GPUs

Jiamin Gan & Bokai Zhang

Overview

- Background
- Programming Model
- CUDA Architecture
- Flow Control
- Memory Hierarchy
- Recent Developments

Background

- First GPU: GeForce 256 (1999)
  - 5M transistors
- First programmable GPU: GeForce 3 (2001)
  - 60M transistors
  - Languages: DX8 and OpenGL
  - Execute vertex shader and pixel shader programs
- Common architecture:
  - NVIDIA: Fermi(2010), Kepler(2012), Maxwell(2014), Pascal(2016), Volta(2017), Turing(2018), Ampere(2020), Hopper(March, 2022)[1]
  - AMD: TeraScale, GCN, RDNA, etc.
- Fastest datacenter GPU (till 10/2020):
  - A40 GPU accelerator: (FP32) 37.4 TFLOPS
  - 10,752 CUDA cores (28.3 billion transistors) & 48 GB GDDR6 memory

1. Everything You Need to Know About GPU Architecture and How It Has Evolved https://www.cherryservers.com/blog/everything-you-need-to-know-about-gpu-architecture

GPU - A Computing Accelerator

- Separate piece of Device
- Specialized hardware
- Connected to CPU through PCIe Bus
- Have its own memory (usually)

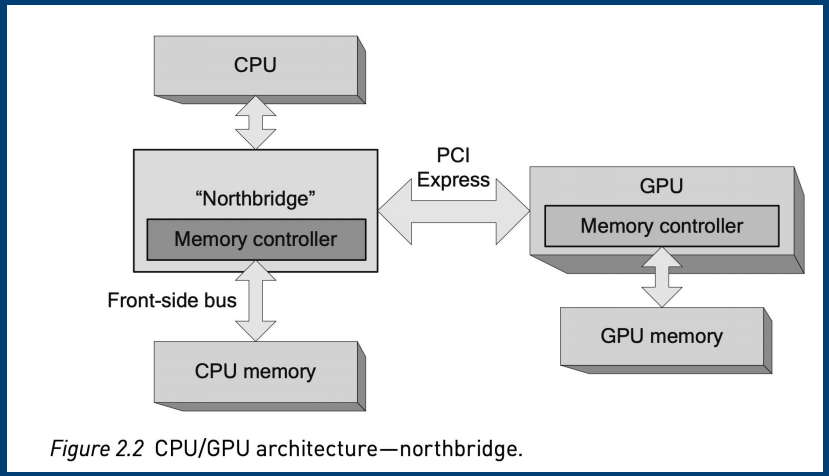


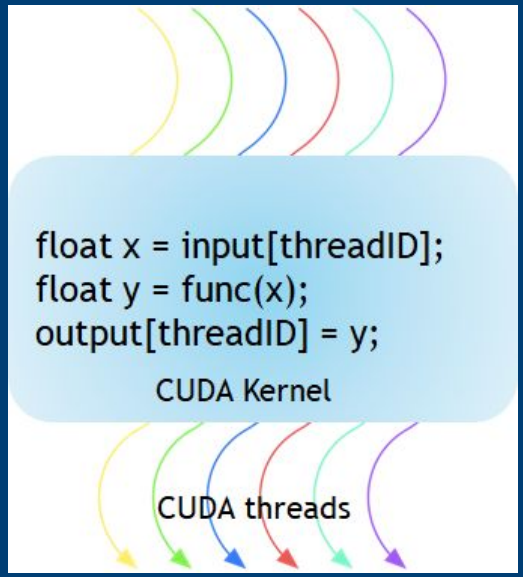
Figure 2.2 CPU/GPU architecture—northbridge.

Programming Model

Programming on GPUs

A typical CUDA program:

1. Copy memory from CPU to GPU
2. Launch a predefined kernel
    a. Each thread will execute this kernel code
    b. Thread have its identifier (similar to thread id)
3. Copy result from GPU back to CPU



[5]

Programming on GPUs - Example

Kernel Definition (saxpy: Single-precision A * X Plus Y)

```
3    __global__
4    void saxpy(int n, float a, float *x, float *y)
5    {
6      int i = blockIdx.x*blockDim.x + threadIdx.x;
7      if (i < n) y[i] = a*x[i] + y[i];
8    }
```

Programming on GPUs - Example (cont)

Memory (and Data) Initialization

```
int N = 1<<20;
float *x, *y, *d_x, *d_y;
x = (float*)malloc(N*sizeof(float));
y = (float*)malloc(N*sizeof(float));
cudaMalloc(&d_x, N*sizeof(float));
cudaMalloc(&d_y, N*sizeof(float));
for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
}
```

Programming on GPUs - Example (cont)

Memory Copying and Kernel Execution

```
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

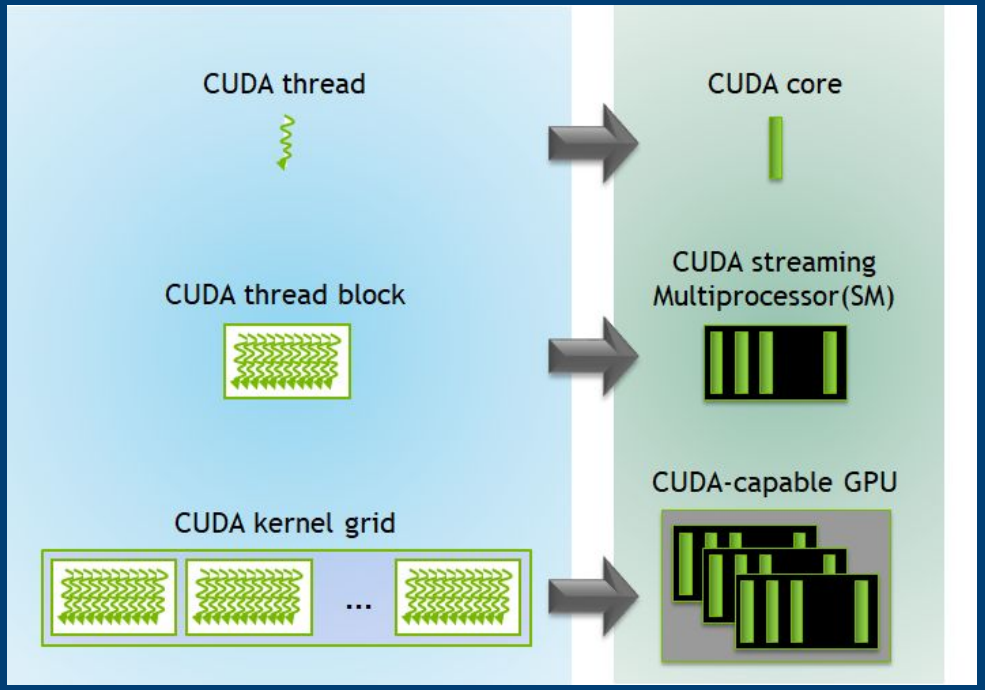
Cleanup

```
cudaFree(d_x);
cudaFree(d_y);
free(x);
free(y);
}
```

Programming on GPUs (cont)

Thread Organization

- Thread Block
  - Collection of warps (multiple threads)
  - All threads in same block have shared memory
- Grid
  - Collection of thread blocks
  - All thread blocks in same grid have shared memory

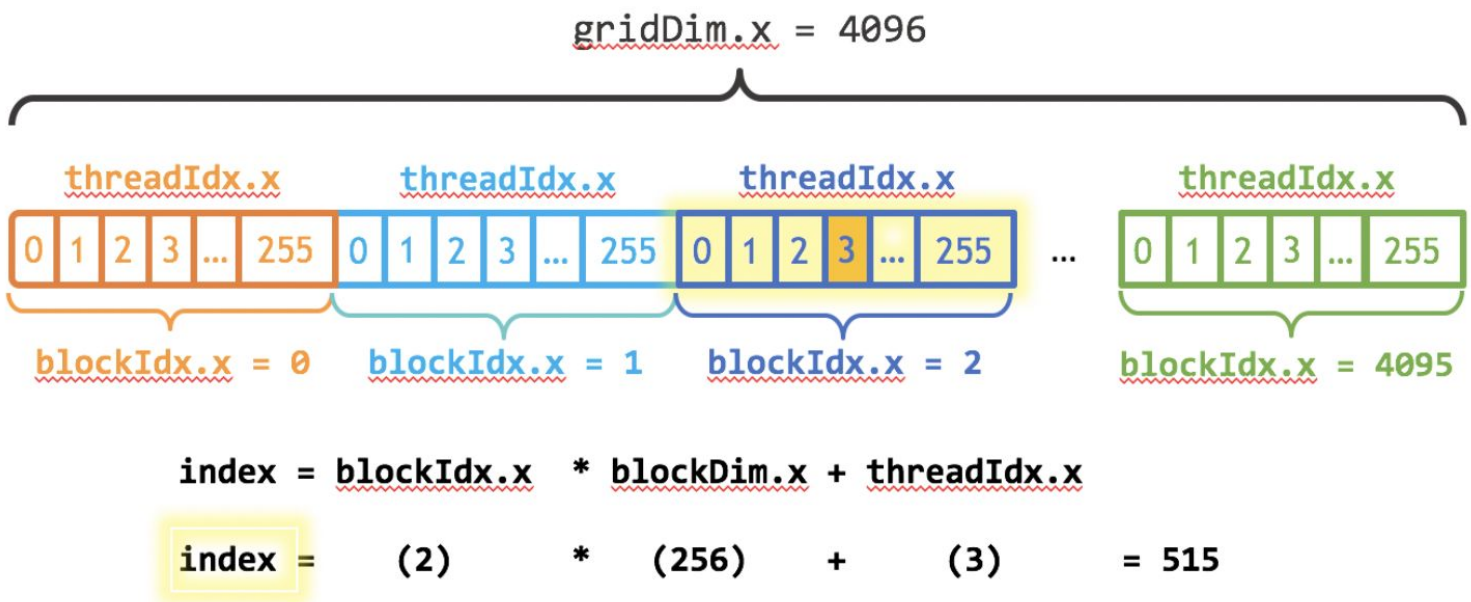


[5]

Programming on GPUs (cont)

