

Introduction to GPUs, CUDA Programming, Optimizations, and Research Directions

[NSM-CAWS 2021]

Vishwesh Jatala

Assistant Professor

Department of EECS

Indian Institute of Technology Bhilai

vishwesh@iitbhilai.ac.in



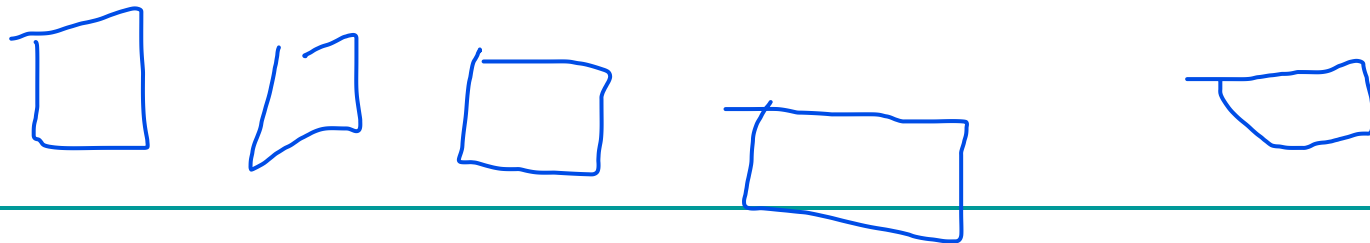
**National
Supercomputing
Mission**

Outline

- Motivation
- GPU Architecture
- CUDA programming
- Instruction execution
- GPU specific optimizations
- Research directions

To compile offline:

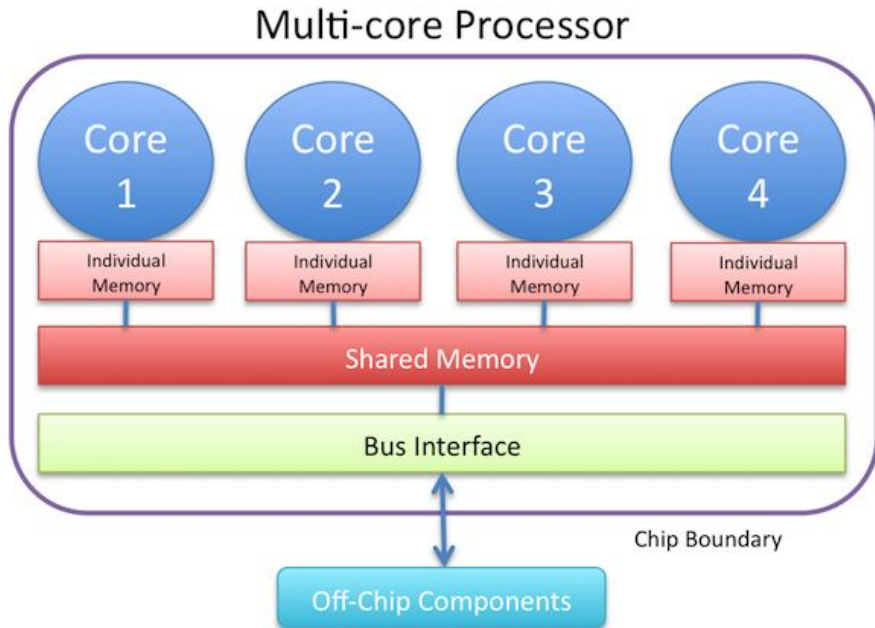
<https://stackoverflow.com/questions/6622454/cuda-incompatible-with-my-gcc-version>



Motivation

- For many decades, the single core processors were popular
 - ❑ Instruction-level parallelism
 - ❑ Core clock frequency
 - ❑ Moore's law
- Mid-to late-1990s - power wall
 - ❑ Power constraints
 - ❑ Heat dissipation
- Multicore processors, accelerators, such as GPUs.

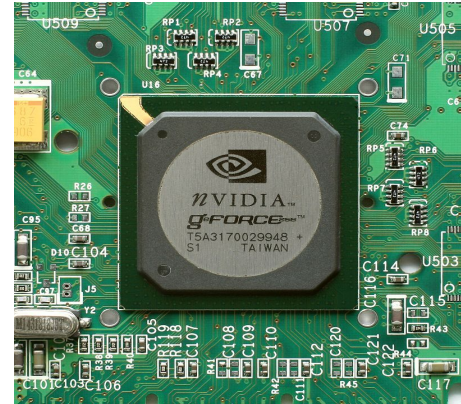
Why GPUs?



- Multicore processors
 - ❑ Task level parallelism
 - ❑ Graphics rendering is computationally expensive
 - ❑ Not efficient for graphics applications

Graphics Processing Units

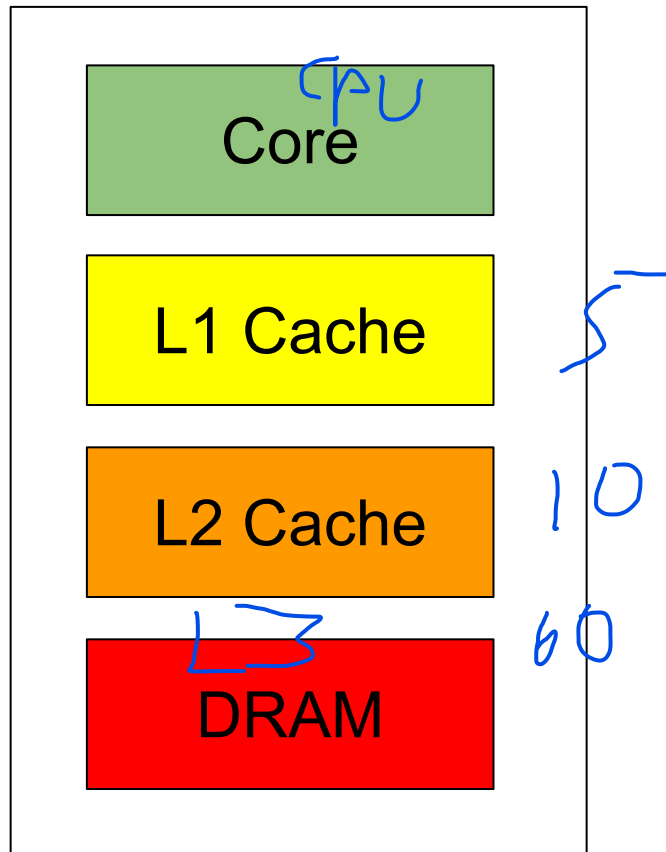
- The early GPU designs
 - ❑ Specialized for graphics processing only
 - ❑ Exhibit SIMD execution
 - ❑ Less programmable
- In 2007, fully programmable GPUs
 - ❑ CUDA released



