Chapter-1

Introduction

OpenCL is the first open, royalty-free standard for cross-platform, parallel programming of modern processors found in personal computers, servers, handheld and embedded devices. OpenCL greatly improves speed and responsiveness for a wide spectrum of applications in numerous market categories from gaming and entertainment to scientific and video encoding and decoding software. There are various video coding techniques emerging in industries for very high quality of picture viewing, such as, MPEG7, H.263, H.264, HD etc. These video codec employ different algorithms for representing their formats, many of them uses only the compression and decompression, some of them uses Motion estimation techniques for video compression. But in MCTI (Motion Compensated Temporal Interpolation) we generate new images between adjacent images of a incoming video sequence which increases no. of frames being displayed per second and this increases viewing experience in multiples. This is achieved by comparing adjacent frames of a video sequence both in forward and backward direction and generates motion vectors for generating new intermediate image. For the generation of Motion Vectors huge computation is needed, and this computation is done through intensive computation Power of Graphics Processing Units (GPUs) .The implementation of MCTI algorithm for generate motion vector is done by OpenCL language.

Currently we are developing a compression algorithm based on motion vectors of consequent frames of videos to give very high picture quality. The most important thing is, we are using GPU with CPU to generate motion vectors which is now first of its kind and more over it will go into both Mobile device which are having capabilities of GPU and for PC's and set-top boxes as well. We are using GPUs for parallel processing of the data for faster results. The goal of this thesis is to implement some of the most widely used motion estimation algorithm on OpenCL, analyze common pitfalls and design choices and point out limits of GPU-accelerated motion estimation. To this end, the algorithms have been embedded with the OpenCL engine framework, making them available instantly in the context of existing applications using the OpenCL library.

Many computational problems have gained a significant performance increase by using the highly parallel properties of the GPU. OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. OpenCL was initially developed by Apple Inc., which holds trademark rights in collaboration with technical teams at AMD, Intel, and Nvidia. Apple submitted this initial proposal to the Khronos Group [1]. On June 16, 2008, the Khronos Compute Working Group was formed [2] with representatives from CPU, GPU, embedded-processor, and software companies. This group worked for five months to finish the technical details of the specification for OpenCL 1.0 by November 18, 2008. This technical

specification was reviewed by the Khronos members and approved for public release on December 8, 2008 [3].

The main goals of the project are;

1.) OpenCL as a multi-core programming tool and its inherent performance and portability properties is of interest. On background of code developed within this project, we wish to explore this area.

2.) Experiments with one or several tools used for performance measuring and profiling of OpenCL code. Nvidias performance measuring and profiling tools should be included here.

3.) For the study of performance tools as mentioned above; include one or more from another vendor; Intel and AMD/ATI.

4.) Based on the experiments; suggest ways to tune portions of the OpenCL code for efficient multi-core/GPU execution.

5.) Study how performance is affected when porting programs between different platforms.

6.) Provide estimates for some OpenCL programs as a function of the number of cores/compute units used.

7.) Compare the performance of benchmark programs implemented in OpenCL with comparable implementations in other languages

8.) Study the interplay of current OpenCL implementations and the operating systems they run on with respect to performance.

9.) A focus on debugging tools for OpenCL is of interest.

The thesis is organized in 6 chapters:

The chapter -2 explains Benchmarking and benchmarking procedure in MCTI. In chapter -3, theory of OpenCL is explained. In chapter -4, the proposed work of this thesis is explained. In chapter -5, Timing analysis or Benchmarking of GPU Code is shown in graphical results. Chapter-6 shows the conclusion and discussion about future directions in this project.

All the results have been achieved on Nvidia Quadro NVS 295 & Nvidia GTX 285.

Chapter -2

Benchmarking Procedure in MCTI

Benchmarking [4] is the process of comparing one's business processes and performance metrics to industry bests and/or best practices from other industries. Dimensions typically measured are quality, time and cost. Improvements from learning mean doing things better, faster, and cheaper.

There is no single benchmarking process that has been universally adopted. The wide appeal and acceptance of benchmarking has led to various benchmarking methodologies emerging.

2.1 Benchmarking Procedure

The methodology for benchmarking consists of selecting a subject for benchmarking (here I have selected MCTI Video coding on OpenCL platform), and defining the process and system requirement. System requirement consist of defining hardware, Operating system and software used in the process of benchmarking.

After defining all the system requirements we should identify the algorithm used in the codec/application and the type of data used in the process (integer, floating etc.), choose different algorithms to do the same functionality by selecting the same data type and precision. Then determine the difference in the time required to complete the task performed by different algorithms, also tell the process difference

(i.e. cycles and memory buffer used in computation) , compare them with the pre specified target and set accordingly the future performance of the code. Share all your experience with the experts in the industries and compare your design with their designs, adjust your product goal and future performance. Finally implement the design by the best algorithm you have used so far and if necessary review/recalibrate the application by the same procedure used. By doing all these steps the application will attain optimum performance and will become an industry standard.

2.2 MCTI (Motion Compensated Temporal Interpolation)

It is a technique for frame repetition or linear interpolation for reconstructing skipped frames in temporally sub-sampled video sequence tends to introduce undesirable artifacts. Object motions must be compensated in order to remove these artifacts.[5]

To smooth out object motions, this proposal is a post processing technique; motion compensated temporal interpolation (MCTI), to increase the instantaneous decoder frame rate. In MCTI, block-based exhaustive motion search is used to establish temporal association between two reconstructed frames. Both forward and backward searches are used to account for uncovered and newly covered areas properly. With MCTI, It is shown that one or more frames can be interpolated with acceptable visual quality. After showing the feasibility of MCTI, In MCTI, compensation for the object motions by tracking the objects between adjacent

received frames. Knowing the trajectory of each object, we can place the object at the appropriate location in the interpolated frames. We use block-based motion estimation to establish block wise association between each pair of adjacent received frames. The criterion for block matching is SUM of Absolute Difference (SAD). Both forward and backward motion estimation would be performed to account for the uncovered regions and the newly-covered regions. These regions can be found in only one but not both of the received frames. The collection of all the motion vectors defines a motion field which is used to set up a database for each interpolated frame. The database would then be used to construct the interpolated frames. The feasibility of the MCTI is to verify by simulation using two videoconferencing test sequences.

2.3 Benchmarking Procedure in MCTI

STEPS

1. **Verify your code!**

2. **Measure runtime**: & compare against the best available code

    1) compile other codes correctly (as good as possible)

    2) use same timing method

    3) be fair

    4) Always sanity check: compare to published results etc.

3. **Measure performance**: flops (number floating point operations/second), compare to peak performance

1) needs peak performance, which can be difficult
2) get instruction count statically (cost analysis) or dynamically (tool that counts, or replace operations by counters through macros (input s sequence of characters))
3) Careful: Different algorithms may have different operations count, i.e., best flops is not always best runtime

How to measure runtime?

1) process specific, low resolution, very portable
2) measures wall clock time, higher resolution, somewhat portable
3) Performance counter
4) measures cycles (i.e., also clock time), highest resolution, not portable
5) measure only what you want to measure (may be subtract overhead)
6) proper machine state
7) measure enough repetitions
8) check how reproducible; if not reproducible: fix it

How to present results (in writing)?

Specify machine

1) processor type, (CPU & GPU Used)
2) Frequency of operation of processors
3) relevant caches and their sizes

4) operating system (windows / Linux)

Specify compilation

1) compiler incl. version
2) Debugger Used
3) Tools and software SDK Used

Explain timing method

1) By Plotting different results
2) Results of operation have to be very readable (colors if possible, thick lines, fonts, etc.)
3) Choose proper type of plot: message as visible as possible
4) X-Y plot and different comparison Plots

2.4 Benefits of Benchmarking

- Increase profit margins
- Improve return on IT investments
- Reduces the unknowns
- Identify best practices and worst practices
- Pinpoint redundancies
- Quantifies the risks

2.5 Benchmarking Procedure

1. Select at least a common target platform to share case studies results & practical experience between AST teams & members located at different sites

2. Take another platform for comparison of results / Experience.

   Common platform

   1. PC with Nvidia Graphics Acceleration Card and OS Linux.

   2. SDK (Software Development kit) : Nvidia

   Alternate Platform

   1. PC with ATI GPU

   2. Ion Nvidia CPU-GPU platform.

   3. Mobile Phones with embedded GPUs.

3 EVAL Board form GPU provider

4 other Requirements

   1. OpenCL Candidate release Library UP and running on the platform(e.g Nvidia one)

   2. Gdebugger from Gremedy.

   3. Set of relevant case studies for Benchmarking and experience gathering.

1. To compare development / Optimization time
2. Compare Programming languages/Models: OpenGL Vs OpenCL Vs C/C++(Vs CUDA)
3. OpenGL Vs OpenCL code performance comparison.
4. Automatic C to OpenCL translation (Code portability & Performance)
5. Data Type Precision
6. Data parsing Overhead (i.e. Cycles Penalty, memory buffer accessibility)

5. Produce & Document case studies depending upon our experience.

2.6 Development Flow

1. Select Platform with GPU installed.
2. Obtain Tools and applications examples as source code for compiling & Debugging.
3. Obtain the OpenCL compiler & tools for selected target platform
4. Identify a simple test application in OpenCL to experiment & Learn the way of Benchmark/profile/test an application.(e.g. Monte Carlo Integration etc.)
5. Cascade the work flow of “How To” by properly documenting & sharing experience in open sharing style.

Chapter 3

Basics of OpenCL

OpenCL: open computing language for writing parallel applications for heterogeneous GPUs. OpenCL (Open Computing Language) is a new cross-vendor standard for heterogeneous computing that runs on the CUDA (Compute Unified Device Architecture) architecture. Using OpenCL, developers can use the massive parallel computing power of NVIDIA GPU's to create forceful computing applications. As the OpenCL standard matures and is supported on processors from other vendors, NVIDIA would continue to provide the drivers, tools and training resources for developers need to create GPU accelerated applications.

Advantage of using OpenCL

i. it extends the life cycle of SOC

ii. lesser time is needed to write the software implementation

iii. saving the chip area by implementing the same functionality in Software rather than implementing in hard ware

iv. NRE cost of software in much lesser than equivalent hardware implementation.

v. OpenCL have a very big development community support

vi. OpenCL gives less time to develop and gives more performance.

vii. Algorithms for OpenCL development is being done by Advance System Technologies.

Processor Parallelism

Processor parallelism is enjoyed by OpenCL Programmer as both the processors can be used together, Heterogeneous computing system as per Khronos Group as shown in the figure 3.1 below [6].

