SSE Exam Solutions

1 2018 Q1

1.1 Part (b)

1.1.1 Code Segment 1

- This code cannot be vectorised using SSE.
- The code requires a dependency in both it's current and previous every iteration, The relience on the previous iteration is what's holding back the iteration, as we will not be able to access the previous updated iteration if we are computing four at a time.

1.1.2 Code Segment 2

- This code can be vectorised using SSE.
- The reason it can be is that the only dependency is the current value found in array[i]. This means that if we work on this 4 times at a time, there will not be a dependency issue there as they are mutually exclusive operations.
- Vectorising this requires us to change the starting position, since we will do four steps at a time but from the top of the array to the bottom, we must decrement by 3 at the start of the loop.

```
[a, b, c, d, 4, 5, 6, 7]
             ^^^^^^^^^^ 1st iteration
[0, 1, 2, 3, e, f, g, h]
 ^^^^^^^^^^ 2nd iteration
```

- As we know, we must also have a postloop for when the array is not divisible by 4. This post loop is the same as our initial loop given, we'll just need to keep track of the i variable so that we can finish the last elements, so the i variable will be put in scope of the two loops.
- An optimisation is to vectorise the constants outside of the loop, so that we do not recompute the same multiplier and SIZE for a vector each iteration.

```
void compute(float* array, int SIZE, float multiplier) {
    int i;
    __m128 v4multipler = _mm_set1_ps(multiplier);
    __m128 v4SIZE = _mm_set1_ps(SIZE);
    for (i = SIZE - 4; i >= 0; i-=4) {
        __m128 v4array = _mm_loadu_ps(&(array[i]));
        __m128 v4array_mul_v4multiplier = _mm_mul_ps(v4array, v4multipler);
        __m128 v4result = _mm_div_ps(v4array_mul_v4multiplier,v4SIZE);
        _mm_storeu_ps(&(array[i]), v4result);
```

```
    }
     for (/* i = i */; i >= 0; i--) {
        array[i] = (array[i] * multiplier) / SIZE;
    }
}
```

1.1.3 Code Segment 3

- This code can be vectorised using SSE.
- The assumptions needed to be made here, is that the array contains non negative integers.
- This code requires intrinsics for integers.
- Calculate max 4 at a time calculating of the array each time, at the end of the vectored loop select the maximum element horziontally out of the 4 lanes. And have a post loop to do the remaining elements not divisible by 4.

```
int compute(int* array, int SIZE) {
    // array is 16-byte aligned address
    __m128i v4max = _mm_setzero_si128(); // set all four lanes of vector to 0
    int i;
    for(i = 0; i < SIZE - 3; i+=4) {
        __m128i v4array = _mm_load_si128((__m128i*)&(array[i]));
        v4max = _mm_max_epi32(v4array, v4max);
    }
    // select the maximum value which is in one of the four lanes in v4max
    v4max = _mm_max_epi32(v4max, _mm_shuffle_epi32(v4max, _MM_SHUFFLE(0, 0, 3, 2)));
    v4max = _mm_max_epi32(v4max, _mm_shuffle_epi32(v4max, _MM_SHUFFLE(0, 0, 0, 1)));
    int max = _mm_cvtsi128_si32(v4max);
    // postloop for when the array is not divsible by 4
    for(/* i = i */; i < SIZE; i++) {
        if (array[i] > max) { max = array[i]; }
    }
    return max;
}
```

