EE-508: Hardware Foundations for Machine Learning GPU Programming

University of Southern California

Ming Hsieh Department of Electrical and Computer Engineering

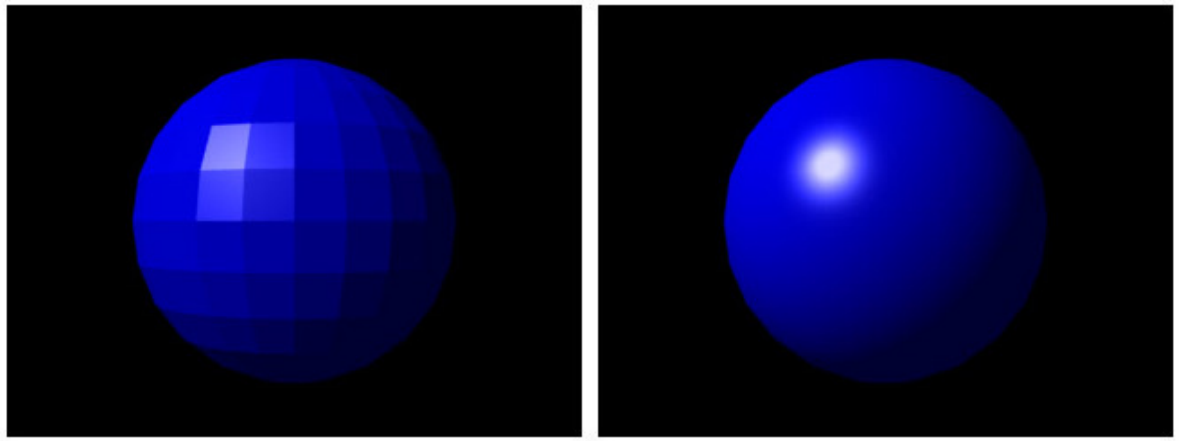
Instructors:
Arash Saifhashemi

Shader vs Cuda Programming


GPU was originally designed for graphics
CALL·DUTY
WARZONE

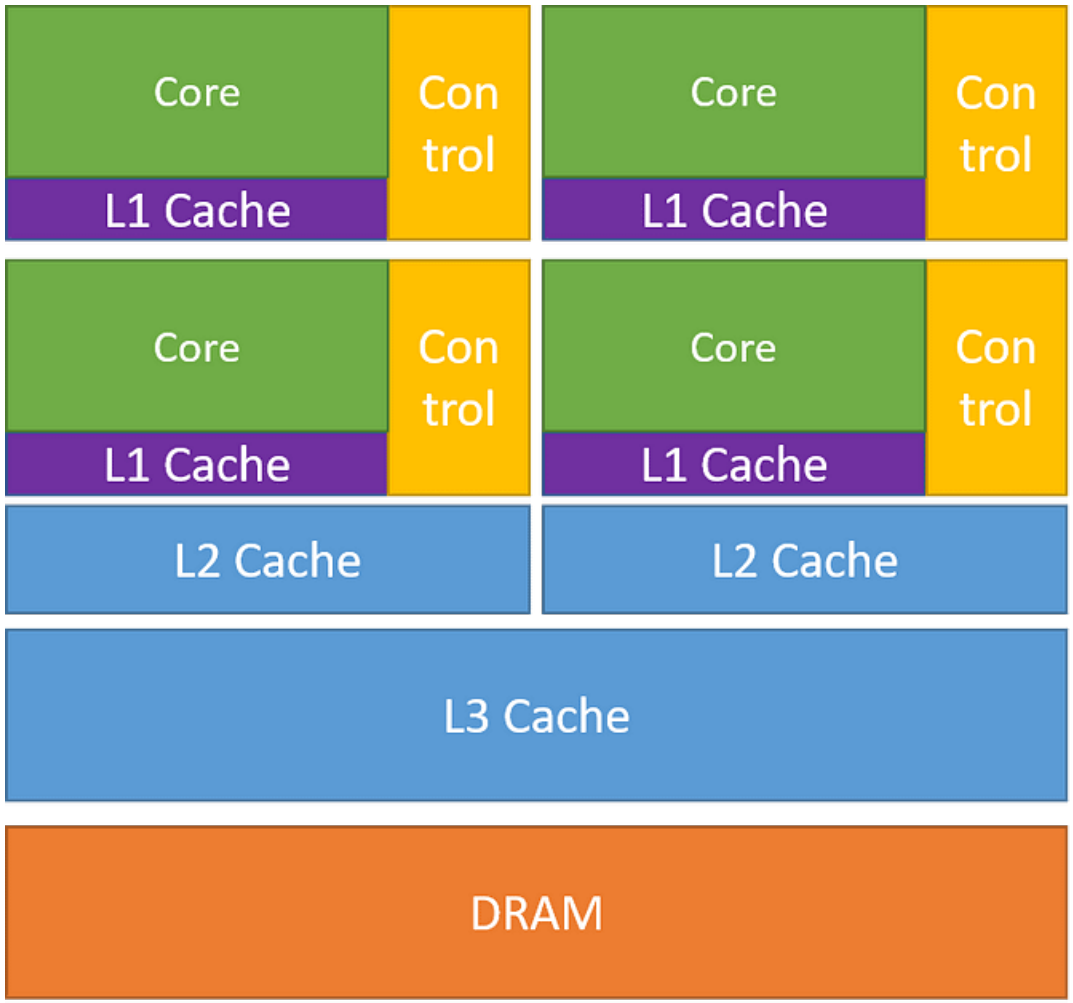
Unreal Engine

Shading and Texture

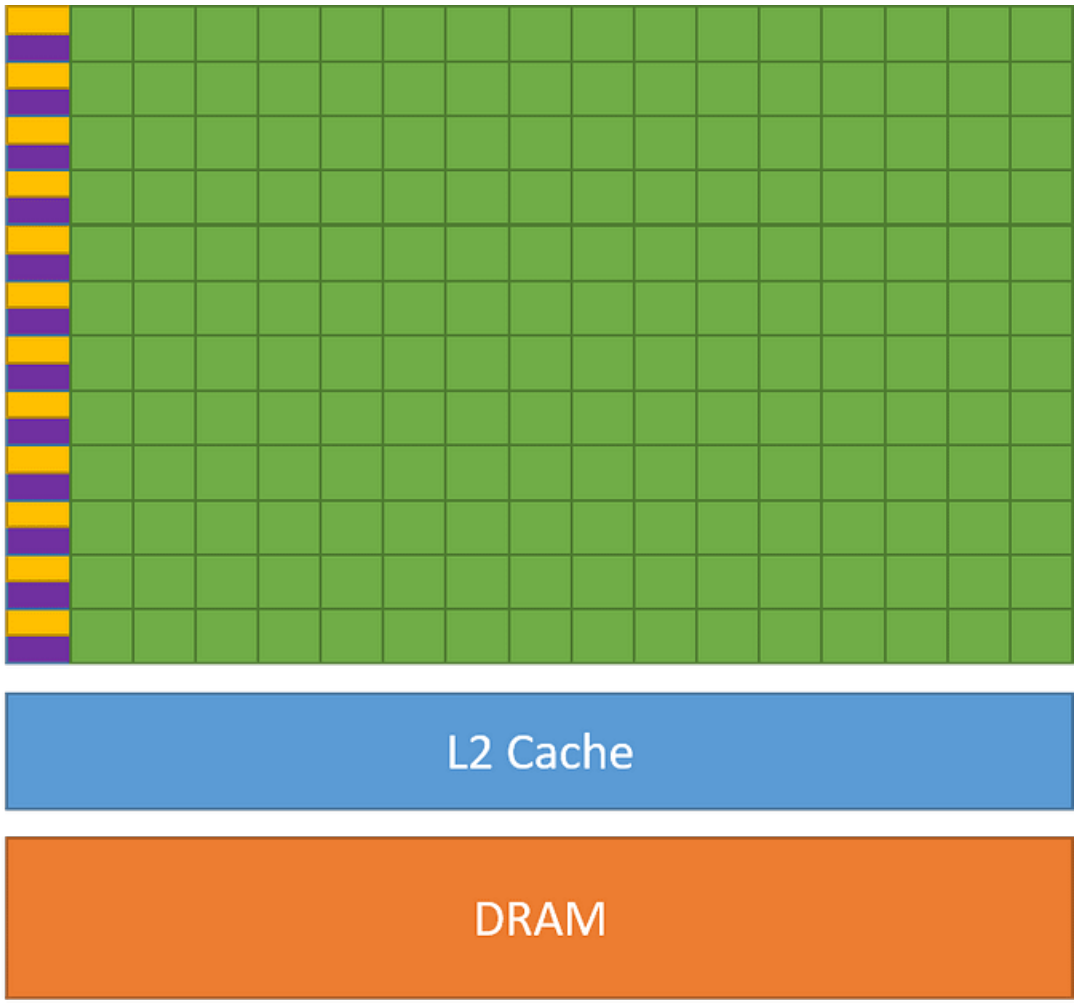


An example of two kinds of shaders: Flat shading on the left and Phong shading on the right.

GPU vs CPU



CPU

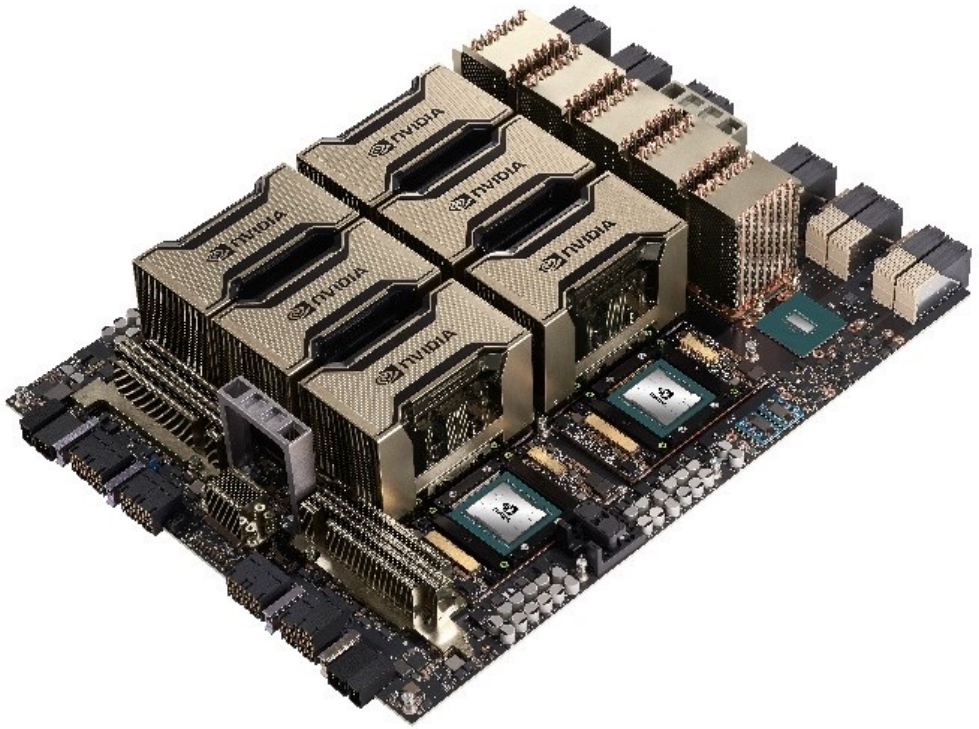


GPU

GPU vs GPGPU

- GPU: Graphics Processing Unit
- GPGPU: General Purpose GPU
  - Using a GPU for non-graphics computations
    - E.g.: AI, deep learning, scientific simulations, and big data processing

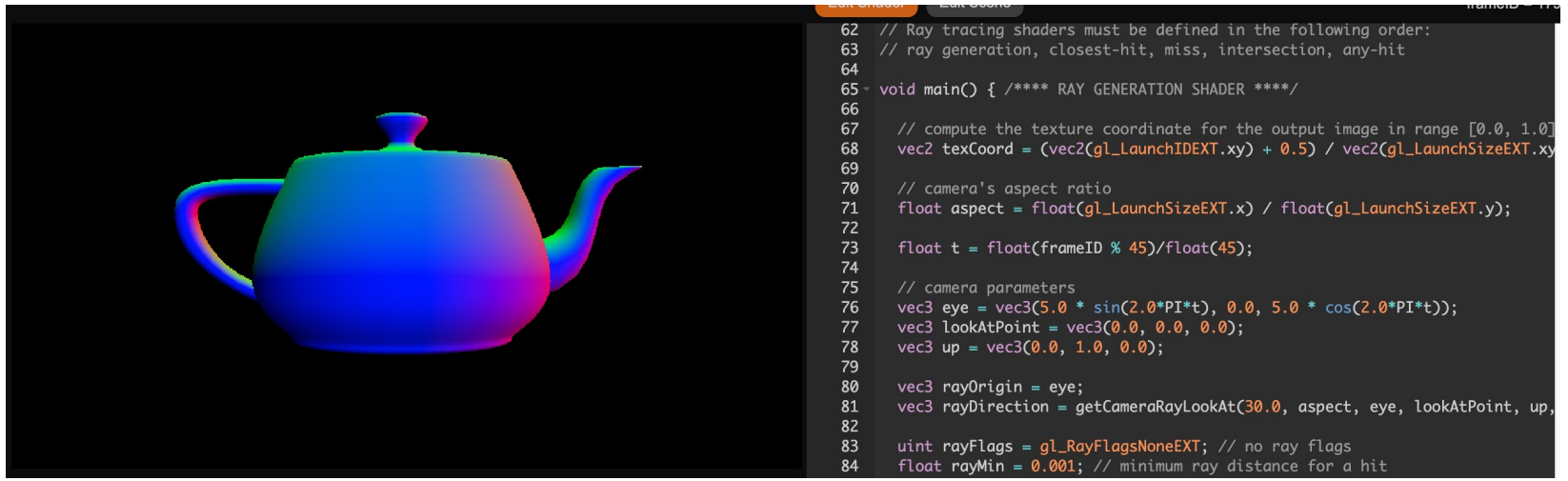
A manrepairing mining equipment in the basement of a home in Zaarouriyeh.
Source CNBC



NVIDIA HGX A100 consisting of 4 or 8 NVIDIA A100 Tensor Core GPUs

Shader Programming

- Small programs that run on the GPU to control how pixels and vertices are processed and displayed on the screen
- **Flexibility**: Shaders can also be used for general-purpose computing tasks. However, this use is somewhat limited by the design and structure of the shading languages (like GLSL or HLSL) .
- **Portability:** Shaders written in languages like GLSL can run on a wide range of GPUs from different manufacturers, offering a degree of portability across platforms.

