Monash University

Electrical Engineering

Final Year Project (ECE4093)

Design Document

Author:
James Anastasiou
ID: 23438940

Supervisor:
Dr. David Boland

September 19, 2016

Contents

1 Literature Review

1.1 Introduction

Optimisation and computational efficiency are two pillars of good program design, much research has been undertaken in the search of improving performance and extracting hardware maximum efficiency. Although the current literature covers an extensive range of research, this review seeks to focus primarily on the topics of automatic vectorisation, alternative hardware (GPU/FPGA) performance in parallel contexts, and finally the utilisation of Static Analysis and Profiling to assist in the process of identifying potential optimisation in the massively parallel computation paradigm. Individually these are all large topics of research, and as such this review will be focusing primarily on computational performance, rather than power consumption or algorithm efficiency. The purpose of this review is to provide a contextualisation around my final year project, in order to identify potential challenges in the problem space, as elucidated by prior research.

1.2 Automatic Vectorisation

Automatic vectorisation is a tool employed by many compiler designers in order to generate ASM which utilises specialised hardware level vector instructions. Same Instruction Multiple Data (SIMD) instructions seek to process multiple elements of a dataset simultaneously, utilising multiple processing units in order to achieve data level parallelism. Roger Espasa and Mateo Valero [1] explore the potential benefits of Data-Level Parallelism by investigating computer architectures to utilise both Instruction Level and Data Level parallel constructs. Within their research they identify and develop an architecture to utilise SIMD instructions to leverage the performance benefits, clearly demonstrating the performance improvement this parallel strategy provides. These techniques have since been further developed by Compiler designers, with the LLVM/Clang team employing two forms of automatic vectorisation within their optimising C compiler [2]. The Clang team focused on developing Loop Level Vectorisation and Super Word Level Vectorisation in order to leverage parallelisation in the target architecture. Their automatic loop vectoriser is capable of providing an increase in processing speed of 3 times, when tuned specifically for the Intel Core-i7 AVX instruction set. This is a clear demonstration of the benefits already being seen by optimising compilers manipulating sequential programs into those which leverage the power of parallel computation. The LLVM optimiser however has some flaws which are identified by Yulei Sui, Xiaokang Fan, Hao Zhou, and Jingling Xue who developed Loop-oriented array and field sensitive Pointer Analysis (LPA) in order to combat the suboptimal performance of the LLVM auto-vectoriser [3]. The authors identified alias analysis as being a potential cause of the LLVM compiler missing optimisation opportunities. They created an analysis framework around both flow-insensitive and flow-sensitive pointer construct. By hooking into the LLVMs partial SSA form they exploited the reduced semantic complexity to algorithmically generate superior memory patterns, and by extension developed a superior loop vectoriser, with performance up to 10% better than the original LLVM. This research again demonstrates the performance opportunities created by parallel computation, however they are all fundamentally limited by the CPUs capacity to perform SIMD operations. Most CPU architectures have at most 8 complex computing cores, limiting the maximum potential throughput, however GPU architectures have the capacity for massively parallel computation. The typical design inherently leverage SIMD concepts containing thousands of simple computational cores with high memory bandwidth. Thus whilst automatic vectorisation has substantial performance potential, most current literature is focused on continuing to improve computational efficiency of host bound programs, rather than looking towards exploiting the massively parallel computational power of modern GPUs.

