Analysis And Characterization Of Popular OpenCL Kernels

BY

Shruti Chhatbar

11MCEC05

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

March - 2013

Abstract

In all computing domains heterogeneous parallel computing platforms composed of CPUs, GPUs, FPGAs and DSPs are widening their user base. GPUs are emerging as a general purpose high performance computing device. GPGPU research is growing very fast as a cost effective approach for accelerating data and compute intensive application which has made many new workloads available. It has been driven by the introduction of C-based programming environments such as NVIDIA's CUDA , OpenCL. OpenCL (Open Computing Language) is an open standard and emerging parallel programming model to write parallel applications for such heterogeneous platforms. However there is no systematic approach to analyse and characterize the workload to assist future work on microarchitecture design, application re-structuring and compiler optimizations. In this paper I will present the software architecture that produces interesting statistics to understand the dynamic behaviour of the GPGPU. In addition to that a set of metrics for evaluation of OpenCL kernels is proposed. ArchOCL is a software model that includes dynamic library that support all OpenCL application to produce statistics to analyse OpenCL application. Development of ArchOCL model includes implementa-tion of OpenCL APIs, Compiler enhancement to support OpenCL kernel compilation and Statistics collection.

Contents

List of Figures

Chapter 1

Introduction

In last few decades, the performance of processors has increased drastically and number of cores per CPU is also found to be increased gradually. With hundreds of in-order cores per chip, Graphics Processing Unit (GPU) provides performance throughput on data parallel and computation intensive applications. Therefore, a heterogeneous microarchitecture, consisting of chip multiprocessors and GPUs seems to be good choice for data parallel algorithms. Nvidia CUDA[3], AMD stream and OpenCL[2]. Emerging CPU-GPU heterogeneous multicore processing has motivated the computer architecture research community to study various microarchitectural designs, optimizations, and analysis for GPU.

Modern processor architectures parallelism is required to increase the performance. Fixed function rendering devices are converted into programmable parallel processors. Now a days since computer systems include highly parallel CPUs, GPUs and other types of processors, it is very important to enable software developers to take full benefit of such heterogeneous processing platforms. Thus OpenCL plays an important role as heterogenous computing lagnuage.

Characterization is the process of classification of kernels or workload, measuring those classes and indentifying their impact. This thesis consists of two parts which includes workload characterization without performance measurements and other is performance characterization of media benchmark workload.

1.1 OpenCL and CUDA

Two comparisons have been drawn between CUDA and OpenCL since its inception. They both draw the same conclusions: if the OpenCL implementation is correctly tweaked to suit the target architecture, it performs no worse than CUDA. Because the key feature of OpenCL is portability (via its abstracted memory and execution model), the programmer is not able to directly use GPU speci c technologies, unless they are willing to give up direct portability by using device speci c technology such as inline PTX. CUDA is more directly connected to the platform upon which it will be executing because it is limited to Nvidia hardware. Compiler technology for both standards supported by Nvidia's toolkit is based upon LLVM and compiles to Nvidia's PTX instruction set abstraction.

CUDA and OpenCL are two di erent frameworks for GPU programming. OpenCL is an open standard that can be used to program CPUs, GPUs, and other devices from di erent vendors, while CUDA is speci c to NVIDIA GPUs. Open Computing Language (OpenCL)[2] and Compute Unifed Device Architecture (CUDA) are two interfaces for GPU computing, both presenting similar features but through di erent programming interfaces. Both OpenCL and CUDA call a piece of code that runs on the GPU "a kernel".

CUDA is a proprietary API and set of language extensions that works only on NVIDIA's GPUs. This may have been ne for students experimenting with a new approach, but mainstream ISVs and other large scale developers need the inherent exibility in industry standards. With a standard, cross platform API, developers can deliver solutions on multiple vendors' hardware while streamlining their development processes and timelines. OpenCL , by the Khronos Group, is an open standard for parallel programming using Central Processing Units (CPUs) , GPUs, Digital Signal Processors (DSPs) and other types of processors.

1.2 GPGPU

GPGPU = General Purpose computation on Graphics Processing Units. Graphics Processing Units (GPUs) can be used to speed up wide range of applications. Graphics chips started as fixed function graphics processors but became increasingly programmable and computationally powerful, which led NVIDIA to introduce the first GPU. In the 1999-2000 timeframe, computer scientists and domain scientists from various elds started us-ing GPUs to accelerate a range of scientific applications. This was the advent of the movement called GPGPU, or General Purpose computation on GPU. While users achieved unprecedented performance (over 100x compared to CPUs in some cases), the challenge was that GPGPU required the use of graphics programming APIs like OpenGL to program the GPU. This limited accessibility to the tremendous capability of GPUs for science[6].

Beyond the obvious need for CPUs to drive execution, most mainstream applications are heterogeneous in nature. They have some functions that accelerate well on multicore CPUs, and others that are perfectly suited for a GPU's data parallel architecture. A good development platform needs to take that into account this is the di erence between GPGPU as a niche accelerator and GPGPU as a new baseline feature, ready for tomorrow's systems and applications

Chapter 2

Literature Survey

Creating applications for heterogeneous parallel processors is difficult. CPU based parallel programming models are standards but have a shared address space and do vector operations are not covered. General purpose GPU programming models covers complex memory hierarchies and vector operations but also they are platform, vendor or hardware specific. Due to this its difficult to check compute power of heterogeneous CPUs, GPUs and other types of processors from a single, multi-platform. There is a need to enable software developers to effectively take full advantage of heterogeneous processing platforms from high performance compute servers, through desktop computer systems to handheld devices that include a diverse mix of parallel CPUs, GPUs and other processors such as DSPs and the cell/B.E. processor.

OpenCL (Open Computing Language) is an open free standard language for programming across CPUs, GPUs and other processors, so that software developers portable and efficient access to these heterogeneous processing platforms. OpenCL supports a wide range of applications, ranging from embedded and consumer software to HPC solutions, through a low-level, high performance, portable abstraction. OpenCL creates metal programming interface and forms the foundation layer of a parallel computing ecosystem of platform independent tools, middleware and applications. OpenCL is particularly suited to play an increasingly significant role in emerging interactive graphics applications that combine general parallel compute algorithms with graphics rendering pipelines. OpenCL is made of an API for communicating parallel computation across heterogeneous architectures; and a cross platform language with a well specified environment. The OpenCL standard:

- Supports both data and task based parallel programming models
- Utilizes a subset of ISO C99 with extensions for parallelism
- Defines consistent numerical requirements based on IEEE 754
- Defines a configuration profile for handheld and embedded devices
- Efficiently interoperates with OpenGL, OpenGL ES and other graphics APIs

2.1 The OpenCL Architecture

OpenCL is a framework for doing programming in parallel way and includes a language, libraries and a runtime system to support software environment. The target of OpenCL is for programmers who wants to write portable and also efficient code. This includes library writers, middleware vendors, and performance oriented application program-mers. So OpenCL is a low-level hardware abstraction with a environment to support programming with overview of underlying hardware[2]. Following is the hierarchy model of OpenCL [2]. Following is the hierarchy model of OpenCL:

- Platform Model
- Memory Model
- Execution Model
- Programming Model

2.1.1 Platform Model

The model is made such that one host is connected to one or many OpenCL devices. As shown in diagram an OpenCL device is divided into one or many compute units (CUs) and then divided into one or many processing elements (PEs). Computations here occurs within the processing elements. An OpenCL application runs on a host according to the models native to the host platform. The OpenCL application gives argumets from the host to do computational work on the processing elements in a device. In a compute unit, the processing elements execute a single stream of instructions as SIMD units or as SPMD units. SIMD - execute in lockstep with a single stream of instructions. SPMD - each PE maintains its own program counter[4]..

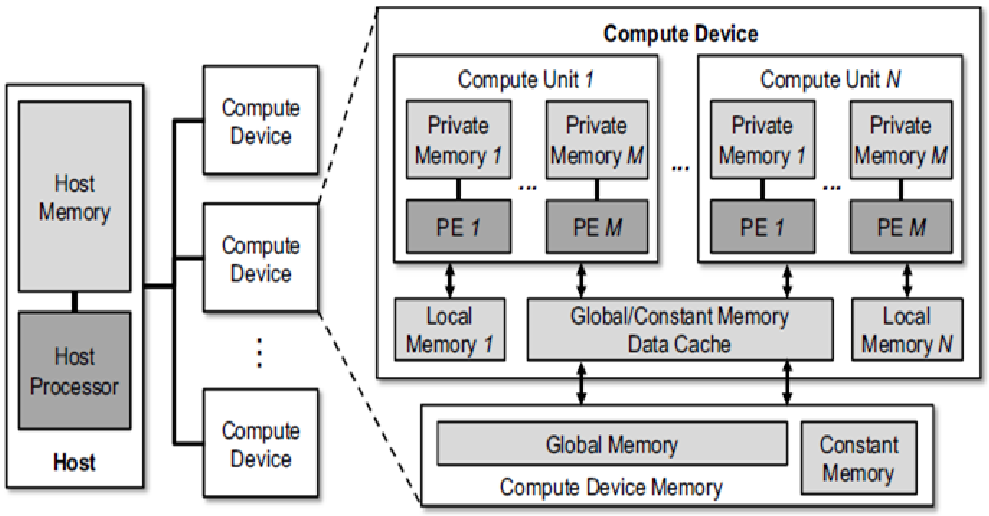
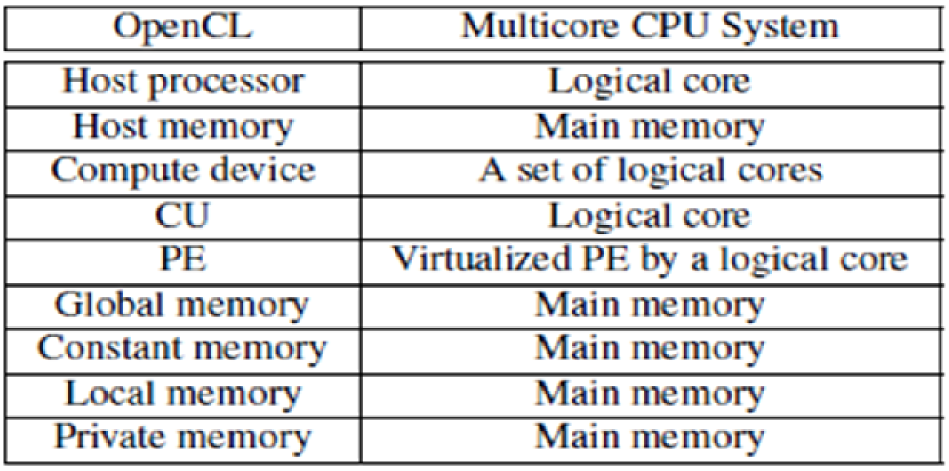


