병렬분산컴퓨팅 (CSE5414)

Fall 2020

Assignment #3

이름: 김인호

학번: 20161577

담당 교수: 박성용 교수님

20161577
김인호

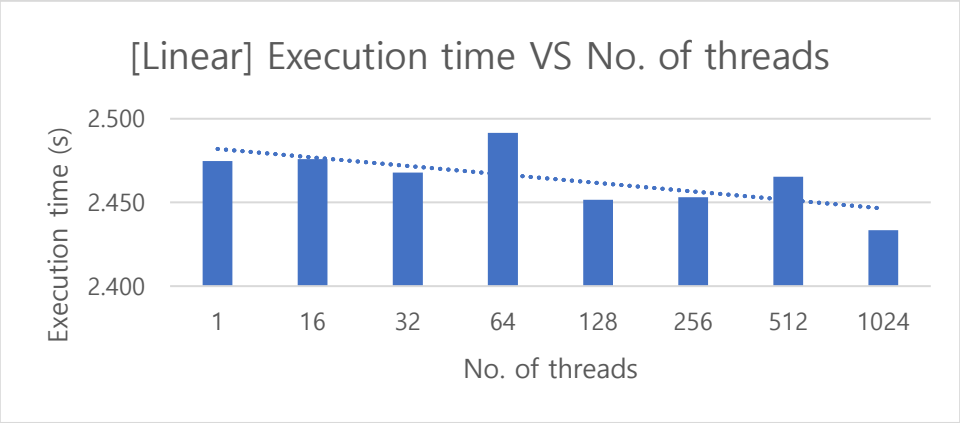
병렬분산컴퓨팅 (CSE5414)

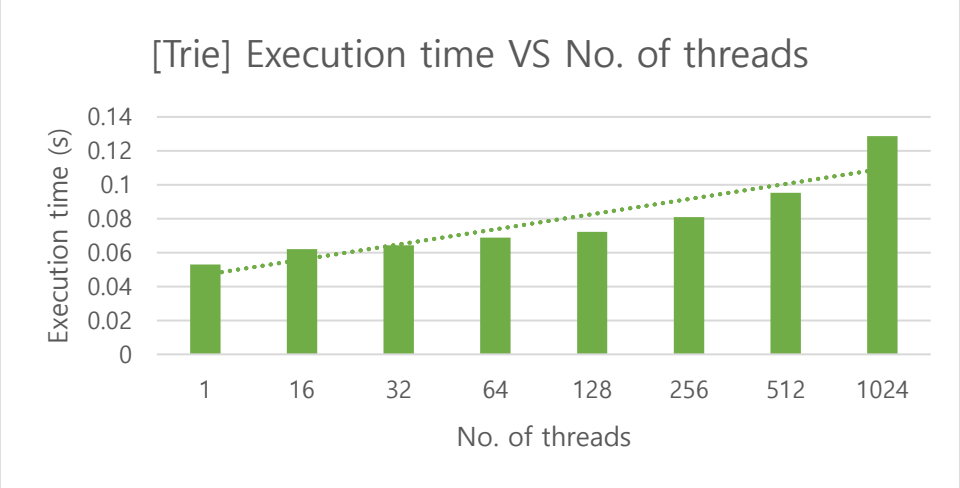
Assignment #3

1. OpenMP Programming

사전의 각 단어가 palindromic 인지의 여부를 알려면 해당 단어를 거꾸로 한 단어도 사전에 등장하는지 확인하는 것으로 충분하다. 이 때 어떤 단어가 사전에 있는지 탐색하는 부분을 linear 탐색과 trie 자료구조를 사용한 탐색으로 구현하였다:

		Number of threads							
		1	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
Time (s)	**Linear**	2.475	2.476	2.468	2.492	2.452	2.453	2.465	2.433
	Trie	0.053	0.062	0.064	0.069	0.072	0.081	0.095	0.129





우선 linear 탐색 구현의 결과를 전체적으로 보면 thread 개수가 증가함에 따라 실행 시간이 줄어드는 것으로 볼 수 있다. 탐색을 병렬화 함으로 인해 꽤 오래 걸리는 탐색 시간을 줄이는 것이 핵심이다. 1 개의 thread 로 실행한 결과 대비 1024 개의 thread 로 실행했을 때의 결과에 대한 speedup 을 계산하면 아래와 같다.

