CUDA Programming Tutorial 1

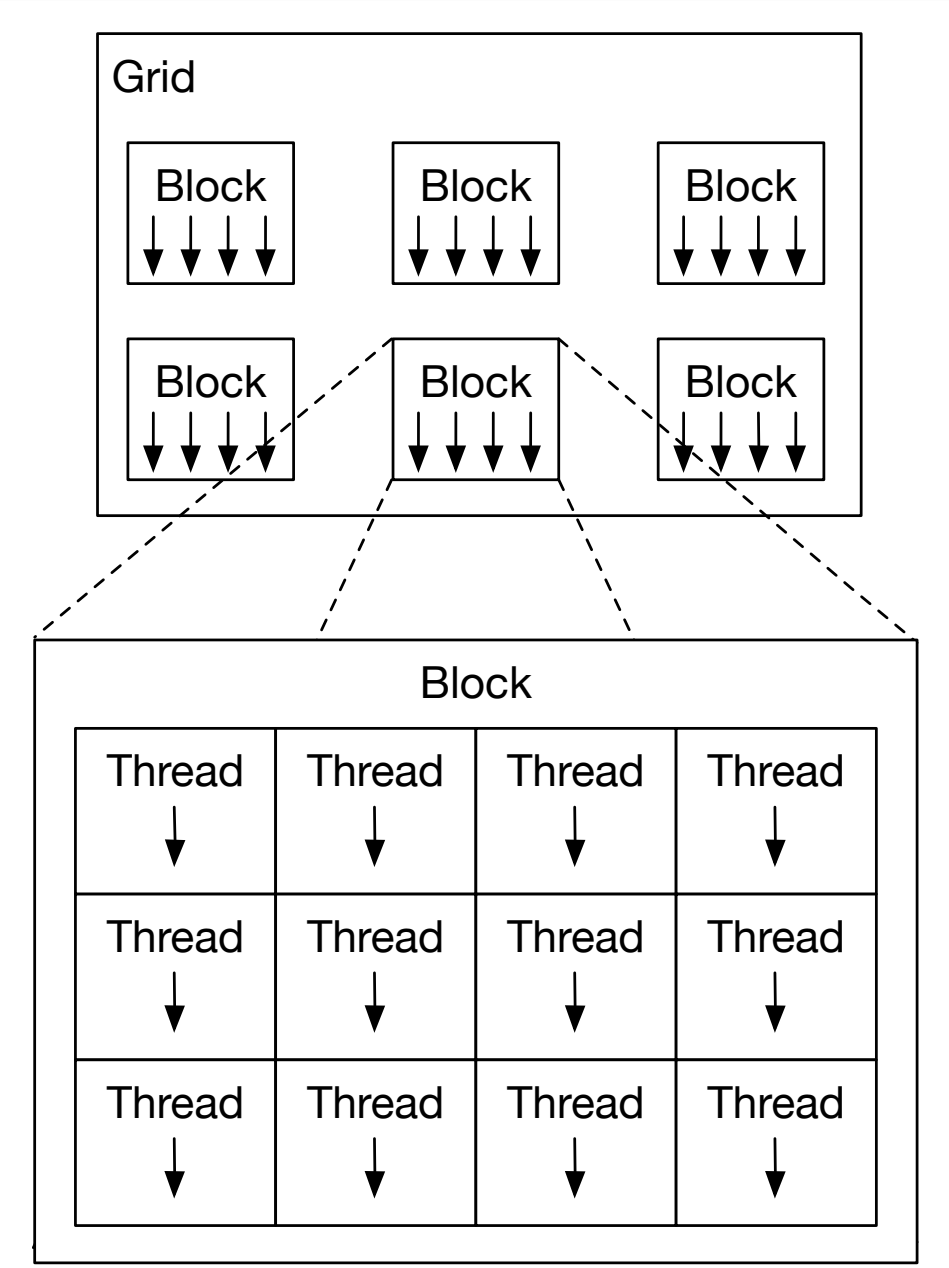
Sungjoo Ha

April 6th, 2017

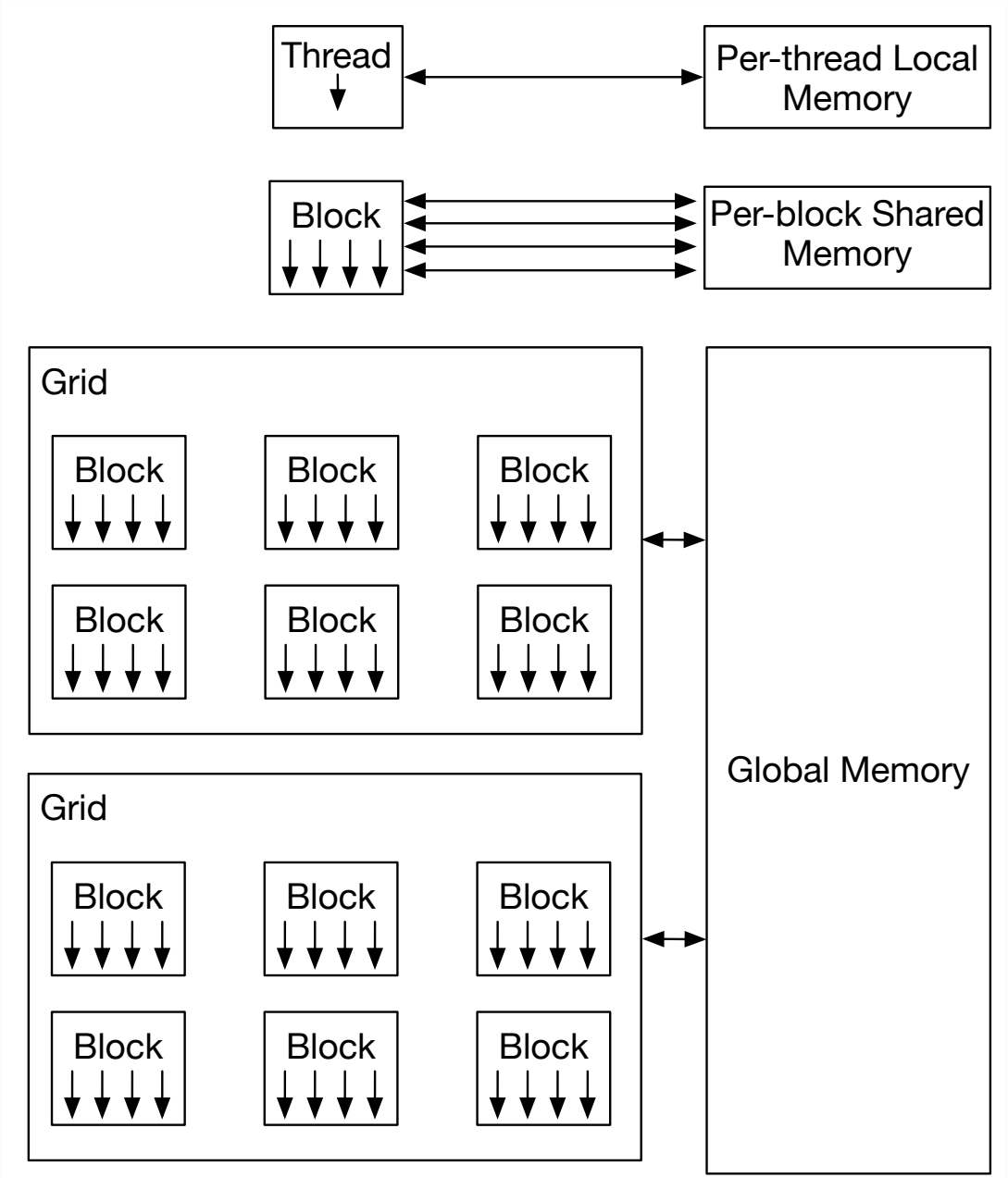
CUDA Platform

- NVidia에서 나온 GPU 병렬 프로그래밍 플랫폼

CUDA Execution Model

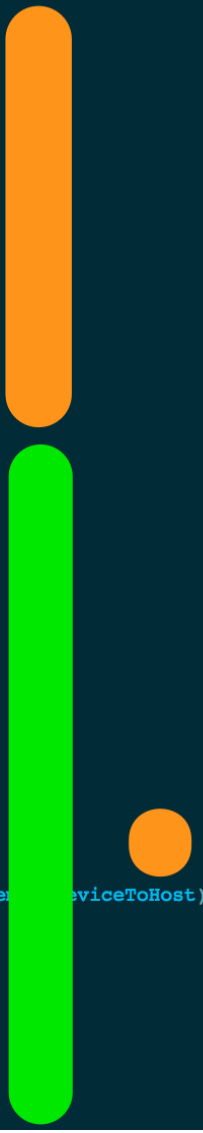


CUDA Memory Model



Heterogeneous Computing

```cuda
#include <cuda.h>
#include <stdio.h>
#include <stdlib.h>


#define blocksPerGrid 64
#define threadsPerBlock 512

__global__ void reduce(int *a, int *result, int length) {
    __shared__ int bucket[threadsPerBlock];

    int offset = blockDim.x * blockIdx.x + threadIdx.x;

    bucket[threadIdx.x] = a[offset];
    __syncthreads();

    int i = blockDim.x / 2;
    while (i != 0) {
        if (threadIdx.x < i) {
            bucket[threadIdx.x] += bucket[threadIdx.x + i];
