Special thanks to Assoc Prof Nicholas Vun (Associate Chair in Academic)
for his initiative and contribution to this module

CE/CZ 3001:
Advanced Computer Architecture

Module 6: GPU Architecture and CUDA Programming

- Programming GPU with CUDA

Asst Prof Liu Weichen
School of Computer Science and Engineering
Nanyang Technological University, Singapore

Outline

- NVIDIA GPU internals
- CUDA C programming

NVIDIA GPUs

Modern GPUs are especially well-suited to address non-graphics problems

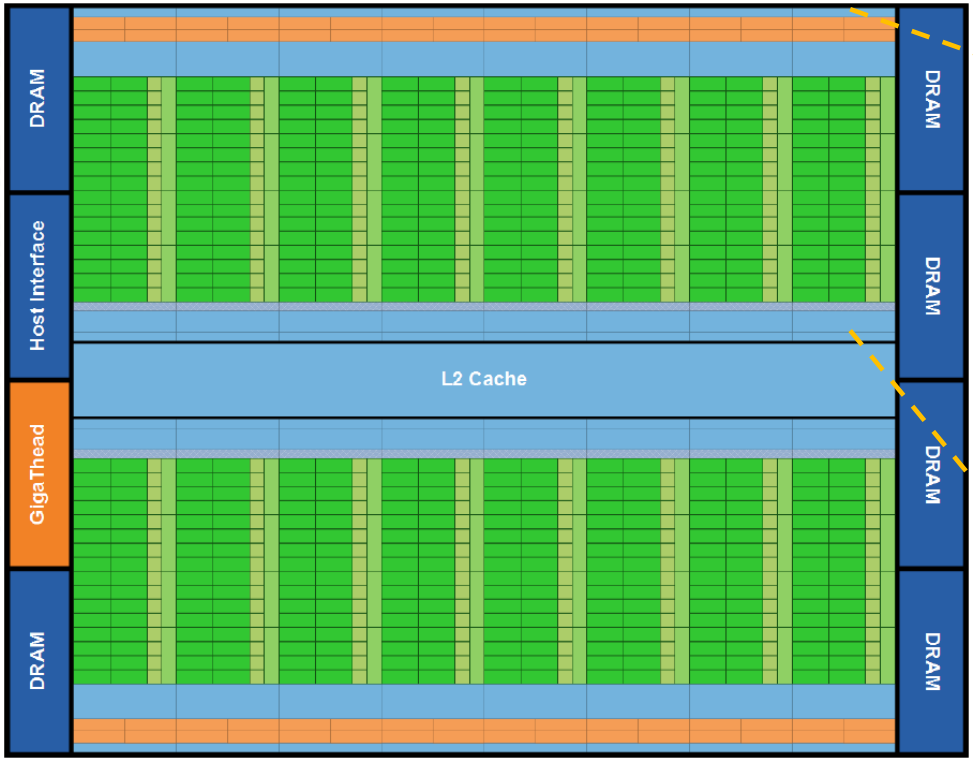
- that can be expressed as data-parallel computations with high arithmetic intensity (c.f. memory based operations)

NVIDIA CUDA enabled GPU families

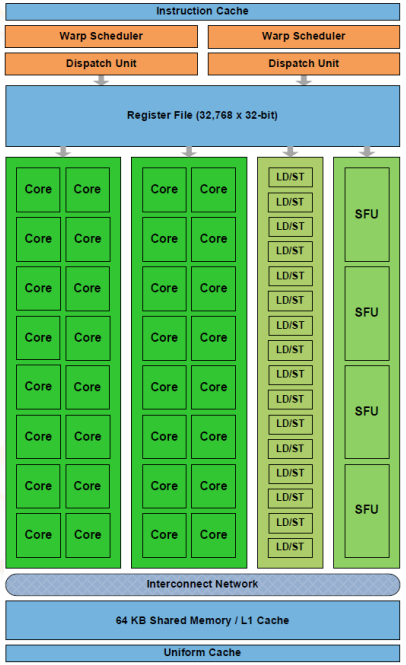
- CUDA programming platform
- Tesla (2008), Fermi, Kepler, Maxwell, Pascal, Volta, Turing (2018)

Overview of NVIDIA GPU Architecture (Fermi)

Consists of 16 Streaming Multiprocessors (SMs)



One SM

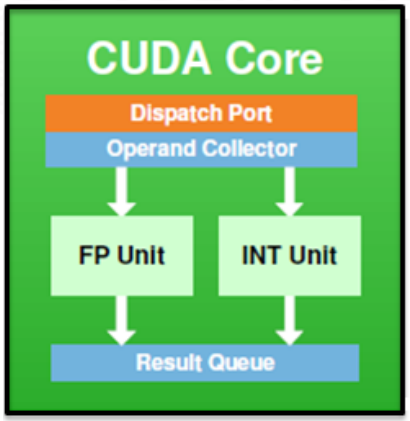


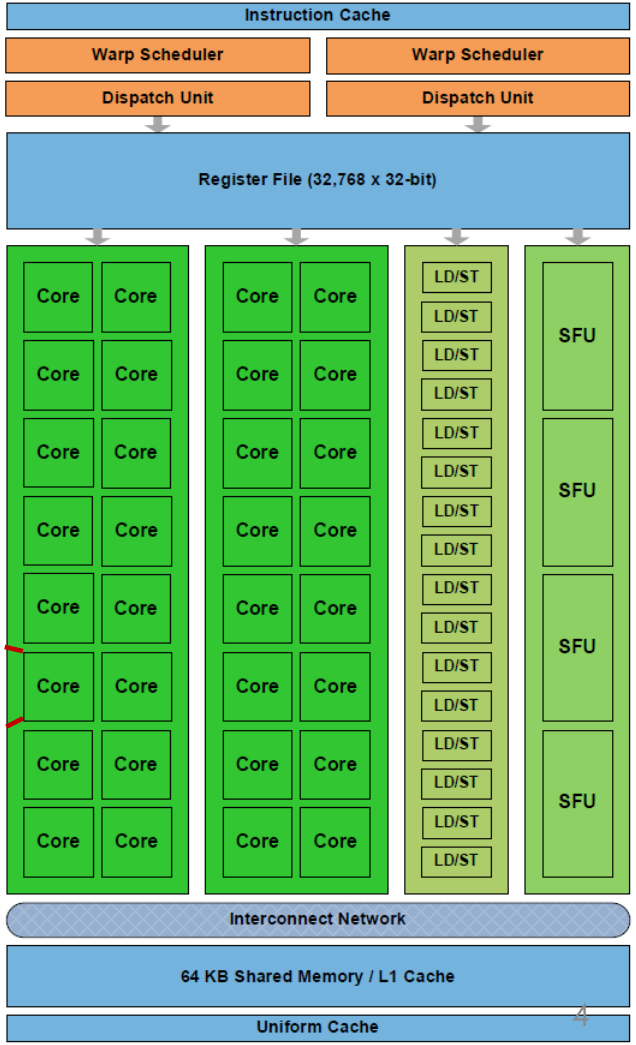
(Source: NVIDIA Fermi Compute Architecture Whitepaper)

