FROM OPENMP TO TORNADOVM: BENCHMARKING JAVA ON HETEROGENEOUS ARCHITECTURES

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2024

Student id: 10813496

Department of Computer Science

Contents

Word Count: 10364

List of Tables

List of Figures

Abstract

FROM OPENMP TO TORNADOVM: BENCHMARKING JAVA ON HETEROGENEOUS ARCHITECTURES
Gaoyang Li
A report submitted to The University of Manchester
for the degree of Bachelor of Science, 2024

This report investigates the performance of TornadoVM by porting the Rodinia benchmark into Java and particularly to the TornadoVM API. The primaty object of this work is the transition from OpenMP, one of programming models used in the Rodinia benchmark, to TornadoVM. The main achievement of this project is to explore the performance and scalability of TornadoVM by using tests with large data to test, and also the comparsion with OpenMP and common Java which is without the use of TornadoVM.

Declaration

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to give thanks to my supervisor, Prof. Christos Kotselidis, for his continuous advice and support during this project. I would also like to thank Thanos Stratikopoulos and Juan Fumero, research fellows at the University of Manchester, for offering important insights about this project and patience to answer my questions. I would like to thank Zheyu Liu, a PhD student of the University of Manchester, for helping me in the lab.Lastly, I would like to thank my family for their help and support so far during the university years.

Chapter 1

Introduction

The advent of heterogeneous hardware accelerators, such as Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), etc., has marked the evolution in software development from single-threaded to highly performant parallel implementations [17]. Then, the evolution of computing systems from single-core to multicore systems has resulted in running programs more efficiently and faster when programming on heterogeneous hardware and in parallel [12]. Parallel programming is different from traditional sequential programming, where multiple tasks of a program or several parts of a single operation or function without data dependency run simultaneously rather than one by one; hence, parallel programming can boost performance [21]. The large variety of hardware accelerators (e.g., GPUs, FPGAs) are used as co-processors to the main CPU to optimize the execution of a program based on their characteristics. CPUs are efficient in dealing with tasks requiring low latency, while GPUs are used for high throughput data processing, and FPGAs lower the latency of the execution of complex algorithms because of their great flexibility and parallel processing capabilities [23]. Programming for heterogeneous hardware enables software applications to run on the hardware unit that is more suitable, hence improving execution efficiency more [31]. Nowadays, more and more programming languages provide support for parallel programming via Application Programming Interfaces (APIs). In parallel programming, programmers using C, C++, and Fortran are allowed to use the Pthread API, while Java programmers can use the Concurrency API. However, these APIs do not support programs utilizing the advantages of heterogeneous hardware, and they also need programmers to manage threads, synchronization, and data sharing manually. The appearance of OpenMP [20], which is an API and a set of compiler directives used in C, C++, and Fortran, and TornadoVM [6], which is a plugin to OpenJDK used in Java, not only simplifies the steps of multi-platform parallel programming but also enables programmers to offload the execution of their programs. OpenMP 4.0 [20] allows programmers to use directives to achieve programming on heterogeneous hardware [14]. Similarly, in Java, TornadoVM [6] was created to offer hardware acceleration for Java programs [7].

1.1 Aims & Objectives

This project aims to find and compare the execution efficiency between OpenMP, TornadoVM [6] and common sequential Java. The focus of this research is to use Java with TornadoVM [6] to implement the algorithms that were implemented in the OpenMP part of the Rodinia benchmark. As a result, this project has the following objectives:

- Explore the basic and advanced usage of OpenMP and TornadoVM.
- Analyse the algorithms in the Rodinia benchmark, especially the part implemented by the features of OpenMP.
- Use common Java code to implement the algorithms without the use of TornadoVM. This will be considered successful when the results are the same as that using OpenMP with various datasets.
- Modify the Java code so that it can run with TornadoVM. This will be considered successful when the results remain the same.
- Find bugs in TornadoVM and provide minimal test cases to the TornadoVM team.
- Analyse the performance difference between the OpenMP Rodinia algorithms and the functionality equivalent algorithms implemented with TornadoVM, as well with common sequential Java code. This will be achieved by finding out the difference in the execution speed across numerous datasets.

1.2 Structure

Chapter 2 presents a comprehensive overview of the background information regarding the OpenMP, TornadoVM [6], and Rodinia benchmark. It explores topics such as CUDA, OpenCL, OpenMP, and an overview of the implemented algorithms. Detailed processes involved in implementing the algorithms are discussed in Chapter 3, followed by a performance comparison of OpenMP and TornadoVM according to the implemented algorithms in Chapter 4. Finally, Chapter 5 includes the conclusion, personal reflections, and future plans.

Chapter 2

Background and Challenges

2.1 OpenMP

OpenMP offers programmers an interface for iterative data processing that can be executed simultaneously, therefore it is useful for large data sets that exhibit no data dependency among the iterations [2]. Figure 2.1 shows a simple parallelized loop example using OpenMP [20], where multiple threads are spawned to print a message at the same time. In this example, each thread corresponds to a loop index. However,

```c
#include <stdio.h>
#include <omp.h>
int main() {
    int i;
    #pragma omp parallel for
    for (i = 0; i < 10000; i++) {
        printf("%d", i);
    }
    return 0;
}
```

Figure 2.1: A C language snippet to demonstrate the simple use of OpenMP

if threads manage many private variables, (variable "i" in Figure 2.1) is not efficient enough due to the large size of copies. OpenMP also provides a way for some variables to be shared among all threads. As an example, consider the C code shown in Figure 2.2. In order to implement the function "summation" which sums all elements stored in "array". In this case, the variable "i" is necessarily private, since each thread is allocated to execute different iterations of the loop, "array" can be shared, as every thread needs to access the same array. The keyword "reduction" for the "sum"

```c
#include <stdio.h>
#include <omp.h>
int summation(int array[], int size){
    int i;
    int sum = 0;
    #pragma omp parallel for private (i) shared (array) reduction
   (+:sum)
    for (i = 0; i < size; i++){
        sum = sum + array[i];
    }
    return sum;
}
```

Figure 2.2: A more practical example of OpenMP

is also important. During the execution, each thread has its own value of "sum", so "reduction+" is used to sum all threads' copies called "sum". Finally, "+" indicates the addition operation is done for the reduction copies. The release of OpenMP 4.0 makes heterogeneous computing available due to effective management of memory across different computing devices, which is achieved by reconfiguring the compatibility between the OpenMP [20] runtime and OpenSHMEM [13]. As shown in Figure 2.3, in order to sum elements of two arrays into an array, "map(to: a[:size], b[:size])" means transferring "a" and "b" to the target device, such as the GPU. Similarly, "map(from: c[:size])" copies back from the device to the host (CPU) after the computation completes.

```c
#include <stdio.h>
#include <omp.h>
void arraySummation(int a[], int b[], int c[], int size){
    #pragma omp target
    map(to: a[:size], b[:size]) map(from: c[:size])
    #pragma omp parallel for
    for (int i = 0; i < size; i++){
        c[i] = a[i] + b[i];
    }
}
```

Figure 2.3: Use of heterogeneous computing by OpenMP

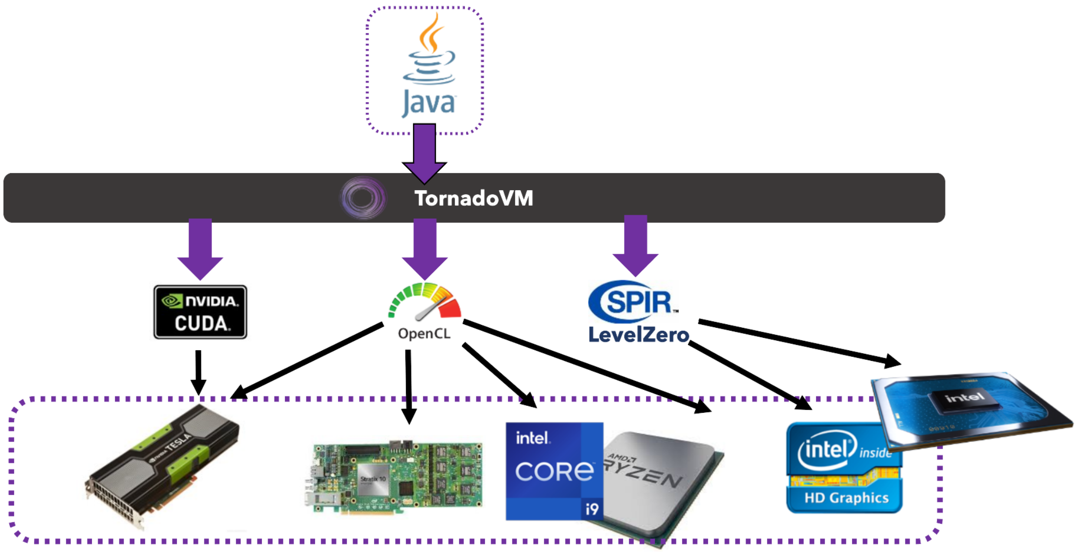

Figure 2.4: Architectural overview of TornadoVM [30]

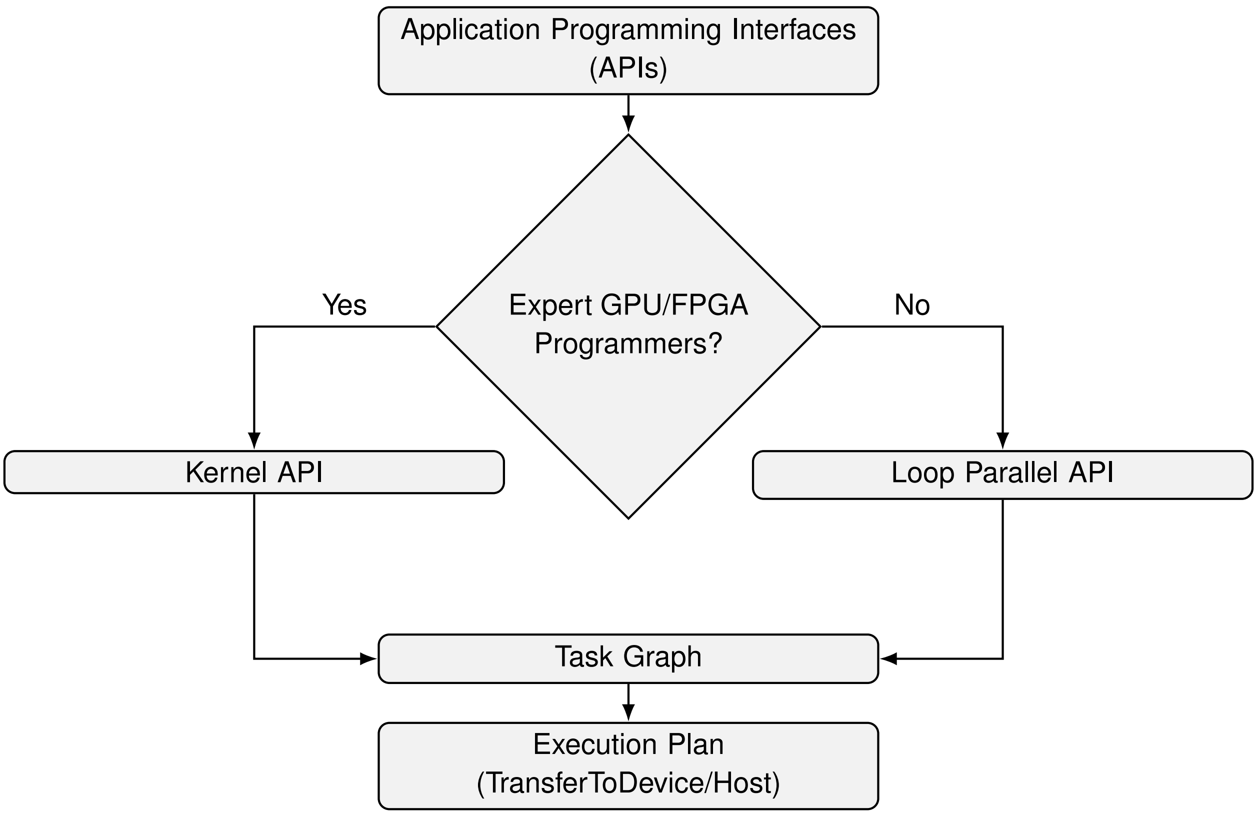
Application Programming Interfaces (APIs)
Expert GPU/FPGA Programmers?
Yes
No
Kernel API
Loop Parallel API
Task Graph
Execution Plan (TransferToDevice/Host)

