CS 6850 - Assignment - Multiple levels of parallelism
Brad Peterson - Weber State University

Overall goal: Efficiently implement three three levels of parallelism in one program:

1) Multiple threads to utilize core parallelism. Instead of using C++11 std:thread, use OpenMP forking.

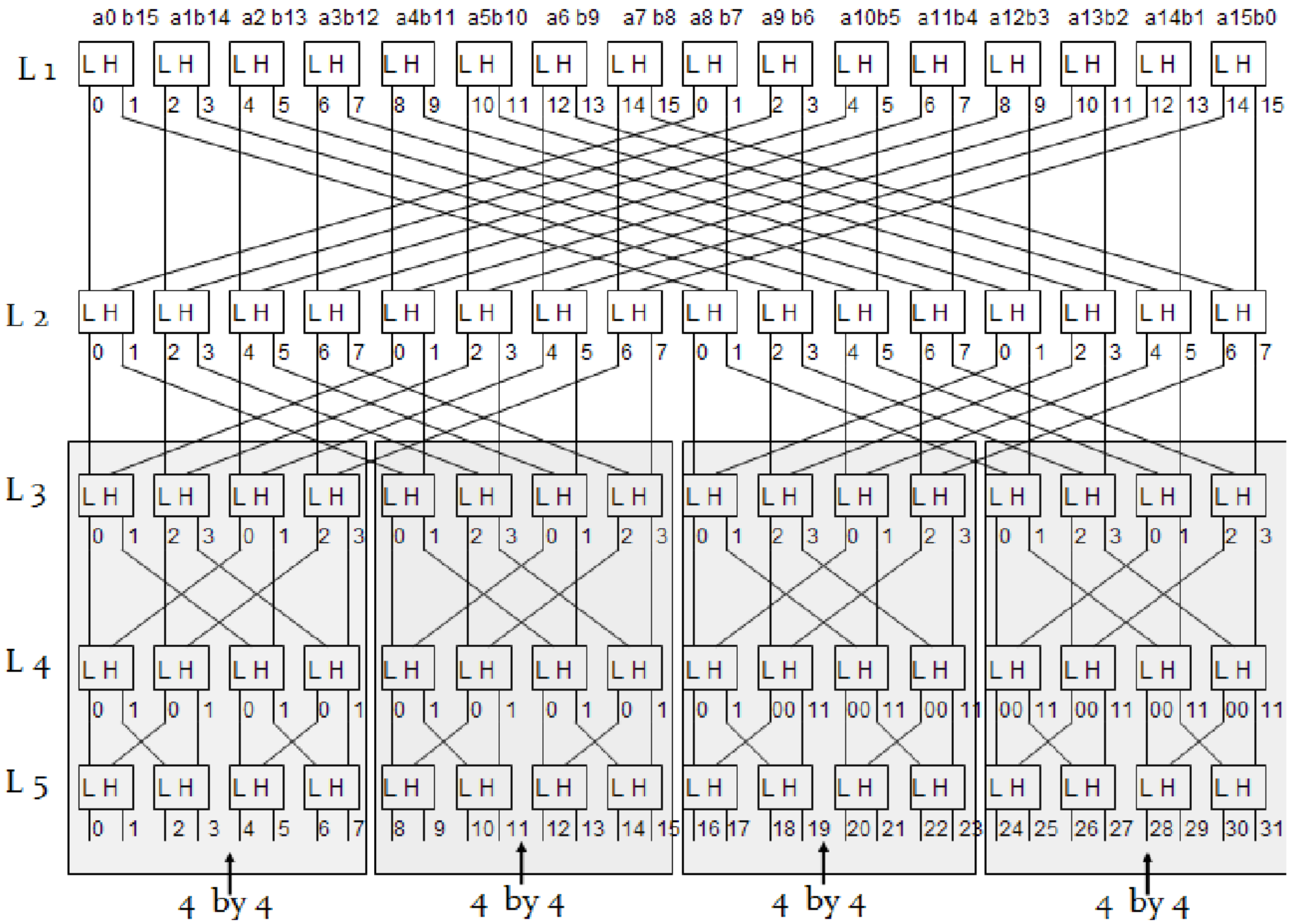
2) SIMD intrinsics to utilize vector parallelism.

3) Interleaving sets of independent instructions to utilize pipeline parallelism

Secondary goals:

- Utilize existing published papers for algorithms
- Know how to get Intel intrinsics to compile
- Sort a large number of randomized 32-bit floats

The core of this assignment is a bitonic merge sort network which works great with vectorization. This assignment is based on an existing paper, “Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture” by Chhugani et al., which utilized 128-bit vectors with SSE instructions. We will expand the same concept to utilize 512-bit vectors. The core idea comes from Figure 5 of the paper:


a0 b15 a1b14 a2 b13 a3b12 a4b11 a5b10 a6 b9 a7 b8 a8 b7 a9 b6 a10b5 a11b4 a12b3 a13b2 a14b1 a15b0
L 1
L 2
L 3
L 4
L 5
L H
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3
0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 00 11 00 11 00 11 00 11 00 11 00 11 00 11 00 11
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
4 by 4
4 by 4
4 by 4
4 by 4

This assignment will utilize a few Intel intrinsics. First, a 2 input permute operation: _mm512_permutex2var_ps:

```
__m512 _mm512_permutex2var_ps (__m512 a, __m512i idx, __m512 b)
```
vpermi2ps,...

