report

2022_28379 협동과정 인공지능 전공 임성준

▼ parallel prefix sum

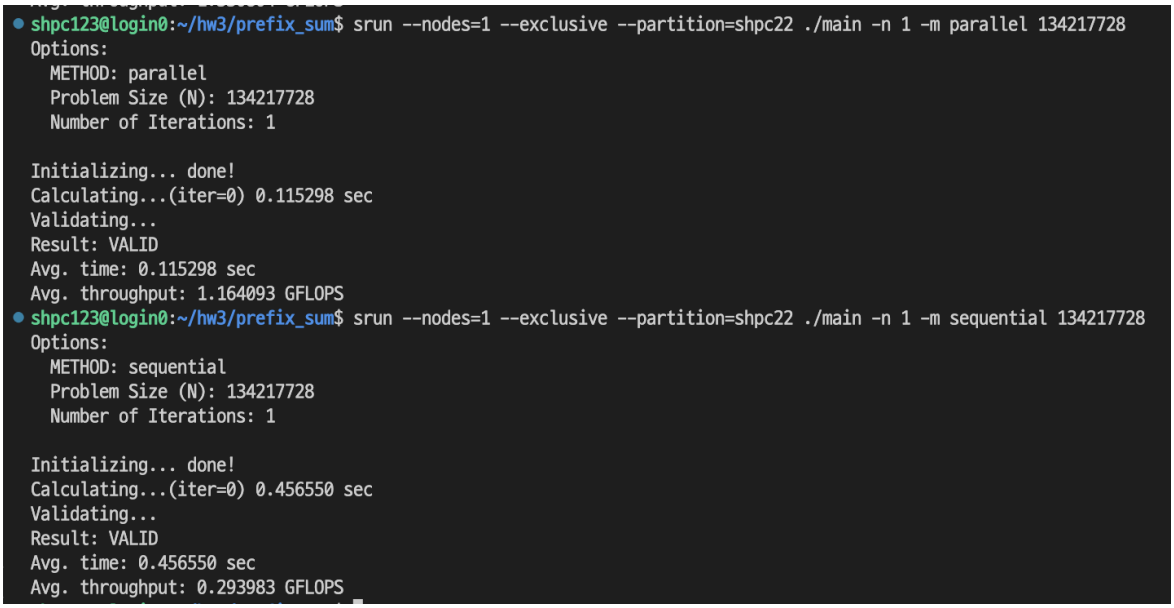
- 구현 설명
  - 반복문을 openMP를 통해서 여러 스레드에 할당해줍니다. 그러면 각 스레드가 부분 배열에서 누적 합을 구할 수 있습니다. 하지만, 특정 스레드 내의 배열을 보면 누적으로 합했지만 전체로 본다면 그렇지 않습니다. 따라서, 부분 배열의 앞 배열의 누적 값도 합해서 최종 값을 구해줘야하는데 앞 배열의 누적 값을 offset라고 하겠습니다. 그러면 모든 스레드에서 누적 합된 배열에 offset을 더해주면 됩니다.
  - 예를 들어서, thread_0이 in[0] ~ in[3]을 누적 합해서 구했다고 가정하고 thread_1이 in[4] ~ in[7]을 누적 합해서 구했다고 가정하면 in[4] ~ in[7]에 offset인 in[3]의 값을 더해주면 됩니다. 이렇게 하면 i[4]로 예를 들자면 i[4] = i[0] + i[1] + … + i[3]가 됩니다.
  - cuda를 참고해서 naive scan으로도 구현할 수 있지만 아래 코드로 구현했습니다.
  - 코드

```
// global하게 설정해야 아래 코드에서 스레드가 offset값을 공유할 수 있다.
double *offset;
#pragma omp parallel
{
  int thread_idx = omp_get_thread_num();
  int threads_num = omp_get_num_threads();
  offset = malloc(sizeof(double) * (threads_num + 1));
  offset[0] = 0;

  // partial sum
  double p_s = 0;
  #pragma omp for
  for(int i = 0; i < N; i++)
  {
      p_s += in[i];
      out[i] = p_s;
  }
  offset[thread_idx + 1] = p_s;

  // partial sum + offset
  #pragma omp barrier // 제일 중요!

  double temp = 0;
  for(int i = 0; i < (thread_idx + 1); i++)
  {
    temp += offset[i];
  }

  // 각 스레드 temp에 offset이 더해졌으므로 각 원소에 합한다. 아까랑 똑같은 배열이 각 스레드마다 할당될테니깐 그건 걱정없어도 되고 temp는 각 스레드마다 구해줘서 상관없다.
  #pragma omp for
  for(int i = 0; i < N; i++)
  {
    out[i] += temp;
  }
}
```