1.1.4 Code Segment 4

- Exam paper has a mistake with the second parameter with function, I will assume it is strcmp from standard C.
- This code can be vectorised using SSE.
- The code requires the use of string comparison intrinsics. We will load 16 bytes at a time because we are restricted to loading 128 bits with SSE. We do a check to see if there's a match, then we check where the match is. If there's no initial match, it'll check to see if there's a null character somewhere therefore false.
- The postloop ensures that if a string isn't divisible by 16, then it can do a postloop that was the same as our initial programme unvectorised programme.

```
bool strcmp_sse(char* string1, char* string2, int strlength) {
    int i;
    for(i = 0; i < strlength - 15; i+=16) {
        __m128i v16string1 = _mm_load_si128((__m128i*)&(string1[i]));
        __m128i v16string2 = _mm_load_si128((__m128i*)&(string2[i]));
        if (_mm_cmpistrc(v16string1,v16string2,_SIDD_UBYTE_OPS |
                                               _SIDD_CMP_EQUAL_EACH |
                                               _SIDD_NEGATIVE_POLARITY |
                                               _SIDD_LEAST_SIGNIFICANT)) {
            int index = _mm_cmpistri(v16string1,v16string2, _SIDD_UBYTE_OPS |
                                                            _SIDD_CMP_EQUAL_EACH |
                                                            _SIDD_NEGATIVE_POLARITY |
                                                            _SIDD_LEAST_SIGNIFICANT);
            if (string1[index] != string2[index])
                return false;
        } else if (_mm_cmpistrz(v16string1,v16string2,_SIDD_UBYTE_OPS |
                                                      _SIDD_CMP_EQUAL_EACH |
                                                      _SIDD_NEGATIVE_POLARITY |
                                                      _SIDD_LEAST_SIGNIFICANT)) {
            return false;
        }
    }
    // postloop
    // i = i;
    for (; i < strlength; i++) {
        if (string1[i] != string2[i])
            return false;
    }
    return true;
}
```

2 2017 Q1

2.1 Part (b)

2.1.1 Code Segment 1

- This code is vectorisable, there are no data dependencies in each iteration so using SSE to do this four steps at a time is doable.
- The approach is to break the code to two parts, the first part does the multiples of four, and the remaining elements that are not divisible by four will have a posloop such as the one given in the question.
- Load your constants before the loop to avoid reloading. These constants are PI, and ONE.
- load first four values to vector, multiply it by itself, divide by a vector of v4ONE / v4aMulv4a
- Finally add the result with v4PI, and store it to the resulting b array.
- The assumptions of this programme is that b and a are at least 1024 elements. And if b is larger the resulting array will not be written to so elements above 1024 could result in undefiend behaviour

if not set before calling function.

```
void compute(float* a, float* b) {
    const float PI = 3.14159;
    const float ONE = 1.0;
    __m128 v4PI = _mm_set1_ps(PI); // set four lanes to PI
    __m128 v4ONE = _mm_set1_ps(ONE); // set four lanes to ONE
    int i;
    for (i = 0; i < 1021; i+=4) { // i < 1024 - 3 = 1021
        __m128 v4a = _mm_loadu_ps(a+i); // load the next four location in a
        __m128 v4aMulv4a = _mm_mul_ps(v4a, v4a); // multiply v4a with itself
        __m128 v4ONE_div_v4aMulv4a = _mm_div_ps(v4ONE, v4aMulv4a); // v4ONE / (v4a * v4a)
        __m128 v4b = _mm_add_ps(v4ONE_div_v4aMulv4a,v4PI); // divAns + PI
        _mm_store_ps(b+i, v4b);
    }
    // postloop not required as 1024 % 4 == 0
}
```

2.1.2 Code Segment 2

- This code can be vectorised using SSE
- The strategy is utilizing the max intrinsic and finding the maximum of every 4 new elements of the array using vectors.
- Finally one of the lanes will have the correct maximum value out of every element, extract it using max again and a few specific shuffles and extract the first lane. This will have the maximum.
- A postloop will finish off the array if the array isn't divisible by 4 and 1 element. The reason is because we are actually starting the vectorisation at i = 1.

```
float find_max(float* array, int size) {
    __m128 v4max = _mm_load1_ps(array); // load first four values
    const int size_minus_three = size - 3;
    int i;
    for (i = 1; i < size_minus_three; i+=4) {
        __m128 v4array = _mm_loadu_ps(array+i);
        v4max = _mm_max_ps(v4array, v4max);
    }
    v4max = _mm_max_ps(v4max, _mm_shuffle_ps(v4max, v4max, _MM_SHUFFLE(0,0,3,2)));
    v4max = _mm_max_ps(v4max, _mm_shuffle_ps(v4max, v4max, _MM_SHUFFLE(0,0,0,1)));
    float max = _mm_cvtss_f32(v4max);
    // i = i;
    for (; i < size; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
    return max;
}
```