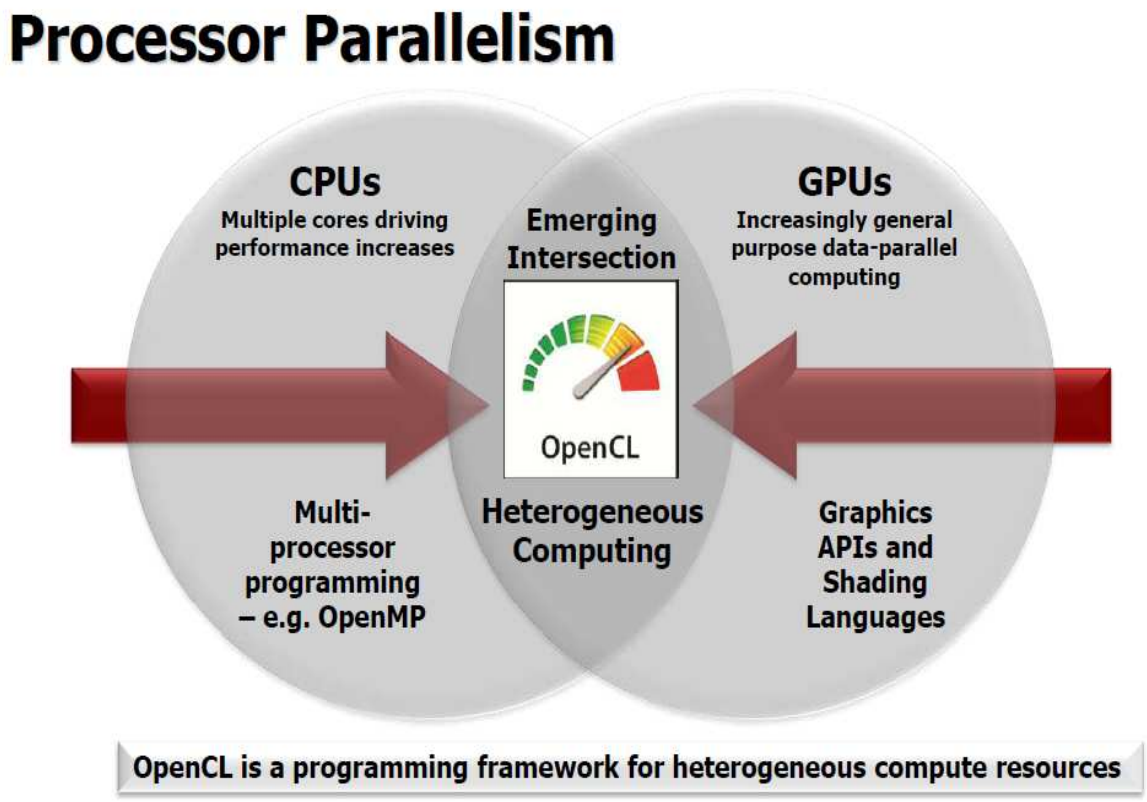

Figure3.1: Processor Parallelism

3.1 History of GPUs

The details in this topic is taken from the official documentation provided by NVIDIA [7 & 8].Graphic processing units have historically been designed to handle the generation and modification of graphical data. This includes rendering of 2D or 3D scenes by means of shader, the name given to software to program the GPUs rendering pipeline. Another task for GPUs were filters, which are small programs that can be applied to an existing stream of images. In 1992, the OpenGL standard was created by Silicon Graphics Inc. (SGI) to allow for cross-platform development of programs which render three dimensional scenes. Microsoft DirectX, with its Direct3D and Direct2D components, is a competing standard and programming API which provides easy facilities to program for graphics cards. The rendering of 3D scenes, while computationally demanding, requires a profoundly different toolset compared to the task normally run by desktop PCs, which is why specialized graphic cards proved to be an efficient way to handle this increasingly important task using dedicated hardware with fast floating point operations, freeing the CPU for other tasks it was more suited for. The speed and detail achieved by programs written for OpenGL, DirectX or the deprecated Glide API surpassed the integrated software renderers of games and professional software, quickly making them obsolete for interactive applications. While OpenGL and DirectX were only aimed at graphics processing, some people tried to harness the computing power of their graphics cards for more general computations, coining the term GPGPU, which is short for general purpose programming using a graphics processing unit. Programs had to be translated to look like graphical problems, meaning that the

program-code had to be written for execution by the programmable shader units, while the data had to be translated into image-data. Because of the lack of native instructions and debug facilities, this was a very limited, cumbersome, and error-prone way to program, but it started the process that would eventually culminate in the 2007 release of NVIDIA's CUDA programming environment. Before the introduction of CUDA, the design philosophy of GPU hardware had already gone from having a fixed-function pipeline for the different stages in three-dimensional image creation to a unified multipurpose processor which could be programmed to handle each of these stages using multiple passes over the input data. This has the advantage that no stage is underutilized while another stage had reached its limit. In 2007, the CUDA toolkit was released by NVIDIA shortly after its new G80 series GPUs, the first one being the GeForce 8800 GTX. With the release of the G80 GPU chip, NVIDIA acknowledged the potential of GPGPU by giving the G80 additional hardware to enable easy programmability. Not only were single-precision floating point operations according to IEEE 754 supported (and eventually double-precision as well), but typical integer operations became possible too. Suddenly, the NVIDIA GPUs transformed into powerful and affordable massively parallel supercomputers with a wide installation base. The compute capabilities of the NVIDIA chips evolved with each new generation, and soon included double-precision floating point operations, efficient scattered memory access, faster and atomic integer operations for 32-bit and 64-bit words and thread synchronization.

In the same timeframe, AMD/ATI released the low-level Close to Metal (CTM) framework to program its GPU device. After the 2007 release of the high-level AMD Stream SDK based on the ATI Brook+ programming language, CTM was renamed to ATI Compute Abstraction Layer. These frameworks provided similar functionality for AMD GPU devices as the CUDA framework. It was not until late 2008 that OpenCL 1.0 was released to the general public. OpenCL (Open Computing Language) is a vendor-agnostic GPGPU framework which was initially developed by Apple Inc. and later handed over to the Khronos group. The Khronos group consists of big vendors like NVIDIA, AMD, Intel, SGI and Sun and is also the home of the OpenGL working group.

Apple had anticipated that it would sell PCs equipped with both NVIDIA and AMD GPUs, which is why they wanted to supply their developers with a portable framework to accelerate algorithms in their applications using a GPU, while AMD started to focus on integrating OpenCL into the AMD Stream SDK and began promoting OpenCL over its own proprietary languages, NVIDIA still offers and develops CUDA alongside OpenCL to the same extent as before.

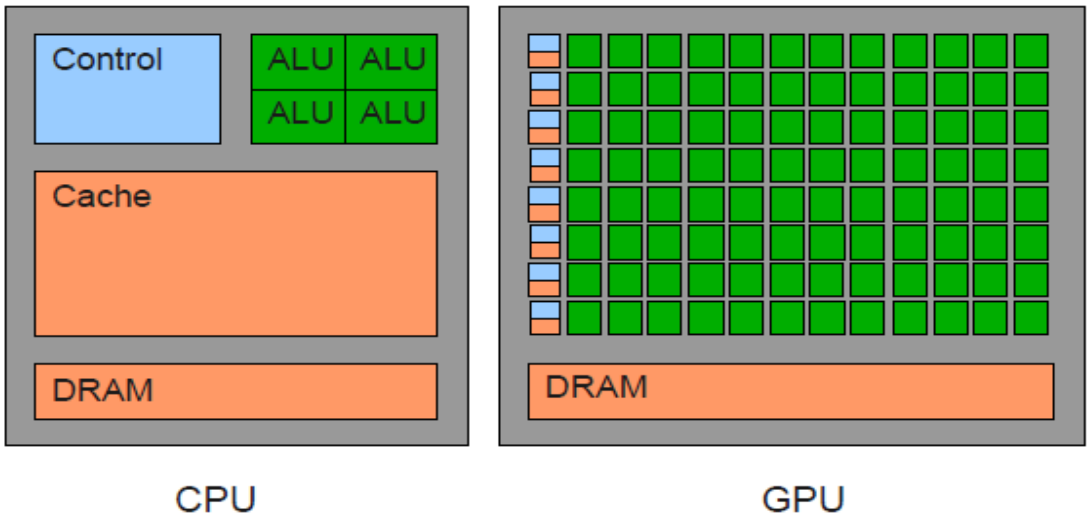

Figure 3.2: Distribution of transistors used in CPU and GPU hardware

Today, the computing power of consumer-grade GPUs has reached about 1.5 teraflops (single-precision) for cards like the GeForce GTX 580 (GF110 chipset), while specialized GPU clusters with four GPU devices offered by NVIDIA perform at a theoretical 5 teraflops. This is similar to the performance achieved by the ASCI Red cluster used by the US Department of Energy starting in 1997, which was the fastest supercomputer in the TOP500 list from June 1997 to June 2000. For comparison, a recent Intel Xeon X5680 CPU can deliver about 150 gigaflops. At the time of writing, the fastest supercomputer on the November 2010 TOP500 list was the Chinese Tianhe-1A system, which was built using 14,336 Intel Xeon X5670 CPUs (each having 6 cores) and 7,168 NVIDIA Tesla M2050 GPUs [TOP10]. It was benchmarked at 2.6 petaflops, a feat which would reportedly have required 50,000 CPUs to achieve the same level of performance without the help of GPUs [8].

The reason for the vast differences in raw floating point operation speed becomes apparent when the distribution of transistors in CPUs and GPUs is observed. Figure 2.1 illustrates how transistors are used in dies of roughly the same size in CPUs and GPUs. The CPU uses a lot of transistors for caching data and control logic to steer program execution and allow for seamless execution of different tasks. All of these transistors are not used for actual computation, only the transistors designated ALU (arithmetic logic unit) are responsible for this task. In a GPU, the area occupied by transistors for arithmetic operations occupies most of the chip, while caches and control logic are kept

3.1.1 Introduction of GPU Programming

Small, fitting the model of executing the same instructions for a large amount of data, a principle termed SIMD (Single Instruction, Multiple Data). Access latency to the global RAM is hidden by having a massive number of threads to schedule if a set of threads is waiting for a memory transaction.

Recently, designers of CPU hardware have reached limits in the acceleration of single die processors imposed by heat dissipation and size. Higher clock speeds can only be achieved with smaller-sized components (i.e. transistors), while increasing the energy consumption per square centimeter means that these processors have increasingly demanding cooling requirements and display wasteful energy consumption in the face of elevated electrical resistance as a result of the temperature. That is why CPU manufactures have begun the move towards multi-

core architectures in the desktop market several years ago, with eight-core CPUs available for the consumer market today.

3.1.2 Graphical Processing Unit (GPU)

The structure of a typical GPU differs from the CPU structure. The main parts of GPU are processor cores, raster operators, texture memory, memory controllers and thread scheduler. Hardware thread scheduler is an important part for parallel computing as it is used to switch threads on hardware level very fast. While one thread is waiting for data from the global memory, other threads are serviced. Threads on GPU are lightweight, which means they contain a small number of instructions, and because of that can work very fast on GPU.

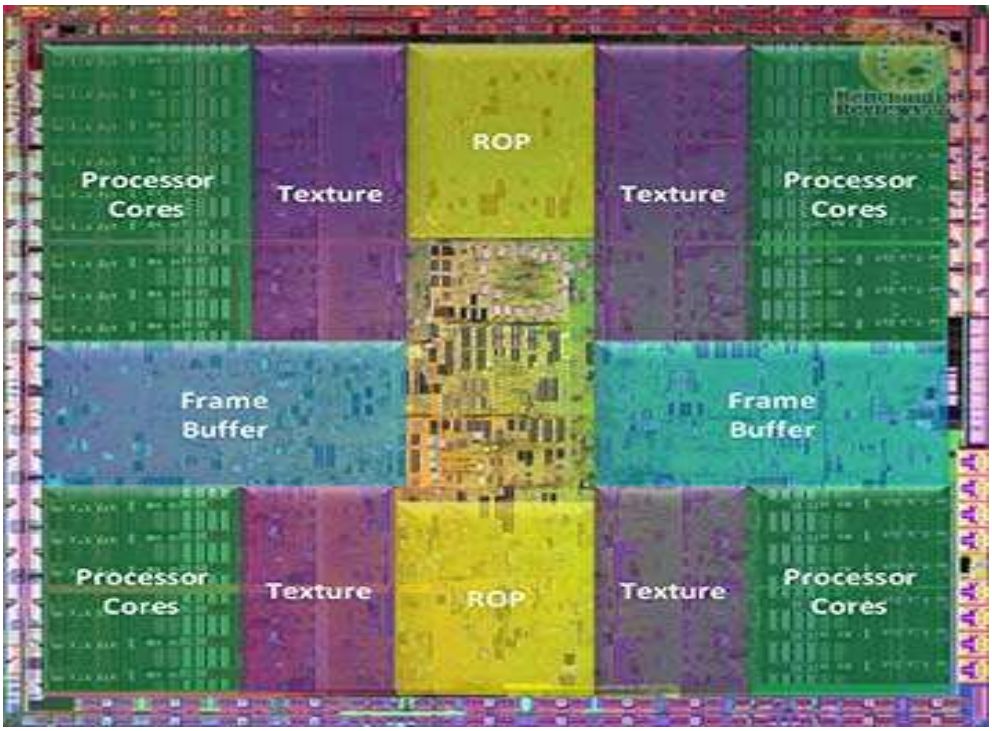

Figure 3.3: Illustrative scheme of GPU

GPU has some streaming multiprocessors (SM), and each of these multiprocessors has some scalar processors (SP) and shared memory. Because most GPU work with single precision float point arithmetic, we can expect that there will be more single precision scalar processors than double precision scalar processors. Every scalar processor has its own small and fast memory called registers. Besides processors, memory, and registers, GPU also has load/store units that supply graphic chip with data from global memory or store the results of computation.