NVIDIA Fermi Architecture

Each SM contains 32 compute engines known as CUDA cores

- each core has an (integer) ALU and FPU, which can execute a floating point or integer instruction per clock



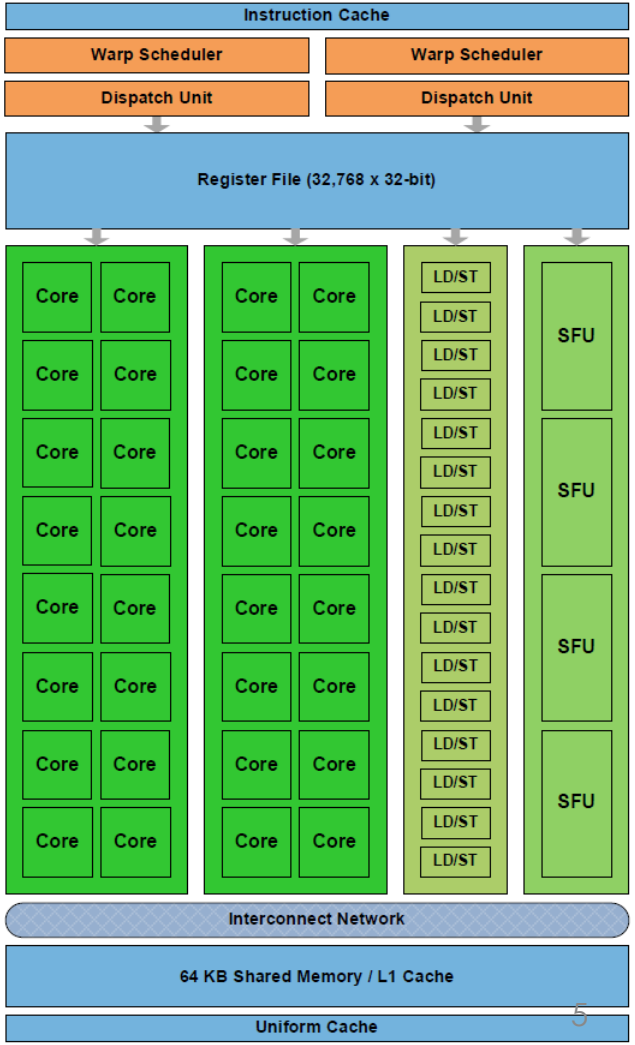


(Source: NVIDIA Fermi Compute Architecture Whitepaper)

NVIDIA Fermi Architecture

Each SM also has

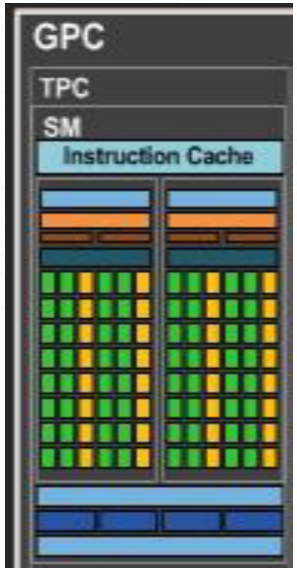
- 16 load/store units
- 4 Special Function Units (SFUs) – to execute transcendental instructions (sin, cosine, reciprocal, and square root).
- 64KB of configurable Shared Memory and L1 Cache.

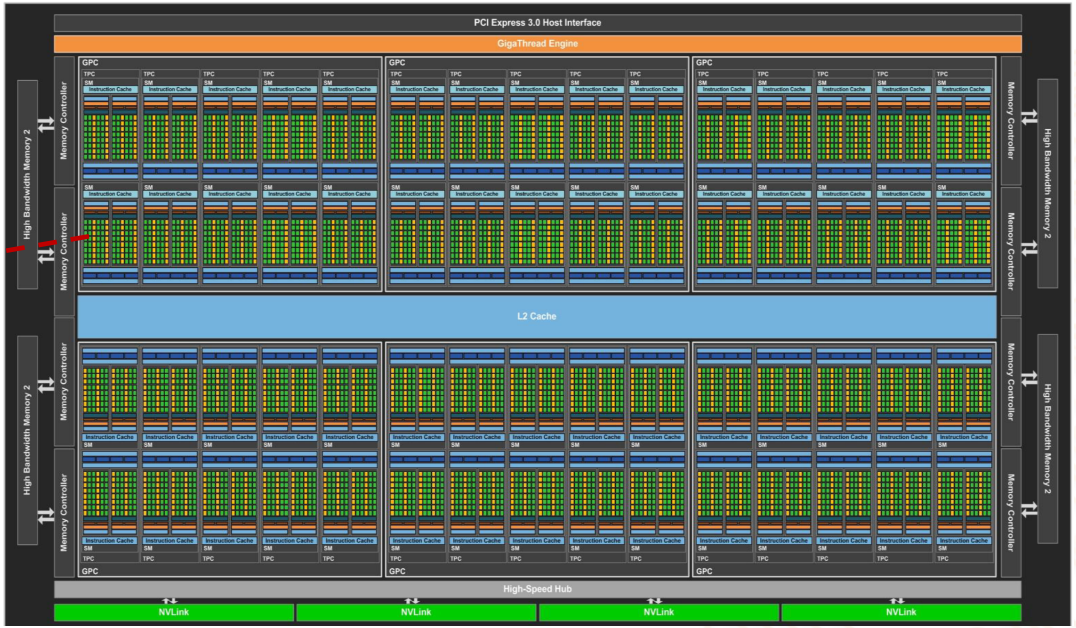


For comparison - NVIDIA Pascal GPU Architecture

Total of 60 SMs with 3840 CUDA cores, arranged in the form of

- 6 GPCs (Graphics Processing Clusters) that each contains 10 SMs, which in turn consists of 64 CUDA cores (per SM)

