COMP3031: GPU Parallel Programming in CUDA

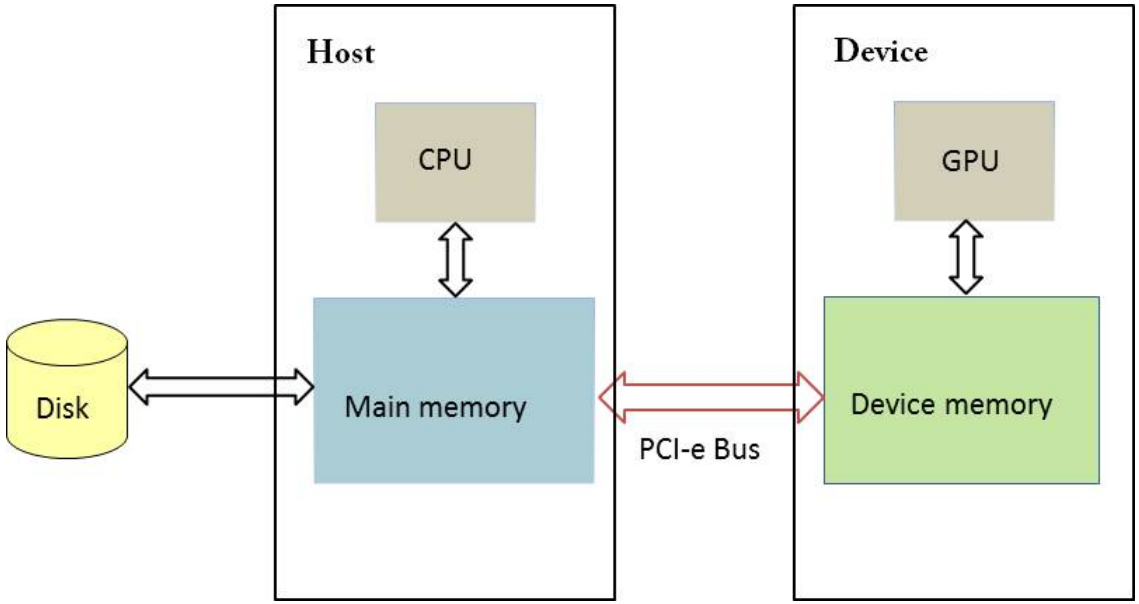
Qiong Luo

HKUST

Graphics Processing Unit(GPU)

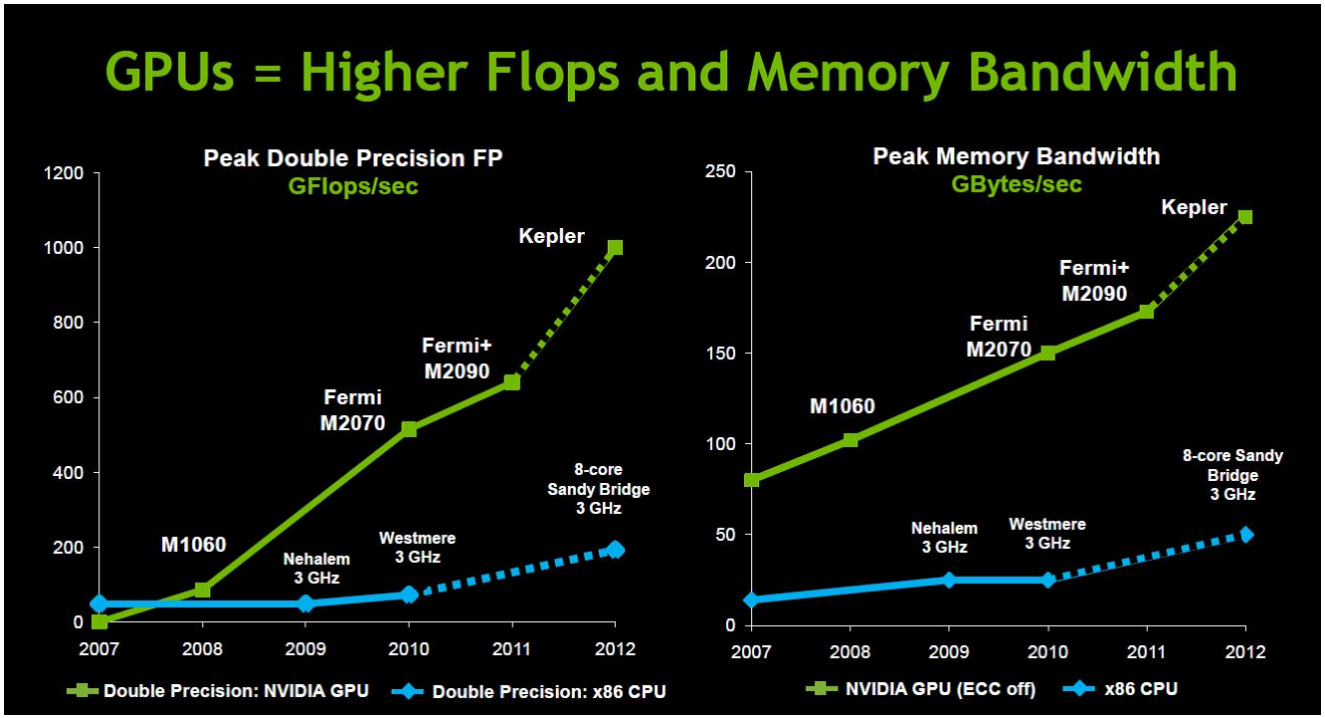
- Traditionally used for game applications
- Currently major accelerators for general-purpose computing applications that exhibit data parallelism
- Work as co-processors, i.e., rely on the CPU for task control, memory allocation, data transfer, etc.

