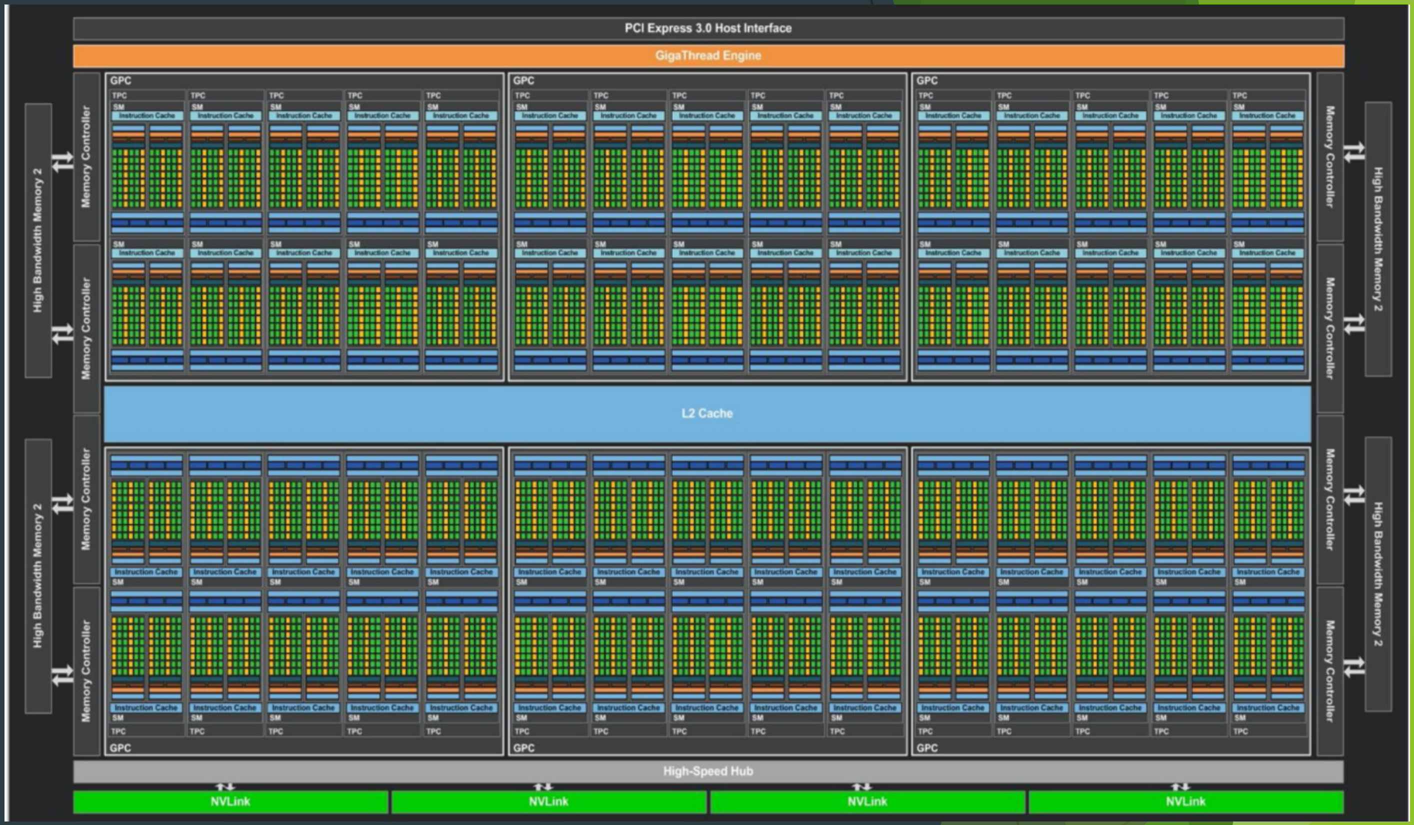
GUDA Introduction

林政宏

國立台灣師範大學電機系

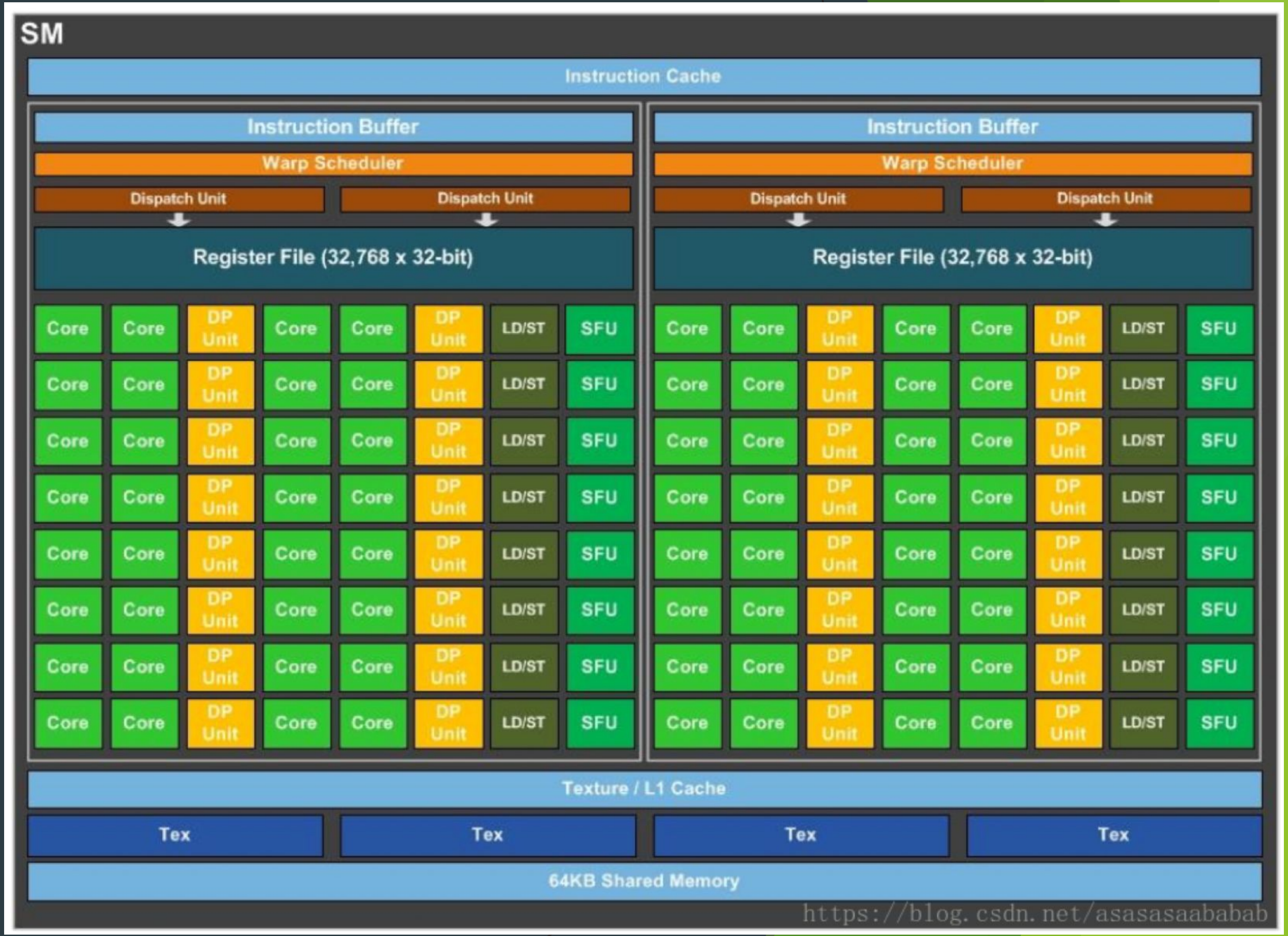
Nvidia Tesla P100

- 共包含60個SM（6GPC x 10SM）
- SM(Streaming Multiprocessor):
  - 多個SP(Streaming Processor), 又稱 CUDA core
  - 其他硬體资源如：warp scheduler，register，shared memory等
  - 同一个block中的warp一定分配给同一個SM
  - 每個SM有自己獨立的local L1 cache，但所有SM會共用Device的L2 cache



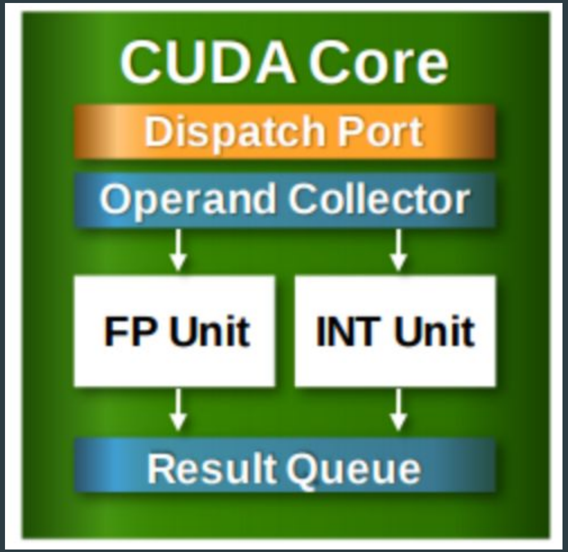
SM(Streaming Multiprocessor)

- 每個SM有自己的內部结構，如下圖所示GP100（Nvidia Tesla P100）中的SM內部结構，在GP100裡，每一個SM有左右兩個SM Processing Block（SMP）
- SP(Streaming Processor):最基本的處理單元，也稱為CUDA core，表示在上圖中為綠色的小格子
  - 相當於一個簡易的CPU
  - 內部包括控制單元Dispatch Port、Operand Collector，以及浮點計算單元FP Unit、整數計算單元Int Unit，另外還包括計算結果隊列。當然還有Compare、Logic、Branch等



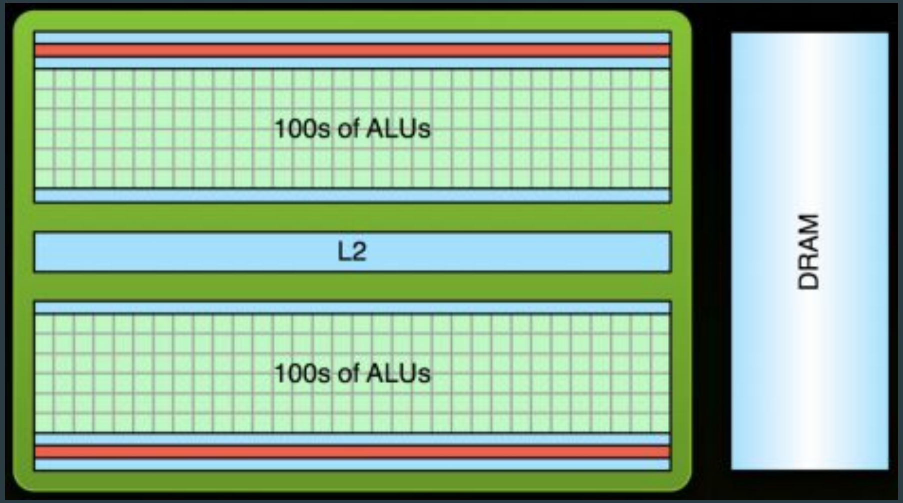
SP(Streaming Processor)

- 最基本的處理單元，也稱為CUDA core，表示在上圖中為綠色的小格子
  - 相當於一個簡易的CPU
  - 內部包括控制單元Dispatch Port、Operand Collector，以及浮點計算單元FP Unit、整數計算單元Int Unit，另外還包括計算結果隊列。當然還有Compare、Logic、Branch等



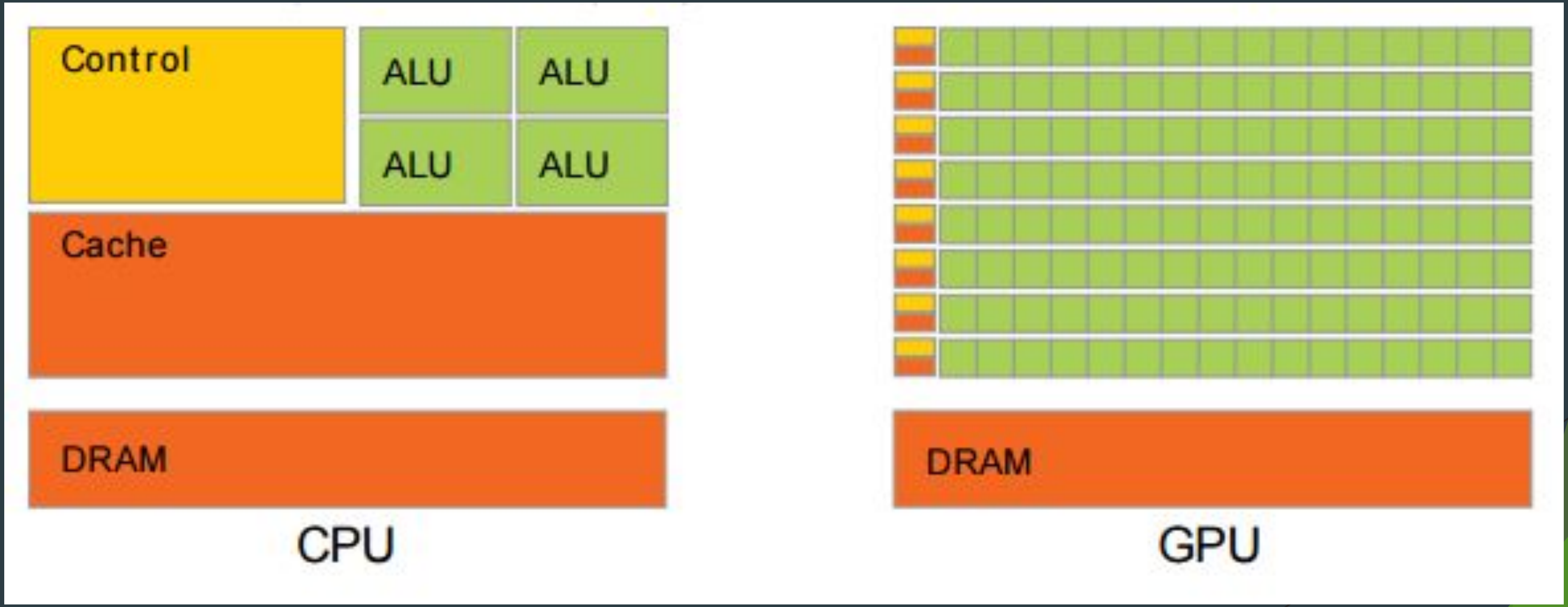
GPU Computing

- GPU: Graphics Processing Unit
- Traditionally used for real-time rendering
- High computational density (100s of ALUs) and memory bandwidth (100+ GB/s)
- Throughput processor: 1000s of concurrent threads to hide latency (vs. large fast caches)



CPU與GPU結構對比

- The CPU relies on larger caches and complex logic control (such as pipeline, out-of-order execution, instruction prediction and preprocessing, etc.) to hide the delay
- GPU relies on higher parallelism to hide the delay



Floating-Point Operations per Second for the CPU and GPU

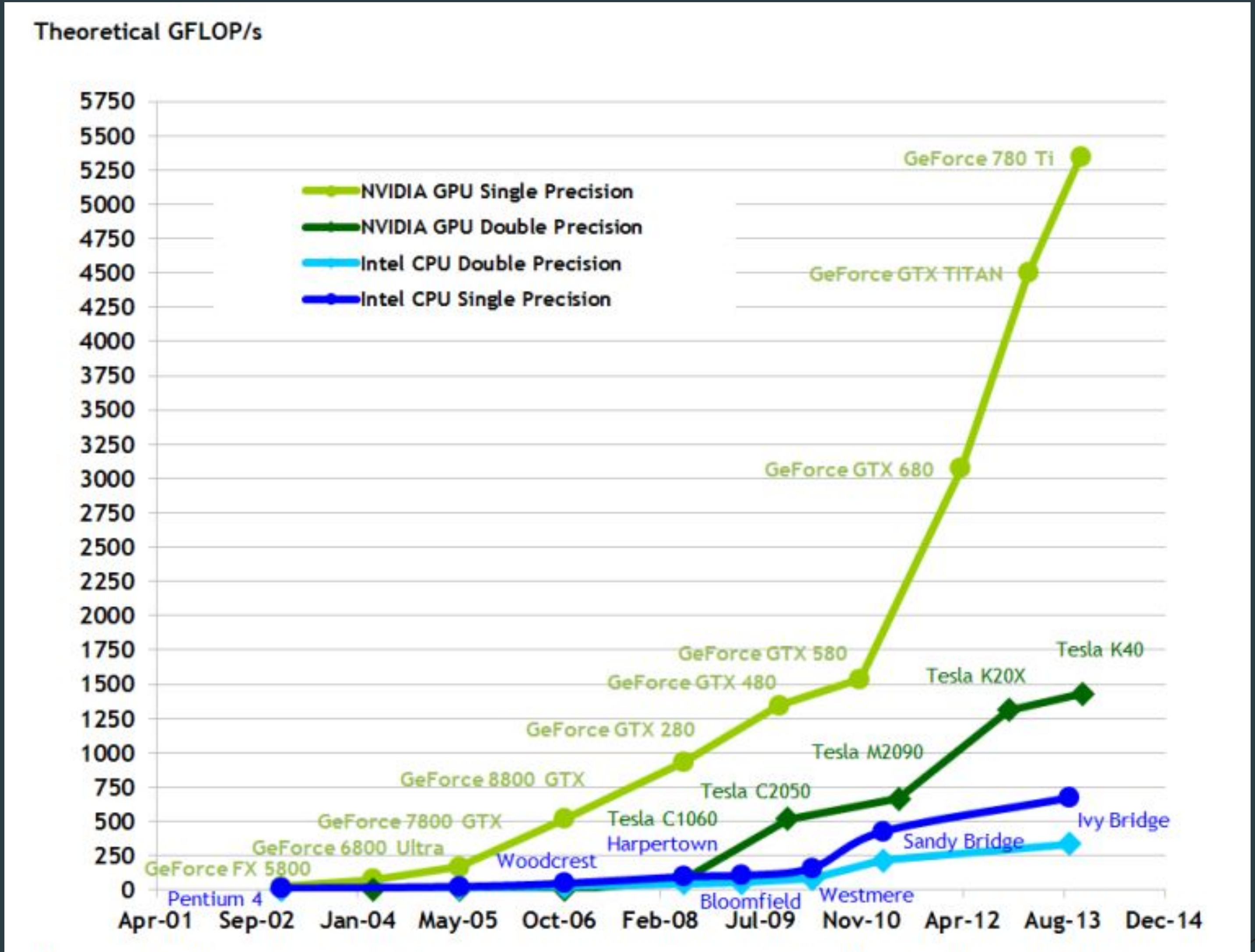
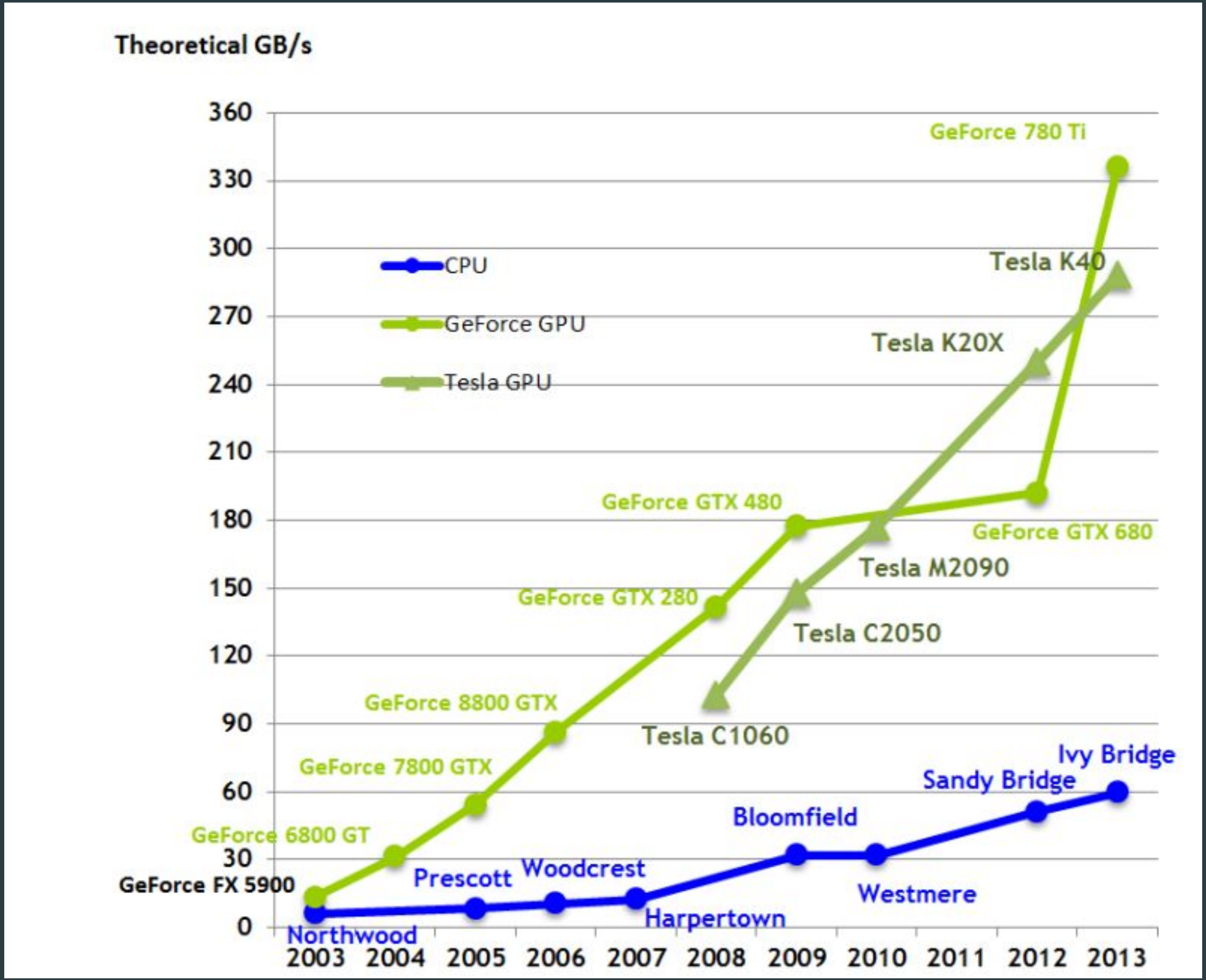


Figure 1 Floating-Point Operations per Second for the CPU and GPU

Memory Bandwidth for the CPU and GPU

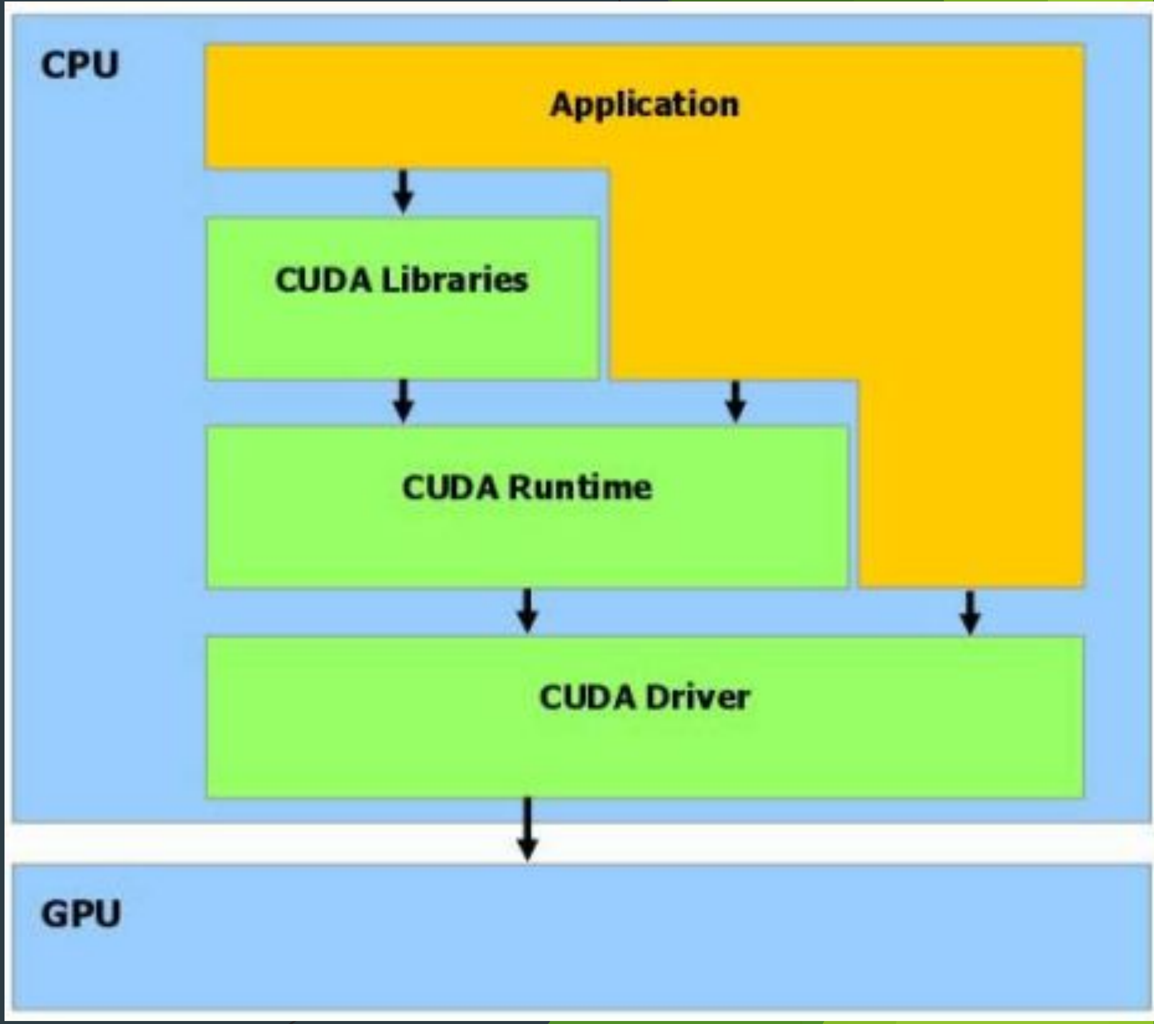


CUDA ™: A General-Purpose Parallel Computing Platform and Programming Model

- CUDA : Compute Unified Device Architecture
  - CUDA is a compiler and toolkit for programming NVIDIA GPUs
  - Enable heterogeneous computing and horsepower of GPUs
  - CUDA API extends the C/C++ programming language
  - Express SIMD parallelism
  - Give a high level abstraction from hardware

