Analysis of performance of Matrix Transposition and Insertion Sort kernels on Xeon Phi and NVIDIA K20X devices

Project Report by
Anusha Balaji and Lalitha Geddapu
Department of Electrical and Computer Engineering
University of Florida

Abstract

This paper presents the comparison and analysis of two kernels - matrix transpose and insertion sort across three devices Xeon E5-2670, Xeon Phi and NVIDIA K20X. The kernels have been implemented using OpenMP and CUDA languages to exploit parallelism. Our implementations of the kernels on the specified devices are not completely new but are based on existing optimized libraries. The results have been presented in form of tables and graphs for execution times of kernels, Computation Density (CD) and Realizable Utilization (RU).

1. Introduction

In matrix transposition, the elements of the matrix are redistributed to form its transpose. Parallelisation of matrix transpose gains importance in a case where the matrix is stored in a row-major order and an operation like fast Fourier transform requires repeated operations on columns. Sorting is nothing but arranging data in an orderly manner. There has been abundant research pertaining to the study on parallelisation of sorting algorithms. We are specifically inclined towards the study on parallelising the insertion sort algorithm. Applications such as Google search use sorting techniques to arrange the search results in descending order of relevance. Parallelisation of such a technique would considerably reduce the time it takes to find the data.

2. Kernels and Languages

2.1 Matrix Transpose

Matrix transposition is one of the fundamental operations of matrices. It has many scientific and engineering implementations. The transpose of a matrix can be obtained by redistributing the elements of the matrix as shown in equation 1.

$$A = \begin{bmatrix} a11 & \cdots & a1n \\ \vdots & \ddots & \vdots \\ an1 & \cdots & ann \end{bmatrix}, A^T = \begin{bmatrix} a11 & \cdots & an1 \\ \vdots & \ddots & \vdots \\ a1n & \cdots & ann \end{bmatrix} \quad (1)$$

Typically, 2D arrays are stored in memory in a row-major or column major format. When matrix is stored as a row-major or column major array, it becomes more difficult to perform the transposition. Traversing row-major into columns is slower than traversing into rows at different location. To achieve good performance, multiprocessor devices need to take care of how the data is read from a single cache line, how the data should be written, how to ensure that different processors read from different cache lines and write lo locations spaced out by the length of the cache line they are reading from etc. If the algorithms used to perform the transposition are architecture unaware, they do not yield high performance. To achieve high bandwidths, multiprocessor systems like Intel Xeon Phi employ access strategies with

granular cache lines[1]. In this type of access, when a new element of matrix is read for writing into a new location, the entire cache line is read. As a result, successive reads from the same row (contiguous locations) will essentially cost no extra time as it is already in cache[1].

2.2 Insertion Sort

Sorting is arranging data in ascending or descending order. It is implemented because it much easier to search for data when it is in-order than when it is out-of-order. Insertion sort is one of such sorting algorithms. Insertion sort is a simple sorting algorithm which builds the array one at a time. It is very efficient for small arrays, but has the complexity of $O(n^2)$ for larger arrays. Insertion sort has the advantage of being efficient with linear time complexity on almost sorted lists. Insertion sort works by iterating one input element from the input and placing it at the precise location in the sorted list. Insertion sort begins with a sorted list of 1 element on the left and N-1 unsorted elements. It then compares the element to its right and moves the element accordingly to create a sorted list of 2 and N-2 unsorted. The procedure is repeated till the N elements are sorted and a complete sorted list of N elements.

2.3 OpenMP

OpenMP is a shared memory thread based programming model with a fork and join execution style. It is an API that can be used with FORTRAN, C and C++ languages to achieve parallelisation i.e. it does not need a specific compiler for its execution. It provides portability across various systems. The fork join execution style provides parallelisation only to specific blocks where needed. A thread, typically master thread, splits into several threads on encountering a parallelisation directive i.e. it indicates start of the parallel region. `#pragma omp parallel` is one such directive that spawns a number of threads that executes the part of the code written following the directive. At the end of parallel region all the threads join into a single thread and continue executing the serial part of the code. OpenMP is a good fit for shared memory programming as it allows high abstraction levels, converts serial code to parallel code with simple directives and provides easy parallelisation of loops.

2.4 CUDA

NVIDIA developed Compute Unified Device Architecture (CUDA) platform to implement across its range of graphic processors (GPUs). NVCC is the compiler that is used to compile the CUDA files which come with the extension .cu. A parallel system in CUDA comprises of a host CPU and computation device typically GPU [2]. The architecture consists of blocks and grids. A block is a group of threads whereas a grid consists of one or two blocks. Programmers do not specify creation of threads and their end. The only inputs from the programmer would be the sizes of grid and block, and workload partitioning.

3. Methodology

The programming languages and methods used for implementing the kernels are presented in this section.

3.1 OpenMP implementation

3.1.1 Matrix Transpose in OpenMP

People involved in benchmarking: Lalitha Geddapu

Devices being targeted: Xeon E5 CPU and Xeon Phi

Implementation: For matrix transpose, we used 2D dynamic memory allocation format to make the logic simple. The number of rows vary from 2^1 to 2^{20} while the columns were varied from 2^{20} to 2^1. Initially, a naïve transpose has been performed where the rows loop of the transpose computation is parallelised always irrespective of the number of rows or columns.

```
……..
Matrix = (int **) malloc(sizeof(*(Matrix)) * m);
for (i = 0; i < m; i++) {
Matrix[i] = (int *) malloc(sizeof(*(Matrix[i]))
* n);
for (j = 0; j < n; j++)
Matrix[i][j] = rand()%100;
}
…...
```

Memory allocation

```
...
#pragma omp parallel for private(j)
for (i = 0; i < m; i = i + 1)
{
for (j = 0; j < n; j = j + 1)
transpose[j][i] = Matrix[i][j];
}
...
```

Naïve transpose

The drawback of this naïve transpose is that, for a 2 X 1048576 matrix two threads take each row of the matrix and then transpose it. This reduces performance as even though there are more number of threads, only two threads perform the computation. Thus depending on whether rows > columns or columns > rows, we change the way we parallelise the outer for loop in the program.

```
...
if(m>=n){
#pragma omp parallel for private(j)
for (i = 0; i < m; i = i + 1){
for (j = 0; j < n; j = j + 1)
transpose[j][i] = Matrix[i][j];}}
else(m<n){
#pragma omp parallel for private(i)
for (j = 0; j < n; j = j + 1){
for (i = 0; i < m; i = i + 1)
transpose[j][i] = Matrix[i][j];}}
...
```

Modified transpose

In order to measure the performance of the code, we used time functions to specify start time and end time and then calculate the elapsed time. The time returned by the system is in seconds. Since it is difficult to compare performance for a single computation, we perform the transposition for 1000 times to obtain significant timing values. For each data size (rows and columns), the corresponding time is stored in an array. The resulting array containing the timing values is then written to a file at the end of the program. The results section of this report presents and compares the performance of the program across Xeon E5 CPU and Xeon Phi.

3.1.2 Insertion Sort in OpenMP

People involved in benchmarking: Anusha Balaji

Devices being targeted: Xeon E5 CPU and Xeon Phi

Implementation: The performance obtained to sort an array using insertion sort is entirely dependent on the input array size. 1 dimensional array of numbers are chosen as the input. The inputs are generated randomly for the entire array and the placement of the elements are further randomised within the array, thus enabling 1D dynamic memory allocation for the array. The size of the array is varied from $2^1$to 2^{20} and the corresponding time to sort the array are noted when the code is run on the Intel Xeon E5 CPU and Intel Xeon Phi. For the OpenMP implementation, insertion sort is implemented as a pure insertion sort which employs sorting sub-arrays of the main array in parallel, followed by external sorting of the sorted sub-arrays. External sorting is done on partially sorted sub-arrays. A modified version of insertion sort was also implemented which applies merge sorting coupled with insertion sorting on the one dimensional array. The code snippets shown below were utilised to implement using OpenMP on the Xeon E5 CPU and Intel Xeon Phi.

```
Insertion sort:
... . .
#pragma omp parallel for
private(j,k,stride,numelem)

for(i=0;i<threads;i++)
 {

 stride=0;
 TID=omp_get_thread_num();

numelem=TID<extra?(avgelem+1):(avgelem);

for(k=0;k<TID;k++)
       stride+=offset[k];

for(j=0;j< numelem ;j++)
       C[j] = A [j+stride];

 insertion_sort(C,numelem);

 for(j=0;j< numelem ;j++)
       I[j+stride]=C[j];

}
```

```
Modified insertion sort:
...
...
void merge(int array[], int n)
{
 int i, m;

 for(m = n/2; m > 0; m /= 2)
 {

 #pragma omp parallel for private
(i)
 for(i = 0; i < m; i++)
 insertion(&(array[i]), n-i, m);

 }

}
...
```

The running time of the OpenMP code for insertion sort are measured on both CPU and Xeon Phi. The number of threads in the CPU is varied between 2 to 16 and Xeon Phi is varied from 4 to 240. As the complexity of insertion sort is of the order n^2, the time taken for sorting increases drastically as the size gets increased. By running multiple threads in parallel (p threads), the complexity of the parallel region is reduced to the order of $\frac{n^2}{p}$. External sorting is then applied on these sorted sub-arrays to achieve a completely sorted array. Other sorting algorithms can be applied for sorting arrays of higher sizes. The study on the insertion sort kernel is essentially performed to observe the advantages of the kernel for smaller data sizes.