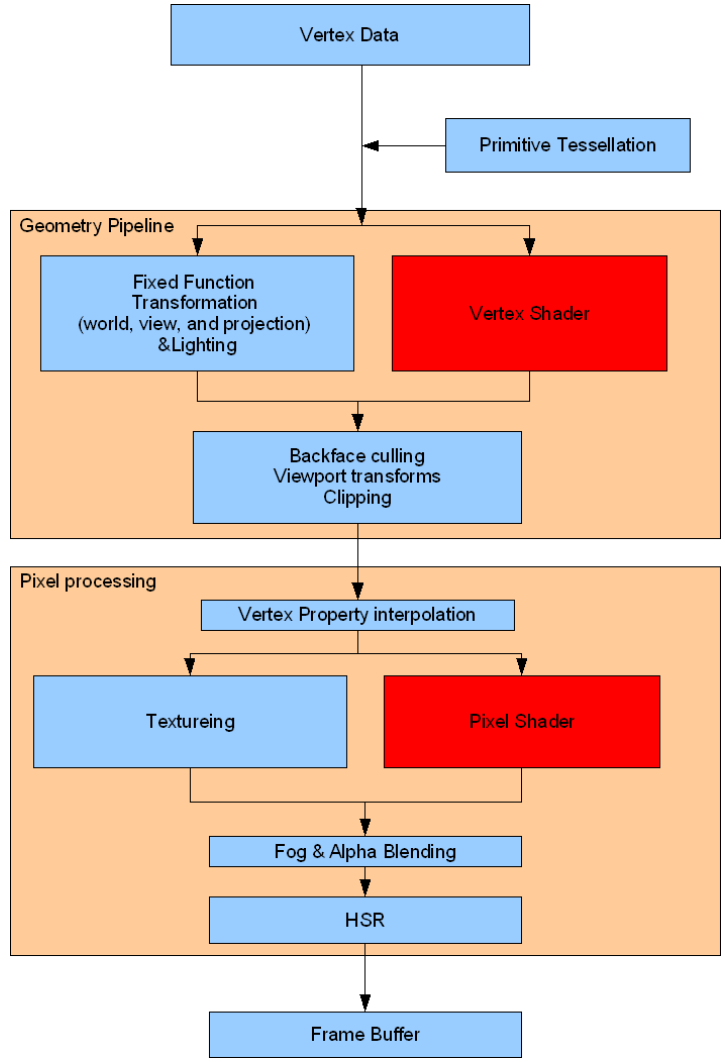


```
// Ray tracing shaders must be defined in the following order:
// ray generation, closest-hit, miss, intersection, any-hit

void main() { /**** RAY GENERATION SHADER ****/

  // compute the texture coordinate for the output image in range [0.0, 1.0]
  vec2 texCoord = (vec2(gl_LaunchIDEXT.xy) + 0.5) / vec2(gl_LaunchSizeEXT.xy

  // camera's aspect ratio
  float aspect = float(gl_LaunchSizeEXT.x) / float(gl_LaunchSizeEXT.y);

  float t = float(frameID % 45)/float(45);

  // camera parameters
  vec3 eye = vec3(5.0 * sin(2.0*PI*t), 0.0, 5.0 * cos(2.0*PI*t));
  vec3 lookAtPoint = vec3(0.0, 0.0, 0.0);
  vec3 up = vec3(0.0, 1.0, 0.0);

  vec3 rayOrigin = eye;
  vec3 rayDirection = getCameraRayLookAt(30.0, aspect, eye, lookAtPoint, up,

  uint rayFlags = gl_RayFlagsNoneEXT; // no ray flags
  float rayMin = 0.001; // minimum ray distance for a hit
```

Source: https://www.gsn-lib.org/docs/nodes/raytracing.php

How is a Shader Program Run on GPU?

- Shader Program Binaries:
  - The application sends GPU shader program binaries through the graphics driver.
- Set Pipeline Parameters:
  - The application configures graphics pipeline settings, such as output image dimensions.
- Vertex Buffer:
  - The application supplies the GPU with a buffer containing vertex data (attributes that define the geometry of objects in 3D rendering).
- Draw Command:
  - The application issues a "draw" command to the GPU.



Source:
https://glasnost.itcarlow.ie/~powerk/GeneralGraphicsNotes/Theory/Shaders.htm

CUDA

- Compute Unified Device Architecture
  - A parallel computing platform and programming model to use NVIDIA GPUs for general purpose processing (GPGPU).
  - **Flexibility**: Provides a wide range of features for complex parallel computations.
    - Memory management, parallel execution of threads, and direct access to GPU hardware features.
  - **Portability**: Proprietary to NVIDIA GPUs.

Standard C Code

```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;



// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

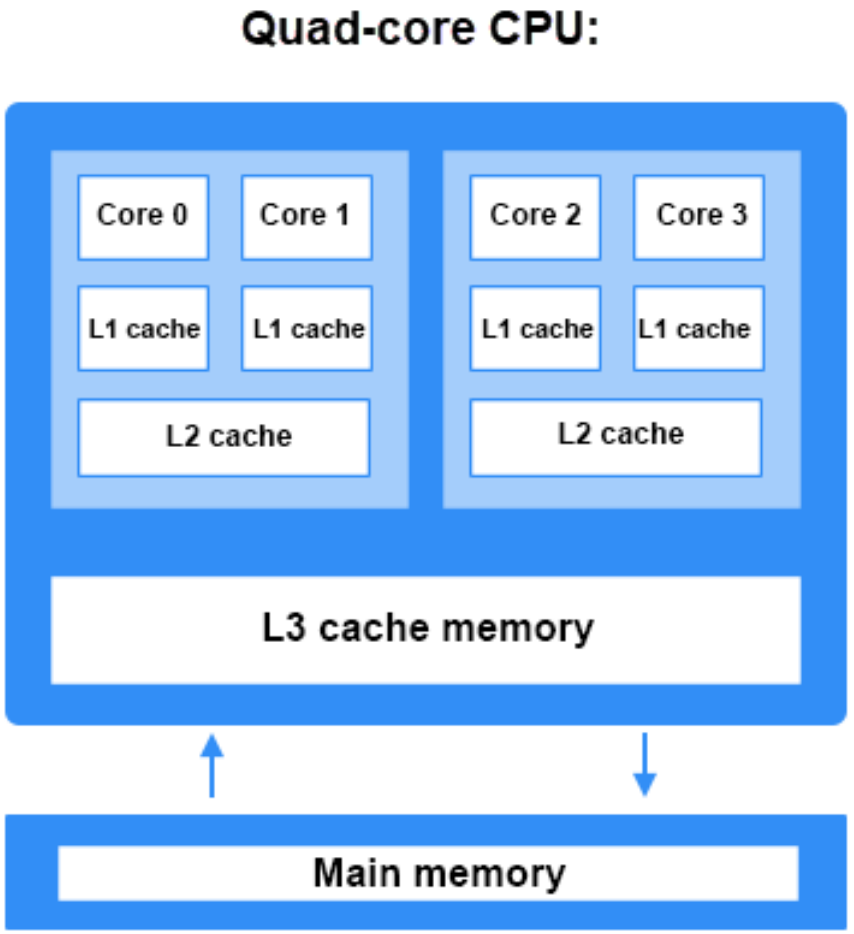
C with CUDA extensions

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

int N = 1<<20;
cudaMemcpy(x, d_x, N, cudaMemcpyHostToDevice);
cudaMemcpy(y, d_y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);

cudaMemcpy(d_y, y, N, cudaMemcpyDeviceToHost);
```

Source: NVIDIA

How is a Program Run on CPU?

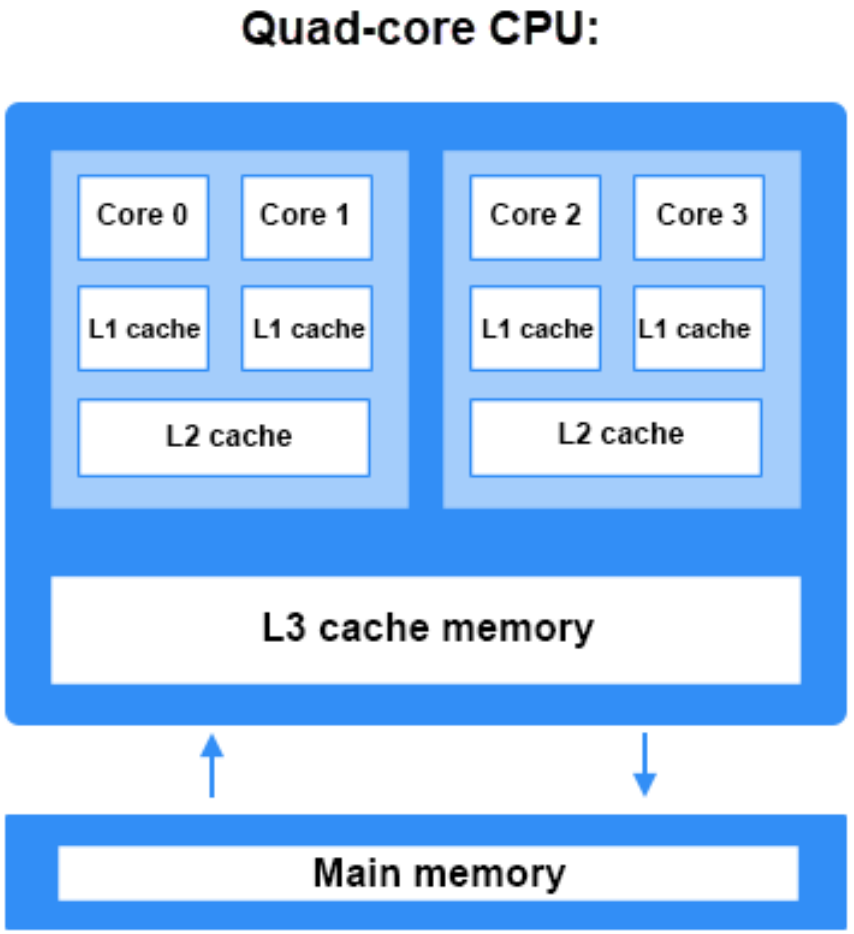
- **Load Program**: OS loads program into memory.



Source: phoenixnap.com

How is a Program Run on CPU?

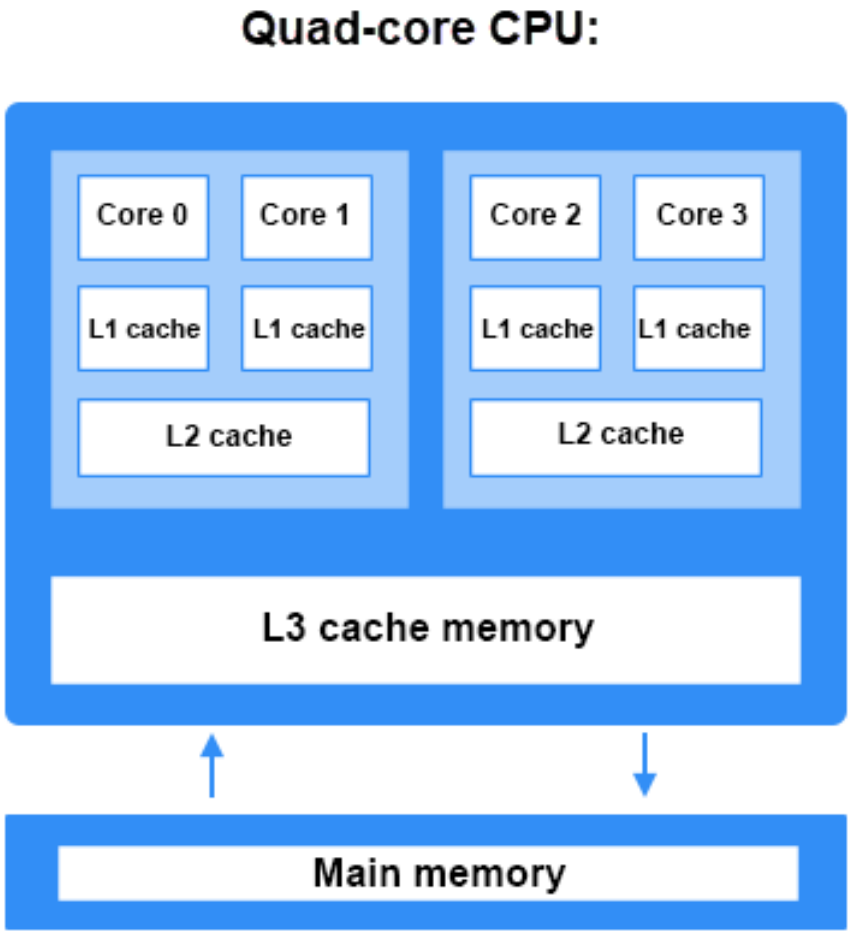
- **Load Program**: OS loads program into memory.
- **Select Context**: OS selects a CPU execution context.
  - An execution context includes information necessary for the CPU to execute the program, such as the state of registers.



Source: phoenixnap.com

How is a Program Run on CPU?

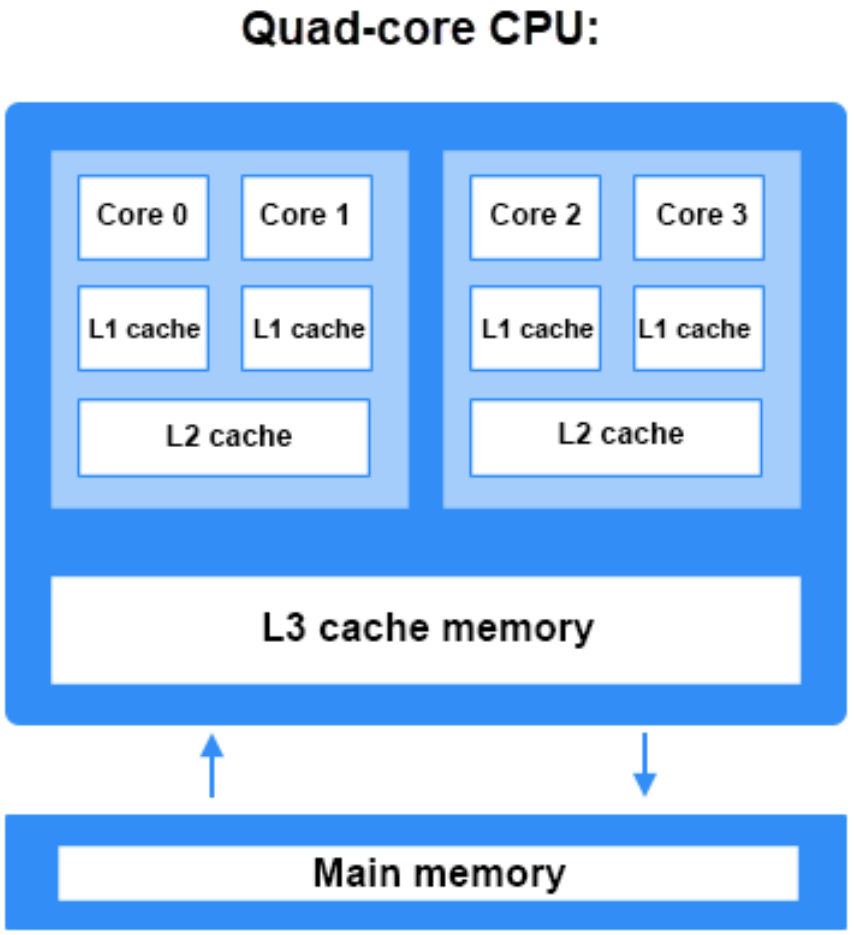
- **Load Program**: OS loads program into memory.
- **Select Context**: OS selects a CPU execution context.
  - An execution context includes information necessary for the CPU to execute the program, such as the state of registers.
- **Prepare Context**: OS sets up execution context.
  - Setting the contents of the CPU's registers, initializing the program counter to point to the start of the program, and configuring other necessary state information.



Source: phoenixnap.com

How is a Program Run on CPU?