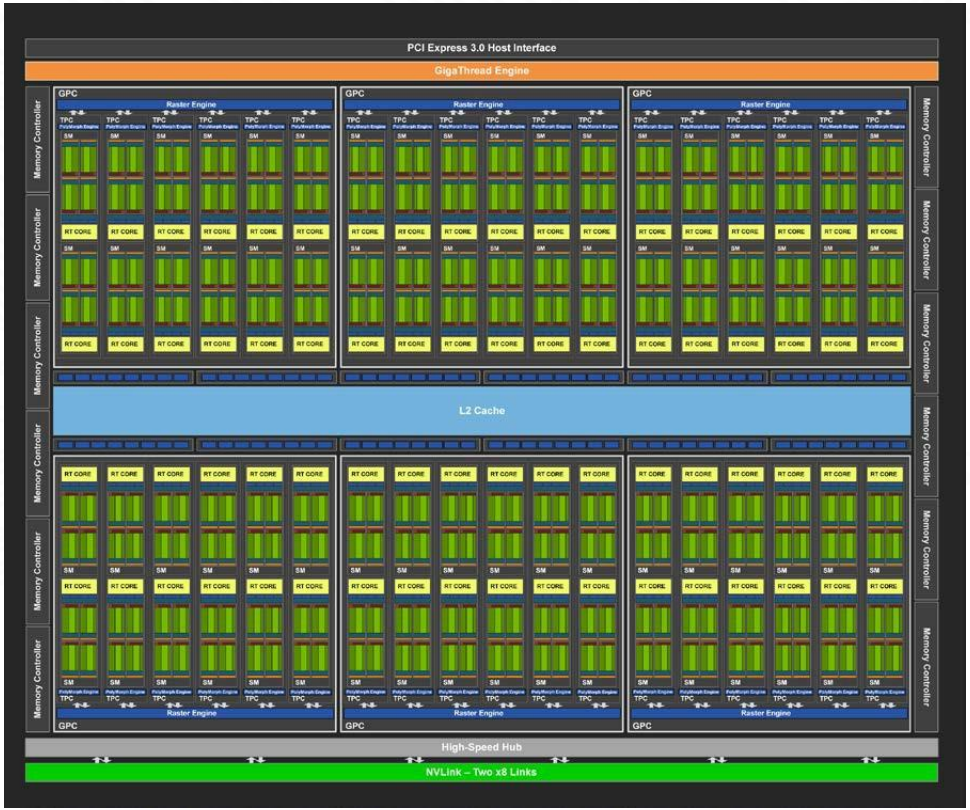


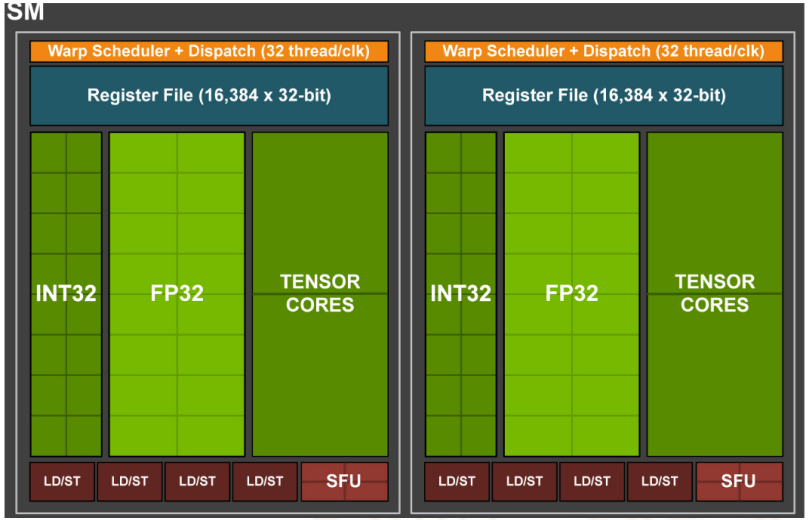


(Source: NVIDIA Pascal Architecture Whitepaper)

For comparison - NVIDIA Turing GPU Architecture

Total of 72 SMs with 4608 CUDA cores (and 576 Tensor cores for matrix operations)





(Source: NVIDIA Turing Architecture Whitepaper)

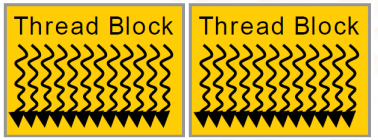
GPU Computing with CUDA Programming

CUDA is a general purpose parallel computing programming model where a problem is first expressed in the form of multiple threads

- and executed in parallel using the CUDA cores in the GPU
- enable many complex computational problems to be solved in a very efficient way

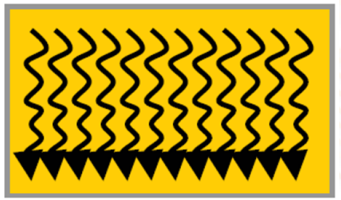
In a CUDA program

- the problem is first decomposed into sub-problems
- each of which is allocated to a 'Thread Block'



Each sub-problem is then separated into finer pieces

- that can be solved cooperatively in parallel by using multiple threads within the thread block.



CUDA Programming Model

CUDA operates on a heterogeneous programming model that consists of a host and a device

- host is typically a CPU and its memory (host memory)
- device is the GPU and its memory (device memory)

Host (CPU)

Device (GPU)

A program will start and run by the host

- which then launches the parallel threads that are executed on the device.

Basic CUDA C program - “Hello, World!”

```
1  int main(void){
2    printf(“Hello, World!”);
3    return 0;
4    }
```

This simple CUDA C program, which only runs on the host just consisting of standard C statements

- program file is stored with “`.cu`” extension
  e.g. hello_world.cu
- which is then compiled using the NVIDIA compiler nvcc
  e.g. nvcc hello_world.cu –o hello_world

So how do we specify the code for execution by the device?

Device Code (kernel)

To indicate the code that is to run on the device

- use the CUDA C keyword __global__ declaration specifier

```
1   __global__ void hello_GPU(void){
2     printf("Hello from GPU!");
3     }
```

Device code are launched as Kernel in CUDA

- which can be called from the host code as follows

```
4   int main(void){
5     hello_GPU<<<1,1>>>();
6     printf("Hello from CPU!");
7     return 0;
8     }
```

Behind the 'scenes'

During compilation

- the source file is split into host and device components

Device code is compiled by NVIDIA's compiler

Host code is compiled by using standard host compiler, such as gcc

```
1  __global__ void hello_GPU(void){
2    printf("Hello from GPU!");
3    }

4  int main(void){
5    hello_GPU<<<1,1>>>();
6    printf("Hello from CPU!");
7    return 0;
8    }
```

Kernel and Parallel Threads

A Kernel is an extended C function

- can be executed N times in parallel by N different CUDA threads when called (as opposed to only once for regular C functions).

