Review of Memory/Cache Management Technologies used on Heterogeneous Computing Systems

Mustafa Ali
Computer Science and Engineering Department
American University of Sharjah
Sharjah, UAE

Tarik Ozkul
Computer Science and Engineering Department
American University of Sharjah
Sharjah, UAE
Tozkul {at} aus.edu

Abstract— **Traditional ways of increasing computer performance has been increasing speed and bit size. Although this kept us going for more than half of a century, methodology has hit a major road block due to power consumption and heat dissipation. The remedy found for this problem has been creating multiple cores/heterogeneous computing systems on a single chip. By introducing multiple cores, efficient memory execution and correct data sharing the performance can be increased without paying power penalty. In this paper, we discuss the implications caused by integrating multiple cores on a single chip and review some of the techniques generated for solving these complications. Among these are techniques for establishing effective communications, data integrity and buffer management.**

Keywords-heterogeneous computing; memory mangement; cache management

I. Introduction

Hardware evolution of CPU, in terms of speed and performance followed the Moore's law. In the past 50 years, CPU hardware has provided exponential growth as predicted by Moore's law with a significant accuracy. Moore's law states that "the number of transistors on integrated circuits doubles approximately every two years". However it is now believed that CPU has reached its threshold in terms of increasing number of transistor on integrated circuits. The reason for reaching this threshold is the power consumption and heat dissipation issues which are the direct result of increased number of transistors and density. Therefore, researchers shifted momentum towards next paradigm i.e. heterogeneous computing systems such as Graphical Processing Units (GPU); which contains multiple cores or computing kernels on a single chip. Rather than increasing speed and clock rate for a single computing unit, multiple core units running in parallel at optimized speed can increase the computational performance.

Nowadays computers include both CPU and a GPU on separate chips combined on a single motherboard or, alternatively CPU's can be located on mother boards and GPU's located on graphic cards.

As we have entered in the Era of GPU Computing [1], there is expectation that the exponential growth may continue at a higher emergent level. GPU's have brought immense cost – effectiveness in terms of computing and visualization, which has shifted the focus to develop GPU's with more and more cores and with greater computing powers. In the future, it is expected that GPU's shall be employed to handle all the computing challenges in place of CPUs [2].

The performance benefits of GPUs are enormous, but as always with new technologies, it gives birth to new complexities. The applicability and performance of computing on heterogeneous computing systems (GPU) is limited and constrained by CPU–GPU memory/cache communication. In heterogeneous computing systems (GPU) and CPU both have separate memories where each of them is able to efficiently access their own memories [3].

However, when a program running on CPU or GPU needs data, there needs to be fast way of sharing data correctly and efficiently. These data–structures and management rules can be addressed as "cache management" in memory operations. However, incorrect communication can cause programs to access stale or inconsistent data. Another major issue of with this architecture is the fact that global memory of the GPU is located outside the chip or graphic card. This is the only way in which all the computing systems of GPU can communicate together. The fetching of data again and again from global memory can be costly. This can result in slowness and threads can starve if data is not available. Moreover, the computing units within the GPU also include instructions that can manage global memory in the GPU. This may result in wasting of clock cycles which can be computationally inefficient [4].

In this paper, discussion will start with the background of heterogeneous computing systems (GPUs) which include their architecture, performance and usability; and will continue with recent technological innovations that deals with complexity of CPU–GPU memory/cache communication and how multiple computing units within a GPU communicate without fetching data again and again from global memory.

There are innovative techniques used for executing threads on different work items so that;

- They can share data correctly and efficiently,
- How work items communicate without accessing GPUs global memory,

- How to handle a response for an instruction from a work items,
- Determination of visibility rules for a data item so that how it can be made visible to other work items,
- How cache operations are executed during the execution of an instruction.

II. HISTORICAL EVOLUTION

The graphical processing unit (GPU) is a chip that contains high number of parallel microprocessors. It was originally manufactured to accelerate 2D or 3D graphic processing to reduce the work load of CPU. However, recent GPUs are composed of large number of computing cores which are able to perform operations in parallel with a very high memory bandwidth which enables them to process large amount of memory data. Due to this reason, researches were then interested in applying its computing power in everyday life.

The development of GPU hardware began from a single core and fixed function hardware pipeline application towards a combination of highly parallel programmable cores which can be used for general purpose computation and scientific computation. GPU technology always progressed by adding more programmability and parallelism in its core architecture. As a result of which it seems that eventually look like CPU core.

