PARALLEL PROGRAMMING...

Parallel Computing Using CUDA

SESSION 3/3

GOAL

Programming Interface for parallel computing With CUDA

- What is CUDA ?
- Concepts
  - Running in parallel (Blocks)
  - Introduction Threads
  - Combining Threads & Blocks (Indexing)
  - Cooperating Threads (Shared memory)
  - Managing the device (Asynchronous operation)

API Examples

What is CUDA ?

What is CUDA ?

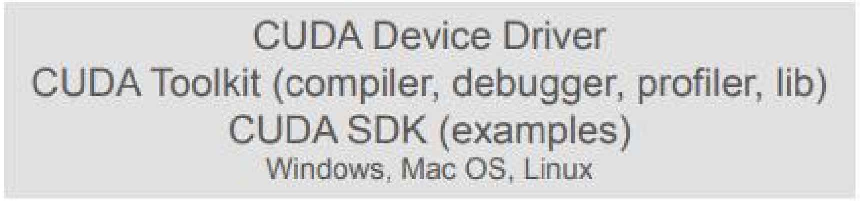
A proprietary platform developed by Nvidia that allows programmers to write C/C++ code that runs directly on Nvidia GPUs.

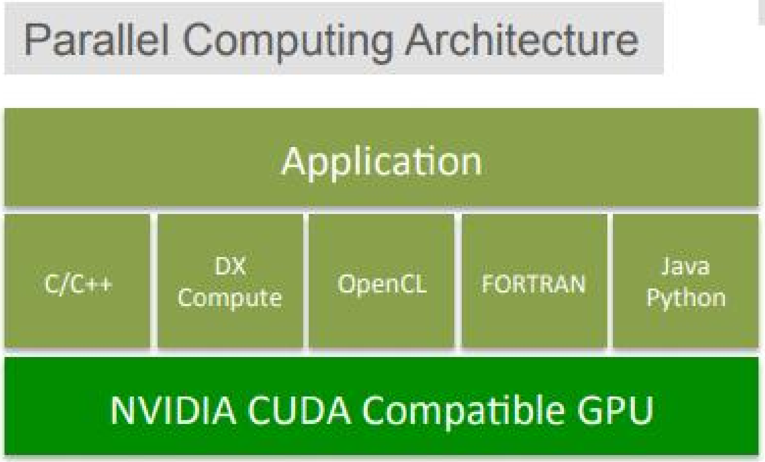
It also provides libraries and tools for various domains such as linear algebra, image processing, deep learning, etc.

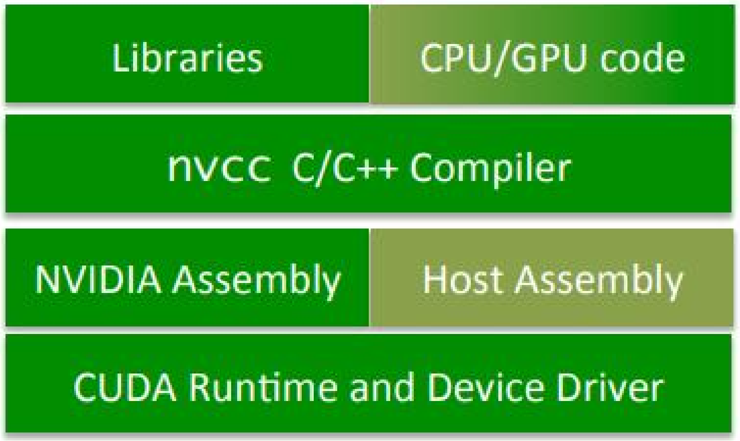
- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance

- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.

CUDA

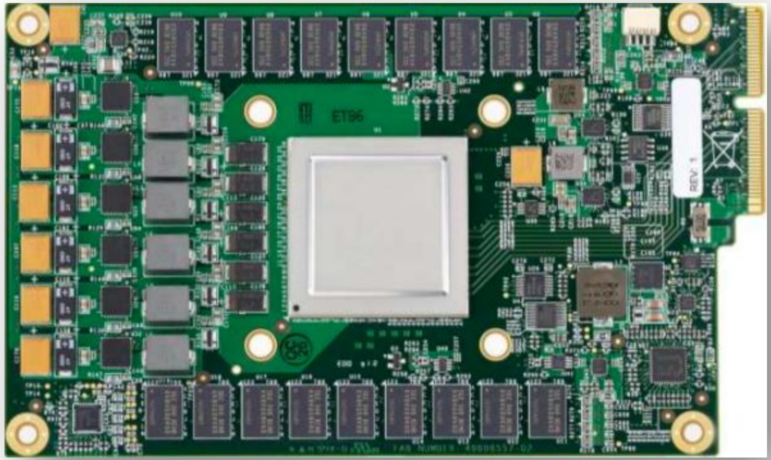

CUDA Device Driver
CUDA Toolkit (compiler, debugger, profiler, lib)
CUDA SDK (examples)
Windows, Mac OS, Linux





Libraries – FFT, Sparse Matrix, BLAS, RNG, CUSP, Thrust…

GPU (Graphics Processing Unit)



A **GPU** is uses to **speed up the process** of creating and rendering computer graphics, designed to accelerate graphics and image processing.

It is the most important hardware.

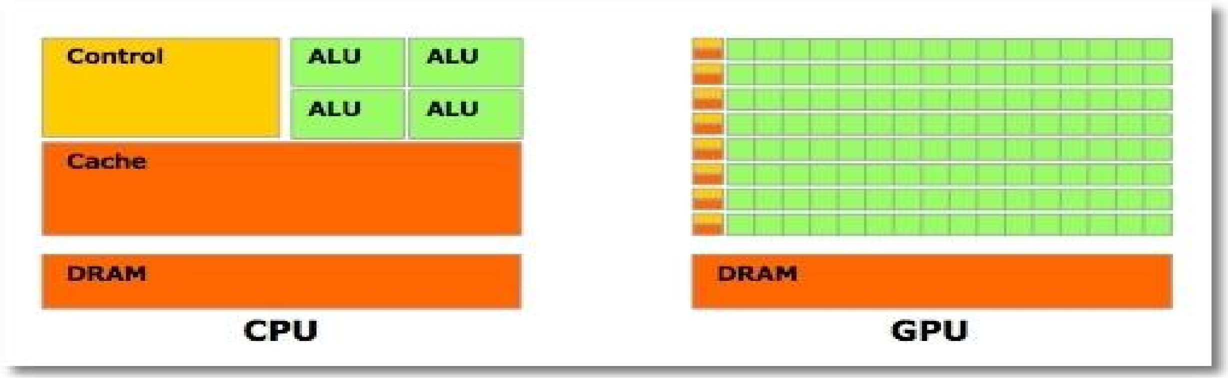
But have later been used for ***non-graphic calculations*** involving embarrassingly parallel problems due to their parallel structure.

What is GPGPU ?

- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application
- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain SIMD parallelism
  - Low-latency floating point (FP) computation
- Applications – see //GPGPU.org
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

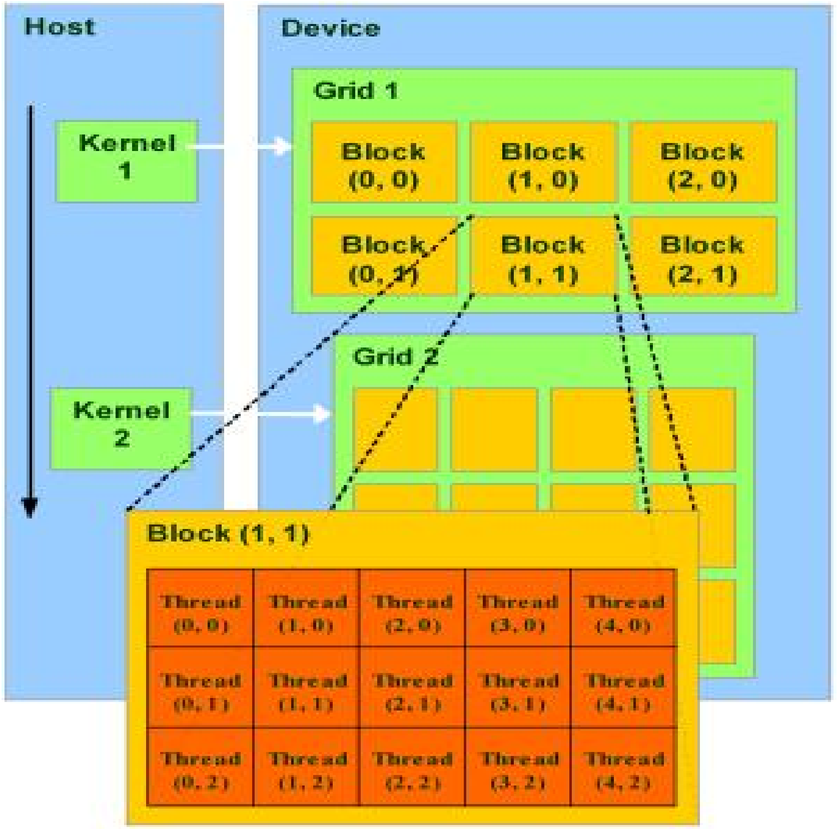
CUDA Goals

- Scale code to hundreds of cores running thousands of threads
- The task runs on the GPU independently from the CPU



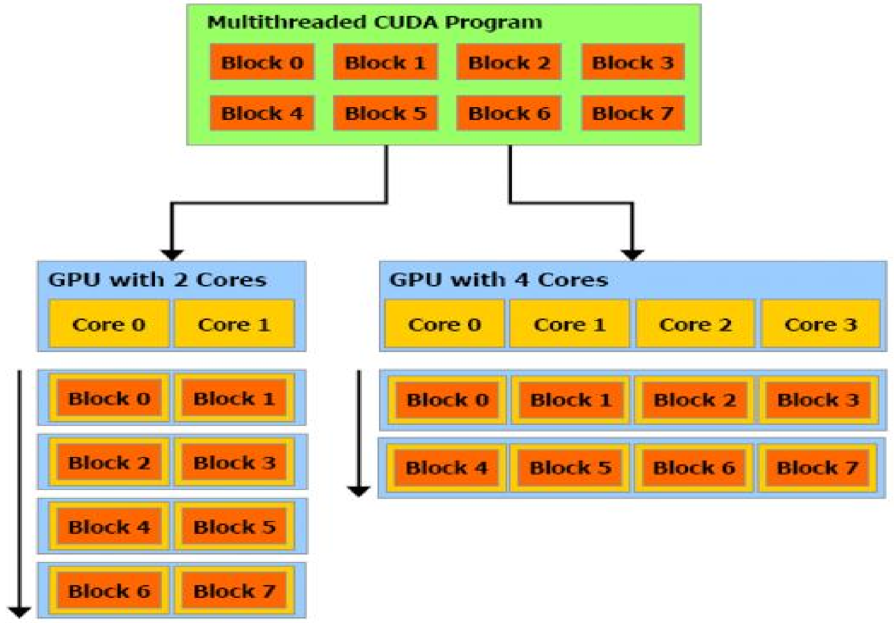
CUDA Structure

- Threads are grouped into thread blocks
- Blocks are grouped into a single grid
- The grid is executed on the GPU as a kernel



CUDA Scalability

- Blocks map to cores on the GPU
- Allows for portability when changing hardware



CUDA Memory Tranfer

cudaMemcpy(void *dst,   void *src,   size_t nbytes, enum cudaMemcpyKind direction);

- returns after the copy is complete blocks CPU
- thread doesn’t start copying until previous CUDA calls complete

enum cudaMemcpyKind

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

CUDA Host Synchronization

All kernel launches are asynchronous

- control returns to CPU immediately
- kernel starts executing once all previous CUDA calls have completed

