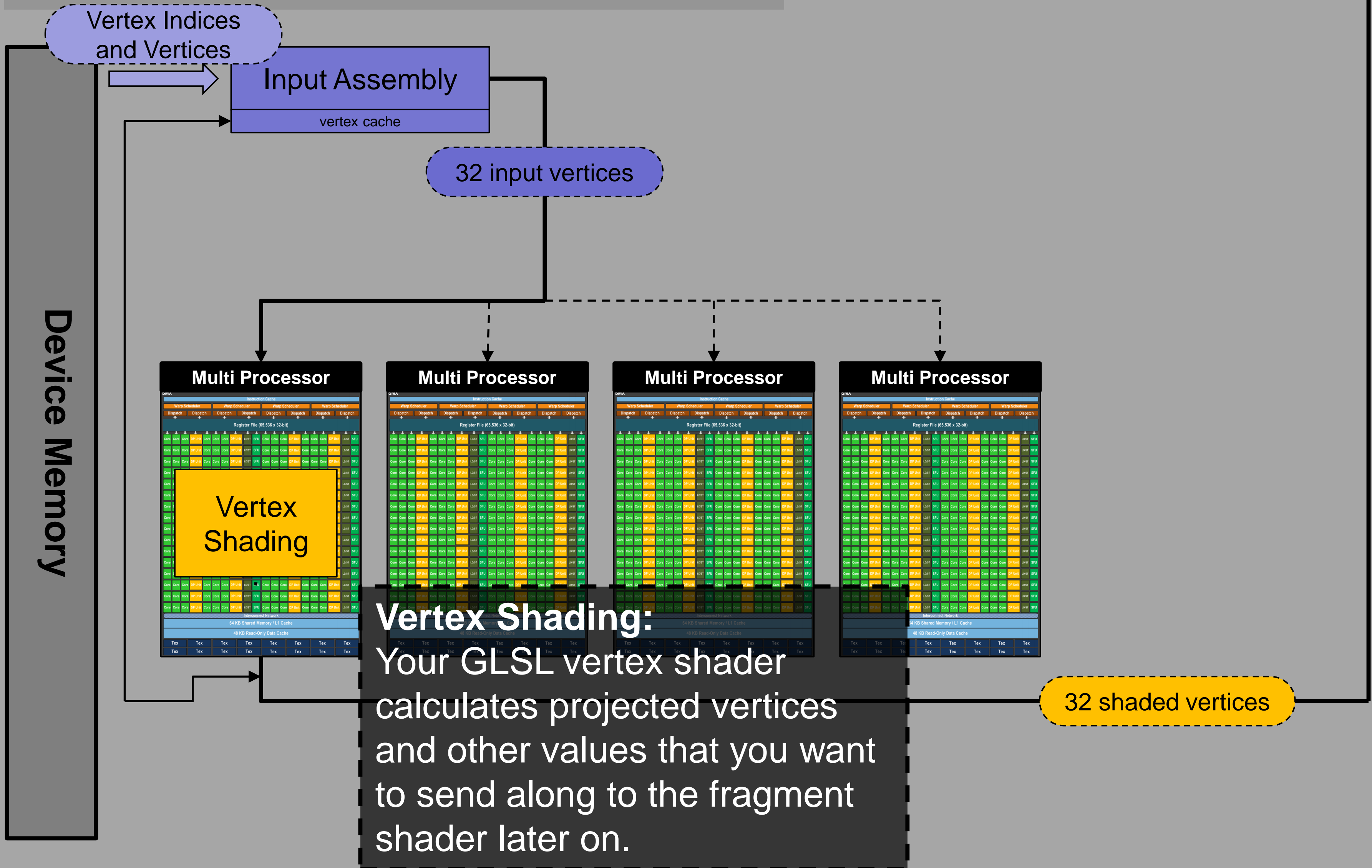
Vertex Processing



Input Assembly:

- Read vertices from indices
  - Data cache helps
- Output vertex shader input
  - Unless already cached
- Fill a queue with 32 vertex inputs

Vertex Processing



Vertex Shading:
Your GLSL vertex shader calculates projected vertices and other values that you want to send along to the fragment shader later on.

Primitive Assembly

Vertex Indices and Vertices

Input Assembly

vertex cache

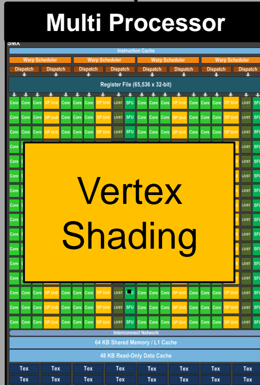
32 input vertices

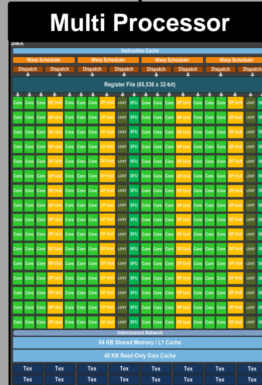
Device Memory

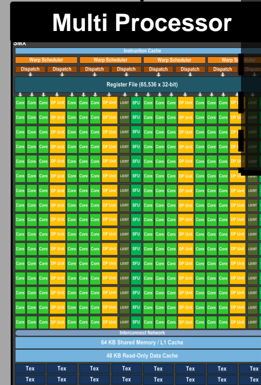
Primitive Assembly

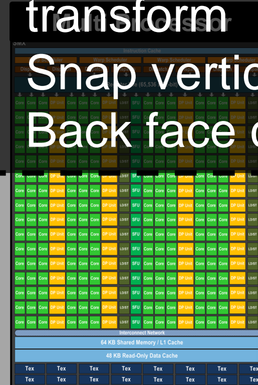
Primitive Assembly & co:

- Grab 3 vertices
- Cull if they are all outside some clipping plane.
- Clip triangles (may produce more primitives)
- Homogenization and viewport-transform
- Snap vertices to subpixel cords
- Back face culling