1.3 Parallelism

General Purpose GPU programming seeks to exploit the parallel performance characteristics of the GPU architecture; identification and development of algorithms which leverage this design pattern can provide substantial computational speedups. The authors of [4] explore the fundamentals of GPGPU programming, and the CUDA architecture. Inherent within the CUDA architecture is hybrid CPU-GPU programming to produce the most efficient solution. In the example described the authors extract maximum parallelism from Matrix operations, and leave complex control flow to the CPU, thus developing a solution which maximises the performance of the individual components of the Hybrid Host-Device model. The proposed solution utilises parallel primitives such as the Reduction, which exploits parallelism by utilising a summation tree. This requires that the data and binary operator form a semigroup (The set of data must be closed under an associative binary operator). In their example the set is of integers, and the operator is addition, which holds these properties. The concept of parallel primitive operations is further expanded by [?] which provides terminology to describe parallel patterns for both computation and communication. Although this research focuses on parallelism within FPGAs the primitive operations are examples of fundamentally parallel operations, exposing high levels of optimisation potential through SIMD data-paths. The authors look to extract performance from their FPGA implementation of potentially parallelisable algorithms; their key focus is on the Parzen window technique of Gaussian PDF estimation, as well as K-Means clustering. From their research they develop a concept of pattern-based algorithm decomposition; as a means of exposing potential parallelism within a codebase to individuals with little or no experience with highly parallel code. The authors describe a limitation in the existing development framework, where FPGA based systems are developed on by only highly specialised and skilled developers. This concept is explored by the authors of [5] in which they describe the limitation upon many programmers is the lack of a breadth of libraries which exploit the benefits of GPGPU programming. Within this paper they present the problem of fragmentation within the GPU programming space, with competing standards and APIs resulting in specialists rolling their own solutions to many problems, resulting in minimal code re-use. The current solution to the code reuse problem is the introduction of parallel algorithms within the C++ STL as well as CUDA based libraries such as CuBLAS and Thrust, which aim to provide programmers with a foundation from which to build larger programs, without being forced into having a full understanding of programming on a GPU. Given the power and performance characteristics of GPUs there is a great incentive to move computationally expensive algorithms onto these devices, currently however there are many roadblocks preventing individuals and organisations from exploiting the potential improvements within their codebase, the high barrier to entry can be difficult to overcome, especially with the limited capabilities of identifying how beneficial any redevelopment may actually be.

1.4 Static Analysis

Static Analysis is the analysis of the original source statements of a program, it provides a method by which the semantics of a program may be identified. Typically, Static Analysis is utilised in order to identify bugs within a codebase [6], there is a litany of literature which describes a variety of processes through which one can employ Static Analysis to search out and find bugs [7] [8]. These tools rely on parsing the source of a program and identifying the anti-patterns within, alerting developers to their potential mistakes. Additionally, many static analysers provide complexity statistics about the analysed source; it is often the intention of static analysers to provide the developer with a summary of information about their codebase which facilitates further investigation and development. Although there has been a large amount of research into static analysis for the purpose of detecting and eliminating bugs, there is a lack of research into the topic of utilising static analysis for the purpose of performance improvements. Static analysis as a tool is rarely used to facilitate optimisation improvements within a codebase; programmers often utilise profiling in order to determine where to invest development time focused on optimisation. Clearly this presents a divide, where programmers often utilise Static Analysis and Profiling in a competing demands structure for developer time. Within the NVidia High-Productivity CUDA Programming presentation [9] they recommend the utilisation of a process called APOD Assess-Parallelise-Optimise-Deploy. Within the Assessment and Optimisation phase of the process they recommend utilising profilers in order to make determinations about what aspect of the code requires further development. The weakness of profiling however is that it is often very time consuming, additionally it requires an individual developer have an understanding of how to potentially translate sequential serial algorithms into their massively parallel equivalents.

2 Introduction

2.1 Purpose and Scope

This document aims to provide a framework which describes the process by which my final year project will be completed, and by extension the re-evaluation of goals initially set forth in the requirements specification. In order to best achieve this aim, this document includes:

- Project Description
- Re-evaluation of the inital goals
- Project Status Overview
- Current Position Review
- Timeline to Expected Completion

3 Project Description

My final year project aims to create a set of static code/compiler analytics tools to help determine which algorithms within a codebase may be easily parallelized. The specific intention it to provide users with a report describing which parts of their codebase may see a potential speed-up from redeveloping them as CUDA (GPU) kernels. In order to achieve this both benchmarking and theoretical analysis is required, as well as static analysis of the C description of the algorithm itself.

3.1 Compiler Analytics

Given a C description of an algorithm, a report is to be generated, highlighting areas within the code that may see a potential speed-up based on matching to known GPU performant patterns. In order to achieve this, a static analysis methodology was invoked. Direct source code analysis is incredibly difficult, and suffers from a number of drawbacks, including the difficulty to parse C and (notoriously difficult) C++. Considering the time constraints involved it was decided that the analysis tool would hook into the Clang tooling library, which provides access to the Clang Abstract Syntax Tree, giving a semi-syntax invariant canvas from which to identify relevant parallel patterns.

