CSI2107_01_PA4 Report 학번: 2022132017 이름: 오성민

과제의 목적은 4096 by 4096 행렬 두개를 곱셈하는 연산 프로그램을 최적화하는 것이 목적이다. 먼저, 주어진 코드를 눈에 보이는 기본적인 최적화를 진행해보았다.

```c
#include <stdio.h>

void fc_layer(size_t data_cnt,
        size_t input_dim,
        size_t output_dim,
        float* matrix,
        float* bias,
        float* input,
        float* output) {
    // loop over input instances
    size_t n1 = 0;
    size_t n2 = 0;

    for ( size_t iidx = 0; iidx < data_cnt; iidx++ ) {
        // loop over weight columns
        for ( size_t oidx = 0; oidx < output_dim; oidx++ ) {
            float outv = bias[oidx];
            size_t idx = oidx;
            // loop over each input's activation values
            for ( size_t aidx = 0; aidx < input_dim; aidx++ ) {
                float inv = input[n1++];
                float weight = matrix[idx];

                outv += inv * weight;
                idx += output_dim;
            }
            n1 -= input_dim;
            if ( outv < 0 ) outv = 0;

            output[n2++] = outv;
        }
        n1 += input_dim;
    }
}
```

Make를 한 뒤, ./nnfc data/inputs.32.bin data/outputs.32.bin를 실행해보면,

```
Model loaded successfuly!
Example data loaded successfuly!
Input count: 32
Starting layer...
Elapsed time: 3.401784 s
Performance (MFLOPS): 315.000000
Average error: 0.000832
```

약간의 성능향상이 있었지만, 우리가 목표까지 도달해야할 14000 MLOPS 까지 가는 데에는 역부족이었다. 위와 같이 계산을 할 경우, 캐시 손실율이 매우 크기 때문에 연산 순서를 바꿀 필요가 있다. 캐시 친화적인 연산을 하기 위해서 코드를 다음과 같이 수정하였다.

```c
#include <stdio.h>

void fc_layer(size_t data_cnt,
        size_t input_dim,
        size_t output_dim,
        float* matrix,
        float* bias,
        float* input,
        float* output) {
    // loop over input instances
    size_t n1 = 0;
    size_t n2 = 0;

    for( size_t i = 0; i < data_cnt; i++){
        for(size_t k = 0; k < input_dim; k++){
            float r = input[n1++];
            for(size_t j = 0; j < output_dim; j+=2){
                output[n2 + j] += r * matrix[output_dim *k + j];
                output[n2 + j + 1] += r * matrix[output_dim *k + j + 1];
                if(k == input_dim-1){
                    output[n2 + j] += bias[j];
                    output[n2 + j + 1] += bias[j + 1];
                    if(output[n2 + j] < 0) output[n2 + j] = 0;
                    if(output[n2 + j + 1] < 0) output[n2 + j + 1] = 0;
                }
            }
        }
        n2 += output_dim;
    }
}

```

이렇게 하게 되면, input matrix의 element를 고정하고 weigted matrix와 output matrix를 sequential하게 접근할 수 있기 때문에 더욱 캐시 친화적인 방법이라고 할 수 있다. 그리고, 가장 내부에 있는 반복문은 loop unrolling을 통해서 파이프라이닝을 효율적이게 설계하였다.

```
Model loaded successfuly!
Example data loaded successfuly!
Input count: 32
Starting layer...
Elapsed time: 0.267868 s
Performance (MFLOPS): 4009.000000
Average error: 0.000880
```