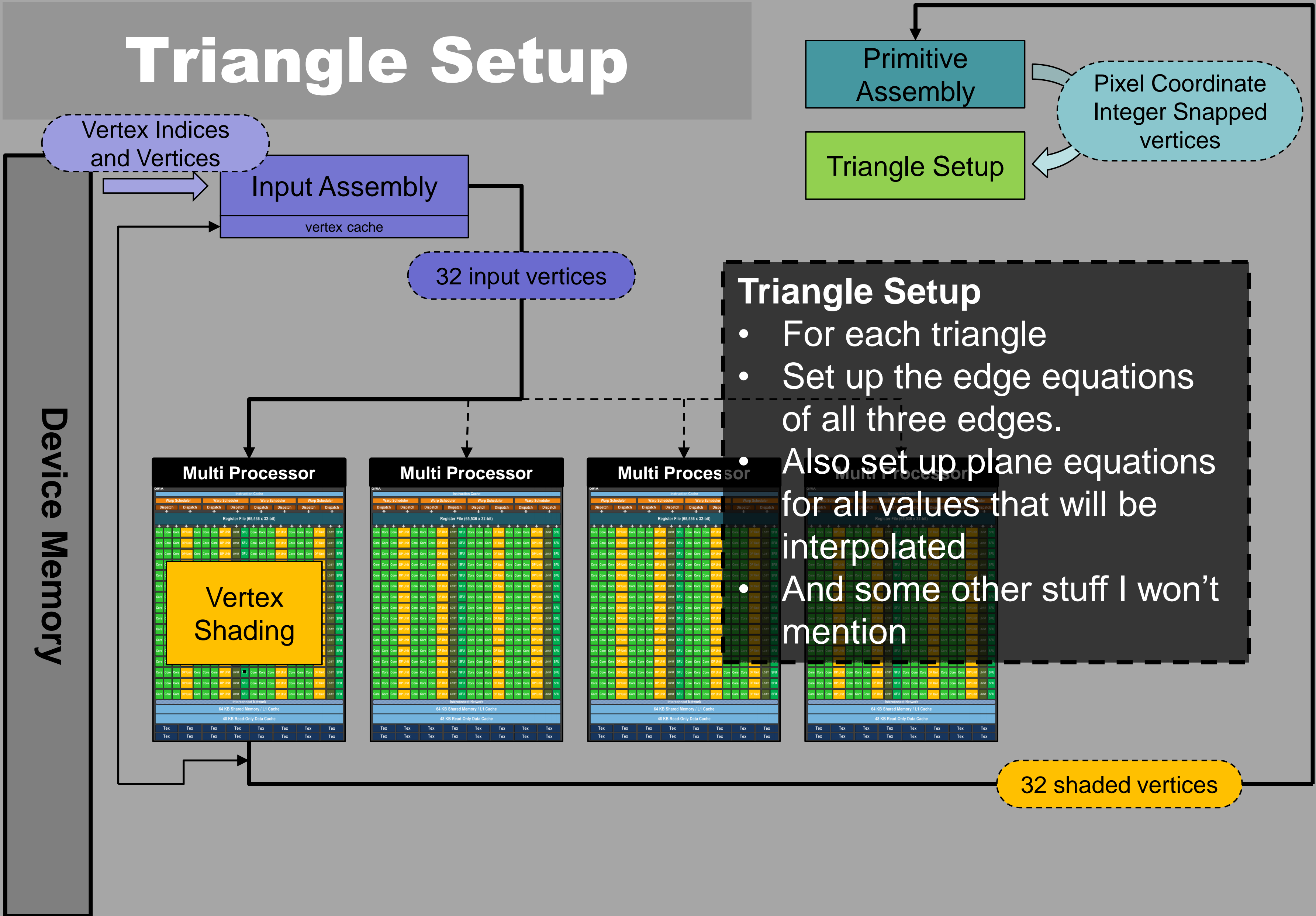




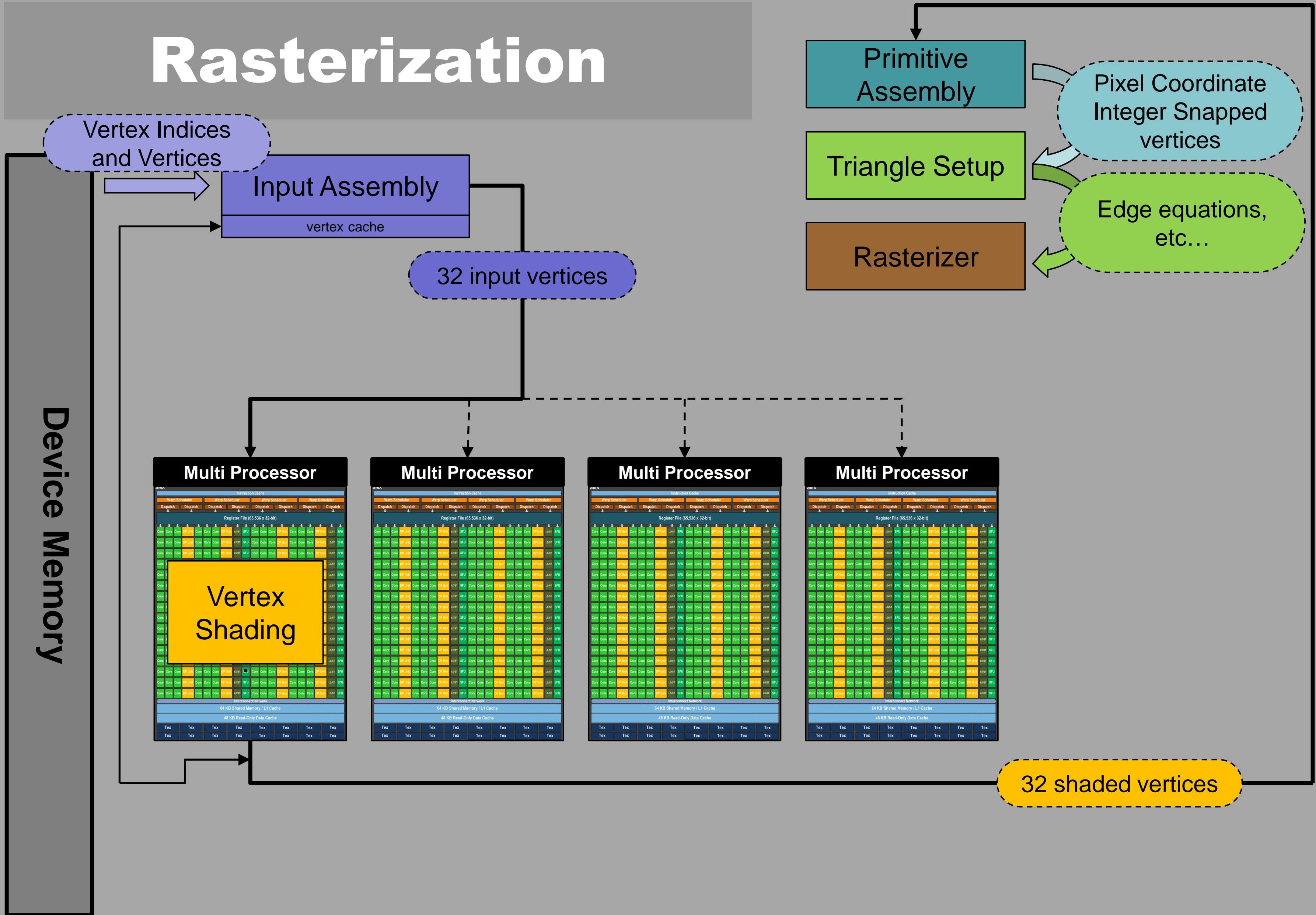




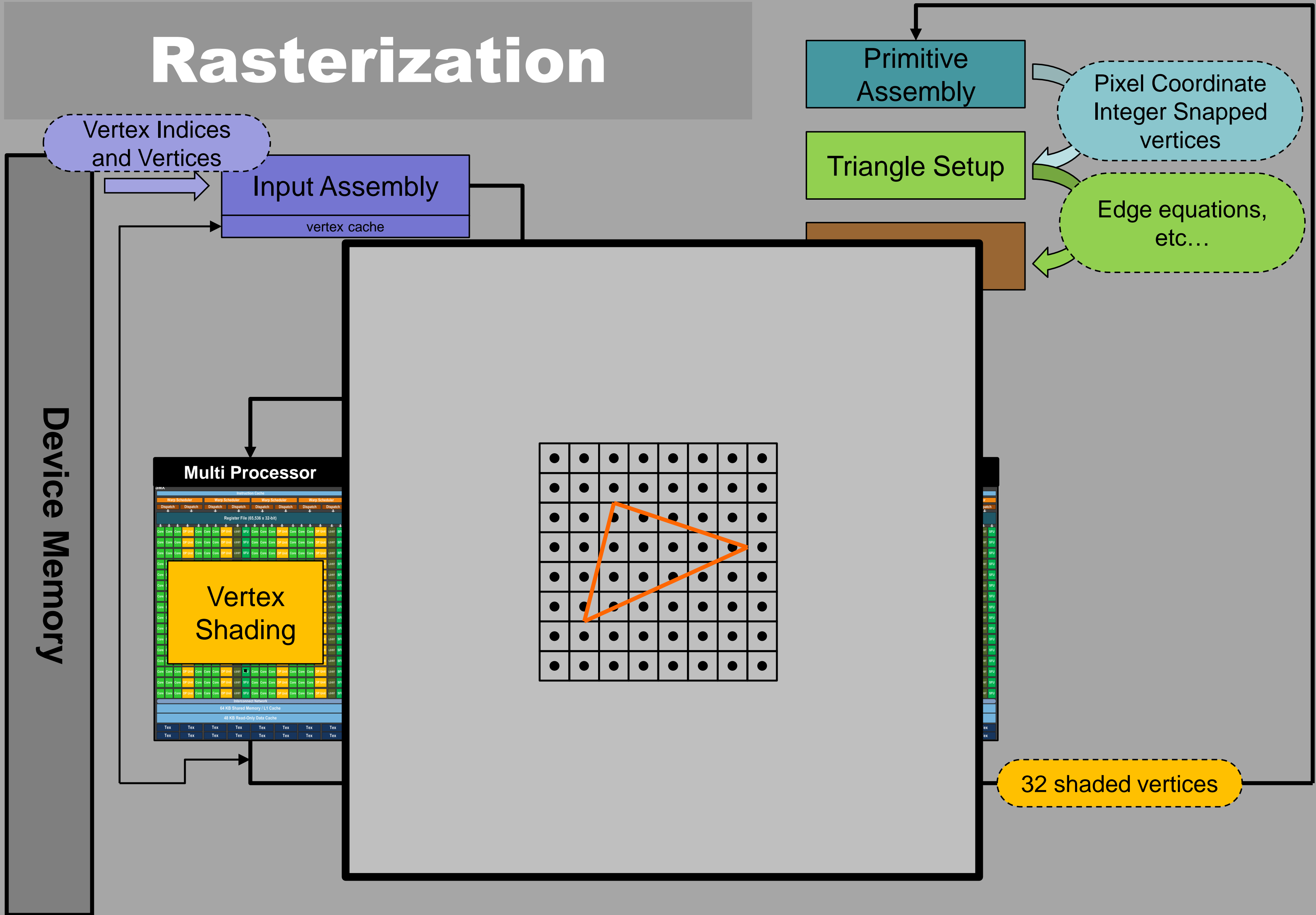
32 shaded vertices

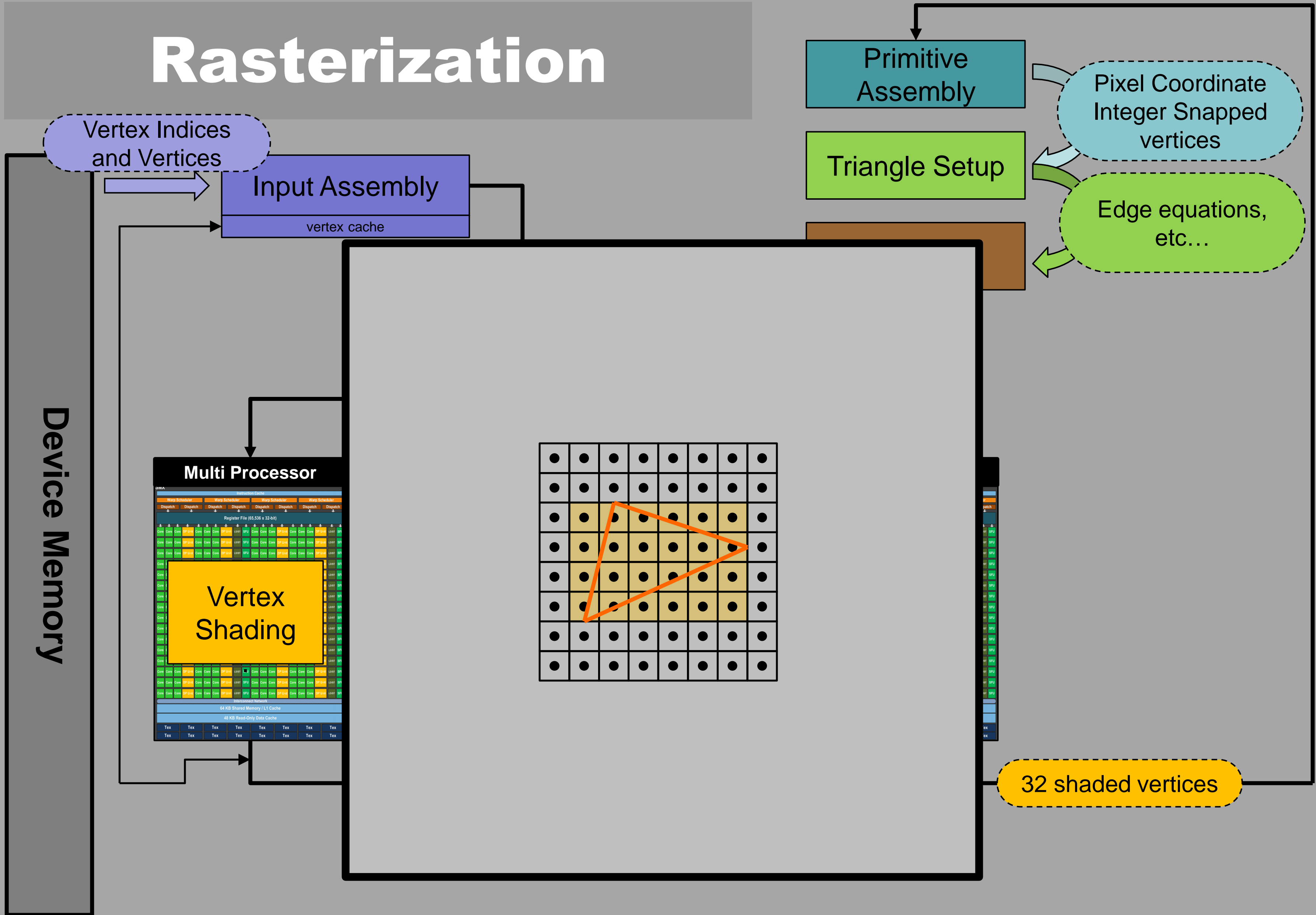


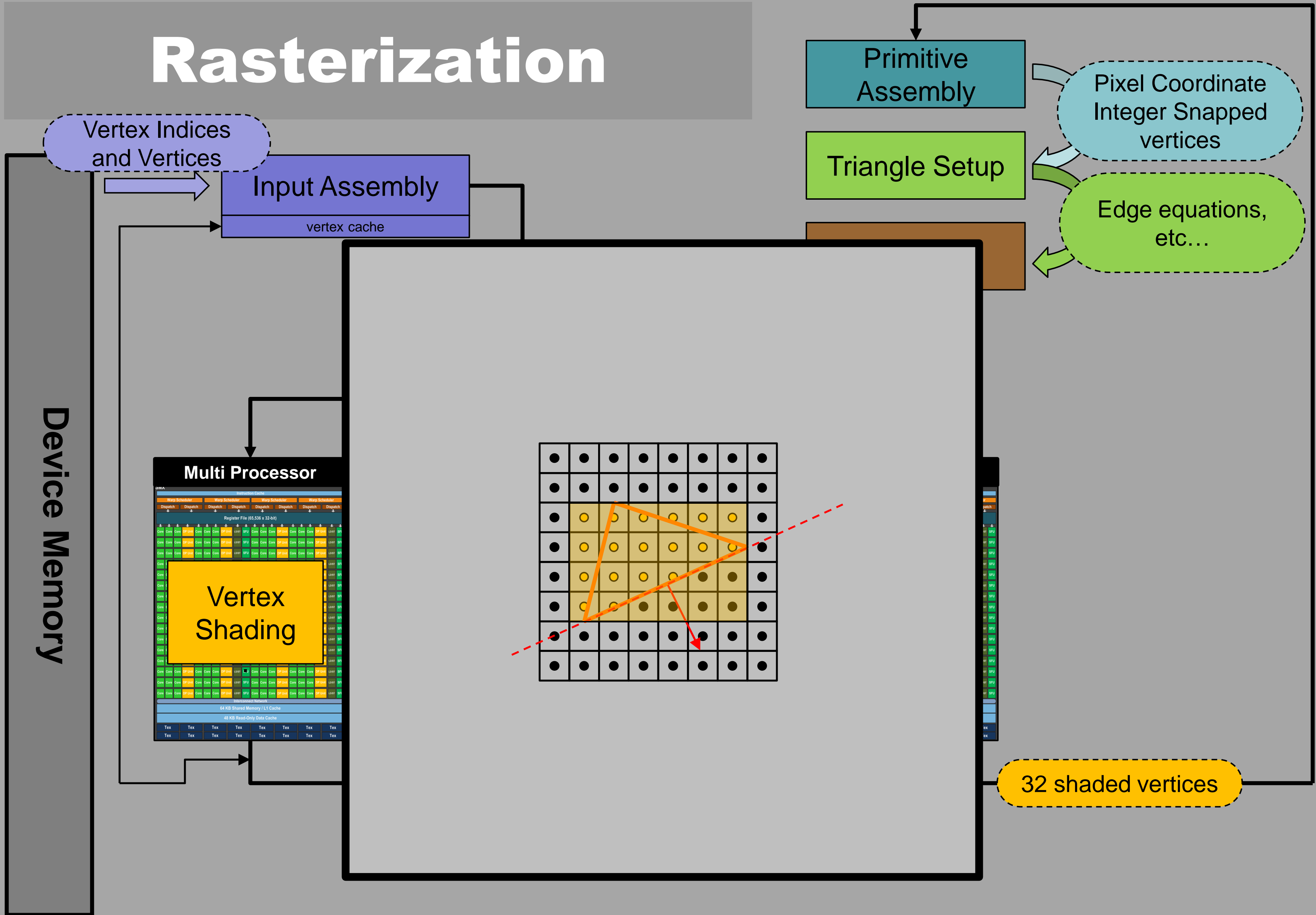
Triangle Setup
Vertex Indices and Vertices
Input Assembly
vertex cache
32 input vertices
Device Memory
Multi Processor
Multi Processor
Multi Processor
Vertex Shading
32 shaded vertices
Primitive Assembly
Pixel Coordinate Integer Snapped vertices
Triangle Setup
Triangle Setup
• For each triangle
• Set up the edge equations of all three edges.
• Also set up plane equations for all values that will be interpolated
• And some other stuff I won't mention