Figure 2.5: Flowchart illustrating API selection in TornadoVM

2.2 TornadoVM

As shown in Figure 2.4, TornadoVM has a compiler with three backends, which can generate OpenCL C code, PTX which are assembly instructions for the NVIDIA GPU model, and SPIR-V modules which are instructions in binary format [27]. This approach enables TornadoVM to dynamically convert Java bytecode into three different forms of parallel implementations which are OpenCL, PTX and SPIR-V, leading to improved performance and a modular backend architecture. As illustrated in Figure 2.5, TornadoVM provides two APIs: one is the Loop Parallel API, and the other is the Kernel API. The Loop Parallel API is a high-level API that enables Java programmers to use annotations similar to the OpenMP "#pragma omp parallel for" directive in order to mark that a for loop can run in parallel. Additionally, TornadoVM exposes the Kernel API which enables programmers to express low-level features, such as allocation of local memory, barriers for data synchronization, etc. This API is used by more experienced programmers and can yield higher performance due to the explicit declaration of those optimizations [6, 28]. Through the use of the Parallel annotation on for loops within Java source code, TornadoVM enables programmers just need to annotate a loop for parallel executions and give a hint to the compiler to generate the corresponding parallel code. These annotations are preserved within the Java Bytecode, allowing TornadoVM to parallelize the designated loops across various hardware types, including CPUs, GPUs, and FPGAs, during runtime in the Java Virtual Machine [26]. This feature improves Java programs by adding a way to indicate parallelization potential directly at the loop level, which eliminates the need to create separate parallelization back-ends for different hardware architectures. In addition, TornadoVM offers the TaskGraph API so that once methods are customized by the Parallel or Kernel API, they are organized in a graph. Developers can use an accelerator to accelerate all methods specified in the task graphs. Furthermore, developers can modify how the execution will take place (e.g., with the profiler backend, or for a specific target device, etc.) by creating an execution plan in TornadoVM. Figure 2.6 shows an example that implements the summation of two arrays, as discussed earlier in the OpenMP example. In this example, the Loop Parallel API is used, and the parallel annotation (line 3) is added to a sequential loop to indicate that the loop can be parallelized because of no data dependency. The TaskGraph is defined in lines 10–17 and contains several methods. The two methods, called "transferToDevice" and "transferToHost", stand for data from the host to the device and from the device to the host individually. First of all, programmers can get the default hardware device available for executing parallel

```
1  public static void arraySummation(
2  int[] a, int[] b, int[] c, int size){
3      for (@Parallel int i = 0; i < size; i++) {
4          c[i] = a[i] + b[i];
5      }
6  }
7  public static void main(String[] args){
8      TornadoDevice device = TornadoRuntime.getTornadoRuntime()
9          .getDefaultDevice();
10     TaskGraph taskGraph1 = new TaskGraph(name: "s1")
11         .transferToDevice(DataTransferMode.FIRST_EXECUTION, a, b,
     size)
12         .task("t1", Draft::arraySummation, a, b, c, size)
13         .transferToHost(DataTransferMode.FIRST_EXECUTION, c);
14     ImmutableTaskGraph immutableTaskGraph1 = taskGraph1.snapshot();
15     TornadoExecutionPlan executor1 = new
16     TornadoExecutionPlan(immutableTaskGraph1).withDevice(device);
17     executor1.execute();
18 }
```

Figure 2.6: Use of heterogeneous computing by TornadoVM

computations (line 8-9). When creating a task graph, two options are provided to select the data transfer mode: one is "FIRST_EXECUTION" (line 11 and 13), which means the transferred buffers are read-only, and the other is "EVERY_EXECUTION", which means data must be transferred in each execution. In order to ensure consistency with hardware, an immutable task graph (line 14) is created so that the transferred data remains the same while the host and device are running. At last, an execution plan (line 15-16) is created from the immutable task graph, and then it can be executed with devices (line 17). In conclusion, using TornadoVM through the Parallel API to define and execute a parallel method consists of three steps: First, use the Parallel annotation to show which for loop in a method can be parallelized. Then, define the hardware resources and create the mutable and immutable task graphs. Lastly, start the execution of the parallelized implementation of the methods on the accelerator. Figure 2.7 indicates the overview of API of TornadoVM in general.

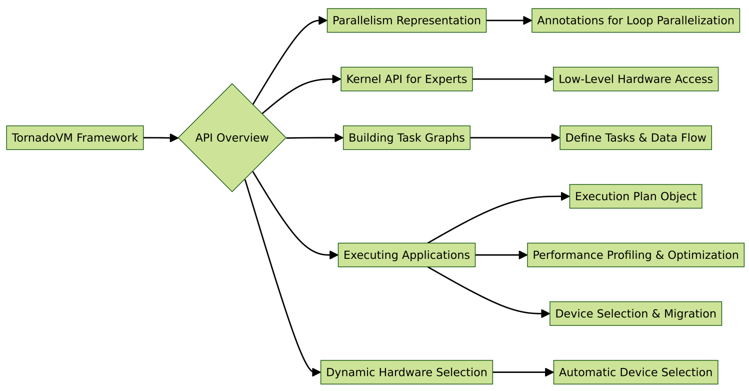
TornadoVM Framework
API Overview
Parallelism Representation
Annotations for Loop Parallelization
Kernel API for Experts
Low-Level Hardware Access
Building Task Graphs
Define Tasks & Data Flow
Execution Plan Object
Executing Applications
Performance Profiling & Optimization
Device Selection & Migration
Dynamic Hardware Selection
Automatic Device Selection

Figure 2.7: General overview of API functionality of TornadoVM

2.3 Rodinia Benchmark

Rodinia is a benchmark suite designed to support research in the area of heterogeneous computing, centred around both multicore CPUs and GPU platforms. It consists of applications and kernels for diverse studies and supports research work on emerging architectures of computation like those involving GPUs and platforms based on CUDA, OpenCL, and OpenMP [4]. The characterization of the suite suggests that such tools could be used to extract essential knowledge about many heterogeneous computing challenges, as well as opportunities [5].

The benchmarks are designed for heterogeneous systems, combining multicore CPUs and GPUs, so this aligns well with TornadoVM's goal of facilitating acceleration of Java program on heterogeneous hardware. However, it also causes challenges in data distribution, synchronization, and workload balancing across different types of hardware units. In addition, Rodinia benchmarks are derived from real-world applications in areas such as computer vision, bioinformatics, data mining, and physics simulation. This relevance ensures that the performance improvements and challenges that occur when porting these benchmarks to TornadoVM reflect practical use cases, providing valuable insights into TornadoVM's applicability in research.

2.3.1 CUDA & PTX

CUDA (Compute Unified Device Architecture) is a parallel and heterogeneous programming platform furnished by NVIDIA due to the power of NVIDIA GPUs [24]. NVIDIA PTX (Parallel Thread Execution) is an instruction set architecture for NVIDIA GPUs consisting of kernel and function declarations, and when a CUDA program is compiled, it is first translated to PTX, which is then further compiled to binary code that is specific to the target GPU architecture [15]. Due to the advent of Unified Memory Access (UMA) that comes with CUDA, nowadays CPUs can access the data residing within a shared memory address space [19]. However, it was found that UMA can achieve better performance than the traditional approach to data transfer in certain applications, such as when transmitting the smallest sizes of data. In the great majority of applications, however, it introduces large performance overhead and limits potential future optimizations since the programming model is simplified [11]. TornadoVM can target NVIDIA GPUs by compiling Java methods to PTX code if an NVIDIA GPU exists.

2.3.2 OpenCL

Similarly to CUDA, OpenCL is also a heterogeneous parallel processing platform [18]. Unlike CUDA, which supports parallelization only for NVIDIA GPUs, OpenCL offers parallelization patterns for both multicore CPUs and many-core accelerated devices [16]. OpenCL defines a platform model that includes a host connected to one or more computing devices, which can be CPUs, GPUs, or other types of processors. Each computing device is further divided into compute units (CUs), and each CU is into processing elements (PEs) [25]. TornadoVM utilizes OpenCL as one of its backends to execute parallel workloads. By abstracting the underlying hardware, TornadoVM allows Java applications to be executed on different devices (CPUs, GPUs, FPGAs) without needing to rewrite the code for each device, and then compiles Java bytecode to OpenCL, enabling the execution of Java programs on hardware that supports OpenCL.

2.3.3 SPIR-V

SPIR-V (Standard Portable Intermediate Representation-V) is an intermediate binary format used for parallel computing and graphical representation [10]. SPIR-V makes parallel processing on heterogeneous hardware—like GPUs and FPGAs—possible. It is an OpenCL extension that supports a broad range of vendors and extensions, enables applications to use binary modules rather than OpenCL C programmes. By allowing high-level language front-ends to generate programmes in a common intermediate format that can be used by OpenGL, Vulkan, or OpenCL drivers, it streamlines driver architecture and fosters the development of a community of tools for analysis, porting, and optimization [8]. By supporting SPIR-V, the flexibility and portability are greatly improved across different hardware devices. Since SPIR-V is designed to be used with both compute (like OpenCL) and graphics (like Vulkan) APIs, TornadoVM is able to target more hardware platforms and capabilities compared to using only OpenCL-supported devices.

2.3.4 OpenMP

Through detailed benchmarking, the results of the study [9] show that OpenMP performance behaviours diversify across different applications, platforms, and dataset sizes; generally, it scales well, with the best performance being observed around the number of hardware cores of the hardware platforms. The current mechanisms of OpenMP for coordinating dependencies across nesting levels can restrict the exploitation of parallelism, which is due to each task defining an isolated domain of dependencies for its direct sub-tasks, which limits the ability to exploit parallelism fully [22].

2.4 Algorithms

The following sections will describe each of the algorithms from the Rodinia benchmark suite that have been examined in this project to develop parallel implementation and access their performance.

2.4.1 Breadth First Search

The Breadth-First Search (BFS) is a classic algorithm in terms of node-level parallelism in graphs and a key point of reference. BFS saturates multicore and many-core platforms like CPUs and GPUs while maintaining peak performance, because it concurrently handles multi-branch graph walks. This is the main reason why the parallel exploration of nodes is an excellent test for verifying performance on a computing system because it shows how excellently distributed and synchronized by parallel frameworks an algorithm is.