Another way of looking at GPU is to look at it as a multiprocessor with MIMD architecture from programming perspective. GPU can also be looked at from execution point of view as a set of SIMD processors that work on shared memory.

Thread scheduler is one of the most important parts on GPU. It switches threads between scalar processors on hardware layer much faster than on CPU. This is an important advantage of GPU, especially for lightweight kernel objects. For example, a chip from Nvidia G80 has GigaThreadTM Thread Scheduler and it can manage 12.288 threads in real time. Its distributed thread management has two level of architecture; with first level distributing schedules thread blocks to various SMs. Next, every SM warp has a scheduler that distributes warps consisting of 32 threads to SM warps execution units.

3.2. Classification of Processor Architectures

Classification of processor architectures with classification based on two factors - data stream and instruction stream .

1. SISD – sequential computer architecture that has no parallelism, with one data flow and one instruction flow per one processor. This is a typical Von Neumann model. SIMD – computer architecture with *N* processors and *N* data flows instructions but with only one flow of commands working with data .
2. MISD – computer architecture with *N* processors and *N* instruction flows working with one data flow.
3. MIMD – computer architecture with *N* processors working on *M* instruction flows and processes *L* flows of data. One example of this architecture is GPU or distributed system.
4. SPMD – Single Process Multiple Data is a method of making one a task effective by splitting the data into parts and running the same process multiple times with different parts of data. Unlike with SIMD, this method does not require vector processors as it can use general purpose processors instead.

3.3 About OpenCL

3.3.1 Introduction to OpenCL

The Khronos Group has presented Open Compute Language in their specification: "*OpenCL is an open industry standard for programming a heterogeneous collection of CPUs, GPUs and other computing devices organized into a single platform. It is more than a language. OpenCL is a framework for parallel programming and includes language, API, libraries and a runtime system to*

support software development. Using OpenCL, for example, a programmer can write general purpose programs that execute on GPUs without the need to map their algorithms onto a 3D graphics API such as OpenGL or DirectX."

The OpenCL standard was suggested by Apple and created by non-commercial Khronos Group, which has created their own standards, such as OpenGL and OpenAL.

How is written in OpenCL specification about main utilization of OpenCL language:

"The target of OpenCL is expert programmers wanting to write portable yet efficient code. This includes library writers, middleware vendors, and performance oriented application programmers. Therefore OpenCL provides a low-level hardware abstraction plus a framework to support programming and many details of the underlying hardware are exposed." [1, p. 19].

3.3.2 The OpenCL Architecture

The OpenCL Architecture is split into four models, the platform model, execution model, the memory model and finally the programming model [9].

• **Platform model**

The platform model consists of a host with one or more connected OpenCL devices. These devices in turn consist of Compute Units (CUs in short) and each Compute Unit consists of several processing elements (PEs). Using a modern multicore CPU as an example the CPU is the compute unit and each core is a processing element. Execution of applications using OpenCL is achieved through

running a native application on the host which then issues commands to the OpenCL devices via an OpenCL context and command queue.

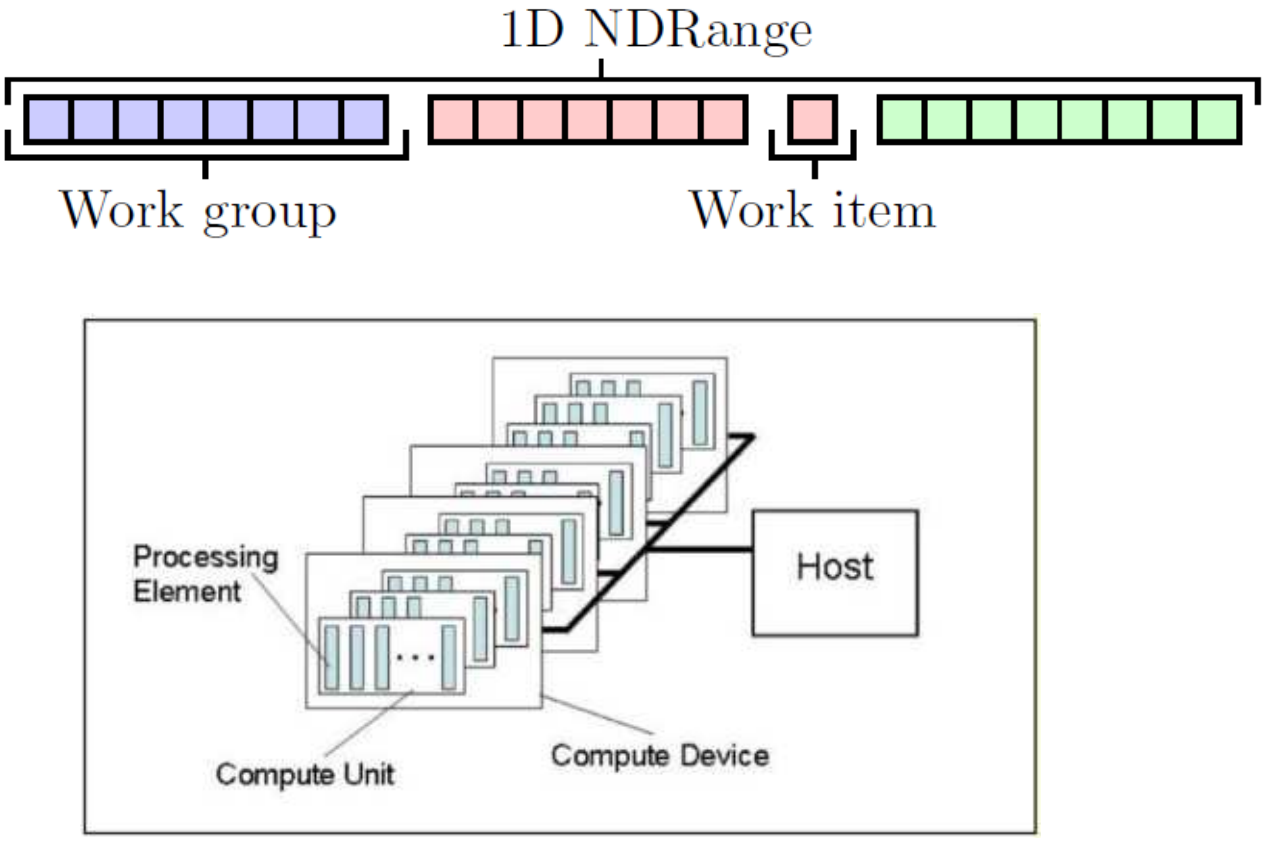

Figure 3.4: Illustration how OpenCL see devices

• **Execution Model**

As the OpenCL platform is designed to utilize multiple additional computational devices the execution of an OpenCL application is split between the host and the compute devices. The part that runs on the host is aptly named host program and the part that runs on the computation devices is called a kernel. To facilitate parallel execution an index space is created when the kernel is submitted to the device for

computation. An instance of the kernel, called work-item is then created for each index and can subsequently by identify by this index. Work-items are then grouped into work-groups. Just as the work-items each work group holds a unique ID derived from the same index space. Apart from its global ID work-items are also given a local ID to identify its location within its work-group. During execution all work-items in a work-group will execute concurrently. The indexing space used to partition problems in OpenCL is called an NDRange. (Khronos OpenCL Working Group, 2008a, p.19).

• **Memory Model**

In OpenCL the memory model contains four independent and distinct memory regions.

a. . Global memory – available to all work-items for read and write

b. . Constant memory – part of the global memory, available to all work items only for read

c. . Local memory – available to work-items in one work-group, it can be mapped into global memory

d. . Private memory – one per every work-item.

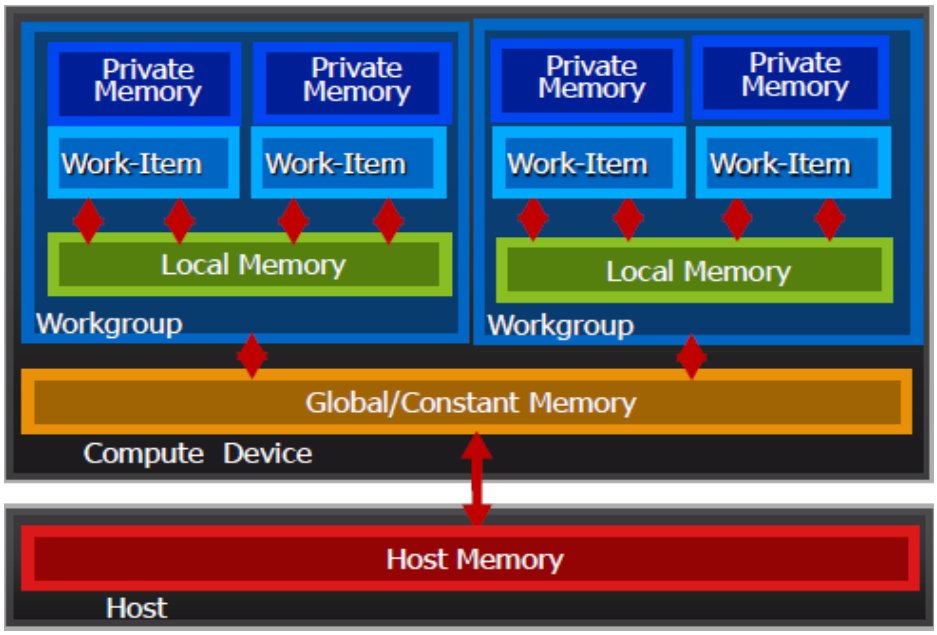

Table 1 describes memory allocation and access from host and kernel to different memory regions. Static allocation is performed during kernel compilation. Dynamic allocation is available from host program. Kernel cannot perform any dynamic allocation.