Rasterization
Vertex Indices and Vertices
Input Assembly
vertex cache
32 input vertices
Device Memory
Multi Processor
Multi Processor
Multi Processor
Multi Processor
Vertex Shading
32 shaded vertices
Primitive Assembly
Triangle Setup
Rasterizer
Pixel Coordinate Integer Snapped vertices
Edge equations, etc…

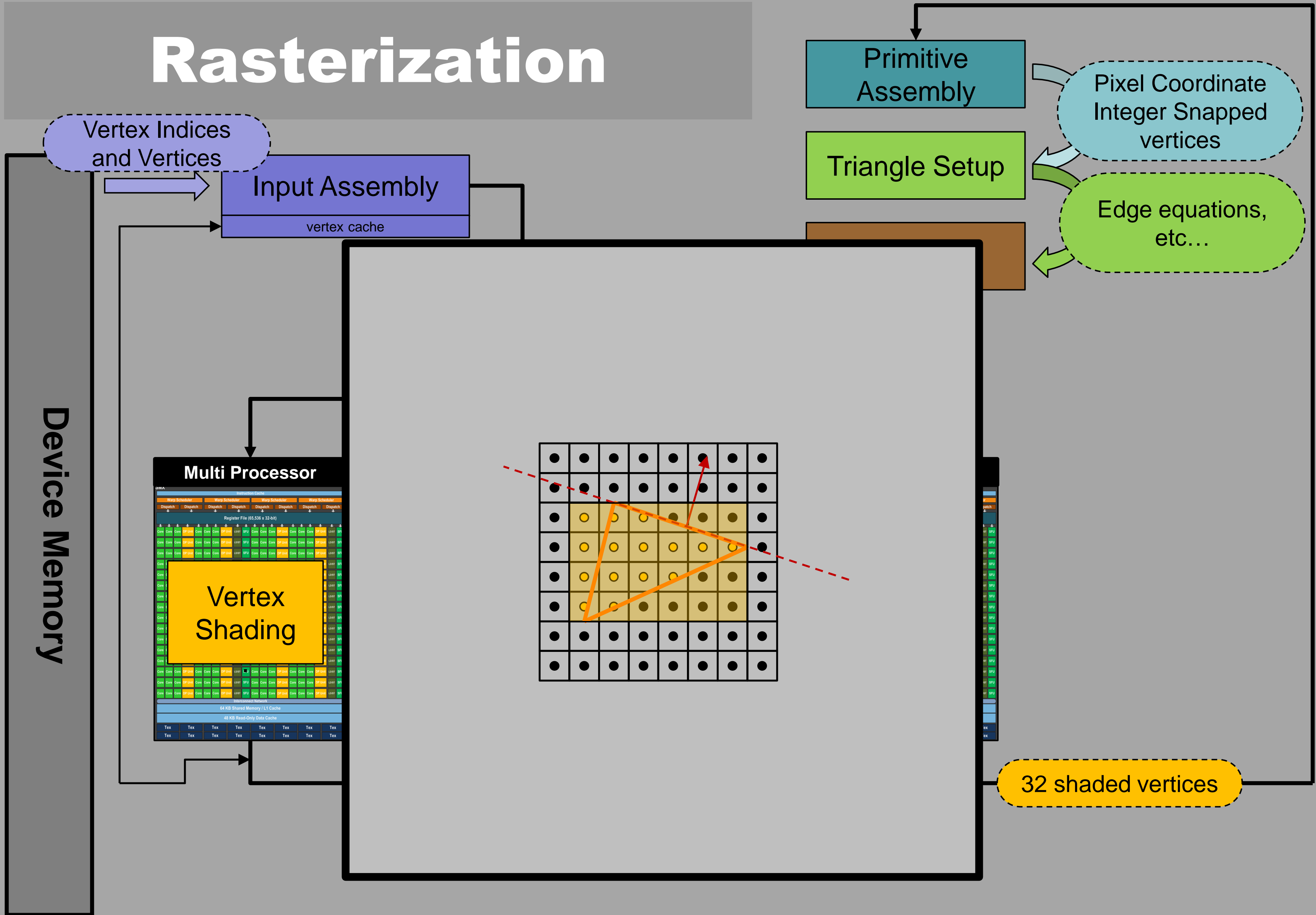
Rasterization



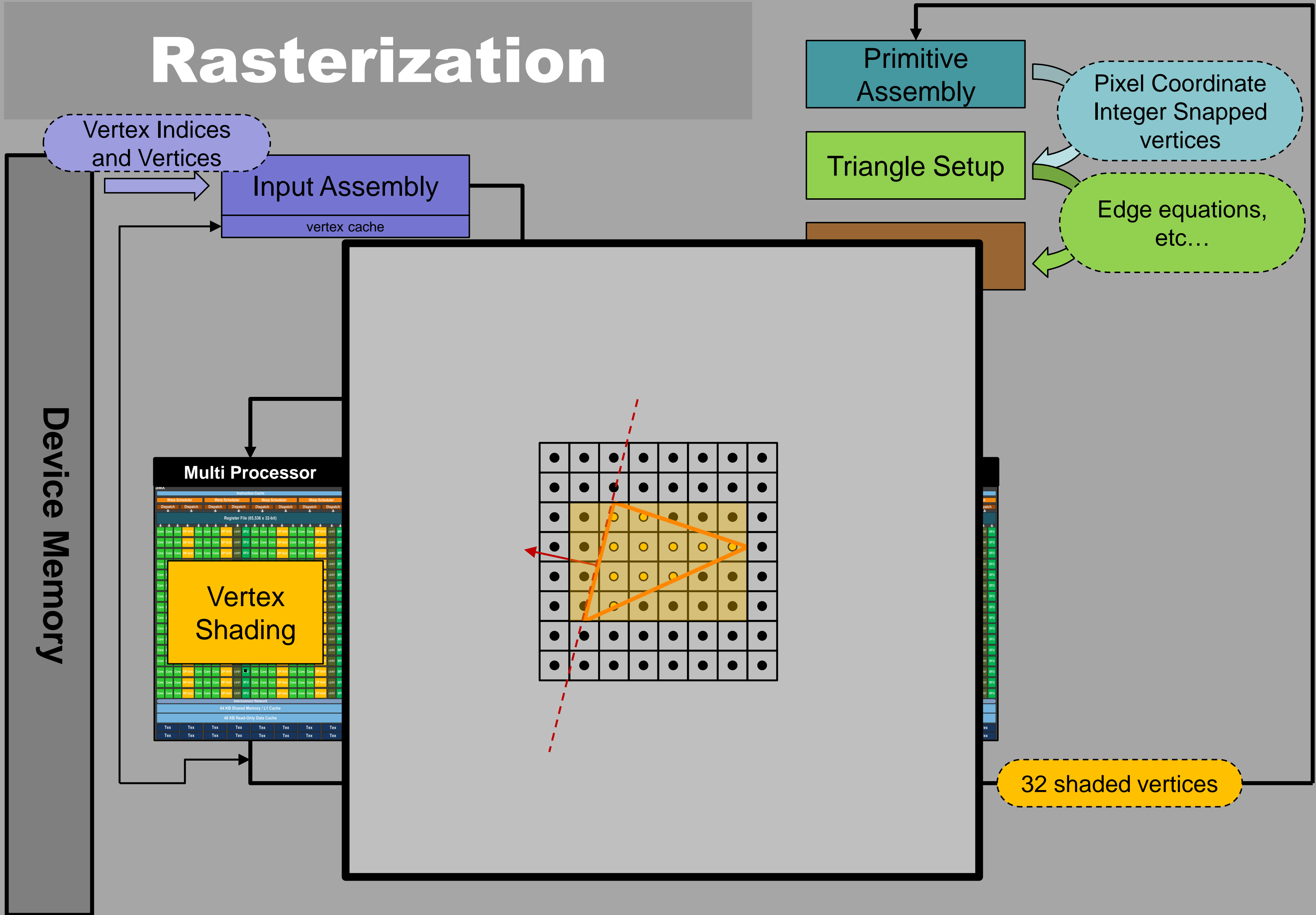

Rasterization
Vertex Indices and Vertices
Input Assembly
vertex cache
Device Memory
Multi Processor
Vertex Shading
Primitive Assembly
Pixel Coordinate Integer Snapped vertices
Triangle Setup
Edge equations, etc…
32 shaded vertices

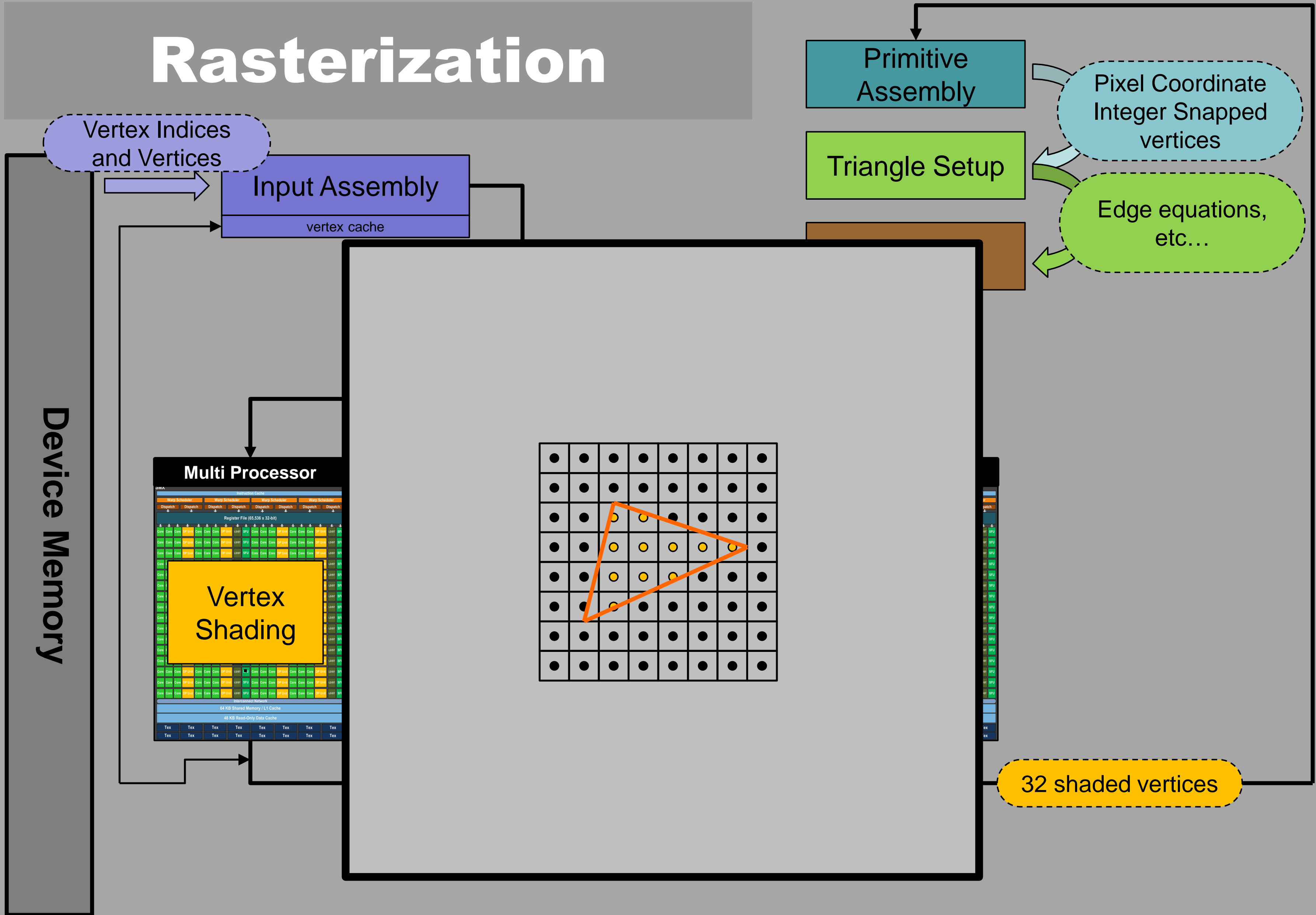

Rasterization
Vertex Indices and Vertices
Input Assembly
vertex cache
Device Memory
Multi Processor
Vertex Shading
Primitive Assembly
Triangle Setup
Pixel Coordinate Integer Snapped vertices
Edge equations, etc…
32 shaded vertices