실행결과 눈에 띄게 MFLOPS가 향상된 것을 확인할 수 있었지만, 아직까지도 부족하다. 우리는 여기서 SIMD라는 레지스터를 활용하여서, 병렬 연산을 하여 float 곱셈 연산을 8배나 빠르게 만들 수 있다.

```
void fc_layer(size_t data_cnt,
        size_t input_dim,
        size_t output_dim,
        float* matrix,
        float* bias,
        float* input,
        float* output) {
    // loop over input instances

    size_t n1 = 0;
    size_t n2 = 0;

    for( size_t i = 0; i < data_cnt; i++){
        for(size_t k = 0; k < input_dim; k++){
            float r = input[n1++];
            __m256 vec1 = _mm256_set1_ps(r);
            #pragma unroll(2)
            for(size_t j = 0; j < output_dim; j += 16){
                __m256 result1 = _mm256_mul_ps(vec1, _mm256_load_ps(matrix + output_dim * k + j));
                _mm256_store_ps(output + n2 + j, _mm256_add_ps(result1, _mm256_load_ps(output + n2 + j)));

                __m256 result2= _mm256_mul_ps(vec1, _mm256_load_ps(matrix + output_dim * k + j + 8));
                _mm256_store_ps(output + n2 + j + 8, _mm256_add_ps(result2, _mm256_load_ps(output + n2 + j + 8)));
            }
            if(k == input_dim-1){
                for(size_t j = 0; j < output_dim; j += 2){
                    output[n2 + j] += bias[j];
                    output[n2 + j + 1] += bias[j + 1];
                    if(output[n2 + j] < 0) output[output_dim * i + j] = 0;
                    if(output[n2 + j + 1] < 0) output[output_dim * i + j + 1] = 0;
                }
            }
        }
        n2 += output_dim;
    }
}
```

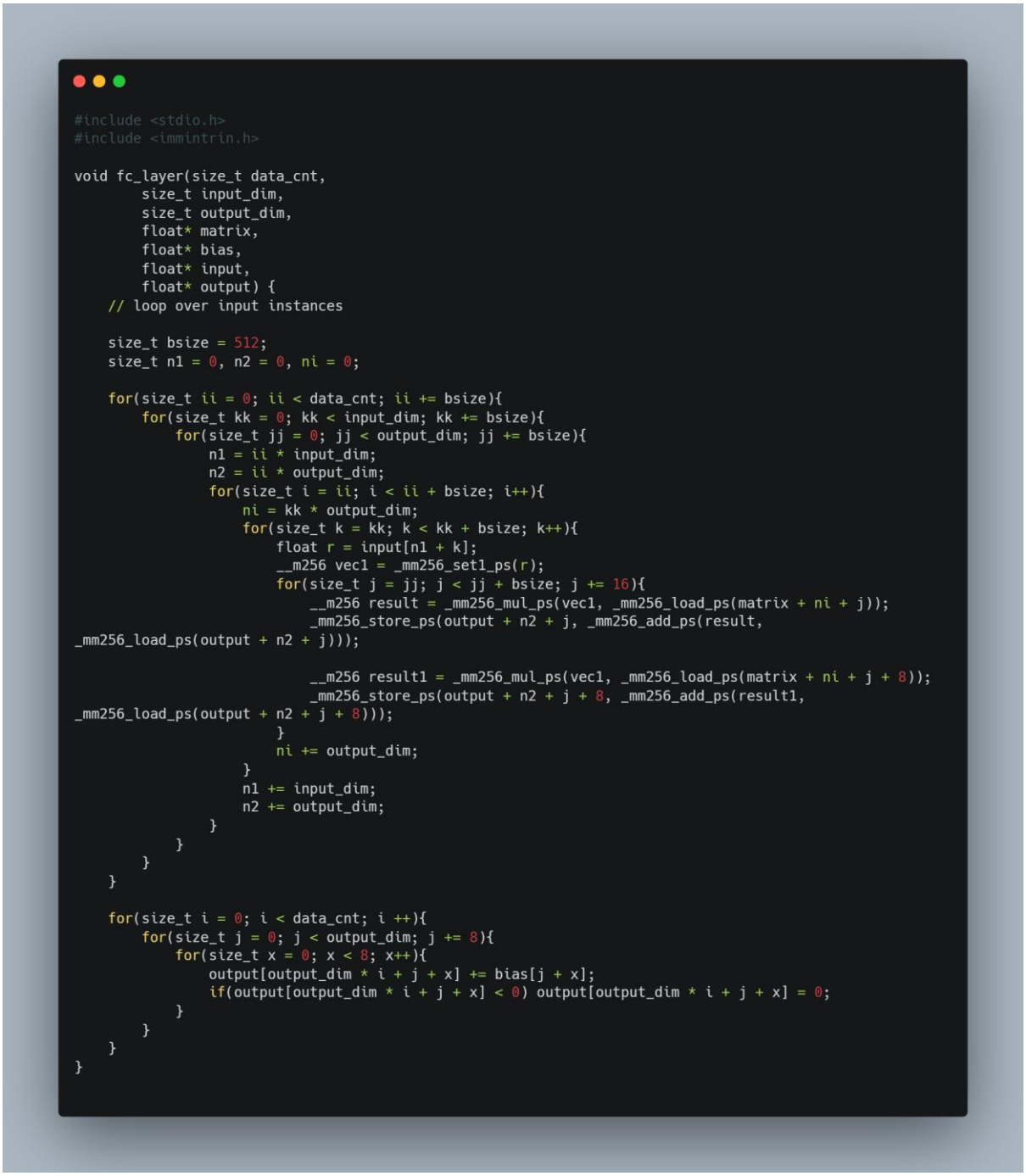
실행하면,

```
Model loaded successfuly!
Example data loaded successfuly!
Input count: 256
Starting layer...
Elapsed time: 1.321544 s
Performance (MFLOPS): 6501.000000
Average error: 0.000895
```

