GPU Programming

Introduction to CUDA Programming

Learning Objectives

1. Write kernel code for vector addition
2. Explain basic CUDA programming terminologies
3. Explain vector addition CUDA code example

Cuda Code Example: Vector Add

```
// Kernel code
__global__ void vectorAdd(const float* A, const float* B, float* C, int numElements) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements) {
        C[i] = A[i] * B[i] + 0.0f;
    }
}

// Host code
int main() {
    // Allocate the device input vector A
    float* d_A = NULL;
    err = cudaMalloc((void**) &d_A, size);

    // Copy the host input vectors A and B in host memory to the device input
    // vectors in device memory
    printf("Copy input data from the host memory to the CUDA device\n");
    err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, numElements);

    // Copy the device result vector in device memory to the host result vector
    // in host memory
    printf("Copy output data from the CUDA device to the host memory\n");
    err = cudaMemcpy(h_C, d_C, size, cudaMemcyDeviceToHost);
```

1. Host code is executed on CPUs
2. Kernel coda is invoked with «<…»>
3. Kernel code is executed on GPUs

SPMD

1. GPU kernel code is single program multiple data
   - All threads execute the same program
   - There is no execution order among threads
   - But we need to make each thread execute different data

threadIdx.x

1. Even though each thread executes the same program, each thread has a unique identifier (each thread has built-in variable that represents the x-axis coordinate)
   - threadIdx.x

Vector Add with Thread IDs

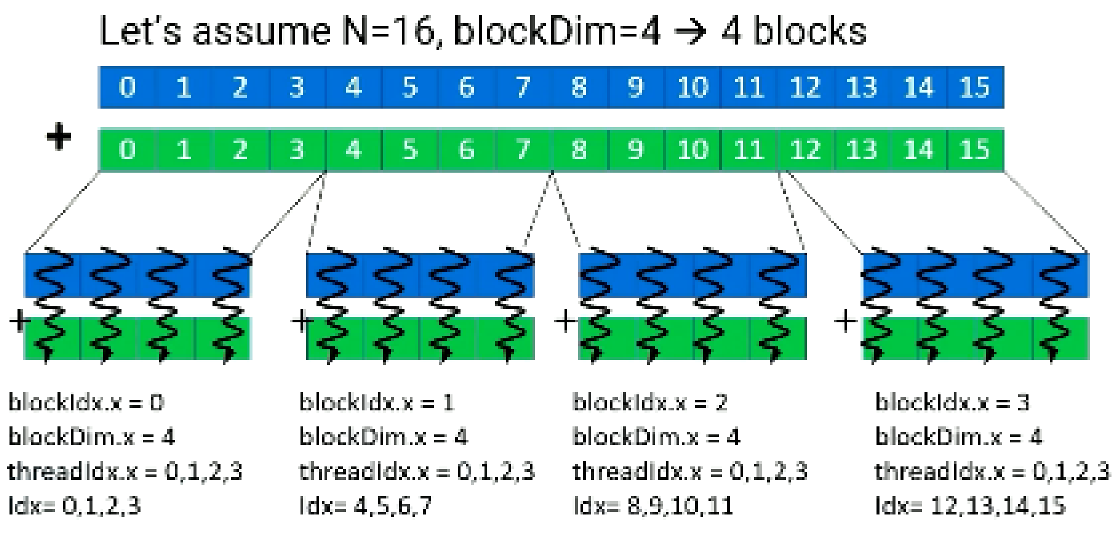
1. Each thread operates on a different element of vector data
    - int idx = threadIdx.x;
    - c[idx] = a[idx] + b[idx];

Execution Hierarchy

1. A group of threads forms a block
    - Group of threads is also called a warp
        - Warp is a microarchitecture concern
            * Pure hardware decision, programmer doesn't need to be aware
    - Block is a critical component of CUDA
2. CUDA block: A group of threads that are executed concurrently
3. Data is divided by block
4. For now, let's just assume that each block is executed on each GPU SM
    - SM: Streaming Multiprocessor
5. No ordering among CUDA block execution