Rasterization

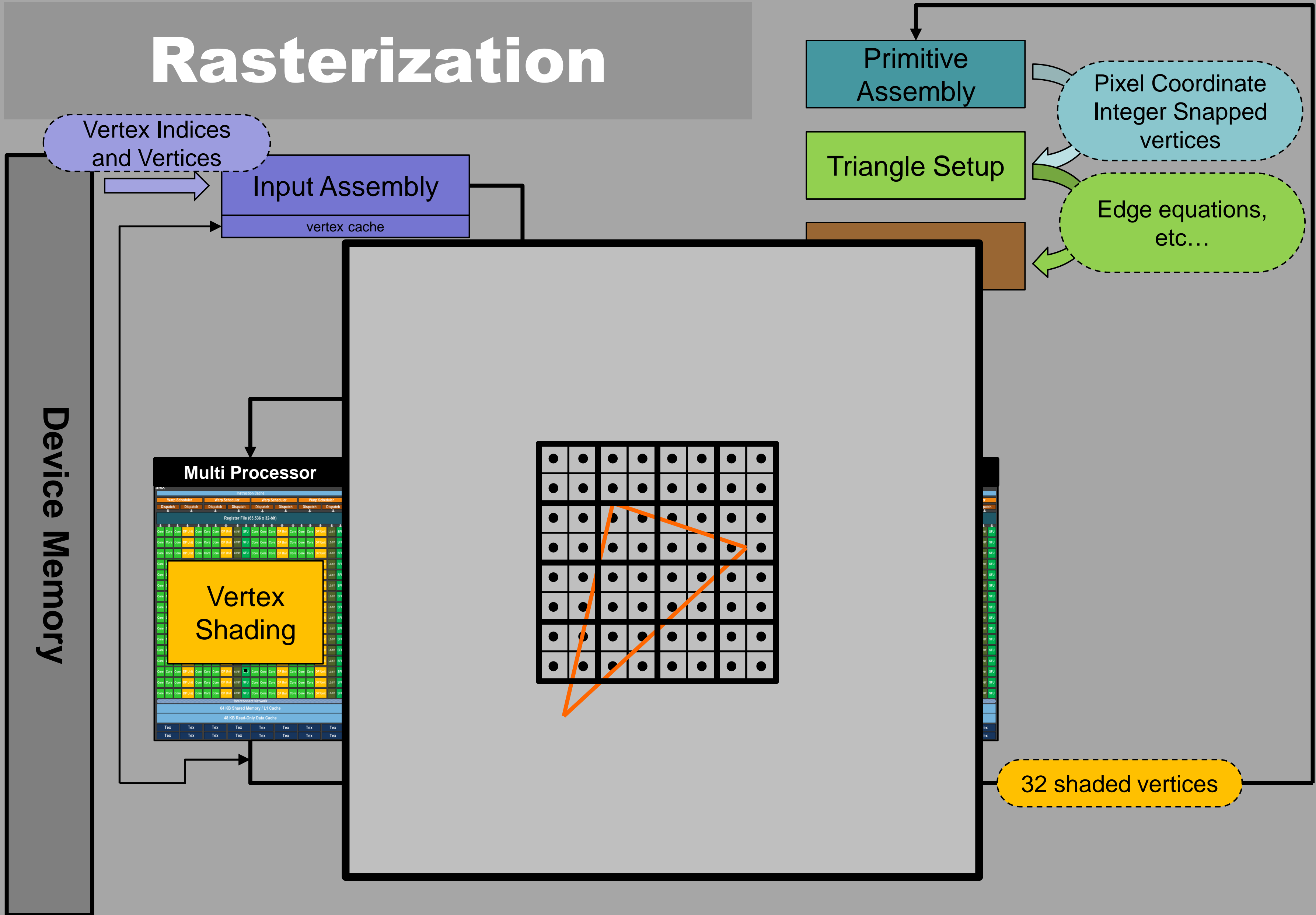


Rasterization

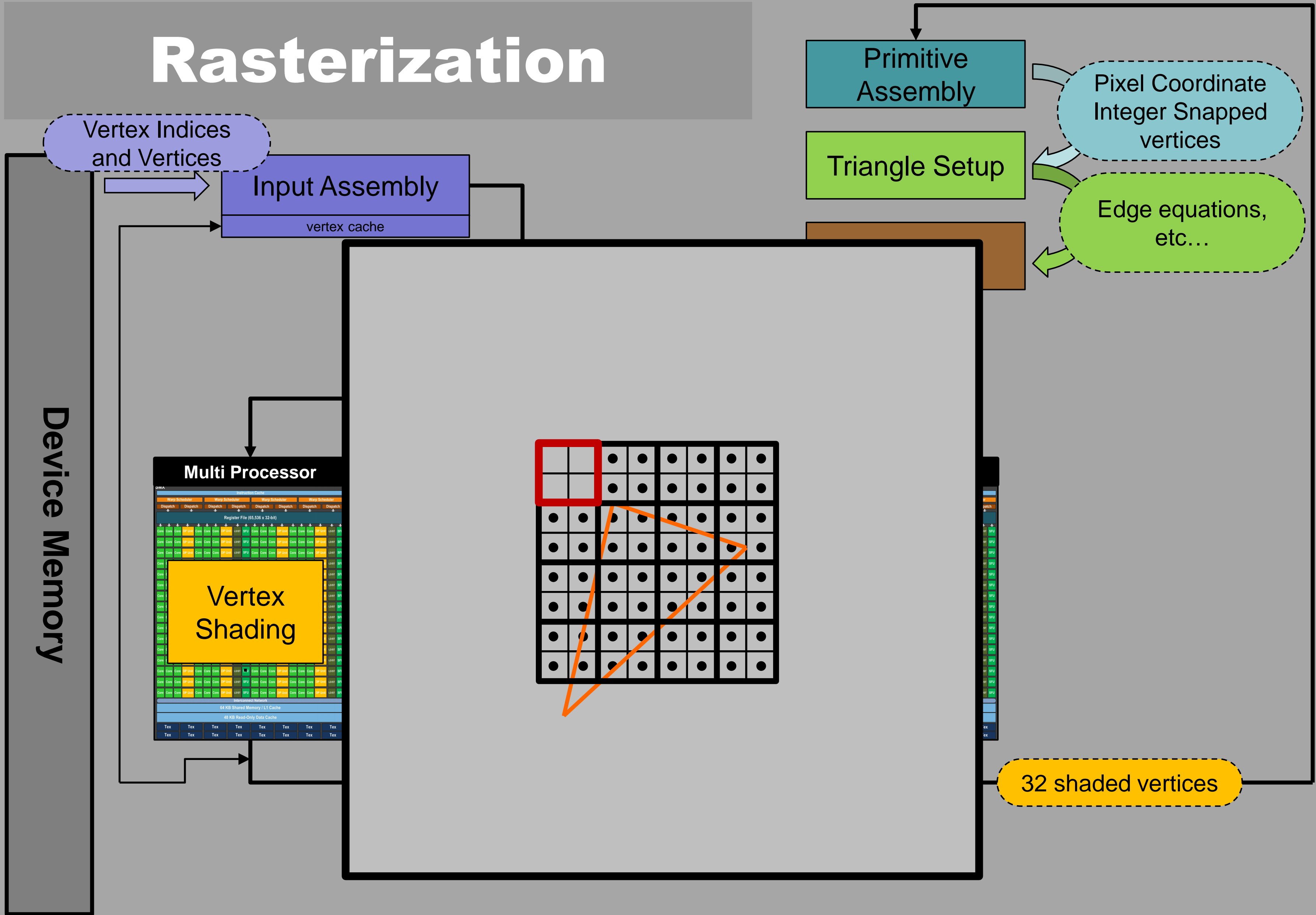


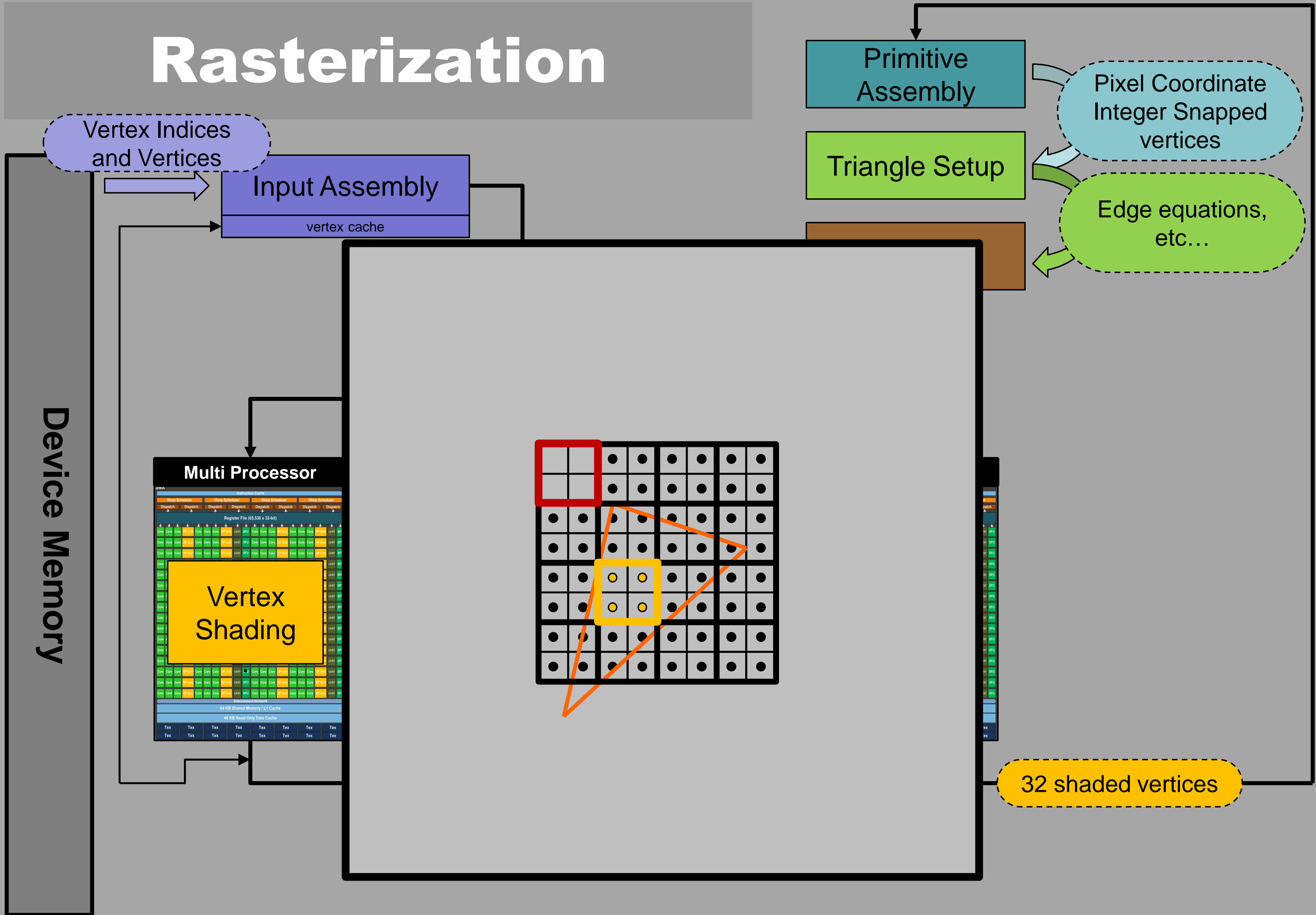

Rasterization
Primitive Assembly
Pixel Coordinate Integer Snapped vertices
Triangle Setup
Edge equations, etc…
Vertex Indices and Vertices
Input Assembly
vertex cache
Device Memory
Multi Processor
Vertex Shading
32 shaded vertices

