CUDA Programming Tutorial 3

Parallel Reduction

Sungjoo Ha

April 27th, 2017

Parallel Reduction

- 가장 기본적인 병렬 알고리즘
- 다른 병렬 알고리즘의 재료

Kernel Launch

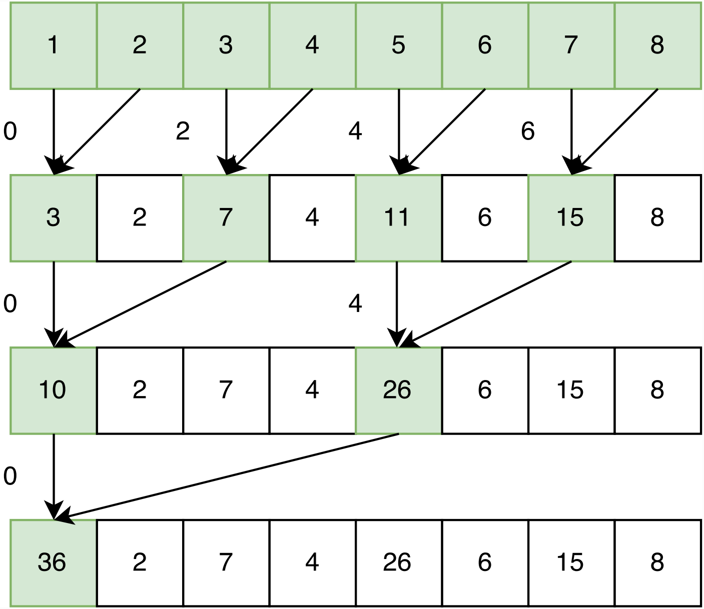
```
reduce<<<blocksPerGrid, threadsPerBlock>>>(d_data, d_block_results, N);
reduce<<<1, blocksPerGrid>>>(d_block_results, d_result, blocksPerGrid);
```

Reduce 1

Naive

```
__global__ void reduce1(int *idata, int *odata, unsigned int n) {
    __shared__ int sdata[threadsPerBlock];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    int result = 0;

    while(i < n) {
        result += idata[i];
        i += blockDim.x * gridDim.x;
    }

    sdata[tid] = result;
    __syncthreads();

    for (unsigned int s = 1; s < blockDim.x; s *= 2) {
        if (tid % (2 * s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    if (tid == 0) {
        odata[blockIdx.x] = sdata[0];
    }
}
```

Reduce 1



Reduce 1

Reduce 1	4.7008 ms

[1]

- % 연산이 느릴 가능성
- Thread divergence

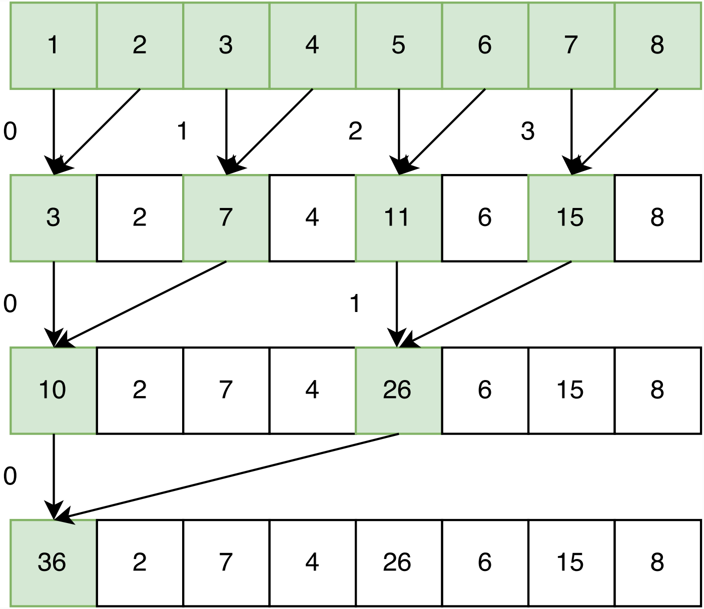
[1] 512 block, 512 thread

Reduce 2

Non-Divergent, Possible Bank Conflict

```
__global__ void reduce2(int *idata, int *odata, unsigned int n) {
    __shared__ int sdata[threadsPerBlock];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    int result = 0;

    while(i < n) {
        result += idata[i];
        i += blockDim.x * gridDim.x;
    }

    sdata[tid] = result;
    __syncthreads();

    for (unsigned int s = 1; s < blockDim.x; s *= 2) {
        int index = 2 * s * tid;

        if (index < blockDim.x) {
            sdata[index] += sdata[index + s];
        }

        __syncthreads();
    }

    if (tid == 0) {
        odata[blockIdx.x] = sdata[0];
    }
}
```

