Bilinear Interpolation

2017202088 신해담

I. Introduction

부동소수점 데이터로 이루어진 NxN 정방행렬에 대한 Bilinear Interpolation을 수행하는 성능이 가장 좋은 코드를 구현한다. 결과 행렬의 크기는 $(N*2^k)x(N*2^k)$이며, 마지막 행과 열의 데이터는 padding을 적용하여 interpolation을 수행한다. 성능의 기준은 code size * $state^2$의 값이 작을수록 좋다고 정의한다.

코드 작성 단계에서 필요한 함수를 순서대로 구현한다. 입력받은 데이터를 저장하는 함수, bilinear interpolation을 수행하는 함수, 행과 열을 이용하여 루프를 수행하는 함수, padding을 수행하는 함수 등을 차례대로 구현한다.

코드 검증 단계에서 구현한 코드가 올바른 결과 행렬을 출력하는지 확인한다. 각종 행렬을 입력하고 무작위로 선별하여 바른 값이 저장되었는지 확인한다.

프로젝트의 수행 일정은 다음과 같다.

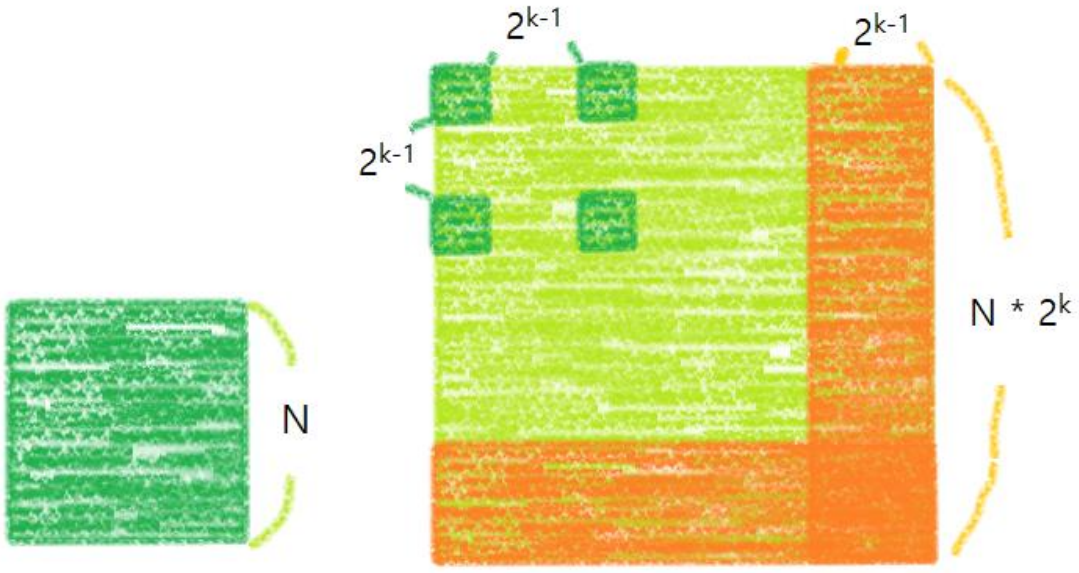
주차	12	13	14	15
제안서				
코드 작성				
코드 검증				
결과보고서				

II. Project Specification

데이터는 Matrix_data label에 DCD 명령어를 통해 N, k, NxN 행렬 순으로 선언되어 있다. 행렬 데이터는 -2000 ~ 2000 사이의 부동소수점 값으로 구성되어 있다. 이를 Bilinear Interpolation하여 $(N*2^k)x(N*2^k)$ 크기의 결과 행렬을 계산한다. 이 때 k는 1~3 사이의 값을 가지며, 기존 행렬을 1~8배 하게 된다. 기존 행렬 사이에는 생성된 값이 k의 값에 따라 2^k-1개만큼 채워진다.

마지막 행과 열의 데이터는 사이에 들어갈 값을 구할 짝이 없으므로, 생성될 데이터와 가장 가까운 위치의 값으로 padding을 적용하여 interpolation을 수행한다.