- double precision을 사용하는 이유는 무엇인가?
  - single precision은 정밀도가 낮아서 0.01을 더하게 되면 0.01같은 수는 2진수로 표현했을때 무한 소수가 돼서 완벽히 표현할 수 없고 에러가 커진다. 반면에 double precision은 정밀도가 single보다 높아서 오차를 개선할 수 있다.

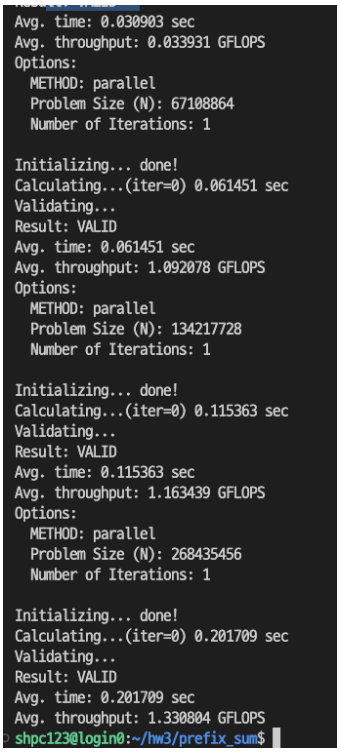
- N = 134217728 일 경우의 순차 버전 및 병렬화 버전의 성능 비교. 만약 병렬화 버전의 성능이 기대보다 느리다면, 그 이유는 무엇인가?

- 스레드가 128개라면 N = 134217728인 경우, 병렬화 버전의 성능이 순차 버전보다 좋다. 하지만 기대보다 느린 이유는 delegate가 배열에서 숫자를 더하는 것보다 시간이 더 걸리기 때문이다. 반복문에는 단순히 더하는 작업 밖에 없기때문에 스레드를 만들고 관리하는 overhead가 있어서 기대보다 느리다고 보여진다.
- 예를 들어서 N = 128인 경우에는 병렬화 버전의 성능이 순차 버전보다 나쁘다. 병렬화 버전의 overhead가 더 크다고 볼 수 있다.
- 추가로 만약 참고 자료에 있던 naive scan방식으로 구현하게 됐다면, 병렬화 버전의 성능이 더 낮을 것이다. 최적의 경우 O(log(n))이 되지만 실제로는 thread에서 하는 연산이 N에 비례해서 늘어나기때문에 time complexity는 O(nlog(n))이 된다.. 순차는 O(n)인데 반해 병렬화 버전이 더 느리게 될 것이다.



