Introduction to GPU Programming in CUDA

Polly Ren

COS / ECE 375: Computer Architecture and Organization
Fall 2025
Princeton University

December 2, 2025

Overview

The Canonical Hello World

PRINCETON UNIVERSITY

CPU Version:

```c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

```
$ gcc hello.c -o hello
$ ./hello
Hello, world!
```

What do we have to change to make use of the GPU?

The Canonical Hello World

PRINCETON UNIVERSITY

CPU Version:

```c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

```
$ gcc hello.c -o hello
$ ./hello
Hello, world!
```

GPU Version:

```c
#include <stdio.h>

__global__ void helloKernel() {
    printf("Hello, world!\n");
}

int main() {
    helloKernel<<<1, 1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

```
$ nvcc hello.cu -o hello
$ ./hello
Hello, world!
```

(Full code in `hello/`)

Hello World on GPU

Let's examine where the GPU code differs from CPU code:

```
1  #include <stdio.h>
2
3  __global__ void helloKernel() {
4      printf("Hello, world!\n");
5  }
6
7  int main() {
8      helloKernel<<<1, 1>>>();
9      cudaDeviceSynchronize();
10     return 0;
11 }
```

```
$ nvcc hello.cu -o hello
$ ./hello
Hello, world!
```

Hello World on GPU: `__global__`

Let's examine where the GPU code differs from CPU code:

```
1  #include <stdio.h>
2
3  __global__ void helloKernel() {
4      printf("Hello, world!\n");
5  }
6
7  int main() {
8      helloKernel<<<1, 1>>>();
9      cudaDeviceSynchronize();
10     return 0;
11 }
```

```
$ nvcc hello.cu -o hello
$ ./hello
Hello, world!
```

- The CPU is called the *host*
- The GPU is called the *device*
- The *kernel* is the function that runs on the device
  - (Must have a `void` return type)
- *Heterogeneous computing* refers to systems that use a combination of different types of processors, e.g., CPUs, GPUs, FPGAs
- The `__global__` keyword is used to modify a function that:
  - Runs on the device
  - Is called from host code
  - Denotes the GPU kernel

Hello World on GPU: `__global__`

Let's examine where the GPU code differs from CPU code:

```
1  #include <stdio.h>
2
3  __global__ void helloKernel() {
4      printf("Hello, world!\n");
5  }
6
7  int main() {
8      helloKernel<<<1, 1>>>();
9      cudaDeviceSynchronize();
10     return 0;
11 }
```

```
$ nvcc hello.cu -o hello
$ ./hello
Hello, world!
```

- The CPU is called the *host*
- The GPU is called the *device*
- The *kernel* is the function that runs on the device
  - (Must have a `void` return type)
- *Heterogeneous computing* refers to systems that use a combination of different types of processors, e.g., CPUs, GPUs, FPGAs

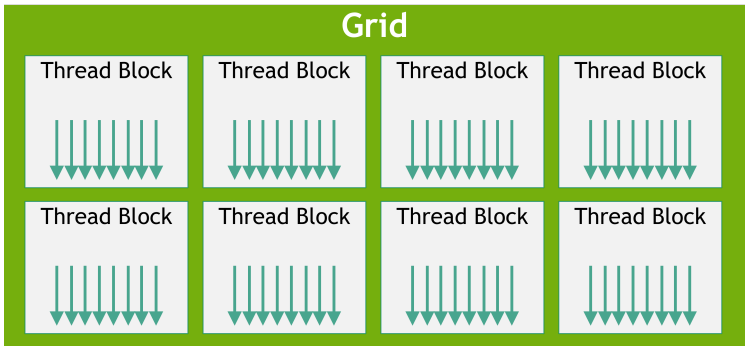
Qualifier	Runs on	Called from
`__host__` (default)	Host	Host
`__device__`	Device	Device
`__global__`	Device	Host

Hello World on GPU: Kernel Launch

Let's examine where the GPU code differs from CPU code:

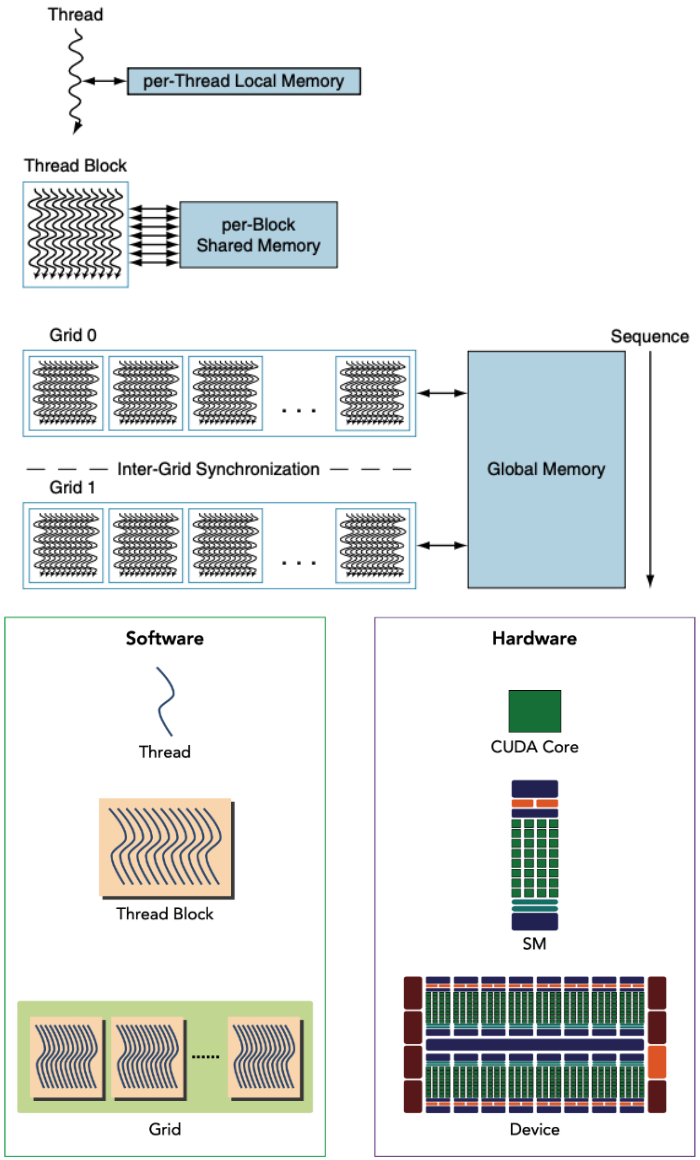
```
1  #include <stdio.h>
2
3  __global__ void helloKernel() {
4      printf("Hello, world!\n");
5  }
6
7  int main() {
8      helloKernel<<<1, 1>>>();
9      cudaDeviceSynchronize();
10     return 0;
11 }
```

