From Shared Memory to Array Processing

An Overview of Distributed Consistency Models, GPGPUs and Vector Processors

Carlos J. Melegrito
Student — Faculty of Science & Faculty of Information Technology
Monash University
Clayton, VIC, Australia
e-mail: cjmel2@student.monash.edu

***Abstract*—This report provides an overview and explanation of three key topics in the field of parallel computing: distributed memory consistency models, general purpose computing on graphics processing units, and vector processors.**

Parallel computing; memory consistency models; GPGPUs; vector processors;

I. INTRODUCTION

Parallel computing — a discipline of computer science and one of the core foundations of modern computing — involves the computation of data or the execution of multiple programs across several processors simultaneously. The following topics covered in this report form a part of the history of parallel computing, introduced as insights from the inception of memory consistency models, the invention and newfound utilization of the GPU to the advancements made by vector processing.

Distributed (Memory) Consistency Models are conceptual models used by programmers and systems designers to understand the semantics of read/write operations on multi-processor systems with shared memory. They are used because it makes it easier to understand the behavior and order of memory operations in a system that requires concurrent memory access.

General Purpose Computing on GPUs (GPGPUs) involves passing non-graphical, general computation — such as raw mathematical data or batch-processing functions — from the **Central Processing Unit** (CPU) to the **Graphics Processing Unit** (GPU). Due to the GPU's parallel architecture, GPUGPUs enables faster processing than a traditional CPU could do alone — and has paved the way for many advancements in modern personal computing.

Vector Processors are special-purpose computers that can process large sets of data concurrently. They are more powerful than traditional CPUs in the sense that they use less bandwidth in terms of memory accesses and that its architecture supports instruction sets suited for vectors. Vector Processors are used in supercomputers because of their bulk and highly-parallel processing abilities.

II. DISTRIBUTED CONSISTENCY MODELS

A. Parallel Computing, Distributed Shared Memory and Memory Consistency Models

One of the biggest advancements in computational performance is the rise of parallel computing architectures through the use of distributed systems. Distributed systems are a collection of independent computers that appear as one single coherent system to its users. That is, within a distributed system, multiple CPUs (for example) can be networked together to spread the workload of compute-intensive programs across multiple processors.

In the general sense, distributed systems can work either one of two ways. Either: each CPU in the system — with an onboard memory — works on its own allocated tasks, then passes messages around the system. Or: the CPUs share a single (potentially, virtual) memory address space, work on their allocated tasks, and then communicate directly via the shared memory using read/write operations. The latter type of distributed computing is classified to be using a **Distributed Shared memory** (DSM) [1].

The use of a DSM brings with it many advantages — in particular, the shielding of the programmer from any send/receive message-passing primitives, thus abstracting the entire distributed system as one computer. However, due to the nature of shared memory systems, it becomes complex to understand the order and behavior of concurrent memory access.

P1	P2
Read(X)	Write(Y)
Write(Y)	Read(X)
Write(Z)	Write(Z)

Figure 1. Two processors operating on a shared memory space

Consider the above diagram. Two processors, P1 and P2, are connected to a shared memory address and perform two read and write operations in parallel. Reading downwards, notice how operations in P1 can happen before, after or even during the operations in P2. Without a "guide" to determine the behavior and order of these operations — it can become problematic to understand how a particular distributed memory system works.

As such, a programmer may require a high-level model of understanding the semantics behind shared memory operations. This conceptual model of determining the behavior of memory access is known as a Distributed Memory Consistency Model (or consistency model, in short). A consistency model provides an interface — or a "contract" — for programmers and systems designers to determine the order in which read/write operations will appear to execute for a system with DSM [2].

B. Different Types of Memory Consistency Models

There are various ways to implement a conceptual model for systems with DSM. Below are some of the most common examples of consistency models in use today.

