CMSC 5743

Efficient Computing of Deep Neural Networks

Lecture 05: CUDA Programming

Bei Yu
CSE Department, CUHK
byu@cse.cuhk.edu.hk

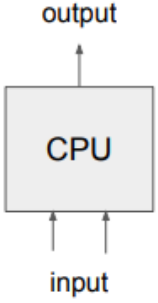
(Latest update: October 22, 2021)

Fall 2021

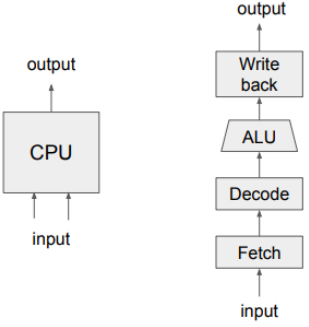
Overview

GPU Architecture

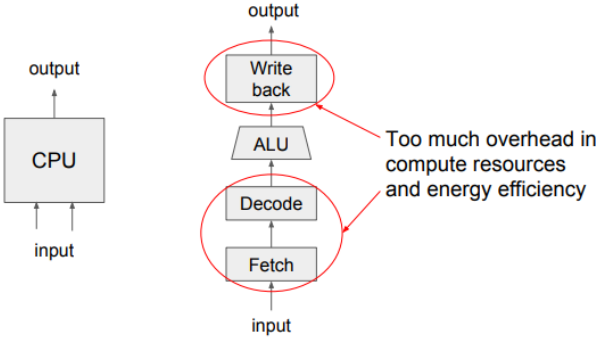
CPU vs GPU



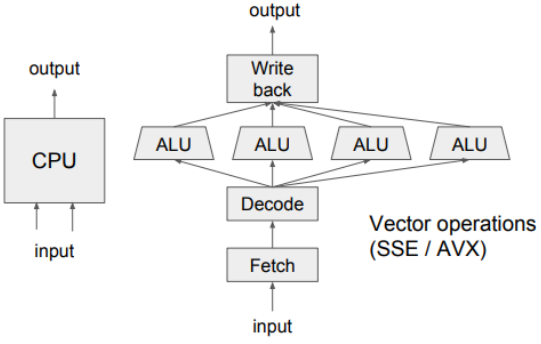
CPU vs GPU



CPU vs GPU

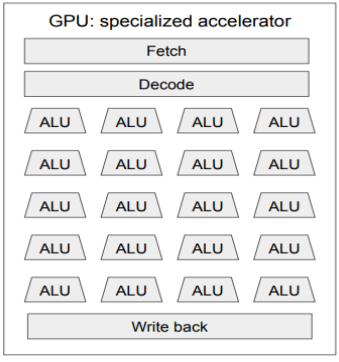

output
CPU
input
output
Write back
ALU
Decode
Fetch
input
Too much overhead in compute resources and energy efficiency

CPU vs GPU


output
CPU
input
output
Write back
ALU
ALU
ALU
ALU
Decode
Fetch
input
Vector operations
(SSE / AVX)

