Performance Model of HPC Application On CPU-GPU Platform

Bayammagari Jothsna Sai
Graduate Student
Department of ISE
Nitte Meenakshi Institute of Technology

Deepti Keshav Bhandari
Graduate Student
Department of ISE
Nitte Meenakshi Institute of Technology

Aishwarya G Nadiger
Graduate Student
Department of ISE
Nitte Meenakshi Institute of Technology

Monika N
Graduate Student
Department of ISE
Nitte Meenakshi Institute of Technology

Chandrashekar B N
Associate Professor
Department of ISE
Nitte Meenakshi Institute of Technology

***Abstract*—In recent years, the world of high-performance computing has been developing rapidly with enormous efforts in the integration of information technology and research. The emergence of CPU-GPU platform computing has made this possible in a very efficient manner. Nowadays, the graphic processing unit (GPU) delivers much better performance than the CPU, because of a few cores with lots of cache memory on the CPU that can handle a few software threads at a time. In contrast, a GPU is composed of hundreds of cores that can handle thousands of threads simultaneously. The CPU-GPU hybrid platform is becoming increasingly important in high-performance computing(HPC) domains such as deep learning, artificial intelligence, etc., because of its tremendous computing power. In this work, we have proposed a performance model to accelerate the performance of the HPC applications on a hybrid CPU-GPU platform. We have tested and analyzed the proposed performance model using different HPC benchmark applications such as Merge sort and Matrix multiplication on different platforms such as sequential, OpenMP, MPI in a single system, MPI in the cluster, and CUDA. We have Observed that parallel computing in a shared and distributed memory architecture gives better performance than sequential computing. After analyzing we have represented it in the terms of graphs for a better view of the results.**

***Index Terms*—hybrid computing, parallel computing, sequential computing, CUDA, MPI, OpenMP, CPU, GPU**

I. Introduction

High Performance Computation(HPC) utilises supercomputers and computer clusters to solve complex computation problems. Nowadays, high performance computing is an advancing field of study. Due to limited cores availability in CPUs, it does not provide better performance on HPC applications. But GPUs can provide astounding performance on HPC applications due to availability of thousands of cores.

In order to determine performance in various platforms, we analysed the performance results in terms of execution time, FLOPS and CPU/GPU time considering existing benchmark application.

We provided clear information on various devices and computing levels used in this work to analyse performance on considered benchmark application by running in various platforms such as sequential computing, parallel computing on a single system, parallel computing on clusters of CPUs, parallel computing on CPU-GPU platform.

A. *Background*

1) CPU: Central Processing Unit(CPU) is the electronic circuitry that executes instructions comprising a computer program. It performs basic arithmetic and logic operations instructed by the program. A core is a brain of the CPU, it receives instructions and performs operations. It can have multi-cores, for instance, they can have 2 cores, 4 cores, 8 cores. . . and so on.

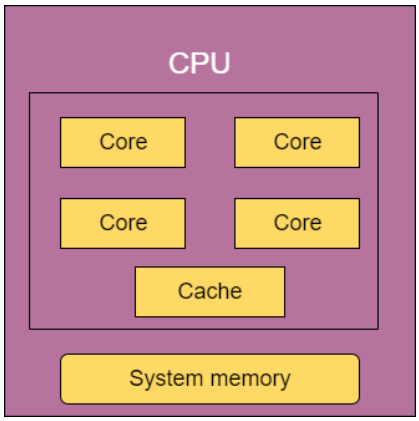

Fig. 1. CPU cores

2) GPU: Graphic Processing Unit(GPU) is popular in today's gaming world. GPUs are more commonly known as graphic cards or video cards. Graphic cards are used for obtaining high quality 2D or 3D images in gaming platforms. GPU in parallel computing is well known for its processing and executing higher complex calculations in lesser time relative to CPU, it is able to do so because of the inbuilt hundred of cores present in them. These cores will have an 'n' number of threads which will perform complex calculations simultaneously.

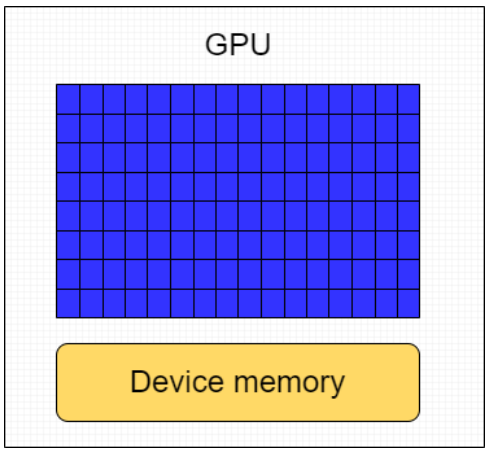

Fig. 2. GPU cores

3) Sequential Computing: In Sequential Computing the execution happens in sequential manner one after the other. Here a program is broken into many instructions and they are executed in the same core one after the other. So, the 2nd instruction waits till the 1st instruction is executed.

