NCI AUSTRALIA

Introduction to CUDA Programming

Dr. Joseph John
Staff Scientist
National Computational Infrastructure (NCI)

Prerequisite

- Experience with C programming.
- Experience with heap memory in C.
- Experience with bash or similar Unix shells.
- Experience with text editors such as vim, emacs or nano.

CPU Vs GPU

- Central processing Unit
- Optimised for Low Latency
- c = a +b
  - will take 0.2 - 0.3 nano sec
- 10 to 100 of concurrent ops
  - via SIMD + multiple cores

- If you want a single fast operation – use CPU
  - If-else

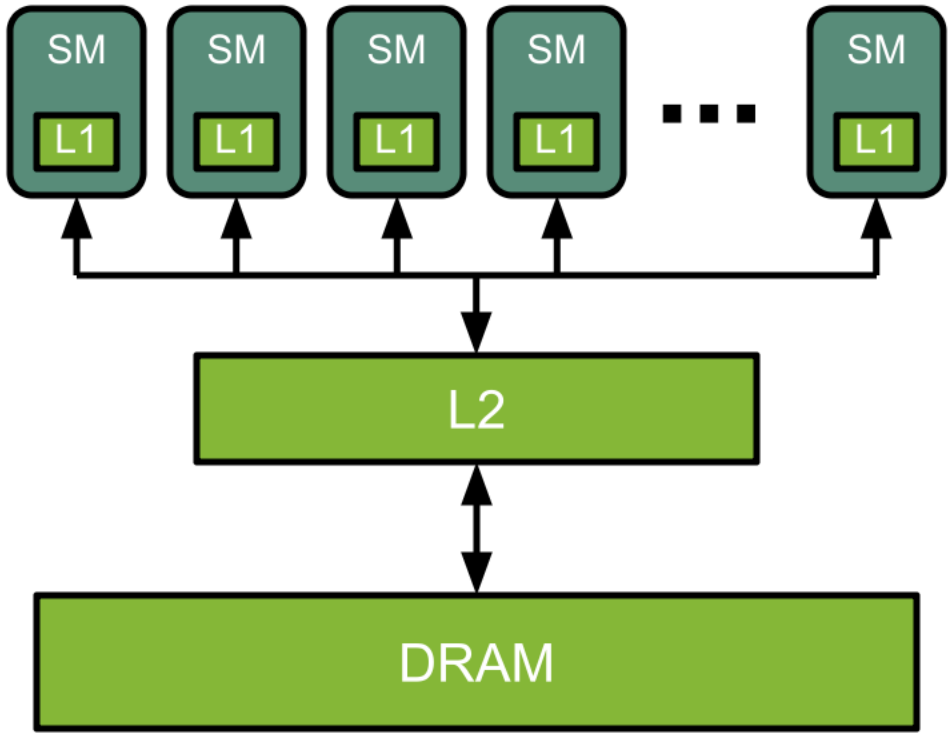
- Graphics processing Unit
- Optimised for throughput
- c = a +b
  - Will take 2.0 - 5.0 nano sec
- 10 of 1000s of concurrent ops
  - via thousands of CUDA cores

- If you want a slower but multiple operation – use GPU
  - Matrix multiplication

NVIDIA GPU Architecture

Streaming Multiprocessor (SM)