GPU-CPU architecture

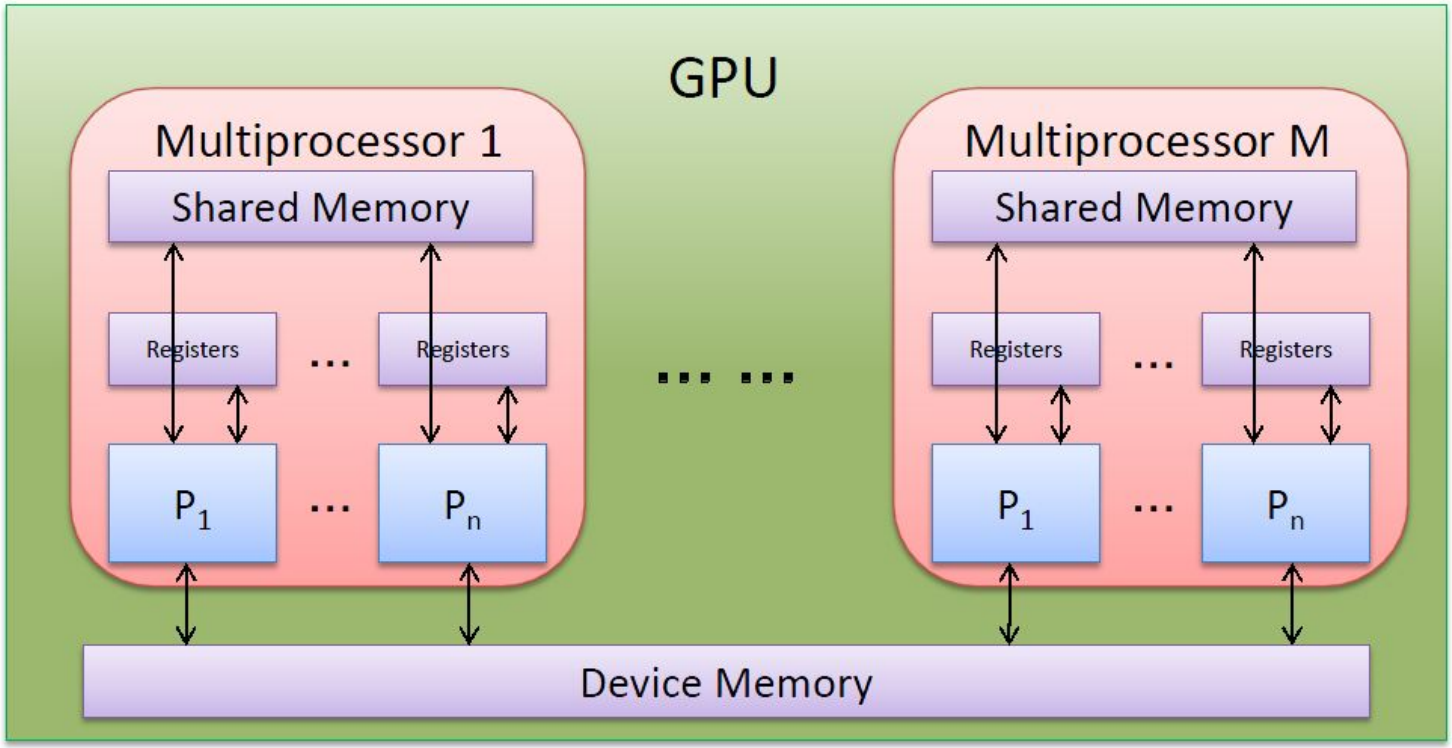


- PCI-e Bus serves as a bridge between the Host and Device
- General workflow of utilizing GPU resources
  - Transfer input data from Host to Device
  - Process data using GPU
  - Transfer output data back from Device To Host