3.2 CUDA implementation

3.2.1 Matrix Transpose in CUDA

People involved in benchmarking: Anusha Balaji

Device being targeted: NVIDIA K20X

Implementation:

For CUDA implementation of the matrix transpose, we use inbuilt functions like `cudaMemcpy()` are used to copy the data from host to memory or vice versa. `cudaMalloc()` and `cudaFree()` are used to allocate the memory linearly and free the same. The initialised source matrix is copied into the device memory. The number of threads per block along with rows and columns are used to compute the block size and grid size. The transpose function is declared with a __global__ declaration specifier which is then called specifying the block size and grid size. The control shifts from host to device at this point. In the device(GPU) based on the block ID and thread ID, each element of the matrix is transposed in parallel and then

```
__global__ void transpose(float* A,float* At,int rows,int cols)
{
        int c = blockIdx.x * blockDim.x + threadIdx.x;
        int r = blockIdx.y * blockDim.x + threadIdx.y;

        if(c < cols && r < rows){
        At[c*rows+r]=A[c+r*cols];}
}
```

Transpose function with __global__ directive

placed into the destination matrix. Once the computation is complete, the transposed matrix is copied from device to the host CPU.

For measuring performance, CUDA provides timing functions like `cudaEventCreate()` and `cudaEventRecord()`. The event record function is used to start or stop the timer. `cudaEventSynchronize()` is used to provide synchronisation. `cudaEventElapsedTime()` is used to compute the elapsed time of the computation. The measured time returned by the CUDA function is in milliseconds. We perform the transposition for 1000 times to obtain significant timing values. The results section of this report presents the timing values of this program when run on NVIDIA K20X.

```
int main()
{
...
/*copy matrix from Host to device memory*/
cudaMemcpy(A_d,A,size,cudaMemcpyHostToDevice);
...
/*CUDA timer declarations*/
cudaEvent_t start_transpose, stop_transpose;
cudaEventCreate(&start_transpose);
cudaEventCreate(&stop_transpose);
cudaEventRecord(start_transpose,0); /*start timer*/

      for (r = 0; r < 1000; r++)
      {
      transpose<<<gridSize,blockSize>>>(A_d,At_d,rows,cols);
      }
cudaEventRecord(stop_transpose,0); /*stop timer*/
cudaEventSynchronize(stop_transpose);
cudaEventElapsedTime(&elapsedTimeTrans, start_transpose,stop_transpose);

/*copy output from device to host memory*/
cudaMemcpy(At,At_d, size, cudaMemcpyDeviceToHost);
...
...
```

Timing functions and memory copy function

3.2.2 Insertion Sort in CUDA

People involved in benchmarking: Lalitha Geddapu

Device being targeted: NVIDIA K20X

Implementation: Insertion sort is implemented in the NVIDIA K20X GPU, by performing insertion sort on blocks of data. The algorithm for insertion sort requires the algorithm to be performed by single thread within each block in the GPU. By parallel execution of the insertion sort on p number of blocks (analogous to the threads in the context for comparison), the order is reduced to $\frac{n^2}{p}$. The array size is varied from 2^1 to 2^{20}. The array consists of randomly generated numbers. The array is copied from the host to the GPU device for execution by utilising `cudaMemcpy()` from host to device. The time for the parallel insertion

sort execution is noted by using the *cudaEventRecord()*. The partially sorted array is then copied from the device to host using the *cudaMemcpy()* from device to host. External merge sorting is required to sort the sorted sub-arrays obtained from the insertion sort carried out in parallel. The code snippet used for the CUDA implementation is shown below.

Insertion sort- CUDA implementation:

```
__global__ void insert( int *a, int n )
{
int c, d, p, t, size;
int i = blockIdx.x *n;
size = ( blockIdx.x + 1 )*n;

for(c=i; c<size; c++)
{      d=c;
       while ( d>i && a[d] < a[d-1] )
       {      t      = a[d];
              a[d]   =a[d-1];
              a[d-1] =t;
              d--;
       }
}
}

int main()
{
...
cudaMemcpy(device_array,host_array,num_bytes,cudaMemcpyHostToDevice);

cudaEvent_t start_insert, stop_insert, start_merge, stop_merge;
cudaEventCreate(&start_insert);
cudaEventCreate(&stop_insert);

cudaEventRecord(start_insert,0);

for (r = 0; r <  1000; r++)
{
       insert<<< block_size, 1>>>(device_array,num_elem/block_size);
}

cudaEventRecord(stop_insert,0);

cudaEventSynchronize(stop_insert);
float elapsedTimeInsert;
cudaEventElapsedTime(&elapsedTimeInsert, start_insert,stop_insert);

cudaMemcpy(host_array,device_array,num_bytes,cudaMemcpyDeviceToHost);
... . .
```

The study of the insertion sort kernel in GPU is done as the kernel has a wide prevalence for usage in other algorithms such as sort and sweep which takes advantage of the spatial coherence associated with it when employed on mostly sorted list acting as a tool for accelerating the algorithm.

3.3 Roadblocks

1. **Matrix transpose:** Minor problems like freeing of 2D memory were the only problems encountered during the execution which did not take much time for debugging.

2. **Insertion Sort:** As the complexity of the insertion sort is of the order n^2, and the algorithm by itself is not designed for efficient parallel implementation, additional external sorting was required for sorting the array. For small size of the arrays, insertion sort gives faster results, the same reason for which the Intel Math kernel library uses the optimised *slasrt* which uses quick sort for sizes greater than 20 and insertion sort for array sizes less than 20. As a result, external sorting was done using merge sort.

3. Understanding the CUDA language structure was a little difficult. Once we understood the memory allocations, transfers, host and GPU code segments, writing the code did not take much time.

4. Results and Analysis

In this section we present the results obtained from Xeon E5 CPU, Xeon Phi and NVIDIA K20X when the kernels were run on them.

Kernel 1: Matrix transpose

When naïve matrix transpose is executed on Xeon E5 CPU, the results obtained are tabulated as below. The timing values are for 1000 tests and are measured in seconds.

TABLE 1: Naïve Matrix Transpose using OpenMP on Xeon E5 CPU.

		Time in seconds/1000 tests			
Rows(m)	Columns(n)	T=4 (s)	T=8(s)	T=12(s)	T=16 (s)
2	**1048576**	7.811307	7.947528	8.026068	8.516942
4	**524288**	3.971526	3.941009	3.972008	4.15156
8	**262144**	5.061659	2.652521	2.669865	2.745125
16	**131072**	8.25912	5.133886	5.018667	2.276537
32	**65536**	23.79283	15.56922	12.35435	7.537556
64	**32768**	38.45093	19.94645	17.10403	11.64249
128	**16384**	8.589735	9.04593	9.473623	10.81735
256	**8192**	8.029593	4.547037	3.78509	3.988808
512	**4096**	6.920661	3.716102	2.913736	2.546128
1024	**2048**	4.220628	2.33689	1.714621	2.361582
2048	**1024**	4.161182	2.227095	1.63183	2.199768
4096	**512**	4.183009	2.24725	1.595303	2.249361
8192	**256**	4.026066	2.107982	1.493463	2.287871
16384	**128**	3.734034	1.944192	1.345377	2.278332
32768	**64**	4.55398	2.363828	1.688204	1.941997
65536	**32**	4.406278	2.282012	1.58121	1.376852
131072	**16**	4.37816	2.265374	1.568306	1.274553
262144	**8**	2.136088	1.097862	0.766704	1.039928
524288	**4**	2.282382	1.100957	0.777959	1.073381
1048576	**2**	3.064965	2.193052	2.12351	2.315159

The graph for the table is presented in Fig.1. Note that the time is in seconds and is measured for 1000 tests.

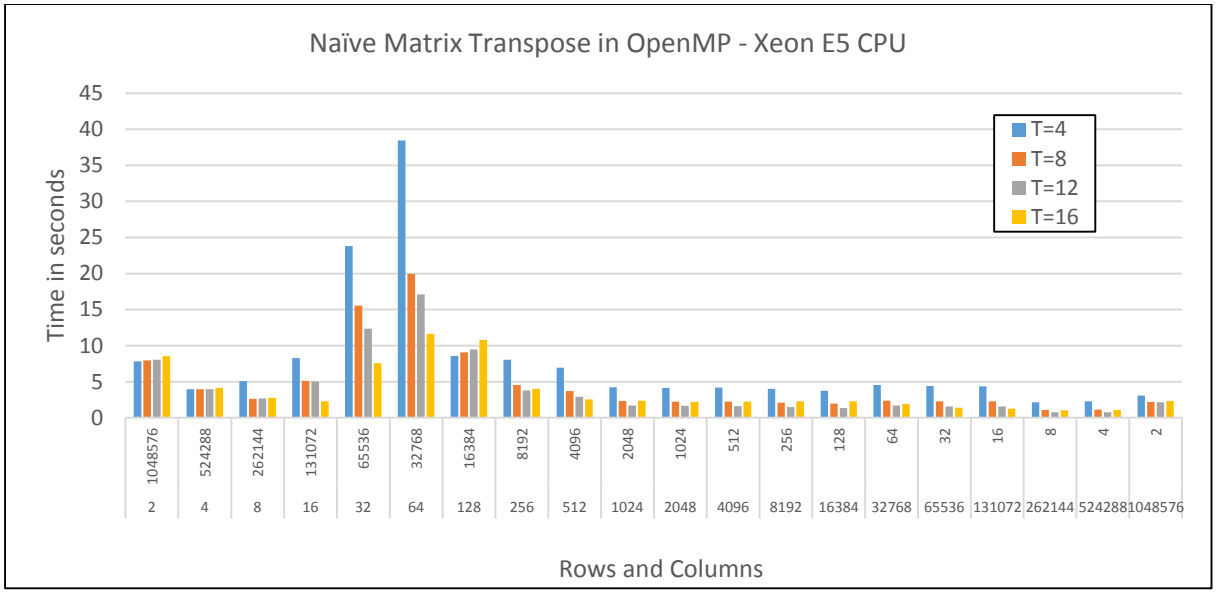

Fig 1: Naïve Matrix Transpose using OpenMP on Xeon E5 CPU. Time taken is for 1000 tests.