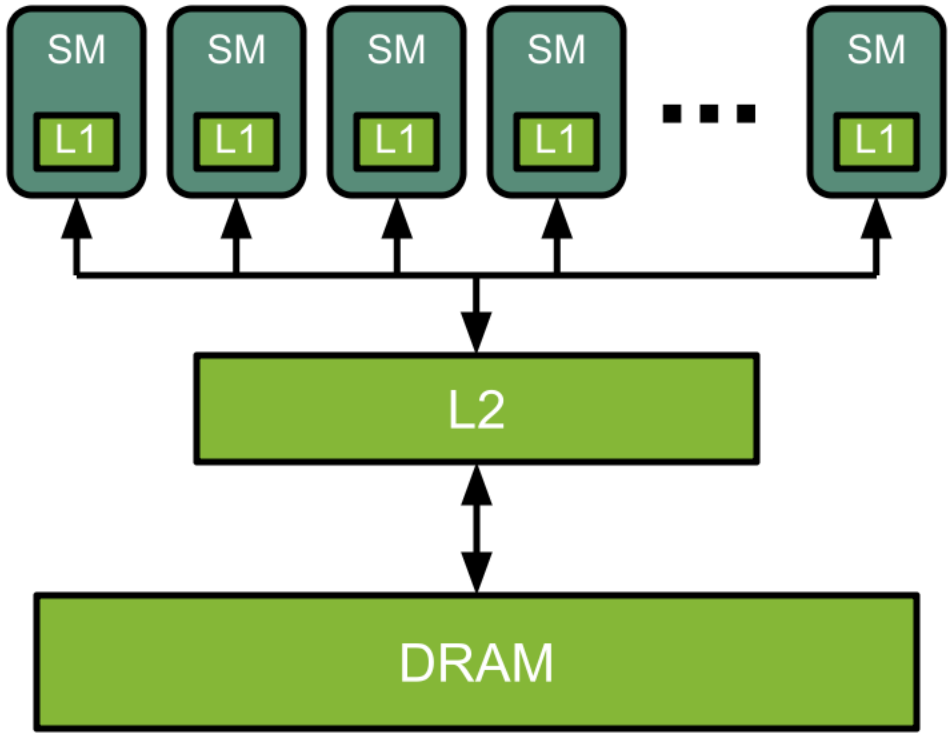
- Basic unit of computation in an NVIDIA GPU
- Multiple CUDA cores
- L1 cache
- Shared memory
- Executes multiple ***warps*** in parallel
  - 32 threads



NVIDIA GPU Architecture

L1 cache

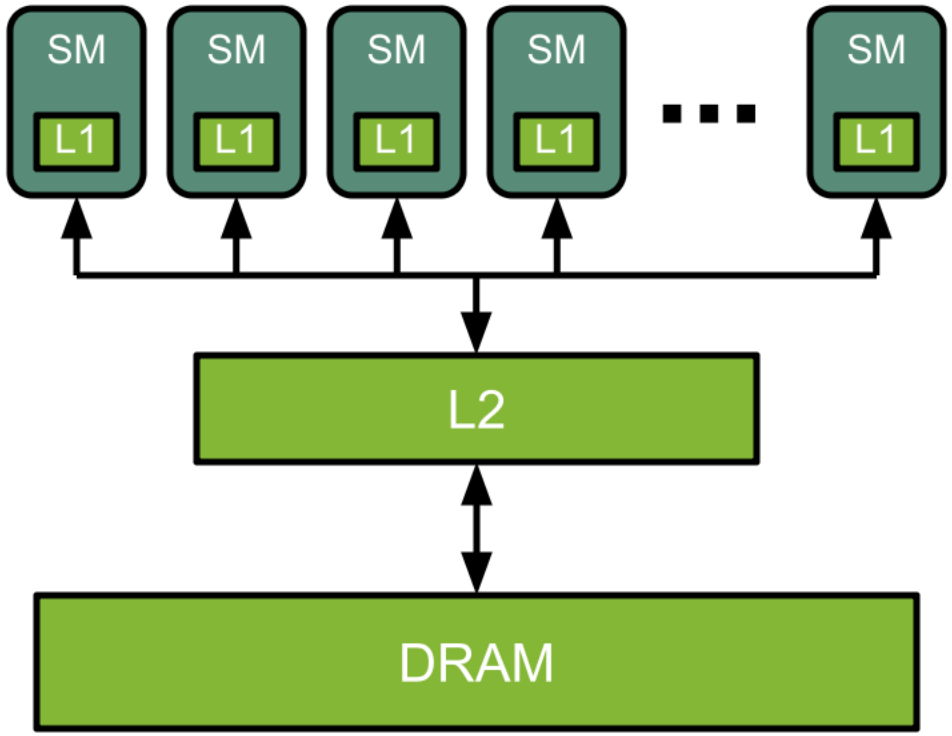
- Private to each SM
- Very low latency
- All threads in the SM shares the L1 cache
- 192 – 256 KB
- On-chip