Synopsis

```
__m512 _mm512_permutex2var_ps (__m512 a, __m512i idx, __m512 b)
#include <immintrin.h>
Instruction: vpermi2ps zmm, zmm, zmm
             vpermt2ps zmm, zmm, zmm
CPUID Flags: AVX512F
```

Description

Shuffle single-precision (32-bit) floating-point elements in a and b across lanes using the corresponding selector and index in idx, and store the results in dst.

Operation

```
FOR j := 0 to 15
        i := j*32
        off := idx[i+3:i]*32
        dst[i+31:i] := idx[i+4] ? b[off+31:off] : a[off+31:off]
ENDFOR
dst[MAX:512] := 0
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	3	1
Skylake	3	1

While the idx vector uses the right 4 bits to determine the source index, the 5th bit is used to determine between input *a* or input *b*. Even better, the math works out so that you can conceptually consider input *a* to own indexes 0 through 15, and input *b* to own index 16 through 31.

The intrinsic _mm512_set_epi32, will help us create the *idx* value. Note that you must enter in the values ***reversed***.

```
__m512i _mm512_set_epi32 (int e15, int e14, int e13, int
e12, int e11, int e10, int e9, int e8, int e7, int e6, int
e5, int e4, int e3, int e2, int e1, int e0)
```

Synopsis

```
__m512i _mm512_set_epi32 (int e15, int e14, int e13, int e12, int e11,
int e10, int e9, int e8, int e7, int e6, int e5, int e4, int e3, int
e2, int e1, int e0)
#include <immintrin.h>
Instruction: Seqeuence
CPUID Flags: AVX512F
```

Description

Set packed 32-bit integers in dst with the supplied values.

Operation

```
dst[31:0] := e0
dst[63:32] := e1
dst[95:64] := e2
dst[127:96] := e3
dst[159:128] := e4
dst[191:160] := e5
dst[223:192] := e6
dst[255:224] := e7
dst[287:256] := e8
dst[319:288] := e9
dst[351:320] := e10
dst[383:352] := e11
dst[415:384] := e12
dst[447:416] := e13
dst[479:448] := e14
dst[511:480] := e15
dst[MAX:512] := 0
```

Also, we need the ability to reverse a vector of 16 values with _mm512_permutexvar_ps:

`__m512 _mm512_permutexvar_ps (__m512i idx, __m512 a)` vpermps

Synopsis

```
__m512 _mm512_permutexvar_ps (__m512i idx, __m512 a)
#include <immintrin.h>
Instruction: vpermps zmm, zmm, zmm
CPUID Flags: AVX512F
```

Description

Shuffle single-precision (32-bit) floating-point elements in a across lanes using the corresponding index in idx.

Operation

```
FOR j := 0 to 15
        i := j*32
        id := idx[i+3:i]*32
        dst[i+31:i] := a[id+31:id]
ENDFOR
dst[MAX:512] := 0
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	3	1
Skylake	3	1

The idx should be `15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00.`
Again recall that when you create the idx value, you will want to pass in these reversed, so you will call `outputvector = _mm512_permutexvar_ps(_mm512_set_epi32(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15), inputVector);`.

Finally, the min and max instrinsics, _mm512_min_ps and _mm512_max_ps (min shown below):

`__m512 _mm512_min_ps (__m512 a, __m512 b)` vminps

Synopsis

```
__m512 _mm512_min_ps (__m512 a, __m512 b)
#include <immintrin.h>
Instruction: vminps zmm, zmm, zmm
CPUID Flags: AVX512F
```

Description

Compare packed single-precision (32-bit) floating-point elements in a and b, and store packed minimum values in dst.

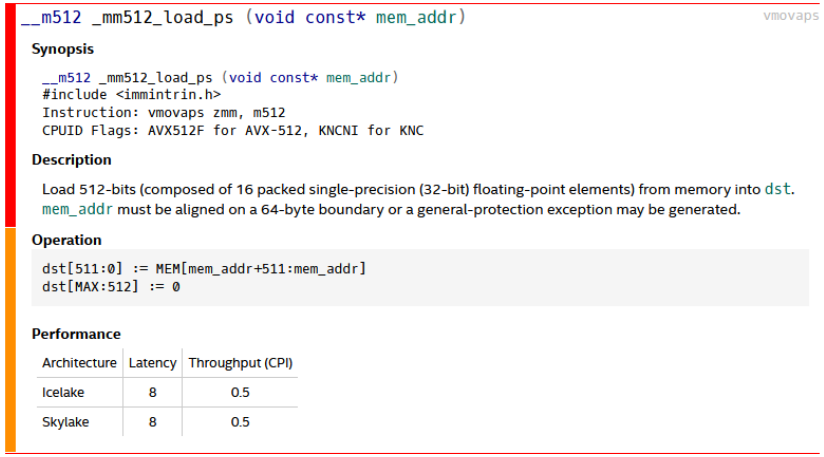
Operation

```
FOR j := 0 to 15
        i := j*32
        dst[i+31:i] := MIN(a[i+31:i], b[i+31:i])
ENDFOR
dst[MAX:512] := 0
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	4	1
Skylake	4	0.5

You will also need the load and store intrinsics: _mm512_load_ps and _mm512_store_ps



`__m512 _mm512_load_ps (void const* mem_addr)` vmovaps

Synopsis