Table 1: Memory allocation and access to devices memory

	Global	**Constant**	**Local**	**Private**
Host	Dynamic allocation Read / Write access	Dynamic allocation Read / Write access	Dynamic allocation No access	No allocation No access
Kernel	No allocation Read / Write access	Static Allocation Read Only Access	Static Allocation Read / Write Access	Static Allocation Read / Write Access

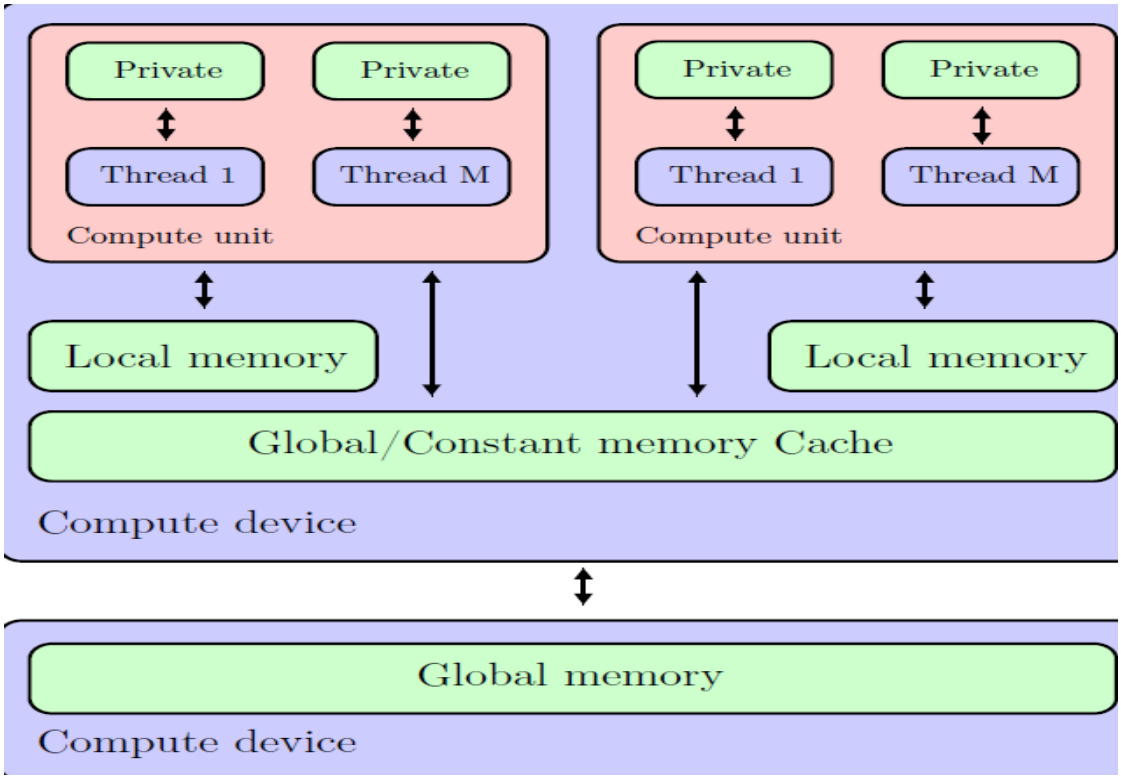

Figure 3.5: [11]The OpenCL memory model. Source: Macresearch.org OpenCL tutorial Podcast, episode 2.

In other words if the kernel instance is reading or writing in other than local memory of his work-group there are required barriers as synchronization points. Exception are private memory and constant memory, that were not changed explicitly from the host.

• Programming Model

OpenCL supports data parallel and task parallel or hybrid of these two programming models. The primary model driving the design of OpenCL is data parallel. Data parallel model defines the way of working with memory. The strict data parallel model defines one-to-one mapping between the work-item and memory object .In explicit model programmer defines total number of work-items

to execute and also defines how the work-items are divided among work-groups. In implicit model programmer defines only total number of work-items and OpenCL take care about division into work-groups.

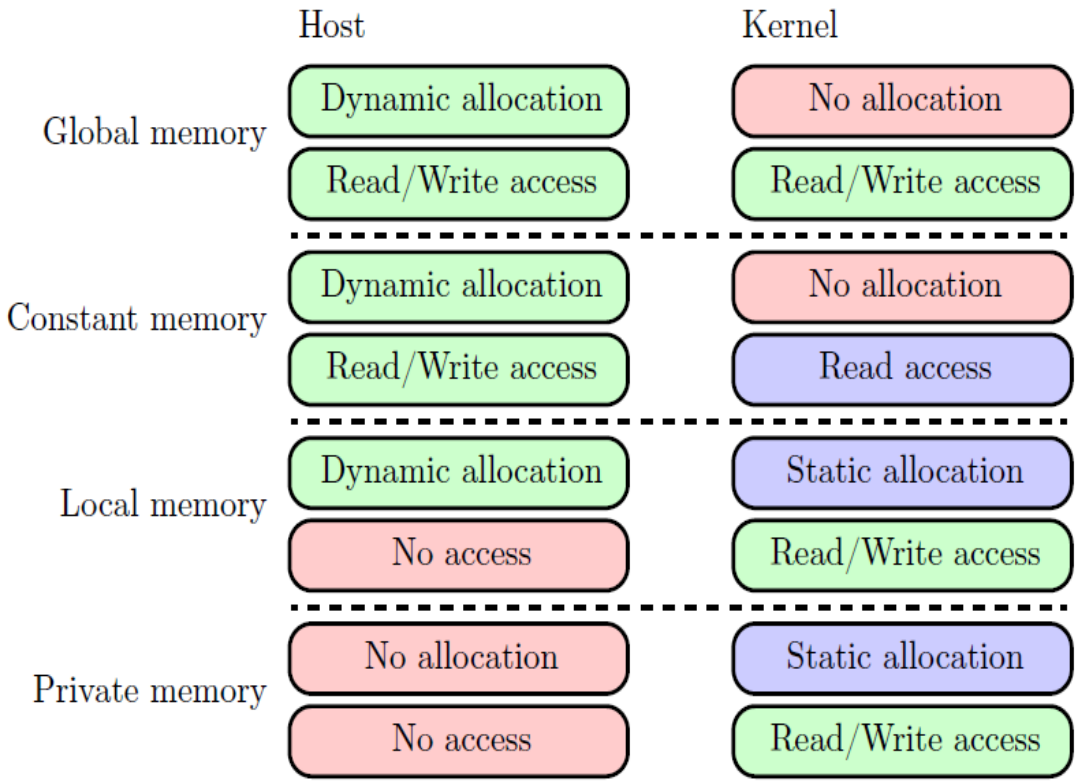

Figure 3.6: The different memory types and access levels for host and kernel(Source:(Khronos OpenCL Working Group, 2008a, p.23))

OpenCL Synch: Queues & Events

Events can be used to synchronize kernel executions between queues

Example: 2 queues with 2 devices

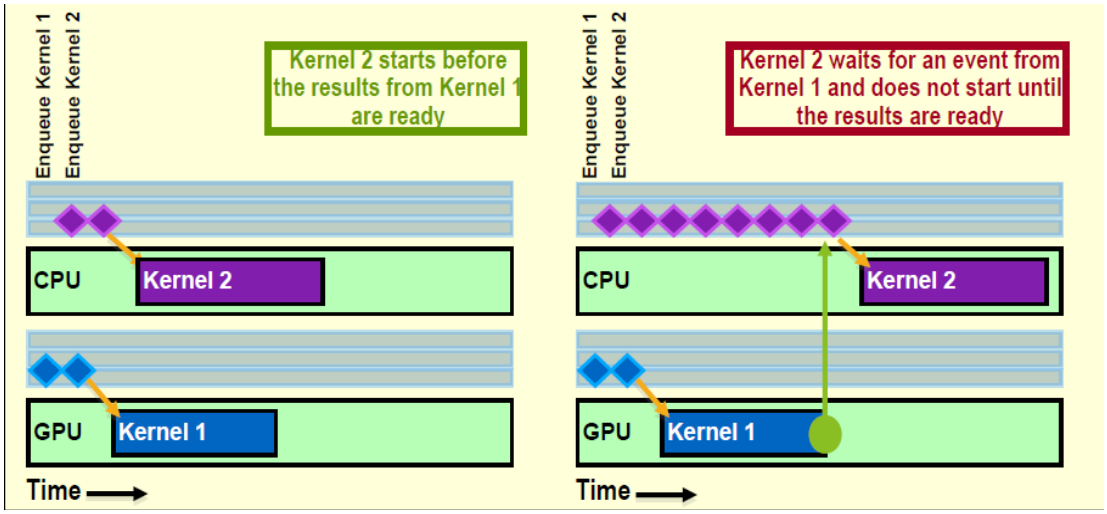

Figure 3.7: OpenCL Synchronization

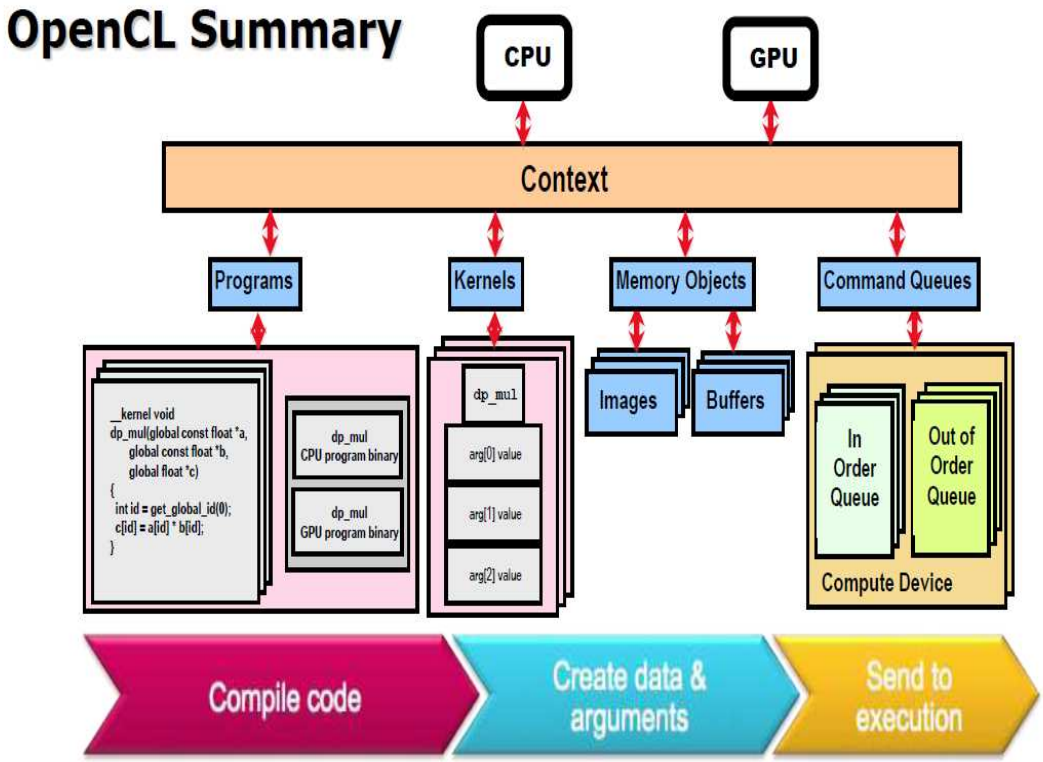

Figure 3.8: OpenCL Summary

3.3.3 Comparison OpenCL to Other Technologies

Writing good and effective program is a little complicated. When we write program, we have to know on what device this program will run. Also we have to know properly how this device and device memory works. At least this is necessary to know in advance to write reasonable program. CPU uses a lot of cache and does not have hardware thread scheduler, so it is better to have small amount of kernels running together on one CPU and have them work with big chunks of memory, which is known as – *relaxed data parallel model*. But GPU is different, there are lot of small Compute Units and cache is very small, so it is better to run program with big count of kernels, which is known as – *strict data parallel model*. If we will not choose right data parallel model computation may run very slowly. CPU will not switch big amount of kernels fast enough, so there may occur slow down of our program, or GPU will not use all compute power and some compute units will unused.