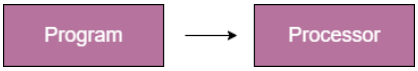

Fig. 3. Sequential computing

4) Parallel Computing: In Parallel Computing, the execution happens in a parallel manner. In this, the program is broken into chunks of instructions and all the broken chunks will start to execute at the same time. Multicores are primarily utilised for the parallel execution. After the execution of instructions in parallel manner the results are combined at the end. In parallel computing, a cluster is formed in order to perform complex calculations simultaneously by harnessing the power of all the systems in a cluster.

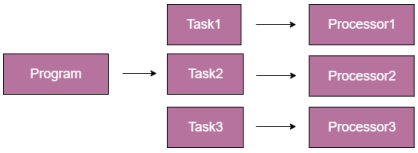

Fig. 4. Parallel computing

The rest of the paper is structured as follows. Section 2, explains the related work. Section 3 Hybrid [CPU-GPU] platforms. Section 4 Proposed performance model. section 5 describes the experimental results. We briefly conclude in section 6.

II. Related Work

In the previous section we studied the various concepts of the CPU-GPU platform. In this section, we will look into the proposed solution that analyses the performance in various platforms by implementing performance calculation formulas in the considered benchmark application.

To analyse better performance of an application we can execute the application in different platforms.

A. *Sequential*

[7] [9] In sequential execution, the execution takes place sequentially from start to finish of a program- opposed to parallel, however, it can handle a restricted amount of workload. In sequential execution, time taken to execute a program will increase. Besides that, if the workload of a program reaches a limit the program execution gets terminated due to lack of availability of cores. Hence, sequential execution can handle a certain limit of workload.

B. *Parallel*

[8] In parallel execution, the execution of a program takes place parallely which in turn gives better performance than sequential according to that of various platforms.

C. *Shared memory*

[11] Shared memory architecture is suitable for parallel computing on various processors sharing single memory. For shared memory systems having multiple multi core processors, all the processors can either connect to main memory directly or each processor can be connected to a block of memory by interconnect. Even though each processor is connected to a block of memory, the processors can have access to other blocks of memory through hardware embedded in the processor. Shared memory architecture for parallel computing gives better performance when compared to sequential computing but not compared to distributed computing when workload is too large to fit in a single machine.

D. *Distributed memory*

[10] [6] [5] Distributed memory model is known as a multi-computer having multiple processor-memory pairs linked together through an interconnection network. Substantially, it describes the large networked cluster of computers each having their private memory. It deals with the applications that are too large to fit in a single machine.

Stijn Heldens, Ana Lucia Varbanescu and Alexandru Iosup has demonstrated [1] at 6th workshop on Irregular Applications in 2016. This paper shows by using dynamic scheduling how they achieved load balancing on hybrid CPU-GPU platform. They have implemented a system of graph-processing called HyGraph.This system increases the performance by scheduling the jobs for CPU and GPU dynamically which outplace the need for workload and provides load balancing. This HyGraph system provides various features like (1) automatically adapt load-balancing nature for CPU-GPU dynamically while executing algorithm (2) hide overlapping computation with hybrid platform communication. HyGraph uses all the resources available to give increased efficiency. This will be further implemented

on the muli-GPUs and distributed cluster.

[2] has been demonstrated by Chandrashekar B.N and Sanjay H.A in 2019,at MECS. In this, using three programming model strategy i.e MPI, OpenMP and CUDA they have developed a framework which predicts asymmetric work load between CPU and GPU. The tools used to deploy the framework are NVIDIA TESLA M2075 and NVIDIA QUADROK2000. To test their model they have used HPC applications like N dynamic vector addition and saxpy. This framework was then compared with Simplice Donfack Hybrid Programming model. There was an mid-range of performance increase of 76.06

Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk and Wen-mei W. Hwu has demonstrated "Optimization Principles and Applications Performance Evaluation of a Multithreaded GPU Using CUDA" [3] at Center for Reliable and High-Performance Computing in 2008. In this work they have used CUDA to evaluate the performance of GeForce 8800 GTX. The main challenge they have faced is to make the right balance of threads resource usage balance and see how many threads are active at the same time. The features which helped them to get the principles of optimising are using more number of cores and using many threads to hide the latency,using local memory to make less severe pressure on global memory bandwidth. Over CPU execution there will be speedup in application kernels if there is no available or limited resources. By the time they published this paper there was two updated version of CUDA which was released. They had to improvise in the resource and optimizing configuration. Then planned to explore more methods to increase the performance applications.

Michal Marks and Ewa Niewiadomska-Szynkiewicz has demonstrated [4] in 2014. In this paper they have addressed the problems associated with distributed computational system and applications while using mixed CPU/GPU. They have formed a cluster of hybrid form from different vendors. Used two software platforms to solve complex problems and engineering problems namely the first platform is HGCC is for preprocessing the data and the second is the extended version of the first platform which is HGCVC. To do high complex calculations, more liable software and hardware should be available which has to be more scalable. Using two technologies they have proved that it is very useful in parallel computations. In future they have planned to use the two platforms for broad range of problems.