Reduce 2



Reduce 2

Reduce 1	4.7008 ms
Reduce 2	4.7484 ms

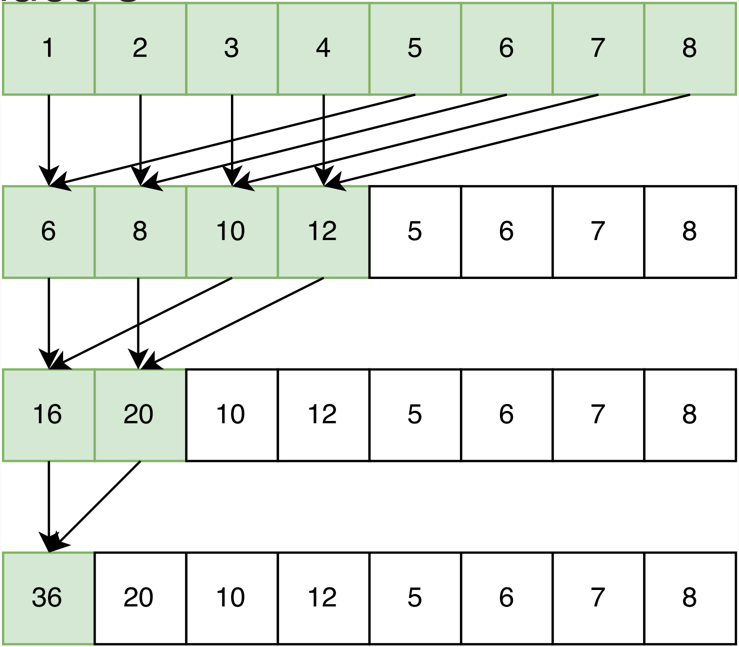
- % 연산자 제거
- Thread divergence 제거
- Bank conflict 가능성?

Reduce 3

Non-Divergent, No Bank Conflict

```
__global__ void reduce3(int *idata, int *odata, unsigned int n) {
    __shared__ int sdata[threadsPerBlock];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    int result = 0;

    while(i < n) {
        result += idata[i];
        i += blockDim.x * gridDim.x;
    }

    sdata[tid] = result;
    __syncthreads();

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

    if (tid == 0) {
        odata[blockIdx.x] = sdata[0];
    }
}
```

Reduce 3



Reduce 3

Reduce 1	4.7008 ms
Reduce 2	4.7484 ms
Reduce 3	4.7218 ms

- Bank conflict 제거 (옛날 하드웨어에서도)

Warp Shuffle Operation

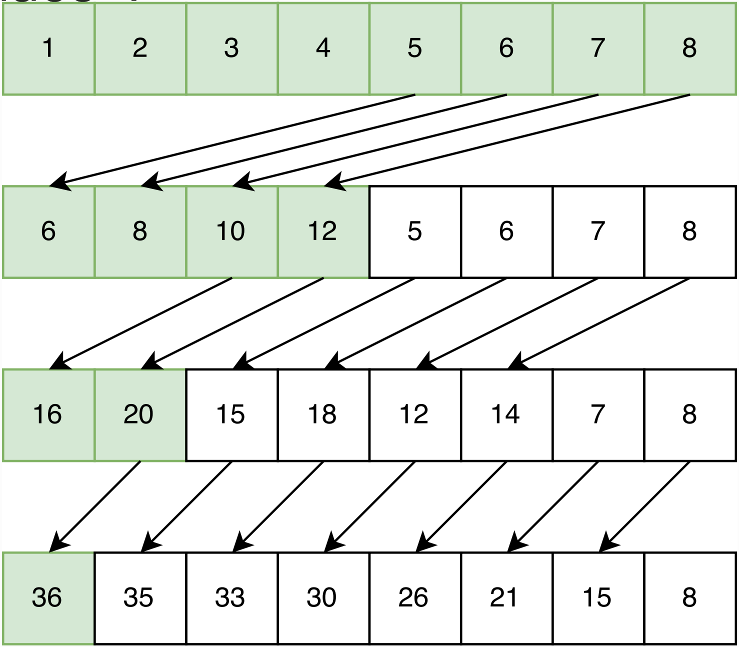
- 동일 warp에 존재하는 thread가 서로의 register를 볼 수 있는 기능
- 기존에는 shared memory를 사용해야만 했던 부분을 우회할 수 있음
- `__shfl`
- `__shfl_up`
- `__shfl_down`
- `__shfl_xor`