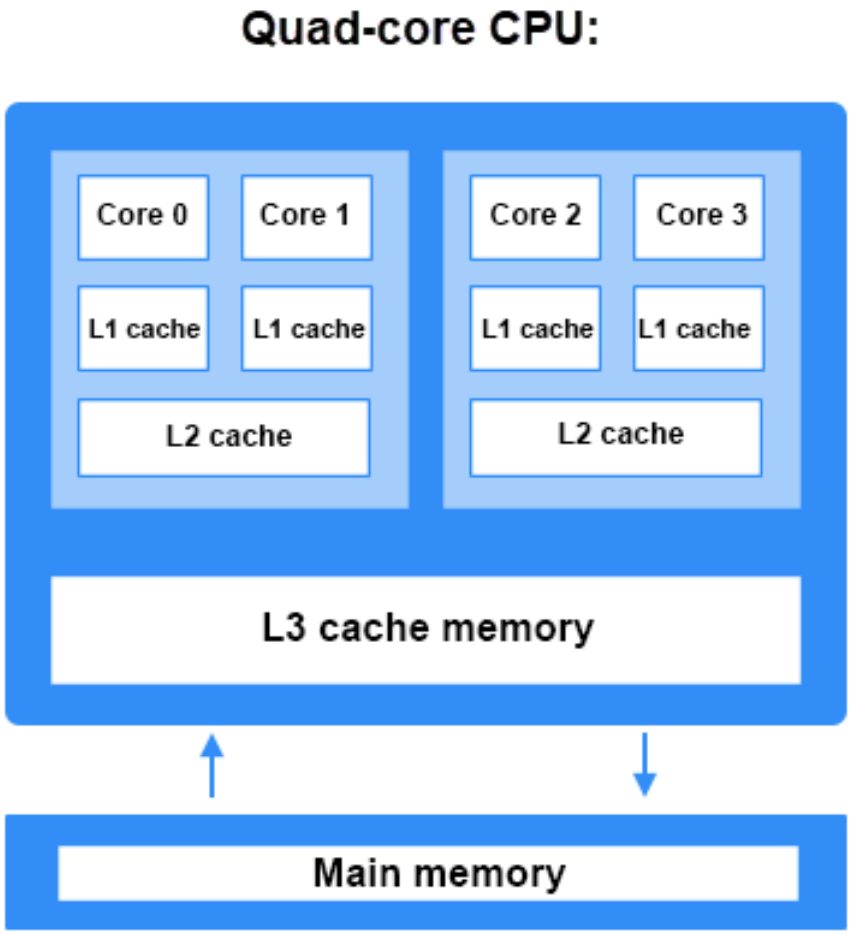
- **Load Program**: OS loads program into memory.
- **Select Context**: OS selects a CPU execution context.
  - An execution context includes information necessary for the CPU to execute the program, such as the state of registers.
- **Prepare Context**: OS sets up execution context.
  - Setting the contents of the CPU's registers, initializing the program counter to point to the start of the program, and configuring other necessary state information.
- **Interrupt CPU and Start Execution:**
  - The OS interrupts the processor to switch its attention to the program's execution context.



Source: phoenixnap.com

How is a Program Run on CPU?

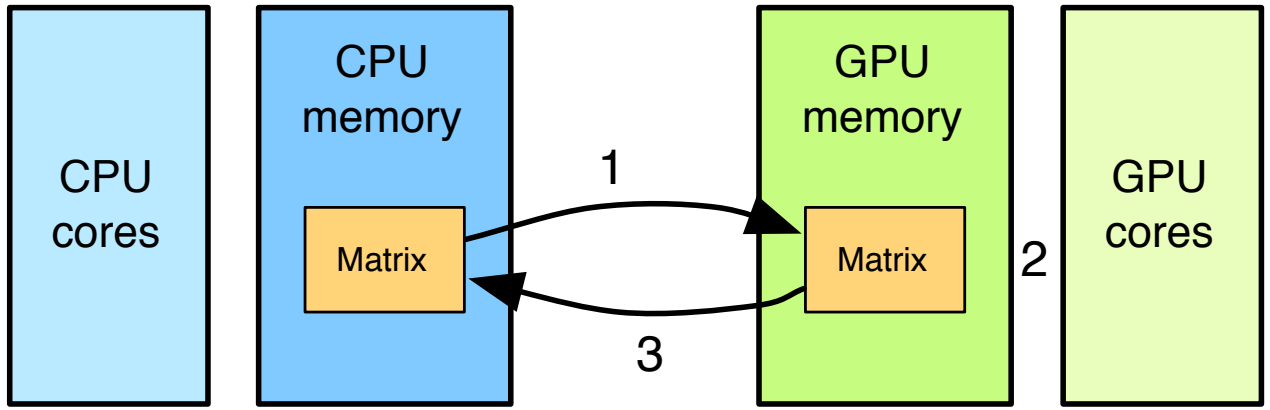
- **Load Program**: OS loads program into memory.
- **Select Context**: OS selects a CPU execution context.
  - An execution context includes information necessary for the CPU to execute the program, such as the state of registers.
- **Prepare Context**: OS sets up execution context.
  - Setting the contents of the CPU's registers, initializing the program counter to point to the start of the program, and configuring other necessary state information.
- **Interrupt CPU and Start Execution:**
  - The OS interrupts the processor to switch its attention to the program's execution context.
- **Execute Instructions**: Processor executes the program's instructions.



Source: phoenixnap.com

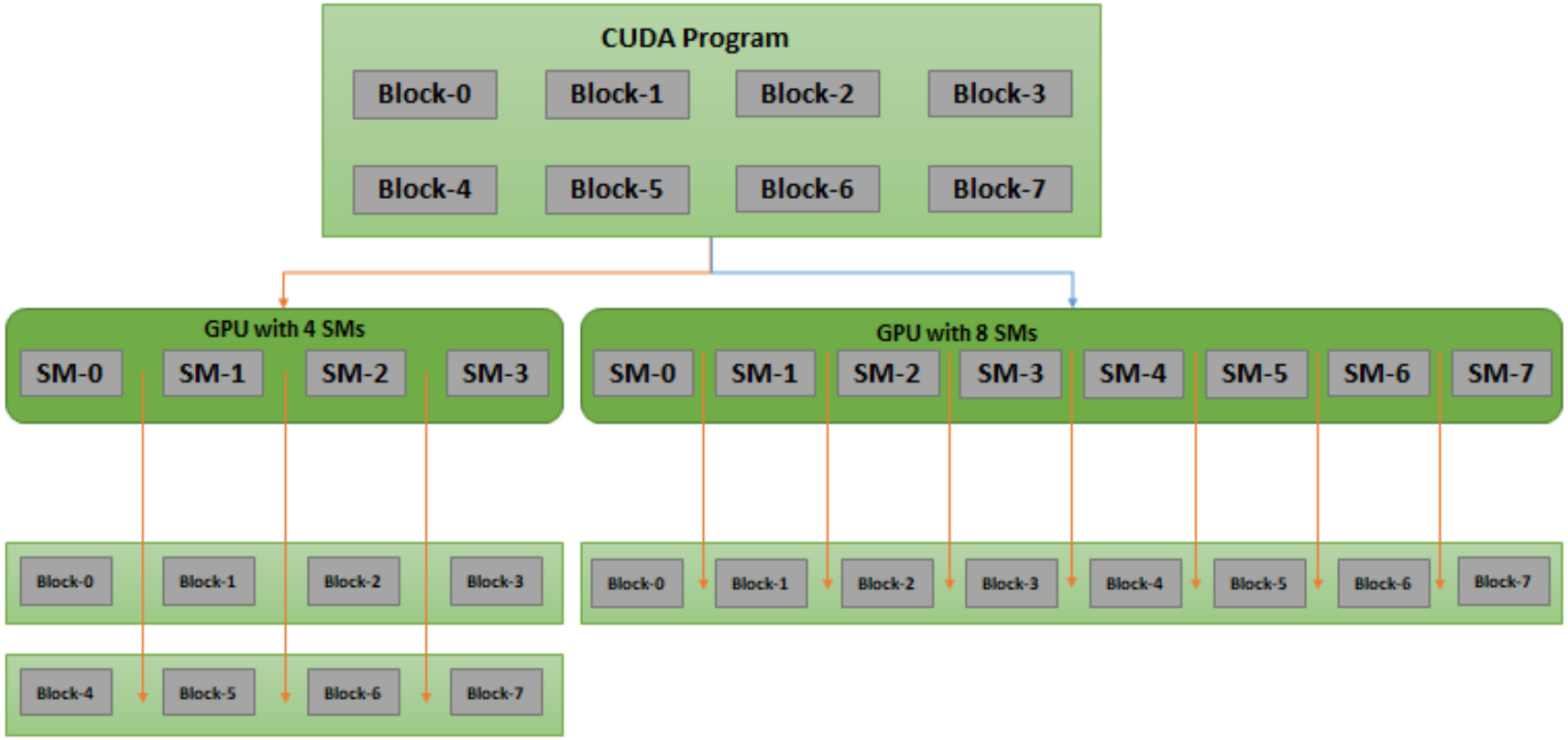
How is a CUDA Program Run on GPU?

- Load Data into CPU Memory
- Copy and allocate Data to GPU Memory
- Execute GPU Kernel
- Synchronize and copy Results to CPU Memory:
  - After processing, transfer the results from the GPU's memory back to the CPU's memory.
- Use Results on CPU



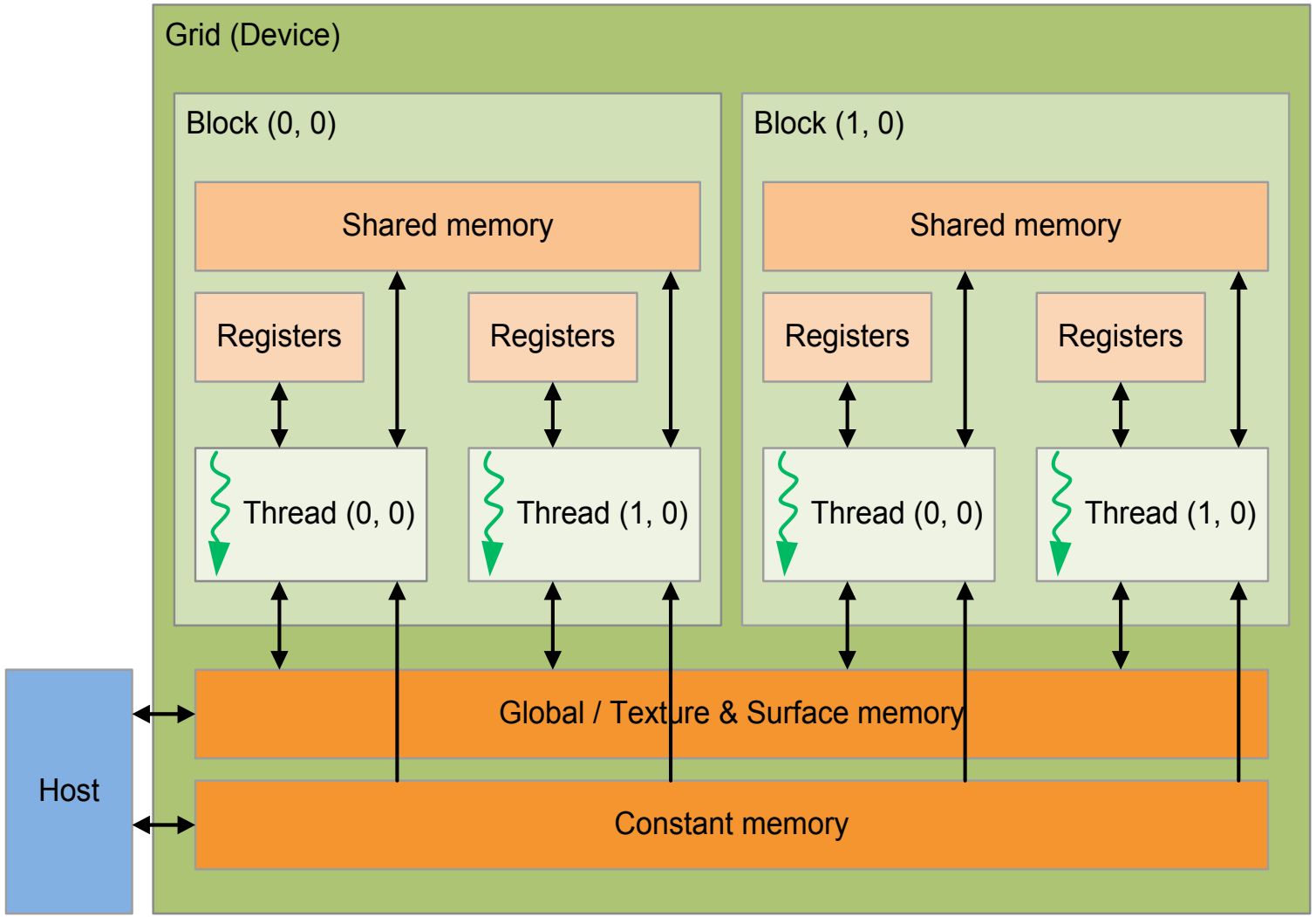
Hardware Abstraction

- Hardware will assign the thread blocks on SMs



Memory Hierarchy

- Hardware will assign the thread blocks on SMs



Traditional Program Structure in CUDA

- Function prototypes

```
float serialFunction(…);
__global__ void kernel(…)
```

- `main()`
  - Allocate memory space on the device – `cudaMalloc(&d_in, bytes);`
  - Transfer data from host to device – `cudaMemCpy(d_in, h_in, …);`
  - Execution configuration setup: #blocks and #threads
  - Kernel call – `kernel<<<execution configuration>>>(args…);`
  - Transfer results from device to host – `cudaMemCpy(h_out, d_out, …);`
- Kernel : `__global__ void kernel(type args,…)`
  - Automatic variables (local variables declared inside a function) transparently assigned to registers
  - Shared memory: `__shared__`
  - Intra-block synchronization: `__syncthreads();`

CUDA Programming Language

- Memory allocation

```
cudaMalloc((void**)&d_in, #bytes);
```

- Memory copy

```
cudaMemcpy(d_in, h_in, #bytes, cudaMemcpyHostToDevice);
```

- Kernel launch

```
kernel<<< #blocks, #threads >>>(args);
```

- Memory deallocation

```
cudaFree(d_in);
```

- Explicit synchronization

```
cudaDeviceSynchronize();
```

NVCC Compiler

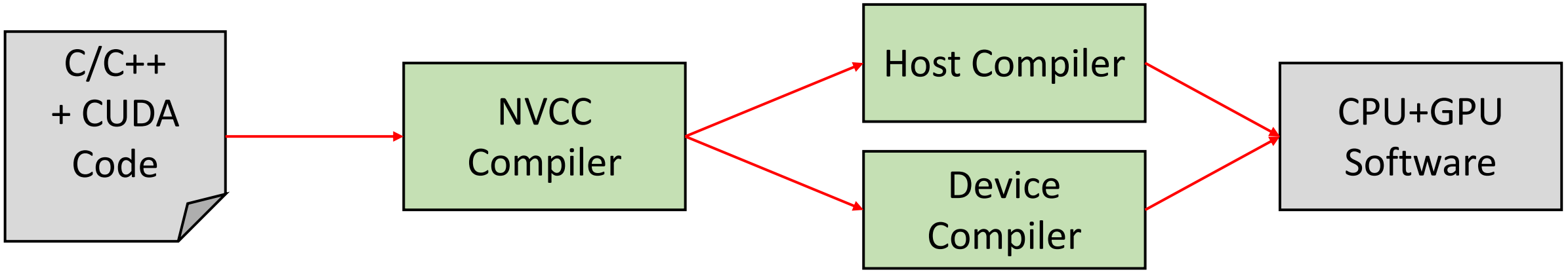
Asynchronous call

CPU side

```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

GPU side

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```



Slide credit: Sang-Woo Jun

Simple CUDA Example