2.4.2 K-means

The K-means algorithm is a clustering method that groups similar features concerning certain characteristics in a dataset and tries to obtain optimal cluster centroids to minimize the variance within the group. Therefore, it is computationally expensive since the algorithm is designed to update centroids, after one assignment of data points to their nearest centroid, several consecutive iterations of it will have to be performed, especially on large datasets. However, the independent tasks are well-defined, and the iterative approach of k-means has shown it to be an excellent candidate for parallel and heterogeneous programming. It supports parallelism and considerably reduces the computation time, since every data point and each centroid calculation are distributed over a portion of different processing units.

2.4.3 Particlefilter

The Particlefilter algorithm is both a method for prediction and a model for representation. It can simulate sets of particles, which support parallel computation, so it can be used in order to test new parallel and heterogeneous programming environments. A task or particle is updated independently about the states at each step, making it possible to distribute a lot of particles simultaneously among many processing units, thus contributing to this algorithm's efficiency. This property is important for real-time applications that call for fast state estimates.

2.4.4 Pathfinder

The Pathfinder algorithm is a solution to the problem where one wants to find the cheapest path through a grid where every cell corresponds to a cost. The movement can only be directly ahead or diagonally adjacent, but it can only change by one increment. Therefore, this algorithm finds direct application in actual problems as well as theoretical concepts that exist within computer science, routing, and network flow. And because of these reasons, this algorithm is significant for the project. The first reason is obvious: it is the capability to support parallel processing, since any outcome of a current step of computation in the algorithm depends solely on the provisions of the result made by some previous step. This provides for a way through which each position in a new row can find the minimum cost to consider the cost of the previous row individually. Therefore, testing parallel and heterogeneous programming paradigms is suitable for the implementation of the Pathfinder algorithm. This algorithm distributes computation over many processing units, highlighting the efficiency and scalability potential of parallel computing architectures in reducing execution times significantly.

2.4.5 Speckle Reducing Anisotropic Diffusion

The Speckle Reducing Anisotropic Diffusion (SRAD) algorithm is used to reduce the noises of images and most typically used for speckle noise in ultrasound and radar images. It does that by applying anisotropic diffusion so that it has the capability of tuning how strong the diffusion is across an image, preserving important features of the image like edges and textures while still suppressing noise. This produces an increase in quality for the whole image, thus facilitating the inspection or interpretation of the same. One important characteristic of SRAD is that updating is based on the immediate surroundings of each pixel, while the pixel is made to remain independent of the general image. This becomes ideal for parallel computation, as the update process could be further divided among several processors. This gives SRAD flexibility to operate on any computing architecture, from multicore CPUs even up to faster GPUs that can afford a large reduction in processing time. The probable parallelization of the diffusion operation makes SRAD not only very useful under the real-time imaging scenario, but also greatly serves for evaluating and enhancing the performance of such parallel computing systems. This contributes to both the field of image processing and the broader field of high-performance computing.

2.4.6 Lower-Upper Decomposition

The Lower-Upper Decomposition (LUD) algorithm for a matrix is one of the simple numerical techniques. The conceptual background behind this method is that it makes linear equations' computation easy and the computation of matrix inversions and calculations related to determinants. The structure of LUD allows independent calculation of sub-matrices; hence, it is highly suited for parallelization. This prerequisite is provided, which gives excellent provision for allowing LUD to effectively use parallel and various computing environments, which allows for a great reduction in time used for large-scale problems. This is how LUD serves as a standard for testing power and performance for parallel systems. The distribution of workload amongst the many processors shows that LUD reflects the capability for potential parallelism of the numerical algorithms. Apart from this, LUD also highlights the need and essence of parallel computing techniques to work out difficult, complex mathematical problems with speed. As such, the LUD algorithm is applicable not only in numerical analysis but also in computational mathematics.

2.4.7 Needleman-Wunsch

The Needleman-Wunsch (Nw) algorithm is a dynamic programming algorithm used for global sequence alignment, i.e., aligning two sequences (e.g., DNA, RNA, and protein sequences) to identify regions of similarity that may indicate functional, structural, or evolutionary relationships between them. This algorithm fills in a matrix, in which each cell represents the highest score it is possible to achieve at a position, based on the scores of neighbouring cells and a specified scoring scheme comprising match/mismatch scores and gap penalties. In parallel heterogeneous computing, the significance of this algorithm lies in its ability to make effective use of the combined computational power of CPUs and GPUs to allow for more efficient processing of the large matrices typical of sequence alignment tasks. By dividing the algorithm into smaller, independent tasks that can be executed concurrently in such environments, the time to calculate the alignment between two very long sequences, which may take days if run on a serial computing architecture, can be reduced to a matter of seconds. Furthermore, the blocks-wise computation optimization described above means that it can make effective use of the memory hierarchies in such joint CPU-GPU architectures and also minimize the overhead of data transfers to and from the GPU, thereby further improving performance.

2.4.8 Hotspot

The Hotspot algorithm is a hot-spot simulator of processor chips. It is useful for the evaluation and optimization of design in such a way that processor chips do not overheat, compromising the reliability and performance of the same. Its unique algorithmic structure allows parallel computation, so it perfectly fits because the temperature calculation depends on its adjacent ones, doing the simultaneous updating of several segments. Such an approach has thus manifested effective results with regard to benchmarking parallel and heterogeneous computing systems. Such an algorithm, because of its computation over several processors or computational units, represents parallel architectures with the ability to carry out complex and high-computing-rate tasks in an economical way. Its main factors of consideration are its applicability in electronic design automation, as well as the impact on further developments of research in parallel computing to give hints at scalable computing solutions for high-performance applications.

2.4.9 Hotspot3D

The HotSpot3D algorithm provides a key solution aimed at thermal analysis in complex electronic systems. This allows accurate three-dimensional multi-layer conduction simulation over the structures according to detailed heat distribution that can help place the hotspots with a view to ensuring the stability of high-density electronic devices. Any system retains its performance levels and reliability so long as it has efficient heat dissipation, because both are directly proportional to coolness. The distinctive layered approach of the HotSpot3D algorithm allows parallel processing of thermal simulation, whereby thermal calculations of each layer are done independently. Such schemes provide for distributing workloads across many such processing units, which have an advantage in processing large and high-resolution three-dimensional thermal models in real-time thermal management of electronics. Therefore, the ease with which it can adapt to parallel processing makes it an elegant and convenient means to evaluate the performances of various parallel computing architectures.

2.5 Challenges when developing benchmarks with TornadoVM

2.5.1 From C/C++ to Java

At the low level, the major difference between C and Java is the hardware platform. While C programs are compiled into the native machine language, Java programs are compiled into Java Bytecode, which is a high level interpretation that ensures the portability of Java programs across multiple hardware ISAs. Additionally, a second difference in coding programs between C/C++ and Java is related to memory management, as well as the way in which pointers are applied. While writing software in C/C++, programmers are supposed to allocate and free memory resources, such as by hand coding through functions malloc (in C) or new (in C++) for allocating memory and free (in C) or delete (in C++) for freeing up memory resources. In contrast, Java comes with its own garbage collector that carries out the work of automatic memory management, leaving Java programmers without the need to write code for memory management.

However, the garbage collection mechanism of the JVM may cause overhead which reduces the performance of the program, and programmers need to spend time considering a suitable method to adjust several parameters manually [1]. Another difference is that C/C++ works with pointers, which are an excellent tool for managing memory and handling arrays. Java does not support pointers; it uses so-called references instead, which compromise efficiency to be better on ease of development and maintenance. What this essentially means is that this project required to be very proficient in and possess good experience in writing programs in both languages: C/C++ and Java.

2.5.2 From OpenMP to TornadoVM

While OpenMP and TornadoVM are grounded in C/C++ and Java, respectively, truly mastering these languages goes beyond mere syntax comprehension. The real challenge lies in adeptness with parallelism and heterogeneous programming. It is tricky to identify which code segments should be parallelized and how to efficiently move data between devices and hosts. Missteps here can degrade performance or yield incorrect results. TornadoVM leverages Java annotations and specialized techniques like task graphs to streamline parallel and heterogeneous computing, while OpenMP opts for direct directives. The stark differences in their coding approaches mean programmers might need extra time to adapt when switching between OpenMP and TornadoVM. Additionally, in parallel computing, a “barrier” is usually required and used to make all threads meet up at a particular point during the execution, but with Loop Parallel API programming, which is more suitable for non-GPU/FPGA expert programmers and mostly used in this project, “barrier” is not supported.

2.5.3 From TornadoVM to Implementing Algorithms

The benchmark suite includes a diverse set of algorithms, each presenting distinct computational challenges and patterns. Transitioning these algorithms to TornadoVM requires a reevaluation of data structures and parallelization approaches to align with the Java and TornadoVM execution environments. This task can be especially challenging for algorithms originally designed for alternative parallel computing models or those dependent on low-level optimizations not easily mapped to the higher-level abstractions of TornadoVM because of the lack of control of memory and hardware-specific optimizations of TornadoVM.

2.5.4 Unsupported Features of TornadoVM

TornadoVM does not support all features of Java due to OpenCL, which is the underlying model of TornadoVM [29]

- Like OpenCL and CUDA, recursions or printing messages are not supported by TornadoVM.
- Structures or classes are very commonly used in programming languages, including C, C++, and Java. One drawback of TornadoVM is that objects, except those replacing primitive type arrays or matrixes, are not supported to be used in task graphs. In consequence, programmers using TornadoVM have to avoid using customized objects, which may cause more development time and lower code readability.
- Unlike OpenMP, in a for loop, TornadoVM can only support the local variable starting from zero. Hence, in some situations, like when the remaining code is to access specific indexes not including zero, it requires programmers to use another logic to implement using TornadoVM.
- Using output messages for debugging is common and useful. However, since OpenCL, CUDA or SPIR-V do not support printing messages when running a kernel, printing messages is not supported by TornadoVM, it may cause programmers to spend more time on debugging.

Chapter 3

Design and Implementation

3.1 Installation of TornadoVM

The prerequisites for installing TornadoVM (version 1.0.1) are about the versions of Maven, CMake, GCC, Python, the operating system, and drivers. There are both instructions for automatic and manual installation, which can be seen on the official website or the public repository of TornadoVM [30]. In this project, Maven 3.6.3, CMake 3.6, GCC 9.0, Python 3.0, JDK 21, and OpenCL & NVIDIA drivers are used. After the installation, there are unit tests that can be run to validate whether the environment and installation of TornadoVM are set properly. If above 90% of the unit tests are passed, then the installation can be proved successful.

3.2 Input Examples

The Rodinia Benchmark [32] provides sample data and data generation functions. As in this project, the main aim is to port the implemented algorithms; the same sample data and data generation functions are used to run and test the programs.

3.3 Data Representation

TornadoVM offers encapsulated array types to help programmers to express data and map the data to vectorized types such as “VectorInt” and “VectorFloat”, which are used to replace “int[]” and “float[]” individually. In addition, TornadoVM also has its own off-heap data types such as “IntArray” and “FloatArray” which are equivalent to

on-heap	off-heap
int[]	IntArray/VectorInt
float[]	FloatArray/VectorFloat
double[]	DoubleArray/VectorDouble
int[][]	Matrix2DInt
float[][]	Matrix2DFloat
double[][]	Matrix2DDouble

Table 3.1: Conversion from on-heap types to off-heap types

"int[]" and "float[]" respectively. Table 3.1 shows the conversions for commonly used types. The reason for the provision and use of the off-heap memory is that declaring memory to be off-heap and making it accessible to the user is one way to get around blocking data transfers, and ensure that garbage collection will not take place [3]. As a result, the CPU threads may be able to go on normally after off-heap memory has been declared.