Example of Data Indexing

1. Thread Index
    - threadIdx.x, threadIdx.y, threadIdx.z
2. Block Index
    - blockIdx.x, blockIdx.y
3. Block Dimensions
    - blockIdx.x * blockDim.x + threadIdx.x

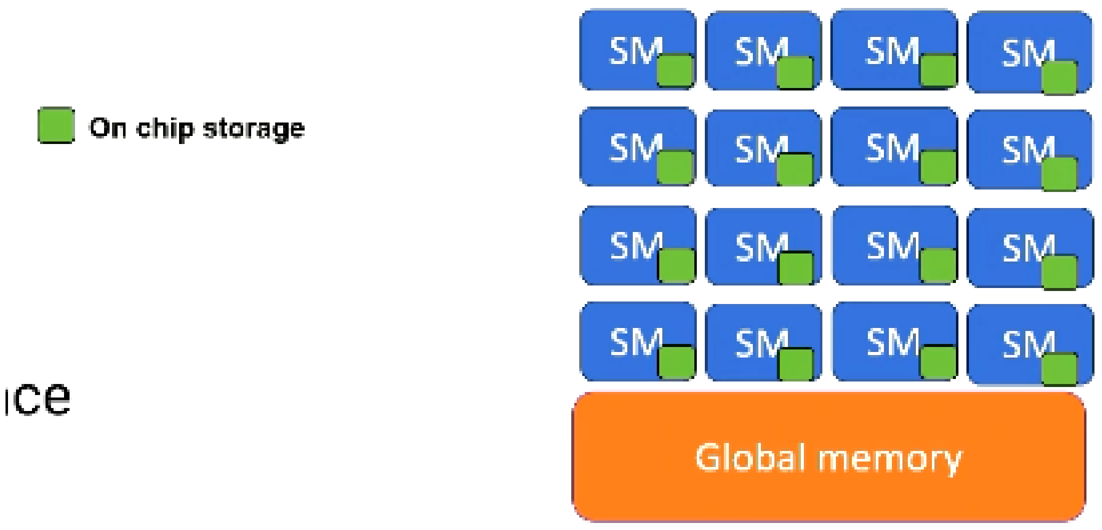


CUDA Blocks

Shared Memory

1. Scratchpad memory
2. Software-controlled memory space
3. Use ___shared___
4. On chip storage -> faster access compared to global memory

5. Accessible only within a CUDA block (later GPUs allow different policy)



Shared Memory

Execution Ordering: Threads and Blocks

1. No predefined ordering among threads or CUDA blocks

Thread Synchronizations

1. Wait for all threads to complete tasks
    - Typical usage: '___syncthreads()' within a CUDA block
    - Interblock synchronization: achieved through different kernel launches

Typical Usage with Thread Synchronizations

1. Typical computation flow
    - Load -> Compute -> Store
    - Load to the shared memory from global memory
    - Compute with the data in the shared memory
    - Store the results into global memory
2. BSP programming model (bulk synchronous parallel)

Kernel Launch

1. Launch kernel from the host code using '«< »>' syntax
    - kernelName«<numBlocks, threadsPerBlock»>(arguments)
2. Total number of threads in a kernel = numBlocks * threadsPerBlock
3. Grid consists of multiple blocks, block consists of multiple threads
4. Multiple different tasks -> multiple kernels
5. Each kernel (sortKernel, addKernel, storeKernel) handles different tasks sequentially

```
int main() {
    sortKernel<<<1,1>>>(d_data, dataSize);
    addKernel<<<gridDim,blockDim>>>(d_data, d_data, d_result, dataSize);
    storeKernel<<<1,1>>>(d_result, dataSize);
}
```

Memory Space and Block/Thread

	Scope	Note
Shared Memory	Within one CUDA block	
Global Memory	All blocks in a kernel	
Local Memory	Within a CUDA thread	
Constant Memory	All blocks in a kernel	Read only (3D-graphics)
Texture Memory	All blocks in a kernel	Read only (3D-graphics)

1. Information sharing is limited across "CUDA execution" hierarchy:
   - Data in the shared memory is visible only within one CUDA block
     - Data in shared memory can stay only in one SM
     - Data in the shared memory of one CUDA block needs explicit communcation (later CUDA supports thread block cluster to allow sharing)

Summary

1. Learned execution hierarchy, shared memory, global memory, kernel launch, thread synchronization

Occupancy

Learning Objectives

1. Explain the concept of occupancy
2. Determine how many CUDA blocks can run on one Streaming Multiprocessor

How Many CUDA Blocks on One Stream Multiprocessor?

1. So far, we assumed on CUDA block per SM
   - In reality, there are multiple blocks per SM

Occupancy

1. How many CUDA blocks can be executed on one SM is decided by the following parameters:
   - # of registers
   - shared memory
   - # of threads
2. Exact hardware configurations are varied by GPU microarchitecture
3. Example
   - Each SM can execute 256 threads, 64K registers, 32 KB shared memory
     - One CUDA block: 32 threads and 2KB shared memory and each thread has 64 registers
     - Constrain by running threads: 256/32 = 8 CUDA blocks
     - Constrain by register size: 64 * 1024 / (64 * 32) = 32 CUDA blocks
     - Constrain by shared memory = 32KB / 2KB = 16 CUDA blocks
   - Final answer: 8 CUDA blocks per SM

Number of Threads Per CUDA Block

1. Host sets the number of threads per CUDA blocks
2. Set up at the kernel launch time
3. # of registers per CUDA block
   - Compiler sets how many architecture registers are needed for each CUDA block (will be covered in later part of the course)
   - Typical CPU ISAs, # of registers per thread is fixed (e.g., 32), but in CUDA, this is flexible
4. Shared memory size is determined from the code

- ___shared___ int sharedMemory[1000]; -> 4000B

Why Do We Care about Occupancy?

1. Higher occupancy means more parallelism
2. Utilization: Utilizing more hardware resources generally leads to better performance
3. Exceptional cases will be studied in later lectures

Summary

1. Occupancy depends on # of threads per CUDA block, register, and shared memory usages
2. Higher occupancy generally leads to more parallelism

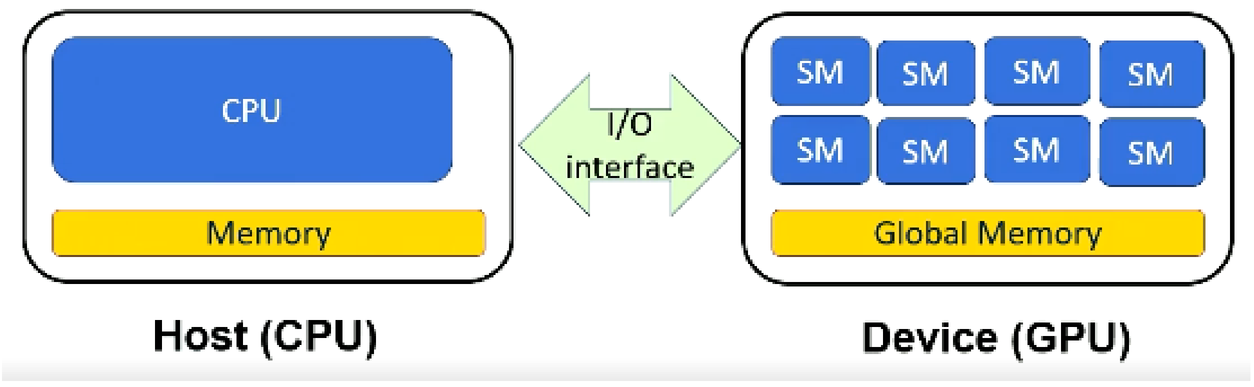
Host Code

Learning Objectives

1. Write the host code for vector addition operation
2. Explain the management of global memory

Managing Global Memory

1. Shared memory: Visible within CUDA block
2. Global memory: Visible among all CUDA blocks
3. A host program needs explicit APIs to manage device memory (global memory)