The graphic pipelining was first introduced in GPUs. This graphic pipelining led to the introduction of conceptual model of stages, through these stages graphic data was processed for computation with the help of hardware (GPU cores) and CPU software (OpenGL, DirectX) combination. The purpose for all the above idea was to simply convert coordinates from 3D space into 2D pixel space so that it can be displayed on the screen.

The evolution of GPU's started in early 1980's at that time they were just integrated frame buffers. They were boards of TTL logic chip that relied on the CPU. Professional Graphics Controller (PGA) was the first 2D/3D video card introduced by IBM. It used an on-board Intel 8086 microprocessor doing all the video related tasks instead of CPU. In late 1980's SGI (Silicon Graphics Inc.) was founded which was a high performance computer graphics hardware and software company. They introduced platform independent 2D/3D application programming interface (API) with the introduction of OpenGL which became a complex part of the design of modern graphics hardware. By 1993 Reality Engine board for graphic processing was released by SGI, due to which GPU hardware and graphics pipeline started to take real shape. At that time GPUs were only able to output one pixel per clock cycle, therefore CPU still had capacity to send more data to GPU for processing. Hence more pipelines were added in parallel to increase pixel processing in parallel for each clock cycle.

First GPU at consumer level was introduced in 1999 as NVIDIA's GeForce256 and ATI's Radeon 7500. In these systems PCI bus was replaced by Accelerated Graphics Port (AGP). These systems used "fixed function" pipeline, as once graphic was data sent, it could not be modified. In 2001 NVIDIA released GeForce 3. This was the first GPU with programmable pipeline and had the ability to program non – programmable parts of the pipeline. In next one year fully programmable graphic cards were invented and first wave of GPU computing started with introduction of DirectX9, by take advantage of the programmability now in the GPU hardware [5].

In 2006 NVIDIA introduce GeForce 8 series. This was a great evolution in the history of GPU because it contained massive parallel processors [6].

The first GPGPU (General Purpose Graphical Processing Unit) was introduced in 2009 as NVIDIA's Fermi architecture featuring true HW cache hierarchy, concurrent kernel execution, better double precision performance, combined memory address space and dual warp schedulers. After that, we had rapid progress in the development of GPUs since 2009.

The evolution of GPU hardware started from single core, fixed function pipeline which were only used for graphic processing, to a set of highly parallel programmable cores to be used for general purpose computations. Hence, the new GPU architecture has started looking like multi–core general purpose CPUs.

TABLE I. Most recent GPUs with their configurations

CARD	CORES	CORES/SM	SM	Comp. POWER
GTX Titan	2688	192	14	3.5
GTX 780	2304	192	12	3.5
GTX 770	1536	192	8	3.0
GTX 760	1152	192	6	3.0
GTX 690	3072	192	16	3.0
GTX 680	1536	192	8	3.0
GTX 670	1344	192	7	3.0
GTX 580	512	32	16	2.0

III. MOTIVATION BEHIND THE TRANSITION

The limitations in the programming environment led GPUs to be only used for graphics acceleration. The graphics acceleration included two dimensional (2D) and three dimensional (3D) graphics processing, this graphic processing was used in graphics and video application programming interfaces (APIs). With the introduction of multi – vendor supported OpenCL®, DirectCompute®, standardAPIs and supporting tools, GPUs are no longer limited to graphical processing only. However, there were still limitations about the environment which allowed the combination of a CPU/GPU to be used as easily as a CPU in programming tasks. Existing computing systems now include multiple processing devices such as CPU on mother board and a GPU on graphic card. This arrangement in a single computing system still includes significant challenges associated with:

(i) Necessity of having separate memory systems for both GPU and CPU,

(ii) Necessity of efficient scheduling of instruction as programs are now divided to be executed on CPU and GPU,

(iii) The service quality,

(iv) Necessity of having an efficient programming model as it will be now executed on both host and device.

All these facts have to be considered along with the necessity of minimizing power consumption.

As discussed above, CPU and GPU are located on different components, and even on boards. CPU typically located on the mother board and GPU on the graphic card. In some cases both of them may be integrated on a same motherboard. CPU and GPU both need to have their own system and global memory. So when a program is running on both CPU and GPU they need to communicate data amongst them, so for this, they need to exchange data among them which negatively affects memory latency and power consumption.