```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

1 block

N threads per block

Should wait for kernel to finish

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Only void allowed

Which of N threads am I?
See also: blockIdx

N instances of VecAdd spawned in GPU

__global__:
In GPU, called from host/GPU
__device__:
In GPU, called from GPU
__host__:
In host, called from host

One function can be both

Slide credit: Sang-Woo Jun

Indexing and Memory Access

- Images are 2D data structures
  - height x width
  - Image[j][i], where 0 ≤ j < height, and 0 ≤ i < width

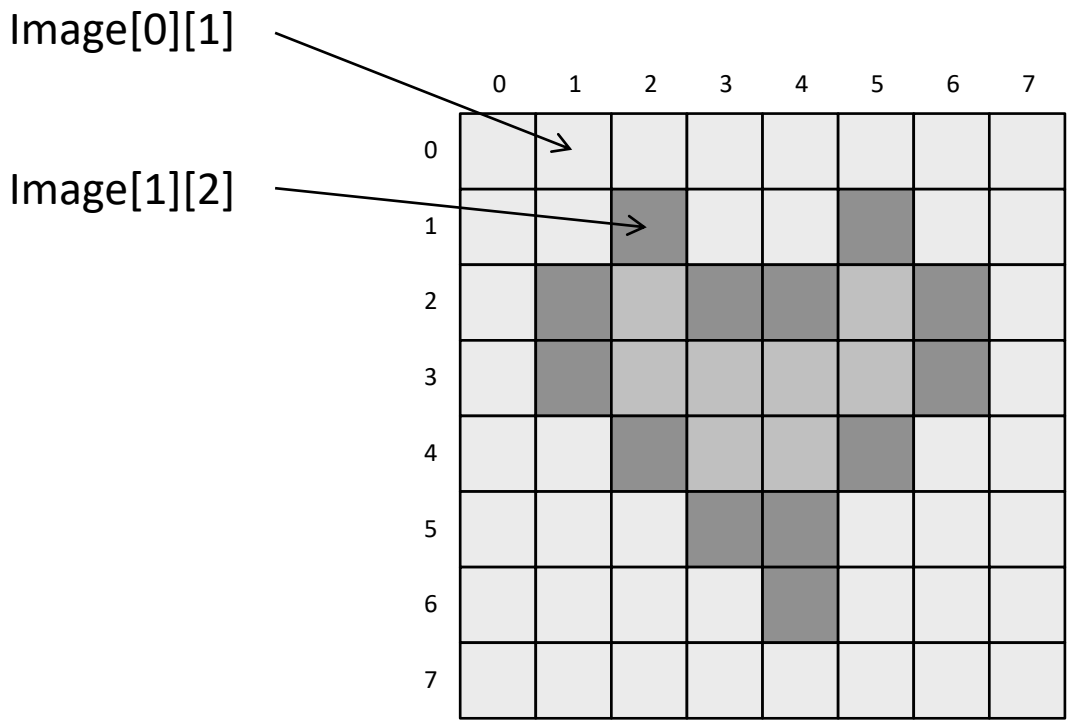
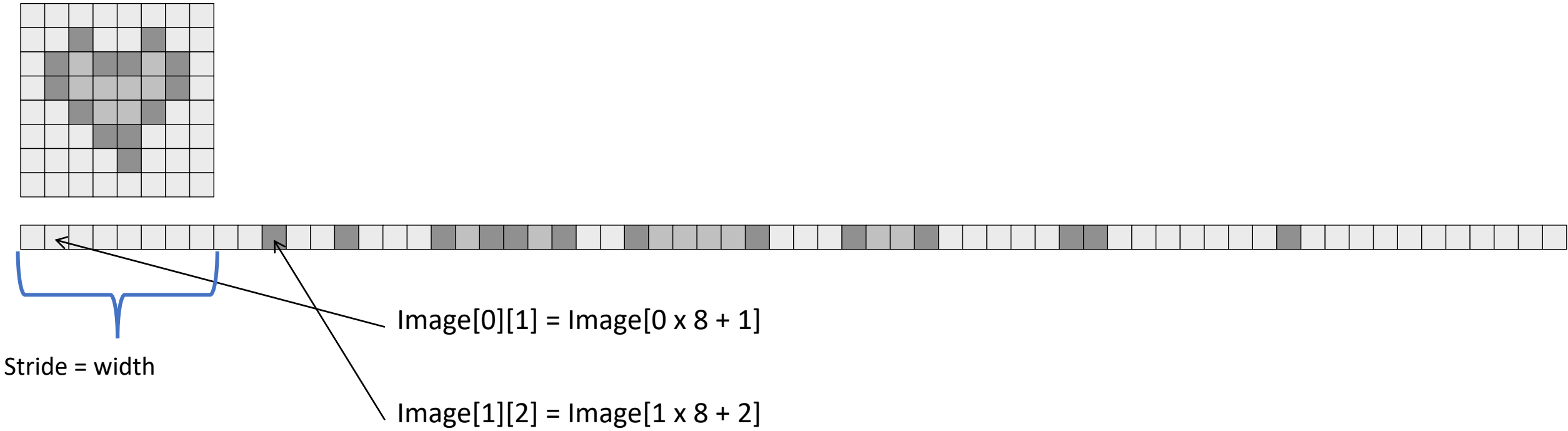


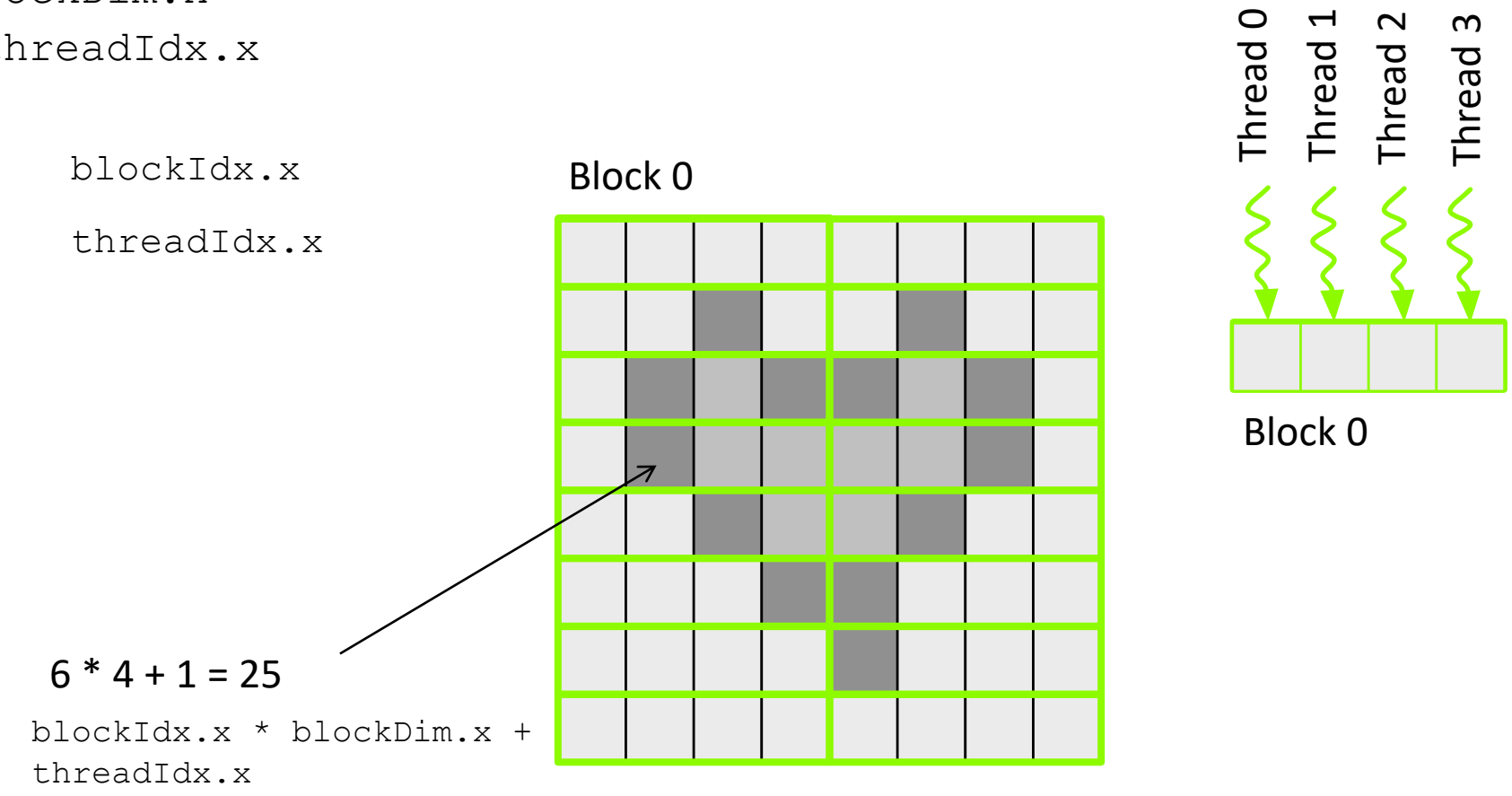
Image Layout in Memory

- Row-major layout
- `Image[j][i] = Image[j x width + i]`



Indexing and Memory Access: 1D Grid

- One GPU thread per pixel
- Grid of Blocks of Threads
  - `gridDim.x, blockDim.x`
  - `blockIdx.x, threadIdx.x`

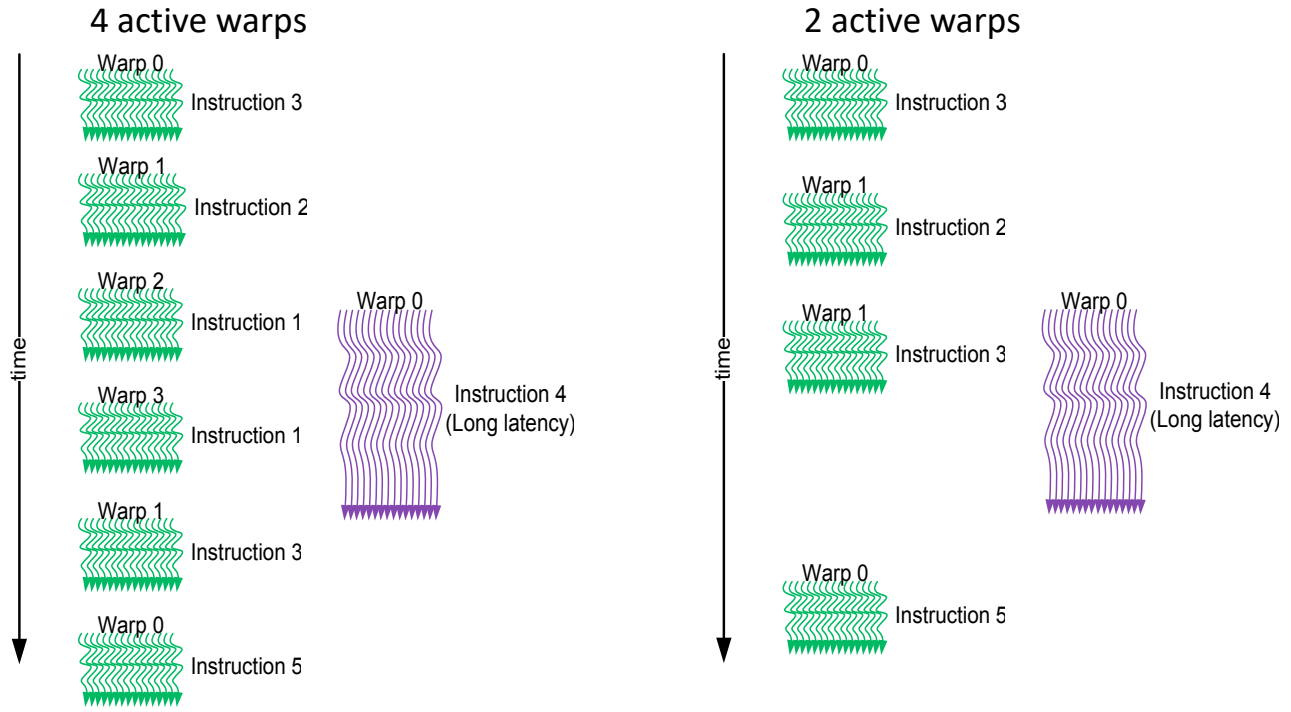


Performance Considerations

- Main bottlenecks
  - Global memory access
  - CPU-GPU data transfers
- Memory access
  - Latency hiding
    - Occupancy
  - Memory coalescing
  - Data reuse
    - Shared memory usage
- SIMD (Warp) Utilization: Divergence
- Atomic operations: Serialization
- Data transfers between CPU and GPU
  - Overlap of communication and computation

Latency Hiding

- FGMT can hide long latency operations (e.g., memory accesses)
- Occupancy: ratio of active warps



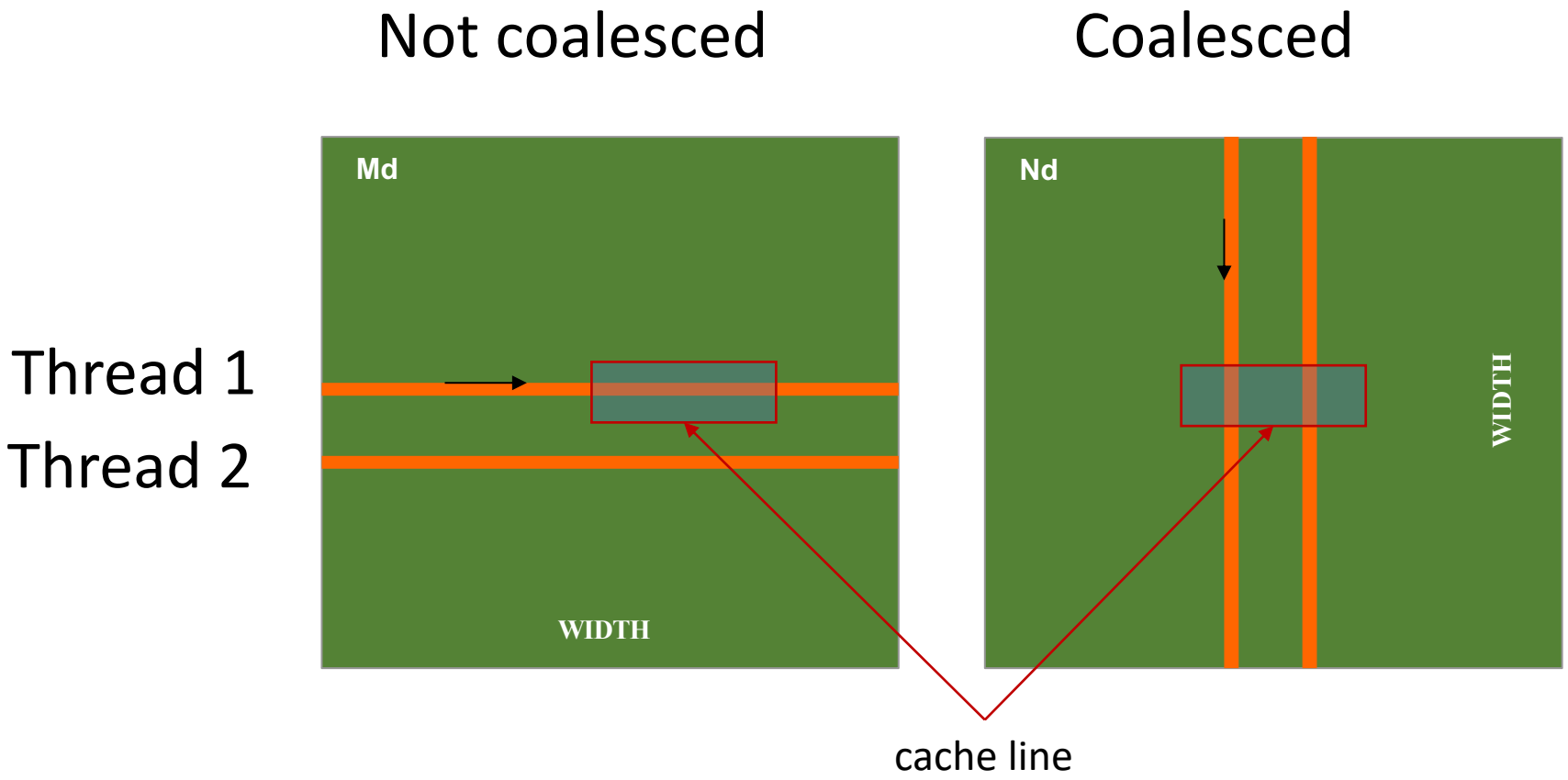
- Idle or stalled warps appear in purple waves.
  - When a warp accesses global memory, it stalls (waiting for memory).
  - If there are enough active warps, the GPU can schedule another warp, hiding the latency.