When Apple and Khronos Group made OpenCL as a multi platform standard they had one very big and strong rival - CUDA from Nvidia corp. CUDA provides the full power of almost all Nvidia graphic cards. CUDA and OpenCL are very similar to C language. Program in CUDA is compiled before program run, meanwhile OpenCL need double compilation, first on the host, native C/C++ language and after kernel compilation for specific device. If we want have program with hybrid computation on multi-core CPU and GPU together, in CUDA language it is in some way more complicated to write, because native C/C++ code on host runs in one

thread, so for full computation power we need to run program in threads (using libraries such as Pthread or OpenMP (see below)).

Compared to CUDA, OpenCL can use all compute power of computer without complicated rewriting host program. From Figure 5 we can see the main differences between platforms. Interesting thing is, that CUDA does not support fully standard for float point arithmetic it may cause some unexpected problems during computation.

Feature	OpenCL™	CUDA
Compilation Methods	Online + Offline	Offline Only
Mathematical Precision	Well Defined	*Undefined*
Math Libraries	Defined Standard	Proprietary
CPU Support	OpenCL™ CPU Device	No CPU Support
Native Host Task Support	Task Parallel Compute Model w/ Ability To Enqueue Native Threads	No Native Thread Support
Extension Mechanism	Defined Mechanism	Proprietary
Vendor Support	Industry-Wide Support AMD, Apple, etc.	NVIDIA Only
C Language Support	Yes	Yes

Figure 3.9: Description of differences between OpenCL and CUDA.

There are two most used access to multi-thread programming on CPU, OpenMP and Ptrhead. OpenMP is collection of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in C/C++ and Fortran programs. OpenMP has simplified hierarchy of memory model, there are just shared memory and private memory used [11].

OpenMP is simple to use and simple to understand, there is no need to use a lot changes in C/C++ code. From forum StacOverflow.com: "OpenMP is taskbased, Pthreads is thread based. It means that OpenMP will allocate the same number of threads as number of cores." [14]. The standard, POSIX.1c, defines an API for creating working with threads. Memory model is similar to OpenMP. Managing with threads may be quite difficult, because there is no one row definition about how program should behave, you have to create threads, run them, synchronize them if you use shared memory [15]. Both OpenMP and Pthread, unfortunately to OpenCL, can use already written libraries for CPU. But program that is written in OpenCL can run on other devices than CPUs, also can run on multiple different devices; therefore OpenCL provides much more compute power on other devices. All these technologies require libraries to run, but only OpenCL needs drivers for each device.

3.4 The Anatomy of OpenCL 1.0

The OpenCL 1.0 specification [9] is made up of three main parts: the language specification, platform layer API and runtime API. The language specification describes the syntax and programming interface for writing compute kernels that run on supported accelerators, such as GPUs and multi-core CPUs. The language used is based on a subset of ISO C99. C was chosen as the basis for the first OpenCL compute kernel language due to its prevalence and familiarity in the developer community. To foster consistent results across different platforms, a

well-defined IEEE 754 numerical accuracy is defined for all floating point operations along with a rich set of built-in functions. The platform layer API gives the developer access to routines that query for the number and types of devices in the system. The developer can then select and initialize the necessary compute devices to properly run their work load. It is at this layer that compute contexts and work-queues for job submission and data transfer requests are created. Finally, the runtime API allows the developer to queue up compute kernels for execution and is responsible for managing the compute and memory resources in the OpenCL system. Table 3 is a concise representation of the various parts of OpenCL.

OpenCL C C-based cross-platform programming interface Subset of ISO C99 with language extensions - familiar to developers Well-defined numerical accuracy - IEEE 754 rounding behavior with defined maximum error Online or offline compilation and build of compute kernel executables Includes a rich set of built-in functions
OpenCL API A hardware abstraction layer over diverse computational resources Query, select and initialize compute devices Create compute contexts and work-queues
OpenCL Runtime Execute compute kernels Manage scheduling, compute, and memory resources

Table 3: Main parts of OpenCL

OpenCL C

OpenCL defines OpenCL C, which is a variant of the familiar C99 language optimized for GPU programming. It incorporates changes necessary to adapt the C programming language for use with GPUs and to support parallel processing.

OpenCL C includes comprehensive support for vector types to streamline data flow and increase efficiency.

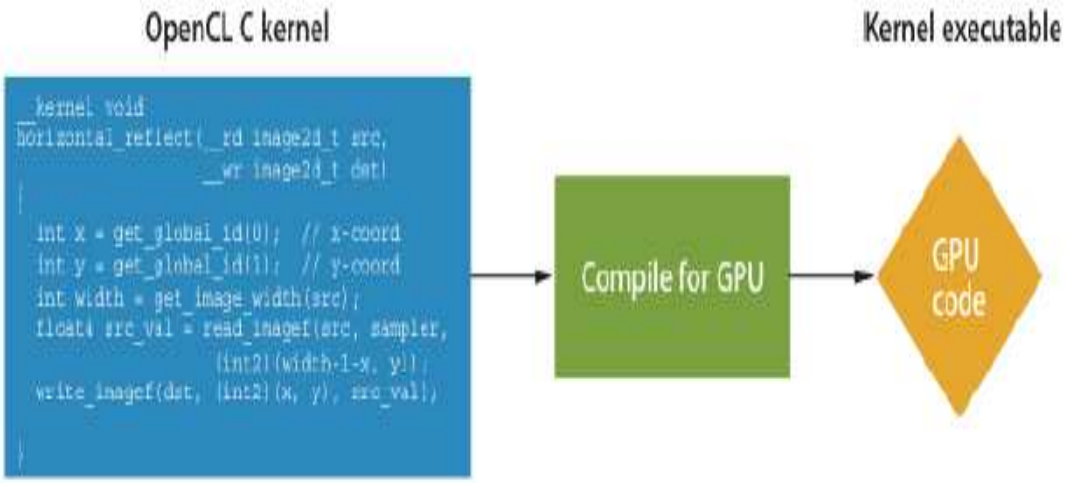

Figure 3.10: Kernel executing as defined by OpenCL 1.0

OpenCL API

The OpenCL API provides functions that allow an application to manage parallel computing tasks. It enumerates the OpenCL-capable hardware in a system, sets up the sharing of data structures between the application and OpenCL, controls the compilation and submission of kernels to the GPU, and has a rich set of functions that manage queuing and synchronization.

OpenCL Runtime

The OpenCL runtime executes tasks submitted by the application via the OpenCL API. The runtime efficiently transfers data between main memory and the dedicated

VRAM used by the GPU, and directs execution of the kernels on the GPU hardware. During execution, the OpenCL runtime manages the in-order or out-of-order dependencies between the kernels, and utilizes the GPU's processing elements in the most efficient manner.

3.3.5 Limitations in the OpenCL C Language

When writing kernels there are some important limitations on what is allowed in the code. The limitations can be found on the Khronos7 web page. The list below is a collection of some of the limitations found on the web page[10]:

The use of pointers is somewhat restricted. The following rules apply:

1. Arguments to __kernel functions declared in a program that are pointers must be declared

with the __global, __constant or __local qualifier.

2. A pointer declared with the __constant, __local, or __global qualifier can only be assigned to a pointer declared with the __constant, __local, or __global qualifier respectively.

3. Pointers to functions are not allowed.

4. Arguments to __kernel functions in a program cannot be declared as a pointer to a pointer(s). Variables inside a function or arguments to non __kernel functions in a program can be declared as a pointer to a pointer(s).

5. Variable length arrays and structures with flexible (or unsized) arrays are not supported.

6. The C99 standard headers assert.h, ctype.h, complex.h, errno.h, fenv.h, float.h, inttypes.h, limits.h, locale.h, setjmp.h, signal.h, stdarg.h, stdio.h, stdlib.h, string.h, tgmath.h, time.h, wchar.h, and wctype.h are not available and cannot be included by a program.

7.The extern, static, auto, and register storage-class specifiers are not supported.

8. Predefined identifiers are not supported.

9. The function using the __kernel qualifier can only have return type void in the source code.

3.4 OpenCL Programming [12]

3.4 .1 Basics

Making the program in OpenCL we should go through some steps. First of all we need to select a platform; this is shown in Figure below.

```
cl_uint numPlatforms;
cl_platform_id platform = NULL;
cl_platform_id * platforms;
clGetPlatformIDs(0, NULL,
&numPlatforms);
platforms = (cl_platform_id*) malloc (
```

```
sizeof(cl_platform_id) * numPlatforms);
clGetPlatformIDs(numPlatforms, platforms,
NULL);
```

Getting platforms

After choosing the platform we can gain access to devices on that platform. In Figure we can select on what device our program will run by creating context.

```
cl_device_type dType = CL_DEVICE_TYPE_DEFAULT; //
setting device
cl_context context;
cl_context_properties cps[3] =
{CL_CONTEXT_PLATFORM,(cl_context_properties)platform,
0};
context =
clCreateContextFromType(cps,dType,NULL,NULL,NULL);
```

Creating context

Each context has to have at least one command queue. We can have multiple command queues with different properties, see Figure below.

```
cl_command_queue commandQueue;
cl_device_id * devices;
size_t deviceListSize;
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0,
NULL, &deviceListSize);
devices = (cl_device_id *) malloc (deviceListSize);
clGetContextInfo(context, CL_CONTEXT_DEVICES,
deviceListSize, devices, NULL);
commandQueue = clCreateCommandQueue(context,
devices[0], 0, &status);
```

Creating command queue

Kernels have to be compiled and linked. Next step it will be memory allocation on the host, setting the kernels arguments and copying data on the host, see Figure below.

```
const char * source;
cl_program = program;
cl_kernel kernel;
cl_mem buffer;
size_t sourceSize = strlen(source);
program =
```

```
clCreateProgramWithSource(context,1,&source,&sourceSize,NULL);
clBuildProgram(program, 1, devices, NULL, NULL, NULL);
kernel = clCreateKernel(program, "Kernel_Name", 0);
buffer = clCreateBuffer(context, CL_MEM_READ_WRITE,
1024*sizeof(cl_float), NULL, NULL);
clSetKernelArg(kernel, 0, sizeof(cl_mem), &buffer);
```

Building kernels and setting arguments

Now we can run kernels with their dimensions set. Wait for finishing the kernel jobs and read data back from device memory, see Figure below

```
unsigned int global_work_size[1] = {1024};
unsigned int local_work_size[1] = {256};
unsigned int work_dim = 3;
status =
clEnqueueNDRangeKernel(commandQueue,
kernel, work_dim, NULL,
global_work_size,
local_work_size, 0, NULL, NULL);
clFinish(commandQueue);
```

Running kernels

Number of work groups in dimension *i* is computed by

$$\textbf{Work Groups} = \frac{\textbf{Global work size[i]}}{\textbf{Local work size[i]}}$$

Count of total running kernels is multiply of each global work sizes from each dimension.

Kernels are independent small programs running on PEs, using C99 syntax but with some restrictions. Figure 11 shows an example of an simple kernel. Kernel uses one global address space. In kernel we can identify kernel identification number in global and local work space and work dimension by inherited functions into OpenCL API. For more information see OpenCL specification

```
kernel void Kernel_Name(__global float * A)
{
__private gid0 = get_global_id(0);
__private lid = get_local_id(0);
A[gid] = lid;
}
```

Example of simple kernel

Chapter-4

Timing Analysis of MCTI Codec

There are several motion estimation algorithms available each having its own advantage and disadvantage. The Motion estimation algorithm based on block matching is used in the present work, where the whole Image is divided into grid of 8x8 pixels blocks. Each block of 8x8 pixels is selected and matched to corresponding nearby blocks using SAD algorithm and if the best SAD is found out the corresponding motion vector is stored in a buffer array. In this way the whole image covering each block of the grid is traversed and motion vector of each image is filled up in the buffer array which is stored for further calculation. Motion Vectors are represented as a dimensional structure covering both(X and Y) directions of motion of object in the images

4.1 Algorithms used in MCTI

The methods for finding motion vectors can be categorized into pixel based methods ("direct") and feature based methods ("indirect"). In pixel based method, a sequence of images is applied as an input sequence to the algorithm first and then each bmp color image is transformed to the YUV image and the Y component of the image is taken. Whole image is divided into blocks of 8x8 pixels and best motion vector is calculated on the basis of SAD algorithm.