Given the difficulty of setting up a generic build environment for the Clang tooling, I decided to fork a project named OCLint, a C, C++ and Obj-C static analysis tool that I had worked with earlier. This tool provides a framework around the Clang tooling, however most importantly it provides sophisticated build scripts which work on a variety of operating systems and distributions.

3.1.1 OCLint Modification

Forking the OCLint project, which is licensed under a modified BSD license has saved significant time and effort from being wasted developing a generic build system around the Clang tooling. The OCLint software provides a direct method for interacting with the Clang AST by revealing the entire Clang library from within its rule system. This increased flexibility has in turn allowed for more time to be spent developing methods to identify parallel patterns within the generated AST. All changes to the OCLint software are unrelated to its original intention and design, and as such no pull requests are likely to be lodged, and no modifications I have written, or will write are likely to move upstream. As such I will be substantially re-engineering the OCLint software into a new tool named the C Algorithm Parallelisation Analyser (CAPA).

3.2 GPU Benchmarking

In order to best provide theoretical performance improvements of algorithms within a codebase, an analysis of the current hardware available is significantly important. As such rather than just provide purely theoretical numbers, part of this project involves developing a simple set of GPU benchmarks which seek to show performance metrics for the identified patterns within the code analyser. This in effect means that reports generated by the analytics tool may contain specific information pertaining to the hardware availble on the current build and test system. In order to achieve the best outcome, CUDA was decided on as the framework for development.

3.2.1 Benchmarks

GPU's are exceptionally good at high throughput calculations, one particular example is SIMD, meaning *Same Instruction Multiple Data.* The performance of GPU's and the algorithms they are particularly useful for is well under continual research, however general problem classes that GPU's are able to solve efficiently are well understood. These problem sets include algorithms that can be described by any of the following:

- Map
- Fold/Reduce
- Scan/Prefix Sum
- Matrix Operations
- Depth first Graph Traversal

The actual speed improvements derived from redeveloping serial code to take advantage of the massively parallel compute power of a GPU differs between each of these operations, however many serial algorithms have equivalent or more performant alternate parallel implementations. As such this project involves developing a small set of benchmarks for GPU's that determine their performance in each of these categories.

3.2.2 Deliberately Difficult Benchmarks

Whilst GPU's can be incredibly powerful, deliberate design decisions in the architecture of the GPU allow for significantly worse performance than one may expect from algorithms that should seemingly perform well on a GPU. There are a number of situations that may arise which decrease GPU computational efficiency including:

- Dispersed Global Memory Reads
- Inefficient use of Shared Memory
- Divergent threads (large control overhead)
- Data dependencies

Some serial algorithms may look easily parallelizable, however they may contain one, or many of these potential inefficenciesm, and then as such they will not perform as well as one might expect on a GPU. It is therefore important in analysing potential speedups to see the performance of algorithms which *appear* to be readily parallelizable, yet in practice do not perform to expectation. As such part of the project is to develop a set of benchmarks which appear to be efficient on a GPU, yet actually perform poorly.

3.2.3 CUDA

CUDA is Nvidia's proprietry library and toolchain for developing parallel software. There are 2 main frameworks in the GPU programming space, CUDA and OpenCL. Whilst OpenCL is a FOSS platform, the development tools are severly lacking in comparison to the CUDA toolkit, and as such it was an easy decision to follow through with the CUDA. This however has limited the performance metrics to only comparisons involving CUDA enabled graphics cards.

4 Re-evaluation of Initial Goals

In order to consolidate the current position of this project, and to best identify a pathway to completion, it's important to take another look at the initial requirements set forth in the requirements analysis. Within the requirements analysis there were a set of goals that defined the project and from which all development so far has stemmed; by looking at these requirements and evaluating the current trajectory of the project a detailed description of what is required and how it will be achieved can be compiled.

The requirements analysis broke the project down into 3 major components:

- GPU Benchmark Development
- Algorithm Analytics Development
- Optimisation Analytics Development

which then allows the breakdown of what so far has happened in the project.