Memcopies are synchronous

- control returns to CPU once the copy is complete
- copy starts once all previous CUDA calls have completed

cudaThreadSynchronize()

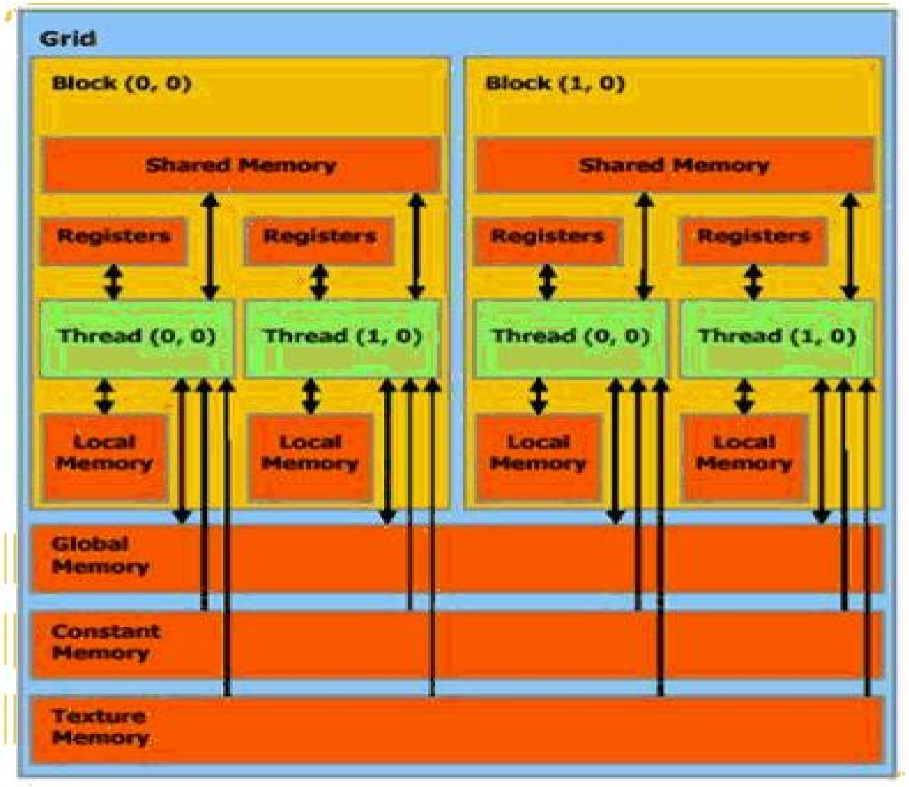
- blocks until all previous CUDA calls complete

Asynchronous CUDA calls provide:

- non-blocking memcopies
- ability to overlap memcopies and kernel execution

CUDA Memory Hierarchy

- Shared memory much much faster than global
- Don’t trust local memory
- Global, Constant, and Texture memory available to both host and cpu



CUDA Global and Shared Memory

Global memory not cached on G8x GPUs

- High latency, but launching more threads hides latency
- Important to minimize accesses
- Coalesce global memory accesses (more later)

Shared memory is on-chip, very high bandwidth

- Low latency (100-150times faster than global memory)
- Like a user-managed per-multiprocessor cache
- Try to minimize or avoid bank conflicts (more later)

CUDA Texture and Constant Memory

Texture partition is cached

- Uses the texture cache also used for graphics
- Optimized for 2D spatial locality
- Best performance when threads of a warp read locations that are close together in 2D

Constant memory is cached

- 4 cycles per address read within a single warp
- Total cost 4 cycles if all threads in a warp read same address
- Total cost 64 cycles if all threads read different addresses

CUDA Coalescing

- A coordinated read by a half-warp (16threads)
- A contiguous region of global memory:
  - 64bytes - each thread reads a word: int, float, …
  - 128bytes - each thread reads a double-word: int2, float2, …
  - 256bytes - each thread reads a quad-word: int4, float4, …

Additional restrictions:

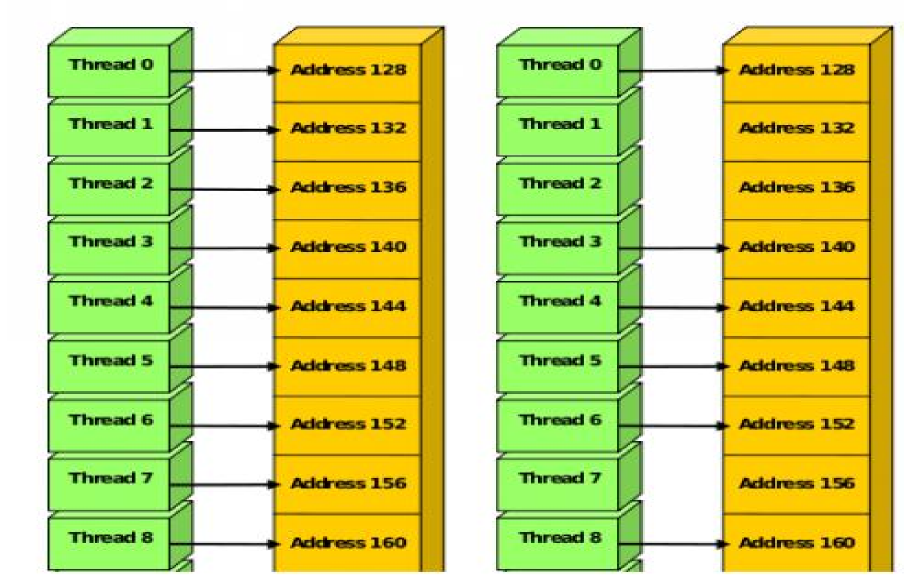
- Starting address for a region must be a multiple of region size
- The kth thread in a half-warp must access the kth element in a block being read

Exception: not all threads must be participating

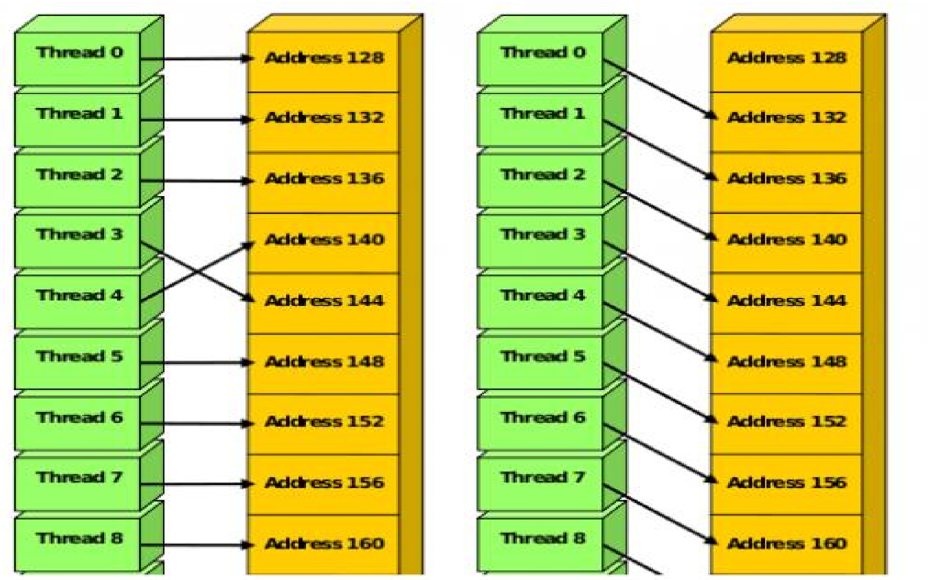
- Predicated access, divergence within a halfwarp

CUDA Coalescing

Coalesced memory accesses

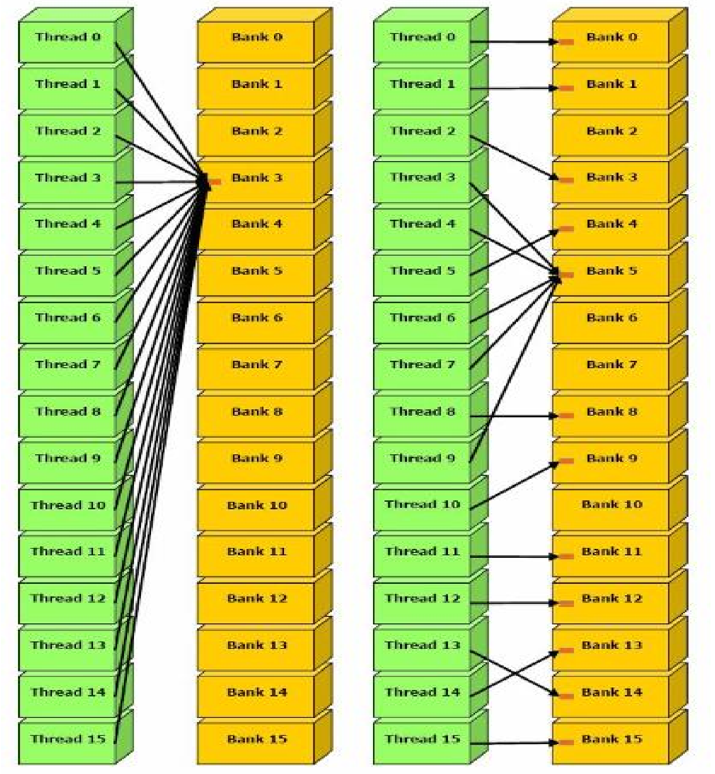


Uncoalesced memory accesses



CUDA Bank Conflicts

- Shared memory is divided into banks
- Each bank has serial read/write access
- Bank addresses are striped
- If more than one thread attempts to access the same bank at the same time, they're accesses are serialized
- This is a bank conflict



CUDA Concepts

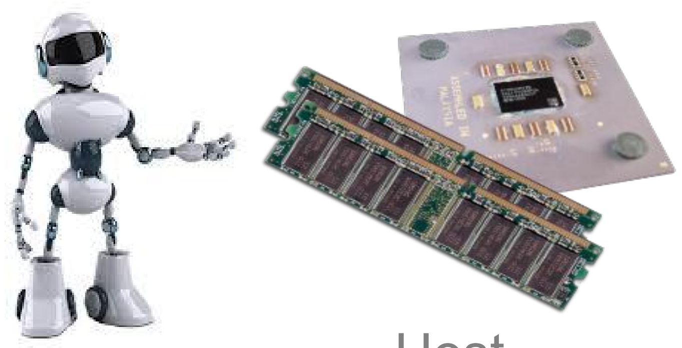
CUDA CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

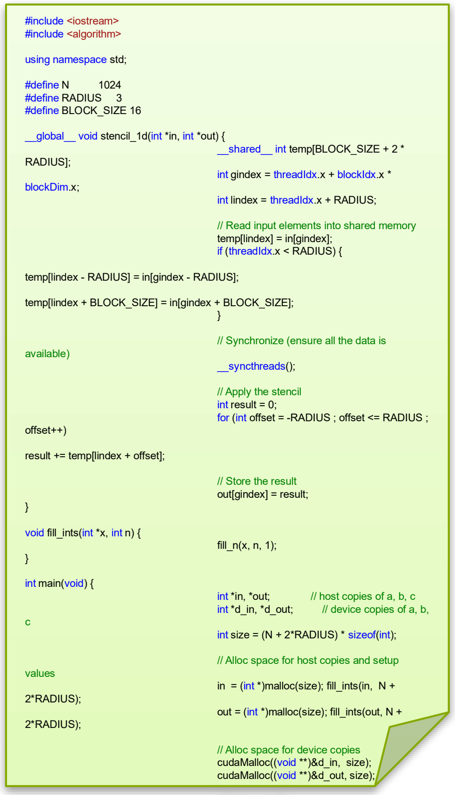
CUDA: Heterogeneous Computing

- Terminology:

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


Host

Device



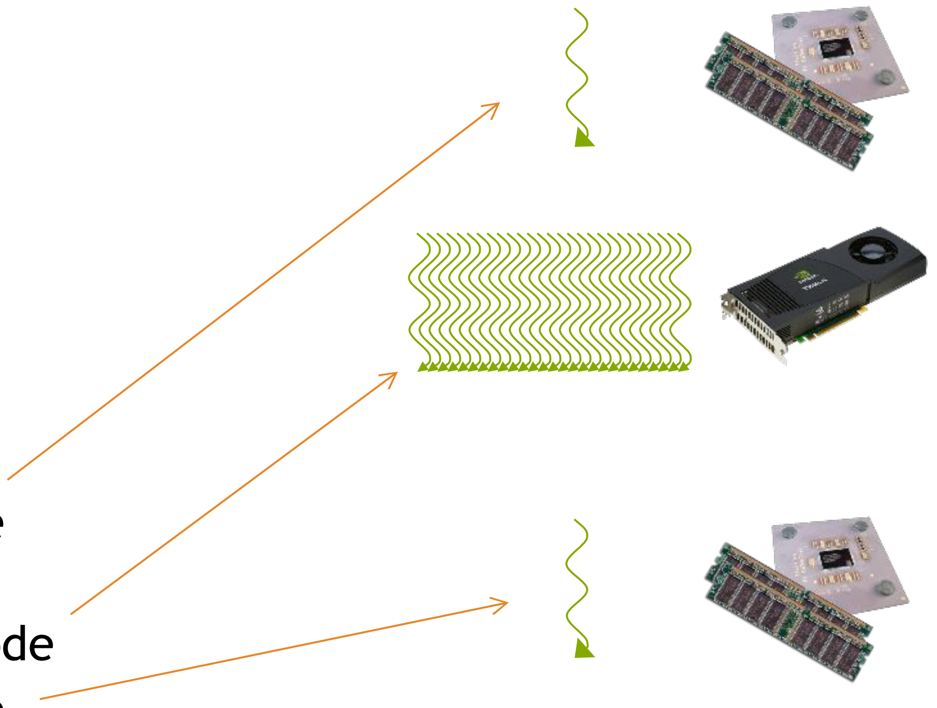
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
                                    __shared__ int temp[BLOCK_SIZE + 2 *
RADIUS];
                                    int gindex = threadIdx.x + blockIdx.x *
blockDim.x;
                                    int lindex = threadIdx.x + RADIUS;

                                    // Read input elements into shared memory
                                    temp[lindex] = in[gindex];
                                    if (threadIdx.x < RADIUS) {

temp[lindex - RADIUS] = in[gindex - RADIUS];

temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
                                    }

                                    // Synchronize (ensure all the data is
available)
                                    __syncthreads();

                                    // Apply the stencil
                                    int result = 0;
                                    for (int offset = -RADIUS ; offset <= RADIUS ;
offset++)

result += temp[lindex + offset];

                                    // Store the result
                                    out[gindex] = result;
}

void fill_ints(int *x, int n) {
                                    fill_n(x, n, 1);
}

int main(void) {
                                    int *in, *out;          // host copies of a, b, c
                                    int *d_in, *d_out;        // device copies of a, b,
c
                                    int size = (N + 2*RADIUS) * sizeof(int);

                                    // Alloc space for host copies and setup
values
                                    in  = (int *)malloc(size); fill_ints(in,  N +
2*RADIUS);
                                    out = (int *)malloc(size); fill_ints(out, N +
2*RADIUS);

                                    // Alloc space for device copies
                                    cudaMalloc((void **)&d_in,  size);
                                    cudaMalloc((void **)&d_out, size);

                                    // Copy to device
                                    cudaMemcpy(d_in,  in,  size,
cudaMemcpyHostToDevice);
                                    cudaMemcpy(d_out, out, size,
cudaMemcpyHostToDevice);

                                    // Launch stencil_1d() kernel on GPU
                                    stencil_1d<<<N/
BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

                                    // Copy result back to host
                                    cudaMemcpy(out, d_out, size,
cudaMemcpyDeviceToHost);

                                    // Cleanup
                                    free(in); free(out);
                                    cudaFree(d_in); cudaFree(d_out);
                                    return 0;
}
```

parallel fn

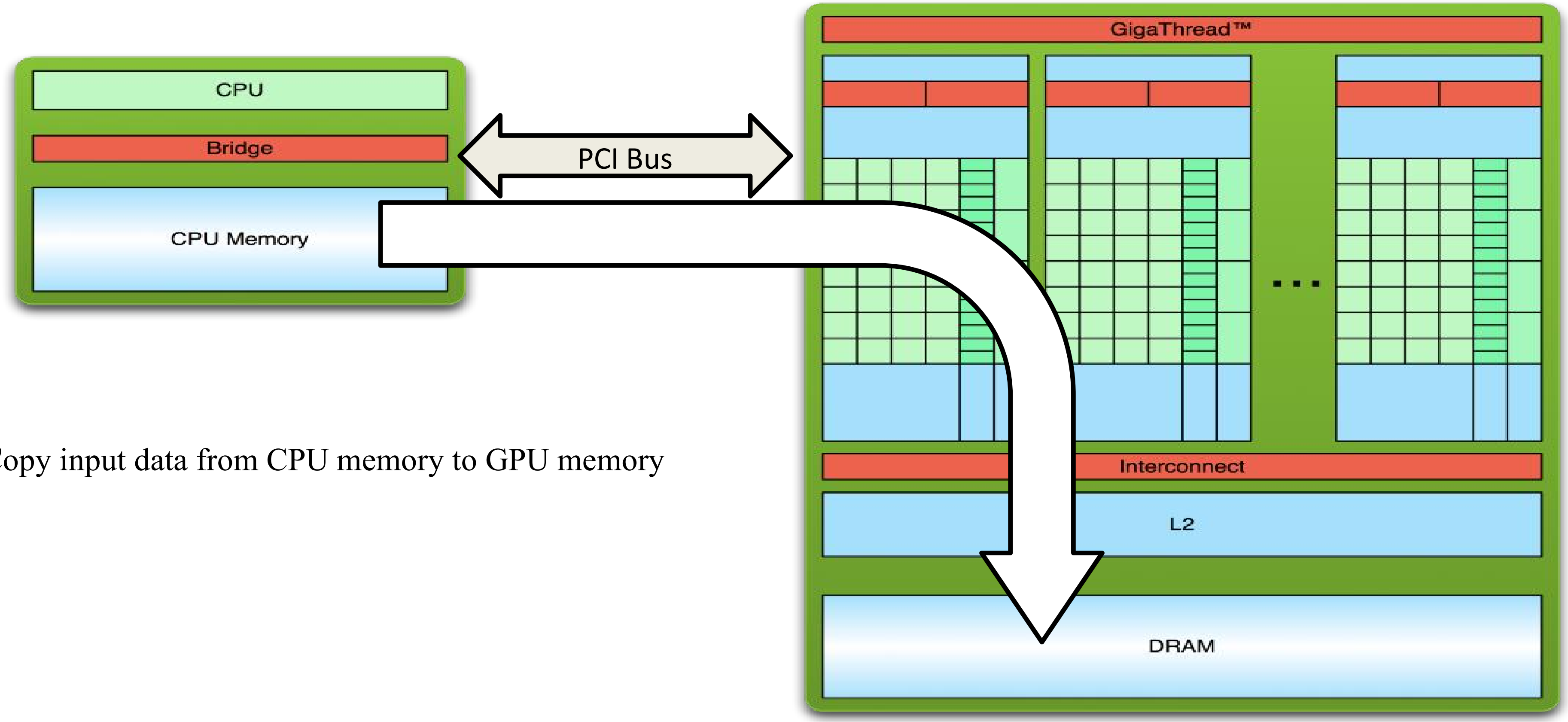
serial code

parallel code

serial code

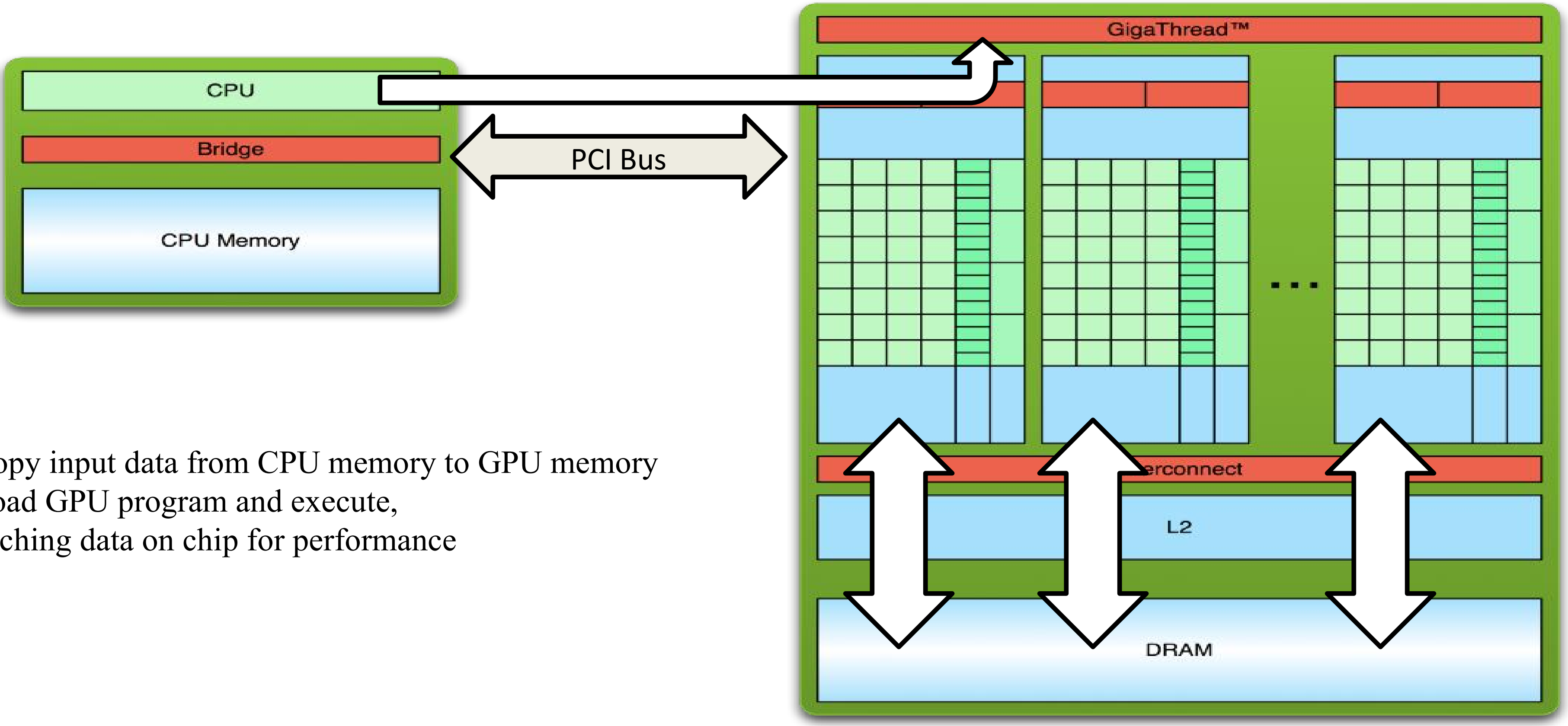


CUDA: Simple Processing Flow



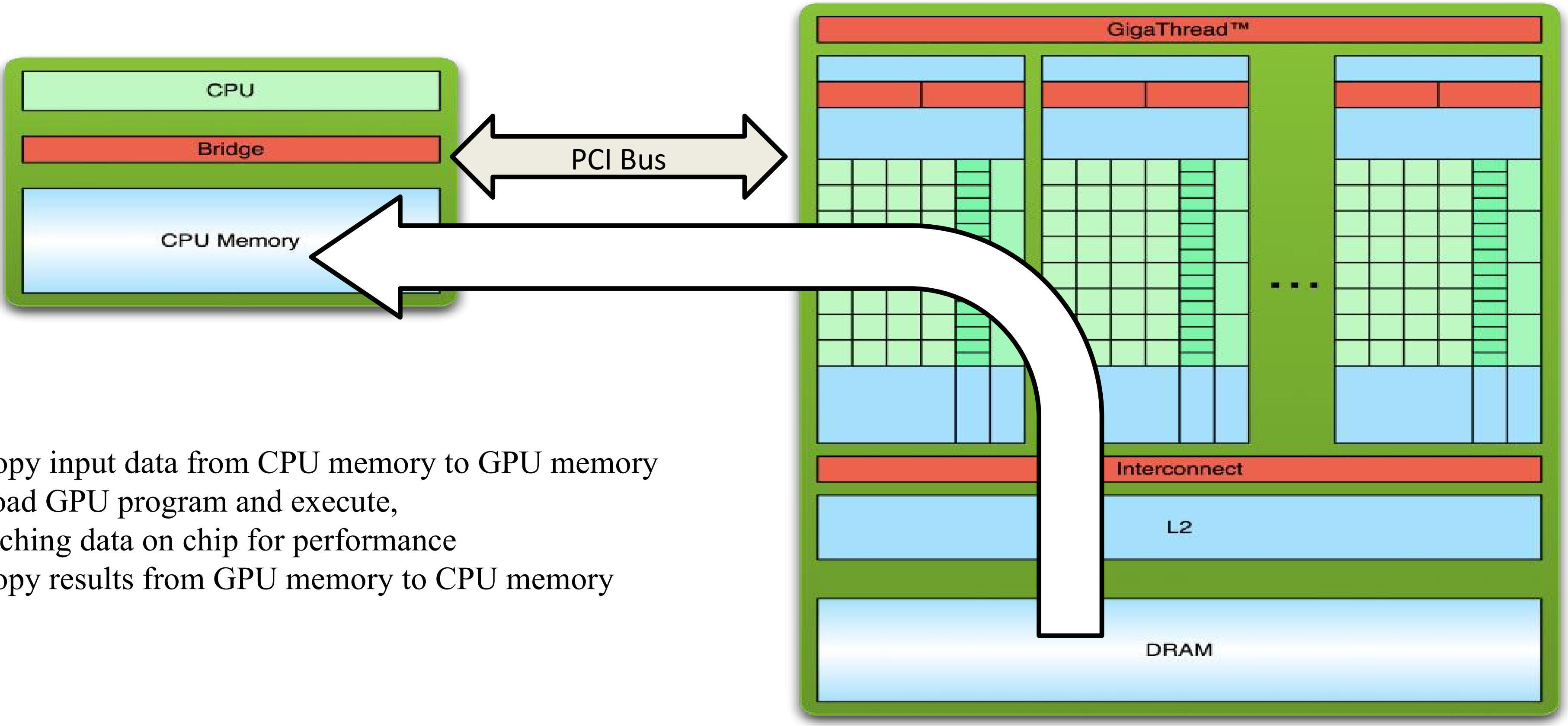
1. Copy input data from CPU memory to GPU memory

CUDA: Simple Processing Flow



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

CUDA: Simple Processing 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: Program

```
__global__ void mykernel(void) {}
```

- CUDA C/C++ keyword __global__ indicates a function that:
  - Runs on the device
  - Is called from host code

- nvcc separates source code into host and device components
  - Device functions (e.g. **mykernel()**) processed by NVIDIA compiler
  - Host functions (e.g. **main()**) processed by standard host compiler
    - **gcc, cl.exe**

With device COde

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a “kernel launch”
  - We’ll return to the parameters (1,1) in a moment

- That’s all that is required to execute a function on the GPU!

CUDA: Program

```
__global__ void mykernel(void){}

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

Output:

```
$ nvcc hello.cu
$ a.out
Hello World!
$
```

CUDA: Memory Management

- Host and device memory are separate entities
  - *Device* pointers point to GPU memory
    - May be passed to/from host code
    - May *not* be dereferenced in host code
  - *Host* pointers point to CPU memory
    - May be passed to/from device code
    - May *not* be dereferenced in device code

- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

CUDA: Addition on the Device

```
__global__ void add(int *a, int *b, int *c)
{
            *c = *a + *b;
}
int main(void) {
int a, b, c;                    // host copies of a, b, c
int *d_a, *d_b, *d_c;           // device copies of a, b, c
int size = sizeof(int);

// Allocate space for device copies of a, b, c
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);

// Setup input values
A = 2; b = 7;

// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size,
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;
}
```

RUNNING IN PARALLEL

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

CUDA: Moving to Parallel

GPU computing is about massive parallelism

So how do we run code in parallel on the device?

add<<< 1, 1 >>>();

add<<< N, 1 >>>();

Instead of executing add() once, execute N times in parallel

CUDA: Vector Addition on the Device

With **add()** running in parallel we can do vector addition

Terminology: each parallel invocation of **add()** is referred to as a block
The set of blocks is referred to as a grid
Each invocation can refer to its block index using **blockIdx.x**

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

By using **blockIdx.x** to index into the array, each block handles a different index

On the device, each block can execute in parallel:

Block 0	Block 1	Block 2	Block 3
c[0] = a[0] + b[0];	c[1] = a[1] + b[1];	c[2] = a[2] + b[2];	c[3] = a[3] + b[3];

CUDA: Vector Addition on the Device

```
#define N 512
int main(void) {
        int *a  *b  *c              // host copies of a, b, c
        int *d_a, *d_b, *d_c;      // device copies of a, b, c
        int size = N * sizeof(int);

        // Alloc space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Alloc space for host copies of a, b, c
        // and setup input values
        a = (int *)malloc(size); random_ints(a, N);
        b = (int *)malloc(size); random_ints(b, N);
        c = (int *)malloc(size);

        // Copy inputs to device
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

        // Launch add() kernel on GPU with N blocks
        add<<<N,1>>>(d_a, d_b, d_c);

        // Copy result back to host
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

CUDA: Review

- Difference between *host* and *device*
  - *Host* CPU
  - *Device* GPU
- Using **__global__** to declare a function as device code
  - Executes on the device
  - Called from the host
- Passing parameters from host code to a device function

- Basic device memory management
  - **cudaMalloc()**
  - **cudaMemcpy()**
  - **cudaFree()**
- Launching parallel kernels
  - Launch **N** copies of **add()** with **add<<<N,1>>>(…)**;
  - Use **blockIdx.x** to access block index

INTRODUCING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

CUDA: Threads

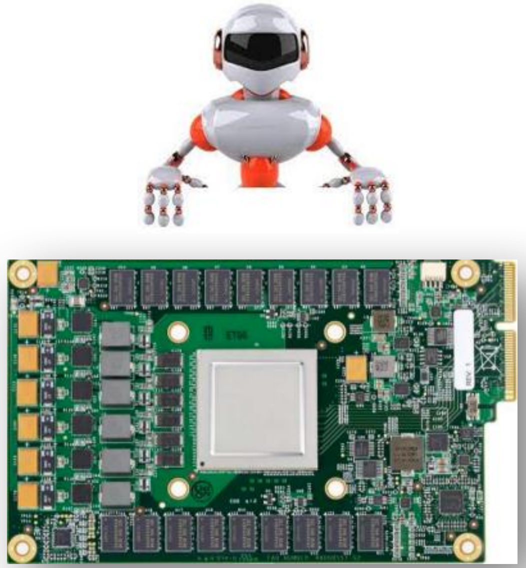
- Terminology: a block can be split into parallel threads
- Let's change add() to use parallel *threads* instead of parallel *blocks*

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

- We use **threadIdx.x** instead of **blockIdx.x**
- Need to make one change in **main()**…

CUDA: Vector Addition Using Thread

```
#define N 512
  int main(void) {
    int *a, *b, *c;              // host copies of a, b, c
    int *d_a, *d_b, *d_c;        // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);

    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
  }
```



COMBINING THREADS AND BLOCKS

CONCEPTS

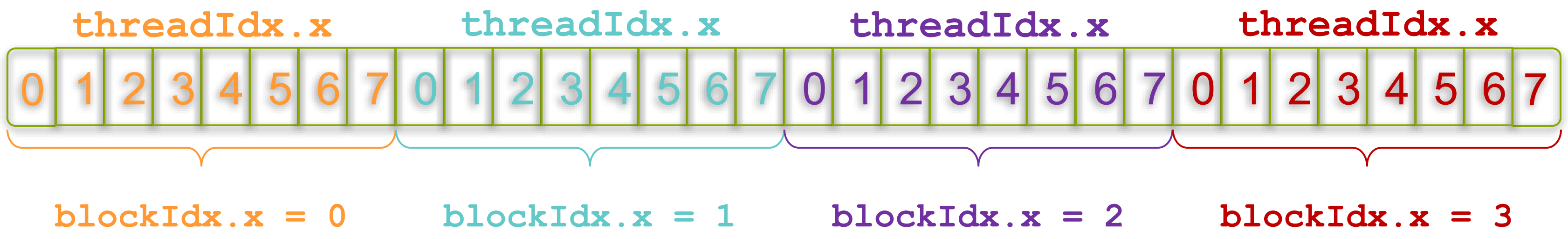
- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

CUDA: 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
- Why? We’ll come to that…
- First let’s discuss data indexing…

CUDA: 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)

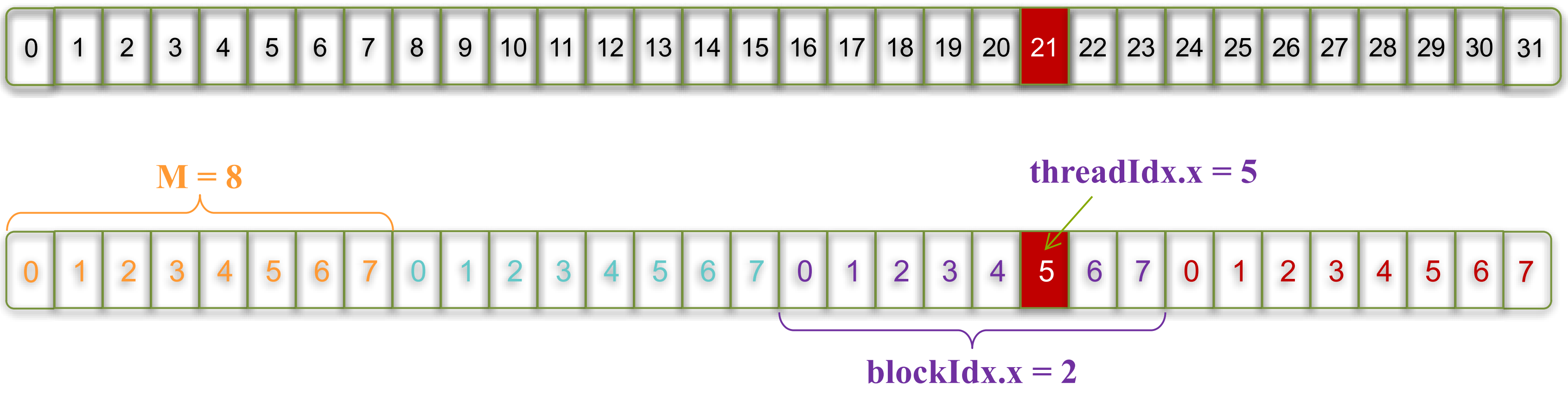


- With M threads/block a unique index for each thread is given by:

  int index = threadIdx.x + blockIdx.x * M;

CUDA: Indexing Arrays example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;
          =     5      +     2      * 8;
          = 21;
```

CUDA: Vector Addition with blocks and Threads

- Use the built-in variable **blockDim.x** for threads per block

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

- Combined version of add() to use parallel threads *and* parallel blocks

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

- What changes need to be made in **main()**?