1) Strict Consistency

CONDITIONS

- Any memory read should return the most recent value written, as per a universal time axis
- All memory operations will appear atomic and sequential
- Any overlapping operations will appear discretely ordered

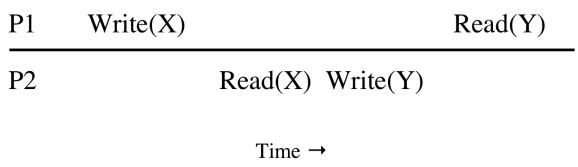

Figure 2. Read/write operations across two processors over a period of time [3]

The above diagram illustrates Strict Consistency in action: P1 writes to memory address a, and subsequently, P2 writes to memory address a. P1 then reads from address a, taking the most-recently written value by P2.

2) Sequential Consistency

CONDITIONS

- The outcome of any execution will be the same no matter the order of memory operations
- The operations of any individual processor appear in the order specified by its local program

```
// Global variables set
x = false
y = false

// In Processor 1
x = true
if (y) {
    … = y
}

// In Processor 2
y = true
if (x) {
    … = x
}
```

Figure 3. Sample pseudocode showing the interaction between two processors P1 and P2

In the figure before, the order of memory accesses is determined by the order in which they were given. That is, accessing X in P2 is guaranteed to contain the new value that was previously assigned in P1.

3) Causal Consistency

CONDITIONS

- At a processor, the local order of events is termed the Causal Order
- A write causally happens before a read is issued by another processor (if the read operation returns the value written by the write)
- The transitive closure of the above two conditions is the Causal Order

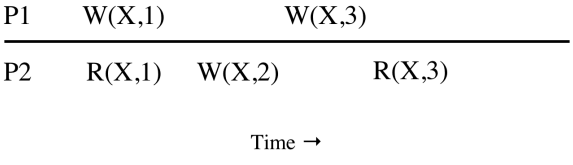

Figure 4. Processors P1 and P2 interacting over time [3]

In the above graph, W(x,1) and W(x,2) are considered to be causally related. This is because R(x,1) happens before W(x,2). *Note: the above notation "W(X,1)" is read as*

"Write the value '1' in address X", and "R(X,1)" is read as "Read the value '1' from address X".

4) Pipelined RAM (PRAM) Consistency

CONDITIONS

- Any write operation issued by a processor is seen by others in the order they were issued
- Write operations from different processors may appear to other processors has having executed in different orders

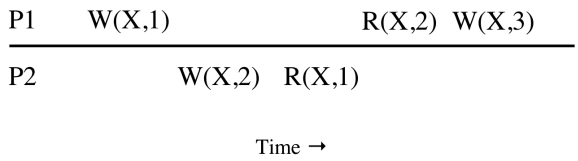

Figure 5. A timeline of two processors P1 and P2 interacting [3]

The timeline above shows that PRAM Consistency follows a first-in-first-out (FIFO) behavior — hence the term "pipeline". W(x,1) is done by P1 and as a result any processor that happens to execute R(x,1) will receive the same result until a new write operation is done by any other processor [4].

5) Weak Consistency

CONDITIONS

- All write operations are propagated to other processes
- All write operations done elsewhere are synchronized locally
- Access to sync variables are sequentially consistent
- Access to any sync variable is not allowed unless all write operations have finished
- No data access is allowed until all previous synchronizations have been performed

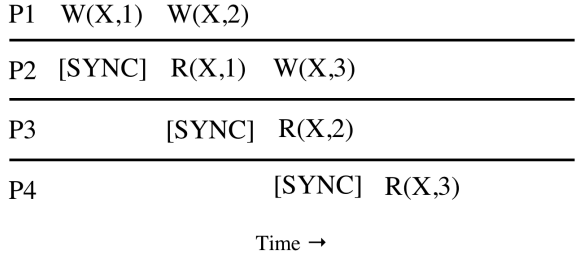

Figure 6. Concurrent memory access across four processors, P1 to P4, over time [4]

The synchronization step S in the above diagram pulls any new write operations from the other processors.

6) Release Consistency

CONDITIONS

- Two types of synchronization variables are used; the first of which is *Acquire*, which indicates that all write operations from other processors must be reflected at this point
- And *Release*, which indicates that any operations done locally in one processor should be sent to all other processors

C. Memory Consistency Models in Practice

The ideal consistency model for a particular system using DSM is one that utilizes performance and is easy to understand. Performance is usually enhanced by more complex, lower-level programming — but at the cost of an intuitive structure or concept of how it should work. Overall however, the balance between the two when deciding over a consistency model is dependent upon two factors: the target application, and the choice of programming easy and maintainability [5].

Distributed Consistency Models have their uses not only in small-scale, physically-networked CPUs, but also in many of today's internet applications. Read/write operations for applications like these can occur concurrently between a user's mobile device and a server stationed remotely — all the while being constantly developed by teams of software engineers and systems designers. As such, it would have been incredibly complex to maintain software at this scale if a consistency model had not been agreed upon [5].

III. GENERAL PURPOSE COMPUTING ON GPUS

A. From Video Games to High-Performance Computing: A Brief History of GPUs

Ever since their inception as an independent display processor in the late 1970s, Video Chips — or now more formally, GPUs — have played an important role in the advancement of modern computing. Originally developed as a means to improve the quality and performance of onscreen computer graphics, GPUs have since made their way to becoming an integral part of parallel computing [6].

The need for a specialized graphics processor came out of the limitations of CPUs. CPUs were specifically made to execute application logic and handle user interaction. But as graphical user interfaces (and in particular, video games) surged in popularity, more graphics processing was needed to be done on top of a CPU's core workload. Memory and processing power was expensive; thus, the GPU was invented [7].

Figure 7. 3D Graphics within a game called “Minecraft”

GPUs were designed to compute graphical data very efficiently. They are not replacements for the CPU — instead, are seen as an extension for it. The CPU would run the main application code, passing any graphical computation to the GPU. This decoupling of processing responsibility resulted in a visually-pleasing and highly-interactive user experience. As such, advancements in GPU technology have since been focused on improving the performance and quality of computer graphics [7].

It wasn't until the early 2000s that another use for GPUs had been discovered. It was found that if raw mathematical data (in particular, matrices, vectors or large arrays) were translated into graphical data, the GPU could perform a much faster computation over it than any traditional CPU. Furthermore, with the introduction of APIs like OpenCL or NVIDIA's **Compute Unified Device Architecture** (CUDA), raw data no longer has to be converted and programs can be passed directly to the GPU, making it easier to use [8].

The speed, performance and efficiency of using non-graphical, General Purpose Computing on GPUs has paved the way for advancements in personal computing and many more modern, practical uses.

B. How It Works: GPGPUs

Unlike CPUs, GPUs were built to process mass amounts of individual data in parallel. In the graphics processing context, if a CPU was tasked to modify a grid of pixels, it would have to pass over each pixel sequentially, and if done trivially, through a `for` loop. On the other hand, a GPU would instead compute all (if not, most of) the pixels at the exact same time. In the context of general computing, this parallelization can be used to batch-compute data [9].

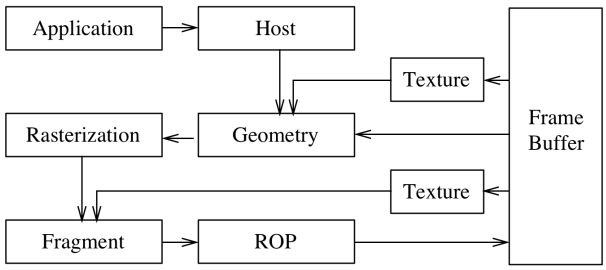

Figure 8. The Graphics Pipeline [9]

The process by which data is passed to the GPU for computation is known as the Graphics Pipeline. From a high-level perspective: graphical data is sent from an **Application** in the CPU, communicated through the **Host** (an interface between the CPU and the GPU), then the data is computed — sometimes combined with data stored in the **Frame Buffer** (video memory). The resulting data is a set of vertices in 3D Space, converted to triangles through **Geometry** that becomes a 3D model. This scene is then mapped onto a 2D projection of pixels through **Rasterization**, rendered with more detail (**Fragment**) and finally, the image is prepared for display via the **Raster Operation** (ROP) [9].