Reduce 4

Warp Shuffle Operation

```
__global__ void reduce4(int *idata, int *odata, unsigned int n) {
    /* same code for initializing shared memory */

    for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
        if (tid < s) {
            sdata[tid] = result = result + sdata[tid + s];
        }

        __syncthreads();
    }


    if (tid < 32) {
        if (threadsPerBlock >=  64) result += sdata[tid + 32];
        for (int offset = warpSize/2; offset > 0; offset /= 2)
        {
            result += __shfl_down(result, offset);
        }
    }

    if (tid == 0) {
        odata[blockIdx.x] = result;
    }
}
```

Reduce 4



Reduce 4

Reduce 1	4.7008 ms
Reduce 2	4.7484 ms
Reduce 3	4.7218 ms
Reduce 4	4.6226 ms

- Warp shuffle 연산 사용
- Warp 아래 단위로는 `__syncthreads` 사용 안함

Reduce 5

Warp Reduce

```
__inline__ __device__ int warp_reduce(int val) {
    for (int offset = warpSize / 2; offset > 0; offset /= 2) {
        val += __shfl_down(val, offset);
    }
    return val;
}
```

Reduce 5

Block Reduce

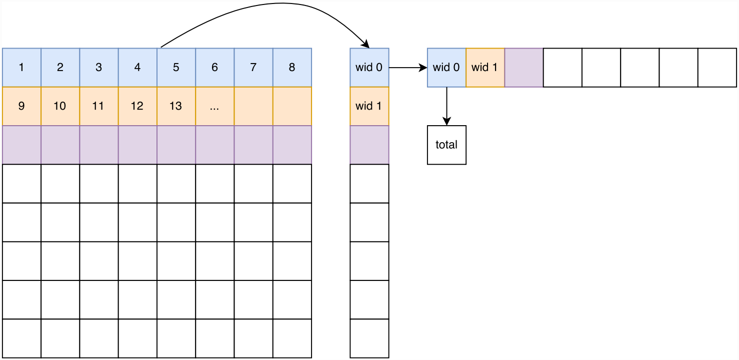
```
__inline__ __device__ int block_reduce(int val) {
    static __shared__ int shared[32];
    int lane = threadIdx.x % warpSize;
    int wid = threadIdx.x / warpSize;

    val = warp_reduce(val);

    if (lane == 0) {
        shared[wid] = val;
    }

    __syncthreads();

    val = (threadIdx.x < blockDim.x / warpSize) ? shared[lane] : 0;

    if (wid == 0) {
        val = warp_reduce(val);
    }

    return val;
}
```

Reduce 5

Inline Warp Shuffle Reduce

```
__global__ void reduce5(int *idata, int *odata, unsigned int n) {
    int result = 0;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    while (i < n) {
        result += idata[i];
        i += blockDim.x * gridDim.x;
    }

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

Reduce 5



Reduce 5

Reduce 1	4.7008 ms
Reduce 2	4.7484 ms
Reduce 3	4.7218 ms
Reduce 4	4.6226 ms
Reduce 5	4.6883 ms

- 모듈화된 구조

Reduce 6

Atomic Warp Reduce

```
__global__ void reduce6(int *idata, int *odata, unsigned int n) {
    int result = 0;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    while (i < n) {
        result += idata[i];
        i += blockDim.x * gridDim.x;
    }

    result = warp_reduce(result);

    if (threadIdx.x & (warpSize - 1) == 0) {
        atomicAdd(odata, result);
    }
}
```

Reduce 6

Reduce 1	4.7008 ms
Reduce 2	4.7484 ms
Reduce 3	4.7218 ms
Reduce 4	4.6226 ms
Reduce 5	4.6883 ms
Reduce 6	0.5460 ms

- Atomic 연산을 사용한 warp 단위 reduction
- Shared memory 안씀
- `__syncthreads` 안씀
- Kernel launch 1회
- Contention이 일어나면 성능이 심하게 떨어질 가능성도 있음

Reduce 7

Atomic Block Reduce

```
__global__ void reduce7(int *idata, int *odata, unsigned int n) {
    int result = 0;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    while (i < n) {
        result += idata[i];
        i += blockDim.x * gridDim.x;
    }

    result = block_reduce(result);
    if (threadIdx.x == 0) {
        atomicAdd(odata, result);
    }
}
```

Reduce 7

Reduce 1	4.7008 ms
Reduce 2	4.7484 ms
Reduce 3	4.7218 ms
Reduce 4	4.6226 ms
Reduce 5	4.6883 ms
Reduce 6	0.5460 ms
Reduce 7	4.6843 ms

- Atomic 연산을 사용한 block 단위 reduction
- Kernel launch 1회
- Warp 단위로 atomic 연산을 쓸 때 contention이 심하다면 사용할 수 있는 전략

References

- CUDA C Best Practices Guide, NVidia
- CUDA C Programming Guide, NVidia
- Optimizing Parallel Reduction in CUDA, Harris, 2008
- Faster Parallel Reductions on Kepler, Luitjens, 2014
  - `https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/`