CUDA: Addition with Blocks and Threads

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;          // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
    // Launch add() kernel on GPU

add<<<N/THREADS_PER_BLOCK THREADS_PER_BLOCK>>>
(d_a, d_b, d_c);

    / Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
  }
```

CUDA: Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of **blockDim.x**

- Avoid accessing beyond the end of the arrays:

```
__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];
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

CUDA

- Launching parallel kernels
  - Launch **N** copies of **add()** with **add<<<N/M,M>>>(…);**
  - Use **blockIdx.x** to access block index
  - Use **threadIdx.x** to access thread index within block

- Allocate elements to threads:

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

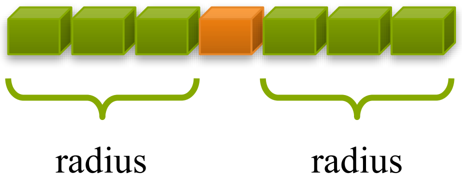
COOPERATING THREADS

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

CUDA: 1D Stencil

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius



- If radius is 3, then each output element is the sum of 7 input elements:

Implementing Within a Block

- Each thread processes one output element
  - blockDim.x elements per block

- Input elements are read several times
  - With radius 3, each input element is read seven times

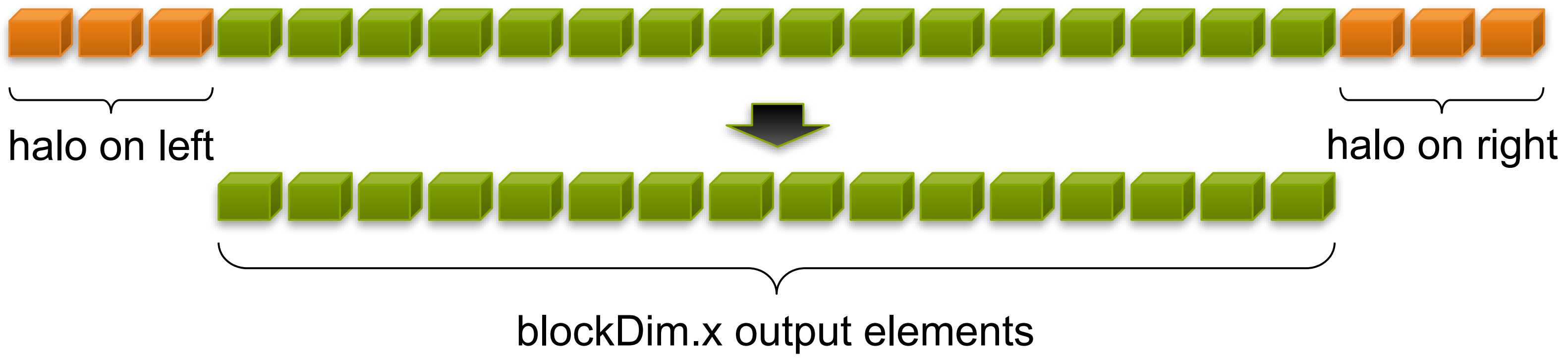
CUDA: Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using __shared__, allocated per block
- Data is not visible to threads in other blocks

CUDA: Implementing with Shared Memory

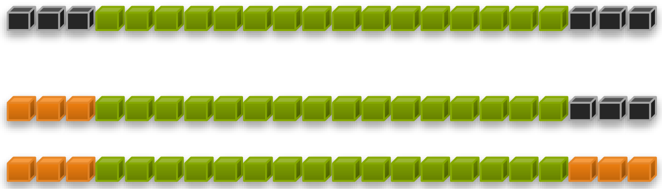
- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory

  - Each block needs a halo of radius elements at each boundary



CUDA: Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
      temp[lindex - RADIUS] = in[gindex - RADIUS];
      temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
```



CUDA: Stencil Kernel

```
  // Apply the stencil
  int result = 0;
  for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
    result += temp[lindex + offset];

  // Store the result
  out[gindex] = result;
}
```

CUDA: Data Race

- The stencil example will not work…
- Suppose thread 15 reads the halo before thread 0 has fetched it…

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS = in[gindex - RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];
```

Store at temp[18]

Skipped, threadIdx > RADIUS

Load from temp[19]

CUDA: __syncthreads()

- void __syncthreads();

- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards

- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

CUDA: Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex – RADIUS] = in[gindex – RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

CUDA: Review

- Launching parallel threads
  - Launch N blocks with M threads per block with **kernel<<<N,M>>>(…);**
  - Use **blockIdx.x** to access block index within grid
  - Use **threadIdx.x** to access thread index within block

- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x
```

- Use **__shared__** to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks

- Use **__syncthreads()** as a barrier
  - Use to prevent data hazards

MANAGING THE DEVICE

CONCEPTS

- Heterogeneous Computing
- Blocks
- Threads
- Indexing
- Shared memory
- __syncthreads()
- Asynchronous operation
- Handling errors
- Managing devices

CUDA: Coordination Host and Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately

- CPU needs to synchronize before consuming the results

cudaMemcpy()	Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
cudaMemcpyAsync()	Asynchronous, does not block the CPU
cudaDeviceSynchronize()	Blocks the CPU until all preceding CUDA calls have completed

CUDA: Reporting Errors

- All CUDA API 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)

  printf("%s\n", cudaGetErrorString(cudaGetLastError()));

CUDA: Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp *prop, int device)
```

- Multiple threads can share a device

- A single thread can manage multiple devices

  `cudaSetDevice(i)` to select current device

  `cudaMemcpy(…)` for peer-to-peer copies †

CUDA

- What have we learned?
  - Write and launch CUDA C/C++ kernels
    - **__global__, blockIdx.x, threadIdx.x, <<<>>>**
  - Manage GPU memory
    - **cudaMalloc(), cudaMemcpy(), cudaFree()**
  - Manage communication and synchronization
    - **__shared__, __syncthreads()**
    - **cudaMemcpy()** vs **cudaMemcpyAsync(), cudaDeviceSynchronize()**

CUDA: Capability

- The **compute capability** of a device describes its architecture, e.g.
  - Number of registers
  - Sizes of memories
  - Features & capabilities

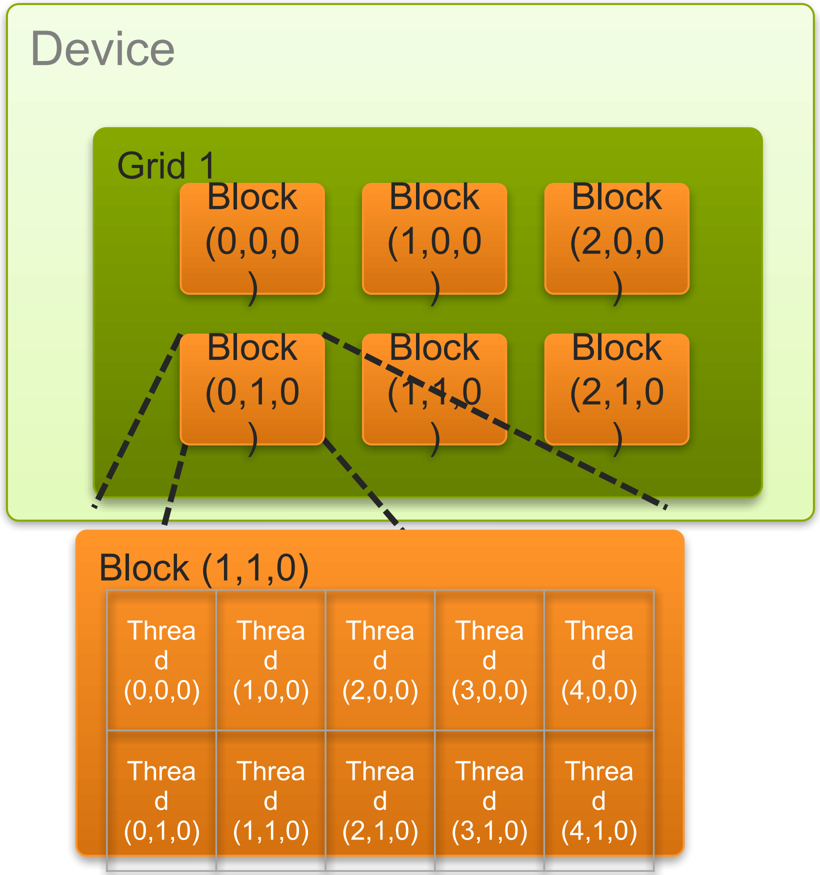
Compute Capability	Selected Features (see CUDA C Programming Guide for complete list)	Tesla models
1.0	Fundamental CUDA support	870
1.3	Double precision, improved memory accesses, atomics	10-series
2.0	Caches, fused multiply-add, 3D grids, surfaces, ECC, P2P, concurrent kernels/copies, function pointers, recursion	20-series

- The following presentations concentrate on Fermi devices
  - Compute Capability >= 2.0

CUDA: IDs and Dimension

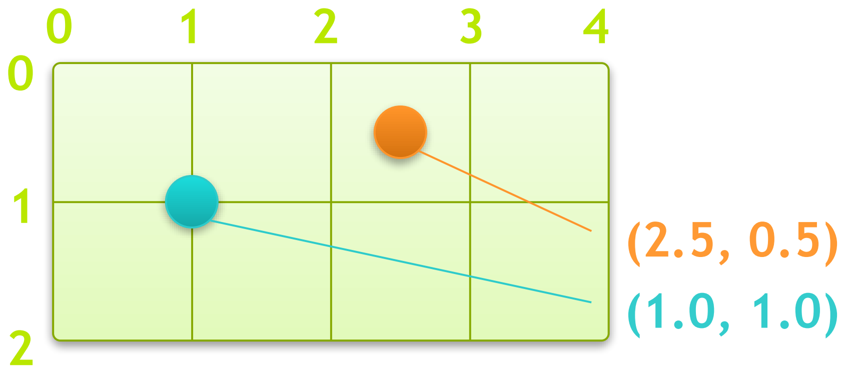
- A kernel is launched as a grid of blocks of threads
  - blockIdx and threadIdx are 3D
  - We showed only one dimension (x)

- Built-in variables:
  - threadIdx
  - blockIdx
  - blockDim
  - gridDim



CUDA: Textures

- Read-only object
  - Dedicated cache
- Dedicated filtering hardware
  (Linear, bilinear, trilinear)
- Addressable as 1D, 2D or 3D
- Out-of-bounds address handling
  (Wrap, clamp)



CUDA API Examples ?

