Assignment1 Report

Student name: Yingzheng Pan Student number: 19336862

Sample sort using SSE

The first sorting algorithm I build is a sample sort using SSE. Sample sort is an algorithm where you take a sample of pivot values from an array of values. As shown in figure 1, I created a float array with size 40, using sample size 40 as an example in my code, Apply for loop to store 40 random values of a[] array with big random data into my array with size 40 as 40 samples of my sample sort. At the same time, I also set five large empty arrays in order to take a sample of 4 items to divide a[] array into five large intervals with four numbers. The other five counts are counting variables used in the algorithms of the five arrays. Also I create five interval arrays with size, which is size of the original array(a[] array).

```
void sample_sort(float a[], int size) {
  int i, j;
  int n = 40;
  float array1[size], array2[size], array3[size], array4[size], array5[size];
  int count1 = 0; int count2 = 0; int count3 = 0; int count4 = 0; int count5 = 0;
  float array[40];
  for (i=0; i < n; i++) {
      array[i] = a[i];
  }
```

Figure 1

As the figure 2 shown, then the main sorting method of my sample sort is most similar to bubble sort. Two for loop and swap functions are used. The basic algorithm is Starting from the first index, compare the first and the second elements. If the first element is greater than the second element, they are swapped. Now, compare the second and the third elements. Swap them if they are not in order. The above process goes on until the last element. I use this algorithm to sort the 40 sample values in the array which I extracted earlier from a[] array into float array[40]. And then now my 40 samples in array have been sorted from smallest to largest successfully.

After that, I randomly choose 4 items to become my four splitters, they are value in the position array[0], array[10], array[20], array[30] in my ordered array[40]. And then I use __m128 _mm_setr_ps(float a, float b, float c, float d) to Insert four values into four lanes of a vector. It take a sample of 4 items and four floats fit in an SSE register called number here.

```
for (i = 0; i < n-1; i++)
for (j = 0; j < n-i-1; j++) if (array[j] > array[j+1])
swap(&array[j], &array[j+1]);

  __m128 number = _mm_setr_ps(array[0], array[10], array[20], array[30]);
```

Figure 2

```
void swap(float *a, float *b)
{
float temp = *a;
*a = *b;
*b = temp;
}
```

As the figure 3 shown, firstly I create an array called arrayVal with size 4, which is used to store the compare result into it. Each position of the array represents a bit. Therefore size four can represent four bits. Now we use a for loop to extract all disorder values in a[] array by using __m128 _mm_set1_ps(float a) to load and set those values into SSE register called value here. And then we use __m128 _mm_cmpgt_ps(__m128 a, __m128 b) to compare corresponding lanes of pairs of vectors. We have a four lanes vector called number that I created earlier. The value of four splitters are in the _m128 number. And then we use compare greater to compare every disorder number in a[] array with _m128 number, which is to check that which one is bigger than the one of the four splitters. The disorder number in a[] array is compared to four splitters in the _m128 number, four splitters so there will be four comparison answers which correspond to four bits of arrayVal. If the _m128 value is not greater than one of the four splitters in the _m128 number, the number in that bit is zero, which means less than that splitter in the number. On the other hand, if it's greater than that, the number at that bit is not zero. Use _mm_store_ps to store compare result into my four bit array arrayVal.

Those if and else if and else statements are used to check which bit of the four-bit array is not zero, it means that it is larger than that splitter and it needs to be placed into the five intervals of the array in the corresponding order position. For example, the four splitters are divided into five intervals. if the fourth bit is not zero, that means it's greater than the fourth splitter, so it needs to be in the fifth interval, and else if it's the third bit is not zero, that means it's less than the fourth splitter but greater than the third splitter, so we're going to put it in the fourth interval. else if the second bit is not zero, it's less than the third splitter but greater than the second splitter, so we're going to put it in the third interval. else if the first bit is not zero, it's less than the second splitter but larger than the first splitter, so we should put it in the second interval. And when none of this is true and the other is if all the four bits are zero (else

statement), then it's less than the first splitter or also means less than all the splitters, then we're going to put it in the smallest interval that's the first interval. That's the logic and algorithms for this part. At the same time, I also have five counts to count how many of the values it is smaller than that part of splitter for the five partitions.

```
float arrayVal[4];
for ( i = 0; i < size; i++ ) {
  __m128 value = _mm_set1_ps(a[i]);
  __m128 comp = _mm_cmpgt_ps(value, number);
  _mm_storeu_ps(arrayVal, comp);

  if (arrayVal[3] != 0) {
      array5[count5] = a[i];
      count5++;
  }
  else if (arrayVal[2] != 0) {
      array4[count4] = a[i];
      count4++;
  }
  else if (arrayVal[1] != 0) {
      array3[count3] = a[i];
      count3++;
  }
  else if (arrayVal[0] != 0) {
      array2[count2] = a[i];
      count2++;
  }
  else {
      array1[count1] = a[i];
      count1++;
  }
}
```

