**Parallel/Distributed Computing (CSEG414)**

**Assignment #3**

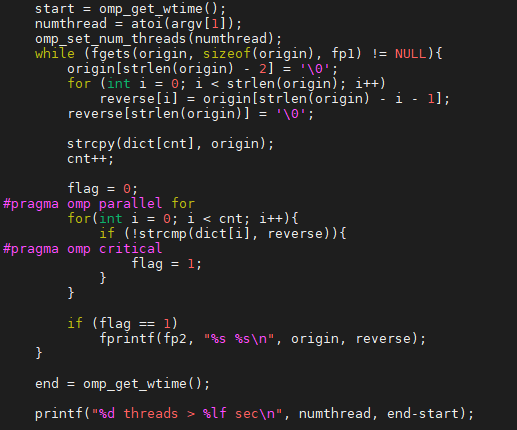
컴퓨터공학과

20171640

박수진

**1. OpenMP Programming**

1) Implementation



- Error 처리 및 File open, close는 execution time 측정에서 제외하였다.

- File에서 한 줄씩 읽어오는 부분을 parallel하게 작성하면 서로 다른 thread가 동시에 같은 줄을 읽을 수도 있으므로 critical section으로 처리하거나 하나의 thread가 처리하도록 구현해야 한다. 그러나 critical section으로 처리하면 overhead가 커질 수 있으므로 file을 읽고 해당 word의 reverse를 구하는 부분은 serial하게 구현하였다.

- File에서 읽은 word는 dict라는 array에 저장하였고, 해당 word의 reverse가 dict array에 존재하면 그 word는 palindrome이다.

i. 양 방향에서 읽어도 같은 단어일 경우, 그 단어의 reverse는 원래 단어와 같고 원래 단어가 dict에 존재하므로 palindrome

ii. 단어의 reverse가 input 파일에 존재하는 경우, word A의 reverse가 word B이며 word A가 먼저 등장한다고 가정하면 word A를 읽어왔을 때 해당 단어의 reverse는 dict에 없지만 word B를 읽어왔을 때 word B의 reverse인 word A가 이미 dict에 존재하므로 두 단어는 palindrome

- 위에서 dict를 탐색하는 부분은 #pragma omp parallel for(**for-loop parallelism**)을 사용하였다.

- dict에서 단어의 reverse를 발견한 경우 flag를 1로 set하는데, flag는 shared 변수이므로 #pragma omp critical을 사용하였다.

- flag가 1일 경우 해당 word는 palindrome이므로 output 파일에 기록한다.

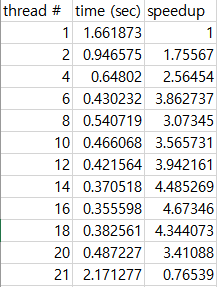
- 출력 양식은 word A의 reverse가 word B라고 할 때,

(word A) (word B)

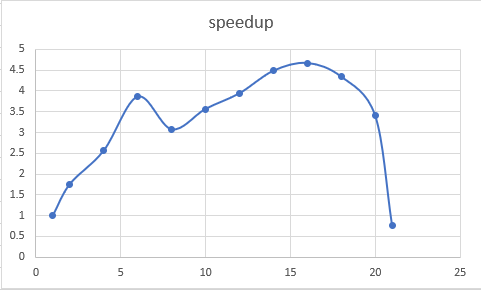
로 하였으며 (word B) (word A) 와 같이 중복되는 경우는 출력하지 않는다.

2) Analysis

<Chart>



<Graph> (가로축 # of threads, 세로축 speed up)



- serial program은 thread 개수가 1인 경우로 하여 측정하였다.

- 실행 시 오차로 인해 그래프가 균일하게 증가하지 않았다.

- thread 개수가 2일 때는 speed up이 1, thread 개수가 4일 때는 speed up이 2, thread 개수가 6~12일 때는 speed up이 3, thread 개수가 14~18일 때는 speed up이 4이다. (speed up 소수점 아래 버림)

- thread 개수가 1 ~ 6일 때는 thread 개수가 증가하는 만큼 speed up이 급격히 증가한다.