Figure 2.1: Platform Model and Memory Model

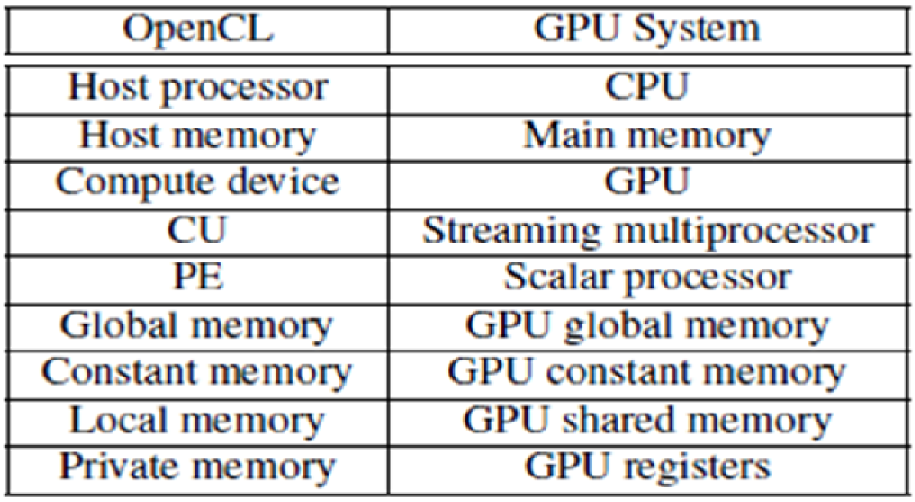
Mapping OpenCL platform model to the multicore CPU system[4]



OpenCL	Multicore CPU System
Host processor	Logical core
Host memory	Main memory
Compute device	A set of logical cores
CU	Logical core
PE	Virtualized PE by a logical core
Global memory	Main memory
Constant memory	Main memory
Local memory	Main memory
Private memory	Main memory

Figure 2.2: Mapping OpenCL platform model to the multicore CPU system

Mapping OpenCL platform model to the GPU system[4]



OpenCL	GPU System
Host processor	CPU
Host memory	Main memory
Compute device	GPU
CU	Streaming multiprocessor
PE	Scalar processor
Global memory	GPU global memory
Constant memory	GPU constant memory
Local memory	GPU shared memory
Private memory	GPU registers

Figure 2.3: Mapping OpenCL platform model to the GPU system

2.1.2 Memory model

There are four memory regions among Work-item(s) executing a kernel:

- Global Memory :

  Depending on devices read and writes to this memory may be cached. This is generally for allowing read/write access among threads and blocks. Thread can read to and from a memory object

- Constant Memory :

  During working of kernel this part of memory remains constant and it is part of global memory. Allocation and initialization of this memory is done by host and then put into constant memory.

- Local Memory :

  A memory portion local to a block. Variables that are shared by all threads in a block are allocated by this memory. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local or private memory region may be aligned to part of the global memory.

- Private Memory :

  This region is private to single thread. Variables defined in one thread private memory are not visible to another thread.

2.1.3 Execution Model

To execute a kernel context is required and created by host. The following resources are included by context:

- Devices: The pool of OpenCL devices that can utilized by the host

- Kernels: The OpenCL devices run this OpenCL functions.

- Objects in Program: The program source and program exes that forms the kernels.

- Memory Objects: A collection of memory objects that can be seen by devices and host.

Kernel operates on values in memory objects. Functions from OpenCL works on context created by host. Command queue is used to do communication between kernels and devices. Command queue schedules various commands and execution is done according. Nd range kernel is passed through device and then results are again given back to host memory. Buffers are used for this purpose[3] [3]

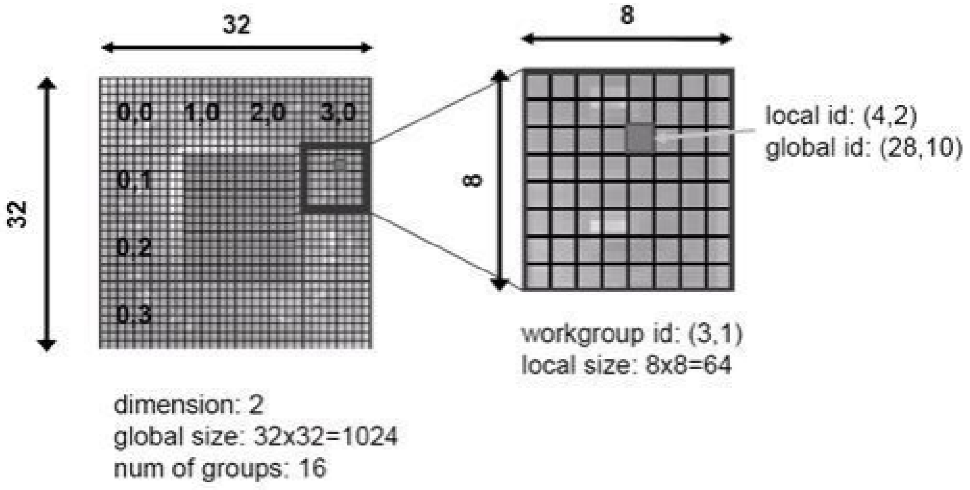


Figure 2.4: Work-item and work-group Example

2.1.4 Programming Model

The OpenCL execution model consists of data parallel and task parallel programming models, also have various combinations of these two models. Data parallelism is the primary model forming the design of OpenCL.

Data Parallel Programming Model : This model tells that a memory object is there which computes in terms of a sequence of instructions given to various factors of object. The index space plays an important role in defining work item and mapping of data with workitems.

Task Parallel Programming Model : In this model no kernels work is dependent on any index space. It is same as running a kernel in a compute unit having a work-item in a block

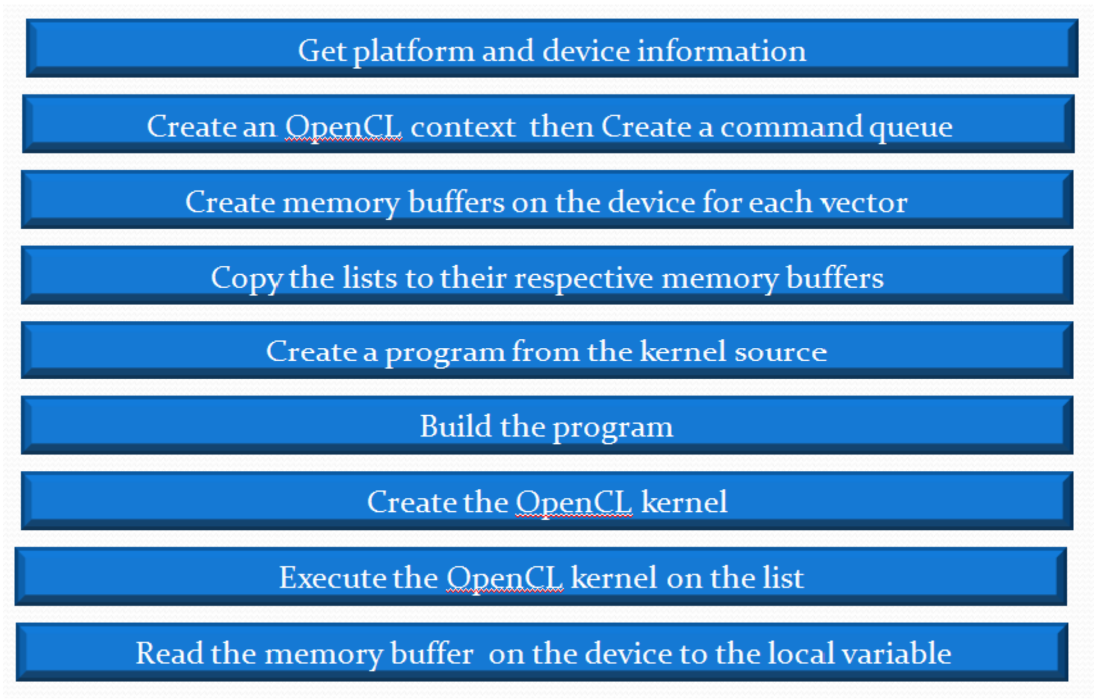

Get platform and device information
Create an OpenCL context then Create a command queue
Create memory buffers on the device for each vector
Copy the lists to their respective memory buffers
Create a program from the kernel source
Build the program
Create the OpenCL kernel
Execute the OpenCL kernel on the list
Read the memory buffer on the device to the local variable

Figure 2.5: Heirarchy of OpenCL program

2.2 OpenCL Framework

The OpenCL environment allows to use one host and many devices as a single heterogeneous system. The framework contains the following components[3][3]:

- OpenCL Platform layer:

  The platform layer helps the host program to create contexts and find OpenCL devices and their capabilities.

- OpenCL Runtime:

  Once context has been created the runtime allows the host program to work on contexts.

- OpenCL Compiler:

  Program executables are formed by opencl compiler that have OpenCL kernels. The compiler forms OpenCL C programming language and it will be a subset of the ISO C99 language with extensions that will have parallelism.