Figure 3

As the figure 4 shown, we will use the sorting algorithms, which is most similar like bubble sort to sort the five intervals array. We use five sorting algorithms (for loop and swap) to sort five partitions array respectively. The basic algorithm is Starting from the first index, compare the first and the second elements. If the first element is greater than the second element, they are swapped. Now, compare the second and the third elements. Swap them if they are not in order. The above process goes on until the last element.
The first two for loop and swap are used to sort first interval array, we use count1 to represent the size of the first interval array. We use _m128 value1 to set the first element in first interval array and we use _m128 value2 to store the second element of first interval array by using __m128 _mm_set1_ps(float a) to load and set those values into SSE register. Also using _m128 comp1 to store the compare result by using __m128 _mm_cmpgt_ps(__m128 a,

__m128 b) to compare corresponding lanes of pairs of vectors whether it is greater than or not. If the first element is greater than the second element, they are swapped. Now, compare the second and the third elements. Swap them if they are not in order. The above process goes on until the last element. Use for loop to keep looping until the first interval array is successfully sorted from smallest to largest. And then The second two for loop and swap are used to sort second interval array, we use count2 to represent the size of second interval array. We use _m128 value1 to set the first element in first interval array and we use _m128 value2 to store the second element of first interval array by using __m128 _mm_set1_ps(float a) to load and set those values into SSE register. Also using _m128 comp1 to store the compare result by using __m128 _mm_cmpgt_ps(__m128 a, __m128 b) to compare corresponding lanes of pairs of vectors whether it is greater than or not. If the first element is greater than the second element, they are swapped. Now, compare the second and the third elements. Swap them if they are not in order. The above process goes on until the last element. Use for loop to keep looping until the second interval array is successfully sorted from smallest to largest. For the third, fourth and fifth interval arrays, they are the same principle. Now we've successfully managed to arrange each of the five interval arrays individually from smallest to largest by using the sample sorting algorithms which is most similar like bubble sort.

```
for (i = 0; i < count1-1; i++) {
for (j = 0; j < count1-i-1; j++) {
    __m128 value1 = _mm_set1_ps(array1[j]);
    __m128 value2 = _mm_set1_ps(array1[j+1]);
    __m128 comp1 = _mm_cmpgt_ps(value1, value2);
    _mm_storeu_ps(arrayVal, comp1);
    if (arrayVal[0] != 0) {
        swap(&array1[j], &array1[j+1]);
    }
}
}
for (i = 0; i < count2-1; i++) {
for (j = 0; j < count2-i-1; j++) {
    __m128 value1 = _mm_set1_ps(array2[j]);
    __m128 value2 = _mm_set1_ps(array2[j+1]);
    __m128 comp1 = _mm_cmpgt_ps(value1, value2);
    _mm_storeu_ps(arrayVal, comp1);
    if (arrayVal[0] != 0) {
        swap(&array2[j], &array2[j+1]);
    }
}
}
for (i = 0; i < count3-1; i++) {
for (j = 0; j < count3-i-1; j++) {
    __m128 value1 = _mm_set1_ps(array3[j]);
    __m128 value2 = _mm_set1_ps(array3[j+1]);
    __m128 comp1 = _mm_cmpgt_ps(value1, value2);
    _mm_storeu_ps(arrayVal, comp1);
    if (arrayVal[0] != 0) {
        swap(&array3[j], &array3[j+1]);
    }
}
}
```

```
for (i = 0; i < count4-1; i++) {
for (j = 0; j < count4-i-1; j++) {
    __m128 value1 = _mm_set1_ps(array4[j]);
    __m128 value2 = _mm_set1_ps(array4[j+1]);
    __m128 comp1 = _mm_cmpgt_ps(value1, value2);
    _mm_storeu_ps(arrayVal, comp1);
    if (arrayVal[0] != 0) {
        swap(&array4[j], &array4[j+1]);
    }
}
}
for (i = 0; i < count5-1; i++) {
for (j = 0; j < count5-i-1; j++) {
    __m128 value1 = _mm_set1_ps(array5[j]);
    __m128 value2 = _mm_set1_ps(array5[j+1]);
    __m128 comp1 = _mm_cmpgt_ps(value1, value2);
    _mm_storeu_ps(arrayVal, comp1);
    if (arrayVal[0] != 0) {
        swap(&array5[j], &array5[j+1]);
    }
}
}
```