GPU versus CPU

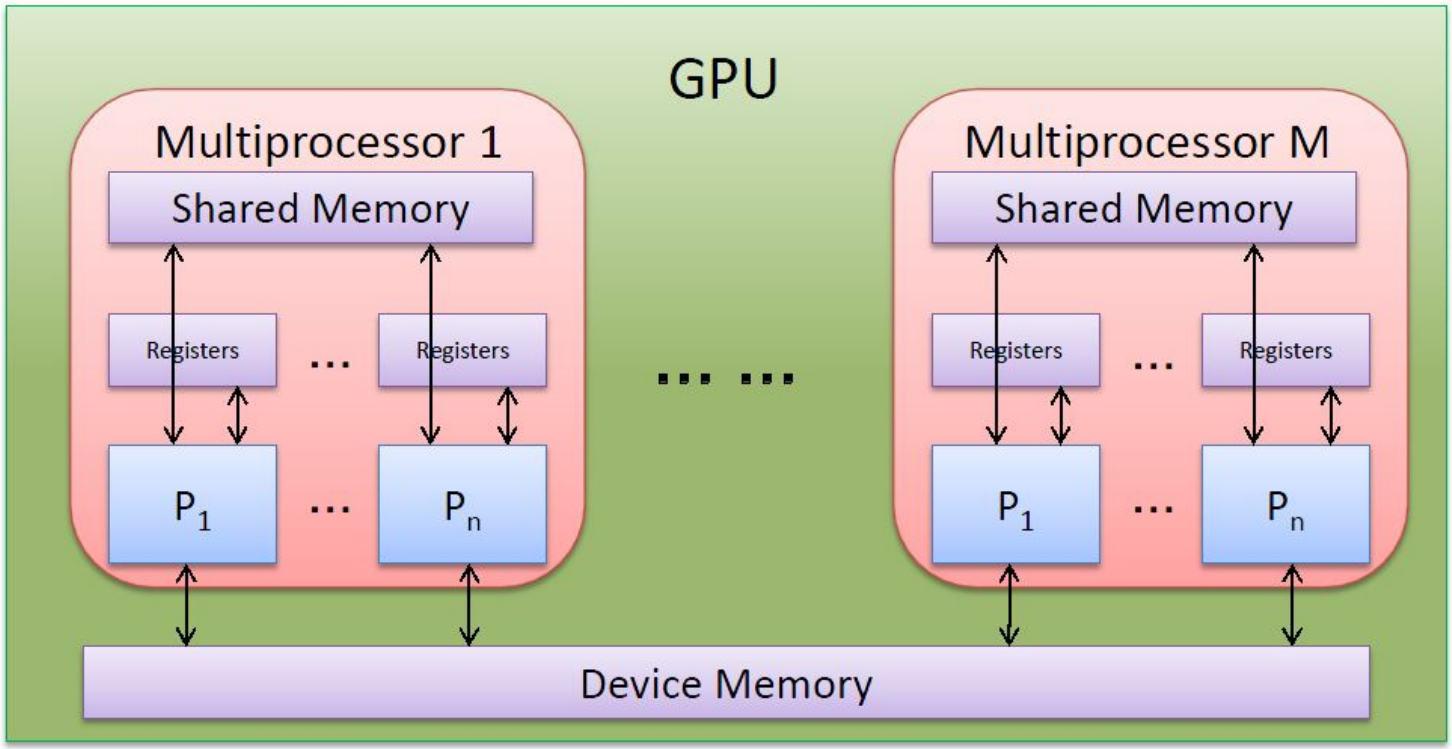


GPU Architecture - Multiprocessors



- Tens of identical Streaming Multiprocessors(SMs)
- Tens of identical uniprocessors in each SM
- Thousands of concurrent threads allowed in the GPU

GPU Architecture - Memory Hierarchy

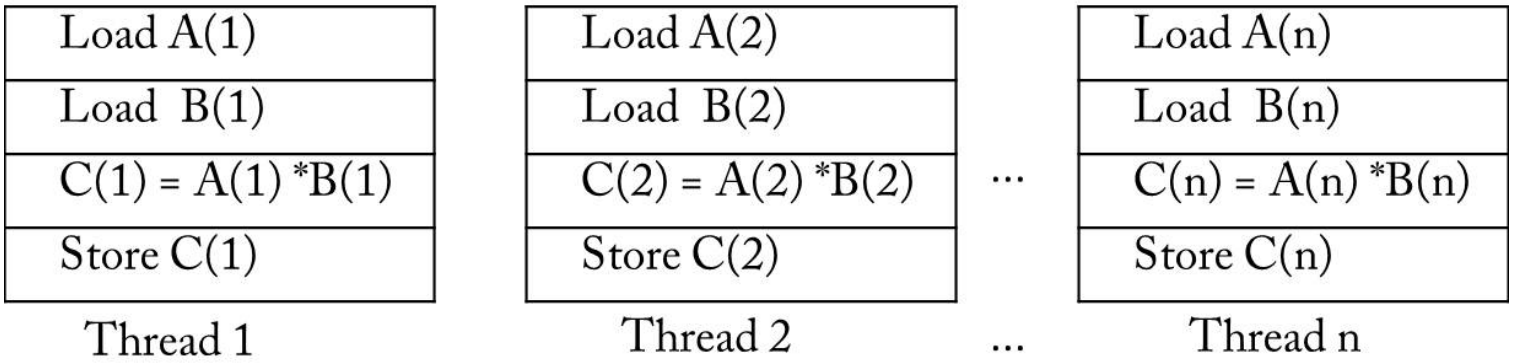


- Off-chip device memory: high-bandwidth, high-latency
- On-chip shared memory: small, fast, programmer use
- Registers: smallest, fastest on-chip memory

GPU Architecture - Single Instruction, Multiple Data

- GPU architecture belongs to the Single Instruction, Multiple Data(SIMD) type
- Each thread executes the same program
- Each thread processes different data element

Input array: A(1…n) and B(1…n)
Output array: C(1…n)