256 by 256 matrix들의 곱셈 계산 성능이 6500MFLOPS로 향상된 것을 확인할 수 있었다. 하지만, 여기서 우리는 간과한 것이 있는데 L1 cache size는 lscpu를 이용해서 얼마인지 알 수 있다.

```
u2022132017@workspace-eber1rx6tf2o-0:~/cs_pa4$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   43 bits physical, 48 bits virtual
CPU(s):                          128
On-line CPU(s) list:             0-127
Thread(s) per core:              2
Core(s) per socket:              32
Socket(s):                       2
NUMA node(s):                    2
Vendor ID:                       AuthenticAMD
CPU family:                      23
Model:                           49
Model name:                      AMD EPYC 7452 32-Core Processor
Stepping:                        0
Frequency boost:                 disabled
CPU MHz:                         1903.157
CPU max MHz:                     2350.0000
CPU min MHz:                     1500.0000
BogoMIPS:                        4699.88
Virtualization:                  AMD-V
L1d cache:                       2 MiB
L1i cache:                       2 MiB
L2 cache:                        32 MiB
L3 cache:                        256 MiB
```

2 MiB는 2097152 bytes이다.2097152 bytes면 4 byte float 자료형을 524288개를 저장할 수 있는 것이다. 아무튼 이러한 캐시 메모리에 있는 행렬 요소들을 최대한 손실율이 나지 않게 연산하기 위해서는 Cache Blocking을 해야한다. 그리고 3B^2 < C이기 때문에 B = 512의 블록 사이즈로 설정하여 Cache Blocking을 진행해 주었다. 아래는 그에 해당하는 코드와, 실행했을 때의 성능 스펙이다.



```
#include <stdio.h>
#include <immintrin.h>

void fc_layer(size_t data_cnt,
        size_t input_dim,
        size_t output_dim,
        float* matrix,
        float* bias,
        float* input,
        float* output) {
    // loop over input instances

    size_t bsize = 512;
    size_t n1 = 0, n2 = 0, ni = 0;

    for(size_t ii = 0; ii < data_cnt; ii += bsize){
        for(size_t kk = 0; kk < input_dim; kk += bsize){
            for(size_t jj = 0; jj < output_dim; jj += bsize){
                n1 = ii * input_dim;
                n2 = ii * output_dim;
                for(size_t i = ii; i < ii + bsize; i++){
                    ni = kk * output_dim;
                    for(size_t k = kk; k < kk + bsize; k++){
                        float r = input[n1 + k];
                        __m256 vec1 = _mm256_set1_ps(r);
                        for(size_t j = jj; j < jj + bsize; j += 16){
                            __m256 result = _mm256_mul_ps(vec1, _mm256_load_ps(matrix + ni + j));
                            _mm256_store_ps(output + n2 + j, _mm256_add_ps(result,
_mm256_load_ps(output + n2 + j)));

                            __m256 result1 = _mm256_mul_ps(vec1, _mm256_load_ps(matrix + ni + j + 8));
                            _mm256_store_ps(output + n2 + j + 8, _mm256_add_ps(result1,
_mm256_load_ps(output + n2 + j + 8)));
                        }
                        ni += output_dim;
                    }
                    n1 += input_dim;
                    n2 += output_dim;
                }
            }
        }
    }

    for(size_t i = 0; i < data_cnt; i ++){
        for(size_t j = 0; j < output_dim; j += 8){
            for(size_t x = 0; x < 8; x++){
                output[output_dim * i + j + x] += bias[j + x];
                if(output[output_dim * i + j + x] < 0) output[output_dim * i + j + x] = 0;
            }
        }
    }
}
```

```
Model loaded successfuly!
Example data loaded successfuly!
Input count: 4096
Starting layer...
Elapsed time: 8.042467 s
Performance (MFLOPS): 17093.000000
Average error: 0.000882
```