```
int blockSize = 256;
int numBlocks = (N + blockSize - 1) / blockSize;
add<<<numBlocks, blockSize>>>(N, x, y);
```

→ =⌈N/blocksize⌉



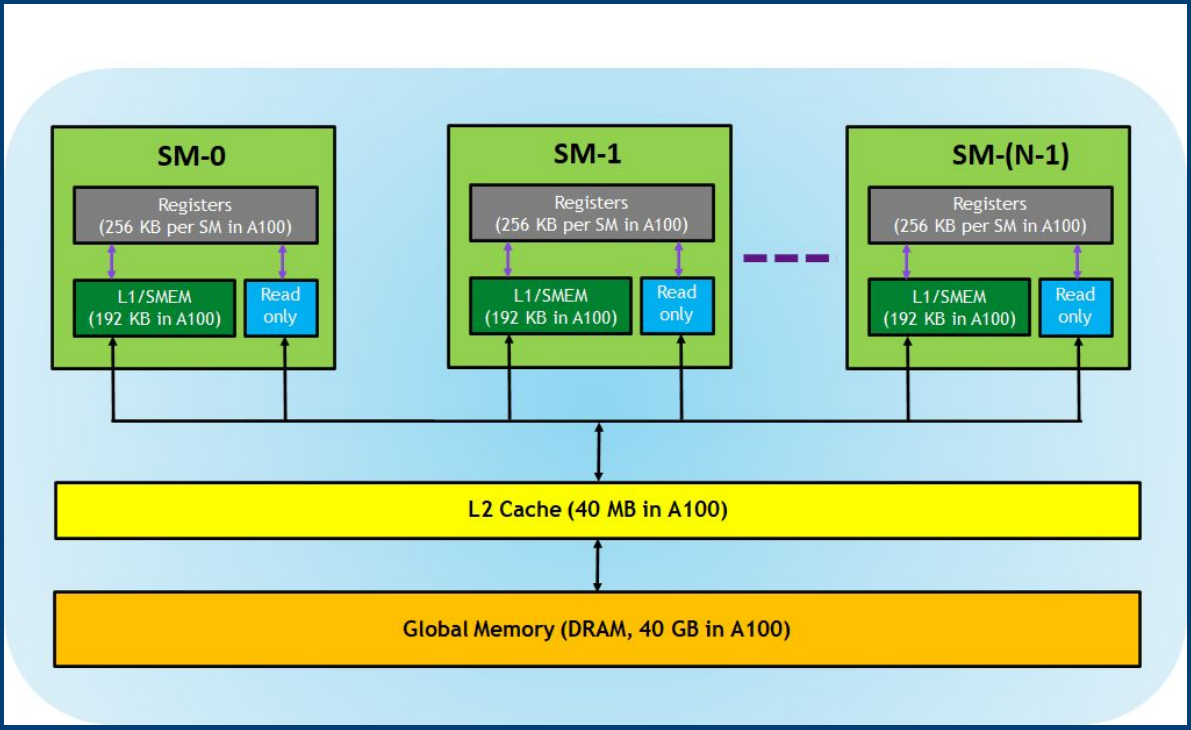
[4]

GPU Architecture

Example based on Nvidia Fermi Architecture and GF100

GPU - Architecture Overview

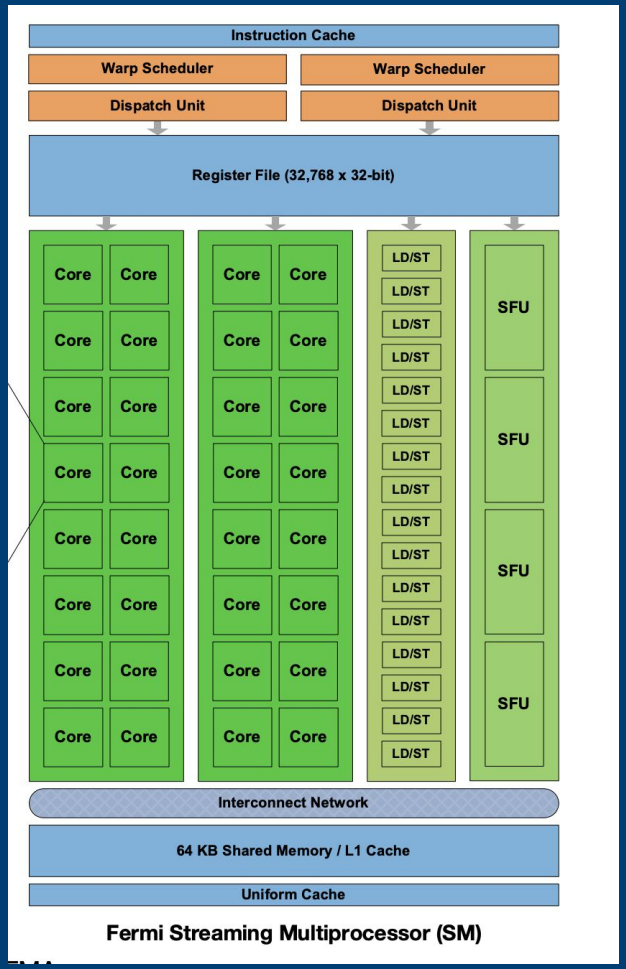
- Collection of Streaming Multiprocessor



[5]

Streaming Multiprocessor

- Consists of massive cores and independent load/store units
- Contains special function units
  - Execute transcendental instructions such as sin, sqrt, Tensor Operations.
- Shared L1 cache
- Lots of registers

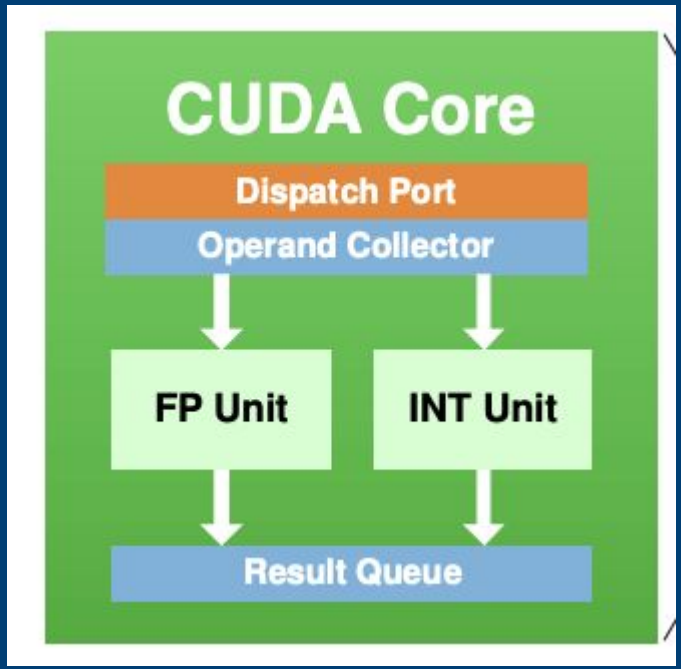


[3]

- Acronym
  - LD/ST: Load Store Units
  - SFU: Special Function Units

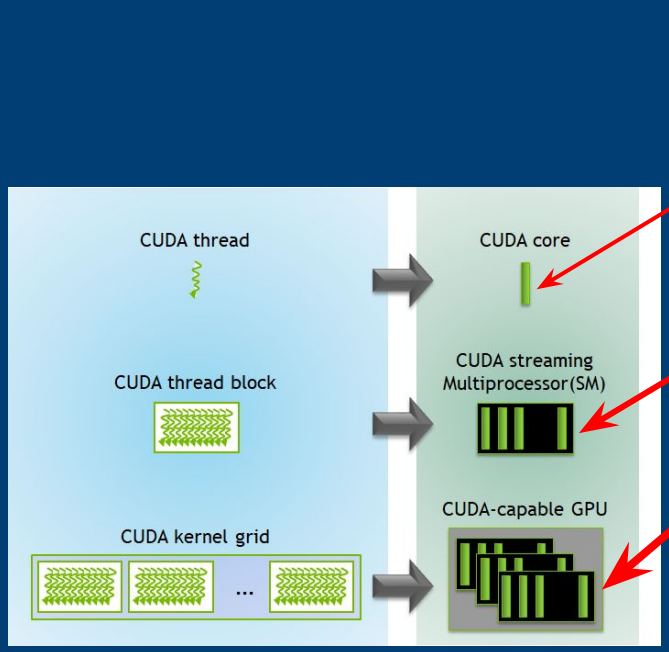
Streaming Processor (or CUDA core)

- Underlying computing hardware
- Takes instruction from warp scheduler and dispatch unit
- Executes through the ALU units(FP and INT in this case)
- Comparing to a CPU core
  - Almost no control logic
  - Much cheaper to build
  - More like an ALU(but they call it a core)
- Can have specialized hardware for specific computations(not necessarily in CUDA core)
  - Single/Double Precision Floating Point Unit
  - Ray Tracing Core, Tensor Core
  - SFUs



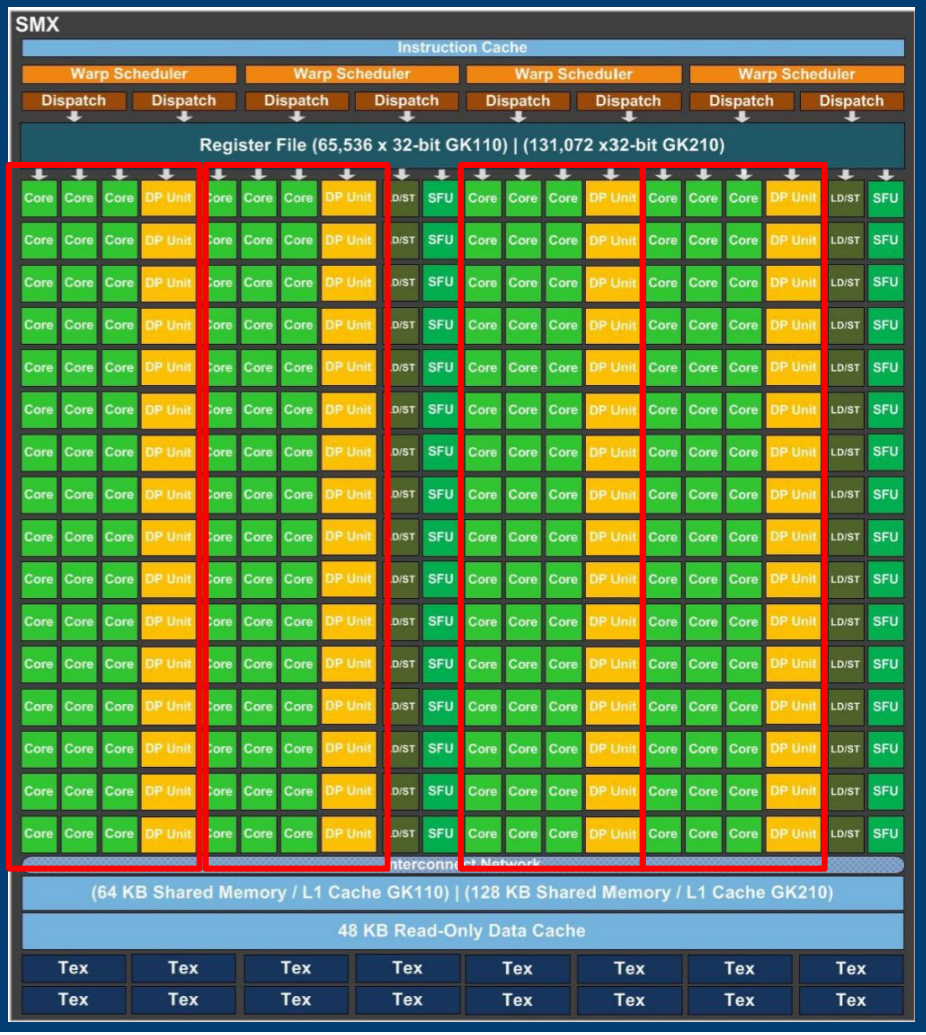
Fermi Architecture [3]

GPU Hardware vs CUDA Model



Streaming Multiprocessor - Warp

- Each SM(thread blocks) have multiple warps
- Example: Kepler GK110
  - 32 threads/core each warp
- Each warp is under Single Instruction Multiple Thread(SIMT) model
  - Warp scheduler give same instruction to each core

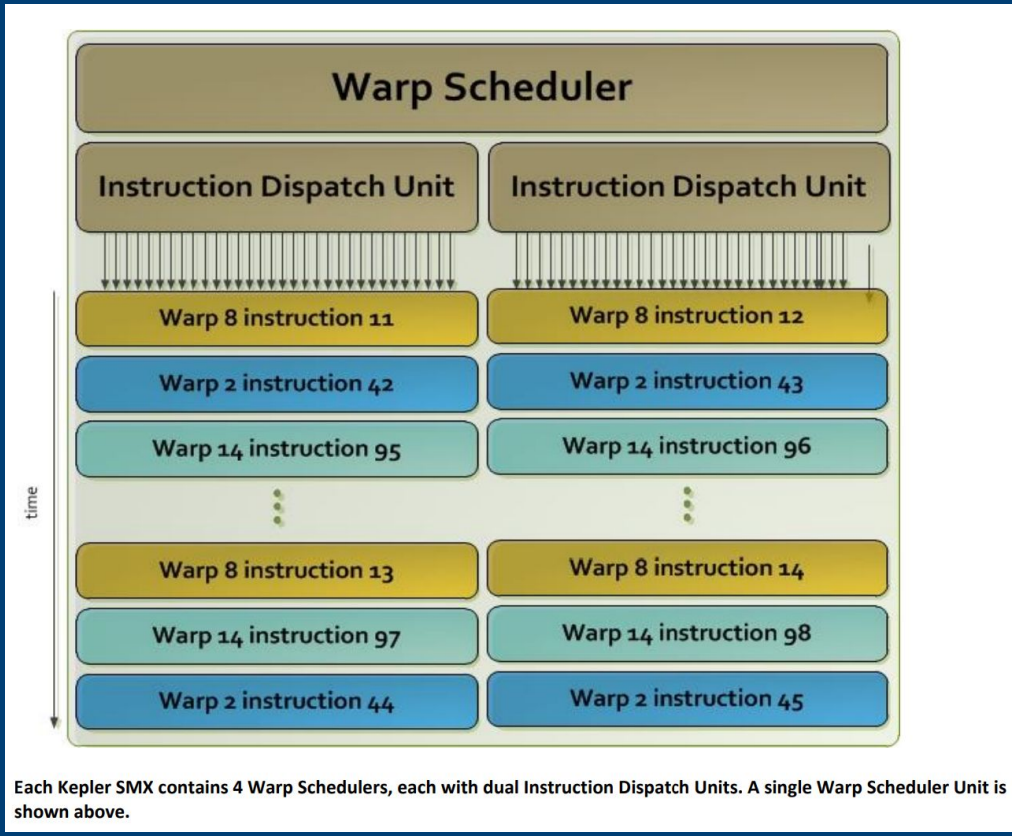


[8]

- Acronym
  - DP Units: Double Precision Units
  - LD/ST: Load Store Units
  - SFU: Special Function Units
  - Tex: Texture Memory

Streaming Multiprocessor - Warp[contd]

- Hardware Warp Scheduler
  - Dispatch specific instructions by each Instruction Dispatch Unit
- All threads in one warp executes the same instruction



Each Kepler SMX contains 4 Warp Schedulers, each with dual Instruction Dispatch Units. A single Warp Scheduler Unit is shown above.

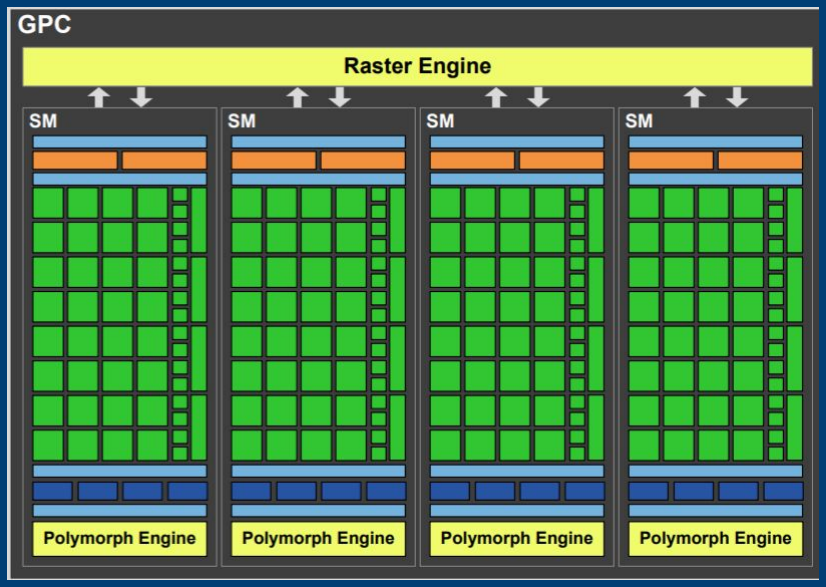
[8]

Graphic Processing Cluster(GPC)

- Cluster for specific graphic operations
  - Basically a collection of (<4) SMs with other aiding hardware
  - Not physically organized(flexible)
- Raster Engine
  - Triangle setup, rasterization, and Z-cull

    //Some Graphic Operation
- Polymorph Engine
  - Vertex attribute fetch and tessellation

    //Some Graphic Operation



GF 100 Graphics Processing Cluster (GPC)[8]

GPU Flow Control
- Divergent Branching

Control Flow Problem In SIMT