```
$ nvcc hello.cu -o hello
$ ./hello
Hello, world!
```



- The triple angle brackets <<<N, M>>> denote a kernel launch
  - M = number of threads per block
  - N = number of blocks in the grid
  - Total threads launched = N × M
- Similar to a C function call, but:
  - The kernel runs on the GPU
  - Many parallel threads are launched, each running the same code

Single Instruction, Multiple Thread (SIMT)



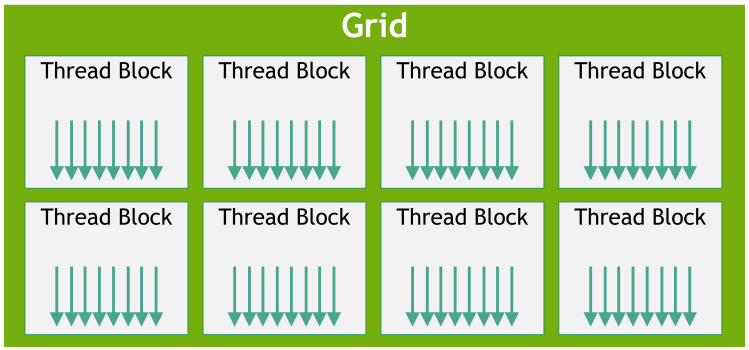
- CUDA uses a SIMT architecture
- All threads in a block are assigned to an SM simultaneously
  - A block cannot be assigned to an SM until it secures enough resources for all its threads to execute
  - Have to share memory resources of SM $\rightarrow$ limit on the number of threads per block (1024 on modern GPUs)
- Threads within a block are executed in groups of 32 called *warps*
  - Threads in a warp execute the same instruction at the same time
  - Each thread has its own program counter, register state, and can have an independent execution path
  - Instruction-level parallelism within a thread

Hello World on GPU: Kernel Launch ($\times$ 32)

Let's examine where the GPU code differs from CPU code:

```
1  #include <stdio.h>
2
3  __global__ void helloKernel() {
4      printf("Hello, world!\n");
5  }
6
7  int main() {
8      helloKernel<<<1, 32>>>();
9      cudaDeviceSynchronize();
10     return 0;
11 }
```

```
$ nvcc hello32.cu -o hello32
$ ./hello32
Hello, world!
Hello, world!
... (29 more times)
Hello, world!
```



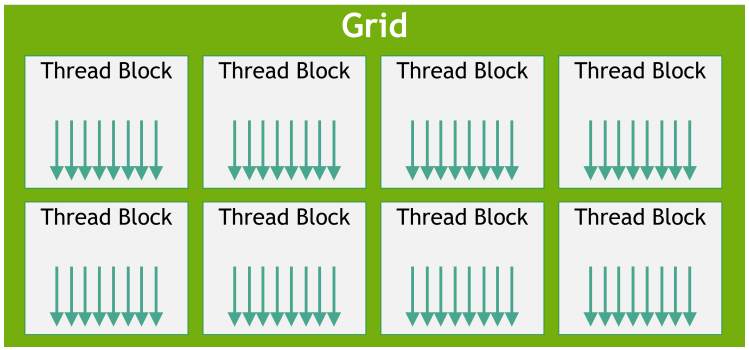
- The triple angle brackets <<<N, M>>> denote a kernel launch
  - M = number of threads per block
  - N = number of blocks in the grid
  - Total threads launched = N $\times$ M
- Similar to a C function call, but:
  - The kernel runs on the GPU
  - Many parallel threads are launched, each running the same code

Hello World on GPU: Kernel Launch ($\times$ 128)

Let's examine where the GPU code differs from CPU code:

```
1  #include <stdio.h>
2
3  __global__ void helloKernel() {
4      printf("Hello, world!\n");
5  }
6
7  int main() {
8      helloKernel<<<4, 32>>>();
9      cudaDeviceSynchronize();
10     return 0;
11 }
```

```
$ nvcc hello128.cu -o hello128
$ ./hello128
Hello, world!
Hello, world!
... (125 more times)
Hello, world!
```



- The triple angle brackets <<<N, M>>> denote a kernel launch
  - M = number of threads per block
  - N = number of blocks in the grid
  - Total threads launched = N $\times$ M
- Similar to a C function call, but:
  - The kernel runs on the GPU
  - Many parallel threads are launched, each running the same code

Hello World on GPU: `cudaDeviceSynchronize`

Let's examine where the GPU code differs from CPU code:

```
1  #include <stdio.h>
2
3  __global__ void helloKernel() {
4      printf("Hello, world!\n");
5  }
6
7  int main() {
8      helloKernel<<<1, 1>>>();
9      cudaDeviceSynchronize();
10     return 0;
11 }
```

```
$ nvcc hello.cu -o hello
$ ./hello
Hello, world!
```

- Unlike a C function call, kernel launches are *asynchronous*
- Control returns to the CPU immediately after the CUDA kernel is invoked
- `cudaDeviceSynchronize` blocks the CPU until all preceding CUDA calls have completed
- What could go wrong if we do not add synchronisation here?
  - Yikes: race conditions and nondeterministic behaviour!

Hello World on GPU: nvcc

Let's examine where the GPU code differs from CPU code:

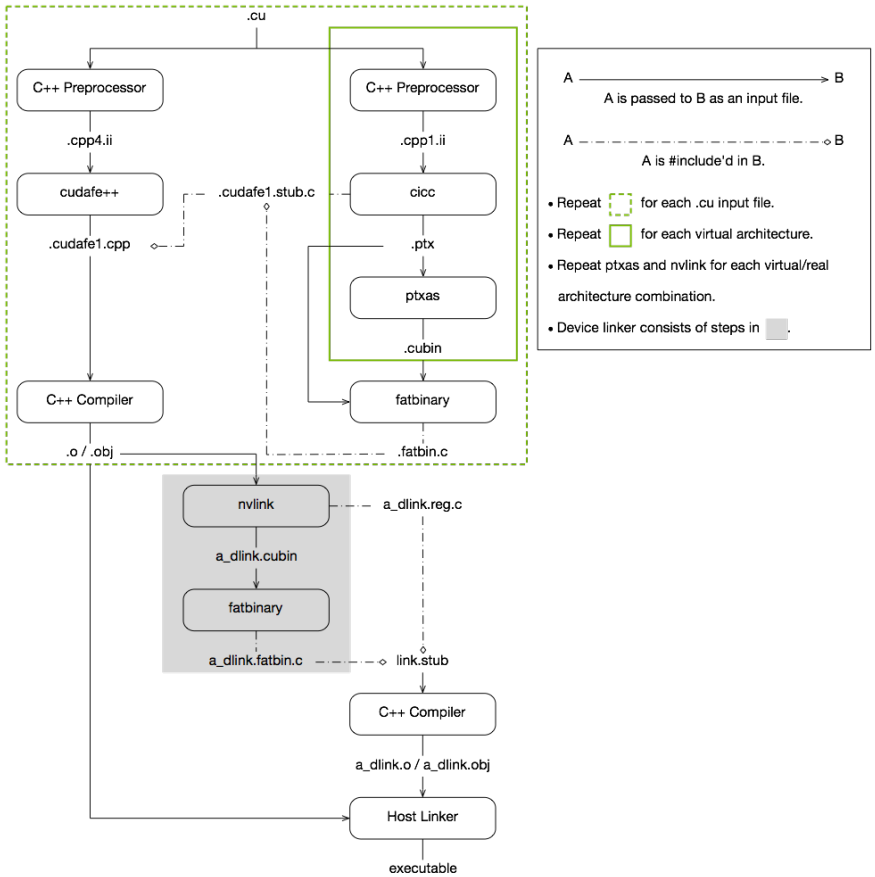
```
1  #include <stdio.h>
2
3  __global__ void helloKernel() {
4      printf("Hello, world!\n");
5  }
6
7  int main() {
8      helloKernel<<<1, 1>>>();
9      cudaDeviceSynchronize();
10     return 0;
11 }
```