$$Speedup = \frac{T_{serial}}{T_{parallel}} = \frac{2.475}{2.433} = 1.017$$

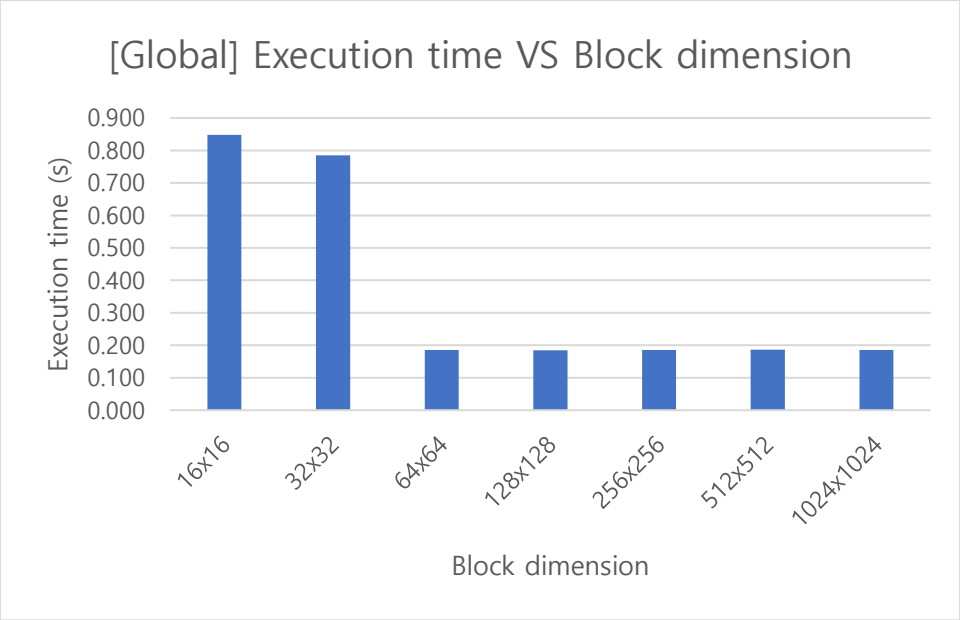
매우 작은 speedup 이지만 이것은 병렬화에 의해 생기는 overhead 에 비해 주어진 사전의 크기(25,143 개의 단어)가 너무 적기 때문이다. 이는 trie 자료구조를 사용한 탐색의 구현의 결과를 통해 더 잘 알 수 있다. 단어의 개수를 N, 단어의 평균 길이를 L 이라 했을 때 trie 자료구조는 삽입이 O(N*L), 탐색이 O(N*L)이다. 이를 사용했을 때의 결과를 보면 thread 개수가 증가함에 따라 실행 시간도 꾸준히 증가하는 것을 알 수 있다. 즉 주어진 사전의 크기로 이 문제를 해결할 때 병렬화로 인한 이득을 얻으려면 고의적으로 탐색 알고리즘을 느린 알고리즘으로 써야 하는 상황인 것이다. 따라서 사전의 단어 개수가 증가함에 따라 병렬화로 얻을 수 있는 speedup 또한 증가할 것으로 판단할 수 있다.

2. CUDA Programming 1

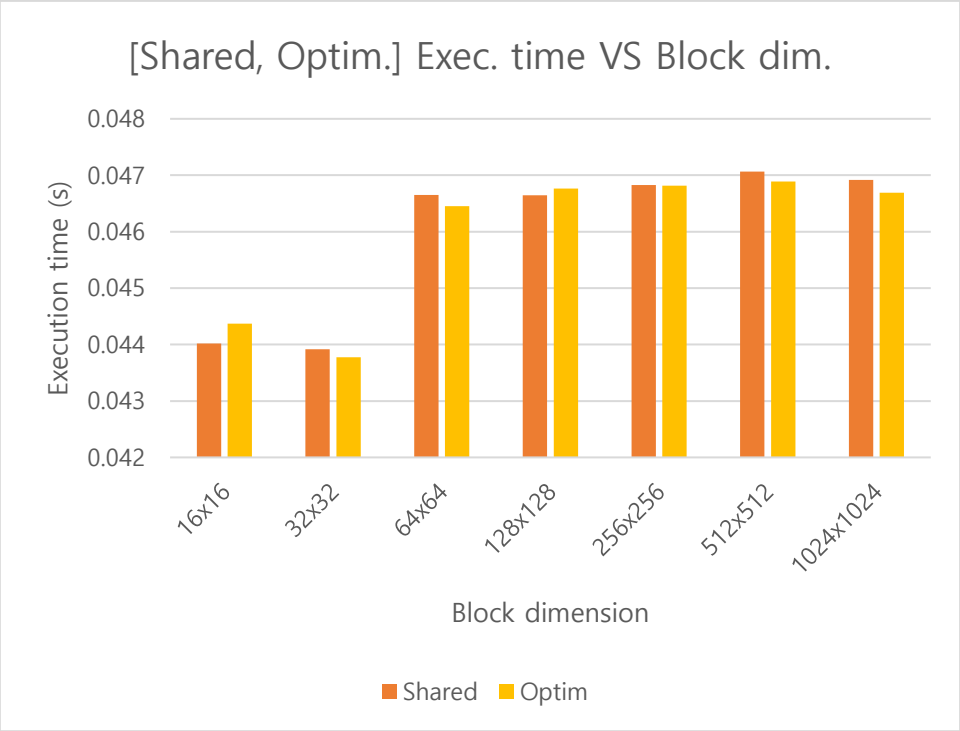
아래는 4096 x 4096 크기의 행렬 곱셈을 3가지 구현 방식으로 해결한 결과이다.