3.4 TornadoVM Device

- The command *$ tornado - - devices* is used to find out the device ids that TornadoVM is discovering. Then each device that TornadoVM finds is linked to a pair of id numbers that match the kind of driver and the particular device:

  ```
  Tornado device=<driverNumber>:<deviceNumber>
  ```

- The dynamic reconfiguration feature of TornadoVM allows programmers to configure the runtime for migrating the execution from one device to another based on three provided pilicies:

  - LATENCY: Instead of evaluating each device's execution before making a choice, the TornadoVM runtime switches context with the first device to complete the execution. As a result, the fastest device can be returned.
  - PERFORMANCE: following a device warm-up (JIT compilation is excluded). Before making a choice, the TornadoVM runtime assesses the execution for every device.
  - END_2_END: take into account the buffer allocations and JIT compilation to get the best-performing device.

3.5 TornadoVM Profiler

- Programmers can use *–enableProfiler* `<silent|console>` to launch the profiler of Tornado. There are two options for programmers to choose from, which are “console” and “silent”. “console” outputs a message in JSON format for each task graph, while “silent” makes the profiling information available in silent mode.

- *executionPlan.getProfilerResult().getCompilationTime()* can be used to obtain the time taken to compile Java bytecode into the device-specific code (e.g., CUDA or OpenCL). Understanding the compilation time can help programmers identify the overhead introduced by the JIT compilation process and explore ways to reduce it.

3.6 Methodology

In Section 3.6.1, some important OpenMP directives which are commonly used in parallel and heterogeneous programming. Following this, Section 3.6.2 shows how TornadoVM deals with common data structures in C/C++ and Java.

3.6.1 Common OpenMP directives

- **get_num_threads**: a part of the OpenMP runtime library and is used to obtain the number of threads currently in the team executing a parallel region. This function is useful for understanding the parallelism level and for making decisions based on the number of threads. In TornadoVM, programmers must use the *KernelContext* object instead of using a single annotation to get the number of threads.

- **parallel** lets the compiler execute the following block of code in parallel by multiple threads. The runtime system creates a team of threads, and each thread executes the code block independently. This primary feature can be implemented by using the annotation *Parallel* in TornadoVM.

- **reduction** ensures that each thread performs its portion of the operation in a thread-private copy of the variable, and then combines these partial results into a single final outcome using a specified operator (e.g., summation, max, min).

Similar to the parallel feature, the reduction can be replaced by the annotation *Reduce* in TornadoVM.

- **shared** makes the listed variables to be shared among all threads in the team. This means that there is only one instance of each shared variable, and it is accessible by all threads.

- **private** is used to declare variables which are private to each thread. Each thread gets its own copy of the variable, and modifications made by one thread do not affect the copies in other threads.

- **firstprivate** is similar to private, but it initializes each thread's private copy of the variable with the original value that the variable had before entering the parallel region.

- **schedule** is used to control how iterations are distributed among threads. The types of scheduling include *static*, *dynamic*, and *auto*, where each has different implications for performance and workload distribution. *Static* scheduling divides loop iterations into fixed-size chunks or nearly equal portions if the chunk size is not specified, and assigns each chunk to a thread in the team before the loop execution starts. The division of iterations is done at compile time or at the start of the loop, making it predictable. When the chunk size is specified, OpenMP divides the loop iterations into chunks of that size and assigns them to the threads in a round-robin fashion. If the chunk size is not specified, OpenMP divides the loop into equal-sized portions or as equal as possible based on the number of threads. As shown in Figure 3.1, loop iterations are divided into

```
#pragma omp parallel for schedule(static, 4)
for(int i = 0; i < N; i++) {
    // Loop body
}
```

Figure 3.1: Example of Static scheduling

chunks of 4 iterations each, and each chunk is assigned to a thread. If there are 8 iterations and 2 threads, thread 0 will execute iterations 0-3, and thread 1 will execute iterations 4-7. Using *Static* scheduling ensures predictability and low overhead since the scheduling is done upfront, but can lead to load imbalance if iterations have varying computational loads, as each thread is assigned a fixed number of iterations. *Dynamic* scheduling is designed to handle workloads where loop iterations have varying computational loads. It assigns a chunk of iterations to a thread dynamically at runtime, as threads become available. The loop iterations are divided into chunks of the specified size. When a thread finishes its current chunk of iterations, it requests the next available chunk until all iterations are completed. If the chunk size is not specified, a default size is chosen. According to Figure 3.2, iterations are dynamically assigned to threads

```
#pragma omp parallel for schedule(dynamic, 4)
for(int i = 0; i < N; i++) {
    // Loop body
}
```

Figure 3.2: Example of Dynamic scheduling

in chunks of 4. Once a thread finishes its chunk, it retrieves the next available chunk, continuing until there are no more iterations. The use of *Dynamic* scheduling can cause better load balancing for loops with irregular workloads, as faster threads can take on more work. However, overhead can be higher due to dynamic scheduling and potential contention among threads requesting new chunks. *Auto* scheduling delegates the decision of how to schedule the loop iterations to the OpenMP runtime. This allows the runtime to choose the scheduling based on the loop characteristics and system load, potentially optimizing for the current execution context. The specifics of how *Auto* scheduling works are implementation-dependent and not defined by the OpenMP standard. The runtime may use heuristics or historical performance data to choose the best scheduling strategy for a given loop. Figure 3.3 shows that the OpenMP runtime

```
#pragma omp parallel for schedule(auto)
for(int i = 0; i < N; i++) {
    // Loop body
}
```

Figure 3.3: Example of Auto scheduling

decides the scheduling strategy for the loop iterations. The decision is based on the runtime's knowledge and heuristics, which may vary by implementation and execution context. Due to the features of *Auto* scheduling, potentially optimal scheduling without requiring the programmer to specify a scheduling strategy.

Useful when the optimal scheduling strategy is unknown or varies between executions. Despite this, there is a lack of predictability and control, as it is chosen during the runtime. Choosing the right scheduling strategy depends on the nature of the workload, the problem domain, and performance testing. *Static* scheduling is often suitable for loops with iterations that have similar computational costs. In contrast, *Dynamic* scheduling can help balance workloads where iteration costs vary significantly. *Auto* scheduling can be a good default choice when the optimal scheduling strategy is not clear, allowing the runtime to make an informed decision.

- **default (none)** is used to enforce explicit data scoping for variables in this parallel region. When default(none) is specified, all variables within the scope of an OpenMP directive in question are required to have their data-sharing attributes explicitly specified. Hence, every variable in the parallel region must explicitly re-state whether it is shared, private, firstprivate, etc. The advantages of using *default (none)* are that as well as preventing the accidental sharing of variables, it also forces the programmer to consider and explicitly specify the desired sharing semantics of each variable. It takes errors and other problems that would occur at run-time and instead makes them compile-time errors, which is easier to rectify. However, this adds additional complexity to the code – the declaration must be entered for every qualified variable in the parallel program. It may be tedious in heavily parallelized regions, where changing the sharing characteristics of numerous variables must be done carefully.

- **master** ensures that the following block of code will be executed by the master thread of the team, which is the one with the ID 0. This is useful for any operation that should be performed only a single time, such as initializing or finalizing the data structure, or for input/output operations.

- **simd** helps to vectorize loops. It is nothing but a hint to the compiler to generate the SIMD instructions to process multiple routine data elements in parallel with a single instruction, which can significantly speed up the process on hardware that supports vectorization.

- **target** is used in heterogeneous computing, which permits part of an application's code to be run on some computing device types. It offloads the code section to a target device, such as GPUs or other accelerators. In order to deal with the heterogeneous computing, TornadoVM utilizes *TaskGraph* instead of annotations like *Parallel* and *Reduce*. Finally, to replace OpenMP features above, *shared*, *private*, *firstprivate*, and *default* need not be stated, thanks to Java's object-oriented components and TornadoVM's programming mode. At the very low level, the implementation of schedule, master, and simd, is currently not supported by TornadoVM.

3.6.2 Data structures in C/C++ and Java

- **Pointers** allow for direct access and manipulation of memory. Arrays can be accessed through pointers, using arithmetic operations to access and manipulate their elements. Instead, Java handles memory management and object access through references, including for arrays. References in Java are similar to pointers in C/C++ because they point to an object's memory location, but they do not allow direct memory manipulation or arithmetic operations like pointers.

- **Struct** groups multiple variables, possibly of different types, into one type. Java does something similar with classes, encapsulating data fields and offering methods to work with the data. Figure 3.4 and Figure 3.5 illustrate the combination of the utilization of *Pointers* and *Struct* by using C/C++ and Java. However, since

```c
struct Point {
    int x;
    int y;
};

int main() {
    struct Point *array = (struct Point*)malloc(10 * sizeof(struct
   Point));
    free(array);
    return 0;
}
```

Figure 3.4: Use of Struct and Pointer in C

TornadoVM does not support user-defined classes, attributes in classes are converted by using arrays or the corresponding off-heap vectors. Figure 3.6 shows the corresponding code which achieves the same functionality. In this example, the attributes *x* and *y* of *Point* which are stored in an array in Figure 3.4 are both converted to separated TornadoVM object types, *VectorInt*, so that the program can be executed successfully using TornadoVM.

```java
class Point {
    int x;
    int y;
}

public class Main {
    public static void main(String[] args) {
        Point[] array = new Point[10];
    }
}
```

Figure 3.5: Use of Class and Array in Java

```java
public class Main {
    public static void main(String[] args) {
        VectorInt PointX = new VectorInt(10);
        VectorInt PointY = new VectorInt(10);
    }
}
```

Figure 3.6: Use of off-heap types in TornadoVM

3.7 Implemented Algorithms

3.7.1 Breadth First Search

To implement the Breadth First Search (bfs) algorithm, the only meaningful parameter here is the path of the input file, which contains the total number of nodes, the stating nodes, the corresponding number of edges, and the source node. As the user-defined type "structure" is supported in C++ and OpenMP, an array named "h_graph_node" can be created to store structures that record the starting nodes and number of edges (shown in Figure 3.7). Figure 3.8 illustrates how to replace the C structure with a

```c
struct Node{
  int starting;
  int no_of_edges;
};
Node* h_graph_nodes = (Node*)malloc(sizeof(Node)*no_of_nodes);
```

Figure 3.7: Use of the structure in bfs using OpenMP

Java class. However, as TornadoVM does not support user-defined classes, two variables of the type "VectorInt" are used to replace the structure and class, as each element of the input file is assumed to be an integer, as shown in Figure 3.9. Fig-

```
class Node {
    int starting;
    int no_of_edges;
}
Node[] h_graph_nodes = new Node[no_of_nodes];
```

Figure 3.8: Use of the class to replace the structure using Java

```
static VectorInt h_graph_nodes_starting;
static VectorInt h_graph_nodes_edges;
h_graph_nodes_starting = new VectorInt(no_of_nodes);
h_graph_nodes_edges = new VectorInt(no_of_nodes);
```

