Workload Characterization Methodology for OpenCL Kernels

By

Hiren A. Kotadiya

09MCE019

Department of Computer Science and Engineering
Institute of Technology, Nirma University
Ahmedabad

May, 2011

Workload Characterization Methodology for OpenCL Kernels

Major Project

Submitted in partial fulfillment of the requirements
for the degree of
Master of Technology in Computer Science and Engineering

By
Hiren A. Kotadiya
09MCE019

Guided By
Mr. Samvit Kaul
Dr. S. N. Pradhan

Department of Computer Science and Engineering
Institute of Technology, Nirma University
Ahmedabad

May, 2011

Declaration

This is to certify that

i) The thesis comprises my original work towards the degree of Master of Technology in Computer Science and Engineering at Institute of Technology, Nirma University and has not been submitted elsewhere for a degree.

ii) Due acknowledgement has been made in the text to all other material used.

Hiren A. Kotadiya

Certificate

This is to certify that the Major Project entitled **"Workload Characterization Methodology for OpenCL Kernels"** submitted by **Mr. Hiren A. Kotadiya (09MCE019)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering, Institute of Technology, Nirma University, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Date: _____

Dr. S. N. Pradhan
Professor and PG-Coordinator,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad.

Mr. Samvit Kaul
Project Guide,
Intel Technology India Pvt. Ltd.,
Bangalore.

Prof. D. J. Patel
Professor and Head,
Department of Computer Engineering,
Institute of Technology,
Nirma University, Ahmedabad.

Dr. K. Kotecha
Director,
Institute of Technology,
Nirma University, Ahmedabad.

Abstract

Graphics Processing Units (GPUs) have enjoyed a dramatic increase in programmability as well as in computational power, which allowed them to be utilized as co-processors for general purpose applications. OpenCL, by the Khronos Group, is an open standard for parallel programming using CPUs, GPUs and other types of processors. The OpenCL standard offers a common API for program execution on systems composed of different types of computational devices. While 3D Graphics workload characterization is a well developed area and standard benchmarks are available, comparatively little has been devoted to the analysis and characterization of GPGPU workloads to assist future work in micro-architecture design, application re-structuring and compiler optimizations. Design goal for ArchOCL software model 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. Development of ArchOCL model includes implementation of OpenCL APIs, Compiler enhancement to support OpenCL kernel compilation and Statistics collection.

Acknowledgements

A journey is easier when you travel together. Interdependence is certainly more valuable than independence. This thesis is the result of work whereby I have been accompanied and supported by many people.

With immense pleasure I express my sincere gratitude, regards and thanks to my external guide **Mr. Samvit Kaul** for his excellent guidance and continuous encouragement at all the stages of my research work. I would like to thank my project manager **Mr. Biju P. Simon** due to his continuous help and support. The chain of my gratitude would be definitely incomplete if I would forget to thank **Mr. M Senthilnathan** who shared with me his experience for serving my thesis work. His interest and confidence in me was the reason for all the success I have made. It is my pleasure to be associated with **Intel Technology India Pvt. Ltd., Bangalore**.

I am also thankful to my internal guide **Dr. S. N. Pradhan**, PG Coordinator, Department of Computer Science and Engineering, Institute of Technology, Nirma University, Ahmedabad, for providing me all the necessary guidance throughout the term. I would like to thank **Dr. K. Kotecha,** Hon'ble Director, Institute of Technology, Nirma University, Ahmedabad for his unmentionable support, providing basic infrastructure and healthy research environment.

Last, but not the least, no words are enough to acknowledge constant support of my parents because of whom I am able to complete my dissertation work successfully.

Hiren A. Kotadiya
09MCE019

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
CU	Compute Unit
CUDA	Compute Unified Device Architecture
DSP	Digital Signal Processor
GPGPU	General-Purpose computation on Graphics Processing Units
GPU	Graphics Processing Unit
ISV	Independent Software Vendor
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
PE	Processing Element
SDK	Software Development Kit
SLM	Shared Local Memory

Chapter 1

Introduction

1.1 OpenCL and CUDA

Graphics Processing Units (GPUs) have enjoyed a dramatic increase in programmability as well as in computational power, which allowed them not only to make an impact on the game industry, but also to be utilized as coprocessors for general purpose applications. Fifteen years ago, GPUs were fixed-function hardware accelerators for the graphics pipeline rendering model defined by APIs such as OpenGL and Direct3D. Over time, GPUs evolved to incorporate progressively more flexible and programmable components, culminating in the vertex and pixel shaders on which modern graphics applications are based. As GPUs became increasingly programmable, many researchers explored ways to harness their parallel-processing potential by mapping relatively general-purpose computations onto the restricted paradigms offered by graphics APIs. [4]

Open Computing Language (OpenCL) [1] and Compute Unified Device Architecture (CUDA) [2] are two interfaces for GPU computing, both presenting similar features but through different 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 fine for students experimenting with a new approach, but mainstream ISVs and other large-scale developers need the flexibility inherent 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. OpenCL promises a portable language for GPU programming, capable of targeting very dissimilar parallel processing devices. The OpenCL standard offers a common API for program execution on systems composed of different types of computational devices such as multicore CPUs, GPUs, or other accelerators. [5]

1.2 ArchOCL - Design Goal

General-purpose application development for GPUs (GPGPU) has gained momentum as a cost-effective approach for accelerating data and compute-intensive applications. It has been driven by the introduction of C-based programming environments such as NVIDIA's CUDA, OpenCL and Intel's Ct. [8]

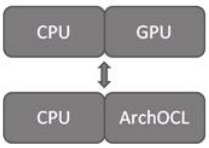

Figure 1.1: ArchOCL and GPU

While significant effort has been focused on developing and evaluating applications and software tools, comparatively little has been devoted to the analysis and characterization of applications to assist future work in compiler optimizations, application re-structuring and micro-architecture design.

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.