The number of threads are varied from 4 to 16 while the data sizes are varied from 2^1 to 2^{20}. On Xeon E5 CPU each core can spawn 4 threads when OpenMP programming model is used. From the table and figure, we can observe that the time for computation decreases as the number of threads is increased from 4 to 12. Initially there is not much time difference. But as the number of rows increases, the time for computation with 12 threads is much lesser than with 4 threads. We can also see the performance degradation when the number of threads is increased to 16. Thus we can determine the optimum thread count to be 12. As described in section 4.1.1, the naïve transpose has its drawbacks. When optimized matrix transpose is run on Xeon E5 CPU the results obtained are tabulated as below. The timing values are for 1000 tests and are measured in seconds.

TABLE 2: Modified Matrix Transpose using OpenMP on Xeon E5 CPU.

		Time in seconds/1000 tests			
Rows (m)	Columns (n)	T=4	T=8	T=12	T=16
2	1048576	3.721216	2.998398	3.262918	2.967072
4	524288	2.455871	1.171522	0.860089	1.062326
8	262144	2.142488	1.098179	0.760362	1.037309
16	131072	2.159768	1.130299	0.775061	1.057593
32	65536	2.162655	1.118599	0.771405	1.080789
64	32768	2.371262	1.215595	0.936294	1.437411
128	16384	3.762908	1.937124	1.342953	2.066667
256	8192	4.064424	2.091643	1.44999	2.138479
512	4096	3.918881	2.1074	1.467595	2.115559
1024	2048	3.933239	2.027573	1.407685	2.091023
2048	1024	4.100674	2.225575	1.648363	2.211068
4096	512	4.153112	2.244497	1.583548	2.191549
8192	256	4.055838	2.111042	1.498658	2.275196
16384	128	3.774404	1.941892	1.346936	2.348613
32768	64	4.564182	2.36471	1.687379	1.387697
65536	32	4.41056	2.282487	1.582867	1.315949
131072	16	4.379573	2.264903	1.631879	1.272895
262144	8	2.10878	1.072912	0.748716	1.043991
524288	4	2.252311	1.060501	0.782834	1.064668
1048576	2	3.080406	2.194308	2.141578	2.250072

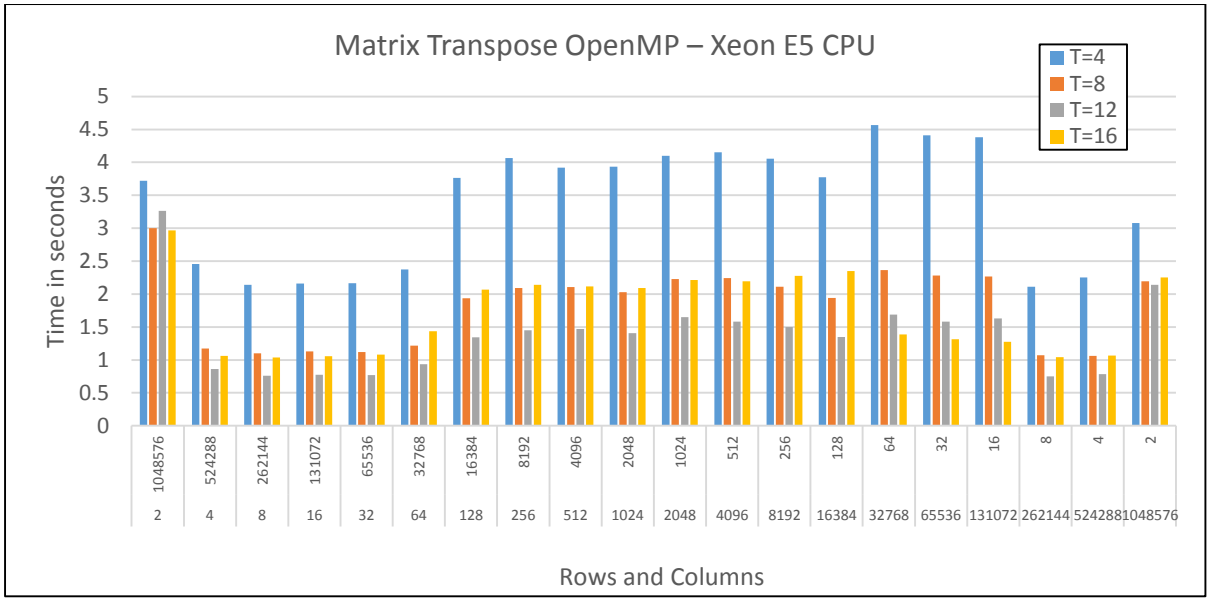

Fig 2: Modified Matrix Transpose using OpenMP on Xeon E5 CPU. Time taken is for 1000 tests.

For the optimized matrix transpose, thread count of 12 gives the best performance. We can observe that as the number of threads increases, the time for execution came down gradually. If the values of naïve transpose are compared with that of the modified one, we can observe that the average time of execution is less for modified transpose. A comparison between the two for a thread size of 16 is presented below. The timing values are for 1000 tests and are measured in seconds.

Table 3: Matrix Transpose using OpenMP on Xeon E5 CPU

		Time in seconds/1000 test	
Rows (m)	Columns (n)	Naïve	Optimized
2	**1048576**	8.516942	2.967072
4	**524288**	4.15156	1.062326
8	**262144**	2.745125	1.037309
16	**131072**	2.276537	1.057593
32	**65536**	7.537556	1.080789
64	**32768**	11.64249	1.437411
128	**16384**	10.81735	2.066667
256	**8192**	3.988808	2.138479
512	**4096**	2.546128	2.115559
1024	**2048**	2.361582	2.091023
2048	**1024**	2.199768	2.211068
4096	**512**	2.249361	2.191549
8192	**256**	2.287871	2.275196
16384	**128**	2.278332	2.348613
32768	**64**	1.941997	1.387697
65536	**32**	1.376852	1.315949
131072	**16**	1.274553	1.272895
262144	**8**	1.039928	1.043991
524288	**4**	1.073381	1.064668
1048576	**2**	2.315159	2.250072

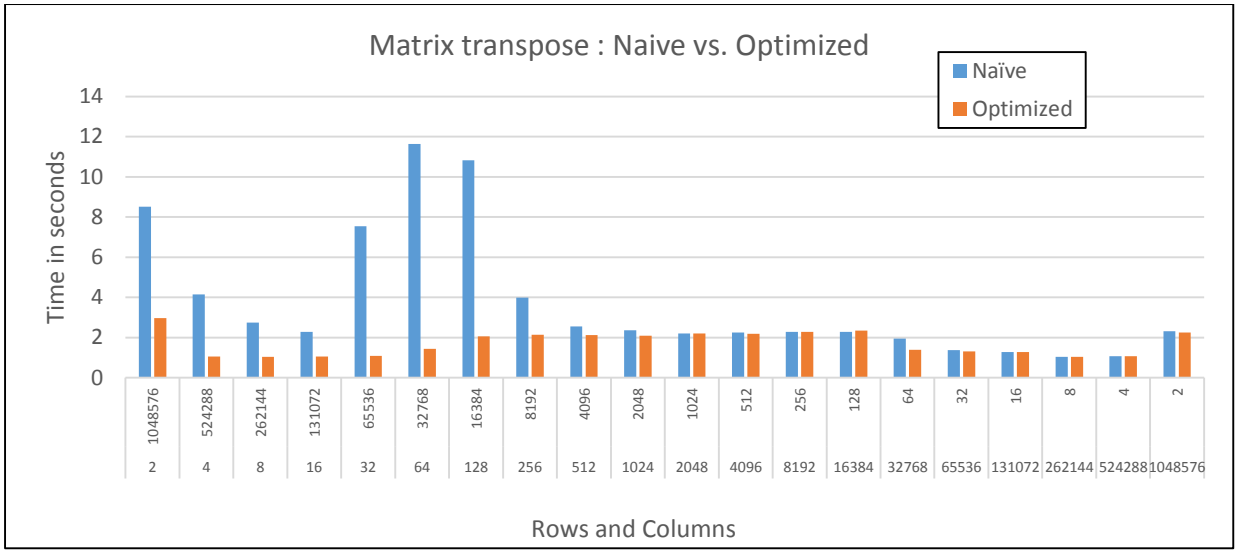

Fig 3: Modified Matrix Transpose using OpenMP on Xeon E5 CPU. Time taken is for 1000 tests.

