

ICE3020 알고리즘설계 과제 1

과제 1 보고서

보고서 작성 서약서

1. 나는 타학생의 보고서를 베끼거나 여러 보고서의 내용을 짜집기하지 않겠습니다.

2. 나는 보고서의 주요 내용을 인터넷사이트 등을 통해 얻지 않겠습니다.

3. 나는 보고서의 내용을 조작하지 않겠습니다.

4. 나는 보고서 작성에 참고한 문헌의 출처를 밝히겠습니다.

5. 나는 나의 보고서를 제출 전에 타학생에게 보여주지 않겠습니다.

나는 보고서 작성시 윤리에 어긋난 행동을 하지 않고 정보통신공학인으로서 나의 명예를 지킬 것을 맹세합니다.

2021년 4월 16일

학부 정보통신공학과

학년 3학년

성명 유지훈

학번 12171810



-코드 사진-

(1번 & 2번 코드)

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

1번과 2번 문제는 QuickSort와 ShellSort의 변종 기법을 만드는 것 입니다.

QuickSort의 경우에는 중간 값 분할 방식을 사용했으며 ShellSort의 경우에는 knuth의 방식을 채택하였습니다. 중간 값 분할 부분의 코드는 21번째 줄부터 24번째 줄까지의 내용입니다. 맨 앞과 중간 그리고 맨 끝의 값을 가지고 와서 3개의 숫자를 비교해 가장 중간의 수를 pivot으로 잡아줍니다. 이렇게 하면 QuickSort의 최악의 경우인 pivot을 매우 작은 수 혹은 매우 큰 수를 잡는 경우를 방지할 수 있습니다. 이 방식을 사용하면서 전체 수행시간이 약 5% 줄어듭니다. (그래도 평균적인 시간복잡도는 O()의 시간복잡도를 가집니다.)

ShellSort의 경우에는 Concrete gaps를 어떻게 정하는지에 따라서 최악의 경우의 시간복잡도가 달라지는데 knuth의 Concrete gaps를 사용하면 최악의 경우에 O()의 시간복잡도를 가집니다.

인풋 n의 크기는 1024개부터 16,777,216(TESTNUM)개까지 2배씩 증가시켜주면서 동일한 랜덤 숫자를 부여하고 정렬 소요시간을 측정했습니다.(srand(2001)로 통일)

기존에 쓰던 ctime의 시간 계산을 이용했을 때 ms기준으로 측정하기 때문에 정확한 계산을 위해서 QueryPerformanceFrequency( )와 QueryPerformanceCounter( ) 두 개의 함수를 이용한 시간 계산을 해줬습니다.

(3번)

<main.cpp>

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

3번의 병렬정렬 알고리즘의 경우에는 CUDA를 사용하는 방법을 선택했습니다. 코드의 경우에는 CUDA에서 기본적으로 주는 예제 코드를 활용해 약간 변형을 시켰습니다.(sortingNetwork)

evenOddMergeSort와 bitonicSort 두 가지의 코드가 있었는데 기존 main.cpp에 적용된 bitonicSort를 그대로 사용했습니다. 코드의 경우에는 동일하게 입력 개수를 1024개 부터 16,777216(N)개까지 2배씩 값을 변경해가면서 시간을 측정해주었습니다. 또한 동일한 시드인 srand(2001)을 입력해 주었습니다.

안의 코드를 간단하게 둘러보면 안정성 검사를 위한 키, 벨류 쌍을 동적할당으로 cpu와 gpu사용을 구분해서 영역을 생성해줍니다. 이후 한 종류의 입력(개수)에 대해서 arraysize와 batch사이즈를 변경해가면서 분할을 다양한 방법으로 해주고 이에 따른 시간을 측정합니다. CUDA 관련 함수에 대해서는 매 실행 시 에러가 있는지 검출하고 있었습니다.

<bitonicSort.cu>

텍스트이(가) 표시된 사진

자동 생성된 설명

bitonicSort.cu 안을 간단하게 확인해보았습니다.

입력으로 데이터 관련 키와 벨류 쌍을 받아오고 이어서 batchSize와 arrayLength를 가져옵니다. 여기서 batchSize와 arrayLength는 arrayLength는 main.cpp에서 계산하는 총 입력의 수 N을 64부터 2씩 증가시키면서 증가하고 batchSize는 총 입력 N을 arrayLength로 나누어 준 것입니다. 처음 확인하는 것은 길이가 1인 경우를 확인하고 이 경우에는 비교를 할 수 없으니 return으로 함수를 끝냅니다. 아닌 경우에는 arrayLength가 최대 사용 가능한 메모리 사이즈인 SHARED\_SIZE\_LIMIT과 비교합니다. 이후 이보다 작은 경우에는 단순 bitonicSortShared( )를 통해서 정렬 및 병합을 진행해줍니다. 만일 SHARED\_SIZE\_LIMIT보다 클 경우에는 먼저 bitonicSortShared1( )를 진행을 하고(두 함수는 내용이 매우 동일합니다.) bitonicMergeGlobal( )과 bitonicMergeShared( )를 통해서 다시 한번 병합 처리를 해줍니다.

bitonicSort 안에 있는 다른 함수들은 아래 첨부하였습니다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