NVIDIA GPU Architecture

L2 cache

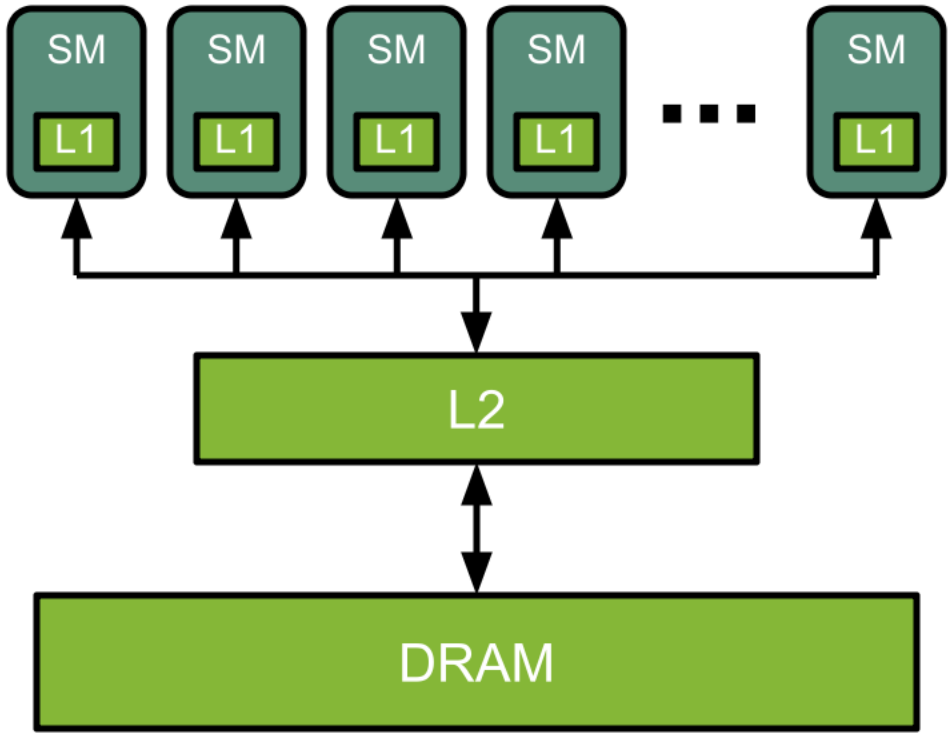
- Shared across SMs
- Latency
  - Higher than L1
  - Much lower than DRAM
- 40 – 50 MB



NVIDIA GPU Architecture

DRAM

- Shared across SMs
- Latency is highest
- 40 – 80 GM
- Off-chip



CUDA Cores

- The smallest execution units inside an SM
- Execute arithmetic and logic instructions for individual threads
- Not all cores are the same

	CUDA Cores				Tensor Cores					
NVIDIA Architecture	FP64	FP32	FP16	INT8	FP64	TF32	FP16	INT8	INT4	INT1
Volta	32	64	128	256			512			
Turing	2	64	128	256			512	1024	2048	8192
Ampere (A100)	32	64	256	256	64	512	1024	2048	4096	16384
Ampere, sparse						1024	2048	4096	8192	

CUDA Kernels

__global__

- __global__ defines a kernel function
  - runs on the device (GPU)
  - called from the host (CPU)
- Kernel function should return void

```
__global__ void add(int *a, int *b, int *c) {
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

// Launch kernel with 1 block of 256 threads
add<<<1, 256>>>(d_a, d_b, d_c);
```