Left Side (Good Warp Scheduling - Full Latency Hiding):
•Many active warps (green waves) execute while some are waiting for memory (purple).
•The **GPU remains fully utilized**, maximizing throughput.

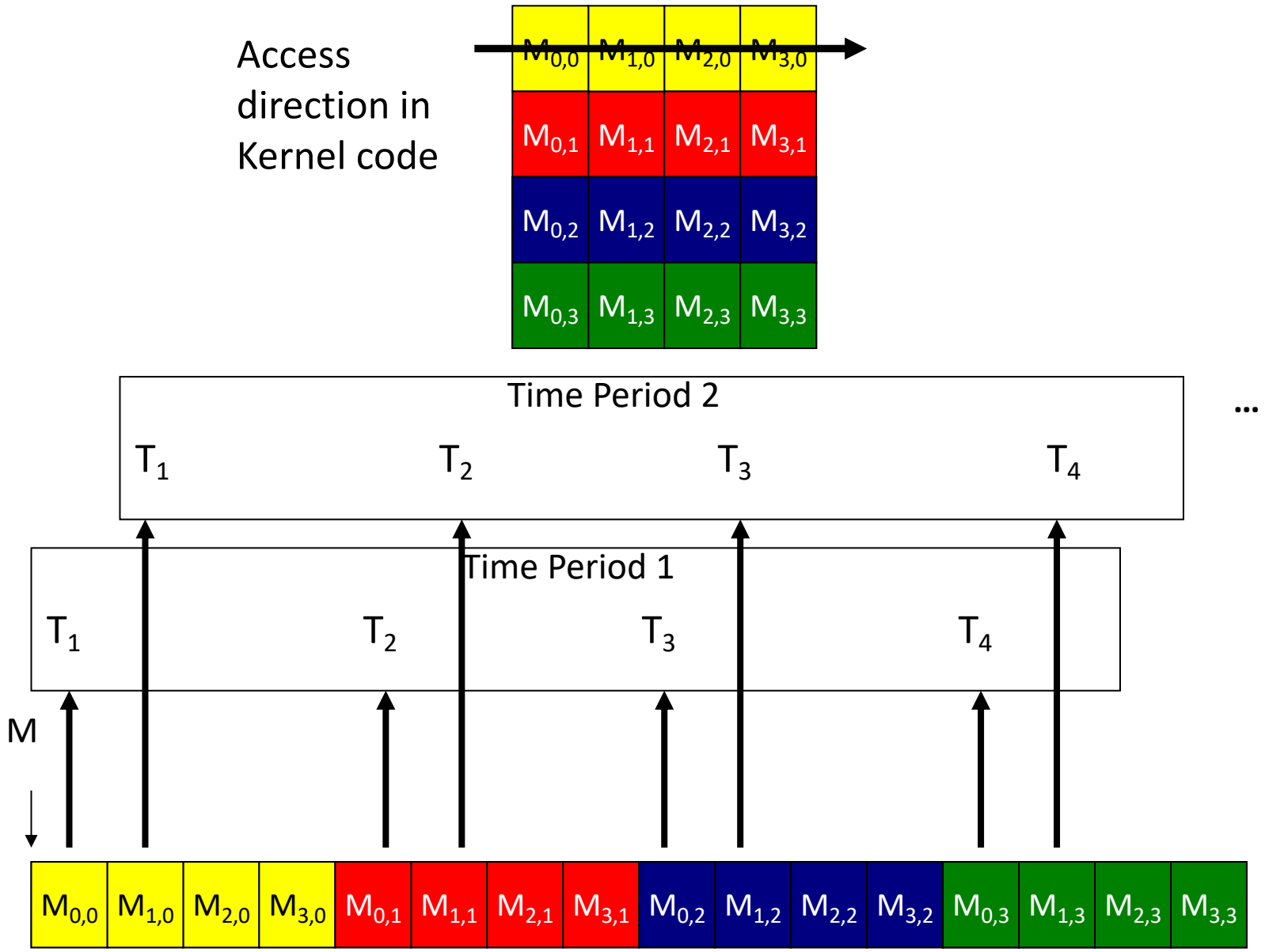
- **Right Side (Bad Warp Scheduling - Partial Latency Hiding)**:
  - Fewer warps are available for execution.
  - When one warp stalls due to memory access, **not enough warps are ready** to take over.
  - This leads to **idle execution slots**, reducing performance.

Memory Coalescing

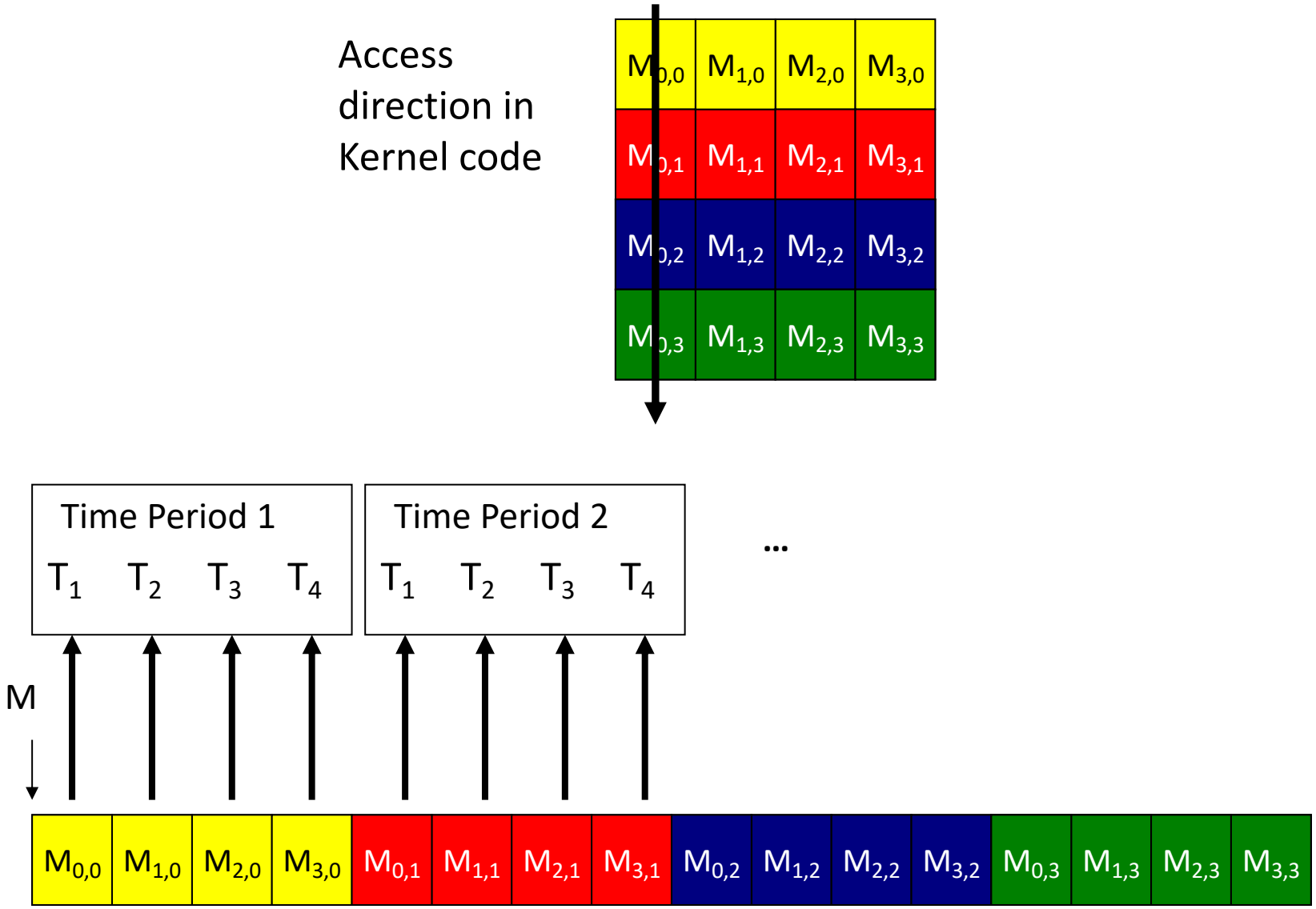
- When accessing global memory, we want to make sure that concurrent threads access nearby memory locations
- Peak bandwidth utilization occurs when all threads in a warp access one cache line



Uncoalesced Memory Accesses



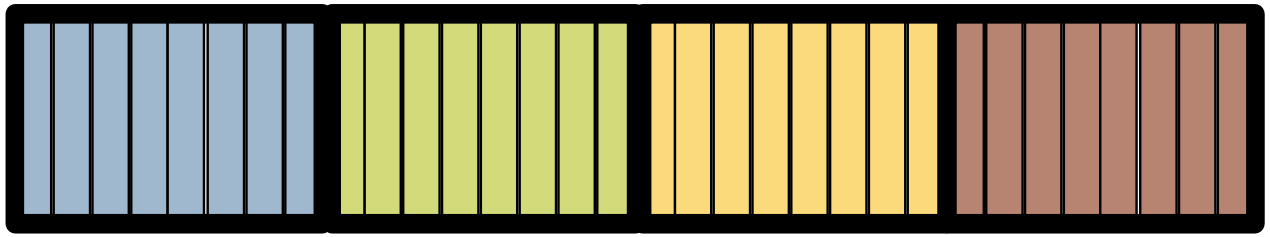
Coalesced Memory Accesses



Array of Structures vs. Structure of Arrays

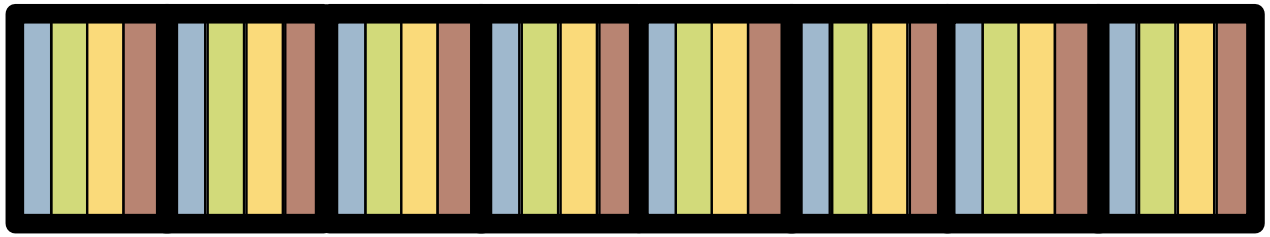
Structure of Arrays (SoA)

```
struct foo{
  float a[8];
  float b[8];
  float c[8];
  int d[8];
} A;
```



Array of Structures (AoS)

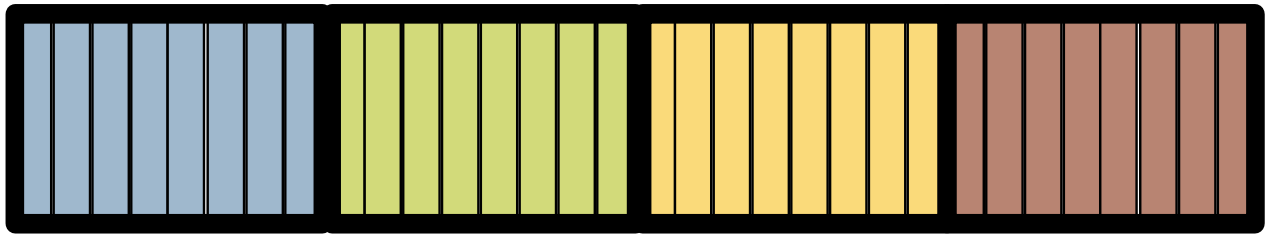
```
struct foo{
  float a;
  float b;
  float c;
  int d;
} A[8];
```



Array of Structures vs. Structure of Arrays

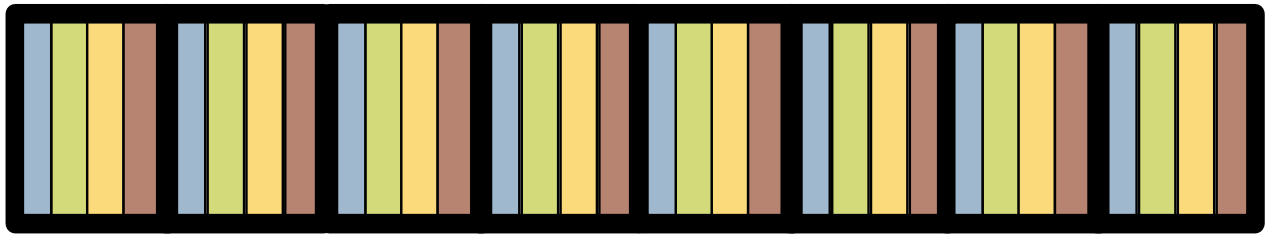
Structure of Arrays (SoA)

```
struct foo{
  float a[8];
  float b[8];
  float c[8];
  int d[8];
} A;
```



Array of Structures (AoS)

```
struct foo{
  float a;
  float b;
  float c;
  int d;
} A[8];
```

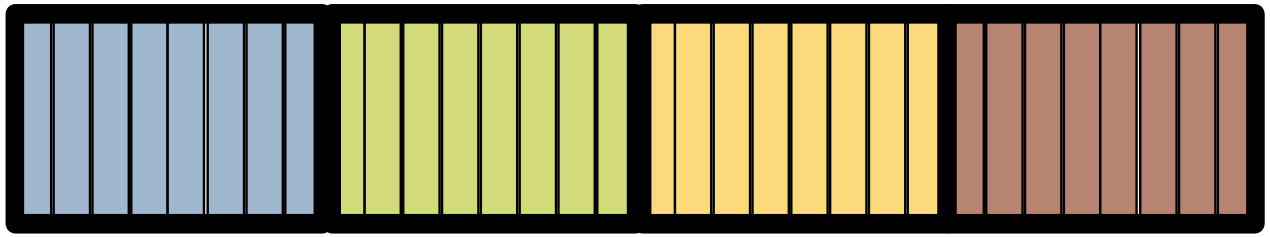


- Which one is better for a program on CPU?