1.3 ArchOCL Software Architecture

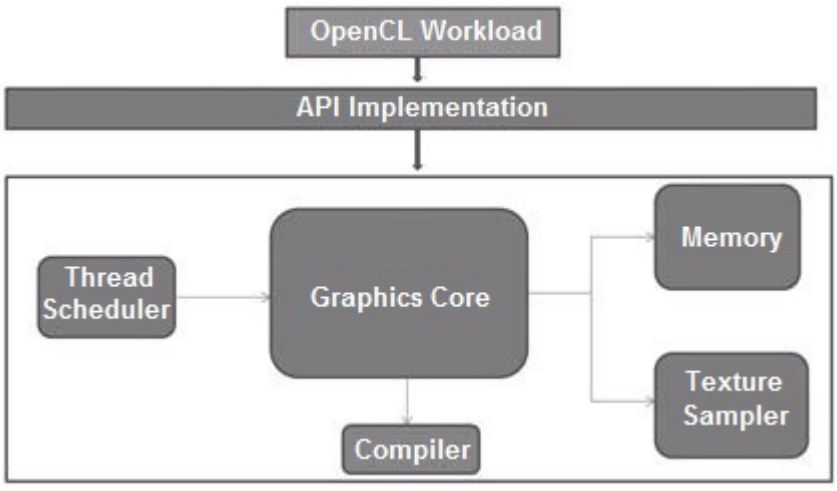

Figure 1.2: ArchOCL Architecture

1.4 Advantages

ArchOCL is useful for

- Manycore architecture research
- Designers to explore architectural design
- Designers to evaluate the impact of ways of parallelization on architectural design

1.5 Limitations

As ArchOCL is a software model, we are not considering the performance impact. It doesn't give the execution time or any other performance related statistics. ArchOCL takes more time to execute kernels compared to GPU, because it executes kernels on CPU only.

1.6 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2, *Literature survey on OpenCL includes details about GPGPU. It describes OpenCL Platform Model, Memory Model, Execution Model and Programming Model. It also includes details about ATI Stream SDK and Mesa Graphics Library.*

Chapter 3, *API Implementation includes description of OpenCL APIs for OpenCL Platform Layer and OpenCL Runtime. It also describes OpenCL APIs that allow applications to use OpenGL buffer, texture and render buffer objects as OpenCL memory objects.*

Chapter 4, *Kernel Execution describes OpenCL kernel and its execution. It also includes pattern that most OpenCL programs follow.*

Chapter 5, *Workload Characterization includes statistics provided by ArchOCL model. ArchOCL provides Number of Kernel Calls, Global and Local Work-Group Size, Memory or Compute Intensive Workload, SLM Density, Instructions Breakdown, Memory Access Spread, Average Channel Utilization, Math Functions Breakdown, Floating Point v/s Integer Operations, Memory Bank Conflict and Cache Line Hit.*

Chapter 6, *Conclusion & Future Work includes concluding remarks and scope for further enhancement.*

Chapter 2

Literature Survey

2.1 GPGPU

GPGPU stands for General-Purpose computation on Graphics Processing Units, also known as GPU Computing. Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput. Once specially designed for computer graphics and difficult to program, today's GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C. Developers who port their applications to GPUs often achieve speedups of orders of magnitude vs. optimized CPU implementations. GPGPU have a major impact on the way programmers parallelize their applications.

Up until now, GPGPU has been a research technology for early adopters - a new, promising experimental capability for scientists, engineers, financial professionals and others running compute-intensive applications. Two elements have kept GPGPU largely in the ivory tower: first, the available APIs were proprietary and second, the GPU has been treated as an independent application accelerator instead of as part of a balanced heterogeneous architecture. OpenCL is a game-changing development

in both respects.

Of course no application runs entirely on the GPU. 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 difference between GPGPU as a niche accelerator and GPGPU as a new baseline feature, ready for tomorrow's systems and applications.

2.2 OpenCL

2.2.1 OpenCL Architecture

OpenCL is an open industry standard for programming a heterogeneous collection of CPUs, GPUs and other discrete computing devices organized into a single platform. It is more than a language. OpenCL is a framework for parallel programming and includes a 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. [1]

OpenCL uses following hierarchy of models:

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

2.2.2 Platform Model

The model consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more compute units (CUs) which are further divided into one or more processing elements (PEs) . Computations on a device occur within the processing elements.

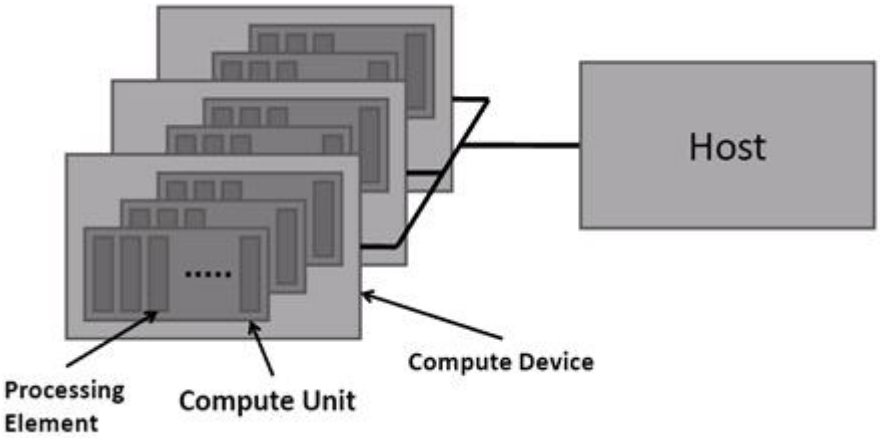

Figure 2.1: Platform model

2.2.3 Memory Model

Work-item(s) executing a kernel have access to four distinct memory regions:

Global Memory: This memory region permits read/write access to all work-items in all work-groups. Work-items can read from or write to any element of a memory object. Reads and writes to global memory may be cached depending on the capabilities of the device.

Constant Memory: A region of global memory that remains constant during the execution of a kernel. The host allocates and initializes memory objects placed into constant memory.

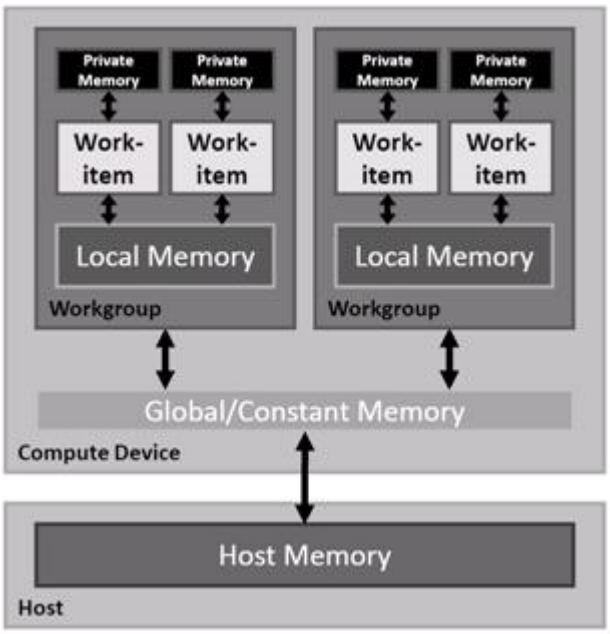

Figure 2.2: Memory Model

Local Memory: A memory region local to a work-group. This memory region can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device. Alternatively, the local memory region may be mapped onto sections of the global memory.