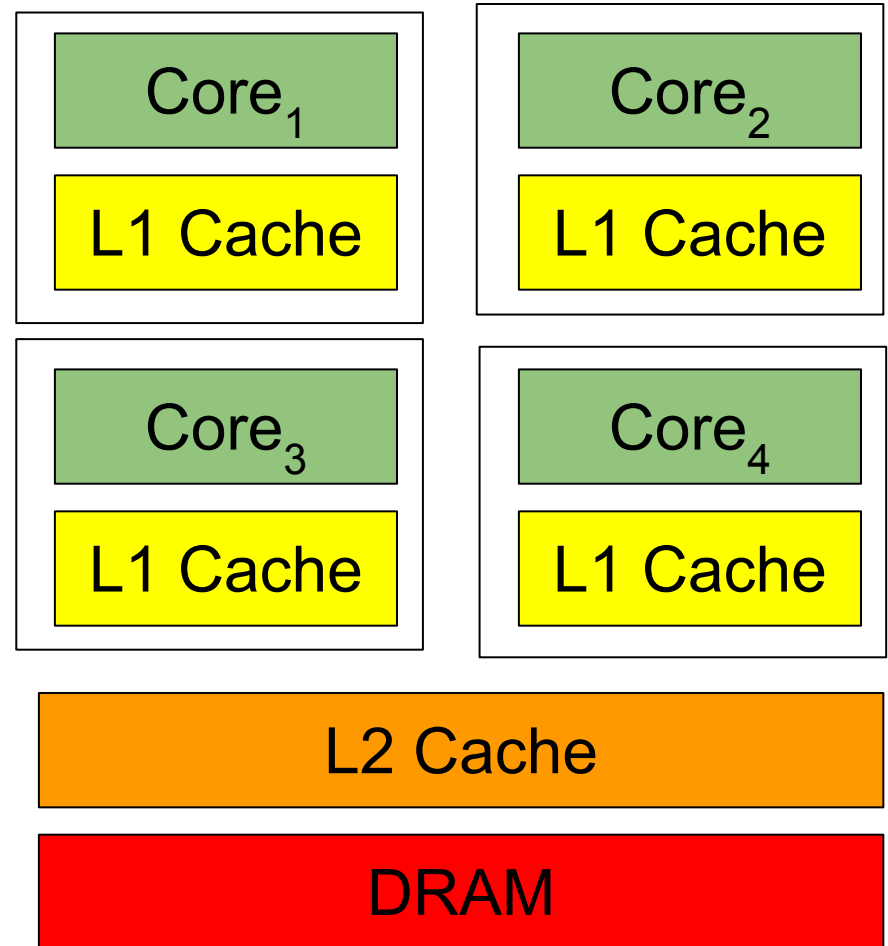
NVIDIA GeForce 256



Single-core CPU vs Multi-core vs GPU



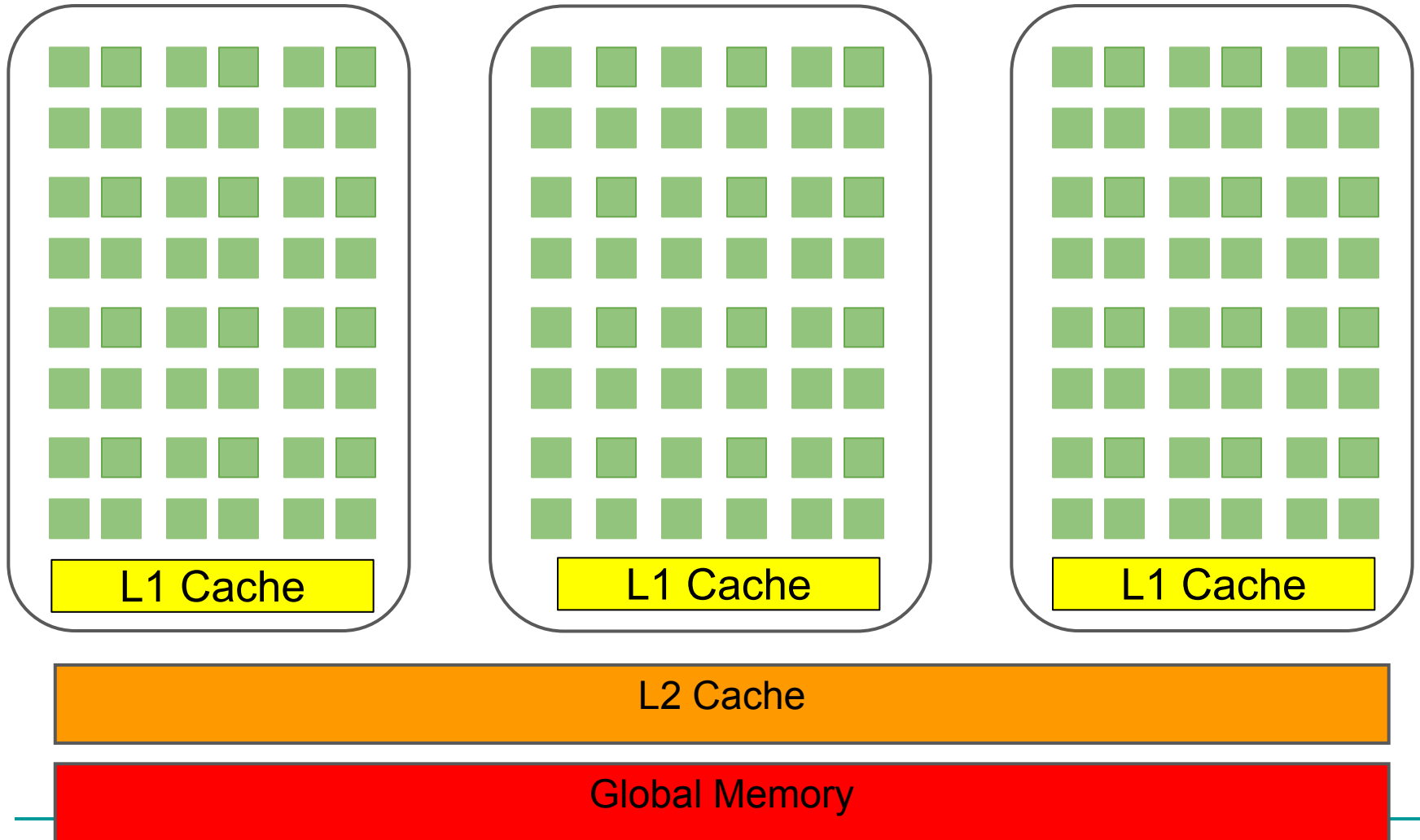
Single-core CPU



Multi-core CPU

Single-core CPU vs Multi-core vs GPU

Streaming Multiprocessor Streaming Multiprocessor Streaming Multiprocessor



~PR AM

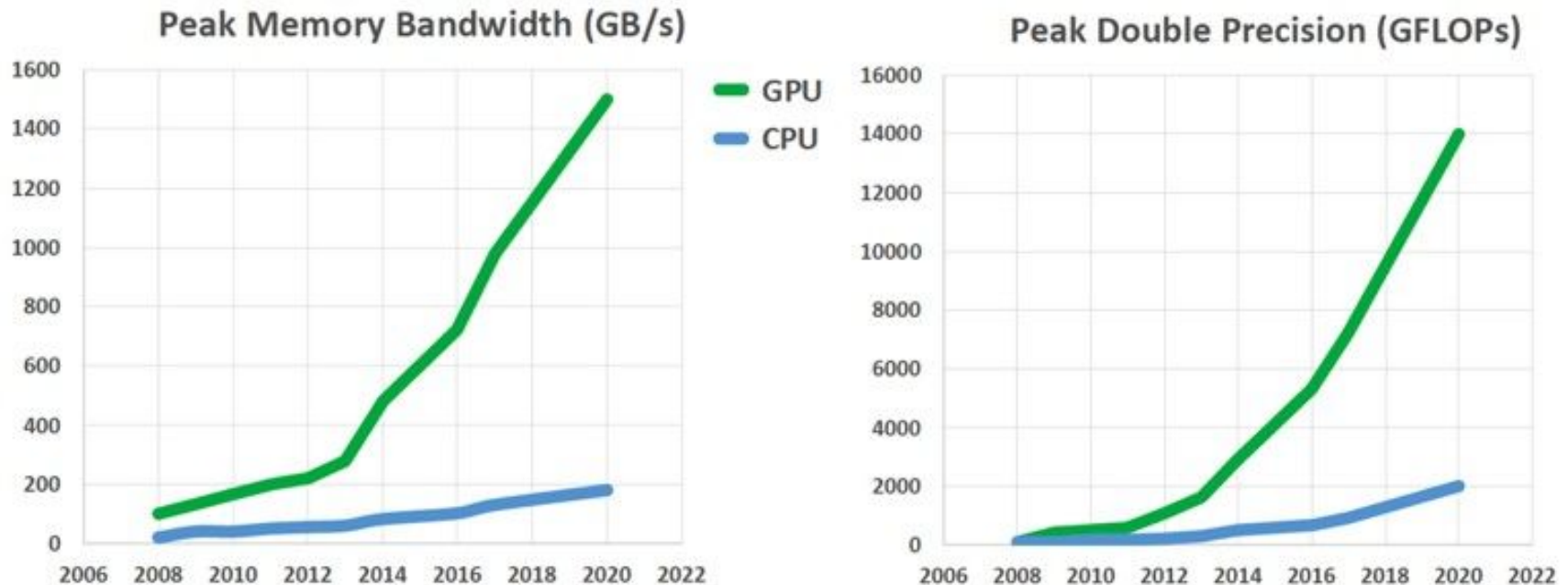
NVIDIA Volta GV100



Specifications

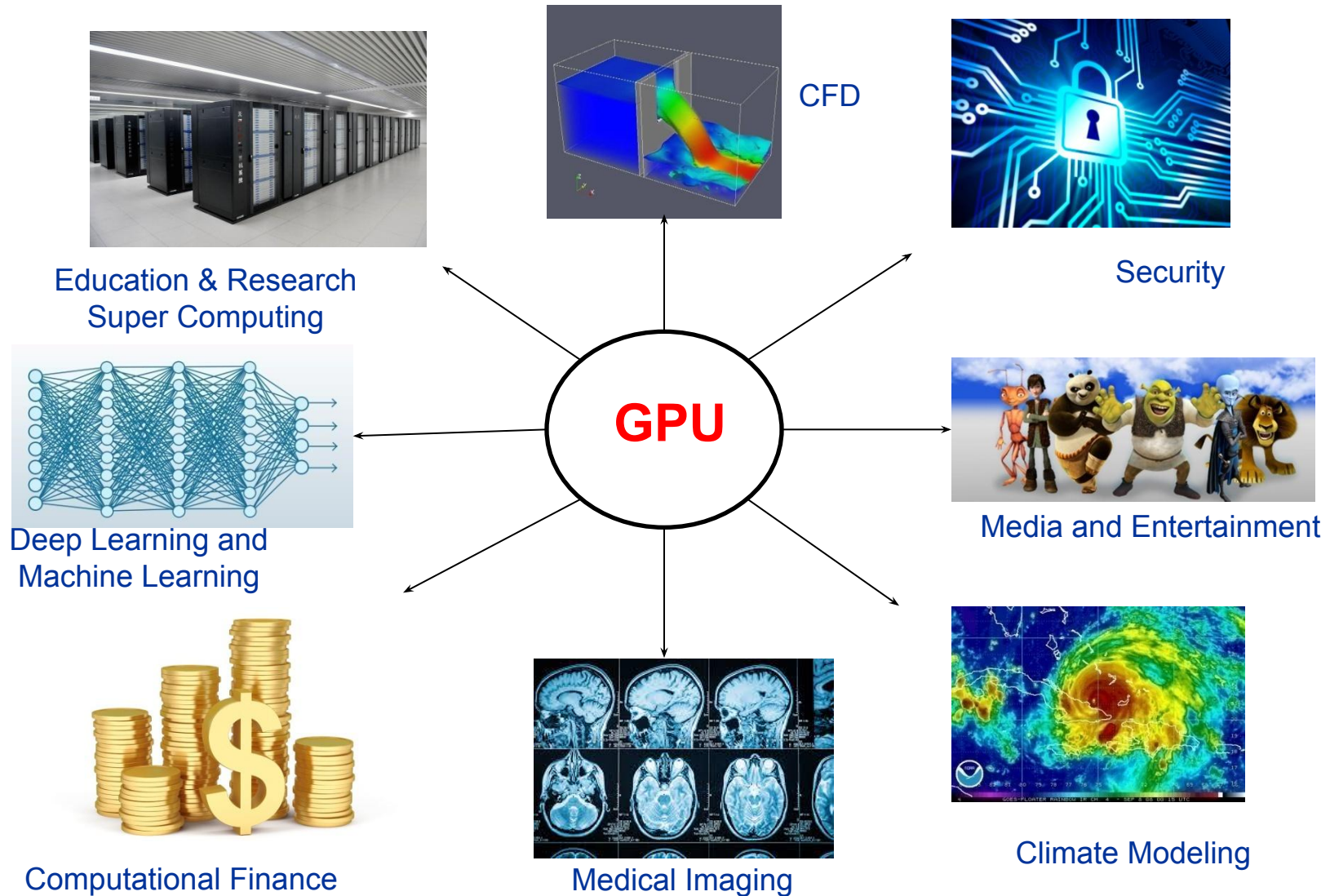
Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS ¹	5	6.8	10.6	15.7
Peak FP64 TFLOPS ¹	1.7	.21	5.3	7.8
Peak Tensor TFLOPS ¹	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm ²	601 mm ²	610 mm ²	815 mm ²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

CPU vs GPU



Chip to chip comparison of peak memory bandwidth in GB/s and peak double precision gigaflops for GPUs and CPUs since 2008.

GPU Applications

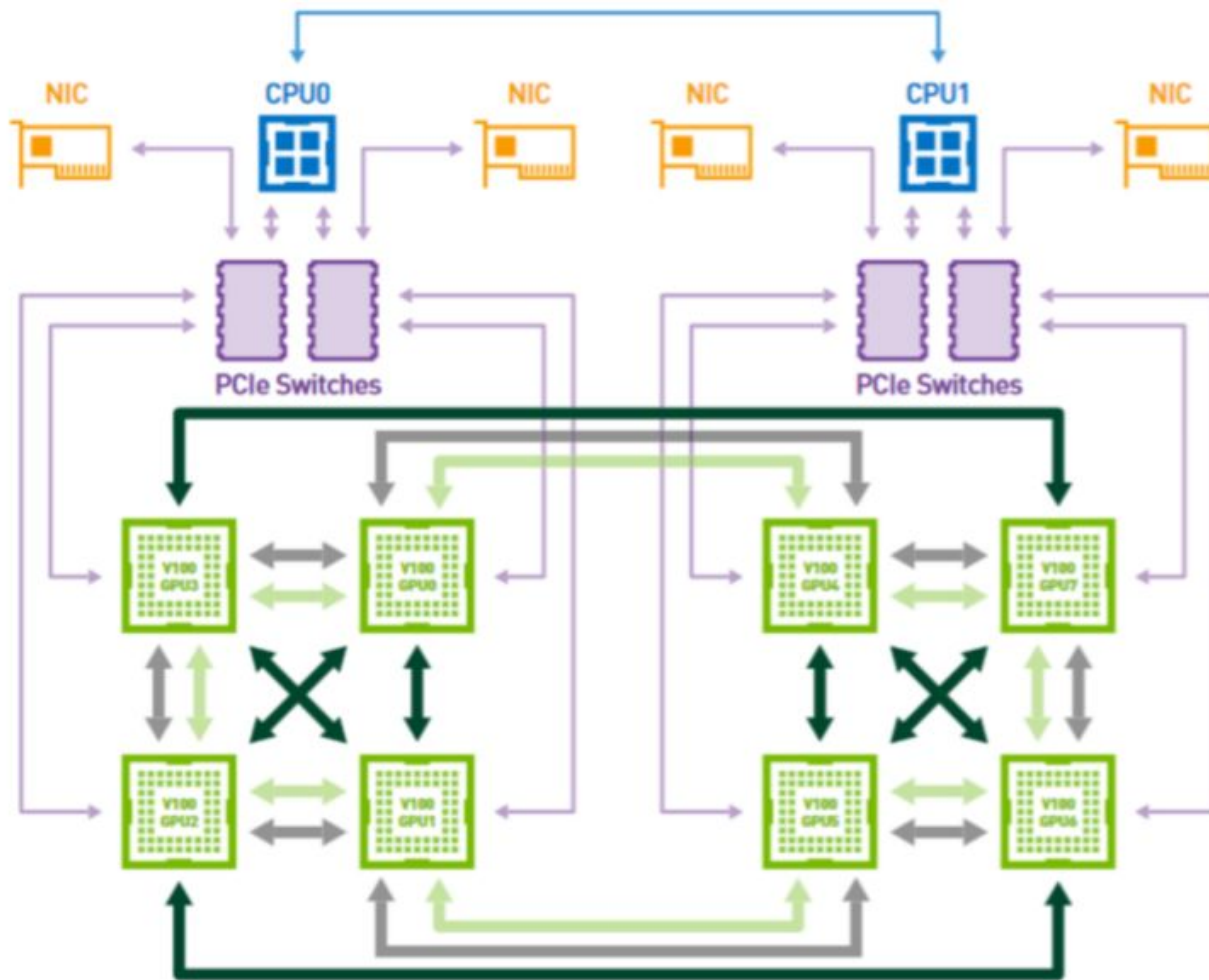


Specifications

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS ¹	5	6.8	10.6	15.7
Peak FP64 TFLOPS ¹	1.7	.21	5.3	7.8
Peak Tensor TFLOPS ¹	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm ²	601 mm ²	610 mm ²	815 mm ²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