Figure 3.9: Use of the VectorInt to replace the structure using TornadoVM

ure 3.10 shows how OpenMP is used in order to achieve heterogeneous computing for bfs. Variables such as *no_of_nodes*, *h_graph_mask*, *h_graph_nodes*, *h_graph_edges*, *h_graph_visited*, and *h_updating_graph_mask* are copied from the host memory to the target device memory to perform the parallel BFS operations on the device. On the other hand, *map(h_cost[0:no_of_nodes])* means that *h_cost* will be copied to the target device before the computation, and the results will be copied back to the host after the computation finishes. There are two parallelized loops. In the first loop, all

```
#pragma omp target data map(to: no_of_nodes,
h_graph_mask[0:no_of_nodes],
h_graph_nodes[0:no_of_nodes],
h_graph_edges[0:edge_list_size],
h_graph_visited[0:no_of_nodes],
h_updating_graph_mask[0:no_of_nodes])
map(h_cost[0:no_of_nodes])
```

Figure 3.10: OpenMP code for bfs heterogeneous computing

nodes in the graph are iterated. In each iteration, the current node *tid* is checked for being active in the BFS frontier indicated by *h_graph_mask[tid]==true*. For active nodes, all adjacent edges are explored by iterating through *h_graph_edges* from the starting index *h_graph_nodes[tid].starting* to the end of the edge list for that node. If an adjacent node (id) is not visited, then its cost *h_cost[id]* is updated to the cost of the current node plus 1 (weight of the edge) and marks the adjacent node for inclusion in the next frontier *h_updating_graph_mask[id]=true*. This parallelization is effective as each iteration operates on independent data segments. Thus, there is less chance of data race conditions, and concurrent execution is safer and more efficient. In the other loop, the BFS frontier is updated. All nodes are iterated, and if they are marked for inclusion in the next frontier by *h_updating_graph_mask[tid]*, then three operations are performed. This node is added to the current frontier *h_graph_mask[tid]=true*, the node is marked as visited *h_graph_visited[tid]=true*, and the update mask is reset *h_updating_graph_mask[tid]=false*. This step is parallelizable, as the state of each node is updated independently. There is no dependency among the states of different nodes, and thus multiple threads can update the state concurrently without interfering with each other. Figure 3.11 shows the use of the two parallelized loops. Fig-

```
#pragma omp parallel for
for (int tid = 0; tid < no_of_nodes; tid++) {
    if (h_graph_mask[tid] == true) {
        h_graph_mask[tid] = false;
        for (int i = h_graph_nodes[tid].starting; i < (h_graph_nodes
    [tid].no_of_edges + h_graph_nodes[tid].starting); i++) {
            int id = h_graph_edges[i];
            if (!h_graph_visited[id]) {
                h_cost[id] = h_cost[tid] + 1;
                h_updating_graph_mask[id] = true;
            }
        }
    }
}

#pragma omp parallel for
for (int tid = 0; tid < no_of_nodes; tid++) {
    if (h_updating_graph_mask[tid] == true) {
        h_graph_mask[tid] = true;
        h_graph_visited[tid] = true;
        stop = true;
        h_updating_graph_mask[tid] = false;
    }
}
```

Figure 3.11: OpenMP code for bfs parallelized loops

ure 3.12 illustrates how TornadoVM is used to replace the above OpenMP code. The two parallelized loops are encapsulated into two methods with an extra *Parallel* annotation for each, which indicates the parallel execution. The data transfer is managed through the *TaskGraph* API, which allows specifying data transfer modes for each task (method containing the parallelized loop). As shown in Figure 3.13, *.transferToDevice(DataTransferMode.FIRST_EXECUTION,...)* ensures that data is moved to the device before the first execution of the task graph. This is analogous to *map(to:*

```java
public static void initMask(VectorInt h_graph_nodes_starting, VectorInt h_graph_nodes_edges, VectorInt h_graph_mask, VectorInt h_graph_visited, VectorInt h_graph_edges, VectorInt h_cost, VectorInt h_updating_graph_mask) {
    for (@Parallel int tid = 0; tid < h_graph_nodes_starting.size(); tid++) {
        if (h_graph_mask.get(tid) == 1) {
            h_graph_mask.set(tid, 0);
            for (int i = h_graph_nodes_starting.get(tid); i < (h_graph_nodes_starting.get(tid) + h_graph_nodes_edges.get(tid)); i++) {
                int id = h_graph_edges.get(i);
                if (h_graph_visited.get(id) == 0) {
                    h_cost.set(id, h_cost.get(tid) + 1);
                    h_updating_graph_mask.set(id, 1);
                }
            }
        }
    }
}
public static void updateMask(VectorInt h_updating_graph_mask, VectorInt h_graph_mask, VectorInt h_graph_visited, VectorInt stop) {
    for (@Parallel int tid = 0; tid < h_updating_graph_mask.size(); tid++) {
        if (h_updating_graph_mask.get(tid) == 1) {
            h_graph_mask.set(tid, 1);
            h_graph_visited.set(tid, 1);
            stop.set(0, 1);
            h_updating_graph_mask.set(tid, 0);
        }
    }
}
```

Figure 3.12: TornadoVM code for bfs parallelized loops

```
TaskGraph taskGraph1 = new TaskGraph("s1")
     .transferToDevice(DataTransferMode.FIRST_EXECUTION,
    h_graph_nodes_starting, h_graph_nodes_edges, h_graph_mask,
    h_graph_visited, h_graph_edges, h_cost)
     .task("t1", Bfs::initMask, h_graph_nodes_starting,
    h_graph_nodes_edges, h_graph_mask, h_graph_visited, h_graph_edges
    , h_cost, h_updating_graph_mask)
     .transferToHost(DataTransferMode.FIRST_EXECUTION, h_graph_mask,
    h_cost, h_updating_graph_mask);
ImmutableTaskGraph immutableTaskGraph1 = taskGraph1.snapshot();
TornadoExecutionPlan executor1 = new TornadoExecutionPlan(
    immutableTaskGraph1);
TaskGraph taskGraph2 = new TaskGraph("s2")
     .transferToDevice(DataTransferMode.EVERY_EXECUTION,
    h_updating_graph_mask, stop)
     .task("t2", Bfs::updateMask, h_updating_graph_mask, h_graph_mask
    , h_graph_visited, stop)
     .transferToHost(DataTransferMode.FIRST_EXECUTION,
    h_updating_graph_mask, h_graph_mask, h_graph_visited, stop);
ImmutableTaskGraph immutableTaskGraph2 = taskGraph2.snapshot();
TornadoExecutionPlan executor2 = new TornadoExecutionPlan(
    immutableTaskGraph2);
```

Figure 3.13: Taskgraphs for parallelized loops

...) in the OpenMP directive, specifying the data to be copied to the device. Furthermore, *.transferToHost(DataTransferMode.FIRST_EXECUTION,...)* specifies the variables whose updated values should be copied back to the host after execution, similar to how *map(from: ...)* would work in OpenMP. This explicit data transfer management in TornadoVM replaces the need for *#pragma omp target data map(...)* by providing fine-grained control over when and how data moves between the host and the device. In order to launch the executions, simply use *executor1.execute()* and *executor2.execute()* (as shown in Figure 3.14)

```
do {
    stop.set(0, 0);
    executor1.execute();
    executor2.execute();
} while (stop.get(0) == 1);
```

Figure 3.14: Launch the executions

3.7.2 Kmeans

The Kmeans algorithm takes three arguments: the files with the data to be clustered, the number of clusters, and the threshold value. Figure 3.15 shows the parallelized section, with numerous common keywords in OpenMP, and that is what Figure 3.16 looks like in the corresponding TornadoVM Java code. The loop iterates over all points in the dataset by assigning each to the closest cluster centre and updating temporary cluster centres appropriately. The loop is parallelizable in that each iteration does not depend on others, i.e., finding the next point's closest cluster centre and then updating the temporary cluster centres do not depend on any other point's results. The explicit shared, *private*, and *firstprivate* variables need not be stated, thanks to Java's object-oriented components and TornadoVM's programming model. Unlike OpenMP, the *reduction* variable *delta* needs to be a reference type instead of a primitive type in TornadoVM. A bug was found when implementing the Kmeans using the TornadoVM on this algorithm: the data type *Matrix2DDouble* could not work with the annotation *Reduce*. Figure 3.17 shows the use of the task graph, as some variables are only used for reading but not writing, so they are only transferred from the host to the device and never transferred from the device back to the host.

```
#pragma omp for \
    private(i,j,index) \
    firstprivate(npoints,nclusters,nfeatures) \
    schedule(static) \
    reduction(+:delta)
for (i=0; i<npoints; i++) {
    index = find_nearest_point(feature[i], nfeatures,
    clusters, nclusters);
    if (membership[i] != index) delta += 1.0;
    membership[i] = index;
    partial_new_centers_len[tid][index]++;
    for (j=0; j<nfeatures; j++)
        partial_new_centers[tid][index][j] += feature[i][j];
}
```

Figure 3.15: OpenMP code for kmeans parallelized loops

3.7.3 Particlefilter

The Particlefilter algorithms requires 4 input: the width and height of the video frames, number of frames, and the number of particles. It consists of many steps and 11 of them

```java
public static void parallel(int npoints, int nfeatures,
                            int nclusters, Matrix2DFloat feature,
                            Matrix2DFloat tmp_cluster_centres,
                            Matrix2DFloat new_centers,
                            VectorInt index,
                            VectorInt membership,
                            VectorInt new_centers_len,
                            @Reduce VectorFloat delta
                            ){
    for (@Parallel int i = 0; i < npoints; i++) {
        find_nearest_point(i, feature, nfeatures,
    tmp_cluster_centres, nclusters, index);
        if (membership.get(i) != index.get(0)) {
            delta.set(0, delta.get(0) + 1.0f);
        }
        membership.set(i, index.get(0));
        new_centers_len.set(index.get(0), new_centers_len.get(index.
    get(0)) + 1);
        for (int j = 0; j < nfeatures; j++){
            new_centers.set(index.get(0), j, new_centers.get(index.
    get(0), j) + feature.get(i, j));
        }
    }
}
```

Figure 3.16: TornadoVM Java code for kmeans parallelized loops

```java
TaskGraph taskGraph1 = new TaskGraph("s1")
    .transferToDevice(DataTransferMode.EVERY_EXECUTION, npoints,
    nfeatures, nclusters, feature, tmp_cluster_centres, new_centers,
    index, membership, new_centers_len, delta)
    .task("t1", Kmeans::parallel, npoints, nfeatures, nclusters,
    feature, tmp_cluster_centres, new_centers, index, membership,
    new_centers_len, delta)
    .transferToHost(DataTransferMode.EVERY_EXECUTION, new_centers,
    index, membership, new_centers_len, delta);
ImmutableTaskGraph immutableTaskGraph1 = taskGraph1.snapshot();
TornadoExecutionPlan executor1 = new TornadoExecutionPlan(
    immutableTaskGraph1);
executor1.execute();
```

Figure 3.17: TornadoVM Java code for task graph

can be parallelized. The methods *setWeights*, *setArrayXY*, *updateArrayXY*, *computeLikelihood*, *updateWeights*, *normaliseWeights*, *findU*, *findNewArray*, *resetWeights* are similar in terms of their functionality and implementation by TornadoVM, because all of them are used for the simple initializations and updates, which makes all modifications of each element is independent of others so that all operations are included in a single *for* loop with the annotation *Parallel*. Figure 3.18 indicates one of these

```
#pragma omp parallel for shared(weights, Nparticles) private(x)
for(x = 0; x < Nparticles; x++){
    weights[x] = 1/((double)(Nparticles));
}
```