Private Memory: A region of memory private to a work-item. Variables defined in one work-item's private memory are not visible to another work-item.

2.2.4 Execution Model

Execution of an OpenCL program occurs in two parts: kernels that execute on one or more OpenCL devices and a host program that executes on the host. The host program defines the context for the kernels and manages their execution.

The core of the OpenCL execution model is defined by how the kernels execute.

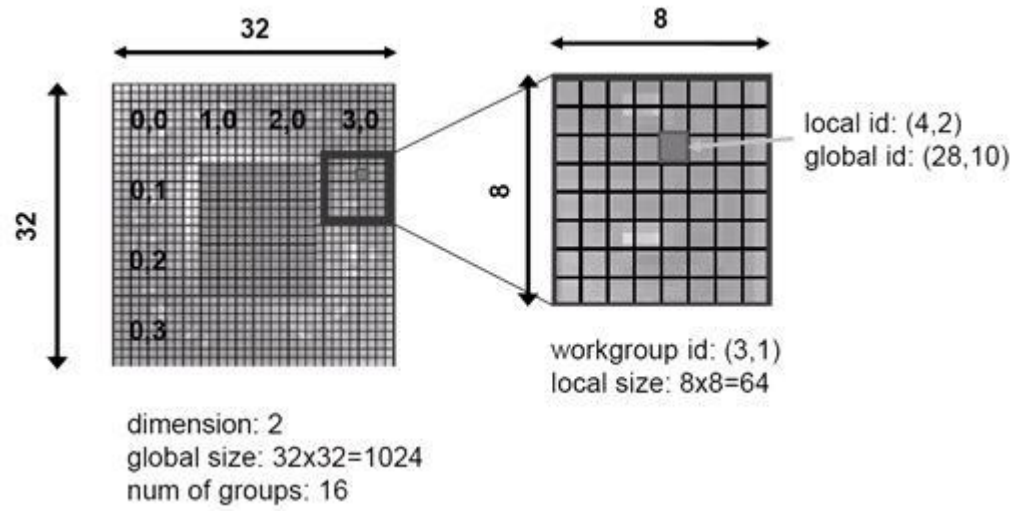

Figure 2.3: Work-item and Work-group Example

When a kernel is submitted for execution by the host, an index space is defined. An instance of the kernel executes for each point in this index space. This kernel instance is called a work-item and is identified by its point in the index space, which provides a global ID for the work-item. Each work-item executes the same code but the specific execution pathway through the code and the data operated upon can vary per work-item. Work-items are organized into work-groups. The work-groups provide a more coarse-grained decomposition of the index space. Work-groups are assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the processing elements of a single compute unit.

The index space supported in OpenCL 1.0 is called an NDRange. An NDRange is an N-dimensional index space, where N is one, two or three. An NDRange is defined by an integer array of length N specifying the extent of the index space in each dimension. Each work-item's global ID and local ID are N-dimensional tuples.

The global ID components are values in the range from zero to the number of elements in that dimension minus one. Work-groups are assigned IDs using a similar approach to that used for work-item global IDs. An array of length N defines the number of work-groups in each dimension. Work-items are assigned to a work-group and given a local ID with components in the range from zero to the size of the work-group in that dimension minus one. Hence, the combination of a work-group ID and the local-ID within a work-group uniquely defines a work-item. Each work-item is identifiable in two ways; in terms of a global index, and in terms of a work-group index plus a local index within a work group.

Context and Command Queues

The host defines a context for the execution of the kernels. The context includes the following resources:

- **Devices:** The collection of OpenCL devices to be used by the host.
- **Kernels:** The OpenCL functions that run on OpenCL devices.
- **Program Objects:** The program source and executable that implement the kernels
- **Memory Objects:** A set of memory objects visible to the host and the OpenCL devices.

The context is created and manipulated by the host using functions from the OpenCL API. The host creates a data structure called a command-queue to coordinate execution of the kernels on the devices. The host places commands into the command-queue which are then scheduled onto the devices within the context. These include:

- **Kernel execution commands:** Execute a kernel on the processing elements of a device.

- **Memory commands:** Transfer data to, from, or between memory objects, or map and unmap memory objects from the host address space.

- **Synchronization commands:** Constrain the order of execution of commands.

The command-queue schedules commands for execution on a device. These execute asynchronously between the host and the device. Commands execute relative to each other in one of two modes:

- **In-order Execution:** Commands are launched in the order they appear in the commandqueue and complete in order. In other words, a prior command on the queue completes before the following command begins. This serializes the execution order of commands in a queue.

- **Out-of-order Execution:** Commands are issued in order, but do not wait to complete before following commands execute. Any order constraints are enforced by the programmer through explicit synchronization commands.

Categories of Kernels

The OpenCL execution model supports two categories of kernels:

- **OpenCL kernels** are written with the OpenCL C programming language and compiled with the OpenCL compiler. All OpenCL implementations support OpenCL kernels.

- **Native kernels** are accessed through a host function pointer. Native kernels are queued for execution along with OpenCL kernels on a device and share memory objects with OpenCL kernels.

2.2.5 Programming Model

The OpenCL execution model supports data parallel and task parallel programming models, as well as supporting hybrids of these two models. The primary model driving the design of OpenCL is data parallel.

Data Parallel Programming Model

A data parallel programming model defines a computation in terms of a sequence of instructions applied to multiple elements of a memory object. The index space associated with the OpenCL execution model defines the work-items and how the data maps onto the work-items.

Task Parallel Programming Model

The OpenCL task parallel programming model defines a model in which a single instance of a kernel is executed independent of any index space. It is logically equivalent to executing a kernel on a compute unit with a work-group containing a single work-item.

2.2.6 OpenCL Framework

The OpenCL framework allows applications to use a host and one or more OpenCL devices as a single heterogeneous parallel computer system. The framework contains the following components:

- **OpenCL Platform Layer:** The platform layer allows the host program to discover OpenCL devices and their capabilities and to create contexts.
- **OpenCL Runtime:** The runtime allows the host program to manipulate contexts once they have been created.
- **OpenCL Compiler:** The OpenCL compiler creates program executables that contain OpenCL kernels.