In the context of general computing, the most useful aspect of the above Graphics Pipeline is a process that happens within the GPU called *Shading*. **Shaders** — programs that add aesthetic value to a 3D image through lighting, color and other image effects — can be programmed in a way to produce a variety of different rasterizations of a 3D image. For example, Shaders can be used to make a 3D sphere look matte, shiny or even made of gold. Beyond visual utility however, Shaders can be programmed to process non-graphical data — that is, general computation [10].

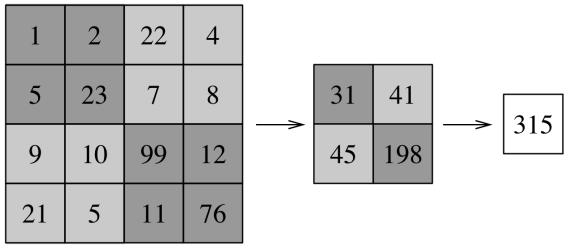

Figure 9. Summing a matrix of values by “Resizing” it

Take for example the task of summing an incredibly large array of data. Done sequentially, the complexity of a trivial algorithm for this would take linear time. However, if the array was converted into an image format — a grid of pixels where each pixel holds the corresponding value of an item in the array (which is a large matrix) — it can then be processed in parallel through the GPU. In this case, a Shader that specializes in image resizing can be re-programmed to sum the values of any 4 adjacent "pixels" (cells in the matrix) into one "pixel" — all at the same time. The GPU would then run through in parallel the entire matrix again, summing any 4 adjacent "pixels" into one — and so on, until eventually ending up with a single "pixel" that holds the value of the final sum.

```
for pixel in grid {
    modify(pixel)
}
```

Figure 10. Code written the traditional way, as a `for` loop

```
kernel function (pixel, grid) {
    modify(pixel)
}
```

Figure 11. The same code written in the form of a *kernel* — a small, reusable program used by individual processes concurrently

Although the above example was simple, the process of having to convert raw data into a format that the GPU can compute becomes more cumbersome as complexity increases. Because of this, the above traditional method of Shader programming has been improved by the more modern technique of simply running the code directly on the GPU via the use of APIs. An example of this is the task of finding an element in a non-sorted array: code could simply be written to compare each element in the 1-dimensional array with the query all in parallel. If the element in the array does not match the query, it is ignored, otherwise, its index is stored and passed as part of the result.

```
#include <stdio.h>
#include <cuda.h>

// Initialise global kernel function
__global__void helloWorld(char*);

// Function to run
int main(int argc, char** argv) {

    // Prepare the string to print
    int i;
    char str[] = "Hello World!";

    // Iterate over the string
    for (i=0; i<12; i++) {
          str[i] = str[i] – i;
    }

    // Prepare the string for parallel
processing
    char *d_str;
    size_t size = sizeof(str);

    // Use CUDA API to allocate string to
memory
    cudaMalloc((void**)&d_str, size);
    cudaMemcpy(d_str, str, size,
cudaMemcpyHostToDevice);
    dim3 dimGrid(2);
    dim3 dimBlock(6);

    // Pass the string to the GPU
    helloWorld<<<dimGrid,
dimBlock>>>(d_str);

    // Deallocate memory
    cudaFree(d_str);

    // Print result
    printf("%s\n", str);

    return 0;
}

// Define the kernel
__global__void helloWorld(char* str) {
    int idx = blockIdx.x * blockDim.x +
threadIdx.x;
    str[idx] = str[idx] + idx;
}
```

Figure 12. Sample C code written with the easy-to-use CUDA API

C. *Art, Science and More: Applications of GPGPUs*

The bulk, parallel processing capabilities introduced by GPGPUs has introduced the possibility for more high-performance computing per individual computer. This idea of using the graphics card for more than just graphics processing has many practical uses today, including advancements in consumer software, data processing in fields like Astrophysics, Chemistry and Medicine, and even for business use such as in data centers for load balancing [11].