2.1.3 Code Segment 3

- This code can be vectorised.
- The approach in vectorising this code is to think vertically instead of horizonatally. Load a vector of 4 pixels ignoring the last value, since my approach does not utilize the 4th lane as a pixel.
- Add together the 4 lanes each time, in practice this will do something the likes of

```
[   0     ,      0     ,     0      ,     0     ]  // ignore last 0
[pixels[i], pixels[i+1], pixels[i+2], pixels[i+3]] // ignore pixels[i+3]
// ------------------------------------------------------ // add them together vertically
// while loop, do the same against the next pixels and the answer of the add computation.
```

- Once finished computing the vertical additions of the three rgb pixels, shuffle them in a way where we take out the first 3 pixels, red, green, and blue and then we vertically multiply them after doing the shuffle, and taking the first value using $\_mm_{cvtssf32}$, this would be our sum product of the 3 pixels.

```
float rgb_sum_product(float* pixels) {
    __m128 v4sum_pixels = _mm_setzero_ps();
    // array must be divisible by 3, and length 1011, last value of v4sum_pixels is undefined
    for (int i = 0; i < 1011; i+=3) {
        __m128 v4pixelsi = _mm_loadu_ps(pixels+i); // first three pixels only matter
        v4sum_pixels = _mm_add_ps(v4pixelsi, v4sum_pixels);
    }
    __m128 v4product_pixels = _mm_mul_ps(v4sum_pixels,
                                         _mm_shuffle_ps(v4sum_pixels,
                                                        v4sum_pixels, _MM_SHUFFLE(0, 3, 2, 1)));
    v4product_pixels = _mm_mul_ps(v4product_pixels,
                                  _mm_shuffle_ps(v4sum_pixels,
                                                 v4sum_pixels, _MM_SHUFFLE(1, 0, 3, 2)));
    return _mm_cvtss_f32(v4product_pixels);
}
```

2.1.4 Code Segment 4

- This code can be vectorised.
- Your inner for loop will do the matrix sum 4 at a time, eventually you'll reach the end of the matrix row so you need to exit inner loop. After you exit it your v4sum vector should have the 4 sums of your loop computation, you horizonatlly add them together in order to retrieve the input and you repeat it for 4096 times to get all the answers to result[i].
- The difficulty in this problem was to break the problem in a way to vertically think about the problem instead of horizontally, the lack of data dependencies allowed for this SSE approach

```
// An efficient way of summing up a vector
// More efficient than doing hadd twice and getting the
// first element
// equivalent to
/*
```

```
v4a = _mm_hadd_ps(v4a);
v4a = _mm_hadd_ps(v4a);
return _mm_cvtss_f32(v4a);
*/
static inline float hsum_ps_sse3(__m128 v4a) {
    __m128 shuf = _mm_movehdup_ps(v4a);
    __m128 sums = _mm_add_ps(v4a, shuf);
    shuf        = _mm_movehl_ps(shuf, sums);
    sums        = _mm_add_ss(sums, shuf);
    return _mm_cvtss_f32(sums);
}

void multiply(float ** restrict matrix, float * restrict vec, float * restrict result) {
    for (int i = 0; i < 4096; i++) {
        __m128 v4sum = _mm_setzero_ps();
        for (int j = 0; j < 4093; j+=4) { // i < 4096 - 3 since j+=4
            __m128 v4vecj = _mm_loadu_ps(&(vec[j]));
            __m128 v4matrixij = _mm_loadu_ps(&(matrix[i][j]));
            __m128 v4vecj_mul_v4matrixij = _mm_mul_ps(v4vecj, v4matrixij);
            v4sum = _mm_add_ps(v4sum, v4vecj_mul_v4matrixij);
        }
        result[i] = hsum_ps_sse3(v4sum); // efficient horiztonal sum vector
    }
}
```