2.2.7 OpenCL Summary

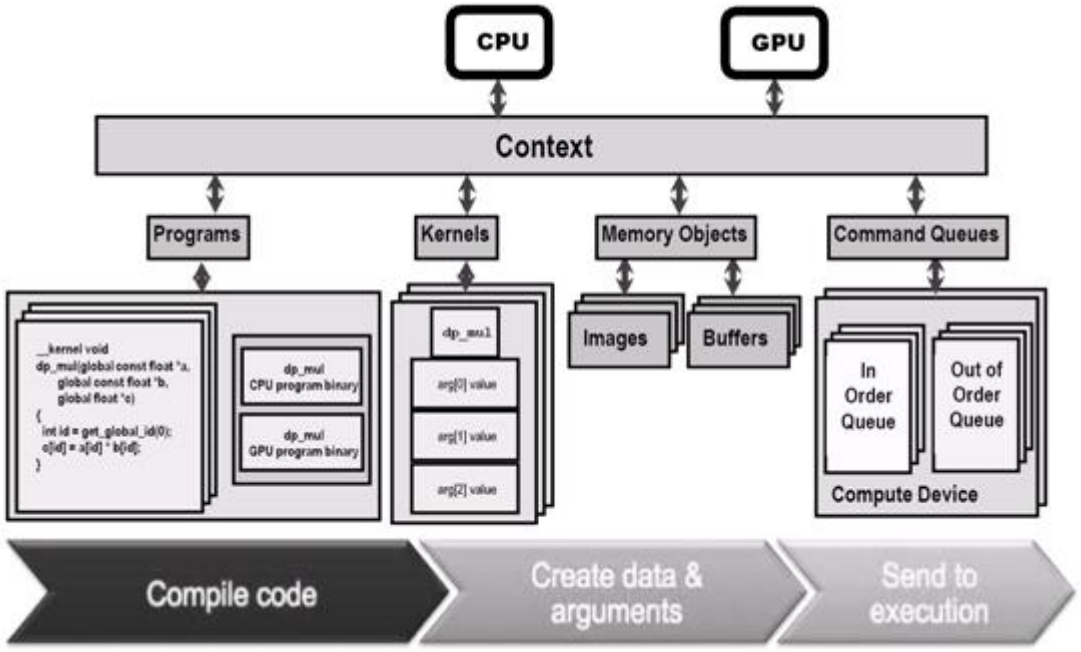

Figure 2.4: OpenCL Program Flow

2.3 ATI Stream SDK

The ATI Stream Software Development Kit (SDK) is a complete development platform created by AMD to allow developer to quickly and easily develop applications accelerated by ATI Stream technology. The SDK allows developer to develop applications in a high level language, OpenCL. [3]

For Microsoft Windows platforms, the ATI Stream SDK installer installs the following packages on your system by default.

- ATI Stream SDK Developer package. This includes:
  - the OpenCL compiler and runtime

  - developer documentation

- ATI Stream SDK Samples package. This includes:
  - sample applications
  - sample documentation
- ATI Stream Profiler package
  The ATI Stream Profiler provides a Microsoft Visual Studio integrated view of key static kernel characteristics such as workgroup dimensions, memory transfer sizes, kernel execution time.
- Stream KernelAnalyzer package
  Stream KernelAnalyzer is a tool for analyzing the performance of OpenCL for ATI graphics cards. It gives accurate performance estimates for kernels and even allows viewing disassembly of the generated hardware kernel, all without having to run the application on actual hardware.

2.4 Mesa Graphics Library

Mesa is an open-source implementation of the OpenGL specification. OpenGL is a programming library for writing interactive 3D applications. [9]

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.

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 clCreateImage{2D—3D}

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 efficient 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

Kernel Execution

4.1 An OpenCL application

A kernel can be executed as a function of multi-dimensional domains of indices. Each element is called a work-item; the total number of indices is defined as the global work-size. The global work-size can be divided into sub-domains, called work-groups, and individual work-items within a group can communicate through global or locally shared memory. Work-items are synchronized through barrier or fence operations.

An OpenCL application is built by first querying the runtime to determine which platforms are present. There can be any number of different OpenCL implementations installed on a single system. The next step is to create a context. An OpenCL context has associated with it a number of compute devices (for example, CPU or GPU devices). Within a context, OpenCL guarantees a relaxed consistency between these devices. This means that memory objects, such as buffers or images, are allocated per context; but changes made by one device are only guaranteed to be visible by another device at well-defined synchronization points. For this, OpenCL provides events, with the ability to synchronize on a given event to enforce the correct order of execution.

Many operations are performed with respect to a given context; there also are many operations that are specific to a device. For example, program compilation and kernel execution are done on a per-device basis. Performing work with a device, such as executing kernels or moving data to and from the device's local memory, is done using a corresponding command queue. A command queue is associated with a single device and a given context; all work for a specific device is done through this interface. Note that while a single command queue can be associated with only a single device, there is no limit to the number of command queues that can point to the same device. For example, it is possible to have one command queue for executing kernels and a command queue for managing data transfers between the host and the device.

Most OpenCL programs follow the same pattern. Given a specific platform, select a device or devices to create a context, allocate memory, create device-specific command queues, and perform data transfers and computations. Generally, the platform is the gateway to accessing specific devices, given these devices and a corresponding context, the application is independent of the platform. Given a context, the application can:

- Create one or more command queues.
- Create programs to run on one or more associated devices.
- Create kernels within those programs.
- Allocate memory buffers or images, either on the host or on the device(s).
- Write data to the device.
- Submit the kernel to the command queue for execution.
- Read data back to the host from the device.