The number of parallel CUDA threads launched for the kernel

- specified in the host code using the triple angle brackets syntax:

```
hello_dev<<<2,4>>>();
```

known as the execution configuration syntax

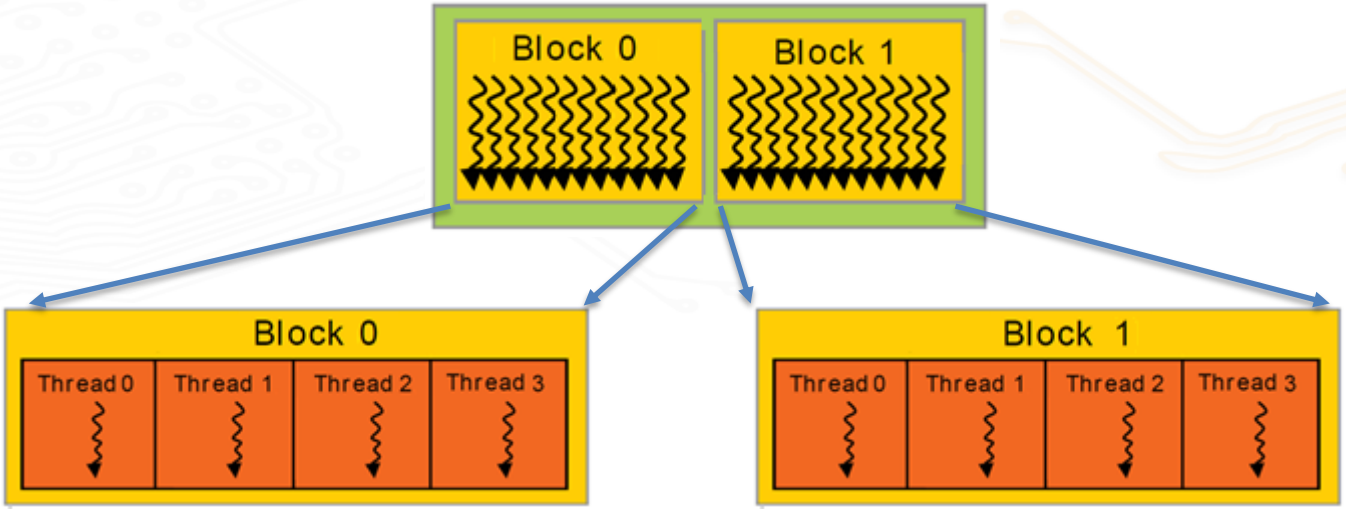
The two parameters between the brackets specifies how the threads are to be launched

- number of thread blocks (2 in the above example)
- number of threads in each block (4 in the above example)

Threads and Thread Blocks

Example: `kernel<<<2,4>>>();`

- will launch 2 thread blocks, each with 4 threads
  → total of 8 threads, executing 8 copies of the kernel



On latest (2019) NVIDIA GPUs

- a thread block may contain up to 1024 threads

Thread ID

How do we then identify the different threads ?

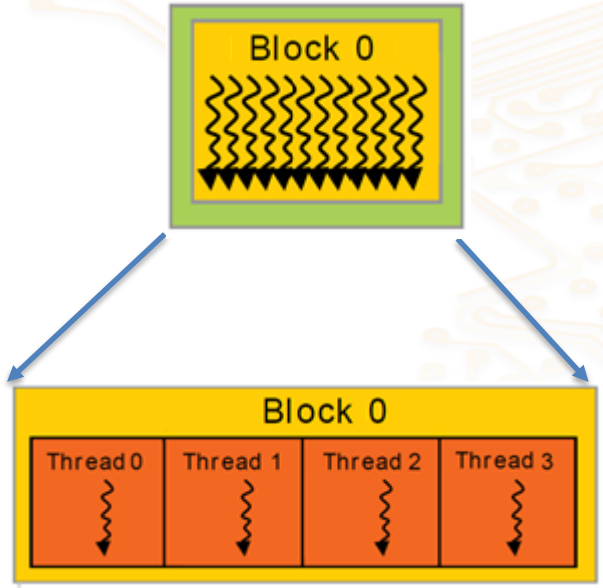
- each thread that executes a copy of the kernel is given a unique thread ID within the block
- accessible within the kernel through the built-in `threadIdx` variable

Example: Launching `hello_GPU<<<1,4>>>()`

```
1   __global__ void hello_GPU(void){
2     int i = threadIdx.x;
3     printf("Hello from GPU[i]!\n");
4     }
```

Output:

```
Hello from GPU[0]!
Hello from GPU[1]!
Hello from GPU[2]!
Hello from GPU[3]!
```



Block ID

With multiple thread blocks, each block that executes the kernel is also given a unique block ID

- accessible within the kernel through the built-in `blockIdx` variable

Threads among different blocks can then be identified through the `threadIdx` and `blockIdx`, together with built-in variable `blockDim` - which corresponds to the number of threads per block

Example: Launching `hello_GPU<<<2,4>>>()`

- 2 thread blocks with 4 threads in each block (i.e. `blockDim` = 4)

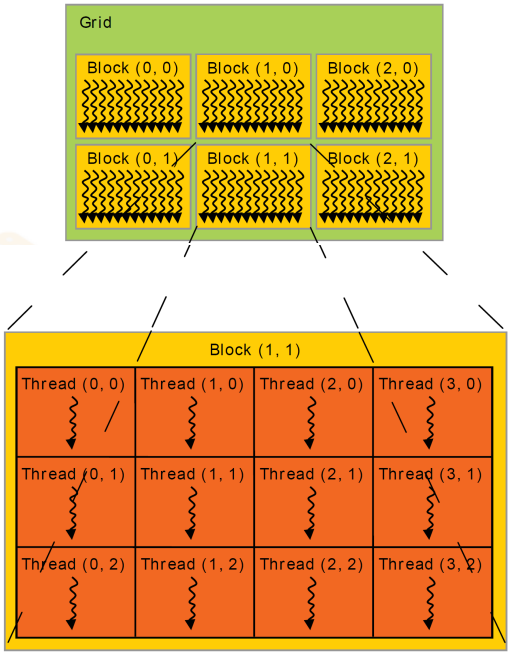
```
1   __global__ void hello_GPU(void){
2     int i = blockIdx.x * blockDim.x + threadIdx.x;
3     printf("Hello from GPU[i]!\n");
4     }
```

Multidimension Threads and Blocks

Why are the thread and Block IDs extended with "`.x`"?