```
__m512 _mm512_load_ps (void const* mem_addr)
#include <immintrin.h>
Instruction: vmovaps zmm, m512
CPUID Flags: AVX512F for AVX-512, KNCNI for KNC
```

Description

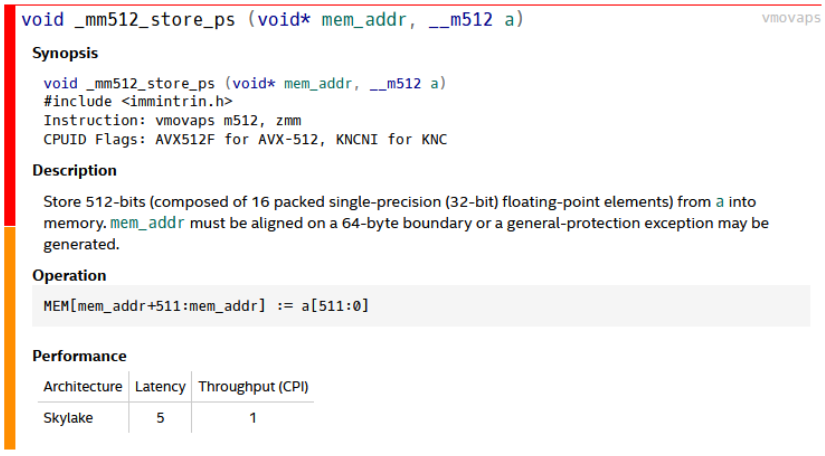
Load 512-bits (composed of 16 packed single-precision (32-bit) floating-point elements) from memory into `dst`. `mem_addr` must be aligned on a 64-byte boundary or a general-protection exception may be generated.

Operation

```
dst[511:0] := MEM[mem_addr+511:mem_addr]
dst[MAX:512] := 0
```

Performance

Architecture	Latency	Throughput (CPI)
Icelake	8	0.5
Skylake	8	0.5



`void _mm512_store_ps (void* mem_addr, __m512 a)` vmovaps

Synopsis

```
void _mm512_store_ps (void* mem_addr, __m512 a)
#include <immintrin.h>
Instruction: vmovaps m512, zmm
CPUID Flags: AVX512F for AVX-512, KNCNI for KNC
```

Description

Store 512-bits (composed of 16 packed single-precision (32-bit) floating-point elements) from `a` into memory. `mem_addr` must be aligned on a 64-byte boundary or a general-protection exception may be generated.

Operation

```
MEM[mem_addr+511:mem_addr] := a[511:0]
```

Performance

Architecture	Latency	Throughput (CPI)
Skylake	5	1

With these tools identified, we can implement the 32-wide bitonic sorter. (If you need a reminder of compiling a program with intrinsics, see the sample code section at the end of the homework)

I strongly advise that you practice implementing the vectorization/instrinsic portion of the homework first, then the ILP portion second, then the multi-core last.

Vectorization: First phase - Creating blocks of 16 sorted elements

Use a simple single threaded sorting algorithm to sort each block of 16 unsorted elements into a block of sorted elements. Because n=16 is such a small number, you can use a $O(n^2)$ algorithm like Bubble Sort or Selection Sort if you want.

As an example, consider the following 64 values.

```
26 61 29 47 67 28 49 35 95 99 09 20 43 45 42 42 04 56 33 72 00 70 50 04 06 68 98 43 64 47 76 48
03 60 91 42 55 37 22 40 26 55 37 45 37 74 88 00 45 54 13 06 80 61 63 91 86 51 66 22 38 26 84 44
```

Sort each block of 16.

```
09 20 26 28 29 35 42 42 43 45 47 49 61 67 95 99 00 04 04 06 33 43 47 48 50 56 64 68 70 72 76 98
00 03 22 26 37 37 37 40 42 45 55 55 60 74 88 91 06 13 22 26 38 44 45 51 54 61 63 66 80 84 86 91
```

Next phase - Merging blocks of 16 sorted elements into blocks of 32 sorted elements

L1:

Now according to the 32-wide bitonic sorter figure, the algorithm needs 16 values for *a* (*a0* through *a15*) and 16 values for *b* (*b0* through *b15*). The first 16 values are taken from above and are *a*. The next 16 values are *b*. These *b* values are reverse to make upcoming operations easier.

```
(a0-a15) 09 20 26 28 29 35 42 42 43 45 47 49 61 67 95 99
(b15-b0) 98 76 72 70 68 64 56 50 48 47 43 33 06 04 04 00
```

Load *a0-a15* and *b15-b0* into zmm register variables.

Use _mm512_min_ps and _mm512_max_ps to generate the L and H as new register variables. The example's result is as follows:

```
min (L): 09 20 26 28 29 35 42 42 43 45 43 33 06 04 04 00
max (H): 98 76 72 70 68 64 56 50 48 47 47 49 61 67 95 99
```

Now according to the figure, we are at the bottom of L1 and ready to feed results into the inputs of the L2 boxes. The input into L2'S L is constructed by taking the first 8 values from min and the last 8 values from max. H is constructed by taking the second 8 values from min and the first 8 values from max.

The intrinsic _mm512_permutex2var is the tool to use to obtain values from the prior L and prior H. Input *a* will be min (L), input *b* will be max (H). From min (L), it must take from indexes in this order: `00 01 02 03 04 05 06 07`, and then it must take from max's `00 01 02 03 04 05 06 07`. Note that to choose the max indexes, the 5th bit must be set to 1. Putting a 5th bit as 1 effectively adds 16 to an integer. And so the idx must be constructed as:

```
00 01 02 03 04 05 06 07 16+0 16+1 16+2 16+3 16+4 16+5 16+6 16+7
```

Or simplified:

```
00 01 02 03 04 05 06 07 16 17 18 19 20 21 22 23
```

The inputs and output for _mm512_permutex2var are thus:

```
a (current L):    09 20 26 28 29 35 42 42 43 45 43 33 06 04 04 00
b (current H):    98 76 72 70 68 64 56 50 48 47 47 49 61 67 95 99
idx:              00 01 02 03 04 05 06 07 16 17 18 19 20 21 22 23
new output(L):    09 20 26 28 29 35 42 42 98 76 72 70 68 64 56 50
```

To create the second output in L2, the process is similar. The first output is created by taking the indexes in the following order: `08 09 10 11 12 13 14 15`, first from min and then from max the same indexes. The indexes from max (input *b*) again need to be offset by 16 (to set that 5th bit)

The inputs and output for _mm512_permutexvar are thus:

```
a (current L):    09 20 26 28 29 35 42 42 43 45 43 33 06 04 04 00
b (current H):    98 76 72 70 68 64 56 50 48 47 47 49 61 67 95 99
idx:              08 09 10 11 12 13 14 15 24 25 26 27 28 29 30 31
new output(H):    43 45 43 33 06 04 04 00 48 47 47 49 61 67 95 99
```

L2 to L3:

Perform a min and max on the two outputs from the last step:

```
first output:  09 20 26 28 29 35 42 42 98 76 72 70 68 64 56 50
second output: 43 45 43 33 06 04 04 00 48 47 47 49 61 67 95 99
min (L):       09 20 26 28 06 04 04 00 48 47 47 49 61 64 56 50
max (H):       43 45 43 33 29 35 42 42 98 76 72 70 68 67 95 99
```

Now according to the figure, a different set of permuting is needed.

The first output (L) is composed from both the min and the max. From min, it must take from indexes in this order: `00 01 02 03`, then it must take from max's `00 01 02 03`, then it must take from min's `08 09 10 11`, then it must take from max's `08 09 10 11`.

Thus:

```
a:             09 20 26 28 06 04 04 00 48 47 47 49 61 64 56 50
b:             43 45 43 33 29 35 42 42 98 76 72 70 68 67 95 99
idx:           00 01 02 03 16 17 18 19 08 09 10 11 24 25 26 27
first output:  09 20 26 28 43 45 43 33 48 47 47 49 98 76 72 70
```

The second output (H) follows a similar process:

```
a:             09 20 26 28 06 04 04 00 48 47 47 49 61 64 56 50
b:             43 45 43 33 29 35 42 42 98 76 72 70 68 67 95 99
idx:           04 05 06 07 20 21 22 23 12 13 14 15 28 29 30 31
second output: 06 04 04 00 29 35 42 42 61 64 56 50 68 67 95 99
```

L3 to L4:

Perform a min and max on the two outputs from the last step:

```
first output:  09 20 26 28 43 45 43 33 48 47 47 49 98 76 72 70
second output: 06 04 04 00 29 35 42 42 61 64 56 50 68 67 95 99
min (L):       06 04 04 00 29 35 42 33 48 47 47 49 68 67 72 70
max (H):       09 20 26 28 43 45 43 42 61 64 56 50 98 76 95 99
```

Now according to the figure, a different set of permuting is needed.

The first output (L) is composed from both the min and the max. From min, it must take from indexes in this order: `00 01`, then it must take from max's `01 01`, then min's `04 05`, then max's `04 05`, then min's `08 09`, then max's `08 09`, then min's `12 13`, then max's `12 13`.
Thus:

```
a:             06 04 04 00 29 35 42 33 48 47 47 49 68 67 72 70
b:             09 20 26 28 43 45 43 42 61 64 56 50 98 76 95 99
idx:           00 01 16 17 04 05 20 21 08 09 24 25 12 13 28 29
first output:  06 04 09 20 29 35 43 45 48 47 61 64 68 67 98 76
```

The second output (H) follows a similar process:

```
a:             06 04 04 00 29 35 42 33 48 47 47 49 68 67 72 70
b:             09 20 26 28 43 45 43 42 61 64 56 50 98 76 95 99
idx:           02 03 18 19 06 07 22 23 10 11 26 27 14 15 30 31
second output: 04 00 26 28 42 33 43 42 47 49 56 50 72 70 95 99
```

L4 to L5:

Perform a min and max on the two outputs from the last step

```
first output:  06 04 09 20 29 35 43 45 48 47 61 64 68 67 98 76
second output: 04 00 26 28 42 33 43 42 47 49 56 50 72 70 95 99
min (L):       04 00 09 20 29 33 43 42 47 47 56 50 68 67 95 76
max (H):       06 04 26 28 42 35 43 45 48 49 61 64 72 70 98 99
```

Now according to the figure, a different set of permuting is needed.

The first output (L) is composed from both the min and the max.
Thus:

```
a:             04 00 09 20 29 33 43 42 47 47 56 50 68 67 95 76
b:             06 04 26 28 42 35 43 45 48 49 61 64 72 70 98 99
idx:           00 16 02 18 04 20 06 22 08 24 10 26 12 28 14 30
first output:  04 06 09 26 29 42 43 43 47 48 56 61 68 72 95 98
```

The second output (H) follows a similar process:

```
a:             04 00 09 20 29 33 43 42 47 47 56 50 68 67 95 76
b:             06 04 26 28 42 35 43 45 48 49 61 64 72 70 98 99
idx:           01 17 03 19 05 21 07 23 09 25 11 27 13 29 15 31
second output: 00 04 20 28 33 35 42 45 47 49 50 64 67 70 76 99
```

L5:

A min and max is needed

