CSED490C Project Final Report

20180085 송수민

I. 문제 제시 및 선정 동기

이번 프로젝트에서는 Project Ideas: 2. CUDA versions for Problem-Based Benchmark Suite (PBBS)를 주제로 진행하고자 한다. 세부 주제로는 선정한 것은 reduceDuplicate이다. 해당 algorithm은 unsigned integer를 담고 있는 array에 중복된 원소를 모두 제거하여 unique element만 남게 만드는 것이다. 예를 들어, [1,2,3,3,4,5,2,6] 이라는 array가 있다고 가정하면, output은 [1,2,3,4,5,6]이 되어야 한다. 해당 알고리즘 설명에 따르면 large size input으로는 1억개의 unsigned integer, small size input으로는 1000만개의 unsigned integer 상황을 두고 있다고 언급한다.

가장 보편화 된 방법으로는 array를 정렬 한 후, 이전 element와 비교하여 같으면 output에 넣지 않고, 같지 않다면 output에 포함하는 방법이 있다. 그렇다면 전체적인 process가 sorting - reduceDuplicates가 될 것이며, sorting 또한 GPU로 성능을 높이고자 하는 항목 중 하나로 overhead가 작지 않을 것으로 예상하여 sorting 없이 시도하고자 한다.

해당 문제를 선정하게 된 동기는 주제에 대해 정보를 찾던 중 한 포럼에서 본 질의응답에서 시작하였다. NVIDIA 포럼에서 본 주제와 같은 문제를 다루기 위해 질문이 게시되었고, 가장 먼저 답변을 단 사용자가 "왜 해당 주제를 GPU를 사용하려고 하느냐, 해당 문제는 CPU가 GPU보다 훨씬 빠르게 처리할 수 있다."라는 맥락으로 답하였다. 이러한 답변을 보고 무의식적으로 당연하다고 생각했던 것들에 대해 다시 생각하게 되었다. 병렬 컴퓨팅 수업을 수강하며 당연하게 CPU보다 GPU로 task를 처리하는 것이 훨씬 빠르고 효율적이라고 생각하고 있었는데, 이 점을 직접 확인함과 동시에 답변자의 답변을 반박하고 싶었다. 이러한 동기로 해당 주제를 선정하게 되었다.

II. 선행 연구 요약

현재 PBBS에서 benchmark로 제시하고 있는 코드[1]는 간단하다. Parlay라는 parallel computing library를 사용하여 sorting 후 중복 원소를 제거한다. 해당 코드는 아래와 같다.

```
gpuTKTime_start(Compute, "Process");
std::sort(v.begin(),v.end());
v.erase(std::unique(v.begin(),v.end()),v.end());
gpuTKTime_stop(Compute, "Process");
```

기존 benchmark는 GPU를 baseline으로 설정하였으나 본 프로젝트에서는 위 코드를 CPU에서 작동시킨 결과를 baseline으로 설정하고 프로젝트를 진행한다. 위 코드를 분석하면, v라는 vector가 parlay library에서 유래한 data container이다. 이는 CUDA에 compatible한 Thrust library와 유사한 개념을 가지고 있으므로 이를 GPU 버전 baseline으로 또한 설정하여 추후에 비교하고자 한다.

[1] https://github.com/cmuparlay/pbbsbench/tree/master/benchmarks/removeDuplicates/serial_sort

III. 구현 방법

본 프로젝트에서 중요하게 생각한 부분은 두가지이다. 첫번째로 원소의 존재 유무이다. 위 주제에서 최종적으로 요구하는 것은 container에 담겨 있는 원소의 종류이다. 이를 다르게 말하면 해당 원소가 존재하는지에 대한 여부를 알고 싶은 것이다. 원소가 몇 번 중복되었는지는 중요하지 않다고 생각하였고, 해당 원소가 있는지, 존재성이 가장 중요하다고 생각하였다. 이점을 고려한다면 parallelization을 활용 할 수 있다. 존재 여부는 True/False로 표현 할 수 있기 때문에, 동일한 원소가 존재한다고 하여도 결국 존재 여부는 True이다. 이를 담는 container를 선언하면 해당 container에 존재여부를 모두 담을 수 있다. 이를 통해 좋은 병렬화를 구현할 수 있는데, True에다 True를 또 assign한다 해서 값이 변하지는 않기 때문에 race condition로부터 자유롭다. 이전 정보 값으로부터 자유롭다는 뜻이다.