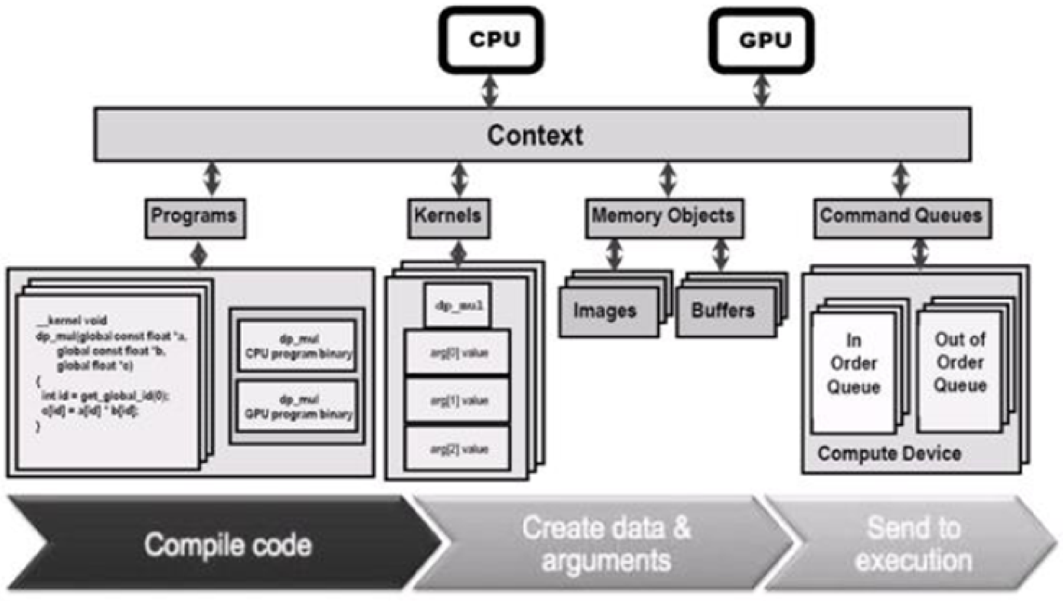


Figure 2.6: OpenCL Framework And Flow

2.3 Clang Compiler

Clang is an "LLVM native" C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles (e.g. about 3x faster than GCC when compiling Objective-C code in a debug configuration), extremely useful error and warning messages and to provide a platform for building great source level tools. The Clang Static Analyzer is a tool that automatically finds bugs in your code, and is a great example of the sort of tool that can be built using the Clang frontend as a library to parse C/C++ code. Some of the goals for the clang include the following:

End-User Features:

- Fast compiles and low memory use
- Expressive diagnostics (examples)
- GCC compatibility

Utility and Applications:

- Modular library based architecture
- Support diverse clients (refactoring, static analysis, code generation, etc)
- Allow tight integration with IDEs
- Use the LLVM 'BSD' License

Internal Design and Implementation:

- A real-world, production quality compiler
- A simple and hackable code base
- A single unified parser for C, Objective C, C++, and Objective C++

2.4 Mesa library

Mesa is an open-source implementation of the OpenGL specification - a system for rendering interactive 3D graphics. A variety of device drivers allows Mesa to be used in many different environments ranging from software emulation to complete hardware acceleration for modern GPUs. Mesa ties into several other open-source projects: the Direct Rendering Infrastructure and X.org to provide OpenGL support to users of X on Linux, FreeBSD and other operating systems [7].

Mesa serves following purposes:

- Mesa is quite portable and allows OpenGL to be used on systems that have no other OpenGL solution.
- Software rendering with Mesa serves as a reference for validating the hardware drivers.
- A software implementation of OpenGL is useful for experimentation, such as testing new rendering techniques.

2.5 Characterization Methodology

Analytical GPU models. Analytical GPU models. Hong et. al. propose a predictive analytical performance model for GPUs. The main components of their model are memory parallelism among concurrent warps and computational parallelism. By tuning their model to machine parameters, static characteristics of applications, and regressions, their performance model predicts kernel runtimes with errors of 13percent or less. Our approach, on the other hand, does not assume particular principal components and instead attempts to determine them based on measurable statistics that may change substantially with the evolution of GPU and CPU micro-architecture[5].

2.5.1 ArchOCL Design goal

Momentum is achieved by GPU as a cost-effective approach in data acceleration and compute-intensive applications.

Design goal for ArchOCL software architecture is to characterize the workload for OpenCL. ArchOCL is a software model which runs any OpenCL application and produces interesting statistics that reveals the dynamic behavior of the application. Statistics collected by ArchOCL is useful for GPU architecture design. It also helps developers to effectively port their applications in OpenCL. Example of statistics are Shared Memory Density which is a ratio of number of shared memory access to number of global memory access, SIMD Channel Utilization which is a number of threads active averaged over all dynamic instructions and many more.[1]

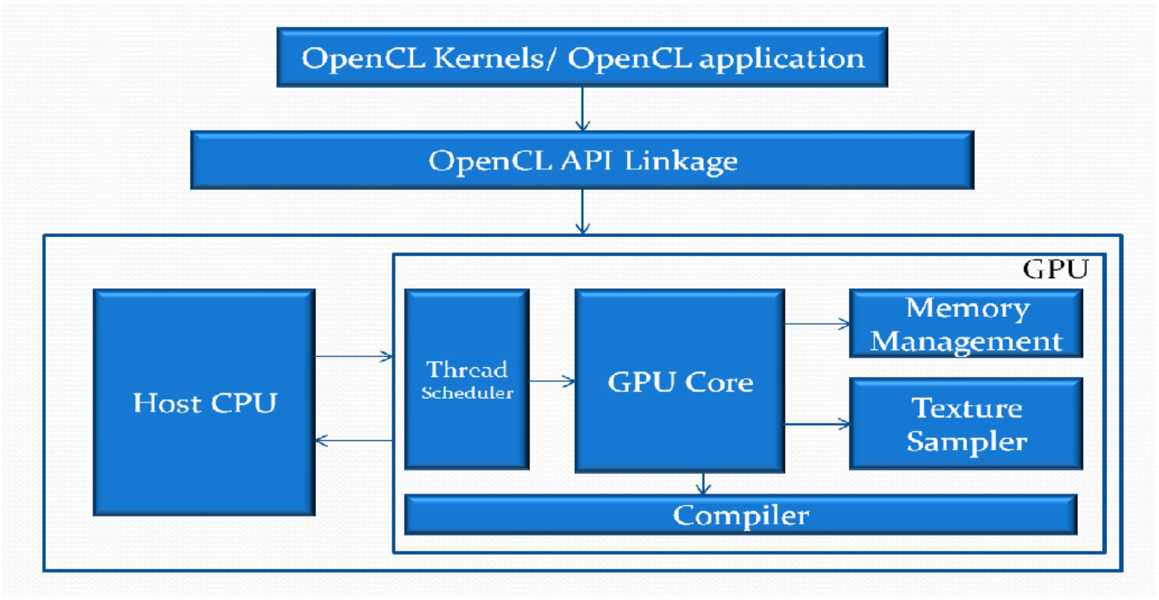


Figure 2.7: Architecture Of ArchOCL

ArchOCL is useful for

- Multicore and manycore architecture research
- Reveals dynamic behaviour of application
- Evaluation of impact on microarchitecture design and parallelization.

2.5.2 Performance characterization of Media Benchmark

Feature Detection, Feature Extraction and Feature Matching are the basic steps in any media benchmark algorithm. Goal of using OpenCL and OpenCV with this media benchmark is to improve the performance on each of these steps. Nearest Neighbor search is an interesting problem in computer vision application. Next goal is to check which nearest neighbor performs best while changing its parameters.

Chapter 3

API Implementation

3.1 OpenCL Platform Layer

This section describes the OpenCL platform layer which implements platform specific features that allow applications to query OpenCL devices, device configuration information, and to create OpenCL contexts using one or more devices.

3.1.1 Querying Platform

clGetPlatformIDs: The list of platforms available can be obtained using this API.
clGetPlatformInfo: This API gives specific information about the OpenCL platform.

3.1.2 Querying Device

clGetDeviceIDs: The list of devices available on a platform can be obtained using the this API.
clGetDeviceInfo: This API gives specific information about an OpenCL device.

3.1.3 Contexts

Contexts are used by the OpenCL runtime for managing objects such as command queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the context.
clCreateContext: This API creates an OpenCL context. An OpenCL context is created with one or more devices.
clCreateContextFromType: This API creates an OpenCL context from a device type that identifies the specific device(s) to use.
clRetainContext: This API increments the context reference count.
clReleaseContext: This API decrements the context reference count.
clGetContextInfo: This API can be used to query information about a context.

3.2 OpenCL Runtime

This section describes the API calls that manage OpenCL objects such as command queues, memory objects, program objects, kernel objects for kernel functions in a program and calls that allow you to enqueue commands to a command-queue such as executing a kernel, reading or writing a memory object.

3.2.1 Command Queues

OpenCL objects such as memory, program and kernel objects are created using a context. Operations on these objects are performed using a command-queue. The command-queue can be used to queue a set of operations (referred to as commands)in order. Having multiple command-queues allows applications to queue multiple independent commands without requiring synchronization.
clCreateCommandQueue: This API creates a command-queue on a specific device.

clRetainCommandQueue: This API increments the command queue reference count.
clReleaseCommandQueue: This API decrements the command queue reference count.
clGetCommandQueueInfo: This API can be used to query information about a command-queue.
clSetCommandQueueProperty: This API can be used to enable or disable the properties of a command-queue.