The purpose of this paper is to address the challenges mentioned above for a system consisting of GPU and CPU as both of them have separate memory and address spaces, CPU and GPU needs to communicate effectively and efficiently with some kind of rules so that they are able to perverse data and keep data integrity. However, another issue with the GPU is that multiple cores or work items can communicate only with the global memory, which is located off chip to the GPU. So if they share data they have to do off chip access which creates latency, slowness and cores or processor may remain idle while data is being fetched. To resolve this issue, there exist a technological innovation that defines “visibility rules” that specifies how CPU and GPU can communicate with each other and buffers and buffer management unit which help cores within a GPU to fetch data from global memory and keep it in their cache memories to avoid latency, slowness and starvation.

A. Some important Definitions:

In this section definitions of the terminologies that will be used throughout the paper will be summarized. These will be frequently used in the sections below; therefore it is important to elaborate them, so that the reader can have proper understanding of the terminologies used [7].

Shaders: Shaders are referred to as programmable computing units which may execute graphical and non – graphical operations (For example, programs written in OpenCL® or C Language).

Heterogeneous Computing System: It refers to computer system that consists of more than one processing units.

Accelerated Processing Devices (APDs): APDs is a combination of CPU, GPU, Multiprocessing devices, Data Parallel Devices.

Processing Logic: It contains control flow instructions, instructions for performing computations and instructions to access resources.

Visibility Ordering: Visibility ordering refers to an order which is represented by partial order in which data items are made visible to work items across multiple processors.

Data Visibility: Data visibility refers to visibility of the data items present in the core’s local memory which accessible to that core. Data item is visible when it present in the global memory and shared by all the cores.

B. Memory and Cache Management:

In this section technological solutions to problems that occur between CPU to GPU memory and cache communication are discussed.

The solution for CPU to GPU memory and cache communication provides how the threads and the other work items are executed on multiple processors while maintaining the integrity of data items when accessed by each computing kernel. Integrity of the data refers to the reading of latest values of the data items regardless on which work item data was updated last.

The solution for reducing latency and communication overhead between the GPU’s global memory and computing kernels within a GPU is dealt by introducing on chip solution to the GPU. We introduce buffer management unit on GPU chip and buffers within the local cache memory of each computing kernels and global memory of GPU. In process details are mentioned in the sections below.

1) Processing System:

The Fig. 1 below [7] shows the block diagram of heterogeneous computing system.

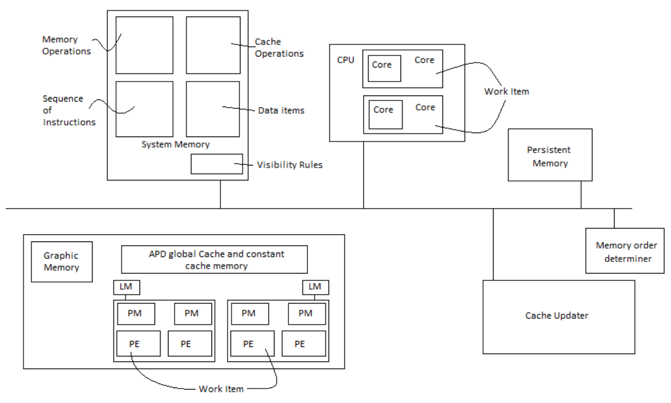

Figure 1: Block diagram of heterogeneous computing system

This is a heterogeneous computing system which consists of one or more CPU, APDs, system memory, persistent storage device, at least one system bus, memory order and a cache updater. All these components are embedded on single silicon die chip combining CPU and APD and providing unified programming environment.

The CPU is treated as host device which is responsible for the initiating and controlling the execution of the applications across CPUs and ADPs. The programming model followed on heterogeneous computing system is mainly SIMD (Single Instruction Multiple Data). Therefore instruction has to be initiated on host and executed on APD or GPU for parallel execution. Furthermore, a CPU is capable of two or more cores; each of them has access to system memory and cache memory to save data.

As discussed earlier APD is responsible for executing applications that can be executed in parallel such as graphic functions, graphic pipeline computations and geometric computations. APD normally consist of a global, one or more computing kernels and a graphic memory. Each computing kernel consists of its own private cache memory. Only global memory in GPU can communicate with CPU's local memory.

All the logical instructions, constant values and variable are kept in the system memory of the CPU during the execution of applications. System memory of CPU is connected to GPU's global by a system bus which is responsible for transporting the data between CPU and GPU.