Rasterization

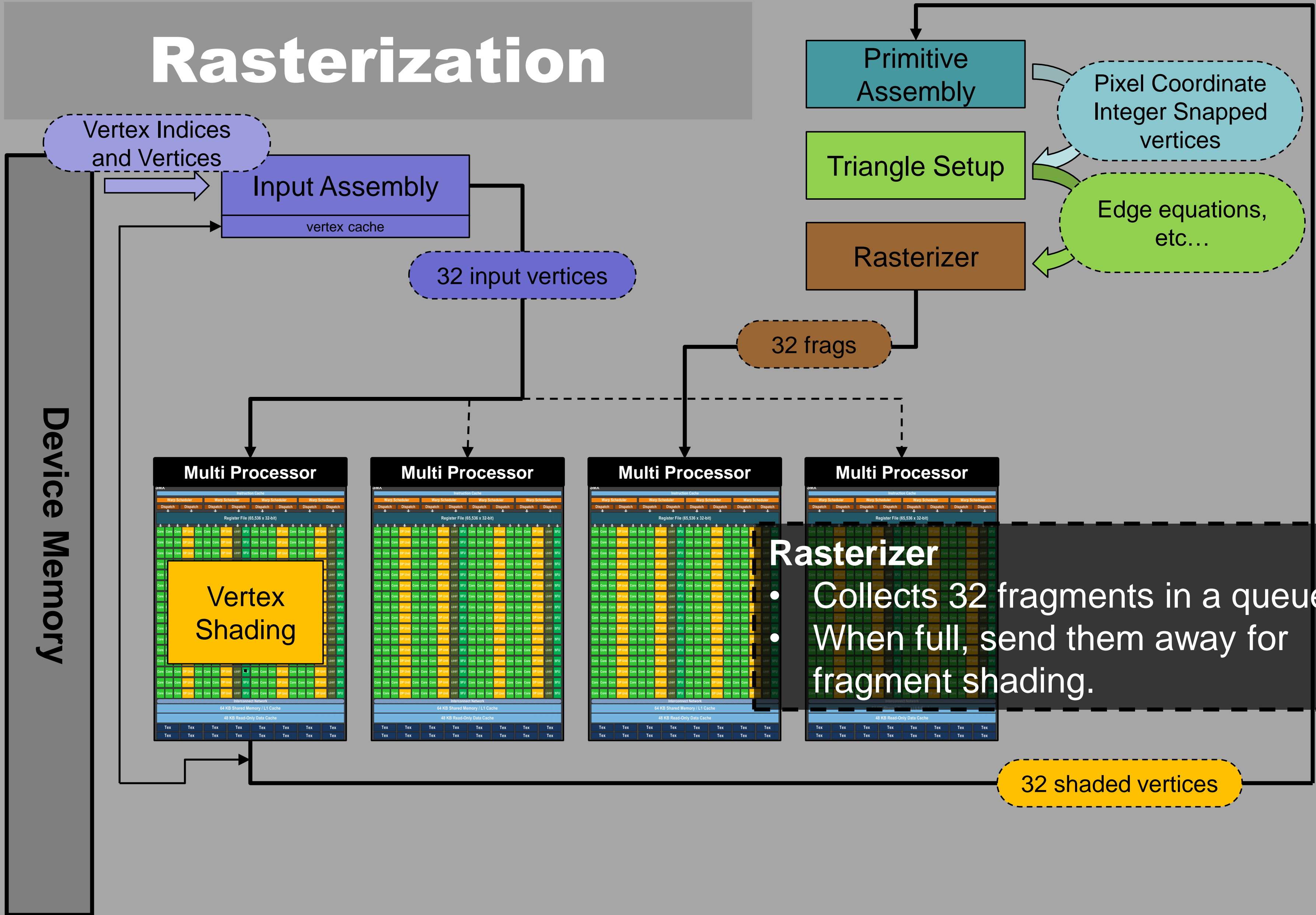


Rasterization




Rasterization
Primitive Assembly
Pixel Coordinate Integer Snapped vertices
Vertex Indices and Vertices
Input Assembly
vertex cache
Triangle Setup
Edge equations, etc…
Device Memory
Multi Processor
Vertex Shading
32 shaded vertices