4.1 GPU Benchmark Development

As described earlier, the importance of developing working GPU benchmarking code for known problem classes allows for better analytics and reporting in the serial algorithm analysis portions. This therefore is a key aspect of satisfactorily completing the project. The GPU benchmark development has a number of requirements that describe what the project necessitates.

4.1.1 Requirements

4.1.1.1 [FR.003] The program shall run developed benchmark algorithms to further analytical information.

This requirement relates directly to the overall aim of the project, which is described in the requirements just proceeding this. As the project currently stands there is only one working benchmark developed, it is a best case memory bandwidth test which requests on device memory, fills it with junk host memory, and requests theen now junk device memory be copied back to the host. This test aims to determine the peak memory bandwidth for the device, which can then be used to determine absolute optimal performance of any subsequent alogrithm.

In order to complete the project more benchmarks need to be written, to specifically cover algorithm optimisation cases identified in the serial analytics portion of the project. Necessary benchmarks are as follows:

- Peak Memory Bandwidth
- Peak Map
- Peak Fold/Reduce
- Peak Scan
- Peak Matrix Multiplication
- Peak Depth first Graph Traversal

Upon completion of these benchmarks this requirement will have been completed, interfacing and using the results of these benchmarks are a seperate requirement.

4.1.1.2 [OA.001] The program shall run custom benchmark algorithms to identify GPU performance.

This requirement describes that the benchmarking algorithms must be utilised to identify GPU performance. In order to satisfy this requirement my intention is to produce benchmarking code for GPU performance in problem sets that are both known to be performant on a GPU as well as benchmarks that may naively appear to be performant, yet further inspection demonstrates that they are not in fact performant. This is a rather large task in and of itself, and has the potential to be an entire FYP on its own, as such significant compromises must be undertaken. In this particular case only 2 non-performant algorithms will be developed, they are:

- Recursive Dependent Matrix Calculations
- Highly Divergent Hashmap Traversal

These problem sets seem to be readily parallelizable, however they in fact exacerbate the weaknesses of the general NVidia GPU architecture (and potentially other co-processor architectures I have not worked with). The results of these benchmarks contextualise the information derived from the code analytics portion, giving rise to more useful metrics defining best and worst case performance.

4.1.1.3 [OA.002] The program shall work on all CUDA devices.

After careful consideration this requirement has been relaxed as it is far too strict. When writing the requirements analysis I was not as familiar with the CUDA toolchain as I am at this point of my project, and as such it is now apparent that writing Compute Capability agnostic code is a very difficult feat. In order to best satisfy the other requirements of this project I shall be limiting benchmarking code to work on CUDA capable devices of Compute Capability 3.5 and above. This compute capability was chosen as the capabilities of CUDA Cards differ significantly pre and post Compute Capability 3. This revision of the initial requirement shall save significant time and effort from being wasted in localisation and highly technical activities that benefit the overall project very little.

4.1.1.4 [OA.003] The program shall provide comparative CPU performance metrics.

This requirement remains as it was initially written, there is no reason why the project should not continue to include a comparison between parallel GPU benchmarks and the relevant serial CPU implementation. This information will be used within the analytics framework.

4.1.1.5 [OA.004] The program shall provide a number of different problem class benchmark algorithms.

This requirement was covered and expanded under sections 4.1.1.1 and 4.1.1.2

4.1.1.6 [OA.005] The program shall provide theoretical performance metrics given a known problem class

In order to provide the best possible analytics for the serial code analysis, theoretical parallel performance must be understood, so that in situations where a GPU is not present, that relevant calculations may be undertaken to provide an estimate on the anticipated performance. Naturally actual performance and theoretical performance differ significantly for a variety of reasons, however the fundamental considerations involved in algorithm analysis can be known or reasonably estimated from which theoretical performance metrics may be provided.

4.1.1.7 [OA.006] The program shall include known FPGA performance metrics given a known problem class

This requirement seems unlikely to be filled by the project as it currently stands. This is mainly due to the change in direction of the project since the submission of the requirements document. The original intention of the project was to provide a benchmarking suite for CPU, GPU and FPGA algorithms. Since then the project has become primrarly about the compiler analytics side, with less emphasis on the relative performances and tradeoffs between the different computing architectures. Due to the size of the project, and the direction it began to proceed, I have elected to not satisfy this requirement, and to remove it from what this project intends to achieve. Removing this requirement has provided more time for solving problems more relevant to the current and final form of this project.

4.2 Algorithm Analytics Development

As described earlier in the design document, this is the crux of my final year project. The core objective is to produce software that analyses a C algorithm description and identifies whether the algorithm described may see some benefit from being parallelised. This is extended by the other aspects of this project which in turn provide extra metrics for comparison between CPU and GPU performance. The stretch goal is to provide analysis of an existing C codebase which may contain a variety of potentially parallel algorithms within. As static code analysis is a rather large task to undertake certain decisions have been made to ensure that this project may be completed, and some of these are reflected wtihin the requirements.

4.2.1 Requirements

4.2.1.1 [FR.001] The program shall analyse an algorithm and produce optimisation analysis

This is the key requirement of the entire project. This requirement can not be compromised on, and as such all other requirements must relate to ensuring this requirement is met. Analysis of the algorithm is defined as static code analysis, and optimisation analysis is defined as the recognition of potentially parallelizable algorithms within the code. This is achieved through integrating with the Clang tooling, utilising the AST to perform the static analysis. The static analysis itself is primrarly concerned with matching known parallel patterns through the AST Matcher library.

As the project currently stands, this requirement will be fulfilled, there is already accurate matching for Map, Reduce and Fold operations. By the time this document is submitted it's expected that 2D matrix operations and depth first graph traversals will also be identified. Currently the most difficult aspect of this requirement is generating specific, yet generic matchers to identify patterns that have been programmed in a *reasonable* manner.

The actual optimisation analysis will be a report generated identifying the tagged regions of the codebase, along with performance metrics. The aim is to utilise aggressive compile time optimisation strategies within Clang to identify as much information about the tagged section as possible. This allows the report to provide the most accurate information it can about potential speedups.

The fundamental intention is to provide an ordered list of potential improvements from within the code, similar to runtime profiling, however attacking the problem from the opposite perspective.

4.2.1.2 [FR.002] The program shall produce theoretical performance metrics of developed algorithms

This relates back to the discussion about theoretical performance metrics in 4.1.1.6. This is the extension of that requirement. Where actual benchmark information is not available, the analyser should provide theoretical performance improvements as described in the literature.

4.2.1.3 [FR.004] The source code shall be released under a FOSS license

All source code will be released under a Modified BSD license.

4.2.1.4 [CA.001] Integrate with the Clang tooling to analyse custom written C code

This requirement has been completely satisfied, a stable build system has been forked from an existing open source project providing the scaffolding around which the entirity of the analyser aspect of this project has been built. All source code analysis is done through the AST Matcher API, and currently is successfully identifying test cases of parallel patterns.

4.2.1.5 [CA.002] Identify simple parallel patterns within analysed code

This requirement has been completely satisfied, the three simple parallel patterns for which significant improvements can be found in GPU implementations are:

- Map Operations
- Reductions
- Scans/Prefix Sum

All three of these simple patterns can be successfully identified within test code. Unit tests are yet to be written however the performance on known specially written problem sets has been highly accurate.

4.2.1.6 [CA.003] Identify medium complexity parallel patterns within analysed code

Medium complexity parallel patterns are considered to be patterns within serial code that are clearly parallelizable yet are difficult to identify in a generic sense. This primrarly means the identification and tagging of 2D Matrix operations. Matrix operations are considered a medium complexity pattern due to the variety of ways in which they may be implemented. As this aspect of the project is primrarly pattern matching and feature detection, identifying the litany of differnet ways a matrix multiplier may be implemented is a significant task. As such writing an accurate, yet generic Matcher and Callback handler for this is a sizeable task. Little work has been done so far in truly analysing the full scale of the complexity of this issue, it may turn out to be significantly more difficult that I anticipate.

4.2.1.7 [CA.004] Identify non-trivial parallel patterns within analysed code

Non-trivial parallel patterns mainly defines a broad set of problems that are not clearly parallelizable. This includes algorithms such as breadth first search and merge sort. This aspect of the project relies on the potential to recommend algorithm change to gain the greatest benefit from any parallelization. In effect this requirement is just as much about identifying non-trivial parallel patterns as it is about identifying algorithms which may be replaced by a highly parallelizable alternative. Recommending algorithm change opens up one of the largest potential optimisations of any code base; choosing a better algorithm, this analyser aims to provide that capability.