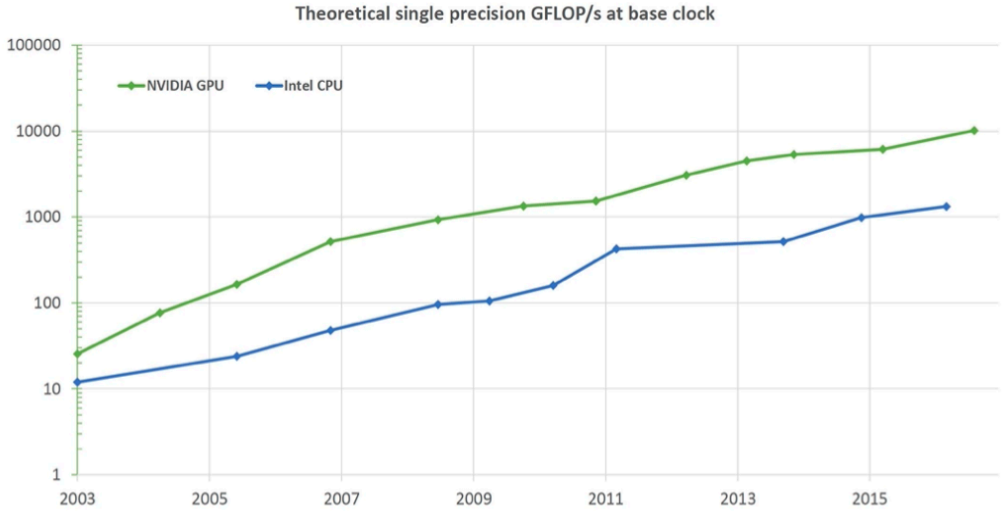
CPU vs GPU

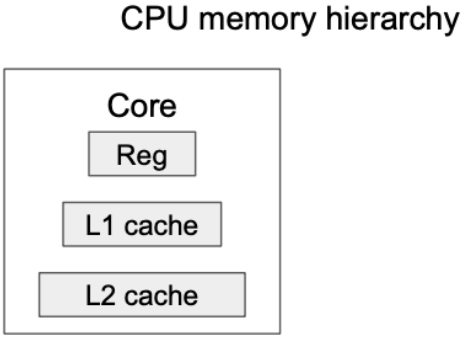
SIMD: single instruction multiple data



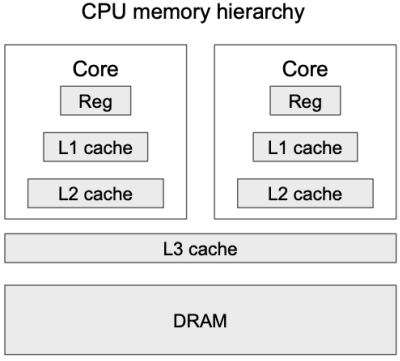
GPU Architecture

Theoretical peak FLOPS comparison

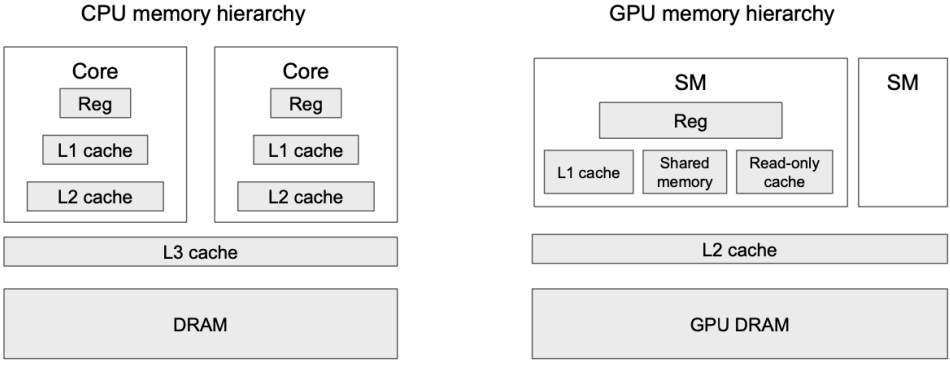



CPU memory hierarchy
Core
Reg
L1 cache
L2 cache

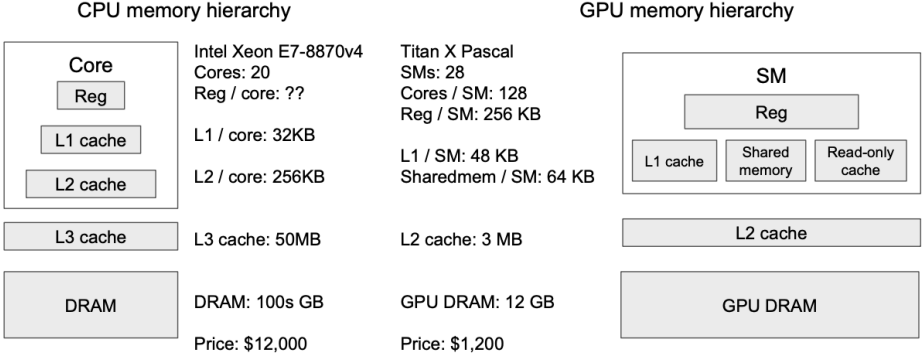
Memory Hierarchy