Figure 3.18: OpenMP C code for a method without Reduction

```
public static void setWeights(DoubleArray weights) {
    for (@Parallel int x = 0; x < weights.getSize(); x++) {
        weights.set(x, 1 / ((double)(weights.getSize())));
    }
}
```

Figure 3.19: TornadoVM Java code for a method without Reduction

methods *setWeights* by OpenMP code, while Figure 3.19 shows the corresponding TornadoVM code. Furthermore, *computeSumWeights*, *moveObjectXe*, *moveObjectYe* utilizes the annotation *Reduce*, as each of them contains a variable to be summed up at last. Figure 3.20, Figure 3.21 shows the OpenMP code and TornadoVM code for the method *computeSumWeights* individually. As shown in Figure 3.22, *weights* with the

```
#pragma omp parallel for private(x) reduction(+:sumWeights)
for(x = 0; x < Nparticles; x++){
    sumWeights += weights[x];
}
```

Figure 3.20: OpenMP C code for a method with Reduction

DoubleArray type stores the weight of each particle, and *sumWeights* with the same data type stores only one element which is the sum of the weights of all particles. It is initialized as zero and then accumulates in the method by the use of *Reduce*.

```java
public static void computeSumWeights(DoubleArray weights, @Reduce
    DoubleArray sumWeights) {
     for (@Parallel int x = 0; x < weights.getSize(); x++) {
          sumWeights.set(0, sumWeights.get(0) + weights.get(x));
     }
}
```

Figure 3.21: TornadoVM Java code for a method with Reduction

```java
TaskGraph taskGraph5 = new TaskGraph("s5")
     .transferToDevice(DataTransferMode.EVERY_EXECUTION, weights,
    sumWeights)
     .task("t5", Particlefilter::computeSumWeights, weights,
    sumWeights)
     .transferToHost(DataTransferMode.EVERY_EXECUTION, sumWeights);
ImmutableTaskGraph immutableTaskGraph5 = taskGraph5.snapshot();
TornadoExecutionPlan executor5 = new TornadoExecutionPlan(
    immutableTaskGraph5);
executor5.execute();
```

Figure 3.22: Task graph for a method with Reduction

3.7.4 Pathfinder

There are two main parameters in Pathfinder algorithm: the width of the path (number of columns), and the number of steps in the path (number of rows). Figure 3.23 and Figure 3.24 show the only parallelized loop in the program. This loop iterates over

```cpp
#pragma omp parallel for private(min)
for(int n = 0; n < cols; n++){
  min = src[n];
  if (n > 0)
    min = MIN(min, src[n-1]);
  if (n < cols-1)
    min = MIN(min, src[n+1]);
  dst[n] = wall[t+1][n]+min;
}
```

Figure 3.23: OpenMP C++ code for the only parallelized loop

each column *n* of a row in a 2D array named matrix. For each column, it computes the minimum value among the current, left, and right elements of the previous row (*src*) and adds it to the current element of wall (*wall[t+1][n]*), storing the result in the destination array (*dst*). In detail, *min = src[n]* initializes min with the value of the current column in *src*. The first *if* condition checks if there is a left neighbour (n > 0)

```java
public static void parallel(int t, VectorInt src, VectorInt dst,
    Matrix2DInt wall){
     for (@Parallel int n = 0; n < cols; n++){
         int min = src.get(n);
         if (n > 0){
             min = TornadoMath.min(min, src.get(n - 1));
         }
         if (n < cols - 1){
             min = TornadoMath.min(min, src.get(n + 1));
         }
         dst.set(n, wall.get(t + 1, n) + min);
     }
}
```

Figure 3.24: TornadoVM Java code for the only parallelized loop

and updates *min* if the left neighbour is smaller, while the second *if* condition checks if there is a right neighbour (n < cols-1) and updates min if the right neighbour is smaller. Finally, *dst[n]* = *wall[t+1][n]* + *min* updates *dst* with the calculated minimum path cost to reach the current cell from the top row. As a result, each iteration of the loop can be executed independently because the calculation for each column *n* only depends on the values of *src* and *wall* at *n*, *n-1*, and *n+1*.

3.7.5 Speckle Reducing Anisotropic Diffusion

The SRAD algorithm takes the following 8 parameters: the number of rows and columns in the image or the domain, the initial row and the last row of the region of interest (ROI) within the image where speckle reduction will be performed, the initial column and the last column of the ROI, and the amount of diffusion that will occur, which is responsible for the trade-off between noise reduction and edge preservation, quantified by the parameter Lambda, and the number of iterations to complete. The main parallelized section is the first section: The gradient of the image intensity for each pixel in north (*dN*), south (*dS*), east (*dE*), and west (*dW*) directions is determined, and is used to compute the diffusion coefficient (*c*). The diffusion coefficient (*c*) describes how much the new value of the current pixel will be influenced by the neighbouring pixel in the next iteration. This is done only with the pixel's eight immediate neighbours and its value. The gradient and the diffusion coefficient depend only on the current pixel under consideration, and earlier output does not impact it. Therefore, this operation can be parallelized. Figure 3.25 and Figure 3.26 illustrates the implemented OpenMP C++ and TornadoVM Java code. As shown in Figure 3.27

```
#pragma omp parallel for shared(J, dN, dS, dW, dE, c, rows, cols, iN
    , iS, jW, jE) private(i, j, k, Jc, G2, L, num, den, qsqr)
for (int i = 0 ; i < rows ; i++) {
    for (int j = 0; j < cols; j++) {
        k = i * cols + j;
        Jc = J[k];
        // directional derivates
        dN[k] = J[iN[i] * cols + j] - Jc;
        dS[k] = J[iS[i] * cols + j] - Jc;
        dW[k] = J[i * cols + jW[j]] - Jc;
        dE[k] = J[i * cols + jE[j]] - Jc;
        G2 = (dN[k]*dN[k] + dS[k]*dS[k] + dW[k]*dW[k] + dE[k]*dE[k])
     / (Jc*Jc);
        L = (dN[k] + dS[k] + dW[k] + dE[k]) / Jc;
        num  = (0.5*G2) - ((1.0/16.0)*(L*L)) ;
        den  = 1 + (.25*L);
        qsqr = num/(den*den);
        // diffusion coefficent (equ 33)
        den = (qsqr-q0sqr) / (q0sqr * (1+q0sqr)) ;
        c[k] = 1.0 / (1.0+den) ;
        // saturate diffusion coefficent
        if (c[k] < 0) {c[k] = 0;}
        else if (c[k] > 1) {c[k] = 1;}
  }
}
```

Figure 3.25: OpenMP C++ code for the first parallelized section

```java
public static void parallel1(VectorInt intParas, VectorFloat doubleParas, VectorFloat J, VectorFloat dN, VectorFloat dS, VectorFloat dW, VectorFloat dE, VectorInt iN, VectorInt iS, VectorInt jW, VectorInt jE, VectorFloat c) {
    for (@Parallel int i = 0; i < intParas.get(0); i++) {
        for (int j = 0; j < intParas.get(1); j++) {
            int k = i * intParas.get(1) + j;
            float Jc = J.get(k);
            // directional derivatives
            dN.set(k, J.get(iN.get(i) * intParas.get(1) + j) - Jc);
            dS.set(k, J.get(iS.get(i) * intParas.get(1) + j) - Jc);
            dW.set(k, J.get(i * intParas.get(1) + jW.get(j)) - Jc);
            dE.set(k, J.get(i * intParas.get(1) + jE.get(j)) - Jc);
            float G2 = (dN.get(k) * dN.get(k) + dS.get(k) * dS.get(k) +
                    dW.get(k) * dW.get(k) + dE.get(k) * dE.get(k)) / (Jc * Jc);
            float L = (dN.get(k) + dS.get(k) + dW.get(k) + dE.get(k)) / Jc;
            float num = (float) ((0.5 * G2) - ((1.0 / 16.0) * (L * L)));
            float den = (float) (1 + (.25 * L));
            float qsqr = num / (den * den);
            // diffusion coefficient (equ 33)
            den = (qsqr - doubleParas.get(1)) / (doubleParas.get(1) * (1 + doubleParas.get(1)));
            c.set(k, (float) (1.0 / (1.0 + den)));
            // saturate diffusion coefficient
            if (c.get(k) < 0) {
                c.set(k, 0);
            } else if (c.get(k) > 1) {
                c.set(k, 1);
            }
        }
    }
}
```

Figure 3.26: TornadoVM Java code for the first parallelized section

and Figure 3.28, the second parallelized section updates the image by determining the new value of each pixel based on the diffusion coefficient (c). The divergence (D) of the flux is computed for each pixel by determining how the flux has diverged at this pixel compared to its neighbouring pixels, and the pixel's value is adjusted. The calculation of divergence (D) depends on the current pixel and its neighbours only, and earlier pixels are independent. These crucial variables are set to be shared as they can be updated by many threads, but all threads need to get their latest values.

```
#pragma omp parallel for shared(J, c, rows, cols, lambda) private(i,
     j, k, D, cS, cN, cW, cE)
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        // current index
        k = i * cols + j;
        // diffusion coefficent
        cN = c[k];
        cS = c[iS[i] * cols + j];
        cW = c[k];
        cE = c[i * cols + jE[j]];
        // divergence (equ 58)
        D = cN * dN[k] + cS * dS[k] + cW * dW[k] + cE * dE[k];
        // image update (equ 61)
        J[k] = J[k] + 0.25*lambda*D;
    }
}
```

Figure 3.27: OpenMP C++ code for the second parallelized section

3.7.6 Lower-Upper Decomposition

The LUD algorithm allows users to input either a text file containing the matrix to be computed, or the size of the matrix to be generated. *default(none)* is used to enforce strict variable scoping within the OpenMP parallel region. It requires that all variables used inside the *for* loop be explicitly classified as either *private*, *shared*, *firstprivate*, *lastprivate*, or *reduction*, which makes sure that each variable's sharing attributes are specified by programmers. The balance between simplicity and potentially a good work result without needing to manually tune the scheduling strategy is achieved by the use of *schedule(auto)*. There are two parallel sections inside this program. The first calculates the perimeter blocks of the matrix after having decomposed the diagonal block. The perimeter calculations consist of "top perimeter" and "left perimeter" processing steps. Parallelism is enforced by dividing the perimeter into chunks, where

```java
public static void parallel2(VectorInt intParas, VectorFloat
    doubleParas, VectorFloat c, VectorInt iS, VectorFloat dN,
    VectorFloat dS, VectorFloat dW, VectorFloat dE, VectorInt jE,
    VectorFloat J) {
    for (@Parallel int i = 0; i < intParas.get(0); i++) {
        for (int j = 0; j < intParas.get(1); j++) {
            // current index
            int k = i * intParas.get(1) + j;
            // diffusion coefficient
            float cN = c.get(k);
            float cS = c.get(iS.get(i) * intParas.get(1) + j);
            float cW = c.get(k);
            float cE = c.get(i * intParas.get(1) + jE.get(j));
            // divergence (equ 58)
            float D = cN * dN.get(k) + cS * dS.get(k) + cW * dW.get(
    k) + cE * dE.get(k);
            // image update (equ 61)
            J.set(k, (float) (J.get(k) + 0.25 * doubleParas.get(0) *
     D));
        }
    }
}
```

Figure 3.28: TornadoVM Java code for the second parallelized section

each chunk is generated by several steps of the processing steps above, i.e., representing a block of the matrix. Then, the other threads process these chunks in parallel. After the perimeter blocks are updated, the interior blocks of the matrix are updated. Another OpenMP section marks this by division of the interior blocks into chunks, and processing them in parallel, similar to the perimeter blocks. However, simply using TornadoVM annotations cannot achieve features like *schedule(auto)* although the final results are not influenced.