Array of Structures vs. Structure of Arrays

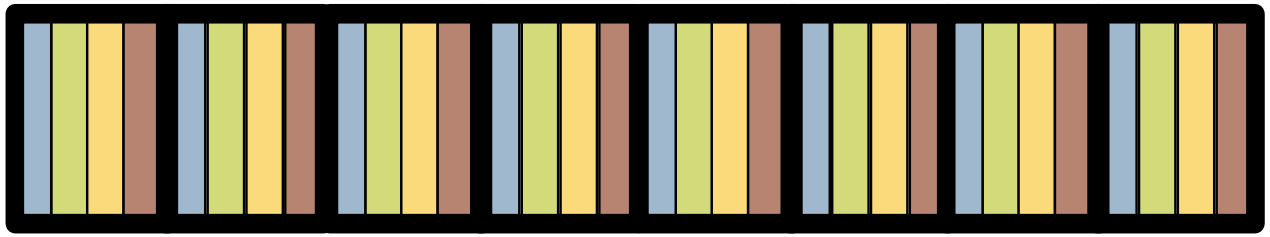
Structure of Arrays (SoA)

```
struct foo{
 float a[8];
 float b[8];
 float c[8];
 int d[8];
} A;
```



Array of Structures (AoS)

```
struct foo{
 float a;
 float b;
 float c;
 int d;
} A[8];
```

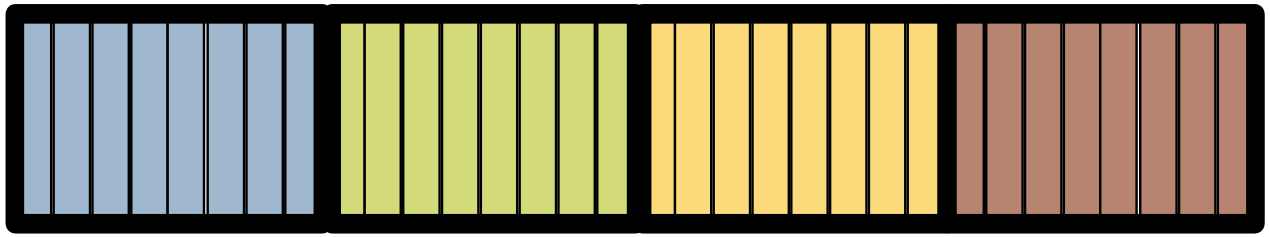


- Which one is better for a program on CPU?
  - If your operations require accessing multiple different fields of a structure at once, AoS might be more efficient because they will likely be in the same cache line.

Array of Structures vs. Structure of Arrays

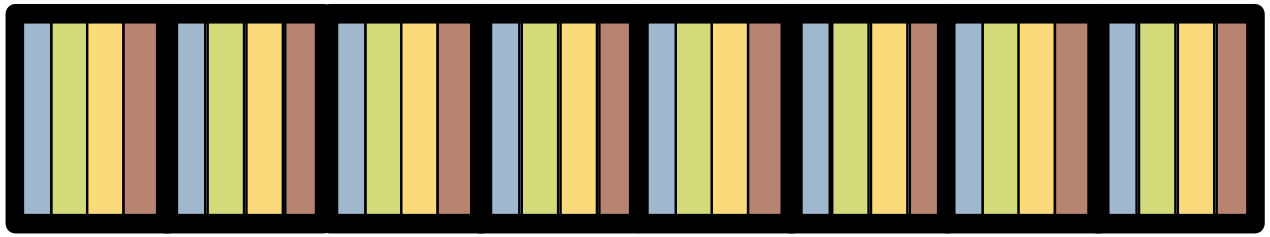
Structure of Arrays (SoA)

```
struct foo{
 float a[8];
 float b[8];
 float c[8];
 int d[8];
} A;
```



Array of Structures (AoS)

```
struct foo{
 float a;
 float b;
 float c;
 int d;
} A[8];
```

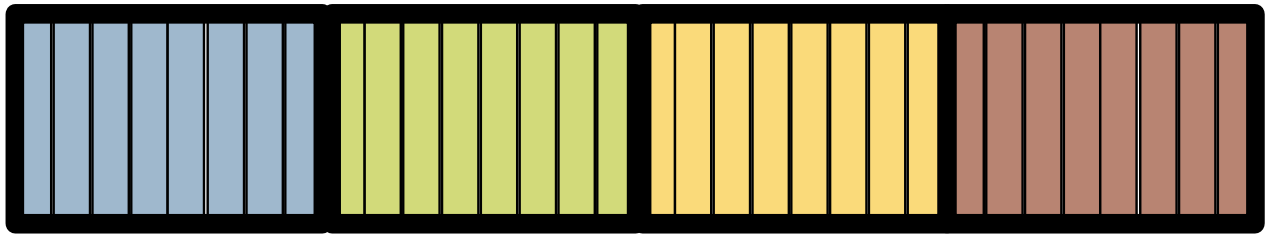


- Which one is better for a program on GPU?

Array of Structures vs. Structure of Arrays

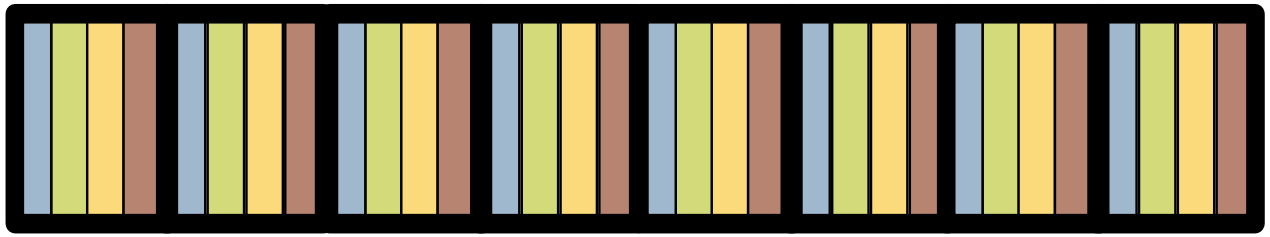
Structure of Arrays (SoA)

```
struct foo{
  float a[8];
  float b[8];
  float c[8];
  int d[8];
} A;
```



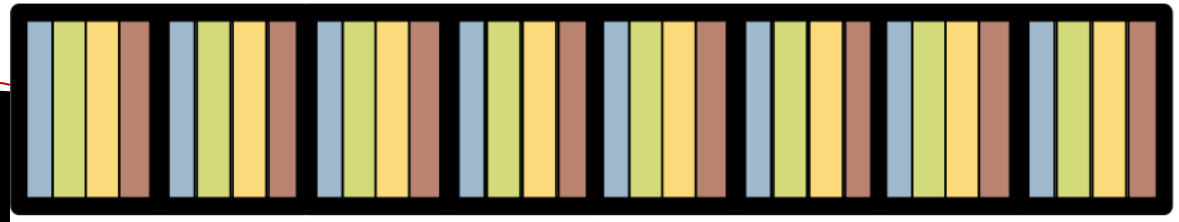
Array of Structures (AoS)

```
struct foo{
  float a;
  float b;
  float c;
  int d;
} A[8];
```

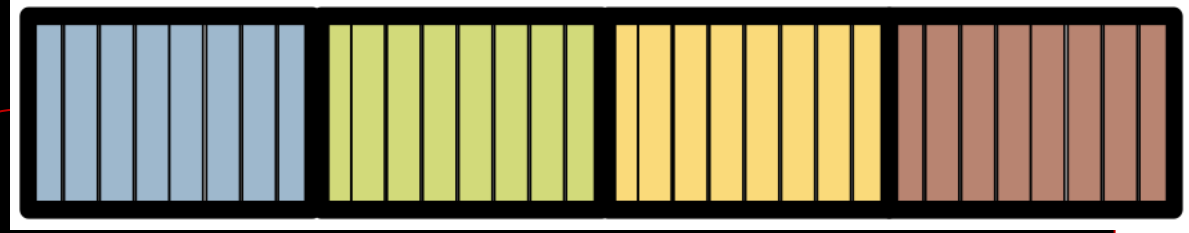


- Which one is better for a program on GPU?
  - SoA allows threads in a warp access contiguous memory addresses.

AoS vs SoA



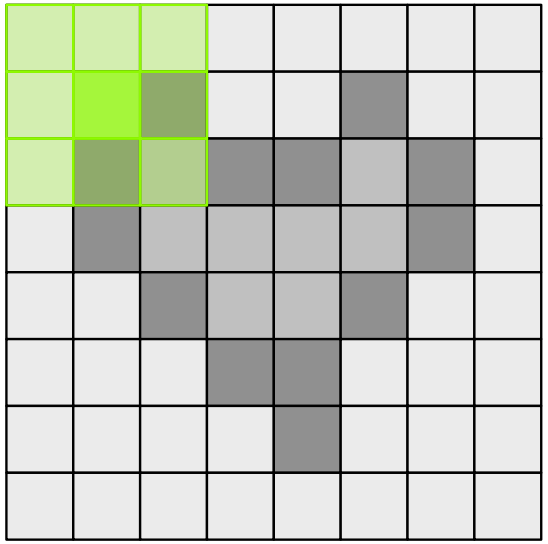
```cpp
struct Particle {
  float x, y, z; // Position
  float vx, vy, vz; // Velocity
};

__global__ void updateParticles(Particle *particles, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) {
    particles[i].x += particles[i].vx; // Uncoalesced
    particles[i].y += particles[i].vy;
    particles[i].z += particles[i].vz;
  }
}
```



```cpp
struct ParticleSoA {
  float *x, *y, *z;
  float *vx, *vy, *vz;
};

__global__ void updateParticles(float *x, float *y, float *z,
                                float *vx, float *vy, float *vz, int N) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) {
    x[i] += vx[i]; // Coalesced memory access
    y[i] += vy[i];
    z[i] += vz[i];
  }
}
```

Data Reuse

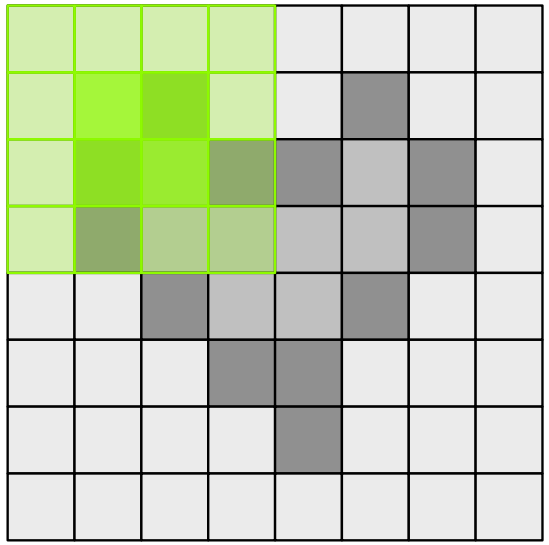
- Same memory locations accessed by neighboring threads



```
for (int i = 0; i < 3; i++) {
  for (int j = 0; j < 3; j++) {
    sum += gauss[i][j] * image[(i+row-1)*width + (j+col-1)];
  }
}
```

Data Reuse: Tiling

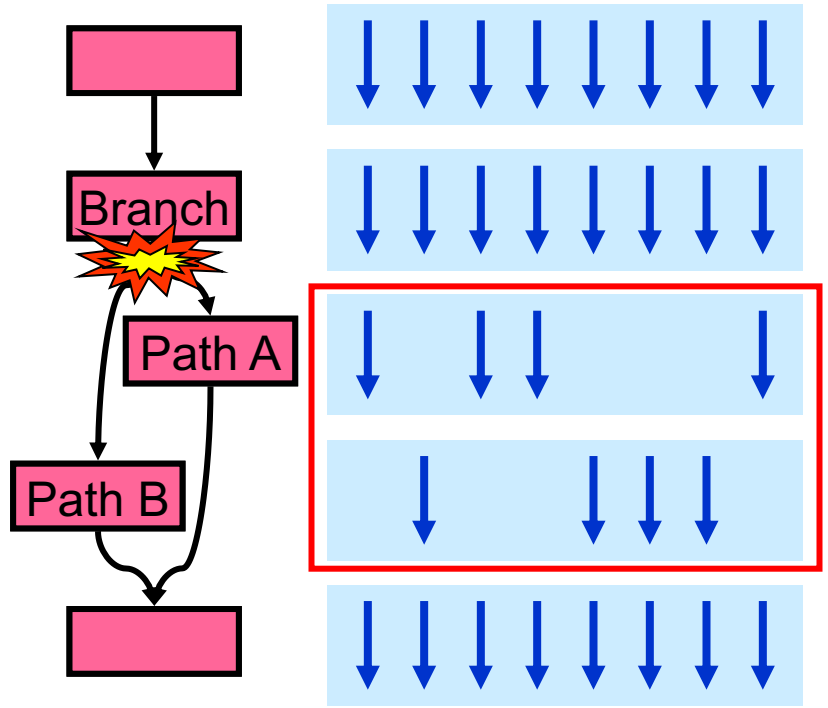
- To take advantage of data reuse, we divide the input into tiles that can be loaded into shared memory



```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
…
// Load tile into shared memory
__syncthreads();
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
  }
}
```

Branching in Threads

- Threads in a warp start together at the same program address, but they are free to branch and execute independently.
- Branch divergence occurs when threads inside warps branch to different execution paths

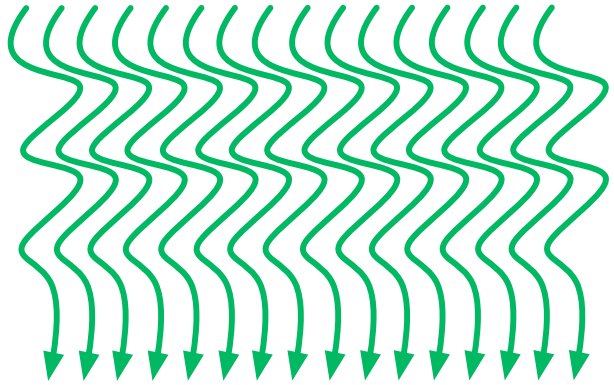


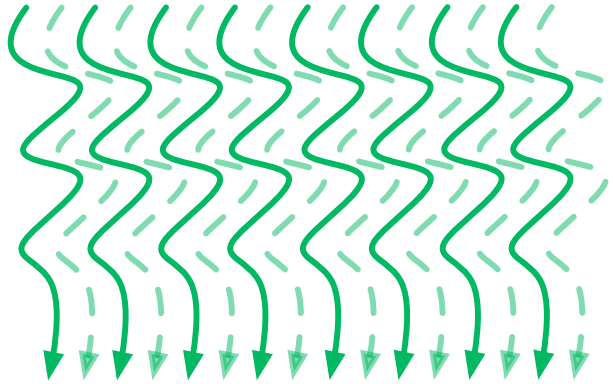
If threads of a warp **diverge** via a data dependent conditional branch, the warp **serially executes** each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads **reconverge** to the original execution path.

