Universidade do Minho
Escola de Engenharia
Departamento de Informática

Gonçalo Medeiros São Pedro Raposo

Efficient execution of Java programs on GPU

November 2021

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Gonçalo Medeiros São Pedro Raposo

Efficient execution of Java programs on GPU

Master dissertation
Master Degree in Informatics Engineering

Dissertation supervised by
João Sobral

November 2021

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Gonçalo Raposo

ACKNOWLEDGEMENTS

The development of this master dissertation was a long process where without some people it could have been a project impossible to complete.

First of all I need to acknowledge the constant dedication and faith that my parents placed in me, I could not even start this adventure if not for their continuous support.

I would like to thank my grandparents for opening my horizons and motivation for always trying to reach further and further so I can have the future that I desire.

My friends were also indispensable for moral support in bad days as well as incredible when asked for a helping hand. Without you, I know that this project would never be finished.

Finally, I have to thank professor João Luis Sobral for his availability, patience and guidance throughout this master dissertation.

ABSTRACT

With the overwhelming increase of demand of computational power made by fields as Big Data, Deep Machine learning and Image processing the Graphics Processing Units (GPUs) has been seen as a valuable tool to compute the main workload involved. Nonetheless, these solutions have limited support for object-oriented languages that often require manual memory handling which is an obstacle to bringing together the large community of object-oriented programmers and the high-performance computing field.

In this master thesis, different memory optimizations and their impacts were studied in a GPU Java context using Aparapi. These include solutions for different identifiable bottlenecks of commonly used kernels exploiting its full capabilities by studying the GPU hardware and current techniques available. These results were set against common used C/OpenCL benchmarks and respective optimizations proving, that high-level languages can be a solution to high-performance software demand.

Keywords:GPU, GPGU, Java, High-Performance, OpenCL, Aparapi, SHOC, Parboil

RESUMO

Com o aumento de poder computacional requesitado por campos como Big Data, Deep Machine Learning e Processamento de Imagens, as unidades de processamento gráfico (GPUs) têm sido vistas como uma ferramenta valiosa para executar a principal carga de trabalho envolvida. No entanto, estas soluções têm suporte limitado para linguagens orientadas a objetos. Frequentemente estas requerem manipulação manual de memória, o que é um obstáculo para reunir a grande comunidade de programadores orientados a objetos e o campo da computação de alto desempenho.

Nesta dissertação de mestrado, diferentes otimizações de memória e os seus impactos foram estudados utilizando Aparapi. As otimizações estudadas pretendem solucionar bottle-necks identificáveis em kernels frequentemente utilizados.

Os resultados obtidos foram comparados com benchmarks C / OpenCL populares e as suas respectivas otimizações, provando que as linguagens de alto nível podem ser uma solução para programas que requerem computação de alto desempenho.

Palavras Chave:GPU, GPGU, Java, High-Performance, OpenCL, Aparapi, SHOC, Parboil

CONTENTS

ACRONYMS

A

AOP Array of Pointers.

AOS Array of Structures.

API Application programming interface.

C

CPU Central processing unit.

CSR Compressed sparse row.

CUDA Compute Unified Device Architecture.

G

GPGU General graphical processing unit.

GPU Graphical processing unit.

GPUS Graphical processing units.

J

JABEE Java Bytecode Execution Environment.

JAR Java Archive.

JIT Just in time.

JVM Java virtual machine.

L

L-J Lennard-Jones.

LRU Least Recently Used.

M

MD	Molecular dynamics.
O	
OPENCL	Open Computing Language.
OS	Operating system.
R	
RAM	Random access memory.
S	
SHOC	The Scalable HeterOgeneous Computing.
SIMD	Single instruction multiple data.
SOA	Structure of Arrays.
W	
WORA	Write once read anywhere.

1

INTRODUCTION

1.1 CONTEXT

The current human society is dependent on computer systems and how well they perform. With technologies predicted to solidify their impact in our lives, from entertainment to science, health and education and the constant demand of the human being for an overall better future forces the amount of work required from these computer systems to increase at a blistering rate as recorded by (Fuller and Millett, 2011). To further increase this problem, researchers and chip manufacturers have increasingly warned about the physical limit of Moore's law. The transistors manufactured are approaching the atom size which escalates the difficulty of suppressing quantum phenomena (Peper, 2017).

Graphics Processing Units (GPUs) may offer a significant performance improvement for many of these challenges. This type of high performance have been demanding more programmers skills since the level of detail, accuracy and the general amount of data used in scientific and industrial areas requires it. This is very clear to observe when Deep Learning algorithms demand millions of data samples to produce an accurate prediction or a fluid simulation that needs to do intensive calculations over a detailed environment. However, to take advantage of GPU capabilities, generally, these programms are programmed in specific and low-level languages. Because of this, a big proportion of the programming community faces a learning barrier while developing GPU software. To tackle this issue, high-level languages are adapting to current demands by constructing APIs and similar tools to enable GPU execution.

Several approaches already exist that enable programmes execution in GPU from high-level programming languages like Java. One that has been gaining popularity in the community is Aparapi. It implements an approach to GPU programming similar to normal Java threads being intuitive and enabling a fast transition from a normal software that does not support GPU execution.

While computational power is a key to solve the demands of science and industry, one can not forget about the algorithms that enable the full capabilities of the hardware. Some optimizations to the original algorithm can have an enormous impact on the overall behaviour of the program. One essential optimization for GPU execution is to ensure good data locality and minimal data transfers between CPU and GPU. In the case of Java, which was studied by Faria et al. (2013), the objects are stored through the heap memory region as Arrays of Pointers (AoP) and the author demonstrated why this approach restricts the program's scalability by making unnecessary memory accesses and instructions while increasing the number of cache misses.

During this master dissertation, different algorithms and optimizations will be studied in Aparapi to understand how high-level languages can compete with low-level solutions by comparing the performance results of both while taking note of the current obstacles and downsides of the solutions.

1.2 MOTIVATION

To address the issues previously mentioned, a performance analysis of platforms that enable GPU programming is critical. Due to necessity, some platforms that enable native Java code execution in GPU are being developed and drawing more attention every day with the promise of fast implementation while taking advantage of the normal Java capabilities. The most prominent of these is Aparapi. In this master dissertation, a study of Aparapi inner works and performance results will be analysed to understand the weak and strong points of this tool and how can it be compared to other GPU programming languages.

1.3 OBJETIVES

This master thesis target is to study Aparapi performance through different GPU benchmarks while applying various kinds of optimizations. The following techniques were studied:

1. Performance impact of run-time OpenCL generation;
2. Analysis of the execution of operations over Java objects and other data layouts in GPU by Aparapi and its performance;
3. Review of the data communication costs between the CPU and GPU;
4. Coarsening performance impact in different benchmark kernels;
5. Usage of the overlapping tiling technique;
6. Privatization of data to tackle bandwidth issues and avoid data races;
7. Memory coalescing ensuring data locality;
8. Padding of data for kernel execution;

Aparapi can execute Java code in GPU by generating OpenCL code in run-time. It makes it by reading the bytecode of the executing program, identifying the kernel and required data to execute it in GPU. By doing this, some execution time is consumed. In point 1, the impact of the run-time OpenCL generation will be studied.

Even tho, Aparapi can execute in some instances objects in the GPU, the data locality diminishes, making it an interesting performance study case. The consequence of object usage in the GPU will be studied with point 2.

The cost of data communication between CPU and GPU is presented in point 3. Here an analysis of the performance degradation and how Aparapi is constructed to help improve this issue are exposed.

Points 4 through 8 present different optimizations. Different algorithms will be used to study how each of these optimizations can be implemented in Aparapi and their performance impact.

Finnaly, a performance comparison between Aparapi and C++ with OpenCL is explored. To do so, some algorithms of the SHOC and Parboil benchmark suites have been rewritten in Java/Aparapi. Each of the selected algorithms has typical bottlenecks that can be overcome with optimizations studied in this master dissertation.

A final performance conclusion can be reached by comparing the execution time of similar kernels in both tools. The data gathered is then used to support the decision of choosing Aparapi as an efficient tool to solve large parallel workloads.

1.4 RESEARCH HYPOTHESIS

Through the comparison of data-transfer times as well as execution times of various standard benchmark algorithms, this project aims to further understand the performance capabilities of high-level languages in GPUs. This study includes a step-by-step approach to the optimizations implemented so accurate conclusions can be drawn. Finally, a direct comparison between Aparapi and C++/OpenCL will be presented so that conclusions can be drawn.

1.5 DOCUMENT STRUCTURE

Following the main objectives and motivation of this master dissertation, a breve exposure of the state of the art is present in the second chapter. In this topic, the main memory system features of CPU and GPU will be presented as well as a brief discussion of both architectures, their differences and objectives. The Java programming language, its qualities and challenges are also exposed following the OpenCL GPU programming language. After this, the different optimizations used throughout this master dissertation are listed and explained. The state of the art chapter ends with an introduction to JaBEE and Aparapi framework.

In the third one, five different algorithms that require different types of optimizations are explained. For every algorithm, the majors bottlenecks are identified and their impact accounted for. For all the bottlenecks an optimization or set of optimizations that can be developed using Aparapi are then proposed.

On the forth chapter, the results of the different optimizations experiments are compiled, presented and compared between different optimizations as well as with the SHOC benchmark algorithms.

The last chapter makes regards to final comparisons between Aparapi and C++/OpenCL by drawing conclusions about both tools.

2 STATE OF THE ART

A wide range of methods have been developed and studied to solve performance-related issues throughout the past. In this section, some of the existing solutions are presented. These range from the memory system to data locality and hardware characteristics. Finally, a platform for executing native Java code on the GPU is contemplated to understand the current challenges that high-level programming languages face while developing code for GPU execution.

2.1 COMPUTATION SYSTEMS

As mentioned by Bryant and O'Hallaron (2002), "In practice, a memory system is a hierarchy of storage devices with different capacities, costs, and access times."

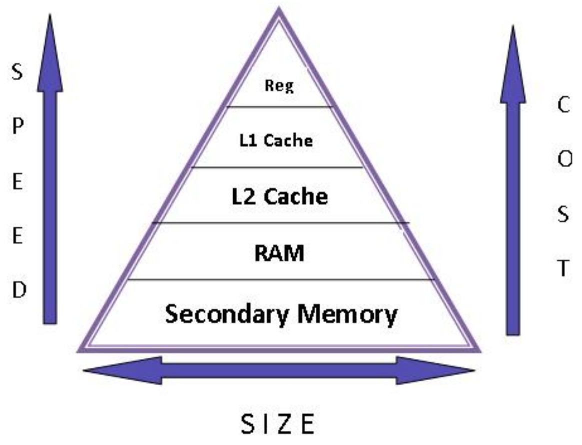

Figure 1: Memory system normally implemented in modern computer systems.World

CPU registers hold the most frequently used data. As can be seen in figure 1 the cache memories, which are fast but small, memory storages act as staging areas for data and instructions stored in the relatively slow main memory. This last one is itself another stage for the large but slow, disks or to other machines connected by a network.

The current memory hierarchy system as proven itself for years since well-written programs tend to access the storage at a particular level more frequently than they access the storage at a lower level. This has great impact in the performance of a program since the time needed to access data at register level is zero cycles during the execution of the instruction. If stored in cache, however, the time to fetch data can be of one to thirty cycles. Access to data stored in memory can have a duration of fifty to two hundred cycles and one access to the last memory layer can be of tens of millions of cycles.

2.1.1 *Locality*

Software ability to have access to recent data or data near recently referenced data objects is known as locality. Locality is an enduring principle which has a tremendous impact on software and hardware systems design and overall performance. Usually, this definition is described as having two forms. Temporal and spatial locality. Farber (2011) ”Temporal locality: assumes that a recently accessed data item is likely to be used again in the near future. Many computational applications demonstrate this LRU (Least Recently Used) behaviour. Spatial locality: neighboring data is stored in cache with the expectation that spatially adjacent memory locations will be used in the near future.” Modern computer systems are planned for locality exploitation. As seen in figure 1, the memory hierarchy is designed to exploit temporal position directly, since the most commonly used data is stored very close to the processing unit. In addition, the memory hierarchy often exploits spatial locality, as each time a memory level is accessed, the data near the referenced point is also retrieved to a upper memory level. As an example, a total of 64 bytes (the size of a cache line) are transferred to the cache memory when the main memory is accessed. The idea of locality is also widely implemented by the operating systems as they use the main memory as a buffer of the virtual address space’s most recent referenced data chunks.

Proving locality

One way to assess a program’s quality in terms of locality is by evaluating a program’s amount of cache hits and cache misses. When software requires data from memory, it first searches for that specific data in all upper levels of the hierarchy. If the program finds the piece of data that it was looking for at a higher level than the one it was supposed to seek, it is called a hit. On the other hand, if the requested data is not found at the cache level, it is considered a miss and the program needs to collect the data one level lower than the one it was searching for. This case drastically increases the time taken and potentially overwrites data already stored in the cache. Furthermore as Patterson and Hennessy (2014) explains, ”For a cache miss, we can stall the entire processor, essentially freezing the contents of the temporary and programmer-visible registers, while we wait for memory”. The overwritten portion of cache memory that happens when the cache is full, is commonly known as a victim block. The victim block can be adjusted as per enforced cache policy. For example, a cache with a replacement policy for LRU (Least-recently used) will select as the victim block the one who was last accessed the furthest in the past, while a random replacement policy will select a random block. Thus, a system with strong locality presents mostly cache hits and few cache misses.

2.1.2 *Data-level Parallelism*

McCool et al. (2012) :"Data parallelism is the key to achieving scalability. Merely dividing up the source code into tasks using functional decomposition will not give more than a constant factor speedup." Data-level parallelism is a general term that applies to any form of parallelism in which the amount of work increases with the problem's size., or as Patterson and Hennessy (2014) describes, "Data-level Parallelism is achieved by performing the same operation on independent data."

SIMD

Another way of approaching the performance issue is the use of Single Instruction Multiple Data (SIMD) or Vector instructions. Vector instructions are a great match to problems with a great amount of data-level parallelism since a vector instruction is fundamentally one instruction that will be repeated over an amount of similar data. With this in mind, a loop can be reduced to a single instruction reducing the fetch and bandwidth required. Also as pointed by Patterson and Hennessy (2014), "By using a vector instruction, the compiler or programmer indicates that the computation of each result in the vector is independent of the computation of other results in the same vector", this way hardware does not have to check for any kind of data hazard during one vector instruction. It is very important to be able to combine multiple loads or storage accesses in a single SIMD instruction to take most advantage of these. This can be done by organising the data to provide spatial locality and aligning array elements in memory.

The initial reason for introducing SIMD instructions to existing architectures was that many microprocessors being connected to graphic displays in PCs and workstations used a lot of processing power for graphic purposes. Computer game industry was a major driving force for developing graphics processing, both on PCs and in dedicated game consoles like the PlayStation. The rapidly growing demand for games prompted many companies to make increasing investments in the production of faster graphics computing hardware which had a positive feedback loop developing this form of hardware faster than traditional microprocessors. Given the differences in goals with the microprocessor development community and the graphics and the game community, this hardware developed its own processing style and terminology. As the graphics processors increased in power, to distinguish themselves from CPUs, they earned the name Graphics Processing Units or GPUs.