III. CPU-GPU PLATFROMS

A. MPI architecture

MPI is an acronym for Message Passing Interface. MPI is a standard used for parallel computing in distributed memory architecture. Distributed memory architecture is when a cluster of systems has each memory unit to its processor. Programs written in C, C++, Fortran can be run in MPI. With the help of MPI processors involved in parallel computing can communicate with each other. In this work, we have installed MPICH 3.3.2. Here MPICH stands for Message Passing Interface Chameleon, which is a free implementation of MPI. In the MPI architecture, one system works as master and remaining as slave. The master system will divide the workload in to chunks of tasks, loads the chunks on to slaves through the communication network for computation. Likewise, execution takes place parallely. Master system keeps on detecting statistics and metrics of slave nodes(systems) such that if any node shows abnormal situations, masters transfers the chunks on existing node to other or new node to avoid conflicts. Once the data gets executed in all nodes the results will be combined and displayed in the master node

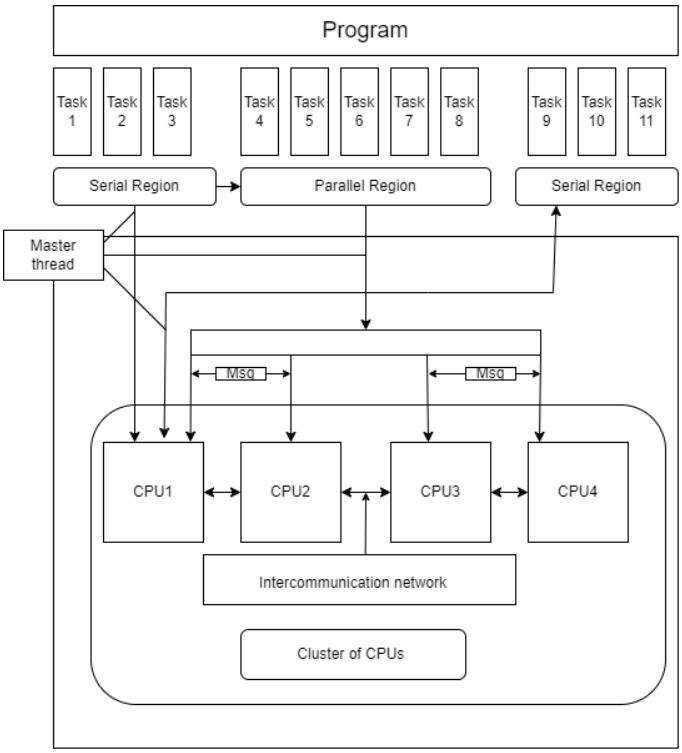

Fig. 5. MPI Architecture

Fig 5 shows the architecture of MPI. In MPI we use Distributed Memory and the cores in the distributed memory will connect through interconnected network. Through this network the cluster of systems will communicate with each other.

B. *OpenMP*

Fig 6 depicts the architecture of OpenMP. Here all the CPU cores shares the same memory space and there are operating system threads which run multiple applications at a time given by user [12].

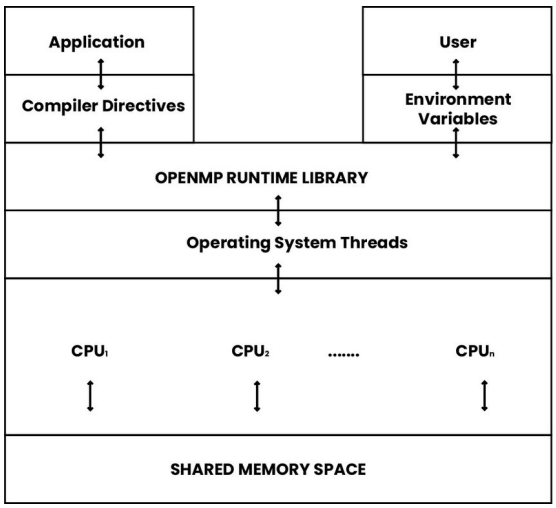

Fig. 6. OpenMP Architecture

All the operating system threads run in parallel have access to a common shared memory space. These threads are controlled by the run-time library present in the architecture. At a given time, a set of threads will be executing the same code. Execution of a OpenMP program starts with a master thread. A set of compiler directive and library routines helps in parallelism in shared memory architecture. Compiler directive is available in the application code. Environment variables helps to change features of OpenMP applications, such as number of threads and to control loop iteration scheduling.