We can observe from the graph that for m<<n, the modified transpose code performs much better than the naïve transpose. With 11.64 seconds being the longest time for naïve transpose, the corresponding time for modified transpose is as low as 1.43 seconds. The modified transpose is chosen to run on Xeon Phi processor as it gives better performance than the naïve one. Thread count is varied between 16 and 256. Some of the results obtained are tabulated as below. The timing values are for 1000 tests and are measured in seconds.

Table 4: Matrix Transpose using OpenMP on Xeon Phi.

		Time in seconds/1000 tests					
Rows (m)	Columns (n)	T=16	T=64	T=160	T=184	T=240	T=256
2	1048576	34.69993	11.35444	6.317913	6.520839	5.716264	8.366576
4	524288	24.75386	9.22035	5.152404	4.671817	4.191395	6.689055
8	262144	24.44642	7.875319	4.533976	4.015914	4.110114	6.320249
16	131072	24.85841	6.815227	4.195649	4.095391	3.963297	6.060922
32	65536	24.9831	6.986328	3.897433	3.932373	3.716219	5.725249
64	32768	26.25804	9.278963	4.474675	3.66638	4.923162	5.816992
128	16384	11.63787	3.530338	1.912001	1.962386	1.73802	3.089207
256	8192	11.45256	3.659316	1.883986	2.034735	1.881416	3.255893
512	4096	13.80652	3.43485	1.78434	1.964804	1.677199	3.427019
1024	2048	14.47084	3.614582	1.858995	2.082346	1.831805	3.764231
2048	1024	12.4359	3.094347	1.844729	1.923664	1.983702	3.747689
4096	512	8.986835	2.079263	1.337151	1.352119	1.255889	3.060621
8192	256	9.356972	1.346031	1.052377	0.989873	0.992199	2.699449
16384	128	8.143112	1.549208	0.82113	0.850648	0.817378	2.354777
32768	64	11.46512	1.788488	0.883954	0.838551	0.80825	2.178016
65536	32	13.19411	1.67774	0.836503	0.773229	0.7732	1.978431
131072	16	8.881404	1.685621	0.813845	0.751038	0.674786	2.287522
262144	8	6.293646	1.482678	0.660826	0.664367	0.562963	2.204366
524288	4	4.357323	1.355077	0.569358	0.580968	0.471497	2.033893
1048576	2	4.255804	1.388931	0.8725	0.92482	0.798309	2.390501

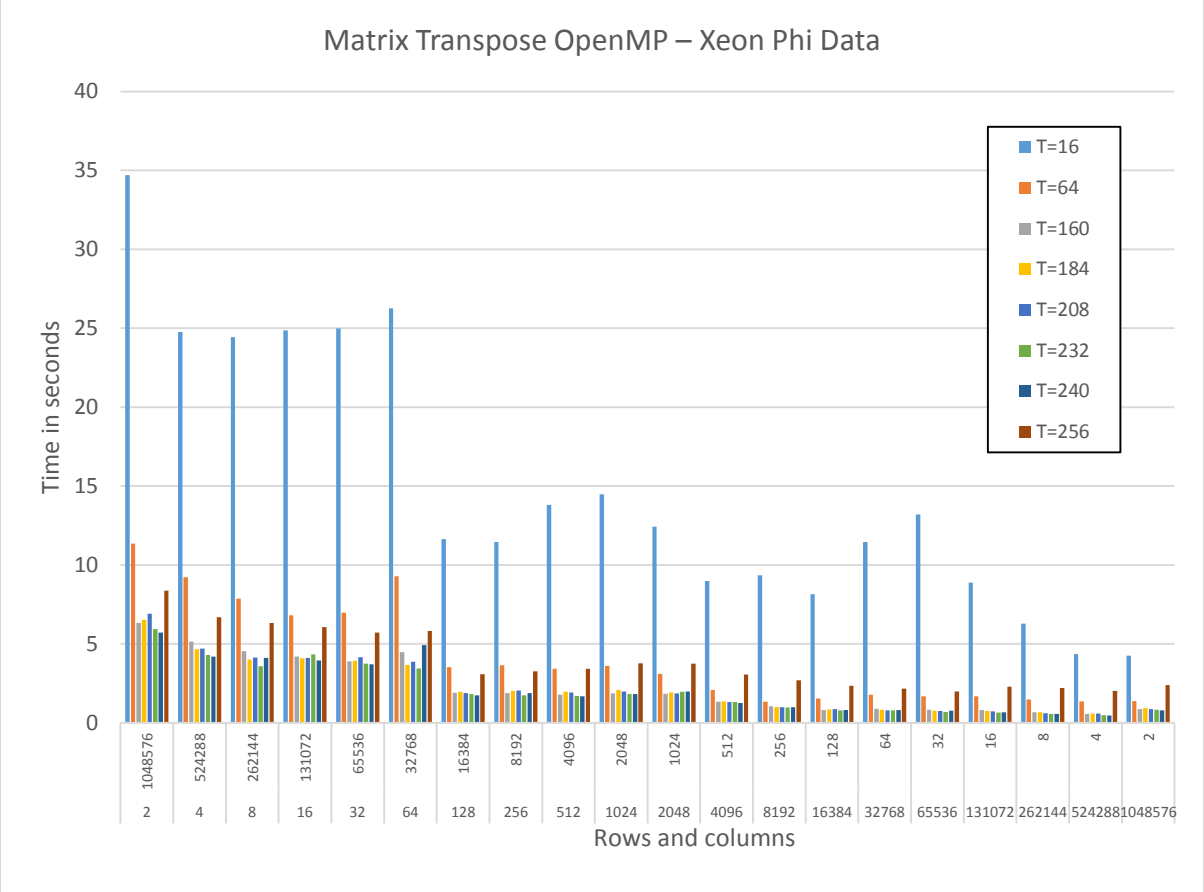

Fig 4: Modified Matrix Transpose using OpenMP on Xeon Phi. Time taken is for 1000 tests.

We can observe that the time for execution is gradually decreasing when the number of threads is increased. At 240 threads, the Xeon Phi gives the best performance as the maximum number of threads in the co-processor that can be used without performance degradation is 240 threads. Clearly, at 256 threads the time for execution spiked up again.

The realizable utilization (RU) for this implementation is very less as all the operations involved are memory operations. Each memory operation requires 25.3 operations to be implemented in Xeon Phi. But the number of operations in our implementation consist only memory operations.

The CUDA implementation of the matrix transpose was run on the NVIDIA K20X using 'nvcc' compiler. The number of threads have been varied from 4 to 2048. The thread count presented here is the number of threads per single block. The number of blocks multiplied by number of threads per block gives the total number of threads in the unit. The upper bound on number of threads per block is 1024 threads and this limit exists because the threads in a single block are expected to be on the same processor and share the memory of that processor [3]. The results obtained from the run are tabulated as below. The timing values are for 1000 tests and are measured in milliseconds.

Table 5: Matrix Transpose using CUDA on NVIDIA K20X.

		Time in seconds/1000 tests								
Rows (m)	Columns (n)	T=16	T=64	T=128	T=240	T=256	T=512	T=1024	T=1200	T=2048
2	1048576	0.25472	0.115232	0.112224	0.139808	0.129024	0.129728	0.135648	0.14	0.152096
4	524288	291.1581	0.098208	0.104736	0.12	0.113888	0.11296	0.1248	0.108928	0.14832
8	262144	152.8461	0.096832	0.102048	0.115712	0.114112	0.110752	0.124928	0.1088	0.137376
16	131072	144.3583	0.098656	0.098176	0.115712	0.113088	0.1208	0.119808	0.110432	0.13664
32	65536	172.0491	0.098688	0.10368	0.114528	0.11488	0.111456	0.119968	0.1088	0.140512
64	32768	173.1685	0.09472	0.095008	0.114592	0.117888	0.111904	0.119392	0.110368	0.13776
128	16384	173.6046	0.095232	0.096	0.11408	0.11072	0.11344	0.125536	0.108864	0.136256
256	8192	174.3492	0.095584	0.096928	0.116032	0.110944	0.114112	0.119648	0.109344	0.143008
512	4096	173.124	0.0984	0.096672	0.114336	0.112384	0.109952	0.119584	0.10992	0.139808
1024	2048	173.81	0.098176	0.099776	0.117504	0.11456	0.11136	0.122144	0.108256	0.136864
2048	1024	174.2533	0.094592	0.095936	0.116736	0.110624	0.111904	0.118944	0.107136	0.137888
4096	512	171.6618	0.096768	0.097056	0.117888	0.11008	0.110944	0.11856	0.110784	0.139296
8192	256	173.4443	0.094944	0.098816	0.125376	0.11104	0.119008	0.118624	0.107456	0.136352
16384	128	173.9648	0.097888	0.095424	0.115584	0.11248	0.113184	0.12288	0.108576	0.135584
32768	64	184.8965	0.096128	0.096576	0.11408	0.115136	0.110432	0.117952	0.109024	0.195744
65536	32	173.856	0.095072	0.096384	0.113504	0.111648	0.112736	0.119488	0.110112	0.141088
131072	16	162.8653	0.095648	0.09648	0.113696	0.112128	0.110656	0.121824	0.107136	0.13696
262144	8	208.9595	0.09664	0.09712	0.114528	0.10336	0.109472	0.1184	0.10656	0.136864
524288	4	315.8961	0.098976	0.096256	0.11504	0.114112	0.114176	0.12032	0.115072	0.139616
1048576	2	0.10784	0.096	0.095616	0.116672	0.129152	0.111584	0.121696	0.10672	0.13584

The graph for the table is presented below. Note that the time is in milliseconds and is measured for 1000 tests.