        }
        __syncthreads();
        i /= 2;
    }

    if (threadIdx.x == 0) {
        result[blockIdx.x] = bucket[0];
    }
}

int main(void) {
    srand(time(NULL));

    int N = blocksPerGrid * threadsPerBlock;
    int data[N];

    for (int i = 0; i < N; ++i) {
        data[i] = rand() % 5;
    }

    int *d_data;
    int *d_result;
    int block_results[blocksPerGrid];
    int size = N * sizeof(int);

    cudaMalloc((void **) &d_data, size);
    cudaMalloc((void **) &d_result, blocksPerGrid * sizeof(int));
    cudaMemcpy(d_data, &data, size, cudaMemcpyHostToDevice);

    reduce<<<blocksPerGrid, threadsPerBlock>>>(d_data, d_result, N);

    cudaMemcpy(&block_results, d_result, blocksPerGrid * sizeof(int), cudaMe viceToHost);

    result = 0;

    for (int i = 0; i < blocksPerGrid; ++i) {
        result += block_results[i];
    }

    printf("Device: %d\n", result);

    return 0;
}
```



Process Workflow

- Host(CPU) 메모리에서 데이터를 Device(GPU) 메모리로 복사
- Device 프로그램을 실행
- Device의 메모리에서 결과를 Host 메모리로 복사

Hello World

```c
#include <stdio.h>

int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

- gcc로 컴파일하는 것이 보통이나 NVidia에서 제공하는 nvcc 컴파일러를 사용해서 컴파일 가능

Device Version of Hello World

```
#include <stdio.h>

__global__ void mykernel(void) {
}

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

- `__global__`
  - 해당 코드가 device에서 수행되어야 함을 표시
  - 또한 host 코드가 해당 함수를 호출함을 알려줌
- `mykernel<<<1,1>>>();`
  - 각괄호는 host에서 device 코드를 수행하는 것을 표시
  - 이를 kernel 실행이라 부름