C. CUDA

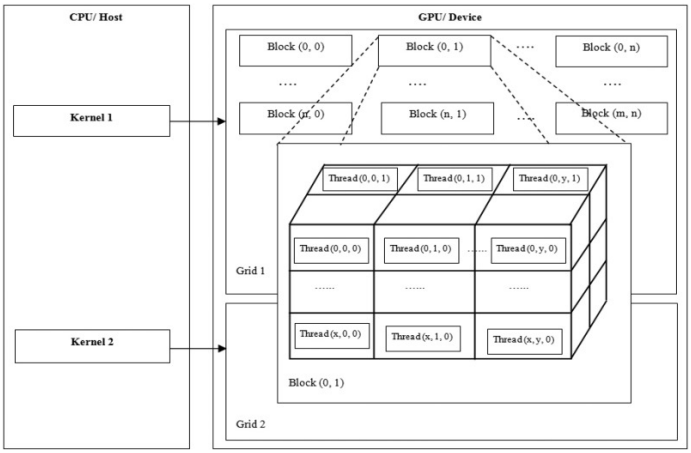

Fig. 7. CUDA Architecture

Fig 7 shows the architecture of CUDA. Here both CPU-GPU is used for execution. CPU acts as host. It will assign all the work to GPU for parallel computation.

Compute Unified Device Architecture is the fullform of CUDA. It is a parallel computing platform as well as a programming model for general computing on GPUs. NVIDIA are the distributors of CUDA. By harnessing the power of GPUs, we can speed the execution of complex applications by CUDA. C, C++, Fortran are used to write programs in CUDA.

We have made use of the latest version available during our work i.e CUDA 11.4. Every GPU installed in the system needs a driver. Driver is nothing but a set of files required to command over a hardware. Driver version installed for our work is 470.103.01.

Apparently CUDA consists of hundreds of cores, where each core acts as a multiprocessor. Total available threads will be arranged into block of threads. There is limit on number of threads per block. Since all the threads in the block must share a limited memory resources. Recent GPUs support up to 1024 threads per block. Blocks are arranged into one-dimensional, two-dimensional, three-dimensional grid of thread blocks. Each block within the grid can be accessible using a unique index called blockIdx variable.Similarly thread can be accessible using threadIdx. Each thread blocks execute independently. These blocks can be executed in parallel or sequential manner.

IV. Proposed Performance Model

A. Performance Evaluation

Initially, we considered merge sort as a benchmark application. We included execution time, FLOPs, CPU/GPU time formulas mentioned below in the program.

FLOPS of CPU tells us no of operations can a CPU perform in a second by considering the execution time. Here, we calculated FLOPS in gigabytes having a number of operations in a merge sort and execution time.

$$CPU_{GFLOPS} = \frac{\text{No. of operations}}{\text{execution time}} \times 10^{-9} \tag{1}$$

where,

no. of operations = number of arithmetic operations in a program,

CPU_{GFLOPS} = no. of giga floating point operations performed in a second by CPU

execution time = Response time of a program

CPU time is the time taken by CPU for processing a computer program which will be less than response time. We calculated the CPU time by considering workload i.e., size and CPUGFLOPS.

$$T_{CPU} = \frac{Workload(Size)}{CPU_{GFLOPS}} \tag{2}$$

where,

T_{CPU} = CPU processing time

Workload = Size of a program

FLOPS of GPU tells us no of operations can a GPU perform in a second by considering the execution time. Here, we

calculated FLOPS in giga having a number of operations in a merge sort and execution time.

$$GPU_{GFLOPS} = \frac{\text{No. of operations}}{\text{execution time}} \times 10^{-9} \quad (3)$$

where,

GPU_{GFLOPS} = no. of floating point operations performed in a second by GPU

GPU time is the time taken by GPU for processing a computer program which will be less than response time. We calculated the GPU time by considering workload i.e., size, GFLOPS of CPU.

$$T_{GPU} = \frac{\text{Workload(Size)}}{GPU_{GFLOPS}} + \text{Transfer time} \quad (4)$$

where,

T_{GPU} = GPU processing time

Transfer time = communication time between CPU and GPU

Transfer time is the time taken to send instructions to the GPU by CPU through the bus for execution.

$$Transfertime = \frac{workload}{BusSpeed} \quad (5)$$

where,

Bus speed = speed of PCI(peripheral component interconnect)

V. Experimental setup and Results

The experimental setup describes how we have arranged and configured different hardware and software for our work.

For executing benchmark applications in sequential platform, system must have gcc compiler installed in it.To verify the installation of gcc compiler run the command "gcc –version". Similarly, executing benchnmark applications in OpenMP platform, system must be installed with gcc compiler as OpenMP takes support from gcc in compiling and running programs. Once the gcc compiler is installed in our system, we should check if OpenMP features are configured or not using the command "echo cpp -fopenmp -dM —grep -i open" .