2.2 JAVA

Java is a programming language and computing platform that is class-based and object-oriented and was first launched in 1995 by Sun Microsystems. It is intended to let application developers write once, run anywhere (WORA), which means that compiled Java code will run on all Java-supporting platforms without recompilation. Java programs are usually compiled to bytecode that can run on any Java virtual machine (JVM), independent of the computer architecture underlying. Java's syntax is identical to C and C++, but it does have less low-level facilities than either. By 2019, with an estimated 9 million developers, Java was one of the most common programming languages in use according to GitHub, particularly for client-server web applications.

2.2.1 *Java objects*

Java objects are one of the most fundamental programming abstractions of java. When a new object is created, a set of memory is automatically allocated. The amount of memory used includes the overhead of metadata as illustrated in Figure 2.

- Class: A pointer to the class information, which describes the object type;
- Flags: A collection of flags that describe the state of the object, including the hash code for the object if it has one, and the shape of the object (that is, whether or not the object is an array);
- Lock: The synchronization information for the object — that is, whether the object is currently synchronized.

The object data, which consists of the fields stored in the object instance, is then placed after the object metadata.

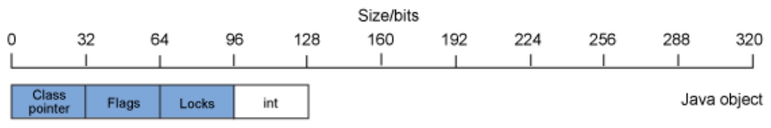

Figure 2: 128 bits of data are used to store the 32 bits of data in the int value, because the object metadata uses the rest of those 128 bits. Bailey

An array object, such as an array of int values, has a similar shape and structure to a standard Java object. The main distinction is that the array object has an extra piece of metadata that indicates the array's size like in figure 3.

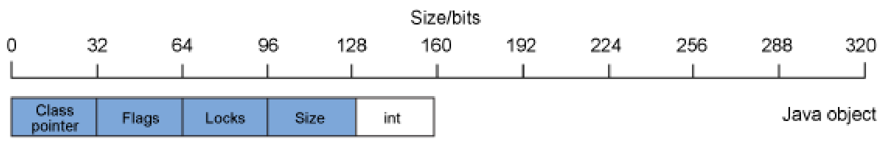

Figure 3: In figure 3, 160 bits of data store the 32 bits of data in the int value, because the array metadata uses the rest of those 160 bits. For primitives such as byte, int, and long, a single-entry array is more expensive in terms of memory than the corresponding wrapper object (Byte, Integer, or Long) for the single field. Bailey

2.2.2 *Java programs performance*

The programming language Java was typically considered slower (execution time) in software development than the fastest languages typed by the 3rd generation, such as C and C++ Lewis and Neumann (2003). The main explanation for this is that a Java program unlike the others above it needs to run in a Java virtual machine rather than directly on the computer's processor as do C. In order to tackle this performance issue, since the late 1990s, Java programs execution speed has greatly increased with the implementation of just-in-time compilation (JIT) (in 1997 for Java 1.1), the development of language features to enable efficient application interpretation and JVM enhancements.

The efficiency of a Java bytecode compiled Java program depends on how well the host Java virtual machine (JVM) handles its assigned tasks optimally, and how well the JVM uses the computer hardware and operating system (OS) features in doing so. So any Java performance check or comparison must always disclose the used JVM's version, vendor, Kernel, and hardware architecture. Similarly, the performance of the comparable native compiled program may depend on the consistency of its created machine code, so the test or comparison must also mention the name, version and vendor of the compiler used, as well as the compiler's disabled optimisation directives.

Performance issues

Due to the traditional object-oriented programming techniques, relatively small methods contribute to more regular method invocations than in other languages T. Suganuma and Nakatani (2000). There may be a method that the only objective is to access a private field variable, for example. Additionally, an object constructor function is immediately generated in Java even though it may not directly be written in the program, and therefore there are several methods for empty object construction. Invocations for the polymorphic method pose another efficiency issue due to the overhead of implicitly calling from a dynamic table lookup method. This problem is exacerbated by the re-usability of object-oriented programming, as programmers are expected to integrate general class libraries or existing frameworks built for general use into new programs to improve their productivity, and this trend has typically resulted in the development of several polymorphic method invocation sites. Even if the class is not explicitly overridden in a given program, the overhead invocation of the polymorphic form is always enforced. Both of these problems are implicit in object-oriented programming, which can be a significant factor in optimal performance.

Second, to ensure the integrity of the object and array accesses, the Java language specification includes run-time exception checks. If a reference or an operation is invalid, an exception must be thrown, and the environment, such as local variables, has to be preserved and made available to an exception handler that catches the exception. It involves a significant penalty in the execution of the program and prevents the use of traditional methods in loop optimisation. Another element in the run-time overhead of testing type conformance is the type inclusion test, which not only comes from the programmer's explicit request using an instance of operations but is implicitly provided by a statement assigning an object to another type.

Third, the clustered methods or clustered blocks used to maintain atomicity for a set of operations in a region introduce overhead run-time by locking a specified object for the execution duration. Once more, the re-usability of object-oriented programming makes this issue worse, since general-purpose class libraries are built often for use in multi-threaded systems, and synchronised methods or blocks are extensively used where thread safety is necessary.

Just-in-time compiling

In programming, compiling just-in-time (JIT) (also dynamic translation or run-time compilations) is a way to execute machine code that requires compiling a program during execution – at run-time – rather than before execution. This most frequently consists of source code or, more generally, bytecode translation into machine code, which is then immediately executed. A system implementing a JIT compiler typically analyses the code being executed on a continuous basis and identifies parts of the code where the speed gained from compiling or recompiling would outweigh the overhead of compiling that code.

JIT compilation is a hybrid of the two conventional computer code translation methods, AOT (ahead of time compilation) and interpretation, this way it can incorporate some of the benefits and disadvantages of both. JIT compilation blends the speed of compiled code with the versatility of interpretation, the overhead of an interpreter and the extra overhead (not only interpreting) of compiling.

Due to the time required to load and compile the bytecode, JIT induces a small but significant delay in initial program execution T. Suganuma and Nakatani (2000). This period is often called ”start time delay” or ”warm-up time” Generally speaking, the more JIT performs optimisation, the greater the code it can produce but the initial delay will also increase. Therefore, a JIT compiler must make a trade-off between the compilation time and the quality of the code which it hopes to produce.

Adaptive optimisation

Adaptive optimisation is a technique that performs a dynamic recompilation of parts of a program depending on the existing execution profile. An adaptive optimiser will easily trade-off between just-in-time compilation and instruction interpretation. Adaptive optimisation at another level can take advantage of local data conditions to optimise branches away and use inline expansion to decrease the cost of procedure calls.

Inline extension or inlining is an enhancement of a manual or compiler that substitutes a function call location with the body of the function called. Assuming that there is a software system that enrols students in a university, and tracks their academic progress throughout their academic life. Because of adaptive optimisation, in the first phase, the compiler will seek to optimise the enrolment of the thousands of students that are going to study at the university. In a later phase where there is few or none, new students, to enrol and the software uses most of its time updating and grading the students, the adaptive optimisation would recompile the assembly code to optimise for this new scenario.

2.2.3 *ByteCode*

Bytecode is a machine-level programming language that runs on the Java Virtual Machine. When a class is loaded, it receives a stream of bytecode for each of its methods. The bytecode for that method is invoked whenever that method is called during the execution of a program. This contributes to Java’s portability, which is lacking in languages like C and C++. Java’s portability ensures that it can be used on a wide range of platforms, including desktops, mobile devices, servers, and more. Following the bytecode interpretation (figure 4), Sun Microsystems dubbed JAVA ”write once, read anywhere” or ”WORA.”

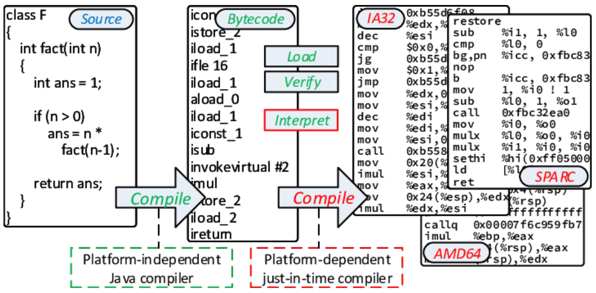
Source
Bytecode
Load
Verify
Interpret
Compile
Compile
IA32
SPARC
AMD64
Platform-independent Java compiler
Platform-dependent just-in-time compiler

Figure 4: ByteCode generation. Orlov (2017)

2.3 GRAPHICAL PROCESSING UNITS

Cedillo et al. "Recently, there has been an increasing interest in general purpose computation on graphics hardware. The ability to work independently alongside the CPU as a coprocessor is interesting but not motivating enough to learn how to apply problems to the graphics domain. However, with full programmability and the computational power of GPUs out pacing CPUs (in terms of a price/performance ratio), the GPU has moved to a place of being the primary processing unit for certain applications, with the CPU managing data and direction of the GPU."

GPUs are usually used as accelerators supplementing a CPU in its workload and they communicate through a Bus (figure 5), so they do not need to be able to perform all of a CPU's functions.

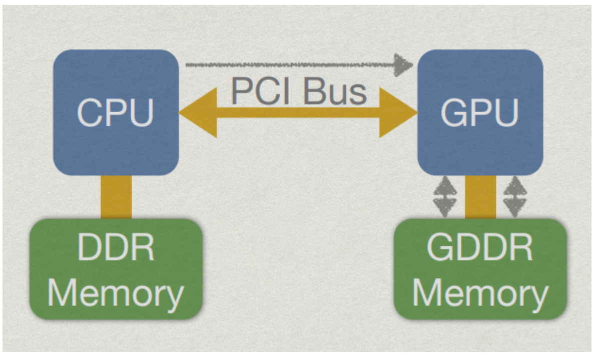

Figure 5: CPU and GPU communication. Pereira

Their function is dedicated exclusively most of the time to graphics. GPUs, unlike CPUs, don't rely heavily on multilevel caches.

To address the long latency of memory data collection they rely on multi threaded hardware. In other words, the GPU will run thousands of threads, between the time of a memory request and the time of data arrival. As a result, the GPU memory is dedicated to prioritising bandwidth over latency. Therefore GPUs are commonly used not only to process graphics but also rendering of video, cryptocurrency mining, statistic algorithms, audio processing, physical-based simulations, neural networks, cryptography and many more areas of study. However, as pointed out by Cedillo et al. ”Successful use of the GPU for general purpose computation requires taking into account the significant overhead incurred in executing and managing GPU kernels. This includes queuing overhead, scheduling overhead and the GPU progress check period.” This overhead can be a bottleneck in some applications caused by the time of transferring data from the CPU to GPU.

2.3.1 *Executing Java programs on GPU*

OpenCL

Apple worked with technical teams from AMD, IBM, Qualcomm, Intel, and NVIDIA to refine OpenCL into an initial proposal, which was then submitted to the Khronos Group. The host application programming interface (API) and the OpenCL C kernel language were defined in OpenCL 1.0 for executing data-parallel programs on heterogeneous devices. Shared virtual memory, nested parallelism, and generic address spaces were among the new features added to the OpenCL standard. These advanced features have the potential to make parallel application development easier and improve OpenCL application performance portability (Group, 2012). The Khronos Group has created an API that is general enough to run on a wide range of architectures while still being adaptable enough to achieve high performance on any hardware platform. Any program written for one vendor’s hardware can run on another vendor’s hardware if it is written in the core language and adheres to the specifications. Although the OpenCL API is written in C, third-party bindings exist for a variety of languages, including Java, C++, Python, and.NET.

Several well-known libraries in fields like linear algebra and computer vision have integrated OpenCL to take advantage of heterogeneous platforms and achieve significant performance gains.

Other languages like CUDA lack portability in comparison to OpenCL and each have it’s own nomenclature.

These differences exposed in figure 6 are important to note since exists different memory types with the same names between languages.

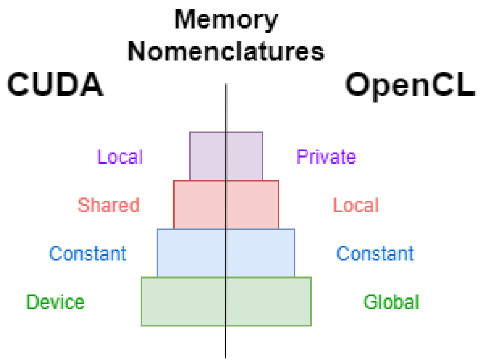

Figure 6: Differences in nomenclature of memory types.

Aparapi

Aparapi is a platform for executing native Java code on the GPU by dynamically translating Java byte code in runtime to an OpenCL kernel. Since Aparapi is backed up by OpenCL, it is compatible with all OpenCL compatible Graphics Cards. Aparapi was originally a project AMD Corporation conceived and developed. It was later abandoned by AMD, and for several years sat largely idle. It is maintained by the community nowadays. Currently, Aparapi supports Windows, Linux and Mac OSX. Aparapi can be easily implemented via a Maven dependency in existing software or by mannualy adding the available JAR. To execute a Java program in a GPU, it is necessary to create one Kernel object and a run method must be override. The run method is programmed in view of the multiple GPU threads running it in parallel having a similar approach as programming a CUDA kernel. All required memory allocation and management can either be done automatically or manually and Aparapi provides simple tools that profile the kernel. According to the study of Joshi (2011) , a sample Java based quantitative finance application JQuantlb, was able to achieve up to 20x improvement in performance after refactoring to use Aparapi kernels, proving the usefulness of this tool.

Aparapi works even in environments without a supported GPU having different algorithms prepared for automatic lookup for the best device to execute the kernel. If it is the first time that a Kernel will be executed, the framework checks if the platform supports OpenCL. If it does, then the Java byte code is converted and the execution is done on the GPU. If any problems arise that prevent the byte code to be generated than the Kernel is executed using the Java Thread Pool. If the Kernel has been used before, then Aparapi checks if it has stored the OpenCL correspondent code, preventing the need of generating it every time.

Aparapi however, does not have pointer support or dynamic dispatch. This means that Aparapi has a lot of limitations regarding object execution on a GPU and does not support common structures such as a matrix[l][c].

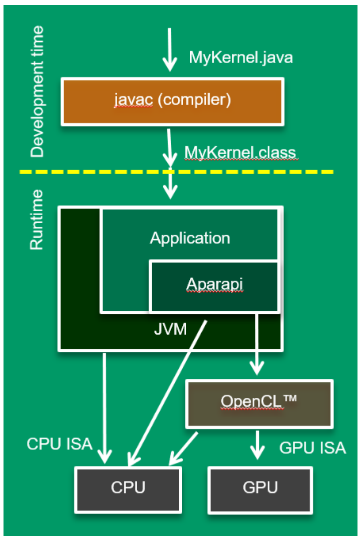

Figure 7: Aparapi System architecture. Frost

The generation of OpenCL supports some optimizations such as access to the shared memory, local and global barriers and precise translation from Java to OpenCL. This means that Aparapi is an optimal tool to reach the objectives of this master dissertation.

One of the most important characteristics that Aparapi offers is the capabilities of deploying the desirable GPU code using all of what OpenCL has to offer without the downside of its verbosity. This happens because most of the work of creating a new OpenCL kernel, initiating it and all the required operations of memory management are done in a couple of Aparapi code lines remaining only the kernel of the problem to program.

JaBEE

Contrary to Aparapi, the Java Bytecode Execution Environment supports object-oriented constructs like dynamic dispatch, encapsulation, and object creation on GPU. This environment makes the preparation, execution and transparent memory management of GPU code simpler.