Figure 4

As the figure 5 shown, the last step of the sample sort is we need to put the five interval arrays into our a[] array in order to output the results. We use five for loops to put five intervals of array into a[] array, first, then second, third, fourth, and finally fifth. These five interval arrays are placed into a[] array in order and in the correct corresponding positions. And when we put the interval array in the correct position starting with the first smallest array, the next thing we need to notice is the size of the interval array each time we put an array in, When we put down an array you need to add the previous size, Because we're going to start dropping an interval array after the last element of the array so we have an array in the condition for loop The range of the size of the array needs to be added to the sum of the original array size so as to be the correct size range. In addition, the starting value of index should also be set to the total value of the space occupied by the previous array after the last element of the previously set array is released.
That's the whole idea of sample sort sorting algorithms.

```
  for (i = 0; i<count1; i++) {
       a[i] = array1[i];
  }
  for (i = count1; i<count1+count2; i++) {
       a[i] = array2[i-count1];
  }
  for (i = count1+count2; i<count1+count2+count3; i++) {
       a[i] = array3[i-count1-count2];
  }
  for (i = count1+count2+count3; i<count1+count2+count3+count4; i++) {
       a[i] = array4[i-count1-count2-count3];
  }
  for (i = count1+count2+count3+count4; i<count1+count2+count3+count4+count5; i++) {
       a[i] = array5[i-count1-count2-count3-count4];
  }
}
```

Figure 5

As the figure 6, the next thing is that we compare the sample sort algorithm's time to the simple sort (david_sort) contained in the source file. After many tests and computation, by presenting times for my sample sort algorithm compared to the simple sort, the algorithm time used by the simple sort is significantly shorter than the sample sort I wrote using SSE. The sort time of simple sort is approximately 33445 microseconds, however the sort time of my sample sort is approximately 13762715 microseconds. Also, the cycle of the simple sort is also much smaller than the sample sort I wrote. The cycle of simple sort is 80255736 cycles. However, the cycle of my sample sort is 33029635698 cycles.

```
void student_sort(float a[], int size) {
  david_sort(a, size);
//  sample_sort(a, size);
//CocktailSort(a, size);

}

void student_sort(float a[], int size) {
//  david_sort(a, size);
  sample_sort(a, size);
//CocktailSort(a, size);

}
```

Figure 6

```
panyi@macneill:~/lab1$ gcc -o sort -O3 -march=native  sort-harness.c student_sor
t.c
panyi@macneill:~/lab1$ ./sort -r 100000
Sorting time: 33445 microseconds, 80255736 cycles
```

david_sort figure

```
panyi@macneill:~/lab1$ gcc -o sort -O3 -march=native  sort-harness.c student_sor
t.c
panyi@macneill:~/lab1$ ./sort -r 100000
Sorting time: 13762715 microseconds, 33029635698 cycles
```

sample_sort figure

Cocktail Shaker sort using SSE

Cocktail sort is a variation of bubble sort. The bubble sorting algorithm always traverses the elements from the left and moves the largest element to the correct position on the first iteration, the second largest element to the correct position on the second iteration, and so on. The cocktail sort traverses the given array alternately in both directions. Cocktail sorting does not go through unnecessary iterations, so it works well for large arrays.

As the figure 7 shown, at first of cocktail shaker sort I create some integer lists at beginning. All integer arrays are both have the two indexes, the reason for this is for cocktail shaker sort each iteration of the algorithm is divided into two phases. The first stage traverses the groups from left to right, like a bubble sort. The first stage loops through the array from left to right, just like the Bubble Sort. During the loop, adjacent items are compared and if the value on the left is greater than the value on the right, then values are swapped. Starting from the first index(second element), compare the second and the first elements. If the first element is greater than the second element, they are swapped. Now, compare the third and the second elements. Swap them if they are not in order. The above process goes on until the last element. At the end of the first iteration, the largest number will reside at the end of the array. The second phase loops through the array in the opposite direction - starting with the item before the most recently sorted item, then moving back to the beginning of the array. Here, adjacent items are also compared and swapped as needed. So basically it's going to sort twice, first in the forward direction and then sort, and then in the reverse order.