<bitonicSortShared> <bitonicSortShared1>

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

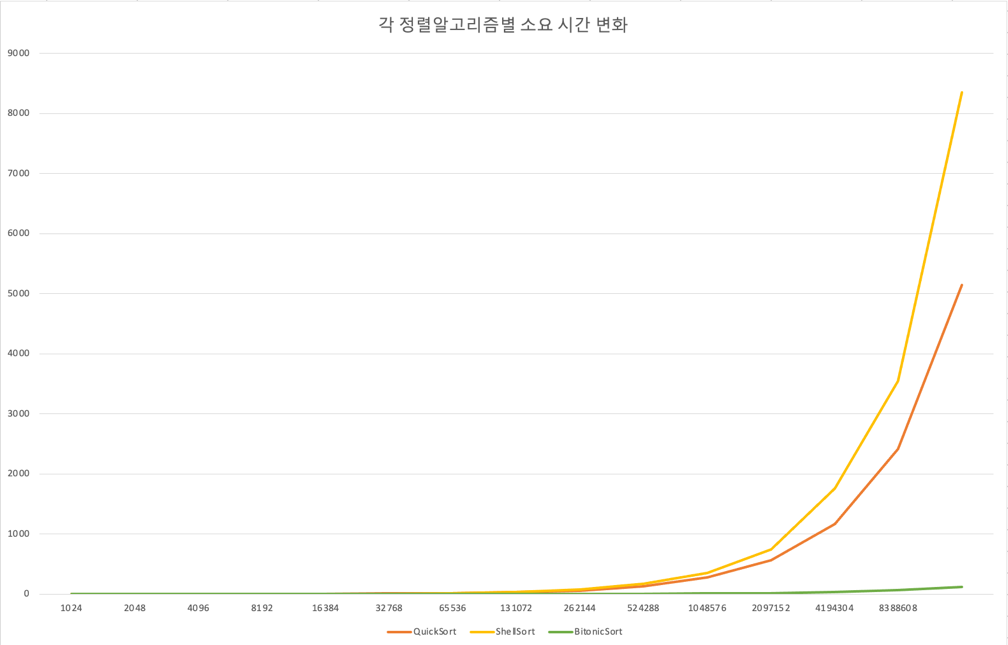
자동 생성된 설명

<bitonicMergeShared> <이 외에도 sortingNetwork 파일들이 있습니다.>

-실행 결과-

(개별 실행결과 캡쳐 사진이 많은 관계로 통합한 표와 그래프 그리고 그에 따른 해석을 우선적으로 적었습니다.)

테이블이(가) 표시된 사진

자동 생성된 설명

각 정렬 알고리즘별 소요 시간을 통합해서 하나의 그래프와 표로 만들었습니다.

입력 개수의 증가에 따른 그래프 기울기를 비교해보면

ShellSort(O())> QuickSort(O()) >> BitonicSort(병렬시 O())

다음과 같습니다. 알고리즘 수업시간에 배웠던 시간복잡도와 잘 맞아 떨어지는 그래프 입니다.

표를 통해 자세히 분석을 해보면 입력 개수가 1024개 이하인 경우에는 오히려 ShellSort가 가장 빠른 알고리즘으로 나오는데 이는 BitonicSort의 경우가 gpu와 cpu간의 병렬 처리를 해주는 과정에서 시간이 좀 더 걸린 것으로 보입니다. QuickSort와 ShellSort 두 개를 비교해주면 입력 개수가 131,072개를 넘어갈 때 소요시간이 역전되는데 이는 두 알고리즘의 C, N의 영향이 있는 것으로 보입니다. 이후에는 정상적으로 ShellSort가 가장 오래 걸리는 것을 볼 수 있습니다. bitonicSort의 경우에는 65,536개 까지는 아주 적은 시간 오차로 진행이 되는데 이는 gpu를 이용한 단순 계산이 빠르게 진행되었기 때문에 병렬 전 처리를 해주는 과정의 시간이 더 크게 적용이 되어서 그런 것으로 보입니다. 이후에는 O() 알고리즘 시간대를 보여주는데 2배씩 증가되는 입력에 맞추어서 약 2배씩 시간이 증가되는 모습을 볼 수 있었습니다. (O()

과 비슷한 성능을 볼 수 있었다.)

개별 실행시간 캡처는 아래와 같습니다.

텍스트, 칠판, 명판이(가) 표시된 사진

자동 생성된 설명텍스트, 칠판, 명판이(가) 표시된 사진

자동 생성된 설명

<QuickSort 와 ShellSort>

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

1,024개 2,048개

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

4,096개 8,192개

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

16,384개 32,768개

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

65,536개 131,072개

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

262,144개 524,288개

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

1.048,576개 2,097,152개

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

4,194,304개 8,388,608개

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

16,777,216개 16,777,216개(최악)

bitonicSort의 경우에는 arrayLength와 batchSize 조합 테스트 중 가장 빠른 경우의 시간을 기준으로 잡았습니다. 대부분의 가장 빠른 케이스의 경우 64 arraySize로 했을 때 인데 이는 batchSize가 클수록 단순 계산 병렬 처리의 양이 많아져서 그런 것으로 보입니다. 최소 arrayLength를 더 낮추었다면 더 빠른 결과가 나왔을 것으로 보입니다.

마지막에 첨부한 두 개의 사진은 bitonicSort에서 arrayLength와 batchSize를 여러 개로 해서 시도를 하는 것 중에서 가장 최선의 시간과 가장 최악의 시간을 캡처한 것입니다. 최악의 시간이 나온 경우를 확인하면 batchSize가 1인 경우인데 이 경우에는 병렬 처리가 아닌 경우라 O()의 알고리즘으로 진행된 것으로 보입니다.