3.7.7 Needleman-Wunsch

The NW algorithm accepts three primary parameters: the size of the matrix used for sequence alignment. Because the program requires the lengths of the two sequences to be an exact multiple of 16 and that the matrix be square, the maximum number of rows and cols will need to be the same. As the dimensions of the matrix, the value that is the maximum dimension of the square matrix also indirectly defines the size of the sequence pairs being aligned. Another parameter required by the NW algorithm is the penalty for each gap between two sequential elements of the matrix. Gaps are generally initiated during sequence alignment between a pair of sequences with disparate lengths, or can be between two sequences and can be clamped. The penalty is a positive integer, which decreases the score of the alignment in question for an actual gap introduced. Assuming all other parameters, a lesser negative gap will prioritize one alignment over another. The first parallelized loop section computes the top-left matrix: the outer Loop iterates over the blocks diagonally. Each iteration corresponds to a diagonal pass through the matrix, starting from the top-left corner. The parallelized inner Loop iterates over the blocks within each diagonal. The variable *b_index_x* represents the x-coordinate of the block, and *b_index_y*, which is derived from *b_index_x* and *blk*, represents the y-coordinate. The *schedule(static)* ensures that iterations are evenly distributed among the available threads. The other loop section processes the top-left part of the matrix: the outer loop also iterates diagonally, but it starts from the second diagonal block since the first diagonal block was covered in the top-left matrix processing, and goes towards the bottom-right corner of the matrix. The parallelized inner loop iterates over the blocks within a diagonal pass. In this section, *b_index_x* starts from the last block of the previous diagonal (*blk - 1*) and goes to the last block in the row. *b_index_y* is calculated based on *b_index_x* and *blk*, representing the block's row index. It was found that TornadoVM does not support a parallelized loop not starting from 0, so the loop is slightly adjusted (shown in Figure 3.29 and Figure 3.30). The use of the variable 'blk' here is to ensure the program run properly even if the start and end index of the loop changes.

```
#pragma omp parallel for schedule(static) shared(input_itemsets,
    referrence) firstprivate(blk, max_rows, max_cols, penalty)
#endif
for( int b_index_x = blk - 1; b_index_x < (max_cols-1)/BLOCK_SIZE;
    ++b_index_x){
     int b_index_y = (max_cols-1)/BLOCK_SIZE + blk - 2 - b_index_x;
     // loop body
}
```

Figure 3.29: OpenMP C++ code for the parallelized section

3.7.8 Hotspot

In the Hotspot algorithm, five parameters are required: the number of rows and columns in the grid, the number of iterations, and the names of the files which contains the original temperature and dissipated power values of each cell. Only one parallelized part

```
public static void parallel2(VectorInt input_itemsets, VectorInt
    reference, VectorInt paras, VectorInt blk) {
    for (@Parallel int b_index_x = 0; b_index_x < ((paras.get(1) -
    1) / BLOCK_SIZE) - (blk.get(0) - 1); ++b_index_x) {
        int b_index_y = (paras.get(1) - 1) / BLOCK_SIZE + blk.get(0)
     - 2 - (b_index_x + blk.get(0) - 1);
        // loop body
    }
```

Figure 3.30: TornadoVM Java code for the parallelized section

exits in this program. The grid is divided into blocks or chunks, each consisting of a subset of the grid's rows and columns. This blocking approach helps in reducing memory access times and improving cache utilization. Each thread processes one or more chunks, computing the new temperature for each grid point in its chunks. The computation takes into account the temperatures of neighbouring points, the power dissipation at the point, and various physical and material properties. For grid points at the edges or corners of the grid, special boundary conditions are applied, as these points do not have a full set of neighbouring points. The results of these computations are written to the result array, which either becomes the new temp array for the next iteration or is used as the final output after all iterations are complete. Each grid point's temperature update is largely independent of others, especially when you consider the computation for a single time step. The new temperature at a grid point depends on its current temperature, the temperatures of its immediate neighbours, and the power dissipated at that point. Given that the original temperatures (*temp*) are read-only during the computation and the results (*result*) are written without reading, there is no data dependency that would prevent simultaneous updates, hence this section can be parallelized.

3.7.9 Hotspot3D

The parameters of the Hotspot3D algorithm are the same as those of the Hotspot algorithm. Inside the nested loops over x, y, and z indices, the temperature for each cell is computed based on its own temperature and the temperatures of its neighbours. This computation also includes the effect of power input (*pIn*) and ambient temperature (*amb_temp*). Because the computation for each cell is independent (given the previous state of the grid), this part of the code can be executed in parallel without needing synchronization between threads after each cell's temperature is computed.

The reason this section can be parallelized is due to the data independence between the calculations for each grid cell within an iteration. As long as each thread works on a different set of cells, there is no need for synchronization between threads during the computation, making it a good candidate for parallel execution.

Chapter 4

Evaluation

4.1 Performance

Device	**QUADRO GP100**	**RTX 4090**
Global Memory Size	15.9GB	23.6GB
Local Memory Size	48.0kb	48.0kb
Workgroup Dimensions	3	3
Total Number of Block Threads	[2147483647, 65535, 65535]	[2147483647, 65535, 65535]
Max Workgroup Configuration	[1024, 1024, 64]	[1024, 1024, 64]

Table 4.1: Device Information

Table 4.2 and Table 4.1 describe the input data size for each benchmark and the information of the GPU used for the performance evaluation. The TornadoVM implementation of the Rodinia benchmarks is not optimized, and in some cases results in lower performance than Java and OpenMP implementations (shown in Figure 4.1). The reason is that the parts that have been offloaded on the GPU device are separated

Algorithm	Size
Bfs	16M nodes
Hotspot	1024 grid and columns with 50000 iterations
Lud	2048 x 2048 matrix
Nw	4090 x 4090 matrix with a penalty of 10
Particlefilter	4096 width and height with 10000 particles
Pathfinder	20000 width and 100 steps
Srad	4096 rows and columns with 2 iterations

Table 4.2: Size of data sets for each benchmark

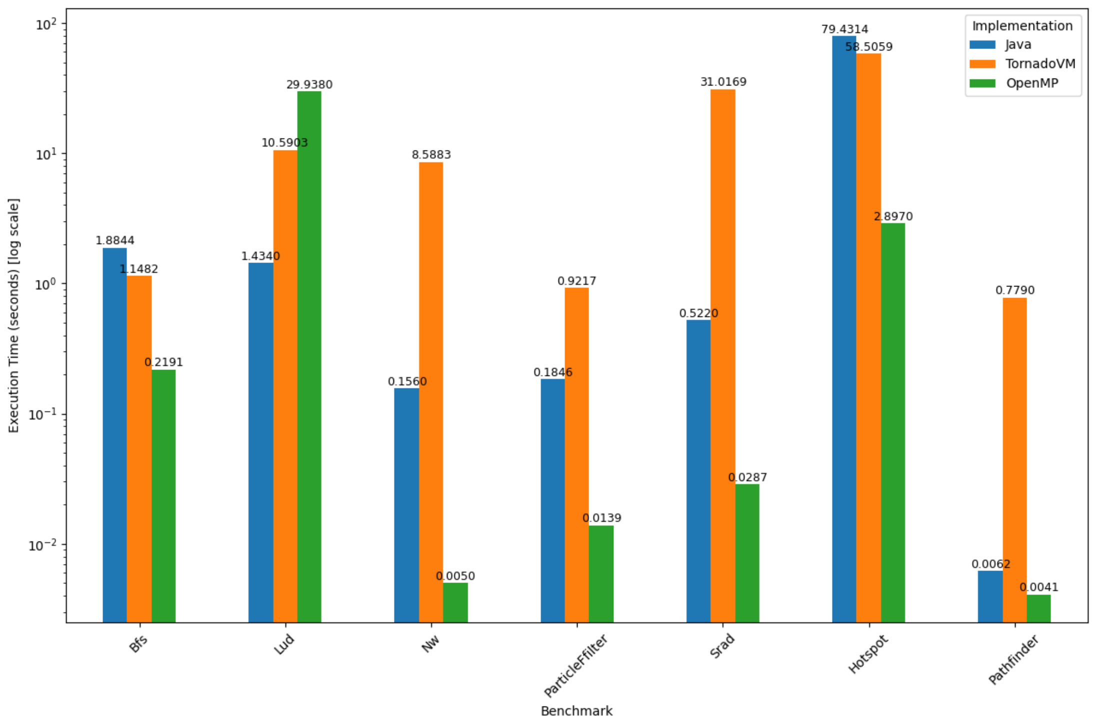

Figure 4.1: Execution time comparison among Java, TornadoVM, and OpenMP

into different TaskGraphs, therefore the data that are consumed by each TaskGraph are transferred to the device multiple times. This is anticipated to be addressed by clustering the accelerated tasks within the same task graphs. In order to test the performance accurately, several iterations with the executions of execution plans should be run before timing to increase the confidence of the obtained measurements.

Additionally, the execution times that are reported in this chapter for all systems are obtained from a single run. This performance methodology is biased in favor of the OpenMP implementation, as the Java programs may not be optimized by the JIT compiler, and the measurement of time for TornadoVM includes the compilation duration. The TornadoVM compilation process is divided into two separate compilation stages. The first stage concern the compilation of Java methods by the TornadoVM JIT compiler, while the second stage regards the compilation of the generated OpenCL kernels and it is performed by the GPU driver. To provide a close performance comparison between the TornadoVM implementations and the OpenMP implementations, the experiments should be revamped in order to perform a warm-up execution prior to benchmarking. Thus, it is anticipated that the performance difference will converge.

Algorithm	OpenCL backend	PTX backend
Bfs	Supported	Supported
Hotspot	Unsupported	Supported
Hotspot3d	Supported	Supported
Kmeans	Unsupported	Unsupported
Lud	Supported	Unsupported
Nw	Supported	Supported
Particlefilter	Supported	Unsupported
Pathfinder	Unsupported	Supported
Srad	Supported	Supported

Table 4.3: Compatibility of algorithms with OpenCL and PTX backends

4.2 Bugs Identified

As can be seen in Table 4.3, several algorithms are not supported to be run with the OpenCL backend. When the methods to be parallelized contain multiple conditions evaluation, there will be an error caused by the OpenCL backend, because in one of the lower-level compiler stages, several conditions are eliminated with the OpenCL backend, and the OpenCL backend is more complicated due to the uncontrolled flow graph to control the flow graph during the code generation of the OpenCL C code. Fortunately, this is not a problem for the PTX backends, since it is available to jump to the same basic block code from different ones in the generated assembly code. This issue is being repaired by the TornadoVM team.

