Accelerating Java on Embedded GPU

Iype P. Joseph

A Thesis submitted to the Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

MASTER OF APPLIED SCIENCE

in Electrical and Computer Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering
University of Ottawa
Ottawa, Canada

January 2014

Abstract

Multicore CPUs (Central Processing Units) and GPUs (Graphics Processing Units) are omnipresent in today's market-leading smartphones and tablets. With CPUs and GPUs getting more complex, maximizing hardware utilization is becoming problematic. The challenges faced in GPGPU (General Purpose computing using GPU) computing on embedded platforms are different from their desktop counterparts due to their memory and computational limitations. This thesis evaluates the performance and energy efficiency achieved by offloading Java applications to an embedded GPU. The existing solutions in literature address various techniques and benefits of offloading Java on desktop or server grade GPUs and not on embedded GPUs. Our research is focussed on providing a framework for accelerating Java programs on embedded GPUs. Our experiments were conducted on a Freescale i.MX6Q SabreLite board which encompasses a quad-core ARM Cortex A9 CPU and a Vivante GC 2000 GPU that supports the OpenCL 1.1 Embedded Profile. We successfully accelerated Java code and reduced energy consumption by employing two approaches, namely JNI-OpenCL, and JOCL, which is a popular Java-binding for OpenCL. These approaches can be easily implemented on other platforms by embedded Java programmers to exploit the computational power of GPUs. Our results show up to an 8 times increase in performance efficiency and 3 times decrease in energy consumption compared to the embedded CPU-only execution of Java program. To the best of our knowledge, this is the first work done on accelerating Java on an embedded GPU.

Acknowledgement

This project and thesis would not have been possible without the help and support of kind people around me. And hence, this document would be incomplete without expressing my gratitude to them.

My parents, Joseph and Beena, and my brothers, Toms and Charles, have always provided me their unequivocal support for pursuing my dreams and I owe my deepest gratitude to them.

This thesis would not have been possible without the invaluable advice, guidance and support from my supervisors Dr. Miodrag Bolic and Dr. Voicu Groza. I have greatly benefited from their insightful comments and suggestions. I would like to use this opportunity to extend my heartfelt gratitude to them.

I would like to show my greatest appreciation to my colleagues at Computer Architecture Research Group (CARG), Dr. Amir Rajabzadeh, Jonathan Parri, Yu Wang, and Wei Wang, for their advice and support. I have greatly benefited from the facilities provided by CARG, University of Ottawa, and have had the support and encouragement from professors at Carleton University and University of Ottawa. I would like to appreciate and thank their efforts.

I would like to acknowledge the financial support provided by the Natural Sciences and Engineering Research Council of Canada (NSERC), YOUi Labs Inc., and my supervisors.

I am particularly grateful for the trainings and technical support I received from CMC Microsystems.

I would like to offer my special thanks to my homestay hosts, Yagu, James and May, for the friendship and warmth I have received.

I owe a very important debt to all my gurus from grade school to graduate school, particularly, Prof. Rajesh Kannan Megalingam who sowed the seeds of research in me.

Last but not the least, I would like to express my gratitude to all my friends, especially, Aseem, Balaji, Breeson, Dave, Kate, Kylea, Manosilah, Muzammil, Parthasarathy, Radhika, Rakesh, Richa, Rijwal, Robin, Sijo, Soumi, and Visakh.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviation	Meaning
CPU	Central Processing Unit
GPU	Graphics Processing Unit
JVM	Java Virtual Machine
JNI	Java Native Interface
JNA	Java Native Access
NUI	Natural User Interface
JPEG	Joint Photographic Experts Group
GPGPU	General Purpose Computing using GPU
POSIX	Portable Operating System Interface
JIT	Just In Time Compiler
TCL	Tool Command Language
VM	Virtual Machine
CUDA	Compute Unified Device Architecture
AMD	Advanced Micro Devices
PCI	Peripheral Component Interconnect
SM	Streaming Multiprocessor
CU	Compute Unit
FPGA	Field-Programmable Gate Array
DSP	Digital Signal Processor
SIMD	Single Instruction, Multiple Data
HPC	High Performance Computing
API	Application Programming Interface
DCT	Discrete Cosine Transform
SP	Stream Processor
PC	Program Counter
RAW	Read-After-Write
WAW	Write-After-Write
ULP	Unit of Least Precision
NaN	Not a Number
SoC	System on Chip
GDDR	Graphics Double Data Rate memory
DRAM	Dynamic Random Access Memory
BSOP	Black-Scholes Option Pricing
MM	Matrix Multiply
NBODY	N-Body Simulation
CP	Columbic Potential
SAD	Sum of Absolute Differences
TPACF	Two Point Angular Correlation Function
RPES	Rys Polynomial Equation Solver
MRI-Q	Magnetic Resonance Imaging Q
MRI-FHD	Magnetic Resonance Imaging FHD

MCS	Monte-Carlo Simulation
FFT	Fast Fourier Transform
HPEC	High Performance Embedded Computing
TDFIR	Time-Domain Finite Impulse Response filtering
FDFIR	Frequency-Domain Finite Impulse Response filtering
CT	Corner Turn via matrix operations
PM	Pattern Matching
CFAR	Constant False-Alarm Rate Detection
GA	Genetic Algorithm
QR	QR Factorization
SVD	Singular Value Decomposition
DB	Database operations
CUBLAS	CUDA Basic Linear Algebra Subprograms
CUFFT	CUDA Fast Fourier Transform
CUDPP	CUDA Data Parallel Primitives
CURAND	CUDA Random number generator
CUSPARSE	CUDA Sparse matrix
SHOC	Scalable HeterOgeneous Computing
APU	Accelerated Processing Unit
HSA	Heterogeneous System Architecture
APP	Accelerated Parallel Processing
SDK	Software Development Kit
LLVM IR	Low Level Virtual Machine Intermediate Representation

Chapter 1 : Introduction and Motivation

This chapter introduces a reader to the objectives of our work, its benefits, and the milestones achieved by the research. It also explains and justifies motivation of the work. The chapter also mentions the contributions of our research and ends by explaining the organization of this thesis document.

i. Introduction

Today's embedded world of high performance smartphones and tablets encompass multicore CPUs and GPUs in their SoC. However, these embedded GPUs are only used to render graphical tasks and not utilized to solve general purpose problems. Using GPUs for general purpose computing is common in desktop and High Performance Computing domains [1]–[17]. Today, GPUs are considered as cost-effective solutions for processing large quantity of data. Their excellent floating point performance and low energy consumption made GPUs popular in supercomputing domain also [18]. The Java programming language is widely used in devices such as smartphones and tablets that have embedded GPUs. Though GPGPU from Java is popular in desktop and server domains [1]–[6], [19], [20], not much research is done in exploiting embedded GPUs for solving general purpose problems. Our research is to evaluate the potential of embedded GPGPU computing from Java by offloading compute-intensive portions of Java programs on an embedded GPU.

The objective of this research is to explore the possibilities of offloading compute-intensive portions of a Java programs on embedded GPU, and to quantitatively analyze the performance and energy gains achieved by doing so. In this research, first we profile the Java programs to identify the compute-intensive and data independent portions. Once these snippets with the potential for acceleration are identified, they are offloaded onto the embedded GPU to see if acceleration is possible. The approach to offload Java programs on embedded GPU could be done by calling the OpenCL host code form Java using Java Native Interface (JNI) or by using existing solutions in literature for Java GPGPU. These solutions include using Java-bindings, which provide wrapper APIs for GPGPU programming

languages such as OpenCL or CUDA, or using user-friendly solutions that abstract the GPU APIs and translate the user program into CUDA or OpenCL program in the background.

In this research, we use JNI-OpenCL, an approach that calls OpenCL host code from Java using JNI, and JOCL, a Java-binding, to achieve acceleration on embedded GPU. The experiments are conducted on a Freescale i.MX6Q SabreLite board, which has a quad-core ARM Cortex A9 CPU, and a Vivante GPU that supports OpenCL 1.1 Embedded Profile. By offloading compute-intensive portions of Java programs on embedded GPU, we achieved a speedup up to 8 times compared to the traditional Java program. The GPU accelerated Java programs were energy efficient and consumed only one-third the energy of their CPU implementations. In this research we prove that performance and energy efficiency can be achieved by enabling Java GPGPU on embedded platforms and this is our major contribution.

ii. Motivation

Multicore CPUs and GPUs are becoming common in today's embedded world of tablets and smartphones. With CPUs and GPUs getting more complex, maximizing hardware utilization is becoming problematic. This is especially true for embedded GPUs due to the lack of interfaces to program them directly. This has resulted in embedded GPUs being used only for graphical rendering tasks. Their computing power cannot be harnessed to solve other general problems like in the High-Performance Computing (HPC), commodity Server Desktop domains. Several studies and solutions [1]–[17] are found in the literature that explain the performance and energy efficiency achieved by employing GPUs for solving general purpose problems. GPUs were initially developed to perform graphics processing and video rendering. Due to commodity GPU standardization (a unified graphics pipeline), these accelerators are now used in a wide range of products including supercomputers, cars, smartphones, tablets, etc. Today, GPUs are highly sought after for their massive parallel processing power.

OpenCL is a royalty free standard for programming heterogeneous systems. It leverages the computing power of GPUs to provide massive speed-up for non-graphical tasks. OpenCL is supported on GPUs, CPUs, FPGAs and other platforms containing SIMD processing

elements. Interpreted languages, like Java, require a virtual machine to make decisions on how to run a program given available hardware. Java is the most ubiquitous programming language used in industry and academia due to its ease of use, platform independence, object-oriented approach, built-in network and multithreading support, automatic memory management, etc. Its platform independent nature allows developers to implement an application once, and run it anywhere without rewriting or recompilation. Java uses a virtual machine to make decisions on how to run a program, given available hardware. These mappings and optimizations are limited with regards to which hardware they run on. This results in most of the advantageous architectural features being underutilized.

Today, over three billion mobile phones run on Java [21]. Accelerating Java on embedded accelerators such as GPUs could give a competitive advantage to vendors [22]. This acceleration should require no hardware changes and use existing embedded GPUs and hence the speed-up and energy saving can be achieved at no additional cost. The major motivation of this work is the success of accelerating Java programs on desktop and server GPU cards [1]–[6]. Desktop and server GPUs use thousands of cores to accelerate a program. This is not the case with embedded GPUs that typically contain tens of GPU cores. Applications using embedded GPUs have to overcome certain limitations such as, size, battery-driven power, limited memory, lower clock frequencies and limited processor instruction sets [23]. Other major differences include the fact that embedded GPUs have only small and very limited memory available compared to desktop GPUs. Also, the memory latency of embedded GPUs is greater due to small or non-existent dedicated memory and thus having to use main memory over the system bus [24]. These limitations have a profound impact on the programming model and realized performance of the embedded GPU.

Power is a critical factor for embedded devices. There is an increased demand in the industry for high performance and low power consuming embedded devices. Because of this reality, any speed-up gained by employing a GPU must be balanced against power consumption. A solution to tackle the energy consumption by today's power hungry tablets and smartphones is inevitable. Our study exposes the energy savings that can be achieved by offloading compute-intensive, data independent and parallel portions of the Java programs onto an embedded GPU. Thus, in this research, we answer the question of how acceleration of

Java programs can be done on embedded GPUs by achieving better energy efficiency while considering the above characteristics of embedded computing.

During this project we collaborated with an Ottawa-based company which provides Natural User Interface (NUI) solutions and they have identified multiple scenarios where GPUs could be useful such as: particle system with collision detection and motion calculations, sorting data, compression and decompression of content stores, complex animations schemes, simulation of physical properties for objects, advanced font rendering, streaming NUI content, JPEG encoding or decoding and encryption or decryption of secure data. The GPU implementation is not trivial and should account for the factors such as bandwidth sharing, power consumption and code complexity. The major limitation to GPU acceleration was the absence of OpenCL on embedded platforms, which leads to increased complexity and cumbersome mapping of algorithms to OpenGL and OpenVG. This limitation was lifted by the introduction of Freescale's i.MX6 triple-play platform. With OpenCL available, proper implementations of OpenCL-accelerated libraries can be researched and integrated to transparently accelerate applications.

This research work is a result of industry's need to accelerate Java applications on embedded platforms that contain GPUs. The challenges faced in GPGPU computing on embedded platforms are different from their desktop counterparts due to the memory and processing element limitations. The solution is to build OpenCL versions of underlying libraries to transparently accelerate different features required by applications. The implementations must not result in any reduction of performance, in any circumstances, compared to the traditional CPU-only version. Proper dispatching decisions must be made in order not to limit the bandwidth available to other applications and external devices.

Our research presents an opportunity for companies in smartphone and tablet industry to exploit recent advances with GPGPU in embedded systems, solidifying their positions as innovators and market leaders. Not only supporting current products, the underlying libraries could themselves be sold as a product to other companies to facilitate the exploitation of heterogeneous multi-core platforms.

We employ two methodologies, JNI-OpenCL and JOCL, in this research to analyze the potential of accelerating Java programs on embedded GPUs. JOCL is also used in comparing

the performance between acceleration from Java and acceleration from native language like C. We also use JOCL in this research to analyze the energy savings achieved by accelerating Java programs on embedded GPU. JNI-OpenCL is the method of invoking OpenCL C host-code from Java using JNI and JOCL is a Java-binding for OpenCL that provide wrapper APIs to enable OpenCL programming in Java. JNI-OpenCL and JOCL are explained in detail in Chapter 5.

iii. Objectives and Contributions

Following are the major objectives of our research.

1. Analyze the feasibility of embedded GPGPU from Java.
2. Analyze the literature and identify Java GPGPU solutions that can be used on embedded platforms.
3. Explore and see if we can accelerate Java programs using embedded GPU.
4. Explore and see if energy savings are possible while accelerating Java programs on embedded GPU.
5. Analyze and compare the achieved results with native GPGPU implementations.

Following are the major contributions of our work along with steps to validate the research questions.

1. Analyzed the feasibility of accelerating Java programs on embedded GPU and established the proof of concept.
2. Analyzed the literature and suggested candidate Java GPGPU solutions that can be ported onto embedded platforms.
3. Ported JOCL, a Java-binding to embedded platform.
4. Accelerating Java on embedded GPU.
5. Compared the acceleration achieved by using GPUs from Java with native GPGPU implementations. Results show similar performance for Java-bindings and native GPGPU solutions for large input sizes.

6. Analyzed the energy savings that can be achieved by enabling embedded GPGPU from Java. Results show 3 times reduction of energy consumption while accelerating Java programs on embedded GPU.
7. DCT, a module in the popular JPEG encoding library, is accelerated on embedded GPU while achieving energy efficiency.

iv. Organization of Thesis

This thesis is organized in seven chapters. Chapter 1 provides an outline of the research to the reader. It also presents the objectives of the work, its benefits, motivation, and the milestones achieved by the research. Chapter 2 describes briefly the technologies used in the work such as, Java, JNI, GPU, GPGPU and OpenCL. Chapter 3 analyzes the existing solutions in the field of GPGPU, embedded GPGPU and programming GPGPU from Java. Metrics to analyze and compare various solutions are also developed in this chapter. The developed metrics are used in identifying a suitable solution that will be employed in our research. Chapter 4 defines the primary and secondary research questions that are to be addressed by the work. It also explains the relevance and benefits of the research. Chapter 5 explains the methodologies employed in this research to answer the research questions. It also introduces the benchmarks used in this work. Chapter 6 analyzes the energy and performance efficiency achieved from experiments. Chapter 7 concludes the work by giving a summary of the research along with its contributions, challenges faced, and future directions. The document ends with References and Appendices.

Chapter 2 : Background

This chapter presents the technologies used in our research. It introduces the reader to Java, Java Native Interface (JNI), GPU, General Purpose Computing using GPU (GPGPU), and OpenCL programming.

i. Introduction to Java

In the recent past, we have seen the growth of various hardware architectures, with each of them supporting different operating systems and each operating with one or more graphical user interfaces. These advancements in technology have been and still are a hurdle for traditional programming languages such as C or C++. Java is an object-oriented programming language that helps the developer to solve many of these challenges faced by the programmer [25]. Java is an interpreted programming language and it eliminates the compile-link-load-test-crash-debug cycle and speeds up the development cycle [25]. The major advantage that attracts millions of programmers around the globe to use Java is its portability. Java is platform independent and once written, an application can be ported to multiple platforms with multiple operating systems and multiple hardware architectures without any modification or recompilation. Also, graphical applications can be easily built using Java with the help of the built-in graphic packages. High performance applications written in Java can exploit the processing power of the hardware by launching multiple concurrent threads with the help of the multithreading feature built into the Java programming language and runtime platform. Java enables the development of secure, highly robust, and high performance applications on multiple platforms in heterogeneous, distributed networks [25]. Its compile-time error checking followed by a second level of run-time checking increase the robustness of the applications developed using Java.

Another appealing feature of Java is its simple, yet efficient memory management that eliminates explicit programmer-defined pointer data types and pointer arithmetic, and including automatic garbage collection. It also replaces the C style ‘malloc’ and ‘free’ routines with a ‘new’ operator which allocates memory for objects. However, there is no

explicit ‘free’ function as the runtime keeps track of the allocated object’s status and it automatically reclaims the memory for future use once the objects are no longer in use. This eliminates the myriads of memory management errors that could occur in C or C++ programming. This memory management by the Java runtime environment also makes applications written in Java robust [25]. The security features incorporated into the language and run-time system helps to build secure Java applications that can work in a distributed environment [25].

The Java compiler generates bytecode, which is an architecture-neutral intermediate format designed to transport Java programs onto multiple software and hardware platforms [25]. This generated bytecode is executed on an architecture-neutral and portable language platform called Java Virtual Machine (JVM). The JVM is primarily based on the POSIX interface specification, which is an industry standard for implementing portable system interface [25]. Portability is also addressed in the syntactic level of a Java program by defining a standard behaviour for the primitive data types and their behaviour to arithmetic operations across all platforms. Java achieves superior performance by making the interpreter to execute bytecode without needing to check the run-time environment. The Java compiler has a strict compile-time static error checking mechanism while the language and run-time systems have dynamic linking stages. This is achieved by linking the classes only when they are needed and also by allowing to link new code modules on demand from various sources, even across a network [25].

For a language to be considered object-oriented, it should support certain features such as, encapsulation, polymorphism, inheritance and dynamic binding. The encapsulation feature supports information hiding, modularity and abstraction from user. Polymorphism helps to send the same message to different objects and the behaviour of the message is determined by the nature of the object that receives the message. Inheritance is the feature that helps the programmer to define a new class based on attributes and behaviors from existing classes to facilitate code reuse and organization. Dynamic binding in Java provides maximum flexibility for the applications developed. It allows the developer to send messages to objects without knowing their type at the time of writing or compiling the code and could deal with the objects coming from anywhere, even from a different network [25]. Java meets all these

requirements and it also adds a considerable amount of runtime support to ease the work of a software developer.

Though applications developed using Java can execute at high speed, there are situations in which higher performance is needed. High Performance Computing and cloud computing are a few examples of this. In such situations, Java bytecode can be translated into machine code on the fly at run time with the help of a Just-In-Time (JIT) compiler. Thus, translated bytecode could achieve roughly the same performance as native C or C++ code [25]. The performance of Java programs is much better than the fully interpreted high-level languages like TCL, Perl, etc. However its performance is lesser than the low-level compiled languages like C or C++ that operate on bare metal and delivers high performance. Java is said to be at a middle ground between very high-level and portable but extremely slow scripting languages and the non-portable, high-performance low-level languages [25]. This attribute of incorporating the desired features of both the worlds make Java one of the popular languages among developers.

a) Introduction to Java Native Interface (JNI)

Java Native Interface (JNI) is a native programming interface that allows Java programs running on the Java Virtual Machine to call libraries and programs written in native programming languages such as C, C++ or assembly. Though applications can be entirely written in Java, there are situations in which we need to use the native code, for instance, certain platform-dependant features might not be supported by the standard Java library and the developers have to resort to native code. Another scenario is when the developer needs to reuse the already existing native language libraries in Java programs. JNI will help to invoke these libraries from Java and they can be easily made available in Java applications without needing to rewrite them. JNI also facilitates the implementation of time-critical portions of the application using assembly or native languages [26]. In our research, we exploit this feature of the JVM to invoke OpenCL programs. With JNI, we can create, update and inspect Java objects including arrays and strings. It also facilitates invoking Java methods from native code and loading classes to obtain their information. JNI also helps to use native methods for throwing and catching exceptions, and in performing runtime type checking [26].

A native program access the Java Virtual Machine features by calling JNI functions. These functions are made available in the native code through a pointer to an array of pointers called an interface pointer [26]. An interface pointer points to an array of pointers, and each element in this array is a pointer that points to an interface function. Figure 2.1 shows how an interface pointer is organized. The usage of interface table separates the JNI namespace from the native code and allows the VM to provide multiple versions of the JNI function tables [26]. The JNI interface pointer is valid only in the current thread and hence, a native method must not pass the interface pointer from one thread to another [26].

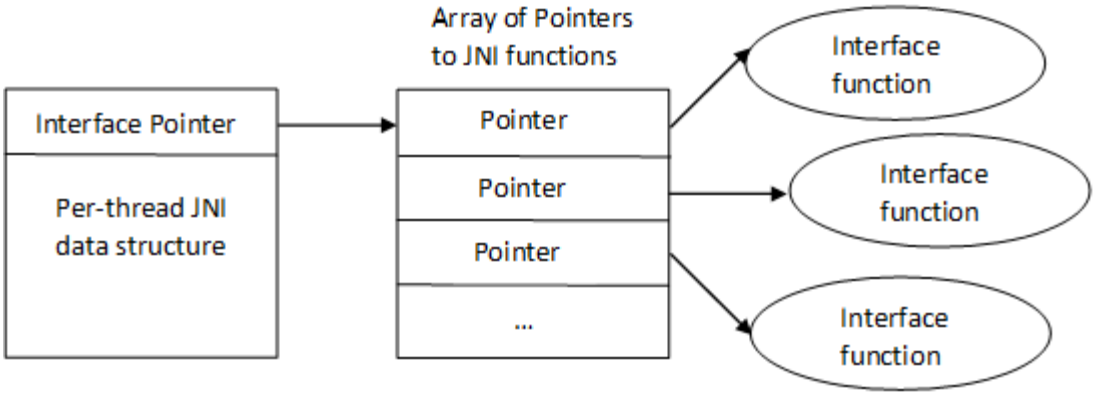

Figure 2.1: Interface Pointer [26]

Once the native method is written, it should be compiled and linked to the Java application. This linking can be done dynamically or statically. The native methods are loaded into the Java program using 'System.loadLibrary' method. If in the case of the underlying operating system not supporting dynamic linking, the native methods must be prelinked with the VM and the 'System.loadLibrary' call is completed without actually loading the library [26]. The VM resolves the native method names by looking for a name match for the methods in the native library. This name matching is done by first looking for the native method with the same short name which is the method name without the argument signature. Then the VM looks for the long name which is the method name including the argument signature [26]. This way, a programmer can overload native methods inside the same library. The first argument to the native method is the JNI interface pointer which is of the type JNIEnv. The second argument for a non-static native method is a reference to the object, whereas for a static method this argument is a reference to the Java class [26]. The remaining arguments for the native method are the regular arguments passed from the Java program.

ii. Introduction to GPU and GPGPU

Graphics Processing Units (GPUs) were originally developed to perform graphics processing and video rendering. Early GPUs employed fixed graphics pipeline to perform their tasks and over successive hardware generations they evolved into programmable cores [27]. These programmable cores facilitate General Purpose computing using GPU (GPGPU). The first GPGPU interfaces introduced were limited to shader languages like OpenGL, DirectX and Cg. Even with the introduction of these shader languages, programmers could express applications only as operations in graphics pipeline [27]. This limitation was solved with the introduction of Brook+ and Sh, which are stream extensions to C language to abstract the graphics pipeline operations [27]. The GPGPU domain was revolutionized with the advent of a new combined software and hardware architecture by NVidia called Compute Unified Device Architecture (CUDA) for their G80 family. Later, with the introduction of other languages such as OpenCL, OpenACC, Microsoft C++ AMP, etc., the term heterogeneous computing subsumed GPGPU [27].

Typically, a GPU is connected to the host processor via PCI Express interconnect as shown in Figure 2.2 and this communication bandwidth becomes the bottleneck in offloading programs to GPU. Offloading programs on GPU can provide benefits only if the acceleration achieved can outweigh the communication bottleneck. A general GPU architecture consists of several Streaming Multiprocessors (SMs) that are connected to the GPU main memory or DRAM. While NVidia uses the term Streaming Multiprocessors (SMs), AMD calls these processing elements Compute Units (CUs). Each SM has many SIMD elements called Stream Processors (SPs) which facilitates multithreaded execution in GPU. These are similar to the thread processors shown in Figure 2.2. In SIMD, a single instruction is fetched, decoded, and scheduled for multiple data operands. This helps GPUs to perform the same operation on multiple data elements at the same time. This allows GPUs to provide high floating-point bandwidth, which is very much desired in both graphics and general-purpose computing [27]. Besides SIMD processing elements, GPU architectures also employ hardware multithreading. This hardware-based context switching helps GPUs to tolerate long memory and operation latencies [27].

In conventional CPU systems, the thread or context switching is expensive as it requires storing all the program states such as Program Counter (PC), registers, stack information, etc. in the memory by the operating system. However, in the case of GPUs, this thread switching overhead is much lower due to the native hardware support [27]. To support hardware multithreading, modern GPU architectures have a large number of PC registers, memory operation buffers, and large register files. The large register file can accommodate a large number of active threads and it also reduces the cost of context switch between threads compared to the conventional CPU architectures [27]. As shown in Figure 2.2, a typical GPU accelerator is connected to the host through the DMA channel. The host processor initiates execution of programs on the GPU using a GPGPU or shader programming framework. The instructions are fetched, decoded and executed on the thread processors. A group of thread processors (SM for NVidia and CU for AMD) share a local memory space and all the SMs or CUs share a global device memory. The host processor also has access to the device memory and it stores the input for the GPU in the global device memory. Once the processing is done the GPU stores the results in the device memory and is eventually read by the host processor. Similar to a traditional CPU, the GPU architecture pipeline also has fetch, decode, scheduler, register read, execution, and write-back stages. The fetch and decode stages are very similar to traditional multithreaded architecture having multiple PC registers.