따라서 우리가 원하던 14000 MFLOPS 이상의 성능을 내는 matrix multplication program을 만들 수 있었다. 하지만...

Top 20 fastest PA4 runners.

Rank	Nickname	MFLOPS	Date
1	winner	23939	12/19/2023, 5:10:11 PM
2	plz	16466	12/19/2023, 4:47:16 PM
3	ButFastRight?	11358	12/19/2023, 10:37:22 AM
4	----------	11166	12/19/2023, 4:41:01 PM
5	(ง'-')ง	10076	12/19/2023, 7:07:22 PM
6	Akaps	3574	12/19/2023, 5:53:51 PM
7	test	2118	12/19/2023, 10:46:17 AM

TOP 3에 들어가기 위해서는 아직 한참 멀었다. 위는 스코어보드가 초기화된 후에 사람들이 제출한 코드들의 성능들이고, 스코어보드가 초기화되기 전에는 40000 MFLOPS 성능을 가지는 코드를 만든 사람도 있었다. (물론 지금과는 CPU 성능이 다르지만.)

아무튼 좀더 최적화를 해보자.

Unrolling을 더욱 진행하고, SIMD 레지스터 명령어에서 load 함수의 경우는 latency가 크기 때문에 주의하면서 load 함수를 loop 바깥으로 최대한 빼주고, for문이 긴 경우에는 branch prediction을 활용하는 등 여러가지 최적화를 진행하였고 그 결과 다음과 같은 코드가 완성되었다.

```c
#include <stdio.h>
#include <immintrin.h>

#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x)     __builtin_expect(!!(x), 0)

void fc_layer(size_t data_cnt,
        size_t input_dim,
        size_t output_dim,
        float* matrix,
        float* bias,
        float* input,
        float* output) {
   // loop over input instances

   size_t bsize = 512;
   __m256 vvec, vvec1, result, result1, result2, result3;

   for(size_t kk = 0; kk < 4096; kk += bsize){
      for(size_t ii = 0; ii < 4096; ii += bsize){
         for(size_t jj = 0; jj < 4096; jj += bsize){
            for(size_t i = ii; i < ii + bsize; i += 4){
                size_t n1 = i << 12;
                size_t n2 = kk << 12;
               for(size_t k = kk; likely(k < kk + bsize); k+=2){
                __m256 vecs[4] = {_mm256_set1_ps(input[n1 + k]), _mm256_set1_ps(input[n1 + 4096 + k]),
_mm256_set1_ps(input[n1 + 8192 + k]), _mm256_set1_ps(input[n1 + 12288 + k])};
                __m256 vecs1[4] = {_mm256_set1_ps(input[n1 + k + 1]), _mm256_set1_ps(input[n1 + 4097 +
k ]), _mm256_set1_ps(input[n1 + 8193 + k]), _mm256_set1_ps(input[n1 + 12289 + k])};
                    for(size_t j = jj; j < jj + bsize; j += 32){
                        __m256 MUL0 = _mm256_load_ps(matrix + n2 + j); //__m256 MUL0 =
_mm256_load_ps(matrix + output_dim * k + j);
                        __m256 MUL1 = _mm256_load_ps(matrix + n2 + j + 8);
                        __m256 MUL2 = _mm256_load_ps(matrix + n2 + j + 16);
                        __m256 MUL3 = _mm256_load_ps(matrix + n2 + j + 24);

                        __m256 mul0 = _mm256_load_ps(matrix + n2 + 4096 + j);
                        __m256 mul1 = _mm256_load_ps(matrix + n2 + 4104 + j);
                        __m256 mul2 = _mm256_load_ps(matrix + n2 + 4112 + j);
                        __m256 mul3 = _mm256_load_ps(matrix + n2 + 4120 + j);

                        for(size_t u = 0; (u < 4); u++){
                            vvec = vecs[u];
                            vvec1 = vecs1[u];

                            result = _mm256_fmadd_ps(vvec, MUL0, _mm256_mul_ps(vvec1, mul0));
                            _mm256_store_ps(output + ((i + u) << 12) + j, _mm256_add_ps(result,
_mm256_load_ps(output + ((i + u) << 12) + j)));

                            result1 = _mm256_fmadd_ps(vvec, MUL1, _mm256_mul_ps(vvec1, mul1));
                            _mm256_store_ps(output + ((i + u) << 12) + j + 8, _mm256_add_ps(result1,
_mm256_load_ps(output + ((i + u) << 12) + j + 8)));

                            result2 = _mm256_fmadd_ps(vvec, MUL2, _mm256_mul_ps(vvec1, mul2));
                            _mm256_store_ps(output + ((i + u) << 12) + j + 16, _mm256_add_ps(result2,
_mm256_load_ps(output + ((i + u) << 12) + j + 16)));

                            result3 = _mm256_fmadd_ps(vvec, MUL3, _mm256_mul_ps(vvec1, mul3));
                            _mm256_store_ps(output + ((i + u) << 12) + j + 24, _mm256_add_ps(result3,
_mm256_load_ps(output + ((i + u) << 12) + j + 24)));

                        }
                    }
                    n2 += 8192;
                }
            }
         }
      }
   }
    size_t nn = 0;

   for (size_t i = 0; likely(i < data_cnt); i++) {
        for (size_t j = 0; j + 16 <= output_dim; j += 16) {

            __m256 bias_vec0 = _mm256_load_ps(bias + j);
            __m256 bias_vec1 = _mm256_load_ps(bias + j + 8);

            __m256 output_vec0 = _mm256_load_ps(output + nn + j);
            __m256 output_vec1 = _mm256_load_ps(output + nn + j + 8);

            output_vec0 = _mm256_add_ps(output_vec0, bias_vec0);
            output_vec1 = _mm256_add_ps(output_vec1, bias_vec1);

            output_vec0 = _mm256_max_ps(output_vec0, _mm256_setzero_ps());
            output_vec1 = _mm256_max_ps(output_vec1, _mm256_setzero_ps());

            _mm256_store_ps(output + nn + j, output_vec0);
            _mm256_store_ps(output + nn + j + 8, output_vec1);
        }
        nn += 4096;
    }

}
```

