ID : 800962452 **Meghana Gudaram** **HPC Assignment 2**

Details of Cache from Likwid-Topology :

CPU type: Unknown Intel Processor
****
Hardware Thread Topology
****
Sockets: 2
Cores per socket: 8
Threads per core: 1
-- **//number of threads and the processor they are allocated with**

HWThread	Thread	Core	Socket
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	0	2	0
5	0	2	1
6	0	3	0
7	0	3	1
8	0	4	0
9	0	4	1
10	0	5	0
11	0	5	1
12	0	6	0
13	0	6	1
14	0	7	0
15	0	7	1

-- **//implying 16 threads are present**
Socket 0: (0 2 4 6 8 10 12 14)
Socket 1: (1 3 5 7 9 11 13 15)

****
Cache Topology
****
Level: 1 **//L1 cache is of size 32 KB, 8 such L1 are present, shared by 2 threads each**
Size: 32 kB
Cache groups: (0 2) (4 6) (8 10) (12 14) (1 3) (5 7) (9 11) (13 15)

Level: 2 **//L2 cache is of size 256 KB, 8 such L2 are present, shared by 2 threads each**
Size: 256 kB
Cache groups: (0 2) (4 6) (8 10) (12 14) (1 3) (5 7) (9 11) (13 15)

Level: 3 **//L3 cache is of size 20 MB, 2 such L3 are present, shared by 8 threads each**
Size: 20 MB
Cache groups: (0 2 4 6 8 10 12 14) (1 3 5 7 9 11 13 15)
****
NUMA Topology
****
NUMA domains: 2

Domain 0:
Processors: 0 2 4 6 8 10 12 14
Relative distance to nodes: 10 21
Memory: 62697.1 MB free of total 65314.3 MB

Domain 1:
Processors: 1 3 5 7 9 11 13 15
Relative distance to nodes: 21 10
Memory: 63326.9 MB free of total 65536 MB

From intel website, following is the information regarding DDR of Intel® Xeon® Processor E5-2667 v3 :

- Maximum BandWidth from figure is 68 GB/s,

Example : my laptop has 2 DDR3 of 1600 MHz, 64 bit bus size. Maximum BW = (2*1600 * 64) / 8 = 25.6 GB/s
Similar way if try to understand assuming this processor has 4 2133 MHz DDR4 for maximum BW, (4 * 2133 * 64) /8 = 68 GB/s (not sure, just based on analysis)

- Maximum Memory of DRAM is ***768 GB*** (again this is from the website, so assuming true)

Memory Specifications

Max Memory Size (dependent on memory type) ?	768 GB	Memory Types ?	DDR4 1600/1866/2133
Max # of Memory Channels ?	4	Max Memory Bandwidth ?	68 GB/s
Physical Address Extensions ?	46-bit	ECC Memory Supported ‡ ?	Yes

1) Highest level of data Cache is L3 of **20 MB** (there are two such), shared by each processor (all cores of a processor)
2) 8 L2 data cache of size **256 KB** each, shared by two threads each (16 such threads)
3) 8 L1 data cache of size **32 KB** each, shared by two threads each (16 such threads)

2) 1kB = 1000 bytes and 1KB = 1024 through out calculations

Submitting job to run on 16 cores results as follows (example taken for read 1000 bytes of data)

```
Time taken for 100 operations of 256000 bytes : 0.004169 BW = 6.140912 GB/s
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004176 BW = 6.129653 GB/s
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004172 BW = 6.136174 GB/s
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004168 BW = 6.142001 GB/s
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004170 BW = 6.139340 GB/s
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004168 BW = 6.141520 GB/s
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004173 BW = 6.135022 GB/s
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004171 BW = 6.137012 GB/s
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004167 BW = 6.143682 GB/s
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004174 BW = 6.133015 GB/s
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004158 BW = 6.156178 GB/s
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004168 BW = 6.142466 GB/s
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004171 BW = 6.137508 GB/s
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004166 BW = 6.144759 GB/s
 Time taken for 100 operations of 256000 bytes : 0.004169 BW = 6.140139 GB/s
value 64000.000000
value 64000.000000
 Time taken for 100 operations of 256000 bytes : 0.004170 BW = 6.139457 GB/s
value 64000.000000
```