하지만, 그러기 위해서는 0 ~ UINT_MAX에 해당하는 정보를 모두 각각 담을 수 있는 크기의 container가 필요한데, 이는 memory를 너무 많이 사용한다는 단점이 있다. UINT_MAX 만큼의 char 배열을 잡는다 하여도 약 4.5GB가 필요하다. 이에 따라 두번째로 중요하게 생각한 것은 메모리 사용량의 절약이다. 이를 해결하기 위해 OS File system에서 사용하는 bitmap 개념을 도입하여 메모리를 절약하고자 하였다. Char 배열의 원소 한 칸은 1byte로 bit로 변환하면 8bit이다. 이는 8가지의 정보를 담을 수 있다는 뜻으로도 해석이 가능하다. 즉, char 배열 원소 한 칸에 8개의 숫자 존재 여부를 담을 수 있다. 결과적으로 UINT_MAX / 8만큼 메모리를 사용하여 기존 구현방법보다 메모리 사용량을 절약 할 수 있다. 하지만, 8개의 숫자를 같은 원소 한 칸에 담는 것이기 때문에 synchronization 문제가 발생한다. 이를 해결해주어야 correctness가 해결 될 것이다.

본 프로젝트의 구현은 4가지로 나누어 구성하였다. 구성은 아래와 같다.

1. CPU Naïve Implementation

II에서 언급한 구현이다. 이를 기본 benchmark로 설정하며 코드는 1번 주석에서 가져와 사용하였다.

2. 1-Lock + Map Implementation

GPU 메모리를 사용하면 4.5GB의 공간을 설정하는 것이 불가능한 것은 아니다. 하지만, 다른 작업들과 같이 활용한다고 하였을 때 어떠한 시나리오가 구성될지는 모르는 일이다. 만약 환경이 가능하다고 가정하고, 3번에서 구현할 항목이 메모리 사용량은 절약되었지만 성능이 나빠지는지를 비교하기 위해 구현한다.

3. Q-Lock + Q-Bitmap Implementation

UINT_MAX / 8 만큼의 char 배열을 할당하여 배열 한 칸마다 8개의 숫자 존재여부를 표현한다. 만약, [1,2,3,4,7,10] 이라는 배열이 존재한다면 char 배열 0번째 칸에는 [10011110], 1번째 칸에는 [00000100]이 채워지게 된다. 2번 구현 방법과 다르게 이는 동기화 문제가 발생한다. 만약, 4와 7을 처리할 때 동시에 진입하게 된다면 최종 assign되는 항목에 따라 4만 반영, 7만 반영이 될 수도 있는 unexpected behavior가 발생할 수 있다. 이를 해결하기 위해 가장 구현이 간단한 방향인 char 배열 한 칸마다 lock을 부여하였다. 즉, char 배열 한 칸을 수정하기 위해서는 lock을 얻어 critical section으로 진입하여야 값을 변경할 수 있다.

4. Check after Sorting with CUDA Thrust Library

II에서 언급한 GPU 버전의 baseline 구현이다. Thrust vector을 선언하여 sorting 후, 각 thread에

서 next thread가 다루고 있는 값과 비교하여 중복 여부를 판단한다.

2, 3, 4의 자세한 구현은 아래에 코드와 함께 설명한다.

2. 1-Lock + Map Implementation

2의 주요 부분 코드는 아래와 같다.

```
__global__ void markBitmap(unsigned int* input, char* d_bitmap, int N, int*
d_zero_flag){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if(tid < N){
        unsigned int value = input[tid];
        if(value == 0){
            d_zero_flag[0] = 1;
            return;
        }
        d_bitmap[value] = 1;
    }
}
```

각 thread는 global memory에 있는 input 원소 하나를 처리한다. 즉, N개의 원소가 들어온다면 GPU에서는 N개의 thread가 활성화되어 작업을 실시한다. 각자 thread ID로 배열에 접근하고 해당 value를 저장하여 이를 map indexing으로 사용한다. 만약, thread ID가 4라면, value는 input[4]가 되며, map[value]에 1을 assign하여 value가 존재한다는 것을 표시한다.

N개의 원소가 input으로 제시된다면 output은 최대 N개로 구성될 수 있다. 즉, 중복되는 원소들이 존재하여 N개보다 작을 수도 있다. 이를 효율적으로 처리하기 위해 추후 과정에서 0인 값들은 모두 pass한다. 하지만, unsigned integer는 0부터 나타낼 수 있기 때문에 0의 존재성을 표현하기 위한 별도의 변수를 하나 두어 이를 host로 추후에 복사한다. 만약 thread가 0을 처리하게 된다면 zero_flag에 1로 ON시키고 return하여 추가적인 global memory 접근을 막아 시간을 절약한다.