4.2 The "Kernel" Concept

Kernels are closed computational functions that execute independently on the GPU core. The CPU first initiates a data transfer to the GPU, then sends a binary or text version of the kernel and finally initiates asynchronous kernel execution on the GPU. After kernel invocation, the CPU control turns to the next instruction, even though the GPU is still executing the kernel. The CPU can probe when the execution is finished and starts initiating the write-back operation of GPU results to the CPU memory. [7]

4.3 OpenCL Kernel Example: Binary Search

```
__kernel void binarySearch(
                __global uint4 * outputArray,
                __const __global uint  * sortedArray,
                unsigned int findMe,
                unsigned int globalLowerBound,
                unsigned int globalUpperBound,
                unsigned int subdivSize)
{
    unsigned int tid = get_global_id(0);
    unsigned int lowerBound = globalLowerBound + subdivSize * tid;
    unsigned int upperBound = lowerBound + subdivSize - 1;
    unsigned int lowerBoundEle = sortedArray[lowerBound];
    unsigned int upperBoundEle = sortedArray[upperBound];

    if((lowerBoundEle > findMe)||(upperBoundEle < findMe))
    {
        return;
    }
    else
    {
        outputArray[0].x = lowerBound; outputArray[0].y = upperBound;
        outputArray[0].w = 1;
    }
}
```

4.4 Partitioning the Work

In OpenCL, each kernel executes on an index point that exists in a global NDRange. The partition of the NDRange can have a significant impact on performance; thus, it is recommended that the developer explicitly specify the global (#work-groups) and local (#work-items/work-group) dimensions, rather than rely on OpenCL to set these automatically. Work-groups cannot be split across multiple compute units, so if the number of work-groups is less than the available compute units, some units are idle. Work-items in the same work-group can share data through local memory and also use high-speed local atomic operations. Thus, larger work-groups enable more work-items to efficiently share data, which can reduce the amount of slower global communication. However, larger work-groups reduce the number of global work-groups, which, for small workloads, could result in idle compute units. Generally, larger work-groups are better as long as the global range is big enough to provide 1-2 Work-Groups for each compute unit in the system; for small workloads it generally works best to reduce the work-group size in order to avoid idle compute units. [3]

4.5 Synchronization

The two domains of synchronization in OpenCL are work-items in a single workgroup and command-queue(s) in a single context. Work-group barriers enable synchronization of work-items in a work-group. Each work-item in work-group must first execute the barrier before executing any beyond the work-group barrier. Either all of, or none of, the work-items in a work-group must encounter the barrier. As currently defined in the OpenCL Specification, global synchronization is not allowed.

Chapter 5

Workload Characterization

Using ArchOCL, we executed and analyzed following ATI Stream SDK workloads.

- BinarySearch
- BinomialOption
- BitonicSort
- DCT
- DwtHaar1D
- FastWalshTransform
- FloydWarshall
- MatrixMultiplication
- MatrixTranspose
- Nbody
- PrefixSum
- RecursiveGaussian
- Reduction
- ScanLargeArrays
- SimpleConvolution
- SobelFilter

5.1 Kernel Call and Work Group Size

Probably the most effective way to exploit the potential performance of the GPU is to provide enough threads to keep the device completely busy. The programmer specifies a three-dimensional NDRange over which to execute the OpenCL kernel. ArchOCL model provides count for total number of kernels executed as well as local and global work group size used to execute those kernels.

Table 5.1: Kernel Call and WorkGroup Size

	Kernel Call	Global Size X	Global Size Y	Global Size Z	Local Size X	Local Size Y	Local Size Z
BinarySearch	1	256	1	1	256	1	1
BinomialOption	1	65280	1	1	255	1	1
BitonicSort	55	28160	55	55	14080	55	55
DCT	1	256	64	1	8	8	1
DwtHaar1D	1	512	1	1	512	1	1
FastWalshTransform	10	5120	10	10	2560	10	10
FloydWarshall	256	16777216	256	256	65536	256	256
MatrixMultiplication	1	16	16	1	8	8	1
MatrixTranspose	1	64	64	1	16	16	1
Nbody	1	1024	1	1	256	1	1
PrefixSum	1	512	1	1	512	1	1
RecursiveGaussian	4	2048	1026	4	576	66	4
Reduction	1	256	1	1	256	1	1
ScanLargeArrays	3	1538	3	3	386	3	3
SimpleConvolution	1	4096	1	1	256	1	1
SobelFilter	1	512	512	1	256	1	1

5.2 Memory or Compute Intensive Workload

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.

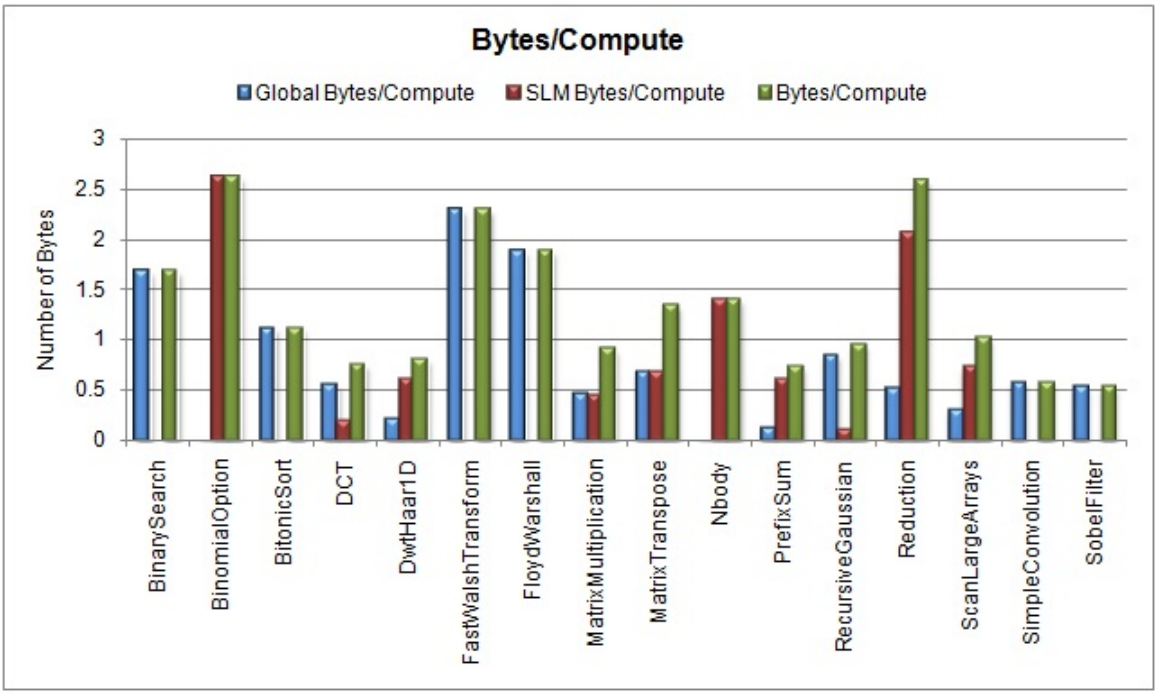

Figure 5.1: Bytes/Compute

Figure 5.1 shows that workloads like BinarySearch, BitonicSort, FastWalshTransform do not use local memory, while workloads like Nbody, PrefixSum use very less global memory. From Bytes/Compute statistics, workloads like BinomialOption, Reduction can be considered as Memory Intensive, while workloads like SimpleConvolution, SobelFilter can be considered as Compute Intensive.