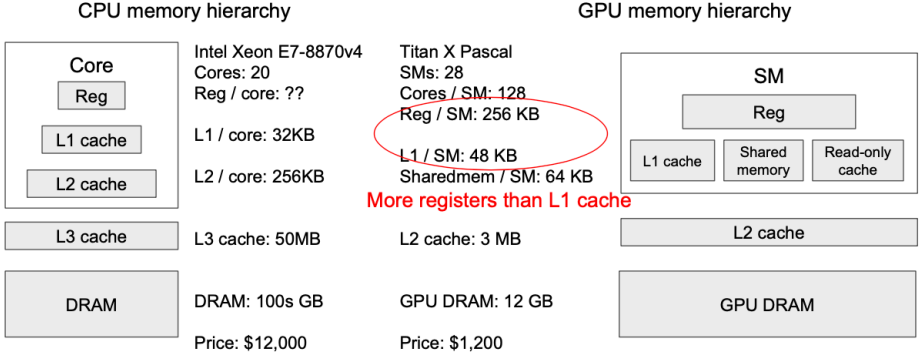
Memory Hierarchy



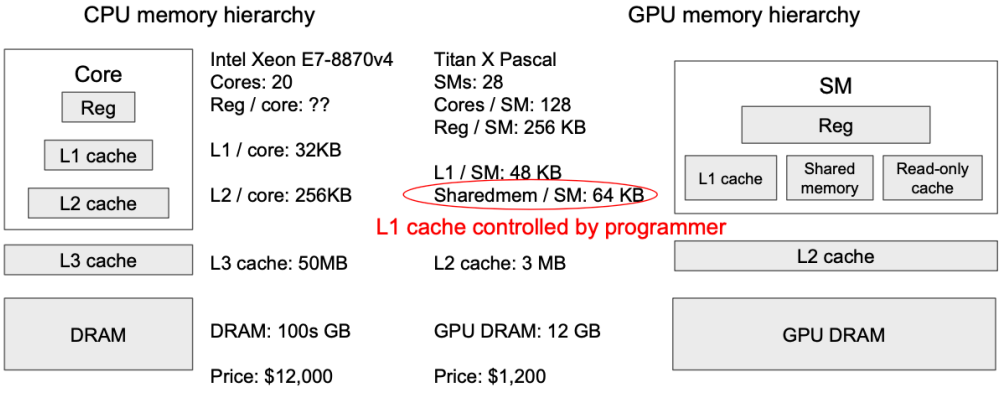
Memory Hierarchy



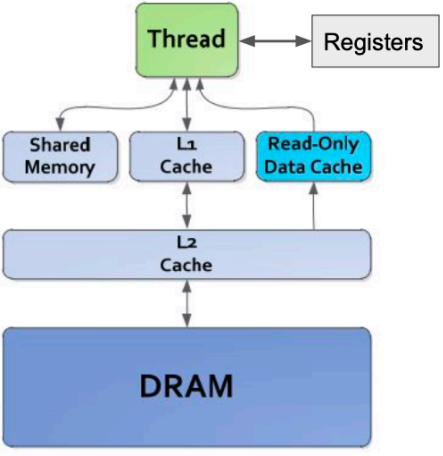
Memory Hierarchy



Memory Hierarchy



Memory Hierarchy



Registers: R 0 cycle / R-after-W ~20 cycles

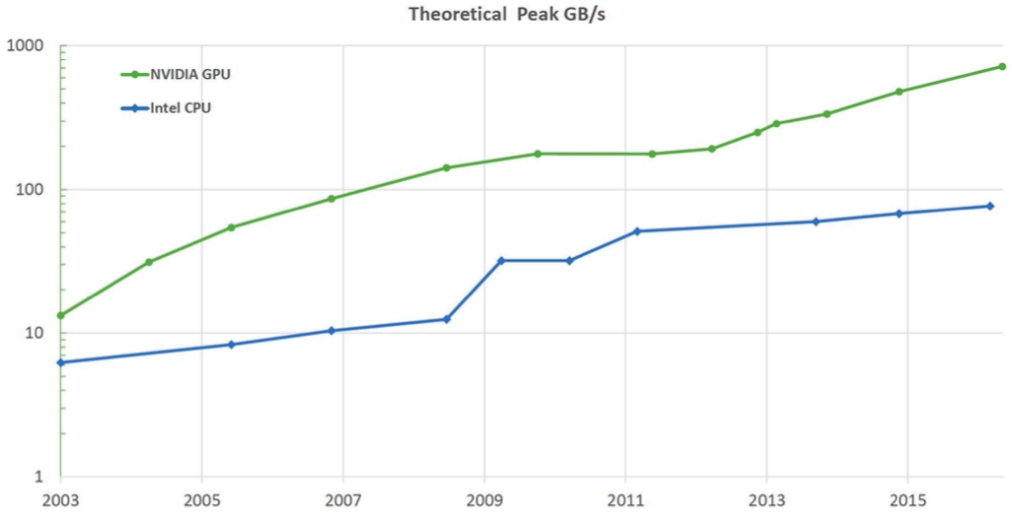
L1/texture cache: 92 cycles
Shared memory: 28 cycles
Constant L1 cache: 28 cycles

L2 cache: 200 cycles

DRAM: 350 cycles

(for Nvidia Maxwell architecture)

Memory bandwidth comparison



Nvidia GPU Comparison

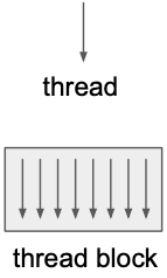
GPU	Tesla K40 (2014)	Titan X (2015)	Titan X (2016)
Architecture	Kepler GK110	Maxwell GM200	Pascal GP102
Number of SMs	15	24	28
CUDA cores	2880 (192 * 15SM)	3072 (128 * 24SM)	3584 (128 * 28SM)
Max clock rate	875 MHz	1177 MHz	1531 MHz
FP32 GFLOPS	5040	7230	10970
32-bit Registers / SM	64K (256KB)	64K (256KB)	64K (256KB)
Shared Memory / SM	16 KB / 48 KB	96 KB	64 KB
L2 Cache / SM	1.5 MB	3 MB	3 MB
Global DRAM	12 GB	12 GB	12 GB
Power	235 W	250 W	250 W

CUDA Programming Model

Programming model: SIMT

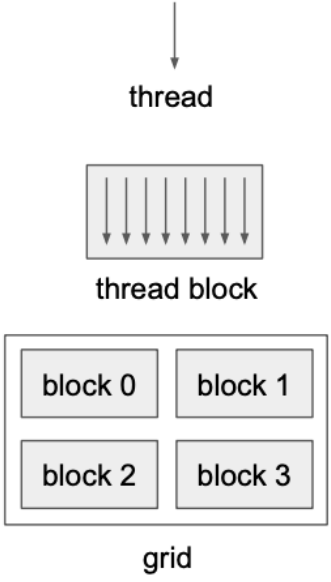
- SIMT: Single Instruction, Multiple Threads
- Programmer writes code for a single thread in simple C program.
  - All threads executes the same code, but can take different paths.

Programming model: SIMT



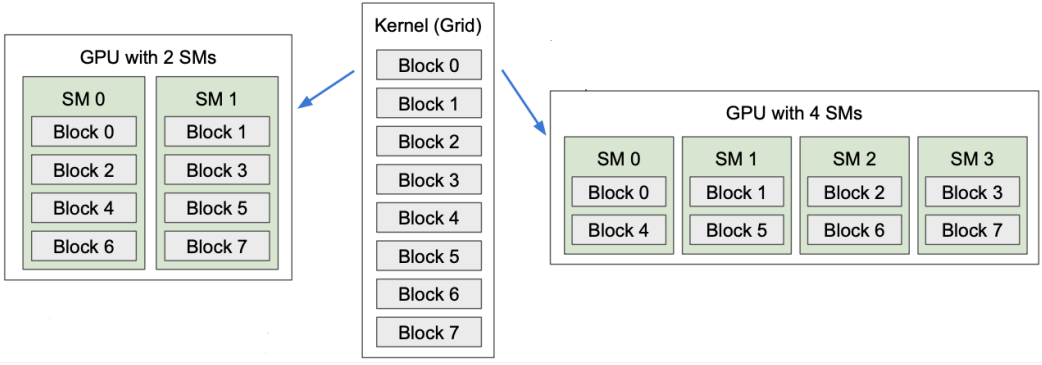
- SIMT: Single Instruction, Multiple Threads
- Programmer writes code for a single thread in simple C program.
  - All threads executes the same code, but can take different paths.
- Threads are grouped into a block
  - Threads within the same block can synchronize execution

Programming model: SIMT



- SIMT: Single Instruction, Multiple Threads
- Programmer writes code for a single thread in simple C program.
  - All threads executes the same code, but can take different paths.
- Threads are grouped into a block
  - Threads within the same block can synchronize execution
- Blocks are grouped into a grid
  - Blocks are independently scheduled on the GPU, can be executed in any order.
- A kernel is executed as a grid of blocks of threads

- Each block is executed by one SM and does not migrate.
- Several concurrent blocks can reside on one SM depending on block's memory requirement and the SM's memory resources.



Kernel Exectuion

- A warp consists of 32 thread.
  - A warp is the basic schedule unit in kernel execution.
- A thread block consists of 32-thread warps.
- Each cycle, a warp scheduler selects one ready warps and dispatches the warps to CUDA cores to execute.