The JaBEE architecture (figure 8) is built on a hybrid paradigm that enables selective bytecode execution on both the CPU and the GPU. The JaBEE system is made up of three major parts. GPUKernel is a base Java class that offers a Java interface for GPU code execution, as well as an online compiler that selectively converts Java byte-code to GPU code and a memory management system that moves data between the CPU and GPU.

The JaBEE does a two-stage on-line application compilation. In the first, the device converts bytecode to LLVM IR using VMKit's modified Java bytecode compiler. The LLVM IR is compiled into PTX in the second stage which is then ready to be executed on GPU. Like Aparapi, a Kernel class with a run method must be provided. The JaBEE passes the

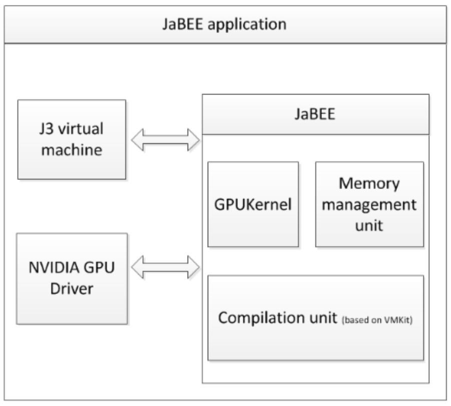

Figure 8: JaBEE System architecture.W. Zaremba and Grover (2012)

object that calls the run method to the GPU, objects passed to the kernel as parameters, all static fields reachable from the run method and recursively all objects pointed to by objects already copied.

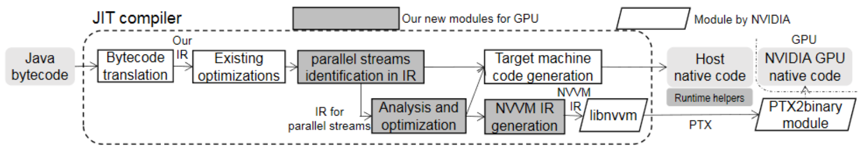

Figure 9: Jit compiler overview. Ishizaki et al.

A call from a Java code seeking GPU kernel execution starts a JaBEE execution. The control is initially handed to the GPUKernel object, which is a Java class instance that provides an interface for GPU code execution and kernel launch request validation. The native code is given more control via the GPUKernel. The system gathers information from native code by building a collection of dependant classes required for compilation, validating variable names, and creating pointer maps, all of which are required during subsequent stages of the execution process. The control is then given to the Compilation Unit, which compiles the bytecode into GPU assembly code (this step is optional because the code may already be built before it reaches this stage). The control then passes to the Memory Management Unit, which moves all required data to the GPU and changes pointers by swapping CPU RAM pointers with GPU RAM pointers. The GPU kernel is being run as one of the final phases in the execution process. Finally, memory is transferred back by the Memory management unit following GPU kernel execution, and control returns to Java code.

2.3.2 *Optimizations*

Most common algorithms share identical bottlenecks. These include locality and bandwidth issues that can be solved or at least diminished through algorithm optimizations. Depending on the particular algorithm these can have different impacts in the overall performance of the software to be executed on the GPU.

Most devices have multiple memory spaces, each with its own set of characteristics based on their intended use. Global, local, shared, texture, and registers are forms of these memory spaces as shown in Table 1.

Memory	Chip location	Access	Scope	Lifetime
Register	On	R/W	Single thread	Thread
Local	Off	R/W	Single thread	Thread
Shared	On	R/W	Thread Block	Block
Global	Off	R/W	All threads + host	Host allocation
Constant	Off	R	All threads + host	Host allocation

Table 1: Device memory features.

Global, local, and texture memory have the greatest access latency, followed by constant memory, shared memory, and the register file Nvidia (2021).

This section presents a some optimizations used across different benchmarks and the type of issues that they solve.

Data structure layouts

As seen before, for a program to be able to scale, locality and data-level parallelism is necessary. It is harder to optimise data-intensive irregular applications that rely on pointers since the pattern of memory access is less predictable. Because of this, and as defended by Hirzel (2007), ”Object-oriented programs rely heavily on objects and pointers, making them vulnerable to slowdowns from cache and TLB misses”,if a program uses a pointer to access an object but the object is not actually in the cache, then the processor has to wait for the data to arrive for many cycles. A performance oriented data layout needs to avoid those waiting times.

In this section, the main data structure layouts used by most high-level languages will be presented in a scenario where three objects (figure 10) are to be stored according to different layouts. .

The Array of Structures represented in figure 11 is the preferred data structure for problems that require all fields of a structure at the same time.

By using this type of data-structure in this type of problems, spatial locality is greatly improved. On the other hand, as noted by Farber (2011) ”Many legacy applications store data

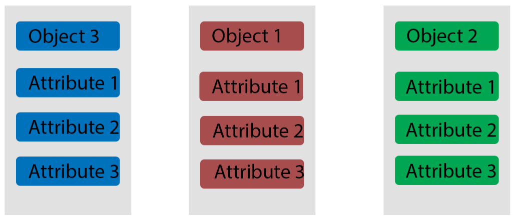

Figure 10: Example of three objects to be stored in memory.

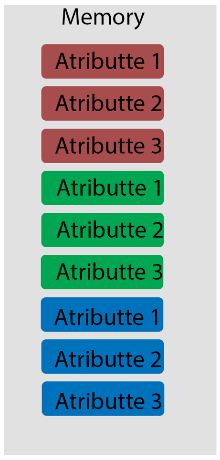

Figure 11: Array of Structures memory representation.

as arrays of structures (AoS) that can lead to coalescing issues. From a GPU performance perspective, it is preferable to store data as a structure of arrays (SoA)." Which is also defended by K. Kofler and Fabringer "programs with AoS data layout are not well suited for GPUs. SoA delivers a much higher performance on all GPU/program combinations".

The Structure of Arrays illustrated in figure 12 provides better locality for algorithms which do not need all fields of the original structure at the same time.

This type of data layout favours problems when similar attributes of different objects interact with each other. This type of data structure layout also improves data level parallelism and makes full use of the memory bandwidth of a GPU even when individual elements of the structure are used as shown by Farber (2011).

The Array of Pointers layout (figure 13) is popular due to the abstract data types it supports. The concrete type of data object is abstracted by using pointers. This form of layout is widely used in virtual machines (e.g., Java) systems with automatic memory management systems such as garbage collectors.

The impact of different data-structure layouts has been a topic of study by many experts. Hirzel (2007) shown that miss rates can vary from ten to twenty percent from layout to layout, and those results have a direct impact in the overall performance as B. Calder and Austin (1998) mentions.

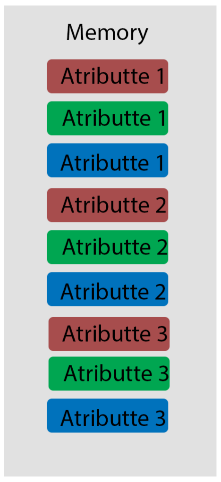

Figure 12: Structure of arrays memory representation.

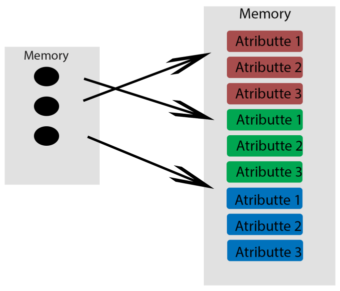

Figure 13: Array of Pointers memory representation.

Communication CPU-GPU

Due to the typical CPU-GPU memory architecture, parallelizing code for GPUs is difficult. The GPU and CPU each have their own memory, and each processing unit can only access its own memory efficiently. When CPU or GPU programs require data structures from outside their memory, they must copy data between the divided CPU and GPU memories explicitly.

Optimizing communication refers to the transformation of communication patterns into acyclic patterns. Cyclic communication patterns result from naively copying data to GPU memory, spawning a GPU function, and copying the results back to CPU memory. An acyclic communication pattern is created by copying data to the GPU in the preheader, spawning many GPU functions, and copying the result back to CPU memory in the end as presented in Nvidia (2012).

The communication CPU-GPU takes place over a PCI bus, and the bandwidth is determined by the number of lanes used on the bus. Typical GPUs use the PCIe 2.0/3.0 link, which can theoretically handle 500MB/s and 1TB/s.

Memory Coalescing

Perhaps the single most important performance consideration in programming for the CUDA architecture is the coalescing of global memory accesses. Global memory loads and stores by threads of a warp (of a half warp for devices of compute capability 1.x) are coalesced by the device into as few as one transaction when certain access requirements are met. Nvidia (2012)

The requirements are simple: the threads of a warp's concurrent accesses will coalesce into several transactions equal to the number of cache lines required to service all of the warp's threads. All accesses are cached by default through L1, which has 128-byte lines. Caching only in L2, which caches shorter 32-byte segments, can be useful for scattered access patterns to reduce over fetch.

The concepts of coalescing are demonstrated in the following simple examples. These case studies assume that all accesses are cached through L1, as is the default and that all accesses are for 4-byte words. Any CUDA-enabled device can achieve the first and most basic case of coalescing: the k-th thread accesses the k-th word in a cache line. All threads don't need to participate in this process. If a warp's threads access adjacent 4-byte words (e.g., adjacent float values), a single 128B L1 cache line and thus a single coalesced transaction will handle the memory access.

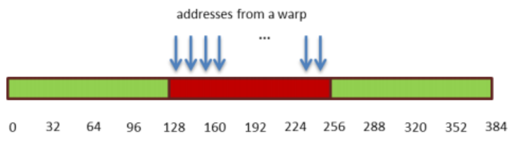

Figure 14: Coalesced access – all threads access one cache line (Nvidia, 2012).

The red rectangle in figure 14 indicates that this access pattern results in a single 128-byte L1 transaction. All data in the cache line is fetched even if some words of the line were not requested by any thread. Two 128-byte L1 caches will be requested if sequential threads in a warp access memory are sequential but not aligned with the cache lines as in figure 15.

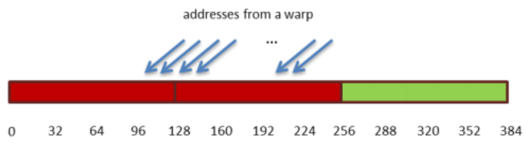

Figure 15: Unaligned sequential addresses that fit into two 128-byte L1-cache lines (Nvidia, 2012).

A similar effect is observed for non-caching transactions, except at the level of the 32-byte L2 segments. In most cases, memory allocation ensures some level of alignment, usually at least 256 bytes. As a result, thread block sizes that are multiples of the warp size make memory accesses by warps that are aligned to cache lines easier.

Shared memory

Following the work of Nvidia (2012), because it is on-chip, shared memory has much higher bandwidth and lower latency than local and global memory—provided there are no bank conflicts between the threads, as detailed in this section. Shared memory is divided into equally sized memory modules (banks) that can be accessed simultaneously to achieve high memory bandwidth for concurrent accesses. As a result, any memory load or store with N addresses that span n memory banks can be serviced at the same time, resulting in an effective bandwidth that is N times that of a single bank. The accesses are serialized if multiple addresses of a memory request map to the same memory bank.

A memory request with bank conflicts is split into as many separate conflict-free requests as needed by the hardware, reducing the effective bandwidth by a factor equal to the number of separate memory requests.The only time this does not apply is when multiple threads in a warp address the same shared memory location, causing a broadcast. Shared memory banks are organized so that each bank has a bandwidth of 32 bits per clock cycle and successive 32-bit words are assigned to each bank. The bandwidth of shared memory is 32 bits per bank per clock cycle.

Shared memory allows threads in a block to work together. When multiple threads in a block need to access the same data from global memory, shared memory can be used to only access the data once. By loading and storing data in a coalesced pattern from global memory and then reordering it in shared memory, shared memory can also be used to avoid uncoalesced memory accesses. Aside from memory bank conflicts, there is no penalty for non-sequential or unaligned accesses by a warp in shared memory. Nvidia (2012)

Coarsening

On GPUs, thread coarsening combines the work of multiple threads into one. The goal is to execute a smaller number of larger, coarser-grained threads than previously. Thread coarsening reduces parallelism in the application, which can have both positive or negative implications for run-time requirements. These effects scale to some extent as the coarsening factor increases, but there will come a point where the application's ability to exploit the available parallelism efficiently is limited by the number of threads available. By increasing the coarsening factor, a kernel's resource consumption, such as register consumption, rises, eventually resulting in lower occupancy. An increased workload per thread can also increase the pressure on the cache in some kernels. Thus, there is a performance trade-off, and the challenge is both to determine whether coarsening should be applied to a given kernel, and if so, what the optimal coarsening factor is, i.e. the optimal number of threads to merge, that maximises the overall performance.

Thread-level coarsening performs the same amount of work with fewer threads in each thread block. As a result, the total number of thread blocks remains unchanged, while the number of threads per block and the total number of threads decreases. This does not always imply a reduction in occupancy, as smaller thread blocks can be scheduled at the same time if enough resources are available. However, because each SM has its own set of limitations in terms of registers, shared memory, and concurrently runnable thread blocks, a decrease in occupancy is inevitable as defended by Stawinoga (2018). At the same time, it's important to avoid destroying any current memory coalescing when selecting the stride parameter. Magni and Dubach (2013)

In block-level coarsening, the number of threads per block remains constant when coarsening at the block level, so the number of executed thread blocks is decreased by the coarsening factor. As with thread-level coarsening, resource requirements per block, in terms of register and shared memory utilization, will normally increase as the workload on each thread block increases. Which blocks will be combined is now determined by the stride parameter. The stride has no effect on memory coalescing, unlike thread-level coarsening, since the original memory access patterns of the uncoarsened code within each block are retained. Changing the stride helps you to have more control over the order in which blocks are scheduled. Stawinoga (2018)

Coarsening some kernels will dramatically increase cache pressure. Higher coarsening factors will result in lower occupancy and less active threads in practice, so the effect is not always linear. However, in the presence of cache line re-use, the cache pressure often rises with the coarsening factor, and this usually outweighs the benefits of coarsening. When data is accessed in a streaming manner, with no data re-use or cache line re-use, cache pressure is not a problem.

Tiling

Tiling consists of identifying blocks/tiles of global memory content that multiple threads access. All the block data is loaded into on-chip faster memory. When the loaded data is no longer required, the block is generally discarded and a new one is loaded to be worked on. "Tiling is a crucial loop transformation for generating high-performance code on modern architectures to expose coarse grain parallelism in multi-core architectures and to maximize data reuse in deep memory hierarchies. Register tiling improves Instruction Level Parallelism and is critical for these architectures to maximize performance improvements." Rajaraman (2009)

Data privatization

The concept behind privatization is to make private copies of widely contended data structures so that each thread (or a subset of threads) can access one. This technique objective is removing dependencies that occur due to the presence of variables that are written and/or read by different threads at the same time. The advantage is that private copies can be accessed with much less contention and, in many instances, with much lower latency. These private copies can dramatically increase the throughput for updating the data structures. ”The downside is that the private copies need to be merged into the original data structure after the computation completes. One must carefully balance between the level of contention and the merging cost.” Solihin (2015)

There are two occasions when a variable can be characterized as ”privatizable”. The first scenario occurs when the variable is (written) before it is (read) by the parallel task in the sequential order of the original program. The conflict/dependency would be eliminated if the task writes to its (private) copy rather than the shared one. The explanation for this is that the program’s sequential order guarantees that the value read by the task is the same as the value written by the task, eliminating any potential conflicts caused by other threads accessing the same variable.