CUDA:Compute Unified Device Architecture

- A parallel computing platform developed by NVIDIA
- Extension to C programming language
- Add easy-to-use APIs to access to the GPU resources
- Requires no knowledge of graphics API like DirectX and OpenGL

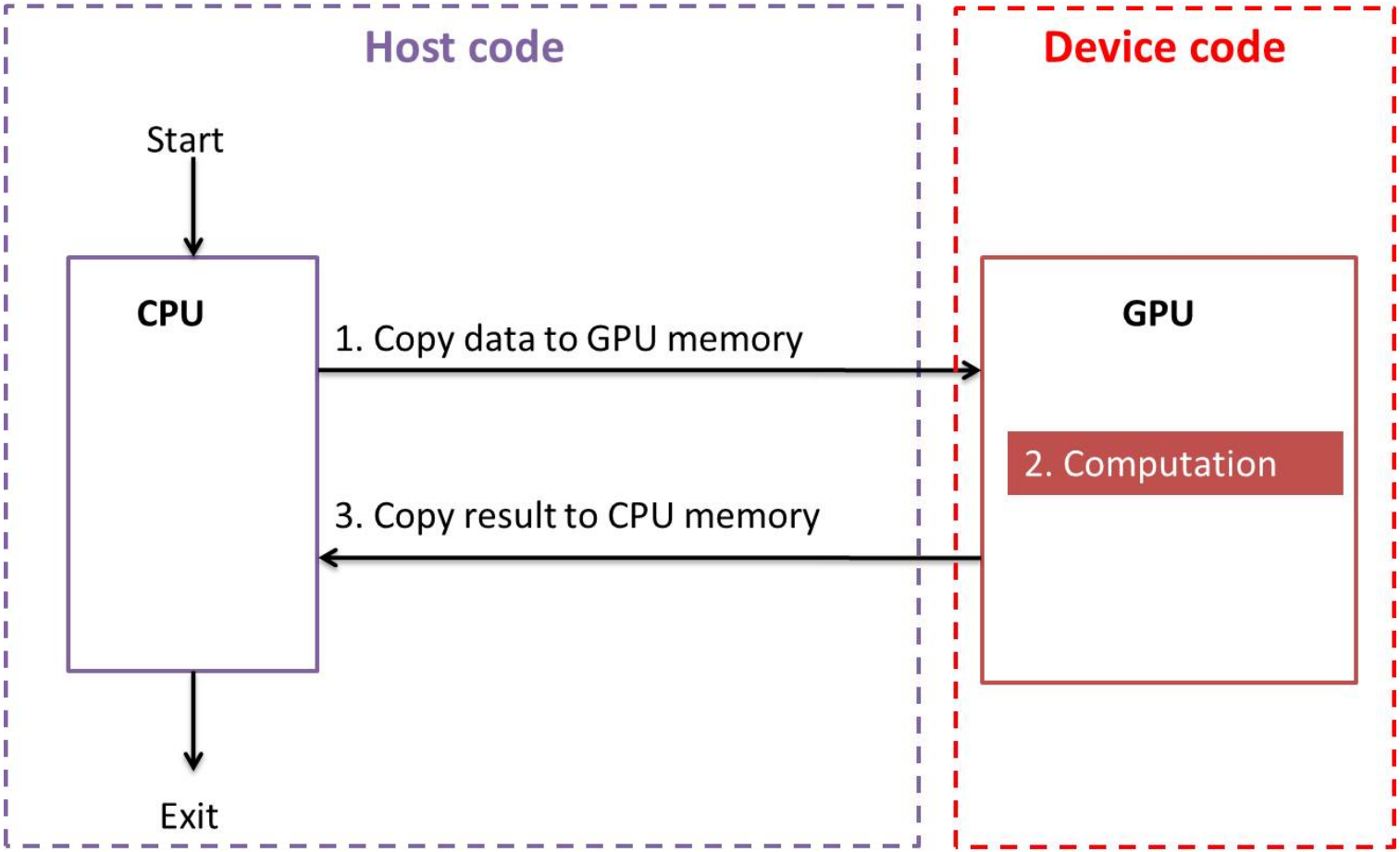
To run CUDA code, programmers need:

- Hardware. NVIDIA GPUs, from graphics cards for laptops and desktops to dedicated server products for computation
- Software. CUDA toolkit, device drivers(video card drivers), and programming SDK.