```
1   do {
2     t1 = tid*N;
3     t2 = t1 + i;
4     t3 = data1[t2];
5     t4 = 0;
6     if( t3 != t4 ) {
7       t5 = data2[t2];
8       if( t5 != t4 ) {
9         x += 1;
10      } else {
11        y += 2;
12      }
13    } else {
14      z += 3;
15    }
16    i++;
17  } while( i < N );
```

A

B

C

D

E

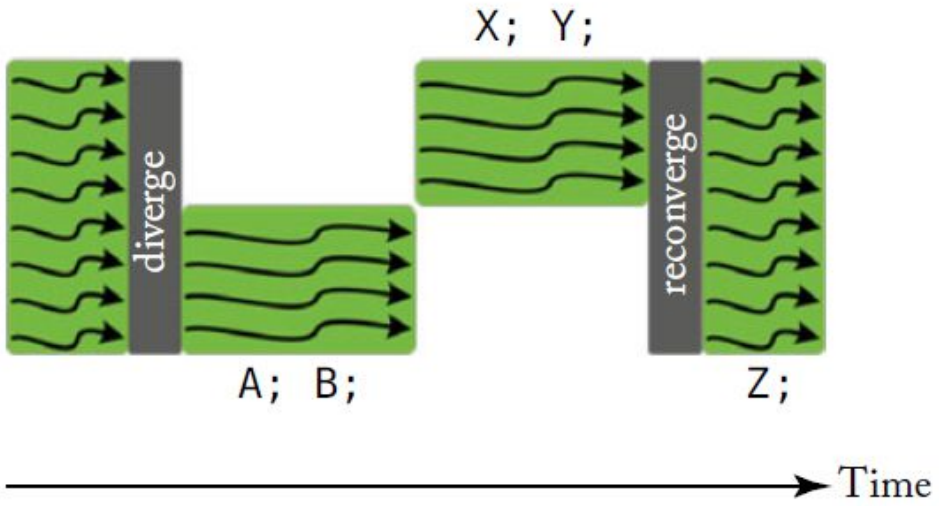
F

G

- Consider this piece of CUDA kernel Code
- A, B, C, D, E, F, G are all instruction address(think of them in assembly)
- Different control path can be taken by different thread (depend on their tid)
- What is the problem?

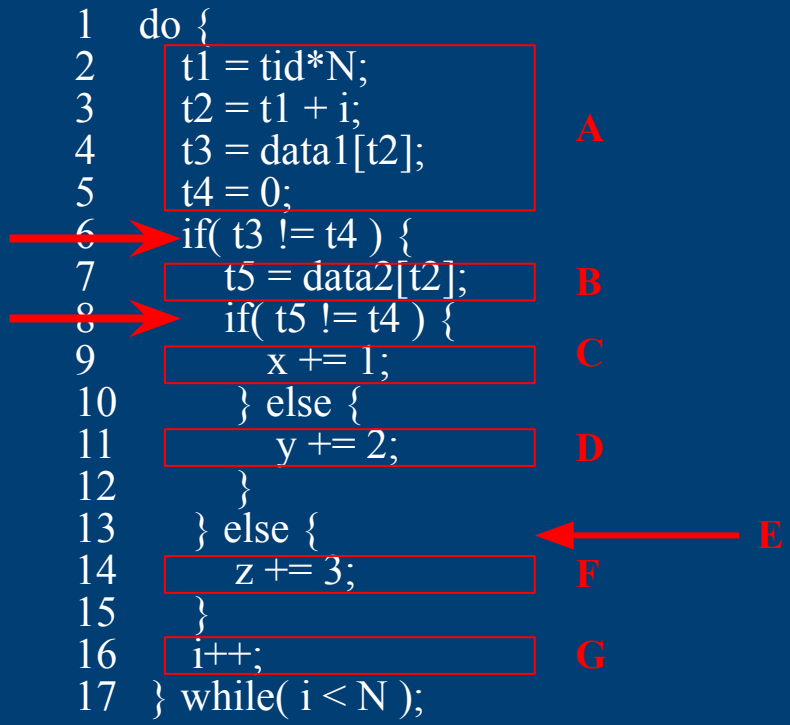
Divergent Branching - Simpler Example

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



[1]

Control Flow Problem In SIMT



- Consider this piece of CUDA kernel Code
- A, B, C, D, E, F, G are all instruction address(think of them in assembly)
- Different control path can be taken by different thread (depend on their tid)
- What is the problem?
  - A shared PC and fetched instruction in all CUDA cores!

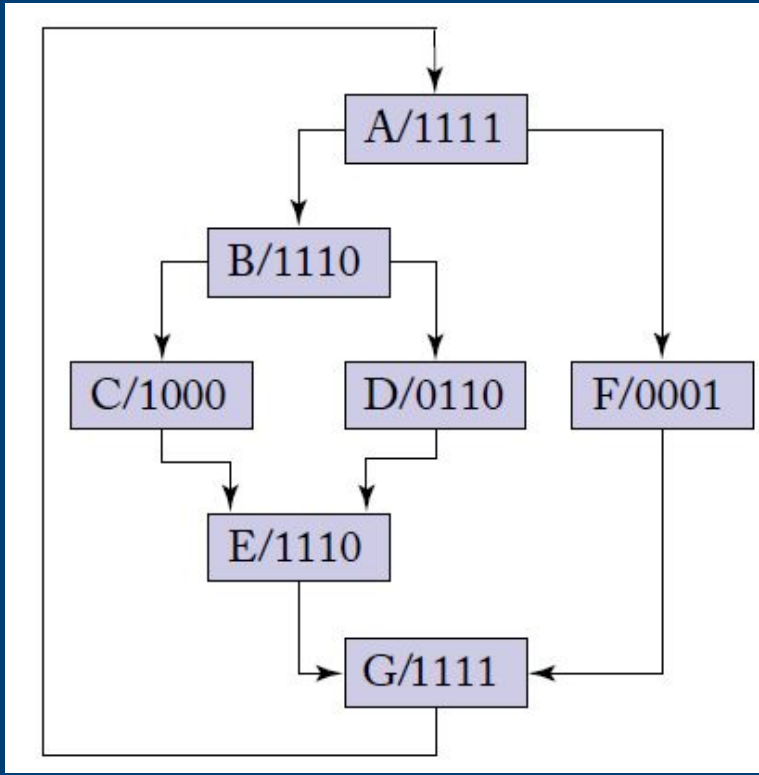
Control Flow Problem In SIMT