Figure 13. Live "8-bit" camera filters in an iOS app called "BitCam" made possible by GPGPUs

In consumer software, due to mass-production of more advanced GPUs and the introduction of easy-to-use, high-level APIs like Apple's *Metal* that enable for GPGPUs, users now have access to features and programs once deemed exclusive to supercomputers — such as image noise reduction, live 3D filters and audio processing.

In the science and medicine industry, clusters of GPUs added on top of CPUs are used by supercomputers to batch-process terabytes of raw mathematical data: from physical calculations (in the form of vectors or matrices) for simulating the movement of distant stars to analyzing protein folding (generating various forms of a chemical in parallel).

D. *The Future of GPGPUs: What's Next?*

Recent advancements include more easy-to-use APIs such as WebGL (using GPGPUs via the web) and NVIDIA's research into Echelon — a new GPU architecture specifically geared towards general computing that stores more memory than a traditional GPU. In terms of future applications, GPGPUs will see more usage in artificial

intelligence (processing more input through parallelism) and in the science research industry — in particular, using asynchronous computation to produce a 1:1 mapping of every molecule in the human brain [12].

By *Moore's Law*, GPUs (and CPUs in turn) will become more efficient, performant and will continue to provide many more practical uses with every advancement of the technology.

IV. Vector Processors

A. Scalar versus Vector Processors

Consider the task of having to add together each element of two very large one-dimensional arrays. The goal is to produce a single array of the same length in which each element in the final array is a sum of the elements from the other two arrays in the same position. To carry out this task, a traditional CPU would have to iterate over each element of both arrays one at a time, store the resulting sum in memory, then continue to build the final array step-by-step until it is ready to be returned.

```
function combine(array_1, array_2) {
    result_array = []
    for (i=0; i<len(array_1); i++) {
        sum = array_1[i] + array_2[i]
        result_array.append(sum)
    }
    return result_array
}
```

Figure 14. A trivial solution to the above problem written in pseudocode

Although trivial and intuitive, this sequential — scalar — way of processing data is very slow. Even with the fact that this task can be done more efficiently and can be accomplished in nanoseconds on today's more modern CPUs, the process still takes linear time complexity. Furthermore, this specific task also requires a lot of memory accesses in terms of fetching instructions, which accumulates on top of the total processing time [13].

The limitations of scalar CPUs such as the problems mentioned above gave rise to the invention of Vector Processors. Also known as Array Processors, Vector Processors work by computing multiple sets of data in parallel. In other words, they are processors that can operate on an entire vector in one instruction. To put this in context, a Vector Processor could accomplish the aforementioned task of combining two arrays in just a handful of steps.

```
function combine(array_1, array2) {
    return array_1 + array_2
}
```

Figure 15. The same solution in pseudocode, except written for Vector Processors in the form of a kernel

Vector Processors also reduce the memory access bandwidth because the number of instructions fetched are less. Moreover, Vector Processors access memory one block at a time, resulting in lower memory latency. Designed in the 70s by the supercomputing pioneer, Seymour Cray, Vector Processors were built to more efficiently operate on data that can be executed in a highly — or even completely — parallel manner [14].

B. Vector Processor Architecture

Vector Processors work like traditional CPUs except operate with a different set of instructions (called Vector Instructions) and parallelize operations either on the instruction level (Instruction-Level Parallelism or ILP), thread level (Thread-Level Parallelism or TLP) or even on the data level (Vector Data Parallelism or DP). Because of this, programs for Vector Processors are written such that the data can be processed concurrently or — if the program happens to be large enough — divided into smaller ones which are then solved simultaneously [14].

