ISSN 2395-1621

CORBA Based Distributed Framework for GPGPU Processing

Shamim Akhter, Tawsif F. Rahman, Md. Rakib Bahadur, Monirul Islam, Yasser Khan, Shantanu Kumar Rahut

shamimakhter@gmail.com,
shantanurahut@gmail.com

Department of Computer Science and Engineering, International University of Business, Agriculture and Technology, Dhaka, Bangladesh
Department of Computer Science and Engineering, East-West University (EWU), Dhaka, Bangladesh.

ABSTRACT

GPU based systems are in demand due to their massively parallel architecture with thousands of cores to handle multiple tasks simultaneously and improve the application performance dramatically. However, GPU based system implementation requires specific hardware and software supports, and thus it is costlier compared to common machines. To solve this problem, the Common Object Request Broker Architecture (CORBA) based distributed framework can be implemented. In this paper, we are proposing and implementing a language and platform-independent distributed framework, which enables GPGPU processing as a service from a remote host to common CPU, enable clients. Besides, the adaptive merged sort is taken as an example application and implemented on GPU based parallel system with a novel approach.

ARTICLE INFO

Article History

Received:23rd December 2020

Received in revised form :

23rd December 2020

Accepted: 29th December 2020

Published online :

6th January 2021

Keywords: GPGPU, CORBA, Distributed Framework, Adaptive Merge Sort

I. INTRODUCTION

Compute and process a humongous amount of data set in minimal time is always an attraction to the programmers. Recently, performance improvement by reducing time is one of the major research domains. In this regard, parallel applications including -Abinit [1], Accelereyes Arrayfire[2], Acceleware AxRecon[3], etc are implemented over the Graphics Processing Unit (GPU)[4] and the number is growing. The term GPU was arrived by NVIDIA in 1999, and it was presented as a "single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines". Compute general computational work using GPU is known as GPGPU (General-purpose GPU). Today's GPU is an integral part of a graphics card and has hundreds of cores that can execute multiple numbers of instructions in parallel. Their performance is also far greater than modern CPUs with 4 or 8 cores. However, GPU based system (GPU enable graphics card) needs specific H/W and S/W supports including OS version, RAM size, Bandwidth, CPU types, etc., to do GPGPU processing. The additional cost is another issue for such system implementation. Let's take an example of a parallel computing lab with several workstations and installed different OSs, H/Ws, and S/Ws. Thus, implementing the GPU enables graphics card may require changing/replacing the available H/W and S/W settings. This may also include additional costs. An alternative to this situation is to implement a distributed framework where two/three workstations (hosts) can be able to do GPGPU processing and the rest can work as a client to request GPGPU servicing from host PCs. It can support to reduce the additional cost as well as provide an infrastructure (GPU based system) as a service to GPU unable devices.

Thus, In this work, we are proposing a platform-independent distributed framework that enables GPGPU processing as a service from a remote host to common CPU enable clients. The framework is also implemented by a Java-based CORBA language implementation. CORBA supports language independence and mobility (platform independence). Common Unified Device Architecture (CUDA) based parallel programming API is chosen at hosts to support the GPGPU processing. Besides, we choose a novel application-adaptive merge sort [5][6] which has not yet been implemented in a parallel system. A new parallel adaptive merge sort algorithm is designed especially for GPU-based systems and the performance improvement is tested on the proposed distributed framework.

The remainder of the work is organized as follows: Section II introduces the background study and state-of-the-art technologies. Section III gives tools, application, and implementation details. Section IV shows the results and performance analysis of GPU-based parallel applications. Finally, the concluding remarks are presented in the conclusion section.

II. BACKGROUND STUDY

A distributed system is implemented with some interconnected (communicate and/or coordinate) independent computers within a communication network. It is organized through the middleware that runs on all machines but offers a uniform interface to the system. Middleware is a kind of software that serves to connect separate and already existing programs or software components. Enterprise applications and web services [7] are examples of such software components. Different middlewares are available for different applications to include- Remote Procedure Call (RPC), Remote Method Invocation (RMI), Common Object Request Broker (CORBA), Distributed Common Object Model (DCOM), etc. CORBA is a software standard that is defined and maintained by the Object Management Group (OMG). It is an architecture and specification that creates, distributes, and manages distributed program objects in a network. It also allows programs at different locations and developed by different vendors can communicate in the network.

A. CORBA System Architecture

CORBA (Common Object Request Broker Architecture) automates many common network programming tasks including object configuration (registration, location, and activation), requests and responses handling (request demultiplexing, parameter marshaling and unmarshalling, stream serializing, operation dispatching), error handling (framing and error detecting and recovering), etc. Fig. 1 presents the CORBA architecture, including client and server communication over IIOP protocol.

In Fig.1, the CORBA architecture includes the Interface Description Language (IDL) language and the platform-independent ORB (Object Request Broker) interface. ORB offers a useful approach for deploying open, distributed, heterogeneous computing solutions to support transparent communication(request-response) between all connected objects located locally or remotely. IIOP (Internet Inter-ORB Protocol) is an ORB transport protocol that enables network objects from multiple compatible ORBs to communicate transparently over TCP/IP. CORBA can establish secure communications channels between clients and object services by allowing a high-level security framework including the authentication and access control of remote users and services, etc.

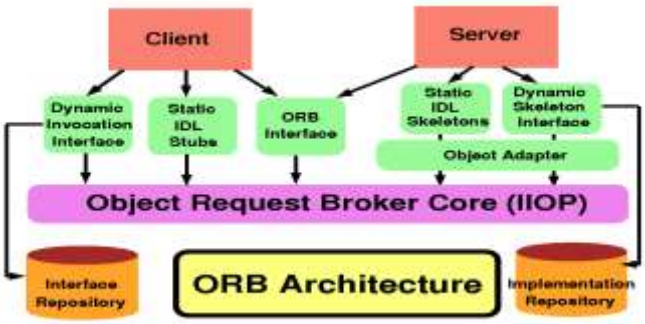

Fig 1. CORBA Architecture

B. GPU Based System Architecture

GPU multiprocessors are worked as co-processors. When the CPU invokes a kernel call for GPU that kernel call executes in parallel several times in GPUs cores. For an instance, the number of tasks a GPU can execute in parallel depends on its architecture including the number of SM (stream multiprocessors) and cores per SM called stream processor (SP), and memory (registers, global, shared). Each SM is allotted with an equal number of cores/streaming processors (SPs). Upon receiving a kernel call or an execution command from the CPU, the GPU SMs are awakened and distributed with an equalized workload of "responsibilities" which are referred to as "kernel". Each kernel is structured with several BLOCKS and several THREADS. SPs are only capable to handle threads. GPU kernel is distributed into SMs, the SMs would distribute all the instructions residing in the kernel to all available SPs.

Under every multiprocessor, there is a large number of 32-bit registers. Register memory is the fastest memory among all other memories in the GPU system. Each thread will be assigned to a set of registers and uses them for fetching and storing data/instructions. Shared memory is comparatively slower than registers but sharable between threads in a block. Because it resides in a chip, it has a higher bandwidth than global or local memory. It can be compared to an L1 cache in a regular CPU. It shares a 64k memory segment per SM. Global memory resides on the device but off-chip from the multiprocessors. Because of that access to global memory is much costlier than accessing shared memory. All threads from any SMs can access global memory. Local memory is the private memory for each thread execution. Local memory is also off-chip and resides on the device. These memories are allocated to the thread when kernel execution needs more memory than registers to hold the thread's local data. Constant memory is accessed like cached. Each multiprocessor cached an amount of constant memory (64k), so that repeated reading from constant memory will be faster.

Fig. 2 shows the architecture of NVIDIA GeForce GTX 650. NVIDIA provides GPU massive parallelism platform named CUDA (Compute Unified Device Architecture). CUDA supports a heterogeneous programming model- where the kernel threads execute at the device and implement with CUDA enable APIs and rest of the program run at CPU in C language. CUDA programming also maintains two separate memory spaces (DRAMs), one is CPU memory and the other is device memory. Therefore, a CUDA program manages the global, constant, and shared memory spaces visible to kernels through built-in API calls including device memory allocation and deallocation as well as data transfer between CPU and device memory.

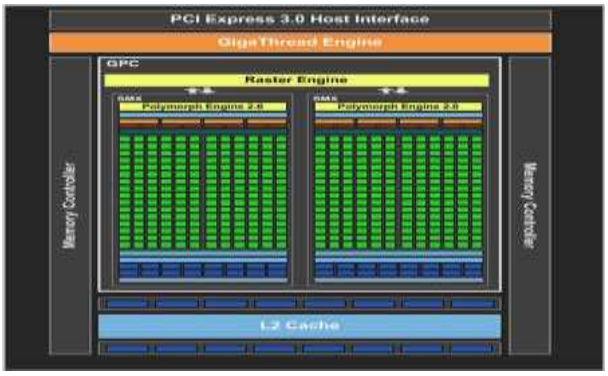

Fig 2. NVIDIA GeForce GTX 650 Architecture

C. Implementation of SMs and SPs using BLOCKS and THREADS

The CUDA programming model maintains two (2) separate memory spaces in DRAM including host memory (utilized by the CPU) and device memory (utilized by the GPU). Host codes execute serially on CPU and the device functions that means kernels execute on GPU. Therefore, a program manages the global, constant, and texture memory spaces (host variables) visible to kernels through calls to the CUDA runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory.

```
#define gridsize 5
#define blocksize 1024
__global__ void adder(int *d_a,int *d_b,int *d_c)
{
        int a = blockDim.x * blockIdx.x + threadIdx.x;
        d_c[a] = d_a[a] + d_b[a];
}
int main()
{
       adder << <gridsize, blocksize >> >(d_a, d_b, d_c);
}
```

The function(adder) labeled with __global__ type is the function that is invoked by the serial code for executing in GPU. The instructions written inside the function are exactly the instructions executed in each thread. The blocksize stands for the number of threads allocated in each block, and the grid size stands for the number of blocks are allocated during the program execution. Thus, GPU will enumerate and create 5 blocks each accommodating 1024 threads in a logical partition of GPU for the above kernel function. Once the enumeration is done blocks would be thrown to SMs to execute. Now some restrictions are imposed on the device. Users are bound to declare the size of the block equal or lesser than the number of threads a single SM is capable to accommodate. In our GTX 650 device, the maximum 1024 number of threads an SM can manage with its 192 SPs.

All user-defined numbers of threads are first allocated into a block and the blocks are then superimposed on the SMs. In the GTX 650 device, there are only two SMs. Each SM would first allocate two 1024/512/256/128/64 sized blocks in them. The SM would then calculate how many threads are available within him to complete 1024 threads. Those rest are allocated to another block. Thus, one SM would be able to execute 1 (1024/1024), 2(1024/512), 4(1024/256), 8 (1024/128), and 16 (1024/64) block(s) consecutively. Thus, in GTX 650, for 5 grid size and 1024 blocksize will execute three (3) iterations and consecutively execute 2x1024 threads in one iteration. If one iteration takes 3.2 microseconds, then the full code execution takes 9.6 microseconds.

Each kernel execution takes the same amount of time due to thread and block-level parallelism rather depending on the number of data to be executed. Let's find out the effect of block size and grid size in execution time. The experiment in Fig. 3 proofs that a single block takes almost the same amount of time regardless of the number of data or threads are taken into consideration. The initial downtrend of the graph might play a little role of counterproof but on average the time taken to execute each of the blocks keeps a relatively static flow.

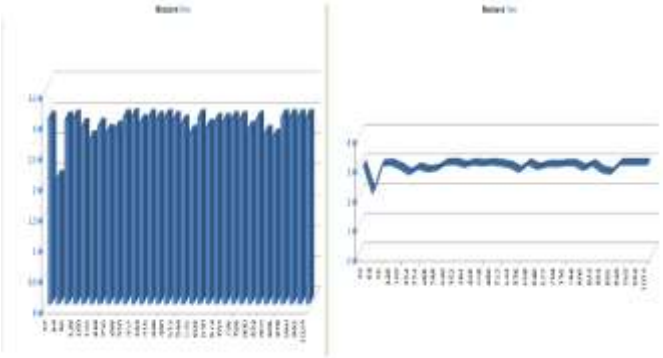

Fig 3. Time graph for Grid size: 1 & Blocksize: 32 to 1024

D. Related Works

Many real-time distributed systems are designed and implemented on the CORBA framework in [8] [9] [10] [11] and [12]. Heterogeneous computing systems provide an opportunity to dramatically increase the performance of parallel and High-Performance Computing (HPC) applications on clusters with CPU and GPU architectures. GPUs are used to speed up many scientific computations; however, to use several networked GPUs concurrently, the programmer must explicitly partition work and transmit data between devices. Message Passing Interface (MPI) works through the message passing paradigm between scalable clusters [13]. However, it is unable to transfer data between CPU and GPU. OpenMP [14] works on a shared memory multiprocessing system but unable to scale beyond 200 nodes due to threads management overhead and cache coherence H/W requirements. OpenCL [15] a standard programming model is jointly developed by Apple, Intel, AMD, and NVIDIA. It supports parallelism and efficient data delivery between parallel processors. However, it uses lower-level programming constructs. Common Unified Device Architecture (CUDA) [24] supports straightforward APIs to manage devices, memory, etc, and higher scalability with low-overhead thread management, easy communication between CPU and GPU, or vice versa. Thus, the CUDA programming API is suitable for hosts to support GPGPU processing.

DistCL[16] framework distributes the execution of OpenCL kernels across a GPU cluster. Many GPUs Package (MGP) [17] is running OpenMP, C++, and unmodified OpenCL applications on clusters with many GPU devices and reduce the complexity of programming and running parallel applications on the clusters-based system. DistCL and MGP work successfully in an integrated, centralized, homogeneous cluster computers system only. Recently major cloud providers, such as Microsoft Azure, Amazon Web Services, and IBM SoftLayer have announced partnerships with Nvidia to provide on-demand GPU cloud computing [18]. Still, they are in the developing phase. Thus, service-oriented and distributed GPGPU processing is in demand. Remote Method Invocation (RMI)[19], Common Object Request Broker Architecture (CORBA)[20][21], Simple Object Access Protocol (SOAP)[22], Remote Procedure Call (RPC)[23][24][25] and etc. are the available middleware to create distributed frameworks. Besides, CORBA supports language independence and mobility (platform independence) and its java-based implementation

can be a good choice for a distributed framework to provide GPGPU processing service.

Merge sort is one of the most efficient sorting algorithms and works on the principle of Divide and Conquer. Adaptive Merge sort is a modified merge sort [5] and reduces the layer of merging. However, our adaptive merge sort has not yet been implemented in parallel. Thus, in this paper the application we choose our proposed adaptive merge short with a new design of implementation for GPGPU based processing.

III. TOOLS AND APPLICATION

The GPU device (used for the experiments) is designed by Kepler architecture and uses a GK107 chip. The Device has a clock rate of 1124 MHz No. of multiprocessors (SM) is 2 with 192 cores in each. The warp size of the device is 32. The CPU specifications are Intel(R) Core (TM) i5-3470 CPU, 3.20GHz (4 CPUs), 3.2GHz, 8192 MB RAM, and Page File is 6004 MB.

A. Application 1: Matrix Multiplication

To multiply two matrices, the condition that must be followed is: 'Number of columns in the 1st matrix must be equal with the number of rows in the 2nd matrix'. The dimension of the result matrix will be (column of the 1st matrix × row of the 2nd matrix). That means if we take two matrices A[2][3] & B[3][4] and multiply then the result matrix will be C[2][4]. See Fig. 4 for a better understanding.

$$\begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} \\ A_{1,0} & A_{1,1} & A_{1,2} \end{bmatrix} \times \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \end{bmatrix} = \begin{bmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \end{bmatrix}$$

Fig 4. Matrix Multiplication

The equation to calculate the value of each position of C is:

C0,0= A0,0 × B0,0 + A0,1 × B1,0 + A0,2 × B2,0
or, C1,2= A1,0 × B0,2 + A1,1 × B1,2 + A1,2 × B2,2

Serial implementation

In serial application the program determines every value in result array one at a time. We can describe the code snippet as follows:

```
for i = 0 to < row of 1st matrix
        begin
    for j = 0 to < column of 2nd matrix
            begin
        for k = 0 to < row of 2nd matrix
                begin
               sum = sum + matrix1[i][k] *matrix2[k][j];
                end
      result¬_matrix[i][j] = sum;
           sum = 0;
           end
    end
```

This part calculates the value of each position of the result matrix. As we can see to calculate the result matrix it needs 24 times to calculate the value of the sum according to the example given above. It may seem very little number to a beginner level programmer and also took a tiny period to execute on the modern processor, but when we multiply matrices with hundreds/thousands of rows and columns then the time cost will be visible to us. We present some data of time to calculate to multiply two matrices in Table 1. To avoid complexity, we are using square matrices (matrix with the same height and width). We can reduce the computation time by implementing it in parallel GPU architecture.

Parallel implementation

Before understanding the parallel application let's recall the concept of two-dimensional thread and block at first. We already know that every thread has a given ID known as threadIDx starts from 0 at every new block. To know the position of a thread from the beginning of the first block we can use the following equation:

Position_of_thread = blockIDx × blockDim + threadIDx

Here, blockIDx is the block ID that contains several active threads, blockDim is the total number of threads in a block and threadIDx is ID of the active thread.

Therefore, the idea is, the value of every position of the result matrix will be calculated by different threads in parallel. That means, the value of C0,0 will be calculated on threadIDx(0,0) and C1,2 on thradIDx(1,2), whether they belong to the same or different block. Which is completely dependent upon the value of blockDim defined by the programmer at the beginning of the program. So, the Kernel function will be like:

```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
        // Each thread computes one element of C
       // by accumulating results into Cvalue
float Cvalue = 0.0;
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;

if(row > A.height || col > B.width) return;
for (int e = 0; e < A.width; ++e)
Cvalue += A[row][col+e] * B[row+e] [ col];
   C.elements[row][col] = Cvalue;
}
```

B. Application 2: Adaptive Merge Sort

Adaptive merge sort is an improved version of the famous merge sort algorithm based on the "divide and conquer" method. Where merge sort has a time complexity of O(nlog2(n)) for the best-case scenario, the adaptive merge sort has the time complexity of O(n). For the worst-case scenario, both the merge sort and the adaptive merge sort has the time complexity of O(nlog2(m)) where m<=n/2. Adaptive merge sort gives the programmers an efficient way of sorting. Also, it can perform even better if coupled with parallelism.

Data sets of random numbers have some natural orders or sequences. Even in the worst-case situation (the most disordered) at least two elements sitting alongside each other have an ordered sequence, either increasing or decreasing. For performing the adaptive merge sort, at first, we find that natural ordered sequence(s) and mark them using a flag based on ascending or descending order, which

means divide the data into smaller sub-list or node. After that, every consecutive pair of one merge and create a new node in sorted order. After creating smaller sub-lists or nodes, the sorting process can be implemented in serial or in parallel.

Serial Implementation

For every merge operation, two types of nodes are needed- high_node and low_node, and it occurs between two consecutive pairs of nodes. For example, node 0 with node 1 or node 4 with node 5. One-time merge loop operation between two consecutive nodes from 0 to N is termed as – total merge. The total merge occurs log2N times for serial implementation.

Parallel Implementation

Parallel implementation differs from serial implementation in one key aspect, two (2) procedures from serial implementation are executed in parallel:

• Conversion of sublists (descending nodes to ascending): For serial implementation, every node needs to be checked and converted from descending order to ascending order once per iteration. The worst-case complexity for this procedure is O(N/2). In parallel implementation, all nodes are checked by different threads and executed in parallel. The proposed parallel implementation can check 1024x2x2=4096 nodes in a single time instant. Thus, complexity will be reduced to O (1).

• Merging the converted sublists: This part works in parallel thus transform into a kernel function and executes in GPU. For parallel execution, a CUDA program was created to do the conversion procedure of the sublists. The following kernel will execute in GPU and will work with 5 threads.

__global__ void merge_myway_ascend(int *d_num, int *d_start_ind, int *d_end_ind, int *d_as_ds, int *swapper)

And invoking kernel function in main

merge_myway_ascend <<<1, nonodes >>>(d_num, d_start_ind, d_end_ind, d_as_ds, swapper);

An example data set with 17, 51, 64, 94, 17, 17, 18, 0, 1, 2, 4, 5, 14, 15, 18, 0, 5, 17, 18, 64 has taken to show the steps of parallel implementation. Conversion algorithm first implements and Fig. 5 presents the sub lists D= (Node 0-Node 4)=5.

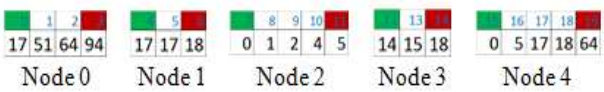

Fig 5. Nodes after conversion

Merging procedure calls the merge kernel, with the following parametes- Height =⌈ ⌊log⌋ _2 (D)⌉ = 3, Size of data (N) = 20 and Aproxnode size = ∑(size[i]*freq[i]) / ∑ freq[i] = 4. After invoking merge kernel following operations are carried out in each thread execution. For finding new position for any data we follow the equation stated below:

New position = start_ind[low] + (fetched index - start_ind[high]) + (index of itself in main dataset - start_ind[low])

Where,

start_ind[low] = starting index of the low node.

fetched index = the index in the high node where the data should probably be at.

start_ind[high] = starting index of the high node.

index of itself in the main dataset is equal the index of the data in consideration in the original dataset.

Below, calculations are shown the merging procedure for Node0 and Node1. Thus low=0 and high=1. This is called first level merging and presented in Fig. 6(i) to Fig. 6(vii).

Thread 0 is responsible to find the position of Data [0] in the result merging array. To do that thread0 reads the starting index of Node0 (start_ind[low] is 0, low=0) and the starting index of Node1 (start_ind[high] is 4, high=1). Node0[data index] = 17 will check in Node1 list from starting index (4) to end sequentially to find a bigger number than 17, and when finds return the Node1 index (fetched index). In our case which is 6. So, New position of 17 = start_ind[low] + (fetched index - start_ind[high]) + (index of itself in main dataset - start_ind[low]) = 0 + (6 - 4) + (0 - 0) = 2. Fig. 6(i) puts 17 into its right position.

Index	0	1	2	3	4	5	6
Data			17				

Fig 6(i). New position of 17 from low_node

Simultaneously thread 1 executes and responsible to find the position of Data [1] in the result merging array. To do that thread1 reads the starting index of Node0 (start_ind[low] is 0, low=0) and the starting index of Node1 (start_ind[high] is 4, high=1). Node0[data index] = 51 will check in Node1 list from starting index (4) to end sequentially to find a bigger number than 51, and when finds return the Node1 index (fetched index). In our case which is none. Thus, fetched index = end index of Node1+1; So, New position of 51 = start_ind[low] + (fetched index - start_ind[high]) + (index of itself in main dataset - start_ind[low]) = 0 + (7 - 4) + (1 - 0) = 4. Fig. 6(ii) puts 51 into its right position.

Index	0	1	2	3	4	5	6
Data			17		51		

Fig 6(ii): New position of 51 from low_node

Simultaneously thread 2 executes similar way and find new position for Data[2]. New position of 64 (Data[2]) = start_ind[low] + (fetched index - start_ind[high]) + (index of itself in main dataset - start_ind[low]) = 0 + (7 - 4) + (2 - 0) = 5. Fig. 6(iii) puts 64 into its right position.

Index	0	1	2	3	4	5	6
Data			17		51	64	

Fig 6(iii). New position of 64 from low_node

Simultaneously thread 3 executes similar way and find new position for Data[3]. New position of 94 (Data[3]) = start_ind[low] + (fetched index - start_ind[high]) + (index of itself in main dataset - start_ind[low]) = 0 + (7 - 4) + (3 - 0) = 6. Fig. 6(iv) puts 94 into its right position.

Index	0	1	2	3	4	5	6
Data			17		51	64	94

Fig 6(iv). New position of 94 from low_node

Similarly, threads 4, 5, and 6 execute and find new positions for Data[4], Data[5], and Data[6]. However, they do not need to compare for finding a bigger value from them. Directly their index is used to find their new position.

New position of 17 (Data[4]) = start_ind[low] + (fetched index - start_ind[high]) + (index of itself in main dataset - start_ind[low]) = 0 + (0 - 4) + (4 - 0) = 0. Fig. 6(v) puts 17 into its right position.

Index	0	1	2	3	4	5	6
Data	17		17		51	64	94

Fig 6(v). New position of 17 from high_node

New position of 17(Data[5]) = start_ind[low] + (fetched index - start_ind[high]) + (index of itself in main dataset - start_ind[low]) = 0 + (0 - 4) + (5 - 0) = 1. Fig. 6(vi) puts 17 into its right position.

Index	0	1	2	3	4	5	6
Data	17	17	17		51	64	94

Fig 6(vi). New position of another 17 from high_node

New position of 18 (Data[6]) = start_ind[low] + (fetched index - start_ind[high]) + (index of itself in main dataset - start_ind[low]) = 0 + (1 - 4) + (6 - 0) = 3. Fig. 6(vii) puts 18 into its right position.

Index	0	1	2	3	4	5	6
Data	17	17	17	18	51	64	94

Fig 6(vii). New position of 18 from high_node

Fig. 7 shows the result after completion of the first level merging.

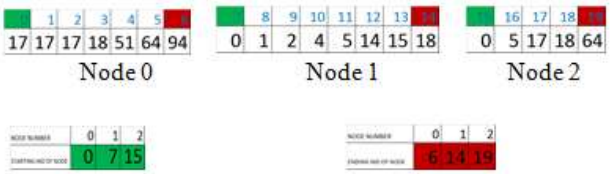

Fig 7. Data set with Start and end INDEX of nodes after the 1st level of merging

And the process described earlier for the first level merging will also continue until the final result is achieved. Fig. 8 shows the result after second-level merging and Fig. 10 shows the final sorted list after final merging. The total number of the level needed is the Height value, which is = 3 here.

Fig 8. Data set with Start and end INDEX of nodes after the 2nd level of merging

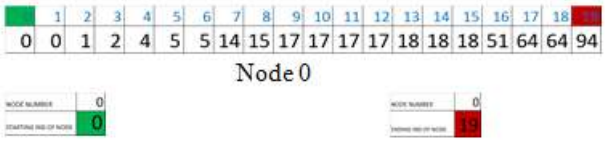

Fig 9. Main data set and Start and end INDEX of the node after final level merging

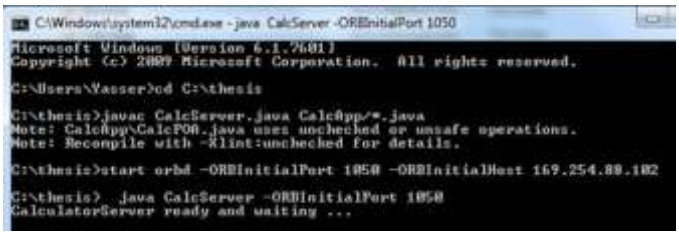

Fig 10. CalcServer is the server code initialized at port 1050.

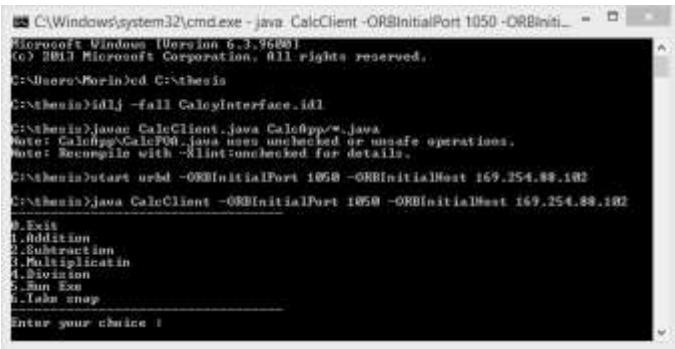

Fig 11. Client console.

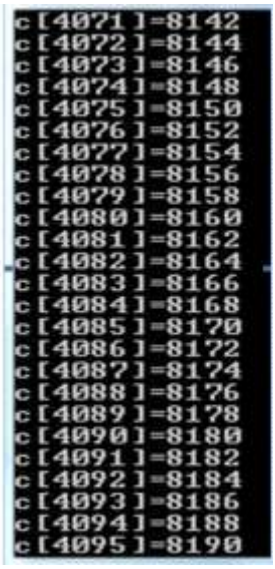

Fig 12. Output of a complete CUDA program of adding two layers

IV. EXPERIMENTS AND RESULTS

A. CORBA Based Client-Server Implementation

CORBD (Object request broker daemon) is used to enable clients to transparently locate and invoke persistent objects on servers in the CORBA environment. Our CORBA-based server (CalcServer) is implemented in IP 169.254.88.102 and port 1050, see Fig. 10.

The client uses port no. and IP address in the command line are used to establish a connection with the server. After successful connection server asks the client to choose one of

the given options as services given in Fig. 11. The client chooses the appropriate service as a number. Thereafter, the server-side code can perform the specific service including ADD, SUB, Run Exe (Adaptive Merge Sort), etc. For running Adaptive Merge Sort in CUDA based GPGPU processing Run Exe option needs to choose. Fig. 12 shows an example running environment of a complete CUDA program that adds two(2) arrays of 4096 integers and the operation uses GPU cores and takes a screenshot of the outputs and saves in on server computer.

B. Experiment on Application1: Matrix Multiplication

At first, we implement the Matrix Multiplication application in the server with serial implementation, and the CPU execution time (in a sec) is noted in Table I. Later, we experiment on the Matrix Multiplication application in GPGPU based server. We already mentioned that the kernel code is the part of CUDA code that is executed by GPU processors. Thus, to visualize the effect of GPU-based implementation, we experiment and test the Matrix Multiplication application's kernel execution time by increasing matrix size and block size (number of threads). Results are presented in Fig. 13. It highlights a noticeable decrement of time on the same size matrix calculation between parallel implementations with different block sizes. A question can arise here, why is there no difference in execution time between 32 to 4 block sizes for 200 size square matrix multiplications but have a huge difference in 1000 size square matrix multiplication?

The answer to the question depends on the GPU architecture. As our GPU is limited to use 2SMs and 1024 threads in each SM. Thus, it restricts us to use the highest 1024x2 threads and 64k local memory to store block data inside an SM. 32,16,8 and 4 block sizes restrict 64,128,256 and 512 corresponding blocks into 2SMs and 32, 16, 8, and 4 corresponding threads execution for solving a block of operations in parallel. 32, 16, 8, and 4 block sizes allow per-thread operation same thus should provide the same time to complete the execution. However, SM internal memory size creates constraints. All block data should be copied into internal SM memory at a single time. When block data crosses the size of the internal memory then left the exceed block to execute in the next phase, which repeats block copy and thread execution and increases overall execution time. Thus, 200 size square matrix multiplication requires 200x200=40,000 results. Thus, 40,000 threads can execute parallel and solve the problem nearly a similar amount of time for the block sizes. However, increasing the data size creates a larger data size for a block and demands a larger memory size. Next level copy time adds much more execution time and creates larger gaps between different block sizes executions.

TABLE I
AVERAGE EXECUTION TIME OF MATRIX MULTIPLICATION SERIAL IMPLEMENTATION

Size of the Square Matrices (S) (A[S][S] & B[S][S])	Time (sec)
200	0.037
400	0.408
600	1.555
800	5.469
1000	13.658

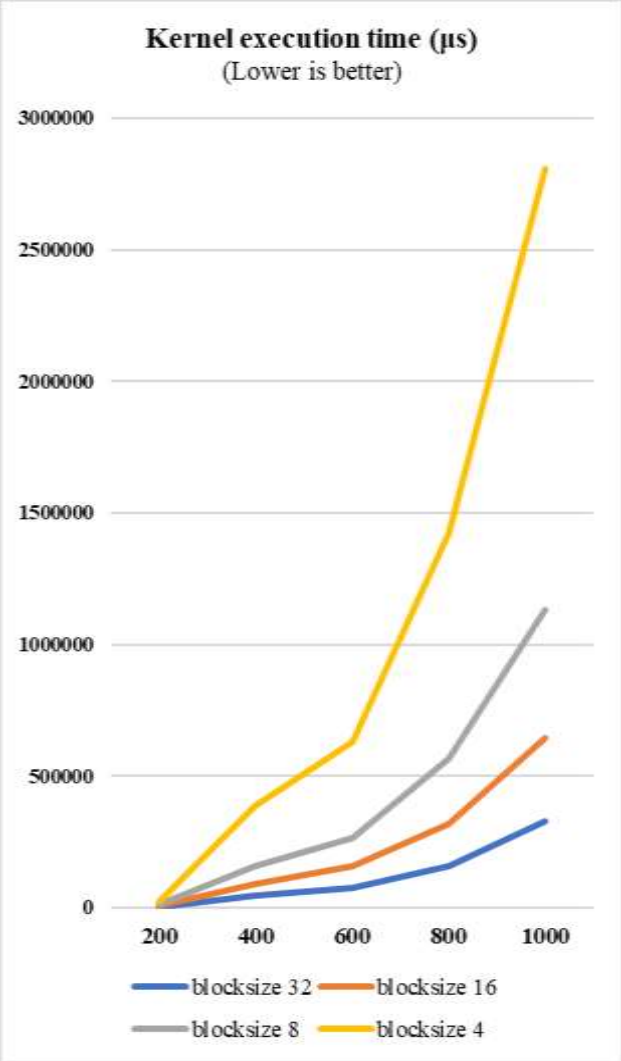

Fig 13. Kernel execution time (µs) vs. matrix size based on various Block Sizes

C. Experiment on Application2: Adaptive Merge Sort

Parallel Adaptive Merge sort requires both CPU and GPU hybrid implementation. Serial parts implement in the CPU and parallel parts implement in GPU in this experiment. Thus, we consider only merging portion to compare the performance of Adaptive Merge sort implementation in CPU and GPU as merging procedure implements in GPU only for parallel version and CPU for the serial version. Fig. 14 depicts a comparison in execution time of merging procedure for various amounts of data using CPU and GPU. We can see the difference in performance for completing an adaptive merge sort on a large number of data using CPU and GPU. Parallel implantation using GPU wins the comparison by a large gap with the serial implementation using CPU. For 13312 data, our proposed parallel implementation works about 6.89% faster than the serial implementation when the block size is 1024 and grid size 1. Fig. 15 depicts the comparison between the execution time of serial implementation by CPU and proposed parallel implementation by GPU for an optimum block size of 128. We see a significant rise in performance by our proposed parallel implementation using the CORBA framework for the block size of 128; about 7.65% faster than serial implementation using CPU only.

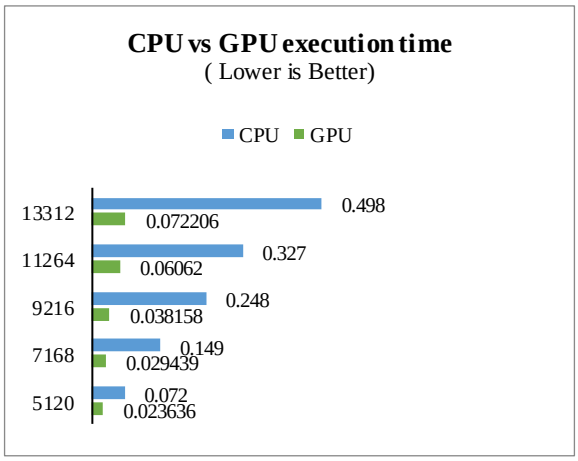

Fig 14. Comparison between the execution time of serial implementation using CPU and proposed parallel implementation using GPU (blocksize 1024)

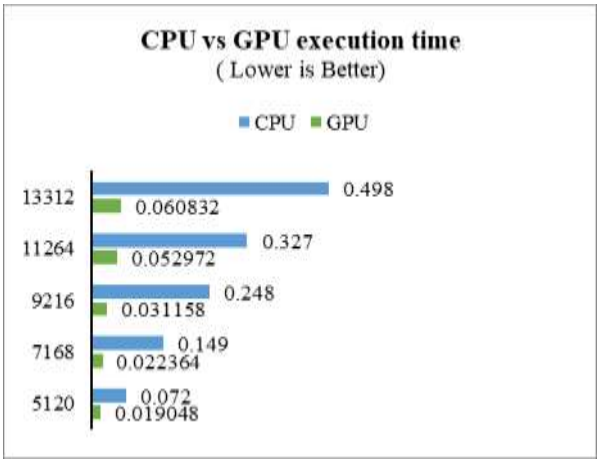

Fig 15. Comparison between the execution time of serial implementation using CPU and proposed parallel implementation using GPU (blocksize 128)

V. Conclusion

Matrix multiplication is a specific type of application where the appeal of CUDA shines. To avoid multifaceted iterations that might prove to be a huge deal of time consumption, it is a good idea to implement GPU parallelism rather than relying upon the CPU's serial implementations. We avoid memory transfer time as it does not serve the purpose of our pursuit and leave for future work. Similar results are also highlighted in adaptive merge sort. Besides, the CORBA framework based parallel implementation for adaptive merge sort yielded the best result in every possible scenario. With optimum blocksize, CORBA framework based parallel implementation is found out to be about 7.65% faster than the serial implementation. Thus two different applications are implemented on GPGPU through the proposed CORBA-based distributed framework. Their successful implementation and performance highlight the workability of our proposed framework.

REFRENCES

[1] X. Gonze, B. Amadon, P.-M. Anglade, J.-M. Beuken, F. Bottin, P. Boulanger, F. Bruneval, D. Caliste, R. Caracas, M. Côté, T. Deutsch, L. Genovese, P. Ghosez, M. Giantomassi, S. Goedecker, D. Hamann, P. Hermet, F. Jollet, G. Jomard, S. Leroux, M. Mancini, S. Mazevet, M. Oliveira, G. Onida, Y. Pouillon, T. Rangel, G.-or futureM. Rignanese, D. Sangalli, R. Shaltaf, M. Torrent, M. Verstraete, G. Zerah, and J. Zwanziger, "ABINIT: First-principles approach to material and nanosystem properties," Computer Physics Communications, vol. 180, no. 12, pp. 2582–2615, 2009. DOI: 10.1016/j.cpc.2009.07.007

[2] "AxRecon Image Reconstruction Solution," Scientific Computing World, 07-Jul-2016. [Online]. Available: https://www.scientific-computing.com/press-releases/axrecon-image-reconstruction-solution. [Accessed: 02-May-2020].

[3] "About ArrayFire," ArrayFire. [Online]. Available: http://arrayfire.org/docs/index.htm. [Accessed: 29-Apr-2020].

[4] "NVIDIA Launches the World's First Graphics Processing Unit: GeForce 256," NVIDIA. [Online]. Available:https://www.nvidia.com/object/IO_20020111_5424.html. [Accessed: 02-May-2020].

[5] S. Akhter and M. T. Hasan, "Sorting N-Elements Using Natural Order: A New Adaptive Sorting Approach," Journal of Computer Science, vol. 6, no. 2, pp. 163–167, Jan. 2010. DOI: 10.3844/jcssp.2010.163.167

[6] V. Estivill-Castro and D. Wood, "A survey of adaptive sorting algorithms," ACM Computing Surveys, vol. 24, no. 4, pp. 441–476, Jan. 1992. DOI: 10.1145/146370.146381

[7] "Web Services Glossary," Web Services Glossary. [Online]. Available: https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice. [Accessed: 02-May-2020].

[8] Y.K. Cho, B.P. Zeigler, H.S. Sarjoughian, Design and implementation of distributed real-time DEVS/CORBA, IEEE International Conference on System, Man, and Cybernetics, Tucson, Arizona October 7-10, 2001. DOI: 10.1109/ICSMC.2001.971989

[9] B. Thuraisingham, P. Kortmann, R. Johnson, G. Cooper, L. Dipippo, and V. Fay-Wolfe, "Real-time CORBA," IEEE Transactions on Parallel and Distributed Systems, vol. 11, no. 10, pp. 1073–1089, 2000. DOI: 10.1109/71.888646

[10] C. O'Ryan, D. C. Schmidt, and J. R. Noseworthy. Patterns and performance of a corba event service for large-scale distributed interactive simulations. In International Journal of Computer System s Science and engineering, 2001

[11] D. A. Karr, C. Rodrigues, Y. Krishnamurthy, I. Pyarali and D. C. Schmidt, "Application of the QuO quality-of-service framework to a distributed video application," Proceedings 3rd International Symposium on Distributed Objects and Applications, Rome, Italy, 2001, pp. 299-308, DOI: 10.1109/DOA.2001.954095.

[12] C. Oryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. L. Levine, "Evaluating policies and mechanisms to support distributed real-time applications with CORBA," Concurrency and Computation: Practice and Experience, vol. 13, no. 7, pp. 507–541, 2001. DOI: 10.1002/cpe.558.

[13] J. Bruck, D. Dolev, C.-T. Ho, M.-C. Roşu, and R. Strong, "Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations," Journal of Parallel and Distributed Computing, vol. 40, no. 1, pp. 19–34, 1997. DOI: 10.1006/jpdc.1996.1267.

[14] A. Basumallik, S.-J. Min, and R. Eigenmann, "Programming Distributed Memory Sytems Using OpenMP," 2007 IEEE International Parallel and Distributed Processing Symposium, 2007. DOI: 10.1109/IPDPS.2007.370397.

[15] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, "Introduction to OpenCL," Heterogeneous Computing with OpenCL, pp. 15–39, 2012.

[16] T. Diop, S. Gurfinkel, J. Anderson and N. E. Jerger, "DistCL: A Framework for the Distributed Execution of OpenCL Kernels," 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems, San Francisco, CA, 2013, pp. 556-566, doi: 10.1109/MASCOTS.2013.77.

[17] A. Barak, T. Ben-Nun, E. Levy and A. Shiloh, "A package for OpenCL based heterogeneous computing on clusters with many GPU devices," 2010 IEEE International Conference On Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), Heraklion, Crete, 2010, pp. 1-7, doi: 10.1109/CLUSTERWKSP.2010.5613086.

[18] A. Abramovici, "GPU Cloud Computing is now a Reality," stratoscale.com. [Online]. Available: http://www.stratoscale.com/blog/compute/gpu-cloud-computing-reality/. [Accessed: 02-May-2020].

[19] "Java Remote Method Invocation - Distributed Computing for Java," Java Remote Method Invocation - Distributed Computing for Java. [Online]. Available: https://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html. [Accessed: 29-Apr-2020].

[20] "About the CORBA Embedded Specification Version 1.0," About the CORBA Embedded Specification Version 1.0. [Online]. Available: https://www.omg.org/spec/CORBAe/1.0. [Accessed: 29-Apr-2020].

[21] K. P. Birman, "CORBA: The Common Object Request Broker Architecture," Guide to Reliable Distributed Systems Texts in Computer Science, pp. 249–269, 2012. DOI: 10.1007/978-1-4471-2416-0_7

[22] "SOAP (Simple Object Access Protocol)," Encyclopedia of Genetics, Genomics, Proteomics and Informatics, pp. 1837–1837, 2008. DOI: 10.1007/978-1-4020-6754-9_15821

[23] "Remote Procedure Call," The JR Programming Language The International Series in Engineering and Computer Science, pp. 91–105. DOI: 10.1007/1-4020-8086-7_8

[24] A. D. Birrell and B. J. Nelson, "Implementing Remote procedure calls," ACM SIGOPS Operating Systems Review, vol. 17, no. 5, p. 3, 1983. DOI: 10.1145/773379.806609

[25] S. Wilbur and B. Bacarisse, "Building distributed systems with remote procedure call," Software Engineering Journal, vol. 2, no. 5, p. 148, 1987. DOI: 10.1049/sej.1987.0020

[26] O. Green, R. Mccoll and D.bader, "GPU merge path: a GPU merging algorithm", 26th International Conference on Supercomputing, Servolo Island, Venice, Italy, 2012. DOI: 10.1145/2304576.2304621