```
A:    mul.lo.u32      t1, tid, N;
      add.u32         t2, t1, i;
      ld.global.u32   t3, [t2];
      mov.u32         t4, 0;
      setp.eq.u32     p1, t3, t4;
@p1   bra             F;
B:    ld.global.u32   t5, [t2];
      setp.eq.u32     p2, t5, t4;
@p2   bra             D;
C:    add.u32         x, x, 1;
      bra             E;
D:    add.u32         y, y, 2;
E:    bra             G;
F:    add.u32         z, z, 3;
G:    add.u32         i, i, 1;
      setp.le.u32     p3, i, N;
@p3   bra             A;
```

[1]

- Consider this piece of CUDA kernel Code
- A, B, C, D, E, F, G are all instruction address(think of them in assembly)
- Different control path can be taken by different thread (depend on their tid)
- What is the problem?
  - A shared PC and fetched instruction in all CUDA cores!

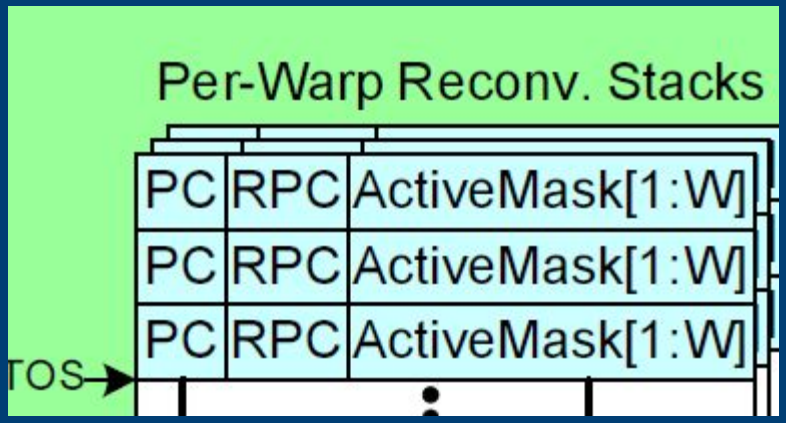
Control Flow Problem In SIMT



[1]

- Consider this piece of CUDA kernel Code
- A, B, C, D, E, F, G are all instruction address(think of them in assembly)
- Different control path can be taken by different thread (depend on their tid)
- What is the problem?
  - A shared PC and fetched instruction in all CUDA cores!

How to issue the instructions - SIMT Stack



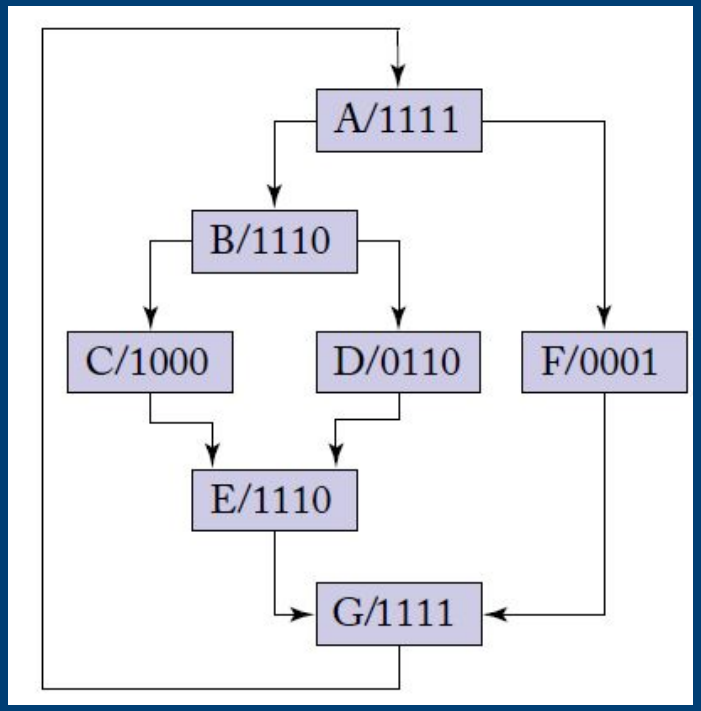
[7]

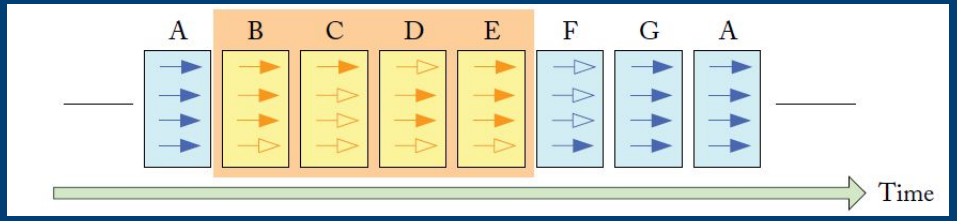
- Located for each warp
- What information should we keep track of
  - PC: Program Counter of the next instruction to execute
  - RPC: Reconvergence Program Counter
  - Active Mask: One bit for each thread, shows if this block of execution includes this thread or not

SIMT Stack

- Initially: Consist of one entry
  - RPC NULL
  - Next PC set to the first instruction in the function
  - Active Mask set to all 1s
- Executes the instructions starting at the PC of the top entry, until
  - Reached the RPC
    - In this case, pop the entry
  - Or a diverge branch happens
    - Needs more branches, leave the entry in the stack
- When a diverge branch happens:
  - Evaluate the RPC of those branches
  - Change the current top entry's next PC to the RPC of those branched instructions
  - Push the two branched entries into the stack with proper active masks

SIMT Stack - Example





[1]

1

#	RPC	PC	Active Mask
1	-	A	1111

TOS

2

#	RPC	PC	Active Mask
1	-	G	1111
2	G	F	0001
3	G	B	1110

New Entries

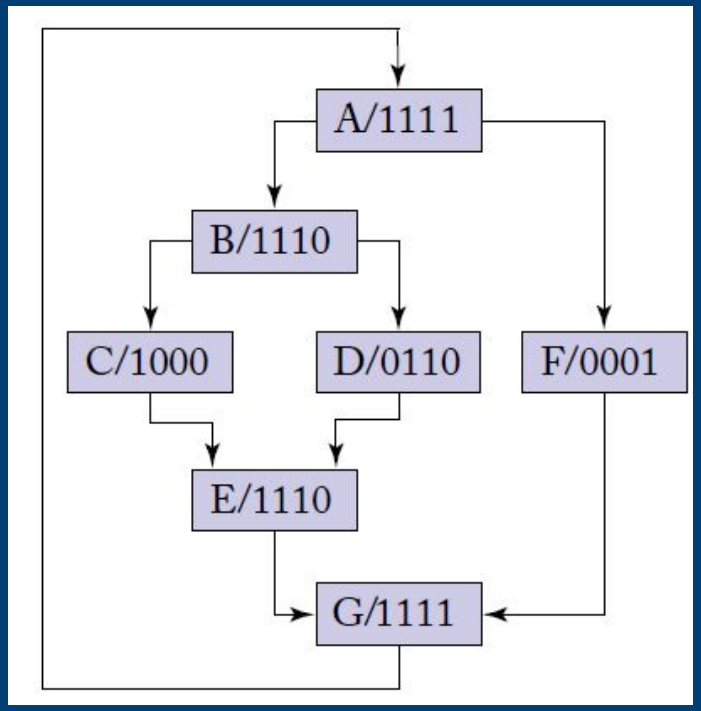
TOS

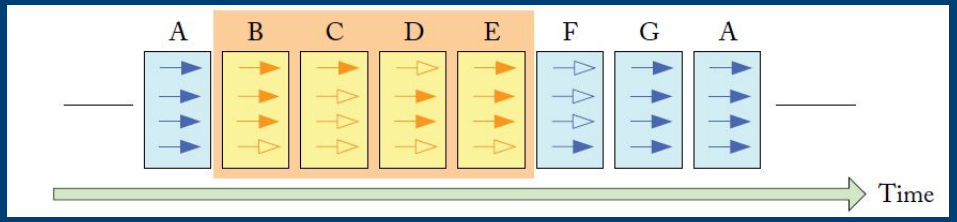
3

#	RPC	PC	Active Mask
1	-	G	1111
2	G	F	0001
3	G	E	1110
4	E	D	0110
5	E	C	1000

TOS

SIMT Stack - Example (cont)





[1]

3

#	RPC	PC	Active Mask	
1	-	G	1111	
2	G	F	0001	
3	G	E	1110	
4	E	D	0110	
5	E	C	1000	← TOS

4

#	RPC	PC	Active Mask	
1	-	G	1111	
2	G	F	0001	
3	G	E	1110	← TOS

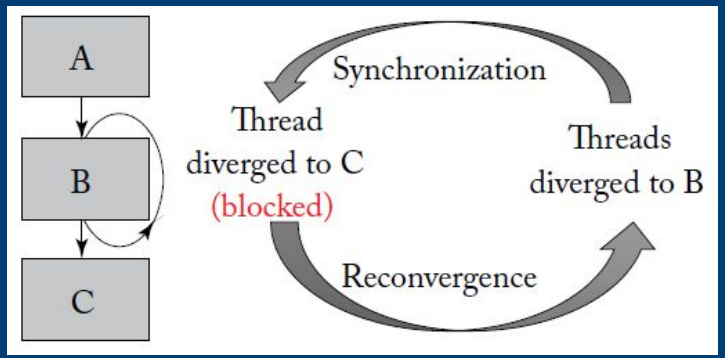
5

#	RPC	PC	Active Mask	
1	-	G	1111	← TOS

Naive SIMT Lock

- Consider this lock
- Threads diverges as different result from the while loop conditions
- C is set to the reconvergence point
- The one succeed to the lock still needs to wait for reconvergence of other threads, causing deadlock