```
$ nvcc hello.cu -o hello
$ ./hello
Hello, world!
```

- nvcc is the CUDA *compiler driver*
- It separates the .cu source file into:
  - host code → compiled by the system's C/C++ compiler
  - device code → compiled by NVIDIA's device compiler into cubin and PTX intermediate code
- The compiled device code is packaged into a *fat binary* and embedded in the host object file
- The host system linker combines these pieces into a final executable containing the GPU kernels

Hello World on GPU: `nvcc`

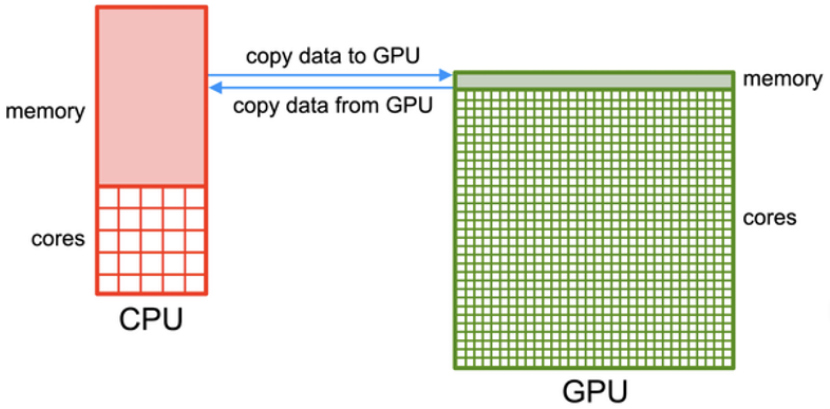
Let's examine where the GPU code differs from CPU code:



- `nvcc` is the CUDA *compiler driver*
- It separates the `.cu` source file into:
  - host code → compiled by the system's C/C++ compiler
  - device code → compiled by NVIDIA's device compiler into cubin and PTX intermediate code
- The compiled device code is packaged into a *fat binary* and embedded in the host object file
- The host system linker combines these pieces into a final executable containing the GPU kernels

Overview

General Programming Workflow

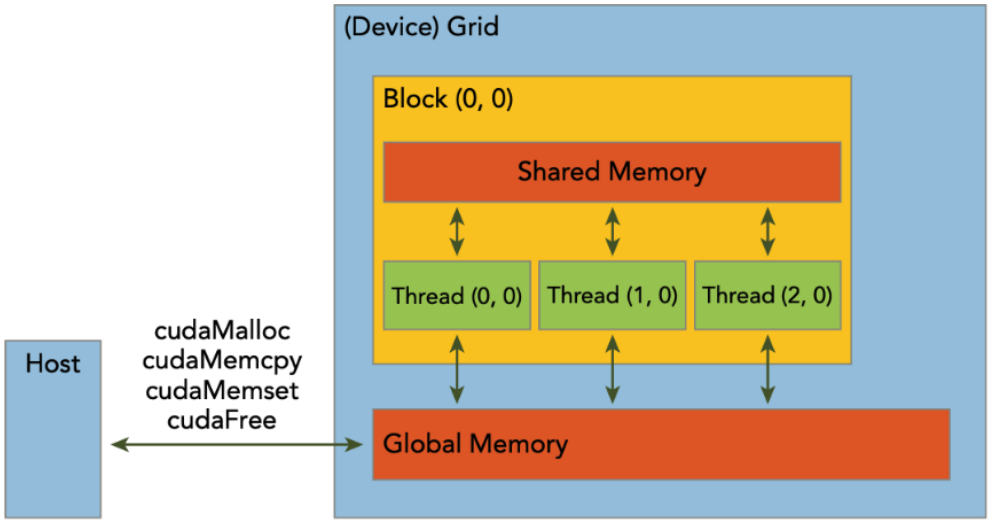


The typical programming flow of a CUDA program follows this pattern:

- Allocate memory on the device
- Copy data from host to device memory
- Invoke kernels to operate on the data stored in GPU memory
- Copy results back from device to host memory

Communication between host and device is slow and can easily become a performance bottleneck $\rightarrow$ limit copying when possible

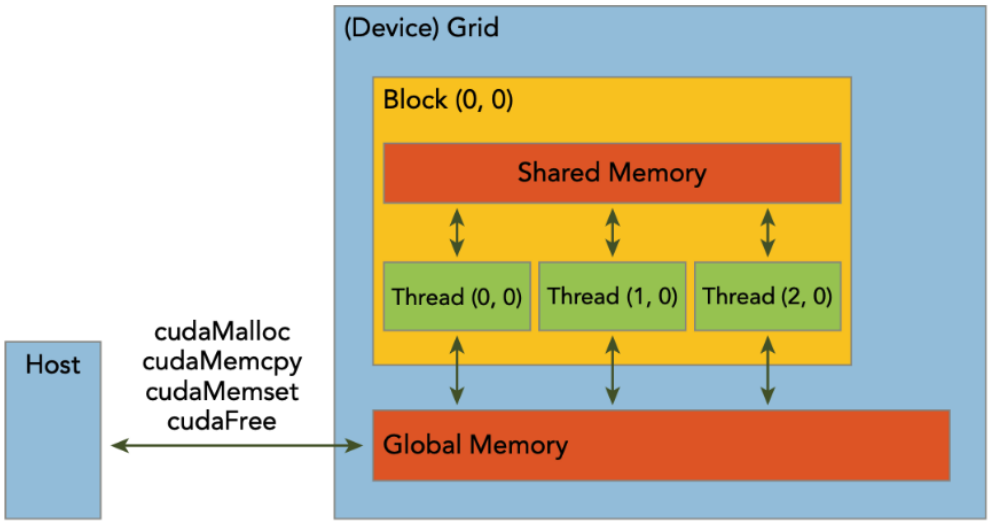
Memory Management



Standard C Function	CUDA C Function
`malloc`	`cudaMalloc`
`memcpy`	`cudaMemcpy`
`memset`	`cudaMemset`
`free`	`cudaFree`

- Host and device memory are separate entities
- Device pointers point to GPU memory; cannot be dereferenced in host code
- Host pointers point to CPU memory; cannot be dereferenced in device code
- CUDA provides an API to allocate and deallocate device memory, and transfer data between the host memory and device memory

Memory Management API



Standard C Function	CUDA C Function
`malloc`	`cudaMalloc`
`memcpy`	`cudaMemcpy`
`memset`	`cudaMemset`
`free`	`cudaFree`

- Allocate memory on device

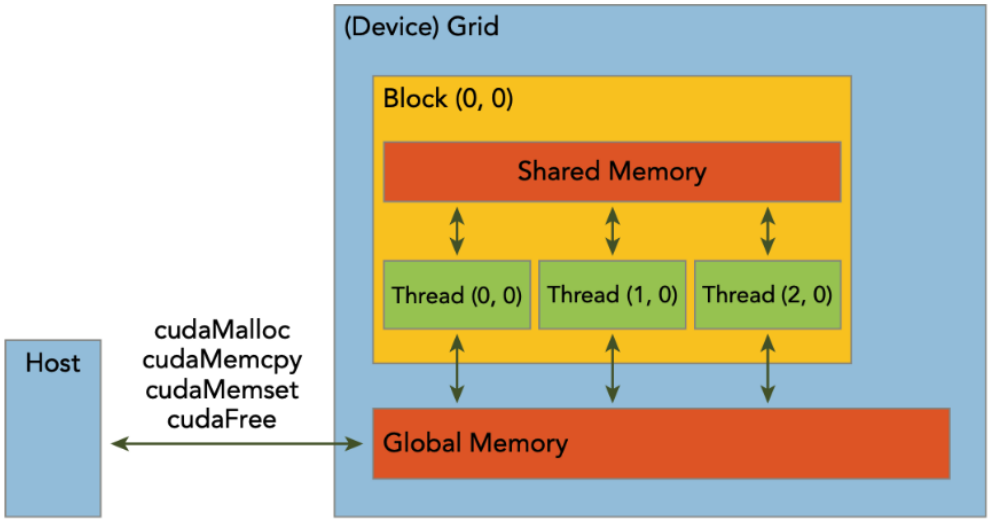
```
cudaError_t cudaMalloc(void**
    devPtr, size_t size)
```

- Transfer data between host and device