Warps (for NVidia) or wavefronts (for AMD) are a group of threads that are scheduled and executed in a lockstep [27]. The scheduler selects a warp to the fetch stage based on scheduling algorithms in place such as round-robin or greedy-fetch. A switching between warps happen in round-robin after a fixed time, whereas the switching in greedy-fetch occurs when there is an I-cache miss or fetching a branch instruction or an instruction buffer full. For this reason, branch predictors are not typically implemented in GPUs [27]. GPUs have an in-order scheduler that can schedule one or more warps at a time. Since there will be multiple warps that are ready to be executed, the scheduler uses a scoreboard to select one of those. Thus, from the programmers' point of view an application might look to have out-of-order execution [27]. The scoreboarding mechanism is used to implement dynamic scheduling in GPUs, it checks for read-after-write (RAW) or write-after-write (WAW) dependencies so that instructions from the same warp can be executed even if the earlier instructions have not

finished executing. This approach helps in increasing the instruction and memory level parallelism [27].

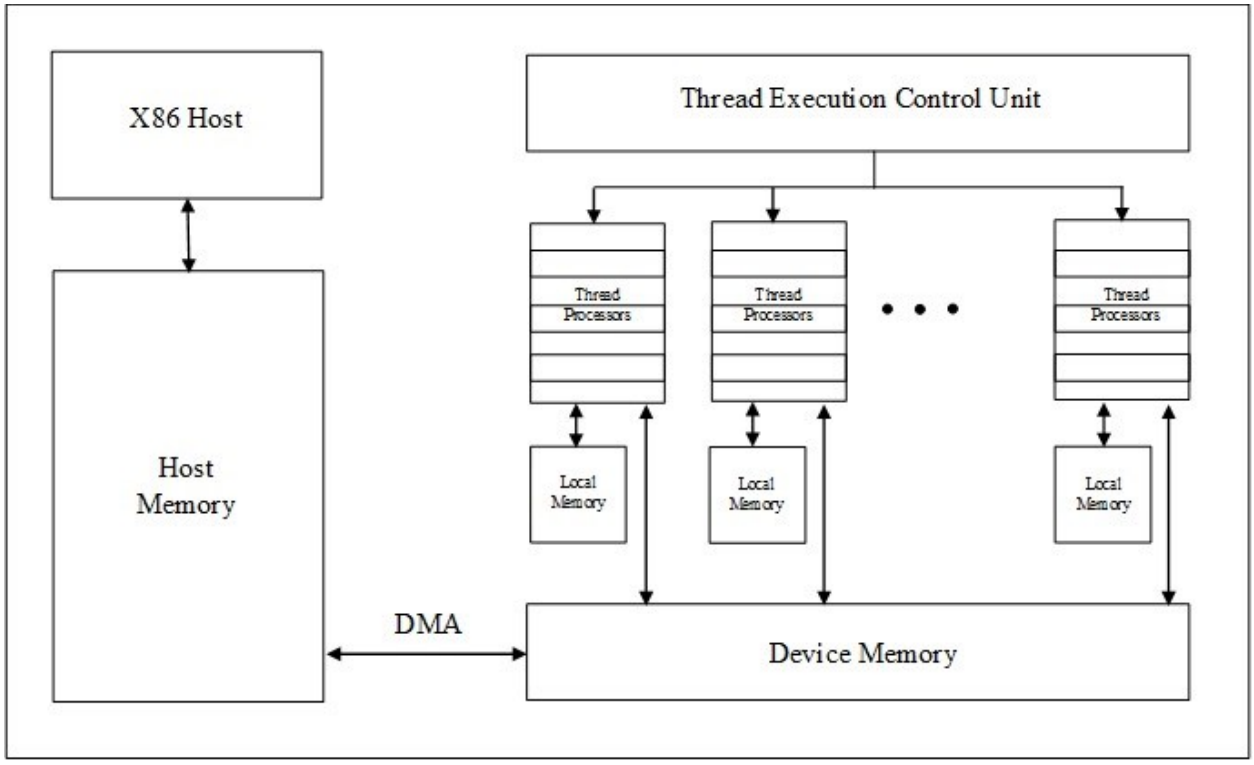

Figure 2.2: GPU Architecture block diagram

The memory system in GPU has several levels of hierarchy and large buffers or queues due to the large number of memory requests on-the-fly. In earlier GPU architectures, there were only software managed first-level caches and later hybrid software-hardware managed caches were introduced [27]. GPUs use GDDR as device memory, which is a high-bandwidth DRAM. It also has the higher number of DRAM banks and wider burst length, which increases the bandwidth [27]. Modern day GPUs employ techniques such as dynamic parallelism to increase their performance. This is a feature available in NVidia Kepler architecture GPUs, which helps the GPU to adapt to data and dynamically launch new threads. Another appealing feature is the introduction of Hyper-Q, which allows different CPU cores to simultaneously run up to 32 tasks on the Kepler device. GPUs that were introduced to perform only graphical tasks have now evolved to solve general purpose problems and are part of HPC domain.

iii. Introduction to OpenCL

OpenCL, or Open Computing Language, is a royalty free framework for heterogeneous programming managed by the non-profit technology consortium Khronos Group. The language is highly parallel and it efficiently maps to single or multiple device systems with homogeneous or heterogeneous architecture consisting of CPUs, GPUs, FPGAs, DSPs, and other SIMD elements. OpenCL supports parallelism by using task-based and data-based parallelism [28]. It currently supports CPUs such as x86, x86_64, ARM V7, ARM Cortex A9 with NEON and PowerPC. It is adopted by the two market leading companies in the GPU manufacturing industry, AMD and NVidia. Embedded GPU manufactures such as ARM, Vivante, etc. also support OpenCL Embedded profile on their devices. FPGA manufacturer Altera supports OpenCL 1.0 on their Stratix V FPGA. This shows the wide range of industry participation ranging from High Performance Computing (HPC) vendors to embedded vendors to support OpenCL. This cross-platform and industry wide support makes OpenCL a suitable and excellent programming model for developers to tap into the full potential of parallel processing. This also promises that OpenCL will continue to be widely available in the years to come with increasing scope and applicability [28].

The major challenge in developing OpenCL programming model is that the Applications Programming Interface (API) should be general enough to run across significantly different architectures and yet adaptable enough to attain high performance from them [28]. The OpenCL API meets these requirements and helps to create portable, device and vendor independent programs that are capable of exploiting the potential of many different platforms. The OpenCL specification is defined as four models including platform model, execution model, memory model and programming model. The platform model specifies a single host processor which is coordinating the execution and one or more devices that executes the OpenCL C program [28]. This model defines an abstract hardware model that is used by developers while writing OpenCL programs. Here, the OpenCL C program or function that executes on the devices is called an OpenCL kernel. The execution model defines how the OpenCL environment is configured on the host and how kernels are being executed on the device [28]. This stage includes setting up an OpenCL context on the host, facilitating host-device communication, defining concurrency model used for kernel execution, etc. The

memory model defines the abstract memory hierarchy the kernels use despite the actual underlying memory hierarchy and architecture [28]. The programming model explains how the concurrency model defined in the execution model is mapped to physical hardware [28].

OpenCL kernels are similar to functions and they execute on a device. The OpenCL API facilitates creating contexts to manage the execution of kernels on the device, transfer data between the device and the host, profiling the kernel execution and data transfer, etc. Let us take a simple example of vector multiplication to understand how different OpenCL is from traditional C programming. The following is a function in C to multiply the values in one vector with the corresponding values in another vector and to store the product in a third vector.

```
//Perform an element-wise multiplication of A & B, and store
the //product in C
//There are N elements in vectors A & B

void vecMult(int *A, int *B, int *C, int N)
{
  for(int i = 0; i < N; i++)
     C[i] = A[i] * B[i];
}
```

Before we look into the OpenCL kernel of this vector multiplication, we need to understand certain OpenCL terms. A work-item is the unit of concurrent execution in OpenCL [28]. In the vector multiplication example, each work-item executes the kernel function body. Here, we need to instruct the OpenCL runtime to create as many work-items as there are elements in the input and output array, and it maps a single iteration of the loop to a work-item. This in turn is mapped onto the underlying hardware of CPUs, GPUs, FPGAs, etc. When a device starts executing an OpenCL kernel, there are intrinsic functions provided in the language for a work-item to identify itself [28]. See Appendix A for the DCT calculation kernel. In the following OpenCL kernel, the `get_global_id(0)` allows the programmer to identify the index of the vector element the kernel instance is operating currently.

```
//Perform an element-wise multiplication of A & B, and store
the //product in C
//N work-items will be created to execute this kernel
__kernel void vecMult(__global int *A,
                      __global int *B,
                      __global int *C)
{
  int idx = get_global_id(0); //OpenCL intrinsic function
  C[idx] = A[idx] * B[idx];
}
```

When an OpenCL kernel is executed, the programmer specifies the number of work-items that are to be created as an n-dimensional range (NDRange) [28]. An NDRange is a one-, two- or three-dimensional index space of work-items that will map to the dimensions of the input or output data. In a typical OpenCL program, the NDRange dimensions are specified as an N-dimensional array of type `size_t`. Here, N denotes the number of dimensions of the work-items that are being created [28]. In our vector multiplication example, if we assume that the size of the input vectors is 2048 elements, then we specify the NDRange as follows:

```
size_t globalIndexSpace[3] = {2048, 1, 1};
```

Scalability is achieved in OpenCL by dividing the NDRange of work-items into smaller equally sized work-groups as shown in Figure 2.3 [28]. An N-dimensional NDRange or index space requires the work-groups to be specified using the same N dimensions. For instance, if the input index space is two-dimensional, the programmers need to specify two-dimensional work-groups in the OpenCL program. An advantage of using work-groups is that, the work-items inside a work-group can perform barrier operations to synchronize between them, and they also have access to a shared memory space [28]. For the vector multiplication example, we can specify the work-group size as shown below. In the given division of global work-items of size NDRange into local work-items inside work-groups, there are 2048 work-items and 64 work-items inside a work-group. Hence, we can see that there are 32 work-groups (2048/64) in the example below.

```
size_t localIndexSpace[3] = {64, 1, 1};
```

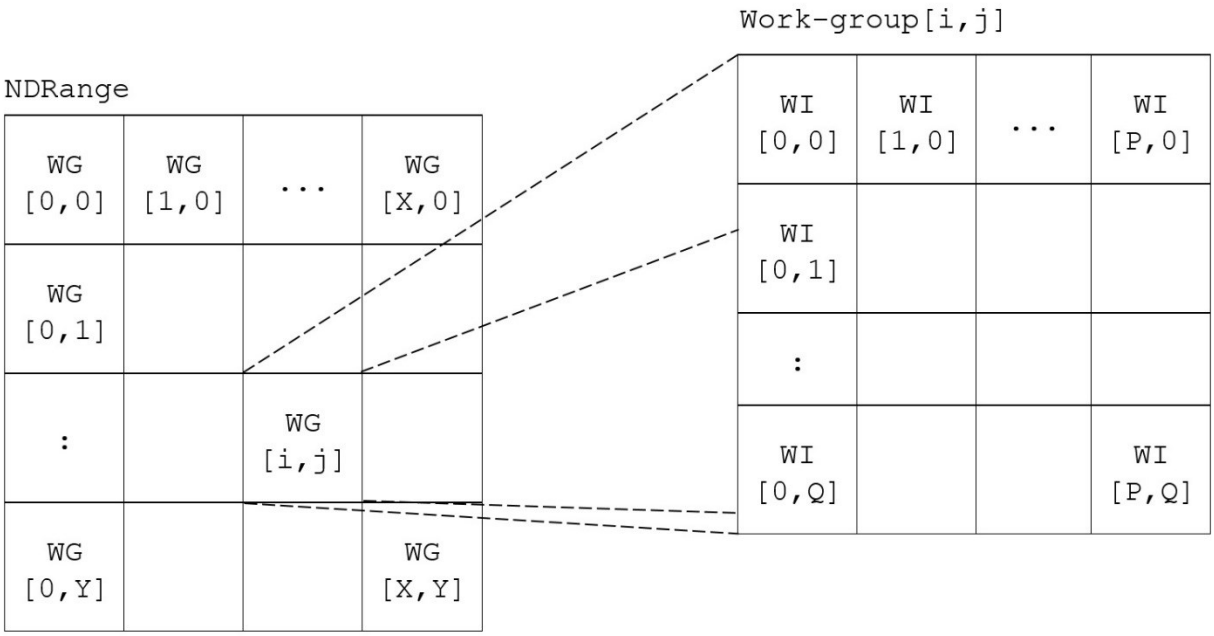

Figure 2.3: Division of OpenCL NDRange into work-groups and work-items [28]

The OpenCL platform model not only defines the roles of the host and the devices, but also provides an abstract hardware model for devices [28]. Platforms are vendor specific implementation of OpenCL API in which a host coordinates the execution on one or more devices. Since these implementations differ from vendor to vendor, the platform designed by a company cannot communicate with the device designed by another company [28]. The platform model defines devices as an array of functionally independent compute units that are further divided into processing elements. The API function `clGetPlatfromIDs()` is used to query the number of platforms available and to query the platform IDs. In a typical OpenCL program, after the first call the API returns the number of available platforms, and the developer will allocate the required space for storing the platform IDs. On the second call, the API is provided with a pointer to a `cl_platform_id` variable and it will return the IDs of the available platforms. Just like we queried the number of platforms and the platform IDs, we need to query the number of devices available and their IDs. The API function used for this is `clGetDeviceIDs()`, which takes additional arguments of a platform and device type. Here, the platform refers to the platform ID returned by invoking the

`clGetPlatfromIDs()` and the device type argument is used to query a specific set of devices available in the platform. Querying and identifying platforms and devices is part of the OpenCL platform model.

Now that we have the platforms and devices identified, we need to create a context on the host and configure it to pass commands and data to the devices for executing kernels [28]. In OpenCL, a context is an abstract group of objects or a container that exists on the host and coordinates host-device interaction, manages memory objects, keeps track of the kernels and programs that are created for each device. The API function `clCreateContext()` is used to create a context. Developers can restrict the scope of a context by specifying the properties argument in the function call. By this way, programmers can create contexts for multiple platforms and fully utilize the available resources from various vendors [28]. The OpenCL specification also allows the developer to create a context that includes all or some of the devices of a specified type using the API function `clCreateContextFromType()`. This way, programmers can create contexts for specific sets of devices such as CPU, GPU, etc. [28].

In OpenCL, the host communicates with the devices using command queues. Once a context is created and the host decides to work with a group of devices, a command queue should be created for each of the devices separately. This way, whenever the host needs to communicate data or to instruct the device to perform certain actions, it will submit the commands to the proper command queue [28]. The API `clCreateCommandQueue()` is used to create a command queue for a device. We can create dependencies in OpenCL by using events. Any operation that enqueues a command into a command queue will create an event. Events can also be used for providing a mechanism for profiling in OpenCL [28]. To transfer data to the device, the developer must encapsulate it as a memory object. Buffers, which are equivalent to arrays in C, and images, which are designed as opaque objects, are the two memory objects available in OpenCL. A memory object is valid only within a single context [28]. The API `clEnqueueWriteBuffer()` and `clEnqueueReadBuffer()` are used to transfer data from host memory to the buffers created on the device and back. The transfer from host memory to the device memory can be blocking or non-blocking. OpenCL programmers can make a blocking read or write by specifying the blocking_read or

blocking_write parameter. If the parameter is `CL_TRUE`, then the read or write is blocking, i.e., the function will return only after completing the data transfer. Similarly, setting the parameter to `CL_FALSE` will create a non-blocking call where the function will return after initiating the memory transfer.

OpenCL program, which is a kernel or collection of kernels, is compiled during runtime through a series of API calls [28]. This runtime compilation allows the code to be optimized for a specific architecture and it also helps the program to be portable across OpenCL runtimes from various vendors. To facilitate this runtime compilation, the first step is to store the source code in a string; if the source code is residing in a separate .cl file, then it is read into a character array. The second step is to convert this character array into a program object of type `cl_program` by calling the `clCreateProgramWithSource()` API. The third step is to compile the program object for one or more devices using the API `clBuildProgram()`. Following these steps will create precise vendor-specific binaries that can efficiently exploit the potential of the intended devices [28]. Once the program object is created, we need to extract the kernels from the program object to create a kernel object of type `cl_kernel` to execute kernels on the device. The name of the kernel is passed to the `clCreateKernel()` API to create a kernel object. After creating the kernel object, we need to specify each argument of the kernel by calling the `clSetKernelArg()` API. The `clEnqueueNDRangeKernel()` API is used to launch kernels on the device. This call is asynchronous and will return immediately after the command is enqueued in the command queue and likely before the kernel starts executing [28]. See Appendix B for more details regarding the OpenCL APIs mentioned above.

So far we have covered the OpenCL host code, however in order to understand the proper working of the OpenCL kernel and to exploit the device memory resources, we need to know the OpenCL memory model. The four types of memory spaces available in OpenCL are global, constant, local and private memories. The global memory is visible to all compute units on the device and whenever the data is transferred from host to device, it is kept in this memory. Since the host can access only the global memory, any data that has to be written back to the host should also reside here. The keyword `__global` is used to qualify a variable that is residing in the global memory. Constant memory holds the variables whose

values never change in an execution. It is also used to store data whose each element is simultaneously accessed by all work-items. Constant memory is part of global memory and the data mapped to this memory is specified using the `__constant` keyword [28]. Local memory is a scratchpad whose address space is unique to each compute device [28]. This memory is shared between work-items in a work-group and it has lower latency and higher bandwidth compared to the global memory. A local memory variable is declared by using the `__local` keyword. Private memory is the memory that is unique to an individual work-item. Local variables and non-pointer kernel arguments are private by default [28].

The OpenCL Embedded Profile, which we are using in this research, relaxes the requirements from Full Profile in order to accommodate the power and area constraints in embedded platforms. The Embedded Profile is a strict subset of the Full Profile and it is introduced to provide a single specification for both profiles, to ensure that OpenCL programs written for the Embedded Profile runs on devices that implement Full Profile, and to allow future revisions of the standard to consider the requirements of desktop and hand-held devices [29]. One of the major differences between the OpenCL Embedded and Full Profiles is making 64 bit integers optional in the Embedded Profile. This means, the scalar data types long and ulong, and the vector data types longn and ulongn need not be supported by a device that implements the OpenCL Embedded Profile [29]. In the OpenCL Full Profile, support for reading of 3D images is required while support for writing 3D images is optional. In OpenCL Embedded Profile, 3D image support for reading and writing is optional [29]. The precision for converting normalized integers to single-precision floating-point values is less than or equal to 1.5 ulp (unit of least precision) for Full Profile, whereas this is less than or equal to 2 ulp for Embedded Profile [29]. OpenCL Full Profile has built-in functions that can perform atomic operations on 32-bit integers to global and local memory while these functions are optional in Embedded Profile [29]. Another major difference between OpenCL Full Profile and Embedded Profile is that the support for positive infinity, negative infinity, and NaN is not required in the Embedded Profile [29]. The OpenCL Embedded Profile allows hand-held device manufactures to implement the specification with relaxed requirements on their devices and helps the developers to utilize the full potential of the SoC using heterogeneous programming.

Chapter 3 : Prior Art

This chapter analyzes the state of the art to gather knowledge on General Purpose computing using GPU (GPGPU), embedded GPGPU, Java GPGPU and the methodologies used to achieve GPGPU from Java. This chapter also develops metrics to analyze various solutions, and sets the framework for defining research questions that will be addressed in the following chapter.

i. Literature Review on GPGPU

The evolution of GPUs from just being graphical processors to General Purpose Computing using GPUs or GPGPUs is introduced in Chapter 2ii. Here, we explore the state of the art to understand and evaluate the benefits of using GPGPU.

Today, GPUs are widely used in industry to accelerate the execution of general purpose problems due to their ability to achieve better performance by consuming less energy compared to CPUs. This paradigm of using GPUs for general purpose computing is called GPGPU. The application of GPGPU domain stretches from supercomputing [18] to embedded domain [13]–[16]. The major advantages that make GPUs more preferred are their popularity, low cost, availability of open GPGPU programming standards, and their high computational power and memory bandwidth. GPUs are manufactured as commodity processors that are widely available at an affordable budget. Programming languages like OpenCL and CUDA help developers to utilize the full potential of the challenging GPU hardware. Every day, these languages are getting new libraries, profilers, debuggers, cross-compilers, etc. and are evolving into portable easy to debug and develop languages. Study [30] shows that when Nehalem CPUs deliver performance of 100 to 150 GFLOPS, a GPU of the same cost can provide over 1 TFLOP. The memory bandwidth of GDDRs used in GPUs also provides higher performance for GPGPU programs. All the above factors contributed in making GPGPU popular in industry and academia. Several studies [9]–[12], [30], [31] are conducted to evaluate and optimize the hardware and software aspects of GPGPU.

In [9], Suda et al. analyze various aspects of using GPUs for general purpose high performance computing. In their study, they run benchmarks on GPU and compare it with a supercomputer to evaluate the potential of GPUs in solving general purpose problems. The experiments in [9] were conducted by comparing the benchmark performance on the T2K supercomputer with a NVidia GeForce GTX 260 GPU. Their study shows that the low-cost GPU can provide performance similar to that of the T2K supercomputer. The study also explains the HPC NVidia Tesla GPU and the power savings that can be achieved by using GPUs to solve high performance problems.

Though GPUs help in accelerating programs, this comes with an additional cost of writing programs in languages such as CUDA, OpenCL, Brook+, etc. These languages require the developer to write explicit GPU functions called kernels, allocating memory on the GPU, managing data transfer between CPU and GPU, and manually optimizing the utilization of GPU memory and processing resources. This process is tedious and error-prone, and could affect the time to develop a product. Han et al. in [10] introduce a high level GPGPU programming framework called hiCUDA. In their work, they define hiCUDA directives, and explain the design and implementation of a prototype compiler that translates these directives into CUDA program. The hiCUDA compiler consists of a GNU 3 Front-end that takes a sequential C program with hiCUDA pragmas and output a WHIRL file which is an intermediate representation. Thus generated WHIRL file is processed by a hiCUDA directive handler to generate another WHIRL file and it is further processed by a CUDA code generator to generate a file containing CUDA program [10]. The CUDA compiler toolchain from NVidia processes this to generate the CUDA binaries, which are then run on the GPU [10]. Several financial, medical and scientific benchmarks are executed on the NVidia GeForce 8800GT GPU using CUDA and hiCUDA to evaluate the performance. The experiments in [10] show that the ease of use achieved by employing hiCUDA directives does not affect the performance.

In [11], Murthy et al. explain the impact of loop unrolling on GPGPU programs. They developed a static, semi-automatic, compile-time approach for identifying optimal loop unroll factors for loops in CUDA programs. Loop unrolling is a technique for replacing a loop with several copies of the logic inside the body of loop and updating the loop control logic

accordingly. If necessary, a remainder loop is also added to the end of the unrolled loop [11]. Loop unrolling also provides better instruction scheduling and Instruction Level Parallelism (ILP). A disadvantage of loop unrolling is that it increases the size of the loop body and if the loop body size exceeds the I-cache size, it can cause cache misses and a reduction in performance gained by ILP. Benchmarks such as Black-Scholes Simulation, N-Body Simulation, Matrix Multiply, Coulombic Potential, Monte-Carlo Simulation and MRI-Q were executed on two NVidia GPUs. All these benchmarks were executed on both the machines with and without loop unrolling. The experiment results in [11] show a performance increase up to 74.26% for MRI-Q while the performance for certain cases of Black-Scholes Simulation is decreased by 42.61%. Their study evaluates the scope of extending the classical compiler optimization technique of loop unrolling to the GPGPU domain. In [12], Chu et al. show that the acceleration of several existing CPU programs are possible by employing GPUs. The existing multi-threaded OpenMP programs running on Intel Xeon and Sun UltraSparc T2 processors are compared to the OpenCL version of the programs running on NVidia GeForce and ATi Radeon GPUs. Their experiments achieve acceleration up to 100 times by employing GPUs. The study in [12] also shows that implementing applications using OpenCL can provide acceleration over various processors used in industry such as UltraSparc T2 and Intel Xeon.

The study in [30] analyzes several data/process dependent and data/process independent algorithms and explains the upsides and downsides of using GPUs for solving them. Small local data (memory and register) requirements, stream computing, arithmetic intensive, memory bandwidth and data parallelism are the general requirements that are to be fulfilled to attain acceleration using GPUs [30]. While better performance and energy efficiency are the upsides of using GPUs to solve certain problems, communication bottleneck between the host and the device is viewed as a downside [30]. The study in [30] also suggests optimizations that can be done on thread deployment and memory usage to improve the efficiency of GPU programs. The advent of NVidia's Kepler architecture GPU that supports virtualization made GPGPU computing popular in cloud computing [31]. The paper also explains a general scheme of GPGPU cloud computing system and the layers that are involved. Studies in [9], [12] analyze the performance and energy efficiency improvements of various benchmarks achieved by using GPUs. Other studies [10], [11] explain how the existing GPGPU solutions

can be modified for better performance and ease of use. Studies [30], [31] are also conducted to analyze the potential of GPGPU domain in solving scientific, industrial, medical and cloud computing problems.

A metric is developed to compare different aspects of GPGU and is explained in Table 3.1. The table explains the benchmark, reference machine, GPU and language used in the solutions. It also shows the speedup achieved by employing various solutions. The top500 website [18], which ranks supercomputers based on their performance have two supercomputers that have GPUs in their architecture in their top 10 list. All these show the potentials of using GPUs in solving general purpose problems. However with the advancements in VLSI technology, today we have smartphones, tablets and other devices that encompass very complex GPUs. So, in the next section, we explore the literature to see if performance and energy gains are achieved by offloading compute-intensive algorithms on embedded GPU.

Solution	Benchmark	Reference Machine	GPU Used	Language Used	Speedup (relative to reference machine)
[9]	Polynomial Approximation of reciprocal function	T2K Supercomputer (X86_64 Cores)	NVidia GeForce GTX 260	CUDA	Similar Performance
[10]	BSOP, MM, NBODY, CP, SAD, TPACF, RPES, MRI-Q, MRI-FHD	NVidia GeForce 8800GT	NVidia GeForce 8800GT	hiCUDA	Similar Performance
[11]	NBODY, BSOP, MM, CP, MCS, MRI-Q	NVidia GTX280/ GTX8800	NVidia GTX280/ GTX8800	CUDA	-42.6% to 74.26%
[12]	Calculation of PI, MM, Matrix Transpose	Intel Xeon, UltraSparc T2	IBM Cell PS3, NVidia GTX285, ATi Radeon HD5850	OpenCL	20X to 100X

Table 3.1: GPGPU Solutions

ii. Literature review on Embedded GPGPU

Today GPUs are highly sought after in general purpose computing for their highly programmable, massively parallel architecture that can provide excellent floating point performance at a low cost. Another appealing feature of GPUs in embedded domain is the energy savings that can be achieved by employing them. Companies such as AMD, NVidia, ARM, Vivante, ZiiLabs, etc. provide GPUs suitable for embedded platforms. Though programmable GPUs are present in embedded devices, languages like OpenGL, Renderscript, etc. that demand knowledge of graphics programming and expertise in graphic pipeline were required to program them. This was changed only with the advent of OpenCL in the embedded domain. Even with several research works done exploring various aspects of GPGPU, most of them are for high-performance desktop or server GPUs and not much research is done to explore the potential of embedded GPGPU. Embedded GPUs have limited or no user-defined memory, limited size of cache, have power and area constraints, etc. [13].