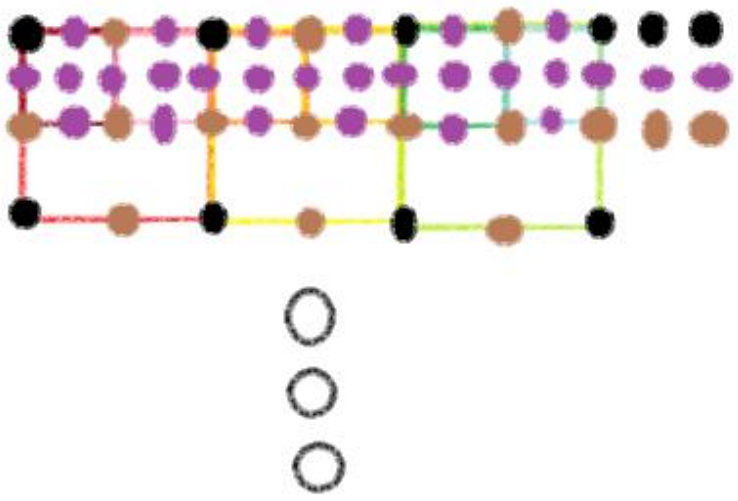
결과 행렬은 Result_data label로 선언된 메모리 주소에 직렬화되어 저장된다. 이 때 주소는 0x60000000로 통일하며, 1 word 단위로 저장된다.



III. Algorithm

- calcBlock
  한 점의 주소와 그 점을 좌상단의 점으로 가지는 2x2 행렬을 이용하여, 각 점들의 사이값을 구하여 3x3 행렬을 계산한다. 이 때, 그리고 주소는 다음 점을 가리키도록 설정한다.
  이 함수만을 이용하여 계산하면 중복으로 계산되는 점이 발생하게 된다. 따라서 이를 변형하여 5개의 점을 모두 구하는 calcFirstPointBlock 함수, 좌상과 좌하의 사이값을 구하지 않는 calcFirstRow 함수, 좌상과 우상의 사이값을 구하지 않는 clacFirstCol 함수, 좌상-좌하와 좌상-우상의 사이값을 구하지 않는 calcMainBlock 함수를 만든다.

  빨간색이 calcFirstPointBlock, 노란색이 calcFirstRow, 초록색이 calcFirstCol, 하늘색이 calcMainBlock 함수를 나타낸다.
- InitMatrix
  입력받은 NxN 행렬을 메모리에 쓰는 작업을 수행한다. 이 때, 최초 한 번은 한 행에서 다음 행으로 넘어가는 데 필요한 메모리 증가량을 확인하여 temp 메모리에 저장한다. 이 함수에서 최초 행렬의 padding 또한 수행햐여 Bilinear Interpolation의 수행 부담을 줄인다.
- Bilinear Interpolation
  모든 행과 열에서 calcBlock 함수를 이용하여 결과 행렬의 값을 구한다. 입력받은 NxN 행렬의 2배 행렬을 구하고, 그렇게 구한 2Nx2N 행렬을 이용하여 4Nx4N을 구하며 계속 확장한다. 이 때 N은 2배씩 증가하여 최대 $N*2^k$의 값을 가진다. k는 매 행렬 계산이 끝날 때마다 -1을 하여, k가 0이 되면 Bilinear Interpolation을 정지한다. 행렬 계산과 동시에 행의 padding 또한 동시에 수행한다. calcBlock을 수행하다가 행의 끝에 다다르면 직전에 구한 우상-우하 값의 계산으로 나온 값을 padding한다.



그림은 Bilinear Interpolation 수행 예시이다. 검은색 점은 최초 입력 행렬 값이다. 검은 점을 꼭지점으로 하는 사각형에서 사각형 안의 갈색 점 값을 구한다. 이를 모든 검은 점에서 반복한다. 그리고 그렇게 구한 갈색 점과 검은색 점을 이용하여 보라색 점들을 구한다. Padding의 경우, 검은색 점을 이용하여 갈색 점을 구할 때 행의 끝에 다다르면 갈색 점을 나열한다.

- colPadding

  Bilinear Interpolation을 수행하면서 행의 padding은 수행되지만, 열의 padding을 수행하지 않으므로 결과 행렬을 모두 구한 다음 마지막에 수행한다. colPadding을 수행하지 않은 행렬의 좌하단 끝의 값을 가리키도록 메모리 주소를 설정한 다음, 그 값을 읽어 2^k-1 만큼 행을 넘겨가며 값을 저장한다. 그리고 메모리 주소를 다음 열의 값으로 넘긴다. 이를 N*2^k(신규 행렬의 한 변의 길이) 만큼 반복한다.