3 2016 Q1

3.1 Part (c)

3.1.1 Code Segment 1

- This can is vectorisable
- Since there aren't any dependencies when writing to a, as we're not reading it's previous contents, we can split our work to four at a time using SSE, we using loading, multiplication, addition, and storing the four results back into a.
  - You must remember that BIMDAS rule, and you must multiply before adding, and my code below does just that.
- A postloop is not required as 1024 is divisible by 4

```
void add_scaled(float* a, float* b, float* c, float factor) {
    __m128 v4factor = _mm_set1_ps(factor);
    for (int i = 0; i < 1024; i+=4) { // postloop not needed since arrays % 4
        __m128 v4b = _mm_loadu_ps(b+i);
        __m128 v4c = _mm_loadu_ps(c+i);
        __m128 v4c_mul_v4factor = _mm_mul_ps(v4c, v4factor);
        __m128 result = _mm_add_ps(v4b, v4c_mul_v4factor);
        _mm_storeu_ps(a+i, result);
    }
}
```

3.1.2 Code Segment 2

- This code is vectorisable
- The idea behind the solution is calculating the sum vertically. if we calculate the sum by doing multiplication and adding it to itself, we can definitely just do this 4 at a time. The issue at the end is needing to sum up the 4 lanes at the end, which I do using an efficient hsum approach by utilizing SSE3. It can also be done using 2 hadd's on itself and $\_mm_{cvtssf32}$ call.
- A postloop is required since the size is not guarnateed to be divisible by 4.

```
// efficient approach for calculating horizontal sum of vector in SSE3
static inline float hsum(__m128 v4) {
    __m128 shuf = _mm_movehdup_ps(v4);
    __m128 sums = _mm_add_ps(v4, shuf);
    shuf = _mm_movehl_ps(shuf,sums);
    sums = _mm_add_ss(sums, shuf);
    return _mm_cvtss_f32(sums);
}

float dot_product(float* restrict a, float* restrict b, int size) {
    const int size_minus_three = size - 3;
    __m128 v4sum = _mm_setzero_ps();
    int i;
    for (i = 0; i < size_minus_three; i+=4) {
        __m128 v4a = _mm_loadu_ps(a+i);
        __m128 v4b = _mm_loadu_ps(b+i);
        __m128 v4a_mul_v4b = _mm_mul_ps(v4a, v4b);
        v4sum = _mm_add_ps(v4sum, v4a_mul_v4b);
    }
    float sum = hsum(v4sum);
    // postloop
    // i = i;
    for (; i < size; i++) {
        sum = sum + a[i] * b[i];
    }
    return sum;
}
```

4 2015 Q1

4.1 Part b

4.1.1 Code Segment 1

- This code is not vectorisable,
- Each iteration requires the previous iterations computation, if we were to use SSE vecotrization we would be doing computation 4 at a time and 4 out of the 3 lanes will have an incorrect value.

4.1.2 Code Segment 2

- This code nearly identically is exactly to the question from 2016 Q1 part c code segment 2
- This code is vectorisable
- Same explanation as seen in 2016 Q1 part c code segment 2 apart from the requirements of needing a postloop

```
// efficient approach for calculating horizontal sum of vector in SSE3
static inline float hsum(__m128 v4) {
    __m128 shuf = _mm_movehdup_ps(v4);
    __m128 sums = _mm_add_ps(v4, shuf);
    shuf = _mm_movehl_ps(shuf,sums);
    sums = _mm_add_ss(sums, shuf);
    return _mm_cvtss_f32(sums);
}

float dot_product(float* restrict a, float* restrict b) {
    __m128 v4sum = _mm_setzero_ps();
    int i;
    for (i = 0; i < 4096; i+=4) { // 4096 % 4, no need for postloop
        __m128 v4a = _mm_loadu_ps(a+i);
        __m128 v4b = _mm_loadu_ps(b+i);
        __m128 v4a_mul_v4b = _mm_mul_ps(v4a, v4b);
        v4sum = _mm_add_ps(v4sum, v4a_mul_v4b);
    }
    return hsum(v4sum);
}
```

4.1.3 Code Segment 3

- This code and explanation is identical to 2016 Q1 part c code segment 2017 Q1 part b code segment 3