4.2.1.8 [CA.005] Provide Theoretical improvement information

Has been covered extensively here 4.2.1.2 and here 4.1.1.6.

4.2.1.9 [CA.006] Provide more accurate theoretical improvement analysis using additional user specified information

Expanding on 4.2.1.2 and 4.1.1.6, if the user decides to provide additional information, then the theoretical performance metrics will take this information into consideration when calculating potential performance improvements.

4.2.1.10 [CA.007] Analyse general C code algorithm descriptions

The current form of the project is capable of working on any Clang compilable C codebase. As such the analyser is already capable of analysing general C code algorithm descriptions from a functional point of view. Relating to actual performance, this has not yet been tested as not enough development has been done to see the benefit of any such test. The current strategy is to continue developing matchers for the medium complexity patterns before beginning to test on an actual codebase.

4.2.1.11 [CA.008] Analyse existing codebases within tagged regions

In order to analyse an existing codebase most effectively, it's useful to identify regions of interest. By only searching for optimisations within the tagged regions, or alternatively, pre-emptively rejecting potential matches based on their location within the source code, the analyser can provide more relevant information to the user. Providing more power to the user to decide what and where to look for optimisations aids in profiling driven design, whereby blind optimisation is avoided in favour of a more rigorous approach.

In order to satisfactorily identify tagged regions, a dual pass will have to be undertaken, once over the source code to identify tagged regions, which will be identified by a particular comment line, and a second pass over the AST whereby the actual analysis occurs.

4.3 Optimisation Analytics Development

Whilst there are no specific requirements relating to this particular component, this is the unifying feature of the entire project. Combining static code analysis with benchmarks to provide a comprehensive optimisation report, without running a profiler allows for fast identification of potential improvements within a codebase of any size. That is the key intention and aim of this project.

5 Project Status

This section provides a summary of what has been completed, and not completed so far within this FYP.

5.1 Completed Tasks

- Stable package and distribution setup
- Successful identification of simple parallel patterns
- Initial scaffolding of GPU benchmarking code
- Basic GPU peak memory performance benchmark
- Scaffolding of performance metric reporting
- Capable of analysing arbitrarily large C codebases

5.2 Incomplete Tasks

- Identification of more complex parallel patterns
- Identification of complex parallel patterns
- Design and coding of GPU benchmarking suite
- Implementation of theoretical performance analyser
- Completion of Poster
- Completion of presentation video
- Generation of documentation

6 Current Position Review

This section aims to look forward at what needs to be achieved and how what has been done so far sets up the rest of the project for completion.

6.1 Incomplete Tasks

6.1.1 Identification of more complex parallel patterns

At the current point within the project, basic parellel patterns are recognisable within given test code. These patterns are Map operations, Reductions and Scan's. Intermediate complexity matrix operations are considered to be a selection of 2D matrix operations. The most optimisable of these is matrix addition, and subtraction, however multiplication is a readily parallelizable task which must be recognised. The current strategy of identifying parallel patterns by utilising the Clang libtooling matcher library has sufficient flexibility to identify these more complex patterns. Given the success of identifying the basic parallel algorithms, it is expected that this will be achieveable within the timeframe set for the project.

6.1.2 Identification of complex parallel patterns

Complex parallel patterns primarily refers to identification of breadth first graph traversal, however it also is in reference to identification of parallel inefficient algorithms and recommending a more efficient parallel implementation, such as depth first search, or certain sorting algorithms. These are significantly more difficult to identify given the large variety of ways in which these algorithms may be written. This introduces significant overhead to both the matcher and the callback, and requires significant effort to develop a generic setup for even one of these problem classes. As such these are considered the most complex patterns for identification, and as such require the largest time investment to bring them up to speed.