CUDA Kernels

kernel launch configuration operator

- << < # blocks per grid, threads per block>>>
- Thread indexing in GPUs are hierarchical
- **Threads** are grouped into **blocks, Blocks** are grouped into a **Grid**
- <<<1, 256>>> - Grid with1 block. Block with 256 threads

```
__global__ void add(int *a, int *b, int *c) {
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

// Launch kernel with 1 block of 256 threads
add<<<1, 256>>>(d_a, d_b, d_c);
```

CUDA Kernels

built-in variables

- **threadIdx** – identifies a thread within its block
  - threadIdx.x
  - threadIdx.y
  - threadIdx.z

CUDA Kernels

built-in variables

- **blockIdx** – identifies the block's position in the grid.
  - blockIdx.x
  - blockIdx.y
  - blockIdx.z

CUDA Kernels

built-in variables

- **blockDim** – tells the size of the block (number of threads in each dimension).
  - blockDim.x
  - blockDim.y
  - blockDim.z

CUDA Kernels

built-in variables

- **gridDim** – tells the size of the grid (number of blocks in each dimension).
  - gridDim.x
  - gridDim.y
  - gridDim.z

Index Translation

1D grid and 1D block

- <<< 4, 16>>>

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
```

Index Translation

2D grid and 2D block

- dim3 gridDim(2, 2)
- dim3 blockDim(4, 4);
- <<< gridDim , blockDim >>>

```
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int globalIdx = y * (blockDim.x * gridDim.x) + x;
```

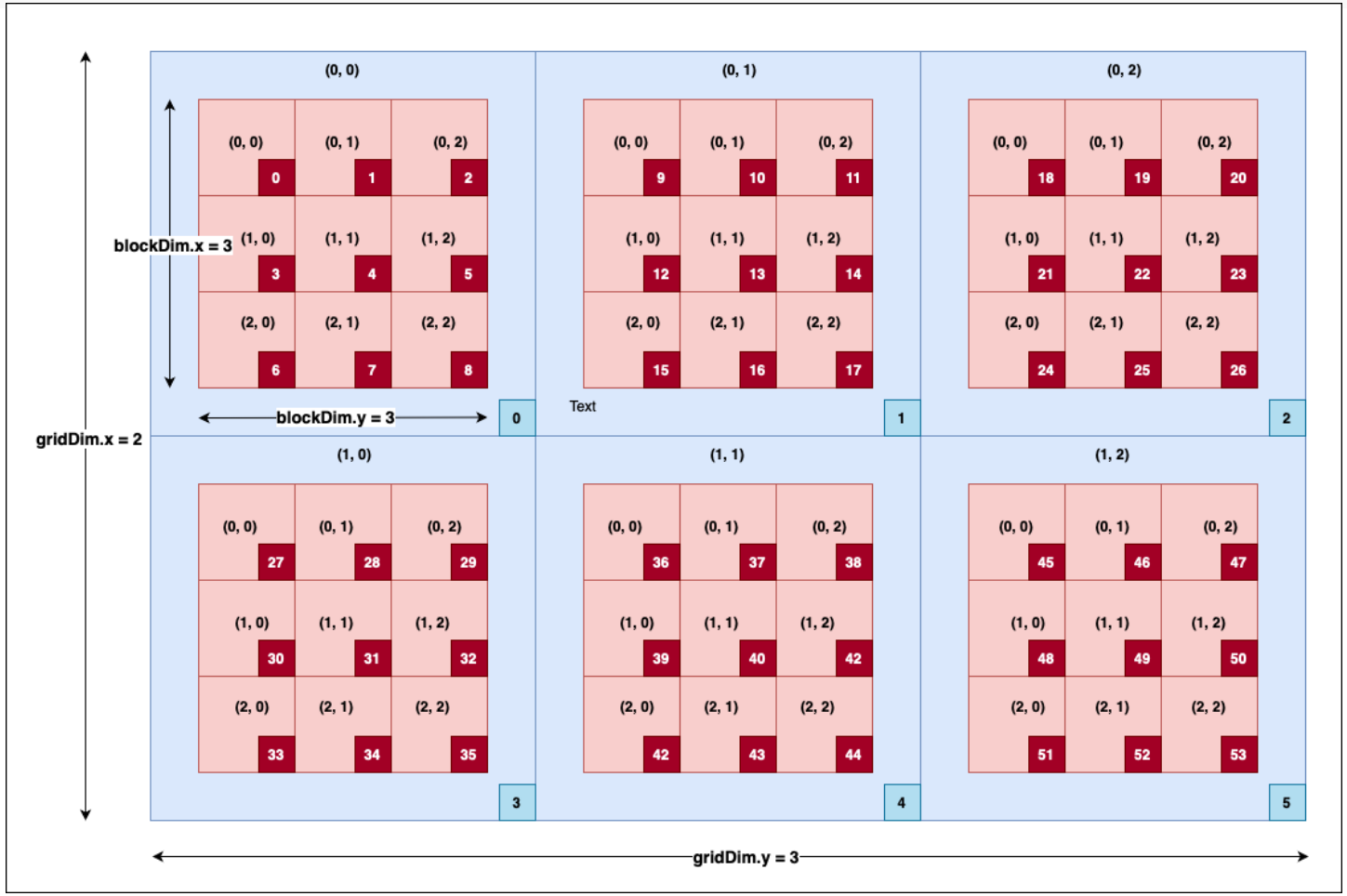
Index Translation

3D grid and 3D block

- dim3 gridDim(2, 2, 2)
- dim3 blockDim(4, 4, 4);
- <<< gridDim , blockDim >>>