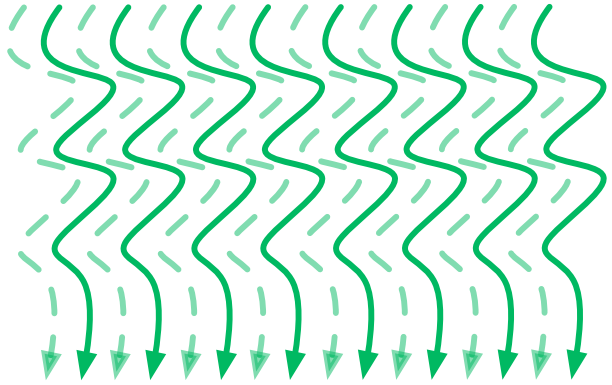
SIMD Utilization

- Divergence in a warp

```
Compute(threadIdx.x);
if (threadIdx.x % 2 == 0) {
  Do_this(threadIdx.x);
}
else {
  Do_that(threadIdx.x);
}
```



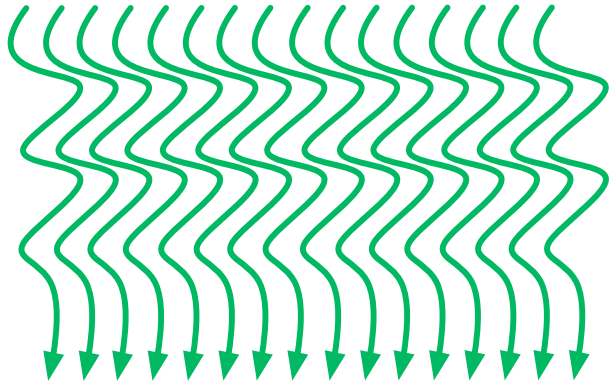


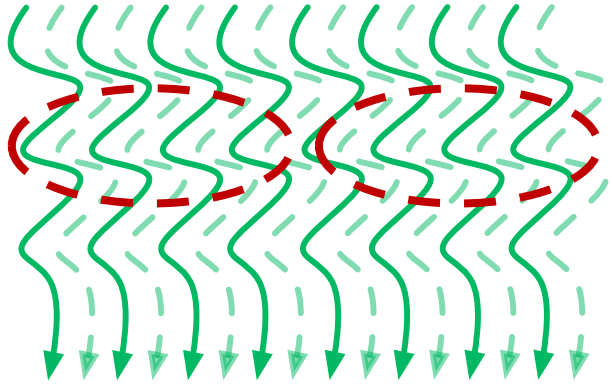


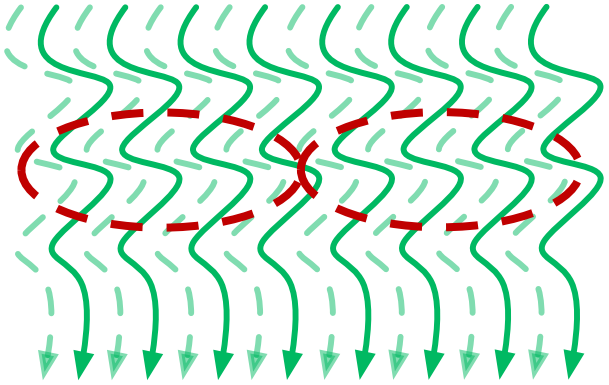
SIMD Utilization

- Divergence in a warp

```
Compute(threadIdx.x);
if (threadIdx.x % 2 == 0) {
  Do_this(threadIdx.x);
}
else {
  Do_that(threadIdx.x);
}
```





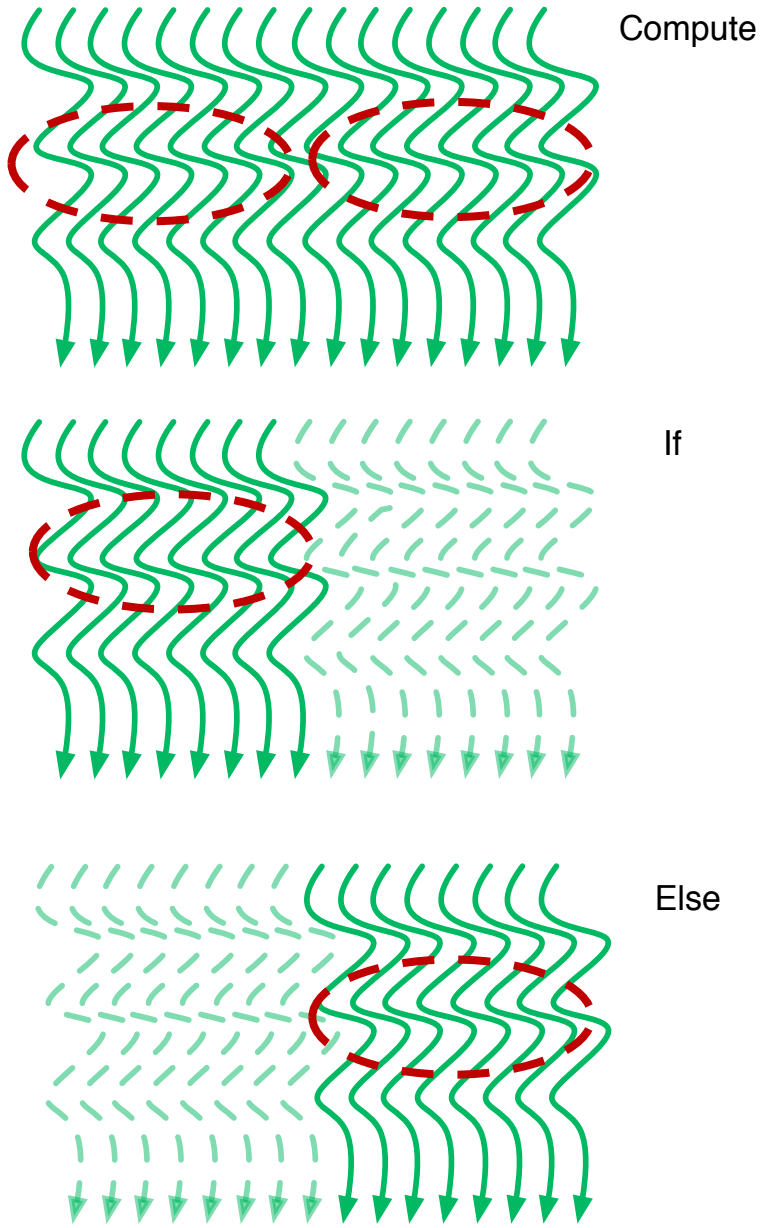


- Half of the threads will diverge: Branching on whether a thread's index is even or odd.
- Threads within a warp must serially execute both paths of the branch, reducing parallelism.

SIMD Utilization

- Divergence-free execution

```
Compute(threadIdx.x);
if (threadIdx.x < 32) {
  Do_this(threadIdx.x * 2);
}
else {
  Do_that((threadIdx.x%32)*2+1);
}
```

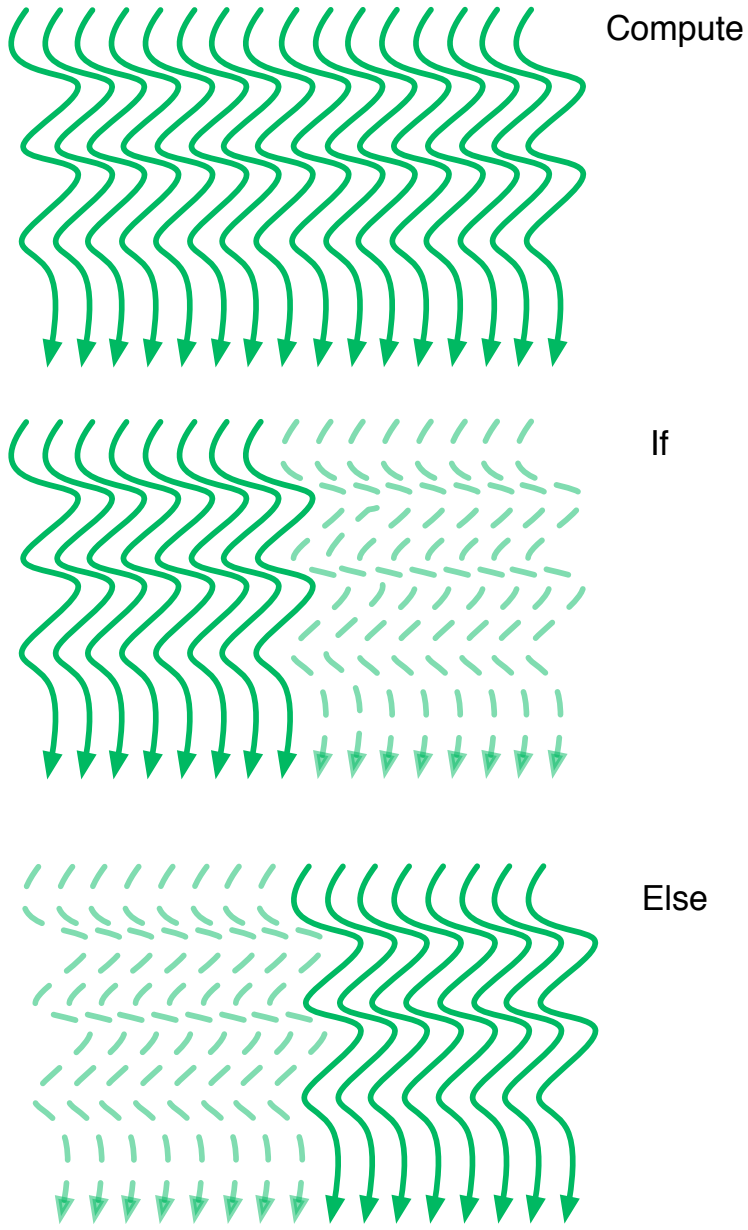


SIMD Utilization

- Divergence-free execution

```
Compute(threadIdx.x);
if (threadIdx.x < 32) {
  Do_this(threadIdx.x * 2);
}
else {
  Do_that((threadIdx.x%32)*2+1);
}
```

All threads within a warp execute the same branch, either `Do_this` or `Do_that`, based on their thread index, significantly reduces **branch divergence**.



cuBLAS

cuBLAS

- A GPU-accelerated library from NVIDIA.
  - API extensions for providing
    - Supports **standard BLAS** APIs
    - **GEMM APIs** with support for **fusions** that are highly optimized for NVIDIA GPUs.

Fusion:
In **traditional non-fused execution**, multiple separate kernel launches are required:
1.Compute **GEMM (Matrix-Matrix Multiplication)**.
2.Store results in **global memory**.
3.Apply an activation function (e.g., ReLU, Sigmoid).
4.Store the final results again.

```
C = A * B // GEMM Operation
C = C + Bias // Bias Addition
C = ReLU(C) // Apply Activation Function
```

Non-Fusion
•Three separate operations
•Three memory accesses → Slower

```
C = ReLU(A * B + Bias)
```

Fusion
•Single kernel execution
•One memory access
•Much faster on NVIDIA GPUs using Tensor Cores

cuBLAS

- A GPU-accelerated library from NVIDIA.
  - API extensions for providing
    - Supports **standard BLAS** APIs
    - **GEMM APIs** with support for **fusions** that are highly optimized for NVIDIA GPUs.

Fusion:
In **traditional non-fused execution**, multiple separate kernel launches are required:
1.Compute **GEMM (Matrix-Matrix Multiplication)**.
2.Store results in **global memory**.
3.Apply an activation function (e.g., ReLU, Sigmoid).
4.Store the final results again.

```
C = A * B // GEMM Operation
C = C + Bias // Bias Addition
C = ReLU(C) // Apply Activation Function
```

cuBLAS

- A GPU-accelerated library from NVIDIA.
  - API extensions for providing
    - Supports **standard BLAS** APIs
    - **GEMM APIs** with support for **fusions** that are highly optimized for NVIDIA GPUs.

Fusion:
In **traditional non-fused execution**, multiple separate kernel launches are required:
1.Compute **GEMM (Matrix-Matrix Multiplication)**.
2.Store results in **global memory**.
3.Apply an activation function (e.g., ReLU, Sigmoid).
4.Store the final results again.

```
// Using cuBLASLt to perform fused GEMM with Bias and ReLU
cublasLtMatmul(handle,
  gemmDesc,
  &alpha, A, descA,
  B, descB,
  &beta, bias, descBias,
  C, descC, workspace, workspaceSize, stream);
```

cuBLAS

- A GPU-accelerated library from NVIDIA.
  - API extensions for providing
    - Supports **standard BLAS** APIs
    - **GEMM APIs** with support for fusions that are highly optimized for NVIDIA GPUs.
  - Extensions for **batched** operations, execution across **multiple GPUs**, and mixed and **low precision** execution with additional tuning for the best performance.

cuBLAS

- A GPU-accelerated library from NVIDIA.
  - API extensions for providing
    - Supports **standard BLAS** APIs
    - **GEMM APIs** with support for fusions that are highly optimized for NVIDIA GPUs.
  - Extensions for **batched** operations, execution across **multiple GPUs**, and mixed and **low precision** execution with additional tuning for the best performance.
  - Optimized for different NVIDIA GPU architectures.

cuBLAS

- A GPU-accelerated library from NVIDIA.
  - API extensions for providing
    - Supports **standard BLAS** APIs
    - **GEMM APIs** with support for fusions that are highly optimized for NVIDIA GPUs.
  - Extensions for **batched** operations, execution across **multiple GPUs**, and mixed and **low precision** execution with additional tuning for the best performance.
  - Optimized for different NVIDIA GPU architectures.
  - Alternative Libraries:
    - Other libraries like Intel MKL or OpenBLAS are popular for CPU-based systems.

cuBLAS

- A GPU-accelerated library from NVIDIA.
  - API extensions for providing
    - Supports **standard BLAS** APIs
    - **GEMM APIs** with support for fusions that are highly optimized for NVIDIA GPUs.
  - Extensions for **batched** operations, execution across **multiple GPUs**, and mixed and **low precision** execution with additional tuning for the best performance.
  - Optimized for different NVIDIA GPU architectures.
  - Alternative Libraries:
    - Other libraries like Intel MKL or OpenBLAS are popular for CPU-based systems.

Writing custom CUDA kernels is an option but requires in-depth knowledge of GPU programming and is generally less efficient than using cuBLAS.

Example code using cuBLAS

```
int main() {
  int N = 256;
  float alpha = 1.0f, beta = 0.0f;
  float *d_A, *d_B, *d_C; // Device pointers

  // Omitted: Memory allocation and initialization for d_A, d_B, d_C

  cublasHandle_t handle;
  cublasCreate(&handle); // Create cuBLAS handle

  // Matrix multiplication: C = A*B
  cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
       N, N, N, &alpha, d_A, N, d_B, N,
       &beta, d_C, N);

  cublasDestroy(handle); // Destroy cuBLAS handle

  // Omitted: Memory deallocation
  return 0;
}
```

Example code using cuBLAS

```
cublasStatus_t cublasSgemm(cublasHandle_t handle,
cublasOperation_t transa, cublasOperation_t transb,
int m, int n, int k,
const float *alpha,
const float *A, int lda,
const float *B, int ldb,
const float *beta,
float *C, int ldc)
```

$$C = \alpha \cdot op(A) \cdot op(B) + \beta \cdot C$$

In this formula:

- A is the matrix located at pointer **`d_A`**.
- B is the matrix located at pointer **`d_B`**.
- C initially contains the pre-multiplication matrix and is overwritten by the result. It is located at pointer **`d_C`**.
- α and β are scalars passed by reference as **`&alpha`** and **`&beta`**, respectively.