5.3 SLM Density

SLM density can be calculated as ratio of Shared Local Memory data access to Global Memory data access.

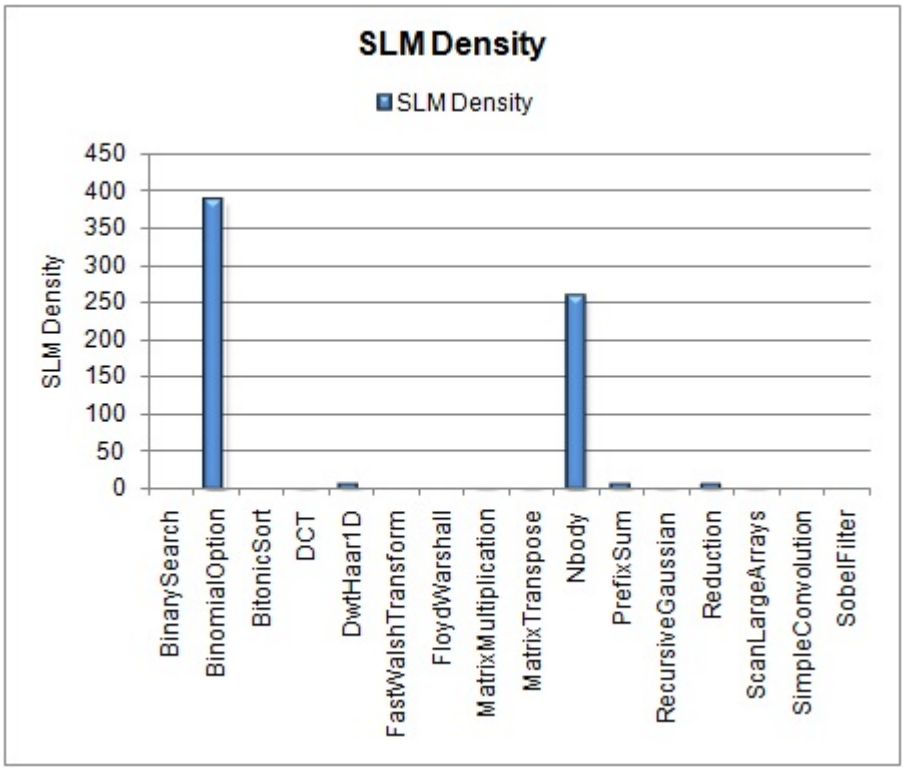

Figure 5.2: SLM Density

Figure 5.2 shows that SLM density is high for workloads BinomialOptions and NBody, which results in good performance because Shared Local Memory latency is less than the Global memory.

5.4 Instructions 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.

5.5 Memory Access Spread

ArchOCL model provides the average memory access spread for each load and store operation during SIMD execution.

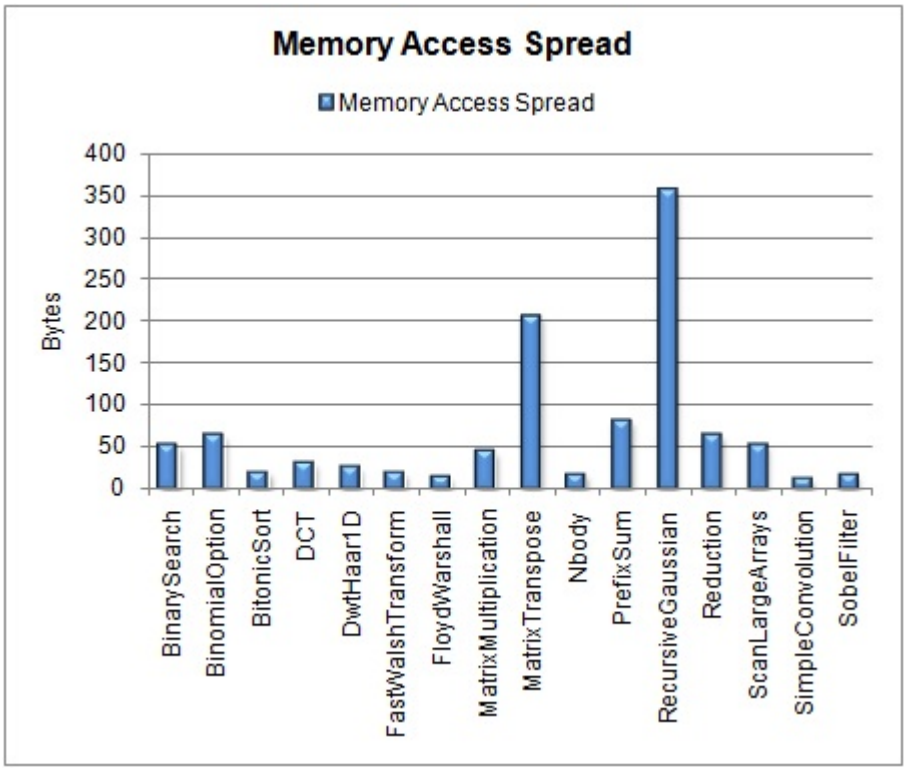

Figure 5.3: Memory Access Spread

Figure 5.3 shows that workloads like RecursiveGaussian, MatrixTranspose have large access spread.

5.6 Average Channel Utilization

ArchOCL model provides average number of active channels during each instruction execution, using which average channel utilization can be determined for different SIMD width.