The motion estimation algorithm is used for generation of Motion vectors (MV) for both x and y directions for the motion direction of object in that direction .each of the motion vectors is stored per frame and shared with the next image for calculation of the upcoming consequent frame .Motion estimation of frames is done the basic of block matching algorithm. Each block of 8x8 pixels is matched with several 8x8 blocks along it on temporal (within different frames) and spatial (within same frame) ways. This block matching generates motion Vectors which is used to make images that shows motion.

4.2 Sum of absolute differences

Sum of absolute differences (SAD) is used for measuring the similarity between image blocks. It works by taking the absolute difference between each pixel in the original block and the corresponding pixel in the block being used for comparison.

The sum of absolute differences may be used for a variety of purposes, such as object recognition and motion estimation for video compression.

4.3 SAD Calculation

SAD Calculation is done with every 8x8 pixel block neighboring to it in both temporal and spatial way. Each SAD value is stored in an array of size of the number of neighbors of it and a SAD Penalty Is added according to the position of the neighboring 8x8 pixel block.

```
GetCandidateMV  will give the candidates of MotionVector for all
blocks
IMAGE Ref_Image
IMAGE Target_image
Int SAD=0
for
        get RefPixel  Linealry from Ref_image
        get Teget_pixel Bilinearly from Target Image
        // Calculate SAD
        SAD =  SAD + ∑ ( Ref_Pixel_i  -  Target_Pixel_i )
                                where i = 1 to  64 for all the pixels of 8x8
block
End

//Add Penalty to sad values
SAD = SAD+ SAD_Penalty
//sort and get Best_SAD & Best_MV
If   SAD < Best_SAD
                Best_SAD = SAD
                Best_MV =  CandidateMV
Endif
//Store in Backward and forward MV Array
```

Pseudo code for calculation of SAD

4.4 Calculation of Reference Pixel and Target Pixel

Reference Pixel calculation in done by simply giving the coordinates of the pixel and getting back the value of that pixel it uses Manhattan distance algorithm for getting the pixel value of nearest pixel , that it uses linear method.

Target pixel calculation is done by giving both the coordinates and the movement value in that direction by using Bilinear Interpolation algorithm.

4.5 Debugging and checking the motion

A Bitmap (bmp) image is made out of the Motion Vectors in 4:2:2 formats where YUV component of the image is manipulated to generate new image for debugging and checking the motion on the sequence of images.

Y component is kept as it is and the motion vectors are calculated.

After calculating Motion Vectors new image is generated by adding motion component of x direction is added to the U component of Image and motion component of y direction is added to the V component of Image. This will generate a new image showing the motion of consecutive frames.

4.6 Best SAD and Motion Vector along the path computation

By sorting we take out the best SAD and the respective motion vectors relating to it. And this Best Motion Vector is stored in an array for generating new frame and calculating new Motion vectors for the next incoming frame and this generation of motion vectors is continued for each frame of the video.

By Using OpenCL- API we have replaced some functions which are made by C language as there are inbuilt by functions for the same.

4.7 Performance analysis of different phases of Motion Estimation on GPU

1. Using GPROF and GDB tools to optimize the code.

Motion Estimation for GPU code for the first time Ported to OpenCL and the result came out 23 seconds per frame for both forward and backward calculation of motion vectors. This was not the full optimized code and for the testing purpose it was successfully ported on OpenCL and it ran perfectly on GPU.

The output results were exactly same as that of simple sequential code written on C language running on CPU. This was a brilliant break through for us as we have achieved first parallelization on GPU successfully.

2. Changing Algorithm and dumping dead codes for the CPU.

Some Changes has been done in the code and the dead codes had been removed to make the ME4GPU code run at 15 seconds per frame. As we have got our code running on GPU next phase of action was to increase the order of parallelism, this code reflected the increment of threads verses timing successfully.

3. Eliminating large array structures into small ones to break most of the time consuming functions,

Here in this version of code data transfer to GPU has been minimized and the constraints have been tightened to achieve max performance. Results reflected at most 50% of time saving to do the same job, and got result 7 second per frame. This was obvious as we have expected because of data overhead reduction.

4. Changing GPU Kernel Code for making Code Even Faster by using inbuilt APIs of OpenCL

Inbuilt API functions take less time to execute the same job, Khronos group have made many functions for Imaging and Graphics, we have implemented the code using inbuilt APIs, which reduces the timing by more than 50%, and this made a path for us to achieve the target timings. After these changes we got result 3 second per frame.

5. Written new algorithm of Chess Board Pattern generation for X and Y coordinates for the image to achieve timings in milliseconds.

Since the timing for launching the kernel was very critical and giving overhead but we written new chess board pattern algorithm to achieve the peak performance of the code, this was a break through implementation in the algorithm as the main execution loop was reduced by several **%** as this saved a lot of time.

Previous main loop execution = **(Height * Width)** times.

New chess board algorithm Loop execution = **(Height+Width-1)** times.

Overall %age reduction in the no. of times execution of the loop=

$$\frac{(\text{Height} * \text{Width}) - (\text{Height} + \text{Width} - 1)}{(\text{Height} * \text{Width})} * 100 = \%\text{ age reduction}$$

This formula gives %age reduction (More than 80 % percent in all cases) in the Main loop which saves a lot of time and we have further moved towards the target timing. After written new chess board pattern algorithm we got the result of 148 milliseconds per frame.

6. Increased thread Parallelization and made the code more complex to achieve the best performance.

Finally we have made our code even complex and increased no. of threads, this gives us very good performance and this was the final timings we have obtained and this time is closest to the target time. And finally we got the result of 108 milliseconds per frame.

Chapter-5

Timing & Result

5.1 Benchmarking of GPU Code

For Benchmarking we used Nvidia visual profiler, and this is done for the last 2 optimized codes.

The specification of CPU and GPU are as below

5.1.1 CPU Info

We implemented our code on an Intel Xeon Processor having 8 CPU cores

Showing 0^{th} core of 8 processors

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 26
model name      : Intel(R) Xeon(R) CPU          E5504  @
2.00GHz
stepping        : 5
cpu MHz         : 1995.005
cache size      : 4096 KB
physical id     : 1
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 16
initial apicid  : 16
fpu             : yes
fpu_exception   : yes
cpuid level     : 11
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep
mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse
sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good pni monitor ds_cpl vmx est
tm2 ssse3 cx16 xtpr dca sse4_1 sse4_2 popcnt lahf_lm
bogomips        : 3993.66
```

```
clflush size    : 64
cache_alignment : 64
address sizes   : 40 bits physical, 48 bits virtual
power management:
```

CPU Specification

5.1.2 GPU Info

It is an Nvidia Graphics Processor (Quadro NVS 295)

```
GPU and OpenCL Info:

OpenCL SW Info:

 CL_PLATFORM_NAME:      NVIDIA CUDA
 CL_PLATFORM_VERSION:   OpenCL 1.0 CUDA 3.2.1
 OpenCL SDK Revision:   5985201


OpenCL Device Info:

 1 devices found supporting OpenCL:

 ---------------------------------
 Device Quadro NVS 295
 ---------------------------------
  CL_DEVICE_NAME:                  Quadro NVS 295
  CL_DEVICE_VENDOR:                NVIDIA
Corporation
  CL_DRIVER_VERSION:               260.19.29
  CL_DEVICE_VERSION:               OpenCL 1.0 CUDA
  CL_DEVICE_TYPE:
CL_DEVICE_TYPE_GPU
  CL_DEVICE_MAX_COMPUTE_UNITS:         1
  CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS:  3
  CL_DEVICE_MAX_WORK_ITEM_SIZES:       512 / 512
/ 64
  CL_DEVICE_MAX_WORK_GROUP_SIZE:       512
  CL_DEVICE_MAX_CLOCK_FREQUENCY:       1300
MHz
  CL_DEVICE_ADDRESS_BITS:              32
  CL_DEVICE_MAX_MEM_ALLOC_SIZE:        128
MByte
  CL_DEVICE_GLOBAL_MEM_SIZE:           255 MByte
  CL_DEVICE_ERROR_CORRECTION_SUPPORT:  no
  CL_DEVICE_LOCAL_MEM_TYPE:            local
```

```
 CL_DEVICE_LOCAL_MEM_SIZE:          16 KByte
 CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE:   64
KByte
 CL_DEVICE_QUEUE_PROPERTIES:
CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE
 CL_DEVICE_QUEUE_PROPERTIES:
CL_QUEUE_PROFILING_ENABLE
 CL_DEVICE_IMAGE_SUPPORT:           1
 CL_DEVICE_MAX_READ_IMAGE_ARGS:       128
 CL_DEVICE_MAX_WRITE_IMAGE_ARGS:      8
 CL_DEVICE_SINGLE_FP_CONFIG:          INF-
quietNaNs round-to-nearest round-to-zero round-to-inf fma

 CL_DEVICE_IMAGE <dim>
2D_MAX_WIDTH    4096
                        2D_MAX_HEIGHT    32768
                        3D_MAX_WIDTH     2048
                        3D_MAX_HEIGHT    2048
                        3D_MAX_DEPTH     2048

 CL_DEVICE_EXTENSIONS:
cl_khr_byte_addressable_store
                        cl_khr_icd
                        cl_khr_gl_sharing
                        cl_nv_compiler_options
                        cl_nv_device_attribute_query
                        cl_nv_pragma_unroll
                        cl_khr_global_int32_base_atomics