- 정확성 테스트: 2개 이상 스레드 사용 ./run_validation.sh 테스트
  - 결과이미지: 모두 valid한 결과를 얻었습니다.

```
shpc123@login0:~/hw3/prefix_sum$ ./run_validate
Options:
  METHOD: parallel
  Problem Size (N): 1
  Number of Iterations: 1

Initializing... done!
Calculating...(iter=0) 0.029329 sec
Validating...
Result: VALID
Avg. time: 0.029329 sec
Avg. throughput: 0.000000 GFLOPS
Options:
  METHOD: parallel
  Problem Size (N): 2
  Number of Iterations: 1

Initializing... done!
Calculating...(iter=0) 0.023918 sec
Validating...
Result: VALID
Avg. time: 0.023918 sec
Avg. throughput: 0.000000 GFLOPS
Options:
  METHOD: parallel
  Problem Size (N): 4
  Number of Iterations: 1

Initializing... done!
Calculating...(iter=0) 0.030503 sec
Validating...
Result: VALID
Avg. time: 0.030503 sec
Avg. throughput: 0.000000 GFLOPS
Options:
  METHOD: parallel
  Problem Size (N): 8
  Number of Iterations: 1

Initializing... done!
Calculating...(iter=0) 0.024596 sec
Validating...
Result: VALID
Avg. time: 0.024596 sec
Avg. throughput: 0.000000 GFLOPS
Options:
  METHOD: parallel
  Problem Size (N): 16
  Number of Iterations: 1

Initializing... done!
Calculating...(iter=0) 0.030442 sec
Validating...
Result: VALID
Avg. time: 0.030442 sec
Avg. throughput: 0.000001 GFLOPS
Options:
  METHOD: parallel
  Problem Size (N): 4096
  Number of Iterations: 1

Initializing... done!
Calculating...(iter=0) 0.031203 sec
Validating...
Result: VALID
Avg. time: 0.031203 sec
Avg. throughput: 0.000131 GFLOPS
Options:
  METHOD: parallel
  Problem Size (N): 1048576
  Number of Iterations: 1

Initializing... done!
Calculating...(iter=0) 0.030903 sec
Validating...
Result: VALID
```



▼ openMP MatrixMultiplication

- 자신의 병렬화 방식에 대한 설명
  - `#pragma omp parallel`는 thread 그룹을 spawn하고 `#pragma omp for`는 loop iterations를 생성한 threads에 나눠줍니다. 이를 동시에 할 수 있는 `#pragma omp parallel for`를 통해서 matrix multiplication을 구현했습니다. 추가로 성능을 높이기 위해서 캐시의 지역성을 이용했습니다. index배치를 i,j,k 순서가 아닌 i,k,j로 했습니다.
  - 코드

```
// TODO: FILL_IN_HERE
#pragma omp parallel for
for (int i = 0; i < M; ++i) {
  for (int k = 0; k < K; ++k) {
    for (int j = 0; j < N; ++j) {
      C[i * N + j] += A[i * K + k] * B[k * N + j];
    }
  }
}
```

- openMP에서 thread 생성 방식 / 여기서 컴파일러, 런타임 시스템의 역할
  - master thread가 필요한 team of threads를 생성합니다. 컴파일러가 complier directives를 통해 parallel region을 만들어주고 해당 지역에서 thread를 생성하므로 명시적으로 thread를 생성하게 됩니다. 그리고 runtime system을 통해서 필요한 openMP 함수들을 제공합니다.

- 스레드 1 - 256에서 성능 측정 / 증가? / 이유?

1	4	32	64	128	256	512
91.42	98.36	78.28	57.38	103.91	101.93	97.27

- `./run.sh -v -n 10 -t 512 4096 4096 4096` 명령어 기준으로 GFLOPS를 측정하여 위와 같은 결과가 나왔습니다.
- 64개까지는 스레드가 증가할수록 오히려 성능이 감소했고, 128개에서는 성능이 증가했지만 다시 감소하기 시작했습니다. 일반적으로 스레드 개수가 증가하면 병렬적으로 처리할 수 있어서 성능이 증가합니다. 하지만 스레드가 많아지는 경우 아래와 같이 성능이 감소하는 경우도 발생합니다. 스레드가 하는 일에 변화가 없고 오히려 switch 등에서 오버헤드가 발생해서 그럴 수 있습니다. 해당 실험에서는 반복문에는 단순히 배열의 원소를 더하는 작업 밖에 없기때문에 스레드를 만들고 관리하는 overhead가 있어서 성능이 감소한 것으로 보여집니다.