CUDA threads and blocks can be defined to be of 1-dimensional (1D, with x only), 2D (with x and y) or 3D (with x, y and z)

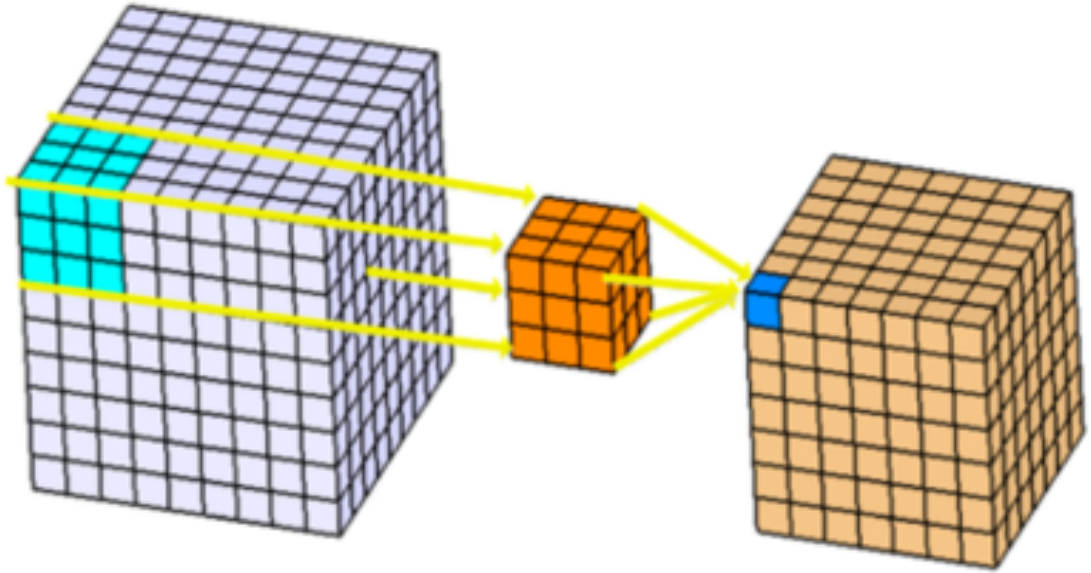
- which are suitable for addressing complex problem best described in multi-dimensions (e.g. Matrix).



(Source: NVIDIA CUDA C Programming Guide)

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
C[i][j] = A[i][j] + B[i][j];
```

3D Convolution



Synchronization between Host and Device

This program will most likely not work properly

- why?

```
1  __global__ void hello_GPU(void){
2    printf("Hello from GPU!");
3    }

4  int main(void){
5    hello_GPU<<<1,4>>>();
6    printf("Hello from CPU!");
7    return 0;
8    }
```

When we need the host to wait for all the threads to complete the kernels execution

- use the CUDA function `cudaDeviceSynchronize()`

Passing of Parameters (Example – Vector Addition)

In practice, we often need to pass parameter(s) from host to device for computation

- and collect the result(s) back from device to the host.

Consider the addition of 2 vectors ($\vec{\mathbf{a}}$+ $\vec{\mathbf{b}}$) such as: $\begin{bmatrix}7\\2\\3\end{bmatrix} + \begin{bmatrix}6\\4\\5\end{bmatrix}$

- result is a vector: ($\vec{\mathbf{a}}$ + $\vec{\mathbf{b}}$) = $\vec{\mathbf{c}}$, which is $\begin{bmatrix}7\\2\\3\end{bmatrix} + \begin{bmatrix}6\\4\\5\end{bmatrix} = \begin{bmatrix}13\\6\\8\end{bmatrix}$

Vector Addition – Host only

```
1  int main(void){
2    int N = 3;
3    int a[N] = {7,2,3};
4    int b[N] = {6,4,5};
5    int c[N];
6
7    vector_add(&c[0], &a[0], &b[0], N);
8    return 0;
9    }
```

```
1  void vector_add(int *h_c, int *h_a, int *h_b, int n){
2    for (int i = 0; i < n; i++)
3      h_c[i] = h_a[i] + h_b[i];
4    }
```

$(\vec{\mathbf{a}} + \vec{\mathbf{b}}) = \vec{\mathbf{c}}$,

Memory management between Host and Device

This vector addition is obviously most suitable to be executed by using the GPU

- by launching multiple (3) threads $\begin{bmatrix}7\\2\\3\end{bmatrix} + \begin{bmatrix}6\\4\\5\end{bmatrix} = \begin{bmatrix}13\\6\\8\end{bmatrix}$

But we need to first pass the data from the host to the device (GPU)

- which is done using the CUDA memory management functions:
  `cudaMalloc(), cudaMemcpy(), cudaFree()`

- similar to the standard C equivalent functions:
  `malloc(), memcpy(), free()`

Vector Addition – Host code

```
1    int main(void){
2      int N = 3;
3      int a[N] = {7,2,3};
4      int b[N] = {6,4,5};
5      int c[N];
```

$$\begin{bmatrix}7\\2\\3\end{bmatrix} + \begin{bmatrix}6\\4\\5\end{bmatrix} = \begin{bmatrix}13\\6\\8\end{bmatrix}$$

Vector Addition – Host code

```
1     int main(void){
2       int N = 3
3       int a[N] = {7,2,3};
4       int b[N] = {6,4,5};
5       int c[N];
6
7       int *d_a, *d_b, *d_c;
8       cudaMalloc((void**)&d_a, sizeof(int)*N);
:          : // repeat for d_b and d_c
```

Vector Addition – Host code

```
1     int main(void){
2       int N = 3
3       int a[N] = {7,2,3};
4       int b[N] = {6,4,5};
5       int c[N];
6
7       int *d_a, *d_b, *d_c;
8       cudaMalloc((void**)&d_a, sizeof(int)*N);
:          : // repeat for d_b and d_c
11      cudaMemcpy(d_a, a, sizeof(int)*N, cudaMemcpyHostToDevice);
12      cudaMemcpy(d_b, b, sizeof(int)*N, cudaMemcpyHostToDevice);
```

Vector Addition – Host code

```
1     int main(void){
2       int N = 3
3       int a[N] = {7,2,3};
4       int b[N] = {6,4,5};
5       int c[N];
6
7       int *d_a, *d_b, *d_c;
8       cudaMalloc((void**)&d_a, sizeof(int)*N);
:          : // repeat for d_b and d_c
11      cudaMemcpy(d_a, a, sizeof(int)*N, cudaMemcpyHostToDevice);
12      cudaMemcpy(d_b, b, sizeof(int)*N, cudaMemcpyHostToDevice);

13      vector_add_cu<<<1,1>>>(d_c, d_a, d_b, N); // note: 1 thread
```

Vector Addition – Host code

```
1     int main(void){
2       int N = 3
3       int a[N] = {7,2,3};
4       int b[N] = {6,4,5};
5       int c[N];
6
7       int *d_a, *d_b, *d_c;
8       cudaMalloc((void**)&d_a, sizeof(int)*N);
:          : // repeat for d_b and d_c
11      cudaMemcpy(d_a, a, sizeof(int)*N, cudaMemcpyHostToDevice);
12      cudaMemcpy(d_b, b, sizeof(int)*N, cudaMemcpyHostToDevice);