		Block dimension						
		16x16	**32x32**	**64x64**	**128x128**	**256x256**	**512x512**	**1024x1024**
Time (s)	**Global**	0.848	0.785	0.185	0.184	0.185	0.186	0.185
	Shared	0.044	0.044	0.047	0.047	0.047	0.047	0.047
	Optim.	0.044	0.044	0.046	0.047	0.047	0.047	0.047

(1) 먼저 global memory를 사용하면서 여러 block dimension으로 실행을 한 결과이다. 16x16 대비 32x32 크기에서 실행 시간이 줄어들었고 64x64 크기에 대폭 감소했다가 그 이후의 크기에는 비슷한 실행 시간을 기록하였다. Block 크기 변화에 따른 SM당 resident blocks 등의 변화로 인해 생기는 차이인 것으로 판단된다.



(2) 그 다음은 global memory 대신 shared memory를 사용한 결과이다. Block dimension의 영향은 거의 없다고 볼 수 있겠지만 global memory를 사용한 구현보다 훨씬 빠른 것을 알 수 있다. Shared memory는 on-chip memory 이기 때문에 off-chip memory인 global memory보다 latency가 훨씬 적기 때문이다.



(3) 마지막으로 shared memory 구현에다가 추가적인 optimization을 추가한 방식이다. Shared memory의 bank conflict를 줄이려면 하나의 thread가 연속적인 memory 공간에 대한 access를 피해야 한다. 따라서 행렬 A*B 연산에서 A의 i번째 행과 B의 j번째 열을 사용한 곱셈을 할 때 A의 i번째 행을 transpose시켜 같은 행의 연속적인 원소들끼리 padding이 존재하도록 하는 것이다. 실제로 이렇게 구현을 한 결과 (2)에서 구현한 일반 shared memory 방식과는 실행 시간의 큰 차이가 없지만 약간 더 빠른 경우가 생긴다는 것을 알 수 있다.

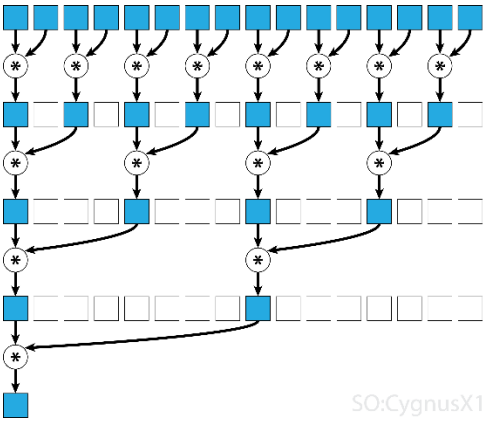
3. CUDA Programming 2

(1) 다음은 array 크기가 10,000일 때의 실행 결과이다.

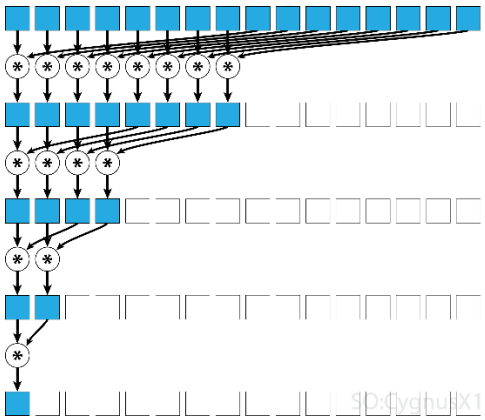
		Block size					
		16	**32**	**64**	**128**	**256**	**512**
Time (ms)	**Seq.**	0.038					
	Div.	117.153	115.113	118.612	116.256	114.606	114.467
	Opt. 1	0.365	0.348	0.361	0.351	0.346	0.345
	Opt. 2	0.450	0.389	0.368	0.344	0.357	0.366