Several studies [13]–[16] are done to showcase the potential of embedded GPUs in providing higher performance per watt and to enable GPGPU on mobile devices. In [13], Maghazeh et al. explain the importance and advantages of using embedded heterogeneous computing. Here, authors run several benchmarks on Vivante and NVidia Tesla GPUs, and compare the power consumption and performance over their serial implementations. In [13], benchmarks such as Rijndael's algorithm, bitcount, genetic programming, convolution and pattern matching are accelerated on an embedded GPU from Vivante. The authors in [13] also compare the architectural differences between an embedded GPU and a desktop GPU and how this affects the execution of programs. They show that since the embedded GPU and the CPU are using the same physical memory, while executing programs of larger data size, the kernel execution is the bottleneck for embedded GPU whereas the data transfer is the bottleneck for the NVidia Tesla GPU. This can be viewed as an important indication which suggests that simply porting the existing desktop versions of the GPGPU programs on embedded platform does not guarantee the similar performance. All the benchmarks in [13] gave better performance on Vivante GC2000 GPU over their single, dual and quad-core implementations on ARM Cortex A9 CPU. The analysis also show that the Rijndael's implementation on GPU consumes slightly more energy than that of the CPU implementation

while all other benchmarks show reduced energy consumption by employing GPU. The genetic algorithm benchmark in [13] uses both the GPU and CPU to solve the problem and it could attain better energy and performance efficiency by exploiting the heterogeneous architecture. The power and performance analysis in [13] shows the potential of using embedded GPUs for GPGPU.

A power consumption approach for evaluating GPGPU on embedded platform is explained in [14]. The benchmark, FFT, is run on Intel Atom D525 CPU and on second generation NVidia ION GPU to evaluate the energy and performance efficiency [14]. The GPU version of FFT consumes 14.21 Watts while the multi-core CPU version consumes only 12.24 Watts [14]. Though the power consumed by GPU FFT computation is more than that of the CPU version, the time taken to compute FFT by GPU is only 0.25 milliseconds and that for CPU is 0.69 milliseconds [14]. This acceleration will not only balance, but also provide a 2.4 times more energy efficiency [14]. This shows that employing embedded GPUs will help in attaining acceleration along with energy savings.

In [15], several benchmarks which are part of the High Performance Embedded Computing (HPEC) benchmark suite were executed on an NVidia GPU. The HPEC benchmark suite contains the time domain and frequency domain FIR filter (TDFIR, FDFIR), corner turn (CT), pattern matching (PM), constant false-alarm rate detection (CFAR), genetic algorithm (GA), QR factorization (QR), singular value decomposition (SVD) and database operations (DB). These benchmarks were then run on the Intel Core 2 Duo CPU and on NVidia Fermi GPU to analyze the performance improvements. The results varied giving a speedup between 1.13 times and 114.1 times, DB being the least accelerated and FDFIR being the most. Selected HPEC benchmarks were also implemented on a DSP (ADSP-TS101) and are also compared with the GPU implementations. The results in [15] show that GPU outperformed DSP by two orders of magnitude. However, the DSP implementations were highly power efficient compared to their GPU counterparts by consuming only 23.8 times less power. The major drawback of the research done in [15] is that the HPEC benchmarks are evaluated on a PC rather than on an embedded platform. However, the study in [15] makes a good comparison between CPU, GPU and DSP implementations providing a platform for future research in embedded heterogeneous computing.

In [16], Leskela et al. explains the benefits of using OpenCL embedded profile in mobile devices. The authors run an image processing algorithm on an ARM Cortex A8 CPU, PowerVR SGX 530 GPU and TI TMS320C64x DSP, and analyze the energy and performance efficiency achieved. The algorithm used is Gaussian Blur with a preprocessing of geometry adjustment and a post-processing of colour adjustment [16]. The measurements in [16] show that employing GPU accelerates the execution time by 3.58 times. The CPU and GPU occupancies are analyzed and several possible scheduling improvements are also suggested in [16]. The study also explains how the GPU and CPU can be used together to solve the algorithm in hand. The power and energy analysis in [16] show that the GPU consumed very less power compared to the CPU implementation. While the CPU consumes 3.93 J per frame, the GPU version of the algorithm consumed only 0.56 J, which is 7 times less. The authors also note that the major portion of time for the GPU implementation is spent on data transfer, and hence more than half of the aforesaid energy consumption is accounted for that matter [16].

The current works in literature [13]–[16] show that both energy and performance efficiency can be achieved by offloading compute intensive portions of programs on embedded GPUs. A metric to analyze this is developed and is shown in Table 3.2. The table compares various implementations of different benchmarks on embedded GPUs and analyses the energy savings and speedup achieved. Solutions [15], [16] evaluates the energy and performance efficiency by running benchmarks on CPU, GPU and DSP, whereas the study in [14] focuses more on balancing the computational load between CPU and GPU to attain better performance. The study in [15] also explores the possibilities of high performance computing on embedded platforms. The platform used in [13] is the first OpenCL conformant embedded board and the same board is used in our research. The analysis reveals that performance and energy efficiency can be achieved for a wide range of benchmarks using this platform. However, the authors in [13] did not explore how to accelerate Java on this platform. The relevance and advantages of accelerating Java on GPU along with the existing solutions are explained in the next section.

Solution	Benchmark	Reference Machine	GPU Used	Energy Efficiency	Speedup (Relative to Reference Machine)
[13]	Rijndael	ARM (single-core)	Vivante GC2000	Slightly negative	2.4X
	Bitcount	ARM (quad-core)	Vivante GC2000	Slightly positive	2.55X
	Convolution	ARM (quad-core)	Vivante GC2000	positive	5.45X
	Pattern Matching	ARM (quad-core)	Vivante GC2000	positive	5.38X
[14]	FFT	Intel Atom D525	NVidia ION 2	2.4X	2.76X
[15]	TDFIR, FDFIR, CT, PM, CFAR, GA, QR, SVD, DB	Intel Core 2 Duo & ADSP-TS101	NVidia Tesla C2050	-23.8X compared to DSP	1.13X to 114.1X
[16]	Gaussian Blur	ARM CPU & TI TMS320C64x DSP	PowerVR SGX 530	7X	3.58X

Table 3.2: Embedded GPGPU Solutions

iii. Literature Review on Java GPGPU

Today, GPUs are used to solve non-graphical problems due to their high floating point efficiency, massively parallel architecture, high memory bandwidth, etc. The popularity of Java, its evolution over the years, its built-in multi-threading support along with the availability of high performance GPUs encouraged developers to explore the Java GPGPU domain. Java GPGPU is the process of utilizing GPUs for general purpose computing from Java programs. Several studies are conducted on how to accelerate Java on GPUs [1]–[8], [32]. The performance of Java has improved in the last years narrowing its gap with native compiled languages like C and FORTRAN [1]. Although this is true for sequential applications, this gap is much more for highly parallel High Performance Computing (HPC) applications [33]. Though Java is one of the popular and preferred programming languages,

most of the GPGPU computing solutions such as OpenCL, CUDA, etc. were developed as libraries to be used from C or C++. Hence, languages like Java should utilize wrappers via JNI (Java Native Interface) or JNA (Java Native Access) to exploit GPUs for general purpose acceleration. The two major approaches available in the literature to provide GPGPU from Java are: to provide Java bindings to a lower level language like OpenCL or CUDA, or to use user-friendly APIs that abstract GPU programming and runtime system which translates Java bytecode into CUDA or OpenCL in a transparent manner. While Java bindings are designed to provide better performance, the increasing complexity of code favors the modified VMs and they help in achieving better productivity. JOCL [3], jCUDA [4] and JavaCL [6] are few of the popular open-source Java bindings which we will be discussing in this section. Meanwhile, Rootbeer [2], Aparapi [5], JaBEE [7] and Japonica [8] are a few popular user-friendly approaches for Java GPGPU that will be discussed in the following sections.

a) Java GPGPU using Java Bindings

Language Bindings include writing wrapper functions to convert data structures in a source language to another destination language. This way, developers can reuse the existing modules in the destination language without needing to rewrite them in the source language. Typically, the source language will be a high-level language such as Java, Python, Ruby, TCL, etc. and the destination language will be a low-level language like C. Here, to enable GPGPU from Java, a possible solution is to write wrapper functions for OpenCL or CUDA methods. Several such Java bindings [3], [4], [6], [34] are available as open source, and can be employed to utilize the GPU for general purpose computation from Java.

JOCL

JOCL [3] is a low-level Java binding that utilizes JNI calls to invoke OpenCL APIs. There are two independent open source Java bindings named JOCL available, one is from the www.jocl.org website [3] and the other from jogamp, which can be found in [34]. Though both these Java bindings use JNI and serve the purpose of accelerating Java programs on GPUs, we limit our discussion to the JOCL in [3]. The bindings offered in JOCL [3] are very

similar to the original OpenCL API. The major advantage of JOCL is its resemblance with the traditional OpenCL API and this makes it easier for new developers to learn programming using the binding. However, this similarity requires the developer the knowledge of OpenCL programming in order to code using JOCL. The portability of Java programs developed using JOCL can be preserved, provided that the JOCL library contains the target machine specific binaries. This being said, writing the host code and the Java code in the same program makes the program cumbersome and difficult to debug. However, this is the case with other Java bindings that do not provide some level of abstraction to hide the underlying OpenCL APIs. Another advantage of JOCL is its light-weight binaries that make porting between platforms easier.

jCUDA

jCUDA [4] is one of the most active Java GPGPU projects and it provides a wrapper over CUDA 5.5 runtime and driver API. Similar to JOCL [3], jCUDA [4] allows direct interaction with the GPU including memory management and launching kernels from Java. However, while JOCL launches OpenCL kernels, jCUDA launches CUDA kernels on the GPU. It also require the developer to manually write kernels, allocate memory, transfer data, etc. to serialize the traditional Java programs. It consists of static methods that are very similar to the native APIs and supports a number of NVidia libraries. This support for NVidia libraries made jCUDA a popular solution for Java GPGPU on CUDA enabled platforms. jCUDA consists of the base jCUDA package, which has the CUDA runtime and driver APIs, JCublas, JCufft, JCudpp, JCurand and JCusparse. JCublas is the Java bindings for the CUBLAS library, a library that provides Basic Linear Algebra Subprograms on CUDA. JCufft is the Java bindings for NVidia CUDA FFT library called CUFFT. JCudpp contains the Java bindings for the CUDPP library which is the CUDA Data Parallel Primitives library. JCurand contains the Java bindings for CURAND, a CUDA Random number generator library. JCusparse provides bindings for the CUSPARSE library, and it is the NVidia CUDA Sparse matrix library. The jCUDA API consists of a set of static methods that are similar to the native library functions. The study in [1] evaluates the execution efficiency of jCUDA against Java and CUDA. For several benchmark algorithms from SHOC benchmark suite, including matrix multiplication and FFT, jCUDA could achieve better performance than the traditional

Java program but was less efficient than CUDA [1]. jCUDA could also achieve performance close to that of CUDA in certain cases of matrix multiplication [1]. Its performance and the support of parallel processing libraries made jCUDA a popular Java GPGPU solution.

JavaCL/ScalaCL/OpenCL4Java

JavaCL [6] is an API that wraps the OpenCL library to make it available in Java. It comprises a low-level API that has a 1-to-1 relationship with the C OpenCL API, an object-oriented API, utilities for parallel reduction and matrix multiplication, and ScalaCL which helps to run Scala code on GPU. JavaCL [6], like other Java bindings, facilitates developing platform-independent programs to run on GPU. It provides a programmer-friendly API and memory management along with automatic and transparent caching of OpenCL binaries. JavaCL maps OpenCL entities that are allocated by the OpenCL driver, typically in the GPU (device) memory to Java objects that are managed by the JVM's garbage collector [6]. These OpenCL entities are freed when their Java object counterparts are garbage collected or when their release() method is called [6]. However, waiting for garbage collector to release these objects can lead to serious issues including failing of the program as the OpenCL driver could run out of memory. A built-in solution available in JavaCL to solve this is that whenever a JavaCL program fails due to the lack of OpenCL memory, it triggers a full garbage collection, waits for a little while and then retires [6]. This however is having an adverse effect on the performance of the program, and the authors of JavaCL highly recommend developers to call release() method often in their programs to release unused OpenCL entities [6]. JavaCL uses JNA as a low level interface to invoke the OpenCL libraries from Java. It provides full support for OpenCL 1.0 specification and partial support for OpenCL 1.1 specification.

b) Java GPGPU by modifying VM

While Java bindings provide wrappers over GPU methods, the solutions that provide Java GPGPU by modifying JVM works on the Java bytecode by translating it into CUDA or OpenCL for GPU. While Java bindings are designed for performance, the solutions that modify VM are designed for better productivity. Since much of the OpenCL and CUDA workloads including, resource allocation on GPU and transfer of data between CPU (host)

and GPU (device) is automated, these solutions do not strictly require the developer the knowledge of OpenCL or CUDA. This abstraction helps in increasing the productivity and it reduces the development cost and time. This can be considered as the major advantage of using Java GPGPU solutions that modify JVM. In this section, we analyze some of the solutions that are available in the literature including Rootbeer [2], Aparapi [5], JaBEE [7] and Japonica [8].

Project Sumatra is an OpenJDK [35] backed project introduced to harness the parallel processing power of GPUs and APUs from Java. The initial focus of the project is on the Hotspot JVM to enable code generation, runtime support and garbage collection on GPU [35]. The goals of the project are to enable Java applications to take advantage of heterogeneous processing units, to extend JVM JITs to generate code for heterogeneous processing units, to integrate the JVM data models with data types that can be processed efficiently by heterogeneous hardware, to allow JVM to interoperate efficiently with high-performance libraries built for heterogeneous processing elements, and to extend the JVM managed runtime to track storage allocation and pointers in the heterogeneous system [35]. The Sumatra project use several features and libraries of Java 8's Lambda project, which is the set of libraries developed for improving multicore support of Java programs. Though the project Sumatra's major focus is on Java, it can come to use in future for utilizing GPUs from other JVM-hosted languages such as JavaScript/Nashorn, Scala and JRuby [35]. There are several challenges to be encountered in the project and a few of them are: the complexities posed by various GPU backends and layered standards such as OpenCL, CUDA, Intel Phi, PTX and forthcoming HSA, to cope with the dynamically varying mixes of serial and parallel code, to deal with exceptions inside loop kernels, to reduce data copying and inter-phase latency between ISAs and loop kernels, etc. [35]. Despite all these challenges, project Sumatra abstracts GPU programming from the developer and helps the developer to concentrate more on the algorithm rather than the implementation. However, the project is still ongoing and is not available to the general public yet [35]. If project Sumatra can overcome the performance drawback of other Java GPGPU techniques that modify the JVM, then it can be considered as a solution that can provide better performance on GPU without compromising the portability or productivity of Java programming.

Rootbeer

In [2], Prat-Szeliga et al. introduce a tool called Rootbeer, which abstracts all the memory allocation, kernel creation, transferring data to and from the GPU, and other prototyping steps. Rootbeer reduces the prototyping time by eliminating these manual steps. It does not automatically parallelize the code and programmer must specify what each GPU core should do. However, the conversion of complex graph of objects into arrays of primitive type is automated [2]. Rootbeer uses a kernel interface that declares a GPU method that is the main entry point of developer's code on GPU. The developer sets values for the private fields of the class that implements kernel interface and Rootbeer copies all reachable fields and objects from the GPU method to GPU and write back the results to CPU after computation [2]. Rootbeer Static Transformer reads the .class file from the input jar and, using Soot [36], it converts the .class file to an intermediate three component representation called Jimple [2]. Then it finds the kernel interface and GPU methods, and generates the CUDA code, which is then compiled using nvcc to generate the cubin file [2]. The Rootbeer Static Transformer then packs this cubin file to an output jar file [2]. The experiments in [2] show that employing Rootbeer could increase the performance by 67X for dense matrix multiplication and 54X for brute force FFT. However, another benchmark, Sobel filter, used in [2] reduced the performance by 3.8X. The study also suggests that Rootbeer can be used for solving other computational bound applications including scientific simulation [2]. Rootbeer lowers the learning curve and provides an easier platform to offload Java programs on GPU compared to Java bindings or other GPU programming languages such as CUDA or OpenCL. Rootbeer is one of the most complete Java GPGPU implementations available with all of the Java features available in the VM [2]. It is extensively tested and is a robust VM for accelerating Java on GPU. Since Rootbeer generates CUDA binaries (cubin), it can run programs only on GPUs that support CUDA and this can be considered as one of the major drawbacks of using Rootbeer.

Aparapi

Aparapi (A PARallel API) [5] is a popular open source project that allows a Java developer to use GPU by executing data parallel code fragments on GPU or APU. Aparapi accelerates the data parallel portions of the Java code by analyzing the bytecode, converting it into OpenCL at runtime and executing it on the GPU [5]. Also, if Aparapi cannot execute the program on GPU, it will then be executed in a Java thread pool [5]. Aparapi provides developer high-level APIs to represent data parallel workloads in Java, abstracting all the GPU implementation steps. This abstraction makes programming using Aparapi very easy and it does not require any knowledge of OpenCL or CUDA programming. Aparapi was made to cater the need of a Java-style programming framework for GPUs, and to replace the existing Java bindings which are considered as literal translation of OpenCL or CUDA to Java. In Aparapi, the developer hints which portion of his code can be parallelized by overloading a run() routine in a kernel class provided with the package [5]. This way, the tedious job of searching for the parallelizable routines and snippets inside the Java program can be easily solved. Another similarity in programming is that the overloading of the kernel class run() routine is fairly similar to overloading the thread class run() routine which is widely used in Java for multi-threaded programming. Once the developer specifies the run() entry points, Aparapi parses through the method, translates it into OpenCL and runs it on the GPU.

Another advantage of Aparapi is its ability to execute this code on multicore CPU if for some reason the code fails to execute on the GPU, thus providing a better performance than that of the serial implementation [5]. Though Aparapi is listed under modified JVM section in this study, it is shipped as a library and works with the Oracle JVM. It is listed here due its similarity with other solutions in user friendliness and ability to translate Java bytecode into OpenCL programs. Since the kernel creation, data transfer between host and device, etc. are automated, Aparapi does an exhaustive copying of buffers between CPU and GPU to ensure that the required data is moved to the GPU prior to kernel execution and the results are copied back to CPU before Java execution resumes [5]. This can adversely affect the efficiency of the program and can be considered as a disadvantage. The project is supported by AMD and earlier Aparapi versions were locked to AMD GPU and AMD APP SDK. Now Aparapi

claims to work on any platform that supports OpenCL 1.1 and is tested on all AMD OpenCL enabled devices and a limited set of NVidia GPUs on various Operating Systems [5]. Though Aparapi is one of the popular Java GPGPU solutions, there are certain limitations that prevent more developers from adopting Aparapi, such as its inability to support multidimensional arrays [5], no support for volatile memory [5], little support for vector execution (SIMD) on CPU [22], and limitation of one kernel and one run() method per class.

JaBEE

In [7] Zaremba et al. introduces a Java Bytecode Execution Environment (JaBEE) that supports dynamic dispatch, object creation and encapsulation on GPUs. The major difference between JaBEE and other GPGPU solutions is its support for object-oriented constructs and its ability to create objects inside a GPU kernel. The JaBEE architecture is based on a hybrid model that allows to execute bytecode selectively on both GPU and CPU [7]. JaBEE consists of three major components such as: a base Java class called GPUKernel that provides Java interface for GPU code execution, an online compiler that selectively compiles Java bytecode to GPU code, and a memory management system that transfers data between CPU and GPU [7].

A JaBEE execution is initiated by an incoming call from Java code with a request to perform GPU Kernel execution [7]. At first, the control is passed to the GPUKernel object, which is a Java class instance. This provides an interface for GPU code execution and validation of correctness of a kernel launch request [7]. The GPUKernel then passes control to the native code. The JaBEE system collects necessary information from the native code by creating a collection of dependent classes required for compilation [7]. It then validates variable names and create maps of pointers that are needed in further phases of the execution process [7]. Once this is done, the JaBEE system passes control to the Compilation Unit, which compiles a bytecode into GPU assembly code [7]. Later, the control is passed onto the Memory Management Unit, which copies all the necessary data to the GPU and adjusts pointers by changing CPU memory pointers to GPU memory pointers [7]. After this, the GPU kernel is executed and once the execution is over, the Memory Management Unit copies results from GPU to CPU memory and the control is returned back to the Java program [7].

JaBEE is built on top of VMKit, which is an open source framework for building Virtual Machines [7]. The VMKit supports three modes of code execution such as, Ahead-of-time compilation, code interpretation, and Just-in-time compilation [7]. Using Ahead-of-time compilation, the VMKit can generate LLVM IR (Low Level Virtual Machine - Intermediate Representation) files which after linking against Java core libraries and native code generation become executable [7]. JaBEE uses this Java bytecode to LLVM IR compiler as the front end for its online GPU compiler and J3, which is a JVM implementation in the VMKit, for bytecode execution on CPU [7].

To execute kernels on the GPU, the developer should extend the GPUKernel class and define run() and start() methods. The GPUKernel class is an abstract class which contains a run() method, which is an abstract method whose implementation must be provided in a subclass, and a start() method, which is a final method that is implemented in GPUKernel [7]. The run() method corresponds to the code that will be executed on the GPU or in other words, a CUDA kernel. This is similar to what we saw in Aparapi [5] and traditional multithreaded program in which the developer extends Thread class. Before and after executing the kernel on GPU, JaBEE copies the entire object graph, i.e. 'this' object which calls the run() method, all objects passed as parameters to kernel method, all static fields reachable form run() method, and recursively all objects which are pointed at by already copied objects [7]. The benchmark used in [7] to evaluate JaBEE is the Julia set computation. The authors use two versions of Julia set, one which uses only primitive data types on GPU, and another which does operations on objects. The results in [7] show that the CUDA C program executes 9.97X faster than the primitive data type Julia set computation Java program running on the J3 JVM, and the JaBEE version executes 4.15X faster than the Java J3 JVM version. The JaBEE version of Julia set computation which uses objects was 3.47X faster than the Java J3 JVM version, however, the Java program running on the Oracle JVM outperformed the JaBEE version of the code by executing 4.23X faster than the J3 JVM version code [7]. JaBEE facilitates processing of Java core library classes such as ArrayList, HashSet, SortedMap, etc. on GPU and hence, JaBEE opens the door for object oriented programming from Java on GPU.

Japonica

A compiler and runtime for making heterogeneous programming possible form Java called Japonica (Java with Auto-Parallelization ON graphIcs Coprocessing Architecture) is introduced in [8]. Japonica facilitates a Java program to scale transparently on GPU-based heterogeneous systems. The Japonica system includes a code translator, profiler, DO-ALL parallelizer and a speculator. Also, there is a scheduler which is a runtime component that balances the workload on the CPU-GPU heterogeneous platform [8]. The code translator statically analyzes and count data dependencies inside target loop, flagging the loops that have deterministic dependencies which can be found by static analysis and dynamic dependencies which can be found only at runtime and require further analysis by the profiler [8]. Several APIs needed for profiling are also inserted into the code at the code translation stage. The profiler gathers dynamic information about the loops that are flagged by the translator by executing them on the GPU in parallel [8]. It also performs analysis of intra-warp and inter-warp memory access dependencies and use this to compute the dependency density with the help of a quantitative model [8]. The data and control flow dependencies between loops are represented by the program dependency graph (PDG) which the scheduler can use to exploit the task-level parallelism [8]. The DO-ALL parallelizer create parallel versions (CUDA for GPU and Java threads for CPU) of the loops that do not have any dependencies [8]. The DO-ALL parallelizer in Japonica also injects optimized communication functions into the code to facilitate data transfer between CPU and GPU. Once the profiling is done, the speculator, or speculative execution engine, performs speculative execution of loops on the GPU by inserting function calls to a light weight Thread-Level Speculation (TLS) library which detects and recovers miss-speculation [8].

Japonica's task scheduler provides two dynamic task scheduling schemes, namely task sharing and task stealing. Task sharing is the scheduling scheme in which the workload of a task is being shared across CPU and GPU according to the computational complexities of the processors and the dependency density of the task [8]. Task stealing scheduling scheme resembles traditional master-slave model in which the CPU and GPU have their respective queues, and the one who empties its queue sooner can steal tasks from the other queue [8]. The scheduler distributes tasks across CPU and GPU based on the inter-task dependency

derived from PDG [8]. The compile-time components of Japonica take annotated loops as input and parallelize them with both analysis and dynamic profiling [8]. Though Japonica automates the kernel creation and load balancing between CPU and GPU, the developer should identify the loop to be parallelized and specify number of threads, scheduling scheme, variables to be copied, etc.

The study in [8] also evaluates the performance of Japonica by executing several benchmarks including dense matrix multiplication, matrix transpose, vector add, breadth first search, Black Scholes, cryptographic algorithms, etc. The experiments were conducted on two Intel Xeon X5650 CPUs with 12 cores operating at a frequency of 2.66 GHz and one NVidia Fermi M2050 GPU with 448 cores and 14 stream processors. The shared scheduling scheme could achieve 1.12X to 5.33X speed up for vector addition, breadth first search and matrix transpose compared to the 16-thread CPU and GPU only implementation [8]. The study in [8] not only shows the benefits of Java GPGPU but also explains the additional advantages that can be achieved by utilizing both GPU and CPU with a proper scheduling mechanism.

c) Analysis of Java GPGPU approaches

The above sections explained several solutions [2]–[8] that can be employed for general purpose computation on GPU from Java. JOCL [3], jCUDA [4] and JavaCL [6] are the Java bindings for GPU programming languages explained in the study. Table 3.3 explains these bindings along with their advantages and disadvantages. The table also shows the low-level language each of these bindings run on the GPU and the interface used to invoke the native code. JOCL [3] and JavaCL [6] are Java bindings for OpenCL whereas jCUDA [4] translates Java programs into CUDA programs. The metric developed and shown in Table 3.3 is later used in our research for deciding on a suitable Java binding for porting to embedded platform.