3.2.2 Memory Objects

Memory objects are categorized into two types: buffer objects, and image objects. A buffer object stores a one-dimensional collection of elements whereas an image object is used to store a two- or three- dimensional texture, frame-buffer or image. Elements of a buffer object can be a scalar data type (such as an int,float), vector data type, or a user-defined structure. **clCreateBuffer:** A buffer object is created using this API.
clEnqueueReadBuffer & clEnqueueWriteBuffer: This APIs enqueue commands to read from a buffer object to host memory or write to a buffer object from host memory.
clEnqueueCopyBuffer: This API enqueues a command to copy a buffer object identified by src buffer to another buffer object identified by dst buffer.
clRetainMemObject: This API increments the memobj reference count.
clReleaseMemObject: This API decrements the memobj reference count.
clCreateImage2D: An image (1D, or 2D) object is created using this API.
clCreateImage3D: A 3D image object is created using this API.
clGetSupportedImageFormats: This API can be used to get the list of image formats supported by an OpenCL implementation.
clEnqueueReadImage & clEnqueueWriteImage: This APIs enqueue commands

to read from a 2D or 3D image object to host memory or write to a 2D or 3D image object from host memory.

clEnqueueCopyImage: This API enqueues a command to copy image objects.

clEnqueueCopyImageToBuffer: This API enqueues a command to copy an image object to a buffer object.

clEnqueueCopyBufferToImage: This API enqueues a command to copy a buffer object to an image object.

clEnqueueMapBuffer: This API enqueues a command to map a region of the buffer object given by buffer into the host address space and returns a pointer to this mapped region.

clEnqueueMapImage: This API enqueues a command to map a region in the image object given by image into the host address space and returns a pointer to this mapped region.

clEnqueueUnmapMemObject: This API enqueues a command to unmap a previously mapped region of a memory object. Reads or writes from the host using the pointer returned by clEnqueueMapBuffer or clEnqueueMapImage are considered to be complete.

clGetMemObjectInfo: This API is used to get information that is common to all memory objects (buffer and image objects).

clGetImageInfo: This API is used to get information specific to an image object created with clCreateImagef2D—3Dg

3.2.3 Sampler Objects

A sampler object describes how to sample an image when the image is read in the kernel. The built-in functions to read from an image in a kernel take a sampler as an argument. The sampler arguments to the image read function can be sampler objects created using OpenCL functions and passed as argument values to the kernel or can be samplers declared inside a kernel.

clCreateSampler: This API creates a sampler object.
clRetainSampler: This API increments the sampler reference count.
clReleaseSampler: This API decrements the sampler reference count.
clGetSamplerInfo: This API returns information about the sampler object.

3.2.4 Program Objects

An OpenCL program consists of a set of kernels that are identified as functions declared with the kernel qualifier in the program source. OpenCL programs may also contain auxiliary functions and constant data that can be used by kernel functions. A program object encapsulates the following information:

- An associated context.
- A program source or binary.
- The latest successfully built program executable, the list of devices for which the program executable is built, the build options used and a build log.
- The number of kernel objects currently attached.

clCreateProgramWithSource: This API creates a program object for a context, and loads the source code specified by the text strings in the strings array into the program object.
clCreateProgramWithBinary: This API creates a program object for a context, and loads the binary bits specified by binary into the program object.
clBuildProgram: This API builds (compiles & links) a program executable from the program source or binary for all the devices or a specific device(s) in the OpenCL context associated with program.
clRetainProgram: This API increments the program reference count.
clReleaseProgram: This API decrements the program reference count.
clGetProgramInfo: This API returns information about the program object.

clGetProgramBuildInfo: This API returns build information for each device in the program object.

3.2.5 Kernel Objects

A kernel is a function declared in a program. A kernel is identified by the kernel qualifier applied to any function in a program. A kernel object encapsulates the specific kernel function declared in a program and the argument values to be used when executing this kernel function.
clCreateKernel: This API creates a kernel object.
clCreateKernelsInProgram: This API creates kernel objects for all kernel functions in program.
clSetKernelArg: This API is used to set the argument value for a specific argument of a kernel.
clRetainKernel: This API increments the kernel reference count.
clReleaseKernel: This API decrements the kernel reference count.
clGetKernelInfo: This API returns information about the kernel object.
clEnqueueNDRangeKernel: This API enqueues a command to execute a kernel on a device.

3.2.6 Event Objects

An event object can be used to track the execution status of a command. The API calls that enqueue commands to a command-queue create a new event object that is returned in the event argument.
clWaitForEvents: This API waits on the host thread for commands identified by event objects in event list to complete.
clRetainEvent: This API increments the event reference count.
clReleaseEvent: This API decrements the event reference count.
clGetEventInfo: This API returns information about the event object.

3.3 OpenCL APIs for OpenGL interoperability

This section describes OpenCL APIs that allow applications to use OpenGL buffer, texture and render buffer objects as OpenCL memory objects. This allows effcient sharing of data between OpenCL and OpenGL. The OpenCL API may be used to execute kernels that read and/or write memory objects that are also OpenGL objects.
clCreateFromGLbuffer: This API creates an OpenCL buffer object from an OpenGL buffer object. API returns a valid OpenCL buffer object based on OpenGL context passed as an argument.

clCreateFromGLTexture2D/3D: This API creates an OpenCL 2D/3D image object from an OpenGL 2D/3D texture object.

clGetGLObjectInfo: The OpenGL object used to create the OpenCL memory object and information about the object type i.e. whether it is a texture, render buffer or buffer object can be queried using this API.

clEnqueueAcquireGLObjects: This API creates is used to acquire OpenCL memory objects that have been created from OpenGL objects. These objects need to be acquired before they can be used by any OpenCL commands queued to a command queue.

clEnqueueReleaseGLObjects: This API is used to release OpenCL memory objects that have been created from OpenGL objects. These objects need to be released before they can be used by OpenGL.

Chapter 4

Testing Mechanism

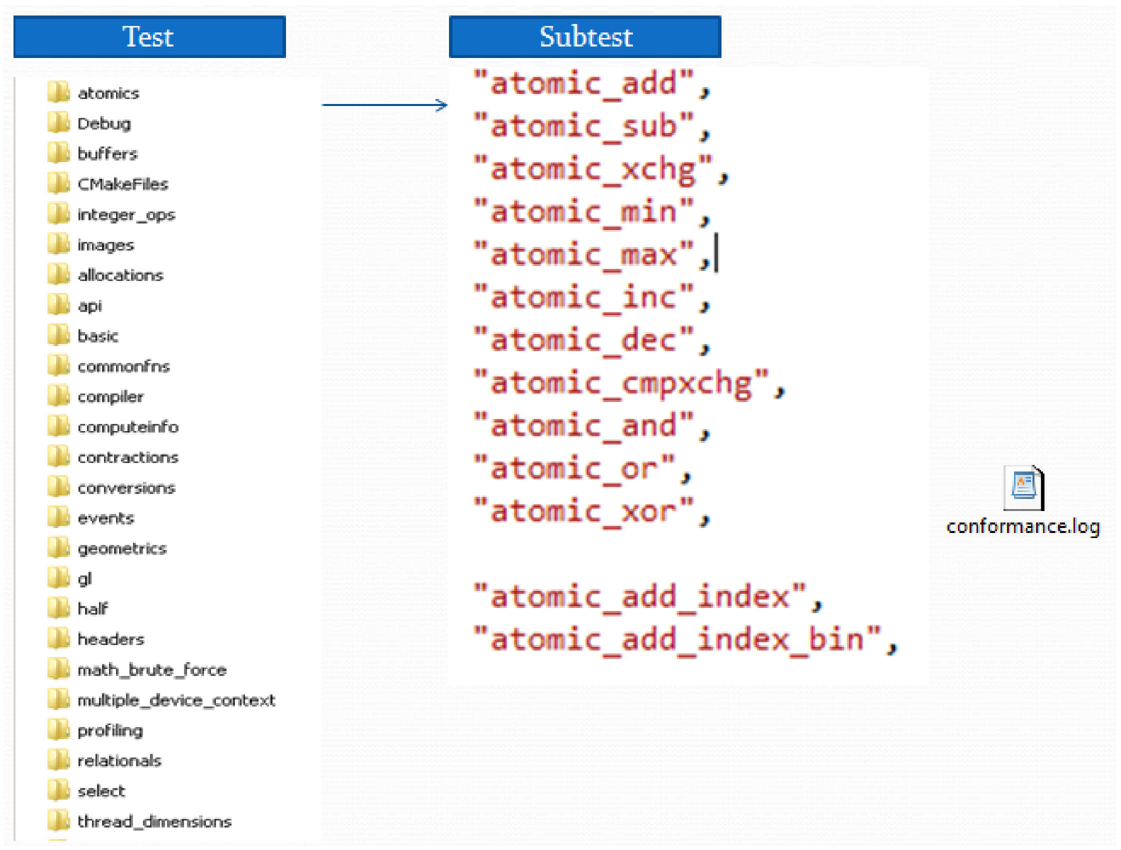
Conformance testing, also known as compliance testing, is a methodology used in engineering to ensure that a product, process, computer program or system meets a defined set of standards.

Conformance testing can be carried out by private companies that specialize in that service. In some instances the vendor maintains an in-house department for conducting conformance tests prior to the initial release of a product or upgrade. In the software industry, once the set of tests has been completed and a program has been found to comply with all the applicable standards, that program can be advertised as having been certified by the organization that defined the standards and the corporation or organization that conducted the tests.

Goals :

- Automate the process of testing.
- Check Pass rate.