```
cudaError_t cudaMemcpy(void*
    dst, const void* src,
    size_t count,
    cudaMemcpyKind kind)
```

- `cudaMemcpyKind` is an enum that specifies the direction of data transfer
  - `cudaMemcpyHostToHost`
  - `cudaMemcpyHostToDevice`
  - `cudaMemcpyDeviceToHost`
  - `cudaMemcpyDeviceToDevice`
  - `cudaMemcpyDefault` (inferred)

Memory Management API



Standard C Function	CUDA C Function
`malloc`	`cudaMalloc`
`memcpy`	`cudaMemcpy`
`memset`	`cudaMemset`
`free`	`cudaFree`

- Free memory on device

```
1 cudaError_t cudaFree(void*
      devPtr)
```

- Calls return an error code (`cudaError_t`) that signify error in API call or in an earlier asynchronous operation
  - Should check if return value is `cudaSuccess`
  - If not, can stringify the error using `cudaGetErrorString`
- Unlike kernel launches, these memory management functions are *synchronous*, i.e., blocks until the operation completes

Vector Addition

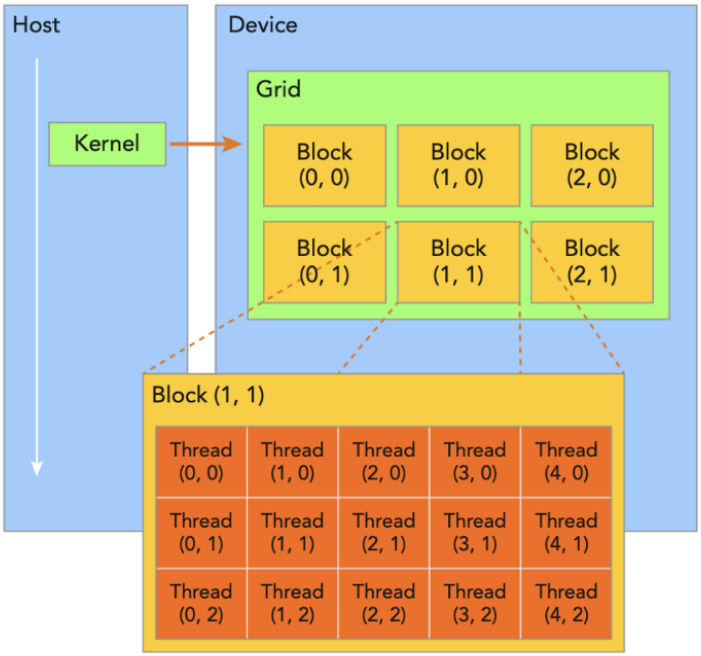
```
1  #define N 16384 * 16384
2
3  void add(float *c, float *a, float *b) {
4      *c = *a + *b;
5  }
6
7  int main() {
8      for (int i = 0; i < N; i++) {
9          add(&c[i], &a[i], &b[i]);
10     }
11 }
```