Responsibility of memory order determiner and cache updater is to hold processing logic so that it can determine the visibility of data items and execute a cache operation according to it. For example, on issuance of load or store instruction memory order has to find data items and cache updater should perform cache operation according to the visibility rules. Visibility rules shall be discussed later in the paper.

System memory also holds sequence of instructions. Sequence of instructions is a representation of instructions appearing in source code or it's the order made by the compiler for the source code instructions.

According to system configuration global memory for the GPU lie on the common chip with other hardware components. The global memory is the only way different work items can work together if they have a common data to be shared amongst them. Although it is shown that each computing kernel has its own local memory, but this local memory is very limited in size and threads which are working outside this computing unit are not able to see this local memory. So it's quite clear that if computing kernel requires the data it has to go on global memory and fetch from there. This fetching of data again and again from global memory can be costly in terms of speed and latency and threads can starve for data.

To resolve this issue, buffers and buffer management techniques are introduced as a part of innovative solution introduced in [7]. These buffers are first in first out buffers and ring buffers. The main idea to resolve this issue is by including a Pipeline Management Unit (PMU) on integrated circuit (IC) which is on the same chip as GPU. This unit keeps the state information of buffers on the global memory. By this information GPU does not need to perform off chip memory access any more. In the sections below describes the functionality and working of pipe management unit along with buffer management techniques to enhance the communication.

a) Buffer Management in GPU:

The PMU functions by storing the starting address and the ending address of the buffers in the global memory. It can also store address of the buffer among many buffers with in buffers in global memory for the consumer thread to read data. It can also keep the track of generation of data from producer thread so that consumer thread can utilize this data for the execution. PMU is generally used to handle producer consumer requests. It may receive a request to store the data for the producer and to store the buffer address so that it is able to entertain the request from consumer without going to global memory of GPU. The Fig. 2 below illustrate the solution for the re–accessing of GPU to global memory, by including on chip pipeline management unit, which keeps the state information of all the buffers in the global memory.

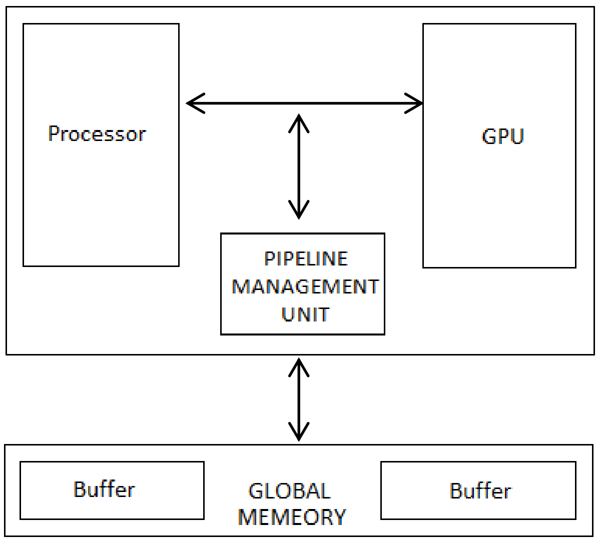

Figure 2: Pipeline Management Unit (PMU) on GPU

Global memory contains the common data for all the work items to be shared. We can see that CPU, GPU and PMU all are integrated on one chip. However, it's not necessary that CPU and GPU should be integrated on same chip. CPU can be referred as host machine; it can run many application programs. Executing instructions for those programs may lie on global memory or system memory. To exploit the power of GPU, CPU unloads the execution of tasks to the GPU which require massive parallel operations.

b) Block Diagram on Pipe Line Management Unit:

The Fig. 3 illustrates the block diagram in more detail. The inner structure of the GPU and Global Memory is explained in here [7].

In this block diagram it can be seen that GPU further consist of fixed computing units (ALU's) and programmable computing units (Shader processor), PMU (Pipeline Management Unit), an internal cache, scheduler and registers. Shaders are referred to as programmable computing units which may execute graphical and non – graphical operations, e.g. programs written in OpenCL® or C Language) [7].

Shader processors are composed of N programmable computing units which may be considered as shader cores, depending on the specifications for the device. Fixed function units are composed of N fixed-function compute units.

The local memory of each programmable computing unit is inaccessible to other computing unit. Producer may store data in one of the buffers in global memory and consumer thread may retrieve it from there. The integrated chip also contains the cache memory, rather than keeping it limited to the GPU. This cache memory can also be utilized to store some intermediate results produced by the producer thread and complete data may reside on global memory. These intermediate results are stored on buffers in the cache memory. These buffers normally functions as a cache memory such that they store partial amount of the produced data, so that it is accessible quickly as compared to accessing data from the global memory. The buffers are included in cache memory to avoid latency and power consumption associated with accessing off – chip memory. But due to space constraints it is not practical to use these buffers completely. Therefore strategy is to split data between these cache and global memory. These buffers in local cache and global memory allow threads to communicate with each other in a pipelining fashion.

A scheduler is also included which has the responsibility for assigning threads to various programmable compute units and fixed function units. For example a scheduler can perform load balancing task on programmable compute unit so that none of them is over utilized or underutilize.

In this block diagram we also include buffers in the global memory. These buffers and buffers in the local cache memory GPU are supposed to assist in organizing the workload scheduling for programmable compute unit and fixed – function unit.

The main component in this block diagram is PMU (Pipeline Management Unit). The functionalities for which PMU is responsible for stated below:

1. Managing states of buffers in global memory and states of buffer in local cache memory.
2. Storing the lengths of buffers and the number of buffers that are available to store the data.
3. PMU stores the header pointer, current offset, maximum depth of the on – chip registers.
4. In many cases buffers may require management such that which buffer to store data in or from which buffer to retrieve the data, determining where to store data and from where to retrieve the data inside the buffers.
5. Avoiding invalid accessing of data from buffers, so that data doesn't get corrupted.
6. PMU can store atomic counter on the registers. This atomic counter usually express whether one or programmable compute units are available. (For example, data is available to be read or whether two or more kernels are attempting to write or read at the same time from the same buffers).
7. Avoids re – ordering of the data by storing additional counter in registers, re – ordering of the data might cause threads to read in correct data. This additional counter can be referred as device atomic counter.
8. Avoid deadlock by storing information in registers.

c) Functioning of Pipeline Management Unit (PMU) :

As for the start programmable computer units execute one or more thread of a kernel on shader processor, as a result PMU receives a request to store data into or retrieve the data from the global memory. PMU must determine whether access is to be allowed for the request or not. If access is not allowed compute unit must start executing additional thread of the kernel. After this, PMU also indicates when access is available. If access is available PMU will determine the address within global memory, i.e. where the data is to be stored or from where data is to be retrieved for the requesting compute unit and bring it to the cache memory of GPU. This allows GPU to store or retrieve the data from the determined location within buffers. PMU is capable of determining the location within one of the buffers where the data is to be stored or retrieved from in global memory. In this case, kernel does not include the instruction for determining the location.

PMU also has ability to read more data than the requested data. In such cases, PMU may store data in local cache memory. PMU is also capable of storing state information of buffers within IC registers. This information may be received from CPU, which contains one or more starting and ending addresses of buffers in global memory along with the buffer where produced data is to be stored and buffer where data is to be retrieved.

The above functioning of PMU is applicable for two or more threads executed on same computing kernels. In such conditions priority is always given to producer thread so that data can be produced before consumer can retrieve it. However, if two threads are executed on different computing kernels, PMU will check will check whether data is available or not by producer thread because there can be a possibility that PMU may receive the request from consumer thread and producer thread simultaneously, prior to, or after receiving the request. In such situations if data is not ready, PMU may indicate other thread to execute and keep it waiting until the data is available.

a) Memory Operations:

Returning back to CPU – GPU memory communication, in this section we discuss how the memory operations are executed in a heterogeneous computing system. The Fig. 4 describes the sequence of memory operations [7].

One more computing units on CPU and APD are executed with a sequence of instructions which can be similar to the one executing CPU and GPU or they can different as well. CPU and APD are able to share a common address space which allows CPU cores and APDs processing elements to access system memory. A sequence of instructions is received from either CPU or APD computing cores; this sequence of instructions received may contain load or store operation which may perform one or more cache operation. As soon as the sequence of instructions is received, visibility ordering of the date items is determined. These visibility rules help data items to maintain their integrity and execution order when they are accessed by multiple computing kernels of APD or CPU. There are chances that compiler may re-order the sequence of instruction in order to enhance its performance, therefore the values of data items may different when executing on same or different processors as latest values might not have been propagated to system memory. Also the multiple computing units are executed they are modified first on their local memories and after that propagated to common memory. Visibility rules make sure that all the work items read or write the correct values.