```
int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int z = threadIdx.z + blockIdx.z * blockDim.z;

int globalIdx = x + y * (blockDim.x * gridDim.x)
                  + z * (blockDim.x * gridDim.x * blockDim.y * gridDim.y);
```

Index Translation



Kernel Execution

- Each threads runs the ***same*** code
- Each thread will have a ***different*** global index
  - Thread 0 – c[0] = a[0] + b[0]
  - Thread 1 – c[1] = a[1] + b[1]
  - Thread n-2 – c[n-2] = a[n-2] + b[n-2]
  - Thread n-1 – c[n-1] = a[n-1] + b[n-1]

```
__global__ void add_vectors(float *a, float *b, float *c, int n) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    if (idx < n)
        c[idx] = a[idx] + b[idx];
}
```

Memory Management

- Memory management in CUDA mainly involves 4 key operations:
  - Device memory allocation
  - Host to Device (H2D) memory copy
  - Device to Host (D2H) memory copy
  - Device memory deallocation

Memory Management

cudaMalloc

```
float *d_a, *d_b, *d_c;
int n = 1024;

// Allocate memory on the GPU
cudaMalloc((void**)&d_a, n * sizeof(float));
cudaMalloc((void**)&d_b, n * sizeof(float));
cudaMalloc((void**)&d_c, n * sizeof(float));
```

Memory Management

cudaMemcpy