Addition on the Device

```
#include <cuda.h>
#include <stdio.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}

int main(void) {
    int a, b, c;
    int *d_a, *d_b, *d_c;
    int size = sizeof(int);

    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    a = 2;
    b = 7;

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<1,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    printf("%d\n", c);

    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

Memory Management and Kernel Launch

- 포인터를 인자로 받는 kernel
  `add(int *a, int *b, int *c)` 를 만들고
- 필요한 메모리 할당을 수행 `cudaMalloc`
- 해당 메모리를 채워 넣고 `cudaMemcpy`
- Kernel 실행 `add<<<1,1>>>(d_a, d_b, d_c)`
- 결과를 메모리로부터 복사 `cudaMemcpy`
- 메모리 해제 `cudaFree`

Vector Addition

```
#include <cuda.h>
#include <stdio.h>

#define N 8

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


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

    int *d_a, *d_b, *d_c;

    int size = N * sizeof(int);

    cudaMalloc((void **) &d_a, size); cudaMalloc((void **) &d_b, size); cudaMalloc((void **) &d_c, size);

    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

    add<<<N,1>>>(d_a, d_b, d_c);

    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

    for (int i=0; i < N; ++i) {
        printf("%d\n", c[i]);
    }

    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

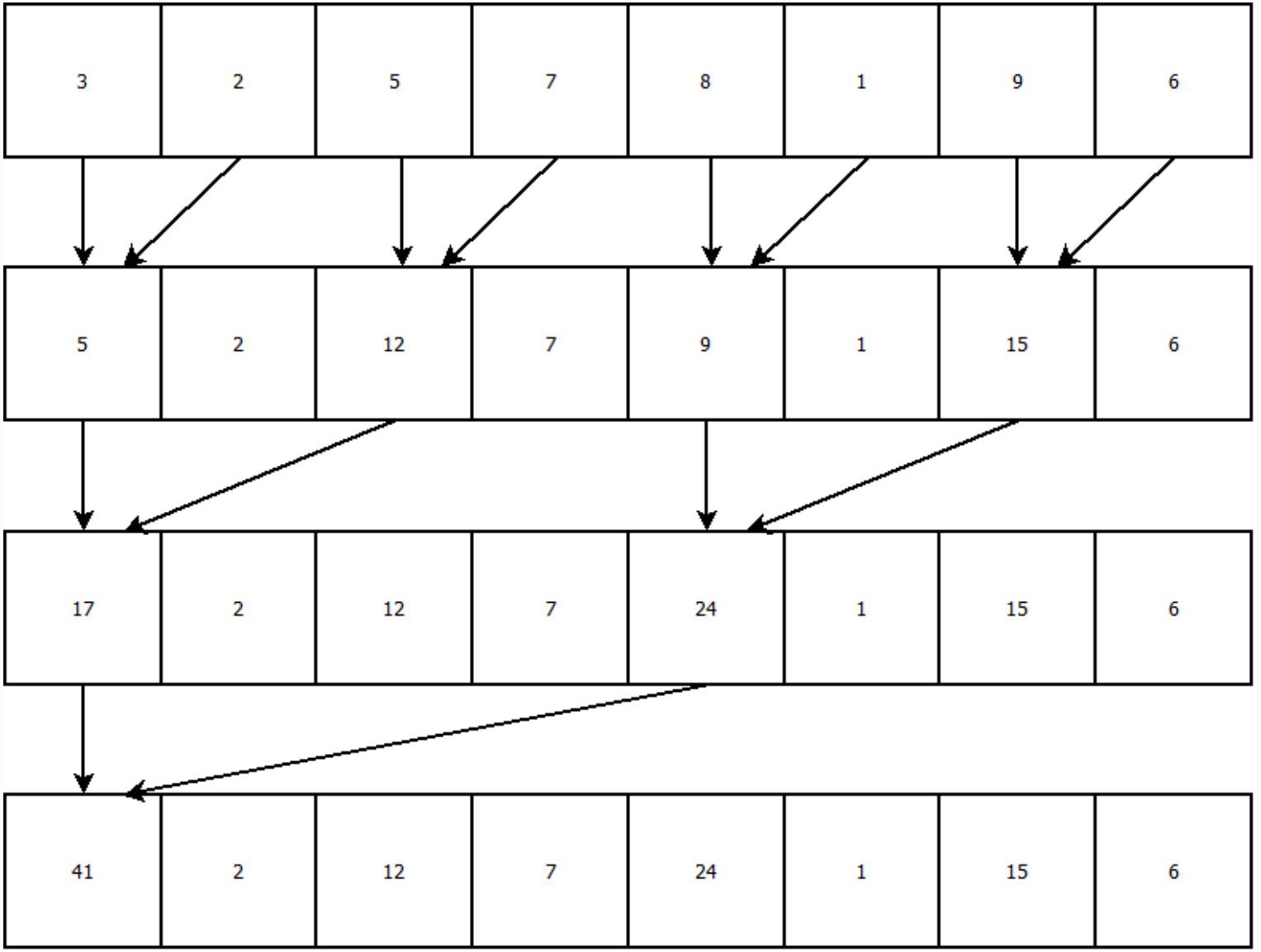
Threaded Version of Vector Addition

- Block 대신 thread 단위로 쪼개기
- `c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];`
- `add<<<1,N>>>(d_a, d_b, d_c);`

Block vs Thread

- 동일 block 안의 thread는 shared memory를 공유
  - 이를 활용한 synchronization 가능
- Block 간의 synchronization은 직접적인 kernel 호출 외에는 불가능

Reduction



Reduction on the Device

```
#include <cuda.h>
#include <stdio.h>
#include <stdlib.h>