Control Flow

```
100: ...
101: if (condition) {
102:     ...
103: } else {
104:     ...
105: }
```

Control Flow

```
100: ...
101: if (condition) {
102:     ...
103: } else {
104:     ...
105: }
```

Control Flow

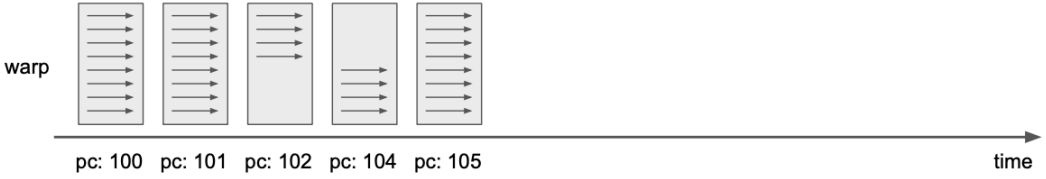
```
100: ...
101: if (condition) {
102:     ...
103: } else {
104:     ...
105: }
```

Control Flow

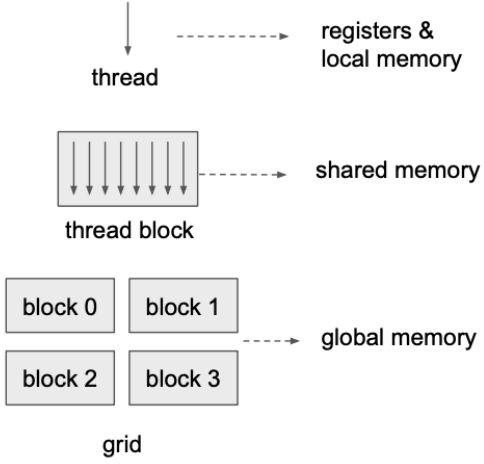
```
100: ...
101: if (condition) {
102:     ...
103: } else {
104:     ...
105: }
```

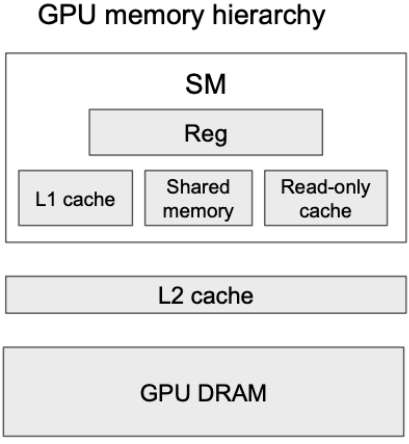
Control Flow

```
100: ...
101: if (condition) {
102:     ...
103: } else {
104:     ...
105: }
```



Thread Hierarchy & Memory Hierarchy





Example: Vector Add

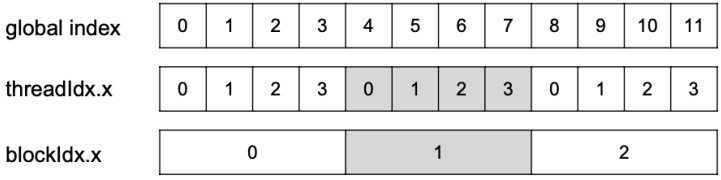
```
// compute vector sum C = A + B
Void vecAdd_cpu(const float* A, const float* B, float* C, int n) {
    for (int i = 0; i < n; ++i)
        C[i] = A[i] + B[i];
}
```

Example: Vector Add

```
// compute vector sum C = A + B
Void vecAdd_cpu(const float* A, const float* B, float* C, int n) {
    for (int i = 0; i < n; ++i)
        C[i] = A[i] + B[i];
}
```

```
__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

Example: Vector Add

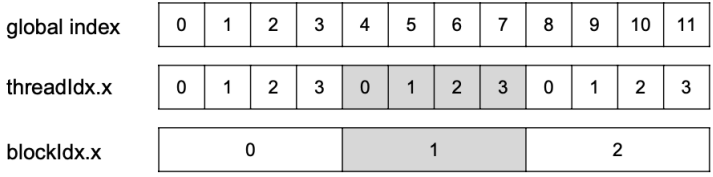


Suppose each block only includes 4 threads:
blockDim.x = 4

```
__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

Compute the global index

Example: Vector Add



Suppose each block only includes 4 threads: blockDim.x = 4

```
__global__ void vecAddKernel(const float* A, const float* B, float* C, int n) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

Each thread only performs one pair-wise addition

Example: Vector Add (Host)