IV. Performance & Result

2x2 행렬을 2^3배 확장할 때의 Performance와 Result

```
Program Size: Code=1216 RO-data=0 RW-data=0 ZI-data=0
States                  8727
```

1216 * 8727^2 = 92611203264로 약 926억의 Performance를 가진다.

```
0x60000000: 2.03125   0.902344  -0.226563  -1.35547   -2.48438   -3.61328   -4.74219   -5.87109   -7         -7         -7
0x6000002C: -7        -7        -7         -7         -7         34.2773    29.4927    24.708     19.9233    15.1387    10.354
0x60000058: 5.56934   0.784668  -4         -4         -4         -4         -4         -4         -4         -4         66.5234
0x60000084: 58.083    49.6426   41.2021    32.7617    24.3213    15.8809    7.44043    -1         -1         -1         -1
0x600000B0: -1        -1        -1         -1         98.7695    86.6733    74.5771    62.481     50.3848    38.2886    26.1924
0x600000DC: 14.0962   2         2          2          2          2          2          2          2          131.016    115.264
0x60000108: 99.5117   83.7598   68.0078    52.2559    36.5039    20.752     5          5          5          5          5
0x60000134: 5         5         5          163.262    143.854    124.446    105.039    85.6309    66.2231    46.8154    27.4077
0x60000160: 8         8         8          8          8          8          8          8          195.508    172.444    149.381
0x6000018C: 126.317   103.254   80.1904    57.127     34.0635    11         11         11         11         11         11
0x600001B8: 11        11        227.754    201.035    174.315    147.596    120.877    94.1577    67.4385    40.7192    14
0x600001E4: 14        14        14         14         14         14         14         260        229.625    199.25     168.875
0x60000210: 138.5     108.125   77.75      47.375     17         17         17         17         17         17         17
0x6000023C: 17        260       229.625    199.25     168.875    138.5      108.125    77.75      47.375     17         17
0x60000268: 17        17        17         17         17         17         260        229.625    199.25     168.875    138.5
0x60000294: 108.125   77.75     47.375     17         17         17         17         17         17         17         17
0x600002C0: 260       229.625   199.25     168.875    138.5      108.125    77.75      47.375     17         17         17
0x600002EC: 17        17        17         17         17         260        229.625    199.25     168.875    138.5      108.125
0x60000318: 77.75     47.375    17         17         17         17         17         17         17         17         260
0x60000344: 229.625   199.25    168.875    138.5      108.125    77.75      47.375     17         17         17         17
0x60000370: 17        17        17         17         260        229.625    199.25     168.875    138.5      108.125    77.75
0x6000039C: 47.375    17        17         17         17         17         17         17         17         260        229.625
0x600003C8: 199.25    168.875   138.5      108.125    77.75      47.375     17         17         17         17         17
0x600003F4: 17        17        17         0          0          0          0          0          0          0          0
```

생성된 16x16 행렬.

3x3 행렬을 2배 확장할 때의 Performance와 Result

```
Program Size: Code=1236 RO-data=0 RW-data=0 ZI-data=0
States                  1383
```

1236 * 1383^2 = 2364083604 로 약 24억의 performance를 가진다.

```
Memory 1
Address: 0x60000000
0x60000000: 110          179          248          116          -16          -16
0x60000018: 13           65.125       117.25       55.6094      -6.03125     -6.03125
0x60000030: -84          -48.75       -13.5        -4.78125     3.9375       3.9375
0x60000048: -42.1758     -9.46289     23.25        -179.391     -382.031     -382.031
0x60000060: -0.351563    29.8242      60           -354         -768         -768
0x60000078: -0.351563    29.8242      60           -354         -768         -768
```

생성된 6x6 행렬.

3x3 행렬을 2^2배 확장할 때의 Performance와 Result
1236 * 5888^2 = 42850320384 로 약 429억의 performance를 가진다.