```
$ gcc add.c -o add
$ ./add
CPU: 0.684974 seconds
```

(Full code in add/add.c)

- For N-dimensional vectors A and B, compute C = A + B
- Vector addition is *embarrassingly parallel* → run the computation in parallel on the GPU
- Assign each GPU thread to be responsible for the element in the A, B or C arrays at some index idx
  - How do we determine idx?

How do we determine `idx`?

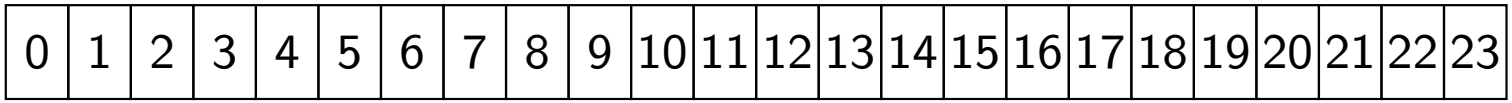


Recall CUDA's thread hierarchy, consisting of blocks of threads and grids of blocks

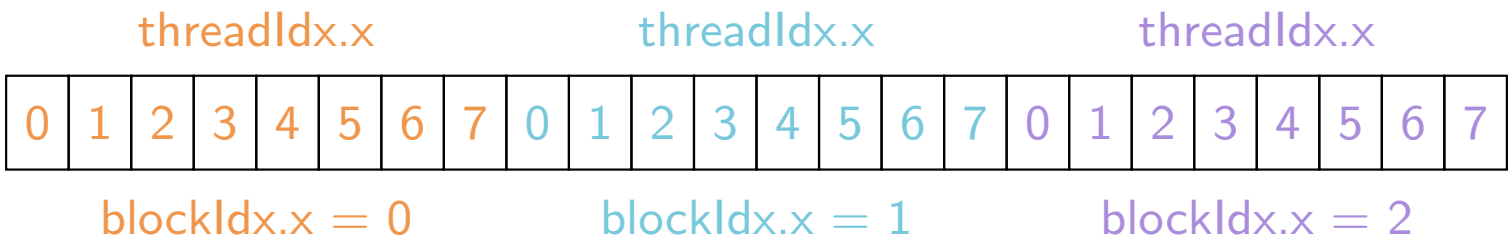
- Each block has a unique ID within the grid, accessed using `blockIdx`
- Each thread has a unique ID within the block, accessed using `threadIdx`
- Grids and blocks are both three dimensional (`x`, `y`, `z` components)
  - Dimensions can be accessed using `gridDim` and `blockDim`
- We will use one-dimensional grids and blocks here, i.e., `y` and `z` dimension is one, `y` and `z` index is always zero

How do we determine `idx`?

Suppose `N = 24`.



Say we launch a kernel with `<<<3, 8>>>` (`blockDim.x = 8, gridDim.x = 3`):



- Index 0 should be worked on by thread 0 in block 0
- Index 1 should be worked on by thread 1 in block 0
- ...
- Index 8 should be worked on by thread 0 in block 1
- In general: `idx = blockIdx.x * blockDim.x + threadIdx.x`

Vector Addition on the GPU

PRINCETON UNIVERSITY

We can write the kernel function by peeling off the for loop and assigning work to different threads (full code in add/add.cu):

```
__global__ void addKernel(float *c, float *a, float *b) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
        c[idx] = a[idx] + b[idx];
}

int main() {
    // allocate host memory for N floats and intialise
    float *a = (float *)malloc(N * sizeof(float)); // same for b, c

    // allocate device memory
    float *d_a, *d_b, *d_c;
    cudaMalloc((void**)&d_a, N * sizeof(float)); // same for d_b, d_c

    // copy input arrays from host to device
    cudaMemcpy(d_a, a, N * sizeof(float), cudaMemcpyHostToDevice); // same for b

    // launch the kernel
    addKernel<<<(N + threadsPerBlock - 1) / threadsPerBlock, threadsPerBlock>>>(d_c, d_a, d_b);
    cudaDeviceSynchronize();

    // copy results back to host
    cudaMemcpy(c, d_c, N * sizeof(float), cudaMemcpyDeviceToHost);

    // do stuff with c and deallocate memory
}
```

(Assignment 8 has you implement something similar.)

How much better did we do?

PRINCETON UNIVERSITY

CPU time:

```
CPU: 0.684974 seconds
```

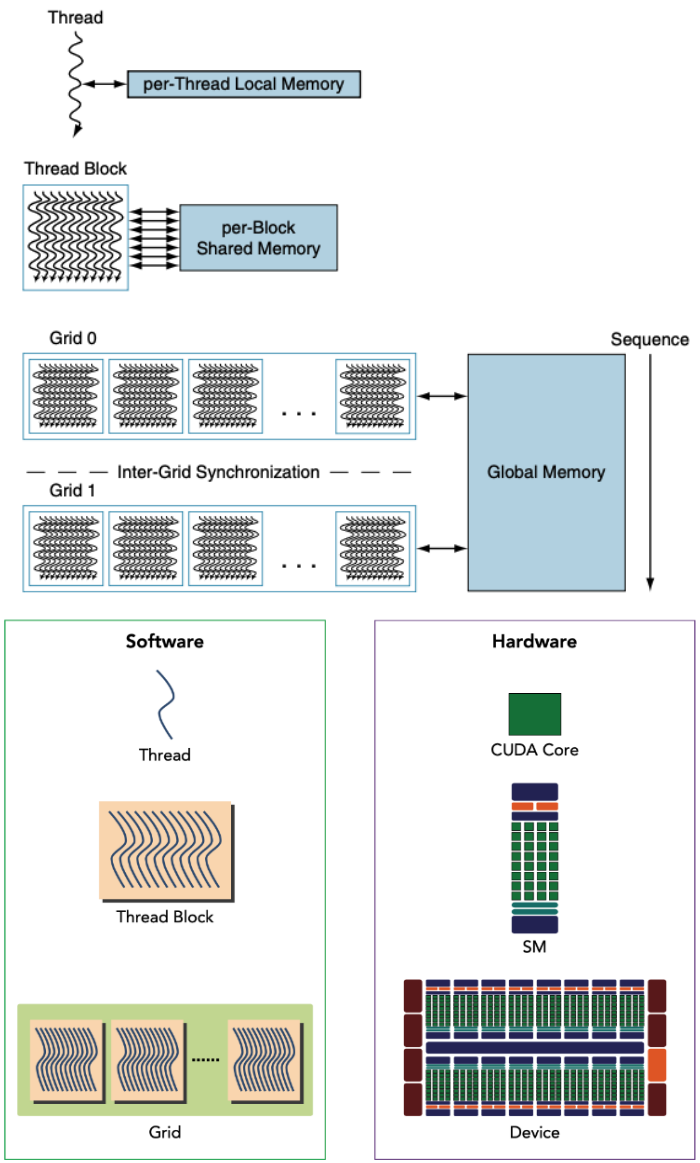
GPU time:

```
<<<4194304,64>>>      Kernel: 0.003667 seconds    Total: 0.109511 seconds
<<<2097152,128>>>     Kernel: 0.002618 seconds    Total: 0.108469 seconds
<<<1048576,256>>>     Kernel: 0.002577 seconds    Total: 0.108430 seconds
<<<524288,512>>>      Kernel: 0.002587 seconds    Total: 0.108402 seconds
<<<262144,1024>>>     Kernel: 0.002620 seconds    Total: 0.108476 seconds
<<<335545,800>>>      Kernel: 0.002699 seconds    Total: 0.108526 seconds
```

How do we choose the optimal block and grid dimensions?

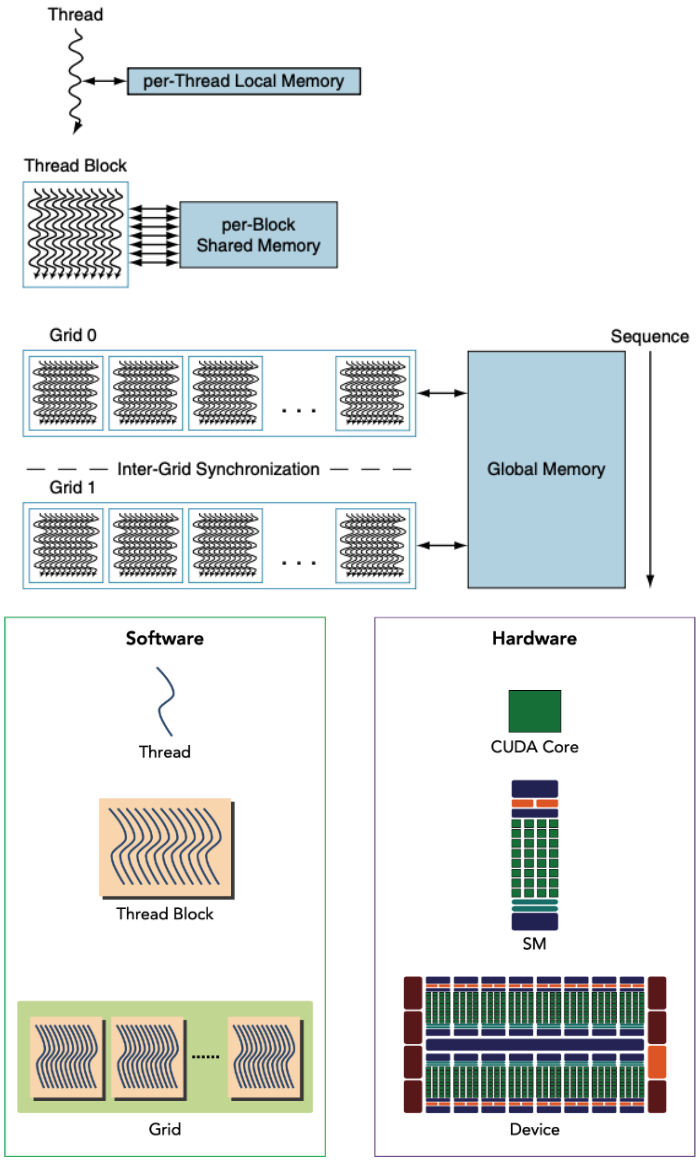
Overview

SIMT, Revisited



- CUDA uses a SIMT architecture
- All threads in a block are assigned to an SM simultaneously
  - A block cannot be assigned to an SM until it secures enough resources for all its threads to execute
  - Have to share memory resources of SM $\rightarrow$ limit on the number of threads per block (1024 on modern GPUs)
- Threads within a block are executed in groups of 32 called *warps*
  - Threads in a warp execute the same instruction at the same time
  - Each thread has its own program counter, register state, and can have an independent execution path
  - Instruction-level parallelism within a thread

SIMT, Revisited



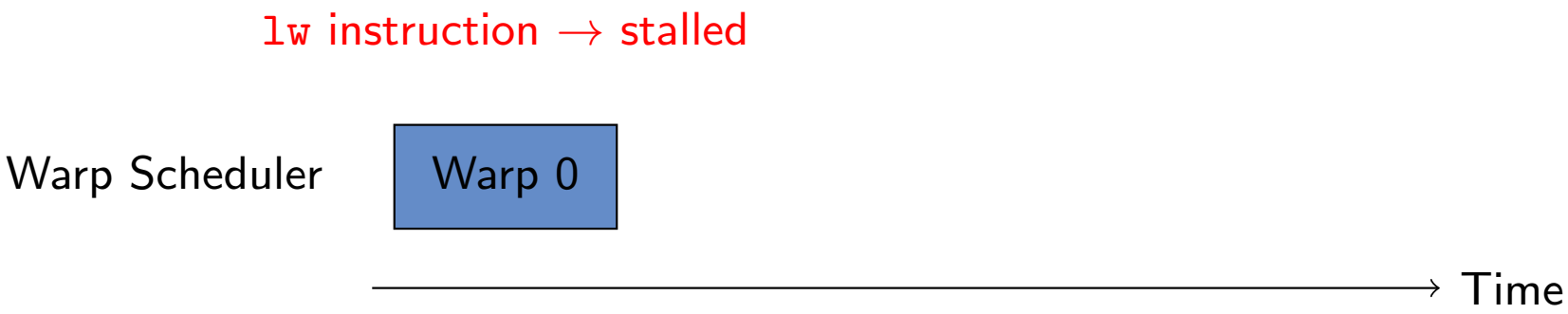
- CUDA uses a SIMT architecture
- All threads in a block are assigned to an SM simultaneously
  - A block cannot be assigned to an SM until it secures enough resources for all its threads to execute
  - Have to share memory resources of SM $\rightarrow$ limit on the number of threads per block (1024 on modern GPUs)
- Threads within a block are executed in groups of 32 called *warps*
  - Why bother with warps? Why not just let each thread execute independently?

Benefits of SIMT: Hardware Simplicity

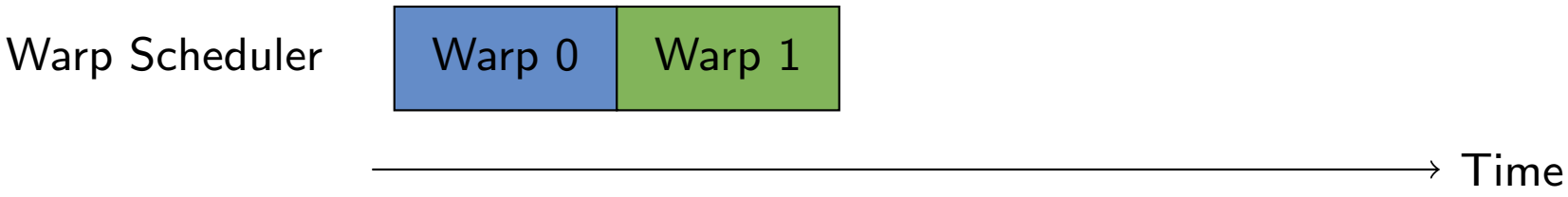
- Warps are not actually part of the CUDA programming model; the API does not expose them directly, but the hardware uses them internally to simplify control logic and improve performance (microarchitectural)
- All threads in a warp execute the same instruction
  - Each processing block only needs one fetch / decode unit
  - *Memory coalescing*: memory requests can be combined into fewer, larger transactions → no need for a load / store unit for each core
  - Implicit synchronisation