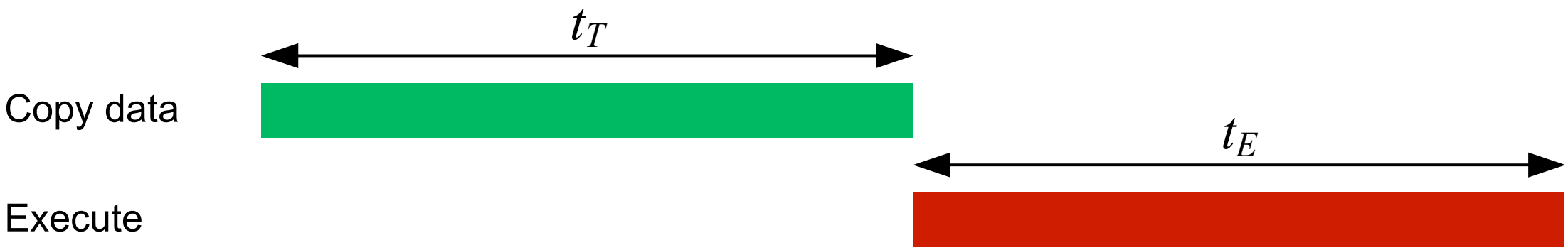
- m: Number of rows in matrices A and C.
- n: Number of columns in matrices B and C.
- k: Number of columns in matrix A and rows in matrix B.

$$\text{op}(A) = \begin{cases} A & \text{if transa} == \text{CUBLAS\_OP\_N} \\ A^T & \text{if transa} == \text{CUBLAS\_OP\_T} \\ A^H & \text{if transa} == \text{CUBLAS\_OP\_C} \end{cases}$$

Data Transfers between CPU and GPU

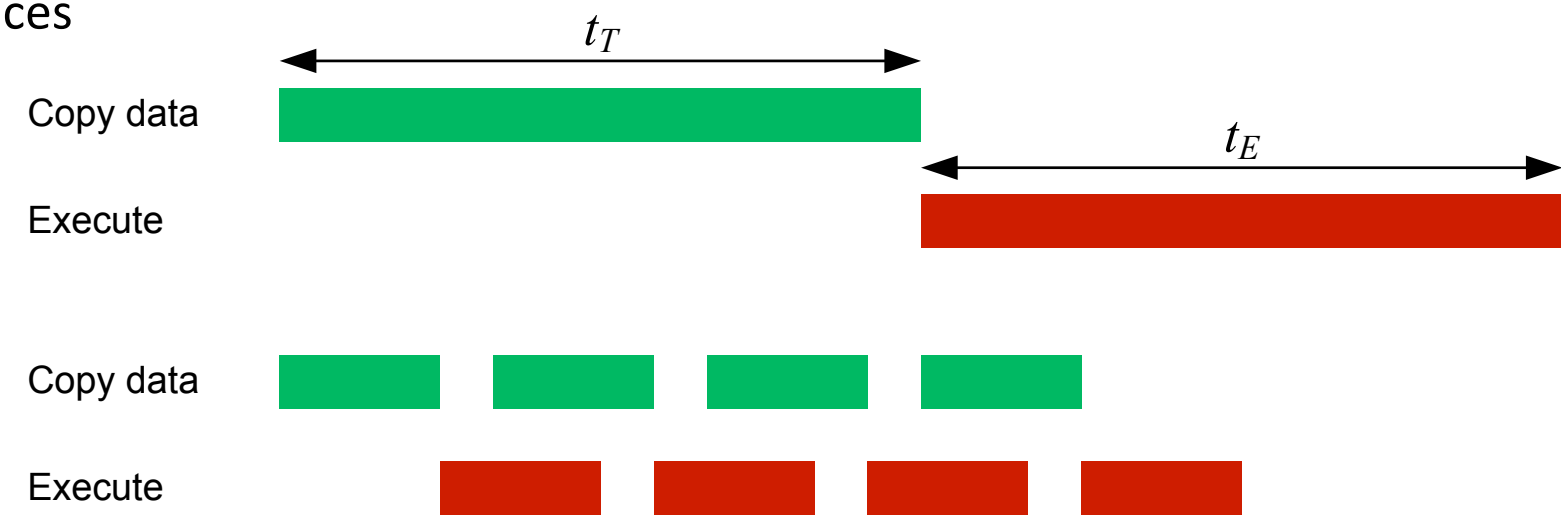
Data Transfers

- Synchronous and asynchronous transfers
- Streams (Command queues)
  - Sequence of operations that are performed in order
    - CPU-GPU data transfer
    - Kernel execution
      - D input data instances, B blocks
    - GPU-CPU data transfer
  - Default stream



Asynchronous Transfers

- Computation divided into nStreams
  - CUDA streams are sequences of operations that execute in issue-order on the GPU.
  - D input data instances, B blocks
  - Number of streams: nStreams, each stream
    - D/nStreams data instances
    - B/nStreams blocks



- Estimates

$$t_E + \frac{t_T}{nStreams}$$

$t_E >= t_T$ (dominant kernel)

$$t_T + \frac{t_E}{nStreams}$$

$t_T > t_E$ (dominant transfers)

Example code using cuBLAS

```
cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);
kernel<<<grid, block>>>(d_data);
cudaMemcpy(h_result, d_result, size, cudaMemcpyDeviceToHost)
```

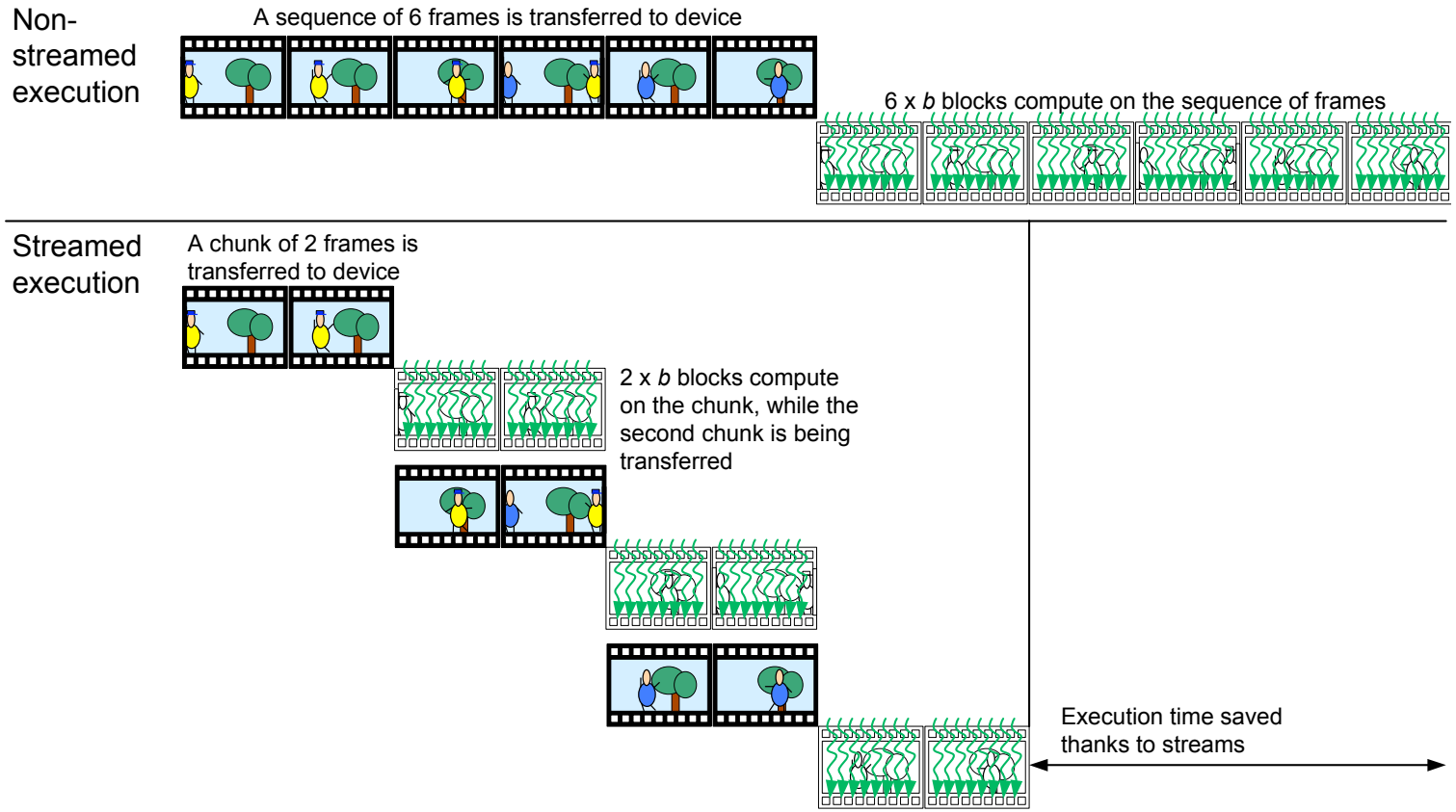
Without Stream

```
for (int i = 0; i < nStreams; i++) {
  cudaMemcpyAsync(d_data[i], h_data[i], size, cudaMemcpyHostToDevice, stream[i]);
  kernel<<<grid, block, 0, stream[i]>>>(d_data[i]);
  cudaMemcpyAsync(h_result[i], d_result[i], size, cudaMemcpyDeviceToHost, stream[i]);
}
```

With Stream

Overlap of Communication and Computation

- Applications with independent computation on different data instances can benefit from asynchronous transfers
- For instance, video processing



Gomez-Luna+, "Performance models for asynchronous data transfers on consumer Graphics Processing Units," JPDC, 2012.

Collaborative Computing

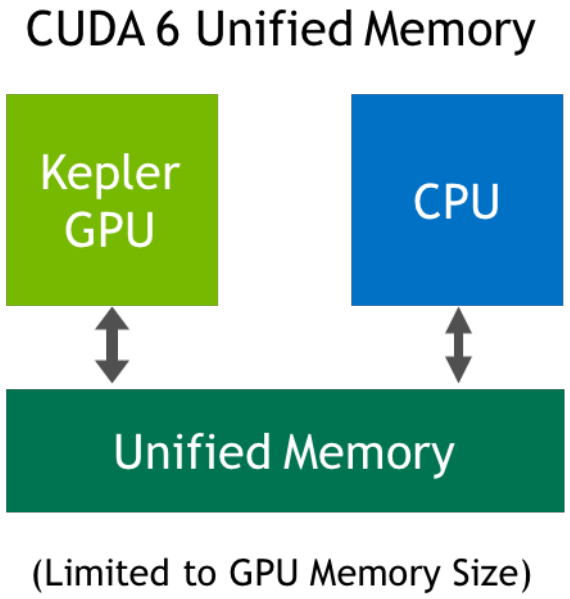
Review

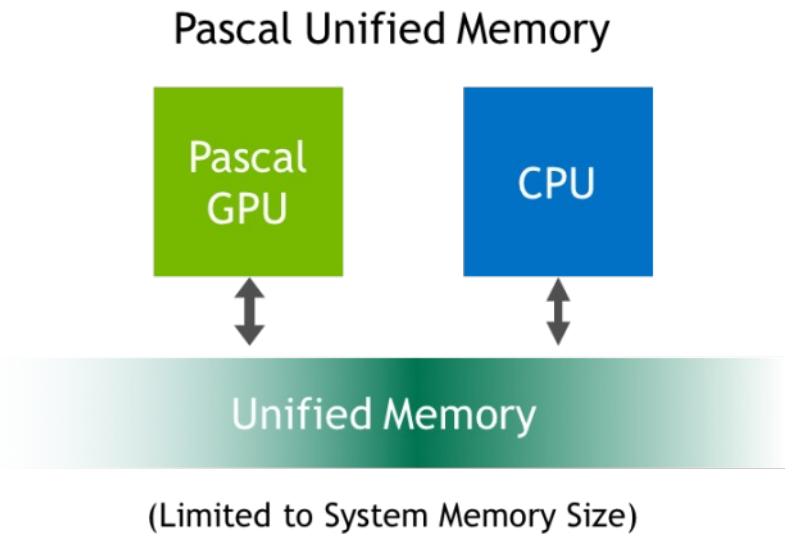
- Device allocation, CPU-GPU transfer, and GPU-CPU transfer
  - `cudaMalloc();`
  - `cudaMemcpy();`

```
// Allocate input
malloc(input, ...);
cudaMalloc(d_input, ...);
cudaMemcpy(d_input, input, ..., HostToDevice); // Copy to device memory

// Allocate output
malloc(output, ...);
cudaMalloc(d_output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_output, d_input, ...);

// Synchronize
cudaDeviceSynchronize();

// Copy output to host memory
cudaMemcpy(output, d_output, ..., DeviceToHost);
```

Unified Memory

- Unified Virtual Address
- Since CUDA 6.0: Unified memory
- Since CUDA 8.0 + Pascal: GPU page faults
  - if the GPU accesses memory that has not been transferred to its local memory, a page fault occurs, and the necessary data is automatically migrated to the GPU.





Non-Unified Memory Example

```cpp
int main(void) {
  int N = 1 << 20; // 1M elements
  float *x, *y; // Host pointers
  float *d_x, *d_y; // Device pointers

  // Step 1: Allocate memory on host
  x = (float*)malloc(N * sizeof(float));
  y = (float*)malloc(N * sizeof(float));

  // Step 2: Allocate memory on device
  cudaMalloc((void**)&d_x, N * sizeof(float));
  cudaMalloc((void**)&d_y, N * sizeof(float));

  // Step 3: Initialize arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Step 4: Copy data from Host to Device
  cudaMemcpy(d_x, x, N * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N * sizeof(float), cudaMemcpyHostToDevice);

  // Step 5: Launch kernel on 1M elements on the GPU
  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, d_x, d_y);

  // Step 6: Wait for GPU to finish before accessing results on host
  cudaDeviceSynchronize();

  // Step 7: Copy results back from Device to Host
  cudaMemcpy(y, d_y, N * sizeof(float), cudaMemcpyDeviceToHost);

  // Step 8: Free device and host memory
  cudaFree(d_x);  cudaFree(d_y);
  free(x); free(y);

  return 0;
}
```

```cpp
// CUDA kernel to add elements of two arrays
__global__ void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

Non-Unified Memory Example

- Easier programming with Unified Memory
  - `cudaMallocManaged();`

```
// CUDA kernel to add elements of two arrays
__global__ void add(int n, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  for (int i = index; i < n; i += stride)
    y[i] = x[i] + y[i];
}
```

```
int main(void)
{
  int N = 1<<20;
  float *x, *y;
  // Allocate Unified Memory -- accessible from CPU or GPU
  cudaMallocManaged(&x, N*sizeof(float));
  cudaMallocManaged(&y, N*sizeof(float));
  // initialize x and y arrays on the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }
  // Launch kernel on 1M elements on the GPU
  int blockSize = 256;
  int numBlocks = (N + blockSize - 1) / blockSize;
  add<<<numBlocks, blockSize>>>(N, x, y);
  // Wait for GPU to finish before accessing on host
  cudaDeviceSynchronize();
  // Free memory
  cudaFree(x);
  cudaFree(y);
  return 0;
}
```