CUDA software stack

- CUDA library
  - Programming language function API
- CUDA runtime API
  - High-level API
- CUDA driver API
  - Low-level API



Heterogeneous Computing

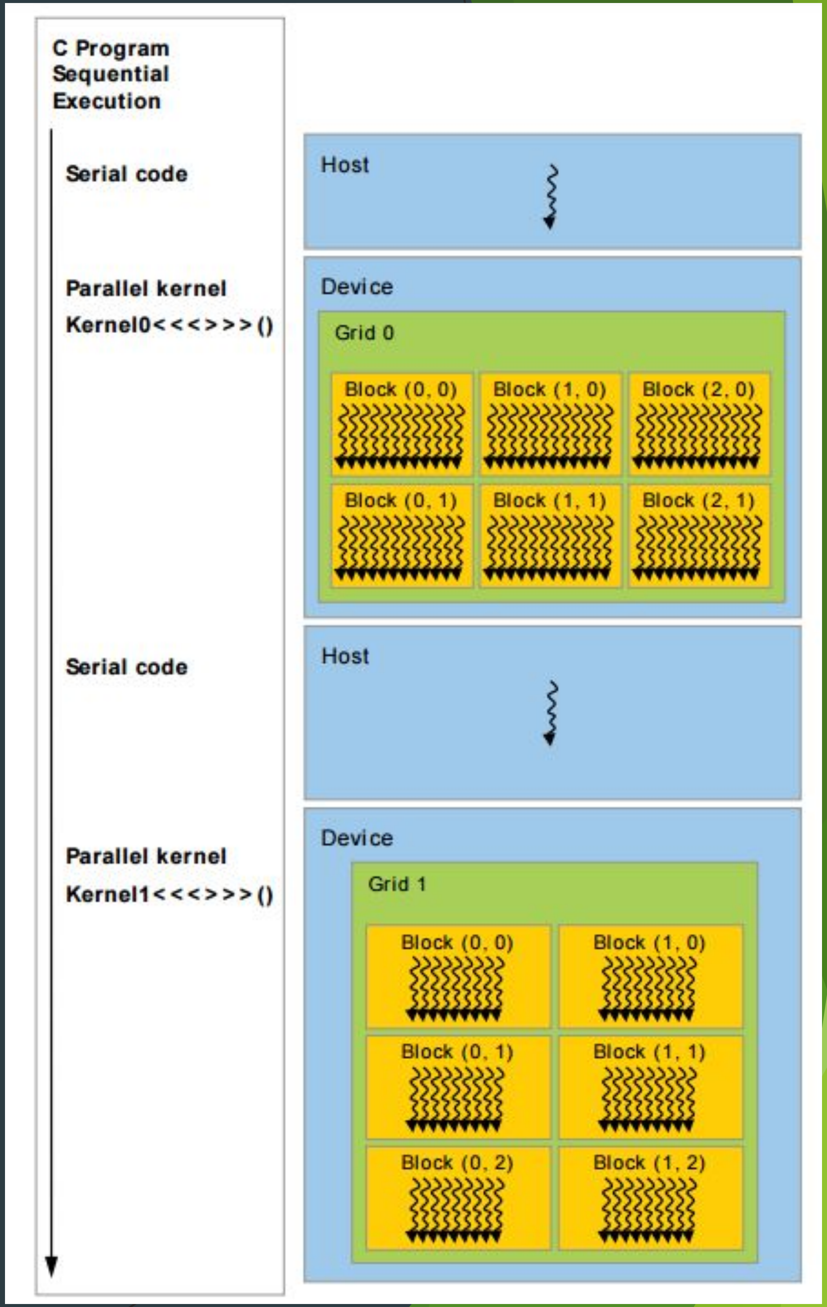
- Terminology:
- Host: The CPU and its memory (host memory)
- Device: The GPU and its memory (device memory)

Host

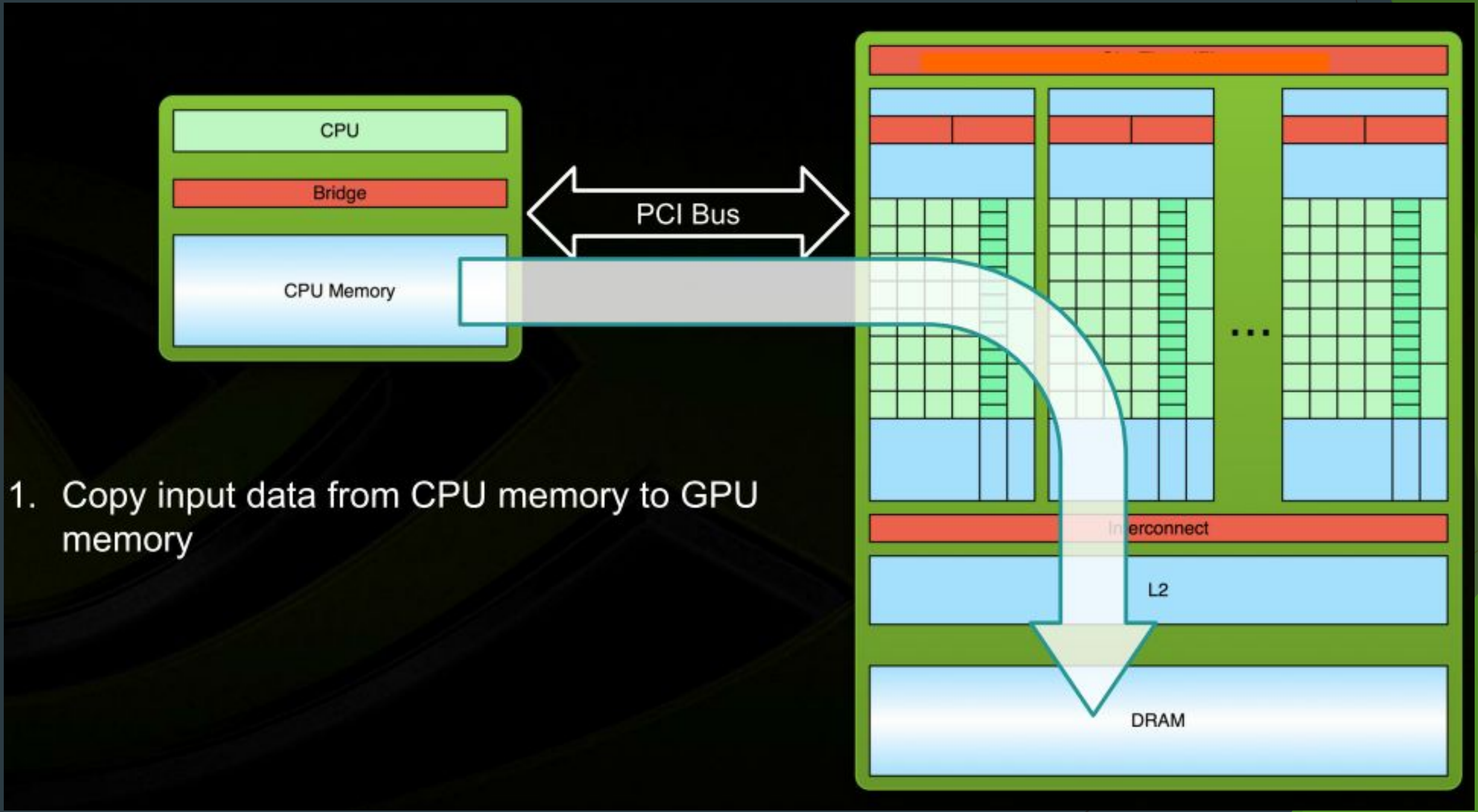
Device

Heterogeneous Programming

- Host : CPU
- Device : GPU
- Kernel : functions executed on GPU
- Thread : the basic execution unit
- Block : a group of threads
- Grid : a group of blocks

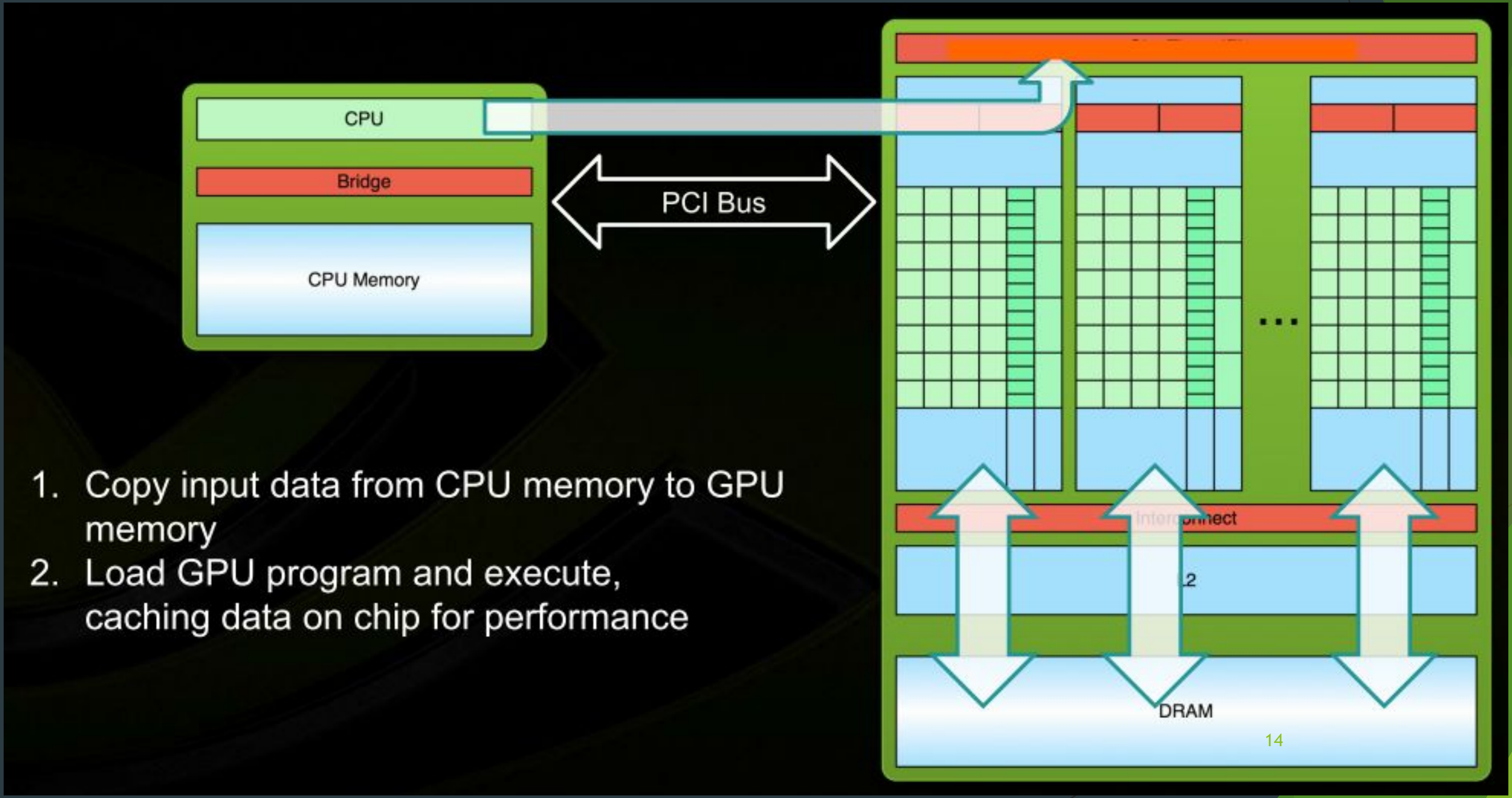


CUDA Program Flow



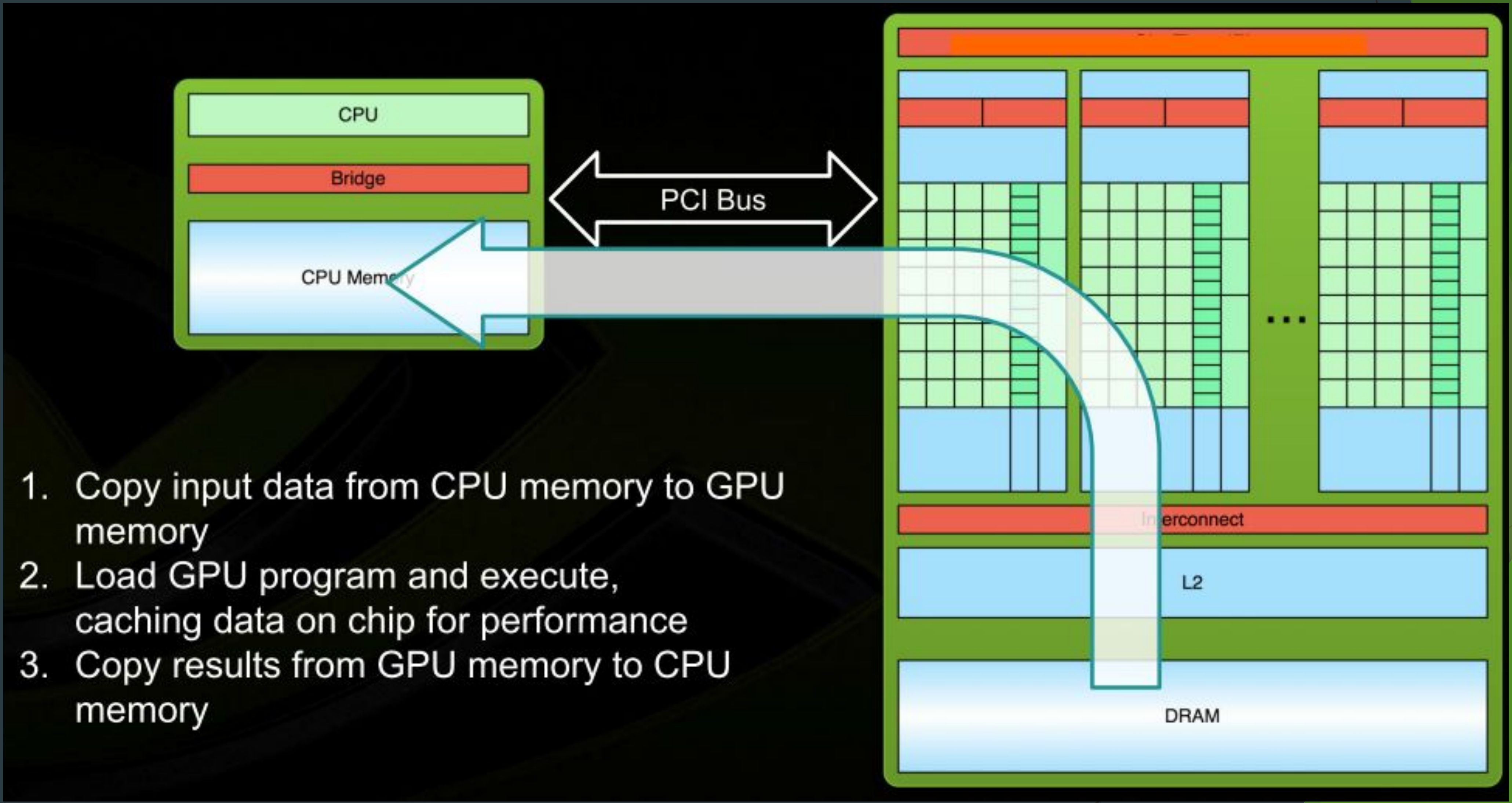
1. Copy input data from CPU memory to GPU memory

CUDA Program Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

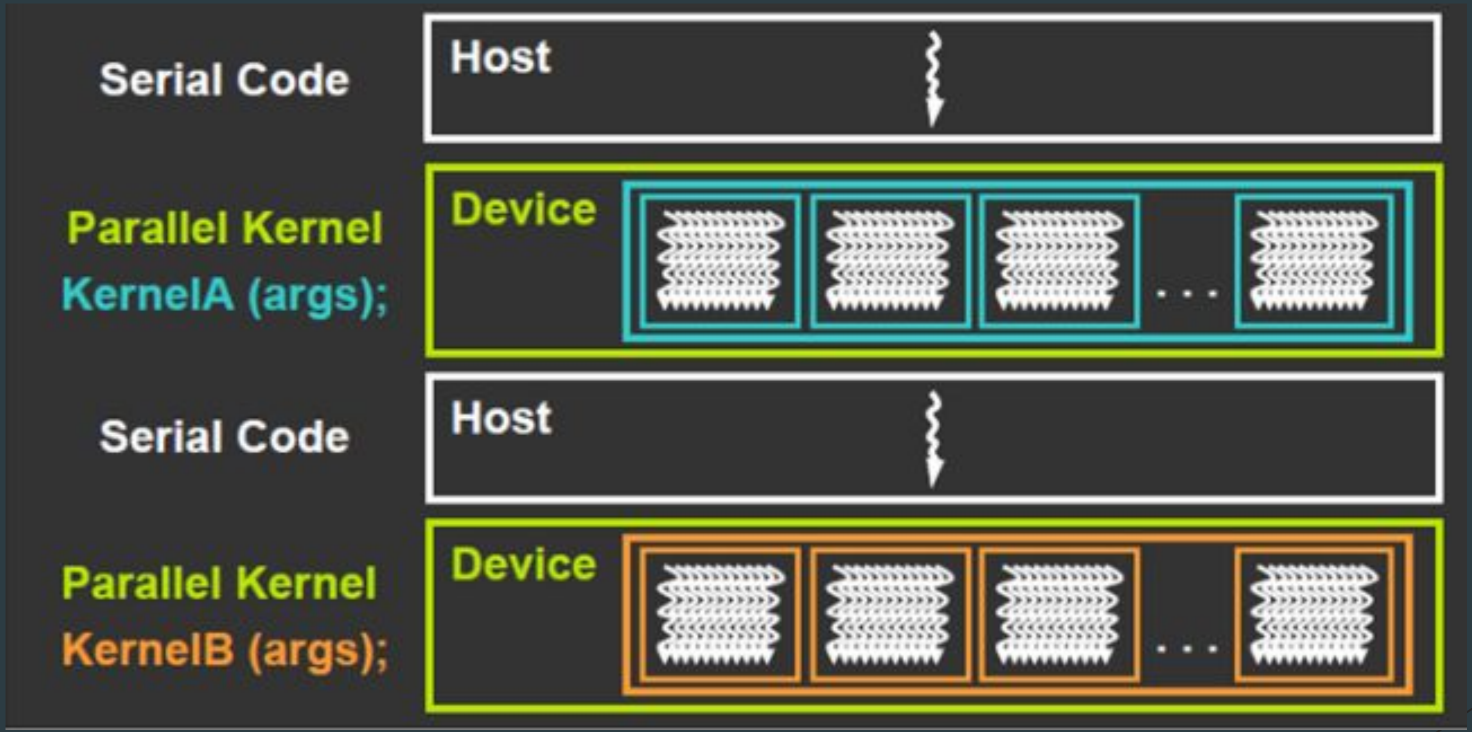
CUDA Program Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

CUDA Programming Model

- CUDA = serial program with parallel kernels, all in C
  - Serial C code executes in a host thread (i.e. CPU thread)
  - Parallel kernel C code executes in many devices threads across multiple processing elements (i.e. GPU threads)



CUDA program framework

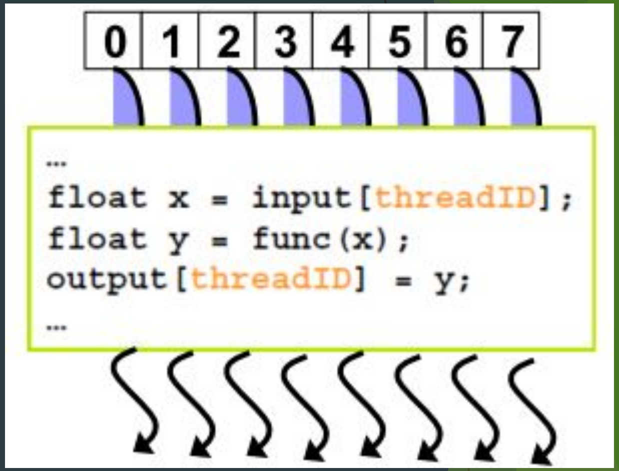
GPU code (parallel)

CPU code
(serial or parallel if p-thread/OpenMP/TBB/MPI is used.)

```
#include <cuda_runtime.h>

__global__ void my_kernel(…) {
   …
}

int main() {
   …
   cudaMalloc(…)
   cudaMemcpy(…)
   …
  my_kernel<<<nblock,blocksize>>>(…)
   …
   cudaMemcpy(…)
   …
}
```