In the second example, the read for the attribute happens before the write, but the value the parallel task is seeking to read is one that was previously established during program execution. Given this, any disputes or dependencies would be addressed during execution if each parallel task wrote to its own private copy of the variable, as they would all read a value known ahead of time and then write their own proper values on their copies.

Padding data

Shared memory space is divided into several banks in the same way that global memory channels are. Banks are allocated to consecutive four-byte data sets. At any given time, each bank can only serve one address. ”When threads in a half warp access K different addresses within one bank, the accesses are serialized N times.” A. Khan and Assayony (2014) To avoid serialize the data accesses padding can be used. Padding consists of adding dummy data that won’t be used. By adding this extra dummy space, the necessary data can be located in a new bank avoiding a bank conflict. Padding can also be used to align data improving the spatial locality of regularly used data. Inter-padding and intra-padding are the two types of array padding. Intra-padding inserts unused spaces into one array, while inter-padding adds dummy space between array variables. Intra-padding can be used to adjust the array access stride, which affects the conflict degree, and thus solves the bank conflicts problem. Rivera and Tseng (1998)

2.3.3 *Benchmarks*

Benchmarks are a common element of current computer system research, and numerous benchmark sets have been established for various reasons. J. Stratton and Hwu (2012)”

Throughput computing, especially targeting a heterogeneous CPU+Accelerator system architecture, has been steadily growing and evolving as an application and architecture area of great interest." The CPU+GPU accelerator system design, which generally uses OpenCL and NVIDIA, has been the most frequently explored version thus far.

SHOC

The Scalable Heterogeneous Computing (SHOC) benchmark suite was created to provide a consistent way to assess the efficiency and reliability of non-traditional high-performance computing architectures A. Danalis and J.Vetter (2010). SHOC benchmarks are distributed using MPI and effectively scale from a single device to a large cluster. Stress assessments and success tests are the two main types of SHOC benchmarks. On several synthetic kernels, as well as typical parallel operations and algorithms, the other tests assess a variety of aspects of machine efficiency. The scope of the performance assessments, as well as the essence of the system capacity they exercise, are divided further. Each benchmark has several versions, serial, embarrassingly parallel and true parallel. The serial versions, execute on a single node and use only one device. The embarrassingly parallel performs on multiple devices or nodes of a cluster without communication between devices or nodes. Finally the true parallel benchmark measure multiple nodes, with one or more devices per node, including all communication.

SHOC provides a Compute Unified Device Architecture (CUDA) version of many of its benchmarks to compare with the OpenCL version. Kernels have been designed for both languages because they support essentially the same constructs. Furthermore, although OpenCL supports CPU devices, the language is designed for GPU-like devices, so exposed constructs like local programmer-managed memory can naturally exploit GPU-like devices better than standard CPUs.

Basic parallel algorithms, such as the Fast Fourier Transform (FFT) or the parallel prefix sum, are measured in this benchmark suite. These algorithms represent typical parallel processing tasks and can be found in a large number of kernels in real applications. The performance characteristics of these algorithms differ significantly. Several of the benchmarks are highly configurable, allowing them to cover a wide variety of problem sizes and other input parameters.

The MD test records the time it takes to calculate the Lennard-Jones potential from molecular dynamics using a simple pairwise estimate. Based on the potential field produced by all particles into a cutoff region, each thread calculates the acceleration for one particle. Coalesced global memory accesses are used by the kernel making this algorithm an interesting study case to explore.

Stencil2D is another interesting study case. It evaluates the results of a two-dimensional nine-point stencil measurement. First, each thread group copies their portion of the data from global memory to local shared memory, including a one-element ghosting field. Each thread computes the stencil for a single matrix element using only shared memory data. All of these can be studied to answer the objectives of this master dissertation.

The two-dimensional histogram of a two-dimensional input is computed by the histogram application. The input has the property of being largely concentrated in the histogram's middle. Privatization is an optimization explored in this test.

Parboil

"The Parboil benchmarks are a collection of accelerated, heterogeneous applications with a focus on throughput". J. Stratton and Hwu (2012) Iterative techniques, dynamic task kernels, dense array operations, and data-dependent memory access patterns are among the benchmarks. With applications taken from astronomy, biomolecular modelling, fluid dynamics, image processing, astronomy, and dense and sparse linear algebra, the Parboil benchmarks test memory bandwidth, floating-point performance, latency tolerance, and even cache effectiveness. Each benchmark is based on a scalable and appropriate method for the datasets supplied with the benchmarks. In addition, each benchmark has several implementations of the method to allow various benchmarking comparisons or circumstances.

The parboil benchmark suite provides a simple run-method were the desirable benchmark can be run with different sizes. It also provides a base CPU version as well as OMP, CUDA and OpenCL. All of the OpenCL and base versions off most algorithms were tested (figure 16) to identify attractive study cases.

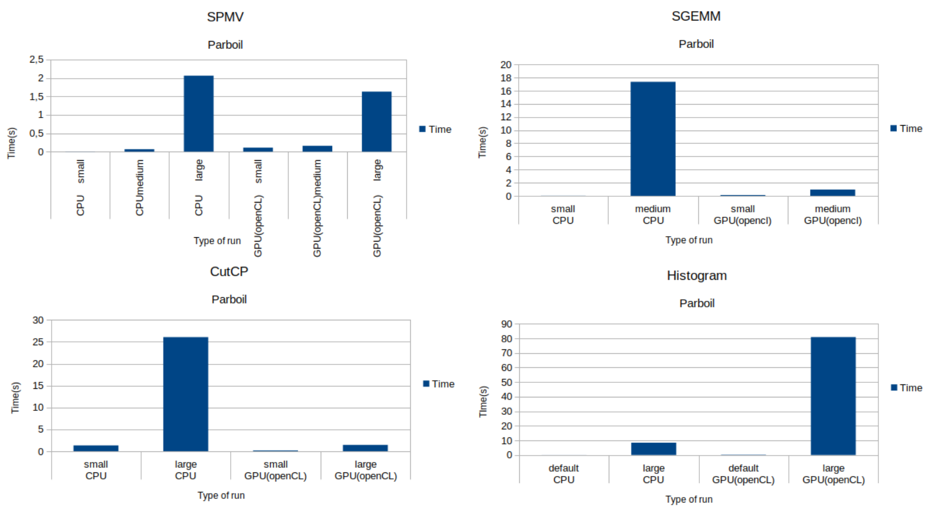

Figure 16: Time of execution of different Parboil algorithms.

Similar to the SHOC benchmarks, all the algorithms present are often used in real applications. Some of these algorithms are very similar to the ones present in SHOC which enables a direct comparison of techniques used to optimize both benchmarks.

2.4 SUMMARY

To take the most advantage of a computer system, one must know about the existing hardware solutions like memory hierarchy and how they work together to achieve maximum performance. By producing an algorithm with good locality the time wasted waiting for data is reduced overall.

Another key factor to improve performance is to structure the data in memory to achieve good data-level parallelism. The GPU, under these circumstances, can be used as an ideal accelerator to execute the heavy work of software.

Current high-level languages like Java can already be used to write code for GPU execution with some limitations. Being Aparapi one of the main platforms that enables this, it already supports the programmer with some optimizations and has the capability to harness most of the power the GPU has to offer.

Some of the optimizations presented in this chapter are not implemented automatically and so the programmer is the one responsible for applying the necessary changes to the software to further improve Aparapi performance.

To compare the Aparapi with other commonly used performance-oriented solutions, the SHOC and Parboil benchmark suites can be used since they cover the most important optimizations for GPUs. This collection of benchmarks has algorithms that have real-life applications ranging from memory-bound to computer-bound. This characteristic makes them ideal candidates to compare performance results with Aparapi.

3 DEVELOPMENT

In this chapter the methodology implemented throughout the development of this master dissertation is presented. By developing different algorithms with different computational workloads and different data layouts the performance impact of run-time OpenCL generation, data communication costs and the effect of using Java objects for GPU execution can be measured.

By analysing different benchmark suits, algorithms with different bottlenecks were used as inspiration to develop similar software written in Java/Aparapi so that different optimizations could be studied in depth while finding the current limitations of Aparapi.

To compare Aparapi with other performance-oriented software solutions, the same benchmarks that were studied were run and its execution times were recorded. After this, the same optimizations and software execution characteristics were replicated in Aparapi to access the performance of the best implementations.

3.1 SELECTING CASES OF STUDY

The parboil benchmark suite contains several interesting benchmarks that have practical utility and contain typical bottlenecks of unoptimized implementations as well as the optimizations that can overcome these. Table 2 presents the typical bottleneck and proposed optimizations that the benchmark suits have.

The stencil benchmark implemented in the simplest version on Parboil has locality issues that can be solved with coarsening and tiling two of the techniques that this master dissertation proposed to study.

The importance of data layout transformations to overcome bandwidth problems can be further explored in the SPMV algorithm of Parboil. This will enable a more in-depth study of more complex data layouts and their performance. It also contains optimizations regarding padding and has a scalar and vector version of the algorithm.

The Histogram present in Parboil can also be a case study to investigate the privatization technique to avoid contention and bandwidth issues. This version contains different kernels for different execution stages which are relevant to the study of kernel creation costs.

Benchmark	**Typical bottleneck**	**Pruposed optimizations**	**Implemented optimizations**
MD (SHOC)	Locality	Data layout transformation	Data layout transformation
SPMV (Parboil)	Bandwidth	Data layout transformation	Data layout transformation
Histogram (Parboil)	Contention, Bandwidth	Privatization, Scatter-to-Gather	Privatization
MD (SHOC)	Locality	Data layout transformation, Privatization	Data layout transformation, Privatization
Stencil2D (SHOC)	Locality, Bank Conflicts	Privatization, Tilling, Coarsening	Privatization, Tilling, Coarsening

Table 2: Algorithms selected from the Parboil and SHOC benchmark suits.

The SHOC benchmark suite also includes several interesting benchmarks that help achieve results to respond to the proposed objectives. The MD algorithm takes advantage of coalesced global memory accesses and faces locality issues once translated to Java using objects.

The Stencil version in SHOC is similar to the one in Parboil but in a two-dimension space. This algorithm uses overlapping tiling and shared memory to improve locality. This stencil algorithm together with the study of the Parboil version is an indispensable case to study the various optimizations used in both. This resulted in mixed optimizations from both benchmarks.

The details of implementation, observed bottlenecks and Aparapi limitations while replicating these versions are described in this chapter.

Note that all the benchmarks include a CUDA and OpenCL version, but only the OpenCL version was studied. This decision was made because and as stated by A. Danalis and J.Vetter (2010) "while CUDA enjoys an advantage in popularity and elegance, it confines codes to NVIDIA GPUs. OpenCL has tremendous potential for code portability, but is rather verbose and may not meet the performance requirements of all developers." Since one of the reasons for using Java is because of portability in the first place, and Aparapi tackles the OpenCL verbosity issue it is only fair to compare it with similar solutions that use OpenCL.

3.2 THE N-BODY AND MD ALGORITHMS

In this section the N-Body and MD algorithms are both explored. The N-Body problem was used as a first experiment to familiarize with Aparapi and its capabilities. The code wasn't inspired in any benchmark but its kernel is similar to the MD algorithm. The MD algorithm of the SHOC benchmark suite follows the Lennard-Jones method in which the most significant difference when compared to the N-Body developed is that their is a cut-off distance. This is, while in the N-Body algorithm all the bodies interacted with each other, in the L-J method only the closest neighbours interact.

3.2.1 *N-Body*

When *N* is greater than two, the N-body problem is notorious for being difficult to solve. It all began in 1687 with the publication of Isaac Newton's "Principia". The equations of motion and gravity inherent in his work converted our seemingly unpredictable cosmos into a precisely timed machine of clockwork predictability. Given their current positions and velocities, Newton's equations might potentially be used to calculate the locations of the solar system's bodies at any distant time. Despite their beauty, Newton's equations only lead to a straightforward solution in one scenario. This approach only works when two bodies orbit each other without being influenced by anything else. In most situations, having only one more body causes the whole motion to become inherently chaotic. Newton's motion equations can be thought of as analytic solutions. Mathematicians Erns Bruns and Henri Poincaré proved in the late 1800s that there are no general analytic solutions and thus the N-Body problem cannot be solved. Even if there isn't a useful analytic solution, approximate solutions can be sought. For most N-Body systems, the motion of the system must be broken down into many small pieces and solved one at a time, assuming everything else remains constant. The main algorithm can be consulted bellow.

```
myBody a {
    position p;
    velocity v;
    mass m;
};

for( N_other_Bodies b: Universe){
    invDistance d=calculateDistance(a,b);
    forces f= calculateAtracttion(m,invDistance);
    newVelocity nv= calculateVelocity(a,b,f)
}
refreshMyVelocity(a,nv);
refreshMyPosition(a);
```

Through the N-Body use case, some of the most practical and simple examples of how GPU execution can improve a program performance can be demonstrated.

In this first experiment, some performance optimizations impact regarding locality and data transference between CPU-GPU are studied. This first trial will be implemented in a naive way to uncover some basic and simple optimizations in Aparapi that have great performance impact.

Identifiable bottlenecks

To better understand the performance results gathered, the OpenCL code generated by Aparapi needs to be studied. The first issue arises from the way Aparapi handled the programmed array of objects ”Body”. As can be seen in the code bellow, the data is stored in memory following an array of pointers to a Body type structure. The work of B. Calder and Austin (1998) mentions that this type of data layout doesn’t exploit locality at the global memory level of the GPU.

```
typedef struct Body_s {
  float x;
  float y;
  float z;
  float vx;
  float vy;
  float vz;
} Body;

typedef struct This_s{
  __global Body *bodies;
  int passid;
}This;
int get_passid(This *this){
  return this->passid;
}
float Body__getZ(__global Body *this){
  return this->z;
}
(...)
```

The second main problem with the developed code can be found by profiling the kernel. Aparapi offers a simple kernel profiling tool that can be enabled to study the times of transference of each array that is passed to/from the GPU as well as the kernel execution time. By doing this, it is possible to understand that the Body data is being continually transferred between CPU-GPU after every iteration of the algorithm.

Optimizations

To solve the data locality issue in the global memory, the data layout generated by Aparapi needs to be changed to a better one. This can be achieved by transforming the Body object into different arrays. The array of objects ”Body” illustrated in Figure 23 where each body had six floats to represent position and speed was deconstructed into two different arrays. The first one has the values of the positions of all bodies and the second has the velocity parameters.