- 가장 높은 성능일때 peak performance 대비 어느정도? 더 높게 하기 위한 방법은?
  - 위의 실험결과에서 스레드 128개를 사용한 경우, 약 104 GFLOPS의 성능을 보였습니다. peak performance의 경우는 5120 GFLOPS로 **2.03**%의 성능을 보이고 있습니다. 더 높은 성능을 위해서는 여러가지 방법이 있습니다. 첫 번째, 앞에서 배운 AVX, FMA 명령어를 사용할 수 있습니다. 두 번째, tiling 기법을 통해 cache hit ratio를 증가시킬 수 있습니다.

- openMP의 loop scheduling 설명 / static, dynamic, guided 설명 / 실험으로 성능 비교
  - loop scheduling을 통해서 loop iteration을 thread 사이에 어떻게 할당할 것인지 정할 수 있다.
  - static 방법은 loop를 동일하게 분배한다. 예를 들어서 N elements를 M subsets로 나누면 N/M만큼 할당된다. 분배한 상태로 작업을 진행한다. 반면에 dynamic은 분배한 상태로 작업하지 않고 그때 그때 배분한다. 만약에 subset의 양이 일정하게 나눠지지 않는 경우에도 사용할 수 있다. 대신 Overhead가 존재한다. guided 방법은 dynamic과 매우 유사하지만 다른 점은 배분되는 작업의 양이 줄어든다.
  - static, dynamic, guided 순서로 아래 결과를 확인할 수 있다. 성능은 static 가장 좋았다. dynamic이 가장 좋지 않았다.

```
shpc123@login0:~/hw3/matmul$ ./run_performance.sh
Options:
  Problem size: M = 4096, N = 4096, K = 4096
  Number of threads: 32
  Number of iterations: 10
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 1.700116 sec
Calculating...(iter=1) 1.713628 sec
Calculating...(iter=2) 1.582655 sec
Calculating...(iter=3) 1.708797 sec
Calculating...(iter=4) 1.578051 sec
Calculating...(iter=5) 1.709305 sec
Calculating...(iter=6) 1.588638 sec
Calculating...(iter=7) 1.573624 sec
Calculating...(iter=8) 1.699476 sec
Calculating...(iter=9) 1.586394 sec
Validating...
Result: VALID
Avg. time: 1.644068 sec
Avg. throughput: 83.596856 GFLOPS
```

```
shpc123@login0:~/hw3/matmul$ ./run_performance.sh
Options:
  Problem size: M = 4096, N = 4096, K = 4096
  Number of threads: 32
  Number of iterations: 10
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 1.606537 sec
Calculating...(iter=1) 1.622152 sec
Calculating...(iter=2) 1.494814 sec
Calculating...(iter=3) 1.490037 sec
Calculating...(iter=4) 1.596670 sec
Calculating...(iter=5) 1.495333 sec
Calculating...(iter=6) 1.616591 sec
Calculating...(iter=7) 1.488303 sec
Calculating...(iter=8) 1.493700 sec
Calculating...(iter=9) 1.622796 sec
Validating...
Result: VALID
Avg. time: 1.552693 sec
Avg. throughput: 88.516482 GFLOPS
```

```
shpc123@login0:~/hw3/matmul$ ./run_performance.sh
Options:
  Problem size: M = 4096, N = 4096, K = 4096
  Number of threads: 32
  Number of iterations: 10
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 1.706382 sec
Calculating...(iter=1) 1.708048 sec
Calculating...(iter=2) 1.572104 sec
Calculating...(iter=3) 1.691771 sec
Calculating...(iter=4) 1.573684 sec
Calculating...(iter=5) 1.705288 sec
Calculating...(iter=6) 1.592647 sec
Calculating...(iter=7) 1.566249 sec
Calculating...(iter=8) 1.586138 sec
Calculating...(iter=9) 1.703424 sec
Validating...
Result: VALID
Avg. time: 1.640574 sec
Avg. throughput: 83.774940 GFLOPS
```