1) Setting up MPI Cluster: We have setup a cluster of two systems. To make a cluster of two computer they must be connected to a network for communication of the two computers. So we have connected both the computers to switch with Ethernet cables as they are rated for maximum bandwidth. After the physical set up we have followed the following steps:

1) Set up network addresses in Settings.
2) Install SSH server on both systems which are forming the cluster.To enable password-less SSH access, we extract a public key from the master node and copy it to the " /.ssh/authorized-keys" file on the slave node.
3) Remote login can be tested by using command "ssh machineName".
4) Install latest version of MPICH (Message Passing Interface Chameleon) in your system.
5) MPI Programs can be compiled using the command "mpicc programname -o outputfilename"
6) MPI Programs can be run using following command using the command "mpirun -np numofprocessor ./outputfilename"

2) Installing CUDA: In order to have CUDA platform in our system NVIDIA GPU must be installed in your computer as a pre- requisite. For the installed NVIDIA GPU system must have a software driver, which controls GPU hardware functions. CUDA toolkit is installed as it provides necessary facilities to develop a GPU accelerate application. The following command demonstrates the step-wise installation of CUDA:

1) Install gcc and all the essential tools from the official package repository of Ubuntu
2) Install CUDA from the official Ubuntu package repository using "sudo apt install nvidia-cuda-toolkit" command.
3) Confirm the installation of CUDA using the command "nvcc –version".
4) Install the latest version of CUDA from the official NVIDIA package repository
5) Add the GPG key of the official NVIDIA package repository -keys
6) Add the NVIDIA CUDA package from the official repository.
7) Update the apt package repository cache and all the existing packages of your Ubuntu 20.04 LTS machine.
8) Install the latest version of CUDA from official package repository, with this latest version of CUDA along with required dependency will be downloaded.
9) Add CUDA to the path by creating a new script cuda.sh.
10) After changes reboot your computer
11) Edit /etc/sudoers configuration file , to run files with superuser privileges.

A. HPC Application

HPC stands for High Performance Computing. Applications which require multi-threaded and multi-process models for successful execution of the task incorporated within them are known as HPC applications. One of the finest example for HPC solutions is supercomputers. In comparison with normal desktop with a 3GHz processor which can perform 3 billion calculations per second, supercomputers are capable of quadrillions of calculations per second. HPC applications have been used in variety of fields and industries like research labs, machine learning, artificial intelligence, financial services like stocks, media and entertainment.

1) Matrix Multiplication: HPC benchmark application we have used in our work is matrix multiplication.The objective of matrix multiplication is two multiply matrix and output

a product matrix. In this case we have used square matrix of 'N' input size. This HPC application multiplies two NxN matrices and produces a NxN matrix. We know that small input size can be easily handled by processor with few inbuilt cores, but as the input size becomes larger in magnitude it becomes impossible for a normal computer to execute the program successfully. So supercomputers with large number of cores and their combined capabilities execute the matrix multiplication program without any breakdown in the performance. Matrix multiplication includes set of for loops which can be divided among available cores. And the cores execute the assigned tasks in parallel providing improved execution time leading to a better performance.

B. Evaluating on various platforms

1) Platform 1 - Sequential: In sequential computing, the program execution happens sequentially. We ran the matrix multiplication program sequentially in a single system and noted execution time, FLOPS of CPU, CPU time on increasing input size. As input size increases, we noticed the decrease in performance and results in core dumped once the input limit exceeds due to limited cores.So, the sequential computing will not give better performance for large input size.

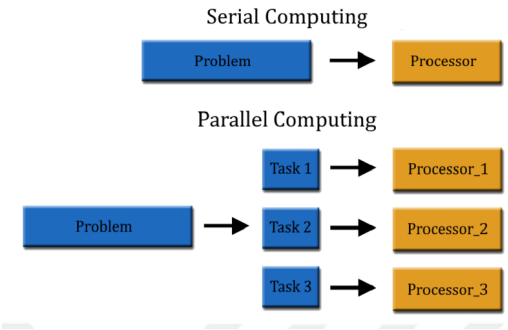

Fig. 8. Sequential Computing

2) Platform 2 - OpenMP: In shared memory architecture, parallel computing will be done by using OpenMP library. We executed a matrix multiplication program parallelly in a single system using the OpenMP library. Here, execution takes place parallelly in the processors by sharing single memory. It uses Fork and Join model, where the program is divided into multiple chunks and each chunk is executed by each processor. It gives better performance when compared to platform 1(sequential). But due to limited cores the core will get dumped after reaching a certain input size limit.

3) Platform 3 - MPI(shared memory): By considering the MPI library, we executed the program in a single system. As MPI is suitable for distributed systems we want to show the performance if MPI is used in a shared memory architecture i.e., single system. As expected the openMP gives better performance than MPI for a shared memory architecture.