As can be seen bellow and inspecting the new OpenCL generated, by modifying the source code deconstructing the object useful data into arrays, the corresponding generated openCL is now in an Array of Structures data layout similar to Figure 11.

```
typedef struct This_s {
  __global float *val$xyz;
  __global float *val$vxyz;
  int passid;
} This;
(...)
```

Regarding the second problem, Aparapi already has various methods that try to minimize the number of times data is transferred. Thus, by using explicit memory management and/or employ the "passes" parameter in the kernel execute method the programmer has full control over the data that is being passed to/from the GPU. Furthermore, by doing so, the kernel creation and other necessary verification steps can be bypassed and thus are only made once improving the time taken to initiate the computation on the device.

```
(...)
kernel.execute(size,steps);
(...)
```

The code can be further improved by transforming the data from a Array of Structures layout to Structure of Arrays as in Figure 12 and as mentioned by Farber (2011). The data of every "Body" object should be organized in six different arrays, one for each coordinate and one for each velocity.

```
typedef struct This_s {
  __global float *val$x;
  __global float *val$y;
  __global float *val$z;
  __global float *val$vx;
  __global float *val$vy;
  __global float *val$vz;
  int passid;
} This;
(...)
```

3.2.2 *Molecular Dynamics*

”Molecular dynamics simulations are important tools for understanding the physical basis of the structure and function of biological macromolecules” Karplus and McCammon (2002). The action of proteins and other biomolecules is captured in atomic detail and at very fine temporal resolution in these simulations. The impact of molecular dynamics (MD) simulations in molecular biology and drug discovery has expanded dramatically in recent years Hollingsworth and Dror (2018), Major advancements in simulation speed, precision, and usability, as well as the abundance of experimental structural data, have expanded the attractiveness of biomolecular simulation to experimentalists—a pattern that can be seen in, but not limited to, neuroscience. Simulations have been useful in understanding the functional structures of proteins and other biomolecules, determining the molecular basis for disease and designing and optimizing small molecules, peptides, and proteins. Importantly, such simulations will determine how biomolecules can react to perturbations including mutation, phosphorylation, protonation, or the addition or removal of a ligand at the atomic level. MD models are often combined with a range of laboratory structural biology methods, such as x-ray crystallography, cryo-electron microscopy (cryo-EM), nuclear magnetic resonance (NMR), electron paramagnetic resonance (EPR), and Förster resonance energy transfer (FRET). The algorithm follows the method of Lennard-Jones. The Lennard-Jones potential is a generalized paradigm that captures the basic characteristics of a simple atom and molecule interactions. The Lennard-Jones possibility is a pair potential, which means it doesn’t cover three- or multi-body interactions. It can be described as a simplistic model that captures the following properties: Two interacting particles repel each other at near range, attract each other at a medium range, and do not interact at a further range.

Identifiable bottlenecks

The MD algorithm present in SHOC is composed of two distinct parts. It starts by building a neighbour list of all the pairs within cutoff distance. This process is made on the CPU. After the neighbour list is built, the kernel is executed on the GPU.

Currently, this benchmark has no special optimizations implemented in SHOC but if the same implementation is made in Java it faces great locality issues since Aparapi generates OpenCL code following an AoP layout. One thing important to mention is that the different sizes of the data set in this particular benchmark are small when compared to other benchmarks here explored. This happens because the process to build the neighbour list is sequential and it may need several hours to build for bigger data sets.

Implemented optimizations

To solve the present bottleneck the solution is to transform the data layout from AoP to SoA according to Farber (2011) and following the same pattern as in the N-Body algorithm. This means that instead of using Java objects, and therefore, using pointers, separate arrays for position and forces in the three dimensions should be used. By implementing this optimization, the number of instructions needed diminishes by reducing the number of memory accesses needed and spatial locality improves.

3.3 STENCIL2D

Stencils are a popular computational pattern used in a wide range of scientific applications, including computational electromagnetics Taflove (1995), PDE solution using finite difference or finite volume discretizations Smith (2004), image processing for CT and MRI imaging J. Cong and Zou (2011). As described by J. Holewinski and Sadayappan ”Stencil computations involve the repeated updating of values associated with points on a multi-dimensional grid, using only the values at a set of neighboring points. For multi-core processors, stencil computations are often memory bandwidth bound when the collective data for all grid points exceeds cache size, since each grid point is accessed at each time step”. Recent studies have been working on using GPUs for solving this kind of computational pattern. The use of overlapped tiling on GPUs has recently shown promise for high performance as mentioned by Meng and Skadron (2011).

3.3.1 *Identifiable bottlenecks*

When designing GPU applications, several inefficiencies may emerge. GPUs have a high off-chip memory bandwidth (up to 192 GB/sec for the GTX 580), but this bandwidth can only be achieved using coalesced access. Since data from the off-chip memory is transmitted to the GPU system in contiguous blocks, high bandwidth can only be obtained when requests from concurrent threads in a warp fall within those contiguous blocks. As threads visit non-contiguous memory locations, the reached bandwidth is always much smaller than the theoretical max, resulting in stalling and wasted compute cycles. Another cause of inefficiency is branch divergence. Threads in a warp that follow divergent control paths are serialized, wasting compute cycles once again.

Traditional approaches to time tiling stencil computations for data reuse on CPUs do not work well on GPUs because they result in uncoalesced memory access and divergent thread branching. Another issue is shared memory that has been introduced as a banked memory scheme. A bank dispute happens when simultaneously running threads in a block make requests to shared memory locations in the same bank. The requests are serialized. As a result, simultaneously running threads can access data from separate banks to make the most use of shared memory.

Data exchange of adjacent tiles makes tiling for stencil computations more difficult. As adjacent tiles are computed by separate processors, cells along the tile's boundary are often required by computations in nearby tiles, necessitating communication between tiles. Data from neighboring cells is needed to compute a stencil on a grid cell. The halo area is the name given to these cells.

Since surrounding tiles will update the values in these cells, the halo region must be re-read from global memory for each time phase on GPU devices. This reduces the amount of re-use that can be done in shared memory before returning to global memory for new records.

Other tiles may need the data in their halo regions, so the new cell values generated in each time step must also be re-written to global memory. In addition to the cost of going to global memory for each time step, global synchronization is needed to ensure that all nearby tiles have finished their computation and written their results to global memory before new halo data is read for each tile. Overlapping tiling has been proposed as a solution to these problems. S. Krishnamoorthy and Sadayappan This technique allows a reduction in the data sharing requirements for stencil computations by adding redundant computation. Instead of requiring tile synchronization after each time stage to update the halo region, each tile computes the necessary values for the halo region redundantly. This enables us to perform time tiling effectively and achieve high efficiency on GPU goals.

3.3.2 *Implemented optimizations*

To take advantage of the GPU memory hierarchy, results must be cached in shared memory wherever possible. The amount of space required needs to account for the redundant computations in order to produce high-performance code using overlapped tilling. In this particular case, it will have a size of:

```
(...)
@Local float[] sh=new float[(lRows+2)*(lCols+2)];
(...)
```

Thread barriers are used between stencil operations to guarantee that all threads have completed the calculation for one stencil operation before any thread begins calculating a value for the next. The thread blocks compute independently, but the threads inside a thread block must be coordinated since one thread may generate a result that another thread in the next stencil operation requires. Fortunately, this barriers can be easily implemented through:

```
localBarrier();
globalBarrier();
```

Two different ways of coarsening were studied. The first and suggested by the benchmark developers uses a scheme of coarsening of 8 rows and every work-group of threads has a size of 256. The second one has 256 rows and work-groups of 8. This was an experiment made to measure the impact that coarsening can have if not developed with other optimizations in mind.

The final optimization done in the SOC benchmark suite was padding. By implementing 3 distinct versions of padding its overall impact can be observed. The most simple of these versions uses no padding while the second has a padding of 8 defined in the SHOC version. The last version has a padding of 16. This number was chosen since it can be used to solve bank conflicts but is big enough to cause a performance degradation.

As last, the same optimizations done in the N-Body and MD kernels were also applied in this algorithm to keep data communication costs to a minimum.

3.4 SPARSE MATRIX-VECTOR MULTIPLICATION

In sparse linear algebra, sparse matrix-vector multiplication (SPVM) is crucial. Sparse operations, in contrast to dense linear algebra's uniform regularity, experience a wide range of matrices, from the regular to the extremely irregular.

The accomplishments of the work done by Volkov and Demme (2008) and S. Barrachina and Quintana-Ortí (2008), on dense matrix operations, in how to achieve large percentages of peak floating-point throughput and bandwidth are owed in part to the dense matrix operations normal access patterns. Sparse matrix operations, on the other hand, must deal with a variety of irregularities in the underlying matrix representation.

CSR format:

$$\begin{aligned}\texttt{ptr} &= [0\ \ 2\ \ 4\ \ 7\ \ 9]\\ \texttt{indices} &= [0\ \ 1\ \ 1\ \ 2\ \ 0\ \ 2\ \ 3\ \ 1\ \ 3]\\ \texttt{data} &= [1\ \ 7\ \ 2\ \ 8\ \ 5\ \ 3\ \ 9\ \ 6\ \ 4]\end{aligned}$$

Figure 17: CSR format. Bell and Garland (2009)

"The compressed sparse row (CSR) format is the most popular, general-purpose sparse matrix representation."C. Shizhao and Zheng (2009) It explicitly stores column indices and nonzero values in arrays indices and data. The CSR format can display rows of varying lengths thanks to the third array of row pointers illustrated in Figure 18. In this section, two different sparse matrix-vector multiplication algorithms will be implemented. The first one assigns a thread to each matrix row, thus being called the scalar kernel. In the vector kernel, however, one warp is responsible for each matrix row.

3.4.1 *Identifiable bottlenecks*

The way threads inside a warp access the CSR indices and data arrays is the most important bottleneck of the scalar version. The column indices and nonzero values for a given row are stored in the CSR data structure together, but they are not accessed at the same time. Instead, each thread sequentially reads the elements of its chain, resulting in the pattern shown in 18.

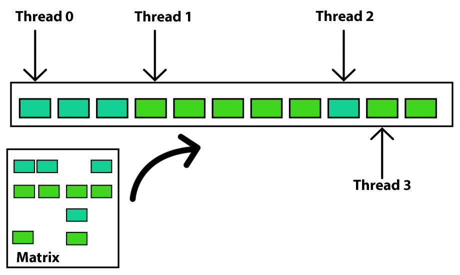

Figure 18: CSR arrays indices, data and the memory access pattern of the scalar CSR SPMV kernel.

"While CSR efficiently represents a broader class of sparse matrices, this additional flexibility introduces thread divergence" Bell and Garland (2009). When the scalar kernel is added to a matrix with an extremely variable number of nonzeros per row, several threads within a warp are likely to stay idle while the thread with the longest row iterates. The scalar kernel has a hard time with matrices whose distribution of nonzeros per row follows a power-law distribution. This form of thread divergence is less pronounced in the vector kernel since warps execute independently. On the other hand, effective vector kernel execution necessitates that matrix rows contain a larger number of nonzeros than the warp size. As a consequence, the vector kernel's output is affected by the size of the matrix lines.

3.4.2 *Implemented optimizations*

Because of the importance of SPMV operations, there is a substantial literature exploring numerous optimization techniques mainly consisting in different representation models. The one choosen was the CSR format.

After the data-transfer optimizations were implemented, the problem was divided in different local work groups of threads. This can be easily done using the Range functionalities offered by Aparapi with:

```
        Range range= Range.create(numRows,localWorkSize);
         (...)
         kernel.execute(range,passes);
         (...)
```

This way, Aparapi could mimic every detail of the CSR scalar implementation of the SHOC benchmark suite.

The vector version includes shared memory to store the values that a warp is working on and padding to avoid shared memory conflicts.

3.5 HISTOGRAM

Histograms are mathematical functions that count the number of observations that fall into disjoint bins. "They allow users to estimate a variable's probability distribution, and they're often employed to get the probability density function of the investigated variable by normalizing the histogram area to one." J. Luna and N.Guil (2012)

Histograms are useful tools in image segmentation and processing as well as pattern recognition and data analytics as studied by S. Kannan and G.Nalini (2014). The principle of the histogram algorithm is to perform the following operation over each element of the image.

```
for( many input values){
    histogram[value]++;
}
```

3.5.1 *Identifiable bottlenecks*

The first major bottleneck encounter was regarding the bandwidth usage and the overhead of global memory transactions. When only taking advantage of the global memory of the GPU the algorithm is very limited.

The second main concern is the synchronization needed to correctly calculate the output that can become a major performance concern after the first bottleneck is solved.

3.5.2 *Implemented optimizations*

The selected algorithm divides the histogram over a number of work-groups, each of which summaries its block of source data into a number of sub-histograms stored in on-chip local memory, to avoid the high degree of contention of access in global memory. Finally, these local histograms are combined into a single global histogram, which is used to store the overall output.

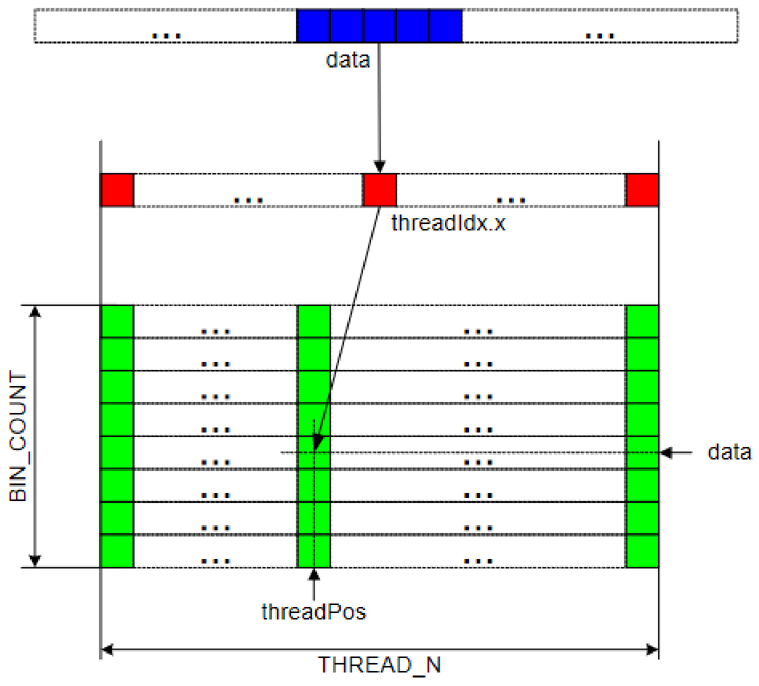

Figure 19: Shared array layout for histogram64. Podlozhnyuk (2013)

The per-block sub-histogram is kept in the s Hist array, which is shared memory. It is a 2D byte array with BIN_COUNTrows and THREAD_Ncolumns as shown in Figure 19. Although it is kept in fast on-chip shared memory, if feasible, a bank-conflict-free access pattern is required for optimal performance. The shared memory bank number is equal to (threadPos + data * THREAD N) / 4) % 16 for each thread with its unique threadPos and data value (which may be the same for some other threads in the thread block).

3.6 SUMMARY

In this chapter all the optimizations proposed in this master thesis objectives were studied with the help of the Parboil and SHOC benchmarks suites.

The N-Body problem was used to study locality, data transference and the impact of Java objects execution on GPU. It was also used to study the time needed to deconstruct these objects into more proper data layouts for GPU execution.

After these experiments, the N-Body kernel was changed to mimic the MD algorithm of SHOC. With this is pretended to gather data for a reliable comparison between Java/Aparapi and C++/OpenCL. Other algorithms present in the Parboil and SHOC were also developed in Java to help answer the objective optimizations that this master dissertation proposed.

A Java version of Stencil2D was developed according to the SHOC benchmark including privatization, tilling and coarsening optimizations. All of these were possible to replicate in Aparapi as well as most of the algorithm. The only thing that couldn't be simply replicated was the swap of pointers since Aparapi has limitations to pointer support on GPU.

The SPMV algorithm of Parboil using the CSR format was implemented in Java without any issue. Both the scalar as well as the vectorial versions were implemented using Aparapi available functionalities like work-groups of threads and range method in the case of the scalar version. Shared memory and padding were implemented in the vectorial one.