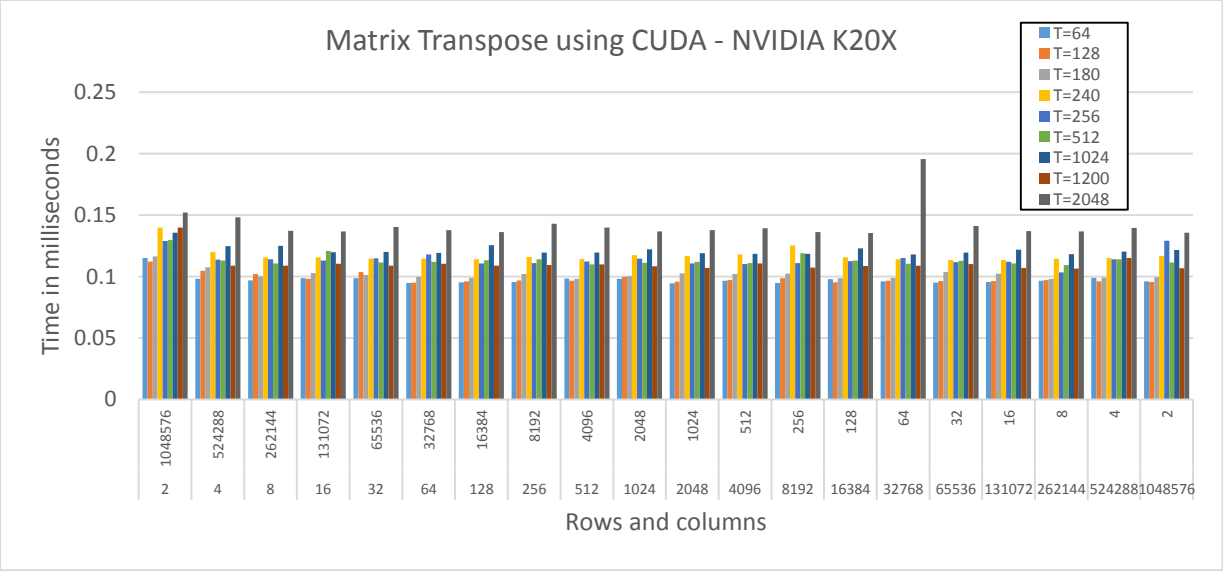

Fig 5: Modified Matrix Transpose using CUDA on NVIDIA K20X. Time taken is for 1000 tests.

We can observe that the performance is the best at the thread count of 64 threads per block. As the number of threads increased the performance degraded as the total number of threads in the unit is much more than required for the data sizes e run on the GPU. The graph provides a comparison between the implementations across Xeon E5 CPU, Xeon Phi and NVIDIA K20X devices for a thread size of 16. Note that the time is in seconds and is measured for 1000 tests.

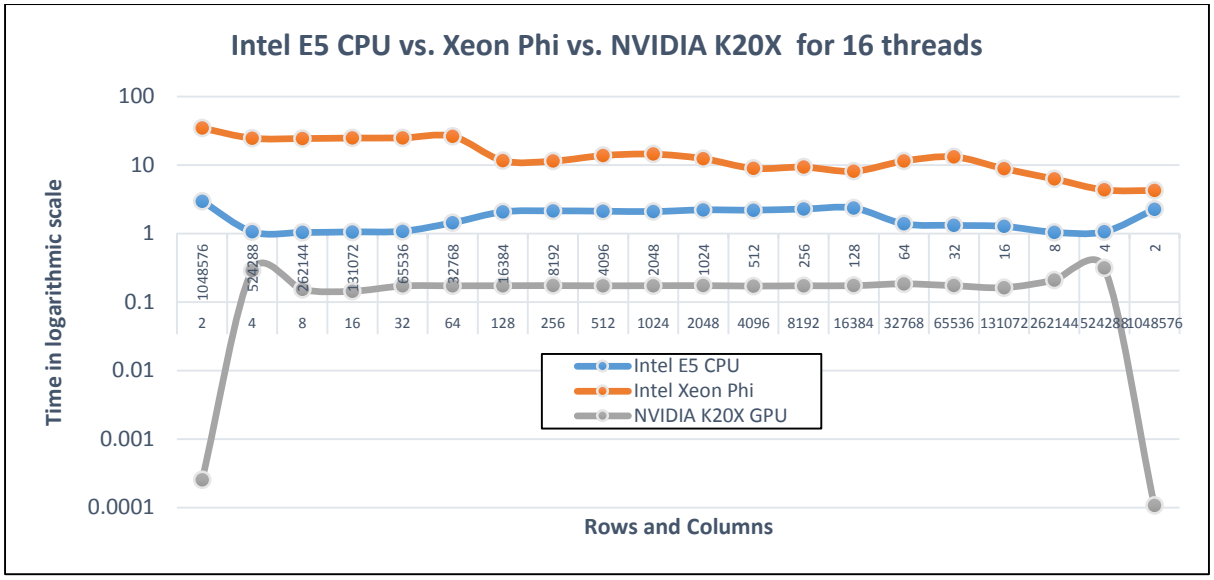

Fig 6: Performance comparison of Matrix Transpose across CPU, Phi and GPU for 16 threads. Time taken is for 1000 tests.

The graph is plotted with time in logarithmic scale as the difference between time values is very high and we needed to accommodate all the values in a single graph. It can be observed that NVIDIA K20X gives the best performance amongst all which is as expected. However, Xeon E5 CPU performed better than Xeon Phi which deviates from our expectations. For 16 threads, at a matrix data size of 4096x512, the GPU performs 12x times faster than E5 CPU and 52x times faster than Xeon Phi. This can explained by the fact that the operation is memory-bound rather than compute-bound and is more suitable for GPU execution.

Kernel 2: Insertion Sort

When we ran insertion sort using OpenMP, the results obtained are tabulated as below. The timing values are for 1000 tests and are measured in seconds.

TABLE 6: Pure insertion sort using OpenMP on Xeon E5 CPU.

Number of elements (n)	Time in seconds/1000 iterations T=4	T=12	T=16
1	0.00138	0.00345	0.00418
2	0.00064	0.00145	0.001
4	0.0005	0.00147	0.001
8	0.0005	0.00147	0.00104
16	0.00055	0.00144	0.00112
32	0.00063	0.00147	0.00109
64	0.00103	0.00165	0.0015
128	0.00249	0.00303	0.00296
256	0.0074	0.00863	0.00967
512	0.02891	0.03431	0.03848
1024	0.10948	0.13801	0.13889
2048	0.44686	0.5348	0.55242
4096	1.77848	2.10859	2.19493
8192	7.04872	8.30462	8.72261
16384	28.9951	32.7088	33.8614

TABLE 7: Pure insertion sort using OpenMP on Xeon Phi.

Number of elements (n)	Time in seconds/1000 iterations T=16	T=64
1	0.05475	0.09563
2	0.00196	0.0037
4	0.00202	0.00381
8	0.002	0.00457
16	0.00238	0.0051
32	0.00286	0.00637
64	0.00481	0.00731
128	0.01266	0.01521
256	0.03861	0.04468
512	0.15889	0.17862
1024	0.60715	0.67808
2048	2.39313	2.67607
4096	9.46802	10.5777
8192	37.5045	41.5864
16384	152.235	161.198

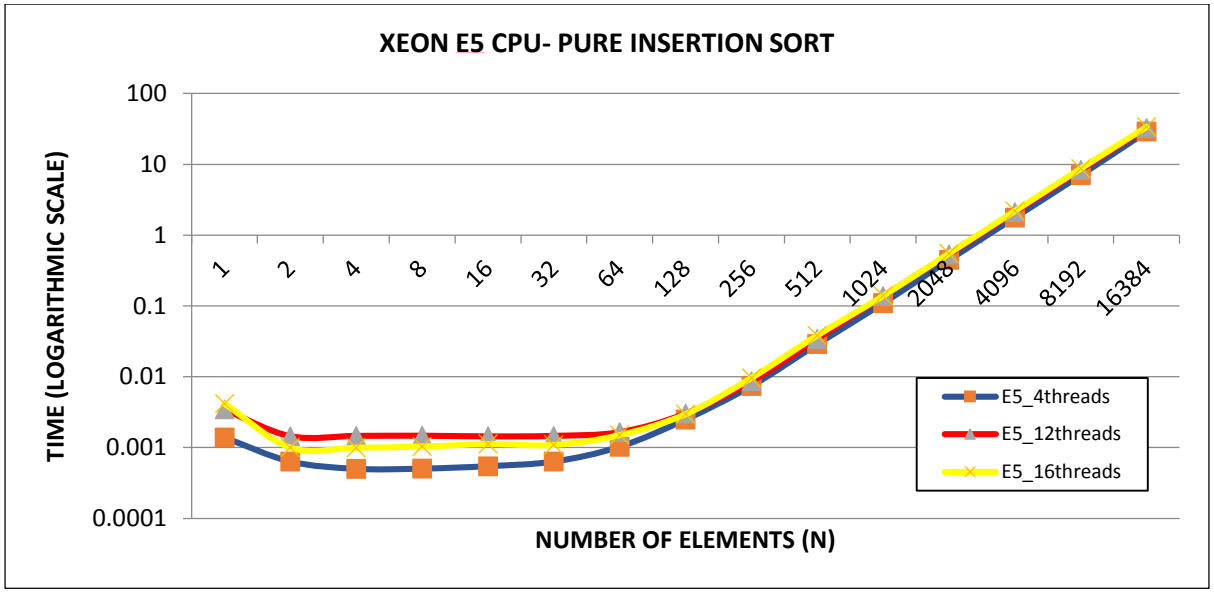

Fig 7: Pure insertion sort using OpenMP on Xeon E5 CPU. Time taken is for 1000 tests.