```
#define THREADS_PER_BLOCK   512
void vecAdd(const float* A, const float* B, float* C, int n) {
    float *d_A, *d_B, *d_C;
    int size = n * sizeof(float);
    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);
    int nblocks = (n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
    vecAddKernel<<<nblocks, THREADS_PER_BLOCK>>>(d_A, d_B, d_C, n);
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

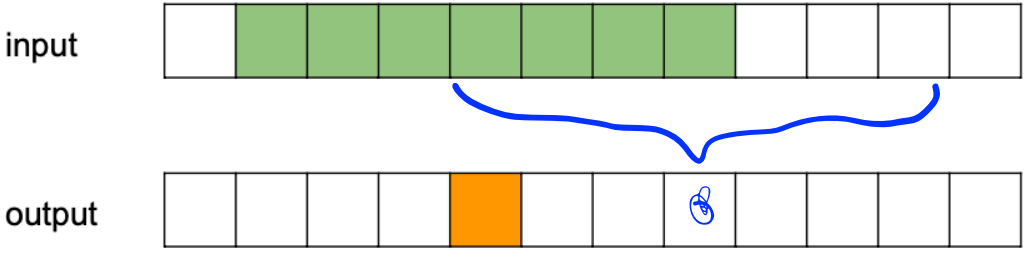
Example: Vector Add (Host)

```
#define THREADS_PER_BLOCK   512
void vecAdd(const float* A, const float* B, float* C, int n) {
    float *d_A, *d_B, *d_C;
    int size = n * sizeof(float);
    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);
    int nblocks = (n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;
    vecAddKernel<<<nblocks, THREADS_PER_BLOCK>>>(d_A, d_B, d_C, n);
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}
```

Launch the GPU kernel asynchronously

Example: Sliding Window Sum

- Consider computing the sum of a sliding window over a vector
  - Each output element is the sum of input elements within a radius
  - Example: image blur kernel
- If radius is 3, each output element is sum of 7 input elements



A naive implementation

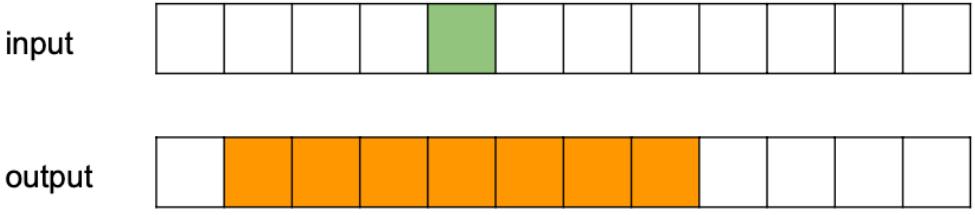
```
#define RADIUS 3
__global__ void windowSumNaiveKernel(const float* A, float* B, int n) {
    int out_index = blockDim.x * blockIdx.x + threadIdx.x;
    int in_index = out_index + RADIUS;
    if (out_index < n) {
        float sum = 0.;
        for (int i = -RADIUS; i <= RADIUS; ++i) {
            sum += A[in_index + i];
        }
        B[out_index] = sum;
    }
}
```

A naive implementation

```
void windowSum(const float* A, float* B, int n) {
    float *d_A, *d_B;
    int size = n * sizeof(float);
    cudaMalloc((void **) &d_A, (n + 2 * RADIUS) * sizeof(float));
    cudaMemset(d_A, 0, (n + 2 * RADIUS) * sizeof(float));
    cudaMemcpy(d_A + RADIUS, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    dim3 threads(THREADS_PER_BLOCK, 1, 1);
    dim3 blocks((n + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK, 1, 1);
    windowSumNaiveKernel<<<blocks, threads>>>(d_A, d_B, n);
    cudaMemcpy(B, d_B, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B);
}
```

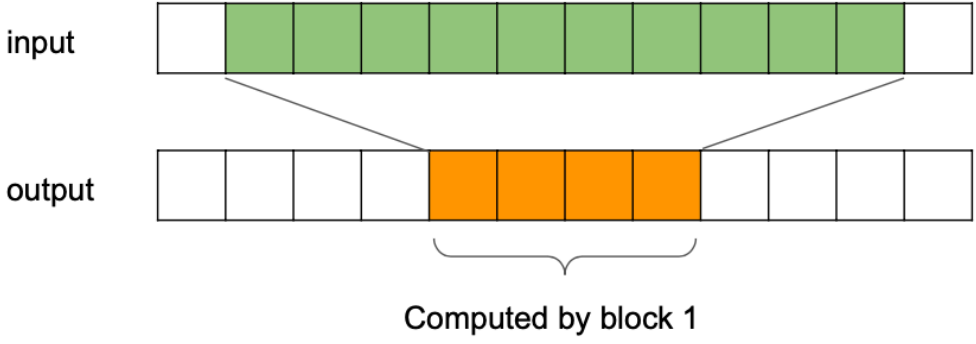
How to improve it?