Kernel = Many Concurrent Threads

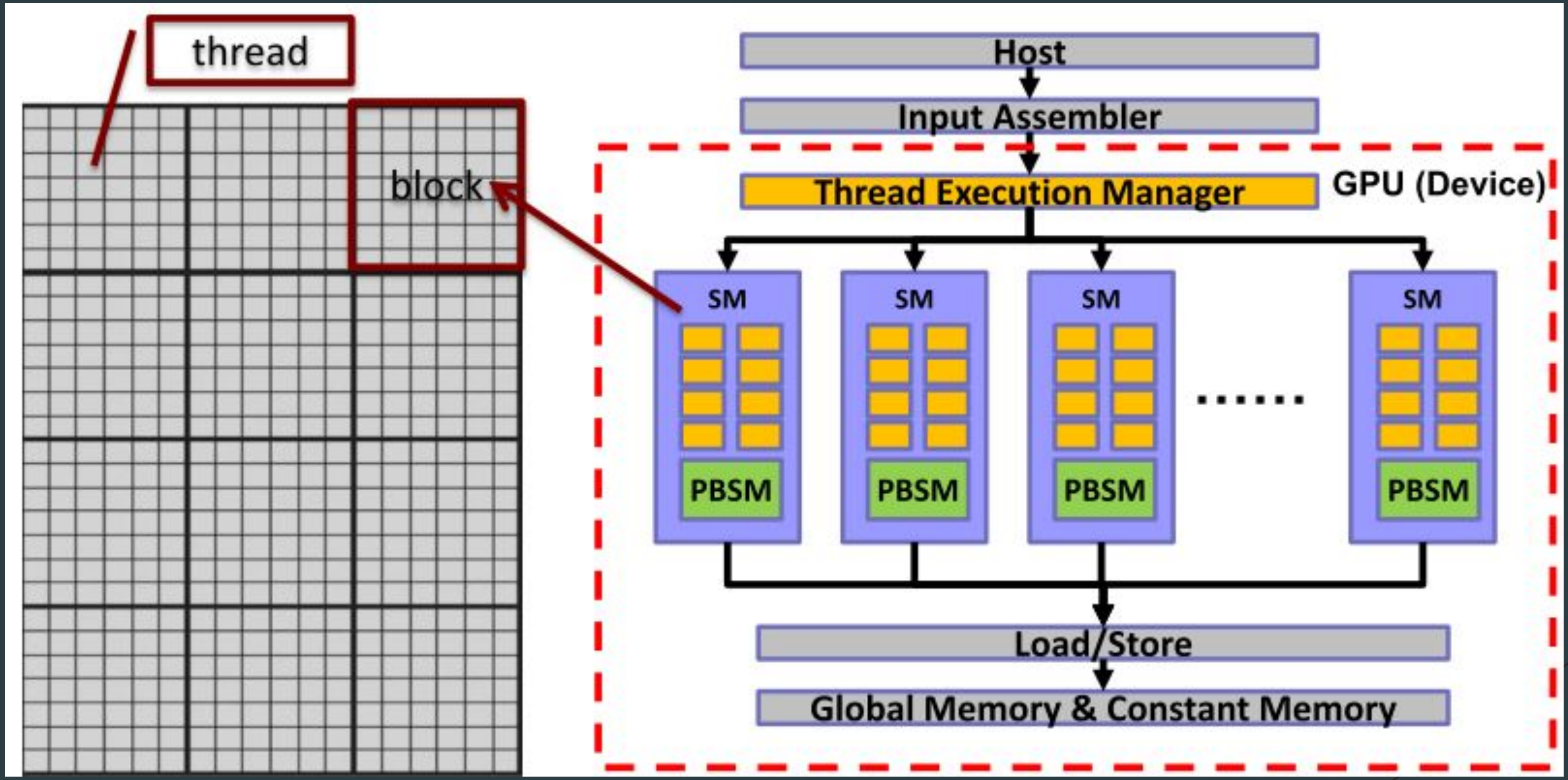
- One kernel is executed at a time on the device
- Many thread execute each kernel
  - Each thread executes the same code
  - … on the different data based on its threadID
- CUDA thread might be
  - Physical threads
    - As on NVIDIA GPUs
    - GPU thread creation and context switching are essentially free
  - Or virtual threads
    - E.g. 1 CPU core might execute multiple CUDA threads



```
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```

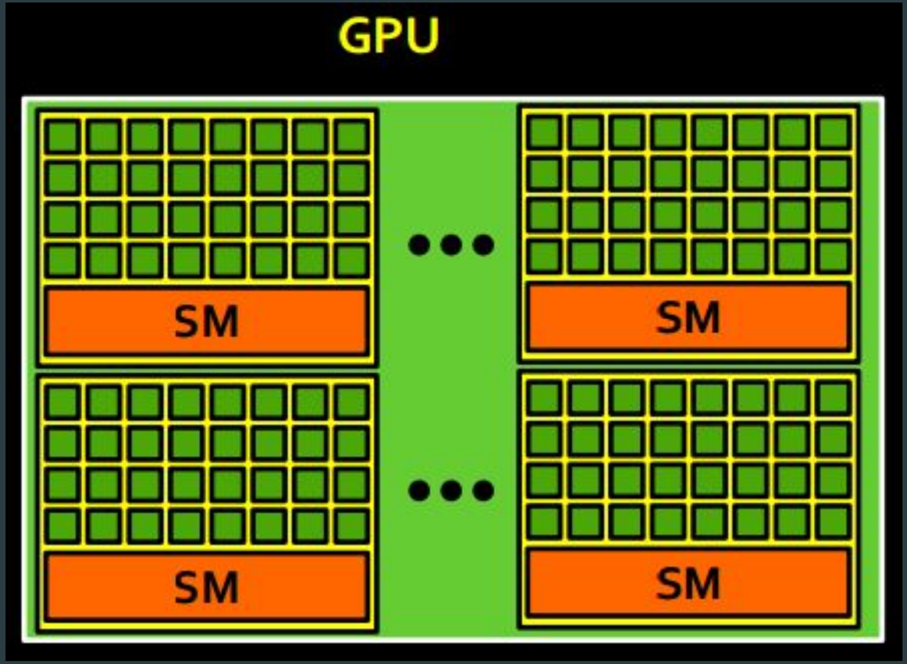
Software Mapping

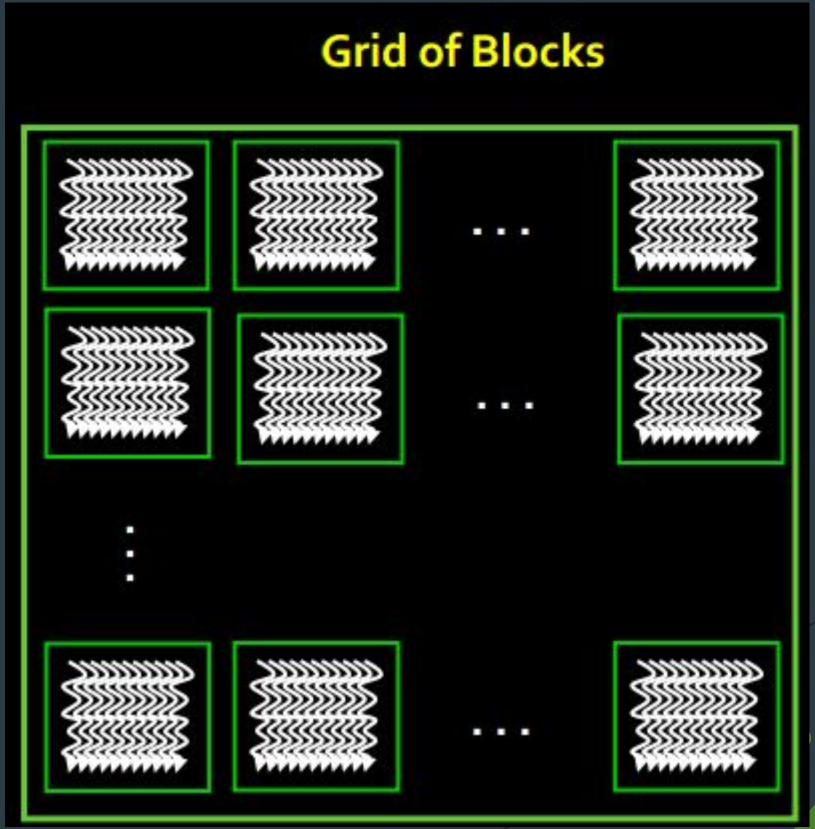
- Software: grid blocks threads
- Hardware: GPU(device) SM(Streaming Multiprocessor) SP(Streaming Processor), 又稱 CUDA core



Scheduling Thread Blocks

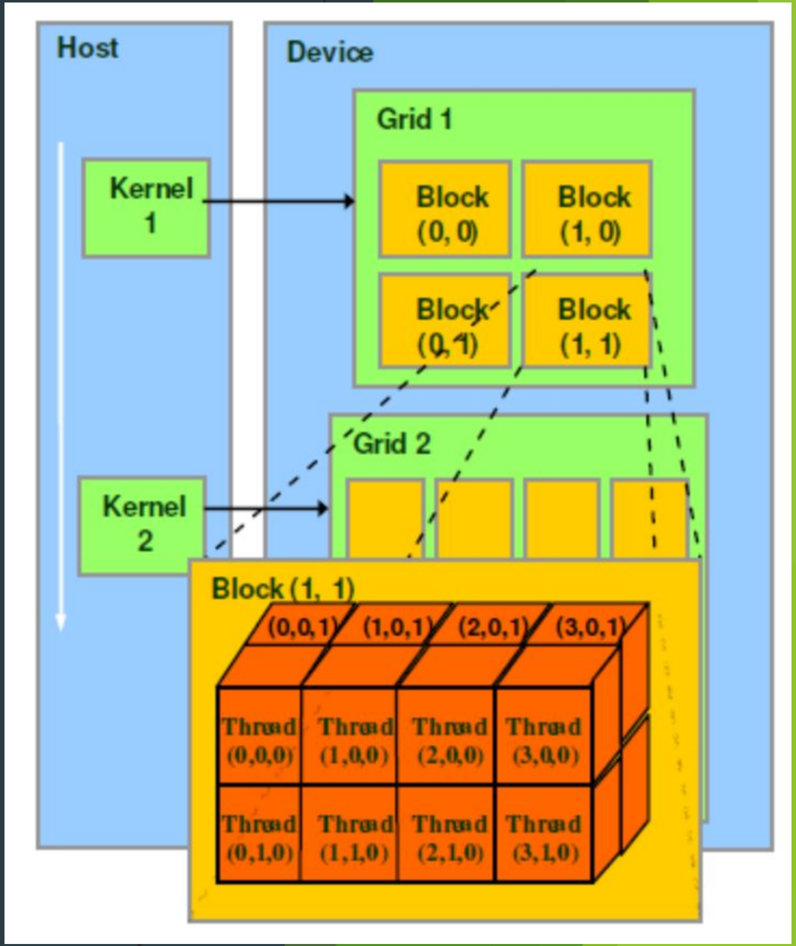
- Hardware dispatches thread blocks to available processor (streaming multiprocessor)





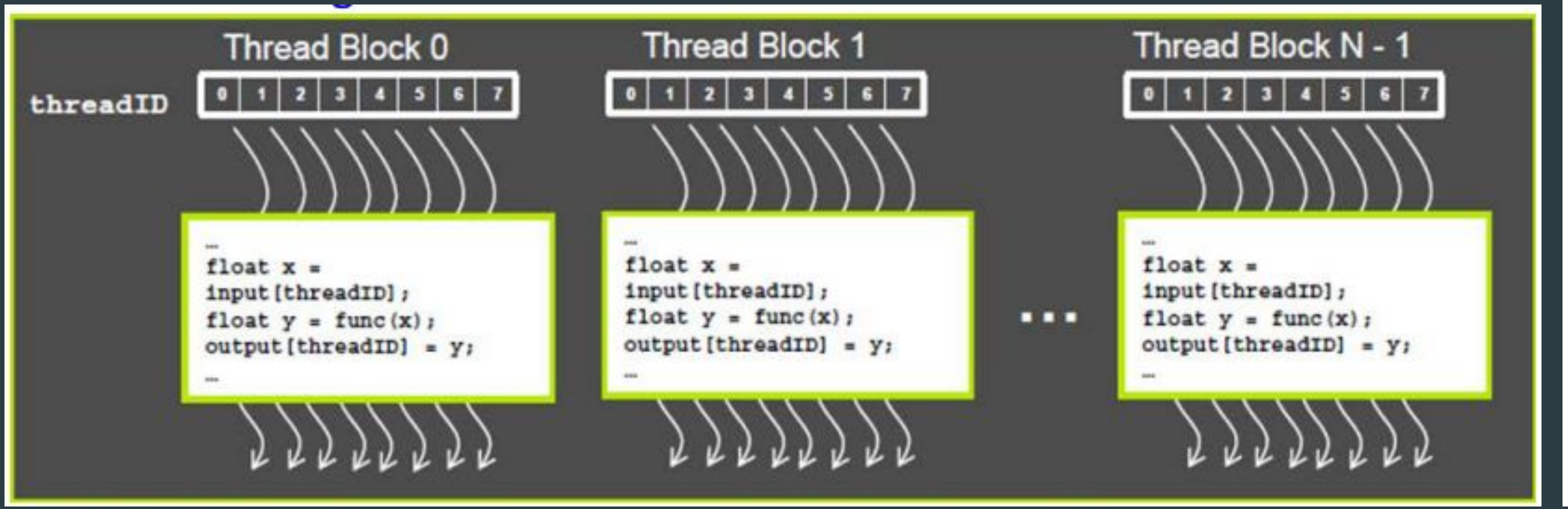
Software Concept in CUDA

- Thread：一個CUDA的並行程序會被分配到多個threads來執行
  - 一個SP(core)對應執行一個thread
  - threadId在cuda中為三維的概念，在一個block中由(x, y, z)來唯一定位
  - threadIdx.x, threadIdx.y, threadIdx.z
- Block：多個thread群組成一個block
  - 同一個block中的threads可以同步，也可以通過shared memory通信
  - blockId在cuda中為三維的概念，在一個grid中由(x, y,z)來唯一定位
  - blockIdx.x, blockIdx.y, blockIdx.z
- Grid：多個block構成grid
  - 一個grid對應一個執行一個kernel函數
- warp：執行任務的調度單元，同一時間一個SM只能運行一個warp
  - 類比於CPU中的進程，當一個warp阻塞時，SM將切換執行另一個warp
  - 當前CUDA的warp大小為32個線程。同在一個warp中的線程，以不同數據資源執行相同的指令，即所謂的SIMT（Single Instruction Multiple Thread）

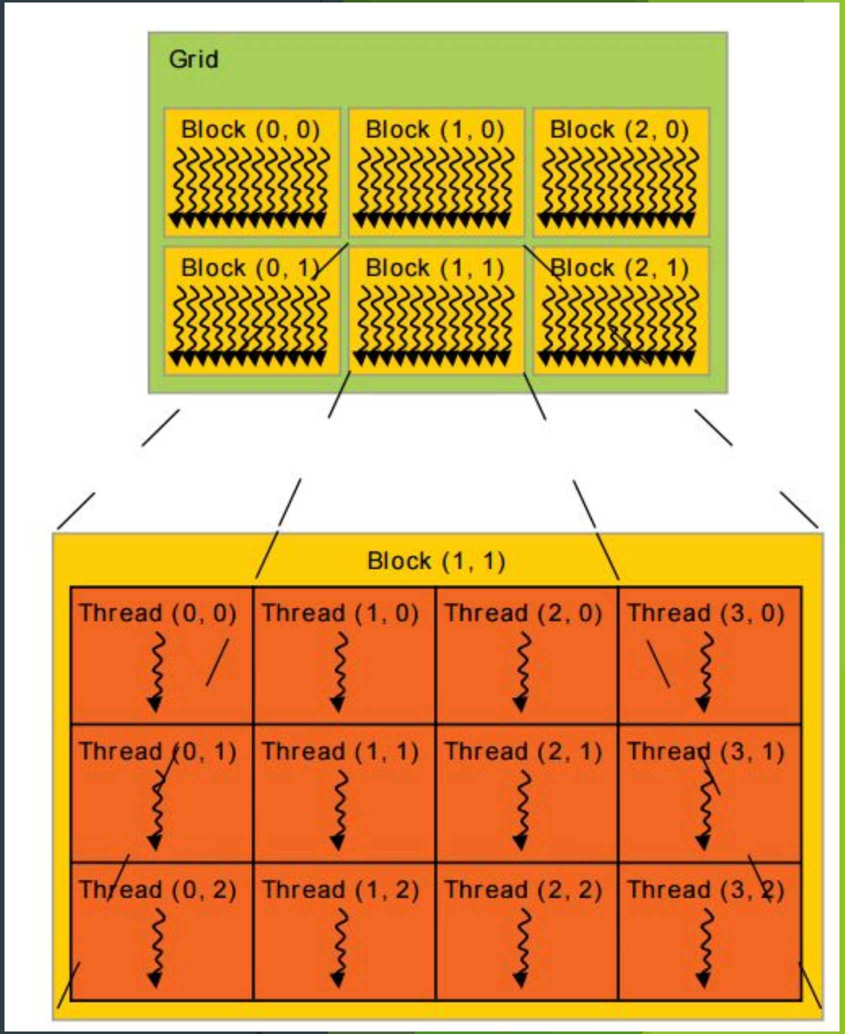


Thread Hierarchy

- Threads are grouped into thread blocks
  - Kernel = gird of thread blocks
- By definition, threads in the same block may synchronized with barriers, but not between blocks



```
…
float x =
input[threadID];
float y = func(x);
output[threadID] = y;
…
```



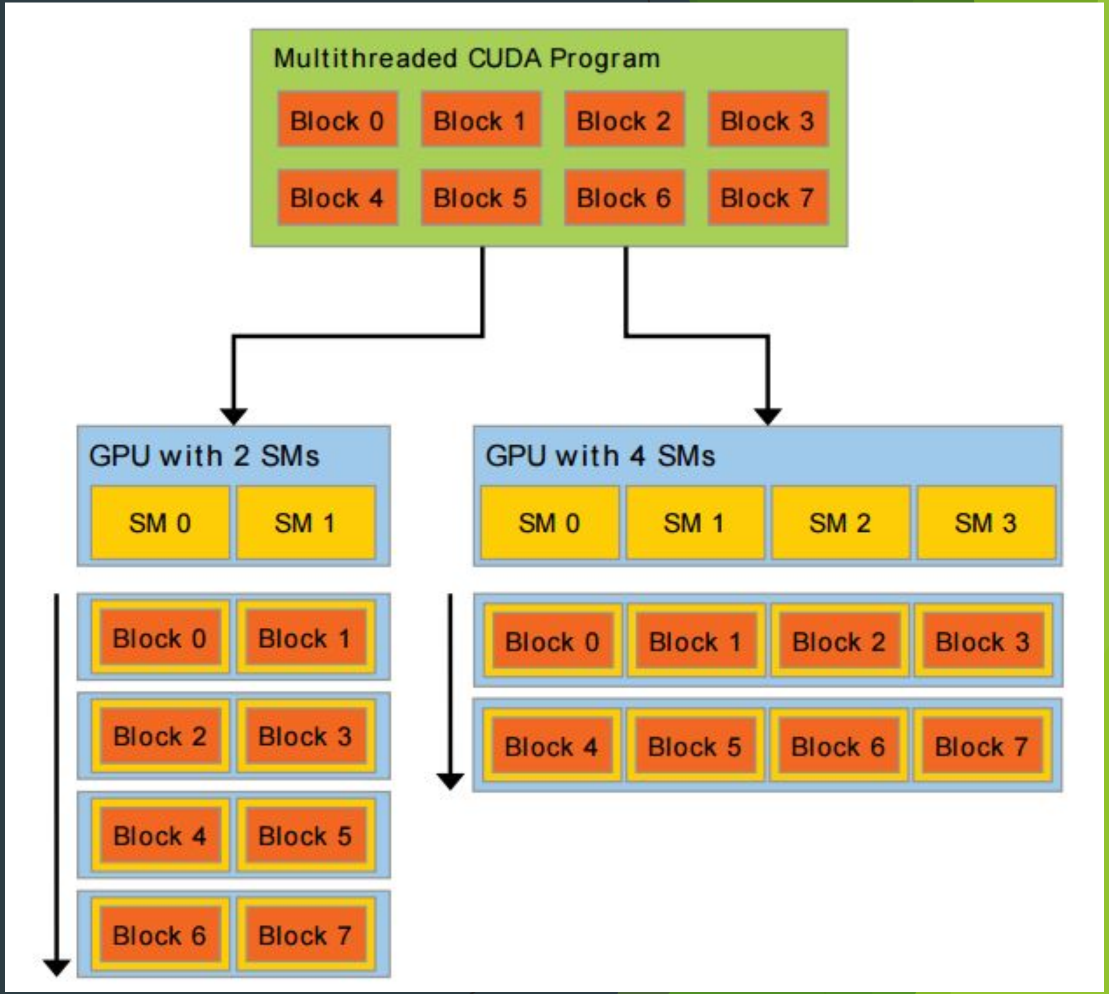
Scheduling Thread Blocks

- A GPU has lots of processors (streaming multiprocessors)
- Each processor (streaming multiprocessors) can execute multiple blocks concurrently
  - Programmers need to ensure that kernel launches creates enough thread blocks to keep machine busy
- Hardware dispatches a block when resources become available, typically when a previous block completes
  - No specific order in which blocks are dispatched and executed
  - Design algorithms to be insensitive to block execution order

Automatic Scalability

- Thread blocks cannot synchronize
  - So they can run in any order, concurrently or sequentially

- This independence gives scalability:
  - A kernel scales across any number of parallel cores



Some Restrictions First

- All threads in a grid execute the same kernel function
- A grid is organized as a 3D array of blocks (gridDim.x and gridDim.y, gridDim.z)
- Each block is organized as 3D array of threads (blockDim.x, blockDim.y, and blockDim.z)
- Once a kernel is launched, its dimensions cannot change.
- All blocks in a grid have the same dimension
- Once assigned to the SM, the block must be fully executed by the SM.

Thread group limits

- The maximum number of threads per block is limited
  - 512 before CUDA 2.0
  - 1024 after CUDA 2.0
- The maximum number of blocks is limited
  - 65535 before CUDA 3
  - $2^{31}-1$ after CUDA 3
- Total number of threads = threads per block * number of blocks

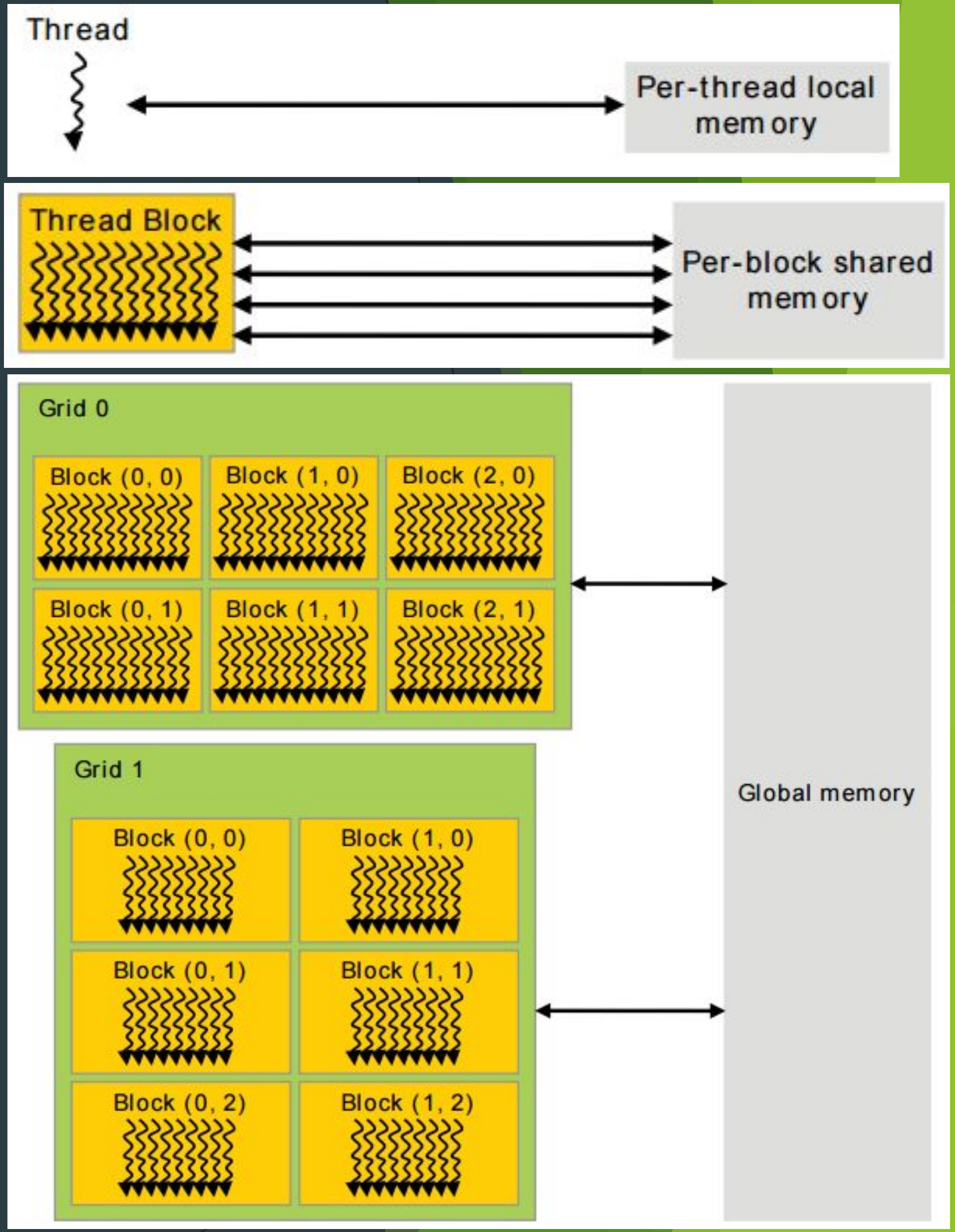
GPU parameters

```
Name:  GeForce GTX TITAN X
Compute capability:  5.2
Clock rate:  1076000
Device copy overlap:  Enabled
Kernel execution timeout :  Enabled
   --- Memory Information for device 0 ---
Total global mem:  12802785280
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 0 ---
Multiprocessor count:  24
Shared mem per mp:  49152
Registers per mp:  65536
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

```
Name:  GeForce GTX 480
Compute capability:  2.0
Clock rate:  1401000
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
   --- Memory Information for device 1 ---
Total global mem:  1545732096
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 1 ---
Multiprocessor count:  15
Shared mem per mp:  49152
Registers per mp:  32768
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (65535, 65535, 65535)
```

Memory Hierarchy

- **1. Registers**
  - **Scope:** Per-thread
  - **Lifetime:** Exists as long as the thread is alive
  - **Access Speed:** Fastest available memory
  - **Usage:** Registers are used to store local variables that are private to each thread. The compiler usually handles register allocation. Excessive usage of registers can lead to spilling to local memory, which degrades performance.
- **2. Shared Memory**
  - **Scope:** Per-block
  - **Lifetime:** Exists as long as the block is executing
  - **Access Speed:** Much faster than global memory but slower than registers
  - **Usage:** Shared memory is explicitly managed by the programmer. It is used to share data between threads within the same block, reducing redundant accesses to global memory and speeding up data interchange between threads.



Memory Hierarchy

- **3. Local Memory**
  - **Scope:** Per-thread
  - **Lifetime:** Exists as long as the thread is alive
  - **Access Speed:** As slow as global memory (essentially a part of global memory)
  - **Usage:** Used to store automatic variables when registers are exhausted. Although named 'local', it is not physically distinct from global memory but has similar access characteristics.
- **4. Global Memory**
  - **Scope:** All threads and host (CPU)
  - **Lifetime:** Exists as long as the application runs
  - **Access Speed:** Slowest among all types of memory on the device
  - **Usage:** Global memory is used to store data that needs to be accessible by all threads or to transfer data between the host and the device. Coalescing global memory accesses can significantly enhance performance.

Memory Hierarchy

- **5. Constant Memory**
  - **Scope:** All threads and host
  - **Lifetime:** Exists as long as the application runs
  - **Access Speed:** Cached and fast when accessed uniformly by threads
  - **Usage:** Optimized for situations where all threads read the same value or when reading relatively static data like coefficients or lookup tables.

- **6. Texture Memory**
  - **Scope:** All threads
  - **Lifetime:** Exists as long as the texture is bound in the application
  - **Access Speed:** Cached, optimized for 2D spatial locality
  - **Usage:** Used for scenarios where data access patterns exhibit spatial locality that can be efficiently cached. Common in graphics and image processing tasks.

Memory Access Patterns

- **Coalesced Accesses:** To achieve the best performance in global memory, accesses by threads within a warp should be coalesced. This means that consecutive threads (e.g., thread 0, 1, 2,...) should access consecutive memory addresses.

- **Bank Conflicts:** When multiple threads in a warp access shared memory, they should avoid simultaneous access to the same memory bank. Shared memory is divided into banks, and simultaneous access to the same bank by different threads causes bank conflicts, which serialize access and decrease performance.