Bias를 더하고 행렬 연산 결과 값이 음수인 경우는 결과 값을 0으로 수정해주는 과정도 loop unrolling을 통해서 하였다. 이 코드를 실행하여 성능을 조사해보면,

```
Model loaded successfuly!
Example data loaded successfuly!
Input count: 4096
Starting layer...
Elapsed time: 3.536091 s
Performance (MFLOPS): 38876.000000
Average error: 0.000835
```

매우 준수한 성능을 내는 것을 확인할 수 있다. 근데 likely 함수를 써서 branch prediction을 한 게 오히려 새로운 분기를 만들어내서 더 성능이 안좋아지는 것같아서 뺐고 결과적으로

```
Model loaded successfuly!
Example data loaded successfuly!
Input count: 4096
Starting layer...
Elapsed time: 3.446798 s
Performance (MFLOPS): 39884.000000
Average error: 0.000835
```

영혼까지 끌어모아서 최적화된 행렬 곱연산 프로그램을 구현할 수 있었다. 근데 가장 안에 있는 u loop도 그냥 4번밖에 루핑을 안하기 때문에 그냥 다 써주게 되면 for문을 돌면서 비교를 안해도 되니까 성능적으로 이득이다. 그리고 루프안에서 초기화 하는 것은 새로운 주소에 메모리를 저장하는 것이기 때문에 레이턴시가 길 수도 있으므로, 초기화를 루프 밖으로 빼준다. 그래서 최종적인 코드를 실행하게 되면,

```
Model loaded successfuly!
Example data loaded successfuly!
Input count: 4096
Starting layer...
Elapsed time: 3.123457 s
Performance (MFLOPS): 44012.000000
Average error: 0.000835
```

4000MFLOPS만큼 성능이 더 좋아졌다.