각 구현의 특징은 다음과 같다:

- Sequential: 병렬 reduction이 아닌 array를 차례대로 탐색해서 최댓값 구함
- Divergent: path divergence가 크게 발생하는 구현

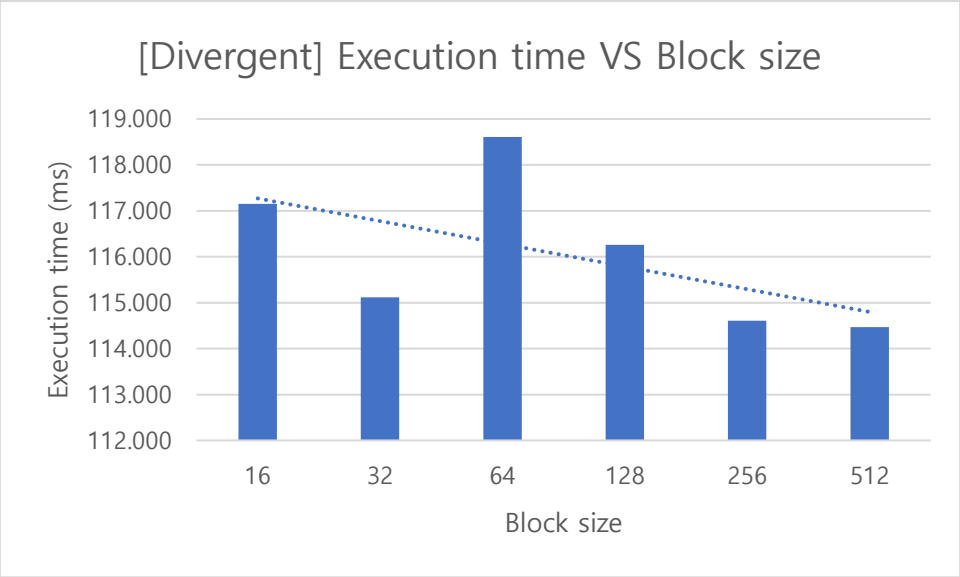


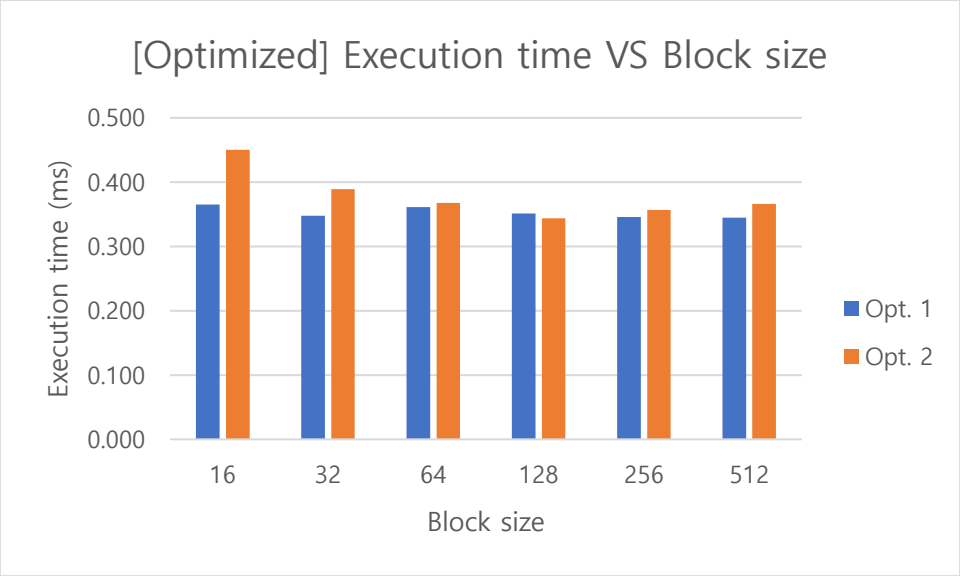
- Opt. 1: path divergence를 최대한 피한 구현



- Opt. 2: shared memory를 사용한 구현

우선 결과를 보면 sequential한 방식이 꽤 빠른 것을 알 수 있다. 이는 array 크기가 sequential 하게 처리를 해도 부담스럽지 않는 크기이며 오히려 병렬화 overhead로 인해 병렬 처리가 더 느린 것을 알 수 있다. Path divergence가 크게 발생하는 구현으로 했을 때 실행 시간이 크게 증가했다. 반대로 path divergence를 최소화하는 구현은 실행 시간을 크게 단축했다. Opt. 2 방식은 shared memory를 활용한 것인데 global이든 shared이든 memory에 쓰고 읽은 회수가 적어서 shared memory 사용으로 인한 큰 이득은 관찰할 수 없었다.