13      vector_add_cu<<<1,1>>>(d_c, d_a, d_b, N); // note: 1 thread

14      cudaMemcpy(c, d_c, sizeof(int)*N, cudaMemcpyDeviceToHost);
```

Vector Addition – Host code

```
1     int main(void){
2       int N = 3
3       int a[N] = {7,2,3};
4       int b[N] = {6,4,5};
5       int c[N];
6
7       int *d_a, *d_b, *d_c;
8       cudaMalloc((void**)&d_a, sizeof(int)*N);
:          : // repeat for d_b and d_c
11      cudaMemcpy(d_a, a, sizeof(int)*N, cudaMemcpyHostToDevice);
12      cudaMemcpy(d_b, b, sizeof(int)*N, cudaMemcpyHostToDevice);

13      vector_add_cu<<<1,1>>>(d_c, d_a, d_b, N); // note: 1 thread

14      cudaMemcpy(c, d_c, sizeof(int)*N, cudaMemcpyDeviceToHost);
15      cudaFree(d_a);
:          : // repeat for d_b and d_c
18      }
```

Vector Addition – Device (kernel) code

The corresponding kernel code:

```
1    __global__
2    void vector_add_cu(int *d_c, int *d_a, int *d_b, int n){
3       for (int i = 0; i<n; i++)
4           d_c[i] = d_a[i] + d_b[i];
5       }
```

- which is the same as the earlier host code:

```
1    void vector_add(int *h_c, int *h_a, int *h_b, int n){
2       for (int i = 0; i < n; i++)
3          h_c[i] = h_a[i] + h_b[i];
4       }
```

But - there is no parallel operation!

Vector Addition – Using Multiple Threads

Using multiple (3) threads to compute the vector addition in parallel.

From the host

- launch 1 thread block with 3 threads

```
13  vector_add_cu<<<1,3>>>(d_c, d_a, d_b);
```

- or launch 3 thread blocks, each containing 1 thread

```
13  vector_add_cu<<<3,1>>>(d_c, d_a, d_b);
```

- Which one is more suitable - which one should we use?
- And how does each thread know the entries that it should compute?

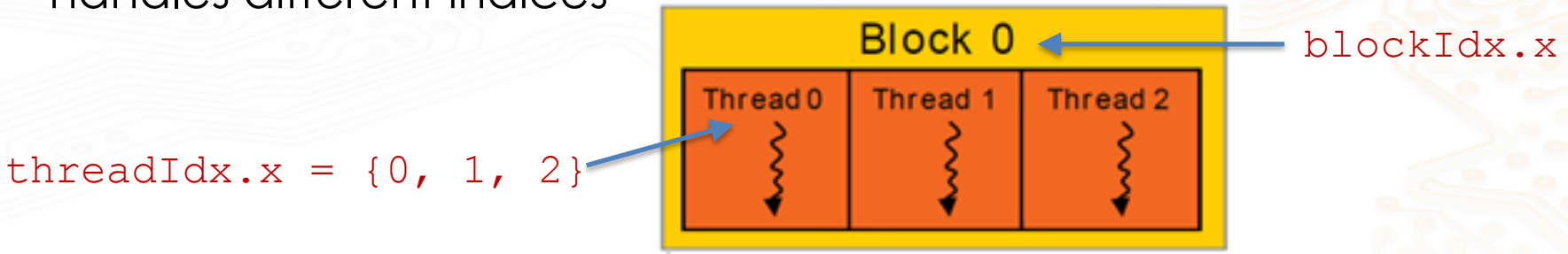
Specifying Thread using Thread ID

With multiple threads running

- need to specify to each thread which elements (indices) of the vectors that it should compute

If we launch 1 thread block with 3 threads

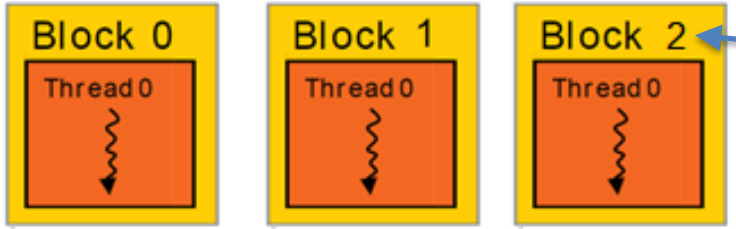
- use `threadIdx.x` to index the vector elements, each thread handles different indices




```
1    __global__
2    void vector_add_cu(int *d_c, int *d_a, int *d_b, int n){
3       d_c[threadIdx.x] = d_a[threadIdx.x] + d_b[threadIdx.x];
4       }
```

Block ID

If we launch 3 thread blocks with 1 thread each:



- use `blockIdx.x` to index the vector elements, each block handles different indices

```
1   __global__
2   void vector_add_cu(int *d_c, int *d_a, int *d_b, int n){
3      d_c[blockIdx.x] = d_a[blockIdx.x] + d_b[blockIdx.x];
4      }
```

The kernel code above may not always work well in practice – why?

Thread ID with Block ID

In practice, we often use multiple blocks, each with multiple threads

- many problems lend themselves to 2D or 3D interpretation of the data

We hence identify a thread by combining its built-in variables

- `threadIdx` and `blockIdx` together with `blockDim`

```
1    __global__
2    void vector_add_cu(int *d_c, int *d_a, int *d_b, int n){
3       int i = blockIdx.x * blockDim.x + threadIdx.x;
4       d_c[i] = d_a[i] + d_b[i];
5       }
```

Q. What is the value of `blockDim.x` with the following Kernel call?