The last algorithm studied was the histogram of Parboil and SHOC. In these benchmarks, privatization and scatter-to-gather optimizations were proposed but only privatization could be implemented. This last study case also included atomic operations which were also not replicated in Java/Aparapi.

With the different algorithms and respective optimizations implemented in this chapter enough data should be gathered to answer the objectives of this master dissertation by comparing the performance results of the benchmarks with the results from Aparapi.

4 EXPERIMENT

In this chapter, all the relevant results from the Aparapi experiments will be exposed. These experiments range from naive implementations to state of the art GPU optimizations recording the results from all of these so that they can be compared with the ones gathered from SHOC. All the experiments were made in the same Linux environment with the software and hardware characteristics listed in table 3. The performance results here presented are an average of the results gathered after 8 separate runs of the same algorithm.

It begins with the results gathered from experiments that involve some basic optimizations that are present in all the algorithms that were developed. From these results, general conclusions can be drawn about the time consumption of transforming object arrays into simple native Java data types and the overall impact of caching the kernel in Aparapi.

After these optimizations are discussed, the N-body problem is used to show the performance impact of the different data layouts studied in this master dissertation. The comparison with algorithms of the SHOC benchmark start after this brief introduction to these general optimizations. For every algorithm chosen, the results of the different versions developed are exposed and compared between themselves in incremented order. When all the optimizations are in place and according to the ones present in SHOC, a final performance comparison is made between Aparapi and C++/OpenCL.

4.1 GENERAL OPTIMIZATIONS RESULTS

Some of the most significant optimizations that can be directly implemented with Aparapi are common to most algorithms including all the ones studied through this master dissertation. As so, in this chapter the results from these general optimizations will be exposed to understand their impact in the general software performance.

4.1.1 *Changing the memory layout*

Since one of the main optimizations that can be made in Aparapi is to not use Java objects, the study of the time needed to extract the data from objects into arrays is of the most importance. Two main kernels were developed to study this necessity. The first one simulates the need of calculating the average age of a country population. This kernel has a very light computation demand per data needed since only two sums are used for every memory access that it performs. The second kernel used in this experiment will be once more the N-Body kernel since it involves a lot more computational power than the previous one.

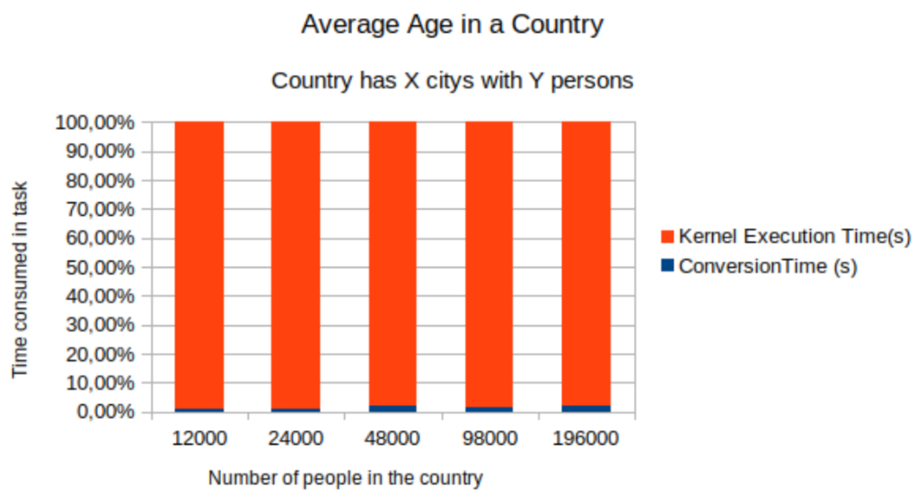

Figure 20: Time taken in % of converting an object to an array for kernel execution.

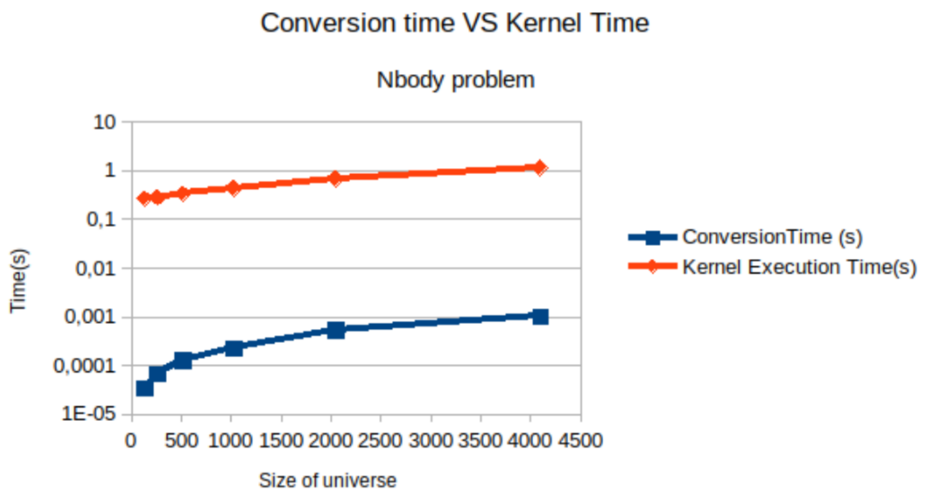

Figure 21: Time taken of converting an object to different arrays for kernel execution.

As can be seen from the results in Figures 20 and 21, the time consumed by reorganizing the necessary data from objects to arrays is small compared to the computation time needed. This happens in both a very light demand computation kernel as well as a heavy one since locality is much more exploited after the data layout conversion.

4.1.2 *Caching the kernel*

As mentioned before, Aparapi has kernel caching capabilities. This functionality can have a great impact, especially in small kernels. At runtime, the Java bytecode needs to be read, analysed and the correspondent OpenCL needs to be generated before the data is transferred to the GPU and the kernel executed.

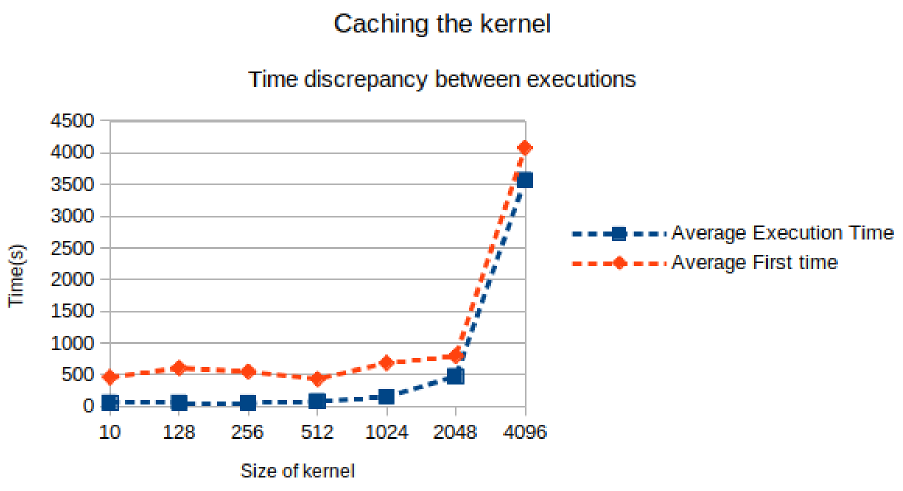

Figure 22: Time discrepancy of various runs of a matrix multiplication kernel.(Legenda no eixo y está errada)

As can be seen in Figure 22 every time the kernel isn't cached the execution time increases around 500 ms.

4.2 N-BODY OPTIMIZATIONS RESULTS

The first developed N-Body kernel was made using Java objects and without any kind of optimizations implemented. The number of bodies change from a minimum of 256 to 4096 and each execution does 1000 steps. The kernel execution time of this first version is presented in Figure 23 with the label "Objects".

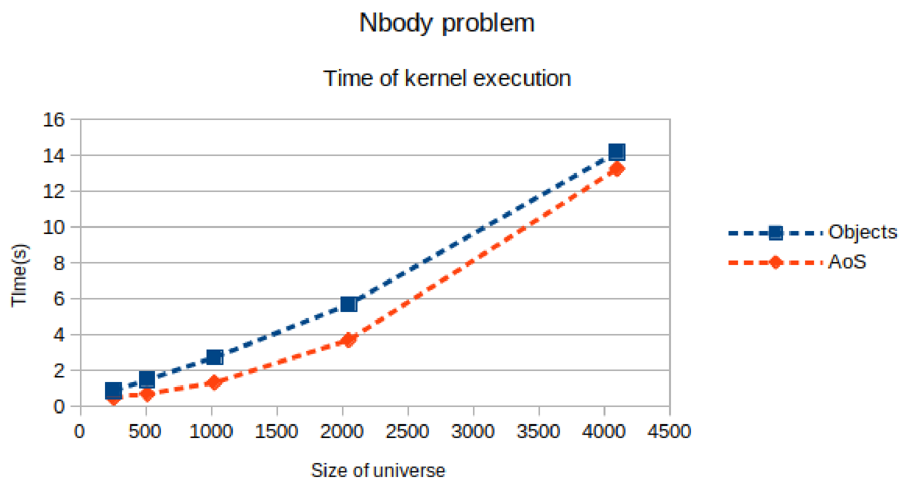

Figure 23: Time of a N-Body simulation changing the data layout to a Array of Structures.

After the algorithm was changed from using Java Objects to simple data arrays improving locality, the time taken for kernel execution diminished in all of the simulation sizes.

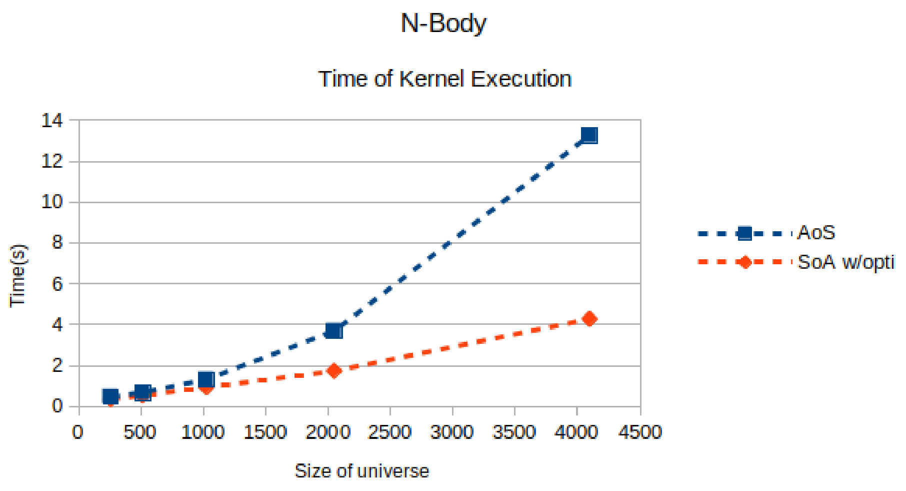

Figure 24: Time of a N-Body simulation with and without using data transfer optimizations and changing the data layout to a Structure of Arrays.

By further improving the developed code by organizing the data into a Structure of Arrays layout and by managing manually the data transfers, the performance results can drastically improve as shown in Figure 24.

4.3 STENCIL-2D OPTIMIZATIONS RESULTS

In the first stencil-2D version, no optimizations were put in place. This is, each thread calculates a single point of the stencil without using masks, padding, shared memory and the memory is managed automatically.

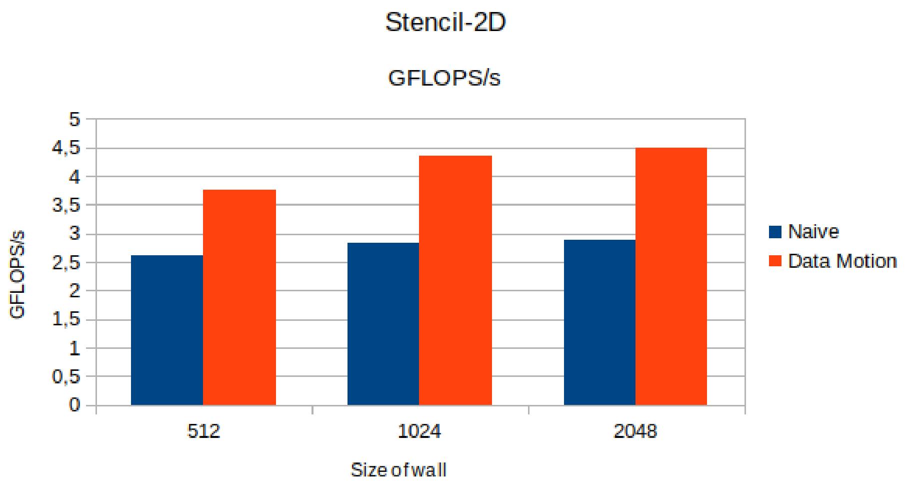

Figure 25: Gflops/s before and after data motion optimizations.

In 25, by explicitly managing the data in/out the GPU, the program managed to break the 3 GFlop/s barrier from the naive version improving for almost 50% in the largest size. However, this version still lacks data locality. To test the effects of coalescing the needed data a second stencil version was made following the advice previously presented in the state of the art 26.

After most of the data locality issues being solved, the next optimization made was to take advantage of the shared memory of the GPU. To ensure a correct comparison with the SHOC benchmark, the same default sizes and specifications were implemented.

By using shared arrays in Figure 29, most of the data that a GPU thread block needs is stored on-chip reducing the latency and greatly improving the software performance.

The next optimization that was studied was coarsening. By allowing a single thread to compute more than a single stencil point and by exploiting locality to take full advantage of the cache system, the performance results can be once more be improved dramatically.

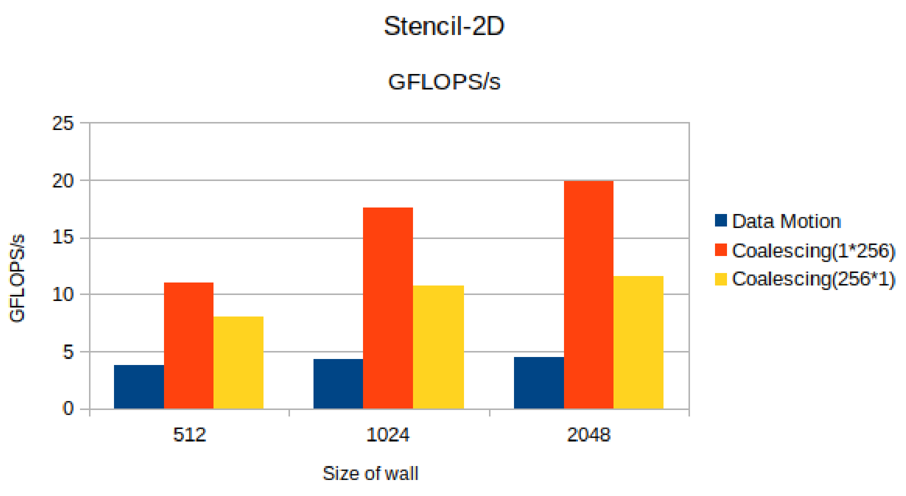

Figure 26: Gflops/s before and after coalescing data accesses.