For int array start [2] is equal to {1, count 1}, which is meaning we extract starting from the first index(second element), compare the second and the first elements(index-1). If the first element is greater than the second element, they are swapped. Now, compare the third and the second elements. Swap them if they are not in order. The above process goes on until the last element. As I mention above, there are two sorts, one in the forward direction, and the second in the reverse direction from the last element. Therefore all the integer arrays are size two. Start[2], the first index is one, the reason is we need to first use our first index to get the second element and then i-1 to get the first element, and then compare them, The process goes on until the last element. The second position is count-1, the reason is for the second sort, we need to sort in the reverse direction, therefore, we need to start from the last element in the array. end[2], the first position is count, the reason is for the

first sort process in the forward direction, the end point is count (last one). The second position is zero, the reason is for the second sore in reverse direction, our end point is zero index, because we will start from the last element and end at the first element. Inc[2], the first position is one, the reason is for the first sort process in forward direction, we need to add our index by one each time. The second position is -1, the reason is for the second sort process in reverse direction, we need to decrease one each time, because we start from the last index and then end at the first element. Also, I create an array called arrayCock to record the compare result into it.

```
while (1)
{
  char flag;
  int start[2] = { 1, count - 1 };
  int end[2] = { count, 0 };
  int inc[2] = { 1, -1 };
  float arrayCock[4];
```

Figure 7

As the figure 8 shown, I have two for loop in my Cocktail Shaker sort, the first for loop is to use for looping two times for sorting element in array, which are the first time is the forward direction and the second direction is the reverse direction. And then the second for loop is to use sort in forward and reverse direction. For the first process is the forward direction, starting from the second element, i=1, and then i-1=0, which is the first element, we use _m128 data1 = _mm_set1_ps(data[i-1]) to store the first element and then _m128 data2 = _mm_set1_ps(data[i]) to store the second element, and then we use _m128 compare = _mm_cmpgt_ps(data1, data2) to compare the first element and the second element which one is greater, and then store the compare result in arrayCock by using _mm_storeu_ps(). During the loop, adjacent items are compared and if the value on the left is greater than the value on the right, then values are swapped. So on so fourth, we use this until we get to the last element. The above process goes on until the last element, then this is the process of the first sort in forward direction. For the second process is the reverse direction, starting from the last element, which index is count-1, and then end at the first element, which index is 0. And then for our index we keep decrease one each time until it reaches the first element. The second stage loops through the array in opposite direction- starting from the item just before the most recently sorted item, and moving back to the start of the array. Here also, adjacent items are compared and are swapped if required. That’s the basic and main idea for Cocktail Shaker sort.

```
    for (int it = 0; it < 2; ++it)
    {
      flag = 1;

      for (int i = start[it]; i != end[it]; i += inc[it])
      {
          __m128 data1= _mm_set1_ps(data[i - 1]);
          __m128 data2= _mm_set1_ps(data[i]);
          __m128 compare = _mm_cmpgt_ps(data1, data2);
          _mm_storeu_ps(arrayCock, compare);
          if (arrayCock[0] != 0) {
          swap(data + i - 1, data + i);
          flag = 0;
      }
      }
      if (flag)
        return;
    }
  }
}
```

Figure 8

As the figure 9 shown, the next thing is that we compare the Cocktail Shaker sort algorithm's time to the simple sort (david_sort) contained in the source file. After many tests and computation, by presenting times for my Cocktail Shaker sort algorithm compared to the simple sort, the algorithm time used by the simple sort is significantly shorter than the Cocktail Shaker sort I wrote using SSE. The sorting time of simple sort is approximately 34741 microseconds. However the sorting time of Cocktail Shaker sort is approximately 39711537 microseconds. Also, the cycle of the simple sort is also much smaller than the Cocktail Shaker sort I wrote. The cycle of simple sort is approximately 83383984 cycles. However the cycle of the Cocktail Shaker sort is approximately 95305106488 cycles.

```
void student_sort(float a[], int size) {
 david_sort(a, size);
//  sample_sort(a, size);
//CocktailSort(a, size);

}

void student_sort(float a[], int size) {
 // david_sort(a, size);
//  sample_sort(a, size);
  CocktailSort(a, size);

}
```

Figure 9

```
panyi@macneill:~/lab1$ gcc -o sort -O3 -march=native  sort-harness.c student_sor
t.c
panyi@macneill:~/lab1$ ./sort -r 100000
Sorting time: 34741 microseconds, 83383984 cycles
```

David_sort figure

```
panyi@macneill:~/lab1$ gcc -o sort -O3 -march=native  sort-harness.c student_sor
t.c
panyi@macneill:~/lab1$ ./sort -r 100000
Sorting time: 39711537 microseconds, 95305106488 cycles
```

Cocktail shaker sort figure