Host and Device Code

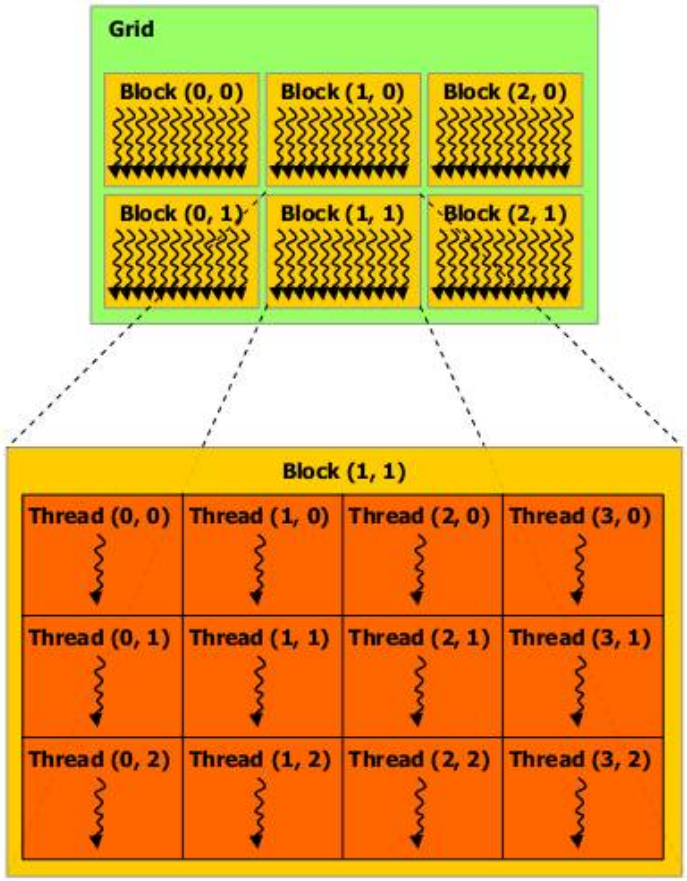
- A CUDA program consists of two parts: host and device (or kernel) code.
- Host code: executed on the CPU
  - Device Memory Allocation/Release
  - Memory copy between The GPU and The CPU
  - Kernel launching
- Device code: executed on the GPU
  - Concurrent computation on the GPU
- A CUDA program always starts from the host code, and then invokes the GPU kernels

Processing Flow of a CUDA Program



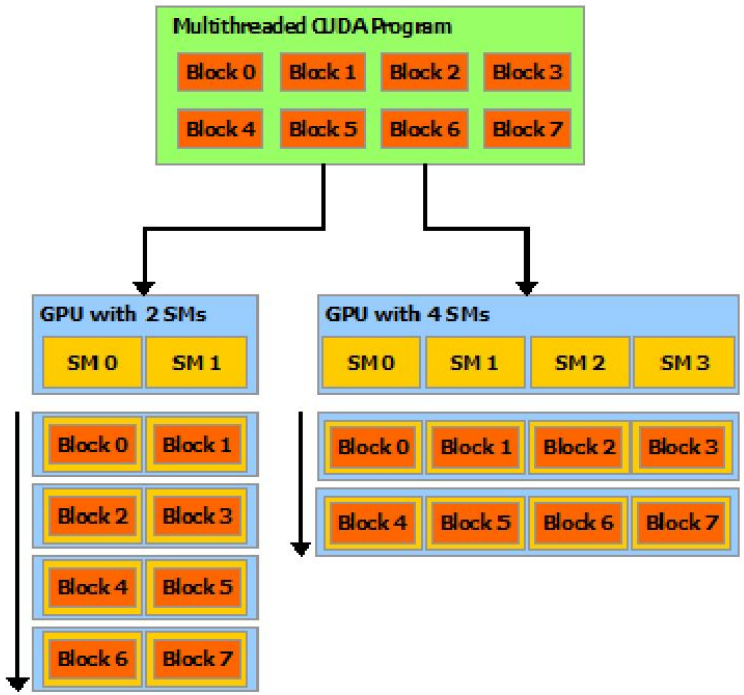
Kernel Launch

- Programmers determine how many concurrent threads to run
- Once the kernel function is launched in host code, threads will be executed on the GPU
- In CUDA programming model, concurrent threads are organized as grid-block-thread mode
  - Each kernel corresponds to a grid
  - Each grid consists of multiple thread blocks
  - Each thread block contains multiple threads

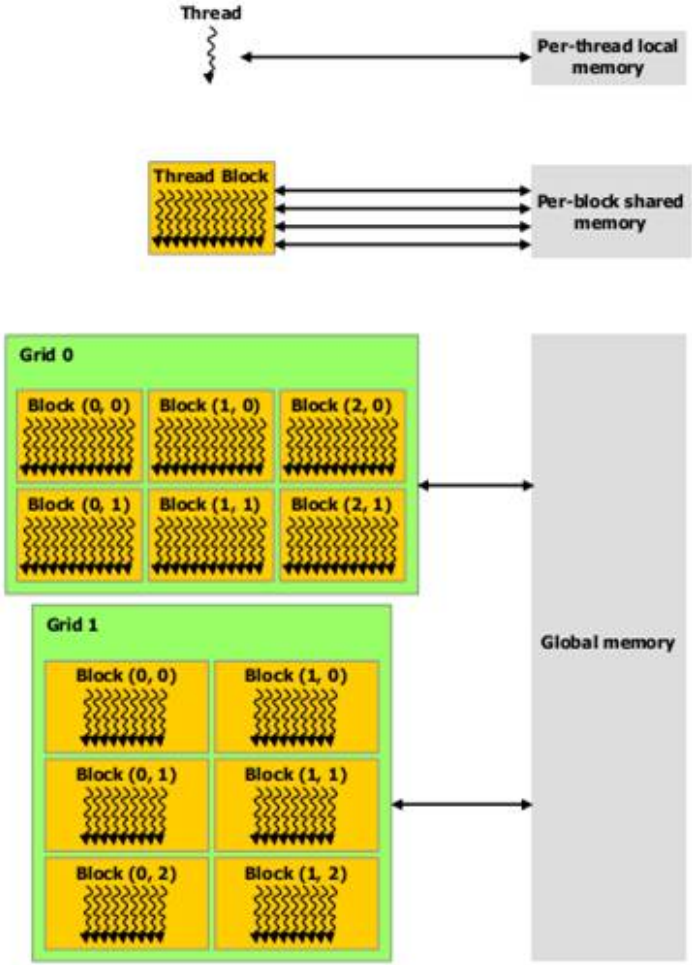


Kernel Launch

- Blocks of threads are scheduled to SMs for execution
- The way how they are scheduled is transparent to programmers
- Given more SMs, more blocks of threads are executed simultaneously