6.1.3 Deisgn and coding of GPU benchmarking suite

The process has already begun for the production of the GPU benchmarking suite. An object heirachy for generic reporting of statistics has been created, and the initial memory benchmark has been implemented, with performances roughly matching theoretical expectations on the test hardware. The benchmarking suite will include problem classes that the algorithm analyser will be targeting, in order to provide the most relevant information for theoretical performance estimation. As it currently stands, this means the production of basic Map, Reduce and Scan kernels, as well as a basic streaming matrix arithmetic kernels, and potentially an implementation of an efficient breadth first search, if time permits.

6.1.4 Implementation of theoretical performance analyser

This is the cumulative goal of the entire FYP, in order to best achieve this goal a combination of the source code analyser and the GPU benchmarker is required; by definition this task is currently incomplete as the two major components that comprise this task are also incomplete. In order to achieve this goal compromises must be made between the depth in whcih both aspects of this project are pursued. In order to provide the best possible outcome it is important that good decision making is undertaken in establishing what aspects of the code analyser and benchmarking suite receive the most attention. The key aim of this task is to provide a report detailing parts of the source code that may see improvements from parallelization. This key aim is enhanced by providing further detail about what sort of potential improvements may be seen in a parallelized implementation. Due to the nature of this relationship it is clear that the code analyser is the *master* and the benchmarks are the *slave* components of the project.

6.1.5 Presentation Related Items

These naturally rely on aspects of the project being either completed, or close to completion and as such cannot be completed until far later in the project timeline.

6.2 Completed Tasks

6.2.1 Identification of Simple Parallel Patterns

This is the initial requirement of the code analyser aspect of this project. Currently simple parallel patterns such as map operations have been successfully completed. This is done utilising the Clang libtooling library, using ASTMatchers and the resulting callback. The current AST Matcher for a map operation is:

```
auto MapMatcher =
forStmt(
    hasLoopInit(anyOf(
        declStmt(hasSingleDecl(varDecl(hasInitializer(
            integerLiteral(anything()))).bind("InitVar"))),
        binaryOperator(
            hasOperatorName("="),
            hasLHS(declRefExpr(to(varDecl(hasType(
                isInteger())).bind("InitVar"))))))),
    hasCondition(binaryOperator(hasRHS(hasDescendant(
        declRefExpr().bind("var"))))),
    hasIncrement(unaryOperator(
        hasOperatorName("++"),
        hasUnaryOperand(declRefExpr(to(varDecl(hasType(isInteger()))
            .bind("IncVar")))))),
    hasBody(hasDescendant(binaryOperator(
        hasOperatorName("="),
        hasLHS(arraySubscriptExpr(
            hasBase(implicitCastExpr(hasSourceExpression(declRefExpr(to(
                varDecl().bind("OutBase")))))),
            hasIndex(hasDescendant(declRefExpr(to(varDecl(hasType(
                isInteger())).bind("OutIndex"))))))),
        hasRHS(hasDescendant(arraySubscriptExpr(hasBase(implicitCastExpr(
                hasSourceExpression(declRefExpr(to(varDecl().bind("InBase")))))),
                    hasIndex(hasDescendant(declRefExpr(to(varDecl(hasType(
                        isInteger())).bind("InIndex")))))))),
        unless(hasDescendant(arraySubscriptExpr(hasDescendant(binaryOperator())))))
        .bind("Assign")))).bind("Map");


addMatcher(MapMatcher);
```

Which identifies the key characteristics of map operations, that being; for loops which increment a loop counter by one with an array assignment which is calculated as a one to one mapping from an input array.

Other matchers for simple parallel patterns are similar in structure to the map matcher, as they are all operations over 1 dimensional arrays. Higher dimension data structures require a different matcher structure to account for their increased complexity.

6.2.2 Inital Scaffolding of GPU Benchmarking Code

The current scaffolding of the GPU Benchmarking code provides a benchmarking class which defines the interface by which other aspects of this project must interface with the benchmarking code. Currently the interface is:

```
class BenchmarkObj
{
protected:
    cudaDeviceProp mDeviceInfo;
    BenchmarkConfig mConfig;

public:
    std::string DeviceName();

    BenchmarkObj();
    BenchmarkObj(int device);
    BenchmarkObj(std::map<std::string, std::string> config);

    std::string Report();

    // Trivial
    void BenchMemBandwidth();

    // Simple
    void BenchMap();
    void BenchReduce();
    void BenchScan();

    // Intermediate
    void BenchMatrixMultiplySimple();
    void BenchMatrixAddition();

    // Complex
    void BenchMatrixMultiplyStreaming();
    void BenchBreadthFirstTraversal();
    void BenchDepthFirstTraversal();
};
```

The current interface design provides a clean API for handling all benchmark requests, as well as providing library users the ability to change configuration information between benchmark operations. The key benefit however is within the restricted interaction beyond initial configuration of the benchmark suite, which provides programs utilising the API a simpler interface from which to work. Internal code within the library is up for change, however as the external API aims to remain constant this should not impact potential library users.