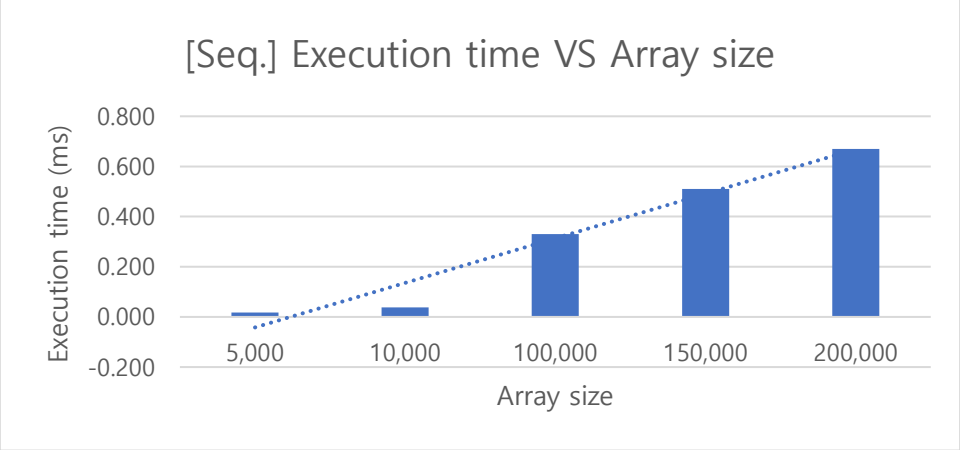




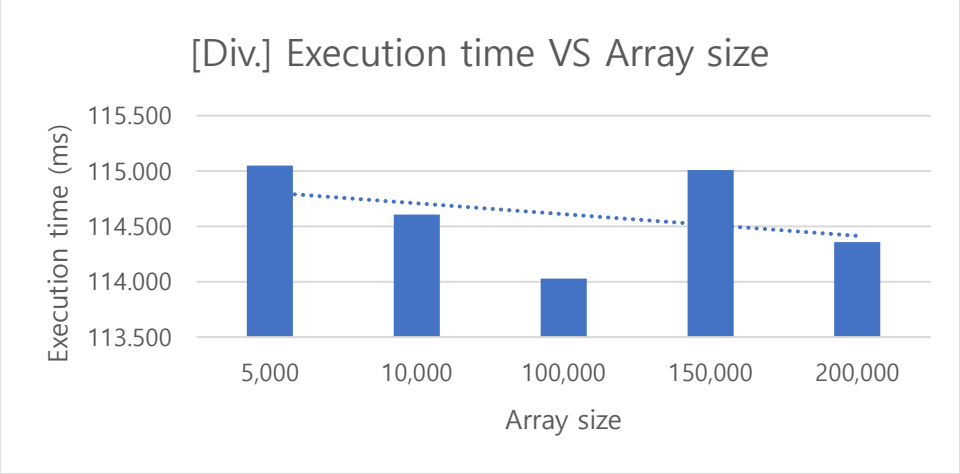
(2) 다음은 block 크기를 256으로 고정시키고 array 크기를 바꿨을 때의 결과이다.

		Array size				
		5,000	**10,000**	**100,000**	**150,000**	**200,000**
Time (ms)	**Seq.**	0.018	0.038	0.331	0.510	0.670
	Div.	115.050	114.606	114.029	115.008	114.360
	Opt. 1	0.326	0.346	0.525	0.616	0.707
	Opt. 2	0.316	0.357	-	-	-

여기서 Opt. 2는 shared memory를 활용하는 구현인데 실행 환경에 포함된 GPU의 shared memory 크기가 64MB이어서 100,000 이상의 크기로는 제대로 작동하지 않았다. 먼저 sequential 구현의 변화는 다음과 같다. 실행 시간이 array 크기에 거의 비례하면서 증가하는 것을 볼 수 있다.



Path divergence를 고려하지 않는 div. 구현의 변화는 다음과 같다. Array 크기에 따라 의미 있는 변화가 있는 것처럼 보이지만 실제로는 1ms 내외 차이로 모두 비슷하다.



마지막으로 Opt. 1구현의 변화이다. Array 크기에 따라 실행 시간이 증가하지만 병렬화 덕분에 array 크기가 10,000에서 100,000로 10배 증가했을 때 실행 시간은 1.5배 정도만 증가하는 등 증가 폭이 sequential 구현의 증가 폭보다 훨씬 적을 것을 알 수 있다.