```
     vector_add_cu<<<3,2>>>(d_c, d_a, d_b);
```

Blocks vs Threads

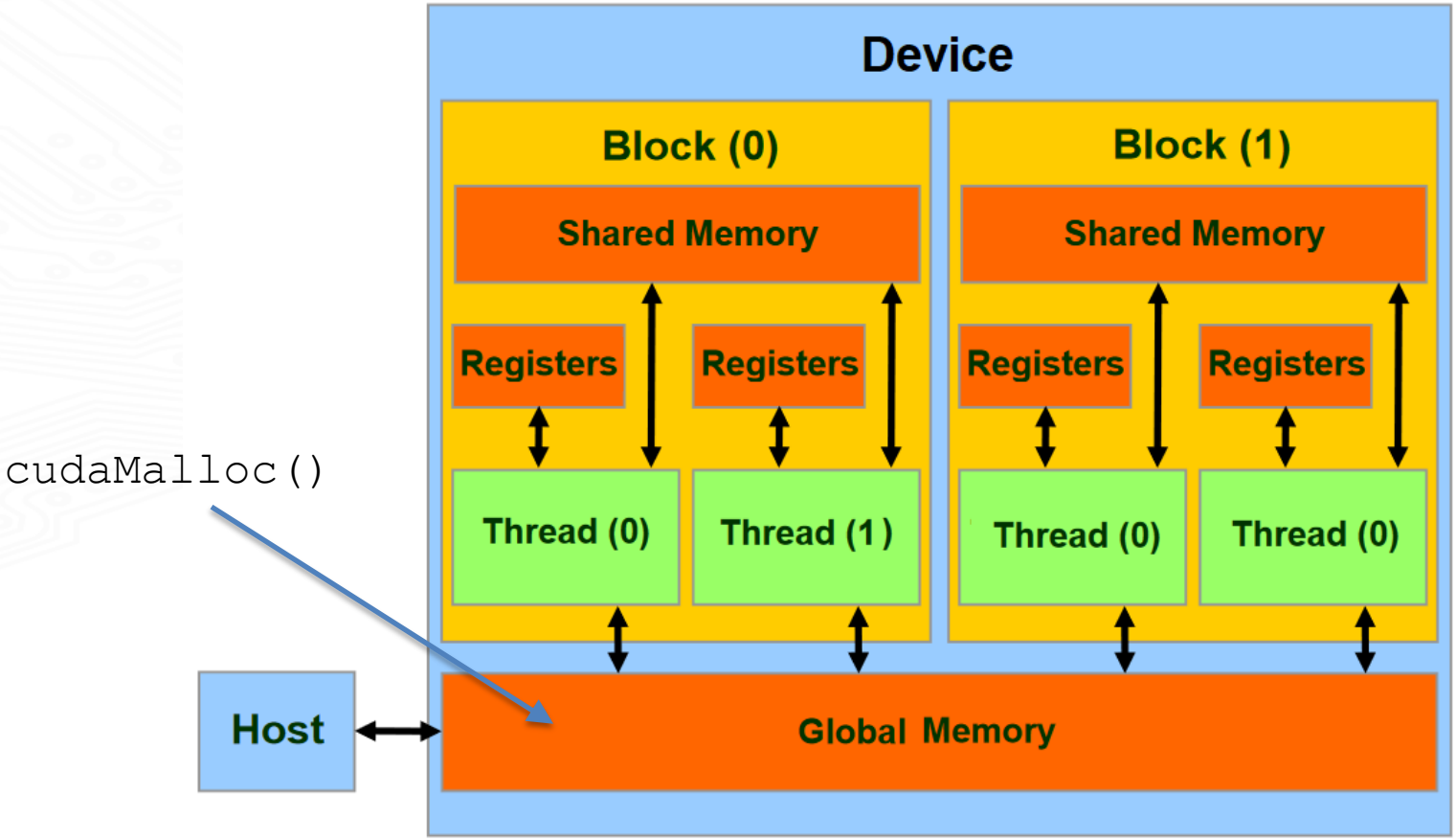
Back to the earlier question - which one should we use?

- multiple parallel blocks with 1 thread each?
- 1 block with multiple parallel threads?

Parallel threads within a block have the addition mechanisms to

- directly communicate and synchronize with each other
- which are usually required in many real-world applications

Memory Hierarchy - Blocks vs Threads



Example: Dot Product Computation

Given two vectors $\vec{\mathbf{a}}$ and $\vec{\mathbf{b}}$, their dot product is defined as

$$\vec{\mathbf{a}} \cdot \vec{\mathbf{b}} = |\vec{\mathbf{a}}| \times |\vec{\mathbf{b}}| \times \cos(\theta)$$

- we can hence find the separation (or angle θ) between the two vectors by calculating

$$\theta = \cos^{-1}\{(|\vec{\mathbf{a}}| \times |\vec{\mathbf{b}}|) / (\vec{\mathbf{a}} \cdot \vec{\mathbf{b}})\}$$

Dot products $(\vec{\mathbf{a}} \cdot \vec{\mathbf{b}})$ of the two vectors (which is a scalar)

- sum of the pairwise products of the vectors' elements

$$\vec{\mathbf{a}} \cdot \vec{\mathbf{b}} = \sum_{i=0}^{n} a_i b_i = a_0 * b_0 + a_1 * b_1 + \ldots + a_{n-1} * b_{n-1}$$

Dot Product Parallel Computation

$$\vec{\mathbf{a}} \cdot \vec{\mathbf{b}} = \sum_{i=0}^{n} a_i b_i = a_0 * b_0 + a_1 * b_1 + \ldots + a_{n-1} * b_{n-1}$$

The dot product can be effectively computed by launching parallel threads,

- each thread computes the product of corresponding pair of elements of the two vectors
- then add (by using one of the threads) all the products to find the final sum

Example: for two 3-element vectors

$$\vec{\mathbf{a}} \cdot \vec{\mathbf{b}} = \sum_{i=0}^{2} a_i b_i = a_0 * b_0 + a_1 * b_1 + a_2 * b_2$$

Dot Product Parallel Computation

$$\vec{\mathbf{a}} \cdot \vec{\mathbf{b}} = \sum_{i=0}^{2} a_i b_i = a_0 * b_0 + a_1 * b_1 + a_2 * b_2$$

```
1   __global__
2   void dot_prod_cu(int *d_c, int *d_a, int *d_b){
3     tmp[3];
4
5     tmp[i] = d_a[i] * d_b[i];
6
7
8
9
10
11
12
13
```

Dot Product Parallel Computation

$$\vec{\mathbf{a}} \cdot \vec{\mathbf{b}} = \sum_{i=0}^{2} a_i b_i = a_0 * b_0 + a_1 * b_1 + a_2 * b_2$$