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int idx = index[0] + tid;
    unsigned int value;
    if(idx <= UINT_MAX){
        if(d_bitmap[idx] == 1){
            value = idx;
        }
        else
            return;
    }
    else
        return;
    bool blocked = true;
    while(blocked){
        if(0 == atomicCAS(&mutex, 0, 1)){
            output[count] = value;
            count++;
            atomicExch(&mutex, 0);
            blocked = false;
        }
    }
}
```

이후, 결과를 merge하는 단계이다. 각 thread는 map 원소 한개를 보아 해당 index가 ON인지를 파악한다. ON이라면 lock을 얻어 critical section으로 진행하여 output에 append하고 count를 증

가 시켜준다. 2번째 줄의 idx 설정 방법은 UINT_MAX만큼의 thread를 만들 수 있어야 하는데, 최대 Grid가 이를 한번에 커버하지 못해 Block size를 1024로하고 Grid 사이즈를 Block * Block으로 하여 4번 loop을 돌아 모두 커버하게 하였다. 이때, thread ID는 0부터 시작할 것이기에 그에 따른 값을 parameter로 넘겨주어 조정하였다. Output을 merge하는 단계에서 busy-waiting으로 구현하였고 이에 대한 synchronization overhead가 클 것으로 예상하였다. 이에 대한 내용은 추후 evaluation 파트에서 설명하고자 한다.

3. Q-Lock + Q-Bitmap Implementation

3의 주요 부분 코드는 아래와 같다.

```
__global__ void markBitmap(unsigned* d_array, unsigned int* locks, char*
d_bitmap, int N, int* d_zero_flag) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N) {
        unsigned int val = d_array[tid];
        if (val == 0){
            d_zero_flag[0] = 1;
            return;
        }
        int idx = val / 8;
        int offset = val % 8;
        int mask = 1 << offset;
        bool blocked = true;
        while(blocked){
            if(0 == atomicCAS(&(locks[idx]), 0u,1u)){
                d_bitmap[idx] |= mask;
                atomicExch(&(locks[idx]),0u);
                blocked = false;
```

위에서 언급한대로 bitmap의 경우 UINT_MAX / 8만큼 할당하였고, 숫자에 따른 접근은 8로 나눈 몫과 나머지를 활용하여 정보를 채워넣는다. 이때, 해당 index에 해당하는 bitmap에 대한 lock을 얻어 critical section으로 진입한 뒤, or operation을 통해 값을 최신화 한다. 0에 대한 처리는 2번 구현과 동일하게 하였다.

```
for (int i = 0; i < 8; i++) {
            if ((val & (1 << i)) >> i == 1) {
                arr[cnt] = tid * 8 + i;
                cnt++;
            }
        }
    }
    if(cnt == 0){
        return;
    }
    bool blocked = true;
    while(blocked){
        if(0 == atomicCAS(&mutex, 0,1)){
            for(int i = 0; i<cnt;i++){
                result[pos[0] + i] = arr[i];
            }
            pos[0] += cnt;
            atomicExch(&mutex,0);
            blocked = false;
```

3 번 구현방법에서는 한 개의 thread 가 최대 8 개의 숫자를 output 에 append 할 수 있으므로, 미리 thread 에서 처리할 숫자의 개수를 센 후에 critical section 으로 진입하여 append 하였다. 전체 개수는 global variable 로 선언하여 critical section 안에서만 operation 을 진행 해 전체 원소 접근에 문제가 없게 하였다.

4. Check after Sorting with CUDA Thrust Library

4 의 주요 부분 코드는 아래와 같다.

```
__global__ void reduceDuplicates(unsigned int* input, int N, int* zero_flag){
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int value = input[tid];
    bool isEqual = false;
    if (tid<N-1){
        if(value == 0){
            zero_flag[0] = 1;
        }
        if (value == input[tid + 1]){
            isEqual = true;
        }
        else{
            isEqual = false;
        }
        __syncthreads();
        if (isEqual){
            input[tid] = 0;
```

각 thread 는 왼쪽에서 오른쪽을 바라보는 시선을 두는 것처럼 비교를 한다. 즉, sorting 된 input 이 GPU 로 넘어오므로 각 thread 에서 옆의 value 만 본다면 중복 여부를 판단할 수 있다. 만약 값이 같다면 local variable 인 isEqual 에 True 를 assign 하여 추후 flag 로 사용한다. 이후, 모든 thread 들이 syncthread()지점까지 오기를 기다렸다가, 모두 도착하였다면 동시에 input 에 중복 된 자리에 0 을 assign 한다. 0 에 대한 처리는 위의 방법과 같다. 이후, 0 인 자리는 pass 하고 0 이 아닌 자리만 다루도록 구현하였다.

Sorting 부분은 library 를 사용하여 간단하며 아래와 같다.