- thread 개수가 6 ~ 20일 때는 speedup에 큰 변화가 없으며, 16일 때 speed up이 4.67로 가장 크다.

- 그러나 thread 개수가 21을 초과하면 overhead가 급격히 커져서 exec time이 커지고 speedup이 급격히 떨어진다. => parallel보다 serial program이 효과적이다.

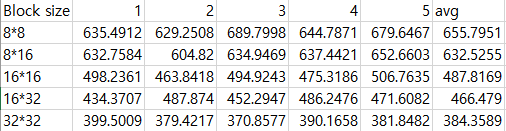
- 따라서, input file이 words.txt인 경우에 thread 개수를 14 ~ 18로 설정하면 이상적이다. (speed up > 4)

**2. CUDA Programming 1**

**(1) Compare the performance of CUDA implementations with different block/thread sizes**

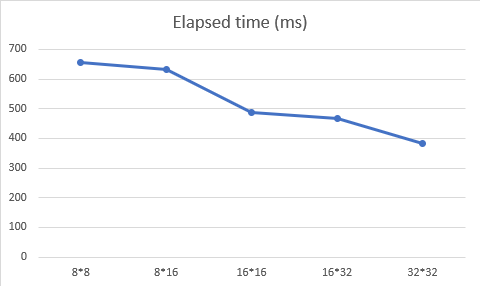
**and discuss your results.**

-> matrixMul\_1.cu



정확도를 위해 5회 측정한 후 elapsed time의 average를 기준으로 하였다.

block은 2 dim, grid도 2 dim이며 block size를 BLOCK\_W \* BLOCK\_H라고 할 때, grid의 dimension은 (N/BLOCK\_W) \* (N/BLOCK\_H)이다.(N = 4096) grid는 block size에 dependent하므로 block size를 warp size(32)의 배수로 바꿔가며 측정하였다. block size는 8\*8부터 32\*32까지 바꿔가며 측정하였는데, block size를 32\*64로 올렸을 때는 수행 시간이 71ms로 급격하게 줄어서 출력 값을 확인해보니 모든 값이 0으로 matrix multiplication이 제대로 수행되지 않았다. 따라서 해당 프로그램은 최대 block size를 32\*32로 하여 실험하였다. 행렬 곱셈 A \* B = C 일 때 CPU에 할당된 matrix A, B, C의 크기는 4096\*4096으로 크기가 커서 전역변수로 선언하여 data 영역에 할당하였다.

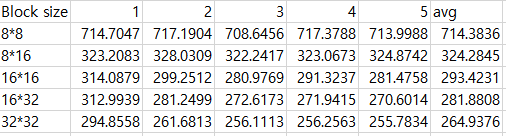


block size가 증가할수록 execution time이 감소하여 성능면에서 좋아지는 것을 확인할 수 있다. block size가 32\*32 thread일 때 가장 좋은 성능을 보였으며 thread 개수가 8\*16에서 16\*16으로 증가할 때 성능면에서 가장 급격한 변화를 보였다. 위 결과를 통해 (1) 프로그램은 병렬화가 잘 이루어졌음을 알 수 있다.

**(2) Modify your program so that you use shared memory instead of global memory.**

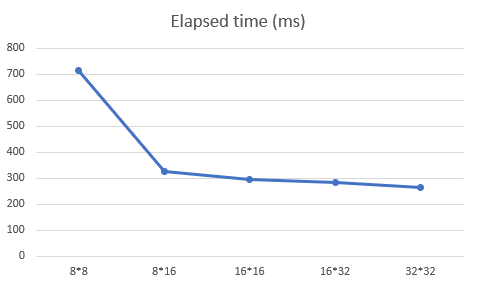
**Discuss your results in terms of memory type.**

-> matrixMul\_2.cu



측정 방법은 (1)과 같다.

shared memory의 size는 block의 개수가 많아지면 한 block에 배정되는 shared memory의 크기가 한정적이므로, SM의 resource를 고려하여 (BLOCK\_W\*BLOCK\_H\*4)으로 하였다.