This study also analyses several user-friendly Java GPGPU solutions such as Rootbeer [2], Aparapi [5], JaBEE [7] and Japonica [8]. Table 3.4 shows these solutions, their corresponding low-level GPU language, along with their advantages and disadvantages. Rootbeer, JaBEE and Japonica enable Java GPGPU by translating the Java program into PTX or CUDA binaries and executing on GPU. Meanwhile, Aparapi provides the same by translating Java

programs into OpenCL binaries at runtime. This tells us that Rootbeer, JaBEE and Japonica is supported only on CUDA enabled NVidia GPUs whereas Aparapi is more portable since OpenCL is widely supported by various GPU vendors. Another advantage of using Aparapi is its ability to execute programs on multicore CPUs in the case of a program failing to execute on the GPU. While Aparapi [5] provides mechanism to access object fields on the device using the run() method without allowing method calls on an object, JaBEE [7] provides support for object creation, dynamic dispatch, and passing objects between a host and a device. Only Rootbeer and Aparapi are available as open source programs, and JaBEE and Japonica are research works in progress. Though these user-friendly Java GPGPU approaches increase productivity, previous research [1] shows that these solutions have reduced performance compared to Java bindings.

Java Binding	Low-level Language	Interface	Pros	Cons
JOCL [3]	OpenCL	JNI	• Light-weight • Similar to OpenCL Programming	• Cumbersome Java program
jCUDA [4]	CUDA	JNI	• Supports NVidia Parallel Libraries • Similar to CUDA programming	• Supports only NVidia GPUs
JavaCL [6]	OpenCL	JNA	• Supports automatic Memory management	• Bulky • Program could easily run out of GPU memory

Table 3.3: Java GPGPU using Java bindings

Java VM	Low -level Language	Pros	Cons
Rootbeer [2]	CUDA	• Well tested • Supports almost all Java features	• Supports only NVidia GPUs
Aparapi [5]	OpenCL	• User-friendly • Supports NVidia, AMD and other GPUs. • Supports multi-threaded execution on CPU	• No support for multi-dimensional arrays • No support for volatile memory • Only 1 kernel and run() method per class
JaBEE [7]	CUDA	• Allow passing objects to GPU	• Supports only NVidia GPUs
Japonica [8]	CUDA	• Enables GPU-CPU heterogeneous programming from Java	• Supports only NVidia GPUs

Table 3.4: Java GPGPU using modified VM

The study in [1] compares various Java GPGPU implementations such as jCUDA [4] and Aparapi [5] with traditional Java and CUDA benchmarks. The authors run selected benchmarks from Scalable Heterogeneous Computing (SHOC) suite on the NVidia Fermi GPU. The benchmarks selected are Matrix Multiplication, FFT and Stencil 2D. The results show that the maximum increase in GFLOPS achieved for single precision floating point operations by employing jCUDA is 97.47%, whereas using Aparapi this is 89.52% compared to the Java version [1]. The double precision floating point operation performance for jCUDA and Aparapi drops to 76.02% and 66.52% respectively [1]. The results in [1] show that Java bindings provide better performance compared to the GPGPU solutions that modify JVM. It is shown in [14] that GPUs are more energy efficient than CPUs in performing FFT operations when the number of FFT points are more than 16K. Rootbeer [2], Aparapi [5], JaBEE [7], Japonica [8], JOCL [3], jCUDA [4], and JavaCL [6] provide tools to offload Java

programs on GPU. However, these tools are developed and tested only for OpenCL full profile running on a Desktop GPU. The studies that evaluate embedded GPGPU [13], [16], [23] suggest that gains can be made on both power consumption and execution time by using mobile GPUs in general applications. However, these studies are all made from native languages like C or C++ and not from a language like Java that runs on a virtual machine. This justifies the relevance of our research and more details regarding this are explained in the next chapter.

The solutions [1]–[8] available in the literature are for accelerating Java programs on desktop or server GPUs and they are not tested on an embedded platform. Other studies [13], [14] evaluate the energy and performance gains that can be achieved by using embedded GPUs. However, these studies do not address the acceleration from Java. Issues like JNI overhead, limited memory, limited processing power, etc. are to be addressed while accelerating Java on embedded GPUs. Another major difference between the desktop and embedded GPU is their architecture difference. While workstation GPUs are connected to the host CPU via PCI, the embedded GPUs share the same heterogeneous chip with the CPU and share the same physical memory space for CPU and GPU. In such architectures, the kernel execution consumes more time than that of the host and device communication time. Though the implementations in [13]–[16] deal with GPGPU on embedded platforms, they does not mention how to accelerate Java on embedded GPUs. Our work is to validate the performance and energy efficiency that can be achieved by offloading compute-intensive portions of Java programs on embedded GPU.

Java is the most popular language used on embedded platforms and we need to analyze the potential of accelerating Java programs on embedded GPUs. The studies [1]–[8] show that both energy and performance efficiency can be achieved by offloading compute-intensive portions of Java programs on desktop and server GPUs. If the same is possible using embedded GPUs, then it can cater for the needs of today's world of power and performance hungry tablets and smartphones. As far as we know, there is no study conducted to explore the advantages or disadvantages of offloading Java programs on embedded GPUs. The major objective of our research is to explore more in this direction and the next chapter will discuss in detail about the problem statement and justification for the research.

Chapter 4 : Problem Statement and Justification

This chapter defines the research questions that will be addressed by this work. It also explains the scope of the research to set bounds for the problems that can be solved at the end of the research. The knowledge gained by analyzing the current literature is used to define the primary and secondary goals that have to be achieved by this research. The latter section of this chapter is to justify the relevance of the research questions and to show that the existing state of the art does not address them. The chapter ends by describing the benefits that can be achieved by addressing the research questions.

i. Problem Statement

Embedded GPUs are omnipresent in today's market-leading smartphones and tablets. However, there is not much work done to explore the potential of using embedded GPUs for General Purpose Computing. Our research is to evaluate the performance and energy gains achieved by offloading compute-intensive portions of Java programs on embedded GPUs. To address this issue, we have formulated 3 primary research questions, and a set of 5 secondary research questions. Following are the primary research questions addressed by this research.

1) Is acceleration using embedded GPU possible from Java?
2) Can Java-bindings deliver the same acceleration compared to native implementations?
3) Is there a reduction in energy consumption achieved by accelerating Java on embedded GPU?

While addressing these primary research questions, it is important to explore and analyze the possible and existing solutions in the literature that can be employed for embedded GPGPU from Java. Such a study can be useful for industries and research associates who want to utilize embedded GPUs for solving their general purpose problems from Java. Keeping this in mind, the secondary research questions are formulated. These are:

a) How can we accelerate Java programs on embedded GPU?
b) Can we employ an alternative solution to solve the portability issue of JNI-OpenCL implementation?

c) What is a suitable candidate Java-binding for OpenCL on embedded platforms?
d) How can we port the desktop implementation of Java-binding to the embedded platform?
e) Can the acceleration be achieved for existing modules of a library used in industry?

The secondary research questions are dependent on the primary questions and the answers can change based on the solutions that we employ to solve the primary questions. We start the research by addressing the secondary research question (a) to see how we can accelerate Java programs using embedded GPUs. A solution for this is to invoke OpenCL host code from Java program using JNI. Here, we profile and identify the compute-intensive portions of the Java program that has the potential for acceleration using GPU. Once this snippet is identified, we replace this compute-intensive portion in the Java program with a JNI call that invokes the OpenCL host code. This host code will launch kernels on the embedded GPU and pass the results after computation to the Java program which then resumes execution. This way, we could reuse the existing GPU accelerated programs from Java without needing to rewrite them. A detailed description of this JNI-OpenCL method is given in Chapter 5.

The major disadvantage of using JNI is that the binaries of the OpenCL host code invoked from Java are hardware specific and would kill the portability of the Java program. Here, we address the secondary research question (b). This portability issue can be addressed by employing the alternative Java GPGPU techniques explained in Chapter 3. The two existing solutions are to use Java-bindings or to use modified JVMs. In this research we explore the Java-bindings approach as the existing modified VMs face issues such as limited support for OpenCL (Rootbeer [2], JaBEE [7] and Japonica [8]) and lack of support for several Java features (Aparapi [5]). Java-bindings create a wrapper set of Application Program Interfaces (APIs) that can cover the OpenCL APIs and can be used from a Java program. This way, we could write the entire OpenCL host code in Java using the Java-bindings. Though this solves the portability issue of the earlier implementation using JNI, Java-bindings comes with an expensive price of rewriting the existing OpenCL host code in Java. As there are advantages and disadvantages for both these approaches, we need to explore them both and let the programmer decide on which one to go for.

The next step of the research is to identify a suitable Java-binding that can be ported onto the embedded platform, which answers the secondary research question (c). The metrics shown in Table 3.3 in Chapter 3 are used to identify a Java binding suitable for embedded platform. JOCL [3] can be considered as a good choice due to its light weight and similarity to the original OpenCL standard. Once the candidate Java-binding is identified, the challenge is to port it onto the embedded platform, which is the secondary research question (d). This is addressed by cross-compiling the Java-binding with the ARM toolchain and Vivante OpenCL SDK provided, and optimizing the Java-binding for the embedded platform. Now that we have both the approaches, JNI and Java-binding, working on the embedded platform, we can evaluate and compare the performance that can be achieved by employing them. Thus, here we address a primary research question (2), which is to see whether Java-bindings can give the same performance as the JNI approach.

Once we evaluate the performance improvements that can be achieved by offloading compute-intensive snippets of Java programs on embedded GPU, we address the primary research question (3) where we explore and see if any energy savings can be achieved by doing this. This is an important concern as most of the devices available in the market that has embedded GPUs are battery operated. Hence, if we can achieve speedup while saving energy, this could be an additional advantage for using embedded GPUs from Java. If all these research objectives specified above can be addressed correctly, then it can be proved that performance and energy gains can be achieved by using embedded GPUs from Java.

However, this achieved acceleration is only a proof of concept and so far; we did not evaluate any modules that are used in the real-world consumer applications. Hence, we identify a compute-intensive and often used module of a popular library in the industry. The module identified for acceleration is Discrete Cosine Transform (DCT) and the library selected is Joint Photographic Expert Group (JPEG). We evaluate whether acceleration of DCT calculation is possible from Java on embedded GPU. Thus we can answer the secondary research question (e). If we can accelerate DCT and IDCT (Inverse Discrete Cosine Transform) while saving energy, then we can hope that a similar approach can be used to gain energy and performance efficiency for other library modules that are used in industrial applications. If we can achieve speedup using JNI-OpenCL or JOCL, then the primary

research question (1), which is to see if embedded GPGPU is possible from Java, can be answered.

ii. Relevance of Research

Now that we have the research questions defined, we need to make sure that these questions are not addressed by any GPGPU solutions existing in the literature. Previous works explained in Chapter 3 deals with embedded GPGPU but only from languages like C or C++ and not from Java. Other studies [1]–[8] explain and evaluate the advantages of Java GPGPU, however, these studies are done on workstation GPUs and not on embedded platforms. As far as we know, this is the first research work done on evaluating the benefits that can be achieved by offloading Java programs onto embedded GPUs. Results in a previous study [13] show that the architectural difference between large-scale GPUs and embedded GPUs along with the memory and processing element limitations on embedded platforms play a vital role in the acceleration achieved. Since it is shown that the acceleration achieved using desktop GPUs does not guarantee the same using embedded GPUs, it is important that we explore and analyze the potentials of Java GPGPU on embedded platforms.

Table 4.1 explains the primary research questions, their relevance, solutions for answering the questions, and the benefits achieved by addressing these questions. Accelerating Java on embedded GPU is relevant for two main reasons: it has never been addressed before and it will enable GPGPU from Java on embedded platforms. The solutions that we use in this research work to offload Java on GPU are JNI-OpenCL and JOCL. If this research question is addressed correctly, it will enable Java programmers to use embedded GPUs for solving general purpose problems. Another advantage of this is that it helps in increasing the performance of Java programs. The next question described in Table 4.1 is about evaluating the efficiency of Java-bindings as opposed to the JNI-OpenCL and native OpenCL implementations. This is to explore and see if using Java-bindings will affect the acceleration efficiency. This can be done by profiling benchmarks developed in JOCL, JNI-OpenCL and native OpenCL separately and comparing their performances. The last question explained in Table 4.1 is to see if we can achieve any energy savings by offloading Java programs on embedded GPU. Most of the embedded GPUs available in the market are battery operated and thus it is important to analyze the reduction in energy consumption achieved, if any, with

embedded Java GPGPU. This can be done by analyzing the energy consumption of traditional Java (single-core and multi-core), GPU accelerated Java and native GPU programs, and comparing the results. A detailed description of how the power consumption is measured and how the energy consumption is calculated is given in Chapter 5.

Table 4.2 does analysis of the secondary questions of this research by explaining the solutions used to address the questions and by giving justifications for choosing the solutions. Here, we utilize the JNI feature of the virtual machine to accelerate Java on embedded GPU. This method is selected since it is supported by all the popular Java virtual machines and can be easily tested on desktop and embedded GPUs.

Research Objectives	Relevance	Solutions	Benefits
Accelerating Java on embedded GPU	• Not addressed before • Enable Java embedded GPGPU	• Java-bindings • JNI-OpenCL	• Better performance • Java programs can utilize GPU
Evaluate efficiency of Java Bindings	• Understanding effects of JNI Overhead	• Compare with Native implementation	• Preserve portability while achieving speedup
Analyze Energy consumption	• Devices with embedded GPU are battery operated	• Compare with Native and Java	• Energy Savings

Table 4.1: Analysis of Primary Research Questions

Secondary Questions	Solutions	Justifications
How to accelerate Java on embedded GPU	• JNI-OpenCL	• Existing solution • Could give good performance on desktop GPUs
How to solve portability issue of using JNI-OpenCL	• Use Java GPGPU solutions in Chapter 3	• Preserve portability
What is a suitable Java-binding for embedded board	• JOCL [3]	• Light-weight
How can we use JOCL on the embedded board	• Optimize • Cross-compile	• We need ARM binaries to run on the board
How can we accelerate modules from existing libraries	• Identify a library and module, then offload to GPU	• To make sure that useful modules can be accelerated

Table 4.2: Analysis of Secondary Research Questions

iii. Scopes of Research

The scope of this research is limited to the following:

1. The study only analyzes the potential benefits achieved by offloading Java programs on embedded GPUs.
2. The study evaluates existing Java GPGPU methodologies on embedded platform and does not introduce a new solution.
3. Only a limited set of GPGPU solutions, JNI-OpenCL and JOCL, are tested in this research.
4. Only a limited set of benchmarks are used in the study.
5. Only one embedded platform is used in the research.
6. The work is limited to Vivante GPU and OpenCL SDK.
7. The experiments are done only in Linaro.

This chapter explained the primary and secondary research questions which will be addressed by this research. The chapter also explained the relevance of doing the research and it sets the bounds of the work by explaining its scope. The methodologies and experiments setup used in this research is explained in the next chapter.

Chapter 5 : Methodologies and Experiment Setup

This chapter explains the methodologies and experiment setup employed in addressing the primary and secondary research questions defined in the problem statement section. The two approaches used to accelerate Java on embedded GPU, JNI-OpenCL and Java-bindings, used in this research are explained here. This chapter also explains the benchmarks which are used to establish the proof of concept. The architecture of the embedded board used in this research along with the APIs used for profiling the benchmarks are explained in this chapter. It also elaborates the energy measurement equipment and techniques used in this research.

In our research, the feasibility of accelerating Java programs on embedded GPU was evaluated by offloading Alpha-blending on Vivante GC2000 GPU of the Freescale i.MX6Q SabreLite board using JNI-OpenCL. Once we establish that Java programs can be accelerated by employing embedded GPU, Mandelbrot set computation benchmark is also offloaded onto the embedded GPU using JNI-OpenCL to evaluate the acceleration achieved. A JOCL version of the benchmark is also developed and run on the embedded board to solve the portability issue. Once the acceleration was achieved by solving the portability issue, the focus of the research concentrated more on seeing if we can accelerate some modules used in industrial applications. Hence, we tried to see if Discrete Cosine Transform (DCT) and its inverse (IDCT) can be accelerated using the embedded GPU from Java. DCT is widely used in the Joint Photographic Expert Group's (JPEG) compression algorithm. Similarly, IDCT is used in the decompression of the stored JPEG images. Modern day smartphones with extreme high resolution cameras demand superior performance in encoding and decoding JPEG images. All these features made DCT a desirable candidate for our further testing. After establishing that the selected benchmarks can be accelerated using the embedded GPU by employing JNI-OpenCL and Java-binding, our focus was then turned into measuring the energy consumption. The energy consumption of the traditional Java, GPU accelerated Java and GPU accelerated native C or C++ implementations are measured. We could accelerate the DCT and IDCT implementations on the GPU by consuming less energy. Energy is a critical factor for battery operated devices like smartphones and tablets and our research establishes that energy savings can be achieved by employing embedded GPUs to solve compute-intensive portions of general purpose Java programs.

i. Methodologies

This section describes the solutions we use to offload Java programs onto the embedded GPU along with their advantages and disadvantages. Here we test two approaches to achieve acceleration, one by calling the OpenCL host code from Java using Java Native Interface (JNI) called JNI-OpenCL and other by using a Java binding for OpenCL called JOCL.

a) JNI-OpenCL

Java Native Interface is an interfacing feature provided by the JVM to facilitate executing native low level languages such as C, C++ or assembly from Java. We can use the same feature to invoke an OpenCL host code from a Java program. We start this approach by identifying the compute-intensive and data-independent portions of Java program and then replace them with a JNI call to the native C or C++ OpenCL host code. The OpenCL host code is invoked via JNI when a JNI call to the library is encountered during execution of the Java program. This OpenCL host code allocates memory on the GPU, manages the data transfer between the CPU and GPU, executes the kernels on the GPU, releases the allocated memory from CPU and GPU, and passes the results and control back to the Java program.

Figure 5.1 shows the basic outline for executing OpenCL kernels on GPU from Java using JNI-OpenCL. In the figure, one can see the quad-core ARM processor, device and host memories, and the Vivante GC2000 GPU. The Vivante GPU shares the same physical memory with the ARM processor on the i.MX6Q SabreLite board. In Figure 5.1, the host (ARM CPU) and device (Vivante GPU) memories are shown separately. This is because the OpenCL programming framework is generic and it demands copying between the host and device at the programming level. The figure also shows how the Java program access the OpenCL host code and executes kernels on the GPU. The block arrows shown in the figure represent the data transfer between different logical blocks in this approach.

The Java program runs on the JVM, which is running on the quad-core ARM processor. This Java program can access the OpenCL host code via JNI and natively execute it on the ARM processor. This native host code has access to the GPU main memory and it is used to

allocate memory on the device and to copy the vectors onto the GPU memory. The host code also schedules, queues, creates, builds and executes OpenCL kernels on the GPU. This host code also includes APIs to process JNI calls. Once the results are calculated and available in the GPU memory, they are then copied back to the CPU memory and are made available in the Java program using JNI. In the approach shown in Figure 5.1, the acceleration is achieved by manually calling OpenCL host code from Java using JNI. The OpenCL host code is cross-compiled with the ARM toolchain to generate a shared object library file. These pre-compiled host code binaries are executed when a JNI call to the accelerated algorithm is encountered from the Java program. The host code is a C or C++ program that facilitates the execution of OpenCL kernels on the GPU. This program executes natively on the ARM processor and it enables executing the accelerated version of the algorithm or kernels on GPU.

The major disadvantage of this approach is that the program executable contains processor-specific binaries. This means that the application is not portable since the host code is cross-compiled for the ARM processor. This is very much undesired as Java is platform independent, and using this approach makes the accelerated program locked to a particular hardware. If the developer wants to port the application to another platform, he or she needs to recompile the host code for the target platform. Another disadvantage is the extra overhead for developer, as he or she must write programs in Java, C and OpenCL to execute any logic on GPU. Besides, the OpenCL host code is different for every application developed and should be cross-compiled every time. Also, debugging of the program takes much time since this approach demands the developer to cross-compile the program for every correction or change made on the host code. All this adversely affects the time to develop and market a new product.

The major advantage of using JNI-OpenCL is its ability to reuse the existing OpenCL programs. While Java-bindings require complete rewriting of the existing OpenCL libraries in Java, JNI-OpenCL requires only a few additions to the existing OpenCL host code which are necessary for processing JNI calls. Consult Appendix B for JNI-OpenCL host code for DCT computation.

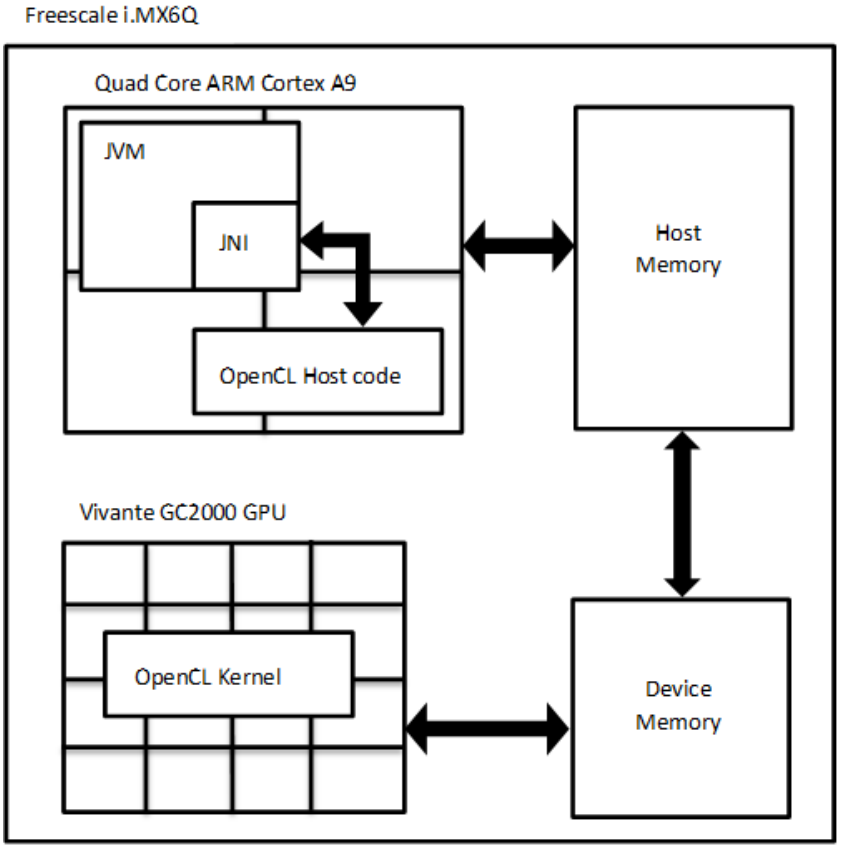

Figure 5.1: JNI-OpenCL

b) JOCL

JOCL is an open source library that offers Java-bindings for OpenCL [3]. JOCL source code contains native C and C++ programs which access OpenCL APIs along with Java programs that facilitate using the OpenCL APIs from Java applications [3]. In a JOCL program, a developer writes the OpenCL kernel, host code and the Java application in a single Java file. During the execution of the JOCL program, when an OpenCL wrapper API call is encountered, it is translated into a JNI call and the precompiled binaries are executed natively. The major advantage of using JOCL is that it preserves the portability of Java code. The programs developed using JOCL can run on any platform, provided the machine specific precompiled binaries are already in the JOCL library. Another advantage is that the developer can call OpenCL APIs directly from Java programs. This eliminates the C or C++ wrapper

development and cross-compiling steps, decreasing development time and effort. Another advantage of JOCL is that it is light weight and the compiled jar file can be easily shipped with the application. The JOCL APIs are very much similar to the original OpenCL APIs, and this make writing JOCL code easy for developers who are familiar with OpenCL programming. All these made JOCL a suitable candidate Java-binding to do Java GPGPU on embedded platforms. The JOCL program contains logics written in Java, OpenCL host code written using JOCL APIs, and other classes and methods in the same Java file. This makes the program cumbersome and can be considered as a disadvantage of using JOCL.

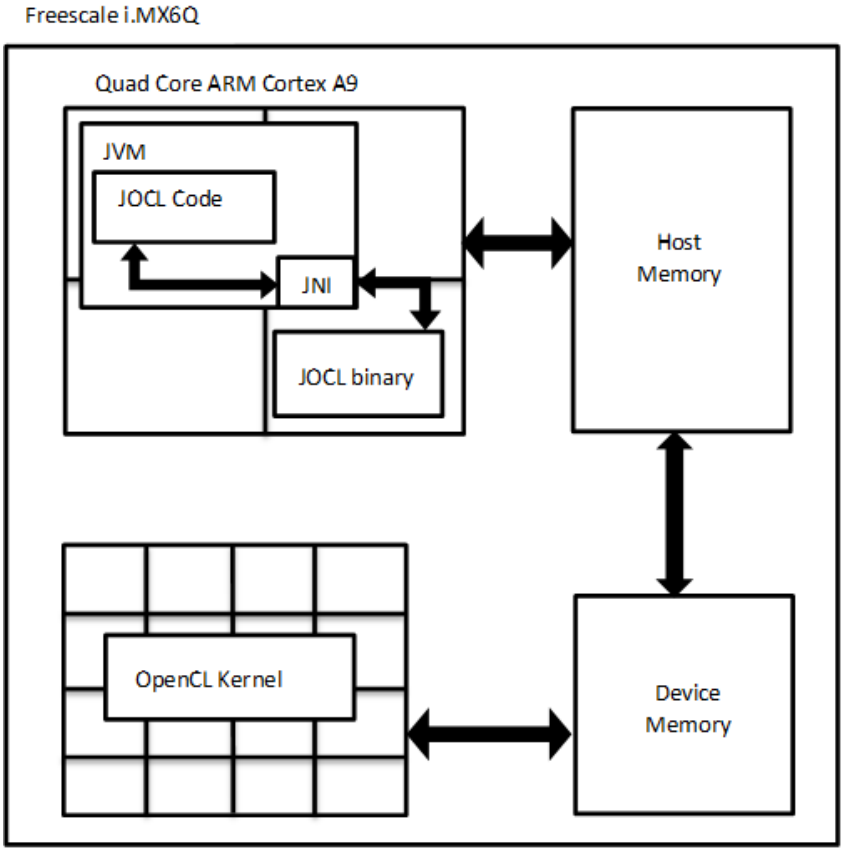

Figure 5.2: JOCL

Figure 5.2 explains the framework for accessing GPU using JOCL. It also shows the quad-core ARM processor, Vivante GC2000 GPU, and the host and device memories. In this figure also, the host and device memories are shown separately, which is not the case in Freescale i.MX6Q SabreLite board. The representation is to show how the movement of data and

control is happening during the execution of a JOCL program from a developer's point of view. The figure shows how the JOCL program running on the JVM accesses the precompiled binaries and executes kernels on GPU. The block arrows in the figure represent data transfers between various logical elements involved. The JOCL program can access the JOCL library which contain the binaries for OpenCL APIs used in the program. The JOCL program access and execute these binaries using JNI. Here, the host code is a Java program which has wrapper APIs to access the OpenCL APIs. These precompiled OpenCL APIs are executed natively on the ARM machine to allocate memory on the GPU, manage data transfer between the CPU and GPU memories, schedule, queue, create and execute OpenCL kernels on GPU. The JOCL code in Figure 5.2 contains the Java program, OpenCL kernel and OpenCL host code. JOCL runs on the JVM like any Java program and when OpenCL APIs are called, the machine specific JOCL binary is invoked from the library. These are then natively executed on the ARM processor. Consult Appendix C for a JOCL host code used for computing DCT.