<https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>

Multi-GPU Systems

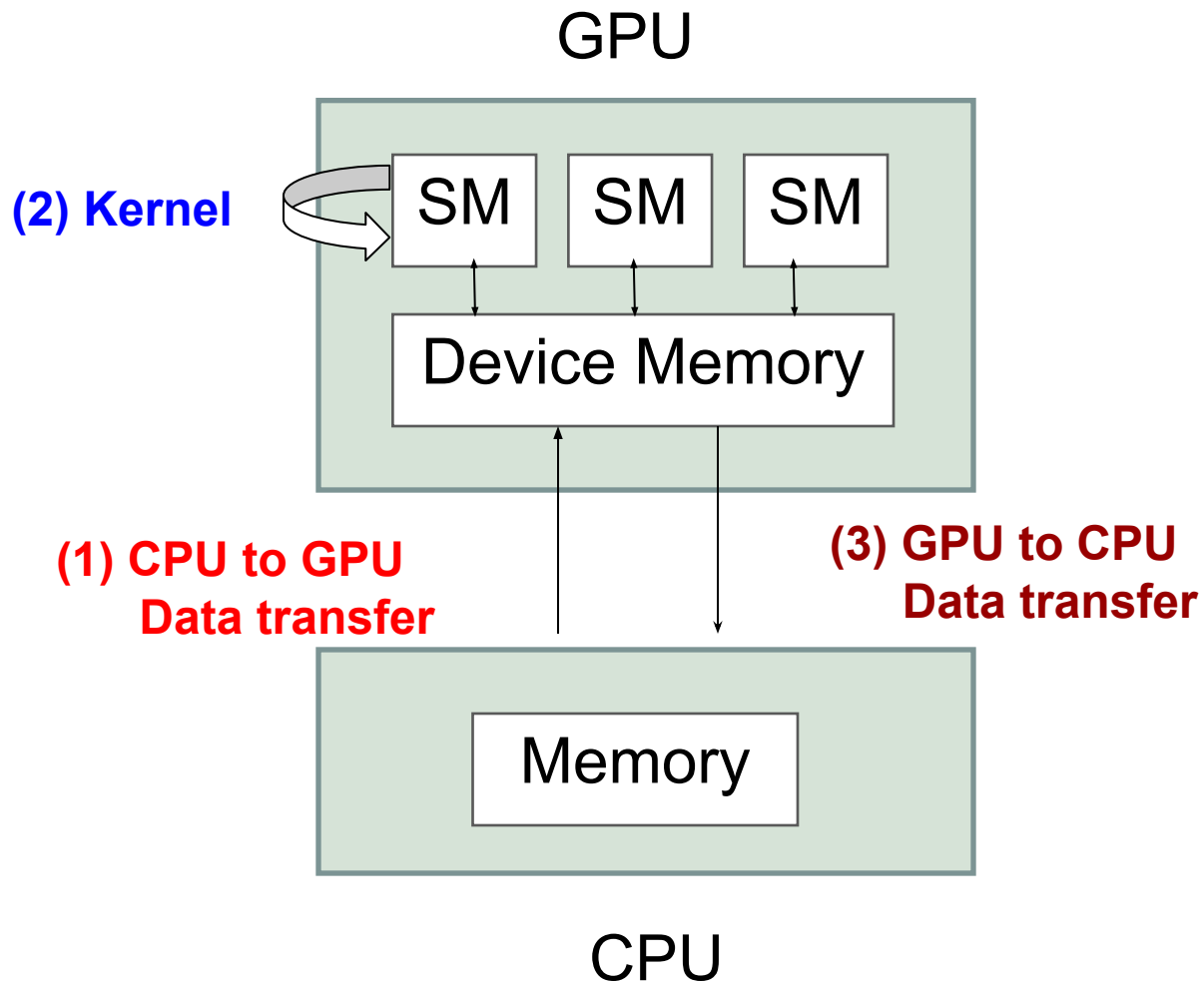


Programming for GPUs

Programming Models

- CUDA (Compute Unified Device Architecture)
 - Supports NVIDIA GPUs
 - Extension of C programming language
 - Popular in academia
- OpenCL (Open Computing Language)
 - Open source
 - Supports various GPU devices

Introduction to CUDA Programming



Simple Example

__global__ - executed in the GPU and called in the CPU
__device__ - executed and called in the GPU
__host__ - executed and called in the CPU

```
__global__ void Square_Kernel(float a){  
    /* Compute index i based on thread id */  
    a[i] = a[i] * a[i];  
}
```

← **Compute Kernel**

```
int main() {  
    h_a = malloc(..) // host array  
    cudaMalloc(d_a,...) // device
```

```
    /* Initialize h_a */
```

```
    cudaMemcpy(d_a, h_a, cudaMemcpyHostToDevice)
```

← **CPU to GPU
Data transfer**

```
    Square_Kernel<<<ThreadConfig>>> (d_a);
```

← **Invoke Kernel**

```
    cudaMemcpy(h_a, d_a, cudaMemcpyDeviceToHost)
```

← **GPU to CPU
Data transfer**

```
    process(h_a);  
    cudaFree(d_a);  
    free(h_a);
```

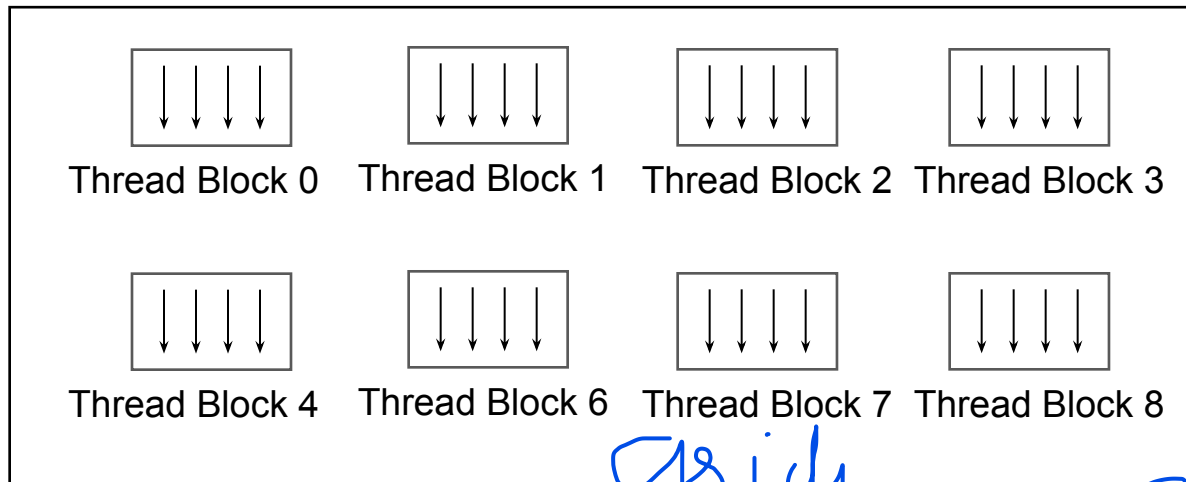
```
}
```

Thread Configuration

`Square_Kernel<<<ThreadConfig>>> (d_a);`



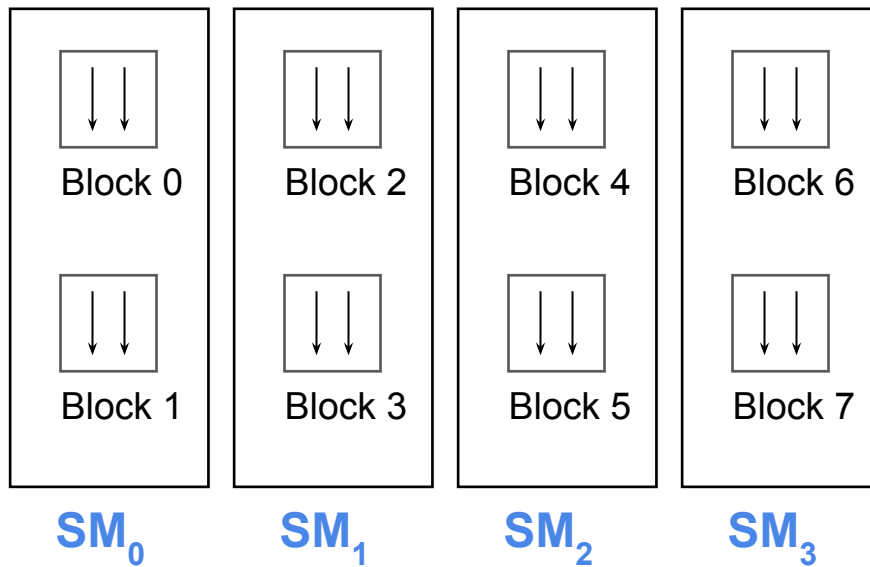
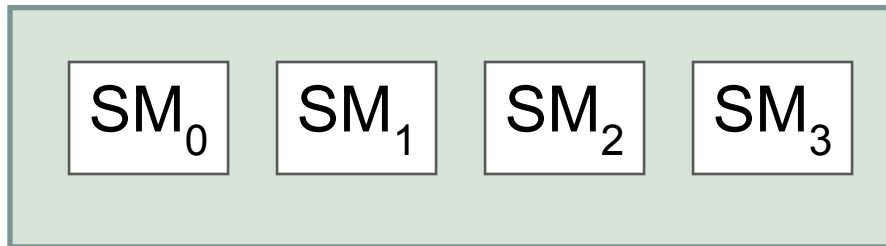
`Square_Kernel<<<ThreadBlocks, Threads>>> (d_a);`



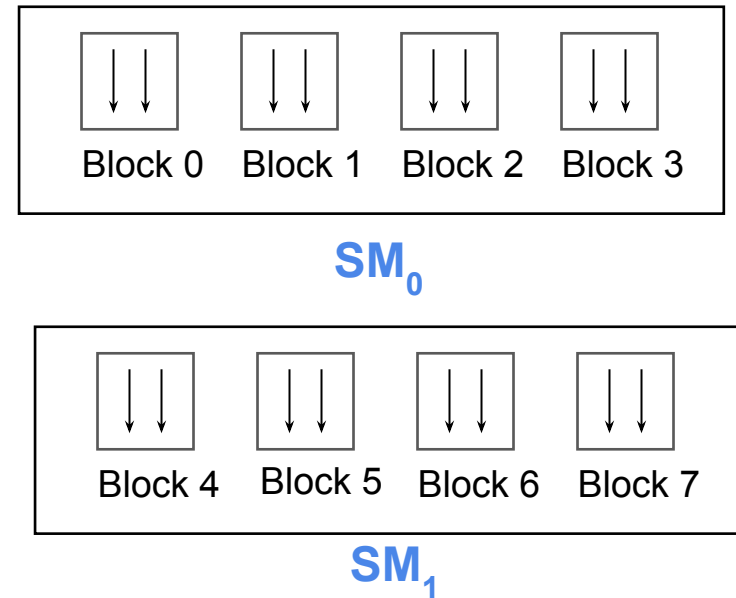
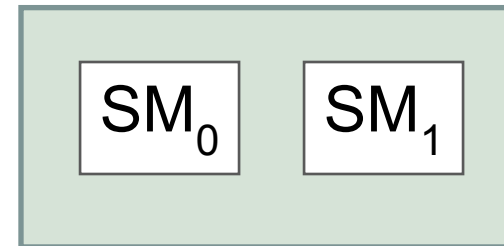
GPU → SMs → grids → Blocks → 32 threads
warps → threads

Scalability

GPU-0



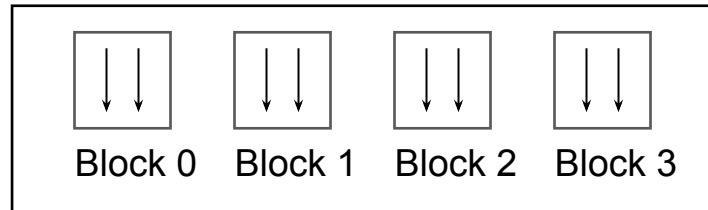
GPU-1



Execution in GPU

Effect of heirarchy?