```
cudaMemcpy(d_a, h_a, n * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, n * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_c, h_c, n * sizeof(float), cudaMemcpyHostToDevice);
```

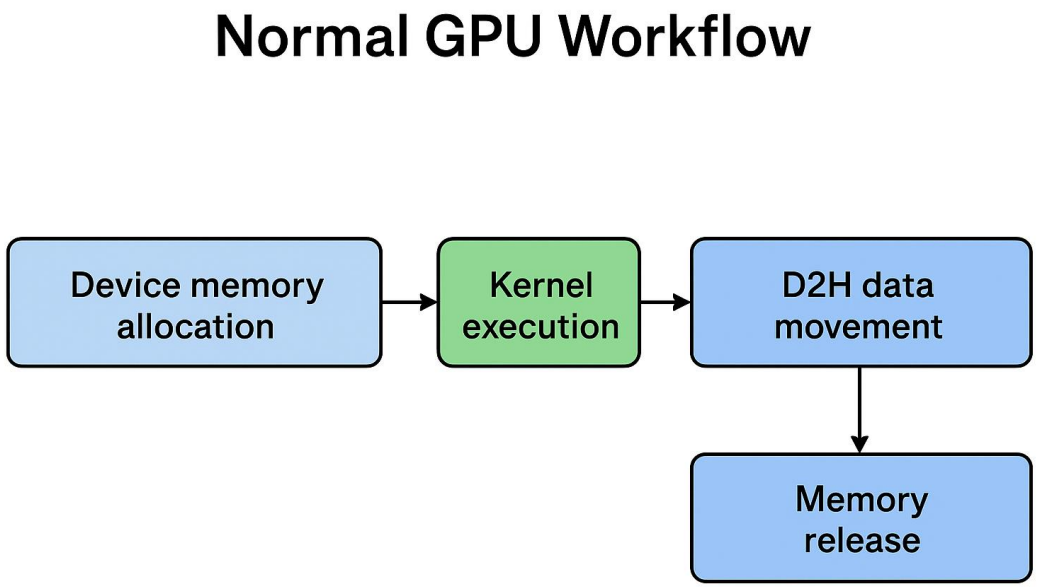
```
// Copy data from host to device
cudaMemcpy(h_a, d_a, n * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(h_b, d_b, n * sizeof(float), cudaMemcpyDeviceToHost);
cudaMemcpy(h_c, d_c, n * sizeof(float), cudaMemcpyDeviceToHost);
```

Memory Management

cudaFree

```
// Free device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```

CUDA Workflow



Warps

- Warps are a grouping of 32 CUDA threads
- The threads in warp always execute the same code
  - Lock step execution
- <<<1, 16 >>>
  - A block only has 16 threads
  - The warp will still have 32 threads
  - 16 threads are idle

Warps

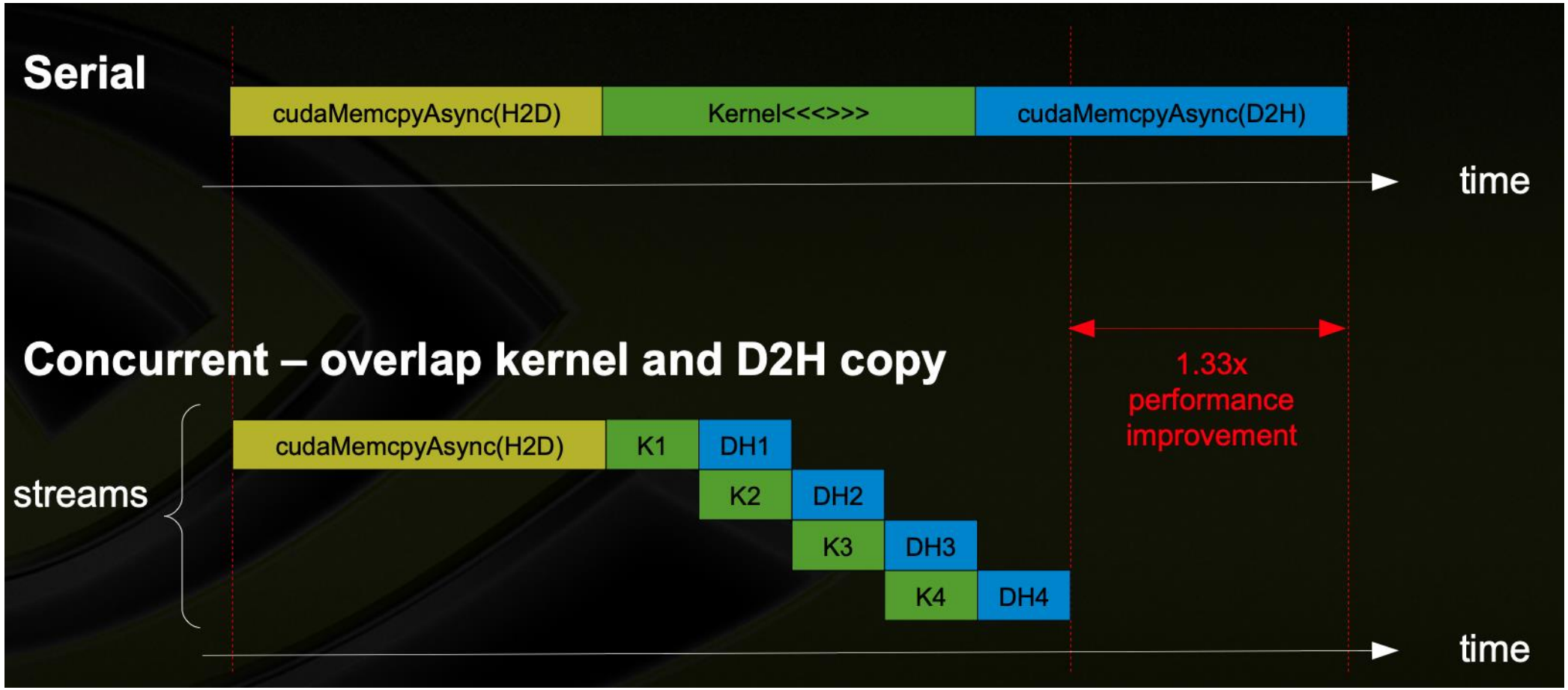
- Warps are a grouping of 32 CUDA threads
- The threads in warp always execute the same code
  - Lock step execution
- <<<1, 50 >>>
  - A block has 50 threads
  - This bock will need 2 warps (2x32)
  - 16 threads in the first warp fully used
  - Last 14 threads in the second warp will be idle