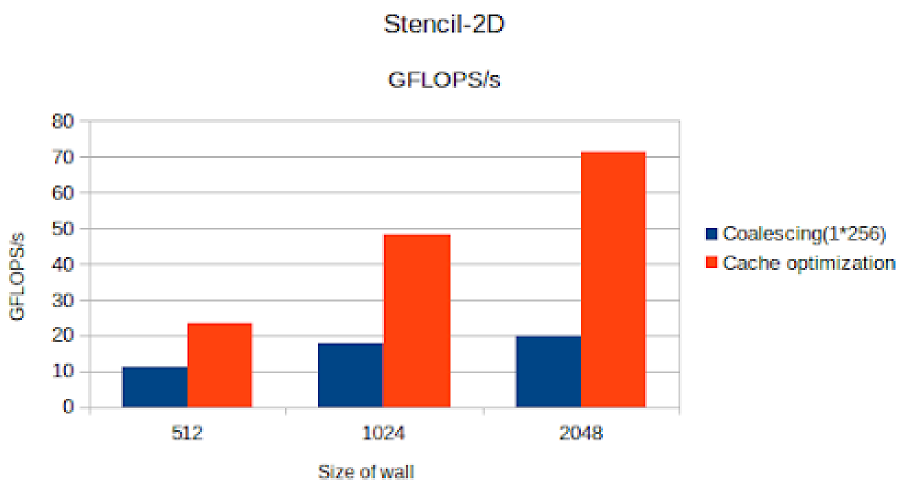

Figure 27: Gflops/s before and after the usage of shared memory.

4.3.1 *SHOC comparison*

To make a correct comparison with the SHOC benchmark Stencil-2D, the same optimizations were implemented. After these were added the same values for coarsening and padding were made common in both the Java and SHOC versions.

As can be observed in figure 29, Aparapi can only compete with the SHOC version when the kernel is big enough to compensate the overhead that Java has at the beginning of the kernel execution. This preliminary results indicate that the produced OpenCL that Aparapi generates can be extremely similar to code produced originally and explicitly in C++/OpenCL.

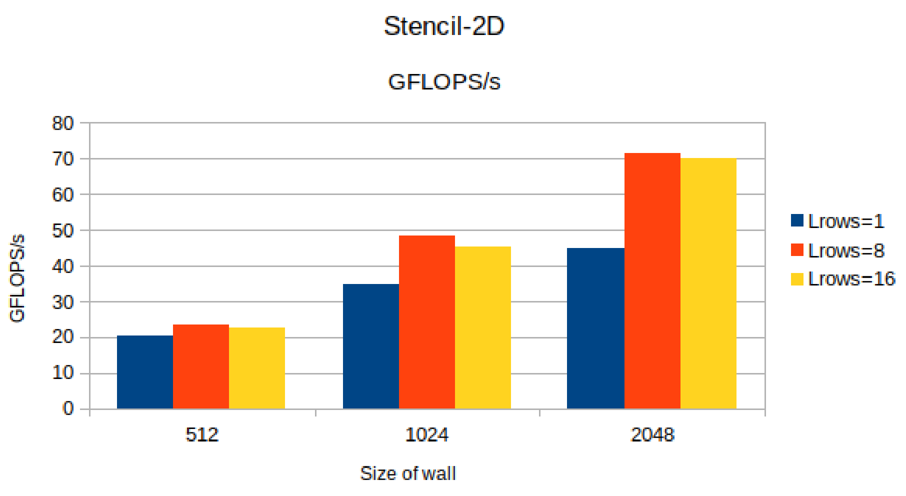

Figure 28: Gflops/s by using different coarsening factors.

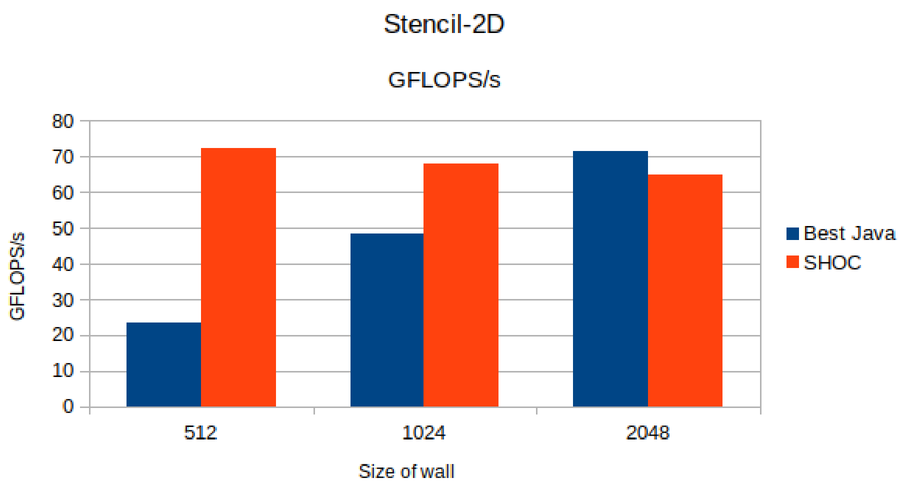

Figure 29: Gflops/s comparison between the two solutions.

4.4 MOLECULAR DYNAMICS OPTIMIZATIONS RESULTS

As previously mentioned, the major bottleneck that the Java version has in comparison to the SHOC version is made by a lack of data locality. Thus, the major optimization that can be implemented is to avoid using Java objects preferring a Structure of Arrays data layout.

Once more, in figure 30 can be verified that the data layout chosen by the developer has an enormous impact on the software performance.

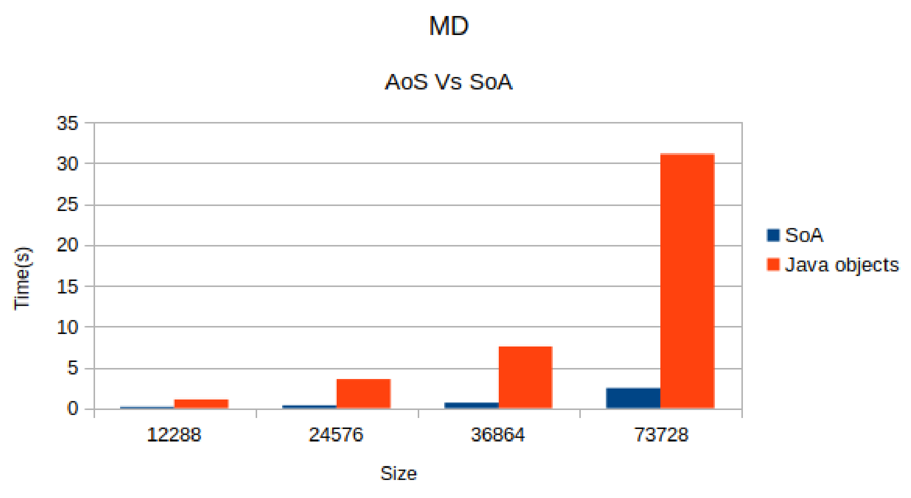

Figure 30: Time of kernel execution of the Lennard-Jones algorithm with 128 neighbours.

4.4.1 *SHOC comparison*

Following the trend of the previous algorithms, the Java/Aparapi MD version struggles in performance results when compared to the C++/openCL scores for small sized data-sets.

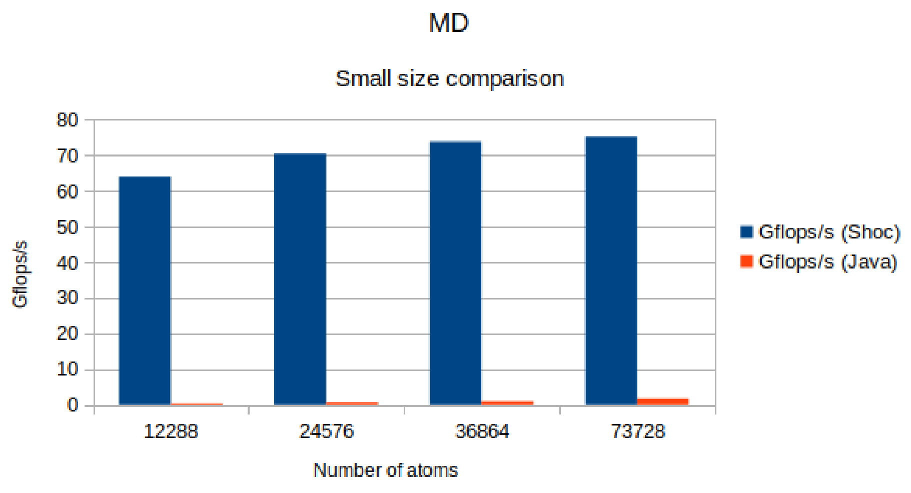

Figure 31: Time of kernel execution of the Lennard-Jones algorithm with 128 neighbours in both versions.

In this particular benchmark, Aparapi as specially bad results overall but the OpenCL kernel generated is identical. If however, some tests with sizes which are bigger then the default are made, both versions have similar results.

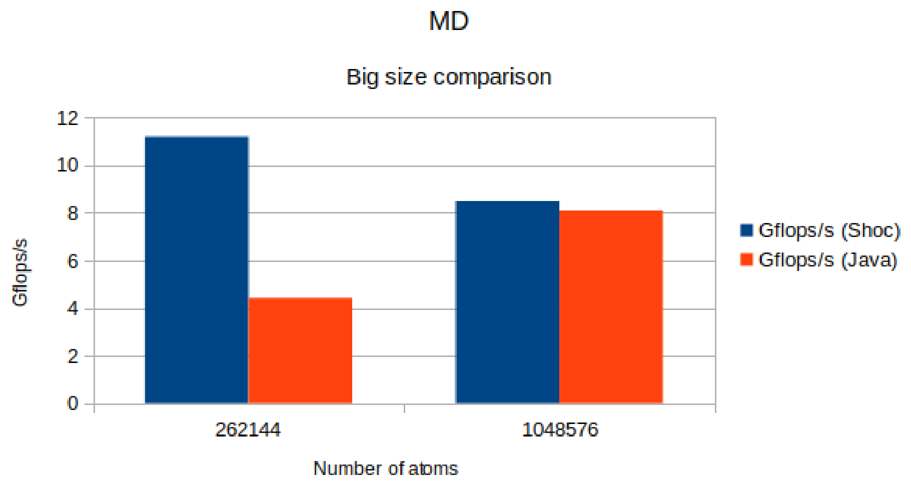

Figure 32: Time of kernel execution of the Lennard-Jones algorithm with 128 neighbours in both versions.

This further extends the trend of the experiments made so far. In small size data sets the overhead that Aparapi has in comparison to the C++/OpenCL exacerbates the performance results.

4.5 SPARSE MATRIX-VECTOR MULTIPLICATION OPTIMIZATIONS RESULTS

As described before, two different algorithms for sparse matrix-vector multiplication were developed mimicking the Parboil benchmark as well as the techniques from Bell and Garland (2009). For each of these algorithms two different versions were made to study the impact of padding.

With the experiment of figure 33, the impact of using a padding technique in a sparse matrix-vector multiplication is exposed. In both test (scalar and vector) padding brought an improvement of .5 to 1 GFlop/s.

The impact of padding in Aparapi as similar results as padding in the Parboil benchmark as shown in figure 34.

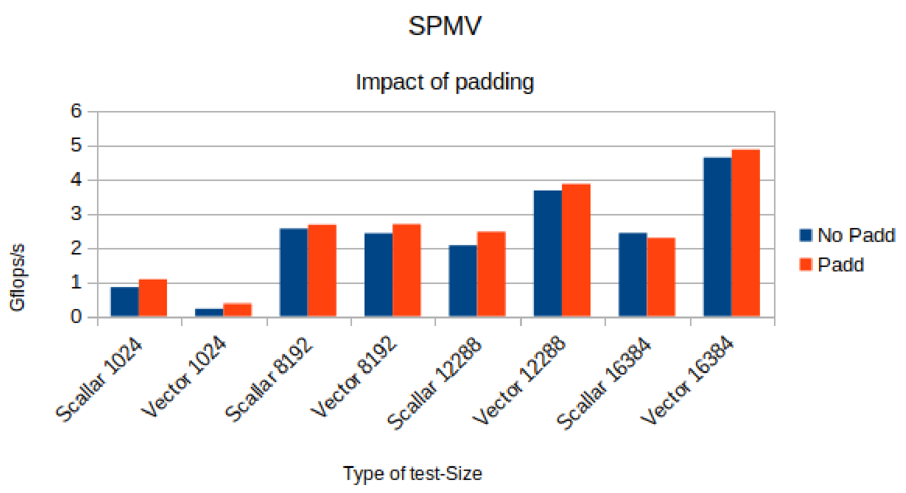

Figure 33: Impact of padding in both versions of the Java SPMV algorithm.

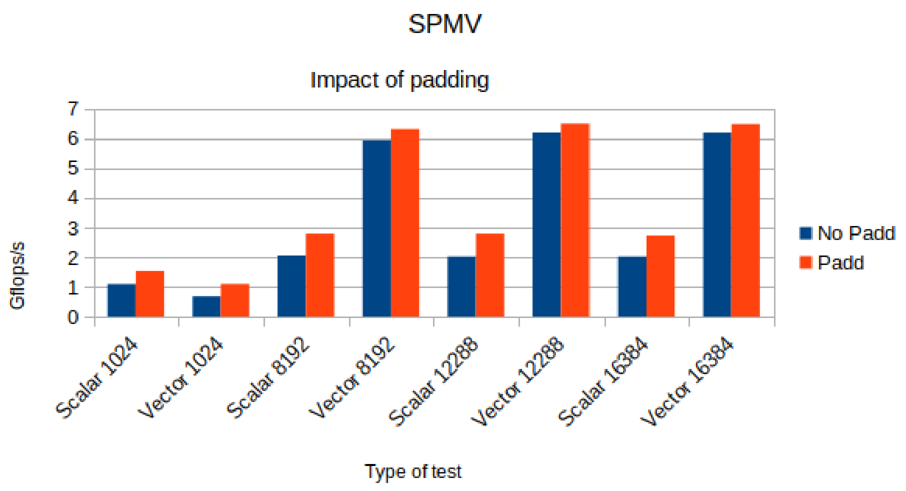

Figure 34: Impact of padding in the Parboil benchmark SPMV.

4.5.1 *Parboil comparison*

The Aparapi version developed follows the same pattern observed so far. In the case of 35, with all of the optimizations implemented so that the generated OpenCL mimics the benchmark, the scalar versions over perform their counter parts with the benchmark having better results overall.

When the number of rows of the matrix reaches 8192, the size is enough for the scalar version of Aparapi to out perform the one of the benchmark. However, the vector Aparapi version has not enough items to compute to compensate its overhead thus achieving poor results.

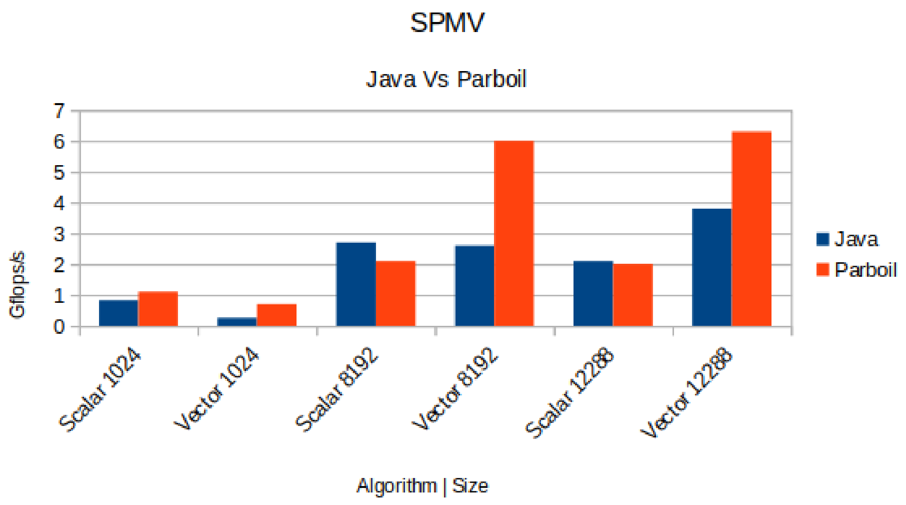

Figure 35

In figure 35 both scalar versions seem to have peaked as well as the vector implementation of the benchmark while the java vector version now has enough items to exceed the scalar ones.