SM_0



Block Size: 128 threads

Number of Blocks: 4

Total threads: 512

Threads grouped into warps
Size of each warp = 32 threads



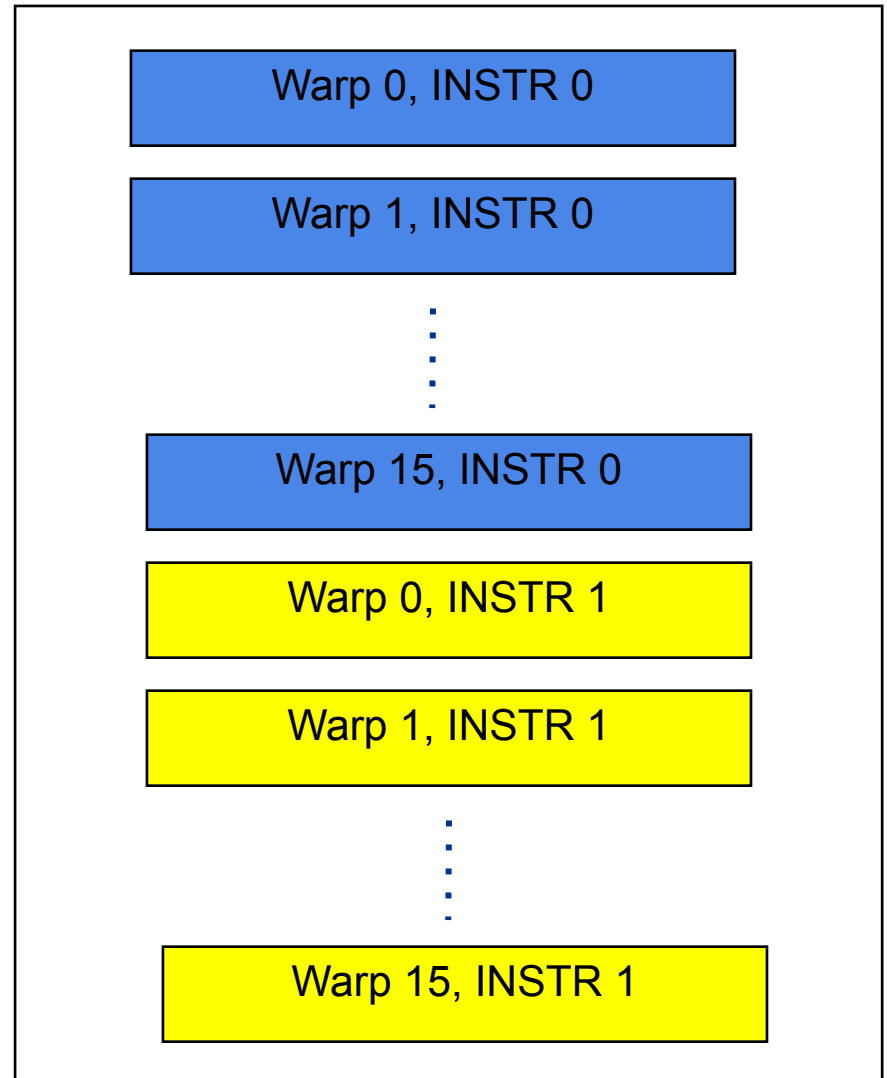
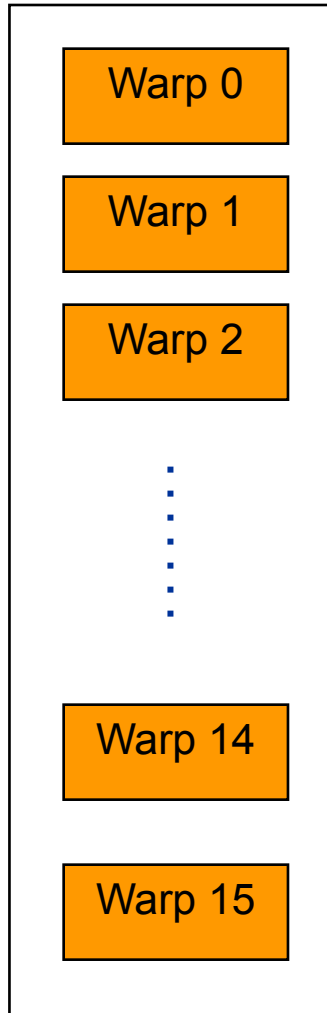
Block 0

Block 1

Block 2

Block 3

Instruction Execution



Warp Scheduler

Simple Example (Revisit)

```
__global__ void Square_Kernel(float a){  
    /* Compute index i based on thread id */  
    a[i] = a[i] * a[i];  
}
```

```
int main() {  
    h_a = malloc(..) // host array  
    cudaMalloc(d_a,...) // device
```

```
    /* Initialize h_a */
```

```
    cudaMemcpy(d_a, h_a, cudaMemcpyHostToDevice)
```

```
    Square_Kernel<<<ThreadConfig>>> (d_a);
```

```
    cudaMemcpy(h_a, d_a, cudaMemcpyDeviceToHost)
```

```
    process(h_a);  
    cudaFree(d_a);  
    free(h_a);
```

```
}
```

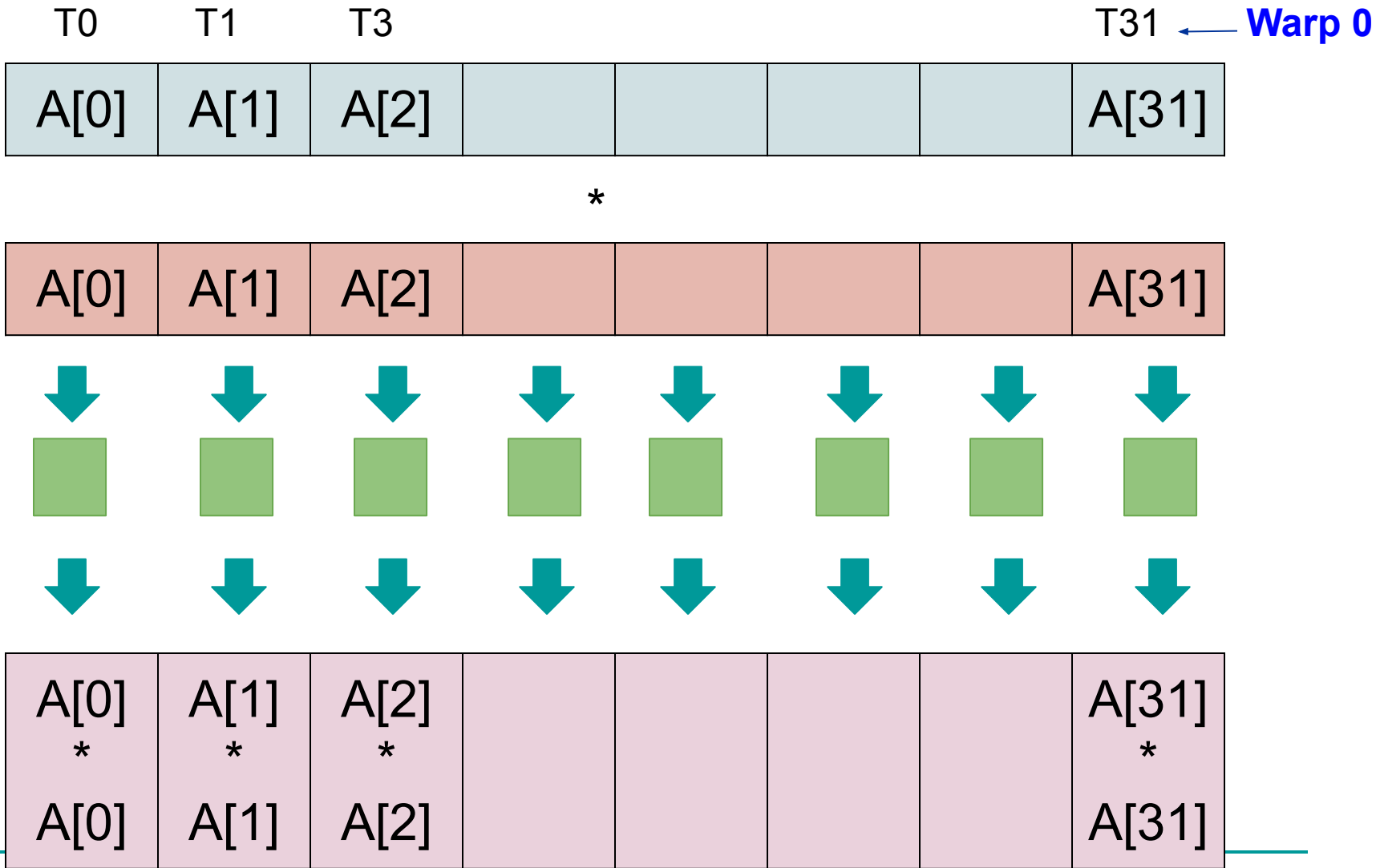
← **Compute Kernel**

← **CPU to GPU
Data transfer**

← **Invoke Kernel**

← **GPU to CPU
Data transfer**

Processing in GPU (SIMD)



Threadblock configuration

`Square_Kernel<<<ThreadBlocks, Threads> (d_a);`

- Thread block configuration
 - ❑ User choice
 - ❑ Depends on problem size
- Problem size = 32768 ($1024 * 32$)
 - ❑ Threadblocks = 32, No of threads/thread block = 1024
 - ❑ Threadblocks = 128, No of threads/thread block = 256

CUDA thread block occupancy:

<https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html>

Compute Capabilities

GPU	Kepler GK180	Maxwell GM200	Pascal GP100	Volta GV100
Compute Capability	3.5	5.2	6.0	7.0
Threads / Warp	32	32	32	32
Max Warps / SM	64	64	64	64
Max Threads / SM	2048	2048	2048	2048
Max Thread Blocks / SM	16	32	32	32
Max 32-bit Registers / SM	65536	65536	65536	65536
Max Registers / Block	65536	32768	65536	65536
Max Registers / Thread	255	255	255	255 ¹
Max Thread Block Size	1024	1024	1024	1024
FP32 Cores / SM	192	128	64	64
Ratio of SM Registers to FP32 Cores	341	512	1024	1024
Shared Memory Size / SM	16 KB/32 KB/ 48 KB	96 KB	64 KB	Configurable up to 96 KB