- 정확성 run_validation.sh

```
shpc123@login0:~/hw3/matmul$ ./run_validation.sh
Options:
  Problem size: M = 831, N = 538, K = 2304
  Number of threads: 26
  Number of iterations: 1
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 0.016293 sec
Validating...
Result: VALID
Avg. time: 0.016293 sec
Avg. throughput: 126.441149 GFLOPS
Options:
  Problem size: M = 3305, N = 1864, K = 3494
  Number of threads: 9
  Number of iterations: 1
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 0.415934 sec
Validating...
Result: VALID
Avg. time: 0.415934 sec
Avg. throughput: 103.501340 GFLOPS
Options:
  Problem size: M = 618, N = 3102, K = 1695
  Number of threads: 38
  Number of iterations: 1
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 0.062364 sec
Validating...
Result: VALID
Avg. time: 0.062364 sec
Avg. throughput: 104.206830 GFLOPS
Options:
  Problem size: M = 1876, N = 3453, K = 3590
  Number of threads: 30
  Number of iterations: 1
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 0.688839 sec
Validating...
Result: VALID
Avg. time: 0.688839 sec
Avg. throughput: 67.520622 GFLOPS
Options:
  Problem size: M = 1228, N = 2266, K = 1552
  Number of threads: 16
  Number of iterations: 1
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 0.040461 sec
Validating...
Result: VALID
Avg. time: 0.040461 sec
Avg. throughput: 213.473577 GFLOPS
Options:
  Problem size: M = 3347, N = 171, K = 688
  Number of threads: 2
  Number of iterations: 1
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 0.013296 sec
Validating...
Result: VALID
```

```
Avg. time: 0.013296 sec
Avg. throughput: 59.228931 GFLOPS
Options:
  Problem size: M = 3583, N = 962, K = 765
  Number of threads: 39
  Number of iterations: 1
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 0.035046 sec
Validating...
Result: VALID
Avg. time: 0.035046 sec
Avg. throughput: 150.480580 GFLOPS
Options:
  Problem size: M = 2962, N = 373, K = 1957
  Number of threads: 30
  Number of iterations: 1
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 0.029614 sec
Validating...
Result: VALID
Avg. time: 0.029614 sec
Avg. throughput: 146.023574 GFLOPS
Options:
  Problem size: M = 3646, N = 2740, K = 3053
  Number of threads: 9
  Number of iterations: 1
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 0.656898 sec
Validating...
Result: VALID
Avg. time: 0.656898 sec
Avg. throughput: 92.859423 GFLOPS
Options:
  Problem size: M = 1949, N = 3317, K = 3868
  Number of threads: 26
  Number of iterations: 1
  Print matrix: off
  Validation: on

Initializing... done!
Calculating...(iter=0) 0.729148 sec
Validating...
Result: VALID
Avg. time: 0.729148 sec
Avg. throughput: 68.589556 GFLOPS
```

- 성능 run_performance.sh 60넘기기

```
shpc123@login0:~/hw3/matmul$ ./run_performance.sh
  Options:
    Problem size: M = 4096, N = 4096, K = 4096
    Number of threads: 32
    Number of iterations: 10
    Print matrix: off
    Validation: on

  Initializing... done!
  Calculating...(iter=0) 1.944801 sec
  Calculating...(iter=1) 1.962685 sec
  Calculating...(iter=2) 1.778158 sec
  Calculating...(iter=3) 1.872474 sec
  Calculating...(iter=4) 1.742819 sec
  Calculating...(iter=5) 1.852544 sec
  Calculating...(iter=6) 1.707272 sec
  Calculating...(iter=7) 1.789439 sec
  Calculating...(iter=8) 1.679515 sec
  Calculating...(iter=9) 1.661890 sec
  Validating...
  Result: VALID
  Avg. time: 1.799160 sec
  Avg. throughput: 76.390635 GFLOPS
```