```
A: *mutex = 0
B: while(!atomicCAS(mutex, 0  ,1));
C: // critical section
   atomicExch(mutex, 0  ) ;
```



[1]

GPU Memory Hierarchy

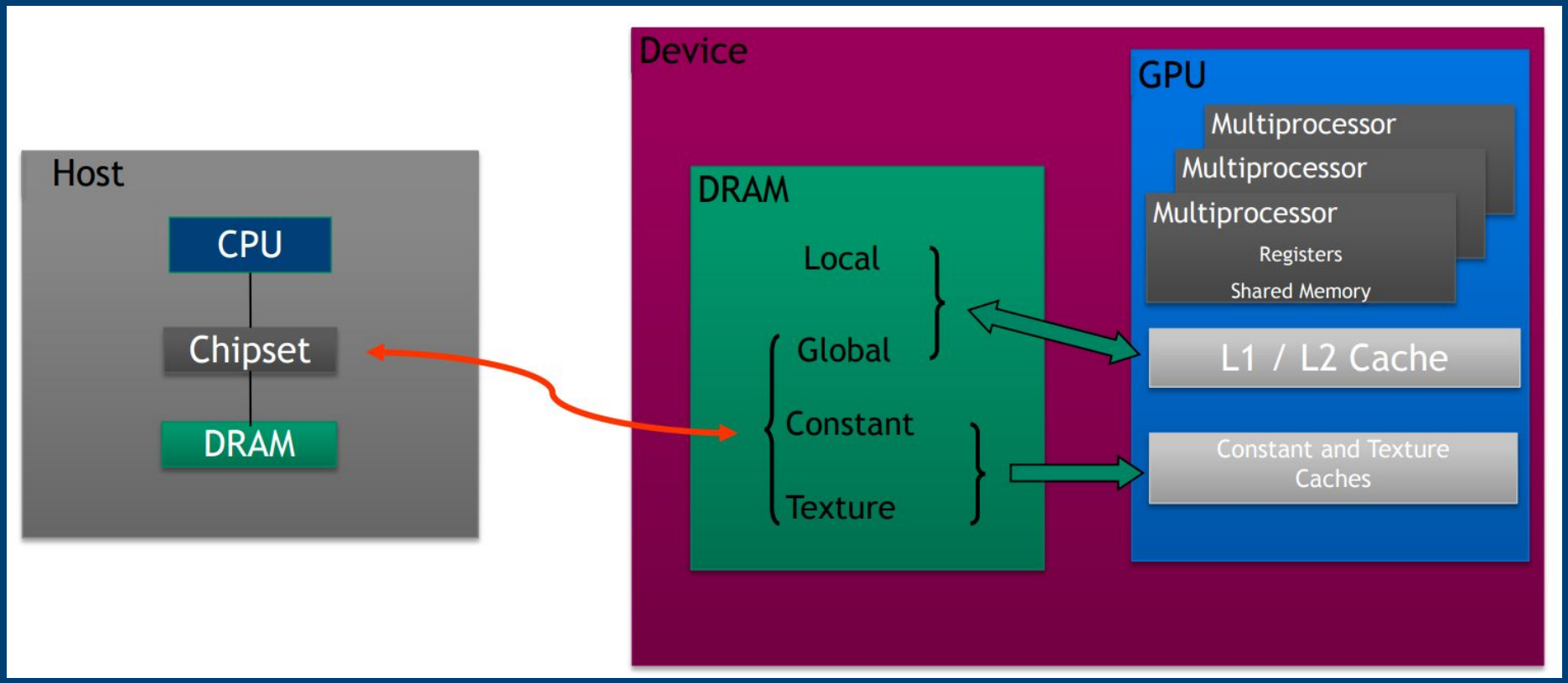
Memory Hierarchy

- Different types of GPU memory
- Comparison between memory for GPU and CPU
- Use of shared memory and texture memory
- First-level memory architectures
- Memory partition unit

Overview of Memory System

- Local: thread-private data determined by the compiler statically. (usually means registers and low-level cache)
- Shared (scratchpad): on-chip memory per thread block managed by programmers using __shared__ specifier.
- Constant: Read-only, Compiler-determined constant values.
- Texture: Read-only memory associated with textures (often used in streaming and rendering).
- Global: Other memory accessible for all blocks and the host.

Overview of Memory System



Size, Throughput, and Latency

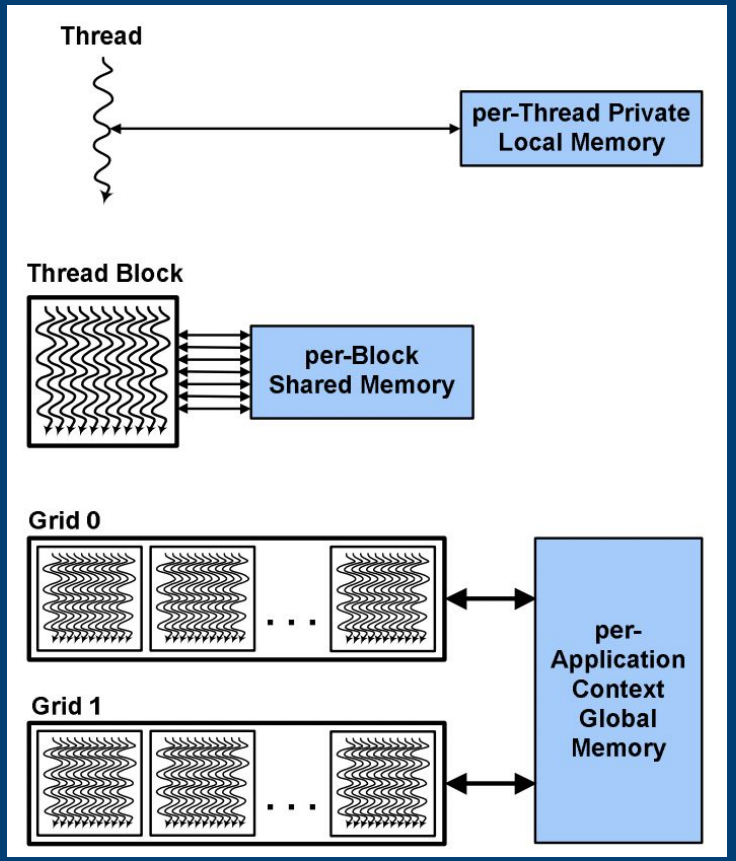
Local(registers): ~256KB per thread, ~1 cycle

Shared: 32 banks w/ 4 bytes per bank, >1TB/s, ~10 cycles

Constant: ~64KB, much slower than shared

Texture: > 100 texture units w/ 48KB per unit

Global: <900GB/s, >100 cycles



Compare Memory for GPUs and CPUs

- GPU:
  - Texture cache works well on streaming (read-only, high throughput, uniform latency).
  - Shared memory works well on intra-block communication.
- CPU:
  - L1d cache, L1i cache, and TLB are used for reducing memory latency.
  - L3 cache & RAM are used for sharing data among cores.

Using Shared Memory

__shared__ specifier to declare

void __syncthreads();

works as a barrier in a thread block

Note: CUDA does not have built-in mutex.

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // Synchronize (ensure all the data is available)
      __syncthreads();
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

Using Texture Memory