- More space on-chip for compute!

Benefits of SIMT: Latency Hiding

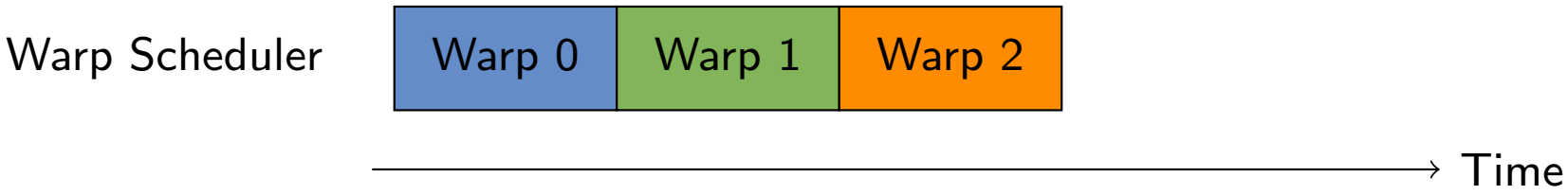
Benefits of SIMT: Latency Hiding



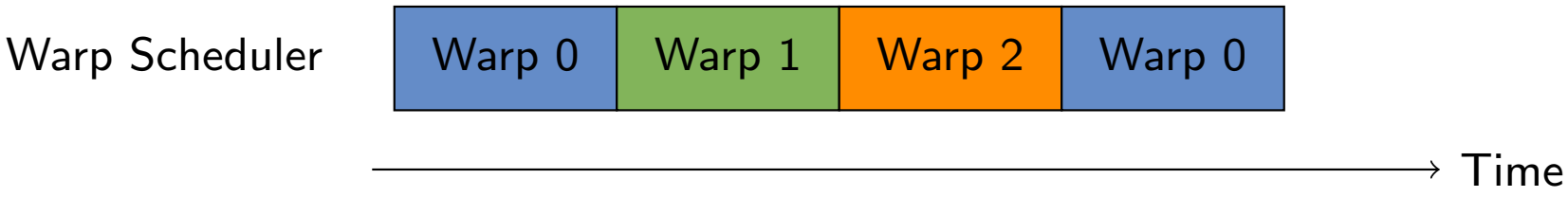
Benefits of SIMT: Latency Hiding



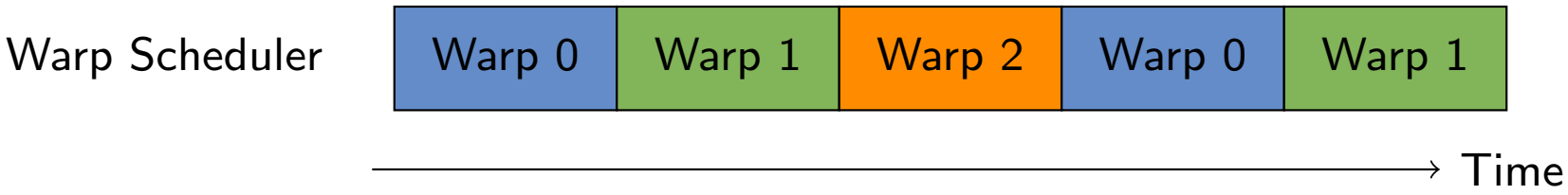
Benefits of SIMT: Latency Hiding



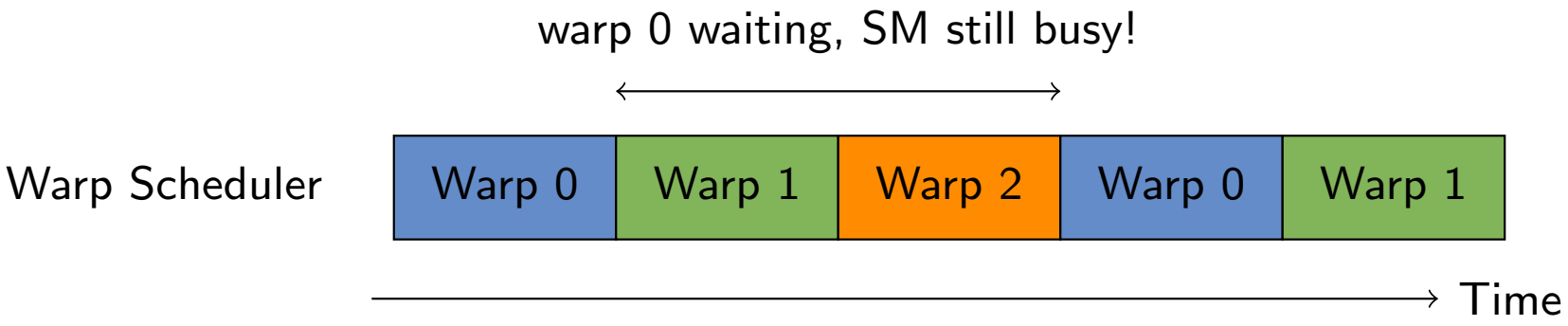
Benefits of SIMT: Latency Hiding



Benefits of SIMT: Latency Hiding

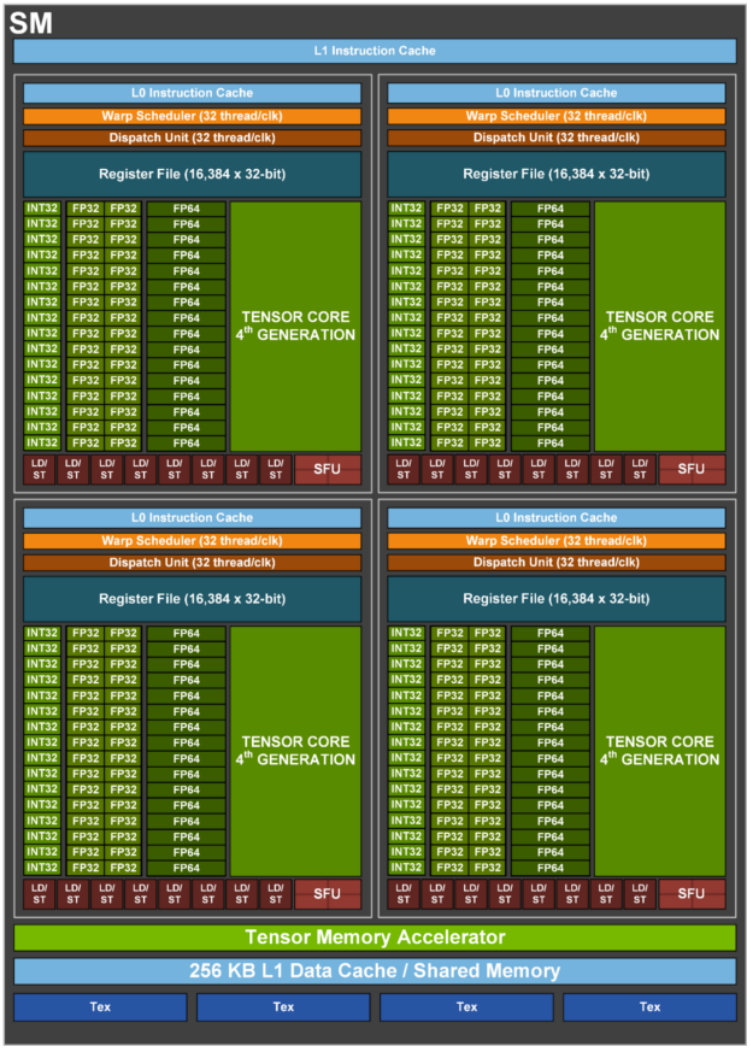


Benefits of SIMT: Latency Hiding



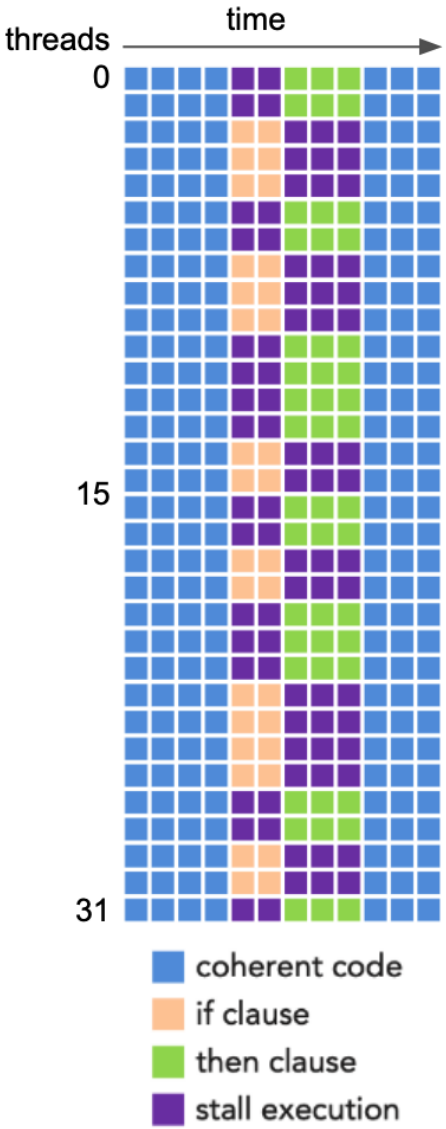
- Idea: Take advantage of thread-level parallelism to fully utilise the hardware → *latency hiding*
- The warp scheduler selects a warp on every cycle based on its state
  - Selected → actively executing
  - Stalled → not ready for execution
  - Eligible → ready for execution but not currently executing
- If a warp stalls, the warp scheduler picks another eligible warp to run
- GPU speedup comes mostly from latency hiding across many threads!
  - CPUs are designed instead for minimising latency

But wait, is there a contradiction?



- Threads are executed in groups of 32 called *warps*
  - Threads in a warp execute the same instruction at the same time
  - Each thread has its own program counter, register state and can have an independent execution path
- In each processing block of the SM, there is only one instruction cache
- Under these constraints, how can threads within a warp be executing independently at all?

Answer: Warp divergence



- Conditional statements in the kernel function can cause *warp divergence*, i.e., threads in the same warp execute different instructions
- If threads in a warp diverge, the warp serially executes each branch path, disabling threads that do not take that path $\rightarrow$ can result in reduced performance (up to 32$\times$)
- Solutions to avoid warp divergence:
  - Partition work such that all threads in the same warp take the same control path
  - Avoid the branch altogether using *branchless arithmetic* or compiler-driven *predication*
  - Move divergent logic outside of the kernel
- Demo code in `divergence/div.cu`

Choosing block and grid dimensions

PRINCETON UNIVERSITY

- How do we choose the optimal block and grid dimensions?
  - As with most things in systems, it depends!