- For each element in the input, how many times it is loaded?
  - Each input element is read 7 times!
  - Neighboring threads read most of the same elements
- How can we avoid redundant reading of data?



Sharing data between threads within a block

- A thread block first cooperatively loads the needed input data into the shared memory.



Kernel with shared memory

```
__global__ void windowSumKernel(const float* A, float* B, int n) {
    __shared__ float temp[THREADS_PER_BLOCK + 2 * RADIUS];
    int out_index = blockDim.x * blockIdx.x + threadIdx.x;
    int in_index = out_index + RADIUS;
    int local_index = threadIdx.x + RADIUS;
    if (out_index < n) {
        temp[local_index] = A[in_index];
        if (threadIdx.x < RADIUS) {
            temp[local_index - RADIUS] = A[in_index - RADIUS];
            temp[local_index + THREADS_PER_BLOCK] = A[in_index+THREADS_PER_BLOCK];
        }
        __syncthreads();
```

Kernel with shared memory

```
        float sum = 0.;
        for (int i = -RADIUS; i <= RADIUS; ++i) {
            sum += temp[local_index + i];
        }
        B[out_index] = sum;
    }
}
```

Case Study: Parallel Reduction

Parallel Reduction

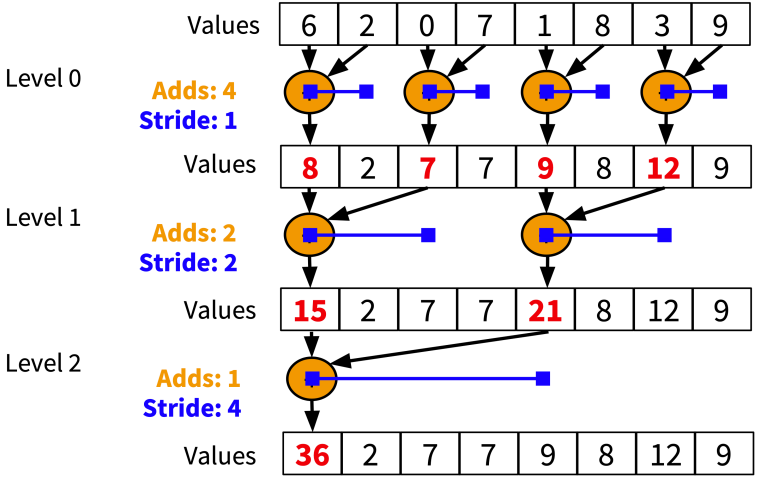
- **Sum Reduction:** Adding up the elements of an array
- Sequential approach:

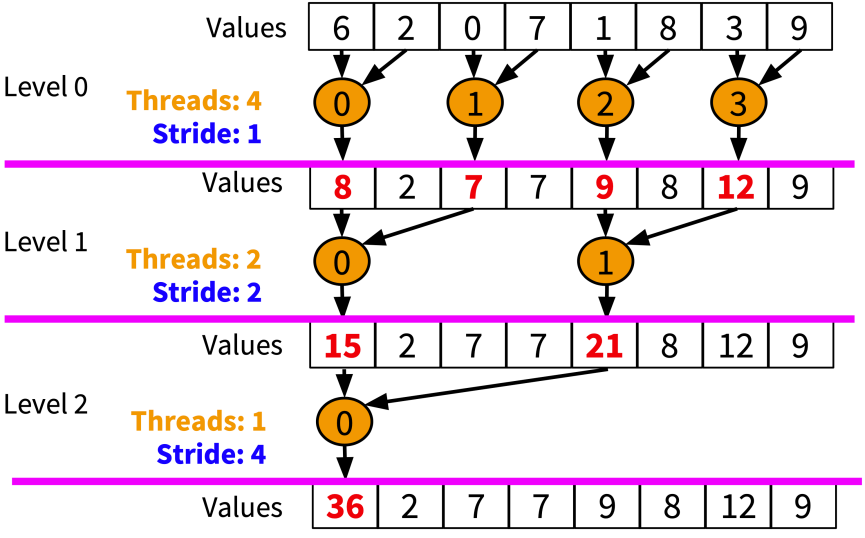
```
1  float sum = 0;
2  for ( int i = 0;i < n;i++ )
3  {
4      sum += array[i];
5  }
```