Regression testing is nothing but full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine. There were total 24 tests with OpenCL each containing some subtest forming total of 458 subtests. The structure of tests is shown in figure 4.1 below:



```
atomics
Debug
buffers
CMakeFiles
integer_ops
images
allocations
api
basic
commonfns
compiler
computeinfo
contractions
conversions
events
geometrics
gl
half
headers
math_brute_force
multiple_device_context
profiling
relationals
select
thread_dimensions
```

```
"atomic_add",
"atomic_sub",
"atomic_xchg",
"atomic_min",
"atomic_max",
"atomic_inc",
"atomic_dec",
"atomic_cmpxchg",
"atomic_and",
"atomic_or",
"atomic_xor",

"atomic_add_index",
"atomic_add_index_bin",
```

Figure 4.1: Structure of Tests

We can run the above test individually or all toghether. The procedure to run the tests is explained in figure 4.2 below: In conversion, API and few others subtests were

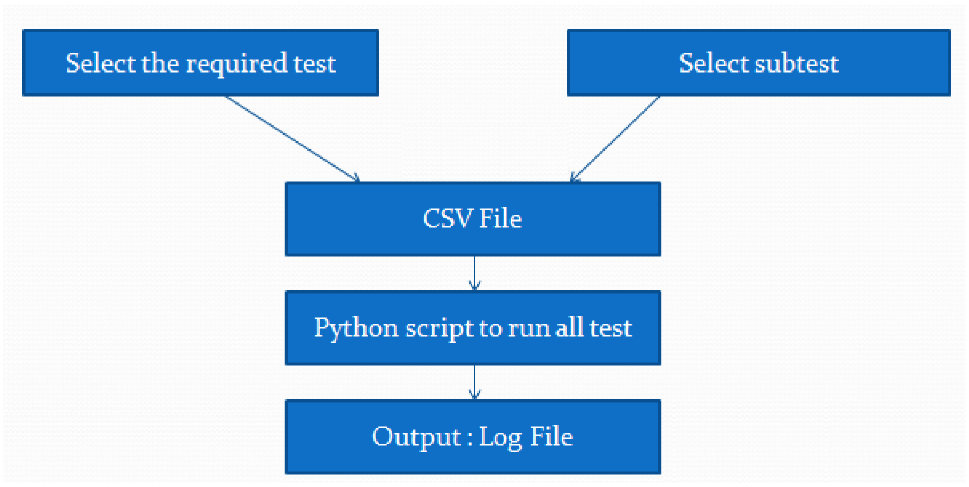


Figure 4.2: Structure of Tests

failing. So support for such subtests which were failing on ArchOCL were enabled. But we have to also check the complete pass rate so that we can verify the changes we made have not affected other tests. This is how we can check the perfectness of our tool. The technique to run tests is already shown above but as few tests were taking much time this running mechanism may take months to get the results. In order to speedup the process and utilize the resource properly grid computing environment was made which is shown in figure 4.3

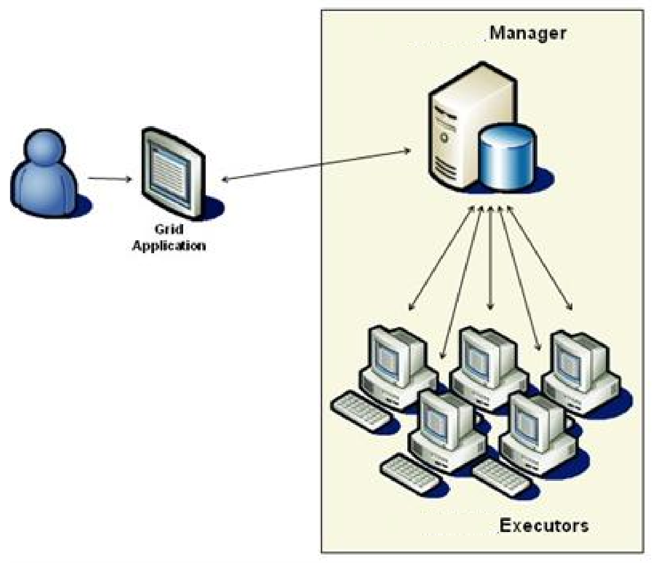


Figure 4.3: Grid Structure

Now each subtest run individually in pool of PC because of grid computing. Database is designed in such a way that instead of taking subtests from CSV it is taking from database and submitted to a PC, The system flow for such system is show in figure 4.4 below

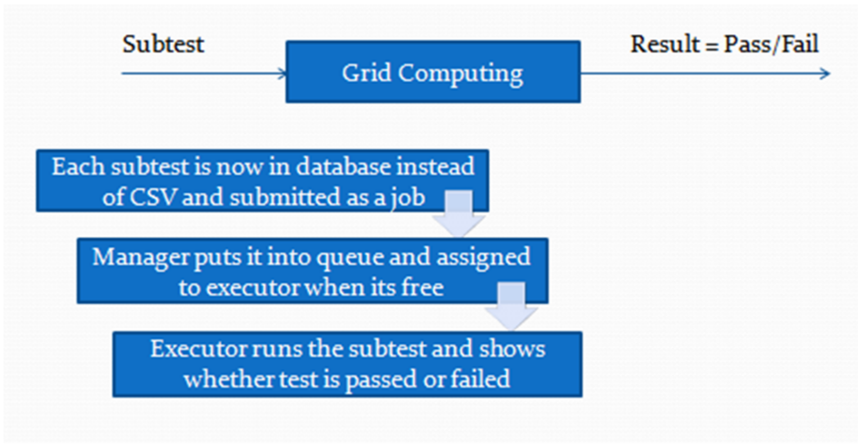
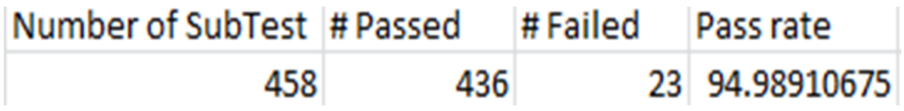

Subtest
Grid Computing
Result = Pass/Fail
Each subtest is now in database instead of CSV and submitted as a job
Manager puts it into queue and assigned to executor when its free
Executor runs the subtest and shows whether test is passed or failed

Figure 4.4: System Flow Diagram

At last we get pass/fail result in database. The final result for ArchOCL pass rate after support of some statistics and test is show in figure 4.5 below Total Test = 24



Number of SubTest	# Passed	# Failed	Pass rate
458	436	23	94.98910675

Figure 4.5: Result of Test

Chapter 5

Characterization of CLBenchmark

CLBenchmark offers an unbiased way of testing and comparing the performance of OpenCL on differerent hardware arhitectures.

Goals:

- Enable the support of CLBenchmark on ArchOCL
- Obtain Statistics for CLBenchmark

5.1 Average Channel Utilization

ArchOCL provides average number of active channels during each instruction execution, using which average channel utilization can be determined for different SIMD width.Workload without branch and barrier shows 100% channel utilization. For SIMD32 channel utilization of Sequential scan is 30.

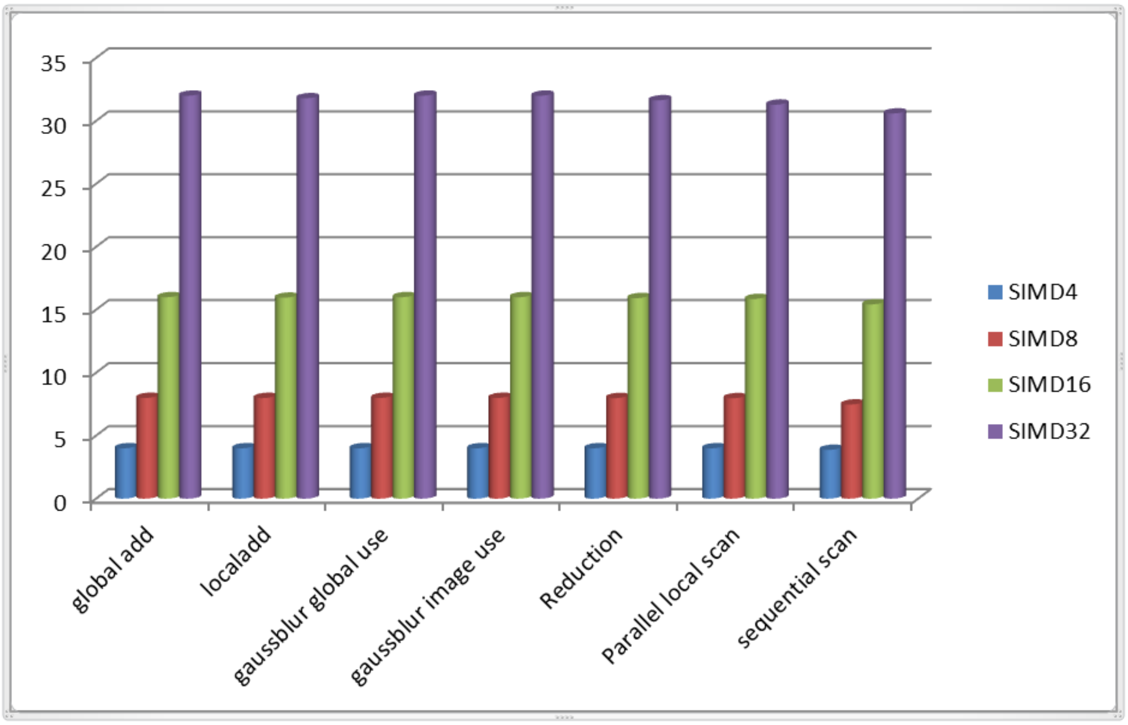


Figure 5.1: Average Channel Utilization

5.2 Instruction Breakdown

ArchOCL model gives dynamic instruction count, which is the total number of instructions executed. It also gives separate count for each instructions like branch instruction, barrier instruction,floating-point instruction, integer instruction, special instruction and memory instruction. Instruction breakdown provides an insight of the usage of different functional blocks in the GPU. Branch instruction count provides the control flow behavior. Due to warp divergence, the frequency of branch instructions plays a significant role in characterizing the workload. Barrier instruction count shows synchronization behavior of the workload.

5.3 Cache Line Hit

The smallest unit between main memory and cache that can be transferred is cache line or block. Rather than reading a single word or byte from main memory at a time, each cache entry holds a certain number of words known as cache line or cache block and whole line is read and cached at once. This takes advantage of principle of locality of reference: if one location is read then nearby location are likely to be read soon afterwards. The cache line is generally fixed in size typically ranging from 16 to 256 bytes.