- In general, conduct experiments to discover the best configuration for your workload and hardware. Some tips:
  - The number of threads per block should be a multiple of warp size to minimise idle threads within a warp
  - Avoid small block sizes to improve *occupancy* and prevent underutilisation of hardware resources
    - Rule of thumb: Start with 128 or 256 threads per block
  - Having too many threads per block can lead to fewer hardware resources (e.g., registers) available to each thread
  - The number of blocks should be much larger than the number of SMs available to exploit latency hiding through warp scheduling
- More detailed profiling results available by using `nsys profile`
- Examine hardware properties with `nvidia-smi` or CUDA Runtime API (`cudaGetDeviceProperties`)

Overview

Additional Topics and Resources

PRINCETON UNIVERSITY

- There are a number of other GPU and architecture-related topics that we could not cover in a single lecture that may be of interest:
  - Block-level synchronisation
  - Stencils and shared memory
  - Texture and constant memory
  - Bank conflicts and memory coalescing
  - Pinned (page-locked) memory
  - Warp shuffling
  - Reduction
  - Asynchronous streams and events
  - Unified memory
- The complete versions of the demo code presented in lecture is available at https://github.com/pollyren/cuda-example

References and Acknowledgements

PRINCETON UNIVERSITY

These slides use material from a number of sources, including:

- NVIDIA (2025). CUDA C Programming Documentation, v13.0
- Dr Giuseppe M. J. Barca (2023). *Introduction to GPU Architecture & Programming*
- Dr Giuseppe M. J. Barca (2023). *GPU SM Architecture & Execution Model*
- Princeton Research Computing (2025). *GPU Computing*
- Cyril Zeller, NVIDIA (2011). *CUDA C/C++ Basics*
- David Patterson & John Hennessy (2021). *Computer Organization and Design RISC-V Edition* (Appendix B)

Many of the graphics used in this presentation are directly sourced or adapted from publicly available documentation or educational materials (linked above). These assets are the property of their respective creators. They are reproduced here under fair use for instructional purposes.

Thanks!

Please remember to fill out the ICQ:

https://tinyurl.com/Fa25ICQ