Optimization Tips

- Minimize Data Transfer: Data transfer between host and device over the PCIe bus is slow. Minimize data transfer and maximize computation on the GPU.
- Utilize Shared Memory: Where possible, use shared memory to cache data needed by multiple threads within a block. Remember to synchronize threads with __syncthreads() after shared memory writes.
- Avoid Divergence: Ensure that threads within the same warp do not diverge significantly in their execution path, as this can lead to warp serialization where different paths are executed sequentially.

CUDA Language

- Philosophy: provide minimal set of extensions necessary
- Kernel launch
  - kernelFunc<<< nB, nT, nS, Sid >>>(…); // nS and Sid are optional
    - nB : number of blocks per grid (grid size)
    - nT : number of threads per block (block size)
    - nS : shared memory size (in bytes)
    - Sid : stream ID, default is 0
- Build-in device variables
  - threadIdx; blockIdx; blockDim; gridDim
- Intrinsic functions that expose operations in kernel code
  - __syncthreads();
- Declaration specifier to indicate where things live
  - __global__ void KernelFunc(…); // kernel function, run on device
  - __device__ void GlobalVar; // variable in device memory
  - __shared__ void SharedVar; // variable in per-block shared memory

Block IDs and Thread IDs

- Each block and thread uses IDs to decide what data to work on
  - `dim3 blockIdx;`
    - blockIdx.x, blockIdx.y, blockIdx.z
  - `dim3 threadIdx;`
    - threadIdx.x, threadIdx.y, threadIdx.z
  - `dim3 gridDim;`
    - gridDim.x, gridDim.y, gridDim.z
  - `dim3 blockDim;`
    - blockDim.x, blockDim.y, blockDim.z

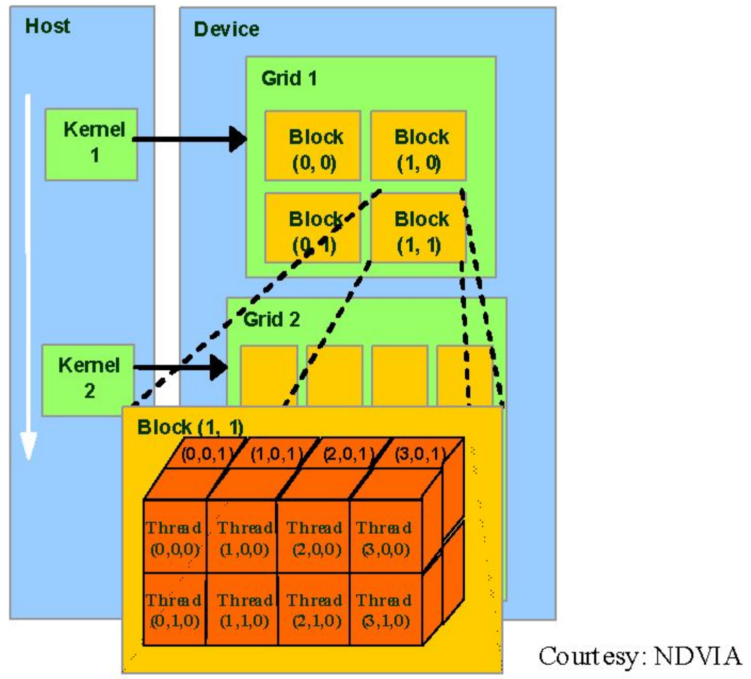


Figure 3.2. An Example of CUDA Thread Organization.

Practical Example

- Suppose you have a problem where you need to process a 2D data array of size 1024x1024. You could organize the threads, blocks, and grid as follows:
- Choose a block size of 16x16 threads. Thus, each block has 16×16=256 threads.
- To cover the entire data set with these blocks, you would need 1024/16=64 blocks in both the x and y dimensions.
- **CUDA Kernel Launch**

```
dim3 threadsPerBlock(16, 16);  // 16x16 threads in a block
dim3 numBlocks(64, 64);        // 64x64 blocks in a grid

yourKernel<<<numBlocks, threadsPerBlock>>>(params);
```

- **In this setup, yourKernel is executed by 64×64×256=1,048,576 threads in total.**

Considerations for Efficient Utilization

- **Maximize Occupancy**: To maximize hardware utilization, you need to maximize the occupancy of the GPU. This is achieved by having enough blocks to cover the latency of memory operations and computations.
- **Memory Coalescing**: Threads in a block can access memory more efficiently if their access patterns are aligned such that they result in coalesced memory accesses.
- **Shared Memory**: Using shared memory within blocks can reduce global memory bandwidth and speed up data access for threads within the same block.
- **Dimensioning Blocks and Grids**: Choosing the right size and shape for your blocks and grids is crucial. It often depends on:
  - The size of the data set
  - The memory usage per thread
  - The GPU architecture (e.g., maximum threads per block, blocks per SM, etc.)

Function Qualifiers

Function qualifiers	limitations
__device__ function	Executed on the device Callable from the device only
__global__ function	Executed on the device Callable from the host only (must have void return type!)
__host__ function	Executed on the host Callable from the host only
Functions without qualifiers	Compiled for the host only
__host__ __device__ function	Compiled for both the host and the device

Variable Type Qualifiers

Variable qualifiers	limitations
__device__ var	• Resides in **device's global memory space** • Has the lifetime of an application • Is accessible from all the threads within the grid and from the host through the runtime library
__constant__ var	• Resides in device's **constant memory space**
__shared__ var	• Resides in the **shared memory** space of a thread block • Has the **lifetime of the block** • Is only accessible from all the threads within the block

CUDA Runtime API

- Device management
  - cudaGetDeviceCount(), cudaGetDeviceProperties()
- Device memory management
  - cudaMalloc(), cudaFree(), cudaMemcpy()
- Graphic interoprability
  - cudaGLMapBufferObject(), cudaD3DMapResources()
- Texture management
  - cudaBindTexture(), cudaBindTextureToArray()

Device memory operations

- Three functions:
- cudaMalloc(), cudaFree(), cudaMemcpy()
- Similar to the C’s malloc(), free(), memcpy()

- cudaMalloc(void **devPtr, size_t size)
  - devPtr : return the address of the allocated device memory
  - size : the allocated memory size (bytes)

- cudaFree (void *devPtr)

- cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)
  - count : size in bytes to copy

cudaMemcpyKind

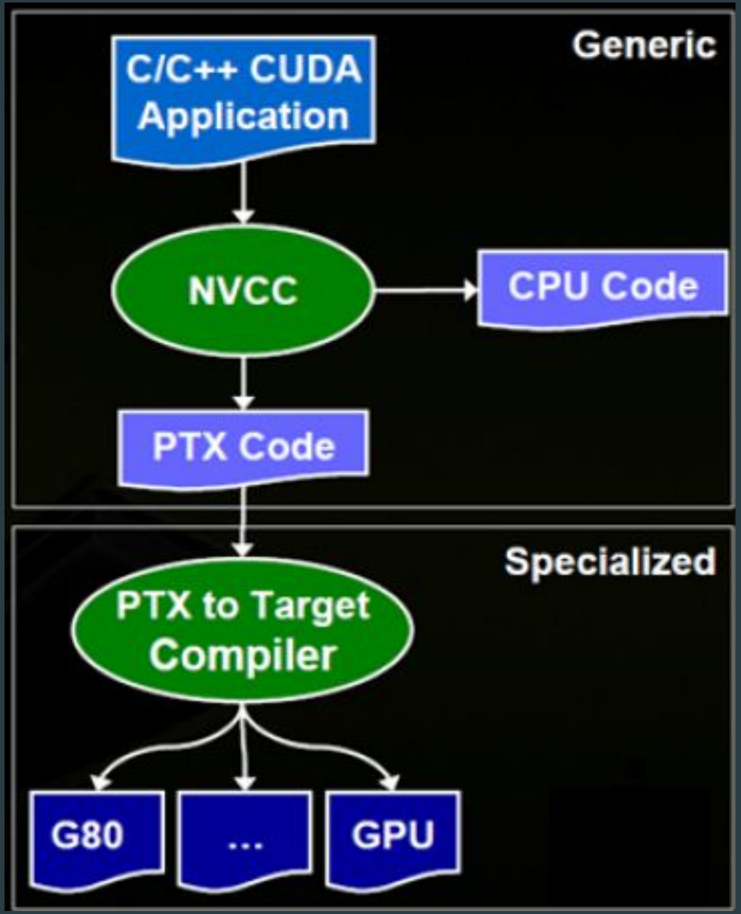
- one of the following four values

cudaMemcpyKind	Meaning	dst	src
cudaMemcpyHostToHost	Host ¤ Host	host	host
cudaMemcpyHostToDevice	Host ¤ Device	Device	Host
cudaMemcpyDeviceToHost	Device ¤ Host	Host	Device
cudaMemcpyDeviceToDevice	Device ¤ Device	Device	Device

host to host has the same effect as memcpy()

Program Compilation

- Any source file containing CUDA language must be compiled with NVCC
  - NVCC separates code running on the host from code running on the device
- Two-stage complication:
  - Virtual ISA
    - PTX: Parallel Threads eXecutions
  - Device-specific binary object



Example 1: Hello World!

- Two new syntactic elements…
  - 1. __global__ indicates a function that runs on the device and is called from host code
  - 2. mykernel<<<1,1>>>();
  - Triple angle brackets mark a call from host code to device code, which is called a “kernel launch”.

```
__global__ void mykernel(void) {
}

int main(void) {
  mykernel<<<1,1>>>();
  printf("Hello World!\n");
  return 0;
}
```

Example 2: add 2 numbers

- This does not work!!
- int ha, hb, hc are in the host memory (DRAM), which cannot be used by device (GPU).
- We need to allocate variables in “device memory”.

```
__global__ void add(int *a, int *b, int *c) {
   *c = *a + *b;
}

int main(void) {
   int ha=1,hb=2,hc;
   add<<<1,1>>>(&ha, &hb, &hc);
   printf(“c=%d\n”,hc);
   return 0;
}
```

X

The correct main()

```
int main(void) {
   int a=1, b=2, c; // host copies of a, b, c
   int *d_a, *d_b, *d_c; // device copies of a, b, c
   // Allocate space for device copies of a, b, c
   cudaMalloc((void **)&d_a, sizeof(int));
   cudaMalloc((void **)&d_b, sizeof(int));
   cudaMalloc((void **)&d_c, sizeof(int));
   // Copy inputs to device
   cudaMemcpy(d_a,&a,sizeof(int),cudaMemcpyHostToDevice);
   cudaMemcpy(d_b,&b,sizeof(int),cudaMemcpyHostToDevice);
   // Launch add() kernel on GPU
   add<<<1,1>>>(d_a, d_b, d_c);
   // Copy result back to host
   cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
   // Cleanup
   cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
   return 0;
}
```

✓

Example 3: add 2 vectors

- Let's first look at the sequential code!

```
// function definition
void VecAdd(int N, float* A, float* B, float* C)
{
    for(int i = 0; i<N; i++)
        C[i] = A[i] + B[i];
}

int main()
{ ...
   VecAdd (N, Ah, Bh, Ch);
  ...
}
```

Parallel CUDA code

- Use threadIdx.x as the index of the arrays
- Each thread processes 1 addition, for the elements indexed at threadIdx.x .

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

int main()
{ ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(Ah, Bh, Ch); ...
}
```

What is the limit of N?

N=4 case

- Four threads can be executed in parallel.
- Each thread has a unique threadIdx , starting from 0.
- threadIdx is a build-in variable, which is a structure of 3 members: x , y , and z .
  - struct unit3 { x; y; z; };
  - struct dim3 { x; y; z; };

Thread 0

c[0] = a[0] + b[0]

Thread 1

c[1] = a[1] + b[1]

Thread 2

c[2] = a[2] + b[2]

Thread 3

c[3] = a[3] + b[3]

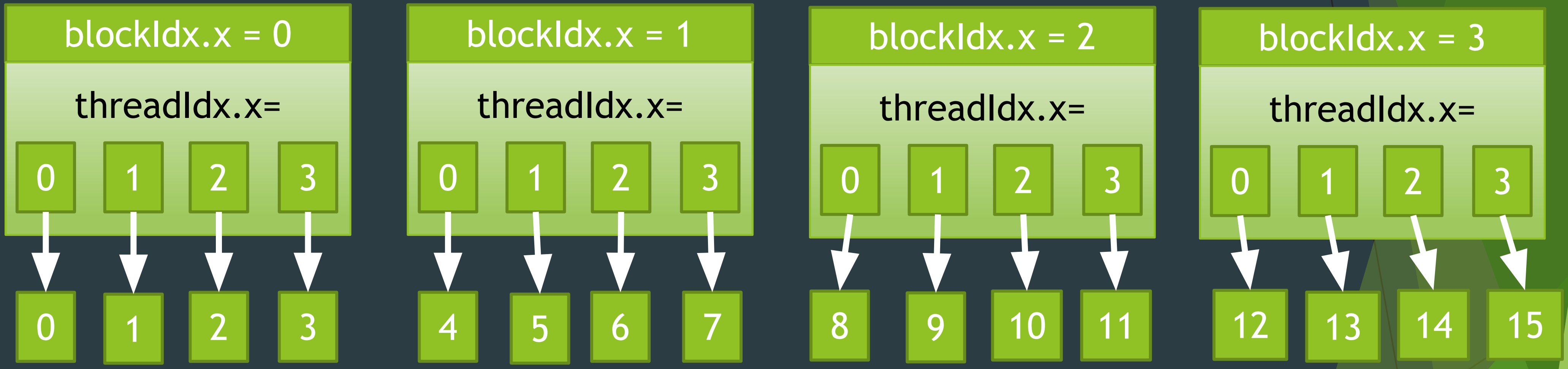
Alternative implementation

- Using parallel thread instead
- N blocks and each block has 1 thread.
- Which one is better?
  - Threads in the same block can communicate, synchronize with others, but the number of threads per block is limited.

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

int main(void) {
    int a[N], b[N], c[N];
    int *d_a, *d_b, *d_c;
    …
    add<<< N, 1 >>>(d_a, d_b, d_c);
    …
}
```

3rd implementation

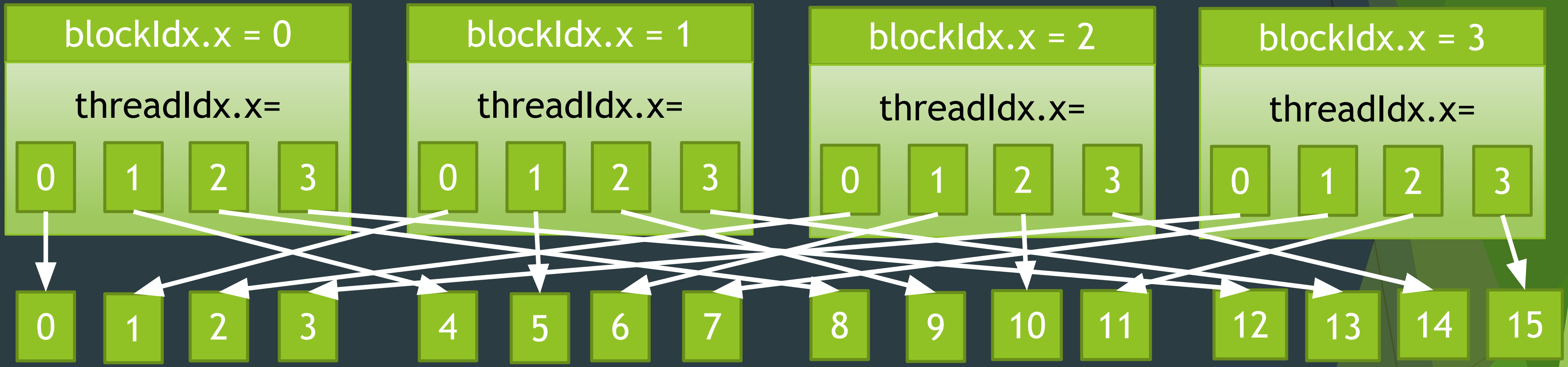
- Using multiple threads and multiple blocks
- Suppose N=16, grid size = 4, and block size = 4



- How to index 16 elements of an array?
  - Method 1: index = blockIdx.x*4+threadIdx.x
  - Method 2: index = threadIdx.x*4+blockIdx.x
- Which one is better?

3rd implementation

- Using multiple threads and multiple blocks
- Suppose N=16, grid size = 4, and block size = 4



- How to index 16 elements of an array?
  - Method 1: index = blockIdx.x*4+threadIdx.x
  - Method 2: index = threadIdx.x*4+blockIdx.x
- Which one is better? memory coalesced

The general case

- Use the built-in variable blockDim.x for threads per block.
- BS is block size (number of threads per block)

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

int main(void) {
   int a[N], b[N], c[N];
   int *d_a, *d_b, *d_c;
   ...
   add<<< N/BS,BS>>>(d_a, d_b, d_c);
   ...
}
```

What if N is not a multiple of BS ?

A even more general case

- The kernel function can have branches, but with a price to pay…

```
__global__ void add(int *a, int *b, int *c, int n) {
   int index = threadIdx.x + blockIdx.x * blockDim.x;
   if (index < n)
      c[index] = a[index] + b[index];
}

int main(void) {
   int a[N], b[N], c[N];
   int *d_a, *d_b, *d_c;
   …
   add<<< (N+BS-1)/BS, BS>>>(d_a, d_b, d_c, N);
   …
}
```

Execution time: in host

- CUDA provides functions to measure the execution time between events.

  cudaError_t cudaEventElapsedTime(float* ms, cudaEvent_t start, cudaEvent_t end)

  - ms: time between start and end in ms
  - start: starting event
  - end: ending event
- The time unit is milliseconds, whose resolution is 0.5 microseconds

CUDA event

- Data type : cudaEvent_t
- cudaError_t cudaEventCreate(cudaEvent_t* event)
  - Create CUDA event
- cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream = 0)
  - Record CUDA event
  - If stream is non-zero, the event is recorded after all preceding operations in the stream have been completed
  - Since operation is asynchronous, cudaEventQuery() and/or cudaEventSynchronize() must be used to determine when the event has actually been recorded
- cudaError_t cudaEventSynchronize(cudaEvent_t event)
  - Wait until the completion of all device work preceding the most recent call to cudaEventRecord()

Example

```
1. cudaEvent_t start, stop;
2. cudaEventCreate(&start);
3. cudaEventCreate(&stop);
4. cudaEventRecord(start);
5. kernel<<<block, thread>>>();
6. cudaEventRecord(stop);
7. cudaEventSynchronize(stop);
8. float time;
9. cudaEventElapsedTime(&time, start, stop);
```

Device information

- Query device information by

  cudaGetDeviceProperties(struct cudaDeviceProp *prop, int device)

  - cudaDeviceProp is a structure, which specifies properties for the queried device
    - char name[256];
    - int maxThreadsPerBlock;
    - int maxThreadsDim[3];
    - int maxGridSize[3];
    - int clockRate;
    - ...
  - Device is the device number, specifying which device's information is queried (used in multi-GPU environment)

Host and device sync

- Kernel launches are asynchronous
  - Control returns to the CPU immediately
  - CPU needs to synchronize before consuming the results.
  - cudaDeviceSynchronize() blocks the CPU until all preceding CUDA calls have completed

- cudaMemcpy() blocks the CPU until the copy is completed.
  - Copy begins when all preceding CUDA calls have completed
  - Using cudaMemcpyAsync() to perform asynchronous memory copy, which does not block the CPU

Reporting errors

- All CUDA calls return an error code cudaError_t
  - Error in the API call itself OR Error in an earlier asynchronous operation (e.g. kernel)

- Get the error code for the last error:
  - cudaError_t cudaGetLastError(void)

- Get a string to describe the error:
  - char *cudaGetErrorString(cudaError_t)

Device Management

- Application can query and select GPUs
  - cudaGetDeviceCount(int *count)
  - cudaSetDevice(int device)
  - cudaGetDevice(int *device)
  - cudaGetDeviceProperties(cudaDeviceProp *prop, int device)

- Multiple CPU threads can share a device

- A single CPU thread can manage multiple devices
  - cudaSetDevice(i) to select current device
  - cudaMemcpy(…) for peer-to-peer copies

Occupancy

- Occupancy: number of concurrent threads per SM
  - Expressed as either:
    - the number of threads (or warps)
    - percentage of maximum threads
- Determined by several factors
  - (refer to Occupancy Calculator, CUDA Programming Guide for full details)
  - Registers per thread
    - SM registers are partitioned among the threads
  - Shared memory per block
    - SM shared memory is partitioned among the blocks
  - Threads per block
    - Threads are allocated at threadblock granularity

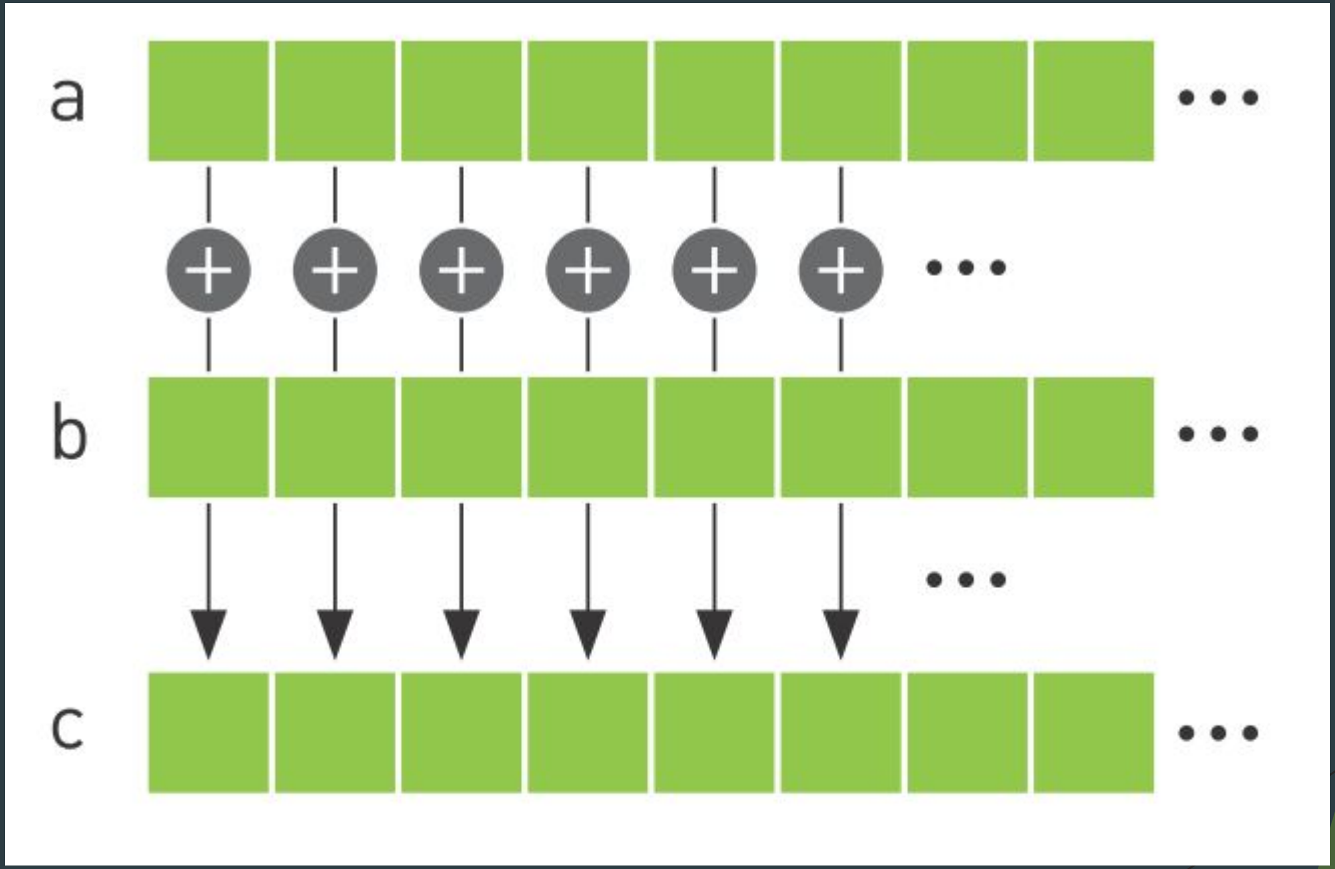
Occupancy and Performance

- Note that 100% occupancy isn’t needed to reach maximum performance
  - Sufficient occupancy to hide latency, higher occupancy will not improve performance

- “Sufficient” occupancy depends on the code
  - More independent work per thread → less occupancy is needed
  - Memory-bound codes tend to need higher occupancy
    - Higher latency (than for arithmetic) needs more work

LAB 1: Vector Addition

Vector Addition



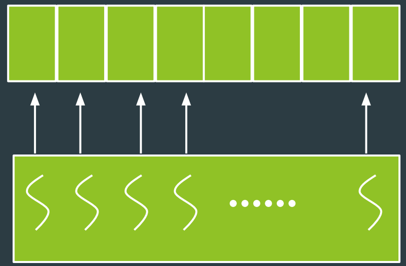
CUDA Built-in Variables

- `dim3 gridDim;`
  - Dimensions of the grid in blocks
  - gridDim.x, gridDim.y, gridDim.z
- `dim3 blockDim;`
  - Dimensions of the block in threads
  - blockDim.x, blockDim.y, blockDim.z
- `dim3 blockIdx;`
  - Block index within the grid
  - blockIdx.x, blockIdx.y, blockIdx.z
- `dim3 threadIdx;`
  - Thread index within the block
  - threadIdx.x, threadIdx.y, threadIdx.z

Single Block Multiple Threads

- Terminology: a block can be split into parallel threads
- add<<<1, N>>>(dev _ a, dev _ b, dev _ c);

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



Thread 0

c[0] = a[0] + b[0];

Thread 1

c[1] = a[1] + b[1];

Thread 2

c[2] = a[2] + b[2];

Thread 3

c[3] = a[3] + b[3];

Vector addition using one block with multiple threads