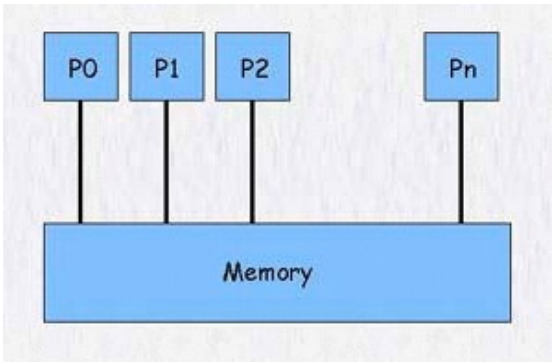

Fig. 9. Sequential Computing

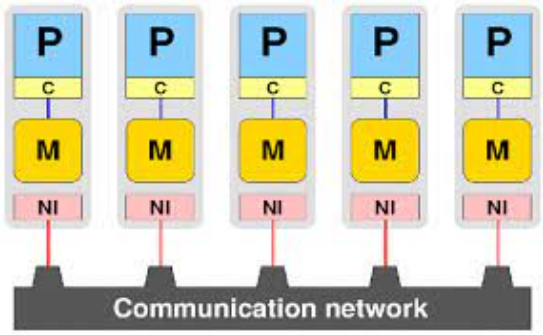

Fig. 10. MPI in shared memory architecture

4) Platform 4 - MPI(Distributed memory): MPI is a suitable library to execute the program in a distributed memory architecture. We executed the program in a CPU-CPU cluster and noted the results to analyse the performance. As input size is more complex, it gives better performance when compared to OpenMP in shared memory architecture. But during certain inputs OpenMP shows better performance due to network communication delay in distributed memory architecture(CPU-CPU cluster).

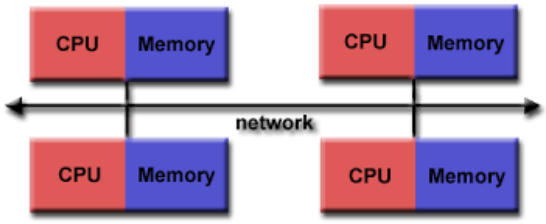

Fig. 11. MPI in Distributed memory architecture

5) Platform 5 - CUDA(CPU-GPU platform): We then considered running the program on the CPU-GPU platform as the GPU contains thousands of cores. Even though there is an increase in input size of a merge sort program the execution time is exiguously less due to thousands of cores. When compared to above platforms the CPU-GPU platform showed better performance.

Fig. 12. CUDA in CPU-GPU Platform

C. *Results and Discussion*

We considered matrix multiplication as the benchmark application. Input size is the size of an mxn square matrix. Here, we considered execution time which is the start and end of execution of a given program, GFLOPS, gives the number of operations CPU/GPU can perform per second, and CPU time gives the start and end time of a program. We executed a matrix multiplication benchmark program taking different input sizes in different platforms and analysed the performance.

Table I shows the results for sequential computing. Increase in GFLOPS, indicates increase in the amount of operations executed per second. And more the GFLOPS, less the CPU time. But here as the input size increases, core dumps due to finite cores.

Table II shows the results for openMP. It is a library for parallel computing in shared memory architecture. After analysing the execution time, GFLOPS, and CPU time values, as input size increases it will show better performance than sequential.

Table III shows the results for MPI in shared memory i.e. in a single system. To compare how the results will vary using openMp and MPI library in shared memory we found that openMP is more suitable for shared memory architecture than MPI library.

Table IV shows the results for MPI in distributed memory, since MPI in a single system did not give efficient results, we clustered two systems to get better performance. We got efficient results compared to all the platforms here.

	Sequential		
Input size	**Execution Time (sec)**	**GFLOPS of CPU (sec)**	**CPU Time(sec)**
150	0.017343	65,678.51583	0.00228385
200	0.048584	1,31,730.6109	0.00151825
300	0.112842	6,46,036.0504	0.00046437
500	0.571663	27,33,253.683	0.00018293
800	4.661697	56,23,359.905	0.00014226
1000	core dumped	-	-
2000	core dumped	-	-
4000	core dumped	-	-
8000	core dumped	-	-

TABLE I
PERFORMANCE RESULTS OF SEQUENTIAL PROGRAMMING

	OpenMP		
Input size	**Execution Time (sec)**	**GFLOPS of CPU (sec)**	**CPU Time (sec)**
150	0.013974	81512.98841	0.0018402
200	0.031179	205266.3652	0.00097434
300	0.070681	1031394.576	0.00029087
500	0.326761	4781782.404	0.00010456
800	2.012232	13027523.67	0.00006141
1000	4.318787	23154649.67	0.00004319
2000	32.8254	194970967.6	0.00001026
4000	344.8599	1187728698	0.00000337
8000	6368.4946	4116263206	0.00000194

TABLE II
PERFORMANCE RESULTS OF PARALLEL COMPUTING IN SHARED MEMORY USING OPENMP

	OpenMP		
Input size	**Execution Time (sec)**	**GFLOPS of CPU (sec)**	**CPU Time (sec)**
150	0.013974	81512.98841	0.0018402
200	0.031179	205266.3652	0.00097434
300	0.070681	1031394.576	0.00029087
500	0.326761	4781782.404	0.00010456
800	2.012232	13027523.67	0.00006141
1000	4.318787	23154649.67	0.00004319
2000	32.8254	194970967.6	0.00001026
4000	344.8599	1187728698	0.00000337
8000	6368.4946	4116263206	0.00000194