```
gpuTKTime_start(GPU, "Sorting");
thrust::sort(thrust::device, d_input, d_input + N);
gpuTKTime_stop(GPU, "Sorting");
```

IV. 평가

평가 방법은 CSED490C 과제에서 사용하였던 libGPUtk 를 사용하여 millisecond 를 측정하였다. Baseline 인 CPU version 은 1st phase, 2nd phase 나누는 것 없이 통합으로 측정하였으며, 2 번째, 3 번째 구현은 map 또는 bitmap marking 을 1st phase, 결과를 merge 하는 과정을 2nd phase 로 설정하였다. 4 번째 구현은 sorting 을 1stphase, 2nd phase 는 각 thread 에서 next thread 를 비교하는 과정으로 설정하였다.

평가 환경은 POSTECH 교육 클러스터 서버를 사용하였으며, 사용 GPU 는 NVIDIA Titan Xp 24GB 환경을 사용하였다.

Input dataset generation 은 코드상에서 N 값을 변화 시켜 N 개의 unsigned integer 를 random 으로 generating 시켜 사용하였다. 해당 코드는 아래와 같다.

```
unsigned int* generateRandomUint(unsigned int* arr, int N) {
    for (int i = 0; i < N; i++) {
        arr[i] = rand() % N;
    }
    return arr;
}
```

위 코드는 Host 에서 실행되어 측정 시간에는 포함하지 않았다.
결과는 아래 표와 같다.

1. Input Size N=**10000000**

	CPU Naïve Baseline	**2nd Impl.**	**3rd Impl.**	**4th Impl**
1st phase		10.3945	35.2608	6.18422
2nd phase		286240	177692	2.48653
Total (ms)	3753.38	286250.394	177727.261	8.67075

2. Input Size = **100000000**

	CPU Naïve Baseline	**2nd Impl.**	**3rd Impl.**	**4th Impl**
1st phase		11.4322	34.1231	51.1323
2nd phase		490321	323521	4.26502
Total (ms)	40296.2	490332.4322	323555.1231	55.39732

본 프로젝트의 주된 주제인 2 번째, 3 번째 구현의 결과를 보면 N 의 size 가 커질 수록 1st 의 결과를 보았을 때, CUDA Thrust library 를 사용할 때보다 결과가 좋아 충분히 경쟁력이 있다고 볼 수 있다. 하지만, 결과를 취합하는 과정에서 synchronization overhead 가 너무 크게 발생해 이점을 못 얻고 있다.

CPU 와 GPU 성능을 비교하였을 때 1 번과 4 번의 비교를 통해 GPU 의 활용이 가능하다면 훨씬 성능이 좋음을 볼 수 있다. 이는 주제 선택 동기였던 사항을 재확인 할 수 있게된 결과라 할 수 있다.

현재 주된 문제점인 synchronization 문제가 2 번째, 3 번째 구현에서도 문제가 발생함을 알 수 있다. 2 번째, 3 번째 구현의 차이는 map, bitmap 에 접근할 때 lock 을 얻어야 하느냐 아니냐의 차이인데 이 차이로 인해 input 의 크기에 상관없이 약 20ms 정도 손해를 얻고 있다. Input 의 크기를 더 증가시켰을 때, 메모리 사용량과 20ms 의 trade off 가 타당한 정도인지 알아볼 필요가 있다.

개선 사항으로 2nd phase 의 성능 향상이 있다. CUDA Thrust library 의 성능이 좋은 이유가 vector 를 다룰 때의 parallelization 이 잘 이루어지기 때문이라는 사실이 있다. 이 점을 활용하여 개선할 수 있을 것이라고 생각한다.

V. Appendix

Code 는 아래의 Github 주소[2]에서 clone 하여 사용가능하다. Source 디렉토리에서 원하는 구현을 make 하고, 생성되는 실행파일을 터미널에서 실행시키면 결과를 알 수 있다.

1 번 구현: make naive

2 번 구현: make no_lock

3 번 구현: make bitmap

4 번 구현: make sort_reduce

[2] https://github.com/songsm921/CSED490C-Project