```
#include <stdio.h>
#define N 256

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

int main( void ) {
    int *a, *b, *c;
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the CPU
    a = (int*)malloc( N * sizeof(int) );
    b = (int*)malloc( N * sizeof(int) );
    c = (int*)malloc( N * sizeof(int) );

    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );
```

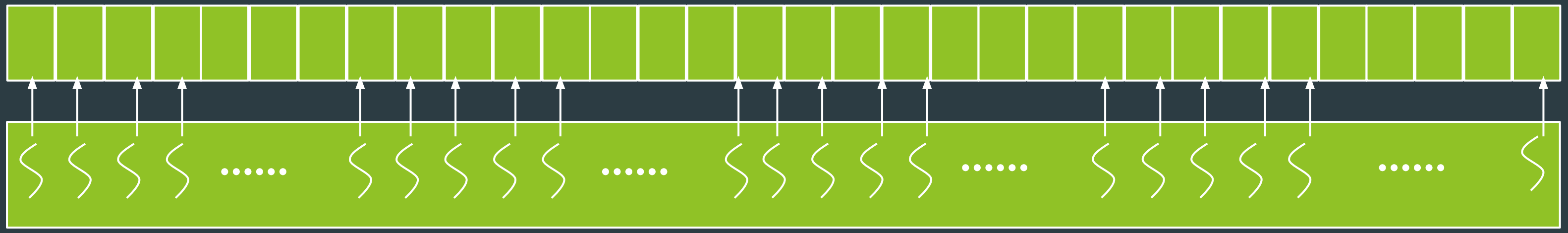
```
// fill the arrays 'a' and 'b' on the CPU
srand ( time(NULL) );
for (int i=0; i<N; i++) {
    a[i] = rand()%256;
    b[i] = rand()%256;
}
// copy the arrays 'a' and 'b' to the GPU
cudaMemcpy(dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, N * sizeof(int),cudaMemcpyHostToDevice);

add<<<1,1024>>>( dev_a, dev_b, dev_c );

// copy the array 'c' back from the GPU to the CPU
cudaMemcpy( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost);
// verify that the GPU did the work we requested
 bool success = true;
 for (int i=0; i<N; i++) {
     if ((a[i] + b[i]) != c[i]) {
         printf( "Error:  %d + %d != %d\n", a[i], b[i], c[i] );
         success = false;
     }
 }
 if (success)    printf( "We did it!\n" );
```

```
    // free the memory allocated on the GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );

    return 0;
}
```

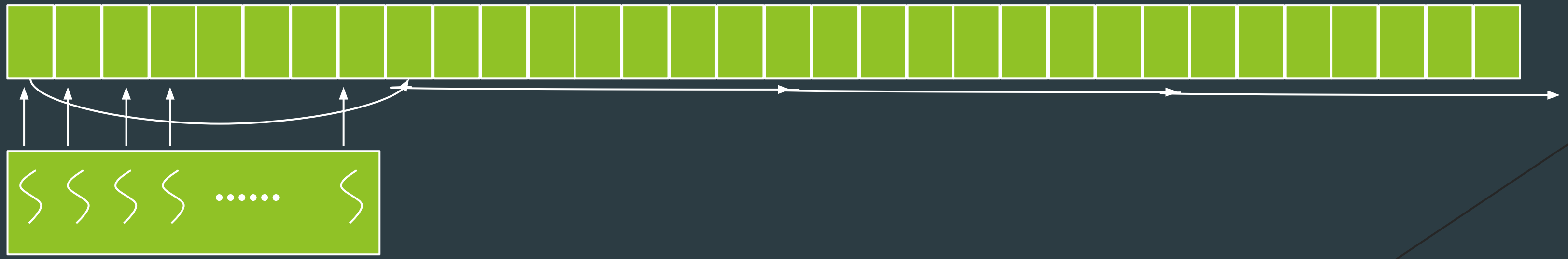
- Each thread handles one position.
- The limits of the number of threads per block?
- How to handle large amount of data?

Problem

- N is changed to 1024x1024
- Increase the number of threads per block???



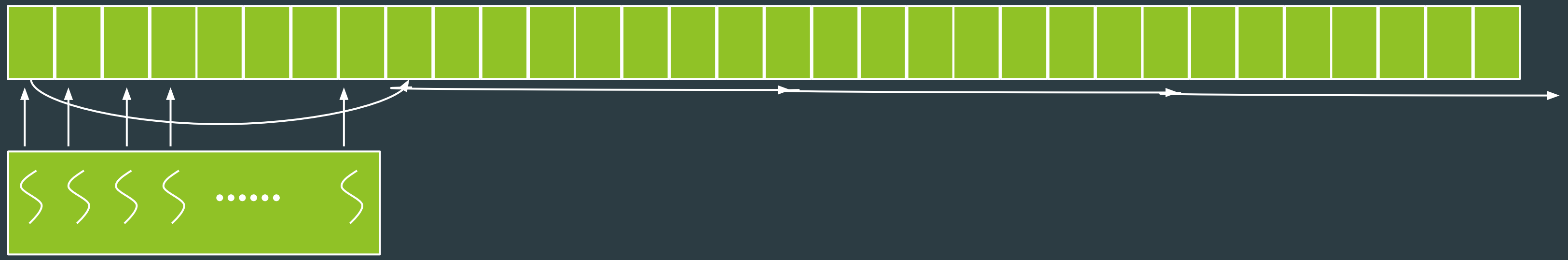
- What is the limitation of threads per blocks???
- Each thread must handle more than one position.



ANS

- add<<<1, N>>>(dev _ a, dev _ b, dev _ c);

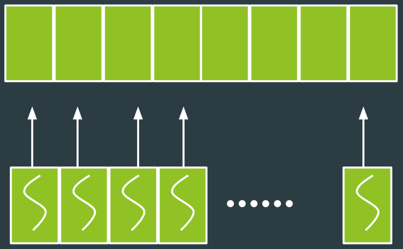
```
__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x;
    while (tid < N)
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x;
}
```



Multiple Blocks Single Thread

- add<<<N,1>>>(dev_a, dev_b, dev_c);
- N blocks x 1 thread/block = N parallel threads

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



Block 0

c[0] = a[0] + b[0];

Block 1

c[1] = a[1] + b[1];

Block 2

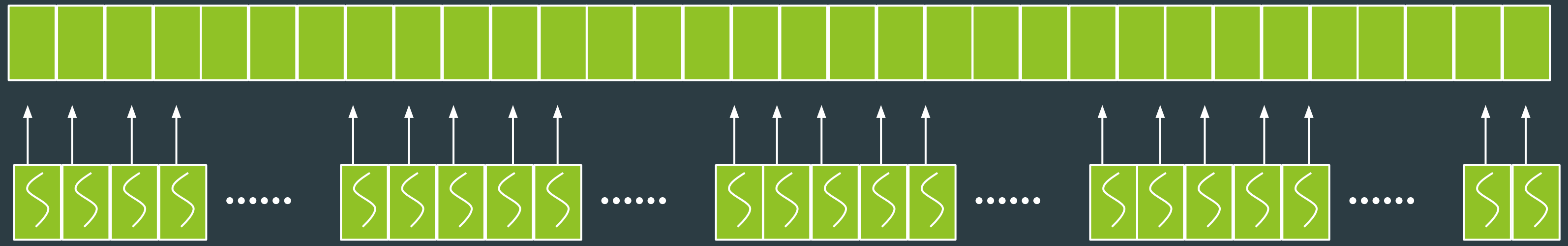
c[2] = a[2] + b[2];

Block 3

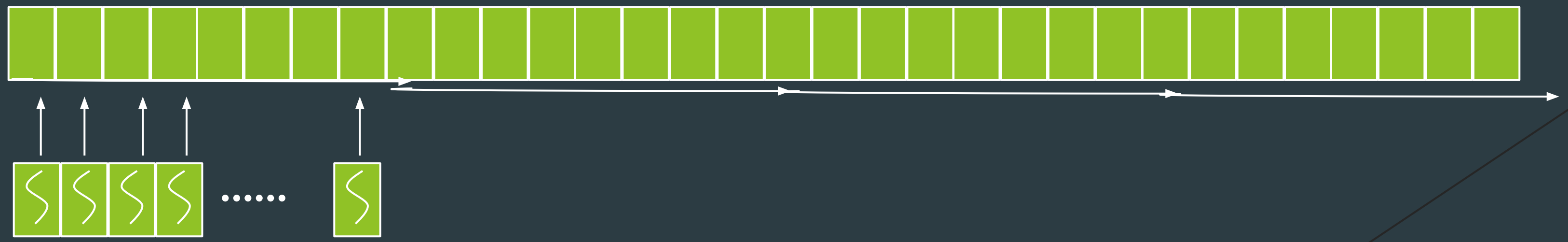
c[3] = a[3] + b[3];

Problem

- N is changed to 1024x1024
- Increase the number of blocks



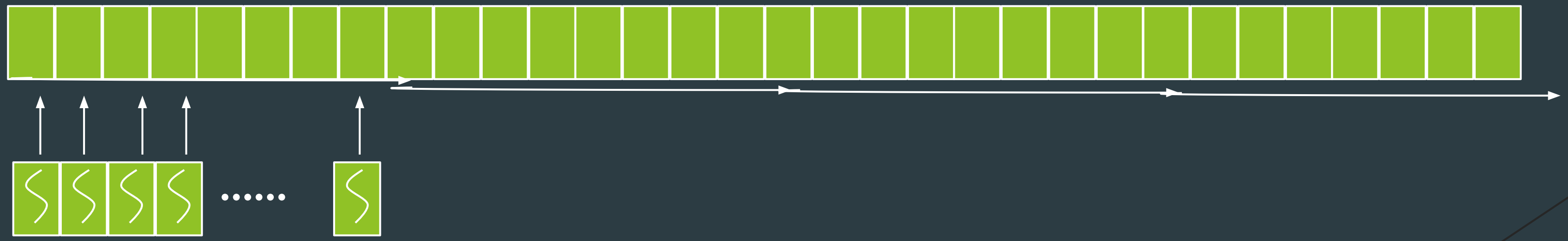
- What is the limitation of blocks???
- Each block must handle more than one position.



ANS

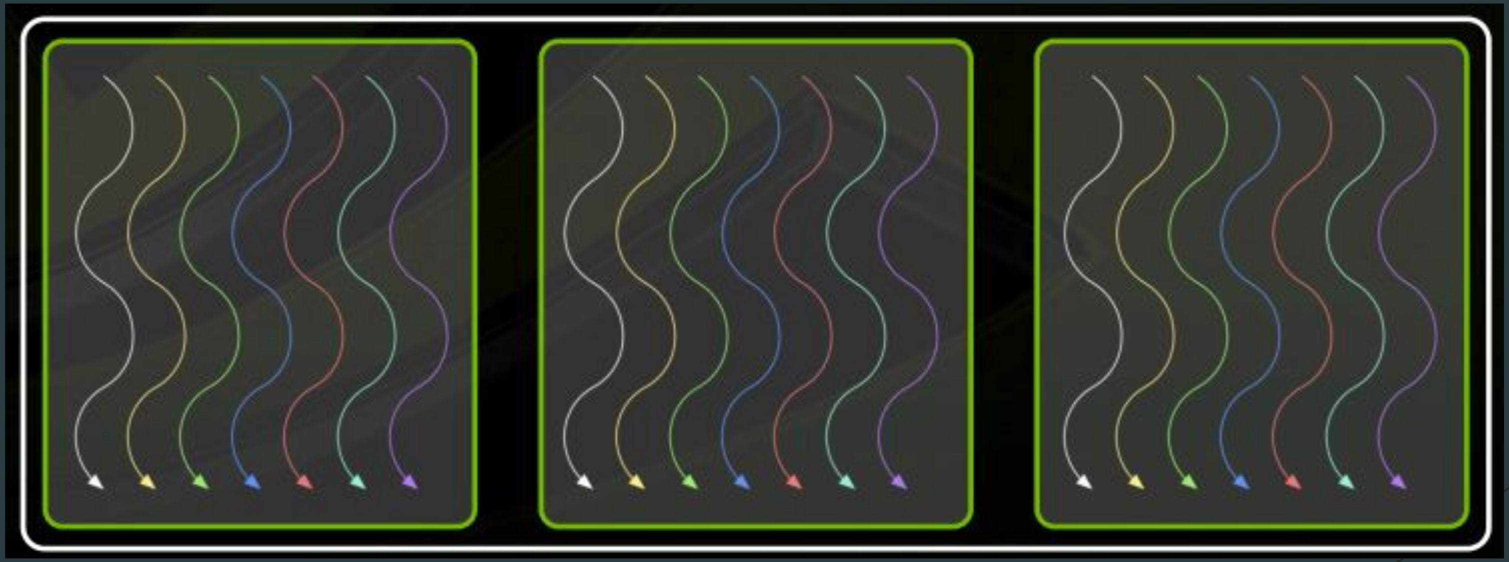
- add<<<N, 1>>>(dev_a, dev_b, dev_c);
- N blocks x 1 thread/block = N parallel threads

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += gridDim.x;//The number of blocks in x direction
    }
}
```



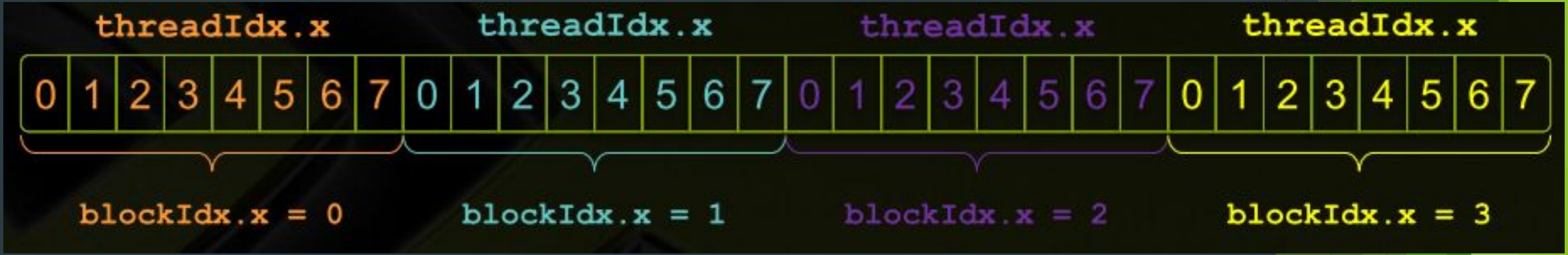
Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Let's adapt vector addition to use both blocks and threads



Indexing Arrays with Blocks and Threads

- No longer as simple as using blockIdx.x and threadIdx.x
  - Consider indexing an array with one element per thread (8 threads/block)

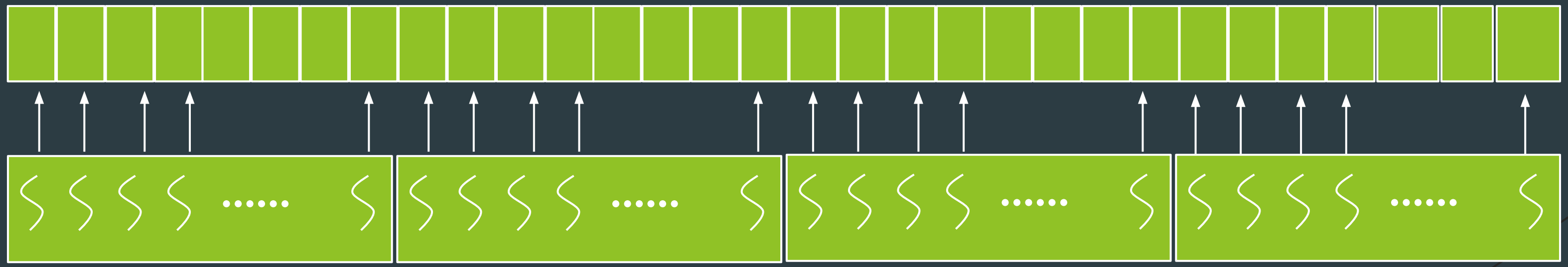


- A unique index for each thread is given by:
  - int index = threadIdx.x + blockIdx.x * blockDim.x;

Multiple Blocks Multiple Threads

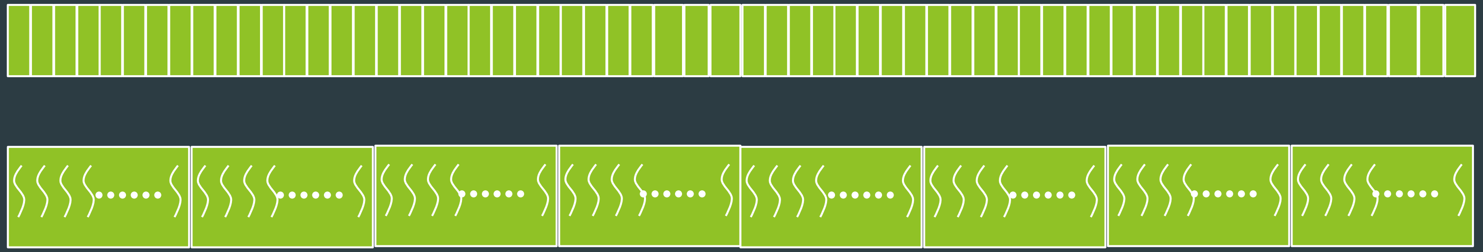
- add<<< 1024/32, 32>>>(dev _ a, dev _ b, dev _ c);
- int tid = threadIdx.x + blockIdx.x * blockDim.x;

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

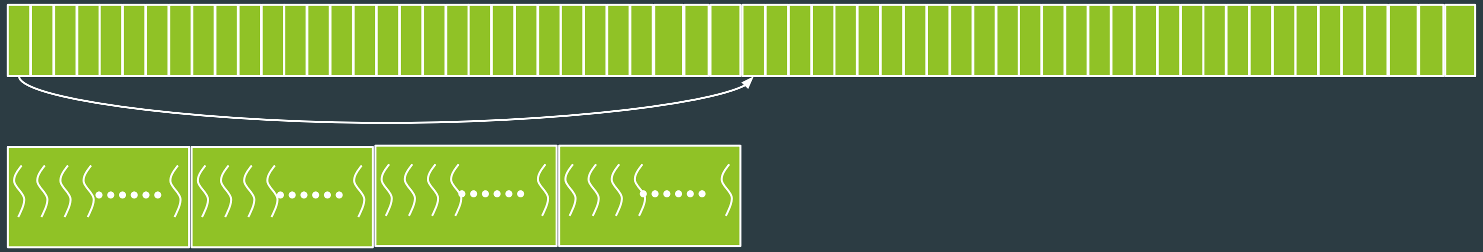


Problem

- N is changed to 1024x1024
- Increase the number of blocks



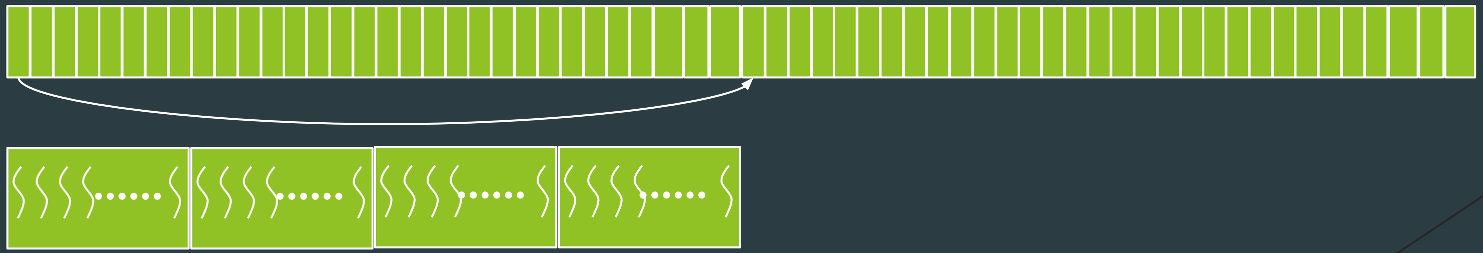
- What is the limitation of blocks???
- Each thread must handle more than one position.



ANS

- unsigned int threadsPerBlock = 32; //最大為1024
- unsigned int blocksPerGrid = (N + threadsPerBlock - 1)/threadsPerBlock;
- add <<< blocksPerGrid, threadsPerBlock >>> (dev_a, dev_b, dev_c);

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += gridDim.x * blockDim.x;
    }
}
```



Note

- blocksPerGrid cannot exceed the maximum gridDim.x
- 因此total最大的thread數量將會是2147483647 x 1024
- 如果N大於2147483647 x 1024 ???

Time counting on Linux

```
struct timespec t_start, t_end;
double elapsedTime;

// start time
clock_gettime( CLOCK_REALTIME, &t_start);

// do something

// stop time
clock_gettime( CLOCK_REALTIME, &t_end);

// compute and print the elapsed time in millisec
elapsedTime = (t_end.tv_sec - t_start.tv_sec) * 1000.0;
elapsedTime += (t_end.tv_nsec - t_start.tv_nsec) / 1000000.0;
printf("elapsedTime: %lf ms\n", elapsedTime);
```

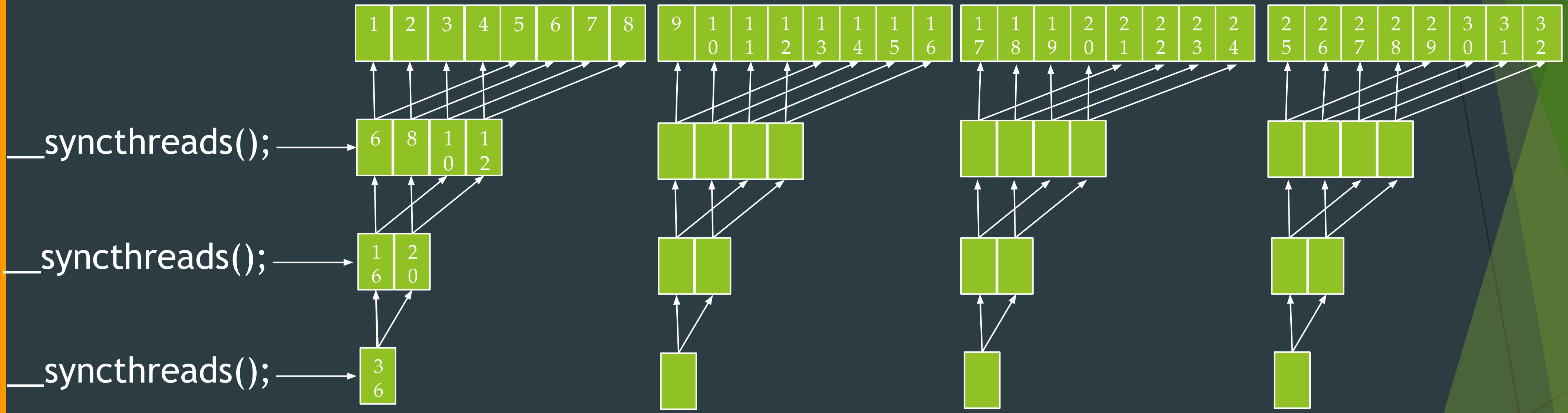
```
Linux:// compile with -lrt
#include <time.h>
```

GPU time counting

```
// Get start time event
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

// Invoke kernel here

// Get stop time event
    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);
 // Compute execution time
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, start, stop);
    printf("GPU time: %13f msec\n", elapsedTime);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
```

Lab2: Reduction Addition

Multiple blocks with multiple threads



```
#include <cuda_runtime.h>
#include <iostream>

// Kernel for reducing an array using global memory
__global__ void reduceSumGlobal(int *input, int *output, int n) {
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Grid stride loop
    for (int stride = blockDim.x/2; stride>0; stride = stride / 2) {
        if(i <  blockIdx.x * blockDim.x + stride)
            input[i] += input[i + stride];
        __syncthreads();
    }

    // Let the thread 0 for each block write its result to the output array
    if (tid == 0) {
        output[blockIdx.x] = input[blockIdx.x * blockDim.x];
    }
}
```

```
int main() {
    const int size = 1024;
    const int byteSize = size * sizeof(int);
    int h_input[size];
    int *d_input, *d_output;

    // Initialize the host input array
    for (int i = 0; i < size; i++) {
        h_input[i] = 1; // All elements are 1 for simplicity
    }

    cudaMalloc(&d_input, byteSize);
    cudaMalloc(&d_output, sizeof(int) * ((size + 255) / 256)); // Assuming block size of 256
    cudaMemcpy(d_input, h_input, byteSize, cudaMemcpyHostToDevice);

    dim3 block(256);
    dim3 grid((size + block.x - 1) / block.x);

    reduceSumGlobal<<<grid, block>>>(d_input, d_output, size);

    int h_output[grid.x];
    cudaMemcpy(h_output, d_output, grid.x * sizeof(int), cudaMemcpyDeviceToHost);
    // Finish reduction on CPU
    int finalSum = 0;
    for (int i = 0; i < grid.x; i++) {
        finalSum += h_output[i];
    }

    std::cout << "Final Sum: " << finalSum << std::endl;

    cudaFree(d_input);
    cudaFree(d_output);
    return 0;
}
```

Improved Reduction Using Shared Memory

- **Shared Memory Utilization:** By using shared memory, which is faster than global memory, the number of global memory accesses is significantly reduced. Each thread loads one element into shared memory, and then all reduction is done in shared memory.

- **Thread Synchronization:** It's crucial to synchronize threads when performing reduction in shared memory to ensure all threads have completed their operations before moving to the next reduction step.

- **Block Size and Grid Size:** Choosing the right block size can significantly affect performance due to its impact on memory access patterns and occupancy.