cl_khr_global_int32_extended_atomics


 CL_DEVICE_COMPUTE_CAPABILITY_NV:     1.1
 NUMBER OF MULTIPROCESSORS:          1
 NUMBER OF CUDA CORES:              8
 CL_DEVICE_REGISTERS_PER_BLOCK_NV:    8192
 CL_DEVICE_WARP_SIZE_NV:            32
 CL_DEVICE_GPU_OVERLAP_NV:          CL_FALSE
 CL_DEVICE_KERNEL_EXEC_TIMEOUT_NV:
CL_TRUE
 CL_DEVICE_INTEGRATED_MEMORY_NV:
CL_FALSE
 CL_DEVICE_PREFERRED_VECTOR_WIDTH_<t>
CHAR 1, SHORT 1, INT 1, LONG 1, FLOAT 1, DOUBLE
0
```

oclDeviceQuery, Platform Name = NVIDIA CUDA, Platform Version = OpenCL 1.0 CUDA 3.2.1, SDK Revision = 5985201, NumDevs = 1, Device = Quadro NVS 295

System Info:

Local Time/Date = 15:05:00, 06/16/2011
CPU Name: Intel(R) Xeon(R) CPU E5504 @ 2.00GHz
# of CPU processors: 8
Linux version 2.6.26-2-amd64 (Debian 2.6.26-22) (dannf@debian.org) (gcc version 4.1.3 20080704 (prerelease) (Debian 4.1.2-25)) #1 SMP Tue Mar 9 17:12:23 UTC 2010

GPU ,OS & OpenCL Specifications

5.2. Hardware and Software used for the Benchmark

The GPU and CPU hardware used in the tests are shown in Table 1 and Table 2 below respectively.

Table 1 The GPU hardware used in all tests.

Graphics board	Nvidia Quadro NVS 295
GPU	NVS 295
CL_DEVICE_VERSION	OpenCL 1.0 CUDA
Memory	1.5 GB GDDR3
Memory Bandwidth	76.8 GB/s (16 PCI lanes)
Number of SMs	24
Number of SPs	192
Clock frequency	1300 MHz
On-chip memory per SM	16 KB
Instructions	FMAD

Table 2 The CPU hardware used in all tests.

CPU	Intel(R) Xeon(R) CPU E5504 @ 2.00GHz
Memory	4GB (2 x 2 GB DIMM) 800Mhz DDR2
Front Side Bus	1333 MT/s
Number of cores	1 (The Second Core is turned off in all tests)
Clock frequency	2.00 GHz
Cache memories	4096 KB L2, 32 KB L1 instruction, 32 KB L1 data
Instructions	X86 SSE4.1

For all GPU benchmarks, **Linux version 2.6.26-2-amd64 (Debian 2.6.26-22)** 64bit and Linux Redhat was used. For all the CPU benchmarks ,For the CPU . **Linux version 2.6.26-2-amd64 (Debian 2.6.26-22)** 64 bits was used. All GPU benchmarks were compiled for compute capability 1.3, using Nvidia drivers version 191.66. All CPU benchmarks were run with only one core running to make interpretation of the results easier as the compilers' and OS's degree of optimizing the code for two cores then becomes irrelevant. All Linux CPU benchmarks were compiled with GCC using the flags 'xc', 'ansi', 'lm' and 'O3', allowing the compiler to add SSE instructions where it deemed possible. The Windows benchmarks where compiled with 'O3'. GFLOPS and GB/s are computed with base 10 throughout the benchmarks, not 1024.

5.3 Benchmark Timing definitions

These parameters have been used when measuring performance of the different benchmarks.

Kernel time refers to the time of running the GPU kernel only, not including any transfer between host and device.

Total GPU time refers to the time of running the GPU kernel only and any transfer between host and device.

Kernel Speedup is how much faster the GPU executes a kernel compared to the CPU.

Total GPU speedup is how much faster the GPU executes a kernel compared to the CPU including any transfers between host and device.

Peak performance is either the maximum GFLOPS or maximum throughput achieved for what can be considered radar-relevant data sizes. GFLOPS is a metric for how many billion floating point operations per second are being performed. The way performance or throughput is calculated is detailed in each respective benchmark. As the aim with performance is to see how hard the GPU is working, the transfer times between host and device are not included; only kernel time is relevant.

5.4 Graphical Results

5.4.1 Benchmarking for optimized code running max of 12*67 threads per execution of the GPU kernel

Case 1:

Specification: code running on GPU having total time for execution of GPU Kernel 145 ms by profiling

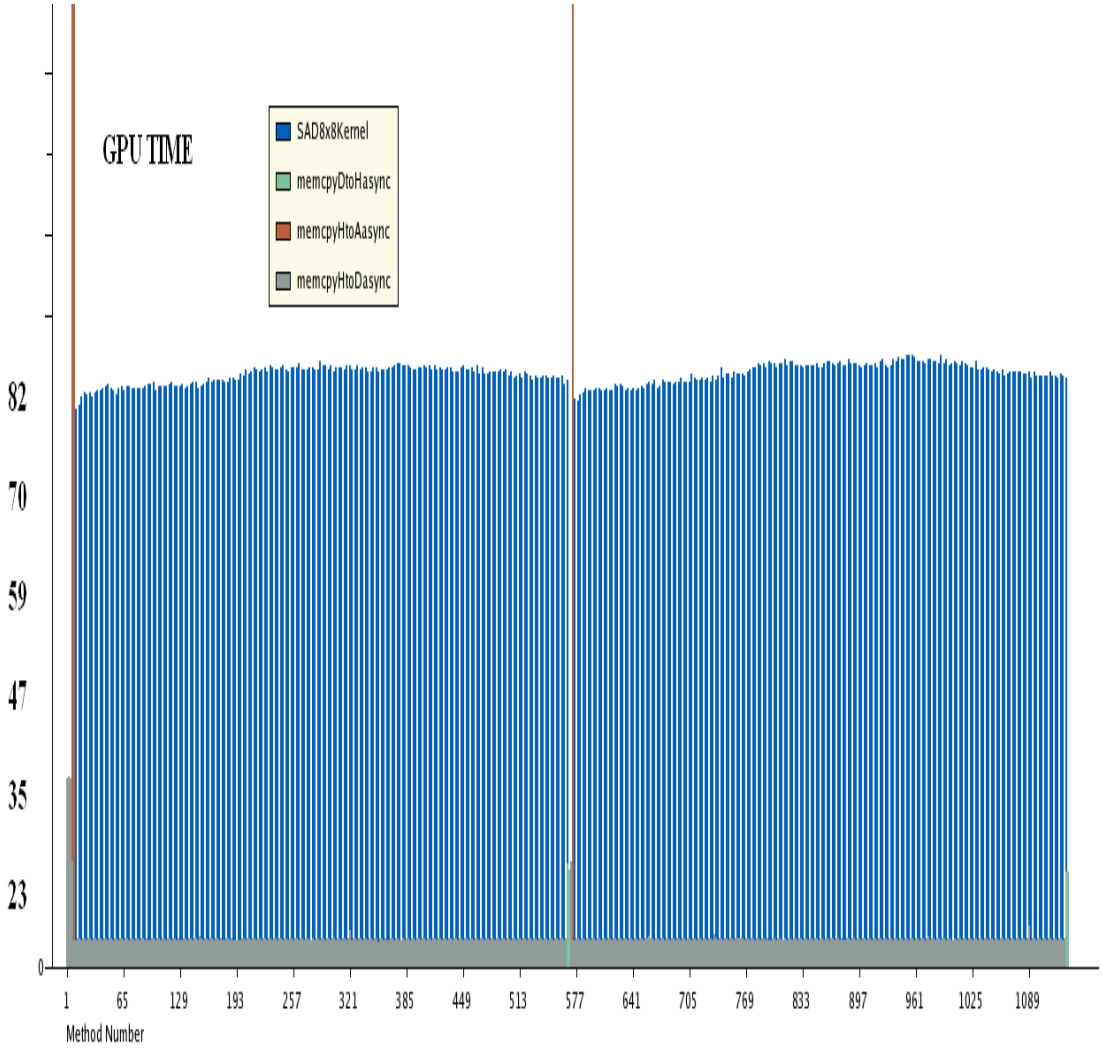

Figure 5.1: Timings for Kernel Execution from Nvidia Visual Profiler

Here maximum kernel launch time for kernel execution is min 82 uc and Maximum 88 microseconds

Table: Kernel Execution Timing

	GPU Time	CPU Time
Total Kernel Execution Time	31902.66 µs	44838 µs
Total kernel Time	35565.15µs	48413µs
% total time	89.70%	92.61%