The insertion sort kernel is implemented on the CPU for varying array sizes from $2^1$to 2^{20} and varying the threads from 4 to 16. It can be observed that the the insertion sort kernel takes the lowest time when implemented with 4 threads on the CPU for lower data sizes and it can be observed that for larger sizes there is no significant difference in the execution times for higher number of threads in the E5 CPU.

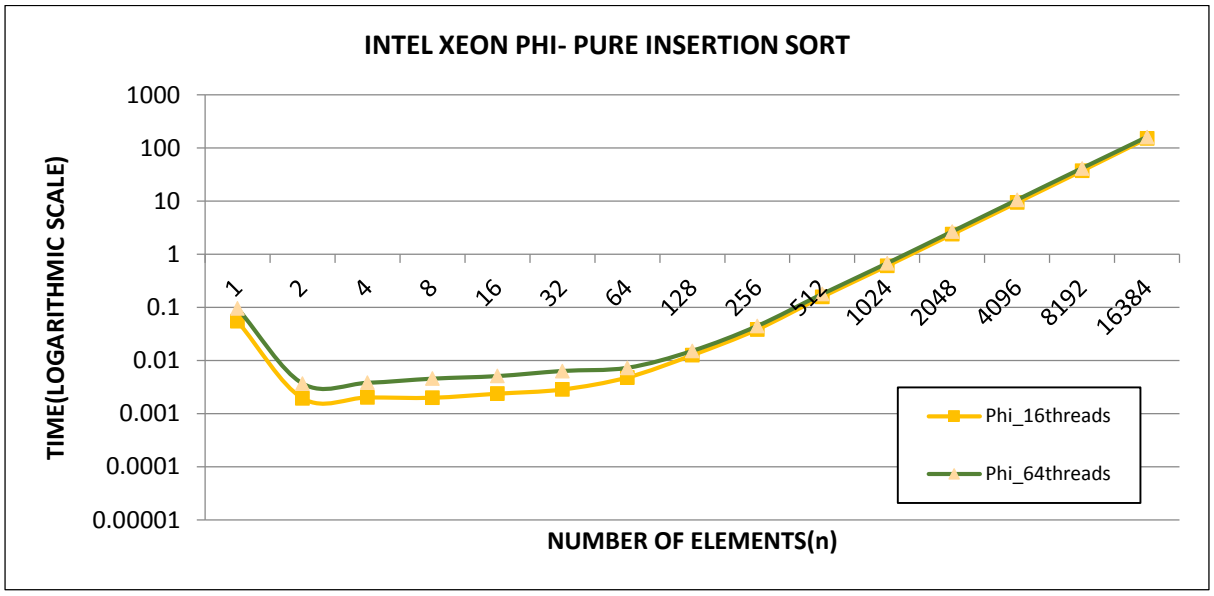

Fig 8: Pure insertion sort using OpenMP on Xeon Phi. Time taken is for 1000 tests.

Implementation on the Intel Xeon Phi results in fewer threads favouring for insertion sort. The results are tabulated in Table 8. A comparison between the CPU and the Xeon Phi for the same number of threads is been plotted. The CPU gives a better performance when compared to the Xeon Phi. This can be attributed to the better data locality in the CPU. In the Xeon Phi, the array is distributed amongst the cores. The recursive nature of the algorithm and the heavy memory operations involved also contribute to the slow execution time.

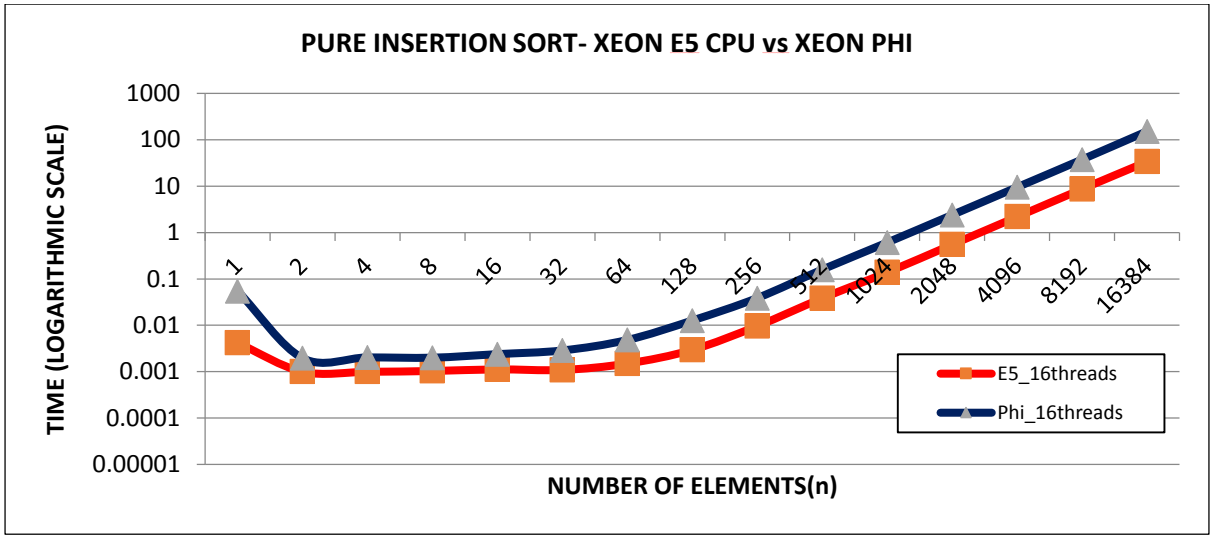

Fig 9: Pure insertion sort comparison Xeon E5 CPU vs. Xeon Phi. Time taken is for 1000 tests.

A modified version of the insertion sort was executed on the Xeon E5 CPU by varying the threads from 2 to 16 and the execution time was plotted for varying array sizes. It can be observed that the for array sizes up to 2^{14} , running using 4 threads gives the best performance. But for higher sizes (beyond 2^{14}) having 16 threads gives the better performance.

TABLE 8: Modified insertion sort using OpenMP on Xeon E5 CPU.

	Time in seconds/1000 tests			
Number of elements(n)	T=2	T=4	T=8	T=16
1	1.3E-05	1.2E-05	1.1E-05	1.2E-05
2	0.00641	0.00474	0.00755	0.00925
4	0.00826	0.00492	0.00653	0.01036
8	0.01264	0.00762	0.0103	0.01533
16	0.01734	0.01026	0.0145	0.02045
32	0.02234	0.01324	0.01929	0.02691
64	0.02949	0.01784	0.02677	0.03542
128	0.0403	0.02763	0.03573	0.04714
256	0.05077	0.04017	0.04998	0.06367
512	0.08358	0.06724	0.08212	0.10418
1024	0.13931	0.1146	0.13066	0.15122
2048	0.22922	0.18668	0.22668	0.24849
4096	0.38282	0.29616	0.40546	0.39418
8192	0.63534	0.53175	0.67049	0.63597
16384	1.12341	1.00978	1.18822	1.06953
32768	2.2281	2.05437	2.58175	2.05367
65536	4.23908	4.56043	4.54948	3.63063
131072	7.6314	8.99616	8.37388	6.93156
262144	14.6258	17.4469	16.0317	13.3416
524288	28.1909	35.0848	31.1044	26.6001

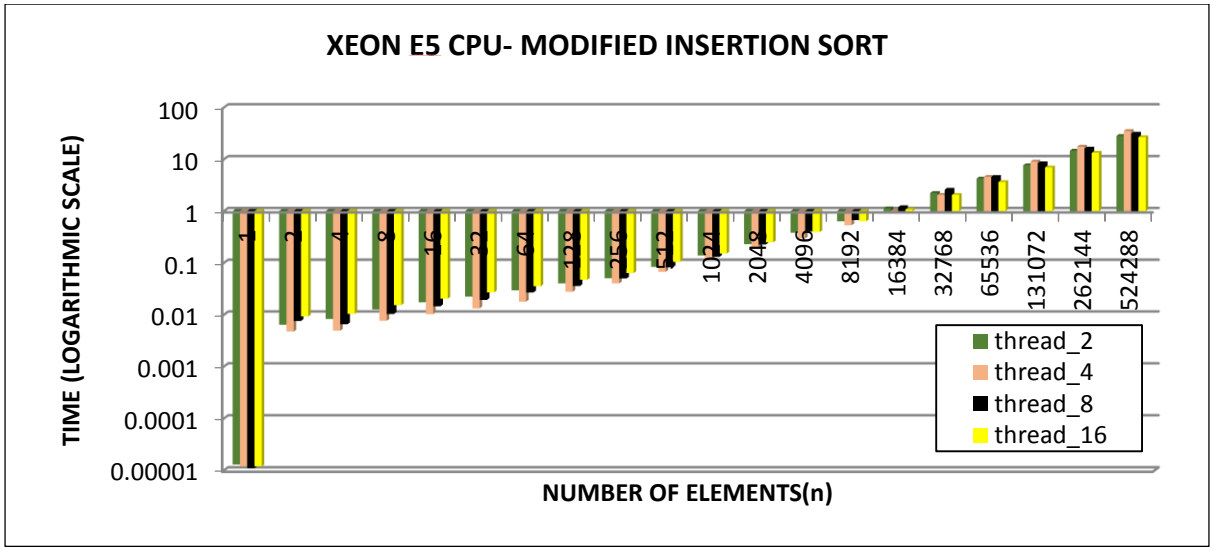

Fig 10: Modified insertion sort on Xeon E5 CPU. Time taken is for 1000 tests.