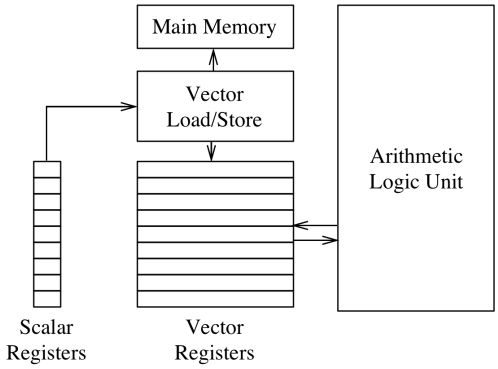

Figure 16. Vector Processor Architecture [14]

From the Main Memory, data is loaded into the Vector Registers which operate like FIFO queues. Each register can hold between 50 to 100 floating point values. The instruction set for Vector Processors loads a vector register from a location in memory, then performs operations on elements in each register, then stores the data back to memory from the registers.

TABLE I. Example VMIPS Instructions [13]

Instruction	Operands	Operation	Comment
ADDV.D	V1, V2, V3	V1 = V2 + V3	Vector + Vector
ADDSV.D	V1, F0, V2	V1 = F0 + V2	Scalar + Vector
MULV.D	V1, V2, V3	V1 = V2 × V3	Vector × Vector
MULSV.D	V1, F0, V2	V1 = F0 × V2	Scalar × Vector
SUBV.D	V1, V3, V3	V1 = V2 – V3	Vector – Vector
SUBSV.D	V1, F0, V2	V1 = F0 – V2	Scalar – Vector
DIVV.D	V1, V2, V3	V1 = V2 ÷ V3	Vector ÷ Vector
DVIVSV.D	V1, F0, V2	V1 = F0 ÷ V2	Scalar ÷ Vector
DVIVVS.D	V1, V2, F0	V1 = V2 ÷ F0	Vector ÷ Scalar
LV	V1, R1		Load vector register v1 from

			memory starting at address R1
SV	R1, V1		Store vector register V1 into memory starting at address R1
LVWS	V1, (R1, R2)		Load V1 from address at R1 and stride at R2 as R1 + i × R2
SVWS	(R1, R2), V1		Store with stride
LVI	V1, (R1 + V2)		Load V1 with vector whose elements are at R1 + V2(i)
SVI	(R1+V2), V1		Store V1 to a vector whose elements are R1+V2(i)
CVI	V1, R1		Create an index vector by storing values i × R1 into V1

Although vector instructions are dependent on the type of processor used, there is a general set of operations used by all vector processors that are specific to vector operations. The table above lists some of the most common sets of vector instructions used in for the VMIPS processor developed in 2001. *Note: the VMIPS processor contains Floating Point Multiply, Add and Divide components, an Integer Add/Shift and a Logical component.*

```
; Load vector from memory
LV V1, R1
; Multiply this vector by a scalar
; And store result as V2
MULVS.D V2, V1, F0
; Load another vector from memory
LV V3, R2
; Add and store result as V4
ADDV.D V4, V2, V3
; Store the result back into memory
SV R3, V4
```

Figure 17. Sample code written in the VMIPS Instruction Set

C. *Vector Processors In Use*

Because of their capacity to run large instruction sets in parallel computers, vector processors are ideal for processing or comparing large sets of data. For example, algorithms used for string processing and cryptography can be used for pattern recognition in biomedical research, such as finding repetitions in DNA or RNA sequences [16].

Vector Processors are not limited to data-processing supercomputers — its use has found its way into consumer electronics like video game consoles. The Cell Processor, developed by Sony in collaboration with Toshiba, is hybrid of vector-scalar processor used in the *PlayStation 3* [17].

V. CONCLUSIONS

A memory consistency model for a system with distributed shared memory provides an overview of how memory operations should appear to behave. Some of these consistency models can be very strict — that is, adhere to one specific condition — or relaxed — which takes advantage of abstractions, such as synchronization, to determine how and when shared memory should be accessed.