```
// Kernel for reducing an array using global memory
__global__ void reduceSumShm(int *input, int *output, int n) {
    extern __shared__ int sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Load input into shared memory.
    sdata[tid] = (i < n) ? input[i] : 0;
    __syncthreads();

    // Grid stride loop
    for (int stride = blockDim.x/2; stride>0; stride = stride / 2) {
        if(  tid < stride)
            sdata[tid] += sdata[tid + stride];
        __syncthreads();
    }


    // Let the thread 0 for each block write its result to the output array
    if (tid == 0) {
        output[blockIdx.x] = sdata[0];
    }
}
```

extern __shared__ int sdata[];

- extern __shared__ int sdata[];
- is used to declare dynamically sized shared memory arrays within CUDA kernels. The size of such arrays is specified when the kernel is launched, not in the code where the array is declared. This approach provides flexibility to use the same kernel code for different block sizes or to share the allocated shared memory among different types of data.
- **Kernel Launch**: When launching the kernel, you specify the amount of shared memory to allocate per block as an additional argument in the execution configuration:
- myKernel<<<numBlocks, blockSize, sharedMemSize>>>(args);
- sharedMemSize is the size in bytes of the shared memory you want to allocate per block.

LAB 3: Vector Dot Product

Dot Product

- (x1, x2, x3, x4)(y1, y2, y3, y4)=x1y1+x2y2+x3y3+x4y4

Using one thread

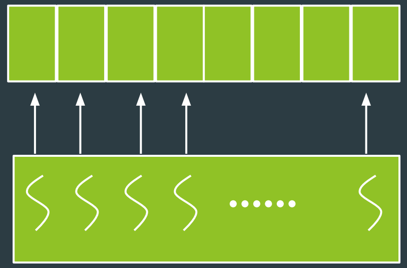
```
#define N   1024
__global__ void dot( int *a, int *b, int *dot ){
    int tid = threadIdx.x;
        int i;
        if(tid==0){
                for(i=0; i<N; i++){
                        *dot += a[i] * b[i];
                }
        }
}
```

Using multiple thread to calculate product and using one thread to obtain dot

```
#define N   1024
__global__ void dot( int *a, int *b, int *c, int *dot ){
        int tid = threadIdx.x;
        int i;

        c[tid] = a[tid] * b[tid];

        __syncthreads(); //need synchronize??

        if(tid==0){
                for(i=0; i<N; i++){
                        *dot += c[i];
                }
        }
}
```

Single Block Multiple Thread

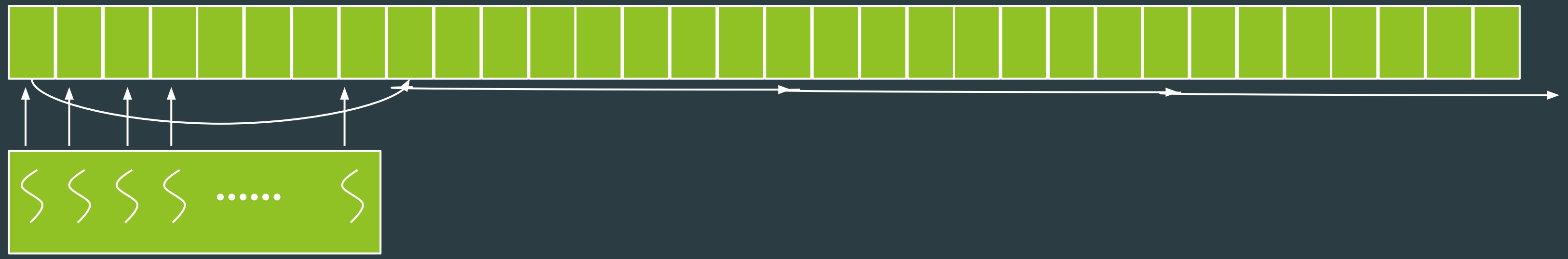
```
const int N = 1024;
const int threadsPerBlock = 1024;
const int blocksPerGrid = 1;
```



```
__global__ void dot( int *a, int *b, int *c ) {

    int tid = threadIdx.x;

    if (tid < N) {
        c[tid] = a[tid] * b[tid];
    }

    __syncthreads();

    int i = N / 2;
    while (i != 0) {
        if (tid < i){
            c[tid] += c[tid + i];
        }
        __syncthreads();

        i /= 2;
    }

}
```

Problem

- N is changed to 1024x1024
- Each thread must handle more than one position.

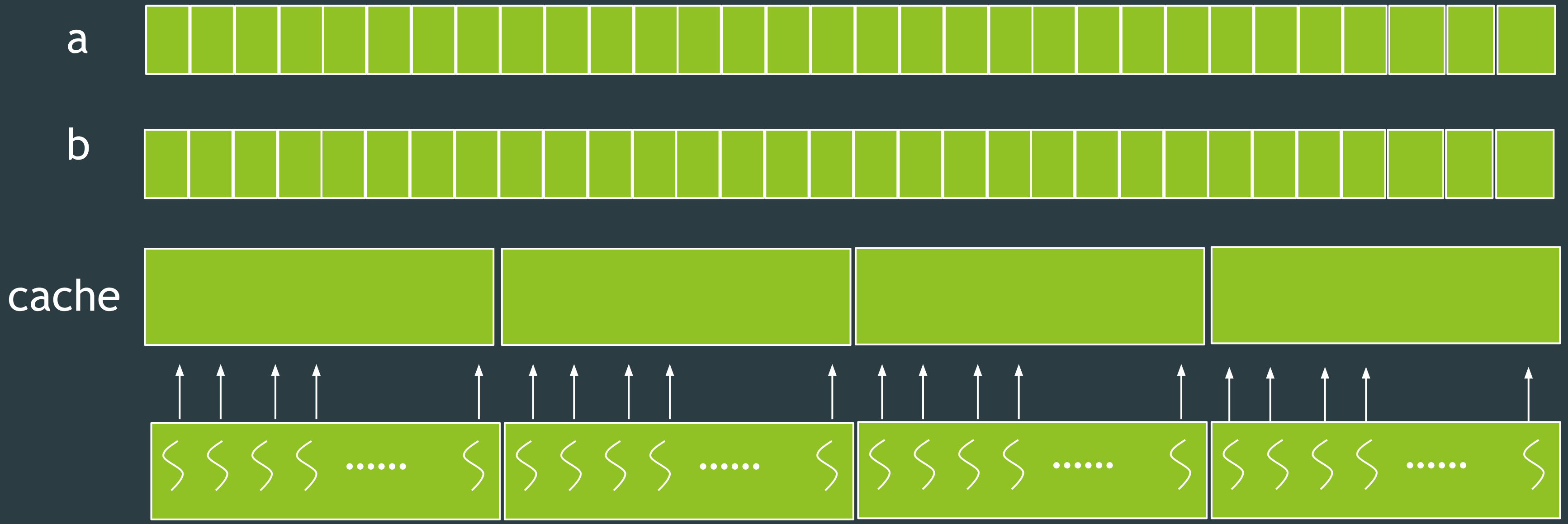


ANS

```
__global__ void dot( int *a, int *b, int *c){
        int tid = threadIdx.x;
        int i;
        int temp_tid = threadIdx.x;
        int temp = 0;

        while(temp_tid < N){
                temp += a[temp_tid] * b[temp_tid];
                temp_tid += blockDim.x;
        }
        // synchronize threads in this block
        c[tid] = temp;

        __syncthreads();
        i = blockDim.x/2;
        while (i != 0) {
             if (tid < i){
                  c[tid] += c[tid + i];
             }
             __syncthreads();
            i /= 2;
        }
}
```

Thinking

- May we modify the above code using multiple block multiple thread?



```
__shared__ int cache[threadsPerBlock];
int tid = threadIdx.x + blockIdx.x * blockDim.x;
int cacheIndex = threadIdx.x;
cache[cacheIndex] = a[tid] * b[tid];
```

```
const int N = 1024 *1024;
const int threadsPerBlock = 1024;
const int blocksPerGrid = N / 1024;

__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    cache[cacheIndex] = a[tid] * b[tid];
    __syncthreads();

    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }
    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
```

Thinking

- If N = 64 *1024 *1024
- const int blocksPerGrid = N / 1024;
- blocksPerGrid will exceed 65535 (the maximum blocks for GTX480)
- How to modify the example code?

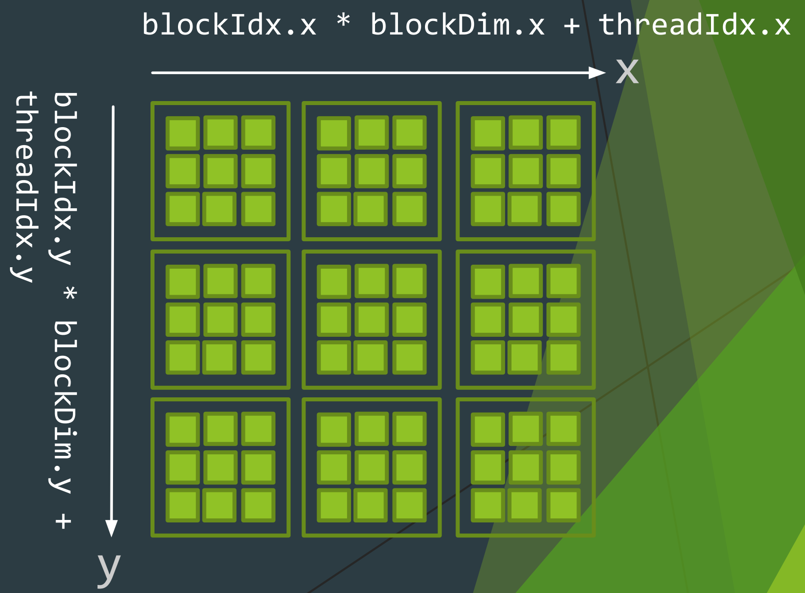
```
const int N = 64 * 1024 * 1024;
const int threadsPerBlock = 1024;
const int blocksPerGrid = 1024;
```

```
__global__ void dot( int *a, int *b, int *c ) {
    __shared__ int cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    int   temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
    // set the cache values
    cache[cacheIndex] = temp;
    __syncthreads();
    i = blockDim.x/2;

    while (i != 0) {
        if (cacheIndex < i){
                cache[cacheIndex] += cache[cacheIndex + i];
        }
        __syncthreads();
        i /= 2;
    }
    if(cacheIndex==0) c[blockIdx.x] = cache[0];
}
```

LAB4: 2D matrix addition

LAB4: 2D matrix addition

```
#include <stdio.h>

// CUDA kernel for adding two 2D matrices
__global__ void addMatrices(int *A, int *B, int *C, int width, int height) {
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    if (col < width && row < height) {
        int index = row * width + col;
        C[index] = A[index] + B[index];
    }
}
```

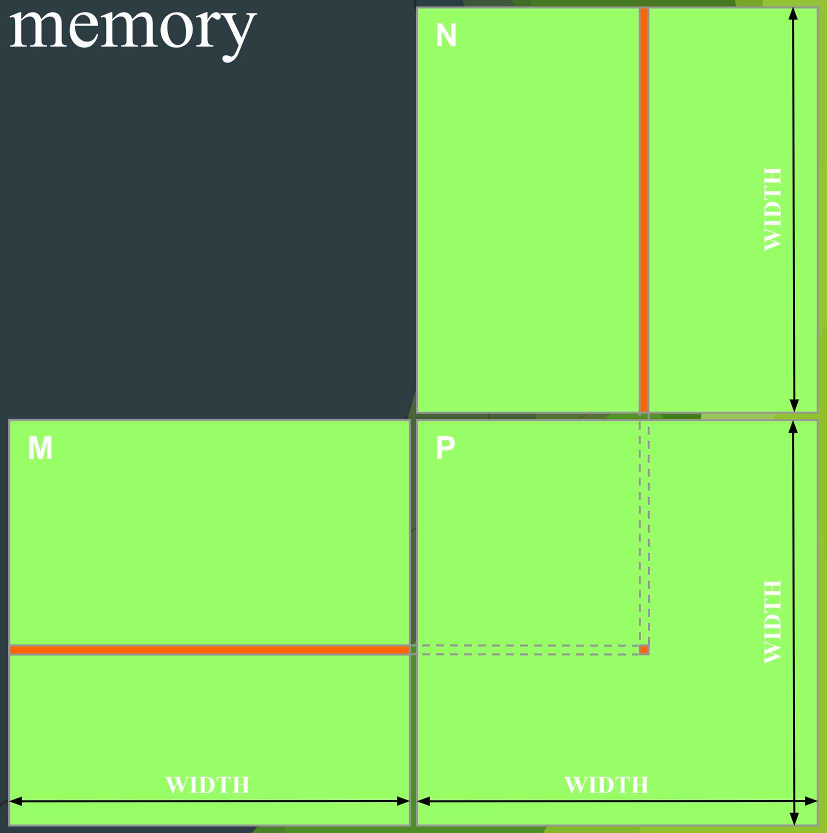

blockIdx.x * blockDim.x + threadIdx.x
x
blockIdx.y * blockDim.y + threadIdx.y
y

```c
int main() {
    int width = 1024;
    int height = 1024;
    int numElements = width * height;
    int size = numElements * sizeof(int);

    int *h_A, *h_B, *h_C;
    int *d_A, *d_B, *d_C;

    // Allocate host memory
    h_A = (int *)malloc(size);
    h_B = (int *)malloc(size);
    h_C = (int *)malloc(size);

    // Initialize host matrices with some values for demonstration
    for (int i = 0; i < numElements; i++) {
        h_A[i] = 1; // Example data
        h_B[i] = 2; // Example data
    }
    // Allocate device memory
    cudaMalloc((void **)&d_A, size);
    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);

    // Copy data from host to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Define block size and grid size
    dim3 blockSize(16, 16);
    dim3 gridSize((width + blockSize.x - 1) / blockSize.x, (height + blockSize.y - 1) / 
blockSize.y);

    // Launch the CUDA kernel
    addMatrices<<<gridSize, blockSize>>>(d_A, d_B, d_C, width, height);

    // Copy result back to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    // Verify the result
    for (int i = 0; i < numElements; i++) {
        if (h_C[i] != h_A[i] + h_B[i]) {
            fprintf(stderr, "Result verification failed at element %d!\n", i);
            exit(EXIT_FAILURE);
        }
    }

    printf("Test PASSED\n");

    // Free device and host memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

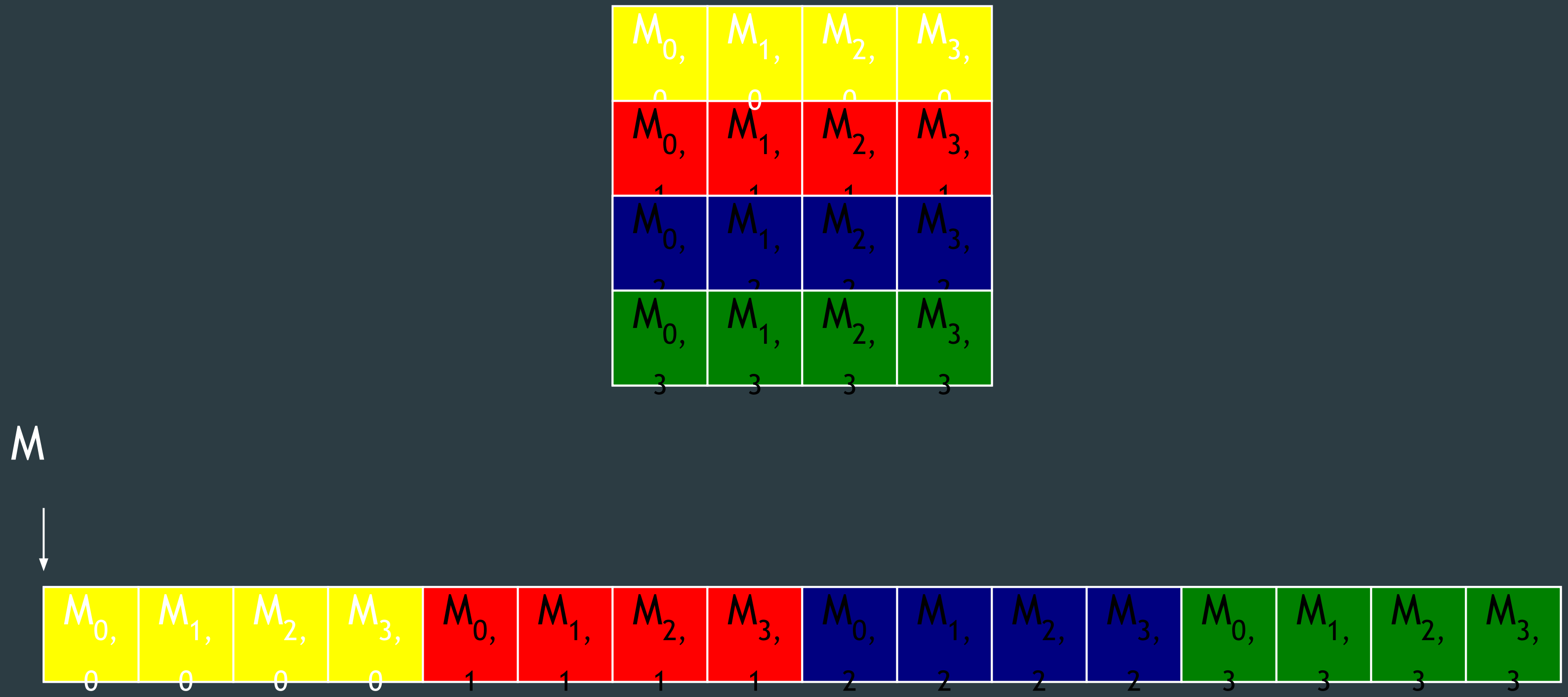
LAB 5: Matrix Multiplication

Square Matrix Multiplication

- P = M * N of size WIDTH x WIDTH
- One thread calculates one element of P
- M and N are loaded WIDTH times from global memory



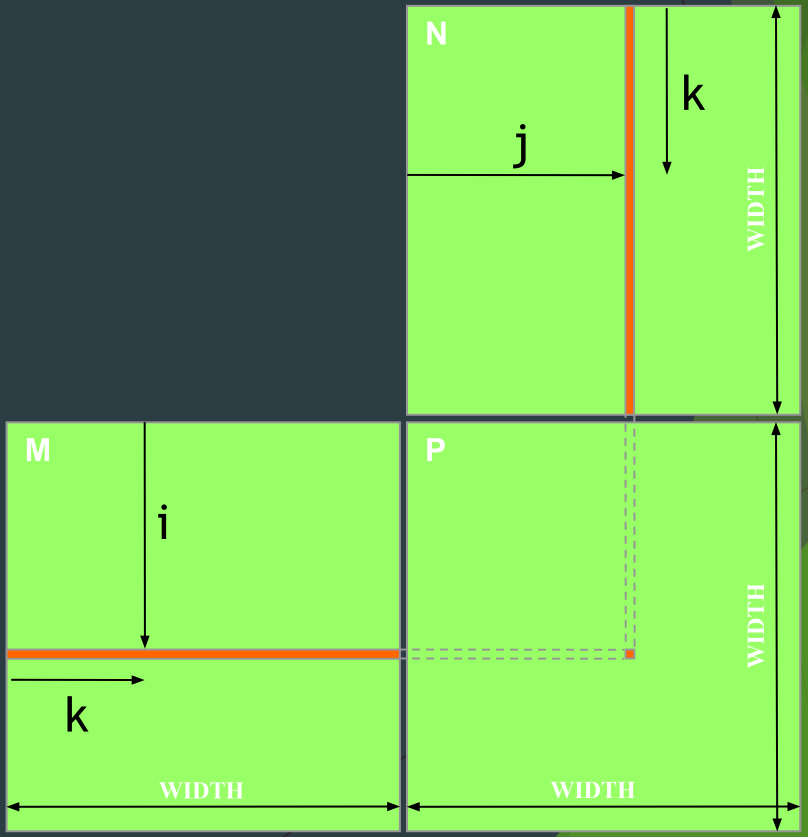
Memory Layout of a Matrix in C



$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$
$M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$
$M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$
$M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$
M
$M_{0,0}$ $M_{1,0}$ $M_{2,0}$ $M_{3,0}$ $M_{0,1}$ $M_{1,1}$ $M_{2,1}$ $M_{3,1}$ $M_{0,2}$ $M_{1,2}$ $M_{2,2}$ $M_{3,2}$ $M_{0,3}$ $M_{1,3}$ $M_{2,3}$ $M_{3,3}$

A Single Thread Version in C

```
__global__ void MatMulKernel(float *Md, float *Nd, float *Pd, int width)
{
    for(int i = 0; i < width; ++i){
        for(int j = 0; j < width; ++j){
            // Computes one element of P
            float Pvalue = 0;
            for(int k = 0; k < width; ++k){
                float Melement = *(Md + i*width + k);
                float Nelement = *(Nd + k*width + j);
                Pvalue += Melement * Nelement;
            }
            *(Pd + i*width + j) = Pvalue;
        }
    }
}
```



Input Matrix Data Transfer

```
void MatMul(float *M, float *N, float *P, int width)
{
    size_t size = width * width * sizeof(float);
    float *Md, *Nd, *Pd;

    // Allocate and Load M, N to device memory
    cudaMalloc((void **)&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **)&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc((void **)&Pd, size);
```

Output Matrix Data Transfer

```
    // Invoke kernel & record time – to be shown later
    ……

    // Read P from device memory
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Md);
    cudaFree(Nd);
    cudaFree(Pd);
}
```

Setting and Invoke kernel

```
// Setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(width, width);

// Get start time event
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

// Invoke kernel
 MatMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, width);

// Get stop time event
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
```

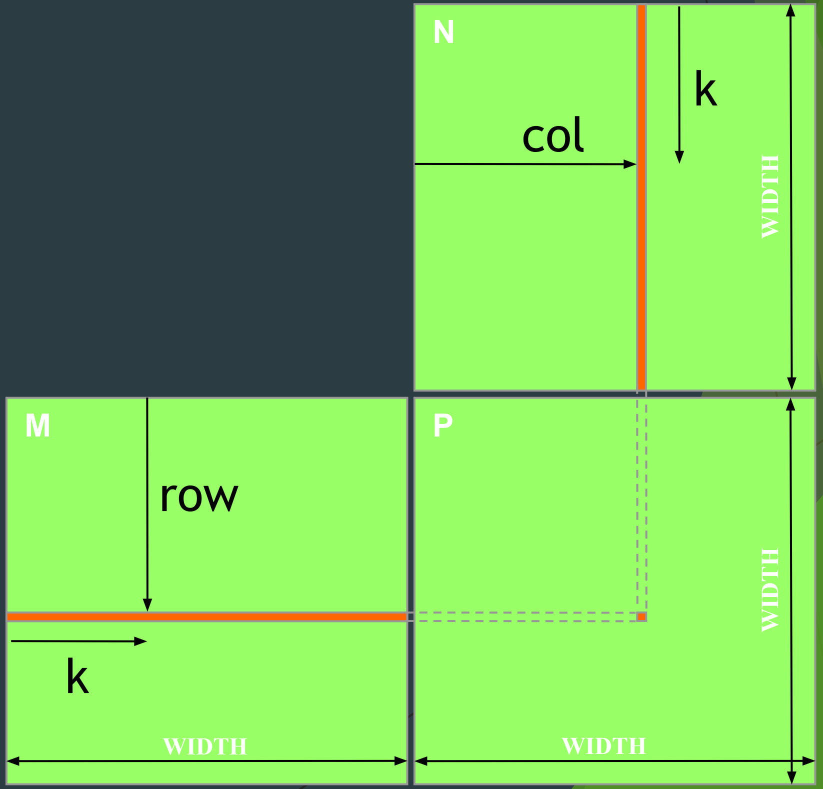
Timing

```
// Setup the execution configuration
dim3 dimGrid(1, 1);
dim3 dimBlock(width, width);

// Get start time event
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

// Invoke kernel
   MatMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, width);

// Get stop time event
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
```

```
// Compute execution time
float elapsedTime;
cudaEventElapsedTime(&elapsedTime,start,stop);
printf("GPU time: %13f msec\n", elapsedTime);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

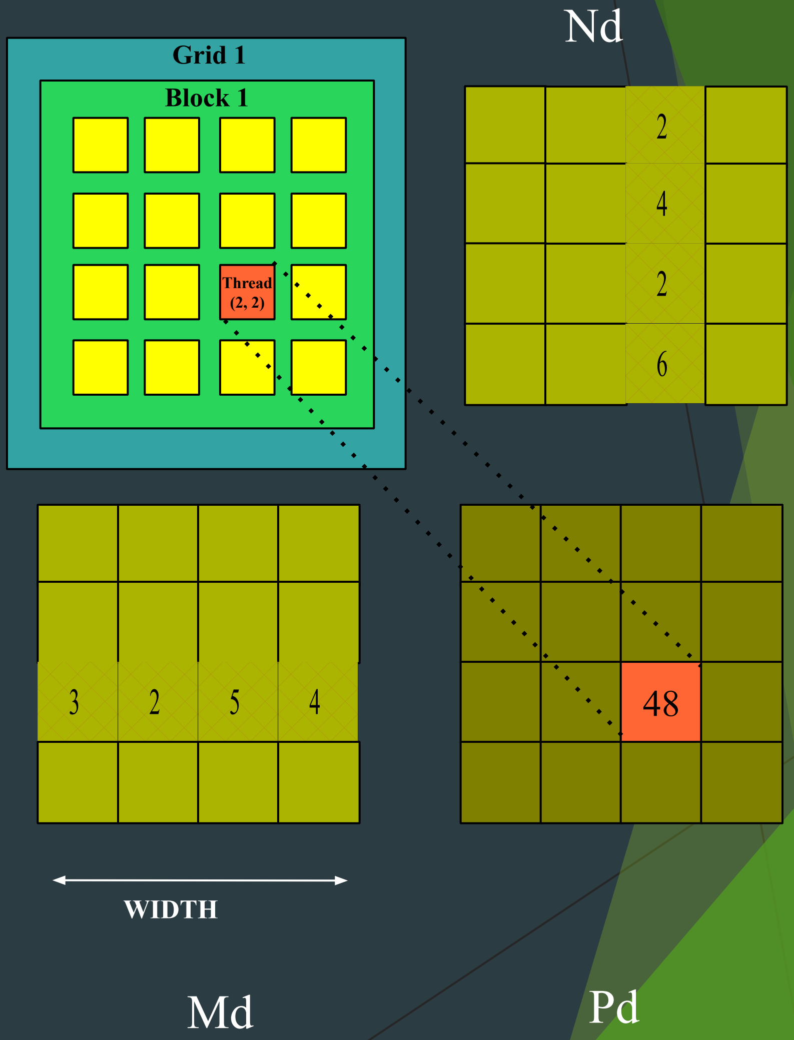
Kernel Function