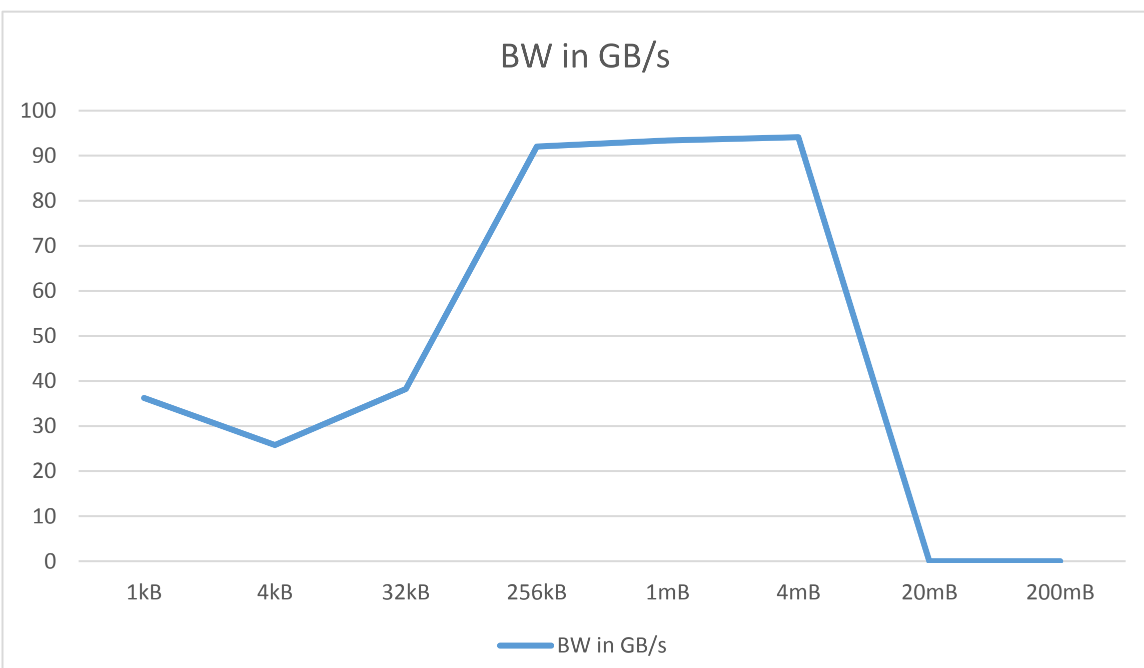
Value printed beside is after the horizontal adds since 256000 bytes, Value printed is 64000