CUDA Memory Hierarchy



- Registers: only available within a thread
- Shared memory: accessed by threads in the same thread block
- Global memory(device memory): can be accessed by all threads

Basic Functions in Host Code

- GPU memory management functions
  - GPU memory allocation
    - ⋆ cudaMalloc(devPtr, size)
    - ⋆ cudaFree(devPtr)
  - Memory copy
    - ⋆ cudaMemcpy(dstPtr, srcPtr, size, direction), direction is a flag indicating the memory copy direction (from Host to Device or from Device to Host)
- Kernel execution
  - kernelName<<<block_size, thread_size>>>(param1, ...)

block_size : number of thread blocks in the grid
thread_size : number of threads per block
(para1, ...) : parameter list of the kernel function

Note : host code has no direct access to GPU memory

Device Code(Kernels)

- The device code is the same for each thread
- A kernel function has the prefix *__global__*, and has a void return type
- Threads are identified with built-in variables.
  - *blockIdx*, *threadIdx*: the index of block and thread.
  - *gridDim*, *blockDim*: the dimension size of grid and block.

Note: device code has no direct access to main memory

Example

- Add two vectors A and B of size N and stores the result into vector C
- The C program without CUDA

```
int main()
{
    int *h_A, *h_B, *h_C;
    int i;
    int N = 4096;
    size_t size = N * sizeof(int);

    // Allocate input vectors h_A and h_B in host memory
    h_A = (int*)malloc(size);
    h_B = (int*)malloc(size);
    h_C = (int*)malloc(size);

    /*initialize h_A and h_B here*/

    //vector Add
    for (i = 0; i < N; i++)
        h_C[i] = h_A[i] + h_B[i];

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

   return 0;
}
```

Example (Cont.d)

- The CUDA program for Vector Addition

Part 1 : Memory Allocation and Memory Copy

```
// Host code
int main()
{
    int *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;
    int N = 4096;
    size_t size = N * sizeof(int);

    // Allocate input vectors h_A and h_B in host memory
    h_A = (int*)malloc(size);
    h_B = (int*)malloc(size);
    h_C = (int*)malloc(size);

    for (int i = 0; i < N; i++)
    {
        h_A[i] = i;
        h_B[i] = i;
    }

    // Allocate vectors in device memory
    cudaMalloc((void**)&d_A, size);
    cudaMalloc((void**)&d_B, size);
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
                    .
                    .
```

Example (Cont.d)

- The CUDA program for Vector Addition

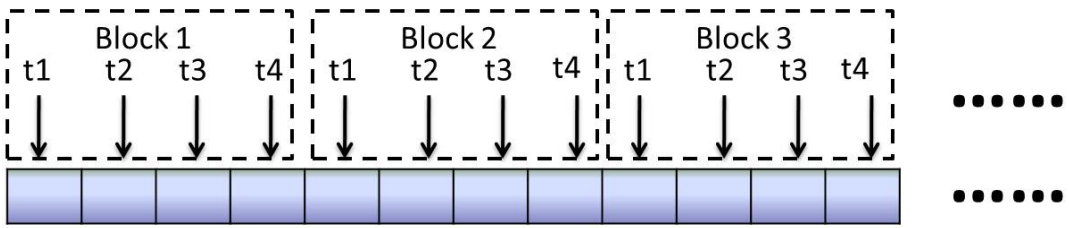
Part 2 : Kernel Launch

```
.
.
// Invoke kernel
int threadsPerBlock = 256;
int blocksPerGrid = N / threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);
.
.
```

The Kernel function is:

```
// Device code
__global__ void VecAdd(int* A, int* B, int* C)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    C[i] = A[i] + B[i];
}
```

Map thread index to array index:



Example (Cont.d)

- The CUDA program for Vector Addition

Part 3 : Copy Memory back to Host and Memory Release

```
    .
    .
    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

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

    //Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

}
```

A recommended common practice is to name a host-resident structure with the prefix 'h_' (host), and a device-resident structure with 'd_'(device)

The Parallelization on the GPU

- Shall we always make one thread handle one element?
- **NO!**
  - The numbers of blocks and threads for a kernel is limited, e.g., up to 65535 blocks and 1024 threads per block
  - A suitable number of threads should balance the degree of parallelism and resource usage
- Sometimes we should make each thread handle multiple elements