```
1   __global__
2   void dot_prod_cu(int *d_c, int *d_a, int *d_b){
3     tmp[3];
4
5     tmp[i] = d_a[i] * d_b[i];
6
7     if (i==0){
8       int sum = 0;
9       for (int j = 0; j < 3; j++)
10        sum = sum + tmp[j];
11      *d_c = sum;
12      }
13    }
```

Dot Product Parallel Computation

$$\vec{a} \cdot \vec{b} = \sum_{i=0}^{2} a_i b_i = a_0 * b_0 + a_1 * b_1 + a_2 * b_2$$

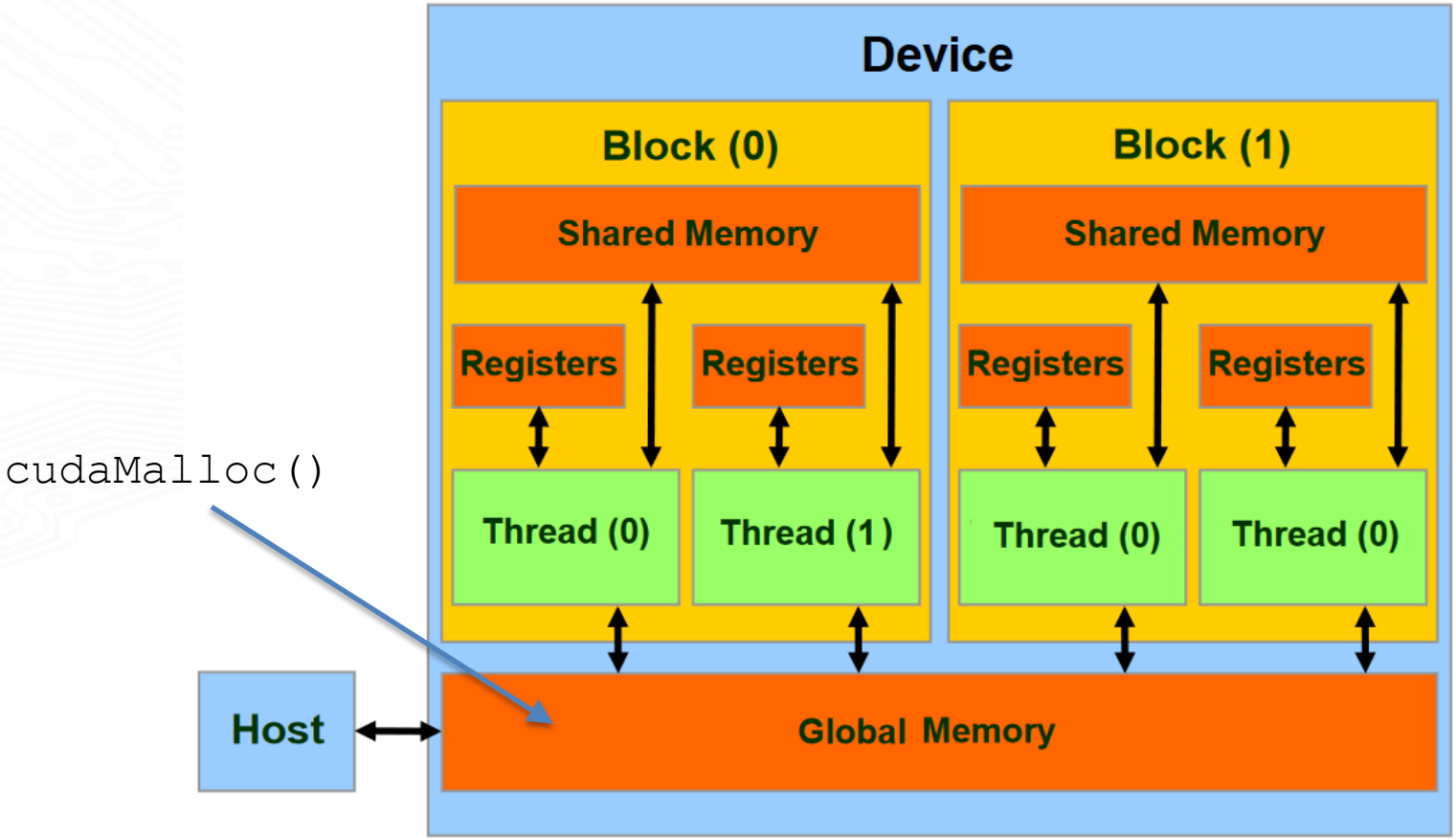
```
1   __global__
2   void dot_prod_cu(int *d_c, int *d_a, int *d_b){
3     tmp[3];
4     int i = ?
5     tmp[i] = d_a[i] * d_b[i];
6
7     if (i==0){
8       int sum = 0;
9       for (int j = 0; j < 3; j++)
10        sum = sum + tmp[j];
11      *d_c = sum;
12      }
13    }
```

Dot Product Parallel Computation

```
Dot_prod_cu<<<3,1>>>(d_c, d_a, d_b);
```

```
1   __global__
2   void dot_prod_cu(int *d_c, int *d_a, int *d_b){
3     int tmp[3];
4     int i = blockIdx.x;
5     tmp[i] = d_a[i] * d_b[i];
6
7     if (i==0){
8       int sum = 0;
9       for (int j = 0; j < 3; j++)
10        sum = sum + tmp[j];
11      *d_c = sum;
12      }
13    }
```

Memory Hierarchy - Blocks vs Threads



Dot Product Parallel Computation

```
Dot_prod_cu<<<1,3>>>(d_c, d_a, d_b);
```

```
1   __global__
2   void dot_prod_cu(int *d_c, int *d_a, int *d_b){
3     int tmp[3];
4     int i = threadIdx.x;
5     tmp[i] = d_a[i] * d_b[i];
6
7     if (i==0){
8       int sum = 0;
9       for (int j = 0; j < 3; j++)
10        sum = sum + tmp[j];
11      *d_c = sum;
12      }
13    }
```

Sharing Data between Threads

Need to be able to access each other data

- use shared memory – declared as `__shared__`

```
1   __global__
2   void dot_prod_cu(int *d_c, int *d_a, int *d_b){
3       __shared__ int tmp[3];
4     int i = threadIdx.x;
5     tmp[i] = d_a[i] * d_b[i];
6
7     if (i==0){
8       int sum = 0;
9       for (int j = 0; j < 3; j++)
10        sum = sum + tmp[j];
11      *d_c = sum;
12      }
13    }
```

Synchronize between Threads

Need to synchronize

- Thread may not be ready to share its data
- use `__syncthreads()`

```
1   __global__
2   void dot_prod_cu(int *d_c, int *d_a, int *d_b){
3     __shared__ int tmp[3];
4     int i = threadIdx.x;
5     tmp[i] = d_a[i] * d_b[i];
6     __syncthreads();
7     if (i==0){
8       int sum = 0;
9       for (int j = 0; j < 3; j++)
10        sum = sum + tmp[j];
11      *d_c = sum;
12      }
13    }
```

Summary

- CUDA C is based on standard C, with extension to support (NVIDIA) GPU based parallel programming.
- Consists of a host and a device.
- Provide a general purpose parallel computing programming model based on using multiple threads running in parallel on the device.
- Kernel entry point is indicated by using the `__global__` declaration specifier.
- Kernel is launched as parallel threads organized in Thread Blocks.
- Threads within the same block can share memories to pass data and synchronize with each other.