Global Memory

CUDA API

1. cudaMalloc: allocate memory on the GPU
2. cudaMemcpy: transfer data between CPU and GPU
3. Unified memory: Eliminates the need for explicit data copies
    - Covered in later lectures

Summary

1. Host launches kernel and manages device memory including allocation and data transfer

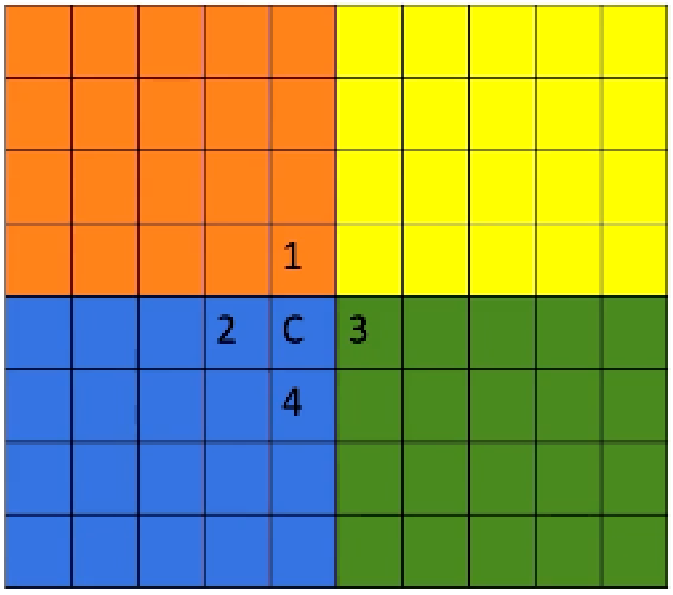
Stencil Operation with CUDA

Learning Objectives

1. Be able to write a stencil code with CUDA shared memory and synchronization

Recap: Stencil Operations

1. A common pattern in HPC, perform computation with neighboring data
2. Each CUDA block performs different calculation
   - Each CUDA block operates each color zone
3. Operations are fully parallel
4. Data decomposition: Each thread handles one element of the stencil operation
5. One element will be used at least 4 times
   - Good candidates for caching



Stencil Operations

Stencil Operations: Operations per Thread

1. Stage 1: Load data from global memory to shared memory (on-chip storage)
2. Stage 2: Perform stencil operations
3. Stage 3: Write results back
4. Thread synchronization between stage 1 and stage 2

How about Boundaries?

1. C position cannot access elements 1 and 3
2. Solution: bring neighboring elements together and pad with zeros outside of the original size

```
// load data into shared memory
__shared__ float sharedInput[sharedDim][sharedDim];
```

```
int sharedX = threadIdx.x + filterSize / 2;
int sharedY = threadIdx.y + filterSize / 2;
```

Use if-else to Check Boundary or Not

1. Load different values on boundaries

```
if (x >= 0 && x < width && y >= 0 && y < height) {
    sharedInput[sharedY][sharedX] = input[y * width + x];
} else {
    sharedInput[sharedY][sharedX] = 0.0f; // handle boundary conditions
}
```

2. Perform computations only on inner elements

```
// apply the filter to the pixel and its neighbors using shared memory
for (int i = 0; i < filterSize; i++) {
    for (int j = 0; j < filterSize; j++) {
        result += sharedInput[threadIdx.y + i][threadIdx.x + j] * filter[i][j];
    }
}
```

Summary

1. Explored writing a stencil operation kernel with shared memory
2. Discussed how to handle boundary elements with if-else statements