Source: <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>

CUDA Optimizations

CUDA Programming

CPU

Bridge

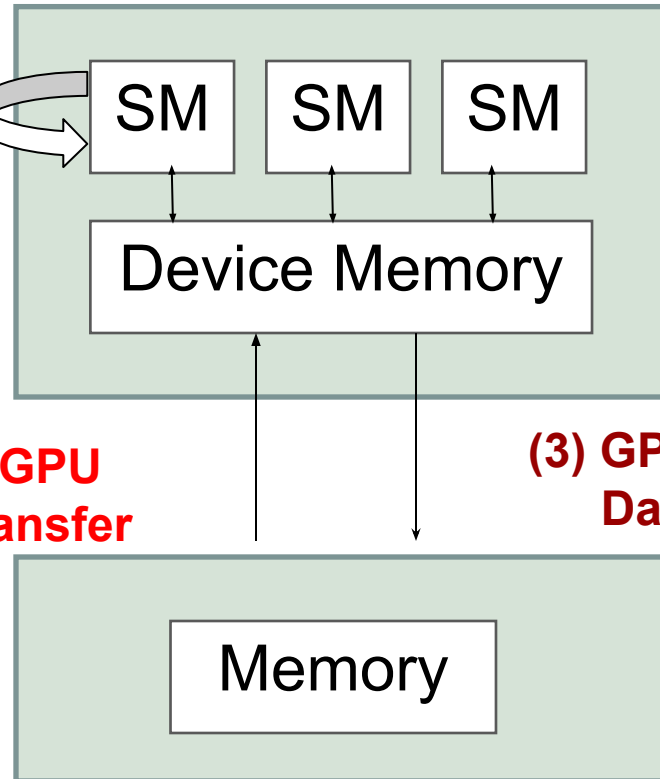
GPU

CPU

(2) Kernel

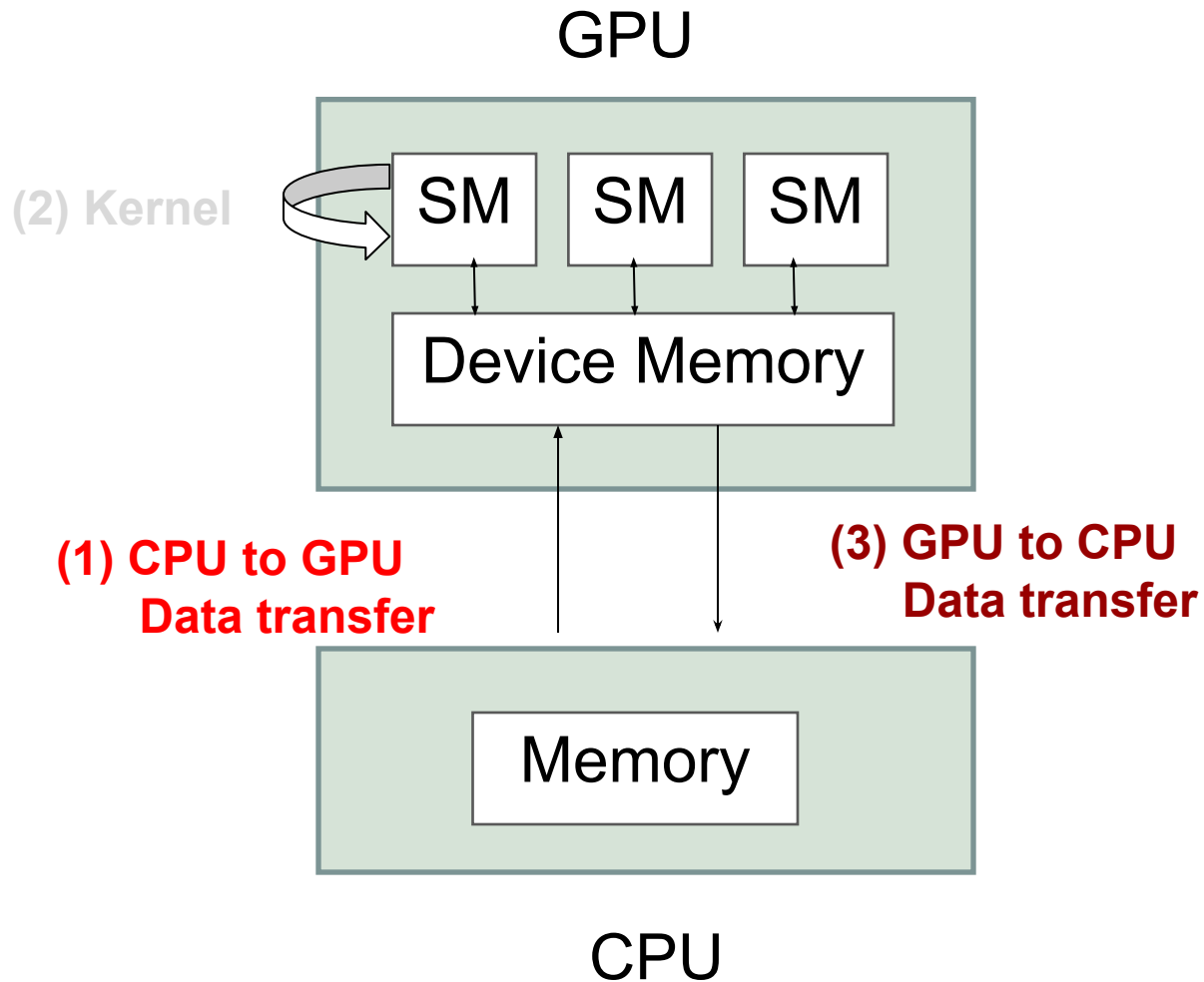
(1) CPU to GPU
Data transfer

(3) GPU to CPU
Data transfer



CPU

CUDA Programming



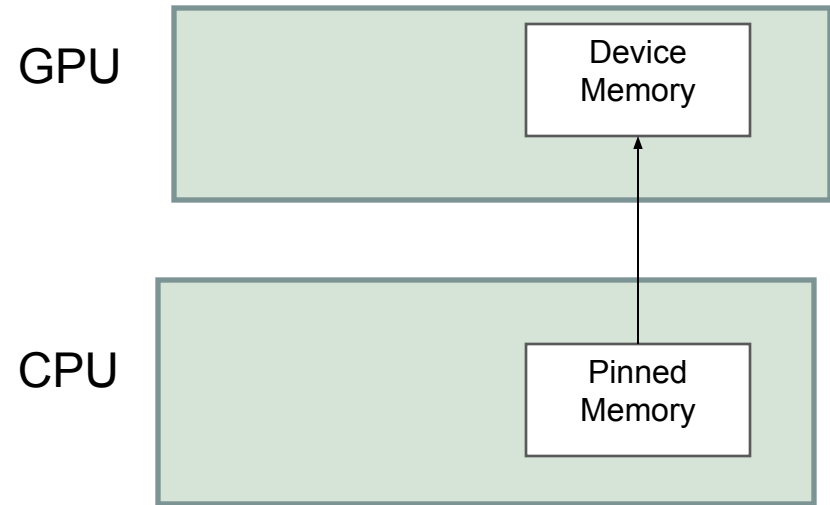
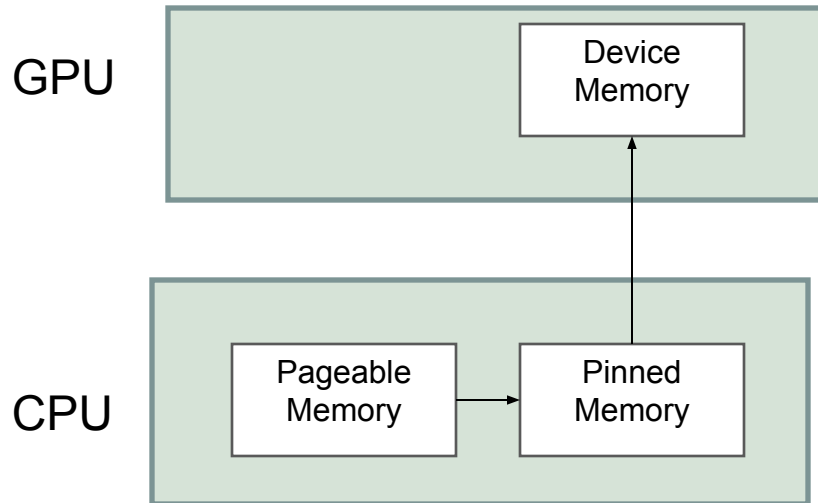
Optimizations

- CPU-GPU data transfers consume time
 - ❑ Pinned memory
 - ❑ Overlapping data transfer using streams
 - ❑ Unified memory
- Other optimizations (kernel)
 - ❑ Global memory access coalescing
 - ❑ Shared memory bank conflicts, etc.

Optimizing Memory Transfer – Pinned Memory

```
int main() {  
    h_a = malloc(..)//host pageable  
    d_a = cudaMalloc(..) //device  
    /* Initialize h_a */  
    cudaMemcpy(d_a, h_a, HostToDevice)  
    /* Process d_a */  
}
```

```
int main() {  
    h_a=cudaMallocHost(..)//host pinned  
    d_a = cudaMalloc(..) //device  
    /* Initialize h_a */  
    cudaMemcpy(d_a, h_a, HostToDevice)  
    /* Process d_a */  
}
```



Overlapping Executions

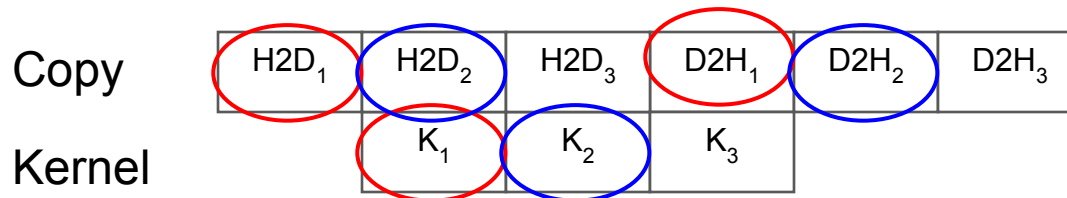
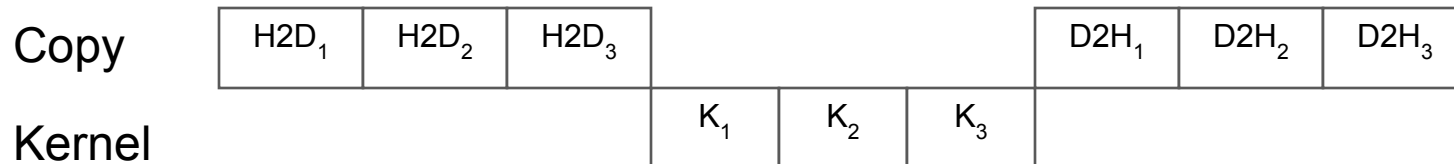
```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
increment<<<1,N>>>(d_a)  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

Blocking
Asynchronous
Blocking

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
increment<<<1,N>>>(d_a) GPU function called in CPU (__global__)  
Compute() CPU function called in CPU (__host__)  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```

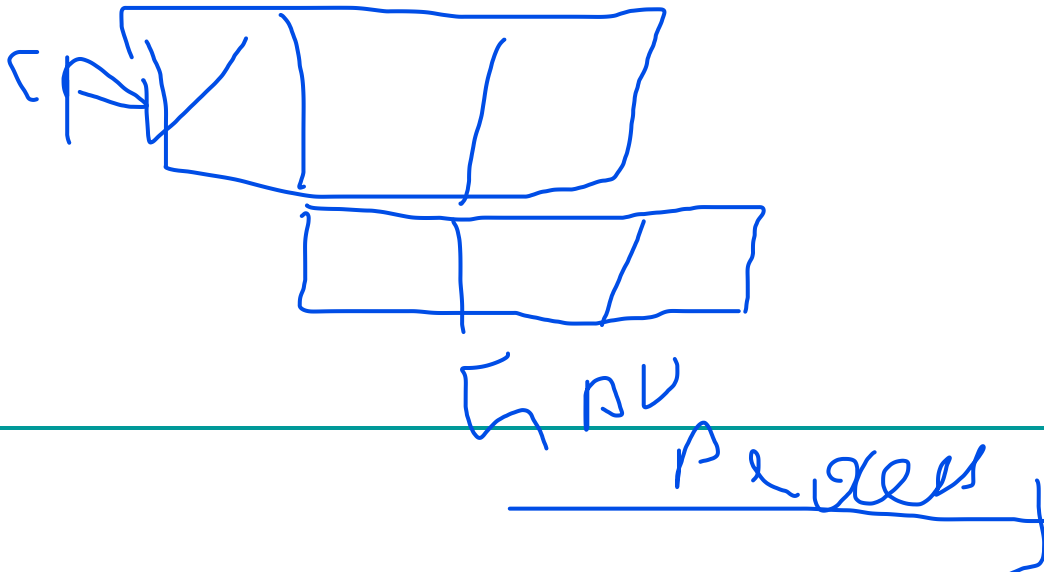
Overlapping Executions Using Streams

```
cudaMemcpy(d_a, a, numBytes, cudaMemcpyHostToDevice);  
increment<<<1,N>>>(d_a)  
cudaMemcpy(a, d_a, numBytes, cudaMemcpyDeviceToHost);
```



Overlapping Executions Using Streams

- Stream is sequence of operations
 - executed on GPU in the same order issued by host
- Operations across different streams
 - can be interleaved
 - can be executed concurrently



Overlapping Data Transfer and Execution in Streams

```

for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync (/*Arguments */, cudaMemcpyHostToDevice, stream[i]);
}
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
}
for (int i = 0; i < nStreams; ++i) {
    int offset = i * streamSize;
    cudaMemcpyAsync ( /* Arguments */, cudaMemcpyDeviceToHost, Stream[i]);
}

```

Stream ₀	H2D ₁	K ₁	D2H ₁
Stream ₁	H2D ₂	K ₂	D2H ₂
Stream ₂	H2D ₃	K ₃	D2H ₃

Copy	H2D ₁	H2D ₂	H2D ₃	D2H ₁	D2H ₂	D2H ₃
Kernel	K ₁	K ₂	K ₃			

Copy	H2D ₁	H2D ₂	H2D ₃	D2H ₁	D2H ₂	D2H ₃
Kernel		K ₁	K ₂	K ₃		

Memory Management - Unified Memory

- Data transfer is automatic

```
int main() {  
    h_a = malloc(..) // host array  
    cudaMalloc(d_a,...) // device  
  
    /* Initialize h_a */  
  
    cudaMemcpy(d_a, h_a, HostToDevice)  
  