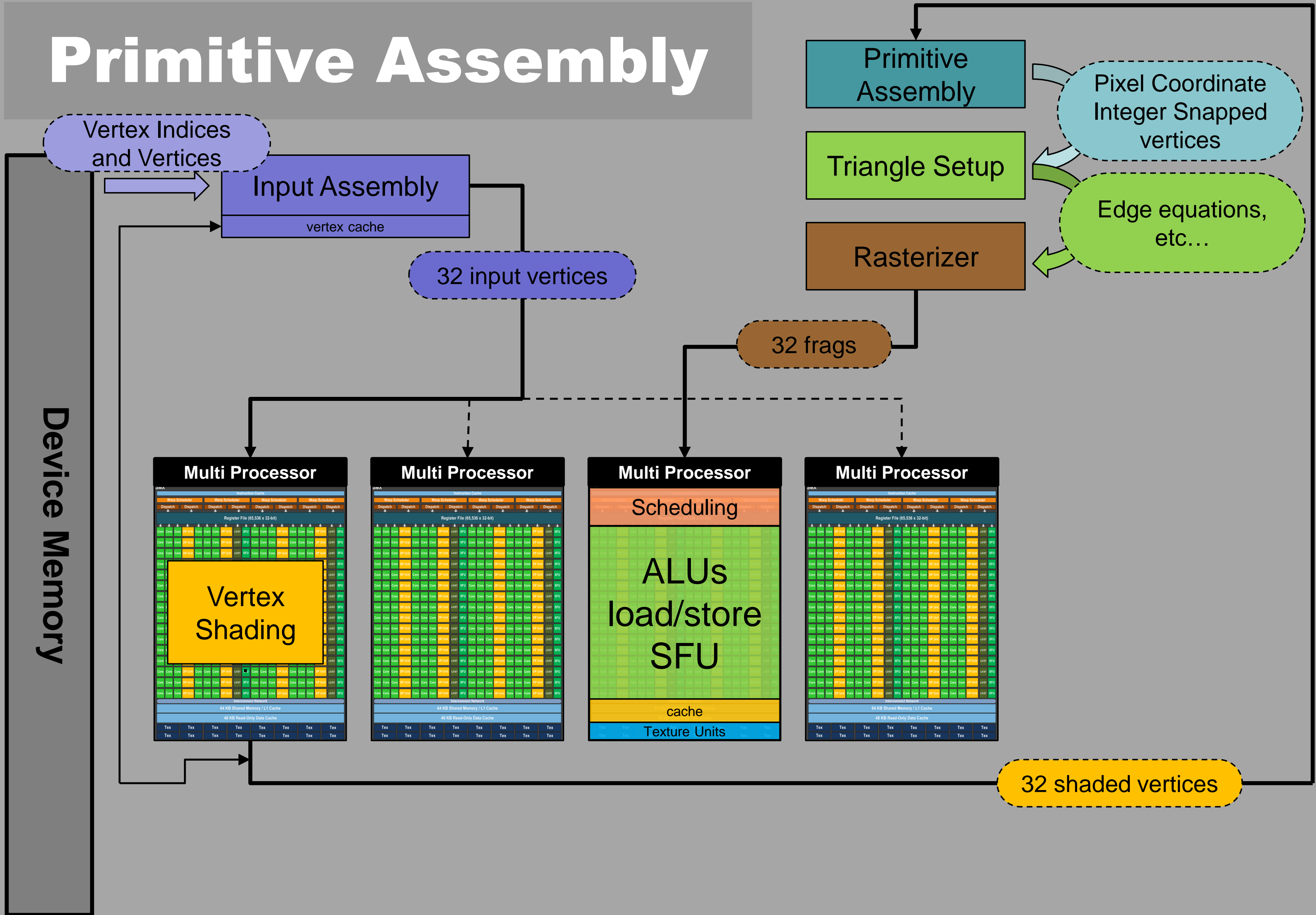
Rasterization



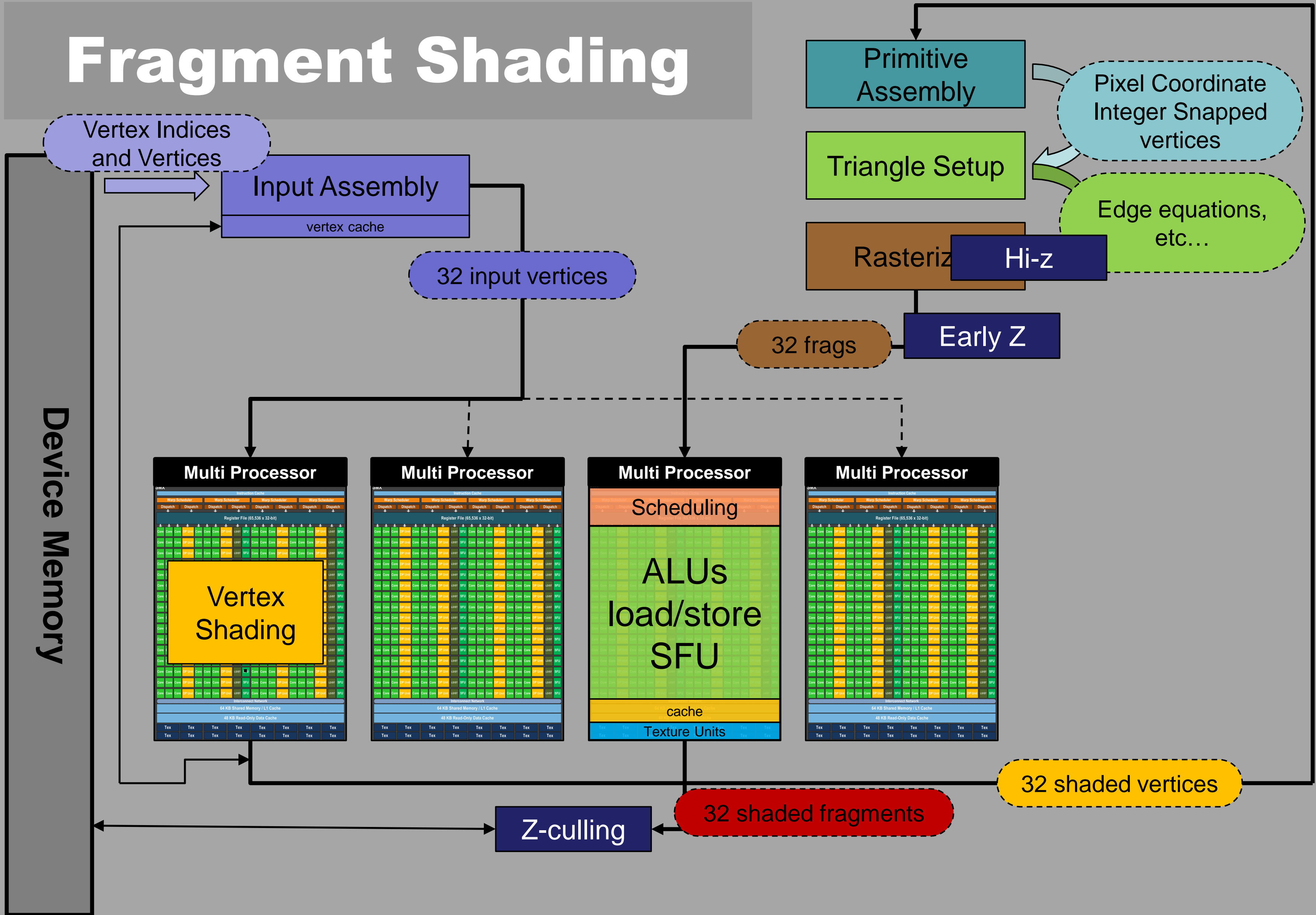
Rasterizer

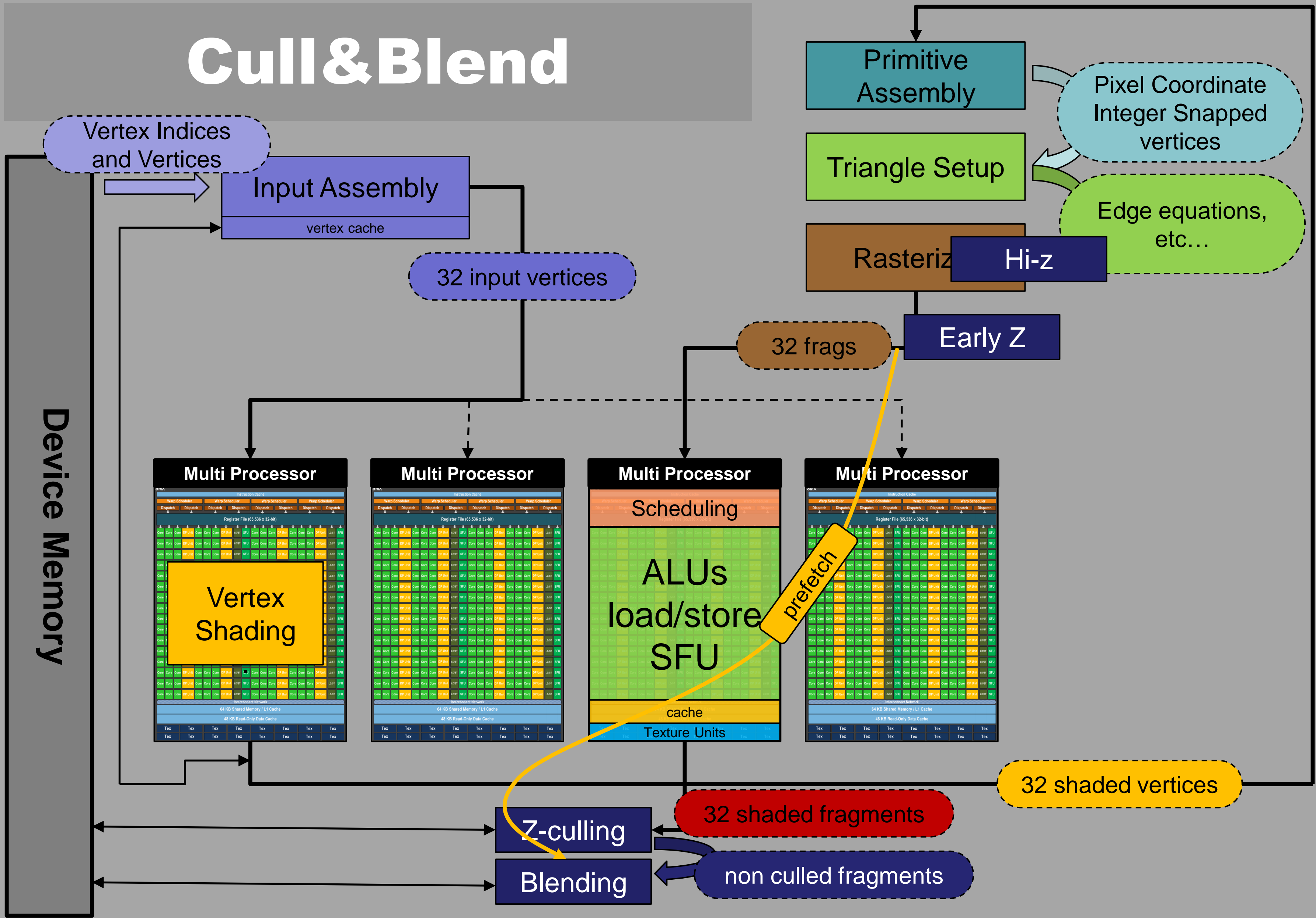
- Collects 32 fragments in a queue.
- When full, send them away for fragment shading.

Primitive Assembly



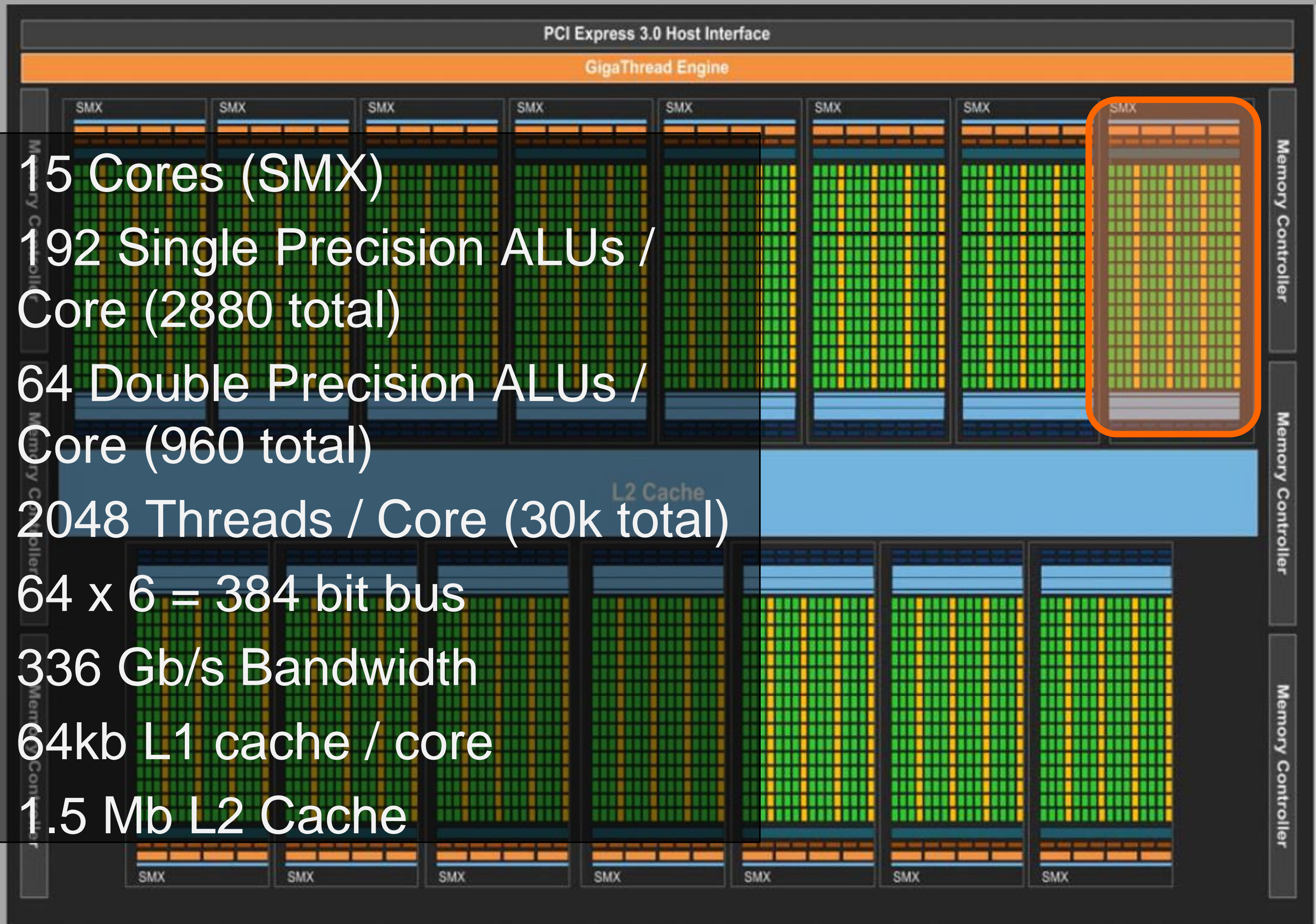
Fragment Shading




Cull&Blend
Vertex Indices and Vertices
Input Assembly
vertex cache
32 input vertices
Device Memory
Primitive Assembly
Pixel Coordinate Integer Snapped vertices
Triangle Setup
Edge equations, etc…
Rasteriz
Hi-z
Early Z
32 frags
Multi Processor
Vertex Shading
Multi Processor
Multi Processor
Scheduling
ALUs
load/store
SFU
cache
Texture Units
prefetch
Multi Processor
32 shaded vertices
32 shaded fragments
Z-culling
Blending
non culled fragments

Kepler GK110 (GTX780/Titan)

- 15 Cores (SMX)
- 192 Single Precision ALUs / Core (2880 total)
- 64 Double Precision ALUs / Core (960 total)
- 2048 Threads / Core (30k total)
- 64 x 6 = 384 bit bus
- 336 Gb/s Bandwidth
- 64kb L1 cache / core
- 1.5 Mb L2 Cache



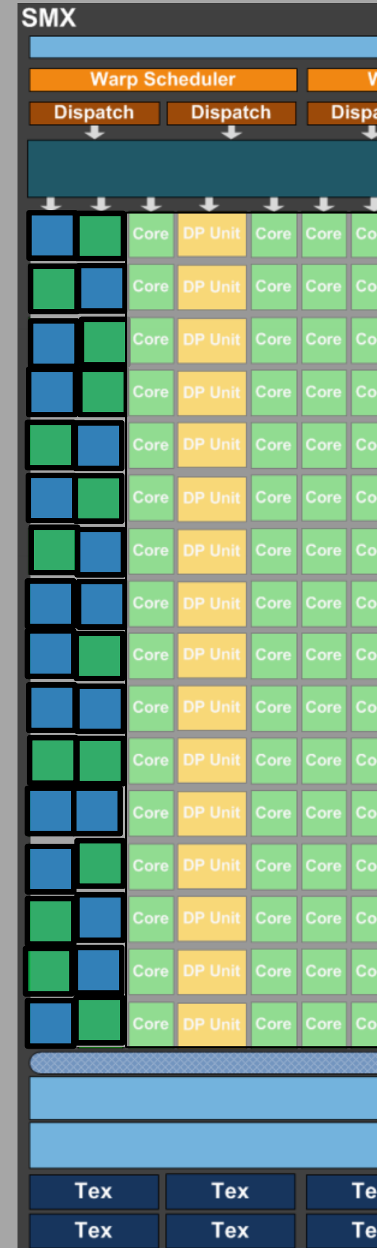
Kepler GK110 (GTX780/Titan)

- A *warp* of 32 threads are executed in lockstep
- Two independent instructions in one warp can run in one clock
- Four warps can run simultaneously

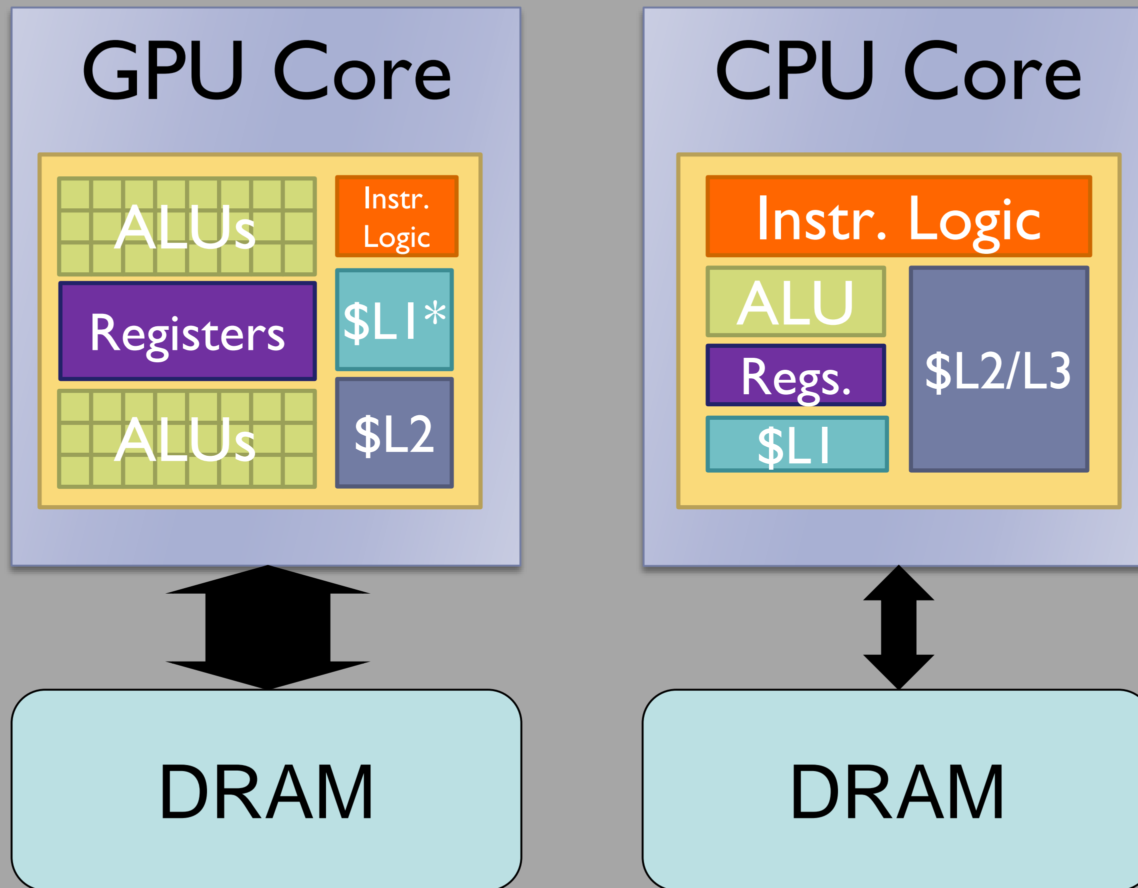
Warp divergence

- 32 threads in a *warp* are run simultaneously.
- So, if we have divergence within a warp, both paths must be taken
  - Threads that don't fulfill the condition are masked out

```
if(something) {
  a = 0.0f;
}
else {
  a = 1.0f
}
```