Modified insertion sort was executed on the Intel Xeon Phi for threads varying between 16 and 240. As there are 60 cores, the maximum threads is limited to 240. The best performance was observed for 180 threads. When the thread size was increased beyond 180, there was degradation in performance. For array sizes till 2^8, 4 threads gives a better performance. But as the array size is increased till 2^{16}, 32 threads gives the best performance when compared to the other thread sizes. For large arrays up to 2^{20}, 180 threads gives the most optimal execution time. The serial portion of the code hinders the parallelizability and cost the performance.

TABLE 9: Modified insertion sort using OpenMP on Xeon E5 CPU.

	Time in seconds/1000 tests								
Number of elements(n)	**4**	**16**	**32**	**64**	**128**	**180**	**184**	**196**	**240**
1	6E-06	7E-06	6E-06	7E-06	7E-06	6E-06	6E-06	6E-06	5E-06
2	0.04143	0.06834	0.08855	0.13204	0.22788	0.27226	0.29132	0.29261	0.37003
4	0.02603	0.0394	0.0462	0.07067	0.07716	0.08178	0.08326	0.08682	0.09402
8	0.0417	0.05916	0.06982	0.10607	0.11514	0.12257	0.12544	0.13088	0.14261
16	0.05695	0.08098	0.09471	0.14165	0.1533	0.16367	0.16644	0.17245	0.18827
32	0.07887	0.10531	0.1194	0.17745	0.19382	0.20432	0.20851	0.21685	0.23555
64	0.12536	0.155	0.16054	0.22102	0.23943	0.2461	0.25176	0.26044	0.2848
128	0.20851	0.24336	0.24277	0.26418	0.29467	0.30121	0.32462	0.31281	0.35107
256	0.32899	0.3726	0.33649	0.4145	0.40871	0.40469	0.45076	0.43317	0.48133
512	0.5559	0.56529	0.62097	0.63345	0.67587	0.68046	0.71428	0.72146	0.72704
1024	0.8814	0.94607	0.93221	1.08158	1.00851	1.01869	1.15213	1.10352	1.17739
2048	1.71904	1.61131	1.62375	1.73228	1.65906	1.70737	1.85876	1.83538	1.91112
4096	3.50765	2.61468	2.72184	2.8705	2.82448	2.85435	3.11864	3.09712	3.18864
8192	7.34685	4.59294	4.63455	4.95485	4.84786	4.90473	5.28718	5.35238	5.41406
16384	15.5921	8.6962	8.07442	8.91171	9.04943	8.84747	9.6682	9.54153	9.78815
32768	37.0677	17.6748	15.4016	15.9963	16.0067	15.8149	17.49	17.4421	17.5629
65536	90.0458	37.5048	31.8768	30.3844	30.6022	30.7172	33.4519	33.723	33.6841
131072	181.72	83.4146	62.8407	59.1499	59.2831	58.3418	63.8543	65.4571	65.3666
262144	419.092	170.762	131.769	122.128	121.562	119.471	131.199	130.736	130.826
524288	915.678	383.474	282.992	252.114	248.515	242.763	266.577	268.316	266.388

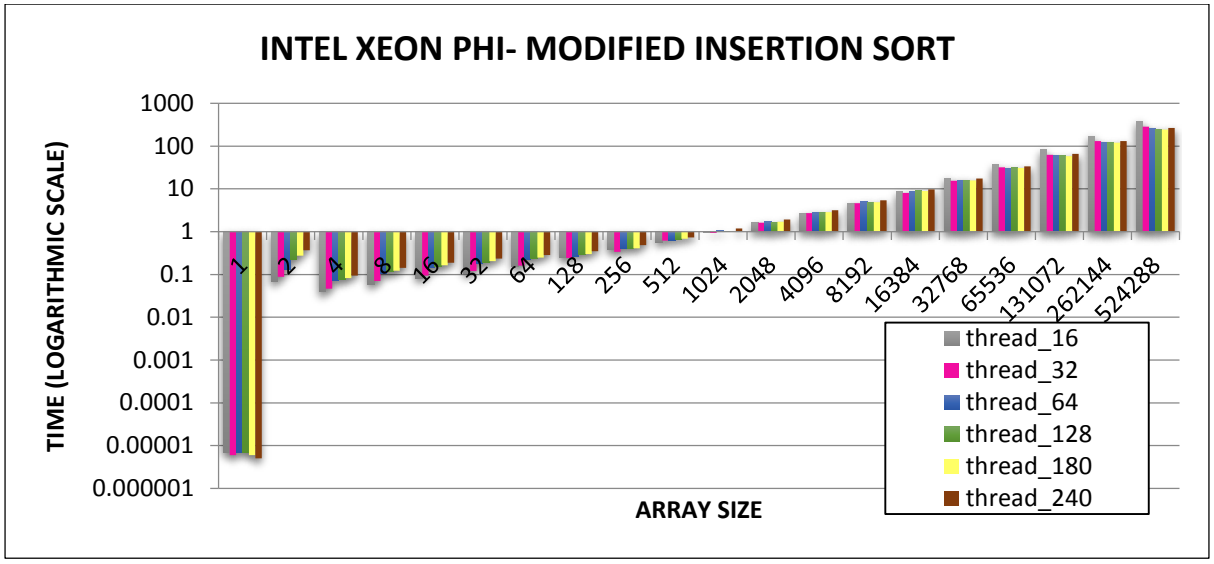

Fig 11: Modified insertion sort on Xeon Phi. Time taken is for 1000 tests.

The fundamental problem lies with the fact that insertion sort is not parallelisable and most importantly not scalable due to its n^2complexity which inhibits the insertion sort from scaling to higher array sizes. It can be observed that the E5 CPU gives a better performance for the insertion sort kernel. Execution insertion sort algorithm, whose algorithm by nature is iterative, is more compatible with the CPU because of its larger cache size and optimized nature for such general purpose workloads.

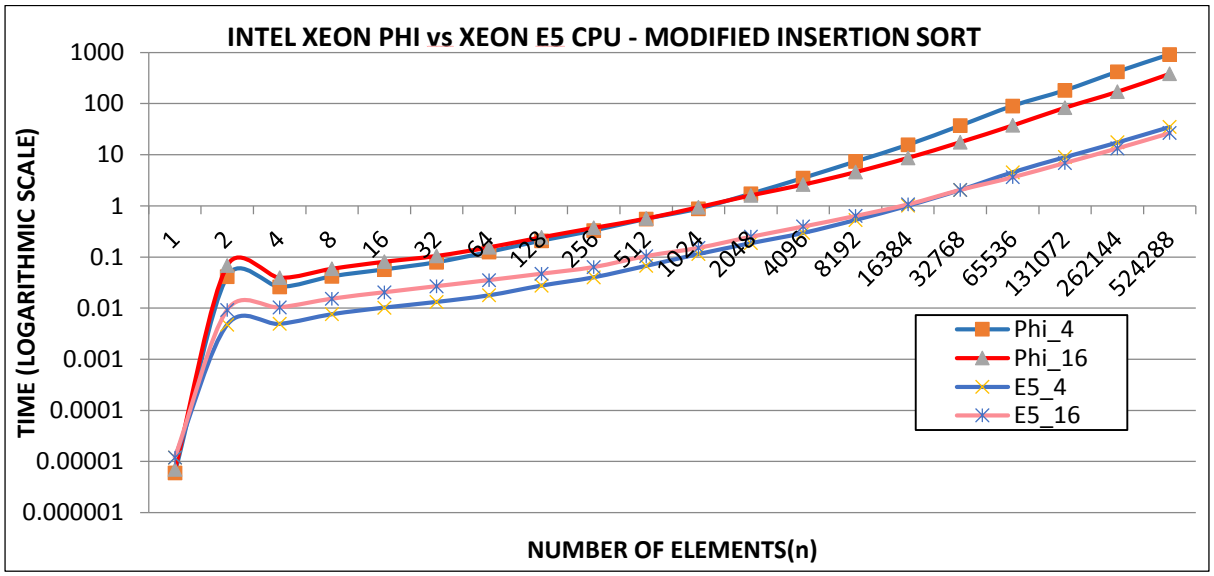

Fig 12: Modified insertion sort on Xeon Phi vs. Xeon E5 CPU. Time taken is for 1000 tests.