    Kernel<<<1, N> (d_a);  
  
    cudaMemcpy(h_a, d_a, DevicetoHost)  
    process(h_a);  
  
    cudaFree(d_a);  
    free(h_a);  
}
```

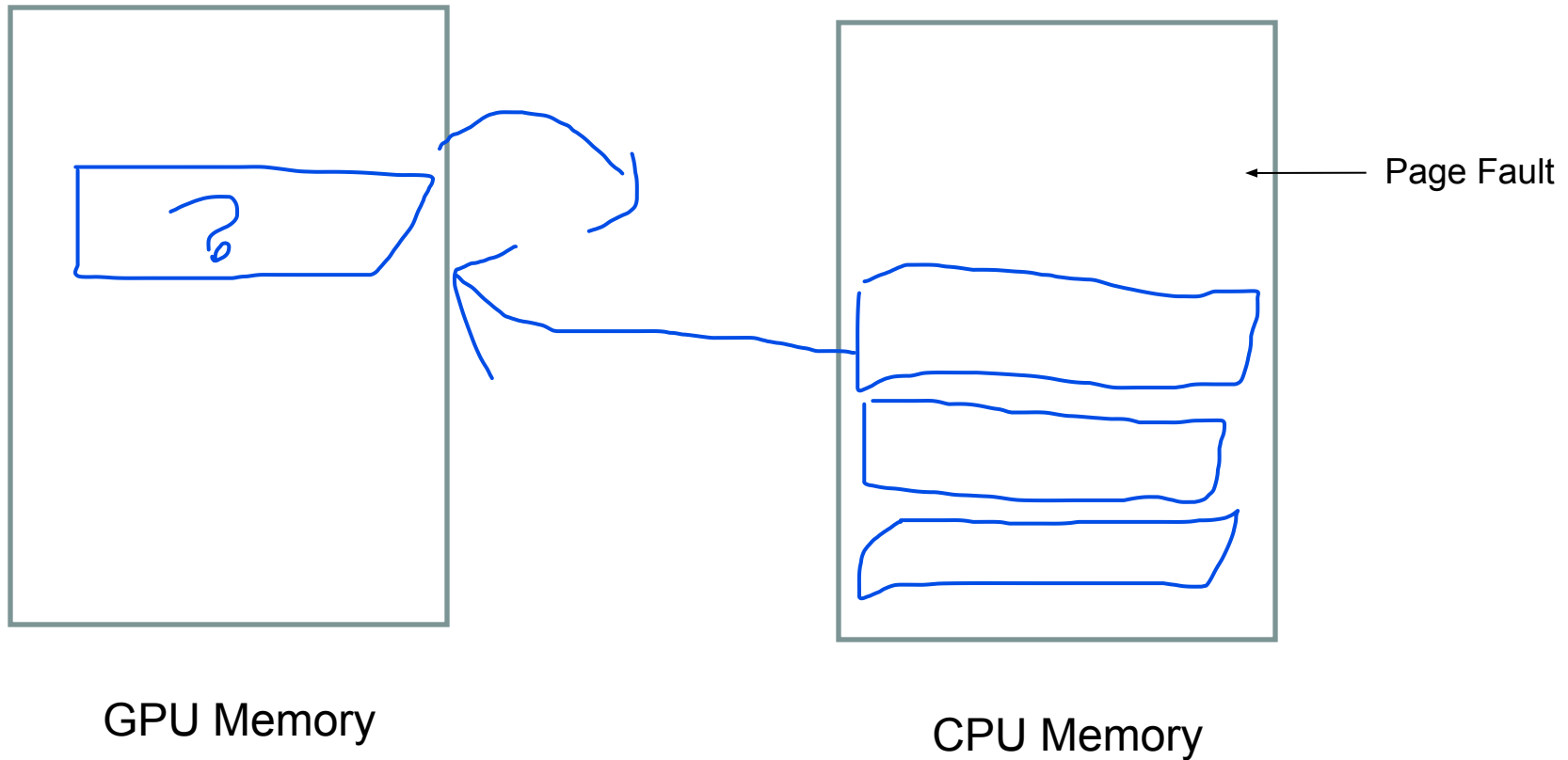
Without Unified Memory

```
int main() {  
  
    cudaMallocManaged(a,...)  
  
    /* Initialize a */  
  
    Kernel<<<1, N> (a);  
    cudaDeviceSynchronize();  
  
    process(a);  
  