TABLE III
PERFORMANCE RESULTS OF PARALLEL COMPUTING IN SHARED MEMORY USING MPI

	MPI(Cluster)		
Input size	**Execution Time (sec)**	**GFLOPS of CPU (sec)**	**CPU Time (sec)**
150	0.019159	59453.12908	0.002523
200	0.0359	178272.9805	0.00112188
300	0.09086	802333.26	0.00037391
500	0.51091	3058268.58	0.00016349
800	1.83511	14284920.25	0.000056
1000	2.96607	33714645.98	0.00002966
2000	28.99138	220755272.8	0.00000906
4000	467.85188	875490764.3	0.00000457
8000	5150.640158	5089542114	0.00000157

TABLE IV
PERFORMANCE RESULTS OF PARALLEL COMPUTING IN DISTRIBUTED MEMORY USING MPI

Table V shows the results for the CPU-GPU platform using CUDA. Here GFLOPS, CPU time, and Transfer time is considered. Transfer time is the communication time between CPU and GPU.

	CUDA		
Input size	**Execution Time (sec)**	**GFLOPS of GPU (sec)**	**GPU Time(sec)**
150	0.000451232	2524338.921	0.0000594402
200	0.000530784	12057635.5	0.000016612
300	0.00150035	48588662.65	0.0000062118
500	0.00664198	235246116.4	0.0000021879
800	0.02307216	1136191843	0.0000008041
1000	0.04750726	2104941434	0.0000006001
2000	0.34083462	18777435227	0.0000003565
4000	2.8300607	144731878012	0.0000005276

TABLE V
PARALLEL COMPUTING IN CPU-GPU PLATFORM USING CUDA

After executing in all the platforms we found that, we will get better performance and efficient results in CPU-GPU platform for enormous input size .

We analysed the performance results of matrix multiplication programs on various platforms upon increasing input size. In comparison of all platforms, sequential is a good fit for small input size, shared memory parallel computing gives better performance for moderate input size, but once we reach core dump then distributed memory parallel computing will execute the program giving better performance. We see that program execution depends on cores availability, hence the CPU-GPU platform which contains thousands of cores gives greater performance out of all platforms.

Fig 13 shows the relation between execution time and input size. As input size increases the CPU-GPU platform(CUDA) gives better performance in terms of execution time than other platforms.

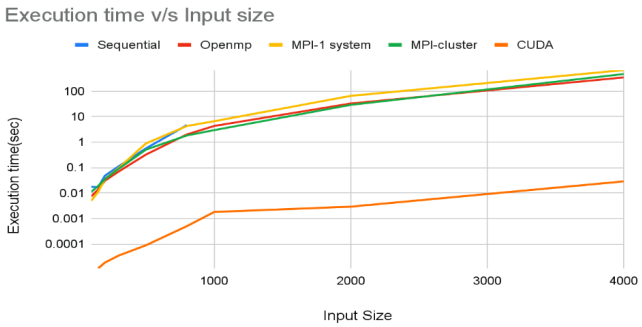

Fig. 13. Input size v/s Execution time

Fig 14 shows us number of floating point operations that CPU/GPU are executing in a second as input size increases. We can see the CPU-GPU platform(CUDA) executes more floating point operations per second when compared to other platforms.

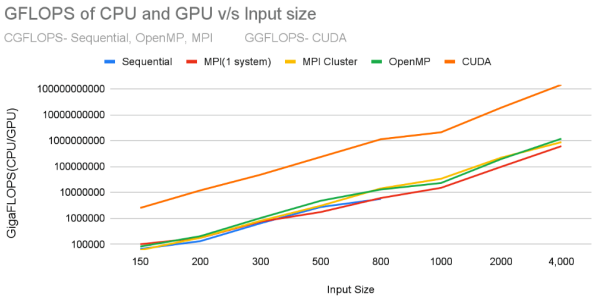

Fig. 14. Input size v/s GigaFLOPS

VI. CONCLUSION AND FUTURE SCOPE

In this work, we have analysed the performance in various platforms, and we have found that parallel computing in a shared memory architecture gives better performance than sequential computing. MPI library will be used for distributed computing for better performance than in a shared memory architecture. Finally, GPU gives better performance results than sequential, parallel in CPU shared memory architecture and CPU-CPU cluster i.e., distributed memory architecture due to the enormous number of cores present in the GPU.

Our future work is to analyse the performance on clusters of GPUs upon increasing workload.A dynamic workload strategy can also be proposed for dividing workload among CPU and GPU. Another future scope would be utilising the virtualisation power of cloud computing for forming a parallel computing platform in our work. Cloud computing will provide us nearly two or three times more resources than physical setup of systems. So with the help of cloud computing virtualisation we can provide more workload and also we can expect more throughput.