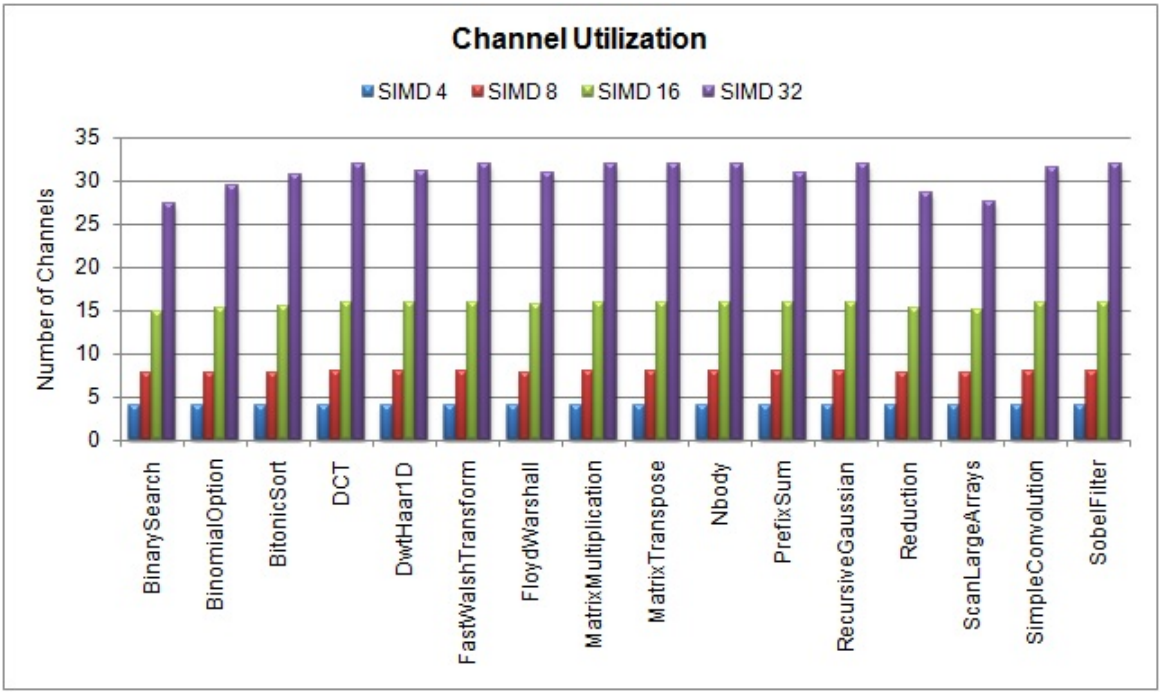

Figure 5.4: Average Channel Utilization

Figure 5.4 shows that most of the workloads behave well up to SIMD 32. For SIMD 32, minimum channel utilization is approximately 28 for BinarySearch and ScanLargeArrays. Workloads without branch and barrier show 100% channel utilization.

5.7 Math Functions Breakdown

Modern programmable GPUs have demonstrated their ability to significantly accelerate important classes of non-graphics applications; however, GPUs' substandard support for floating-point arithmetic can severely limit their usefulness for general-purpose computing. Accuracy is going to be dependent on the vendor and GPU.

Some functions are supported natively and others are emulated.

Hence total number of math function called during kernel execution is one of the important characteristics. ArchOCL model gives statistics that how many times each math function is being executed. Using this statistics, exact number of integer operations and floating point operations can be calculated.

5.8 Floating Point v/s Integer Operations

ArchOCL model provides separate count for Floating Point Operations and Integer Operations executed during kernel executions.

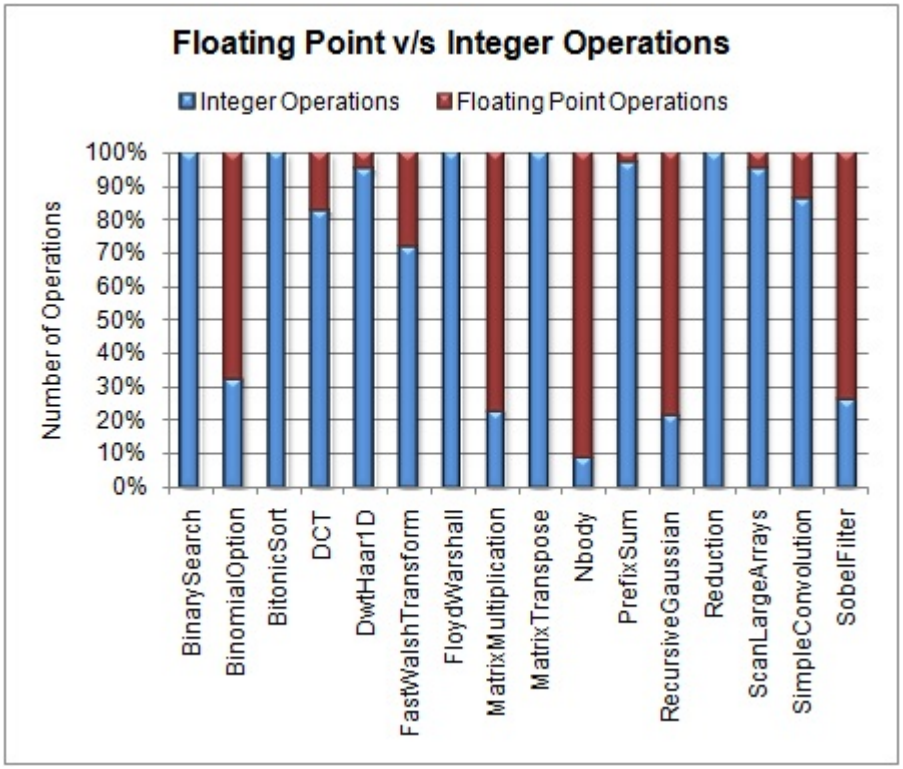

Figure 5.5: Floating Point v/s Integer Operations

Figure 5.5 shows that Workloads like BinarySearch, BitonicSort do not have floating point operations while workloads like NBody, RecursiveGaussian have more floating

point operations compared to integer operations. This result depends on the type of input data.

5.9 Memory Bank Conflict

In GPUs, the local memory is divided into memory banks. Each bank can only address one dataset at a time, so load/store from/to the same bank leads to bank conflict.