Vector would like 8 floats of 64000

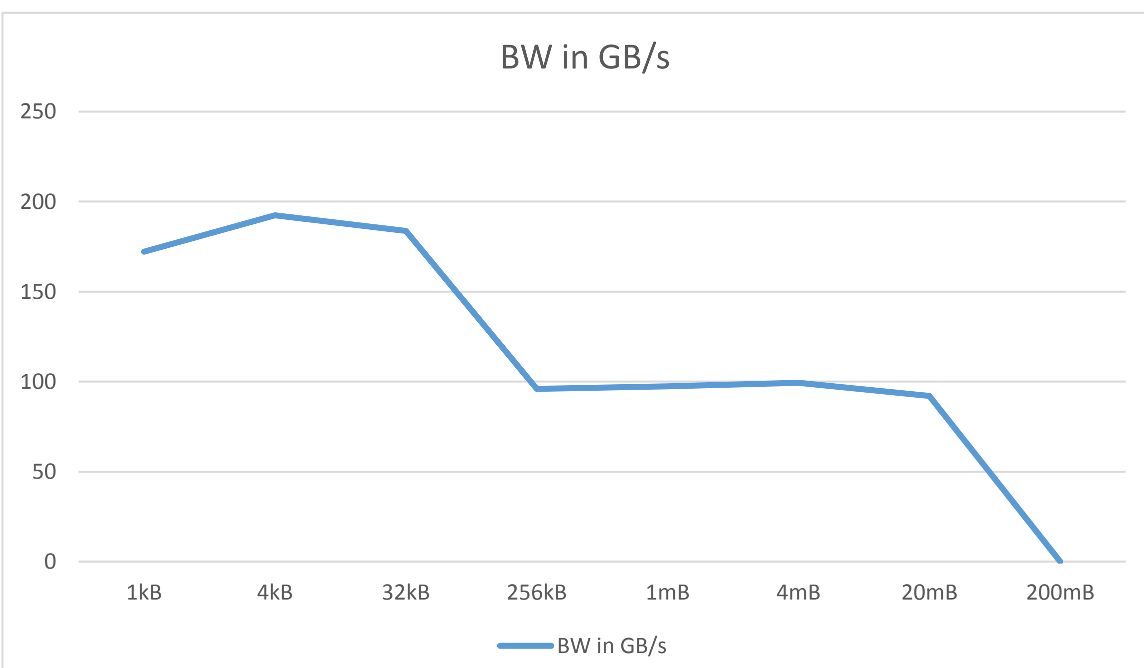
Adding BW by all 16 cores results around 98.5

Values for Read are not as expected, instead of continuous decrease, we see an increase inbetween, code is below.