REFERENCES

[1] Heldens, S., Varbanescu, A. & Iosup, A. Dynamic load balancing for high-performance graph processing on hybrid cpu-gpu platforms. *2016 6th Workshop On Irregular Applications: Architecture And Algorithms (IA3)*. pp. 62-65 (2016)

[2] Chandrashekhar, B. & Sanjay, H. Performance framework for hpc applications on homogeneous computing platform. *International Journal Of Image, Graphics And Signal Processing*. **11**, 28 (2019)

[3] Ryoo, S., Rodrigues, C., Baghsorkhi, S., Stone, S., Kirk, D. & Hwu, W. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. *Proceedings Of The 13th ACM SIGPLAN Symposium On Principles And Practice Of Parallel Programming*. pp. 73-82 (2008)

[4] Marks, M. & Niewiadomska-Szynkiewicz, E. Hybrid CPU/GPU Platform For High Performance Computing.. *ECMS*. pp. 508-514 (2014)

[5] Chen, W. Dynamic Workload Division in GPU-CPU Heterogeneous Systems. (The Ohio State University,2013)

[6] Ma, K., Li, X., Chen, W., Zhang, C. & Wang, X. Greengpu: A holistic approach to energy efficiency in gpu-cpu heterogeneous architectures. *2012 41st International Conference On Parallel Processing*. pp. 48-57 (2012)

[7] Hassani, R., Aiatullah, M. & Luksch, P. Improving HPC application performance in public cloud. *IERI Procedia*. **10** pp. 169-176 (2014)

[8] Syberfeldt, A. & Ekblom, T. A comparative evaluation of the GPU vs the CPU for parallelization of evolutionary algorithms through multiple independent runs. *International Journal Of Computer Science Information Technology (IJCSIT) Vol*. **9** (2021)

[9] Rajsbaum, S. & Raynal, M. Mastering concurrent computing through sequential thinking. *Communications Of The ACM*. **63**, 78-87 (2019)

[10] Sanders, J. & Kandrot, E. CUDA by example: an introduction to general-purpose GPU programming. (Addison-Wesley Professional,2010)

[11] Kalamkar, D., Georganas, E., Srinivasan, S., Chen, J., Shiryaev, M. & Heinecke, A. Optimizing deep learning recommender systems training on cpu cluster architectures. *SC20: International Conference For High Performance Computing, Networking, Storage And Analysis*. pp. 1-15 (2020)

[12] Chandrasekhar B N., Sanjay .H.A ., Performance Study of OpenMP and Hybrid Programming Models on CPU-GPU Clusters. *SC20: Fifth Scopus International Conference on 'Emerging Research in Computing, Information, Communication, and Applications, (ERCICA-2018)* . springer publisher

[13] B. N. Chandrashekhar ., H. A. Sanjay., Performance Framework for HPC Applications on Homogeneous Computing Platform. *International Journal of Image, Graphics and Signal Processing (IJIGSP) MECS Press Publishers Vol. 11, No. 8, pp,28-39,2019, DOI: 10.5815/ijigsp.2019.08.03*.

[14] B. N. Chandrashekhar ., H. A. Sanjay., Dynamic Work Load Balancing for Compute Intensive Application Using Parallel and Hybrid Programming Models on CPU-GPU Cluster. *Journal of computational and theoretical Nanoscience American scientific Publishers Volume 15, Numbers 6-7, June 2018, pp. 2336-2340(5), DOI: https://doi.org/10.1166/jctn.2018.7464*.

[15] B. N. Chandrashekhar, H. A. Sanjay., Performance Analysis of Sequential and Parallel Programming Paradigms on CPU-GPUs Cluster. *IEEE 3rd International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV 2021) India — 978-1-6654-1960-4/20/2021 IEEE — DOI: 10.1109/ICICV50876.2021.9388469* .

[16] B. N. Chandrashekhar, H. A. Sanjay., Performance Evaluation of CPU-GPU with CUDA Architecture. *International Conference on Communication and Computing (ICC-2015) Bangalore, India, on 9th to 11th July 2015. Published in PP 265-274 McGraw - Hill Education 2015* .

[17] Noaje, Gabriel, Michael Krajecki, and Christophe Jaillet, Multi-GPU computing using mpi or OpenMP. *Intelligent Computer Communication and Processing (ICCP)IEEE International Conference 2008* .

[18] G. Teodoro, E. Valle, N. Mariano, R. Torres, W. M. Jr, and J. H. Saltz, Approximate similarity search for online multimedia services on distributed CPU-GPU platforms. *Intelligent Computer Communication and Processing (ICCP)IEEE International Conference 2008* .

[19] A. Brodtkorb, Simplified ocean models on the GPU. *Norsk Informatikkonferanse, 2018* .

[20] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk, Merge pathparallel merging made simple. *Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International. IEEE, 2012, pp. 1611–1618* .

[21] O. Green, R. McColl, and D. A. Bader,Gpu merge path: a gpu merging algorithm. *in Proceedings of the 26th ACM international conference on Supercomputing. ACM, 2012, pp. 331–34010. Grama A., Gupta A., Karypis G., and Kumar V, Introduction to Parallel Computing, 2nd ed., Addison-Wesley, 2003* .