```
__global__ void MatMulKernel(float *Md, float *Nd, float *Pd, int width)
{
    int row = threadIdx.y;
    int col = threadIdx.x;
    float Pvalue = 0;
    for (int k = 0; k < width; ++k) {
        float Melement = *(Md + row*width + k);
        float Nelement = *(Nd + k*width + col);
        Pvalue += Melement * Nelement;
    }
    *(Pd + row*width + col) = Pvalue;
}
```



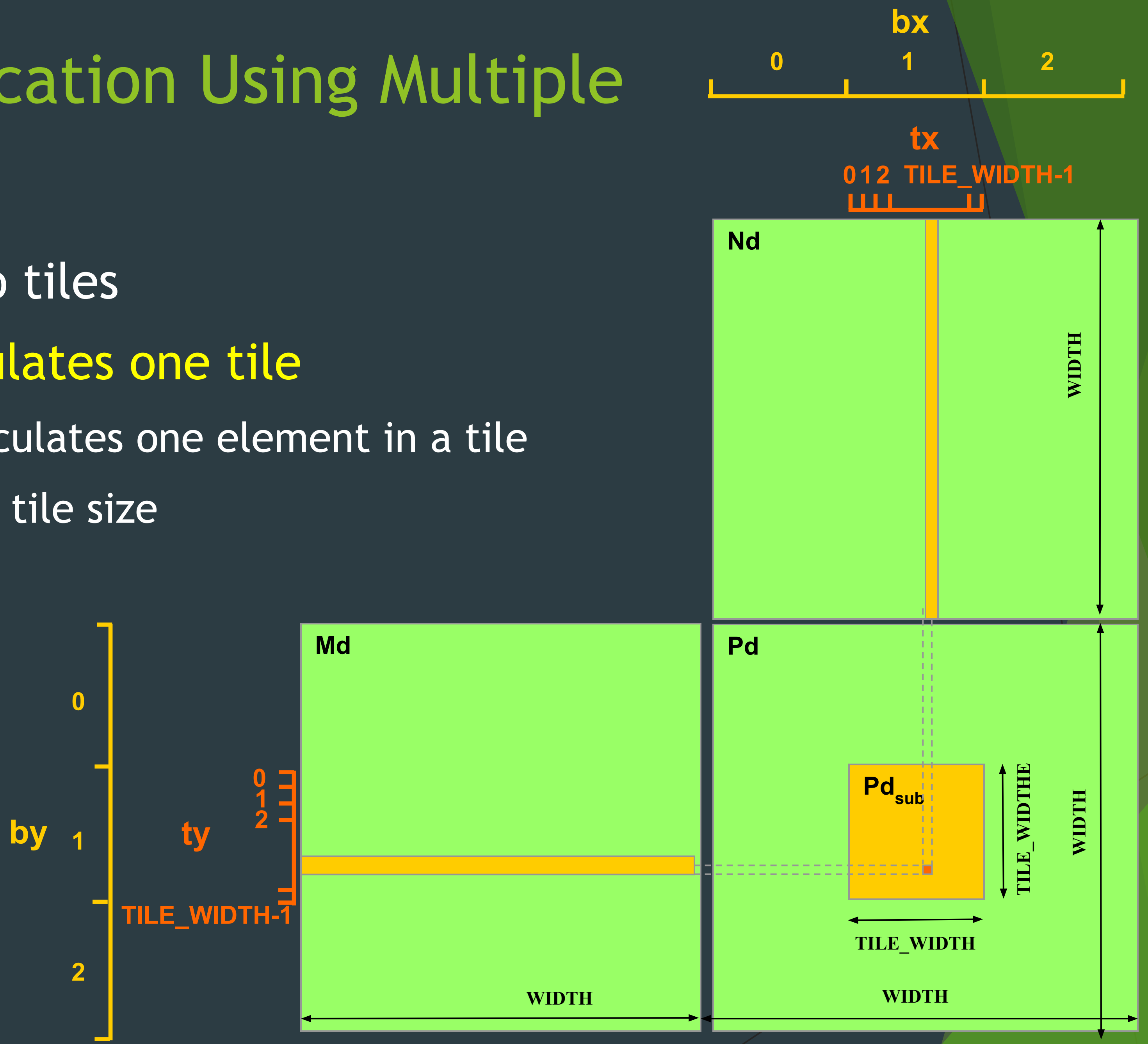
Only One Thread Block Used

- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and one addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block
- How to handle matrices larger than 32 x 32??

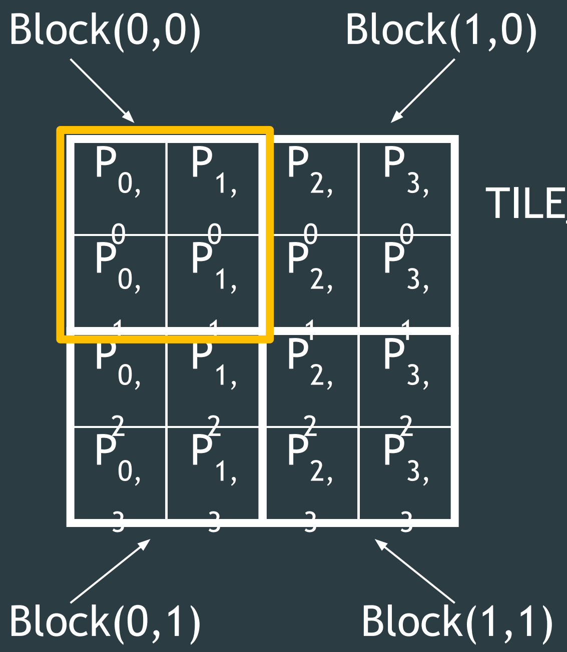


Matrix Multiplication Using Multiple Blocks

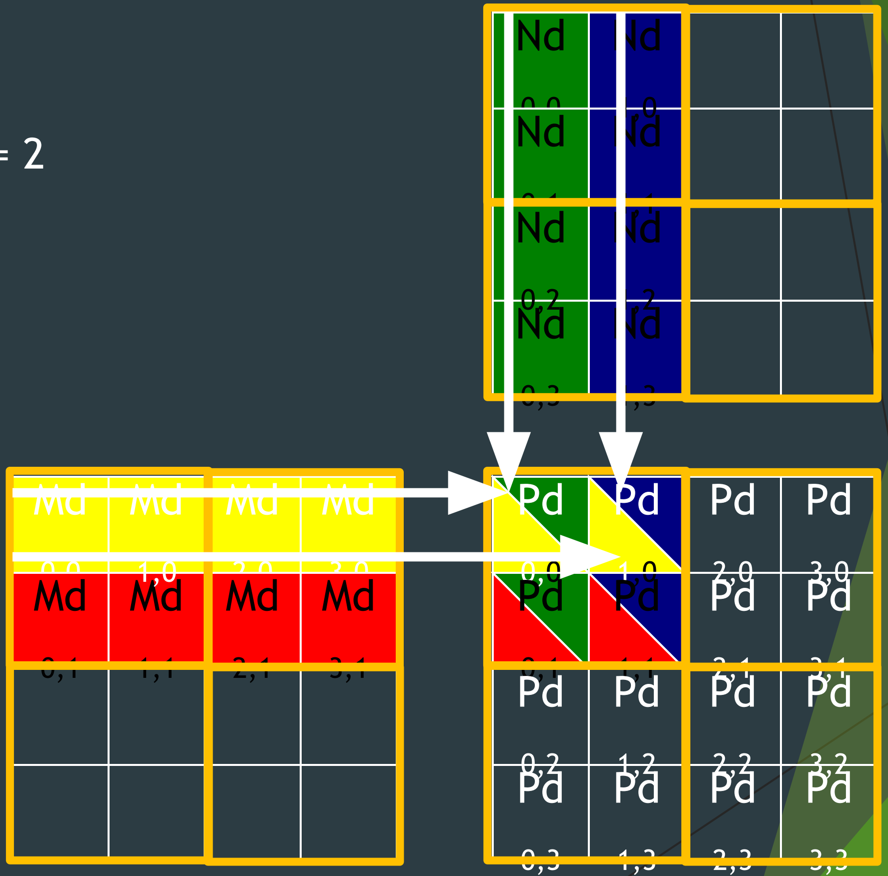
- Break-up Pd into tiles
- Each block calculates one tile
  - Each thread calculates one element in a tile
  - Block size equal tile size



A Small Example



TILE_WIDTH = 2

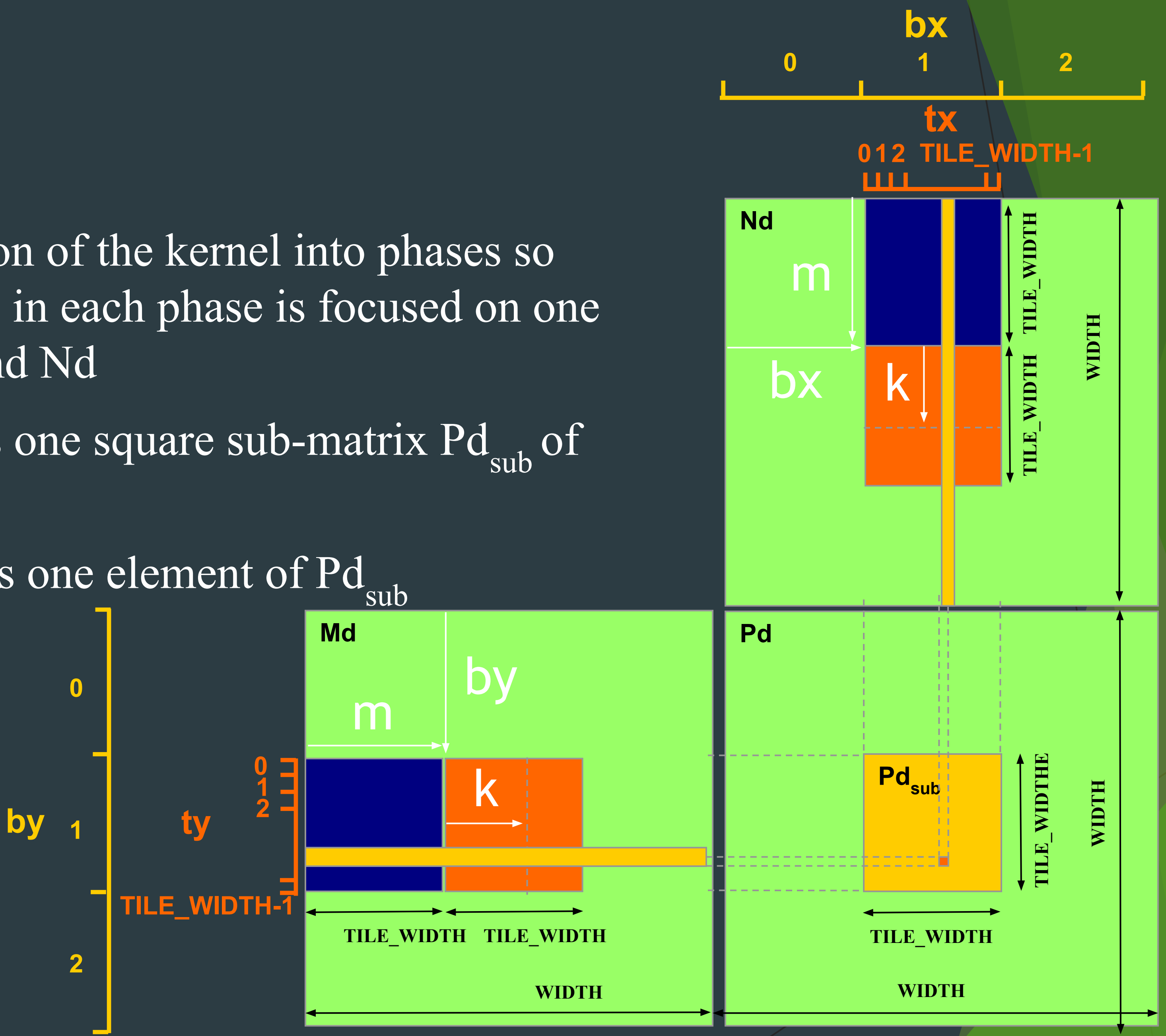


```
#include <stdio.h>
#include <cuda_runtime.h>

// Kernel definition for matrix multiplication
__global__ void matrixMul(const float* A, const float* B, float* C, int N) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < N && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < N; ++k) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}
```

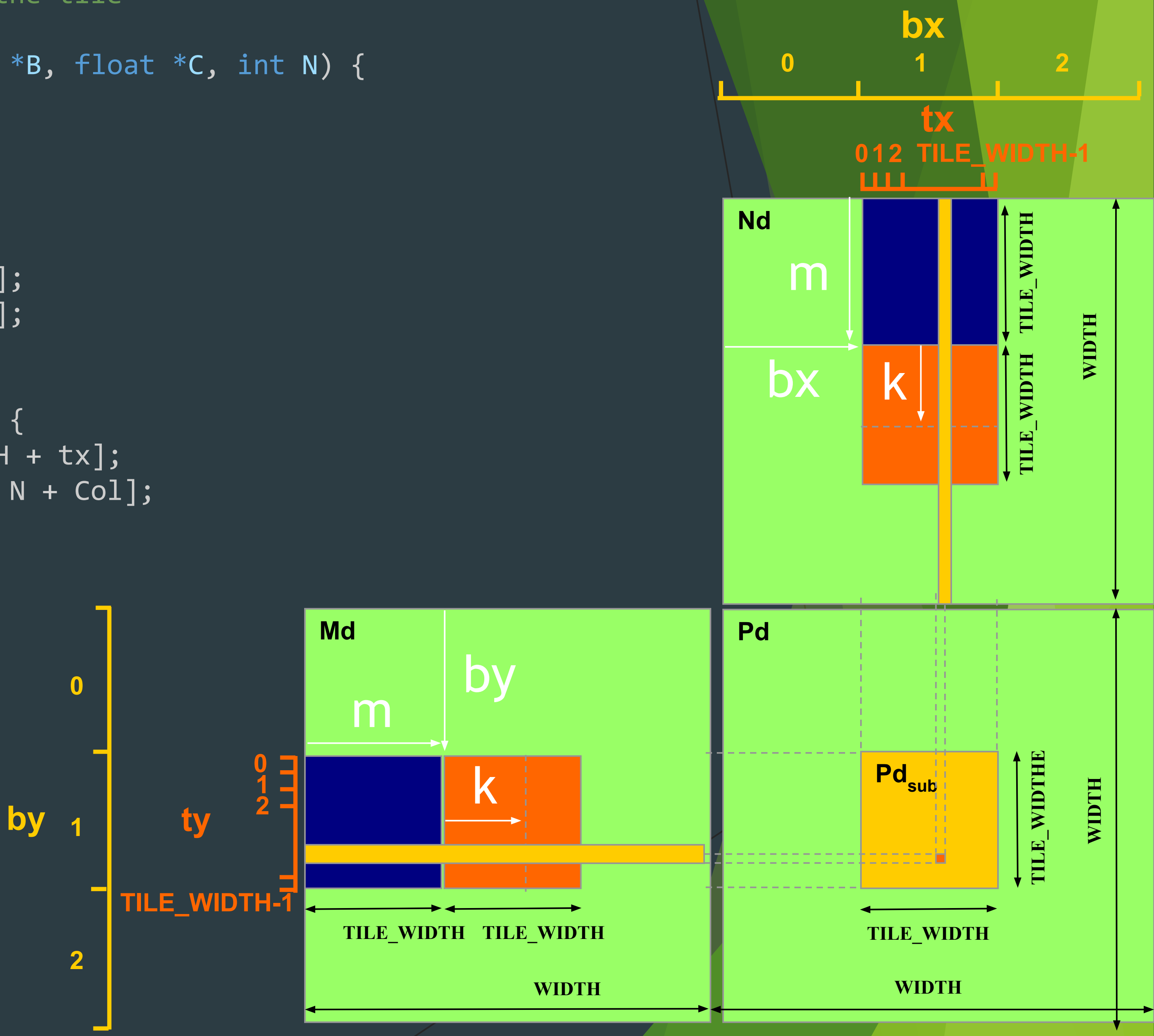
```
int main() {
    int N = 1024;  // Size of the matrix (1024x1024)
    size_t bytes = N * N * sizeof(float);

    float *h_A, *h_B, *h_C;
    float *d_A, *d_B, *d_C;

    // Allocate host memory
    h_A = (float*)malloc(bytes);
    h_B = (float*)malloc(bytes);
    h_C = (float*)malloc(bytes);

    // Initialize matrices on the host
    for (int i = 0; i < N * N; i++) {
        h_A[i] = 0.01f;
        h_B[i] = 0.02f;
    }
    // Allocate device memory
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);

    // Transfer data from host to device memory
    cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, bytes, cudaMemcpyHostToDevice);

    // Thread block and grid dimensions
    dim3 threadsPerBlock(16, 16);
    dim3 blocksPerGrid((N + threadsPerBlock.x - 1) / threadsPerBlock.x, (N + threadsPerBlock.y - 1) / threadsPerBlock.y);

    // Execute the matrix multiplication kernel
    matrixMul<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy the result matrix from device to host memory
    cudaMemcpy(h_C, d_C, bytes, cudaMemcpyDeviceToHost);
    // Free device memory
     cudaFree(d_A);
     cudaFree(d_B);
     cudaFree(d_C);

     // Free host memory
     free(h_A);
     free(h_B);
     free(h_C);

     return 0;
}
```

Tiled Multiply

- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd
- Each block computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each thread computes one element of Pd_{sub}



```
#define TILE_WIDTH 16  // Define the width of the tile

__global__ void matrixMulTiled(float *A, float *B, float *C, int N) {
    int bx = blockIdx.x, by = blockIdx.y;
    int tx = threadIdx.x, ty = threadIdx.y;
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    __shared__ float As[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Bs[TILE_WIDTH][TILE_WIDTH];

    float Cvalue = 0;
    for (int m = 0; m < (N / TILE_WIDTH); ++m) {
        As[ty][tx] = A[Row * N + m * TILE_WIDTH + tx];
        Bs[ty][tx] = B[(m * TILE_WIDTH + ty) * N + Col];
        __syncthreads();

        for (int k = 0; k < TILE_WIDTH; ++k) {
            Cvalue += As[ty][k] * Bs[k][tx];
        }
        __syncthreads();
    }

    C[Row * N + Col] = Cvalue;
}
```



```
int main() {
    int N = 1024; // Assume N is a multiple of TILE_WIDTH
    size_t bytes = N * N * sizeof(float);

    float *h_A = (float*)malloc(bytes);
    float *h_B = (float*)malloc(bytes);
    float *h_C = (float*)malloc(bytes);

    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, bytes);
    cudaMalloc(&d_B, bytes);
    cudaMalloc(&d_C, bytes);

    // Initialize matrices
    for (int i = 0; i < N * N; i++) {
        h_A[i] = 0.01f * i;
        h_B[i] = 0.02f * i;
    }
    cudaMemcpy(d_A, h_A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, bytes, cudaMemcpyHostToDevice);

    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
    dim3 dimGrid((N + TILE_WIDTH - 1) / TILE_WIDTH, (N + TILE_WIDTH - 1) / TILE_WIDTH);

    matrixMulTiled<<<dimGrid, dimBlock>>>(d_A, d_B, d_C, N);

    cudaMemcpy(h_C, d_C, bytes, cudaMemcpyDeviceToHost);

    // Free resources and end
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_A);
    free(h_B);
    free(h_C);

    std::cout << "Matrix multiplication completed successfully." << std::endl;

    return 0;
}
```

Performance Comparison

- Multiple Block Multiplication
  - Each thread: **WIDTH*2** load from **global** memory (2 for M and N memory access)
  - Each thread: **WIDTH*2** mul/add operation (2 for mul and add)
  - load:operation=1:1
- Tiled Algorithm with **shared memory**
  - Each tile: **TILE_WIDTH^2*2** load from **global** memory (2 for M and N memory access)
  - Each tile: **TILE_WIDTH^2*2*TILE_WIDTH** mul/add operation

    Because each tile has **TILE_WIDTH^2** threads $\rightarrow$
  - Each thread: **2** load from **global** memory
  - Each thread: **TILE_WIDTH*2** mul/add operation
  - **load:operation=1:TILE_WIDTH**

Invoke Tiled CUDA Kernel

```
size_t size = width * width * sizeof(float);
size_t shared_size = TILE_WIDTH * TILE_WIDTH * sizeof(float) * 2;
float *Md, *Nd, *Pd;

// Setup the execution configuration
dim3 dimGrid(width/TILE_WIDTH, width/TILE_WIDTH);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

// Invoke kernel
MatMulKernel<<<dimGrid, dimBlock, shared_size>>>(Md, Nd, Pd, width);
```

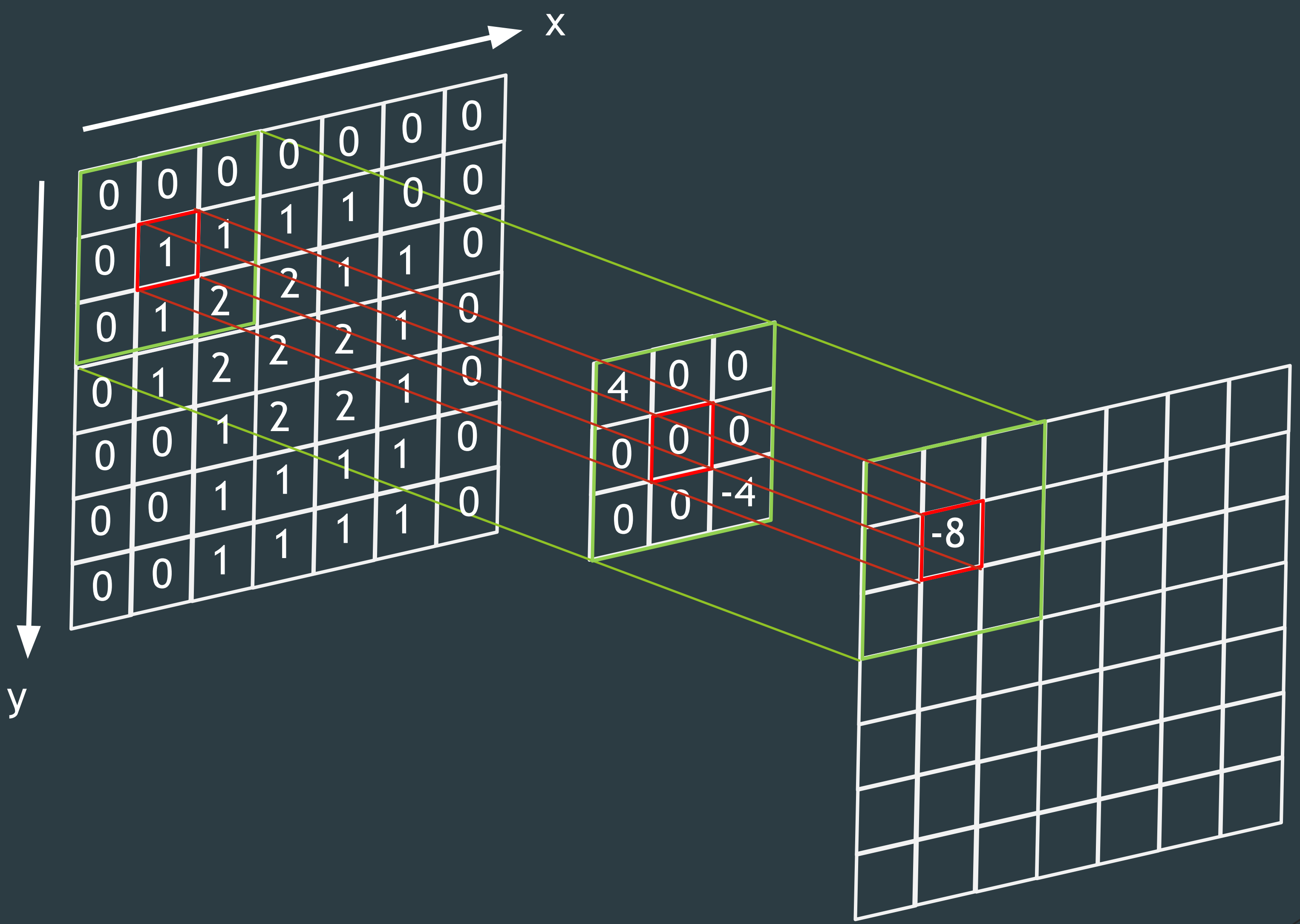
Functions May be Used