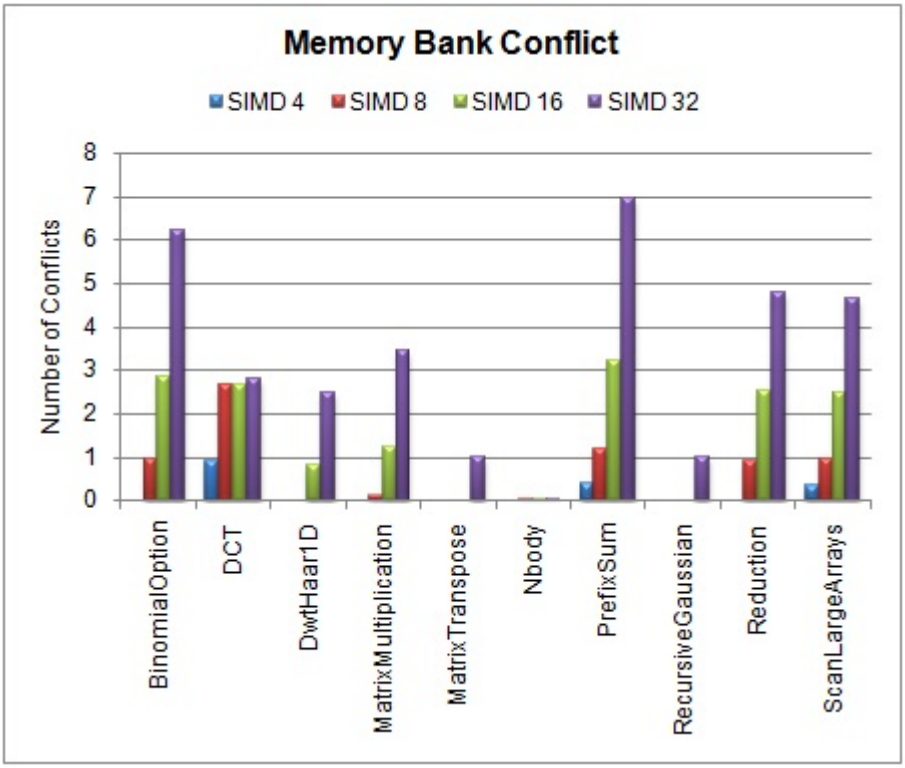

Figure 5.6: Bank Conflict

Figure 5.6 shows bank conflict with 16 banks for different SIMD width. Workloads like MatrixTranspose, RecursiveGaussian have no bank conflict for SIMD 4, 8 and 16. NBody has very less bank conflict for all SIMD width.

5.10 Cache Line Hit

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

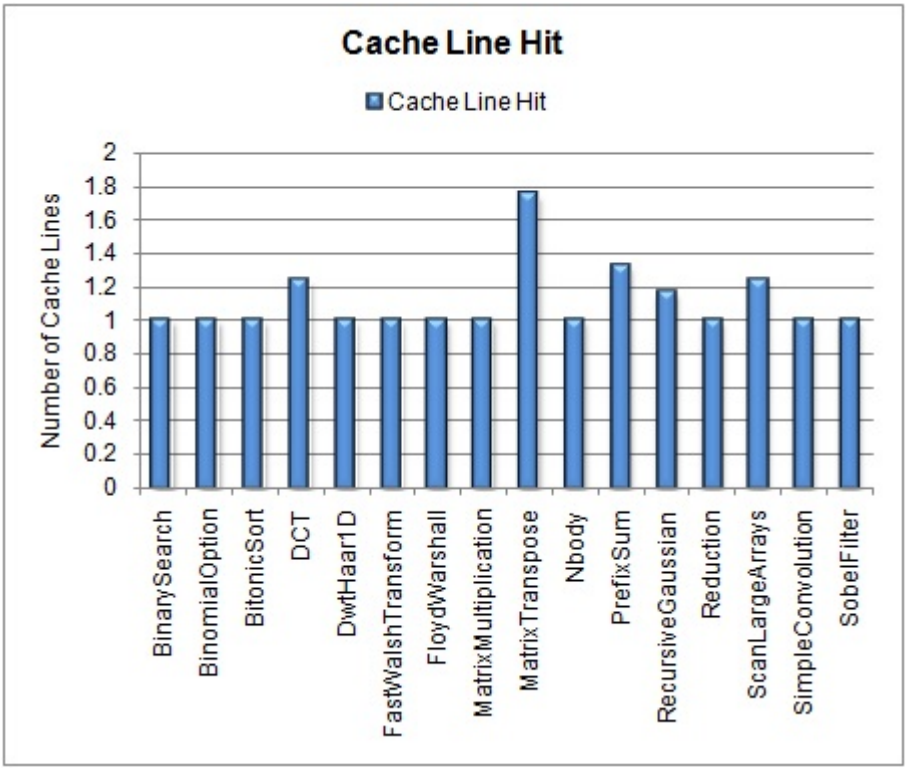

Figure 5.7: Cache Line Hit

ArchOCL model provides statistics for average number of cache line hit during kernel execution. Figure 5.7 shows average number of cache line hit with 64 bytes cache line. Workloads like MatrixTranspose, PrefixSum hit more than 1 cache line.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

ArchOCL is a software model to characterize OpenCL workload. Development of ArchOCL model includes implementation of OpenCL APIs, Compiler enhancement to support OpenCL kernel compilation and Statistics collection. ArchOCL model gives many interesting statistics that reveals the dynamic behavior of the OpenCL application. Statistics collected by ArchOCL is useful for GPU architecture design. It also helps developers to effectively port their applications in OpenCL.

6.2 Future Work

- **Upgrade ArchOCL Model for OpenCL 1.1**
  Current ArchOCL model supports OpenCL 1.0. OpenCL 1.1 has been released by Khronos Group. OpenCL 1.1 contains features like new data types including 3-component vectors and additional formats, enhanced use of events to drive and control command execution, additional OpenCL C built-in functions such as integer clamp. Future plan is to upgrade ArchOCL model for OpenCL 1.1.

References

[1] *"The OpenCL Specification"*, Version 1.0, Published by Khronos OpenCL Working Group, Aaftab Munshi (ed.), 2009
`http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf`

[2] 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`

[3] Programming Guide V2.2, ATI Stream Computing, OpenCL, August 2010
`http://www.amddevcentral.com/gpu/ATIStreamSDK/pages/Documentation.aspx`

[4] J. Cohen and M. Garland, *"Solving Computational Problems with GPU Computing"* Computing in Science & Eng., vol. 11, no. 5, 2009, pp. 58-63.

[5] John E. Stone, David Gohara, and Guochun Shi, *"OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems"* Computing in Science & Eng., vol. 12, no. 3, 2010, pp. 66-73.

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

[7] Amr Bayoumi, Michael Chu, Yasser Hanafy, Patricia Harrell, and Gamal Refai Ahmed, *"Scientific and Engineering Computing Using ATI Stream Technology"* Computing in Science & Eng., vol. 11, no. 6, 2009, pp. 92-97.

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

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

Index