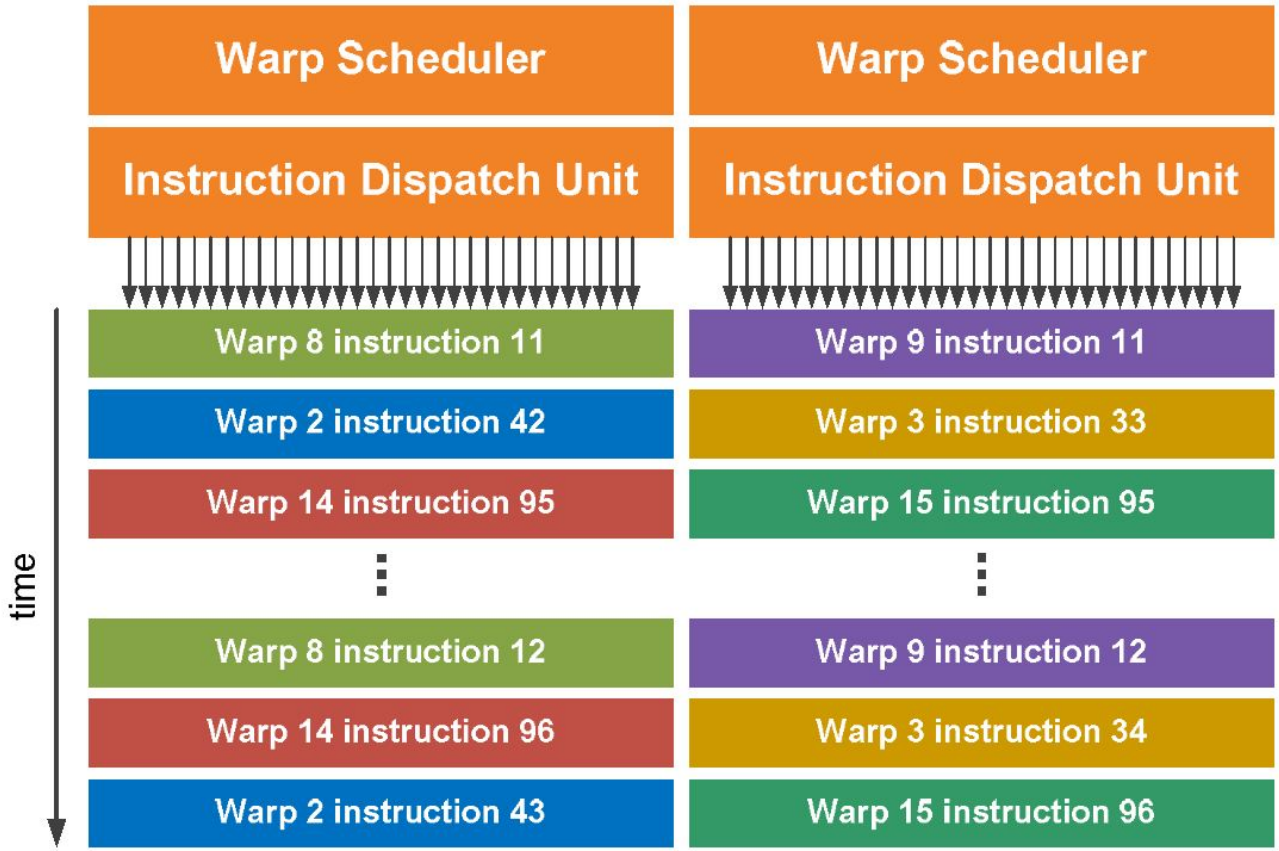
Performance Issue

- Writing Cuda program is simple
- The difficulty is how to fully utilize GPU resources
- We'll cover three performance-related issues
  - Warp
  - Shared Memory Usage
  - Coalesced Memory Access

Warp

- The Streaming Multiprocessor schedules threads in groups of 32 parallel threads called *warps*
  - Assume number of threads per block is 128, there're 4 warps.
  - warp 0: thread 0 - thread 31
  - warp 1: thread 32 - thread 63
  - warp 2: thread 64 - thread 95
  - warp 3: thread 96 - thread 127
- At each given time, threads of a warp are issued and are executed concurrently
- Threads of the same warp share a program counter, they are always synchronized
  - execute the same instruction

Warp (Cont.d)



Warp scheduling of NVIDIA Fermi Architecture

Warp (Cont.d)

Sometimes execution of threads of the same warp may diverge

- flow control instructions, e.g. if, switch, do, for, while

```
__global__ void kernel_fun(int *A, int *B)
{
        int i = blockIdx.x*blockDim.x + threadIdx.x;

        if (i % 2 == 0)
        {
                /*Execution Path 1: thread 0, 2, 4, 6...30... reach here*/

                A[threadIdx.x] += B[threadIdx.x];
        }
        else
        {
                /*Execution Path 2: thread 1, 3, 5, 7...31... reach here*/

                A[threadIdx.x] -= B[threadIdx.x];
        }

 /*even threads and odd threads go back to the same exexution path*/
}
```

In the above example, two execution paths must be serialized.