Which GPU do I have ?

```
#include <stdio.h>
int main()
{
     int noOfDevices;
     /* get no. of device */
     cudaGetDeviceCount (&noOfDevices);

     cudaDeviceProp  prop;
     for (int i = 0; i < noOfDevices; i++)
     {
          /*get device properties */
          cudaGetDeviceProperties (&prop, i );

          printf ("Device Name:\t %s\n",  prop.name);
          printf ("Total global memory:\t %ld\n",
                    prop.totalGlobalMem);
          printf (”No. of SMs:\t %d\n",
                    prop.multiProcessorCount);
          printf ("Shared memory / SM:\t %ld\n",
                    prop.sharedMemPerBlock);
          printf("Registers / SM:\t %d\n",
                    prop.regsPerBlock);

     }
     return 1;
}
```

Use
`cudaGetDeviceCount`
`cudaGetDeviceProperties`

Compilation

```
> nvcc whatDevice.cu –o whatDevice
```

Output

```
Device Name:           Tesla C2050
Total global memory:   2817720320
No. of SMs:            14
Shared memory / SM:    49152
Registers / SM:        32768
```

For more properties see
`struct cudaDeviceProp`

For details see CUDA Reference Manual

Timing with CUDA Event API

```
int main ()
{
        cudaEvent_t start, stop;
        float time;

        cudaEventCreate (&start);
        cudaEventCreate (&stop);

        cudaEventRecord (start, 0);

        someKernel <<<grids, blocks, 0, 0>>> (...);

        cudaEventRecord (stop, 0);
        cudaEventSynchronize (stop);

        cudaEventElapsedTime (&time, start, stop);

        cudaEventDestroy (start);
        cudaEventDestroy (stop);

        printf ("Elapsed time %f sec\n", time*.001);

        return 1;
}
```

Ensures kernel execution has completed

CUDA Event API Timer are,

- OS independent
- High resolution
- Useful for timing asynchronous calls

Standard CPU timers will not measure the timing information of the device.

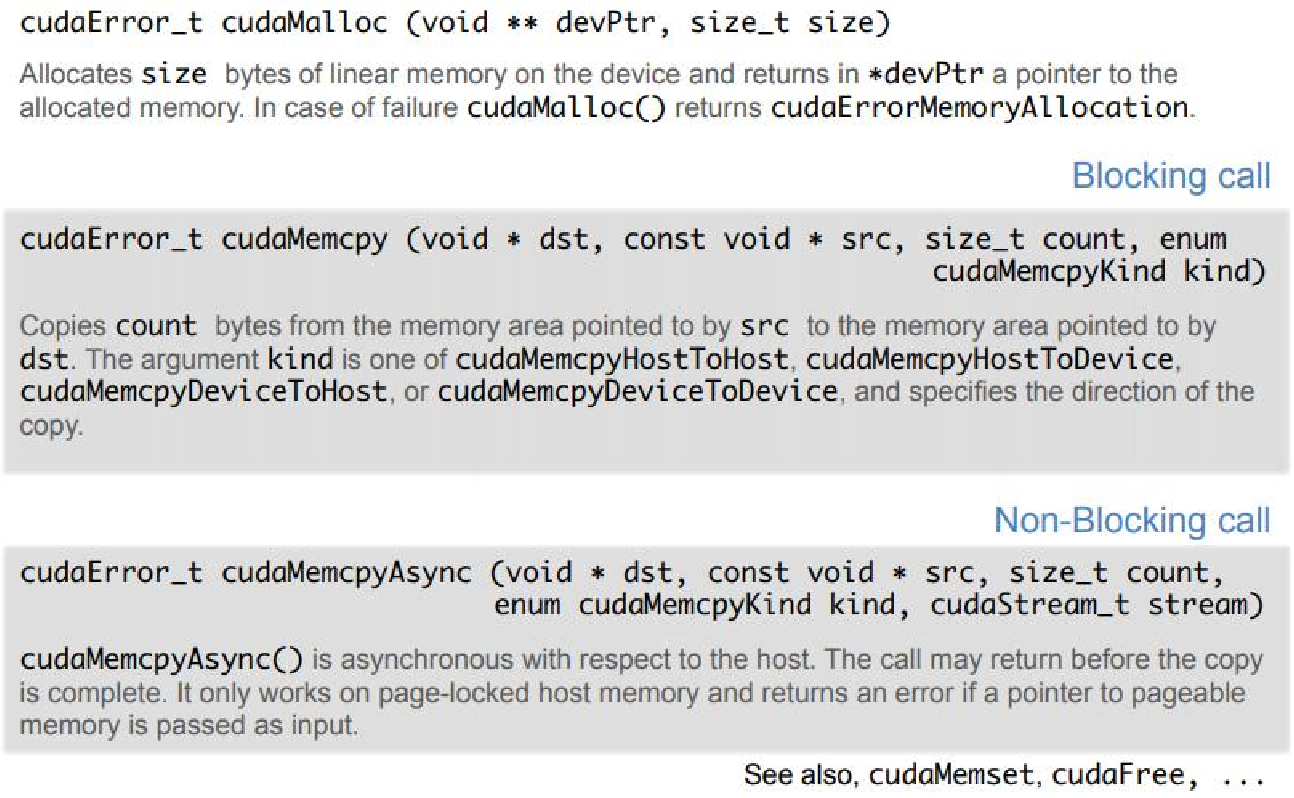
Memory Allocations / Copies

```
int main ()
{
  ...

  float host_signal[N]; host_result[N];
  float *device_signal, *device_result;

  //allocate memory on the device (GPU)
  cudaMalloc ((void**) &device_signal, N * sizeof(float));
  cudaMalloc ((void**) &device_result, N * sizeof(float));

  ... Get data for the host_signal array

  // copy host_signal array to the device
  cudaMemcpy (device_signal, host_signal , N * sizeof(float),
                 cudaMemcpyHostToDevice);

  someKernel <<<< >>> (...);

  //copy the result back from device to the host
  cudaMemcpy (host_result, device_result, N * sizeof(float),
                 cudaMemcpyDeviceToHost);

  //display the results
  ...
  cudaFree (device_signal); cudaFree (device_result) ;
}
```

Host and device have separate physical memory

Cannot dereference host pointers on device and vice versa

Basic Memory Methods



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

Allocates `size` bytes of linear memory on the device and returns in `*devPtr` a pointer to the allocated memory. In case of failure `cudaMalloc()` returns `cudaErrorMemoryAllocation`.

Blocking call

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

Copies `count` bytes from the memory area pointed to by `src` to the memory area pointed to by `dst`. The argument `kind` is one of `cudaMemcpyHostToHost`, `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`, or `cudaMemcpyDeviceToDevice`, and specifies the direction of the copy.

Non-Blocking call

```
cudaError_t cudaMemcpyAsync (void * dst, const void * src, size_t count,
                             enum cudaMemcpyKind kind, cudaStream_t stream)
```

`cudaMemcpyAsync()` is asynchronous with respect to the host. The call may return before the copy is complete. It only works on page-locked host memory and returns an error if a pointer to pageable memory is passed as input.

See also, `cudaMemset`, `cudaFree, ...`

Kernel

The CUDA kernel is,

Run on device

Defined by `__global__` qualifier and does not return anything

```
__global__ void someKernel ();
```

Executed asynchronously by the host with <<< >>> qualifier, for example,

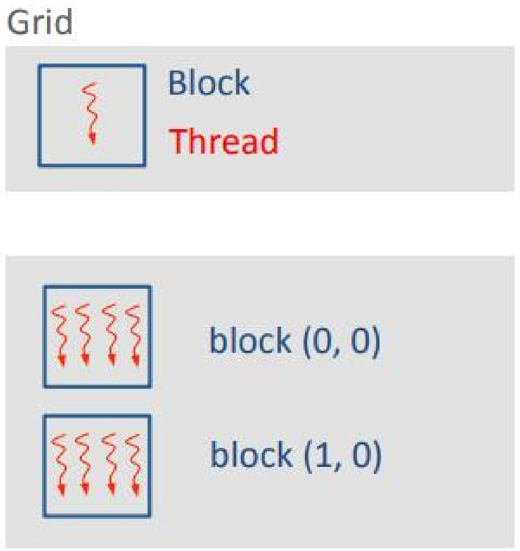
```
someKernel <<<nGrid, nBlocks, sharedMemory, streams>>> (...)
someKernel <<<nGrid, nBlocks>>> (...)
```

The kernel launches a 1- or 2-D **grid** of 1-, 2- or 3-D **blocks** of **threads**
Each thread executes the same kernel in parallel (SIMT)
Threads within blocks can communicate via shared memory
Threads within blocks can be synchronized

Grids and blocks are of type `struct dim3`

Built-in variables `gridDim`, `blockDim`, `threadIdx`, `blockIdx` are used to traverse across the device memory space with multi-dimensional indexing

Grids, Blocks and Threads



```
someKernel<<< 1, 1 >>> ();
gridDim.x    = 1
blockDim.x   = 1
blockIdx.x   = 0
threadIdx.x  = 0
```

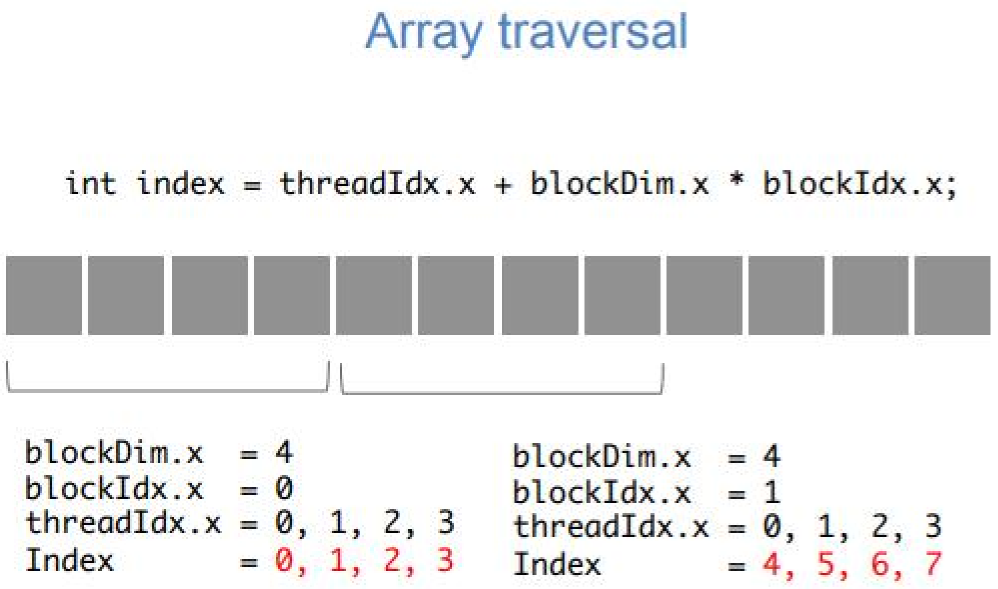
```
dim3 blocks (2,1,1);
someKernel<<< (blocks, 4) >>> ();
gridDim.x    = 2;
blockDim.x   = 4;
blockIdx.x   = 0,1;
threadIdx.x  = 0,1,2,3,0,1,2,3
```

<<< number of blocks in a grid, number of threads per block >>>

Useful for multidimensional indexing and creating unique thread IDs

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

Thread Indices



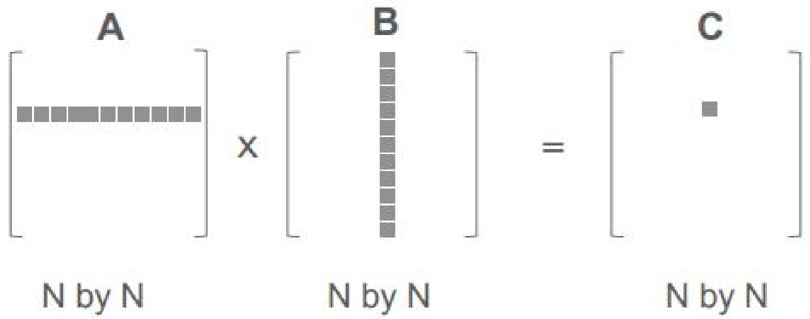
```
int index = threadIdx.x + blockDim.x * blockIdx.x;
```

```
blockDim.x  = 4              blockDim.x  = 4
blockIdx.x  = 0              blockIdx.x  = 1
threadIdx.x = 0, 1, 2, 3     threadIdx.x = 0, 1, 2, 3
Index       = 0, 1, 2, 3     Index       = 4, 5, 6, 7
```

Example

Matrix-multiplication

Each element of product matrix **C** is generated by row column multiplication and reduction of matrices **A** and **B**. This operation is similar to inner product of the vector multiplication kind also known as vector dot product.