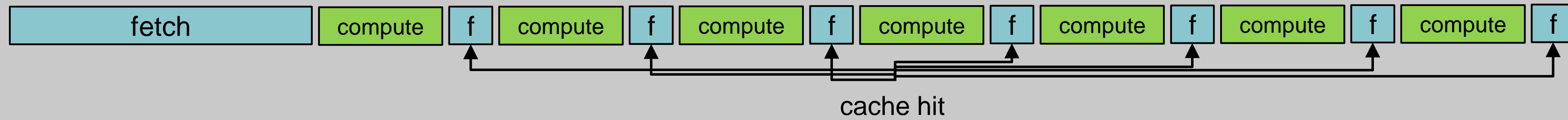


The Memory Hierarchy



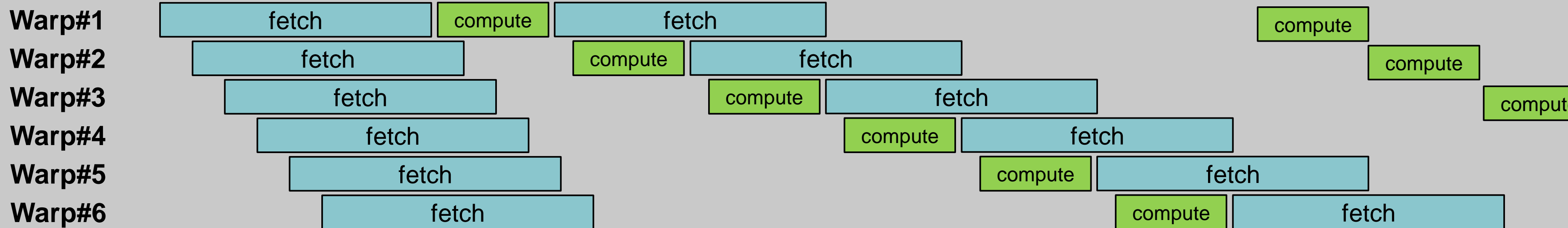
Latency hiding

CPU



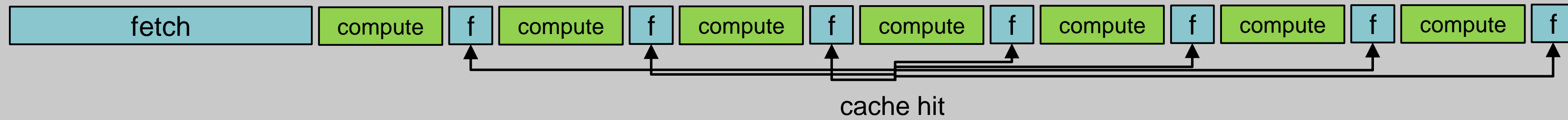

```
for(int i=0; i<…; i++)
{
  a = <fetch data>;
  b = compute(a);
}
```

GPU



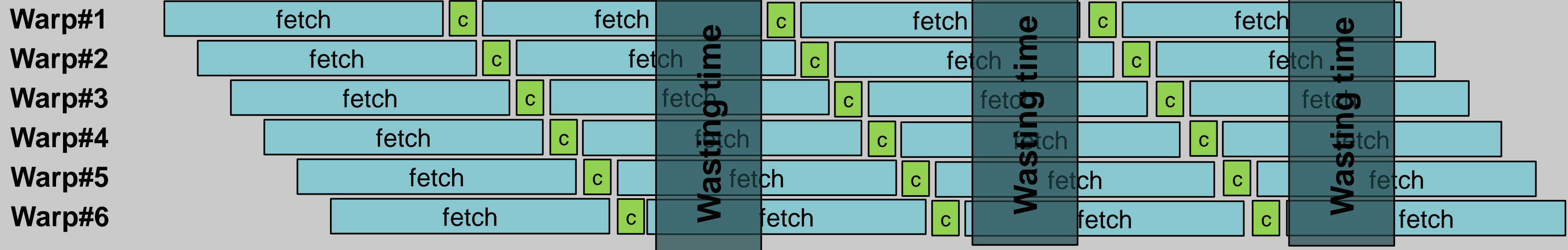
Latency hiding

CPU



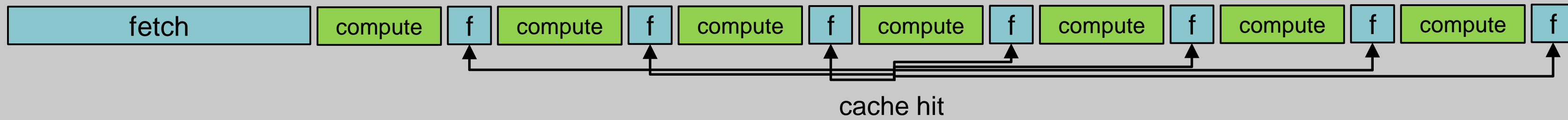

```
for(int i=0; i<…; i++)
{
  a = <fetch data>;
  b = compute(a);
}
```

GPU



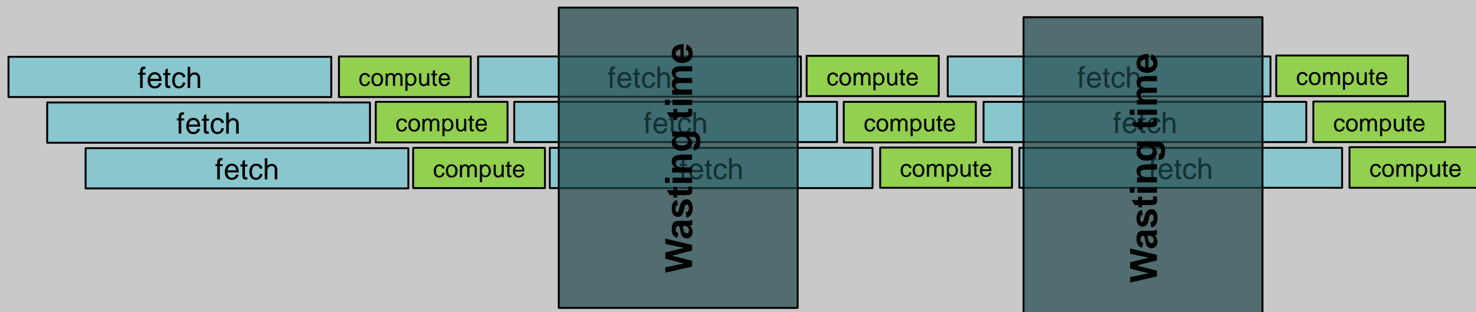
Latency hiding

CPU




```
for(int i=0; i<…; i++)
{
  a = <fetch data>;
  b = compute(a);
}
```

GPU

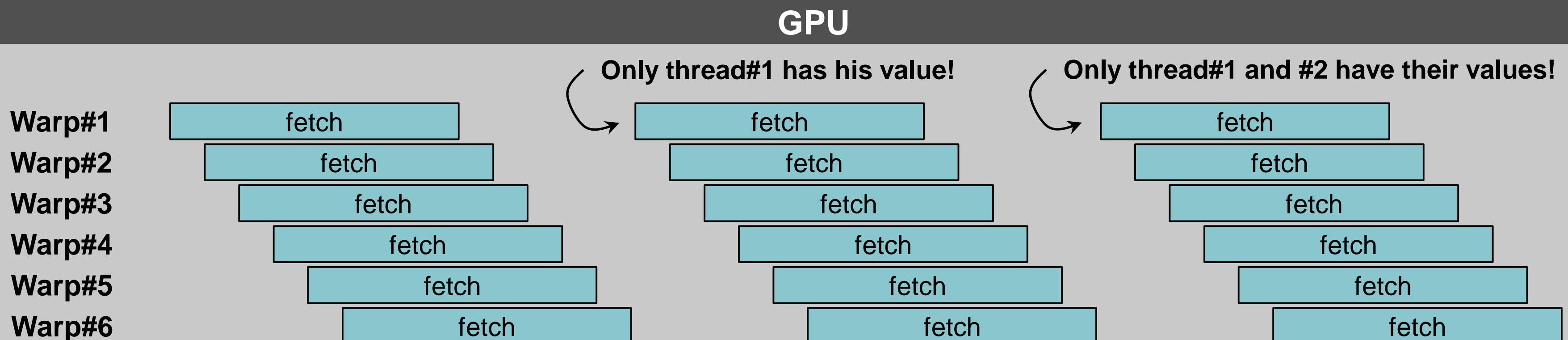


Context Switching

- In order for this approach to work, context switching must be very fast
  - No time to push state to memory (obviously)
- All threads must be live
  - Need huge register file
  - Need to keep number of used registers down (for full occupancy < 32 registers / thread)
  - That's really hard (consider any recursive algorithm…).

Coalesced Reads/Writes

- Scattered reads or writes within a warp can be very expensive.
  - If all threads in a warp read consecutive 32 bit words they will be *coalesced* into as few fetches as possible.
  - If they cannot be coalesced…



Shared Memory

- Each SMX has 64kb that can be used either as L1 cache or as fast scratchpad (shared) memory.
  - It's not much (32 bytes per thread at full occupancy)
  - But very useful if e.g. each block (group of warps) can cooperate on something.

Atomic Operations and Synchronization

- If two threads write to the same memory location, the result is undefined.
- But most modern GPUs support *atomic operations*
  - E.g. atomicAdd(float *mem, float val) will safely add val to the value in *mem.
  - Works on both shared and global memory
  - Should be used with care though, don’t serialize your program!
- We can also synchronize threads
  - (in CUDA kernel) __syncthreads() waits for all threads in a *block* to complete
  - (in CUDA host code) cudaDeviceSynchronize() waits for all running kernels and memory transfers to complete.

Shuffle and vote instructions

- Additionally, most GPUs and APIs support "shuffle" and "vote" instructions
  - These allow threads *within a warp* to exchange information without going through memory
  - E.g:

```
// Everyone get the value that the thread to the left has
int val_left = __shfl(val, (threadIdx.x - 1) % 32);

// Find out what all threads think of something
uint32_t bitmask = __ballot(pixel_covered);
```

Comparison

CPU	GPU
• Large Cache – Pre-fetch and retain data.	• More threads. – Do something else (useful)
• Branch Prediction – Pipeline instruction past.	• More threads. – Do something else.
• Instruction re-order. – Find independent.	• More threads. – Do something else.
• High clock speed.	• Low Clock speed.
• Lower efficiency – The above is not free! • Higher single thread throughput	• Higher efficiency – ALUs are kept busy. • Higher total throughput

Suggested Reading

- Pipeline and Hardware:
  - A trip through the Graphics Pipeline
    Fabian Giesen
    Very detailed text on what is publicly known about graphics hardware
    https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/
  - Render Hell – Book II
    Simon Schreibt
    Haven't read but Dan recommends it as a friendlier introduction to the graphics pipeline
    https://simonschreibt.de/gat/renderhell-book2/
- NVIDIA GeForce Whitepapers (starting from 8800GTX)
  Quite a lot of interesting information hidden in quite a lot of marketing nonsense
  Can't seem to find a collection, google e.g., "GTX 980 Whitepaper"
- *A Brief History of Graphics*
  Not about GPU hardware at all, just how graphics have evolved in games. Sort of fun.
  https://www.youtube.com/watch?v=QyjyWUrHsFc

Outline

Part 1: Motivation

If you're not into hardware, why should you care?