CUDA Programming for Parallel Reduction

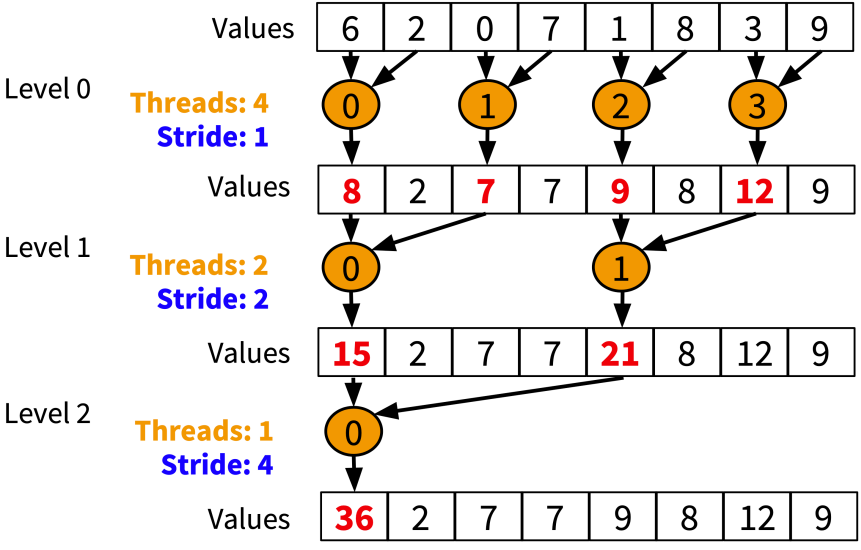
- How can we do this in parallel using threads?
- 1. Create one thread for every pair of elementes
  2. All threads add in parallel
     - This gives us a bunch of partial sums
  3. Repeat from (1) using the partial sums
     - Until we have left with a single value

An In-Place Array Implementation

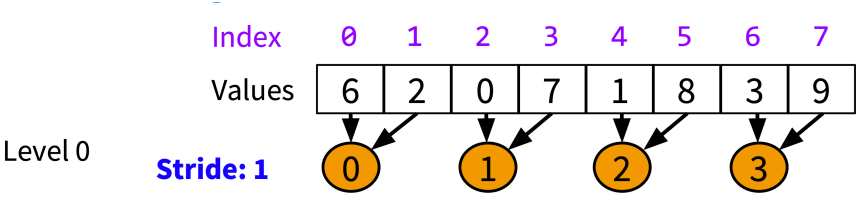



Values
6 2 0 7 1 8 3 9
Level 0
Threads: 4
Stride: 1
0 1 2 3
Values
8 2 7 7 9 8 12 9
Level 1
Threads: 2
Stride: 2
0 1
Values
15 2 7 7 21 8 12 9
Level 2
Threads: 1
Stride: 4
0
Values
36 2 7 7 9 8 12 9

Looping through Levels

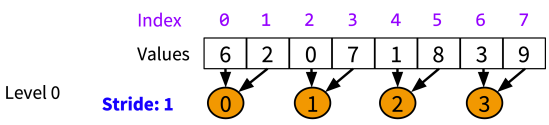


Calculating Indices



- What is the index of each left number?
  - 0, 2, 4, 6
- We have:
  - level 0: $left = id \times 2$
  - level 1: $left = id \times 4$
- What changes between levels?
  - stide:
    - level 0: $stride = 1$
    - level 1: $stride = 2$
- We can get left = id $\times$ (strides $\times$ 2)

A General Formula



- So we have: left = id $\times$ (strides $\times$ 2)
- If we know left, then: right = left + stride

Writing a Kernel

```
1   __global__ void reduce(float *array, int n) {
2       int id = threadIdx.x;
3       int threads;
4       int stride;
5       int left, right;
6       threads = n / 2;
7       for (stride = 1; stride < n; stride *= 2, threads /= 2)
8       { if (id < threads) {
9           left = id * (stride * 2);
10          right = left + stride;
11          array[left] = array[left] + array[right];
12      }
13      __syncthreads();}
14  }
```

Limitations to this approach

- Our kernel assumes we have enough threads to run level 0 in a single block
  - $\frac{n}{2}$ threads
- **Problem:** Max block size on our GPU is 1024 threads
  - At level 0, each threads adds 2 elements
  - With 1024 threads, we can add a 2048 element array
  - Any more, and we need to use another block
- Can we split the summation across multiple blocks?