#define blocksPerGrid 64
#define threadsPerBlock 512

__global__ void reduce(int *a, int *result, int length) {
    __shared__ int bucket[threadsPerBlock];

    int offset = blockDim.x * blockIdx.x + threadIdx.x;

    bucket[threadIdx.x] = a[offset];
    __syncthreads();

    int i = blockDim.x / 2;
    while (i != 0) {
        if (threadIdx.x < i) {
            bucket[threadIdx.x] += bucket[threadIdx.x + i];
        }
        __syncthreads();
        i /= 2;
    }

    if (threadIdx.x == 0) {
        result[blockIdx.x] = bucket[0];
    }
```

Reduction on the Host

```
int main(void) {
    srand(time(NULL));

    int N = blocksPerGrid * threadsPerBlock;
    int data[N];

    for (int i = 0; i < N; ++i) {
        data[i] = rand() % 5;
    }

    int *d_data;
    int *d_result;
    int block_results[blocksPerGrid];
    int size = N * sizeof(int);

    cudaMalloc((void **) &d_data, size);
    cudaMalloc((void **) &d_result, blocksPerGrid * sizeof(int));
    cudaMemcpy(d_data, &data, size, cudaMemcpyHostToDevice);

    reduce<<<blocksPerGrid, threadsPerBlock>>>(d_data, d_result, N);

    cudaMemcpy(&block_results, d_result, blocksPerGrid * sizeof(int), cudaMemcpyDeviceToHost);

    result = 0;

    for (int i = 0; i < blocksPerGrid; ++i) {
        result += block_results[i];
    }

    printf("Device: %d\n", result);

    return 0;
}
```

Reduction

- `__shared__`로 shared memory 할당
  - 이는 각 block 마다 따로 할당되는 메모리
- `int offset = blockDim.x * blockIdx.x + threadIdx.x;`
  - 몇 번째 block의 몇 번째 thread인지 판단
- `__syncthreads();`로 해당 block의 thread를 synchronize
- `result[blockIdx.x] = bucket[0];`로 특정 block의 계산 결과를 device 메모리에 저장
- `result += block_results[i];`로 모든 block의 연산을 최종 취합

References

- CUDA C/C++ Basics, http://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf