Chapter 4

GPU Computing Taxonomy

Abdelrahman Ahmed Mohamed Osman

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/intechopen.68179

Abstract

Over the past few years, a number of efforts have been made to obtain benefits from graphic processing unit (GPU) devices by using them in parallel computing. The main advantage of GPU computing is that it provides cheap parallel processing environments for those who need to solve single program multiple data (SPMD) problems. In this chapter, a GPU computing taxonomy is proposed for classifying GPU computing into four different classes depending on different strategies of combining CPUs and GPUs.

Keywords: host, device, GPU computing, single program multiple data (SPMD)

1. Objective

The objective of this chapter is as follows:

- Divide graphic processing unit (GPU) computing into four different classes.
- How to code different classes.
- Speedup computations using GPU computing.

2. Introduction

Despite the dramatic increase in computer processing power over the past few years [1], the appetite for more processing power is still rising. The main reason is that as more power becomes available, new types of work and applications that require more power are generated. The general trend is that new technology enables new applications and opens new horizons that demand further power and the introduction of some newer technologies.

Developments at the high end of computing have been motivated by complex systems such as simulation and modelling problems, speech recognition training, climate modelling and the human genome.

However, there are indications that commercial applications will also be in demand for high processing powers. This is mainly because of the increase in the volumes of data treated by these applications [2].

There are many approaches to increase computer processing power like improving the processing power of computer processors, using multiple processors or multiple computers to perform computations or using graphics processing unit (GPU) to speed up computations.

2.1. Why we need parallel computing

There are many reasons for parallelization, like speed up execution, overcome memory capacity limit and execute application that is distributed in its nature. The main reason for parallelization is to speed up the execution of applications. Another problem that arises in the era of big data is that the huge data, which need to be processed, do not fit in a single computer memory. Some applications are distributed in their nature, where parts of an application must be located in widely dispersed sites [3].

2.2. Important terminology

There are many important terminologies that help in understanding parallelization, here in this section we will talk about some of them.

2.2.1. Load balancing

The load balancing is an important issue for performance. It is a way of keeping all the processors busy as much as possible. This issue arises constantly in any discussion of parallel processing [3].

2.2.2. Latency, throughput and bandwidth

Latency, throughput and bandwidth are important factors that affect the performance of computations. Here is a brief definition for them.

- Communication latency is the time for one bit to travel from source to destination, e.g. from a CPU to GPU or from one host/device to another.
- Processor latency can be defined as the time to finish one task. While throughput can be defined as the rate at which we complete a large number of tasks (a number of tasks done in a given amount of time).
- Bandwidth is the number of bits per unit time that can be travelling in parallel. This can be affected by factors such as the bus width in a memory or the number of parallel network paths in a cluster and also by the speed of the links [3].

2.2.3. Floating point operation (FLOP)

It is a unit for measuring performance. It is about how many floating-point calculations can be performed in 1 s. The calculations can be adding, subtracting, multiplying or dividing two floating-point numbers. For example, 3.456 + 56.897 is equal to one flop.

Units of flops are as follows:

- Megaflop = million flops.
- Gigaflop = billion flops (million megaflops).
- Teraflop= trillions flops.
- Petaflop = quadrillion.
- Exaflop = quintillion.

3. Introduction to GPU

The revolutionary progress made by GPU-based computers helps in speeding up computations and in accelerating scientific, analytics and other compute intensive codes. Due to their massively parallel architecture with thousands of smaller, efficient cores, GPU enables the completion of computationally intensive tasks much faster than conventional CPUs, because CPUs have a relatively small number of cores [4, 5].

Due to these features, GPU devices are now used in many institutions, universities, government labs and small and medium businesses around the world to solve big problems using parallelization. The acceleration of application is done by offloads the parallel portions of the application to GPU's cores, while the remainder serial code runs on the CPU's core. GPU computing can be used to accelerate many applications, such as image and signal processing, data mining, human genome, data analysis and image and video rendering [6].

Currently, many of the fastest supercomputers in the top 500 are built of thousands of GPU devices. For example, Titan achieved 17.59 Pflop/s on the Linpack benchmark using 261,632 of its NVIDIA K20x accelerator cores.

The reasons for the spread of using GPU devices in high performance computing is that it has many features such as it is massively parallel, contains hundreds of cores, is able to run thousands of threads at the same time, is cheap and anyone can use it even in laptops and personal computers and is highly available and it is programmable [7].

4. GPU architecture

GPU is a device that contains hundreds to thousands of arithmetic processing units (ALUs) with a same size. This makes GPU capable of runing thousands of threads concurrently (able

to do millions of similar calculations at the same time in parallel), **Figure 1**. These threads need to be independent of each other without synchronization issues to run concurrently. Parallelism of threads in a GPU is suitable for executing the same copy of a single program on different data [single program multiple data (SPMD)], i.e. data parallelism [8].

> SPMD is different from single instruction, multiple data (SIMD). In SPMD, the same code of the program executed in parallel on different parts of data, while in SIMD, the same instruction is executed at the same time in all processing units [9].

CPUs are low latency, low throughput processors (faster for serial processing), while GPUs are high latency, high throughput processors (optimized for maximum throughput and for scalable parallel processing).

GPU architecture consists of two main components, global memory and streaming multiprocessors (SMs). The global memory is the main memory for the GPU and it is accessible by both GPU and CPU with high bandwidth. While SMs contain many simple cores that execute the parallel computations, the number of SMs in a device and the number of cores in SMs differ from one device to another. For example, Fermi has 16 SMs with 32 cores on each one (with the total cores equal to 16 × 32 = 512 cores), see **Table 1** for different GPU devices.

There are different GPU memory hierarchies for different devices. **Figure 2** shows an example of NVIDIA Fermi memory hierarchy with following memories:

- Registers.
- Shared memory and L1 cache (primary cache).
- Constant memory.
- Texture memory and read-only cache.
- L2 cache (secondary cache).
- Global (main) memory and local memory.

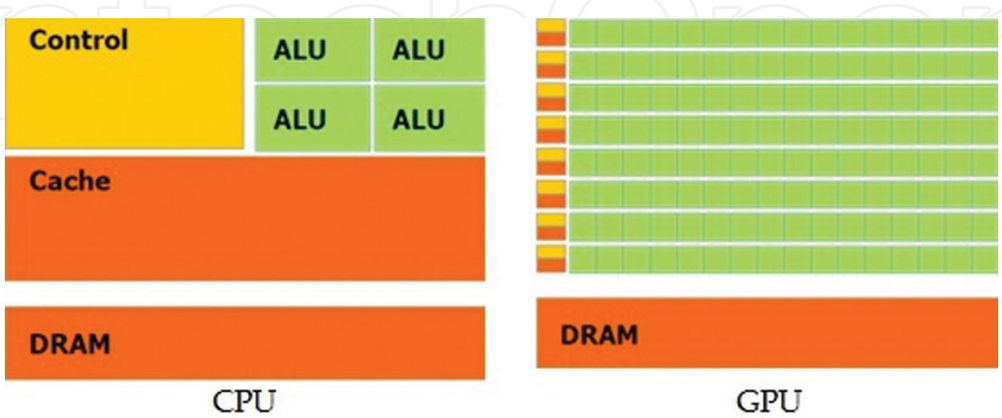

Figure 1. CPU vs. GPU, from Ref. [8].

	GTX 480	GTX 580	GTX 680
Architecture	GF100	GF110	GK104
SM/SMX	15	16	8
CUDA cores	480	512	1536
Core frequency	700 MHz	772 MHz	1006 MHz
Compute power	1345 GFLOPS	1581 GFLOPS	3090 GFLOPS
Memory BW	177.4 GB/s	192.2 GB/s	192.2 GB/s
Transistors	3.2B	3.0B	3.5B
Technology	40 nm	40 nm	28 nm
Power	250 W	244 W	195 W

Table 1. Comparisons between NVIDIA GPU architecture, from Ref. [10].

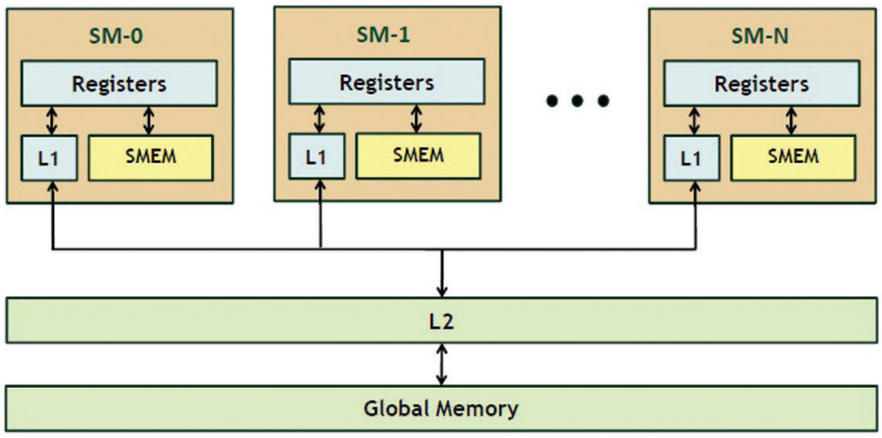

Figure 2. NVIDIA Fermi memory hierarchy, from Ref. [11].

5. GPU taxonomy

GPU computing can be divided into four different classes according to the different combinations between hosts and devices, **Figure 3**. These classes are as follows:

(1) Single host with single device (SHSD).

(2) Single host with multiple devices (SHMD).

(3) Multiple hosts with single device in each (MHSD).

(4) Multiple hosts with multiple devices (MHMD).

In the rest of this section, we will talk about each class separately.

	Single Device	Multiple Device
Single Host	SHSD	SHMD
Multiple Host	MHSD	MHMD

Figure 3. GPU computing taxonomy.

5.1. Single host, single device (SHSD)

The first class (type) in GPU taxonomy is the single host with single device (SHSD), as shown in **Figure 4**. It is composed of one host (computer) with one GPU device installed in it. Normally, the host will run the codes that are similar to conventional programming that we know (may be serial), while the parallel part of the code will be executed in the device's cores concurrently (massive parallelism).

The processing flow of SHSD computing includes:

- Transfer input data from CPU's memory to GPU's memory.
- Load GPU program and execute it.
- Transfer results from GPU's memory to CPU's memory [12].

The example of SHSD is shown in **Figure 5**; data transferred from CPU host to GPU device are coming through a communication bus connecting GPU to CPU. This communication bus is of type PCI-express with data transfer rate equal to 8 GB/s, which is the weakest link in the connection (new fast generation is available, see Section 6.1 for more details). The other links in the figure are the memory bandwidth between main memory DDR3 and CPU (42 GB/s), and the memory bandwidth between GPU and its global memory GDDR5 (288 GB/s).

Bandwidth limited

> Because transfer data between the CPU and GPU is expensive, we will always try to minimize the data transfer between the CPU and GPU. Therefore, if processors request data at too high a rate, the memory system cannot keep up. No amount of latency hiding helps this. Overcoming bandwidth limits are a common challenge for GPU-compute application developers [14].

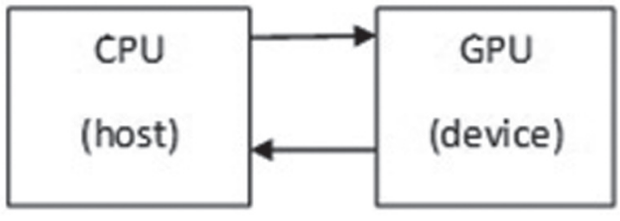

Figure 4. Single host, single device (SHSD).

http://dx.doi.org/10.5772/intechopen.68179

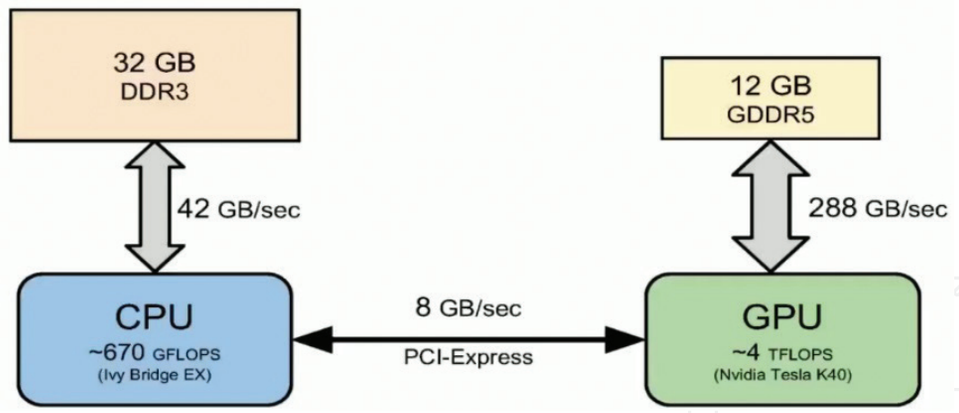

Figure 5. Example of SHSD class, from Ref. [13].

5.2. Single host, multiple device (SHMD)

In this section, we can use single host with multiple devices installed in it (SHMD), **Figures 6** and **7**. SHMD can be used to run parallel tasks in installed GPUs, with each GPU run sub-tasks in in their cores.

We can use a notation like SHMD (d, to show the number of devices that installed in the host. For example, if we have a single host with three GPU devices installed in it, we can write this as SHMD (3).

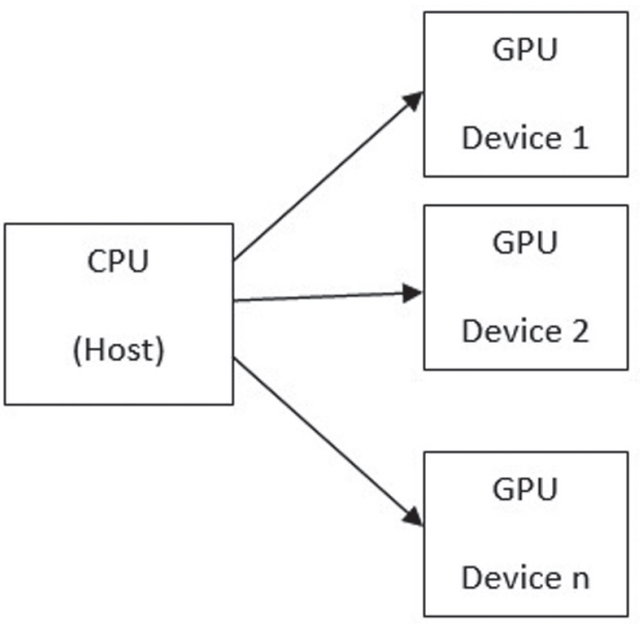

Figure 6. Single host, multiple device [SHMD (n)].

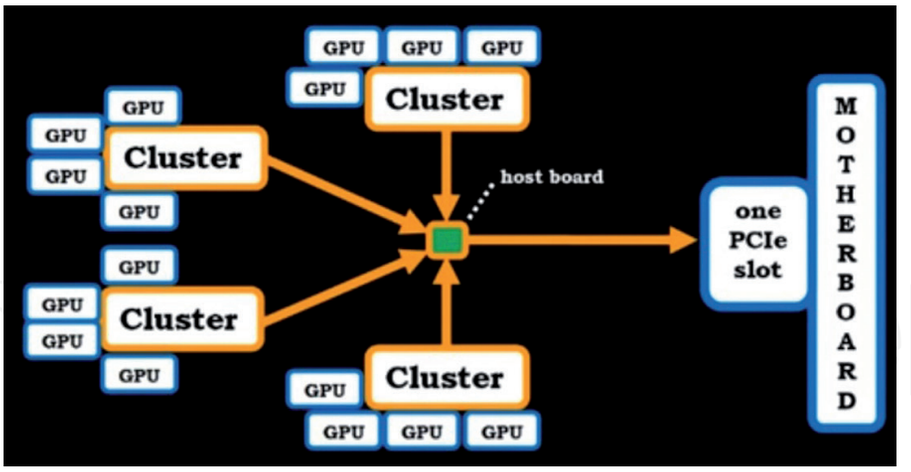

Figure 7. Server of type SHMD (16). Image from: https://www.youtube.com/watch?v=Vm9mFtSq2sg.

5.3. Multiple host, single device (MHSD)

Multiple host with single device in each host (MHSD) is an architecture for using a GPU cluster connecting many SHSD nodes together, **Figure 8**.

We can use the notation MHSD (h) to define the number of hosts in this architecture. For example, the architecture in **Figure 7** is an MHSD (4), where four nodes (SHSD) are connected in a network.

5.4. Multiple host, multiple device (MHMD)

Multiple host with multiple devices in each host (MHMD) is a GPU cluster with a number of SHMD nodes (where all nodes may have the same number of devices or may have a different number of devices), **Figure 9**.

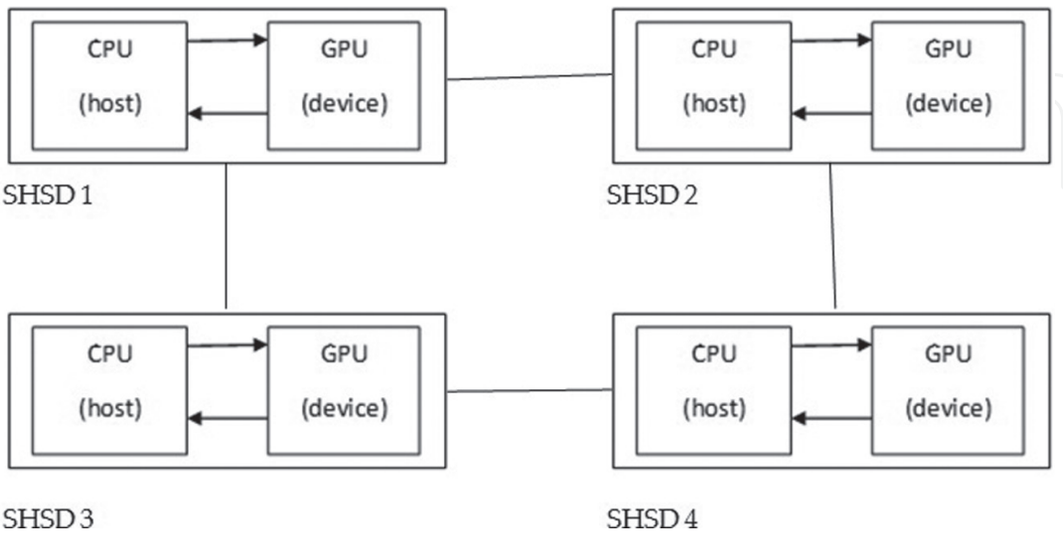

Figure 8. MHSD (4).

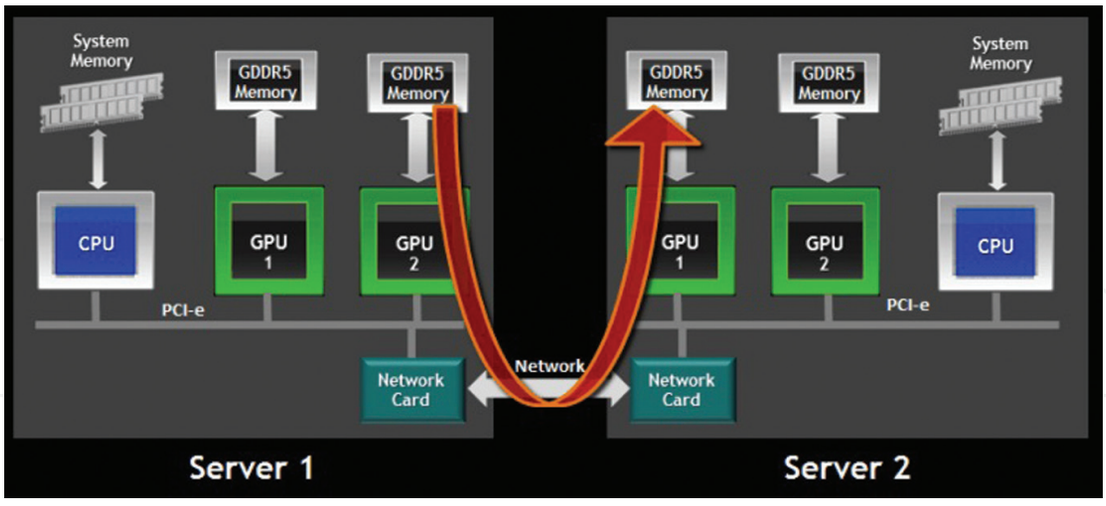

Figure 9. GPU cluster [MHMD (2,2)], image from: http://timdettmers.com/2014/09/21/how-to-build-and-use-a-multi-gpu-system-for-deep-learning.

We can use MHMD (h, d) notation to denote the number of hosts and the number of devices in each host, where h is for the number of hosts and d is for the number of devices. If the number of devices in each node is not equal, we can ignore the second parameter by putting x as do not care, MHMD (h, x).

6. Communications

There are two types of connections that can be used in GPU computing:

(1) The connection between the GPU devices and host (in SHSD and SHMD).

(2) The connection between different hosts (in MHSD and MDMD).

In this section, we will discuss about each one below.

6.1. Peripheral component interconnect express (PCIe)

PCIe is a standard point-to-point connection used to connect internal devices in a computer (used to connect two or more PCIe devices). For example, you can connect GPU to CPU or other GPU, or connect network card like InfiniBand with CPU or GPU, because most of these devices now are PCIe devices.

PCIe started in 1992 with 133 MB/s and increased to reach 533 MB/s in 1995. Then in 1995, PCIX appeared with a transfer rate of 1066 MB/s (1 GB/s). In 2004, PCIe generation 1.x came with 2.5 GB/s and then generation 2.x in 2007 with 5 GB/s, after that generation 3.x appeared in 2011 with a transfer rate equal to 8 GB/s and the latest generation 4.x in 2016 came with a transfer rate reaching 16 GB/s [15–17].

PCIe is doubling the data rate in each new generation.

6.2. Communication between nodes

In GPU cluster (MHSD or MHMD), the main bottleneck is the communications between nodes (network bandwidth) that is how much data can be transferred from computer to computer per second.

If we use none direct data transfer between different nodes in GPU cluster, then the data transferred in the following steps:

- GPU in node 1 to CPU in node 1.
- CPU in node 1 to network card in node 1.
- Network card in node 1 to network card in node 2.
- Network in node 2 to CPU in node 2.
- CPU in node 2 to GPU in node 2.

Some companies such as Mellanox and NVIDIA recently have solved the problem by using GPUDirect RDMA, which can transfer data directly from GPU to GPU between the computers [18].

6.3. GPUDirect

GPUDirect allows multiple GPU devices to transfer data with no CPU intervention (eliminate internal copying and overhead by the host CPU). This can accelerate communication with network and make data transfer from GPU to communication network efficient. Allow peer-to-peer transfers between GPUs [19]. CUDA supports multiple GPUs communication, where data can be transferred between GPU devices without being buffered in CPU's memory, which can significantly speed up transfers and simplify programming [20]. **Figures 10** and **11** show how GPUDirect can be used in SHMD and MHMD, respectively.

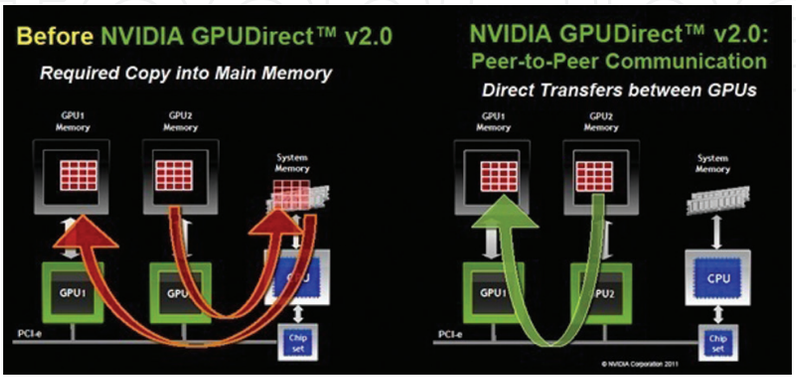

Figure 10. GPUDirect transfer in SHMD, from Ref. [19].

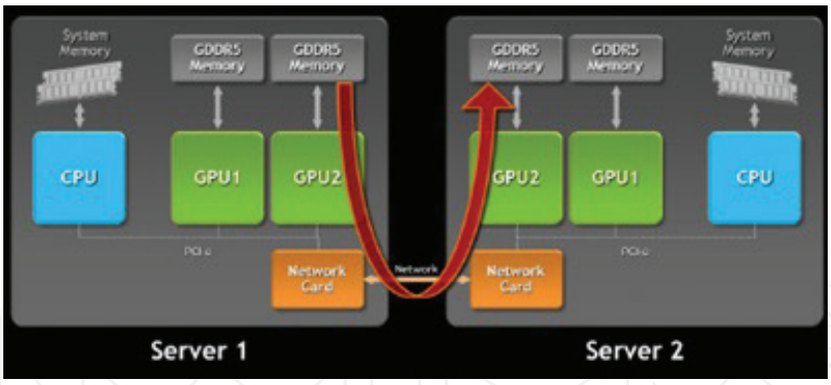

Figure 11. GPUDirect transfer in MHMD. Image from: http://www.paulcaheny.com/wp-content/uploads/2012/05/RDMA-GPU-Direct.jpg.

7. Tools for GPU computing

Many tools are available for developing GPU computing applications, including development environments, programming languages and libraries. In this section, we will give a little overview of some of these tools.

7.1. Compute unified device architecture (CUDA)

CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables increases the computing performance by harnessing the power of the GPU devices. CUDA splits a problem into serial sections and parallel sections, serial sections are executed on the CPU as a host code and parallel sections are executed on the GPU by launching a kernel, **Figure 12**.

7.1.1. CUDA in SHSD

CUDA can run in SHSD using the following sequence of operations [21]:

- Declare and allocate host and device memory.
- Initialize host data.
- CPU transfer input data to GPU.
- Launch kernel on GPU to process the data in parallel.
- Copies results back from the GPU to the CPU.

Kernel code: the instructions actually executed on the GPU.

The following codes demonstrate a portion of code that can be run in SHSD.

```
//Kernel definition --- device code
__global__ void VecAdd(float* A, float* B, float* C)
{
int i = threadIdx.x;
C[i] = A[i] + B[i];
}
//////host code invoking kernel
int main() {
…
//Kernel invocation with N threads (invoke device code from host code)
VecAdd<<<1, N>>>(A, B, C);
…
}
```

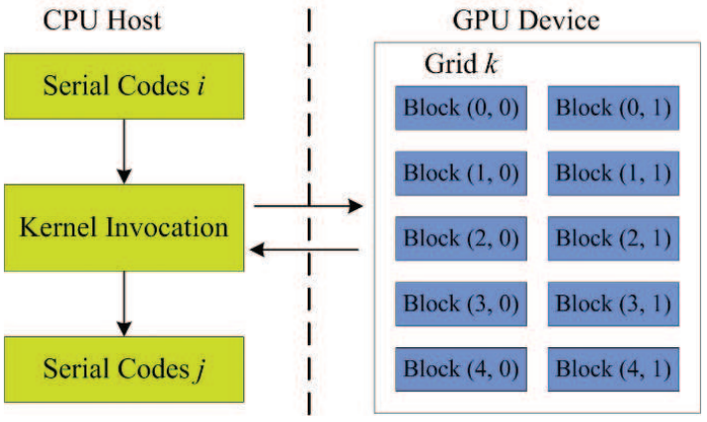

Figure 12. Using CUDA in SHSD. Image from: http://3dgep.com/wp-content/uploads/2011/11/Cuda-Execution-Model.png.

SHSD example from

7.1.2. CUDA in SHMD

The need for multiple GPUs is to gain more speed and overcome the limit of GPU device's memory (has smaller capacity compared to CPU's memory). For example, 32–64 GB is a typical size for the host memory, whereas a single GPU device has between 4 and 12 GB device memory. When dealing with large-scale scientific applications, the size of the device memory may thus become a limiting factor. One way of overcoming this barrier is to make use of multiple GPUs [22]. Now, systems with multiple GPUs are becoming more common. For SHMD, CUDA can manage multiple GPU devices from a single CPU host thread [23].

To use multiple GPU devices in single host, we need a thread for each device to control it (attach a GPU device to a host thread). The following codes show how one can invoke different kernels in different devices from a single host.

```
//Run independent kernel on each CUDA device
int numDevs = 0;
cudaGetNumDevices(&numDevs);//number of devices available
…
for (int d = 0; d < numDevs; d++) {
cudaSetDevice(d);//Attach a GPU device to a host thread (select a GPU)
kernel<<<blocks, threads>>>(args);//invoke independent kernel in each device
}
```

SHMD example using CUDA from Ref. [24]

7.1.3. Multiple host, single device (MHSD) and multiple host, multiple device (MHMD)

MPI is a programming model that used for a distributed memory system. If we have a MHSD or a MHMD system, MPI can be used to distribute tasks to computers, each of which can use their CPU and GPU devices to process the distributed task.

For example, if we want to do matrix multiplication on MHSD, then we can:

- Split the matrix into sub-matrices.
- Use MPI to distribute the sub-matrices to hosts (processes).
- Each host (process) can call a CUDA kernel to handle the multiplication on its GPU device(s).
- The result of multiplication would be copied back to each computer memory.
- Use MPI to gather results from all hosts (processes), and re-form the final matrix [25].

One way for programming MHSD and MHMD is to use MPI with CUDA, where MPI can be used to handles parallelization over hosts (nodes), and CUDA can be used to handle parallelization on devices (GPUs). We can use one MPI process per GPU and accelerate the computational kernels with CUDA.

For transferring data between different devices in different hosts, we can use the following steps:

- Sender: Copy data from a device to a temporary host buffer.
- Sender: Send host buffer data.
- Receiver: Receive data to host buffer.
- Receiver: Copy data to a device [26].

For example, if we need to transfer data between node 0 and node N-1 as shown in **Figure 13**, then we can use MPI_Send and MPI_Recv function as follows:

```
//MPI rank 0
MPI_Send(s_buf_d,size,MPI_CHAR,n-1,tag,MPI_COMM_WORLD);
//MPI rank n-1
MPI_Recv(r_buf_d,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
```

MHSD using MPI and CUDA

Where s_buf_d is from device 0 and r_buf_d is from device N-1.

The following code is a simple MPI with CUDA example that shows how to collect and print the list of devices from all MPI processes in the MHSD/MHMD system.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <mpi.h>
#include <cuda.h>
#define MAX_NODES 100
#define BUFF_LEN 256
//Enumeration of CUDA devices accessible for the process.
void enumCudaDevices(char *buff)
{
char tmpBuff[BUFF_LEN];
int i, devCount;
cudaGetDeviceCount(&devCount);//number of devices
sprintf(tmpBuff," %3d", devCount);
strncat(buff, tmpBuff, BUFF_LEN);
for (i = 0; i < devCount; i++)
{
cudaDeviceProp devProp;
cudaGetDeviceProperties(&devProp, i);
sprintf(tmpBuff, " %d:%s", i, devProp.name);
strncat(buff, tmpBuff, BUFF_LEN);
}
}
int main(int argc, char *argv[])
{
int i, myrank, numprocs;
char pName[MPI_MAX_PROCESSOR_NAME],
buff[BUFF_LEN];
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Get_processor_name(pName, &i);
sprintf(buff, "%-15s %3d", pName, myrank);
//Find local CUDA devices
enumCudaDevices(buff);
//Collect and print the list of CUDA devices from all MPI processes
if (myrank == 0)
{
char devList[MAX_NODES][BUFF_LEN];
```

```
MPI_Gather(buff, BUFF_LEN, MPI_CHAR,
devList, BUFF_LEN, MPI_CHAR,
0, MPI_COMM_WORLD);
for (i = 0; i < numprocs; i++)
printf("%s\n", devList + i);
}
else
MPI_Gather(buff, BUFF_LEN, MPI_CHAR,
NULL, 0, MPI_CHAR,
0, MPI_COMM_WORLD);
MPI_Finalize();
return 0;
}
```

The output of the program look similar to this:
g01n07.pdc.kth.se 0 3 0:Tesla M2090 1:Tesla M2090 2:Tesla M2090
g01n06.pdc.kth.se 1 3 0:Tesla M2090 1:Tesla M2090 2:Tesla M2090
MHMD simple example from
(https://www.pdc.kth.se/resources/software/old-installed-soft-ware/mpi-libraries/cuda-and-mpi)

Most recent versions of most MPI libraries support sending/receiving directly from CUDA device memory; for example Cray's implementation of MPICH supports passing GPU memory buffers directly to MPI function calls, without manually copying GPU data to the host before passing data through MPI. The following codes show how initialize memory on the GPU and then perform an MPI_Allgather operation between GPUs using device buffer [28].

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <mpi.h>
void main(int argc, char** argv)
{
MPI_Init (&argc, &argv);
int direct;
int rank, size;
int *h_buff = NULL;
int *d_rank = NULL;
int *d_buff = NULL;
size_t bytes;
int i;
//Ensure that RDMA ENABLED CUDA is set correctly
direct = getenv("MPICH_RDMA_ENABLED_CUDA")==NULL?0:atoi(getenv ("MPICH_RDMA_ENABLED_CUDA"));
if(direct != 1){
printf ("MPICH_RDMA_ENABLED_CUDA not enabled!\n");
exit (EXIT_FAILURE);
```

```
}
//Get MPI rank and size
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);
//Allocate host and device buffers and copy rank value to GPU
bytes = size*sizeof(int);
h_buff = (int*)malloc(bytes);
cudaMalloc(&d_buff, bytes);
cudaMalloc(&d_rank, sizeof(int));
cudaMemcpy(d_rank, &rank, sizeof(int), cudaMemcpyHostToDevice);
//Preform Allgather using device buffer
MPI_Allgather(d_rank, 1, MPI_INT, d_buff, 1, MPI_INT, MPI_COMM_WORLD);
//Check that the GPU buffer is correct
cudaMemcpy(h_buff, d_buff, bytes, cudaMemcpyDeviceToHost);
for(i=0; i<size; i++){
if(h_buff[i] != i) {
printf ("Alltoall Failed!\n");
exit (EXIT_FAILURE);
}
}
if(rank==0)
printf("Success!\n");
//Clean up
free(h_buff);
cudaFree(d_buff);
cudaFree(d_rank);
MPI_Finalize();
}
```

Direct transfer data, code from Ref. [28]

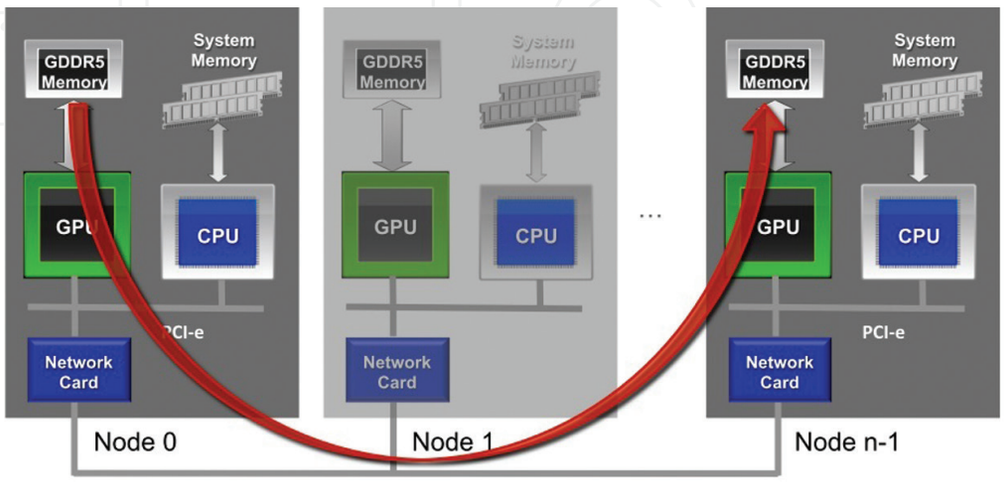

Figure 13. MPI with CUDA for MHSD, from Ref. [27].

7.2. GPU computing using MATLAB

MATLAB is a widely used simulation tool for rapid prototyping and algorithm development. Since MATLAB uses a vector/matrix representation of data, which is suitable for parallel processing, it can benefit a lot from CPU and GPU cores.

We can use two tools for parallelization in MATLAB:

- Parallel computing toolbox: to run applications on SHSD and SHMD.
- Distributed computing server with parallel computing toolbox: for applications that will run in MHSD and MHMD.

7.2.1. Parallel computing toolbox

Parallel computing toolbox can be used to speed up MATLAB code by executing it on a GPU. There are more than 100 built-in functions in MATLAB that can be executed directly on the GPU by providing an input argument of the type GPUArray, a special array type provided by parallel computing toolbox. MATLAB GPU-enabled functions such as fft, filter and several linear algebra operations that can be used in GPU computing. In addition, there are other GPU-enabled functions in many toolboxes like image processing toolbox, communication system toolbox, statistics and machine learning toolbox, neural network toolbox, phased array systems toolbox and signal processing toolbox. So the CUDA kernel can be integrated in MATLAB applications by only a single line of MATLAB code [29].

Using MATLAB for GPU computing is good for those who have some or a lot of experience on MATLAB coding, but not enough depth in either C coding or the computer architecture for parallelization [30].

For example, FFT can be used to find the discrete Fourier transform of a vector of pseudorandom numbers on the CPU with normal arguments like this:

```
A = rand(2^16,1);
B = fft(A);
```

The same operation can be executed on the GPU by just using data type of gpuArray like this:

```
A = gpuArray(rand(2^16,1));
B = fft(A);
```

The result, B, is stored on the GPU. However, it is still visible in the MATLAB workspace. You can return the data back to the local MATLAB workspace by using gather command, for example, C = gather(B) [29].

7.2.2. MATLAB-distributed computing server

MATLAB-distributed computing server is suitable for MHSD and MHMD. The server provides access to multiple workers that receive and execute MATLAB code and Simulink models. Multiple users can run their applications on the server simultaneously.

MHSD and MHMD can use MATLAB workers in parallel computing toolbox and MATLAB-distributed computing server.

> MATLAB support CUDA-enabled NVIDIA GPUs with compute capability 2.0 or higher. For releases 14a and earlier, compute capability 1.3 is sufficient. In a future release, support for GPU devices of compute capability 2.x will be removed. At that time, a minimum compute capability of 3.0 will be required.

7.2.3. SHSD code examples

The following codes show how to perform matrix multiplication in SHSD.

```
Z = X*Y; % computation on CPU

x = gpuArray(X); % create copy from X on GPU

y = gpuArray(Y); % create copy from Y on GPU

z = x*y; % computation on GPU

ZZ = gather(z); % return data from GPU to CPU
```

Example 1: SHSD matlab code

The following codes using gpuArray to push data to GPU and then any function call on this array will be executed on GPU. To return result back from GPU device memory to host memory, we use gather function.

```
A = someArray(1000, 1000);

G = gpuArray(A); % Push to GPU memory

…

F = fft(G);

x = G\b;

…

z = gather(x); % Bring back into MATLAB
```

Example 2: SHSD code example, from Ref. [31]

7.2.4. SHMD code examples

In MTALAB, the parallel computing toolbox (PCT) can be used easily to perform computations on SHMD systems. PCT support CPU parallelism by using MATLAB pool. It allows you to use a number of workers run concurrently in the same time. The default number of workers is equal to the number of cores (for local pool). When you run a PARFOR loop, for example, then the work for that loop is broken up and executed by the MATLAB workers.

To perform computations in the SHMD system, you need to open a MATLAB pool with one worker for each GPU device. One MATLAB worker is needed to communicate with each GPU. Each MATLAB session can use one GPU at a time.

If you have only one GPU in your computer that GPU is the default. If you have more than one GPU device in your computer, you can use the gpuDevice function to select which device you want to use.

If you have 2 GPUs, you can assign one local worker for each device, as shown below:

```
matlabpool local 2 % two workers
spmd
gpuDevice(labindex); % select device for each work
g = gpuArray(…);
… operate on g…
End
```

SHMD (2)

7.2.5. Multiple host, single device (MHSD)/multiple host, multiple device (MHMD)

MATLAB® Distributed Computing Server™ lets you run computationally intensive MATLAB programs and Simulink® models on computer clusters, clouds and grids. You develop your program or model on a multicore desktop computer using parallel computing toolbox and then scale up to many computers by running it on MATLAB-distributed computing server. The server supports batch jobs, parallel computations and distributed large data. The server includes a built-in cluster job scheduler and provides support for commonly used third-party schedulers [32].

A parallel pool is a set of workers in a compute cluster (remote) or desktop (local). The default pool size and cluster are specified by your parallel preferences. The workers in a parallel pool can be used interactively and can communicate with each other during the lifetime of the job [33].

In the following multiple GPU example, we can have more than one workers: if the workers are local in the same host, then we can name this type as SHMD; if the workers are remote on cluster, then this means we are dealing with multiple hosts (MHs) and if there are more than one GPU device in each host (local workers in each remote host, with one worker for each remote GPU device, this can be multiple devices (MDs) and hence the system is MHMD; otherwise, it is MHSD.

```
N = 1000;
A = GPUaRRAY(a);
for ix = 1:N
x = myGPUFunction(ix,A)
xtotal(ix,:) = gather(x);
end
SHSD
```

```
N = 1000;
spmd
gpuDevice(labindex)%worker for each device
A = GPUaRRAY(A);
end
parfor ix = 1:N
x = myGPUFunction(ix,A)
xtotal(ix,:) = gather(x);
end
Multiple GPUs, from Ref. [32]
```

7.3. Open accelerator (OpenACC)

OpenACC is an application-programming interface, stands for open accelerators, it came to simplify parallel programming by providing a set of compiler directives that allow developers to run parallel code with the modifying underlying code (like OpenMP), and it was developed by CAPS, Cray, NVidia and PGI. OpenACC uses compiler directives that allow small segments of code, called kernels, to be run on the device. OpenACC divides tasks among gangs (blocks), gangs have workers (warps) and workers have vectors (threads) [9, 34–36].

OpenACC is portable across operating systems and various types of host CPUs and devices (accelerators). In OpenACC, some computations are executed in the CPU, while the compute intensive regions are offloaded to GPU devices to be executed in parallel. The host is responsible for

- allocation of memory in the device,
- initiating data transfer,
- sending the code to the device,
- waiting for completion,
- transferring the results back to the host,
- deallocating memory and
- queuing sequences of operations executed by the device [37].

A small code example for using OpenACC is as follow:

```
main()
{
<serial>
#pragma acc kernels
//automatically runs on GPU device
{
<parallel code>
}
}
```

#pragma acc kernels: tells the compiler to generate parallel accelerator kernels that run in parallel inside the GPU device [38].

8. Conclusion

In this chapter, taxonomy that divides GPU computing into four different classes was proposed. Class one (SHSD) and two (SHMD) are suitable for home GPU computing and can

used to solve problems of single program and multiple data (SPMD) like image processing, where each task processes a part of the image, doing the same work in different data. The time spent in transferring data between the host and the device is affecting the overall performance, so try to minimize data transfer between the host and the device as possible as you can. Most successful GPU applications are doing a lot of computation in a small amount of data. This means, GPU will not work effectively for the small amount of threads, but it is efficiently launching many threads to use GPU effectively. It gives good performance when running a large number of threads in parallel.

Class three (MHSD) and four (MHMD) are used for GPU clustering, where data is transferred between devices on the same host or between devices in different hosts through a network connection. One of the reasons that affect performance is the speed of the used network.

Different types of GPU devices with different architectures and compute capabilities give different performance result, so choose the suitable one for your work before you start.

Author details

Abdelrahman Ahmed Mohamed Osman

Address all correspondence to: aamosman@uqu.edu.sa

Faculty of Computer at Al-Gunfudah, Umm AL-Qura University, Al-Gunfudah, Saudi Arabia

References

[1] Tanenbaum AS, Van Steen M. Distributed Systems. Prentice-Hall; © 2007 Pearson Education. Inc. Pearson Prentice Hall Pearson Education, Inc. Upper Saddle River, NJ 07458.

[2] Osman AAM. A multi-level WEB based parallel processing system a hierarchical volunteer computing approach. World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering. 2008;**2**(1):176-181.

[3] Matloff N. Programming on Parallel Machines. Davis: University of California; 2011.

[4] Yuen DA, et al. GPU Solutions to Multi-scale Problems in Science and Engineering. Springer; 2013. http://www.springer.com/us/book/9783642164040

[5] Zahran M. Graphics Processing Units (GPUs): Architecture and Programming (Multi-GPU Systems) – Lecture. [cited January 9, 2017] [Internet]. Available from: http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture9.pdf

[6] NVIDIA. What is Gpu-Accelerated Computing. [Internet] 2016 [cited October 8, 2016]. Available from: http://www.nvidia.com/object/what-is-gpu-computing.html

[7] Cruz FA. Tutorial on GPU Computing With an Introduction to CUDA. [Internet] 2009 [cited January 10, 2017]. Available from: http://lorenabarba.com/gpuatbu/Program_files/Cruz_gpuComputing09.pdf

[8] An Introduction to Modern GPU Architecture, Ashu Rege, Director of Developer Techno logy NVIDIA. [cited December 31, 2016]. http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf

[9] Kirk DB. W-mWH. Programming Massively Parallel Processors: A Hands-on Approach. Elsevier Inc. 2013.

[10] Rosenberg O. NVIDIA GPU Architecture: From Fermi to Kepler. Internet 2013 [cited January 15, 2017]. Available from: https://moodle.technion.ac.il/pluginfile.php/375213/mod_resource/content/1/Lecture%2313-FromFermitoKepler.pdf

[11] Vu Dinh DM. Graphics Processing Unit (GPU) Memory Hierarchy. Internet 2015 [cited January 3, 2017]. Available from: http://meseec.ce.rit.edu/551-projects/spring2015/3-2.pdf

[12] Harris, M. Tesla GPU Computing, A Revolution in High Performance Computing. Internet 2009 [cited January 3, 2017]. Available from: http://www.lsr.nectec.or.th/images/f/f2/Overview.pdf

[13] Kardos J. Efficient Data Transfer, Advanced Aspects of CUDA. Internet 2015 [cited January 3, 2017]. Available from: http://www.youtube.com/watch?v=Yv4thF9tvPo

[14] Rosenberg O. Introduction to GPU Architecture Internet [cited January 17, 2017]. Available from: http://haifux.org/lectures/267/Introduction-to-GPUs.pdf

[15] Evolution of PCI Express as the Ubiquitous I/O Interconnect Technology. Internet 2016 [cited January 17, 2017]. Published on Apr 8, 2016 In this video from the 2016 OpenFabrics Workshop, Debendra Das Shama presents: Available from: https://www.youtube.com/watch?v=eU5-6ogW1iY

[16] Fun and Easy PCIE – How the PCI Express Protocol works. 2016 [cited January 17, 2017]. Available from: https://www.youtube.com/watch?v=sRx2YLzBIqk

[17] Lawley J. Understanding Performance of PCI Express Systems. 2008. https://www.xilinx.com/support/documentation/white_papers/wp350.pdf

[18] How To Build and Use a Multi GPU System for Deep Learning 2014-09-21 by Tim Dettmers. Internet 2014 [cited January 17, 2017]. Available from: http://timdettmers.com/2014/09/21/how-to-build-and-use-a-multi-gpu-system-for-deep-learning/

[19] NVIDIA. GPUDirect. Internet [cited January 17, 2017]. Available from: https://developer.nvidia.com/gpudirect

[20] Marsden O. What is the Best Option for GPU Programming. Internet 2014 [cited January 14, 2017]. Available from: https://www.researchgate.net/post/What_is_the_best_option_for_GPU_programming

[21] Harris M. An Easy Introduction to CUDA C and C++. Internet 2012 [cited December 28, 2016]. Available from: https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/

[22] Sourouri M, et al. Effective multi-GPU communication using multiple CUDA streams and threads. In 2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS). Hsinchu, Taiwan. IEEE: 2014.

[23] Micikevicius P. Multi-GPU Programming. Internet 2011 [cited December 29, 2016]. Available from: https://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf

[24] Harris MBM. CUDA Multi-GPU Programming Internet [cited February 19, 2017]. Available from: http://people.maths.ox.ac.uk/gilesm/cuda/MultiGPU_Programming.pdf

[25] Badgujar HY. How to Mix Mpi and Cuda in a Single program. Internet 2014 [cited February 19, 2017]. Available from: https://hemprasad.wordpress.com/2014/12/19 how-to-mix-mpi-and-cuda-in-a-single-program/

[26] Alfthan SV. Introduction GPU Computing. Internet 2011 [cited February 19, 2017]. Available from: http://www.training.prace-ri.eu/uploads/tx_pracetmo/GPU_intro.pdf

[27] Bernaschi M. Multi GPU Programming (With MPI) Internet 2014 [cited February 19, 2017]. Available from: http://twin.iac.rm.cnr.it/Multi_GPU_Programming_with_MPI.pdf; http://on-demand.gputechconf.com/gtc/2014/presentations/S4236-multi-gpu-programming-mpi.pdf

[28] Laboratory ORN. GPUDirect: CUDA aware MPI. Internet 2016 [cited February 19, 2017]. Available from: https://www.olcf.ornl.gov/tutorials/gpudirect-mpich-enabled-cuda/

[29] Jill Reese SZ. GPU Programming in MATLAB. Internet [cited February 14, 2017]. Available from: https://www.mathworks.com/company/newsletters/articles/gpu-programming-in-matlab.html

[30] Suh JW, Kim Y. Accelerating MATLAB with GPU Computing: A Primer with Examples. Newnes; Morgan Kaufmann; 2nd December 2013; 258. eBook ISBN: 9780124079168, Paperback ISBN: 9780124080805

[31] Dean L. GPU Computing with MATLAB. MATLAB Products MathWorks. 2010 [cited February 19, 2017]. http://on-demand.gputechconf.com/gtc/2010/presentations/S12267-GPU-Computing-with-Matlab.pdf

[32] Mathworks. MATLAB Distributed Computing Server. Internet 2016 [cited February 20, 2017]. Available from: https://www.mathworks.com/help/mdce/index.html

[33] Mathworks. What Is a Parallel Pool. Internet 2016 [cited February 20, 2017]. Available from: https://www.mathworks.com/help/distcomp/parallel-pools.html

[34] Zoran Dabic RL, Simonian E, Singh R, Tande S. An Introduction to OpenACCl. Internet [cited January 17, 2017]. Available from: http://heather.cs.ucdavis.edu/OpenACCDir/Intros158/DabicLutrellSimonianSinghTandel.pdf

[35] Rasmuss GMMNMR. An Introduction to OpenAcc ECS 158 Final Project Robert. Internet 2016 [cited January 17, 2017]. Available from: http://heather.cs.ucdavis.edu/OpenACCDir/Intros158/GonzalesMartinMittowRasmuss.pdf

[36] Killian W. An Introduction to OpenACC. 2013 Internet [cited January 17, 2017]. Available from: https://www.eecis.udel.edu/~wkillian/latest/resources/OpenACC.Lecture.CIS-C879.Spring2013.pdf

[37] Oscar Hernandez RG. Introduction to OpenACC. Internet 2012 [cited January 17, 2017]. Available from: https://www.olcf.ornl.gov/wp-content/training/electronic-structure-2012/IntroOpenACC.pdf

[38] Larkin J. An OpenACC Example. Internet 2012 [cited February 20, 2017]. Available from: https://devblogs.nvidia.com/parallelforall/openacc-example-part-1/