```
first output:  04 06 09 26 29 42 43 43 47 48 56 61 68 72 95 98
second output: 00 04 20 28 33 35 42 45 47 49 50 64 67 70 76 99
min (L):       00 04 09 26 29 35 42 43 47 48 50 61 67 70 76 98
max (H):       04 06 20 28 33 42 43 45 47 49 56 64 68 72 95 99
```

At this point, we need to put the values back together.
The first sorted output is from both the min and the max.

Thus:

```
a:             00 04 09 26 29 35 42 43 47 48 50 61 67 70 76 98
b:             04 06 20 28 33 42 43 45 47 49 56 64 68 72 95 99
idx:           00 16 01 17 02 18 03 19 04 20 05 21 06 22 07 23
first output:  00 04 04 06 09 20 26 28 29 33 35 42 42 43 43 45
```

The second output (H) follows a similar process:

```
a:             00 04 09 26 29 35 42 43 47 48 50 61 67 70 76 98
```

```
b:             04 06 20 28 33 42 43 45 47 49 56 64 68 72 95 99
idx:           08 24 09 25 10 26 11 27 12 28 13 29 14 30 15 31
second output: 47 47 48 49 50 56 61 64 67 68 70 72 76 95 98 99
```

Thus the first output and second output combined are the 32 numbers merged together! At this point, the 32 values should be written back out to an output array.

Repeating L1 through L5:

Recall that the example was to sort the 64 numbers. So far, this example has only sorted the first 32. Suppose that algorithm proceeds on the next group of 32 values. The result will be:

```
00 03 06 13 22 22 26 26 37 37 37 38 40 42 44 45 45 51 54 55 55 60 61 63 66 74 80 84 86 88 91 91
```

When written out to memory, the new output will contain:

```
00 04 04 06 09 20 26 28 29 33 35 42 42 43 43 45 47 47 48 49 50 56 61 64 67 68 70 72 76 95 98 99
00 03 06 13 22 22 26 26 37 37 37 38 40 42 44 45 45 51 54 55 55 60 61 63 66 74 80 84 86 88 91 91
```

Had our example used *n* elements, where *n* is some multiple of 32, this process would continue for all blocks of 16 sorted elements so they all become blocks of 32 sorted elements.

Next phase - Merging blocks of 32 sorted elements into blocks of 64 sorted elements

The next step is merging these sorted blocks of 32 elements together into sorted blocks of 64 elements. As described previously, the approach is to reuse 32-wide bitonic sort. This process is done by taking the first 16 element block (indexes 0 through 15) and the second 16 element block (indexes 32 through 47)

```
Input 1:       00 04 04 06 09 20 26 28 29 33 35 42 42 43 43 45
Input 2:       00 03 06 13 22 22 26 26 37 37 37 38 40 42 44 45
```

Reverse input 2 to properly feed it in.

```
Input 1:       00 04 04 06 09 20 26 28 29 33 35 42 42 43 43 45
Input 2:       45 44 42 40 38 37 37 37 26 26 22 22 13 06 03 00
```

Upon running it through the bitonic sorter, the outputs will be:

```
First output:  00 00 03 04 04 06 06 09 13 20 22 22 26 26 26 28
Second output: 29 33 35 37 37 37 38 40 42 42 42 43 43 44 45 45
```

Store the first output to memory. The second output is then used as an input back into the bitonic sorter. The second input must be either indexes 16 through 31 or indexes 48 through 63. Since the data index 16 is 47 and the data at index 48 is 45, the algorithm chooses index index 48, as the 45 < 47.

```
Input 1:       29 33 35 37 37 37 38 40 42 42 42 43 43 44 45 45
Input 2:       45 51 54 55 55 60 61 63 66 74 80 84 86 88 91 91
```

Reverse input 2 to properly feed it in.

```
Input 1:       29 33 35 37 37 37 38 40 42 42 42 43 43 44 45 45
Input 2:       91 91 88 86 84 80 74 66 63 61 60 55 55 54 51 45
```

Upon running it through the bitonic sorter, the outputs will be:

```
First output:  29 33 35 37 37 37 38 40 42 42 42 43 43 44 45 45
Second output: 45 51 54 55 55 60 61 63 66 74 80 84 86 88 91 91
```

Store the first output to memory. The second output is then used as an input back into the bitonic sorter. The second input has only one choice left, the data at index 16 with value is 47.

```
Input 1:       45 51 54 55 55 60 61 63 66 74 80 84 86 88 91 91
Input 2:       47 47 48 49 50 56 61 64 67 68 70 72 76 95 98 99
```

Reverse input 2 to properly feed it in.

```
Input 1:       45 51 54 55 55 60 61 63 66 74 80 84 86 88 91 91
Input 2:       99 98 95 76 72 70 68 67 64 61 56 50 49 48 47 47
```

Upon running it through the bitonic sorter, the outputs will be:

```
First output:  45 47 47 48 49 50 51 54 55 55 56 60 61 61 63 64
Second output: 66 67 68 70 72 74 76 80 84 86 88 91 91 95 98 99
```

Store the first output to memory. As there are no more possible inputs, also store the second value to memory.

Thus, merging two blocks of 32 sorted values took 3 rounds of the 32-wide bitonic sorter. Repeat this process for all upcoming 32 sorted blocks to make 64 sorted blocks.

Vectorization: Merging blocks of 64 sorted elements into blocks of 128 sorted elements

Similar as before, work with blocks of 16 at a time, one starting at relative index 0, the other starting at relative index 64. Overall, this will take 7 rounds of the 32-wide bitonic sorter.

Vectorization: Finishing

Continue this pattern, forming sorted blocks of 256, 512, 1024, etc.

Instruction Level Parallelism: 4 Independent Instructions Per Thread

The _mm512_permutex2var cannot proceed until the _mm512_min_ps and _mm512_max_ps complete, as there are data dependencies. This dependency problem can limit a core's pipeline to 1-way. To exploit any potential 2-way or 4-way pipelining, your homework should interleave 4 mins, 4 maxes, the first 4 _mm512_permutex2var, and the second 4 _mm512_permutex2var.

For example, suppose the initial phase has been completed, where the array now formed into blocks of 16 sorted elements. Instead of merely performing a min and max on the first 2 blocks of 16 and then the permutes, the code instead should perform these in groups of 4 (which are labeled A, B, C, and D for clarity):

A is 0-31, B is 32-63, C is 64-95, D is 96-127.
A's inputs a vector from indexes 0-15 and a vector from 16-31 (inputFirstA inputSecondA)
B's inputs a vector from indexes 32-47 and a vector from 48-63 (inputFirstB inputSecondB)
C's inputs a vector from indexes 64-79 and a vector from 80-95 (inputFirstC inputSecondC)
D's inputs a vector from indexes 96-111 and a vector from 112-127 (inputFirstD inputSecondD)

inputSecondA = reverse inputSecondA
inputSecondB = reverse inputSecondB
inputSecondC = reverse inputSecondC

inputSecondD = reverse inputSecondD

(A bitonic merge level)
minA = min on inputFirstA, inputSecondA
minB = min on inputFirstB, inputSecondB

minC = min on inputFirstC, inputSecondD
minD = min on inputFirstD, inputSecondD
maxA = max on inputFirstA, inputSecondA
maxB = max on inputFirstB, inputSecondB
maxC = max on inputFirstC, inputSecondC
maxD = max on inputFirstD, inputSecondD
firstOutputA = first permuteex2var on minA and maxA
firstOutputB = first permuteex2var on minB and maxB
firstOutputC = first permuteex2var on minC and maxC
firstOutputD = first permuteex2var on minD and maxD
secondOutputA = second permuteex2var on block minA and maxA
secondOutputB = second permuteex2var on block minB and maxB
secondOutputC = second permuteex2var on block minC and maxC
secondOutputD = second permuteex2var on block minD and maxD
(Repeat for each level)

When L5 is done, the result of the four firstOutputX are written to memory. The result of the secondOutputX are either written out to memory, or used as an input for another 32-wide bitonic sort. When it is used as the input, the code should:

Determine which of the two candidate second inputs of A is the second input
Determine which of the two candidate second inputs of B is the second input
Determine which of the two candidate second inputs of C is the second input
Determine which of the two candidate second inputs of D is the second input

Repeat the 32-wide bitonic sort (starting with the reverse calls)

Multi-core: Adding Multiple Threads

Utilize OpenMP's dynamic scheduling model. Each unit of work should consist of 2^{16} = 65,536 elements. (L1 data cache is 32 KB per core or 16 KB per hyperthread. L2 cache is 1024 KB per core or 512 KB per hyperthread. This assignment is targeting L2 cache. Since 2^{16} floats require are 256K of memory, that will fit comfortably into L2 cache while leaving more than enough room for other program variables.)

Each thread should determine an independent 65,536 element region in the array in which to work. A thread starts with the initialization phase by ensuring the array has blocks of 16 sorted elements. Then using ILP, have the thread work with 4 sets of pairs of 16-value blocks. Thus, this first round should obtain 4 sets * 2 blocks in a pair * 16 values per block = 128 elements at a time per thread. After the thread has processed 128 elements into 4 sorted blocks of 32 elements, move to the next 128 array elements and repeat into 4 sorted blocks of 32 elements. Continue processing groups of 128 elements until all 65536 elements have been processed. At this point, the work is done and let OpenMP's dynamic scheduling determine if any remaining blocks of 65536 elements remain.

The next round obtains 256 elements at a time to create 4 blocks of sorted 64 elements. The next round obtains 512 elements with 4 blocks of sorted 128 elements. This continues until it completes the round which sorts 4 blocks of 16384 elements. At this point, the thread is done with this work unit, and should go back to the thread pool to look for more work, if any still exists.

The assignment itself should use 64 threads on the Linux server. The homework should use at a minimum 2^{25} values to average at least 8 work units per thread.

Concluding the Homework

Your code should finish with a simple checker, ensuring that each block of 16384 elements in the output array is indeed sorted. You will not sort the entire array!

Had the assignment continued to finish the sort (*which it will not*) here the program would move to a new phase, where all threads cooperate to merge together these blocks of 16384 elements, with each thread merging its own independent region. Further, the merging process could again utilize the 32-wide bitonic sort. The paper mentioned at the beginning of this assignment has details into this process.

Prepare a very brief report detailing the wall time overall. Compare to a normal single threaded merge sort. (You can ask a classmate for any number they have computed.)

Sample code

Using intrinsics:

```
#include <cstdlib>
#include "immintrin.h"
#include <chrono>
#include <cstdio>
int main() {
  int max = 10000000;
  int *a = (int*)aligned_alloc(64, sizeof(int) * max);
  int *output = (int*)aligned_alloc(64, sizeof(int) * 16);
  for (int i = 0, j = max - 1; i < max; i++, j--) {
        a[i] = j;
  }
  auto t1 = std::chrono::high_resolution_clock::now();
  __m512i first = _mm512_load_si512(&a[0]); // Go get the first vector
  for (int i = 16; i < max; i += 16) {
        __m512i second = _mm512_load_si512(&a[i]); // Get the next vector
        first = _mm512_max_epi32(first, second);
  }
  _mm512_store_si512(&output[0], first);

  auto t2 = std::chrono::high_resolution_clock::now();
  std::chrono::duration<double, std::milli> fp_ms = t2 - t1;
  for (int i = 0; i < 16; i++) {
        printf("%d ", output[i]);
  }
  printf("\n");
  printf("It took %g milliseconds\n", fp_ms.count());
  t1 = std::chrono::high_resolution_clock::now();
  unsigned int maxIndex = 0;
  for (int i = 0; i < max; i++) {
        if (a[i] > a[maxIndex]) {
        maxIndex = i;
        }
  }
  t2 = std::chrono::high_resolution_clock::now();
  fp_ms = t2 - t1;
  printf("The max was %d at index %u\n", a[maxIndex], maxIndex);
  printf("It took %g milliseconds\n", fp_ms.count());
  free(a);
  free(output);
  return 0;
}
```

Compile with: g++ -mavx512f -mavx2 vector_avx_512_max.cc -o vector_avx_512_max.x

Code to help print an __m512 vector:

```
void printVectorInt(__m512i v, string name)
{
#if defined (__GNUC__)
        int* temp = (int*)aligned_alloc(64, sizeof(int) * 16);
#elif defined (_MSC_VER)
        int* temp = (int*)_aligned_malloc(sizeof(int) * 16, 64);
#endif

        _mm512_store_si512(temp, v);
        printf("The vector called %s contains: ", name.c_str());
        for (int i = 0; i < 16; i++)
        {
        printf("%02d ", temp[i]);
        }
        printf("\n");

#if defined (__GNUC__)
        free(temp);
#elif defined (_MSC_VER)
        _aligned_free(temp);
#endif
}


void printVectorFloat(__m512 v, string name)
{
#if defined (__GNUC__)
        float* temp = (float*)aligned_alloc(64, sizeof(float) * 16);
#elif defined (_MSC_VER)
        float* temp = (float*)_aligned_malloc(sizeof(float) * 16, 64);
#endif

        _mm512_store_ps(temp, v);
        printf("The vector called %s contains: ", name.c_str());
        for (int i = 0; i < 16; i++)
        {
        printf("%3f ", temp[i]);
        }
        printf("\n");

#if defined (__GNUC__)
        free(temp);
#elif defined (_MSC_VER)
        _aligned_free(temp);
#endif
}
```

OpenMP Dynamic Scheduling:

OpenMP Example of Dynamic Scheduling:

```
#include <unistd.h>
#include <cstdlib>
#include <omp.h>
#include <cstdio>
#define N 33'554'432 // 2^25

int main() {
  #pragma omp parallel for schedule(dynamic) num_threads(64)
  for (unsigned int i = 0; i < N; i += 65536) {
   sleep(1);
   printf("Thread %d is ready to work within range [%d, %d).\n", omp_get_thread_num( ), i, (i + 65536));
 }
 return 0;
}
```

Pseudocode structure

I recommend creating a bitonicSort function that accepts 16 arguments (8 inputs and 8 outputs). Further, I recommend the bitonicSort() have no built-in indexing logic. In other words, a clean approach is to make the bitonicSort accept 16 __m512 registers. All 16 are passed in by reference, the latter 8 can be const. To process this initially, I strongly recommend you start programming this *without* ILP. In my naming convention, anything referring to "A" is the first lane of ILP (such as `startA1, startA2, endA1, endA2, A1in, A2in, A1out, A2out`). The "B", "C", and "D" variables are for the 2-4 lanes of ILP. Thus, ignore all B-D pseudocode initially and just get your code working for A. Then when it works, go back in and add all the B, C, and D logic.

```
main() {

  // openMP thread dynamic scheduler with for loop {
  for () {
    // This thread is now responsible for doing all work to sort the 65536 elements into 4 sorted chunks of 16384 elements each.
    int startIndex = ;         // determine the start index of this block of 65536 elements
    int endIndex = ;           // determine the end index of this block of 65536 elements

    int sortedBlockSize = 16;  // Start with sorting 16 blocks into 32
    int endingSortedBlockSize = 16384;
    Create __m512 vectors for 8 inputs and 8 outputs (I called mine A1in, A2in, B1in, . . . , D2in and A1out, A2out, B1out, . . .,  D2out)
    // Sort on sets of 16 (this also has a nice effect of loading all values into L2 cache)
    Create an input pointer at the 0th item of your 65536 elements.
    Create an output pointer.  Allocate an array of 65536 elements.
    while (sortedBlockSize <= endingSortedBlockSize) {
      // sortedBlockSize starts at 16 and is doubled every while loop iteration until it reaches 16384.

      for (int arrIndex = startIndex; arrIndex  < endIndex; arrIndex += sortedBlockSize * 8) {
        // Suppose the arrSize is 2048, sortedBlockSize is 16, and ILP is 4.
        // Then on the first round, this thread must work with a 128 element block (each ILP doing a 32 element block).  After
        // the 128 element block is complete, this loop should advance to the next 128 element block, and so on.
```

```
      Compute 8 starting indexes (startA1, startA2, startB1, startB2, startC1, startC2, startD1, startD2)
      Also compute the ending indexes (endA1, endA2, . . . , endD2). These will help us see when this inner while is done
      // You can use both sortedBlockSize and arrIndex to easily compute these.  For example, if arrIndex = 256 and sortedBlockSize is 32,
      // then startA1 = 256, startA2 = 288, startB1 = 320, startB2 = 352, startC1 = 384, . . ., startD2 = 480.
      // Also endA1 = 256 + 32 = 288, endA2 = 320, . . . , endD2 = 512.