For size N × N matrices the matrix-multiplication **C = A · B** will be equivalent to N^2 independent (hence parallel) inner products.

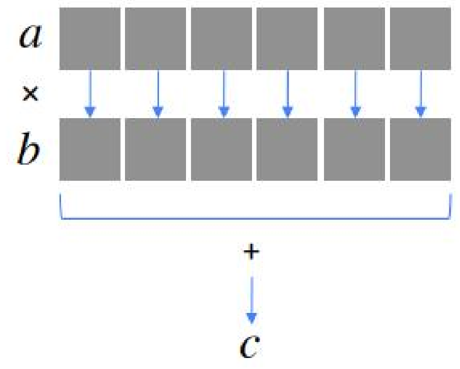
Example

$$c = \sum_i a_i b_i$$

Serial representation

```
double c = 0.0;

for (int i = 0; i < SIZE; i++)
    c += a[i] * b[i];
```

Simple parallelization strategy



Multiplications are done in parallel

Summation is sequential

Example

```
__global__ void innerProduct (int *a, int *b, int *c)
{
  int product[SIZE];

  int i = threadIdx.x;

  if (i < SIZE)
    product[i] = a[i] * b[i];

}
```

```
__global__ void innerProduct (...)
{
      ...
}

int main ()
{
    ...

    innerProduct<<<grid, block>>> (...);

    ...
}
```

Called in the host code

Example

```
__global__ void innerProduct (int *a, int *b, int *c)
{
  int product[SIZE];

  int i = threadIdx.x;

  if (i < SIZE)
    product[i] = a[i] * b[i];







}
```

Qualifier `__global__` encapsulates device specific code that runs on the device and is called by the host

Other qualifiers are, `__device__`, `__host__`, `host__and__device`

`threadIdx` is a built in iterator for threads. It has 3 dimensions x, y and z.

Each thread with a unique `threadIdx.x` runs the kernel code in parallel.

Example

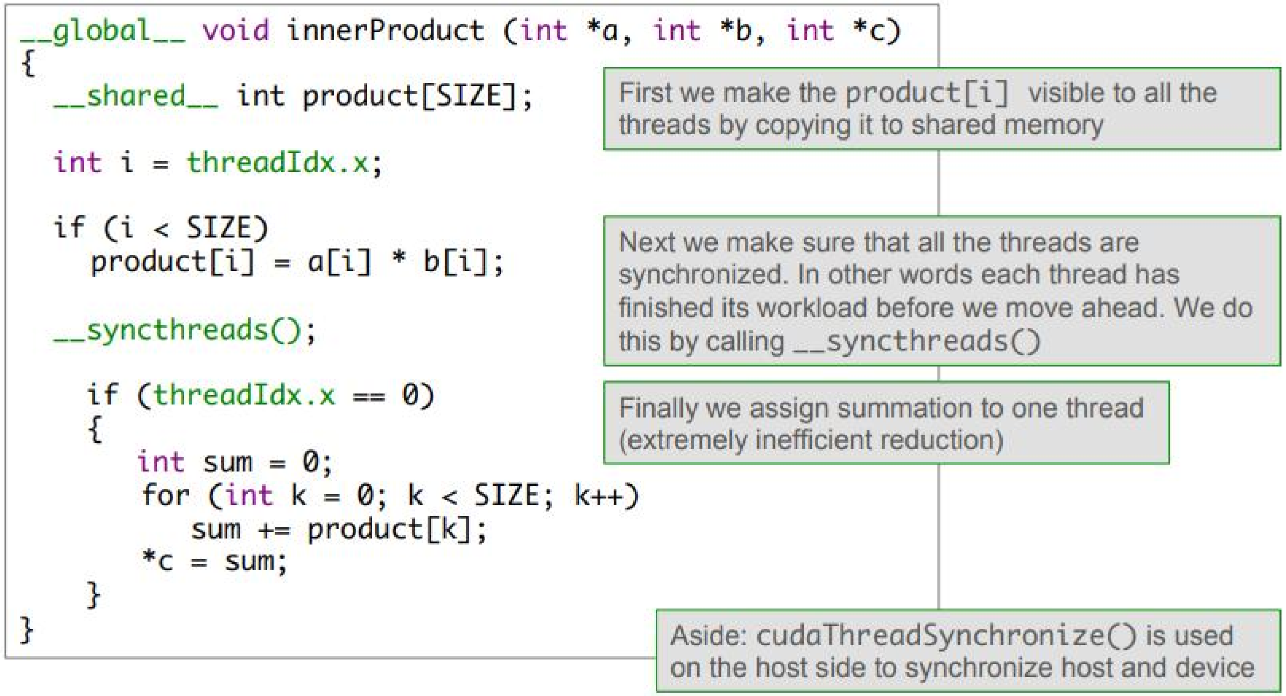
```
__global__ void innerProduct (int *a, int *b, int *c)
{
  int product[SIZE];

  int i = threadIdx.x;

  if (i < SIZE)
    product[i] = a[i] * b[i];




       int sum = 0;
       for (int k = 0; k < N; k++)
           sum += product[k];
       *c = sum;

}
```

Now we can sum the all the products to get the scalar c

Unfortunately this won't work for following reasons,

- `product[i]` is local to each thread
- Threads are not visible to each other

Example



```
__global__ void innerProduct (int *a, int *b, int *c)
{
  __shared__ int product[SIZE];

  int i = threadIdx.x;

  if (i < SIZE)
    product[i] = a[i] * b[i];

  __syncthreads();

    if (threadIdx.x == 0)
    {
       int sum = 0;
       for (int k = 0; k < SIZE; k++)
          sum += product[k];
       *c = sum;
    }
}
```

First we make the product[i] visible to all the threads by copying it to shared memory

Next we make sure that all the threads are synchronized. In other words each thread has finished its workload before we move ahead. We do this by calling __syncthreads()

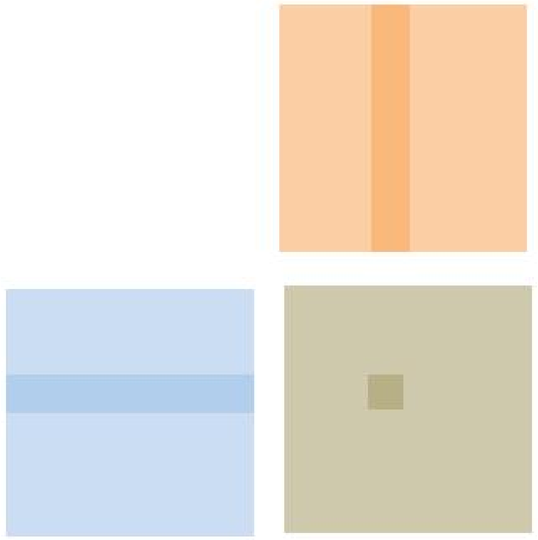
Finally we assign summation to one thread (extremely inefficient reduction)

Aside: cudaThreadSynchronize() is used on the host side to synchronize host and device

Example: Matrix Multiplication

CPU Version

```
void matrixMultiplication ( float* A, float* B, float* C, int WIDTH)
{

    for (i → 0 : WIDHT)

        for (j → 0 : WIDTH)

            for (k → 0 : WIDTH)

                a = $A_i$;
                b = $B_j$;
                sum += a * b;

        $C_{ij}$ = sum;
}
```



Example: Matrix Multiplication

GPU Version (Memory locations)

Constant memory

```
__global__ void matrixMultiplication (float* A, float* B, float* C, int WIDTH)
{
```

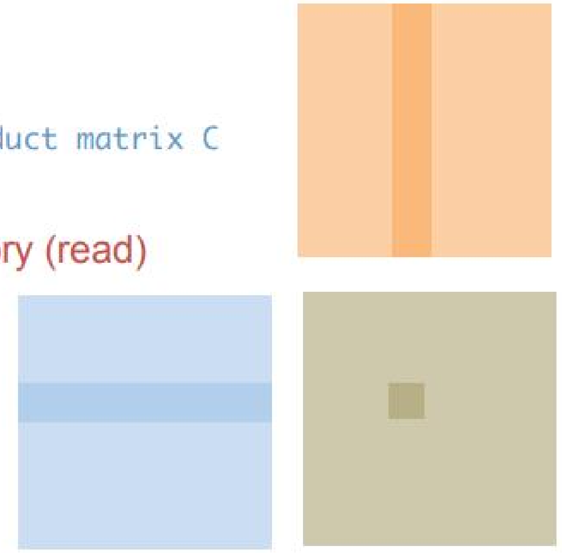
Shared memory

```
    int i = blockIdx.y * WIDTH + threadIdx.y;
    int j = blockIdx.x * WIDTH + threadIdx.x;

    // each thread computes one element of product matrix C
    for (k → 0 : k)

        sum += A[i][k] * B[k][j];
```

Global memory (read)

```
    C[i][j] = sum;
}
```

Global memory (write)