These two approaches, JNI-OpenCL and JOCL, are used in our research to offload programs on the embedded GPU to achieve acceleration. Considering the advantages and disadvantages of both these approaches, we use these solutions to achieve better performance and energy efficiency. Once we offload the Java program onto the embedded GPU, the execution time and the energy consumption of the GPU accelerated Java program is compared to the traditional single-core and quad-core Java programs. It is also compared to the acceleration achieved on embedded GPU from C or C++ program to evaluate the effects of JNI overhead.

ii. Experiment Setup

The experiments in our research are carried out on a CPU-GPU heterogeneous Freescale i.MX6Q SabreLite board. The Freescale i.MX family processors are designed for low-power consumption and they have multiple processing elements such as CPU, GPU, video rendering machines, etc. on the same die. These processors are prevalent in industry, consumer and automotive domain. The i.MX6Q SoC encompasses a quad-core ARM Cortex A9 processor clocked at 1.2GHz per core, a Vivante GC 2000 GPU IP for 3D graphics, a GC 355 GPU for

2D or vector graphics, a Vivante GC 320 and a NEON SIMD accelerator [37]. Vivante GC 2000 IP passed OpenCL 1.1 Embedded Profile conformance testing and it facilitates OpenCL programming on the i.MX6Q SoC. The Vivante GC2000 GPU has one compute device (GPGPU core) with four compute units per device (shader cores) and four processing elements per compute unit [37]. This means that the GC2000 GPU can process $1*4*4=16$ elements at a time and thus the preferred work-group or thread-group size is 16. The maximum number of global work-items allowed in each dimension is 64K and the maximum number of work-items in each dimension per work-group is 1K [37].

The OpenCL Embedded Profile on GC2000 is a scaled down version of the Full Profile running on workstation GPUs, and thus it requires extra optimizations to utilize the embedded GPU efficiently. Some of these optimizations are to: take advantage of algorithm and data parallelism, overlap memory transfer from different levels of OpenCL memory hierarchy with simultaneous thread execution, and maximize instruction throughput and minimize instruction count [37]. The programmer should also maximize memory bandwidth and minimize data transfers since large transfers are more beneficial than many smaller transfers due to the impact of latency [37]. Besides these, choosing the correct execution configuration is also necessary to achieve superior performance using embedded GPUs. An important execution configuration is to use preferred multiple of work-group size. This means that the work-group size should be a multiple of the thread-group size and if not, several threads will remain idle and the developer cannot utilize the full potential of GPU [37]. As an example, if the work-group size is 8 on a GPU that supports 16 threads, then only half of the processing elements will be utilized and the other half is wasted. Another optimization technique is to use multiple work-groups of reduced size rather than having a large work-group. This will reduce the synchronization penalties and increase performance [37]. These guidelines are followed while developing applications on the Vivante GC2000 GPU to achieve better performance.

Figure 5.3 shows the Freescale i.MX6Q SabreLite board. The board has three display ports supporting HDMI, LVDS and RGB. In our research, we use the LVDS display to interface with the board. The operating system used to conduct experiments is Linaro Ubuntu provided by Freescale which has the Vivante OpenCL SDK. The arm-fsl-linux-gnueabi toolchain is used to cross-compile OpenCL host code and JOCL source code.

Vivante GC 2000 does not have hardware support for local memory which means that there is no on-chip memory that physically exists which can be used to share data between GPU cores. However, OpenCL framework is standard and it ensures consistency across various platforms and GPU vendors. Hence, the local memory is emulated in the global memory which is the system memory. Here, when the programmer transfers data between CPU and GPU, he or she is not moving data close to the GPU core; instead it only creates software overhead.

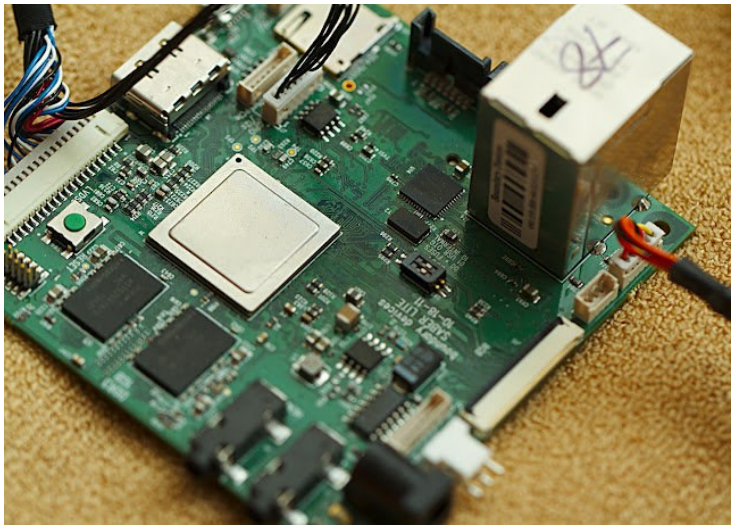

Figure 5.3: i.MX6Q SabreLite board

The profiling of benchmark programs is done using APIs available in C, OpenCL and Java. The OpenCL API, `clGetEventProfilingInfo()` in `cl.h` header is used to profile the time taken for data transfers between CPU memory and GPU memory, and executing kernels. The Java API `System.nanoTime()` is used to profile various portions of the Java program and to identify compute-intensive snippets that have the potential for acceleration. This API is also used to calculate the round trip time from Java which is the Java-C++-OpenCL-Java time taken for performing GPU acceleration including JNI overhead, OpenCL setup time, and kernel execution time. The `clock()` function in `time.h` is used to profile the native OpenCL host code written in C.

a) Setup for Measuring Energy Consumption

After profiling the benchmarks, we need to measure and analyze the power consumption of the ARM CPU and Vivante GPU while executing programs. However, there are no measuring points on the Freescale i.MX6Q SabreLite board to measure the CPU and GPU power separately. Consequently, the board was powered using a DC power supply through a resistor. The voltage drop across the resistor is measured to calculate the current using Equation 5.1. Here, I_{drawn} is the current drawn by the board and it is calculated from the measured voltage drop V_{drop} across the resistor R. This current is the total current drawn by the board and not by the CPU or GPU alone. Also, we need to find the current drawn for executing the benchmarks and not the total current.

In order to isolate the current drawn while executing the benchmark and the current drawn by the board while sitting idle, we need to measure the voltage drop, calculate and define a base current. This base current or I_{idle} is the current drawn by the board while sitting idle. Now we are ready to calculate the current drawn by the board while executing a benchmark. For this, first the current drawn by the board in idle state is calculated, and then the single-core, quad-core and GPU accelerated versions of the benchmark are executed on the board. The total current drawn by the board while executing the various benchmarks is calculated. The current consumed by the benchmark programs alone is calculated by isolating the idle current using Equation 5.2. The energy consumed by the board for executing the benchmark program is calculated using Equation 5.3.

$$I_{drawn} = \frac{V_{drop}}{R} \qquad \text{Equation 5.1}$$

$$I_{benchmark} = I_{total} - I_{idle} \qquad \text{Equation 5.2}$$

$$E = V * I_{benchmark} * t \qquad \text{Equation 5.3}$$

Here, E is the energy consumed by the board while executing the benchmark, V is the voltage supplied to the board, I_{idle} is the average current drawn by the board in idle state, I_{total} is the total average current drawn by the board while executing the program, $I_{benchmark}$ is the average current consumed for executing the benchmark program alone, and t is the time taken for executing the benchmark. The voltage drop across the resistor is measured using LeCroy WaveRunner 64xi Oscilloscope and this is used to calculate the current.

iii. Benchmarks

We use three benchmarks: Alpha-Blending, Mandelbrot set computation and Discrete Cosine Transform, to evaluate GPGPU computing from Java. Alpha-blending is used to evaluate the feasibility of offloading compute-intensive portions of Java code on embedded GPU. Once the feasibility is tested, Mandelbrot set computation is used to evaluate the speedup achieved using JNI-OpenCL and JOCL. We compare the performance of these two approaches to compare and identify the fastest. Mandelbrot set computation benchmark is selected in our research due to its popularity in GPU computing domain. Discrete Cosine Transform (DCT) is used in this research to represent a module from a popular library in industry. These benchmarks are used to address and evaluate the research questions defined in Chapter 4.

a) Alpha-Blending

Alpha-blending is the process of combining a translucent foreground color with a background color, thereby producing a new blended color. The degree of the foreground color's translucency may range from completely transparent to completely opaque. If the foreground color is completely transparent, the blended color will be the background color. Conversely, if it is completely opaque, the blended color will be the foreground color. The translucency can range between these extremes, in which case the blended color is computed as a weighted average of the foreground and background colors. The weight used in this

calculation is alpha and thus the name Alpha-blending. The value of alpha can range from 0.0 to 1.0, where 0.0 represents the fully transparent color and 1.0 represents the fully opaque color. In our Java program, first we read the two input images from file, and then we separate the Red, Green and Blue (RGB) components of these two images, latter Equation 5.4 is applied to each of the components separately to attain the blending. Once this blending is completed, the RGB components are combined to reconstruct the composite image and it is saved in a file. While executing the program, it is this separation of RGB components, blending, and reconstruction that is profiled and identified as the compute-intense portions of the program and it is offloaded onto the embedded GPU to attain acceleration.

$$\left.\begin{aligned} R_3 &= R_1 * \alpha + R_2 * (1-\alpha) \\ G_3 &= G_1 * \alpha + G_2 * (1-\alpha) \\ B_3 &= B_1 * \alpha + B_2 * (1-\alpha) \end{aligned}\right\} \quad \text{Equation 5.4}$$

Here, R_1, G_1, B_1 are the Red, Green and Blue components of the first input image, R_2, G_2, B_2 are the RGB components of the second input image, and R_3, G_3, B_3 are the RGB components of the resultant composite image. Also, α is the blending factor that determines the transparency or opacity of images. A pixel in an RGB image is 24 bits long with the various components stored in an order with the least significant 8 bits (0-7) for Blue, the next 8 bits (8-15) for Green, and the most significant 8 bits (16-23) for Red. Before processing, we need to separate these components and after processing, we should recombine the new RGB components in order to make the composite image. We use bit shifting and masking operations as shown in Equation 5.5 to separate the components.

$$\left.\begin{aligned} R &= Picture[pixel] >> 16 \,\&\, 0XFF \\ G &= Picture[pixel] >> 8 \,\&\, 0XFF \\ B &= Picture[pixel] \,\&\, 0XFF \end{aligned}\right\} \quad \text{Equation 5.5}$$

In Equation 5.5, $Picture[pixel]$ is a pixel in the image, and R,G,B represents the Red, Green, and Blue components of a pixel. While combining the components, the operation in Equation 5.5 is reversed as shown in Equation 5.6 and the composite image is reconstructed.

$$\left.\begin{aligned} Picture[pixel] &= R \\ Picture[pixel] &= Picture[pixel] << 8 \mid G \\ Picture[pixel] &= Picture[pixel] << 8 \mid B \end{aligned}\right\} \quad \text{Equation 5.6}$$

Here, R,G,B corresponds to the RGB components of the processed composite image. This benchmark is used in our research to test the feasibility of offloading Java programs on embedded GPUs to achieve acceleration.

b) Mandelbrot Set Computation

Mandelbrot set is a two dimensional fractal named after the mathematician Benoit Mandelbrot. Fractals are objects that display similarity at different scales, which means that magnifying a fractal will reveal features that are similar to the large scale version. Mandelbrot set is a mathematical set of points whose boundary is a distinctive and easily recognizable two-dimensional fractal shape [38]. Images of the Mandelbrot set are made by sampling complex numbers and determining for each whether the result tends towards infinity when a particular mathematical operation is iterated on it. Treating the real and imaginary parts of each complex number as image coordinates, pixels are coloured according to how rapidly the sequence diverges, if at all. The Mandelbrot set is popular for its aesthetic appeal and as a parallel processing benchmark [39]–[41]. Mandelbrot set is a group of complex numbers plotted in a complex plane, it can be explained as the set of all complex numbers C for which the sequence given in Equation 5.7 [38] remains bounded.

$$Z_{n+1} = Z_n^2 + C \quad \text{Equation 5.7}$$

Here, Z_n is the n^{th} term in the iteration and Z_{n+1} is the $(n+1)^{th}$ term, and the values of n are ranging from 1 to maximum number of iterations which in our case is 255. We start iteration by assuming $Z_0 = 0$ and use Equation 5.7 to generate the consequent terms. As we iterate through Equation 5.7, the value of Z changes every cycle, however we are more interested in finding the magnitude of Z. The magnitude of a complex number is its distance from origin when represented on a complex plane. This distance can be found using Equation 5.8.

$$| Z | = \sqrt{a^2 + b^2}$$ Equation 5.8

Here, the complex number, $Z = a + ib$ with a being real part and b being the imaginary part. The magnitude of Z can either stay equal to or less than 2 or it will eventually be greater than 2. Once the magnitude of Z surpasses 2, it will increase forever and is said to be unbounded. In the case of the magnitude staying less than or equal to 2, the complex number C is said to be the part of Mandelbrot set and if the magnitude becomes greater than 2, then C is not considered to be part of the set. We test every point in the 600x600 complex plane and paint the thousands of complex number points that are in the Mandelbrot set with one color and the ones outside the set with another. This will result in generating the Mandelbrot set fractal. We can also paint the points that are outside the Mandelbrot set differently. This can be done depending on how many iterations it took before magnitude of Z to surpass 2. Adding color will not only enhance the visual appeal of the fractal, but also highlights the parts of Mandelbrot set that are too small or otherwise not visible in the graph. This benchmark is used in our research to understand and compare the performance differences between using JNI-OpenCL and JOCL to offload compute-intensive portions of Java programs on embedded GPUs.

c) DCT and IDCT

Discrete Cosine Transform (DCT) [42] was proposed by Ahmed et al. in 1974. While Fourier Transform represents the signal as a mixture of sines and cosines, DCT expresses the signal as an expansion of cosine series only. DCT is popular due to its high energy compaction property, i.e. the cosine transformed signal can be easily analyzed using few low-frequency components [43]. DCT is used in the encoding/compression of JPEG images and its inverse IDCT is used in the decoding/decompression process of JPEG images. The DCT of a sequence $f(x), x = 0,1,2,...,(N-1)$ can be calculated by using Equation 5.9 [42], [43].

$$\left.\begin{aligned} C(u) &= \alpha(u)\sum_{x=0}^{N-1} f(x)\cos\left[\frac{\pi(2x+1)u}{2N}\right] \\ where,\ \alpha(u) &= \begin{cases} \sqrt{\frac{1}{N}}, & u = 0 \\ \sqrt{\frac{2}{N}}, & u \neq 0 \end{cases} \end{aligned}\right\} \quad \text{Equation 5.9}$$

Here, $C(u)$ is the u^{th} DCT coefficient and u ranges from 0 to N-1. Also, $C(0)$ is the mean of the sample and this is called the DC coefficient of the cosine transform, meanwhile other terms in the frequency domain are called the AC coefficients. The two dimensional variation of DCT, DCT8x8 is used in the JPEG compression algorithm [43], [44]. This is done by dividing the input signal, which in this case is a 2D image, into a set of several non-overlapping 8x8 blocks and then each of them are processed independently. This approach makes the block-wise processing of the image possible and it facilitates performing the transformation in parallel, which makes it a good candidate for processing on GPU. JPEG is the most widely used format in smartphones and tablets. In this work, we implement the two dimensional DCT8x8 on the embedded GPU. We get the 2D cosine transformation equations by extending Equation 5.9 into two dimensions. For an input sample size of NxN the 2D-DCT can be represented by Equation 5.10 [43].

$$\left.\begin{aligned} C(u,v) &= \alpha(u)\alpha(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\cos\left[\frac{\pi(2x+1)u}{2N}\right]\cos\left[\frac{\pi(2y+1)v}{2N}\right] \\ where,\ \alpha(p) &= \begin{cases} \sqrt{\frac{1}{N}}, & p=0 \\ \sqrt{\frac{2}{N}}, & p\neq 0 \end{cases} \end{aligned}\right\} \quad \text{Equation 5.10}$$

We can perform DCT of a vector of size N by using Equation 5.9. However, this can also be done as a two-step process, by pre-computing the cosine values of N vectors with N elements offline and storing them in a cosine matrix called A, and then by performing a dot product between A^T and the input vector X [43]. This is the more common method used for calculating DCT using computers. If we extend this 1D version of DCT into two dimensional space, then instead of using Equation 5.10, we can use matrix multiplication to calculate 2D-DCT of an NxN input image matrix. Since the input sample is a 2D signal or matrix of size NxN, we perform two 1D-DCTs on the input image matrix by transforming the rows and columns separately. This is expressed as a matrix notation in Equation 5.11. DCT8x8 is a limited case of the NxN DCT explained above. Here, since the input is divided into many small fragments of 8x8, we need to pre-calculate only 8x8 cosine values. This is an advantage when compared to calculating many cosine values and storing them. Equation 5.11 can be used to calculate DCT8x8.

$$C(u,v) = A^T XA \quad \text{Equation 5.11}$$

Here, $C(u,v)$ is the cosine transformed fragment of size 8x8, A is the 8x8 cosine matrix containing the cosine values, A^T is the transpose of A and X is an 8x8 fragment of the two dimensional input matrix for DCT. Here, one can see the Equation 5.10 is broken down into a series of many small matrix multiplications as shown in Equation 5.11. This is very much ideal for GPU acceleration and we offload these floating point matrix multiplications from

Java onto the embedded GPU to achieve better performance. DCT benchmark is used in our research to evaluate the energy and performance efficiency achieved by using embedded Java GPGU.

iv. Outline of Research

Figure 5.4 is the project flow diagram that explains various steps taken in this research to analyze the potential of accelerating Java programs using embedded GPUs. We start the research by analyzing literature to find similar studies existing in GPGPU, embedded GPGPU and Java GPGPU domains. The studies in GPGPU domain explain the relevance of GPUs in High Performance Computing, Cloud Computing, and other areas where desktop GPUs can be employed to achieve performance efficiency, and cost and energy savings. The embedded GPGPU studies concentrate on energy and performance efficiency that can be achieved by using embedded GPUs which are part of the SoC in tablets, smartphones, etc. These studies understand the memory and processing element limitations of embedded GPUs and try to take advantage of them. The Java GPGPU studies explain the potentials of using GPU from high level languages such as Java. Most of the studies in this area are open-source projects that utilize GPUs by modifying JVM or providing Java-bindings or wrappers for low-level GPU APIs. However, these studies are targeted, tested, and optimized only for desktop and workstation GPUs that are connected to CPUs via PCI, and not for embedded GPUs. Given the popularity of Java programming language and embedded devices, along with the presence of GPUs in them, it is necessary that we explore the potential Java GPGPU on embedded platforms. Our research is to analyze the benefits that can be achieved by offloading Java programs on embedded GPUs. Once we have analyzed the literature to build sufficient knowledge and understanding of current scenarios in GPGPU domain, we define the research questions that are addressed in our work. After defining the research questions, we configure the embedded platform, Freescale i.MX6Q SabreLite board, used in this research. This is done by installing Linaro-Ubuntu and Oracle JDK SE Embedded version, and Vivante OpenCL SDK on the board. After installing the required software and packages, we test the environment by executing sample Java, C and OpenCL programs on the embedded board.

Once we have the experiment setup ready and tested, we write the benchmark programs in Java and port them to the embedded board. These Java programs are then profiled to identify compute-intensive and data parallel portions that can be offloaded onto the embedded GPU to achieve speedup. These portions are then rewritten using OpenCL and is replaced with JNI hooks to the written OpenCL Library from Java. This is then executed and profiled on the Freescale board, and the performance of JNI-OpenCL version of benchmark is compared with the previous Java version. The performance efficiency achieved by offloading Java programs on embedded GPU is analyzed here. However, this JNI-OpenCL version of benchmark is hardware-dependent and would affect portability of Java program. Hence, we analyze the acceleration using JOCL, a Java-binding, to offload programs on embedded GPU. JOCL is used due to its similarities with the original OpenCL API and since it is light-weight. The JOCL source code is cross-compiled and optimized for the Freescale board using ARM toolchain and Vivante OpenCL SDK.

After porting JOCL onto the embedded platform, benchmarks are rewritten using the Java-binding, and executed and profiled on the Freescale board. The performance results achieved by employing Java-binding are compared with the JNI-OpenCL and native C-OpenCL implementations. Once we evaluate the acceleration from Java using JOCL and JNI-OpenCL, we focus on evaluating the energy efficiency achieved by embedded GPGPU from Java. The power consumption of traditional Java, GPU accelerated Java and native GPU accelerated versions of benchmarks are measured. The energy savings achieved by offloading Java programs on embedded GPU is calculated from the measured power. The performance and energy efficiency by enabling GPGPU from Java on embedded platforms are analyzed and the work is concluded.

Figure 5.4: Project Flow Diagram

We have discussed the research questions, the methodologies adopted to address the questions, the benchmarks used in this research, and the experiment setup for the work. Now we execute these selected benchmarks on the embedded platform, profile and calculate energy consumption to answer the problem statement defined in Chapter 4. The analysis of these experiment results are done in the next chapter.

Chapter 6 : Analysis of Results

This chapter discusses the results after executing various benchmarks on the i.MX6Q SabreLite embedded platform. The chapter analyzes the performance and energy efficiency achieved by offloading Java programs on embedded GPU by comparing the Java, JNI-OpenCL, JOCL, and C++-OpenCL implementations.

The JNI-OpenCL version of all the benchmarks requires the `LD_LIBRARY_PATH` to be set to the directory which has the cross-compiled OpenCL host code shared object file. In our experiments, this binary is kept in the same directory of the Java class file and thus we set `LD_LIBRARY_PATH` to the current directory. Both JOCL and JNI-OpenCL require the `LD_PRELOAD` environment variable to point at the OpenCL library file, this can be found in the OpenCL SDK installation directory. In our platform, the SDK is installed in /usr/lib and we configure the environment variable accordingly. Following is the export commands for setting these two environment variables.

```
$ export LD_LIBRARY_PATH=.
$ export
LD_PRELOAD=/usr/lib/libOpenCL.so:/usr/lib/libGAL.so:/usr/lib/
libCLC.so
```

i. Alpha-Blending Results

As a proof of concept and to establish the feasibility of offloading Java programs on embedded GPU, we accelerate the Alpha-blending of two 1024x768 images on the iMX6Q GPU. The first step is to execute the Java code for alpha-blending on i.MX6Q CPU and profile to identify the compute-intensive portions in the program. Profiling is done using the Java method `System.nanoTime()`. The time taken to perform alpha-blending and the time taken to read and write images were measured as the two time consuming operations. The read and write operations of images are part of the Java library and not part of the benchmark, hence we concentrate on accelerating the alpha-blending process. The alpha-blending logic in the Java program is replaced with a JNI call and it is rewritten in OpenCL.

The OpenCL host code is invoked from the Java program when the JNI call is encountered. This OpenCL host code then allocates memory on the GPU, copies the image matrices onto the GPU memory, launches the OpenCL kernels that have the alpha-blending logic, copies the results from GPU memory to CPU memory after execution, and passes the control and results back to the Java program. In this JNI-OpenCL version of alpha-blending also, we use the `System.nanoTime()` method to measure the time taken to perform alpha-blending. However, here the time taken for alpha-blending is the round trip time for Java-C++-OpenCL-Java which includes the JNI overhead, OpenCL setup time, data transfer time, and kernel logic execution time. The kernel logic execution time is measured in the JNI-OpenCL using the OpenCL API `clGetEventProfilingInfo()`. The benchmark is run ten times on the embedded platform to achieve consistent results. The results show that Java alpha-blending program took an average time of 316.03 ms with a standard deviation of 6.88 ms to execute on the ARM CPU. The accelerated JNI-OpenCL version of alpha-blending completed execution in an average time of 169 ms with a standard deviation of 12.23 ms on the Vivante GPU. This shows that the GPU accelerated version of the benchmark was 1.86 times or 46.35 % faster than the traditional Java program.

Following is a typical execution output for non-accelerated version of Alpha-blending Java program on i.MX6Q SabreLite board. The output shows time taken to read the two input images and the time taken to blend the images to create the composite image.

```
$ java alphablending
Time taken to read images 4314.49 ms
Time taken to blend images 244.67 ms
```

Following is a typical execution output for GPU accelerated JNI-OpenCL version of Alpha-blending on i.MX6Q SabreLite board. The output shows time taken to read the two input images, the number of OpenCL enabled platforms on the embedded board, the OpenCL kernel logic execution time, and the total round trip time taken to perform Alpha-blending.

```
$ java AlphaBlending
Time taken to read images 4310.61 ms
```

```
There are 1 platforms on this machine
Kernel Execution     36564000 ns
Time taken to blend images 170.01 ms
```

Table 6.1 compares the execution time of Alpha-blending using JNI-OpenCL and Java. It shows the time taken to execute the blending logic in Java, the round trip time in JNI-OpenCL and percentage of speedup achieved by using embedded GPU from Java.

Alpha-Blending program	Total Execution time (ms)	Acceleration (%)	Acceleration (times)
Java	316.03	N/A	N/A
JNI-OpenCL	169.55	46.35	1.86

Table 6.1: Alpha-blending using Java and JNI-OpenCL

Table 6.2 compares the Alpha-Blending algorithm logic execution time using Java and JNI-OpenCL. In the case of JNI-OpenCL, the logic execution time is only the time taken to execute the OpenCL kernel and the rest of the time (131.81 ms) in round trip time is due to data transfer, JNI overhead, and other OpenCL setup steps.