      Create 4 indexes to track where to write back out to the array (writeA, writeB, . . ., writeD).  These are initialized to startA1, startB1, . . ., startD1.
      Load 8 vectors (A1in, A2in, B1in, B2in, . . . , D2in) using your start indexes from the input array.
      Increment the 8 start indexes by 16, as you have just read 16 values from them.

      for (int j = 0; j < (sortedBlockSize  / 8) - 1; j++) {
        // This part should loop 1 time when sortedBlockSize = 16, 3 times when sortedBlockSize = 32, 7 times when sortedBlockSize = 64, 15 times when sortedBlockSize = 128, and so on.

        Call the bitonicSort() function, passing in the 8 outputs and the 8 inputs

        Store the relevant 4 outputs back to the output array (A1out, B1out, C1out, D1out) using the write indexes.
        Increment the 4 write indexes by 16 each.

        // Determine the other input.  For example, whatever value is smaller at indexes startA1+16 or startA2+16 determines the next input.
        if (j == (sortedBlockSize  / 8) - 2) {
          // This j loop is on its last iteration.
          Write to the output array A2out, B2out, C2out, D2out
          Increment the 4 write indexes by 16 each.
        }
        else {
          Copy the other 4 outputs into the inputs so they can be reused (A2out into A1in, . . . , D2out into D1in)
          if (startA1 == endA1) {
            // 1's side has been used up, so just use 2's side
            Load the A2in vector from the input array index startA2
            startA2 += 16
          } else if (startA2 == endA2) {
            // 2's side has been used up, so just use 1's side
            Load the A2in vector from the input array index startA1
            startA1 += 16
          } else if the value at startA1 is less than the value at startA2
            // use A1's value
            Load the A2in vector from the input array index startA1
            startA1 += 16
          } else {
            // use A2's value
            Load the A2in vector from the input array index startA2
            startA2 += 16
          }
          // Repeat this if/else if/else if/else code for B, C, and D.  This part unfortuntaely doesn't use ILP.
        }
      }
      }
      sortedBlockSize *= 2;
      exchange input and output pointers, so that your output becomes the new input, and the input is now the space for your new output.
    }
    Deallocate the output array.
  } // End OpenMP for loop
} // End main
```

Provide merging two 64s into 128 using the pseudocode above. Apply the code to a concrete example:

Suppose thread 0 receives the block to work with elements [0, 65536)

Create 16 instrinic vectors (A1in, A2in, B1in, B2in, C1in, C2in, D1in, D2in, A1out, A2out, B1out, B2out, C1out, C2out, D1out, D2out)
startIndex is 0, endIndex is 65536.
endingBlockSize is 16384.

Suppose the while loop has iterated twice already (sortedBlockSize = 16 and 32), and now sortedBlockSize is now 64.

```
While loop is true {
  For loop is true, as startIndex (0) is less than endIndex (65536)
    startA1 = 0, startA2 = 64, startB1 = 128, startB2 = 192, startC1 = 256, startC2 = 320, startD1 = 384, startD2 = 448
    endA1 = 64, endA2 = 128, endB1 = 192, endB2 = 256, endC1 = 320, endC2 = 384, endD1 = 448, endD2 = 512
    writeA = 0, writeB = 128, writeC = 256, writeD = 384
    A1in = load in vector starting at a[startA1]
    A2in = load in vector starting at a[startA2]
    …
    D2in = load in vector starting at a[startD2]
    Increment startA1 through startD2 by 16 each.  So startA1 becomes 16, startA2 becomes 80, etc.
    for (int j = 0; j < (sortedBlockSize  / 8) - 1; j++) { (this for loop will loop 7 times, with j = 0 through j = 6.)
      Call bitonic sorter. (Pass in A1in through D2in, and A1out through D2out.  All 16 by reference.  The former 8 as constant)
      Store A1out to memory starting at output[writeA1], B1out to memory starting at output [writeA2], . . . , D1out to memory starting at output [writeD1]
      Increment the four write indexes by 16 (so they become 16, 144, 272, and 400)
      Copy the other 4 outputs into the inputs so they can be reused (A2out into A1in, . . . , D2out into D1in)
      if ( this will evaluate to false when j == 0)  {
      } else {
        Copy A2out to A1in, B2out to B1in, . . . , D2out to D1 in, as needed
        Determine how to load A2in.  Increment startA1 or startA2 accordingly.
      }
      (if statement repeats for B, C, and D)
    }

     For this example, the j loop will iterate again with j == 1
    {
      The prior j = 0 loop iteration loaded in A1in, A2in, . . . , D1in, D2in.  Thus the next step is…
      Call bitonic sorter. (Pass in A1in through D2in, and A1out through D2out.  All 16 by reference.  The 8 “in” vectors as const)
      Store A1out to memory starting at output[writeA1], B1out to memory starting at output [writeA2], . . . , D1out to memory starting at output [writeD1]
      Increment the four write indexes by 16 (so they become 16, 144, 272, and 400)
      Copy the other 4 outputs into the inputs so they can be reused (A2out into A1in, . . . , D2out into D1in)
      if ( this will evaluate to false when j = 0)  {
      } else {
        Copy A2out to A1in, B2out to B1in, . . . , D2out to D1 in, as needed
        Determine how to load A2in.  Increment startA1 or startA2 accordingly.
      }
      (if statement repeats for B, C, and D)
    }
    The j loop will iterate again for j == 2, j == 3, . . . , j == 6.  At j == 6, the last if statement will be true and another set of indexes written out.
  }
}
```