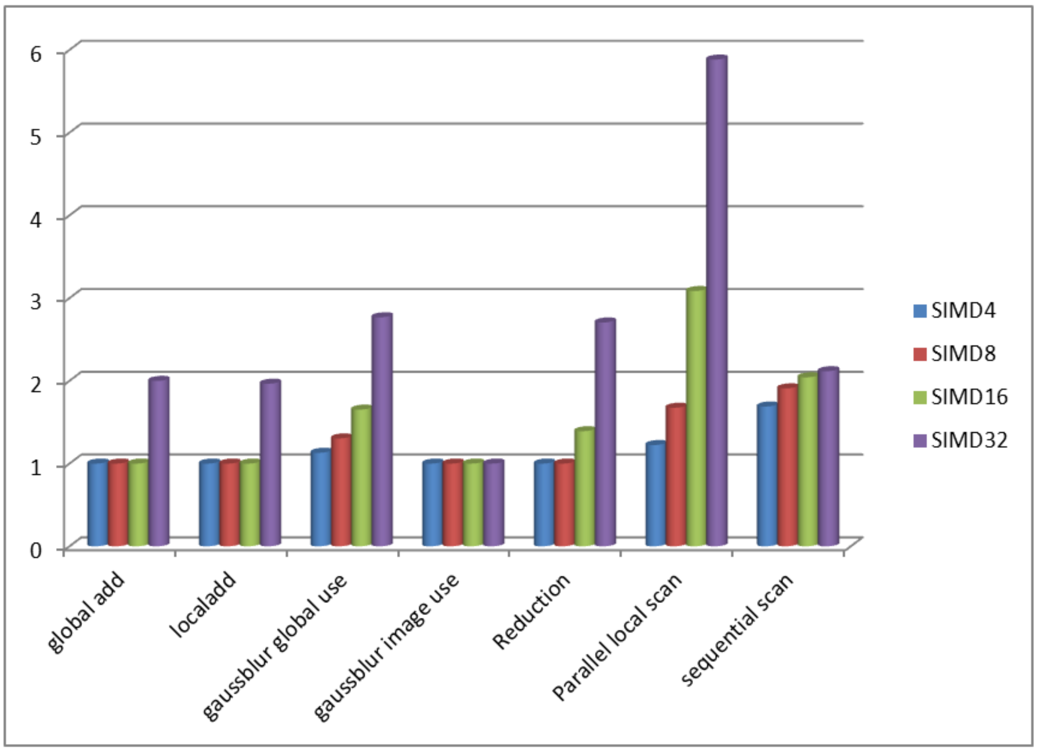


Figure 5.2: Cache Line Hit

5.4 Bank Conflicts

In GPUs the local memory is divided into memory banks. Each bank can only address one dataset at a time, so load/store to/from the same bank leads to bank conflict. Figure above shows bank conflict with 16 banks for different SIMD width.

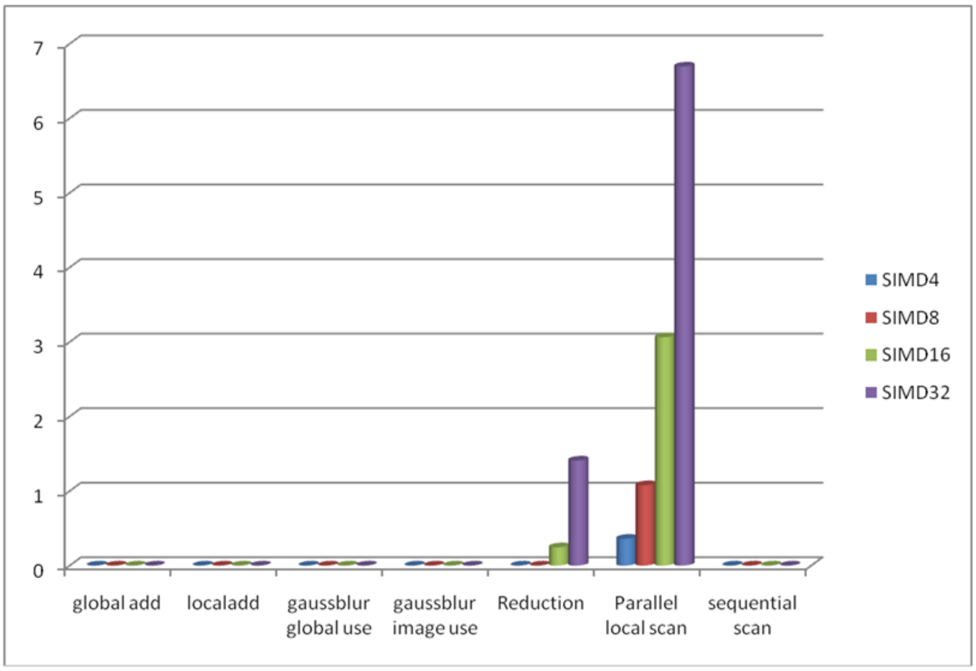


Figure 5.3: Bank Conflicts

For workloads like Reduction and parallel local scan bank conflict increases. Other has no bank conflict for all SIMD width.

5.5 Global Memory Access Per Instruction

ArchOCL model gives statistics like Amount of Local, Global and Constant memory used and Total number of Integer and Floating Point operations executed. Using these statistics, Global Bytes/Compute, SLM Bytes/Compute and Total Bytes/Compute can be determined and workload can be characterized as either compute intensive or memory intensive Global Memory Acces per instruction for global add is maximum.

Global memory access is combination of global load and store of all instruction.

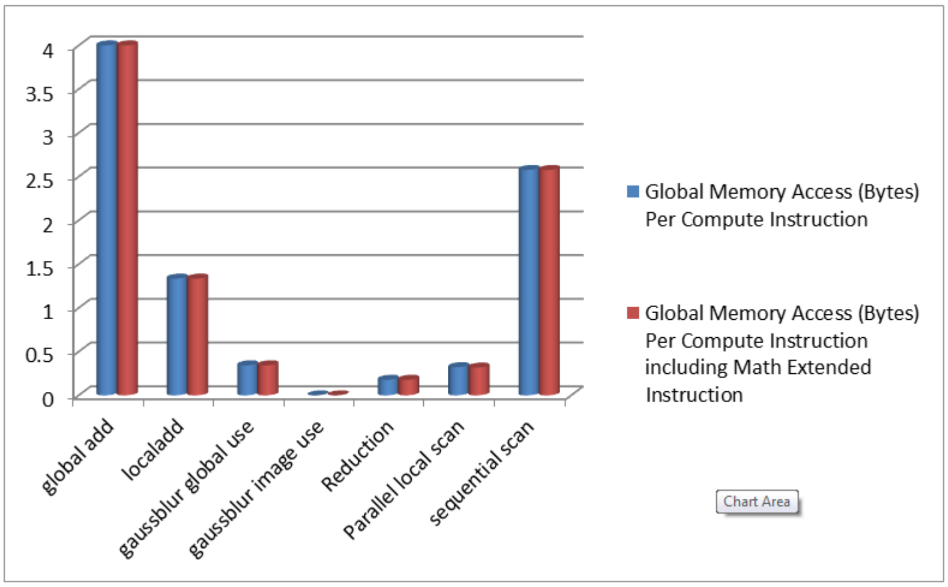


Figure 5.4: Global Memory Access Per Instruction

5.6 SLM Reuse

SLM Reuse is the ratio of shared Local Memory data access to Global Memory data access. The figure above shows that SLM is reused properly by Reduction and Parallel Local Scan which results in good performance because Shared Local Memory latency is less than the Global memory

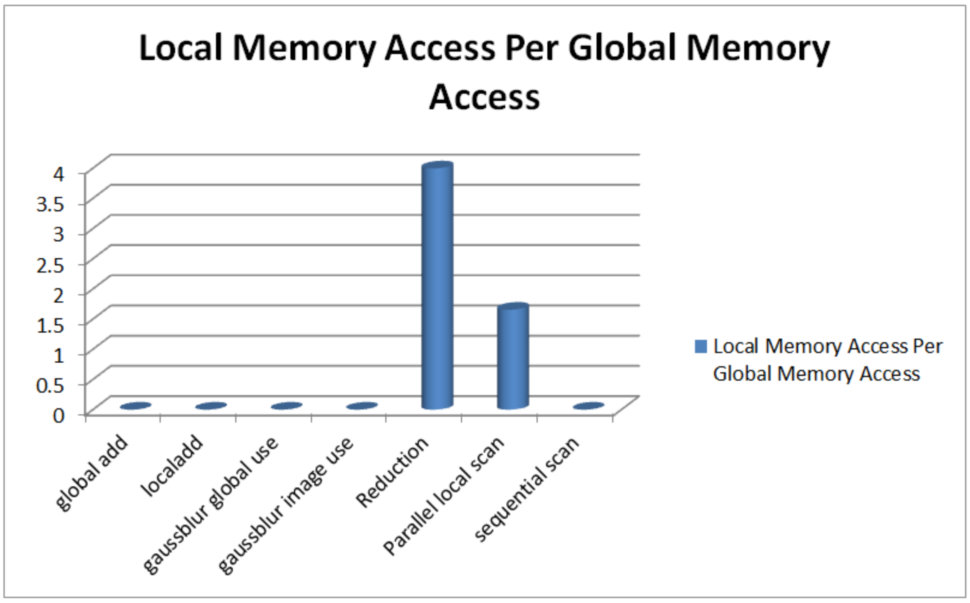


Figure 5.5: SLM Reuse

Chapter 6

Characterization using OpenCV and OpenCL

OpenCV is a free open source computer Vision Library aimed at real time computer vision problem. It is free to use under BSD license and it is cross platform. It supports C, C++, python, java interface, OpenCL and CUDA[8]

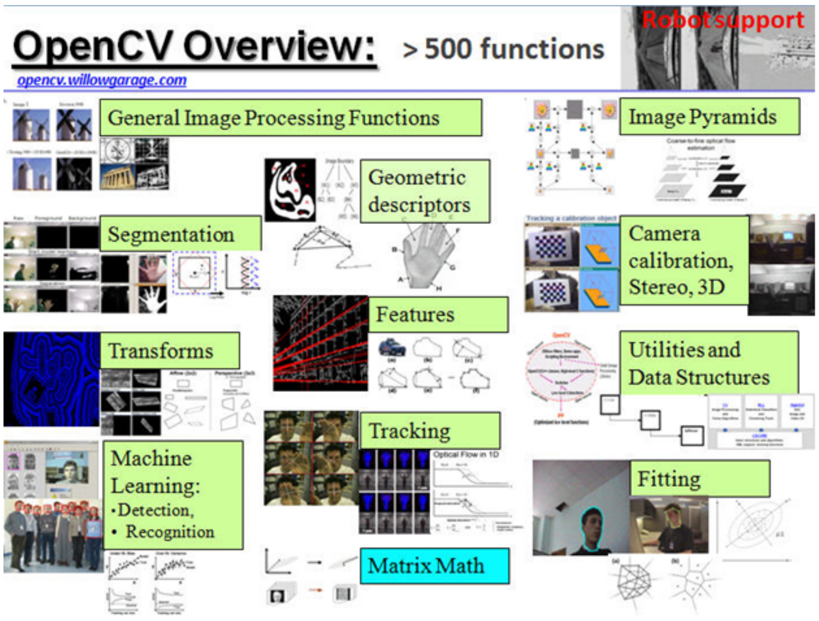


Figure 6.1: OpenCV Overview

The modular structure of OpenCV consists of modules namely core, imgproc, video, calib3d, features2d objdetect, highgui and gpu. Other helpers modules are also there such as FLANN and google test wrappers, python bindings and others.