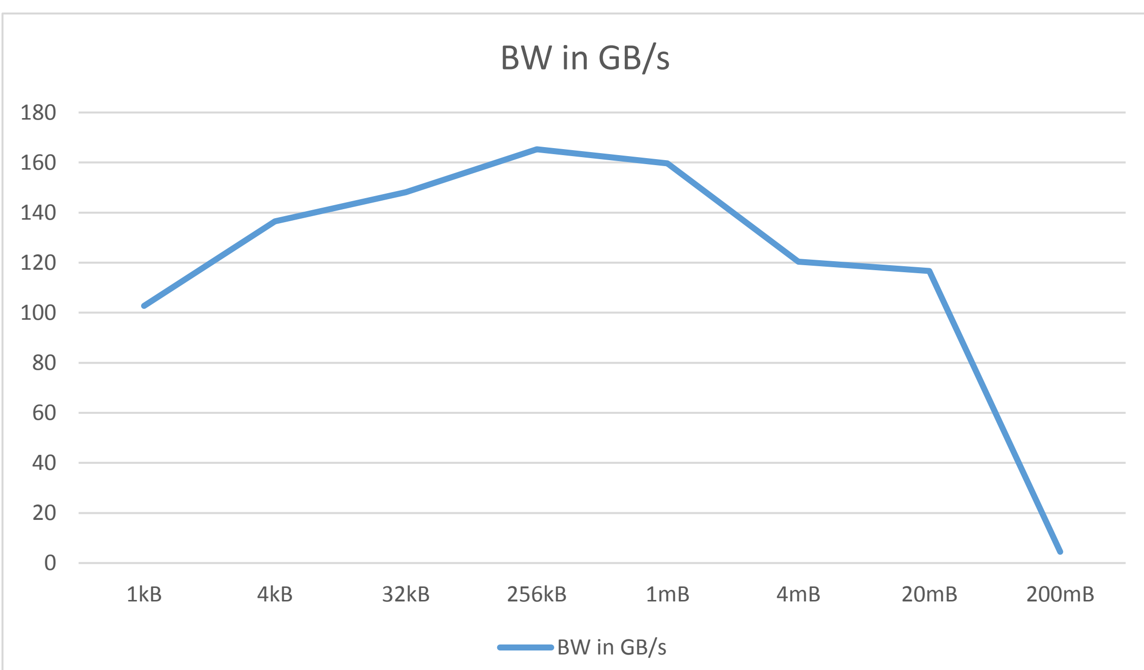
```
#include<stdio.h>
#include<omp.h>
#include<immintrin.h>
#include<iostream>
#include<chrono>
#include<ctime>

using namespace std;
using namespace std::chrono;

__m256 avx_memoryRead(float* array, int size)
{
  __m256 sum = _mm256_set1_ps(0);

  for (int i = 0; i < (size); i=i+8)
  {
        sum= _mm256_add_ps(_mm256_loadu_ps(&array[i]),sum); // Adding elements to sum, impying read from memory
  }
        sum=_mm256_hadd_ps(sum,sum); // Horizonatal addition of vector values i.e. vect1=vect0+vect0
        sum=_mm256_hadd_ps(sum,sum); // Horizonatal addition of vector values i.e. vect2 =vect1+vect1
        sum=_mm256_hadd_ps(sum,sum); // Horizonatal addition of vector values i.e. vect3=vect2+vect2

return sum;
}

int main()
{

  #pragma omp parallel
  {
    int size=262144; // 256KB is taken as size
    float *array = (float*)malloc(sizeof(float) * size); // Allocating memory in heap
    for(int j=0;j<size;j++) // Initialize array to 1
    array[j]=1;
    double bw=0;
    __m256 val= _mm256_set1_ps(0);

    high_resolution_clock::time_point t1 = high_resolution_clock::now();
    for(int i=0;i<100;i++)
    {
       val=avx_memoryRead(array,size); // function call to make sure loops are not optimized
    }
       high_resolution_clock::time_point t2 = high_resolution_clock::now();
       duration<double> time_span = duration_cast<duration<double>> (t2 - t1);
       bw = (size*4*100)/(1000000000 * time_span.count()); // Bandwidth is size in bytes / time in sec i.e. scaled to GB/s
       printf(" Time taken for 100 operations of %d bytes : %lf BW = %lf GB/s\n", size*4, time_span.count(), bw);
       printf("value %f\n",val[0]);

  }

return 0;
}
```



```
#include<stdio.h>
#include<omp.h>
#include<immintrin.h>
#include<iostream>
#include<chrono>
#include<ctime>

using namespace std;
using namespace std::chrono;

__m256 avx_memoryWrite(float* array, int size)
{
    __m256 sum = _mm256_set_ps(0,0,0,0,0,0,0,0);

    for (int i = 0; i < size; i=i+8)
    {
        _mm256_store_ps(&array[i],sum); // strore a value implies writing into memory
    }

    return sum;
}
int main()
{
#pragma omp parallel
{

    static int size=1000000; // size of 4000000 bytes and not 40MB
    float *array=(float*)malloc(sizeof(float) * size);
    for(int j=0;j<size;j++)
    array[j]=1;
    float bw=0;
    __m256 val= _mm256_set1_ps(0);

    high_resolution_clock::time_point t1 = high_resolution_clock::now();

    for(int i=0;i<100;i++)
    {
                val=avx_memoryWrite(array,size); // function call to make sure code is not optimized
    }
    high_resolution_clock::time_point t2 = high_resolution_clock::now();
    duration<double> time_span = duration_cast<duration<double>> (t2 - t1);

    bw = (size*4*100)/(1000000000 * time_span.count());
    printf(" Time taken for 100 operations of %d bytes : %lf BW = %lf GB/s\n",size*4,time_span.count(), bw);
    printf("value %f\n",val[0]);

}

    return 0;
}
```



```cpp
#include<stdio.h>
#include <omp.h>
#include <immintrin.h>
#include <iostream>
#include<chrono>
#include<ctime>

using namespace std;
using namespace std::chrono;

__m256 avx_memoryRead(float* array, int size)
{
        int j=0;
        __m256 sum = _mm256_set1_ps(0);

        for (int i = 0; i < size ; i=i+8)
        {
                __m256 vect = _mm256_load_ps(&array[i]); // load and store implies read and write operations
        _mm256_store_ps((float*)&vect,sum);
        }

return sum;
}
int main()
{
#pragma omp parallel
        {
                int size=1000000; // Size is 4000000 bytes and not 40MB
                float *array=(float*)malloc(sizeof(float) * size);
                for(int j=0;j<size;j++)
                        array[j]=1;
                long bw=0;
                __m256 val= _mm256_set1_ps(0);

                high_resolution_clock::time_point t1 = high_resolution_clock::now();
                for(int i=0;i<100;i++)
                {
                        val=avx_memoryRead(array,size);
                }
                high_resolution_clock::time_point t2 = high_resolution_clock::now();
                duration<double> time_span = duration_cast<duration<double>> (t2 - t1);
                bw = (size*4*100)/(1000000000 * time_span.count());
                printf(" Time taken for 100 operations of %d bytes : %lf BW = %lf GB/s\n",size*4,time_span.count(), bw);
                printf("value %f\n",val[0]);

        }
    return 0;
}
```