The modified version of insertion sort is run on the NVIDIA K20X for thread sizes varying from 8 to 1024. As insertion sort works iteratively, each block had to run on a single thread and the number of blocks were varied to study the performance of the insertion sort kernel. The presence of a serial portion becomes very critical to the performance of the kernel as the

size gets scaled. The performance increases as the thread size is increased as the array size is increased. For arrays of very small sizes fewer threads gives a good performance. For sizes in the range 2^6 to 2^{10} , 128 threads gives the optimal performance. 2^{14} to 2^{20}, 1024 threads gives the optimal performance. 2^{11} to 2^{15}, utilizing 180 threads gives the best performance. The major hindrance to the performance of the NVIDIA K20X for insertion sort is the presence of the serial external sort, which is employed to sort the sorted sub-arrays.

TABLE 10: Insertion sort using CUDA on NVIDIA K20X.

	Time in seconds/1000 tests						
Number of elements(n)	**T=16**	**T=64**	**T=128**	**T=180**	**T=256**	**T=512**	**T=1024**
2	3.9832	4.069888	4.179232	4.357408	4.530496	5.265344	6.625312
4	3.821472	4.051456	4.112128	4.290752	4.472384	5.175232	6.52608
8	3.994336	4.209632	4.25696	4.467808	4.642688	5.351584	6.698176
16	4.617888	4.590432	4.645728	4.849344	5.009664	5.72848	7.09888
32	6.059008	5.518432	5.53616	5.736832	5.88128	6.63888	7.981536
64	8.903072	7.324736	7.304096	8.07712	7.637376	8.436544	9.793472
128	15.28096	12.10653	11.52266	11.54576	12.35747	12.93302	13.81478
256	28.13075	21.77482	20.83965	21.34368	20.59523	21.13846	22.52042
512	57.08707	42.71642	40.70681	40.76403	41.96179	40.84627	41.43818
1024	116.1352	89.85235	85.343	84.40787	86.23945	84.41309	84.13325
2048	242.139	188.6918	179.665	177.5228	179.4481	177.5526	176.7495
4096	510.1568	399.0492	381.3871	376.4966	380.961	376.7484	374.4882
8192	1077.136	846.6828	810.2391	799.9834	809.486	800.121	796.8626
16384	2302.112	1794.419	1720.732	1698.024	1719.23	1703.953	1692.506
32768	4994.723	3796.682	3639.041	3598.364	3635.11	3601.735	3585.863
65536	11182.52	8054.48	7698.051	7606.723	7679.558	7612.582	7577.196
131072	26225.35	17093.01	16256.88	16045.94	16180.63	16025.33	15968.14
262144	66460.17	36590.98	34325.17	33793.63	34108.57	33743.45	33573.84
524288	185592.5	79599.83	72694.45	71294.3	71915.34	70898.38	70490.13
1048576	585641.8	178676.9	155250.4	150799.7	151787.4	148936.6	147785.3

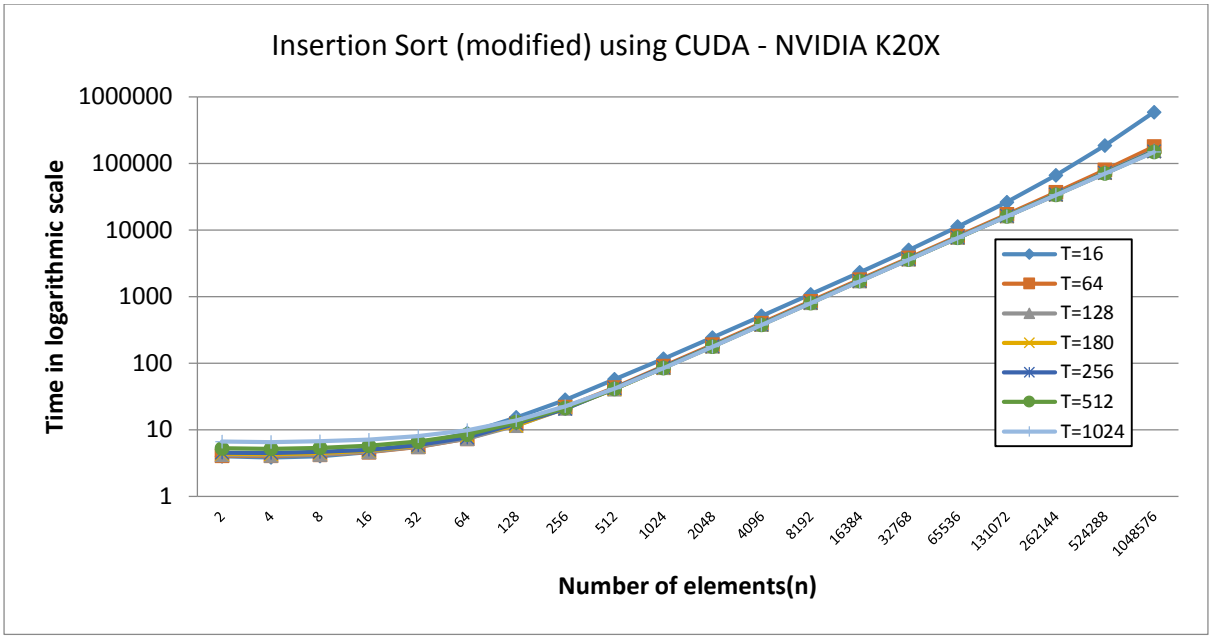

Fig 13: Insertion sort on NVIDIA K20X. Time taken is for 1000 tests.

On comparing the insertion sort kernel amongst the three devices, it can be seen that the for array sizes till 2048, for the same thread size of 26, NVIDIA K20X is faster than both Xeon Phi and E5. But for higher array sizes, the E5 CPU, with the presence of large caches, it performs well for the insertion sort as it requires repeated iteration by each thread. Xeon Phi's poor performance can be owed to higher memory demand placed on each of the distributed cores. The ring topology for communication amongst the 60 cores weighs heavily on the device, making the memory access a cumbersome operation on the Xeon Phi co-accelerator, as the Xeon Phi involves offloading the parallel code from the host to the accelerator.

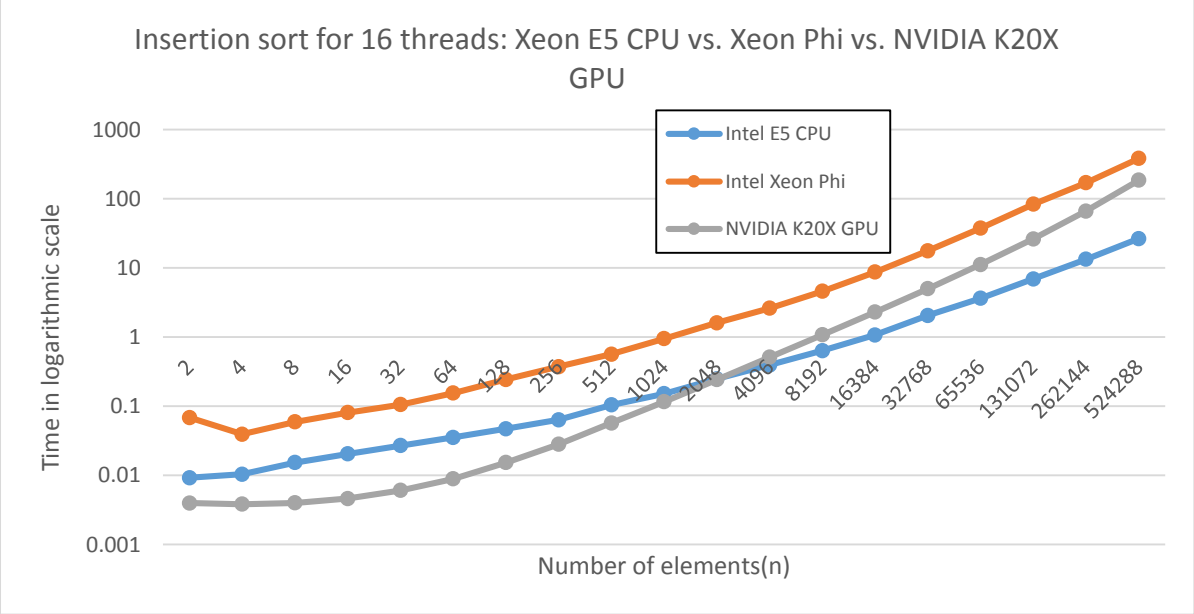

Fig 14: Performance comparison of insertion sort across CPU, Phi and GPU for 16 threads. Time taken is for 1000 tests.

6. Conclusion

The performances of the devices for both kernels have been analysed. NVIDIA's K20X GPU proved to be the right choice for the matrix transposition and while CPU performed better for insertion sort kernel. The memory bound nature of operations of the kernels brought down the performance of Xeon Phi with Xeon E5 2670 CPU performing better.

Acknowledgement

We thank Dr. Alan D. George for providing us with all the resources we needed for the project. We thank Andrew Milluzzi for assisting us in every step of the project and patiently clearing our doubts.

References

[1] "Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Coprocessors" by Andrey Vladimirov for Colfax Research, August 12, 2013.

[2] "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era" By Javier Diaz, Camelia Mun˜oz-Caro, and Alfonso Nin˜o, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 23, NO. 8, AUGUST 2012.

[3] CUDA C programming guide by NVIDIA