Warps

- Warps are a grouping of 32 CUDA threads
- The threads in warp always execute the same code
  - Lock step execution
- <<<1, 64 >>>
  - A block has 64 threads
  - This bock will need 2 warps (2x32)
  - Both warps fully used
  - No idle threads

Asynchronous API

CUDA Streams



Asynchronous API

CUDA Streams

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
```

Asynchronous API

cudaMemcpyAsync()

```
// Copy and launch kernel on stream1
cudaMemcpyAsync(d_data1, h_data1, size1, cudaMemcpyHostToDevice, stream1);
int threads = 256;
int blocks1 = (n1 + threads - 1) / threads;
kernel1<<<blocks1, threads, 0, stream1>>>(d_data1, n1);

// Copy and launch kernel on stream2
cudaMemcpyAsync(d_data2, h_data2, size2, cudaMemcpyHostToDevice, stream2);
int blocks2 = (n2 + threads - 1) / threads;
kernel2<<<blocks2, threads, 0, stream2>>>(d_data2, n2);
```

Shared Memory

- Shared among the threads within the block
- Faster than global memory
- Reduces global memory access

Shared Memory

Static shared memory

```
 __global__ void addKernel(int *input, int *output)
{
    __shared__ int sdata[256];  // dynamic shared memory

    int tid = threadIdx.x;
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Load data from global to shared memory
    sdata[tid] = input[idx];
    __syncthreads();

    // Do something with shared memory
    sdata[tid] += 10;
    __syncthreads();