4.6 HISTOGRAM OPTIMIZATIONS RESULTS

The histogram algorithm was developed according the suggestion present in the Parboil benchmark suite. These include the use of work-groups of threads which compute over an assigned block of data and exploit shared memory to store it.

The algorithm was developed to avoid shared memory bank conflicts and reduces the number of multiple kernels. The results in figure (36) Show that for a small size kernel the overhead of having to generate multiple kernels is relevant. The total time taken for Aparapi to generate the OpenCL code (2x 500ms) corresponds to half of the total time needed for the execution of the entire program.

When the input size rises, the time taken to generate the different kernels gets less relevance as the execution time increases and the first remains constant.

4.6.1 *Parboil comparison*

When comparing the default size input in both tools, Parboil has better results when compared to Aparapi. The main reason is because Aparapi has to generate two different OpenCL kernels when Parboil doesn't have this burden. This results in a overhead that is quite significant in the overall results.

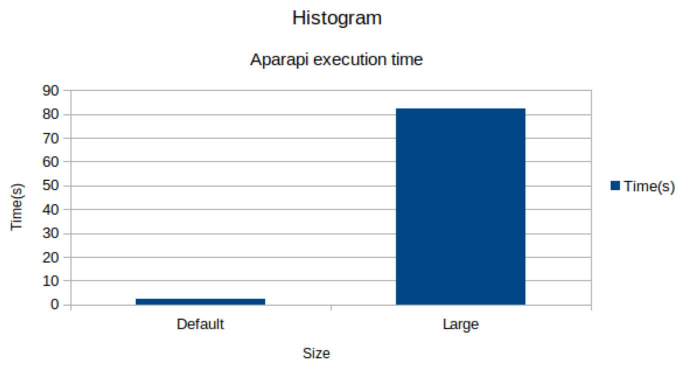

Figure 36: Kernel execution time of Aparapi with the inputs of the Parboil histogram.

The Parboil benchmark suite version of the histogram is also better with large input but the difference is not as much accentuated has in the small inputs.

Another reason why Aparapi can't outperform or have better results then the Parboil version is because the atomic operations where not replicated in Aparapi.

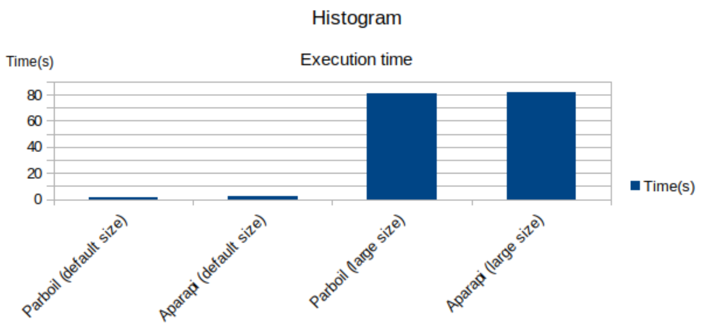

Figure 37: Execution time of the histogram algorithm in Aparapi and Parboil benchmark suite.

4.7 SUMMARY

After all the algorithms from the selected benchmarks has been run the gathered data about their execution has permitted to achieve the objectives that were proposed to study.

With the developed Aparapi kernels, the impact of changing the memory layout was found to be one of the most important optimizations that can be made in Aparapi especially if one is using Java Objects to be executed on GPU. This optimization can be easily implemented by deconstructing the desirable objects into a structure of arrays layout and the time needed for this task can be considered redundant both in a heavy or light workload computation kernel.

To try to minimize the impact of generating the OpenCL code in run-time Aparapi caches the kernels through an execution. This is particularly important to note when dealing with small sized kernels because the results will be heavily affected.

Through the N-Body algorithm, the different data layouts presented in this master dissertation were implemented and their results exposed and compared.

The effects of coalescing data in Aparapi was directly observed in the results of the stencil-2D improving the performance significantly which were then further improved by taking advantage of the shared memory low latency capabilities. Finally, by applying a coarsening technique, the best version of Aparapi stencil-2D was achieved.

The MD version due to its small size kernel was not the best choice to use to compare to Aparapi. This is because the time taken for execution is too small to compensate for the run-time generation of OpenCL code. Some modifications to the size of the kernel were experimented with but, since the time taken by the CPU stage rises exponentially with the size of the kernel further investigation of optimizations for the algorithm were abandoned.

By studying and implementing an Aparapi version of the SPMV algorithm of the parboil benchmark suite the results of padding in a scalar and vector version were analysed.

Finally, the different optimized algorithms of Aparapi results were compared with their counterparts from the benchmark suits. With this comparison, the final objective of this document could be achieved. Aparapi through all the tests cannot compete with C++/OpenCL principally because of its necessity of managing and constructing the code to be generated for the execution.

5 CONCLUSION AND FUTURE WORK

In this last chapter, conclusions are drawn from the results gathered through the various sections of the development and experiment chapters. The final section of this master degree dissertation is related to future work that can be done based on the current conclusions.

5.1 CONCLUSION

In this master degree dissertation, Aparapi a tool that permits the execution of developed Java software in GPU was studied in detail and several benchmarks and experiences were made to extract the data needed to fulfill the proposed objectives of this project.

The performance impact of run-time OpenCL generation was tested in all the experiments made. Since the time needed for the generation remains constant independently of the size of the kernel, it has a great impact when executing for the first time a small kernel. By caching a kernel, Aparapi greatly reduces the executing time if it is several times needed throughout the program execution.

It is important to mention that if the software has more than one kernel and if these are intertwined Aparapi performance will deteriorate.

Regarding the **analysis of the execution of Java objects in GPU by Aparapi** some simple experiments where Aparapi could work correctly with Java objects has proved that the use of pointers in the GPU is responsible for degrading performance results. Java, however, can rapidly transform these objects into better data structures to be executed. This process also has the inconvenience of having to be manually done and deprives the developer of main features that Java has to offer.

Aparapi can easily deal with **data communication costs between CPU and GPU** through the methods available by it. The software developer can choose to manually handle the data transfers or automatically optimize the process by employing the steps argument in the execute method.

The **study of different data layouts transformations** and the generated OpenCL of Aparapi through the experiments made confirmed that Aparapi follows the same pattern of favouring a structure of arrays type of data layout over the rest as other tools that enable GPU execution. By opting for an SoA structure layout, the developer can see a speedup up to 7x when compared to other data structures. Once more, Aparapi OpenCL generator does not have the capabilities of generating SoA from any type of data structure that the developer provides.

Following the implementation of the **coarsening** technique in the stencil experiment the software performance jumped in some situations up to 1.5x. The coarsening results gathered from Aparapi have similar behaviours to coarsening in other software solutions since the conversion of the OpenCL code for coarsening is straightforward and it only depends on the developer.

By employing **tilling** and exploiting the shared memory of the GPU the stencil-2D performance results saw a speedup of 3.5x. Again, Aparapi can easily generate OpenCL code automatically similar to the source code of the benchmark used for comparison. Thus, both the tilling technique as well as the code that enables shared memory usage to produce results on par with other solutions once the size of the kernel is enough to subdue the overhead of Aparapi.

Throughout the experiments made with stencil-2D, different ways of organizing the data in memory were implemented. By achieving a **coalesced** pattern of data in memory, the program experienced a speedup of 5x times in comparison with data layouts that were not aligned with the cache lines. These results are expected since a coalesced memory access was often pointed through the state of the art chapter as one of the most important optimizations that could be done for a GPU program to perform as expected.

The impact of **padding** in the SPMV and in the stencil-2D experiment didn't had as much influence in Aparapi has they had in experiments with the SHOC version of the same algorithms only improving the performance by 0.5 to 1 GFlop/s.

In the histogram experiment, **data privatization** was implemented to handle the high degree of contention present in global memory.

Finally, **Aparapi in comparison with similar software solutions** has its advantages and disadvantages. Aparapi is capable of generating reliable and optimized OpenCL code from optimized Java. Most of the Aparapi generated code mimicked in perfection the algorithms of the benchmark suites. The verbosity of OpenCL was completely abstracted by Aparapi only damaging the performance results of the first execution.

5.2 FUTURE WORK

Aparapi current way of handling Java objects is still primitive having restrictions in the way objects can be used and presents some difficulties tracking down the object hierarchy and correspondent classes which makes it not being able to generate code in more complex case studies.

The current generation technique could also be improved by detecting automatically the objects while reading the Bytecode and convert it to an SoA layout to be used on execution. This way Aparapi can truly be a software tool that enables fast deployment and optimized OpenCL with minimum effort giving the developer a fast learning curve.

BIBLIOGRAPHY

C. McCurdy J. Meredith P. Roth K.Spafford V. Tipparaju A. Danalis, G. Marin and J.Vetter. The scalable heterogeneous computing (shoc) benchmark suite, 2010.

A. Fatayer A. Almousa A. Baqais A. Khan, M. Mouhamed and M. Assayony. Padding free bank conflict resolution for cuda-based matrix transpose algorithm., 2014.

S. John B. Calder, C. Krintz and T. Austin. *Cache-conscious data placement*. Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, 1998.

C. Bailey. From java code to java heap.

N. Bell and M. Garland. Implementing sparse matrix-vector multiplicationon throughput-oriented processors, 2009.

R. Bryant and D. O'Hallaron. *Computer Systems A programmer's perspective*. Pearson education, 2002.

C. Donglin X. Chuanfu C. Shizhao, F. Jianbin and W. Zheng. Optimizing sparse matrix-vector multiplication on emerging many-core architectures, 2009.

A. Cedillo, R. Zárate, C. Lozada, and M. Carbajal. *GPU Programing paradigm*. Journal of Theoretical and Applied Information Technology.

R. Farber. *Cuda Application design and development*. Morgan Kaufman, 2011.

N. Faria, R. Silva, and J. Sobral. *Impact of Data Structure Layout on Performance*. 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2013.

G. Frost. Aparapi: An open source tool for extending the java™ promise of 'write once run anywhere' to include the gpu.

S. Fuller and L. Millett. The future of computing performance: Game over or next level?, 2011.

Khronos OpenCL Working Group. The opencl specification, 2012.

M. Hirzel. *Data Layouts for Object-Oriented Programs*. SIGMETRICS, 2007.

S. Hollingsworth and R. Dror. Molecular dynamics simulation for all., 2018.

K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. Compiling and optimizing java 8 programs for gpu execution.

M. Huang J. Cong and Y. Zou. Accelerating fluid registration algorithm on multi-fpga platforms., 2011.

L.Pouchet J. Holewinski and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures.

J. Benevides J. Luna, J. Linares and N.Guil. An optimized approach to histogram computation on gpu, 2012.

I.Sung N. Obeid L.Chang N. Anssari G.Liu J. Stratton, C. Rodrigues and W. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing, 2012.

S. Joshi. *Leveraging Aparapi to Help Improve Financial Java Application Performance*. 2011.

B.Cosenza K. Kofler and T. Fabringer. *Automatic Data Layout Optimizations for GPUs*.

M. Karplus and J. McCammon. Molecular dynamics simulations of biomolecules., 2002.

J. Lewis and U. Neumann. Performance of java versus c++, 2003.

A. Magni and C. Dubach. A large-scale cross-architecture evaluation of thread-coarsening, 2013.

M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming*. Morgan Kaufman, 2012.

J. Meng and K. Skadron. A performance study for iterative stencil loops on gpus with ghost zone optimizations., 2011.

Nvidia. Cuda c best practices guide, 2012.

Nvidia. Cuda toolkit documentation, 2021.

M. Orlov. Evolving software building blocks with finch, 2017.

D. Patterson and J. Hennessy. *Computer organization and design*. Morgan Kaufman, fifth edition, 2014.

F. Peper. The end of moore's law: Opportunities for natural computing?, 2017.

A. Pereira. High performance computing with cuda.

V. Podlozhnyuk. Histogram calculation in cuda, 2013.

B. Rajaraman. Implementation and evaluation of register tiling for perfectly nested loops., 2009.

G. Rivera and C. Tseng. Data transformations for eliminating conflict misses, 1998.

F. D. Igual R. Mayo S. Barrachina, M. Castillo and E. Quintana-Ortí. Solving dense linear systems on graphicsprocessors., 2008.

V. Gurusamy S. Kannan and G.Nalini. Review on image segmentation techniques, 2014.

U. Bondhugula J. Ramanujam A. Rountev S. Krishnamoorthy, M. Baskaran and P. Sadayappan. Effective automatic parallelization of stencil computations.

G. Smith. Numerical solution of partial differential equations: Finite difference methods., 2004.

Y. Solihin. Fundamentals of parallel multicore architecture., 2015.

N. Stawinoga. Predictable thread coarsening, 2018.

M. Takeuchi T. Yasue M. Kawahito K. Ishizaki H. Komatsu T. Suganuma, T. Ogasawaram and T. Nakatani. Overview of the ibmjava just-in-timecompiler, 2000.

A. Taflove. Computational electrodynamics: The finite-difference time-domain method., 1995.

V. Volkov and J. W. Demme. Benchmarking gpus to tunedense linear algebra, 2008.

Y. Lin W. Zaremba and V. Grover. Jabee - framework for object-oriented java bytecode compilation and execution on graphics processor units, 2012.

Linux World. http://tuxthink.blogspot.com/2010/06/memory-hierarchy.html.

A
APPENDIX

Manufacturer	Intel Corporation
Arquitecture	Haswell
Model	Core i7-4702MQ
# Cores	4
# Threads	8
Processor frequecy	2.2 GHz
Extensions	Intel SSE4.1, Intel SSE4.2, Intel AVX2
L1 Cache	256 KiB
L2 Cache	1 MiB
L3 Cache	6 MiB
Associativity	8/8/12
RAM memory	2 * 8GiB DDR3L
RAM latency	11 clocks
Memory bandwidth peak	12625.1MiB/s
#Memory channels	2
GPU manufacturer	NVIDIA
GPU arquitecture	Maxwell
GPU model	GTX 850M
CPU-GPU link	PCIe 3.0 16 lanes
GPU global memory size	2 GiB
GPU global memory cache line size	128 Bytes
GPU local memory size	48 KiB
GPU L1 Cache	64 KiB
GPU L2 Cache	2 MiB
GPU #threads	640 cuda cores
GPU memory bandwidth peak	32.03 GiB/s
GPU FP performance peak	1155 GFlops/s
GPU max work item sizes	1024x1024x64
GPU max work group size	1024
Warp size	32
OS	Ubuntu 18.04.4LTS
Java	v11.0.11
OpenCL	v1.2
Aparapi	v1.10.0

Table 3: Laptop Hardware and Software Characterisation.