Alpha-Blending Program	Logic Execution time (ms)	Acceleration (%)	Acceleration (times)
Java	316.03	N/A	N/A
JNI-OpenCL	37.74	88.06	8.37

Table 6.2: Alpha-blending Logic using Java and JNI-OpenCL

Figure 6.1 shows the Alpha-Blending process when $\alpha = 0.5$. The two input images along with the actual composite output of the program is shown in the figure.

Figure 6.1: Alpha-blending input and output images

ii. Mandelbrot Set Computation Results

After testing the feasibility of offloading Java programs on embedded GPUs to achieve speedup and establishing the proof of concept, we now need to solve the problems that are inherent to the JNI-OpenCL method. The major disadvantage of this method is its hardware dependency and to solve this we employ JOCL, which is a popular Java-binding described in Chapter 5. Here, we analyze the difference in performance between using JNI-OpenCL and JOCL for GPGPU acceleration from Java. In JNI-OpenCL, the OpenCL host code which contains all the JNI processing steps, OpenCL setup and kernel is written in C++ or C. This is cross-compiled and executed on the Freescale i.MX6Q SabreLite board to analyze performance. After profiling the JNI-OpenCL version, the JOCL benchmark program which contains the Java program, and OpenCL host code and kernel in the same Java file is compiled and executed on the embedded board. Here, we execute and profile Java, JOCL and JNI-OpenCL versions of calculating 600x600 pixel Mandelbrot set computation on the embedded platform.

The API `System.nanoTime()` is used to profile the Java version of the benchmark. Meanwhile, the OpenCL API `clGetEventProfilingInfo()` is used to profile the JNI-OpenCL version of Mandelbrot set computation. In JOCL, a wrapper API for the OpenCL profiling API with the same name is created for profiling. The benchmark is run ten times on the embedded platform to achieve consistent results. The results show that the Java version of Mandelbrot Set computation took an average time of 2893.42 ms with a standard deviation of 36.12 to execute, whereas the JNI-OpenCL Mandelbrot Set computation program took an average execution time of only 368.51 ms with a standard deviation of 16.32 ms to execute. This shows that the JNI-OpenCL version of Mandelbrot set computation is 7.85 times or 87.26% faster than the traditional Java program. The JOCL program took an average time of only 399.37 ms with a standard deviation of 3.92 ms to execute Mandelbrot set computation on the GPU. This is 7.24 times or 86.20% faster than the Java version of Mandelbrot set computation. This shows that the acceleration attained by employing JOCL and JNI-OpenCL are comparable, with JNI-OpenCL being slightly faster.

A detailed profiling of the code is performed to identify the bottlenecks in the execution. The following is a typical output for non-accelerated version of Mandelbrot set computation

on i.MX6Q SabreLite board. The results show the time taken for generating complex numbers, time taken for calculating Mandelbrot set, and the time taken for painting the Mandelbrot set on display.

```
$ java Mandelbrot
Time taken for making complex numbers is 247.96 ms
Time taken to Calculate Mandelbrot Set is 2907.09 ms
Time taken to draw Mandelbrot Set is 1877.80 ms
```

The following is a typical output for accelerated version of Mandelbrot set computation using JNI-OpenCL on i.MX6Q SabreLite board. The detailed profiling shows the time taken to copy the two input vectors from CPU memory to GPU memory, Kernel execution time, and time taken to copy the resultant vector from GPU memory to CPU memory after execution. It also shows the round trip time from Java which includes the JNI overhead, and OpenCL setup and execution time.

```
$ java Mandelbrot
Time taken for making complex numbers is 248.91 ms
There are 1 platforms on this machine
Copy Time from CPU to GPU     :   10.49 ms
Kernel Execution Time         :  260.17 ms
Copy Time from GPU to CPU     :    6.74 ms
Time taken to Execute Host code and Calculate Mandelbrot Set
is 345.16 ms
Time taken to paint Mandelbrot Set is 1654.22 ms
```

As explained before, the potential of using JOCL for solving the platform dependency issue is also explored. The Mandelbrot set computation program is rewritten using JOCL and is executed on the embedded board. This version of the program takes 399.37 ms to execute which is 7.24 times or 86.20% faster than the non-accelerated Java program. The following is a typical output for accelerated version of Mandelbrot set computation using JOCL on

i.MX6Q SabreLite board. The output shows the time taken for generating complex numbers, the time taken for copying data from CPU to GPU and back, time taken for executing Mandelbrot set computation kernel on Vivante GPU, the total host code and kernel execution time, and the time taken to paint Mandelbrot set on the display.

```
$ java -classpath .:armJOCL.jar MandelbrotJocl
Time taken for making complex numbers is 292.84 ms
Waiting for events...
Time taken for creating, building and executing kernel is
299.97 ms
Copy Time from CPU to GPU     :   9.10 ms
Kernel Execution Time         : 259.48 ms
Copy Time from GPU to CPU     :   6.99 ms
Time taken to execute Host code and to calculate Mandelbrot
set on GPU is 387.77 ms
Time taken to paint Mandelbrot Set is 1376.45 ms
```

Table 6.3 compares the execution efficiency of Mandelbrot set computation using Java, JNI-OpenCL and JOCL. It shows the percentage increase in speedup and the ratio of the Java version to the accelerated version.

Mandelbrot set computation program	Total Execution time (ms)	Acceleration (%)	Acceleration (times)
Java	2893.42	N/A	N/A
JNI-OpenCL	368.51	87.26	7.85
JOCL	399.37	86.20	7.24

Table 6.3: Mandelbrot Set using Java, JNI-OpenCL and JOCL

Table 6.4 compares the Mandelbrot set logic execution using Java, JOCL, and JNI-OpenCL. The logic for calculating Mandelbrot set is written as an OpenCL kernel for both JNI-OpenCL and JOCL and the same is written as the part of program in Java. Since there is no time wasted on data transfer, the time taken for computing Mandelbrot set is same as logic

execution time in the case of Java, whereas this is the kernel execution time for JNI-OpenCL and JOCL. Here, the JOCL and JNI-OpenCL kernels executed 11 times or 91% faster than the Java program. This shows that an average of 139.98 ms for JOCL and 109.59 ms for JNI-OpenCL is wasted for data transfer and JNI overhead.

Mandelbrot set computation program	Logic Execution time (ms)	Acceleration (%)	Acceleration (times)
Java	2893.42	N/A	N/A
JNI-OpenCL	258.92	91.05	11.17
JOCL	259.39	91.04	11.15

Table 6.4: Mandelbrot Set logic using Java, JNI-OpenCL and JOCL

Figure 6.2 shows the Mandelbrot set computation output on Freescale i.MX6Q SabreLite board.

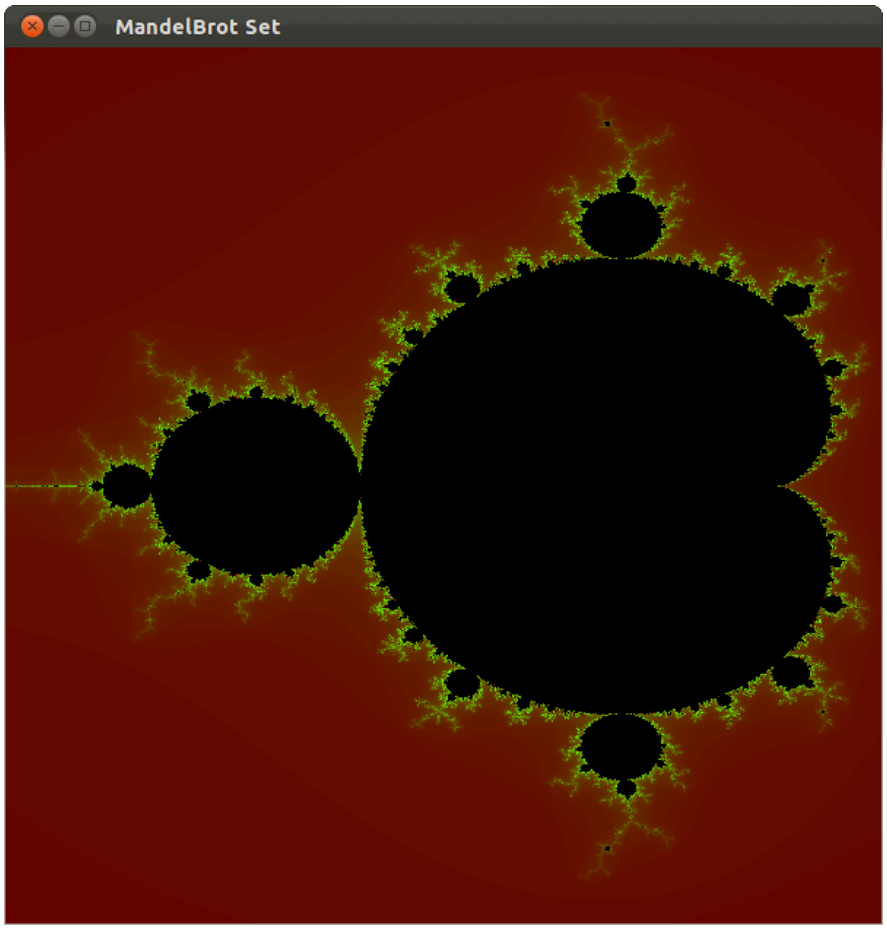

Figure 6.2: Mandelbrot Set Output

The results achieved after executing the CPU and GPU versions of Mandelbrot set computation are shown in Figure 6.3. The logic execution explained in the figure is the time taken for the Mandelbrot set kernel to execute, whereas the total execution includes the JNI overhead, time taken to copy data from CPU to GPU and back, etc. Since these overheads are not in Java, the total execution time and the logic execution time for the Java version is the same. The Java version of the Mandelbrot Set computation took 2893.42 ms to execute, whereas the JNI-OpenCL and JOCL Mandelbrot Set computation programs were 7.85X and 7.24X faster respectively. The OpenCL kernels of JOCL and JNI-OpenCL were 11X faster than the traditional Java program. This shows that JNI-OpenCL and JOCL can deliver similar acceleration by offloading Java programs on embedded GPU.

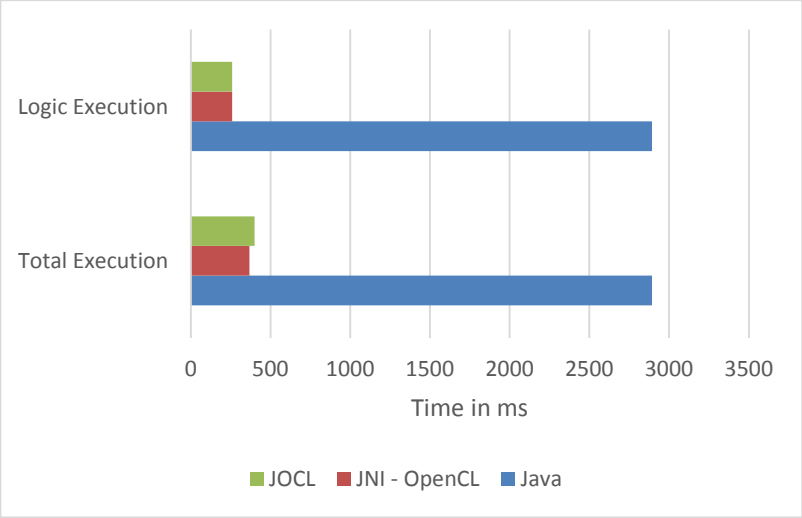

Figure 6.3: Performance of Mandelbrot Set Computation on CPU and GPU

iii. DCT Results

Discrete Cosine Transform benchmark is used in our research to evaluate and compare the acceleration achieved from Java and native language like C or C++ on embedded GPU. Here, we execute the two GPU accelerated versions of DCT, one from Java and other from C on i.MX6Q SabreLite board. We profile the benchmark using built-in APIs in Java, OpenCL and C to find the time taken for calculating DCT, memory transfers, kernel execution, etc. DCT is also used to analyze the energy efficiency achieved by offloading Java programs on embedded GPU. The method explained in Chapter 5 is used to measure the voltage drop

across a resistor while executing various versions of DCT benchmark and to calculate the energy consumption. A detailed performance and energy efficiency analysis of single-core, quad-core, Java-GPU, and native-GPU versions of DCT benchmark is done here. Consult Appendix A for the kernel used for calculating DCT, Appendix B for JNI-OpenCL host code for DCT computation, and Appendix C for JOCL program used for computing DCT on the embedded board.

The following is a typical output of a traditional single-core Java DCT benchmark. It shows the input matrix size, which in this case is 2048x2048 floating point values, and the time taken for computing DCT on the embedded board.

```
$ java DCT
The input size is 2048x2048
Calculating DCT took 3563.32 milliseconds.
```

Following is an output of multi-threaded Java DCT benchmark executed on i.MX6Q SabreLite board. This version of benchmark runs on the Quad-Core ARM Cortex A9 CPU for computing DCT. It shows the input size of $2048 * 2048 = 4194304$ floating point values, the threshold size, the number of processors available, and the time taken for computing DCT. The `-Xmx512m` parameter is specified while creating the JVM to dictate the maximum memory (heap size), which in our case is 512MB, the JVM can occupy while executing a program. If this parameter is not specified for larger problem sizes of DCT benchmark, the JVM crashes while executing and throws a `java.lang.OutOfMemoryError`.

```
$ java -Xmx512m ForkDCT

The input size is 2048x2048
Array size is 4194304
Threshold is 1398101
4 processors are available
Calculating DCT took 2502.25 milliseconds.
```

The following is the output of executing JOCL version of DCT benchmark on the embedded platform. It shows the input matrix size and the time taken for creating, building and executing kernels on GPU. The output also shows the time taken for transferring data from CPU memory to GPU memory and back. We can also see the OpenCL kernel execution time profiled using the API `clGetEventProfilingInfo()`. It also shows the complete DCT calculation time including the OpenCL setup and execution time measured from Java, which includes the querying and memory allocation for platforms and devices, creating command queue, program, and kernel object, building the kernel, and executing the kernel on the Vivante GC2000 GPU. Please see Appendix C for details regarding the JOCL version of DCT.

```
$ java -classpath .:armJOCL.jar JoclDCT


The input size is 2048x2048
Waiting on Kernel event...
Time taken for creating, building and executing kernel is
700.79 ms
Time taken to copy from CPU to GPU is   41.35 ms
Time taken to execute kernel is  644.29 ms
Time taken to copy form GPU ot CPU is  125.03 ms


The Complete OpenCL setup and execution time for DCT is:
1042.82 ms
```

The following is the output of GPU accelerated DCT benchmark from a native C program. The output shows the input size of 2048x2048, number of OpenCL enabled platforms available on the board, time taken for data transfer between CPU and GPU, and the kernel execution time. It also shows the total time taken for calculating DCT which includes the OpenCL setup time and kernel execution time.

```
$ ./imxDCT
The input Size is 2048 x 2048
There are 1 platforms on this machine
Waiting on Kernel event...

Time taken to copy from CPU to GPU is  37.52 ms
Time taken for kernel execution is  632.38 ms
Time taken to copy form GPU to CPU is  121.99 ms

The time taken to calculate DCT is 1.00 Seconds
```

Table 6.5 compares the total time taken for calculating DCT using various versions of the benchmark such as Single-Core Java, Quad-Core Java, GPU Java, and GPU C. For single-core Java, the total execution time is the time taken to execute the entire DCT method, and in the case of multi-threaded version of DCT, the total execution time includes the DCT method execution time along with the time taken to spawn enough threads that are needed for executing on the quad-core processor. The total execution time for GPU C program includes the OpenCL setup time, data transfer time and kernel execution time. In the case of GPU Java, the total time also includes the JNI overhead. The table contains a special case of the DCT benchmark where the problem size or input size is of 2048x2048 floating point values. It shows the average case of the execution outputs given above. In Table 6.5, the acceleration is calculated in comparison with the single-core version of DCT benchmark. The acceleration is shown as percentage increase in speedup and also as the ratio of single core to the accelerated version. It also shows the standard deviation of samples from the mean.

DCT Program	Total Execution time (ms)	Standard Deviation/σ (ms)	Acceleration (%)	Acceleration (times)
Single-Core Java	3580.91	4.85	N/A	N/A
Quad-Core Java	2486.85	8.67	30.55	1.44
GPU Java	1044.95	0.62	70.82	3.43
GPU C	1034	5.48	71.12	3.46

Table 6.5: Comparison of 2048x2048 DCT8x8 benchmark

Table 6.6 compares the logic execution times for single-core Java, quad-core Java, GPU Java and GPU C versions of DCT computation when the problem size is 2048x2048. The logic execution time for quad-core Java and single-core Java versions of the benchmark are same. However, for the GPU versions, the logic execution time is the OpenCL kernel execution time and it excludes the OpenCL setup time and data transfer time from total execution time. Table 6.6 shows the percentage of increase in speed up, the ratio between single-core Java and accelerated versions, and the standard deviation of execution times from mean.

DCT Program	Logic Execution time (ms)	Standard Deviation/σ (ms)	Acceleration (%)	Acceleration (times)
Single-Core Java	3580.91	4.85	N/A	N/A
Quad-Core Java	2486.85	8.67	30.55	1.44
GPU Java	643.84	0.13	82.02	5.56
GPU C	638.31	0.09	82.17	5.61

Table 6.6: Comparison of 2048x2048 DCT8x8 kernel logic execution

Figure 6.4 shows the performance results after executing DCT on the i.MX6Q CPU and GPU. Here, the single-core Java DCT, the quad-core Java DCT, and the DCT algorithm accelerated on GPU from Java and C using OpenCL are executed on the board. The time taken for calculating DCT is measured from the Java and C programs to evaluate performance efficiency. In Figure 6.4, the time shown for GPU execution includes the time taken for OpenCL setup, copying data from CPU to GPU and back, building the kernel and executing kernel. In the case of accelerating from Java, this time is the round trip time for Java-C-OpenCL-Java which also includes the JNI overhead. The results show that the traditional Java version of the benchmark is faster when the problem size is small and when the problem size grows, the performance gain achieved by employing a GPU increases.

For each problem size the benchmark is run five times on the embedded platform to achieve consistent results. When the input for DCT is a 2048x2048 floating point matrix, the traditional single-core Java version of the code executes in 3580.91 ms with a standard deviation of 4.85 ms, and the quad-core Java version is 1.44 times faster and execute in 2486.85 ms with a standard deviation of 8.67 ms, whereas the GPU accelerated version of the

Java code computes DCT 3.43 times faster compared to the single-core version and takes only 1044.95 ms with a standard deviation of 0.62 ms. Also, the kernel logic for this problem size executes in only 643.84 ms with a standard deviation of 0.13 ms. Here, by using the GPU, we can achieve a 3.43 times acceleration or 70.82% speedup compared to the single-core version of DCT. By employing GPU, we can achieve acceleration up to 2.38 times or 57.98% compared to the quad-core version of the DCT Java program. We can see from Figure 6.4 that when the problem size grows, the execution time of the GPU accelerated Java code is comparable to the GPU accelerated C program. This shows that for large problem sizes, the JNI overhead in accelerating Java programs on GPU has little effect on the acceleration efficiency. Also, from the figure we also can see that the Java GPU accelerated program and the GPU accelerated DCT from C or C++ are slower than single-core and quad-core Java versions when the problem size is small. This is due to the additional overhead in executing OpenCL APIs from Java and C along with the data transfer overhead.

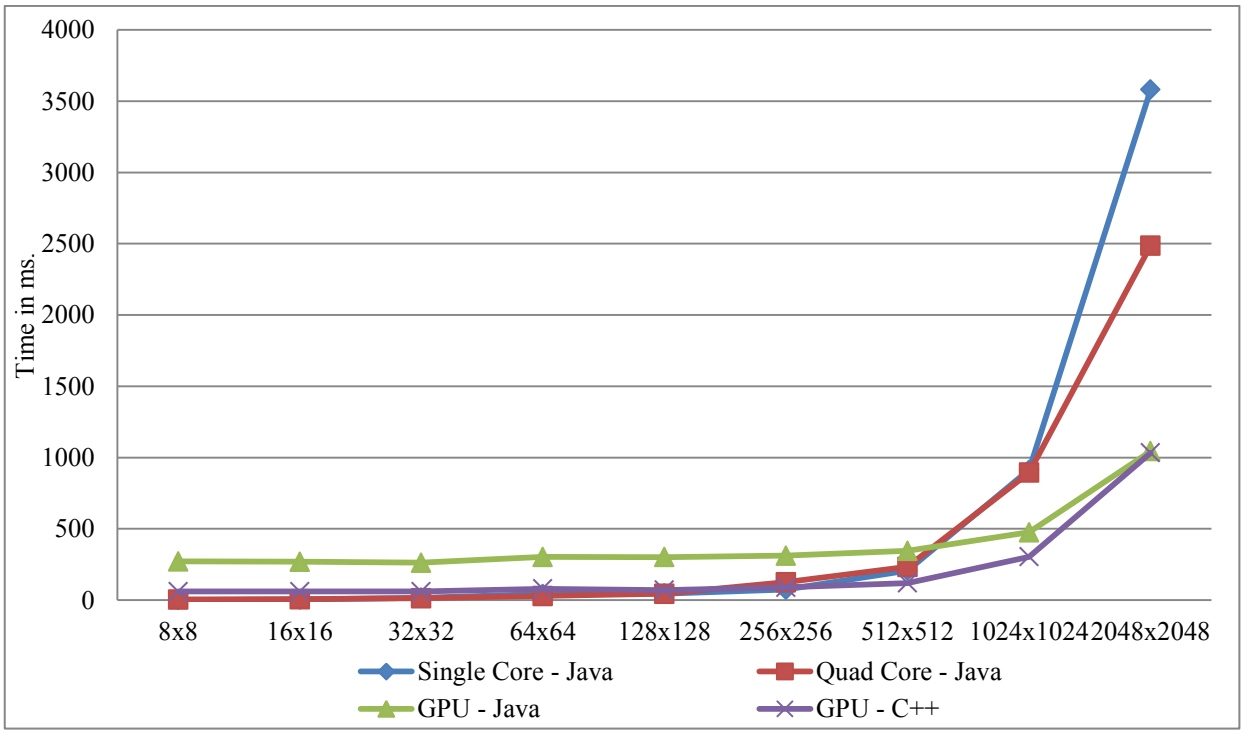

Figure 6.4: Performance of DCT8x8 on CPU and GPU

Table 6.7 compares the total execution time for different versions of DCT benchmark. Here, the problem sizes are varied from 8x8 through 2048x2048. The table analyzes the performance gain or loss happened by using various acceleration methods. The speedup is represented as a ratio of single-core Java version to the accelerated version. If this speedup ratio is greater than 1, it means that there is a performance gain and if this ratio is less than 1, it means there is a performance loss. The places where we get a performance gain are shaded in the table. From Table 6.7 one can observe that GPU accelerated versions are efficient for large problem sizes, the multi-threaded Java version is efficient for medium problem sizes, and the single-core Java version is better for small problem sizes. When the problem size is very small, the single-core version of DCT benchmark outperforms the GPU Java version by 180.67 times. However, this is due to the unwanted JNI and data transfer overheads, and also due to the severe underutilization of GPU compute resources. This trend diminishes when problem size grows and for larger problem sizes we can see the GPU versions outperforming single-core and quad-core versions of the benchmark. This is the same case whether the acceleration on GPU is done from Java or C. One of the requirements for acceleration using GPU is the availability of large number of elements to process; only then GPUs can utilize the data parallel execution methods for achieving speedup. This is the reason for reduced performance of GPU versions of DCT benchmark for smaller problem sizes.

Problem Size	Single-Core Java	Quad-Core Java	4-Core Speedup (times)	GPU-Java	GPU-Java Speedup (times)	GPU-C	GPU-C Speedup (times)
8x8	1.50	4.27	0.35	271.84	0.01	60.00	0.03
16x16	4.81	6.37	0.76	269.46	0.02	60.00	0.08
32x32	16.48	13.56	1.22	262.00	0.06	60.00	0.27
64x64	43.04	29.51	1.46	303.31	0.14	80.00	0.54
128x128	45.90	44.70	1.03	301.54	0.15	70.00	0.66
256x256	74.39	125.69	0.59	311.59	0.24	90.00	0.83
512x512	207.43	232.94	0.89	345.02	0.60	120.00	1.73
1024x1024	913.16	894.26	1.02	475.61	1.92	304.00	3.00
2048x2048	3580.91	2486.85	1.44	1044.95	3.43	1034.00	3.46

Table 6.7: DCT Total Execution Comparison

Figure 6.5 compares the logic execution time for various DCT implementations when the problem size is varied from 8x8 through 2048x2048 floating point values. The logic execution time is same as the total execution time for single-core and quad-core Java versions. However, this is different for the GPU Java and GPU C versions of DCT. Here, we exclude the OpenCL setup time and data transfer time, and consider only the kernel logic execution time. The execution times vary from very small values like 1.1 ms to large values like 3580.91 ms, and hence it is difficult to show these values in a regular graph. In Figure 6.5, these execution times are represented in a logarithmic scale to the base 10. The speedup is measured as a ratio of single-core or quad-core Java version to the GPU accelerated version. The results in Figure 6.5 show that the logic (kernel) of GPU accelerated versions from C and Java executes faster than the single-core and quad-core versions of the DCT benchmark.

Even though all the GPU kernels in Figure 6.5 execute faster than the Java only versions, we know from Table 6.7 that this is reflected in total execution only for large problem sizes. This is because the JNI and data transfer overheads overshadow the acceleration achieved by employing parallel processing elements in the GPU. We can see in Figure 6.5 that the GPU accelerated DCT kernel executes 34 times faster than the single-core version and 23 times faster than the quad-core version when the problem size is 64x64. However the total execution for GPU Java version of DCT when problem size is 64x64 is 7 times slower than the single-core Java version. This is due to the overhead caused by JNI calls and data transfers. Another interesting thing we can notice in Figure 6.5 is the variation of speedup ratio among different problem sizes. For small problem sizes, this ratio is small due to the underutilization of the GPU compute units. The speedup ratio then increases with the problem size, this is because there are enough data elements to be processed by GPU compute units and not too much data to make managing them difficult. Once the problem size reaches 512x512 and more, the speedup ratio saturates and stays around 5.