3) Calculating Latency for byte transfer with different sizes of data : 1kB, 4KB, 32KB, 256KB, 1MB, 4MB, 20MB, 200MB

Code used for generating Latency:

```
#include<stdio.h>
#include <iostream>
#include<chrono>
#include<ctime>

using namespace std;
using namespace std::chrono;

int main()
{
	int size=52428800,rnd=0,i=0,current=0; // data size taken 200MB
	int *randa=(int*)malloc(sizeof(int) * size); // Array considered to fill random values in randb
	int *randb=(int*)malloc(sizeof(int) * size); // Array of unique random values
	int *next=(int*)malloc(sizeof(int) * size); // Array considered to traverse linearly
	int resize=size; // resize the length of randa
	for(int j=0;j<size;j++) // Array initialization
	{
		randa[j]=j;
		randb[j]=0; // so that when *last reaches randa[0], remaining randb is 0
		next[j]=j;
	}
	int *last=&randa[size-1]; // pointer to the last element of randa

	while(*last!=0) // until pointer becomes randa[0]
	{
		rnd = rand()%resize--; // random variable according to resize of randa[]
		randb[i++]=randa[rnd]; // assign randb with random value
		randa[rnd]=*last--; // swap the last element of array with randa[rnd], also shift
pointer from last element to last element-1
}

	high_resolution_clock::time_point t1 = high_resolution_clock::now();
	for(int k=0;k<size;k++)
	{
		current=next[current]; // traversing linearly
	}
	high_resolution_clock::time_point t2 = high_resolution_clock::now();
	duration<double> time_span = duration_cast<duration<double>> (t2 - t1);
	printf("Time taken for linear is %lf \n",time_span.count());

	current=0;
	high_resolution_clock::time_point t3 = high_resolution_clock::now();
	for(int k=0;k<size;k++)
	{
		current=randb[current]; // traversing randomly
	}
	high_resolution_clock::time_point t4 = high_resolution_clock::now();

	duration<double> time_span1 = duration_cast<duration<double>> (t4 - t3);
	printf("Time taken for random is %lf \n",time_span1.count());
return 0;
}
```

Now, plotting graphs for the time values generated in milliseconds:

Memory SIze	Time for total linear accesses (millisec)	Time for total random accesses (millisec)	Time per byte for linear accesses (nanosec)	Time per byte for random accesses (nanosec)
1KB	0.001	0.001	0.9	0.9
4KB	0.004	0.004	0.9	0.9
32KB	0.03	0.032	0.9	0.9
256KB	0.238	0.456	0.9	1.8
1MB	0.953	3.94	0.9	3.7
4MB	3.807	15.743	0.9	3.72
20MB	19.026	103.64	0.9	4.9
200MB	190.326	3787.43	0.9	17