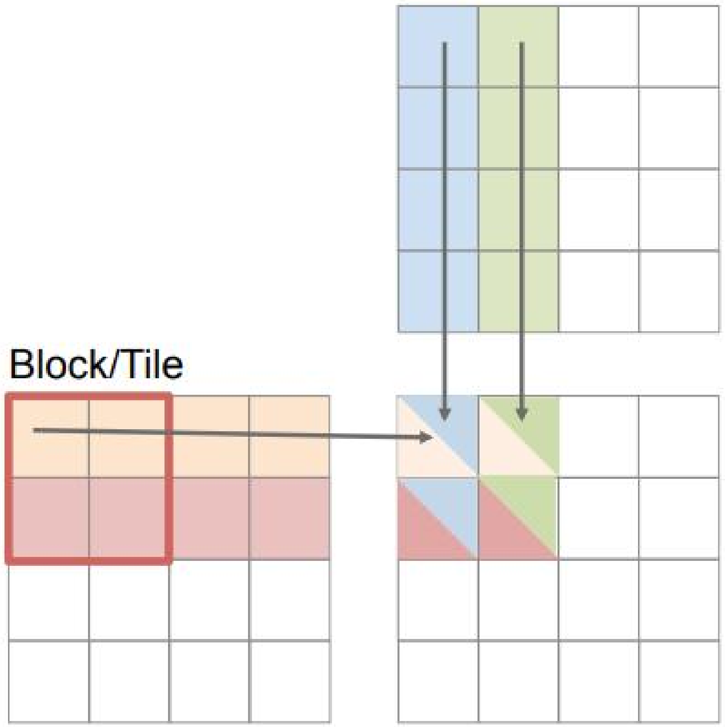
Example: Matrix Multiplication

Partial rows and columns are loaded in shared memory

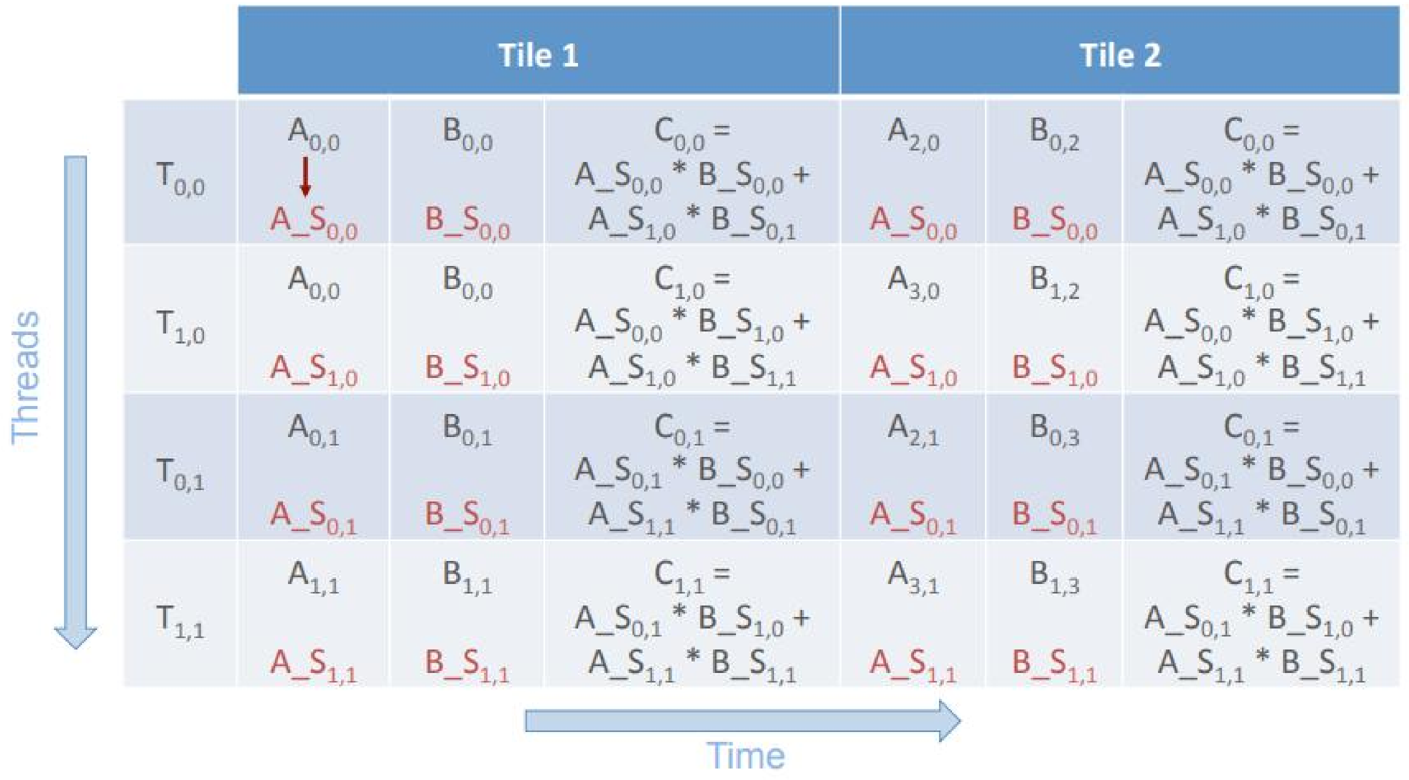
One row is reused to calculate two elements.

Multiple blocks are executed in parallel.

For a 16 x 16 tile width the global memory loads are reduced by 16.



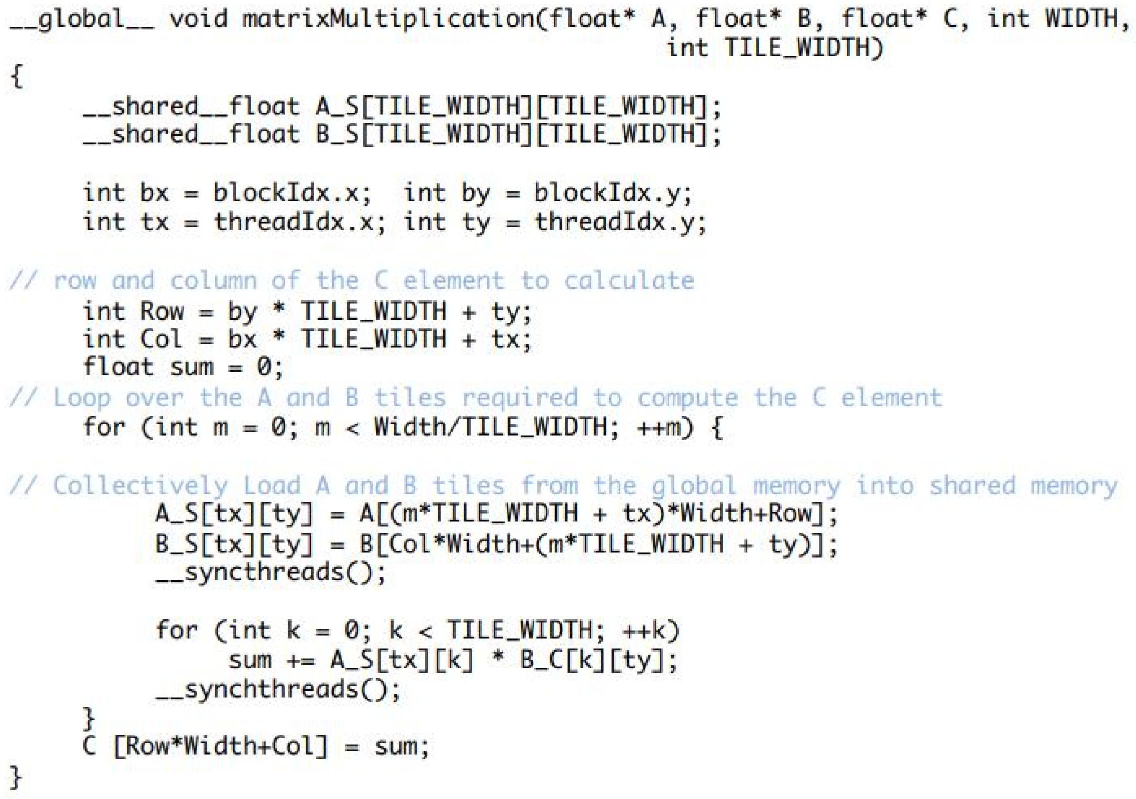
Example: Matrix Multiplication



Threads
Time

	Tile 1			Tile 2		
$T_{0,0}$	$A_{0,0}$ $\downarrow$ $A\_S_{0,0}$	$B_{0,0}$ $B\_S_{0,0}$	$C_{0,0} = A\_S_{0,0} * B\_S_{0,0} + A\_S_{1,0} * B\_S_{0,1}$	$A_{2,0}$ $A\_S_{0,0}$	$B_{0,2}$ $B\_S_{0,0}$	$C_{0,0} = A\_S_{0,0} * B\_S_{0,0} + A\_S_{1,0} * B\_S_{0,1}$
$T_{1,0}$	$A_{0,0}$ $A\_S_{1,0}$	$B_{0,0}$ $B\_S_{1,0}$	$C_{1,0} = A\_S_{0,0} * B\_S_{1,0} + A\_S_{1,0} * B\_S_{1,1}$	$A_{3,0}$ $A\_S_{1,0}$	$B_{1,2}$ $B\_S_{1,0}$	$C_{1,0} = A\_S_{0,0} * B\_S_{1,0} + A\_S_{1,0} * B\_S_{1,1}$
$T_{0,1}$	$A_{0,1}$ $A\_S_{0,1}$	$B_{0,1}$ $B\_S_{0,1}$	$C_{0,1} = A\_S_{0,1} * B\_S_{0,0} + A\_S_{1,1} * B\_S_{0,1}$	$A_{2,1}$ $A\_S_{0,1}$	$B_{0,3}$ $B\_S_{0,1}$	$C_{0,1} = A\_S_{0,1} * B\_S_{0,0} + A\_S_{1,1} * B\_S_{0,1}$
$T_{1,1}$	$A_{1,1}$ $A\_S_{1,1}$	$B_{1,1}$ $B\_S_{1,1}$	$C_{1,1} = A\_S_{0,1} * B\_S_{1,0} + A\_S_{1,1} * B\_S_{1,1}$	$A_{3,1}$ $A\_S_{1,1}$	$B_{1,3}$ $B\_S_{1,1}$	$C_{1,1} = A\_S_{0,1} * B\_S_{1,0} + A\_S_{1,1} * B\_S_{1,1}$

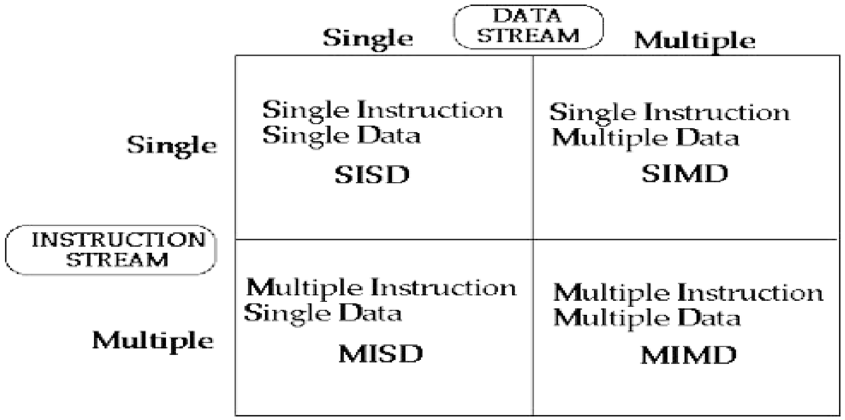
Example: Matrix Multiplication



```
__global__ void matrixMultiplication(float* A, float* B, float* C, int WIDTH,
                                                        int TILE_WIDTH)
{
     __shared__float A_S[TILE_WIDTH][TILE_WIDTH];
     __shared__float B_S[TILE_WIDTH][TILE_WIDTH];

     int bx = blockIdx.x;  int by = blockIdx.y;
     int tx = threadIdx.x; int ty = threadIdx.y;

// row and column of the C element to calculate
     int Row = by * TILE_WIDTH + ty;
     int Col = bx * TILE_WIDTH + tx;
     float sum = 0;
// Loop over the A and B tiles required to compute the C element
     for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Collectively Load A and B tiles from the global memory into shared memory
          A_S[tx][ty] = A[(m*TILE_WIDTH + tx)*Width+Row];
          B_S[tx][ty] = B[Col*Width+(m*TILE_WIDTH + ty)];
          __syncthreads();

          for (int k = 0; k < TILE_WIDTH; ++k)
               sum += A_S[tx][k] * B_C[k][ty];
          __synchthreads();
     }
     C [Row*Width+Col] = sum;
}
```

Thank you for your attention !

Flynn’s Taxonomy


DATA STREAM
Single
Multiple
INSTRUCTION STREAM
Single
Multiple
Single Instruction Single Data
SISD
Single Instruction Multiple Data
SIMD
Multiple Instruction Single Data
MISD
Multiple Instruction Multiple Data
MIMD

Classification of computer architectures, proposed by Michael J. Flynn

- SISD – traditional serial architecture in computers.
- SIMD – parallel computer. One instruction is executed many times with different data (think of a for loop indexing through an array)
- MISD - Each processing unit operates on the data independently via independent instruction streams. Not really used in parallel
- MIMD – Fully parallel and the most common form of parallel computing.

Mechnics of Using Shared Memory

- __shared__ type qualifier required
- Must be allocated from global/device function, or as “extern”
- Examples:

```
extern __shared__ float  d_s_array[];

/* a form of dynamic allocation */
/* MEMSIZE is size of per-block  */
/* shared memory*/
__host__ void outerCompute() {
   compute<<<gs,bs,MEMSIZE>>>();
}
__global__ void compute() {
     d_s_array[i] = …;
}
```

```
__global__ void compute2() {
   __shared__ float d_s_array[M];

   /* create or copy from global memory */
   d_s_array[j] = …;

  /* write result back to global memory */
   d_g_array[j] =  d_s_array[j];
}
```

Hardware Architecture