    cudaFree(a);  
}
```

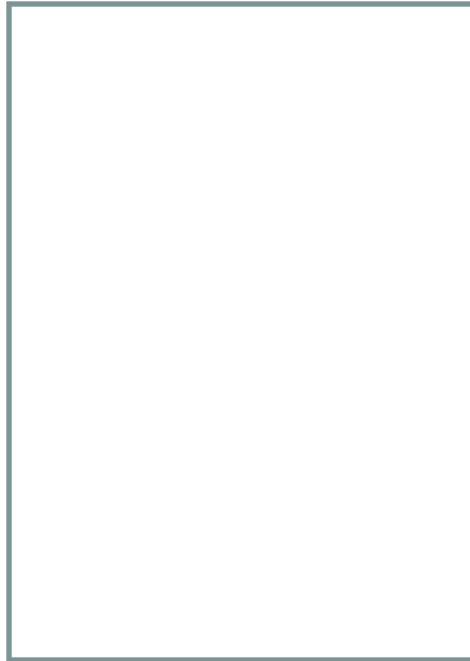
With Unified Memory

Unified Memory in Pascal

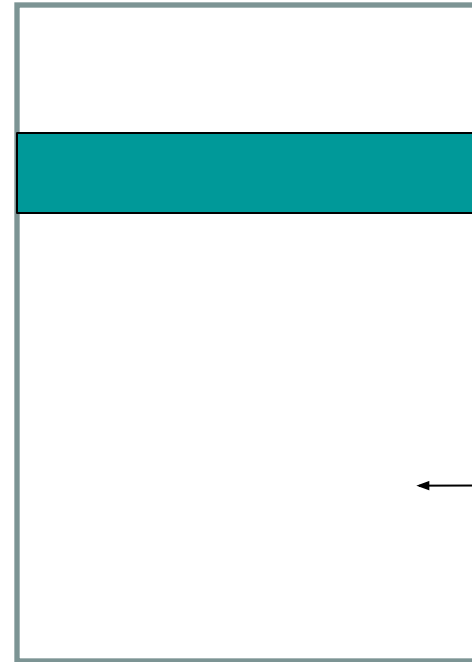
Initially may not have any pages!



Unified Memory in Pascal



GPU Memory



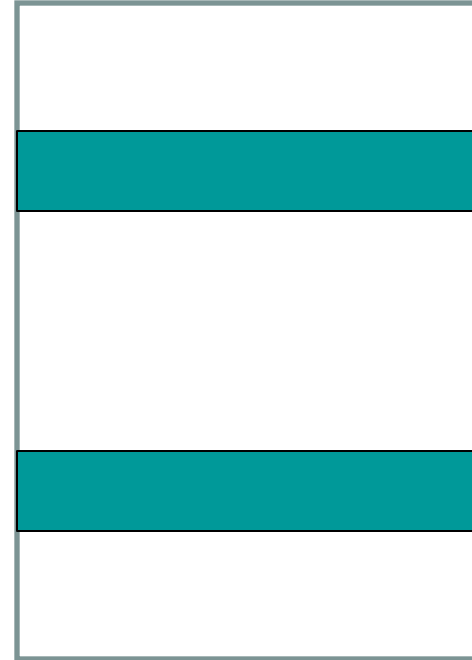
← Page Fault

CPU Memory

Unified Memory in Pascal

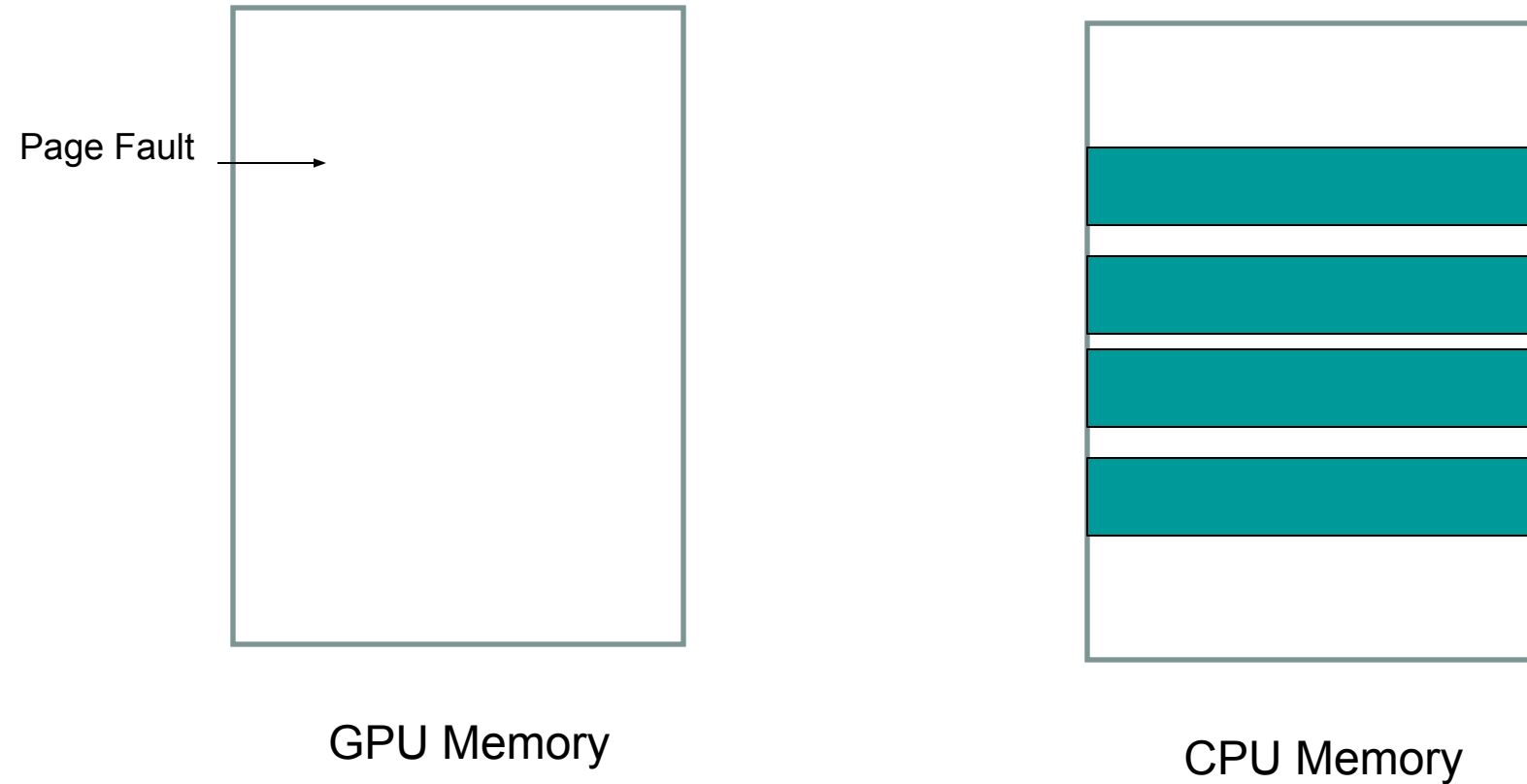


GPU Memory



CPU Memory

Unified Memory in Pascal



Unified Memory in Pascal

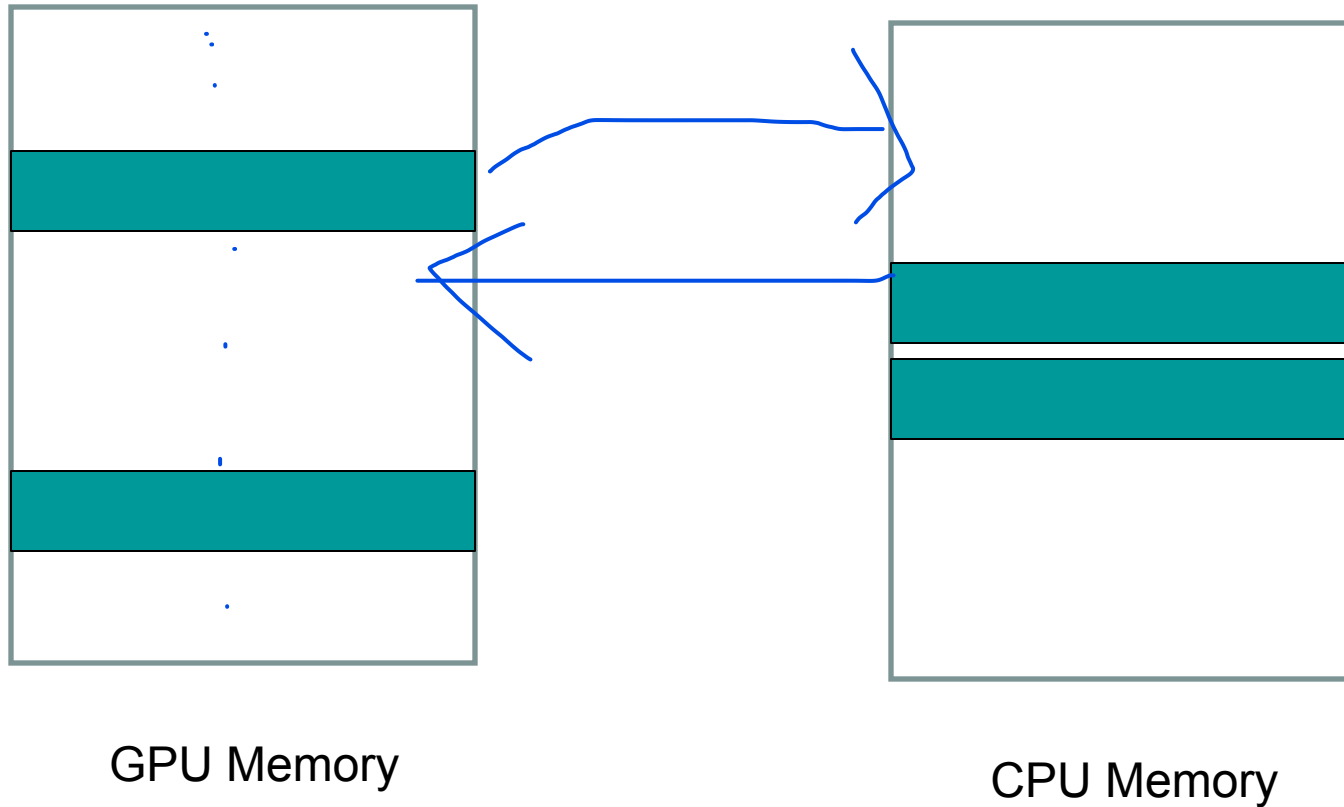


GPU Memory



CPU Memory

Unified Memory in Pascal



Unified Memory in Pascal

- Supports processing large inputs
 - ❑ Pages may not be allocated on GPU upon `cudaMallocManaged()`
 - ❑ Page faults can occur both in CPU and GPU
 - ❑ Oversubscription of pages is possible by evicting the pages in GPU

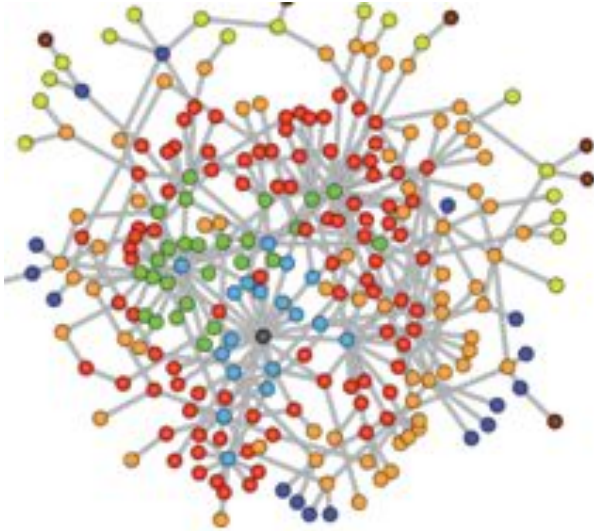
Will there be data loss?

Quick Summary

- Optimizations are crucial for performance
 - Programmers
 - Architects
- Optimizations
 - Data transfers
 - Warp divergence, access patterns, resource utilization etc.

Research Directions

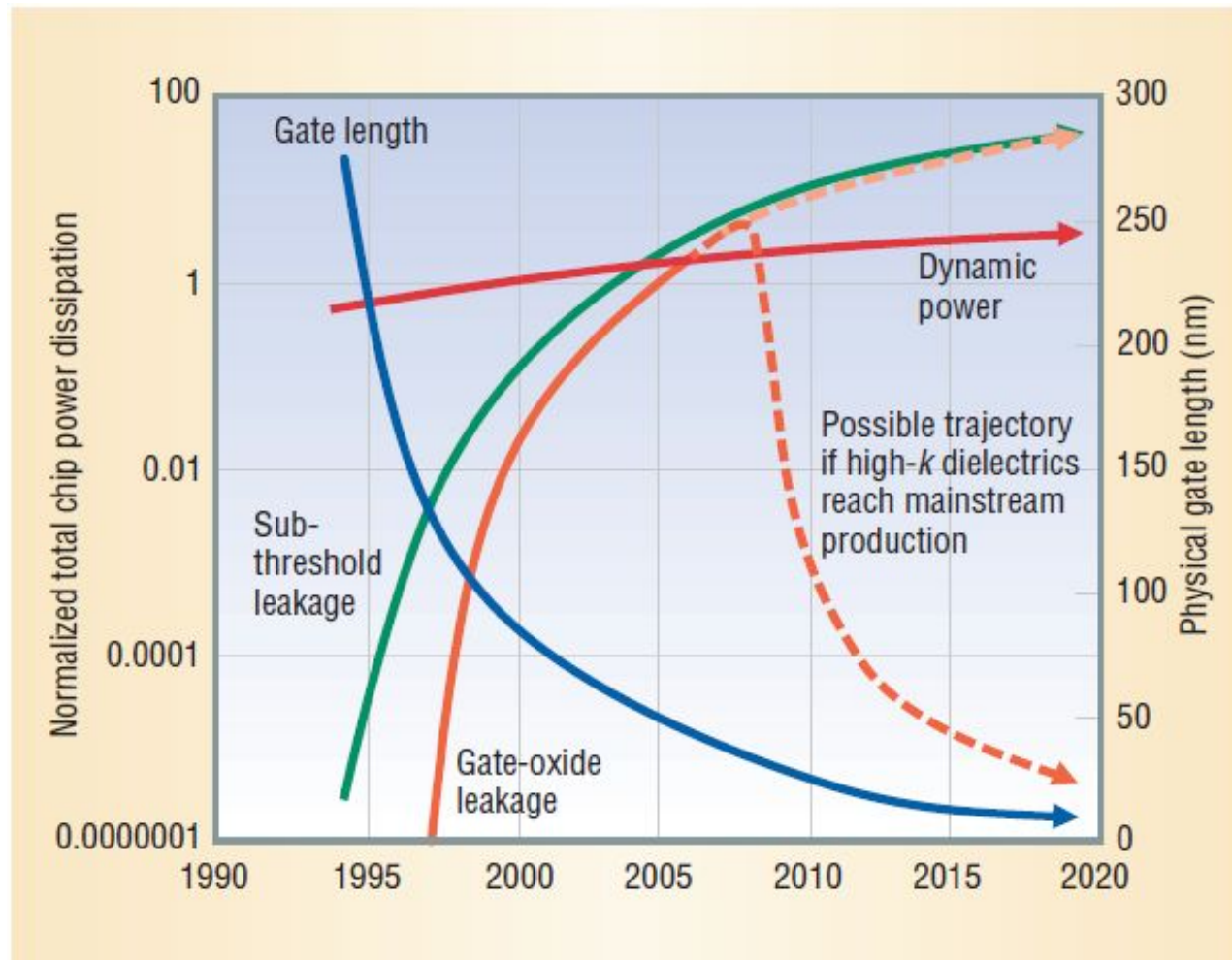
Challenge-1: Data Growth



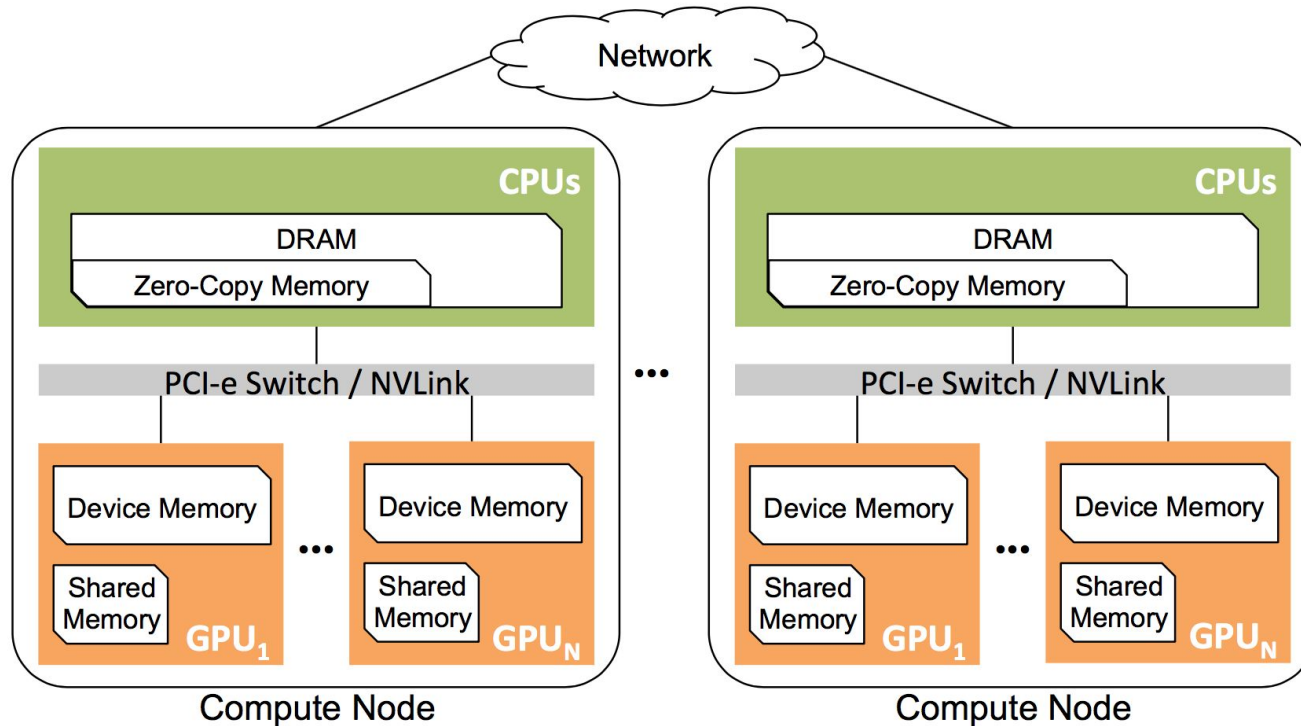
Graph Analytics

- Real World Graphs
~ TB
- Limited GPU Memory
 - NVIDIA P100 has
16 GB memory