```
struct cudaTextureDesc
{
    enum cudaTextureAddressMode addressMode[3];
    enum cudaTextureFilterMode  filterMode;
    enum cudaTextureReadMode    readMode;
    int                         sRGB;
    int                         normalizedCoords;
    unsigned int                maxAnisotropy;
    enum cudaTextureFilterMode  mipmapFilterMode;
    float                       mipmapLevelBias;
    float                       minMipmapLevelClamp;
    float                       maxMipmapLevelClamp;
};
```

- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#texture-object-api

Using Texture Memory

Kernel Code:

```
// Simple transformation kernel
__global__ void transformKernel(float* output,
                                cudaTextureObject_t texObj,
                                int width, int height,
                                float theta)
{
    // Calculate normalized texture coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    float u = x / (float)width;
    float v = y / (float)height;

    // Transform coordinates
    u -= 0.5f;
    v -= 0.5f;
    float tu = u * cosf(theta) - v * sinf(theta) + 0.5f;
    float tv = v * cosf(theta) + u * sinf(theta) + 0.5f;

    // Read from texture and write to global memory
    output[y * width + x] = tex2D<float>(texObj, tu, tv);
}
```

Using Texture Memory

Host Code:

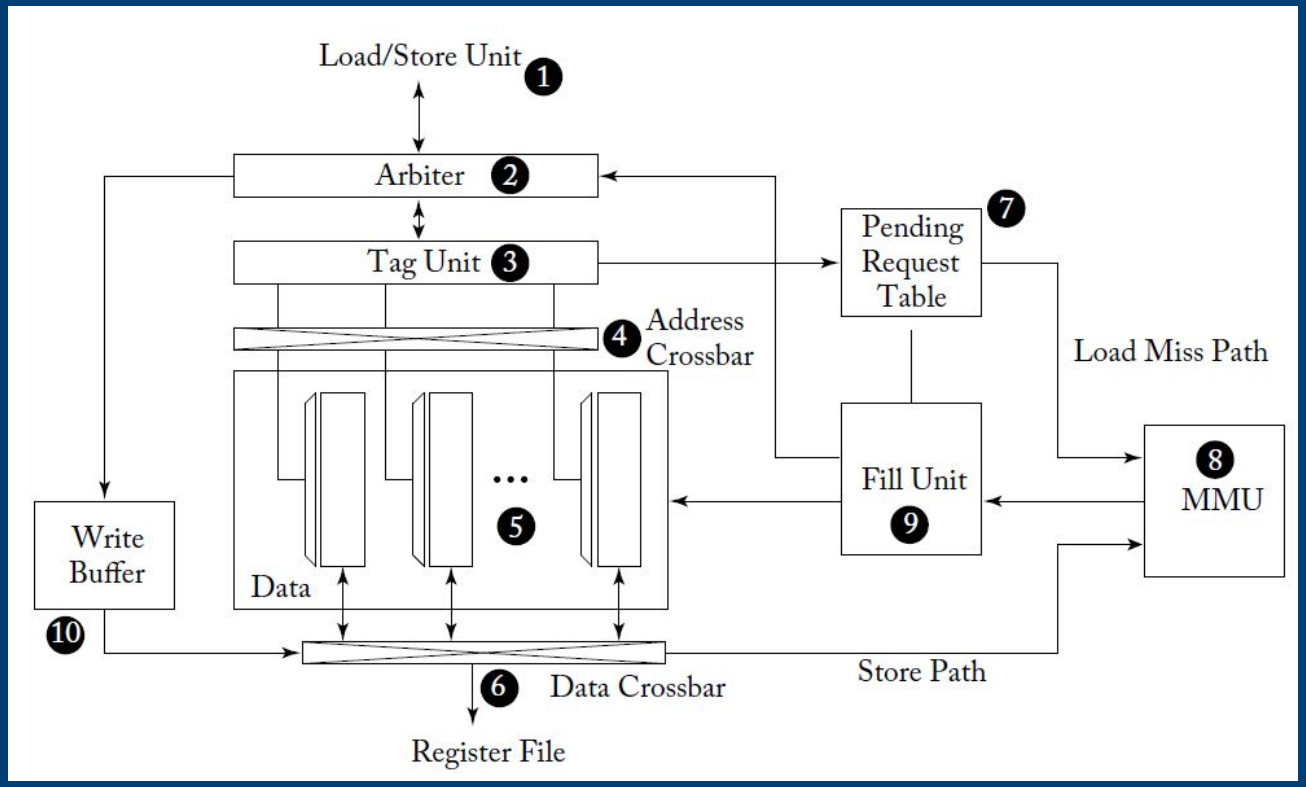
```
// Specify texture
struct cudaResourceDesc resDesc;
memset(&resDesc, 0, sizeof(resDesc));
resDesc.resType = cudaResourceTypeArray;
resDesc.res.array.array = cuArray;

// Specify texture object parameters
struct cudaTextureDesc texDesc;
memset(&texDesc, 0, sizeof(texDesc));
texDesc.addressMode[0] = cudaAddressModeWrap;
texDesc.addressMode[1] = cudaAddressModeWrap;
texDesc.filterMode = cudaFilterModeLinear;
texDesc.readMode = cudaReadModeElementType;
texDesc.normalizedCoords = 1;

// Create texture object
cudaTextureObject_t texObj = 0;
cudaCreateTextureObject(&texObj, &resDesc, &texDesc, NULL);
```