Memory 1

Address: 0x60000000

0x60000000: 58	43.0234	28.0469	13.0703	-1.90625	-2.92969
0x60000018: -3.95313	-4.97656	-6	-6	-6	-6
0x60000030: 15.5	19.2676	23.0352	26.8027	30.5703	24.4902
0x60000048: 18.4102	12.3301	6.25	6.25	6.25	6.25
0x60000060: -27	-4.48828	18.0234	40.5352	63.0469	51.9102
0x60000078: 40.7734	29.6367	18.5	18.5	18.5	18.5
0x60000090: -69.5	-28.2441	13.0117	54.2676	95.5234	79.3301
0x600000A8: 63.1367	46.9434	30.75	30.75	30.75	30.75
0x600000C0: -112	-52	8	68	128	106.75
0x600000D8: 85.5	64.25	43	43	43	43
0x600000F0: 14	49.5	85	120.5	156	126.313
0x60000108: 96.625	66.9375	37.25	37.25	37.25	37.25
0x60000120: 140	151	162	173	184	145.875
0x60000138: 107.75	69.625	31.5	31.5	31.5	31.5
0x60000150: 266	252.5	239	225.5	212	165.438
0x60000168: 118.875	72.3125	25.75	25.75	25.75	25.75
0x60000180: 392	354	316	278	240	185
0x60000198: 130	75	20	20	20	20
0x600001B0: 392	354	316	278	240	185
0x600001C8: 130	75	20	20	20	20
0x600001E0: 392	354	316	278	240	185
0x600001F8: 130	75	20	20	20	20
0x60000210: 392	354	316	278	240	185
0x60000228: 130	75	20	20	20	20

생성된 12x12 행렬.

10x10 행렬을 2^3배 확장할 때의 Performance와 Result

```
Program Size: Code=1600 RO-data=0 RW-data=0 ZI-data=0
```

States 339232

1600 * 339232^2 = 184125359718400 로 약 184조의 performance를 가진다.

Memory 1

Address: 0x60000000

0x60000000: 160	102	44	-14	-72	-130	-188	-246	-304	-233	-162
0x6000002C: -91	-20	51	122	193	264	223	182	141	100	59
0x60000058: 18	-23	-64	-22	20	62	104	146	188	230	272
0x60000084: 237.555	203.109	168.664	134.219	99.7734	65.3281	30.8828	-3.5625	49.8828	103.328	156.773
0x600000B0: 210.219	263.664	317.109	370.555	424	366.75	309.5	252.25	195	137.75	80.5
0x600000DC: 23.25	-34	-30.5156	-27.0313	-23.5469	-20.0625	-16.5781	-13.0938	-9.60938	-6.125	-5.64063
0x60000108: -5.15625	-4.67188	-4.1875	-3.70313	-3.21875	-2.73438	-2.25	-2.25	-2.25	-2.25	-2.25
0x60000134: -2.25	-2.25	-2.25	140.156	86.7617	33.3672	-20.0273	-73.4219	-126.816	-180.211	-233.605
0x60000160: -287	-215.75	-144.5	-73.25	-2	69.25	140.5	211.75	283	240.281	197.563

생성된 80x80 행렬. 이하 생략.

V. Conclusion

Bilinear Interpolation을 정상적으로 수행하는 코드를 처음 제작한 이후, 10x10 행렬의 8배 행렬을 테스트하자 state가 50만을 넘었다. 무엇이 state를 많이 사용하는지 조사한 결과, calcBlock 함수를 이용하여 구현하면 같은 점을 중복하여 계산하는 경우가 발생한다는 것을 알아내었다. 따라서 이를 해결하기 위해 모든 점을 구하지 않고 특정 점을 제외하는 함수를 만들어 4개의 함수로 분리하였다.

calcBlock 함수를 분리하여 삽입하면서 문제가 발생했는데, n이 2이면 calcBlock 함수가 한 번만 수행되어야 하지만 이를 제한하는 코드가 없어서 무한루프로 빠지는 버그였다. Conditional branch를 사용하여 해결할 수 있었다.

어떤 함수가 link register를 사용하면, 그 함수를 이용하는 함수는 link register를 사용할 수 없다는 문제점이 있었다. 따라서 Branch를 이용하여 함수를 마치고 돌아올 지점을 설정하여 사용하였다.

Loop unrolling을 사용하면 branch에 사용되는 state를 줄일 수 있다. 하지만 이번에 구현한 코드는 calcBlock 함수를 통해 계속 메모리의 값을 read/write를 수행하는 코드로써, 쉬운 알고리즘과 뛰어난 유연성이 장점이라고 생각하지만 같은 값을 반복적으로 load 하면서

생기는 state 증가량이 다른 요소들을 압도한다. Branch 사용을 줄이는 정도로는 눈에 띄는 변화는 없을 것으로 예상하며, state를 줄일 수 있는 획기적인 알고리즘을 고안하는 것이 해결 방법일 것으로 추정한다.

이 프로젝트를 완료함으로써, 어셈블리어를 이용한 프로그램을 설계하는 능력과 구현한 코드를 분석하는 능력이 향상될 것으로 기대된다. 또한 state를 줄이는 방법들이 어느 코드에 어떻게 사용할 수 있을지 판단하고 수행하는 능력 또한 함양될 수 있을 것이다.