Challenge-2: Leakage Energy

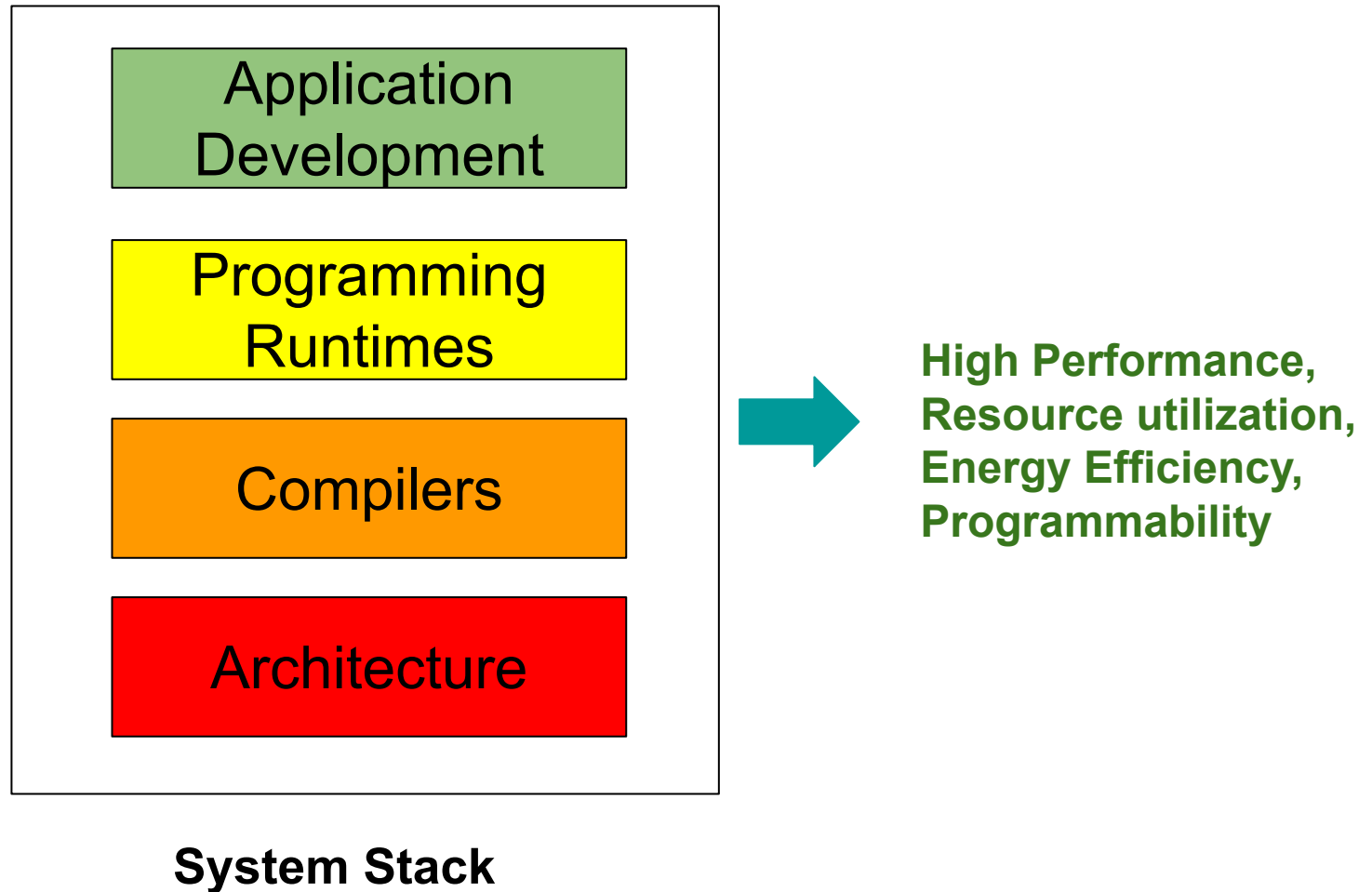


Challenge-3: High-Performance

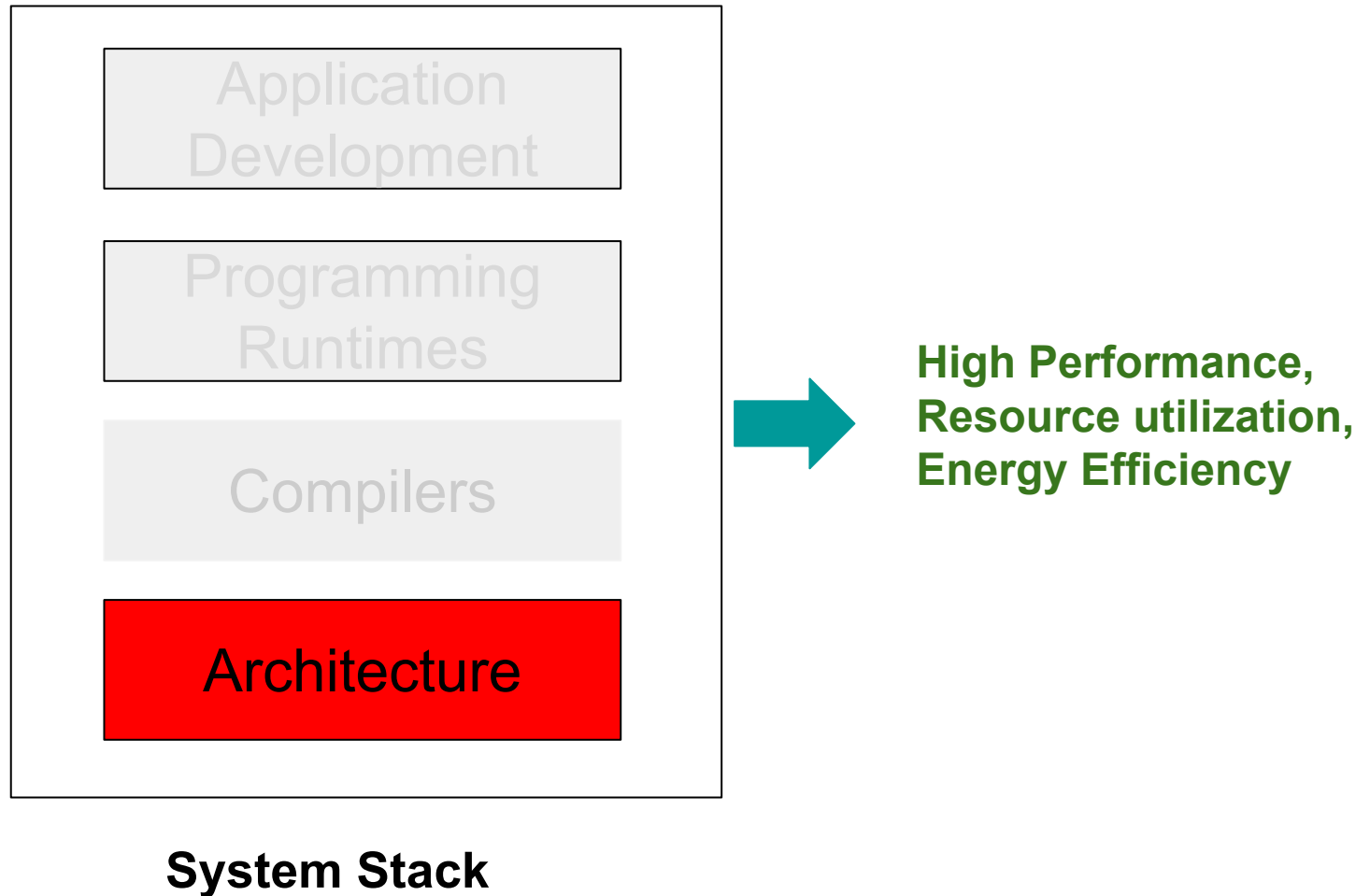


HPC Challenges: Partitioning, Synchronization, and Computation!

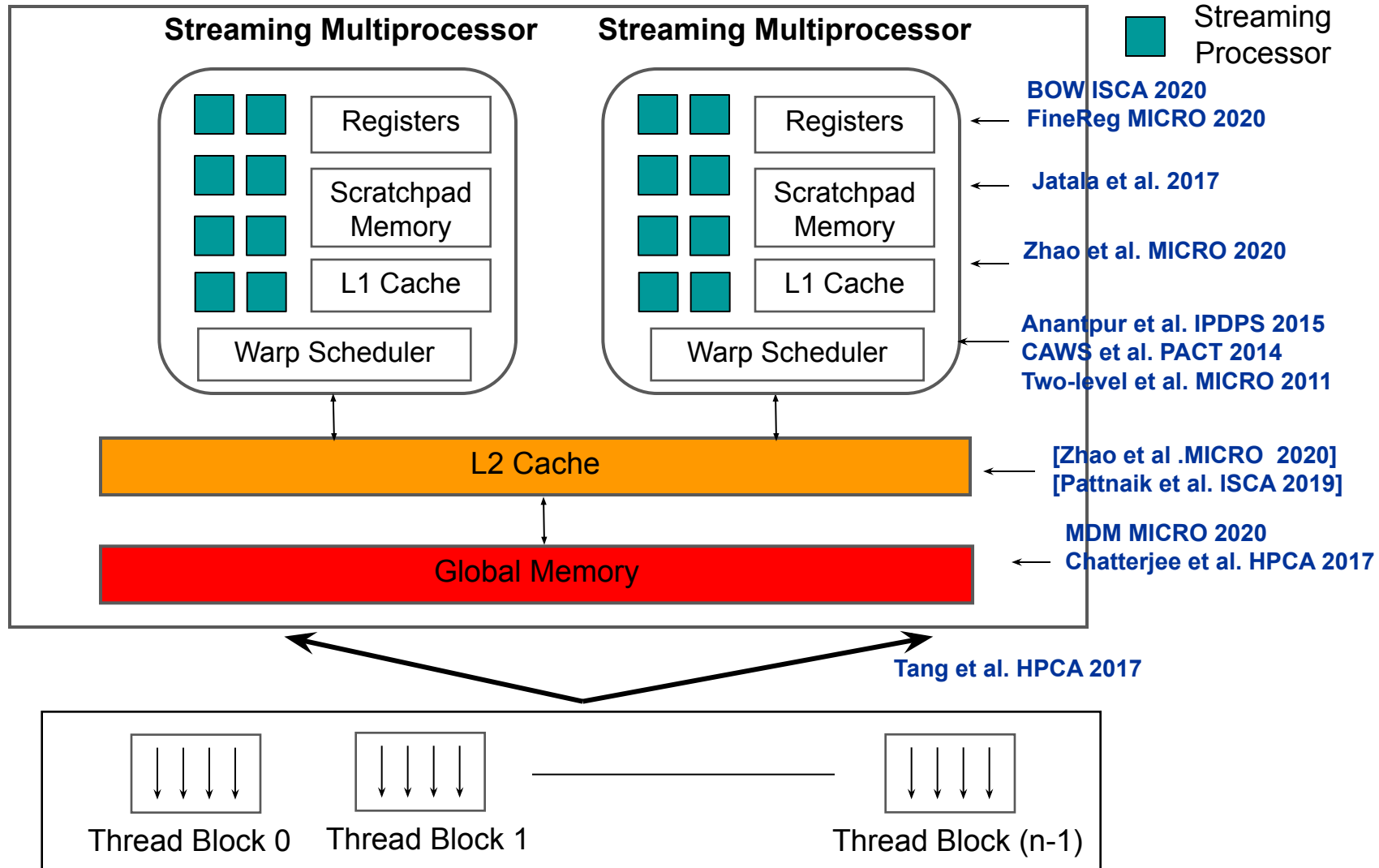
Research Directions



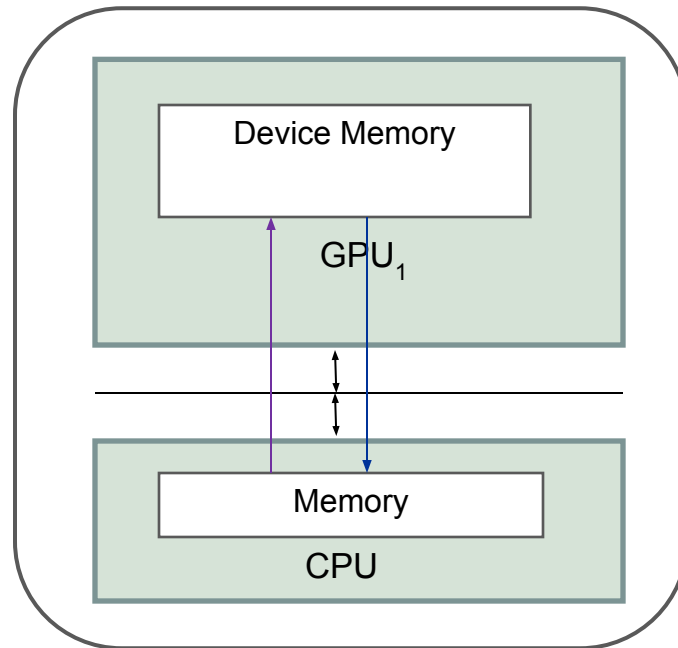
Research Overview: Architecture



Architectural Research: Performance and Energy



Architectural Research: Memory Limitations



Compute Node

Unified Memory

Griffin: HPCA 2020

Shin ISCA 2018

Ganguly et al. HPCA 2017

Mosaic MICRO 2017

Other Architectural Research Directions

- CPU-GPU heterogeneous architectures
 - [Mittal et al. ACM Computing Surveys 2015]
- Machine learning <----> GPUs
 - [Duplo MICRO 2020], [Poise HPCA 2019], [DeftNN MICRO 2017]
- High-performance distributed GPUs
 - [Chu et al. HiPC 2019], [Chu et al. GPGPU 2019], [Awan et al. EuroMPI 2018]
- Developing/Improving the GPU simulators
 - [Accel-Sim ISCA 2020], [HMG: HPCA 2020], [MGPUSim ISCA 2019]

Summary

- GPUs are popular platforms for developing applications
 - Introduction to GPU; CUDA Programming; Optimizations
- Limitations:
 - Memory
 - Performance
 - Energy efficiency
- Research:
 - High-performance and energy efficient architectures; Distributed systems

Questions?