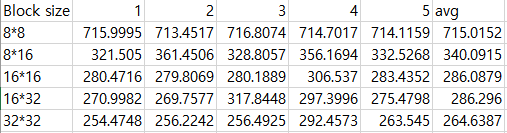


(1)과 같이 block size가 증가함에 따라 elapsed time은 감소하는 감소 그래프를 그림을 확인할 수 있다. 그러나 thread 개수가 8\*8일 때 수행시간이 매우 크게 나타나는데, 이 때 block size가 작기 때문에 shared memory의 size도 작게 책정된다. 또한 shared memory의 size가 작기 때문에 shared memory에 들어갈 수 있는 data의 수가 한정적이고, 따라서 여러 번 global memory에 access 해야 한다. 이 과정에서 발생하는 overhead로 인해 block size 8\*8에서 수행 시간이 오래 걸리는 것으로 보인다. global memory와 shared memory를 사용할 때 수행시간 차이는 (3)에서 설명하도록 하겠다.

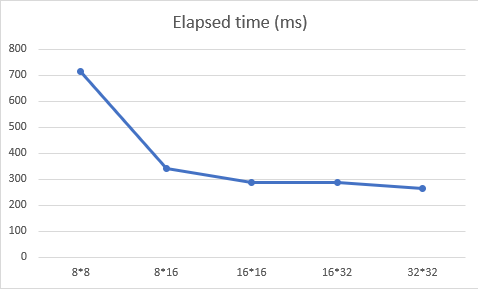
**(3) Try to further optimize your program using any techniques learned in the class and**

**discuss your results.**

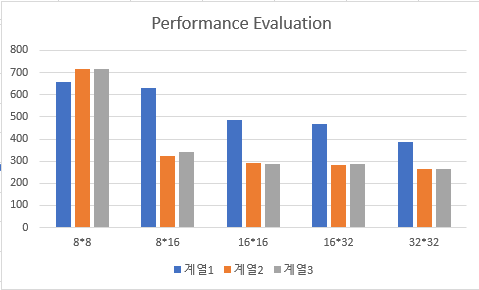
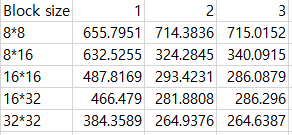
-> matrixMul\_3.cu



측정 방법은 (1)과 같다.



stream을 이용하여 asynchronous하도록 optimization하였다. 행렬의 곱셈을 A\*B라고 할 때 A는 row로 곱하므로 stream으로 row의 배수만큼씩 data를 잘라서 보내도록 구현하였고 B는 col을 곱하기 때문에 stream으로 따로 잘라서 보내지 않고 한번에 cudaMemcpy하였다. kernel을 호출한 후 C에서 stream으로 다시 result를 보낼 수 있도록 구현하였다. 그러나 shared memory를 사용할 때는 block-wise multiplication을 사용하므로 stream의 개수를 증가시키면 오히려 프로그램의 수행시간이 증가하는 것을 확인할 수 있었다. 이러한 이유로 stream의 개수를 1로 고정하였는데, stream이 1이면 data를 잘라 보내는 과정이 없으므로 본질적으로 보면 (2)와 수행시간이 비슷하게 나올 수 밖에 없다.



위 결과를 종합하여 (1), (2), (3)의 cuda program의 block size별 수행시간을 비교해보았다. block size가 작을 때는 (2), (3)에서는 global memory에 접근해서 data를 가져온 후 다시 shared memory에서 data를 접근하는 overhead가 커서 global memory를 사용할 때가 성능이 더 좋게 나타난다. 그러나 그 외의 block size에서는 shared memory에 access하는 시간이 global memory에 access하는 시간보다 훨씬 작기 때문에 (2), (3)이 (1)보다 좋은 성능을 보인다. 또한 shared memory를 사용할 때는 partial sum을 이용한 block-wise matrix multiplication을 하기 때문에 연산 과정 자체도 훨씬 빠르다. 결론적으로 4096x4096 matrix multiplication을 할 때는 shared memory를 사용하여 block-wise matrix multiplication algorithm을 활용하는 것이 좋으며 block size (thread 개수)는 32\*32가 가장 이상적이다.

**3. CUDA Programming 2**

1) Implementation

(a) Write a sequential version in C.

-> reduction\_a.c

(b) Write a CUDA version that does not take path divergence into account.