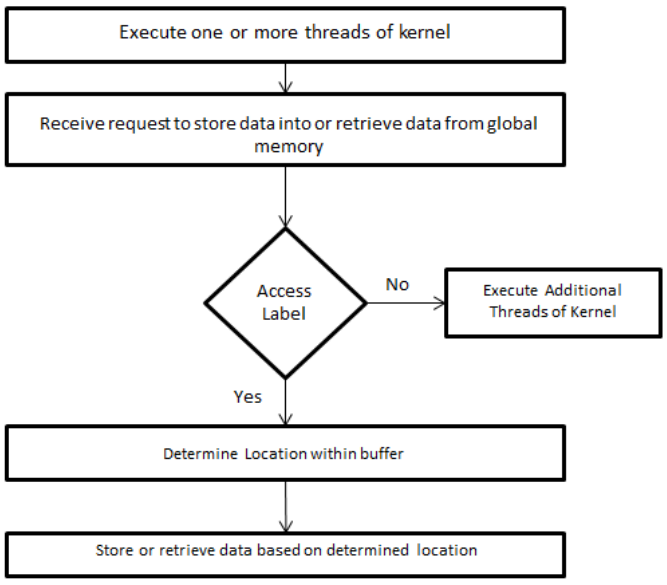
Execute one or more threads of kernel
Receive request to store data into or retrieve data from global memory
Access Label
No
Execute Additional Threads of Kernel
Yes
Determine Location within buffer
Store or retrieve data based on determined location

Figure 3: Flow chart for Pipeline Management Unit

After determining the visibility rules for memory operations, the cache operations which include cache flush and cache validation are performed. The purpose of cache flush operation is to write data items to system memory which are already updated in local or cache memory so that they can be accessed globally. Cache invalidate is performed to invalidate the copy of a data item when that data item is updated in any of the local memory of work items. The visibility rules are described in next section [7].

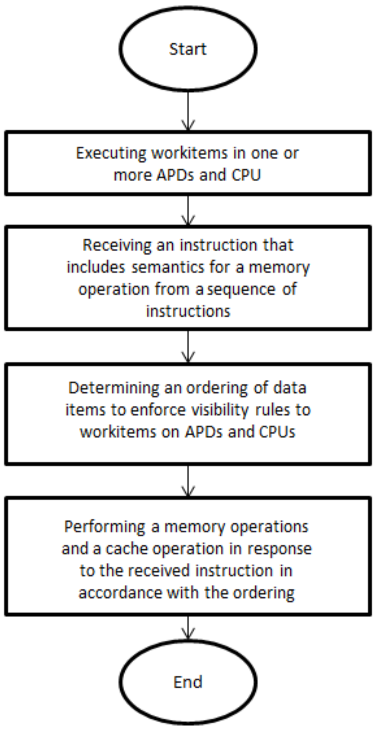
Start
Executing workitems in one or more APDs and CPU
Receiving an instruction that includes semantics for a memory operation from a sequence of instructions
Determining an ordering of data items to enforce visibility rules to workitems on APDs and CPUs
Performing a memory operations and a cache operation in response to the received instruction in accordance with the ordering
End

Figure 4: Sequence of memory operations

d) Visibility Rules:

Before the cache operations are executed, it is necessary to analyze visibility rules for the data items in the sequence of instructions in accordance with a received instruction. Following are the rules on the basis of which a certain data item X or certain data item Y is being flushed or invalidated.

- Sequence before ordering. It is a kind of asymmetric, transitive, pairwise relation, and it enforces partial ordering among those instructions. Let suppose if X is sequenced before Y, then X shall precede Y and vice versa. Let suppose if X is not sequenced before Y and Y is not sequenced before X. X and Y are not sequenced. **"X sb Y"** (X sequenced before Y) which basically means, X>>Y that Y must only be visible after X is visible. It applies transitivity as well such that if X>>Y and Y>>Z then X>>Z.
- Ordering of the sequence of instructions should be according to conditions that include the following [7].

  i. If X and Y are to same address or lie in same memory area, then X>>Y

  ii. If X and Y are to different address or lie in different memory area and there is barrier() or synchronization operation between them, then X>>Y

  iii. If X is load acquire or atomic (un – interrupted) operation and Y is any other operation then, X>>Y

iv. If X is a load operation and Y is any other memory operation such that an address dependency exists with the value returned by X, where the value of the address changes to the value returned by X then, X>>Y.

v. If instruction X refers to a memory operation and Y is a store release, then X>>Y.