General Purpose Computing on GPUs work by passing large amounts of individual data from the CPU to the GPU via the Graphics Pipeline and then parallelizing the computation. This insight of passing non-graphical, general computation has advanced the mere GPU from a simple extension for processing onscreen graphics, to become the more versatile and high-performance processor used in a lot of today's modern computing.

Unlike traditional scalar CPUs, Vector Processors are parallel computers designed to operate on an entire vector in one instruction. They reduce the fetch and decode bandwidth when accessing instructions in memory, and their architecture supports instruction sets that operate on vectors stored in FIFO registers. Originally founded by Seymour Cray in the early 70s, Vector Processors have since found their way to modern supercomputers and consumer electronics.

REFERENCES

[1] G. Kourosh. "Memory Consistency Models for Shared-Memory Multiprocessors," Standford University Technical Report CSL-TR-95-685. 1995.

[2] J. Protic, I. Tartalja and M. Tomasevic, "Memory consistency models for shared memory multiprocessors and DSM systems," Proceedings of 8th Mediterranean Electrotechnical Conference on Industrial Applications in Power Systems, Computer Science and Telecommunications (MELECON 96), Bari, 1996, pp. 1112-1115 vol.2.

[3] G. Radhika, et al. "Consistency Models in Distributed Shared Memory Systems," International Journal of Computer Science and Mobile Computing, Vol. 3, Issue 9, pg. 196-201. 2014.

[4] A. Kshemkalyani, et. al. "Distributed Shared Memory," Distributed Computing: Principles, Algorithms and Systems, Cambridge University Press. 2008.

[5] J. Protic, M. Tomasevic and V. Milutinovic, "Distributed shared memory: concepts and systems," in IEEE Parallel & Distributed Technology: Systems & Applications, vol. 4, no. 2, pp. 63-71, 1996.

[6] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone and J. C. Phillips, "GPU Computing," in Proceedings of the IEEE, vol. 96, no. 5, pp. 879-899, May 2008.

[7] H. Gamaarachchi, M. Fawsan, F. Fasna and D. Elkaduwe, "User-friendly interface for GPGPU programming," 2017 6th National Conference on Technology and Management (NCTM), Malabe, Sri Lanka, 2017, pp. 99-104.

[8] NVIDIA. (2017). What Is GPU-Accelerated Computing? [Webpage]. Available http://www.nvidia.com/object/what-is-gpu-computing.html

[9] L. Wei. (2005). A Crash Course on Programmable Graphics Hardware [PDF]. Available http://graphics.stanford.edu/~liyiwei/courses/GPU/paper/paper.pdf

[10] W. Mark, R. Glanville, et al. "A system for programming graphics hardware in a c-like language", ACM Trans. Graph. 22, 3, 896-907. 2003.

[11] M. Harris. (2005). General-purpose computation using graphics hardware [Website]. Available http://www.gpgpu.org/

[12] F. James and M. Steve. "Using Multiple Graphics Cards as a General Purpose Parallel Computer : Applications to Computer Vision," University of Toronto, Department of Electircal and Computer Engineering, Canada. 2004.

[13] A. Eman, et al. (2012). Vector Processors [PDF]. Available https://www.cs.uic.edu/~ajayk/c566/VectorProcessors.pdf

[14] J. Hennessy, et al. "Computer Architecture, A Quantitative Approach," Morgan Kaufmann. 1990.

[15] R. Kirchner and U. Kulisch, "Arithmetic for vector processors," 1987 IEEE 8th Symposium on Computer Arithmetic (ARITH), Como, Italy, 1987, pp. 256-269.

[16] P. David. (1998). Lecture 7: Vector Processing [PDF]. Available https://people.eecs.berkeley.edu/~pattrsn/252S98/Lec07-vector.pdf

[17] A. Brian, et al. (2002). Vector Processors [PPT]. Available www.cct.lsu.edu/~scheinin/Parallel/VectorProcessors.pp