```
// Invoke kernel
dim3 threadsperBlock(16, 16);
dim3 numBlocks((width + threadsperBlock.x - 1) / threadsperBlock.x,
               (height + threadsperBlock.y - 1) / threadsperBlock.y);
transformKernel<<<numBlocks, threadsperBlock>>>(output, texObj, width, height,
                                                angle);
```

First-Level Memory Structures

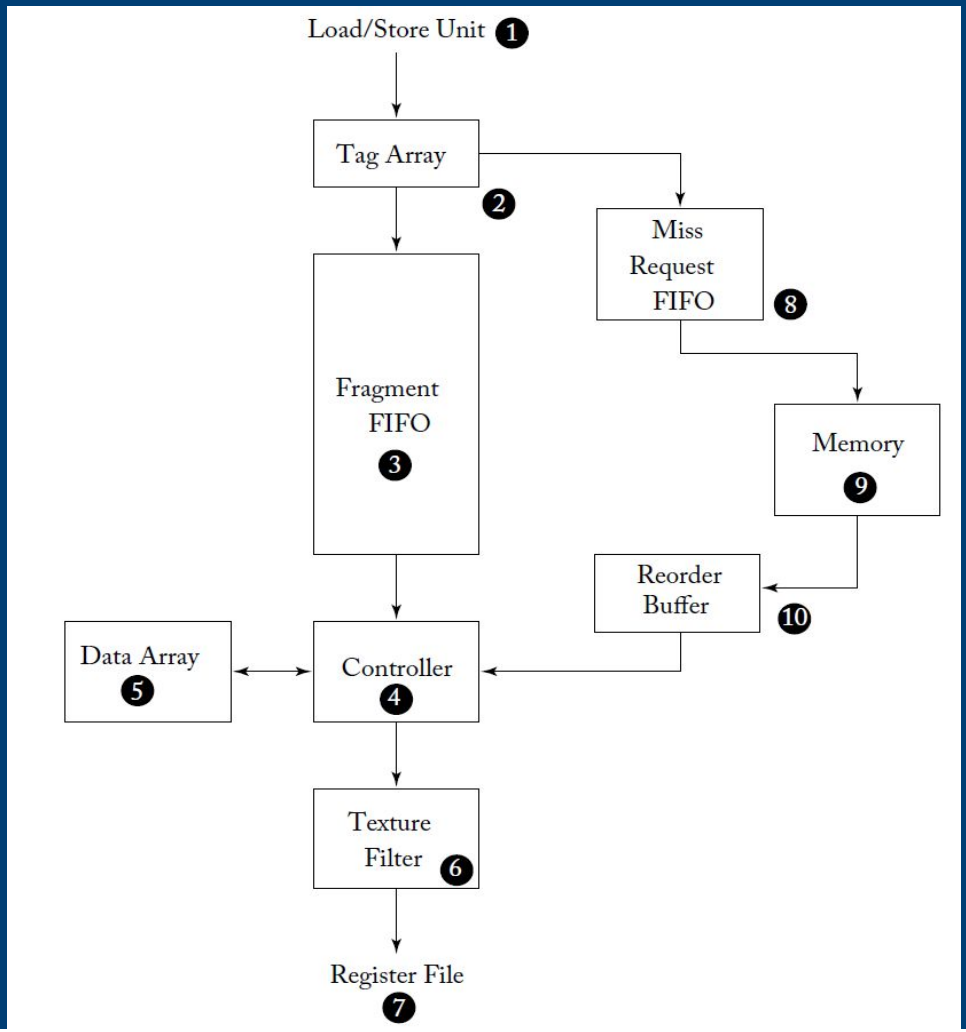
- Unified L1 data cache and shared memory
  - a subset of the global memory address space in the cache
  - shared memory access: bank conflicts (i.e. two threads want to access the same bank of shared memory at the same time)
  - cache read & write
- Texture cache
  - often used in streaming and rendering
  - High throughput, uniform latency

Unified L1 data cache and shared memory



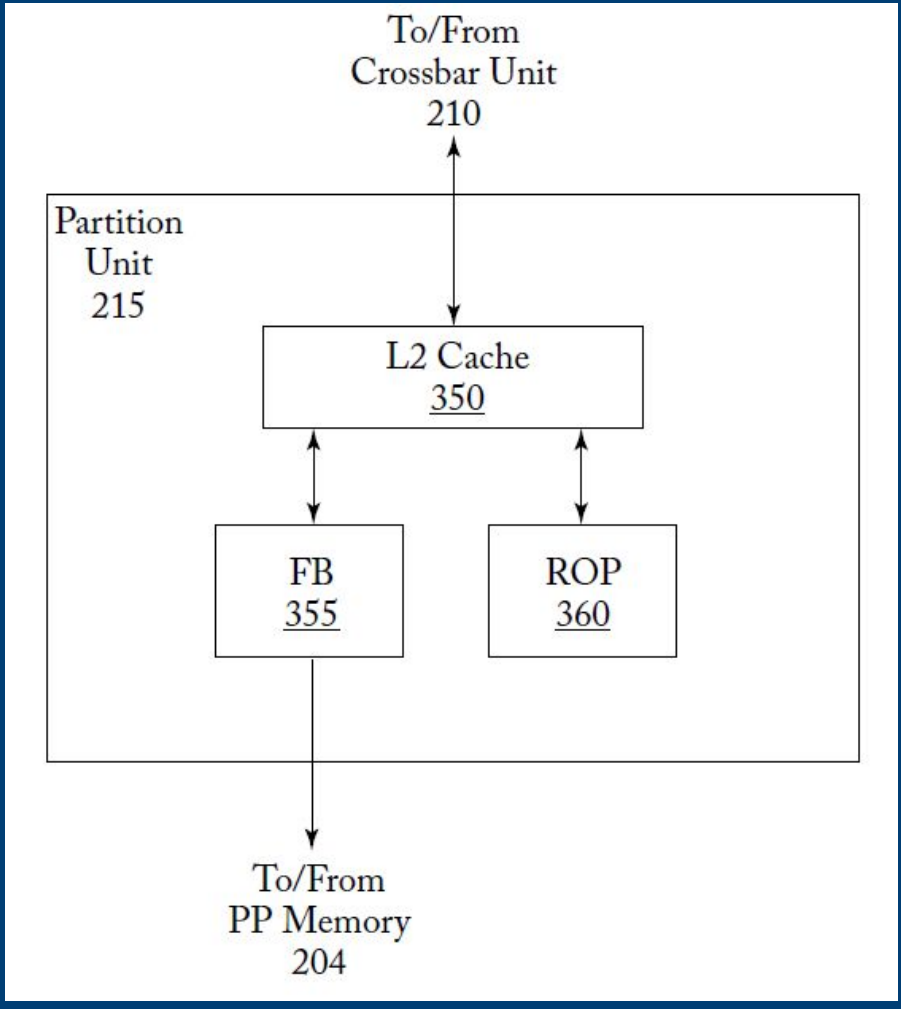
- Arbiter(2): handle bank conflicts ("replay")
- Shared memory load, read (hit & miss), write (through & back)

Texture Cache



- Read-only
- FIFO: hide the latency of miss requests that may need to be serviced from DRAM

Memory Partition Unit

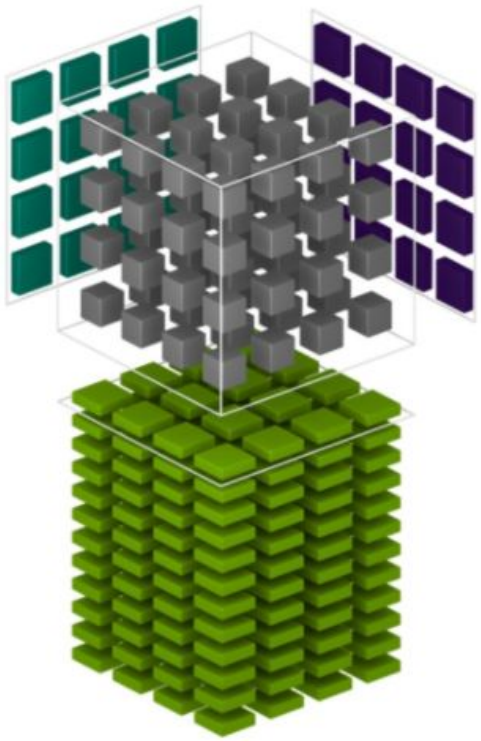


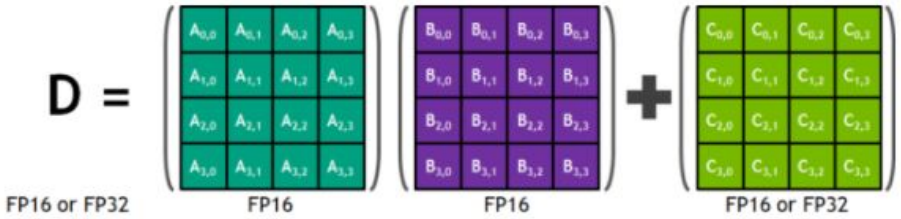
- L2 Cache: contains both graphics and compute data.
- Frame Buffer (FB)
- Raster Operations Pipeline (ROP):
  - Graphics operation (e.g. alpha blending).
  - Atomic operations (e.g. in CUDA programming model).

GPU Special Function Hardware - Tensor Core

Volta GV100 Streaming Multiprocessor

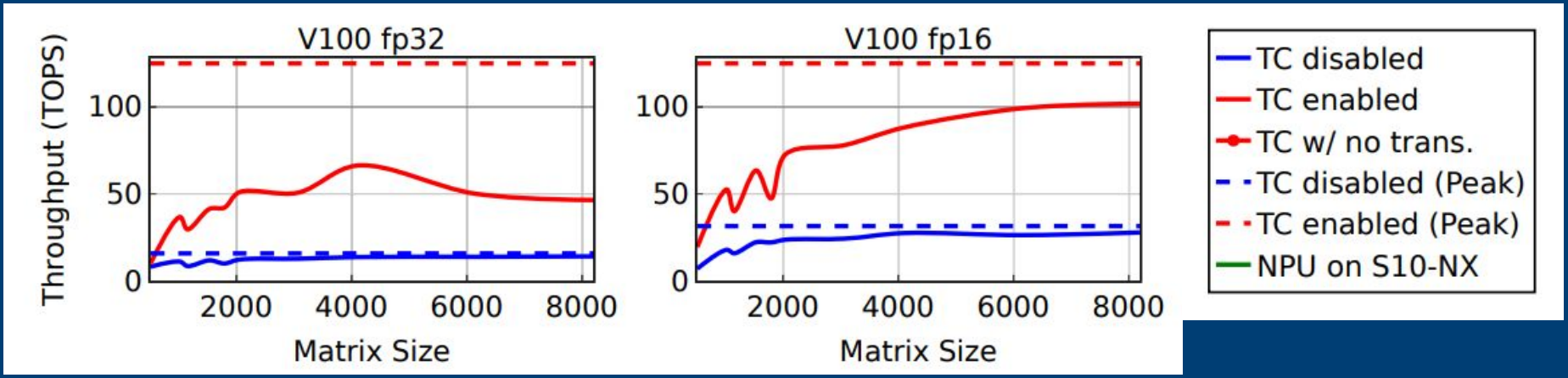
Volta Tensor Core Matrix Multiply and Accumulate





Using Tensor Cores to Accelerate Matrix Multiplications

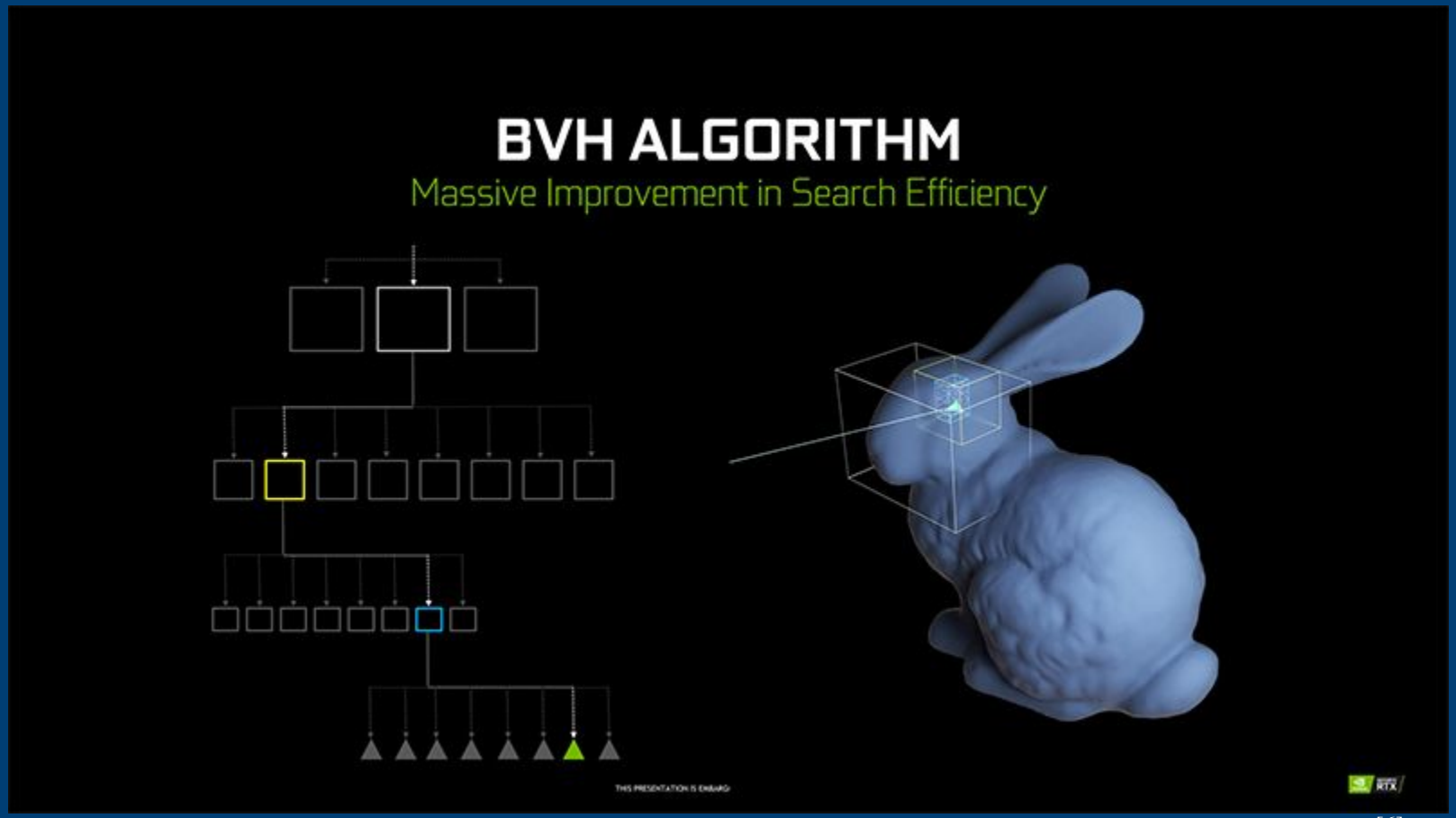
- Example - Neural Networks:
  - CNN, ResNet, …



Boutros, Andrew, et al. "Beyond peak performance: Comparing the real performance of AI-optimized FPGAs and GPUs." *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020.