```
// Get a matrix element
__device__ float GetElement(float *matrix, int row, int col, int width)
{
    return *(matrix + row*width + col);
}

// Set a matrix element
__device__ void SetElement(float *matrix, int row, int col, int width, float value)
{
    *(matrix + row*width + col) = value;
}

// Get the TILE_WIDTHxTILE_WIDTH sub-matrix matsub of matrix that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of matrix
__device__ float *GetSubMatrix(float *matrix, int blockrow, int col, int blockwidth)
{
    return (matrix + blockrow*TILE_WIDTH*width + blockcol*TILE_WIDTH);
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatMulKernel(float *Md, float *Nd, float *Pd, int width) {
    int blockRow = blockIdx.y;  int blockCol = blockIdx.x;
    float *Pd_sub = GetSubMatrix(Pd, blockRow, blockCol, width);
    int row = threadIdx.y;  int col = threadIdx.x;
    float Pvalue = 0;
    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
    for (int m = 0; m < (width / TILE_WIDTH); ++m) {
        float *Md_sub = GetSubMatrix(Md, blockRow, m, width);
        float *Nd_sub = GetSubMatrix(Nd, m, blockCol, width);
        Mds[row][col] = GetElement(Md_sub, row, col, width); // put Md_sub into shared memory
        Nds[row][col] = GetElement(Nd_sub, row, col, width); // put Nd_sub into shared memory
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += Mds[row][k] * Nds[k][col];
        __syncthreads();
    }
    SetElement(Pd_sub, row, col, width, Pvalue);
}
```

Lab6: 2D Convolution



$$\begin{array}{r} (4 \times 0) \\ (0 \times 0) \\ (0 \times 0) \\ (0 \times 0) \\ (0 \times 1) \\ (0 \times 1) \\ (0 \times 0) \\ (0 \times 1) \\ + \; (-4 \times 2) \\ \hline -8 \end{array}$$

```
#include <iostream>
#include <cuda_runtime.h>

// Define the convolution kernel size
#define KERNEL_DIM 3


void convolve2D_CPU(float *input, float *output, float *kernel, int width, int height) {
    int i, j, m, n;
    float outputValue;
    int halfKernel = KERNEL_DIM / 2;

    for (i = halfKernel; i < height - halfKernel; ++i) {      // rows
        for (j = halfKernel; j < width - halfKernel; ++j) {   // columns
            outputValue = 0.0;

            for (m = -halfKernel; m <= halfKernel; ++m) { // kernel rows
                for (n = -halfKernel; n <= halfKernel; ++n) { // kernel columns
                    int ii = i + m;
                    int jj = j + n;
                    outputValue += input[ii*width + jj] * kernel[(m + halfKernel) * KERNEL_DIM + (n + halfKernel)];
                }
            }
            output[i*width + j] = outputValue;
        }
    }
}
```

```
// CUDA kernel to perform 2D convolution
__global__ void conv2D(float *input, float *output, float *kernel, int width, int height) {
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;

    int halfKernel = KERNEL_DIM / 2;
    float value = 0.0;

    if (ix >= halfKernel && ix < (width - halfKernel) && iy >= halfKernel && iy < (height - halfKernel)) {
        for (int kx = -halfKernel; kx <= halfKernel; kx++) {
            for (int ky = -halfKernel; ky <= halfKernel; ky++) {
                int x = ix + kx;
                int y = iy + ky;
                value += input[y * width + x] * kernel[(ky + halfKernel) * KERNEL_DIM + (kx + halfKernel)];
            }
        }
        output[iy * width + ix] = value;
    }
}
```

```
int main() {
    int width = 1024;
    int height = 1024;
    int imgSize = width * height;
    float *h_input, *h_output, *h_kernel, *h_output_CPU;
    float *d_input, *d_output, *d_kernel;

    // Allocate host memory
    h_input = new float[imgSize];
    h_output = new float[imgSize];
    h_output_CPU = new float[imgSize];
    h_kernel = new float[KERNEL_DIM * KERNEL_DIM];

    // Initialize the kernel (example)
    float exampleKernel[KERNEL_DIM * KERNEL_DIM] = {
        0, -1, 0,
        -1, 5, -1,
        0, -1, 0
    };

    for (int i = 0; i < KERNEL_DIM * KERNEL_DIM; i++) {
        h_kernel[i] = exampleKernel[i];
    }

    // Initialize the input image with arbitrary data
    for (int i = 0; i < imgSize; i++) {
        h_input[i] = 1.0;  // All pixels 1.0 for simplicity
    }
```

```
// Allocate device memory
cudaMalloc((void **)&d_input, imgSize * sizeof(float));
cudaMalloc((void **)&d_output, imgSize * sizeof(float));
cudaMalloc((void **)&d_kernel, KERNEL_DIM * KERNEL_DIM * sizeof(float));

// Copy data from host to device
cudaMemcpy(d_input, h_input, imgSize * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_kernel, h_kernel, KERNEL_DIM * KERNEL_DIM * sizeof(float), cudaMemcpyHostToDevice);

// Define block size and grid size
dim3 blockSize(16, 16);
dim3 gridSize((width + blockSize.x - 1) / blockSize.x, (height + blockSize.y - 1) / blockSize.y);

// Launch the CUDA kernel
conv2D<<<gridSize, blockSize>>>(d_input, d_output, d_kernel, width, height);

// Copy result back to host
cudaMemcpy(h_output, d_output, imgSize * sizeof(float), cudaMemcpyDeviceToHost);

//Launch CPU
convolve2D_CPU(h_input, h_output_CPU, h_kernel, width, height);
```

```
    int pass = 1;

    for(int i=0; i<height; i++){
        for(int j=0; j<width; j++){
            if(h_output_CPU[i*width + j] != h_output[i*width + j]){
                pass = 0;
            }
        }
    }
    if(pass==1){
        printf("Pass!!!\n");
    }
    else{
        printf("Fail!!!");
    }

    // Free device and host memory
    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_kernel);
    delete[] h_input;
    delete[] h_output;
    delete[] h_output_CPU;
    delete[] h_kernel;

    return 0;
}
```

Using constant memory

- Constant memory is a special type of memory that is read-only for kernels, cached, and optimized for scenarios where all threads access the same value or when many threads read the same location.
- Initializing Constant Memory in CUDA
- Declare the constant memory space in your CUDA code outside of any function, typically at the global level.

```
#include <iostream>
#include <cuda_runtime.h>
// Define the convolution kernel size
#define KERNEL_DIM 3
// Define a constant array in constant memory
__constant__ float d_kernel[KERNEL_DIM * KERNEL_DIM];
```

- **Initialize Constant Memory and copy data to constant memory**

```
// Initialize the kernel (example)
float exampleKernel[KERNEL_DIM * KERNEL_DIM] = {
    0, -1, 0,
    -1, 5, -1,
    0, -1, 0
};

for (int i = 0; i < KERNEL_DIM * KERNEL_DIM; i++) {
    h_kernel[i] = exampleKernel[i];
}

// Copy data to constant memory
cudaMemcpyToSymbol(d_kernel, h_kernel, KERNEL_DIM * KERNEL_DIM * sizeof(float));
```

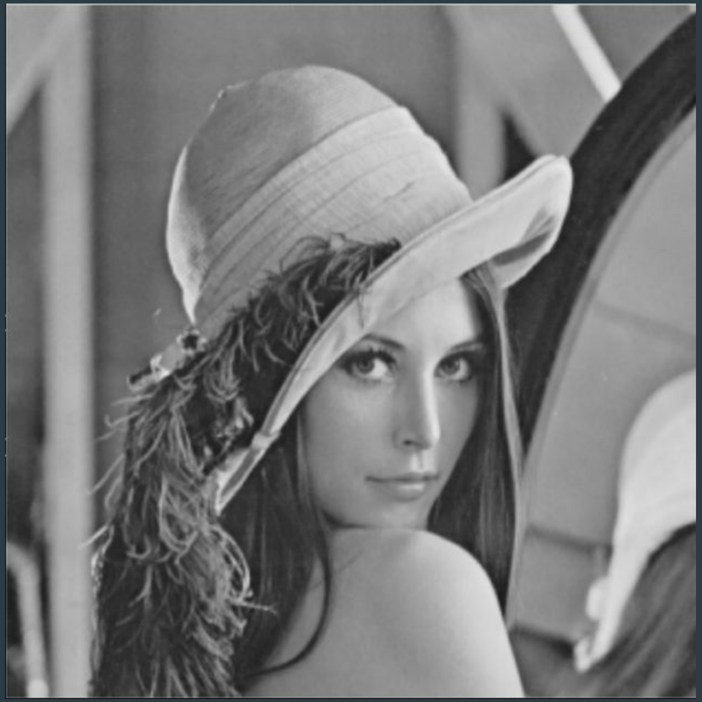
```
// CUDA kernel to perform 2D convolution

__global__ void conv2D(float *input, float *output, int width, int height) {
    int ix = blockIdx.x * blockDim.x + threadIdx.x;
    int iy = blockIdx.y * blockDim.y + threadIdx.y;

    int halfKernel = KERNEL_DIM / 2;
    float value = 0.0;

    if (ix >= halfKernel && ix < (width - halfKernel) && iy >= halfKernel && iy < (height - halfKernel)) {
        for (int kx = -halfKernel; kx <= halfKernel; kx++) {
            for (int ky = -halfKernel; ky <= halfKernel; ky++) {
                int x = ix + kx;
                int y = iy + ky;
                value += input[y * width + x] * d_kernel[(ky + halfKernel) * KERNEL_DIM + (kx + halfKernel)];
            }
        }
        output[iy * width + ix] = value;
    }
}
```

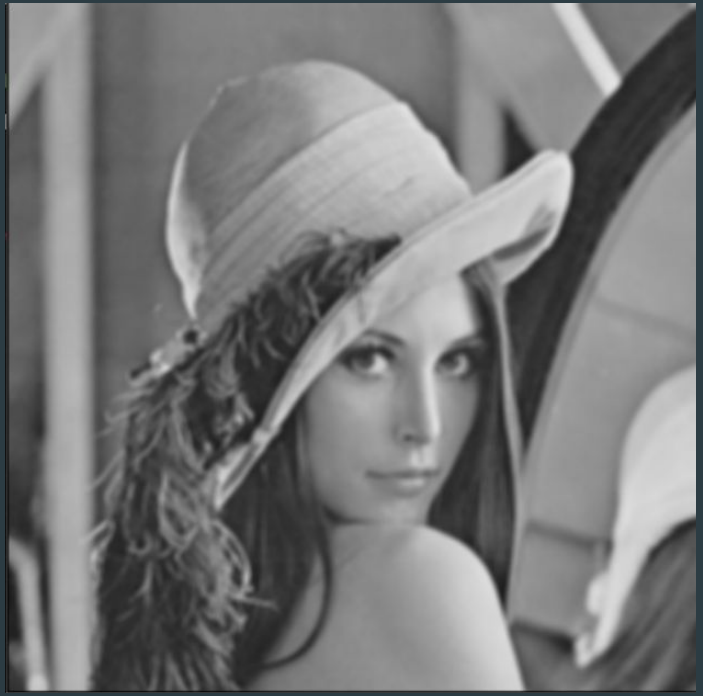
```
int main() {
    int width = 16;
    int height = 16;
    int imgSize = width * height;
    float *h_input, *h_output, *h_kernel, *h_output_CPU;
    float *d_input, *d_output;//*d_kernel

    // Allocate host memory
    h_input = new float[imgSize];
    h_output = new float[imgSize];
    h_output_CPU = new float[imgSize];
    h_kernel = new float[KERNEL_DIM * KERNEL_DIM];

    // Initialize the kernel (example)
    float exampleKernel[KERNEL_DIM * KERNEL_DIM] = {
        0, -1, 0,
        -1, 5, -1,
        0, -1, 0
    };

    for (int i = 0; i < KERNEL_DIM * KERNEL_DIM; i++) {
        h_kernel[i] = exampleKernel[i];
    }

    // Copy data to constant memory
    cudaMemcpyToSymbol(d_kernel, h_kernel, KERNEL_DIM * KERNEL_DIM * sizeof(float));
```

```
// Initialize the input image with arbitrary data
for (int i = 0; i < imgSize; i++) {
    h_input[i] = 1.0;  // All pixels 1.0 for simplicity
}

// Allocate device memory
cudaMalloc((void **)&d_input, imgSize * sizeof(float));
cudaMalloc((void **)&d_output, imgSize * sizeof(float));
//cudaMalloc((void **)&d_kernel, KERNEL_DIM * KERNEL_DIM * sizeof(float));

// Copy data from host to device
cudaMemcpy(d_input, h_input, imgSize * sizeof(float), cudaMemcpyHostToDevice);
//cudaMemcpy(d_kernel, h_kernel, KERNEL_DIM * KERNEL_DIM * sizeof(float), cudaMemcpyHostToDevice);

// Define block size and grid size
dim3 blockSize(16, 16);
dim3 gridSize((width + blockSize.x - 1) / blockSize.x, (height + blockSize.y - 1) / blockSize.y);

// Launch the CUDA kernel
conv2D<<<gridSize, blockSize>>>(d_input, d_output, width, height);

// Copy result back to host
cudaMemcpy(h_output, d_output, imgSize * sizeof(float), cudaMemcpyDeviceToHost);
```

```
    //Launch CPU
    convolve2D_CPU(h_input, h_output_CPU, h_kernel, width, height);


    int pass = 1;
    for(int i=0; i<height; i++){
        for(int j=0; j<width; j++){
            if(h_output_CPU[i*width + j] != h_output[i*width + j]){
                pass = 0;
            }
        }
    }
    if(pass==1){
        printf("Pass!!!\n");
    }
    else{
        printf("Fail!!!\n");
    }

    // Free device and host memory
    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_kernel);
    delete[] h_input;
    delete[] h_output;
    delete[] h_output_CPU;
    delete[] h_kernel;

    return 0;
}
```

Blur operation





Blur an image using convolution

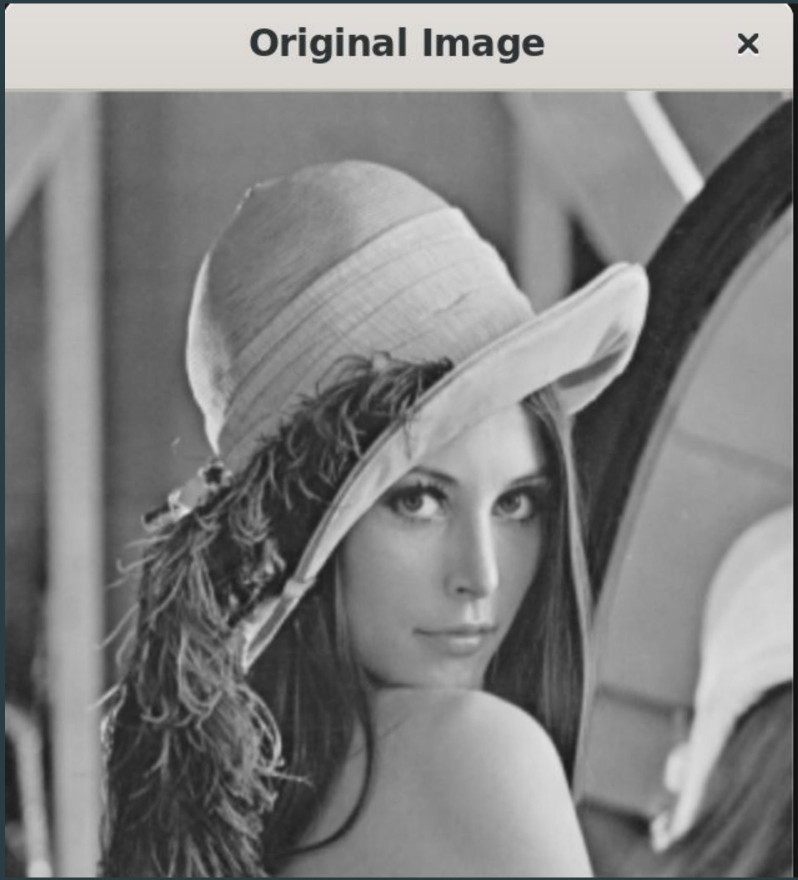
```
#include <iostream>
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>

// CUDA kernel to perform 2D convolution
__global__ void conv2D(const uchar *input, uchar *output, int width, int height, const float *kernel, int kWidth,
int kHeight) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int kHalfWidth = kWidth / 2;
    int kHalfHeight = kHeight / 2;

    if (x >= kHalfWidth && x < (width - kHalfWidth) && y >= kHalfHeight && y < (height - kHalfHeight)) {
        float sum = 0.0f;
        for (int ky = -kHalfHeight; ky <= kHalfHeight; ky++) {
            for (int kx = -kHalfWidth; kx <= kHalfWidth; kx++) {
                int pixel = input[(y + ky) * width + (x + kx)];
                float coeff = kernel[(ky + kHalfHeight) * kWidth + (kx + kHalfWidth)];
                sum += pixel * coeff;
            }
        }
        output[y * width + x] = static_cast<uchar>(sum);
    }
}
```

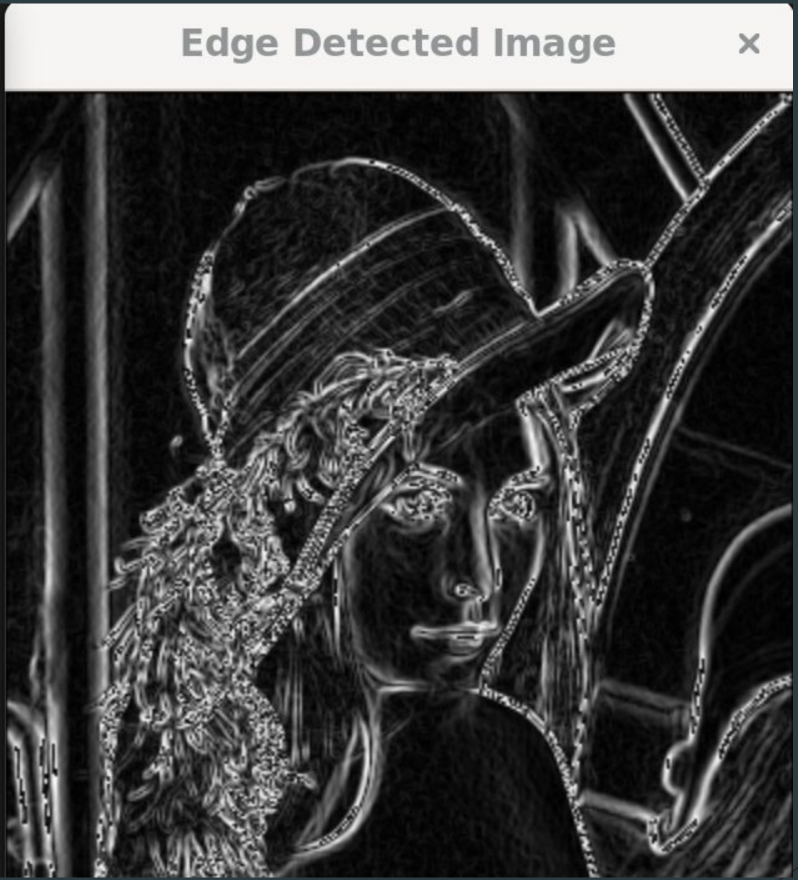
```
int main() {
    // Load an image in grayscale
    cv::Mat image = cv::imread("Lenna.jpg", cv::IMREAD_GRAYSCALE);
    if (image.empty()) {
        std::cerr << "Could not read the image." << std::endl;
        return -1;
    }

    // Display the original image
    cv::imshow("Original Image", image);
    cv::waitKey(0);

    // Define the kernel for averaging (blur)
    float h_kernel[3][3] = {
        {1/9.0, 1/9.0, 1/9.0},
        {1/9.0, 1/9.0, 1/9.0},
        {1/9.0, 1/9.0, 1/9.0}
    };

    // Prepare data for CUDA
    uchar *d_input, *d_output;
    float *d_kernel;
    cudaMalloc((void **)&d_input, image.total());
    cudaMalloc((void **)&d_output, image.total());
    cudaMalloc((void **)&d_kernel, sizeof(h_kernel));
    cudaMemcpy(d_input, image.data, image.total(), cudaMemcpyHostToDevice);
    cudaMemcpy(d_kernel, h_kernel, sizeof(h_kernel), cudaMemcpyHostToDevice);
    // Define block size and grid size
    dim3 blockSize(16, 16);
    dim3 gridSize((image.cols + blockSize.x - 1) / blockSize.x, (image.rows + blockSize.y - 1) / blockSize.y);

    // Launch the CUDA kernel
    conv2D<<<gridSize, blockSize>>>(d_input, d_output, image.cols, image.rows, d_kernel, 3, 3);

    // Allocate memory for the output image
    cv::Mat result(image.rows, image.cols, CV_8UC1);

    // Copy the result back to host
    cudaMemcpy(result.data, d_output, image.total(), cudaMemcpyDeviceToHost);

    // Display the result
    cv::imshow("Convolved Image", result);
    cv::waitKey(0);

    // Free memory
    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_kernel);

    return 0;
}
```

Makefile

```
# Makefile for compiling CUDA and OpenCV project

OPENCV_FLAGS = `pkg-config --cflags --libs opencv4`  # Change to 'opencv' if opencv4 doesn't work
CUDA_FLAGS = -lcuda -lcudart

NVCC = nvcc
CC_FLAGS = -std=c++11

TARGET = blur
SOURCE = blur.cu

all: $(TARGET)

$(TARGET): $(SOURCE)
	$(NVCC) $(CC_FLAGS) $(SOURCE) -o $(TARGET) $(OPENCV_FLAGS) $(CUDA_FLAGS)

clean:
	rm -f $(TARGET)
```

Edge detection using Sobel





```
#include <iostream>
#include <opencv2/opencv.hpp>
#include <opencv2/highgui/highgui.hpp>

__global__ void sobelEdgeDetection(const uchar *input, uchar *output, int width, int height, const float *kernelX,
const float *kernelY, int kWidth, int kHeight) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    int kHalfWidth = kWidth / 2;
    int kHalfHeight = kHeight / 2;

    if (x >= kHalfWidth && x < (width - kHalfWidth) && y >= kHalfHeight && y < (height - kHalfHeight)) {
        float sumX = 0.0f, sumY = 0.0f;
        for (int ky = -kHalfHeight; ky <= kHalfHeight; ky++) {
            for (int kx = -kHalfWidth; kx <= kHalfWidth; kx++) {
                int pixel = input[(y + ky) * width + (x + kx)];
                sumX += pixel * kernelX[(ky + kHalfHeight) * kWidth + (kx + kHalfWidth)];
                sumY += pixel * kernelY[(ky + kHalfHeight) * kWidth + (kx + kHalfWidth)];
            }
        }
        output[y * width + x] = sqrt(sumX * sumX + sumY * sumY);
    }
}
```

```cpp
int main() {
    cv::Mat image = cv::imread("Lenna.jpg", cv::IMREAD_GRAYSCALE);
    if (image.empty()) {
        std::cerr << "Could not read the image." << std::endl;
        return -1;
    }

    cv::imshow("Original Image", image);
    cv::waitKey(0);

    // Sobel operator kernels for edge detection
    float h_kernelX[] = {
        -1, 0, 1,
        -2, 0, 2,
        -1, 0, 1
    };
    float h_kernelY[] = {
        -1, -2, -1,
        0, 0, 0,
        1, 2, 1
    };
    uchar *d_input, *d_output;
    float *d_kernelX, *d_kernelY;
    cudaMalloc((void **)&d_input, image.total());
    cudaMalloc((void **)&d_output, image.total());
    cudaMalloc((void **)&d_kernelX, sizeof(h_kernelX));
    cudaMalloc((void **)&d_kernelY, sizeof(h_kernelY));
    cudaMemcpy(d_input, image.data, image.total(), cudaMemcpyHostToDevice);
    cudaMemcpy(d_kernelX, h_kernelX, sizeof(h_kernelX), cudaMemcpyHostToDevice);
    cudaMemcpy(d_kernelY, h_kernelY, sizeof(h_kernelY), cudaMemcpyHostToDevice);

    dim3 blockSize(16, 16);
    dim3 gridSize((image.cols + blockSize.x - 1) / blockSize.x, (image.rows + blockSize.y - 1) / blockSize.y);

    sobelEdgeDetection<<<gridSize, blockSize>>>(d_input, d_output, image.cols, image.rows, d_kernelX,
d_kernelY, 3, 3);

    cv::Mat result(image.rows, image.cols, CV_8UC1);
    cudaMemcpy(result.data, d_output, result.total(), cudaMemcpyDeviceToHost);

    cv::imshow("Edge Detected Image", result);
    cv::waitKey(0);

    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_kernelX);
    cudaFree(d_kernelY);

    return 0;
}
```