-> reduction\_b.cu

(c) Update the version in (b) to take path divergence into account.

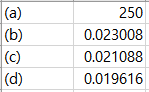
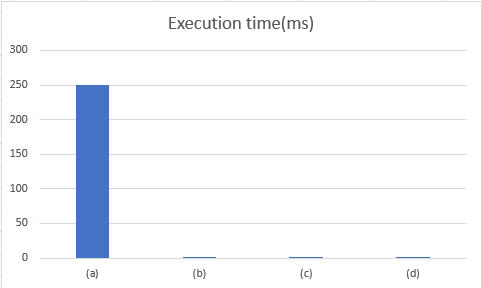
-> reduction\_c.cu

(d) Optimize the version in (c) by using different block/thread sizes, shared memory instead of global memory, and other memory optimizations techniques.

-> reduction\_d.cu

2) Analysis

(1) Compare the performance of each version and discuss your results and findings.

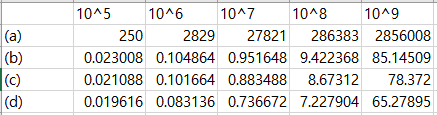
(Array size = 10000, 단위 = ms)

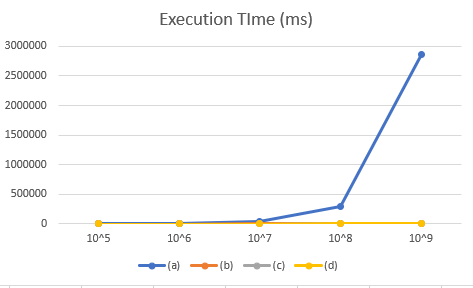
(a)는 sequential program으로 for문으로 array 전체를 탐색하면서 max 값을 찾는다. 이 CPU program은 (b), (c), (d)에 비해 현저히 오래 걸린다. (b)는 reduction max를 구하는 일반적인 알고리즘이다. block size는 128이며 global memory를 사용하였다. 또한 path divergence 문제가 발생할 수 있다. (c)의 경우에는 (b)에서 반복문 내의 조건문을

if ((id / **WARP\_SIZE**) % (2\*i) == 0)

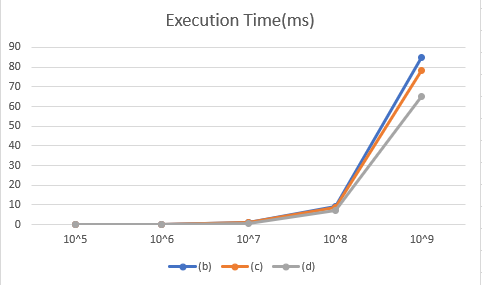
으로 수정하여 path divergence 현상을 해결하였다. 여기서 WARP\_SIZE는 32로 설정하였다. path divergence 문제가 해결되어 (b)보다 빠른 처리 속도를 보이는 것을 확인할 수 있다. (d)는 실험적으로 가정 빠른 block size인 32로 설정하였고 shared memory로 접근하도록 수정하였다. 이러한 optimization을 통해 모든 프로그램 중 가장 좋은 성능을 보이는 것을 실험적으로 확인하였다.

(2) Also repeat the same comparison with different array sizes. Did you get the same results as (1) ?





위 그래프를 보면 array size가 커질수록 CPU program의 실행 시간은 GPU program에 비해 매우 급격하게 커지는 것을 확인할 수 있다. 따라서 array size가 클 경우에는 GPU program이 매우 유용하다. (1)에서의 결과와 같이 (a)의 실행 시간은 (b), (c), (d)에 비해 현저히 크다. GPU program (b), (c), (d)만 다시 살펴보면 다음과 같다.



array size가 커져도 결과는 (1)과 같이 (b), (c), (d) 순으로 좋은 성능을 보이는 것을 알 수 있다. 이는 각각의 optimization이 잘 이루어졌음을 의미한다. 또한 array size가 커질수록 execution time의 차이가 커지므로 data size가 큰 problem의 경우에 (a)와 같은 CPU program은 굉장히 성능이 떨어지며 GPU program도 optimization이 반드시 필요하다.