Warp (Cont.d)

A way to avoid warp divergence

- Let the first half threads of a block handle even indices of array
- Let the second half thread of a block handle odd indices of array

```
__global__ void kernel_fun(int *A, int *B)
{
        int base = blockIdx.x*blockDim.x;

        if (threadIdx.x < blockDim.x/2)
        {
                /*Execution Path 1: the first half threads of a block reach here*/

                int even_index = base + threadIdx.x*2;

                A[even_index] += B[even_index];
        }
        else
        {
                /*Execution Path 2: the second half threads of a block reach here*/

                int odd_index = base + (threadIdx.x - blockDim.x/2)*2 +1;

                A[odd_index] -= B[odd_index];
        }

 /*even threads and odd threads go back to the same exexution path*/
}
```

Warp (Cont.d)

A summary

- Number of threads per block should be multiple of 32
  - Assume the Block Size is 257, the last warp of this block only has 1 thread!
- Try to avoid divergence of threads in the same warp

Shared Memory Usage

- Delay of accessing global memory is high (several hundreds of clock circles)
- On-chip shared memory is much faster (tens of clock circles)
- A general procedures of using shared memory to improve performance
  - Load input from global memory to shared memory
  - Computation on data stored in share memory
  - Write output back from shared memory to global memory
- Shared Memory is declared using *__shared__* qualifier.
- Shared Memory is visible to all threads in a block
- Function *__syncthreads()* should be called before using shared memory
  - Block threads until all threads have reached that point
  - Ensure all threads in a block has loaded data to shared memory

Shared Memory Usage (Cont. d)

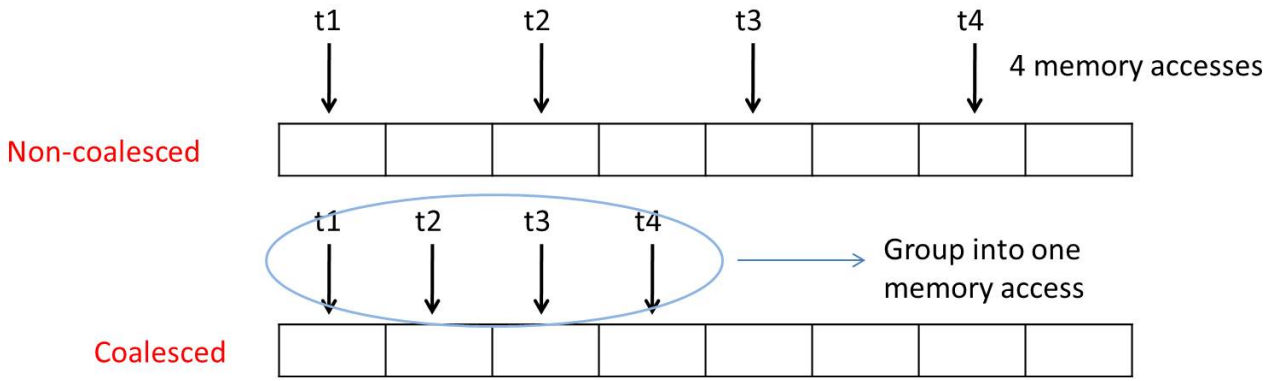
An Example

```
__global__ void compute_fun(int *data)
{
    int tid = threadIdx.x;
    __shared__ int myblock[1024];
    int tmp;

    // load data from global memory to shared memory
    myblock[tid] = data[tid];

    // ensure that all threads have loaded their values into
    // shared memory; Otherwise, one thread might be computing
    // on unitialized data.
    __syncthreads();

    // compute the average of this thread's left and right neighbors
    int pre = tid > 0 ? tid-1 : 1023;
    int pos = tid < 1023 ? tid+1 : 0;
    tmp = (myblock[pre] + myblock[pos])/2;

    // square the previous result and add my value, squared
    tmp = tmp*tmp + myblock[tid];

    // write the result back to global memory
    data[tid] = myblock[tid];
}
```

- This function requires many memory reads
- Performance may suffers if not use shared memory

Coalesced Memory Access

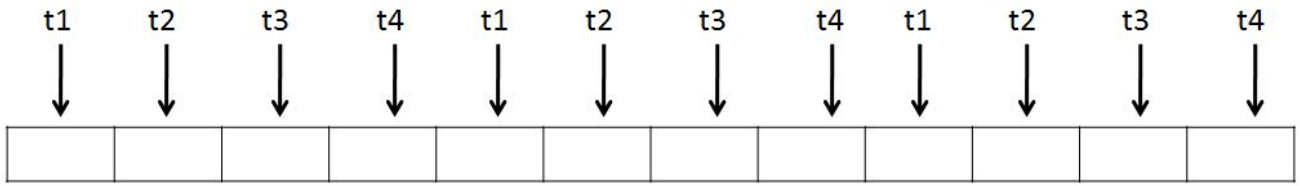
If memory addresses accessed by threads in the same block are consecutive, then these memory accesses are grouped into one memory transaction



Coalesced Memory Access (Cont. d)

A kernel function with Coalesced Memory Access

```
__global__
void kernel2(int* d_data, const int numElement) {
        const int tid = blockDim.x*blockIdx.x + threadIdx.x;
        const int nthread = blockDim.x*gridDim.x;

        for(int i = tid; i < numElement; i += nthread) {
                d_data[i] += 1;
        }
}
```



Some Resources

- CUDA C Programming Guide
- LLNS's tutorials on Parallel Computing