This Chapter focuses on OpenCV OCL module. OpenCV recently started supporting classes and functions so that we can utilize OpenCL compatible device. Here I would describe OpenCV media benchmark work. There are many media benchmark namely

- Brief
- Freak
- SURF
- SIFT

There are three steps in all of these algorithm
1) Feature Detection

- Feature vector creation
- Corner, edge, important points detection

2) Feature Extractor (Descriptor)

- Descriptor creation

3) Feature Matching

- Matching of two vectors using any distance algorithm.

In this chapter we will discuss some points to convert a serial program into parallel form. After that we will check the time/performance of both sequential and parallel applications. Steps for Creating a Parallel Program
1) If you are starting with an existing serial program, debug the serial code completely
2) Identify the parts of the program that can be executed concurrently:

- Requires a thorough understanding of the algorithm
- Exploit any inherent parallelism which may exist.
- May require restructuring of the program and/or algorithm. May require an entirely new algorithm.

3) Decompose the program:

- Functional Parallelism
- Data Parallelism
- Combination of both

4) Code development

- Code may be influenced/determined by machine architecture
- Choose a programming paradigm
- Determine communication
- Add code to accomplish task control and communications

5) Compile, Test, Debug 6) Optimization

- Measure Performance
- Locate Problem Areas
- Improve them

OpenCV supports OCL matrix and few other things of OpenCL. It does not support feature detection method directly. So my goal was to convert few of above into OpenCL modules and charaterize its performance. OpenCV have serial program for BRIEF. For Feature detection Brief uses FAST algorithm. Corner detection is an approach used within computer vision systems to extract certain kinds of features and infer the contents of an image. AST is an acronym standing for Accelerated Segment Test. Instead of evaluating the circular disc only the pixels in a Bresenham circle of radius around the candidate point are considered. If contiguous pixels are all brighter than the nucleus by at least or all darker than the nucleus by, then the pixel under the nucleus is considered to be a feature. The first corner detection algorithm based on the AST is FAST (Features from Accelerated Segment Test). Although can in principle take any value, FAST uses only a value of 3 (corresponding to a circle of 16 pixels circumference), and tests show that the best results are achieved with being 9.

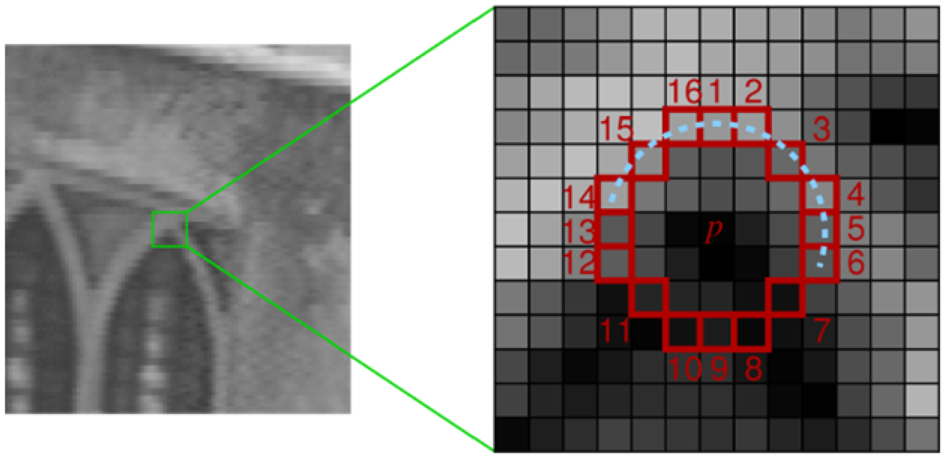


Figure 6.2: FAST Detector

Fast can be of various type fast8, fast12, fast16. Among all Fast 16 gives best performance. Above figure 6.2 show how fast16 works. In FAST detector we have to check opposite points first and check such boundary condition to see if p is feature point or not. After that score for each feature point is calculated. Here if all such points p are tested in parallel way it would really improve performance. So if image contains 200 pixels we can pass that many threads in kernels. Thus all pixels are checked in parallel way if they are keypoints or not. Therefore optimizing overall performance. Experiments were done when application was sequential and after converting it in parallel form. That's how keypoints are detected.

Results for sequential approach for FAST feature detector

CPU time 0.0166461 sec

Results for parallel approach for FAST Feature detector

Average Kernel Total time 0.000846659 sec

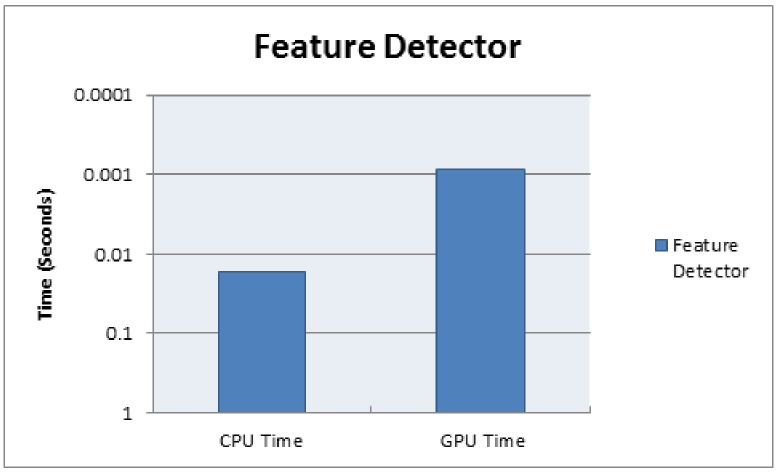


Figure 6.3: Performance Comparison for Fast Feature Detector

After Keypoints detection, it's time for Feature descriptor phase. For each keypoint they will form 16,32,64 or 128 bit descriptor. This this also can be done in parallel manner. Suppose we want to form 32 bit descriptor and there are 100 keypoints. We can calculate 32 bit descriptor for each keypoint in parallel manner passing 100 threads. Thus descriptor part also can be made parallel and the experiments were done on this.

Results for sequential approach for computing descriptor

CPU Time 0.0324986 sec

Results for parallel approach for computing descriptor

Average Kernel Total Time 0.000182724 sec

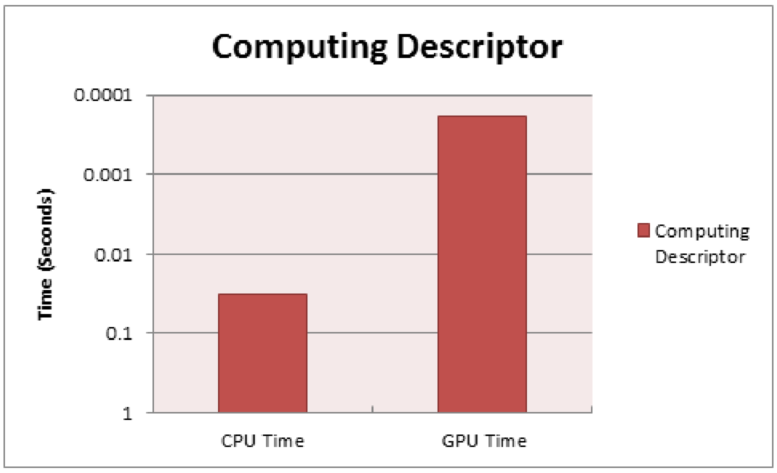


Figure 6.4: Performance Comparison for Computing Descriptor

Thus we can see that using appropriate technique we can get a good speed up.

6.1 Matching algorithms

The problem of nearest neighbor search is one of major importance in a variety of applications such as image recognition, data compression, pattern recognition, machine learning, document retrieval systems, statistics and data analysis. FLANN (Fast Library for Approximate Nearest Neighbors) is a library for performing fast approximate nearest neighbor search. To experiment with matching we have taken 3 matching algorithms namely Kd Tree, Kmeans and LSH (Locality Sensitive Hashing) [9]. Nearest neighbor search is a very interesting problem in computer vision application. The algorithm for Kdtree is as follows:

1) Training part
2) Matching Part

In training KdTree is formed using base descriptor. Now matching part matches two descriptors. In kmeans algorithm, cluster is formed and then matching is done. Both are explained briefly below. When a search is restricted to some maximum number of searched nodes, the probability of finding the true nearest neighbour increases with the increasing limit. Priority search increases search performance, compared with a tree backtracking. The problem with diminishing returns in priority search is that searches of the individual nodes in a tree are not independent, and the more searched nodes, the further away the nodes are from the node that contain the query point. To address this problem, we investigate the following strategies.

1. We create m different KD-trees each with a different structure in such a way that searches in the different trees will be (largely) independent.
2. With a limit of n nodes to be searched, we break the search into simultaneous searches among all the m trees. On the average, n/m nodes will be searched in each of the trees.