- Rules below are specified among multiple work items.

i. Sequential consistency must be obtained by all load acquire and store release operations.

ii. There is single order for all stores per memory location, by any work item.

iii. X->Y implies that a store in X in any work item synchronizes with a load Y, in another work – item.

iv. If X is an operation such as an atomic store from a work item, which provides data to any load Y on another work item, X will be visible to all Z instructions such that Y>>Z and X->Y>>Z.

v. The barrier and sync operation synchronize work – items and act as full A>>B.

e) Determination of Visibility Rules:

In this step the relative visibility ordering of data items accessed are determined. Fig. 5 below, shows the sequence of operation performed to enforce visibility ordering.

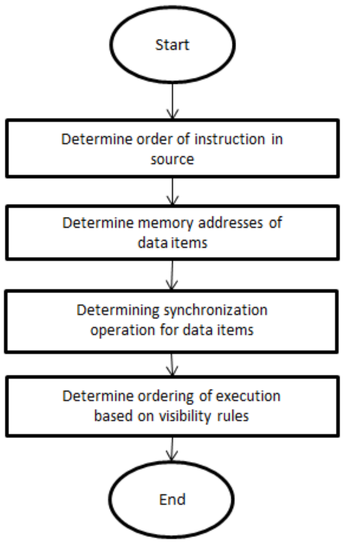
Start
Determine order of instruction in source
Determine memory addresses of data items
Determining synchronization operation for data items
Determine ordering of execution based on visibility rules
End

Figure 5: Sequence of cache operations

f) Sequence of Memory Operations:

This flow chart explicitly explains how the sequencing is carried out on heterogeneous computing system. The Fig. 6 below shows the flow chart diagram.

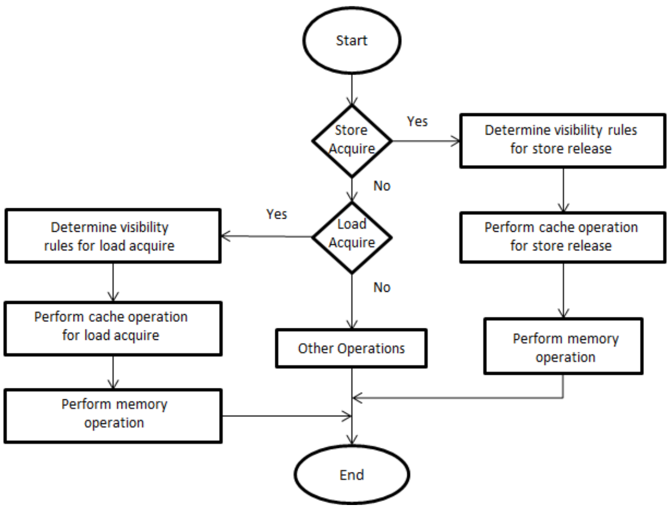
Start
Store Acquire
Yes
Determine visibility rules for store release
No
Yes
Determine visibility rules for load acquire
Load Acquire
Perform cache operation for store release
No
Perform cache operation for load acquire
Other Operations
Perform memory operation
Perform memory operation
End

Figure 6: Sequence of memory operations

First, which kind of memory operation needs to be performed should be determined (whether store release or load acquires). If it is store release or atomic store the semantics for it should be determined which means an instruction Y writing data item Y to a memory and ensuring that, for any data item X such that X sb Y, then X is visible to Y [5]. In case of load acquire the semantics for the instructions would be an instruction Y reading data item Y only after any other data item X such that, X sb Y have already been made visible to Y. After deriving the visibility rules for store or load instruction, we determine which cache operation is to be performed, which can be cache flush or cache invalidate. It may be applied to all the data items or selected ones according to visibility rules. Hence, after cache operation we are able to write the data in system or local memory or read the data from system or local memory.

g) Sequence of Cache Operations:

The cache operations mainly performed are cache invalidate and cache flush operations. This flow of operations may be repeated for each memory operation for store release and load acquire instruction. The Fig. 7 represents the flow of cache operations.

We determine which cache operation to perform either cache flush or invalidate. If both load and store instruction requires one or more cache flush operation such that store Y instruction may require cache flush operation for any data item X, such that X sb Y and X is visible to work items before Y is written and load acquire Y instruction may require cache flush operation for any data item X, such that X sb Y and X is visible to work items before Y is read we perform flush operation. Usually cache flush is used to enforce “sequence before” ordering rules.