Part 2: History lesson

Where did GPUs come from, and what were they meant to do?

Part 3 GPU Architecture

What is a modern GPU, and how does it work?

Part 4: General Purpose GPU Programming

A little bit about non-gfx GPU programming

Example #1

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Example #1

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Example #1

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Example #1

```
__global__ void square_kernel(int N, float *d_data)
{
	int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
	if(thread_id >= N) return;
	d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
	float *d_data;
	cudaMalloc(&d_data, N * sizeof(float));
	cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
	dim3 block_dim(256);
	dim3 grid_dim(N/block_dim.x + 1);
	square_kernel<<<grid_dim, block_dim>>>(N, d_data);
	cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
	cudaFree(d_data);
}
```

Example #1

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Example #1

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Example #1

```
__global__ void square_kernel(int N, float *d_data)
{
	int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
	if(thread_id >= N) return;
	d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
	float *d_data;
	cudaMalloc(&d_data, N * sizeof(float));
	cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
	dim3 block_dim(256);
	dim3 grid_dim(N/block_dim.x + 1);
	square_kernel<<<grid_dim, block_dim>>>(N, d_data);
	cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
	cudaFree(d_data);
}
```

Example #1

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Example #1

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Example #1

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Example #1

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Example #1

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Will this be faster than just squaring on the CPU?

```
__global__ void square_kernel(int N, float *d_data)
{
	int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
	if(thread_id >= N) return;
	d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
	float *d_data;
	cudaMalloc(&d_data, N * sizeof(float));
	cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
	dim3 block_dim(256);
	dim3 grid_dim(N/block_dim.x + 1);
	square_kernel<<<grid_dim, block_dim>>>(N, d_data);
	cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
	cudaFree(d_data);
}
```

Will this be faster than just squaring on the CPU? NO!

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Will this be faster than just squaring on the CPU? NO!

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Allright… the actual kernel call. Will that run at full speed?

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Allright... the actual kernel call. Will that run at full speed?

```
__global__ void square_kernel(int N, float *d_data)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        d_data[thread_id] = d_data[thread_id] * d_data[thread_id];
}

void square(int N, float *h_data)
{
        float *d_data;
        cudaMalloc(&d_data, N * sizeof(float));
        cudaMemcpy(d_data, h_data, N * sizeof(float), cudaMemcpyHostToDevice);
        dim3 block_dim(256);
        dim3 grid_dim(N/block_dim.x + 1);
        square_kernel<<<grid_dim, block_dim>>>(N, d_data);
        cudaMemcpy(h_data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
        cudaFree(d_data);
}
```

Nah...

- Not enough compute to hide latency
- Probably not enough independent instructions to fully utilize SMX
- Only doing SP FP and Load/Store, so we CAN'T fully utilize SMX
- But fully coalesced reads and writes
- And certainly faster than CPU

Example #2: Reduction

- Say we want to find the maximum of N elements

```
array = [12 1 6 65 3 87 55 2 85 … 23]
```

- CPU code is extremely simple:

```
result = 0;
for(int i=0; i<N; i++) { result = max(result, array[i]; }
```

- But how to do this on the GPU?

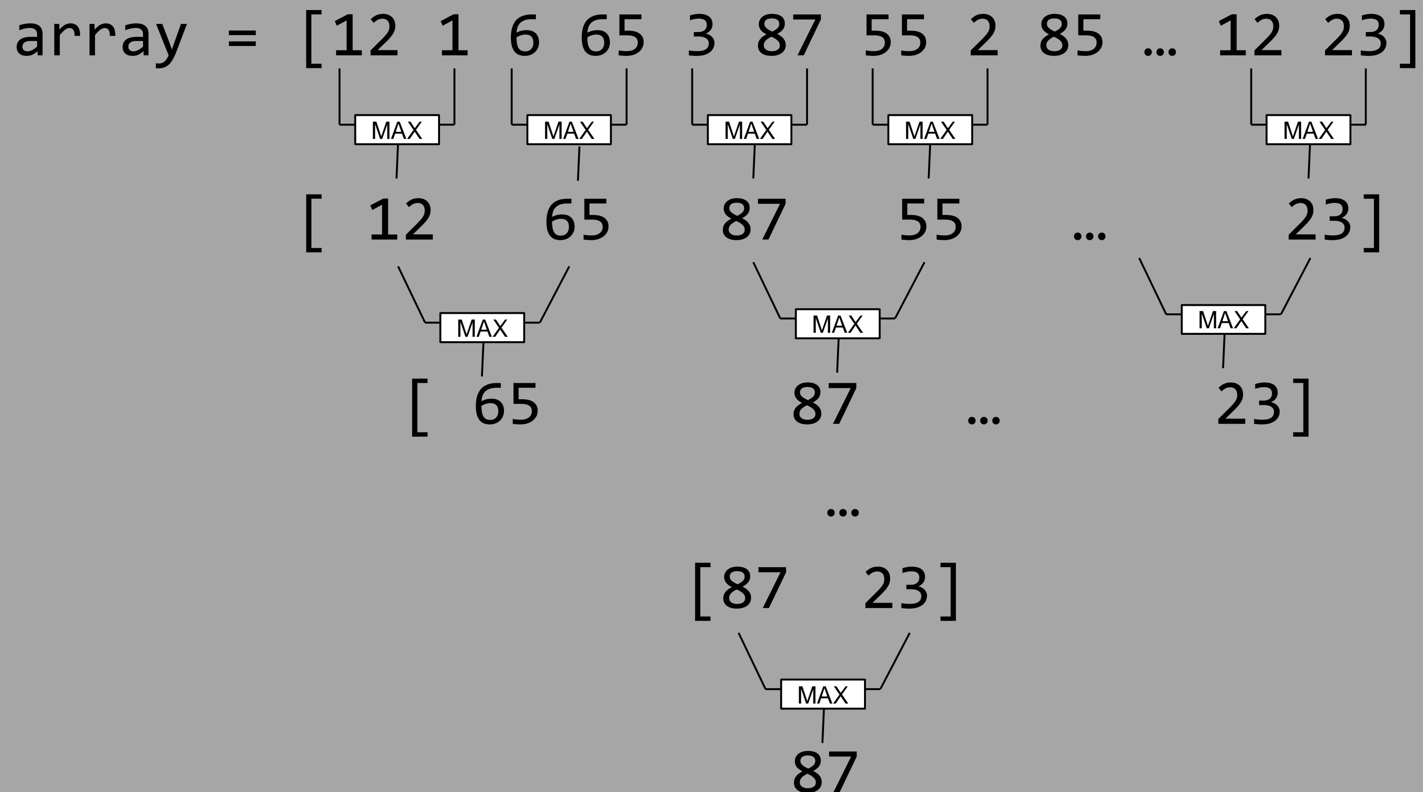
Example #2: Reduction

- An idiots approach:

```
__global__ void find_max(int N, float *d_data, float *d_result)
{
        int thread_id = blockIdx.x * blockDim.x + threadIdx.x;
        if(thread_id >= N) return;
        atomicMax(*d_result, d_data[thread_id];
}
```

Example #2: Reduction

- A slightly smarter approach:

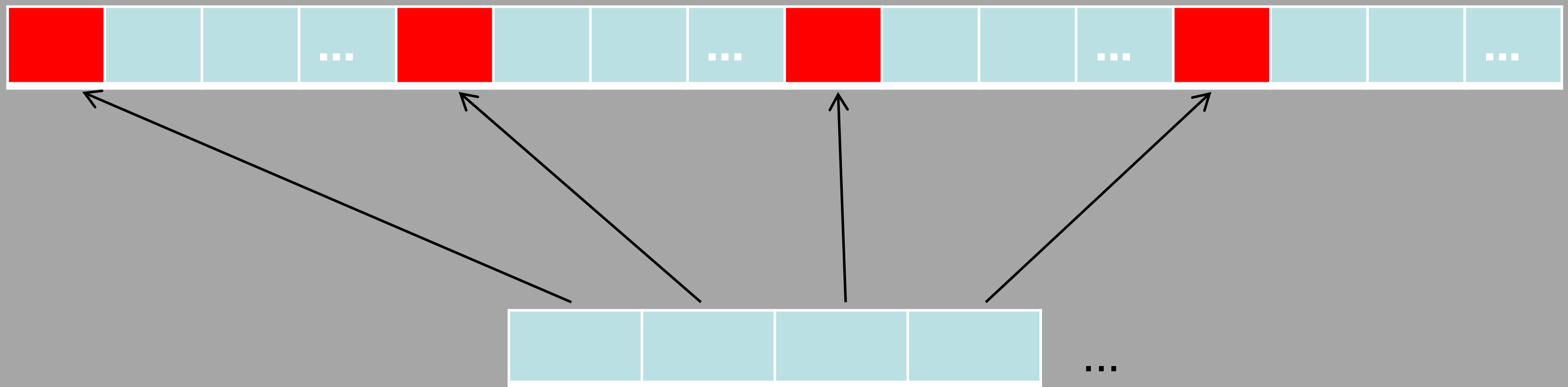


Example #2: Reduction

- An even smarter approach:
- We want to have at least 30k threads running.
  - Less and the GPU will be underutilized
  - More is okay, but no benefit
- Divide the input into 30k parts
- Let each thread do a sequential reduction over his part
- Output the result into a 30k array.
- Reduce that with the previous method.

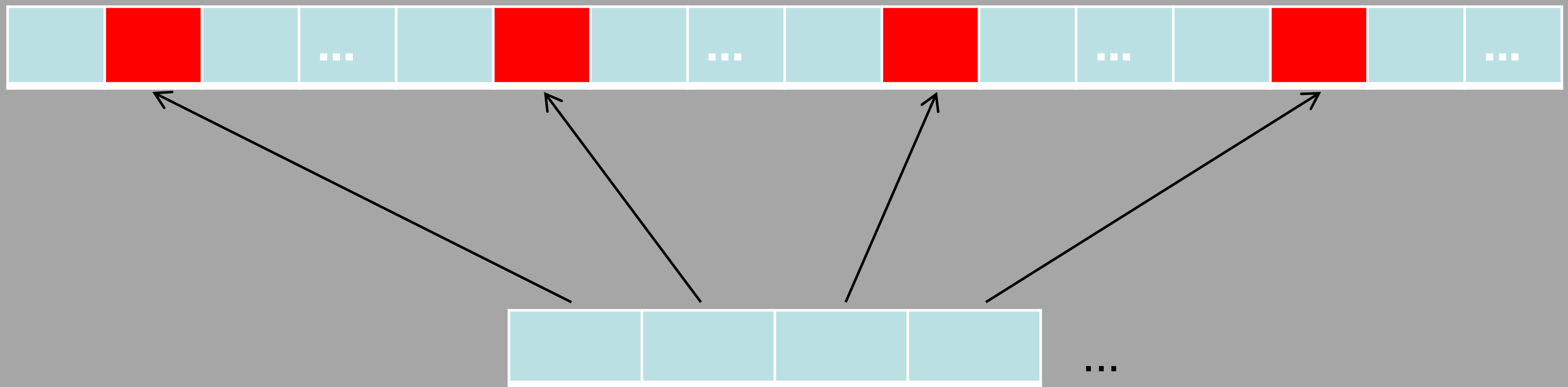
- Is this a good idea?

Example #2: Strided Memory Access



Warp = 32 threads
First memory request

Example #2: Strided Memory Access



Warp = 32 threads
Second memory request

Example #2: Strided Memory Access

```
// Divide input into 960 (#warps) parts instead.

start = warp_id * (N/960);
end = (warp_id + 1) * (N/960);
result = 0;
for(int i=start + threadIdx.x; i<end; i+=32) {
  result = max(result, d_data[i]);
}

result = warpReduce(result);

if(threadIdx.x == 0) d_output[warp_id] = result;
```

Conclusion

- Seems difficult?
  - It is! Turning your problem into a massively parallel one can be very hard (sometimes impossible).
- But so is CPU programming (if you want full speed)
  - Today, you need to at least parallelize enough to use all cores *and SIMD lanes*.
  - CPUs and GPUs are merging quickly.
    - Most recent CPUs have a *large* GPU part
    - Future intel CPUs will have GPUish SIMD
- You need to learn this stuff!
  - Luckily, you'll realize it's also much fun!