    // Store result back to global memory
    output[idx] = sdata[tid];
}
```

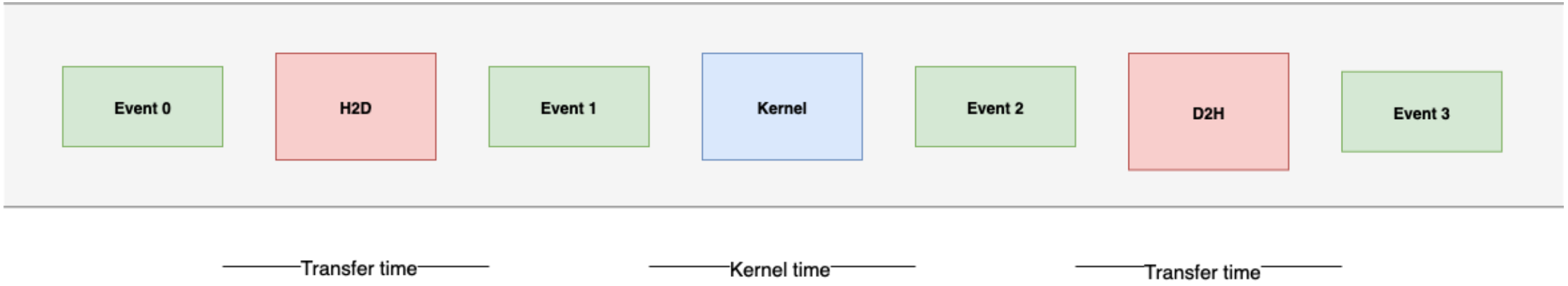
Shared Memory

Dynamic shared memory

```
__global__ void addKernel(int *input, int *output)
{
    extern __shared__ int sdata[];  // dynamic shared memory
    int tid = threadIdx.x;
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Load data from global to shared memory
    sdata[tid] = input[idx];
    __syncthreads();

    // Do something with shared memory
    sdata[tid] += 10;
    __syncthreads();

    // Store result back to global memory
    output[idx] = sdata[tid];
}

int main()
{
    int threads = 256;
    int sharedMemSize = threads * sizeof(int);  // Size of dynamic shared memory

    ...
    ...
    ...

    addKernel<<<1, threads, sharedMemSize>>>(d_input, d_output);
}
```

CUDA Events

- Measure time between two operations in the same stream
- Can measure time between two events in different streams
  - Generally not recommended
  - Synchronization required



Unified Memory

- Write applications without worrying about the complexities of managing memory between the host (CPU) and device (GPU)
- Single address space for both the host and device, enabling seamless data sharing and access
- Unified Memory automatically migrates data between the host and device as needed.
  - When the host accesses data that is currently on the device, Unified Memory will automatically transfer it to the host memory
  - and vice versa
  - This migration is managed by the CUDA runtime, which tracks memory accesses and performs the necessary transfers transparently

Unified Memory

cudaMallocManaged

- Allocate memory that is accessible from both the host and device.

```
int *data;
cudaMallocManaged(&data, size * sizeof(int));

// Use data on the host
for (int i = 0; i < size; i++) {
    data[i] = i;
}

// Use data on the device
kernel<<<blocks, threads>>>(data);

// Synchronize to ensure all operations are complete
cudaDeviceSynchronize();
```

Unified Memory

cudaMemPrefetchAsync

- Prefetch data in advance
- cudaCpuDeviceId – denotes the host

```
// Example of using cudaMemPrefetchAsync
int *data;
cudaMallocManaged(&data, size * sizeof(int));

// Prefetch data to the host from GPU 0
cudaMemPrefetchAsync(data, size * sizeof(int), cudaCpuDeviceId);

// Use data on the host
for (int i = 0; i < size; i++) {
    data[i] += 1;
}

// Prefetch data back to GPU 0
cudaMemPrefetchAsync(data, size * sizeof(int), 0);

// Launch kernel on the device
kernel<<<blocks, threads>>>(data);

// Synchronize to ensure all operations are complete
cudaDeviceSynchronize();
```

Unified Memory

Pinned Memory

- The host memory to be pinned
- It cannot be paged out by the operating system

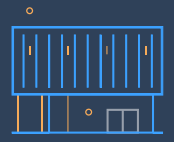
NCI AUSTRALIA

NCI Contacts

General enquiries: +61 2 6125 9800

Support: help@nci.org.au

Email: first.last@anu.edu.au



Address

NCI, ANU Building 143
143 Ward Road
The Australian National University
Canberra ACT 2601

License

Slide Title

Slide Subtitle

- Slide text
  - Slide sub-points