In case of cache invalidate operation, store release instruction normally requires cache invalidate operation. It used when store Y instruction writes the new value of data item

Y, it needs to invalidate all cache copies. Load instruction might not need cache invalidate operation.

IV. Technological Improvements:

As CPU reached its limitations in terms of increasing speed and computation power, the paradigms shifted to next generation GPU (Graphical Processing Units) to exploit speed up and enhancement in computation power. Due to dense architecture of GPUs which contain hundreds of computing cores, massive parallelism increases computational power. In the beginning due to limited programming environment GPUs were mainly used for graphic processing, but as the era progressed and due to multi–vendor support technologies GPUs were able to be utilized for general purpose computations. However CPU and GPU memory communication and communication within the cores of GPU remained a great challenge. The technologies discussed above brought solutions to CPU–GPU memory/cache communication. The visibility rules for data items were enforced on memory operation to make them atomic and synchronize with all the threads executed and computing cores which are executed in parallel. The main purpose of these visible rules was to have data integrity when an instruction is being executed on CPU as well as on GPU. The cores on the GPU are not able see CPU system memory; therefore all the accessible data items are required to be transported to GPU global memory with correctness for efficient computation. This is why visibility rules and visibility ordering was established to maintain data integrity.

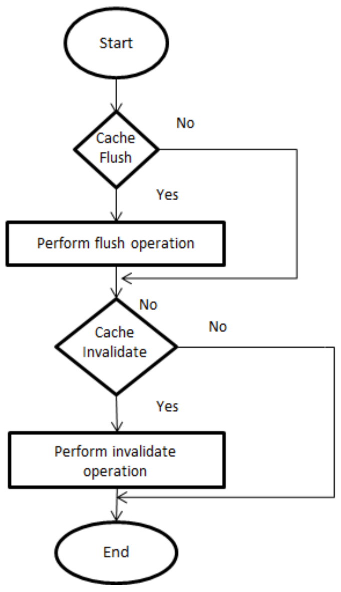

Figure 7: Cache operations

Another problem with the GPU was that, the computing cores with in a GPU does not have access to each other local memory, so if the these cores are sharing some kind of data they have to fetch it from off chip global memory, which could be expensive in terms of speed and latency, and if data is not available threads or computing kernels may result in starvation. This technological issue has been handled by innovative solution of introducing buffers in the global memory as well as in the cache memory of GPU along with buffer management unit [7]. This buffer management unit is able to keep track of starting and ending addresses of the buffers, where to store the data inside the buffers of the global memory, where to retrieve the data from in the buffers of global memory and bringing in with additional information from the global memory and saving it on local caches. This solution provided with less latency, reduction in slowness and starvation free threads execution on GPU. Hence both the technologies had a great impact towards the progress of heterogeneous computing systems. One can expect more better and efficient solutions coming up in next few years.

V. Conclusion:

No doubt, the future of the computing speed lies with GPUs. But as we introduce new technologies there are always some consequences. In case of GPUs the consequence is communication between CPU and GPU and communication between the cores in the GPU. In this paper, we review two technologies; each one of them addresses one problem with its functionality. However, researchers are still working on the issue, and trying to have more effective communication between them to enhance the performance, speed and latency.

References

[1] Chien – Ping Lu (nVIDIA) K3 Moore's Law in the Era of GPU Computing .In Inter national Symposium on VLSI Design Automation and Test (VLSI-DAT) 2010.

[2] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. In PLDI, 2011.

[3] R. Moussalli, R. Halstead, M. Salloum, W. Najjar, V.J. Tsotras, Efficient xml path filtering using gpus, in: ADMS, 2011.

[4] R. Yang, Processing Dependent Tasks on a Heterogeneous GPU Resource Architecture, 2nd IEEE International Conference on Parallel Distributed and Grid Computing (PDGC), 2012.

[5] C. McClanahan. History and Evolution of GPU Architecture. A Paper Survey http://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf

[6] A. V. Bourd, V. Goel Graphics processing unit buffer management, US Patent Application 20130194286 A1.

[7] A.Asaro, K. Normoyle, M. Hummel, N. Rubin, M. Fowler Cache Management for Memory Operations US Patent Application 20130262775 A1.