And drawing the curve by the data obtained from Nvidia Visual profiler

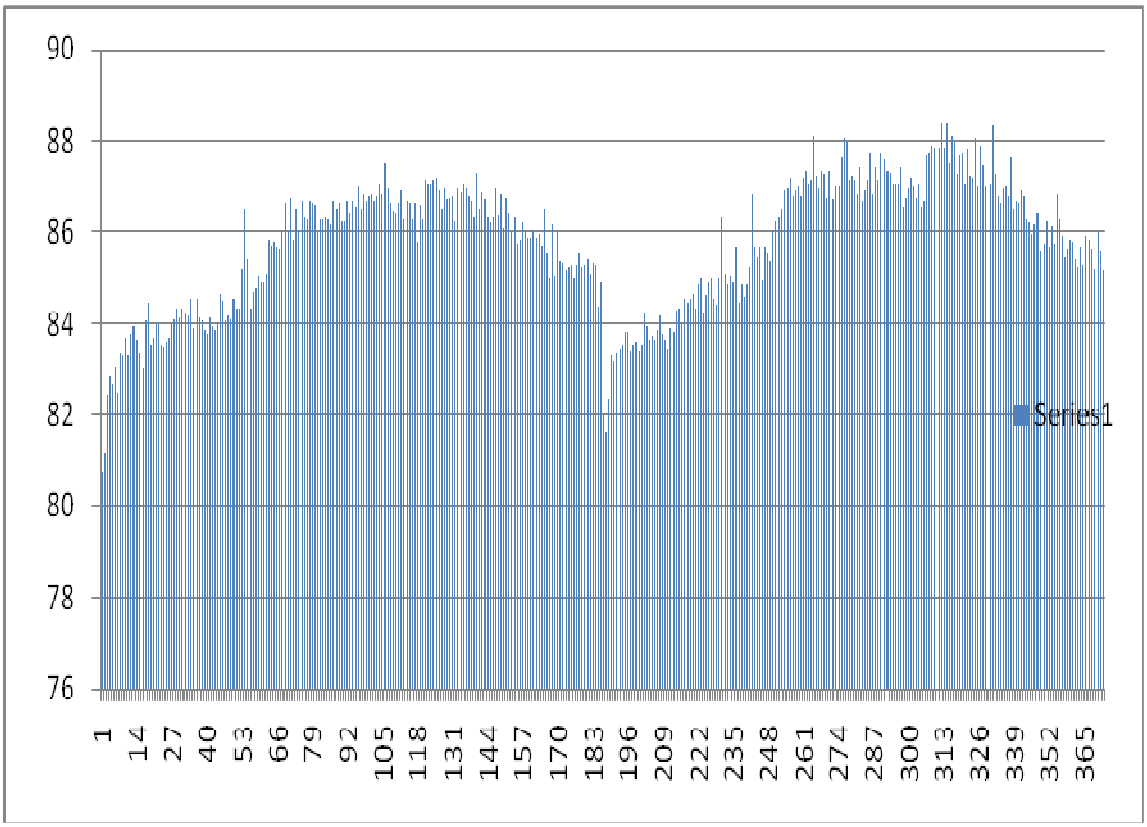

Figure 3.2: GPU Individual Timings for launching the kernel obtained from the the Nvidia GPU profiler

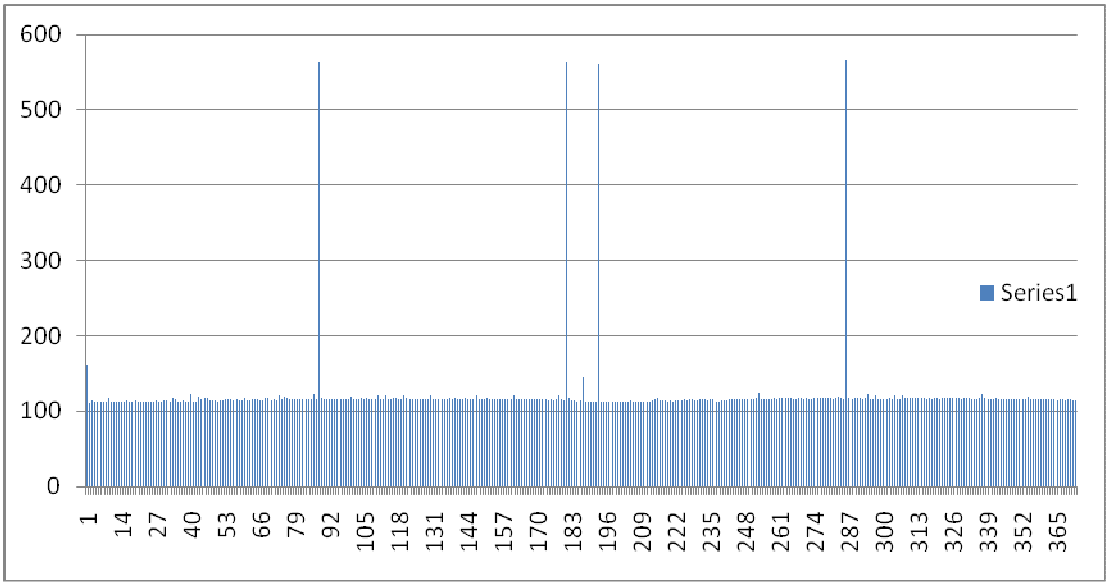

Figure 3.3: CPU Individual Timings for launching the kernel obtained from the the Nvidia GPU profiler

5.3.2 Case 2:

Final Result
Specification: code running on GPU having total time for execution of GPU Kernel 108 ms by profiling

Latest 108 ms

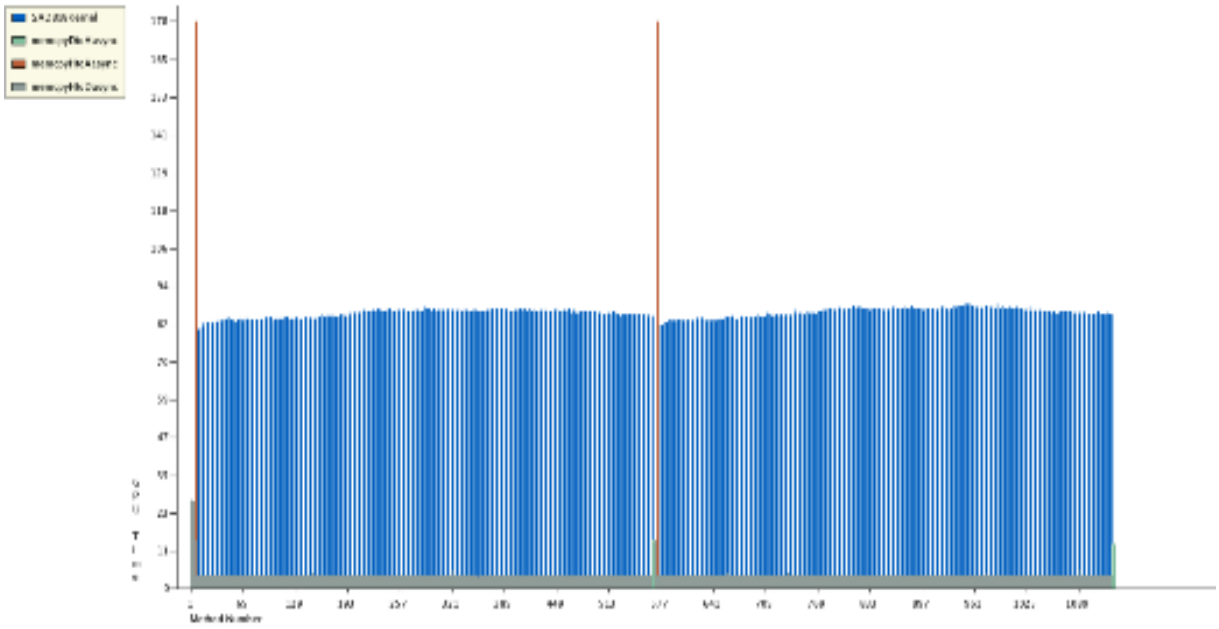

Figure 3.4: Timings for Kernel Execution from Nvidia Visual Profiler

Here maximum kernel launch time for kernel execution is 66 microseconds and minimum is 58 microseconds

Which gives total time to process one frame both in backward and forward direction

Table: Kernel Execution Timing

	GPU Time	CPU Time
Total Kernel Execution Time	24701.73µs	36648 µs
Total kernel Time	21453.44 µs	33489 µs
% total time	86.85%	91.38%

And drawing the curve by the data obtained from Nvidia Visual profiler

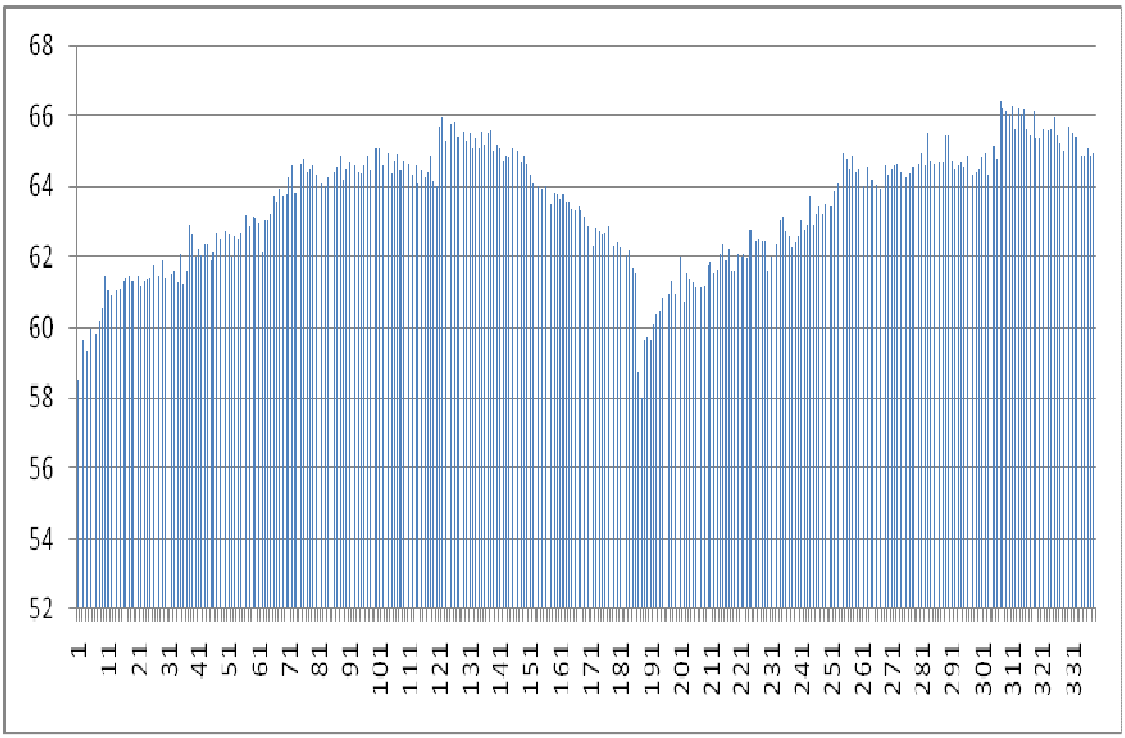

Figure 3.5: Individual Timings for Kernel Execution on GPU Ploted on Excel Sheet

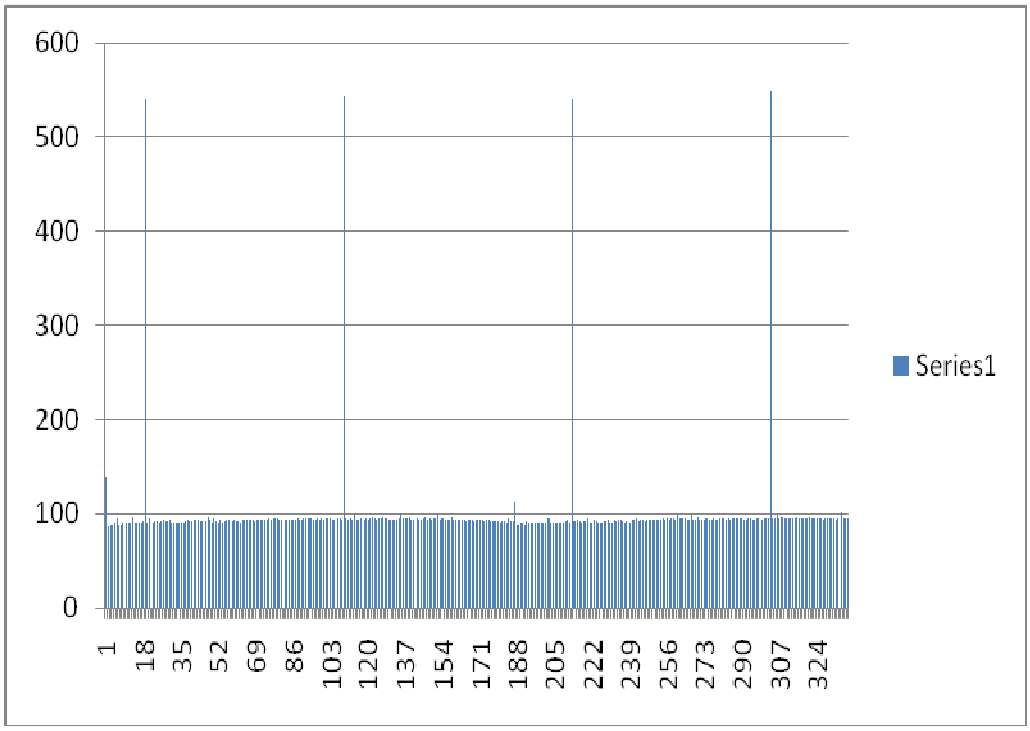

Figure 3.6: Individual Timings for Kernel Execution on CPU Ploted on Excel Sheet

Chapter 6

Conclusion

In this project Motion estimation for GPU code of MCTI has been successfully ported on OpenCL platform and parallelization has been done using the new chessboard pattern algorithm which makes the code faster to execute and got the result as per the target time.

Timing analysis of MCTI codec on Nvidia GPU has given very surprising results, as there was a vast difference between the time taken to execute the function made by us in pure C language and time taken by its inbuilt APIs doing the same functionality. We used maximum inbuilt APIs of OpenCL to reduce the timing of the MCTI codec for the calculation of Motion Vectors between adjacent frames.

We have reduced the computational overhead by optimizing the Motion estimation Code with OpenCL platform and using Nvidia GPUs by breaking the computation to some parts and give part of it to CPU and some to the GPU to enhance the performance.

On different phases of the development of code we calculated the GPU kernel execution timing and time to launch the kernel and pointed out the major factors for that consume more time, every time the phase of the codec changes and timings improves. There is still a possibility of improving the timing of kernel execution by reducing the time overhead to launch the kernel and to reduce the overhead of reading and writing the buffer of GPU, which could be seen in the next release of OpenCL Specification by Khronos Group. OpenCL has many possibilities in

parallel programming as whole electronics industry in moving towards miniaturization and parallelization. The Timing Analysis of MCTI Codec has been done successfully and the results were very close to that of target timings, and this was shown by Nvidia OpenCL visual profiler in our experiment and research.

Adding to this we have imposed some solutions to the existing Problem of Parallelization and overcome it. Development & Benchmarking of this application had reduced the CPU-GPU cycles penalty and memory buffer overhead uses.

References

1.) http://www.khronos.org/opencl/

2.)http://www.khronos.org/news/press/releases/khronos_launches_heterogeneous_computing_initiative/

3.)http://www.khronos.org/news/press/releases/the_khronos_group_releases_opencl_1.0_specification/

4.) Benchmarking Process ,Provided by ST documents.

5.) MCTI ,STMicroelectronics Confidential propriety.

6.) Khronos OpenCL Working Group. *The OpenCL Specification* ,Introduction and Overview-June 2010.

7.) NVIDIA CUDA C Programming Guide 3.2. 2010.

8.)NVIDIA Tesla GPUs Power World's Fastest Supercomputer. http://pressroom.nvidia.com/easyir/customrel.do?easyirid=A0D622CE9F579F09&prid=678988&releasejsp=release_157, 2010. [Online; accessed 31-April-2011].

9.) OpenCL 1.0 specification

10.) http://www.khronos.org/

11.) macresearch.org/ OpenCL tutorial podcast: episode 2.

12.) OpenCL Programming Reference Card 1.0