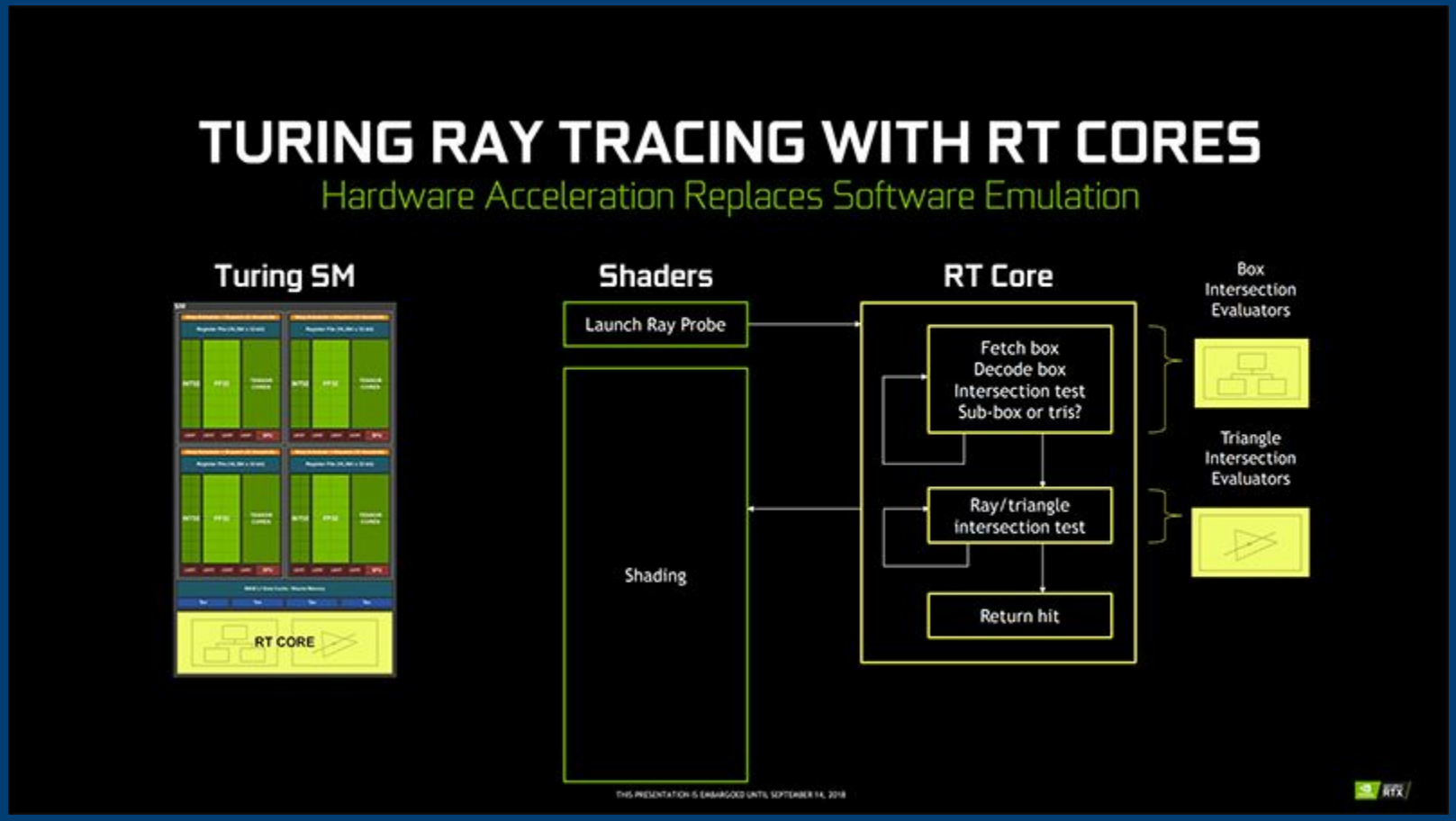
GPU Special Function Hardware - Ray Tracing Core

Ray Tracing Problem



[6]

Accelerate Ray Tracing Algorithms - RT Core



[6]

Using Ray Tracing Hardware For General Problem

- Example - Neighbor Search
  - Construct BVH tree using the given data
    - Already HW accelerated
  - Use the RT Core hardware to compute
    - Good if querying multiple times
  - By professor Yuhao Zhu, University of Rochester
    - RTNN: Accelerating Neighbor Search Using Hardware Ray Tracing[9]
- Example - Tet-Mesh Point Location
  - RTX Beyond Ray Tracing: Exploring the Use of Hardware Ray Tracing Cores for Tet-Mesh Point Location[10]

References

- [1]Aamodt, T. M., Fung, W. W. L., & Rogers, T. G. (2018). General-purpose graphics processor architectures. *Synthesis Lectures on Computer Architecture*, 13(2), 1-140.
- [2]https://pages.cs.wisc.edu/~sinclair/courses/cs758/fall2019/handouts/lecture/cs758-fall19-gpu_memSys.pdf
- [3]https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
- [4]https://developer.nvidia.com/blog/even-easier-introduction-cuda/
- [5]https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/
- [6]https://www.hardwarezone.com.sg/feature-what-you-need-know-about-ray-tracing-and-nvidias-turing-architecture/rt-cores-and-tensor-cores

References

- [7]W. W. L. Fung and T. M. Aamodt, "Thread block compaction for efficient SIMT control flow," 2011 IEEE 17th International Symposium on High Performance Computer Architecture, 2011, pp. 25-36, doi: 10.1109/HPCA.2011.5749714.
- [8]https://www.ece.lsu.edu/gp/refs/gf100-whitepaper.pdf
- [9]https://www.cs.rochester.edu/horizon/pubs/ppopp22.pdf
- [10]I. Wald, W. Usher, N. Morrical, L. Lediaev, and V. Pascucci, “RTX Beyond Ray Tracing,” p. 7.

Q&A