Here, we can see that the single-core Java version of DCT is faster than the quad-core Java version when problem size is 8x8 and 16x16. Once this size grows, the quad-core Java version achieves and maintains better performance than the single-core Java version of DCT benchmark. We can see from Figure 6.5 that the GPU accelerated versions of DCT benchmark from Java and C has lesser logic execution time compared to the single-core Java

and quad-core Java implementations. However, this trend is not followed for the total execution time and it can be justified by the overheads caused by JNI and data transfers between CPU and GPU memories as explained above.

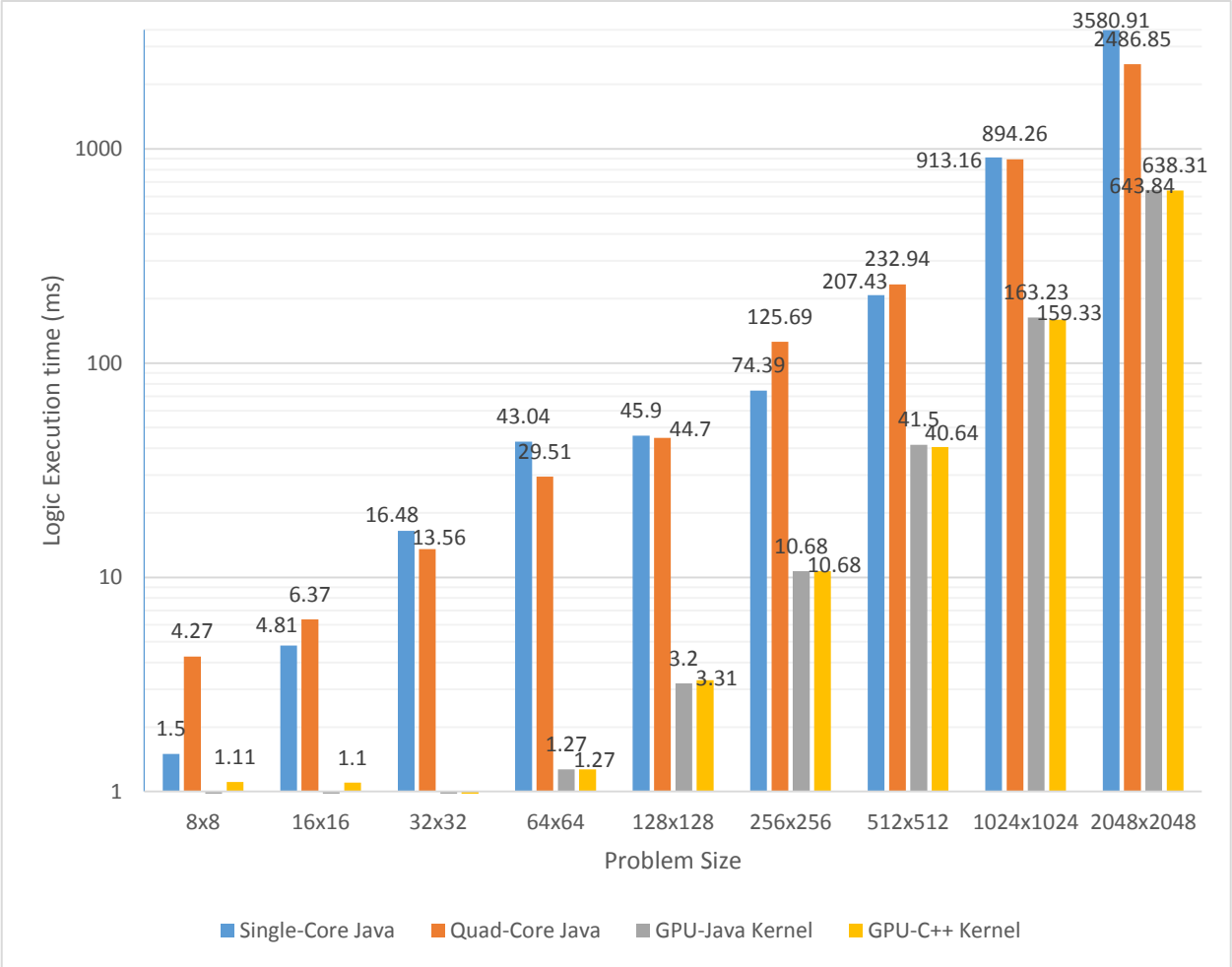

Figure 6.5: Logic execution time of DCT benchmark on CPU and GPU

Figure 6.6 shows the analysis of overheads for different problem sizes of DCT calculation. Here, we compare the total execution time of the GPU accelerated Java program with the GPU kernel execution time (also referred to as logic execution time). The total execution time includes the actual kernel execution time, the data transfer time, the OpenCL setup time, and the JNI overhead. Meanwhile, the logic execution time only shows the time taken for OpenCL kernels to execute on the GPU. The overhead time in Figure 6.6 is the difference between

total execution time and the logic execution time of the GPU accelerated version of Java program. If we analyze the overhead time taken for various problem sizes, we can see that only when the problem size is 2048x2048, the overhead is smaller than the kernel execution time. This shows that there are more computations being done than there are data transfers. This ensures that we get good acceleration. Another interesting observation is that there is significant difference between total execution time and kernel execution time only for problem sizes larger than 512x512. If we take a look into Figure 6.4, we can find that we get an acceleration using GPU only after this point. This emphasises the importance of large number of data-independent instructions that are required to achieve acceleration by employing GPU. The analysis in Figure 6.6 is only takes the DCT computation into account and the break-even point for other algorithms could be different. However, similar approach can be used to estimate and predict the problem sizes that are required to achieve acceleration by employing GPUs.

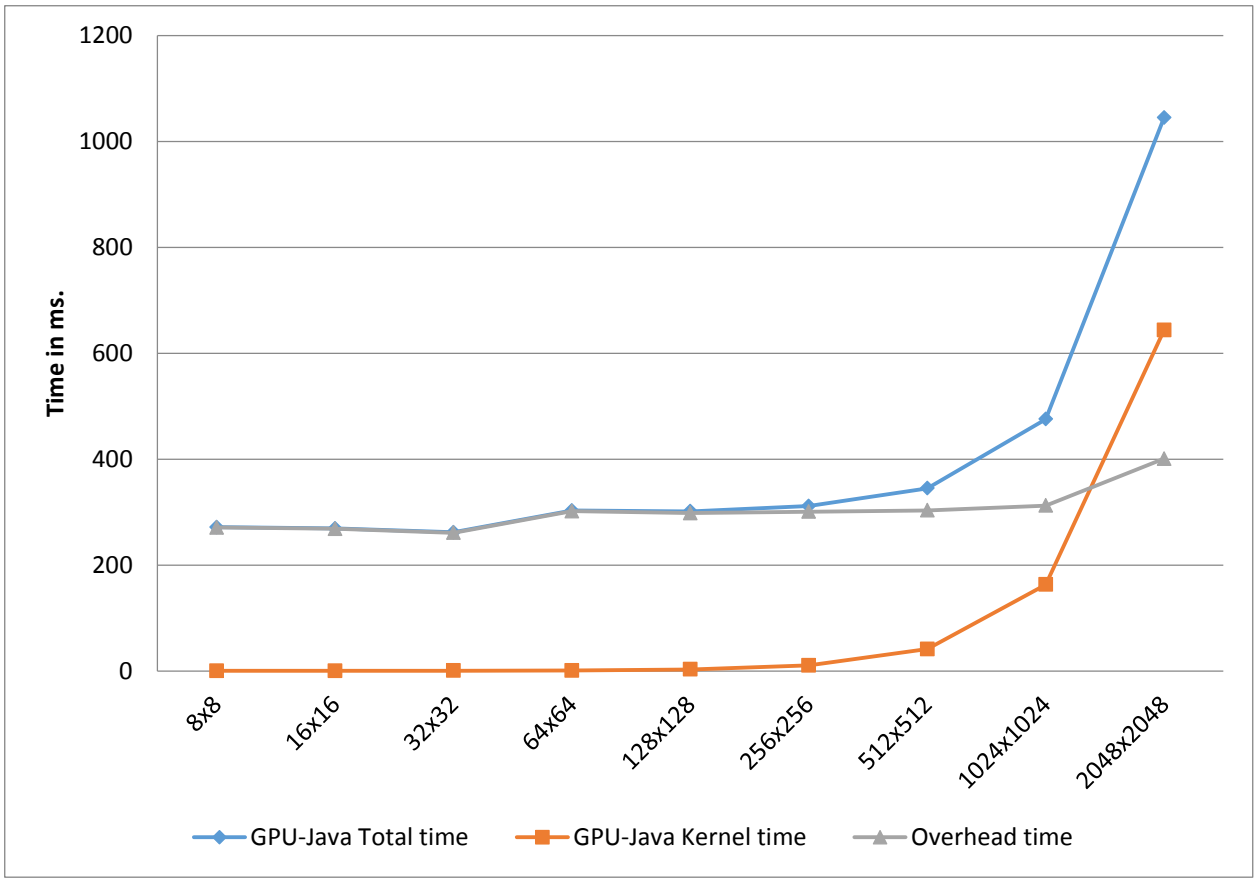

Figure 6.6: Analysis of Overheads

iv. Energy Consumption Results

As explained earlier, we are measuring the power consumption of the entire board and not the CPU and GPU separately. The LVDS display used in the research draws power from the board and the noise in the measured voltage is high that a base power could not be calculated. This requires that the display has to be removed in order to evaluate and analyze the power consumption of the board while executing benchmarks. Due to this drawback, the benchmarks like Mandelbrot set computation and Alpha-blending that utilize a display cannot be used to measure the power consumption. Hence, we removed the display and the DCT benchmark that does not require a display to execute is used to evaluate the energy consumption of the board.

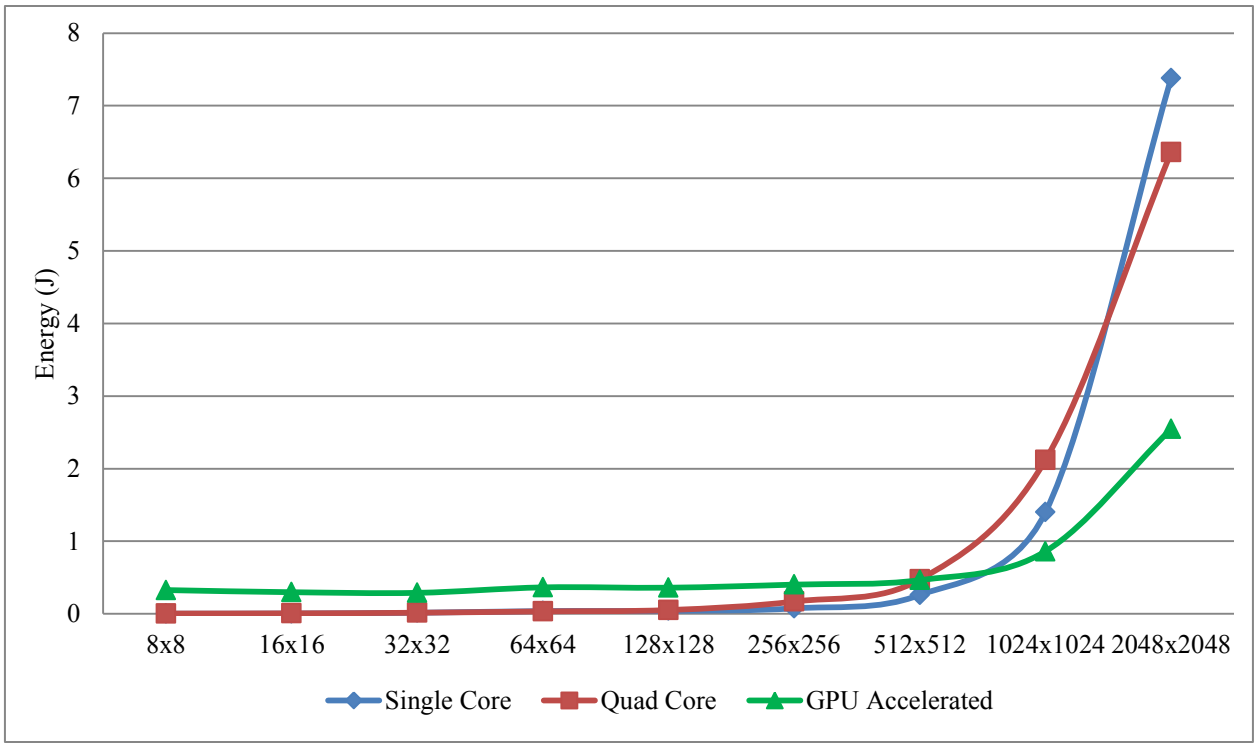

Figure 6.7: Energy Consumption Analysis

Figure 6.7 shows the energy consumption analysis of DCT execution for different input problem sizes. The voltage drop across the input resistor is measured and the energy is calculated as explained in Chapter 5. The energy consumption of traditional single-core Java, quad-core Java, and GPU Java are calculated and compared in Figure 6.7. The energy

consumption of traditional single-core Java version of DCT is 7.38 J when the problem size is 2048x2048, whereas the energy consumption of the quad-core Java version of DCT for the same problem size is 6.36 J which is 1.16 times lesser than the single-core version. The energy consumption of the GPU accelerated version of Java program for problem size of 2048x2048 is only 2.54 J. Here, by using the GPU, we achieve up to a 2.91 times or 65.58% less energy consumption compared to the single-core version and up to a 2.5 times or 60.06% less energy consumption compared to the quad-core version of DCT. From Figure 6.7 we can observe an energy gain for larger problem sizes. The GPU computation consumes more power than their CPU counterparts. However, this additional power consumption is balanced in GPU programs and energy efficiency is achieved by executing benchmarks with much less time than the CPU versions. This shows that the acceleration achieved by offloading DCT Java benchmark on embedded GPU can balance the power consumption by GPU and provide energy efficiency.

Figure 6.8 shows the output from oscilloscope after executing the traditional single-core Java program and the accelerated Java GPU versions of DCT for an input size of 2048x2048 floating point values. It shows the voltage drop across a ~0.7 Ω resistor. This measured voltage is used to calculate current consumed by the board while executing DCT benchmark. This however is the total current consumed, and the current consumed for executing benchmark alone is calculated by subtracting the base or idle current from this measured total. This is explained in Chapter 5. The energy consumed by the board while executing the DCT computation for various problem sizes is then calculated. From Figure 6.8, it is evident that the traditional single-core Java program which uses only the CPU consumes less power and takes much more time to execute compared to the accelerated GPU version of the Java code which consumes more power but takes a fraction of the time to calculate results. The y-axis in Figure 6.8 indicates the voltage drop measured across the resistor and the x-axis is the time taken for execution. The figure does not show execution times, but they are measured by profiling the Java and OpenCL programs.

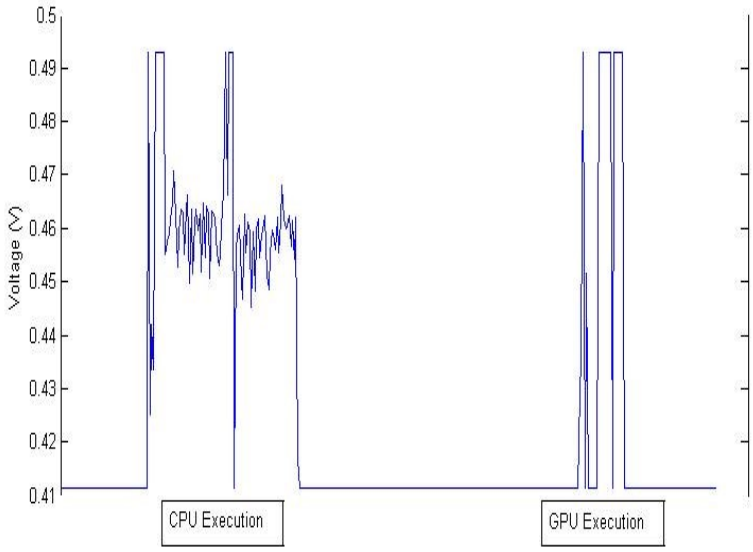

Figure 6.8: Voltage graph (Output from Oscilloscope)

v. Summary

Three benchmarks Alpha-blending, Mandelbrot set computation and DCT are offloaded from Java to embedded GPU in order to achieve speedup. Alpha-blending benchmark is used in this study to check the feasibility of embedded GPGPU from Java, and Mandelbrot set computation is used to compare the performance difference between JOCL (Java-binding) and JNI-OpenCL. DCT is used to explore the impact of different input sizes and to compare the acceleration achieved by using embedded GPU from Java and C (native language). DCT is also used to evaluate the energy efficiency achieved by offloading Java programs on embedded GPU. This chapter explains and analyzes the results which demonstrate that performance and energy efficiency can be achieved by employing embedded Java GPGPU. Next chapter concludes the research by explaining the how the results discussed here answers the research questions defined in Chapter 4.

Chapter 7 : Conclusion

This chapter summarises the work by explaining contributions of our research. The chapter also elaborates how the research questions defined in Chapter 4 are answered by this research. This chapter also mentions the challenges faced while doing research, their workarounds, and future research directions.

i. Summary of Work

The major objective of our research is to demonstrate the feasibility of accelerating Java programs on embedded GPU by achieving energy efficiency. We started our research by configuring the i.MX6Q SabreLite board with Linaro Ubuntu. After installing Linux, Oracle JDK SE Embedded is installed on the board, and then the Vivante OpenCL SDK is installed and tested using test examples. Once the experiment setup is configured, we test the feasibility of accelerating Java on embedded GPU by executing Alpha-blending benchmark. The JNI-OpenCL version of Alpha-blending which uses the embedded GPU could perform blending 1.86X or 46.35% faster than the Java version of benchmark. Now that we have established the feasibility of accelerating Java programs on embedded GPU, we explore other solutions in the literature and also try to solve the hardware dependency issue of JNI-OpenCL.

Mandelbrot set computation was used to evaluate the acceleration that can be achieved with and without using Java bindings for OpenCL. The experiments show a speedup of 7.9X using JNI-OpenCL and 7.2X using JOCL compared to the Java version of benchmark. This shows that similar performance can be achieved using traditional JNI-OpenCL approach and by using Java bindings. Today's smartphones, tablets and other consumer electronics with high resolution cameras and large displays demand large processing power to compress and decompress images. DCT and IDCT are integral parts of image compression and decompression algorithms and is one of the benchmarks used in this work. DCT is used in our research to compare the performance of GPU accelerated program form from Java and C. It is also used to evaluate the energy efficiency that can be achieved by offloading Java programs on embedded GPU. Accelerating the Java version of DCT by reducing energy consumption is the major milestone of this work. The results show that the single-core DCT Java code can be

accelerated 71% by consuming only 65.58% less energy by using GPU offload techniques. This approach can be used to offload other commonly used Java libraries to embedded GPUs to achieve speed-up and energy efficiency. The study also reveals that, for large problem sizes, the acceleration achieved by using embedded GPU from Java is close to that of the acceleration achieved from native C or C++. This work also evaluates the performance and energy gains by using Java bindings for OpenCL to offload programs on embedded GPUs. Our observations show similar speedup for GPU acceleration achieved with and without using Java bindings for OpenCL, also for large problem sizes, Java-bindings give the same performance as that of GPU accelerated code form C.

ii. Contributions of Work

Smartphones, tablets, and other devices with embedded GPUs are common in today's market. These battery-operated devices running Java programs demand high performance and low energy consumption. Our research can be considered as a solution to cater the needs of such devices. In this research, we could accelerate Java programs on embedded GPU while achieving energy efficiency. We also compare the acceleration achieved on embedded GPU from Java and native C program. This is useful in understanding the impact of JNI overhead in accelerating Java programs on embedded GPUs. Our work also compares the acceleration achieved by using JNI-OpenCL and Java-bindings. This work also provides a platform for exploring future embedded heterogeneous computing solutions from interpreted languages like Java.

Following are the major contributions of this research.

- Evaluated and achieved Performance gain using embedded Java GPGPU
- Evaluated and achieved Energy efficiency using embedded Java GPGPU
- Identified a candidate Java-binding, JOCL, and ported it to the embedded platform
- Evaluated and compared the Performance of Java-bindings with other Java GPGPU and native GPGPU implementations
- Demonstrated the Potential of Embedded Heterogeneous Computing from high level interpreted languages.

Now that we have highlighted the contributions of the research, we need to analyze and see if the research questions defined in Chapter 4 are answered correctly by the work. Table 7.1 explains the primary research questions, answers and how the questions are answered in this research.

Primary Questions	Answers	How it is Answered?
Is acceleration using embedded GPU possible from Java?	Yes	By offloading various benchmarks on embedded GPU and achieving speedup.
Can Java-bindings deliver similar performance as native implementations?	Yes	By comparing the performance of Java-bindings with JNI-OpenCL and C++-OpenCL.
Is there a reduction in energy consumption achieved by accelerating Java on embedded GPU?	Yes	By comparing energy consumption of Java and GPU accelerated versions of DCT.

Table 7.1: Answering Primary Research Questions

Table 7.2 explains the secondary research questions and answers.

Secondary Questions	Answers
How can we accelerate Java programs on embedded GPU?	Using JNI-OpenCL or Java-bindings such as JOCL.
How can we solve the portability issue of JNI implementation?	Use Java-bindings.
What is a suitable candidate Java-binding for OpenCL on embedded platforms?	JOCL, since its light-weight.
How can we port the desktop implementation of Java-binding to the embedded platform?	Cross-compile and Optimize for the embedded platform.
Can the acceleration be achieved for existing modules of a library used in industry?	Yes, DCT which is a module in JPEG compression is accelerated in this research.

Table 7.2: Answering Secondary Research Questions

Today, with the introduction of desktop and server grade fusion APUs such as AMD's Kaveri architecture that utilizes a fully shared memory between CPU and GPU, the modern day workstation processors look similar to heterogeneous embedded SoCs, but only with much more processing power. In the case of such systems where the memory is being shared between CPU and GPU, the communication bottleneck due to data transfer between the host and device is not the major problem but the kernel execution time itself is. This is the case with our Freescale SoC and our research can potentially be extrapolated to understand the acceleration capabilities of Java programs on such future processors.

iii. Challenges Faced

Several challenges were encountered while doing this research. The challenges along with their workarounds used to solve them are explained below.

- Android is the most popular smart phone and tablet OS, and it is very much desired to accelerate Java programs on Android platform. Despite all its advantages, Google is yet to release OpenCL drivers for Android. Hence, the Android ICS image from Freescale could not be used to perform the tests. The solution was to install Freescale Linux that has Vivante SDK with OpenCL drivers.
- Oracle Java with full AWT and swing support was not working as desired on Freescale Linux. The workaround was to install a version of Linux which gives full support of AWT features. Linaro-Ubuntu was installed on the board and we could execute full version of Java with AWT and swing features.
- Vivante GC2000 GPU has a limitation of 64K global work items in each dimension. This is an issue while performing Alpha-blending of images of size 1024x768, i.e. 786432 pixels cannot be processed in a work group. The solution was to perform Alpha-blending on tiles of data. So we divided the image into tiles of size 16 (Vivante recommends this work-group size for maximum performance [37]) and performed blending.
- JOCL does not have binaries to support ARM machines and was not tested for OpenCL Embedded Profile. The workaround was to cross-compile the JOCL source code using the ARM toolchain and to generate binaries for the embedded platform.

- There are no measuring points on the i.MX6Q SabreLite board to measure the power consumption of CPU and GPU separately. A work around was to connect the board to a voltage supply and measure the voltage drop across a resistor and calculate the power consumption of the entire board. This is done multiple times to find a base or idle power consumed by the board, and this base power is used to calculate the energy consumed by traditional Java and GPU accelerated Java versions of benchmarks.
- The benchmarks like Mandelbrot set computation and Alpha-blending that utilize a display cannot be used to measure the power consumption. Since we are measuring the power consumed by the entire board and not the CPU and GPU independently, the display can cause too much noise and a base power level could not be identified. The color, brightness, refresh rate, screen activity, all could affect the power consumed by the display and will overshadow the power consumption of the chip. A solution is to remove display and use benchmarks like DCT that does not require a display and feed commands through the serial debug port.

iv. Future Works

Our research is to establish the feasibility of accelerating Java programs using embedded GPUs and to explore the energy savings that can be achieved by embedded Java GPGPU. The future directions for this research include evaluating embedded GPU acceleration that can be achieved by employing Java GPGPU solutions that modifies the virtual machine. This includes projects such as Rootbeer [2], Aparapi [5], etc. Another future direction is to try automating the acceleration by analyzing Java bytecode and dynamically mapping the execution to GPU or CPU. This requires a bytecode parser which can look for parts of Java code that has the potential for acceleration and then direct the execution to GPU or CPU based on a cost function. Another future direction for this research is to accelerate Java libraries for encryption, image processing, etc. on embedded GPU. Now, we are manually identifying the data-independent and compute-intensive portions of the Java program. A possible future direction is to employ profilers or scripting languages to identify these portions. Accelerating Java programs using embedded GPU on other operating systems such as Android and IOS could also be a future direction of this research.

References

[1] J. Docampo, S. Ramos, G. L. Taboada, R. R. Exposito, J. Tourino, and R. Doallo, "Evaluation of Java for General Purpose GPU Computing," *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pp. 1398–1404, Mar. 2013.

[2] P. C. Pratt-Szeliga, J. W. Fawcett, and R. D. Welch, "Rootbeer: Seamlessly Using GPUs from Java," *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pp. 375–380, Jun. 2012.

[3] "jocl.org - Java bindings for OpenCL." [Online]. Available: http://www.jocl.org/. [Last Visited in January 2014].

[4] "jcuda.org - Java bindings for CUDA." [Online]. Available: http://www.jcuda.org/. [Last Visited in January 2014].

[5] "aparapi - API for data parallel Java." [Online]. Available: https://code.google.com/p/aparapi/. [Last Visited in January 2014].

[6] "javacl - OpenCL bindings for Java - Google Project Hosting." [Online]. Available: https://code.google.com/p/javacl/. [Last Visited in January 2014].

[7] W. Zaremba, Y. Lin, and V. Grover, "JaBEE: framework for object-oriented Java bytecode compilation and execution on graphics processor units," ... *Processing with Graphics Processing Units*, no. 2, pp. 74–83, 2012.

[8] G. Han, C. Zhang, K. T. Lam, and C.-L. Wang, "Java with Auto-parallelization on Graphics Coprocessing Architecture," *2013 42nd International Conference on Parallel Processing*, pp. 504–509, Oct. 2013.

[9] R. Suda, T. Aoki, S. Hirasawa, A. Nukada, H. Honda, and S. Matsuoka, "Aspects of GPU for general purpose high performance computing," *Design Automation Conference, 2009. ASP-DAC 2009. Asia and South Pacific*. pp. 216–223, 2009.