In the case of *Matrix2DDouble* and *Parallel*, there will be thrown an exception generated by TornadoVM, so the algorithm Kmeans which requires the use of *Matrix2DDouble* and *Parallel* cannot be implemented by using TornadoVM. In addition, in some situations, there will be an error when using off-heap types such as *VectorInt* and *VectorFloat*, hence on-heap types such as *int[]* and *float[]* are still used in some algorithms.

4.3 Overhead

TornadoVM compiles Java bytecode into an intermediate representation that results in generating code that can be executed on various hardware devices. Since it compiles the code at runtime (Just-In-Time, JIT), there will be a startup overhead: the first execution of a program code may work slower due to the compilation time, potentially leading to negative initial performance results. Furthermore, there can be memory transfer overhead: When using devices like GPUs and FPGAs, data often needs to be transferred between the host (CPU) and the device (GPU or FPGA). The data transfer process can cause large overhead, particularly for applications that frequently read and write data from large data sets. In some cases, this overhead can be so significant that it cancels out the advantages of executing computations more rapidly on the device, highlighting the need for efficient data structure design and algorithm optimization to reduce communication costs. Moreover, the first time a device like a GPU is used, there can be an initialization overhead. Devices need a "warm-up" to reach peak performance, which can add overhead to short-running tasks or tasks that are run infrequently, because "warm-up" contains the time for initializing the diver and compiling the code.

Chapter 5

Reflection and Conclusion

5.1 Challenges

In this project, numerous challenges needed to be overcome. When I started, I had difficulties downloading and installing TornadoVM due to the configurations of my OpenCL drivers. The TornadoVM members were friendly and patient to help me solve the installation problem.

When I first started the project, it was also difficult for me to understand the OpenMP code. The introduction of OpenMP directives, pragmas in C/C++, was complicated for me. These pragmas have a syntax that is uncommon in basic C or C++ programming; they are used to tell the compiler to parallelize specific code blocks. One popular directive for parallelizing a for loop is *#pragma omp parallel for*. It took me a long time to make the conceptual leap from sequential to parallel execution to understand how these pragmas impact the program's flow. Furthermore, another crucial topic that was challenging to understand is data sharing. OpenMP offers a range of data-sharing features, including reduction, shared, private, and first-private, each of which has a distinct function during parallel processing. Understanding the distinction between shared and private variables, as well as knowing when to utilize each, is essential to guaranteeing accurate and effective execution of parallel code. As a consequence, I had to spend a lot of time in learning OpenMP in order to ensure that I was able to understand the OpenMP programmes that Rodinia Benchmark offered.

Although Java is the language I know best and excel in, it also took me a long time to be familiar with TornadoVM. TornadoVM has its special APIs to specify the parallel sections and task graphs. In order to utilize these APIs well, I had to make good use of time by understanding the unit tests and examples provided by TornadoVM.

Algorithm	Status
Bfs	Implemented
Hotspot	Implemented
Hotspot3d	Implemented
Kmeans	Implemented
Lud	Implemented
Nw	Implemented
Particlefilter	Implemented
Pathfinder	Implemented
Srad	Implemented
B+tree	Unimplemented
Backprop	Unimplemented
Cfd	Unimplemented
Heaertwall	Unimplemented
LavaMD	Unimplemented
Leukocyte	Unimplemented
Mummergpu	Unimplemented
Myocyte	Unimplemented
Nn	Unimplemented
Streamcluster	Unimplemented

Table 5.1: Status of Implemented and Unimplemented Algorithms

In addition, when I finished the code part and started to test the performance and compare it with the sequential Java and OpenMP. I found that the selection of the size of test data sets was important but also difficult. In order to find a balance between the memory and workload for a GPU, the test data sets were not supposed to be small, because this could not only make most cores become idle but also let the data transfer time become much longer than the actual computation time. Also, the size of the data set should not be too large, since it can cause unnecessary time waiting for the completion of the computation.

5.2 Future Plans

At present, as shown in Table 5.1, 9 out of 19 algorithms have been ported from the Rodinia benchmark to the TornadoVM. If I had more time on the project, I would have to implement the remaining 10 algorithms with TornadoVM, and also try the optimization for each algorithm. All the algorithms are commonly used in a wide range of fields, such as computer vision and natural language processing. I am also studying and interested in these fields. The more algorithms I implement, the more I learn about the fields. In addition, perhaps more TornadoVM's bugs can be found, or more useful features can be suggested during the process of implementing the algorithms. Furthermore, some implemented algorithms are not implemented by the use of heterogeneous computing in the Rodinia Benchmark, hence in the future, I plan to add the OpenMP's heterogeneous computing feature to the algorithms, and then compare the new implementations with the corresponding old versions and also the TornadoVM versions in order to find more about the efficiency of the OpenMP's heterogeneous computing. Moreover, in this project, when using TornadoVM, only fundamental annotation APIs such as *Parallel* and *Reduce* are used, so that would be more helpful for me to be more familiar with both TornadoVM and GPU programming if I make the use of Kernel API to get more access to GPUs' local memory and then implement the algorithms. After that, the performance difference can be evaluated by comparing the algorithms that use different techniques (non-Kernel API and Kernel API).

5.3 Conclusion

Through detailed benchmarking and analysis, I have shown that TornadoVM is a practical option for using the computational power of GPUs and FPGAs in Java applications. The performance improvements seen in multiple Rodinia benchmarks highlight TornadoVM's ability to greatly improve the performance of parallel computing tasks. This project has deepened my understanding of parallel programming and has improved my skills in Java and my ability to adapt to new technologies like TornadoVM. Although there were several challenges, every challenge was a chance to learn, pushing me to explore the subject further and sharpen my problem-solving skills. These experiences have given me a deeper appreciation for the complex interaction between software and hardware in high-performance computing.

I had an amazing time working on this project, and it was a great learning experience for me. Now, I have gained a deep understanding of parallel and heterogeneous computing by using both OpenMP and TornadoVM. In addition to the thoughts, I have also learned how the classic algorithms are implemented in practice. Additionally, I know how to report a bug to developers in a GitHub public repository efficiently by providing minimal test cases.

Overall, I am pleased to contribute to TornadoVM, an open-source project, and I look forward to seeing the continuous progress of TornadoVM.

Bibliography

[1] Deleli Mesay Adinew, Zhou Shijie, and Yongjian Liao. Spark performance optimization analysis in memory tuning on gc overhead for big data analytics. In *Proceedings of the 2019 8th International Conference on Networks, Communication and Computing*, pages 75–78, 2019.

[2] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, March 2009.

[3] Florin Blanaru, Athanasios Stratikopoulos, Juan Fumero, and Christos Kotselidis. Enabling pipeline parallelism in heterogeneous managed runtime environments via batch processing. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2022, page 58–71, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.

[5] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the rodinia benchmark suite with comparison to contemporary cmp workloads. In *IEEE International Symposium on Workload Characterization (IISWC'10)*, pages 1–11. IEEE, 2010.

[6] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James

Clarkson, and Christos Kotselidis. Dynamic Application Reconfiguration on Heterogeneous Hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '19. Association for Computing Machinery, 2019.

[7] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. Dynamic application reconfiguration on heterogeneous hardware. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, page 165–178, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Juan Fumero, György Rethy, Athanasios Stratikopoulos, Nikos Foutris, and Christos Kotselidis. Experiences in building a composable and functional api for runtime spir-v code generation. *arXiv preprint arXiv:2305.09493*, 2023.

[9] Zheming Jin. The rodinia benchmarks in sycl. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2021.

[10] John Kessenich, Boaz Ouriel, and Raun Krisch. Spir-v specification. *Khronos Group*, 3:17, 2018.

[11] Raphael Landaverde, Tiansheng Zhang, Ayse K. Coskun, and Martin Herbordt. An investigation of unified memory access performance in cuda. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sep. 2014.

[12] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495):eaam9744, 2020.

[13] Wenbin Lu, Shilei Tian, Tony Curtis, and Barbara Chapman. Extending openmp and openshmem for efficient heterogeneous computing. In *2022 IEEE/ACM Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM)*, pages 1–12, Nov 2022.

[14] Matt Martineau, Simon McIntosh-Smith, and Wayne Gaudin. Evaluating openmp 4.0's effectiveness as a heterogeneous parallel programming model. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 338–347, May 2016.

[15] Kazuaki Matsumura, Simon Garcia De Gonzalo, and Antonio J Peña. A symbolic emulator for shuffle synthesis on the nvidia ptx code. In *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, pages 110–121, 2023.

[16] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. Benchmarking opencl, openacc, openmp, and cuda: Programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, ARMS-CC '17, page 1–6, New York, NY, USA, 2017. Association for Computing Machinery.

[17] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009.

[18] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314, Aug 2009.

[19] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. Unified memory in cuda 6.0. a brief overview of related data access and transfer issues. *SBEL, Madison, WI, USA, Tech. Rep. TR-2014-09*, 2014.

[20] OpenMP Architecture Review Board. OpenMP 4.0 Specifications Released. https://www.openmp.org/uncategorised/openmp-4-0-specifications-released/, 2013. Accessed: April 11, 2024.

[21] Heinrich Martin Overhoff, S. Bußmann, and D. Sandkühler. A mathematical speedup prediction model for parallel vs. sequential programs. In Jos Vander Sloten, Pascal Verdonck, Marc Nyssen, and Jens Haueisen, editors, *4th European Conference of the International Federation for Medical and Biological Engineering*, pages 2556–2559, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[22] Josep M. Perez, Vicenç Beltran, Jesus Labarta, and Eduard Ayguadé. Improving the integration of task nesting and dependencies in openmp. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 809–818, May 2017.

[23] Luis Fernando Rodríguez Ramos, Jose Javier Diaz Garcia, Juan Jose Fernández Valdivia, Haresh Chulani, Carlos Colodro-Conde, and Jose Manuel Rodriguez Ramos. The use of cpu, gpu and fpga in real-time control of adaptive optics systems, 2015.

[24] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[25] Sangmin Seo, Gangwon Jo, and Jaejin Lee. Performance characterization of the nas parallel benchmarks in opencl. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 137–148, Nov 2011.

[26] Rishi Sharma, Shreyansh Kulshreshtha, and Manas Thakur. Can we run in parallel? automating loop parallelization for tornadovm, 2022.

[27] Athanasios Stratikopoulos, Florin Blanaru, Juan Fumero, Maria Xekalaki, Orion Papadakis, and Christos Kotselidis. Cross-language interoperability of heterogeneous code. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming*, pages 17–21, 2023.

[28] TornadoVM Project. TornadoVM Documentation. https://tornadovm.readthedocs.io/en/latest/programming.html. Accessed: April 11, 2024.

[29] TornadoVM Project. TornadoVM Documentation. https://tornadovm.readthedocs.io/en/latest/. Accessed: April 11, 2024.

[30] TornadoVM Project. TornadoVM Documentation. https://tornadovm.readthedocs.io/en/latest/, 2024. Accessed: April 11, 2024.

[31] Liu Yuan, Yinggang Dong, Yangyang Li, Rui Zhang, and Haiyong Xie. A task parallel programming framework based on heterogeneous computing platforms. In Sabu M. Thampi, Ljiljana Trajkovic, Sushmita Mitra, P. Nagabhushan, El-Sayed M. El-Alfy, Zoran Bojkovic, and Deepak Mishra, editors, *Intelligent Systems, Technologies and Applications*, pages 169–184, Singapore, 2020. Springer Singapore.

[32] YUHC. GPU-Rodinia: Benchmark Suite for Heterogeneous Computing, Accessed 2024.