The tree building process starts with all the points in the dataset and divides them into K clusters, where K is a parameter of the algorithm, called the branching factor. The clusters are formed by first selecting K points at random as the cluster centers followed by assigning to each center the points closer to that center than to any of the other centers. The algorithm is repeated recursively for each of the resulting clusters until the number of points in each cluster is below a certain threshold (the maximum leaf size), in which case that node becomes a leaf node. In kmeans few parameter like branching plays an important role. The behavior due to change in branching factor is show in figure 6.5

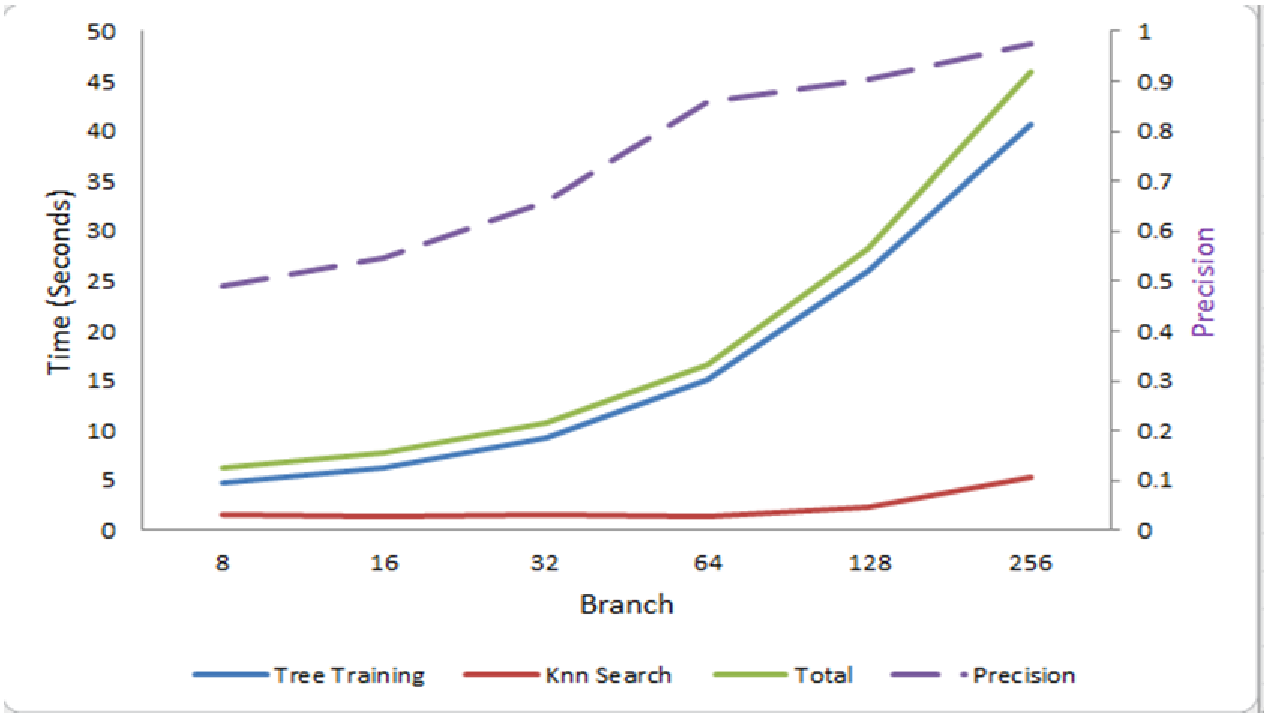


Figure 6.5: Precision and time consumed over linear search for different branching factors

Number of parallel trees to be created also plays an important role in Kmeans. The effect of variation in number of trees is shown in Figure 6.6 below

After few experimental results using different descriptor size it was proved that KDTree shows better performance compared to KMeans. Below Figure 6.7 is for 64 bit descriptor size.

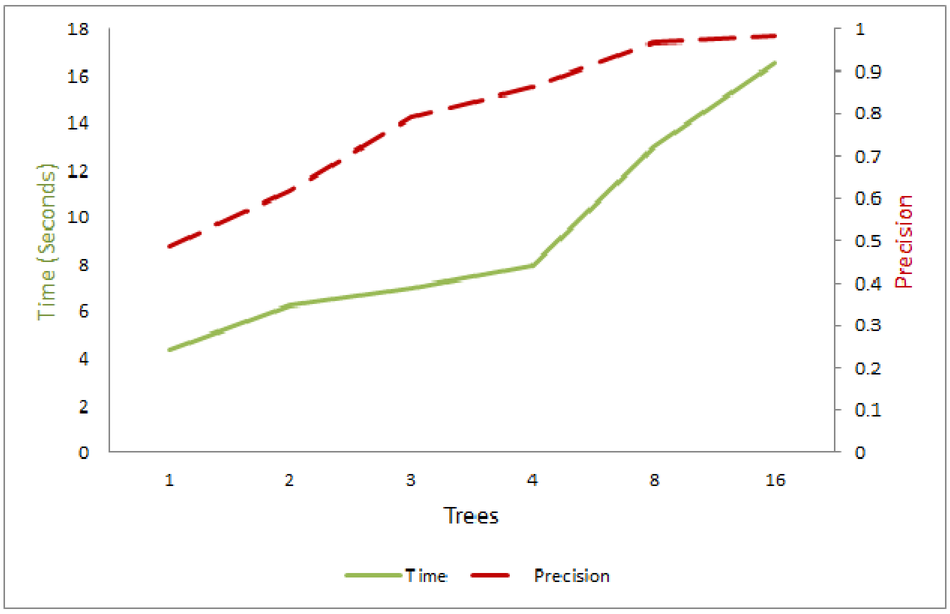


Figure 6.6: Precision and time consumed over linear search for different number of parallel randomized trees used by the index

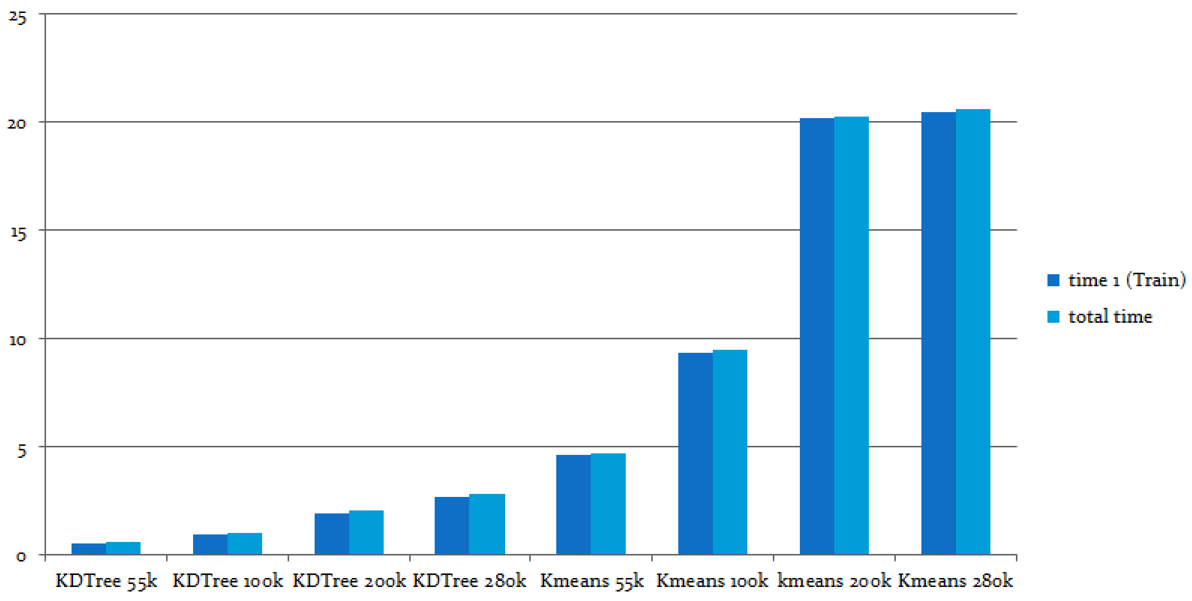


Figure 6.7: Comparison of KDTree and KMeans Index

Chapter 7

Conclusion

ArchOCL is a software model to characterize OpenCL workload. Output of this tool is the statistics which are really interesting and helps to reveal the dynamic behavior of the application. It also helps in GPU architecture design. Testing of software also plays an important role. Effective and fast results can be obtained if proper way of testing is used. OpenCL increases the performance using proper optimization techniques and it can be further improved. In matching algorithm while tuning the parameters like branching factor in KDTree improves the precision but at the cost of increased runtime. After many experiments we could conclude, in most cases KDTree perfroms better than Kmeans.

Bibliography

[1] A. Kerr, G. Diamos, and S. Yalamanchili, "Characterization and Analysis of PTX Kernels", Georgia Institute of Technology, May-2009, CERCS, GIT-CERCS-09-06

[2] "The OpenCL Specification", Version 1.2, Published by Khronos OpenCL Working Group, Aaftab Munshi (ed.), 2011 http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[3] NVIDIA, NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, 2nd ed, June 2008 http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf

[4] Sangmin Seo, Gangwon Jo and Jaejin Lee, "Performance Characterization of the NAS Parallel Benchmarks in OpenCL", School of Computer Science and Engineering, Seoul National University, Seoul, 151-744, Korea

[5] Nilanjan Goswami, Ramkumar Shankar, Madhura Joshi, and Tao Li, "Exploring GPGPU Workloads: Characterization Methodology, Analysis and Microarchitecture Evaluation Implications ," 2010 IEEE International Symposium on Workload Characterization (IISWC), pages 110. IEEE

[6] . Kerr, G. Diamos, and S. Yalamanchili, "Modeling gpu-cpu workloads and systems," in Third Workshop on General-Purpose Computation on Graphics Procesing Units, Pittsburg, PA, USA, March 2010.

[7] The Mesa 3D Graphics Library http://www.mesa3d.org

[8] http://opencv.willowgarage.com/wiki/

[9] M. Muja and D. G. Lowe, Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration, in International Conference on Computer Vision Theory and Application VISSAPP09), 2009, pp. 331340.