[10] T. D. Han and T. S. Abdelrahman, "hiCUDA: High-Level GPGPU Programming," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1. pp. 78–90, 2011.

[11] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for GPGPU programs," *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–11, 2010.

[12] S. Chu and C. Hsiao, “OpenCL: Make Ubiquitous Supercomputing Possible,” *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pp. 556–561, Sep. 2010.

[13] A. Maghazeh, U. D. Bordoloi, P. Eles, and Z. Peng, “General purpose computing on low-power embedded GPUs: Has it come of age?,” *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 1–10, Jul. 2013.

[14] G. Calandrini, A. Gardel, P. Revenga, and J. L. L’zaro, “GPU Acceleration on Embedded Devices. A Power Consumption Approach,” *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pp. 1806–1812, Jun. 2012.

[15] S. Mu, C. Wang, M. Liu, D. Li, M. Zhu, X. Chen, X. Xie, and Y. Deng, “Evaluating the potential of graphics processors for high performance embedded computing,” *2011 Design, Automation & Test in Europe*, pp. 1–6, Mar. 2011.

[16] J. Leskela, J. Nikula, M. Salmela, R. Projects, and N. Corporation, “OpenCL embedded profile prototype in mobile device,” pp. 279–284, 2009.

[17] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb, “Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs,” in *Proceedings of the 2011 International Conference on Parallel Processing*, 2011, pp. 176–185.

[18] “Home | TOP500 Supercomputer Sites.” [Online]. Available: http://www.top500.org/. [Last Visited in January 2014].

[19] K. a. Seitz Jr. and M. C. Lewis, “Virtual Machine and Bytecode for Optimization on Heterogeneous Systems,” *2012 Ninth International Conference on Information Technology - New Generations*, pp. 528–533, Apr. 2012.

[20] J. Parri, J.-M. Desmarais, D. Shapiro, M. Bolic, and V. Groza, “Design of a custom vector operation API exploiting SIMD intrinsics within Java,” *Electrical and Computer Engineering (CCECE), 2010 23rd Canadian Conference on*. pp. 1–4, 2010.

[21] “Java Technology.” [Online]. Available: http://www.java.com/en/about/. [Last Visited in January 2014].

[22] J. Parri, D. Shapiro, M. Bolic, and V. Groza, “Returning Control to the Programmer: SIMD Intrinsics for Virtual Machines,” *Queue*, vol. 9, no. 2, pp. 30:30–30:37, Feb. 2011.

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

[24] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-GPU synchronization," *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 354–365, Feb. 2013.

[25] "The Java Language Environment: Contents." [Online]. Available: http://www.oracle.com/technetwork/java/index-136113.html. [Last Visited in January 2014].

[26] "Oracle JNI Specification." [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html. [Last Visited in January 2014].

[27] H. Kim, R. Vuduc, S. Baghsorkhi, J. Choi, and W. Hwu, "Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)," *Synthesis Lectures on Computer Architecture*, vol. 7, no. 2, pp. 1–96, Nov. 2012.

[28] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*. 2011.

[29] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*. 2011.

[30] M. Ujaldon, "High performance computing and simulations on the GPU using CUDA," *2012 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 1–7, Jul. 2012.

[31] L. Hu, X. Che, and Z. Xie, "GPGPU cloud: A paradigm for general purpose computing," *Tsinghua Science and Technology*, vol. 18, no. 1. pp. 22–23, 2013.

[32] F. Jacob, D. Whittaker, S. Thapaliya, P. Bangalore, M. Mernik, and J. Gray, "CUDACL: A tool for CUDA and OpenCL programmers," *2010 International Conference on High Performance Computing*, pp. 1–11, Dec. 2010.

[33] G. L. Taboada, S. Ramos, R. R. Expósito, J. Touriño, and R. Doallo, "Java in the High Performance Computing Arena: Research, Practice and Experience," *Sci. Comput. Program.*, vol. 78, no. 5, pp. 425–444, May 2013.

[34] "Java Bindings for the OpenCL API." [Online]. Available: http://jogamp.org/jocl/www/. [Last Visited in January 2014].

[35] "OpenJDK: Project Sumatra." [Online]. Available: http://openjdk.java.net/projects/sumatra/. [Last Visited in January 2014].

[36] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java Bytecode Optimization Framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, 1999, p. 13–.

[37] "Vivante OpenCL Guide," no. August, pp. 1–28, 2012.

[38] D. Ashlock, "Evolutionary Exploration of the Mandelbrot Set," *2006 IEEE International Conference on Evolutionary Computation*, pp. 2079–2086, 2006.

[39] A. Leung, O. Lhoták, and G. Lashari, "Automatic Parallelization for Graphics Processing Units," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, 2009, pp. 91–100.

[40] J. Burnim and K. Sen, "Asserting and checking determinism for multithreaded programs," *Communications of the ACM*, vol. 53, no. 6, p. 97, Jun. 2010.

[41] J. R. Vaughan and G. R. Brookes, "The Mandelbrot Set As a Parallel Processing Benchmark," *Univ. Comput.*, vol. 11, no. 4, pp. 193–197, Dec. 1989.

[42] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete Cosine Transform," *Computers, IEEE Transactions on*, vol. C–23, no. 1. pp. 90–93, 1974.

[43] A. Obukhov and A. Kharlamov, "Discrete cosine transform for 8x8 blocks with CUDA," *NVIDIA white paper*, no. October, 2008.

[44] "The JPEG committee home page." [Online]. Available: http://www.jpeg.org/. [Last Visited in January 2014].

Appendix A: DCT Kernel

```
uint   getId(uint blockIdx, uint blockIdy, uint localIdx, uint localIdy, uint
blockWidth, uint globalWidth)
{
       uint globalIdx = blockIdx * blockWidth + localIdx;
       uint globalIdy = blockIdy * blockWidth + localIdy;
       return (globalIdy * globalWidth  + globalIdx);
}

__kernel void kernelDCT(__global float *dct8x8,
                        __global float *input,
                        __global float *output,
                        __local float *temp,
                        const uint width,
                        const uint blkWidth,
                        const uint inverse)
{
       //Get Global Ids of elements

       uint globalIdx = get_global_id(0);
       uint globalIdy = get_global_id(1);

       //Get the block Id
       uint groupIdx = get_group_id(0);
       uint groupIdy = get_group_id(1);


       //Get Indices relative to the block
       uint i = get_local_id(0);
       uint j = get_local_id(1);

       uint idx = globalIdy * width + globalIdx;

       //Initializing accumulator variable
       float tmp = 0.0f;

       //Multiplying AT and X
       for(uint k = 0; k < blkWidth; k++)
       {
              uint index1 = (inverse) ? i * blkWidth +k : k * blkWidth +i;
              uint index2 = getId(groupIdx, groupIdy, j, k, blkWidth, width);

              tmp += dct8x8[index1] * input[index2];
       }

       temp[j * blkWidth + i] = tmp;
       barrier(CLK_LOCAL_MEM_FENCE);

       tmp = 0.0f;

       //Multiplying (AT * X) with A
       for(uint k = 0; k < blkWidth; k++)
       {
              uint index1 = i * blkWidth + k;
              uint index2 = (inverse) ? j * blkWidth +k : k * blkWidth +j;

              tmp += temp[index1] * dct8x8[index2];
       }

       output[idx] = tmp;
}
```

Appendix B: JNI-OpenCL Host Code for DCT Calculation

```
/*****************************************************************************
This is the OpenCL Host code that will be called from the Java program. Here, the
programSource variable contains the OpenCL kernel shown in Appendix A stored as a
string.
*****************************************************************************/
#include <jni.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "DCT.h"

#include <CL/cl.h>

#define _USE_MATH_DEFINES

//Initializing the 8x8 Cosine Values

float dct[64];
void initialize()
{
	for(int j=0;j<8;j++)
		dct[j] = 1/sqrt(8.0);

	for(int i=1;i<8;i++)
		for(int j=0;j<8;j++)
		{
			double idob = (double) i;
			double jdob = (double) j;

			dct[i* 8 + j] = sqrt(2.0/8.0) * cos(((2.0 * jdob + 1) * idob *
M_PI) / (2.0 * 8.0));
		}
}

JNIEXPORT void JNICALL Java_DCT_GpuDCT(JNIEnv * env,
                                       jobject obj,
                                       jfloatArray jniinput,
                                       jfloatArray jnioutput,
                                       jint jwidth,
                                       jint jblkwidth,
                                       jint jinv)
{
	jfloat *jip, *jop;


	float *ip, *op;
	int width, blkwidth, inv;

	width = (int) jwidth;
	blkwidth = (int) jblkwidth;
	inv = (int) jinv;

	initialize();

	int n = env -> GetArrayLength(jniinput);

	int global_size_x = width;
	int global_size_y = width;

	int local_size_x = blkwidth;
```

```
int local_size_y = blkwidth;

size_t dataSize = sizeof(float) * n;
ip = (float*) malloc(dataSize);
op = (float*) malloc(dataSize);

jip = env -> GetFloatArrayElements(jniinput, NULL);
if(jip == NULL)
{
       printf("Error! Cannot read input matrix\n");
       return;
}

jop = env -> GetFloatArrayElements(jnioutput, NULL);
if(jop == NULL)
{
       printf("Error! Cannot read output matrix\n");
       return;
}

ip = (float*) jip;
op = (float*) jop;

cl_int status;

//---------------------------------------------------
// STEP 1: Discover and initialize the platforms
//---------------------------------------------------

cl_uint numPlatforms = 0;
cl_platform_id *platforms = NULL;

// Use clGetPlatformIDs() to retrieve the number of platforms
status = clGetPlatformIDs(0, NULL, &numPlatforms);

// Allocate enough space for each platform
platforms =(cl_platform_id*)malloc(numPlatforms*sizeof(cl_platform_id));

// Fill in platforms with clGetPlatformIDs()
status = clGetPlatformIDs(numPlatforms, platforms, NULL);

printf("There are %d platforms on this machine\n", numPlatforms);

//---------------------------------------------------
// STEP 2: Discover and initialize the devices
//---------------------------------------------------

cl_uint numDevices = 0;
cl_device_id *devices = NULL;

// Use clGetDeviceIDs() to retrieve the number of devices present

status = clGetDeviceIDs(
                    platforms[0],
                    CL_DEVICE_TYPE_ALL,
                    0,
                    NULL,
                    &numDevices);

// Allocate enough space for each device
devices = (cl_device_id*)malloc(numDevices*sizeof(cl_device_id));

// Fill in devices with clGetDeviceIDs()
```

```
status = clGetDeviceIDs(
                    platforms[0],
                    CL_DEVICE_TYPE_ALL,
                    numDevices,
                    devices,
                    NULL);

//---------------------------------------------------
// STEP 3: Create a context
//---------------------------------------------------

cl_context context = NULL;

// Create a context using clCreateContext() and associate it with devices
context = clCreateContext(
                    NULL,
                    numDevices,
                    devices,
                    NULL,
                    NULL,
                    &status);

//---------------------------------------------------
// STEP 4: Create a command queue
//---------------------------------------------------

cl_command_queue cmdQueue;

// Create a command queue using clCreateCommandQueue(),
// and associate it with the device you want to execute on

cmdQueue = clCreateCommandQueue(
                    context,
                    devices[0],
                    CL_QUEUE_PROFILING_ENABLE,
                    &status);

//---------------------------------------------------
// STEP 5: Create device buffers
//---------------------------------------------------

cl_mem inputBuffer, outputBuffer, dctBuffer;

inputBuffer = clCreateBuffer(
                    context,
                    CL_MEM_READ_ONLY,
                    dataSize,
                    NULL,
                    &status);

outputBuffer = clCreateBuffer(
                    context,
                    CL_MEM_WRITE_ONLY,
                    dataSize,
                    NULL,
                    &status);

dctBuffer = clCreateBuffer(
                    context,
                    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                    dataSize,
                    &dct,
                    &status);
```

```
//---------------------------------------------------
// STEP 6: Write host data to device buffers
//---------------------------------------------------

cl_event writeEvent;

status = clEnqueueWriteBuffer(
                    cmdQueue,
                    inputBuffer,
                    CL_FALSE,
                    0,
                    dataSize,
                    ip,
                    0,
                    NULL,
                    &writeEvent);

//---------------------------------------------------
// STEP 7: Create and compile the program
//---------------------------------------------------

cl_program program = clCreateProgramWithSource(
                    context,
                    1,
                    (const char**)&programSource,
                    NULL,
                    &status);

status = clBuildProgram(
                    program,
                    numDevices,
                    devices,
                    NULL,
                    NULL,
                    NULL);

//---------------------------------------------------
// STEP 8: Create the kernel
//---------------------------------------------------

cl_kernel kernel = clCreateKernel(
                    program,
                    "kernelDCT",
                    &status);

//---------------------------------------------------
// STEP 9: Set the kernel arguments
//---------------------------------------------------

status = clSetKernelArg(
                    kernel,
                    0,
                    sizeof(cl_mem),
                    &dctBuffer);

status |= clSetKernelArg(
                    kernel,
                    1,
                    sizeof(cl_mem),
                    &inputBuffer);

status |= clSetKernelArg(
```

```
                    kernel,
                    2,
                    sizeof(cl_mem),
                    &outputBuffer);

status |= clSetKernelArg(
                    kernel,
                    3,
                    blkwidth * blkwidth * sizeof(cl_float),
                    NULL);

status |= clSetKernelArg(
                    kernel,
                    4,
                    sizeof(cl_uint),
                    &width);

status |= clSetKernelArg(
                    kernel,
                    5,
                    sizeof(cl_uint),
                    &blkwidth);

status |= clSetKernelArg(
                    kernel,
                    6,
                    sizeof(cl_uint),
                    &inv);

//-----------------------------------------------------
// STEP 10: Configure the work-item structure
//-----------------------------------------------------

size_t globalWorkSize[2];
size_t localWorkSize[2];

globalWorkSize[0] = global_size_x;
globalWorkSize[1] = global_size_y;
localWorkSize[0] = local_size_x;
localWorkSize[1] = local_size_y;

//-----------------------------------------------------
// STEP 11: Enqueue the kernel for execution
//-----------------------------------------------------

cl_event kernelEvent;
status = clEnqueueNDRangeKernel(
                    cmdQueue,
                    kernel,
                    2,
                    NULL,
                    globalWorkSize,
                    localWorkSize,
                    0,
                    NULL,
                    &kernelEvent);

printf("Waiting on Kernel event...\n");
status = clWaitForEvents(
                    1,
                    &kernelEvent);
```

```
//---------------------------------------------------
// STEP 12: Read the output buffer back to the host
//---------------------------------------------------

cl_event readEvent;
status = clEnqueueReadBuffer(
                    cmdQueue,
                    outputBuffer,
                    CL_TRUE,
                    0,
                    dataSize,
                    op,
                    0,
                    NULL,
                    &readEvent);

//---------------------------------------------------
// STEP 13: Profile the copying and execution time
//---------------------------------------------------

cl_ulong startTime, endTime;
float elapsed;

clGetEventProfilingInfo(
                    writeEvent,
                    CL_PROFILING_COMMAND_QUEUED,
                    sizeof(cl_ulong),
                    &startTime,
                    NULL);

clGetEventProfilingInfo(
                    writeEvent,
                    CL_PROFILING_COMMAND_END,
                    sizeof(cl_ulong),
                    &endTime,
                    NULL);

elapsed = (float) (endTime - startTime)/1e6;
printf("\nTime taken to copy from CPU to GPU is %8.3f ms", elapsed);

clGetEventProfilingInfo(
                    kernelEvent,
                    CL_PROFILING_COMMAND_QUEUED,
                    sizeof(cl_ulong),
                    &startTime,
                    NULL);

clGetEventProfilingInfo(
                    kernelEvent,
                    CL_PROFILING_COMMAND_END,
                    sizeof(cl_ulong),
                    &endTime,
                    NULL);

elapsed = (float) (endTime - startTime)/1e6;
printf("\nTime taken for kernel execution is %8.3f ms", elapsed);

clGetEventProfilingInfo(
                    readEvent,
                    CL_PROFILING_COMMAND_QUEUED,
                    sizeof(cl_ulong),
                    &startTime,
                    NULL);
```

```
        clGetEventProfilingInfo(
                            readEvent,
                            CL_PROFILING_COMMAND_END,
                            sizeof(cl_ulong),
                            &endTime,
                            NULL);

        elapsed = (float) (endTime - startTime)/1e6;
        printf("\nTime taken to copy form GPU to CPU is %8.3f ms\n", elapsed);

        //---------------------------------------------------
        // STEP 14: Release OpenCL resources
        //---------------------------------------------------

        clReleaseKernel(kernel);
        clReleaseProgram(program);
        clReleaseCommandQueue(cmdQueue);
        clReleaseMemObject(inputBuffer);
        clReleaseMemObject(outputBuffer);
        clReleaseMemObject(dctBuffer);
        clReleaseContext(context);

        env -> ReleaseFloatArrayElements(jniinput, jip,0);
        env -> ReleaseFloatArrayElements(jnioutput, jop, 0);
        free(platforms);
        free(devices);
}
```

Appendix C: JOCL Host Code for DCT Computation

```
/**************************************************************************
This is the OpenCL Host code and the Java program written using JOCL. Here, the
programSource variable contains the OpenCL kernel shown in Appendix A stored as a
string.
**************************************************************************/


import static org.jocl.CL.*;

import org.jocl.*;

import java.util.*;

public class JoclDCT{


	private static final int width = 2048;
	private static final int height = 2048;
	private static final int blkwidth = 8;

	public static int nelements = width * height;
	public float[] A = new float[nelements];


	public JoclDCT()
	{
		initialize();
	}

	void initialize()
	{
		for(int j=0;j<8;j++)
			A[j] = (float) 1/ (float) Math.sqrt(8.0);

		for(int i=1;i<8;i++)
			for(int j=0;j<8;j++)
			{
				double idob = (double) i;
				double jdob = (double) j;

				A[i* 8 + j] = (float) Math.sqrt(2.0/8.0) * (float)
Math.cos(((2.0 * jdob + 1) * idob * Math.PI) / (2.0 * 8.0));
			}
	}


	public void oclwork(float[] ip, float[] op, int inv){

		Pointer d_ip = Pointer.to(ip);
		Pointer d_A = Pointer.to(A);
		Pointer d_op = Pointer.to(op);

		int[] widPar = new int[1];
		int[] blkwidPar = new int[1];
		int[] invPar = new int[1];

		widPar[0] = width;
		blkwidPar[0] = blkwidth;
		invPar[0] = inv;
```

```
            //Declare the Platform, Device and Device Index

            final int platformIndex = 0;
            final long deviceType = CL_DEVICE_TYPE_DEFAULT;
            final int deviceIndex = 0;

            CL.setExceptionsEnabled(true);

            //Query the number of platforms

            int numPlatformsArray[] = new int [1];
            clGetPlatformIDs(0, null, numPlatformsArray);
            int numPlatforms = numPlatformsArray[0];

            //Query PlatformID

            cl_platform_id platforms[] = new cl_platform_id[numPlatforms];
            clGetPlatformIDs(platforms.length, platforms, null);
            cl_platform_id platform = platforms[platformIndex];

            //Initialize the context properties

            cl_context_properties contextProperties = new
cl_context_properties();
            contextProperties.addProperty(CL_CONTEXT_PLATFORM, platform);

            //Obtain the number of devices for the platform

            int numDevicesArray[] = new int [1];
            clGetDeviceIDs(platform,
                                deviceType,
                                0,
                                null,
                                numDevicesArray);
            int numDevices = numDevicesArray[0];


            //Obtain a Device ID

            cl_device_id devices[] = new cl_device_id[numDevices];
            clGetDeviceIDs(platform,
                                deviceType,
                                numDevices,
                                devices,
                                null);
            cl_device_id device = devices[deviceIndex];

            //Create a context for the selected device

            cl_context context =
                  clCreateContext(contextProperties,
                                1,
                                new cl_device_id[]{device},
                                null,
                                null,
                                null);

            //Create a command-queue for the selected device

            cl_command_queue commandQueue =
                  clCreateCommandQueue(context,
                                device,
                                CL_QUEUE_PROFILING_ENABLE,
```

```
                    null);

//Allocate the memory objects for the input and output data

cl_mem inputBuffer, outputBuffer, dctBuffer;

inputBuffer = clCreateBuffer(context,
                    CL_MEM_READ_ONLY,
                    Sizeof.cl_float * nelements,
                    null,
                    null);

outputBuffer = clCreateBuffer(context,
                    CL_MEM_WRITE_ONLY,
                    Sizeof.cl_float * nelements,
                    null,
                    null);

dctBuffer = clCreateBuffer(context,
                    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                    Sizeof.cl_float * blkwidth * blkwidth,
                    Pointer.to(A),
                    null);

cl_event cp1_tim_event = new cl_event();

clEnqueueWriteBuffer(commandQueue,
                    inputBuffer,
                    CL_FALSE,
                    0,
                    Sizeof.cl_float * nelements,
                    d_ip,
                    0,
                    null,
                    cp1_tim_event);

long startOcl = System.nanoTime();

//Create the program from Source

cl_program program =
       clCreateProgramWithSource(context,
                    1,
                    new String[]{ programSource },
                    null,
                    null);

//Build the program

clBuildProgram(program, 0, null, null, null, null);

//Create the kernel

cl_kernel kernel = clCreateKernel(program,
                    "kernelDCT",
                    null);

//Set arguments for the kernel

clSetKernelArg(kernel,
                    0,
                    Sizeof.cl_mem,
                    Pointer.to(dctBuffer));
```

```
        clSetKernelArg(kernel,
                       1,
                       Sizeof.cl_mem,
                       Pointer.to(inputBuffer));

        clSetKernelArg(kernel,
                       2,
                       Sizeof.cl_mem,
                       Pointer.to(outputBuffer));

        clSetKernelArg(kernel,
                       3,
                       blkwidth * blkwidth * Sizeof.cl_float,
                       null);

        clSetKernelArg(kernel,
                       4,
                       Sizeof.cl_uint,
                       Pointer.to(widPar));

        clSetKernelArg(kernel,
                       5,
                       Sizeof.cl_uint,
                       Pointer.to(blkwidPar));

        clSetKernelArg(kernel,
                       6,
                       Sizeof.cl_uint,
                       Pointer.to(invPar));

        //Set the work-item dimensions

        long global_work_size[] = new long[] {width, height};
        long local_work_size[] = new long[] {blkwidth, blkwidth};

        //Execute the kernel

        cl_event kernelEvent = new cl_event();

        clEnqueueNDRangeKernel(commandQueue,
                       kernel,
                       2,
                       null,
                       global_work_size,
                       local_work_size,
                       0,
                       null,
                       kernelEvent);

        //Wait for the events, i.e. until the kernels have completed
execution

        System.out.println("Waiting on Kernel event...");
        clWaitForEvents(1, new cl_event[]{kernelEvent});

        long endOcl = System.nanoTime();

        double kerTime = (endOcl - startOcl)/1000000d;

        System.out.println("Time taken for creating, building and executing
kernel is "+kerTime+" ms");
```

```
            //Read the DCT output

            cl_event readEvent = new cl_event();

            clEnqueueReadBuffer(commandQueue,
                                outputBuffer,
                                CL_TRUE,
                                0,
                                nelements * Sizeof.cl_float,
                                d_op,
                                0,
                                null,
                                readEvent);

            long startTime[] = new long[1];
            long endTime[] = new long[1];
            long tx1_duration, tx2_duration, kernel_exe_time;

            clGetEventProfilingInfo(cp1_tim_event,
                                CL_PROFILING_COMMAND_QUEUED,
                                Sizeof.cl_ulong,
                                Pointer.to(startTime),
                                null);

            clGetEventProfilingInfo(cp1_tim_event,
                                CL_PROFILING_COMMAND_END,
                                Sizeof.cl_ulong,
                                Pointer.to(endTime),
                                null);

            tx1_duration = endTime[0] - startTime[0];

            System.out.println("Time taken to copy from CPU to GPU is
"+String.format("%8.3f", tx1_duration/1e6)+" ms");

            clGetEventProfilingInfo(kernelEvent,
                                CL_PROFILING_COMMAND_QUEUED,
                                Sizeof.cl_ulong,
                                Pointer.to(startTime),
                                null);

            clGetEventProfilingInfo(kernelEvent,
                                CL_PROFILING_COMMAND_END,
                                Sizeof.cl_ulong,
                                Pointer.to(endTime),
                                null);

            kernel_exe_time = endTime[0] - startTime[0];

            System.out.println("Time taken to execute kernel is
"+String.format("%8.3f", kernel_exe_time/1e6)+" ms");

            clGetEventProfilingInfo(readEvent,
                                CL_PROFILING_COMMAND_QUEUED,
                                Sizeof.cl_ulong,
                                Pointer.to(startTime),
                                null);

            clGetEventProfilingInfo(readEvent,
                                CL_PROFILING_COMMAND_END,
                                Sizeof.cl_ulong,
                                Pointer.to(endTime),
                                null);
```

```
            tx2_duration = endTime[0] - startTime[0];

            System.out.println("Time taken to copy form GPU ot CPU is
"+String.format("%8.3f", tx2_duration/1e6)+" ms");

            //Release kernel, program and memory objects

            clReleaseMemObject(inputBuffer);
            clReleaseMemObject(dctBuffer);
            clReleaseMemObject(outputBuffer);
            clReleaseKernel(kernel);
            clReleaseProgram(program);
            clReleaseCommandQueue(commandQueue);
            clReleaseContext(context);

      }

      public static void main(String[] args){

            JoclDCT objDCT = new JoclDCT();

            System.out.println("\nThe input size is "+width+"x"+height);

            float[] input = new float[nelements];
            float[] output = new float[nelements];
            float[] invop = new float[nelements];
            int inverse = 0;

            for(int i=0;i<width;i++)
                  for(int j=0;j<height;j++)
                        input[i * width +j] = (float) ((float)i*8.0+j);

            long sfulltime = System.nanoTime();
            objDCT.oclwork(input, output, inverse);
            long efulltime = System.nanoTime();
            System.out.println("\nThe Complete OpenCL setup and execution time
for DCT is: "+(efulltime-sfulltime)/1e6+" ms\n");

            inverse = 1;

            sfulltime = System.nanoTime();
            objDCT.oclwork(output, invop, inverse);
            efulltime = System.nanoTime();
            System.out.println("\nThe Complete OpenCL setup and execution time
for IDCT is: "+(efulltime-sfulltime)/1e6+" ms\n");

      }
}
```