Additionally in order to increase code re-use I've attempted to implement the concept of Type-Classes into CUDA C++. This allows for the single case of higher order functions such as Map, Reduce and Scan. This implementation utilises template meta-programming and templated type alias's to produce type safe higher order polymorphism within CUDA compute kernels.

```
template <typename T>
using uCat = T(*)(T);

template <typename T>
using mCat = T(*)(T, T);

__device__ int square(int a)
{
    return a * a;
}

__device__ int mult(int a, int b)
{
    return a * b;
}

template <typename T>
__device__ T mult(T a, T b)
{
    return a * b;
}

template <typename T, mCat<T> func>
__global__ void biMapKernel(T *a, T *b, T *c, size_t size)
{
    size_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < size)
        c[i] = func(a[i], b[i]);
}

template <typename T, uCat<T> func>
__global__ void MapKernel(T *a, T *c, size_t size)
{
    size_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < size)
        c[i] = func(a[i]);
}
```

This code segment demonstrates typeclass instances for operations over both *Functors* and *BiFunctors*, the *Functor* typeclass is the alias uCat, a function which accepts a single input of type T and returns a single output of type T. The second typeclass definition is the *mCat* defition, defining the monoidal typeclass requirement of *mConCat*. This is any function that accepts 2 inputs of type T and returns an output of type T. These typeclasses are then used to validate the parametric polymorphism of the *biMapKernel*, which implements a polymorphic bimap operation, the key aspect of the *bifunctor* typeclass, as well as the common map kernel, which is the single requirement of the *functor* typeclass. This parametric polymorphism provides an easy to use, clear and concise framework from which to build larger GPU benchmarking routines, by utilising the higher order nature of the GPU *primatives*. To summarise the category theory, this essentially provides a type safe way to write less code.

6.2.3 Basic GPU Peak Memory Performance Benchmark

The current peak memory performance benchmark is an incredibly simple memory test, the base benchmark is merely allocates memory on the device, copies memory from the host to the device, performs a no-op on the device and copies the memory back from the device to the host. This is all timed in order to calculate a peak theoretical memory bandwidth. The benchmark was written before the current version of the benchmark object existed, and as such it does not have any dynamic reponse to changing structure. It is merely a proof of concept for the peak memory performance benchmark.

```
void BenchmarkObj::BenchMemBandwidth()
{
```

```
size_t size = 10000000;
uint8_t *h_a, *h_b;
uint8_t *d_a, *d_b;
timespec ts, te;

h_a = (uint8_t*) malloc(sizeof(uint8_t) * size);
h_b = (uint8_t*) malloc(sizeof(uint8_t) * size);

vectorFill(h_a, size);
vectorFill(h_b, size);

cudaMalloc((void **) &d_a, sizeof(uint8_t) * size);
cudaMalloc((void **) &d_b, sizeof(uint8_t) * size);

for (int i = 0; i < 10; ++i)
{
    clock_gettime(CLOCK_REALTIME, &ts);
    cudaMemcpy(d_a, h_a, sizeof(uint8_t) * size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, sizeof(uint8_t) * size, cudaMemcpyHostToDevice);
    clock_gettime(CLOCK_REALTIME, &te);

    timespec diff = diffTime(ts, te);
    std::cout << diff.tv_sec << "s " << diff.tv_nsec << "ns "
              << ((float) size / diff.tv_nsec) << "GB/s" << std::endl;
}
std::cout << std::endl << "INTERNAL" << std::endl;
for (int i = 0; i < 10; ++i)
{
    clock_gettime(CLOCK_REALTIME, &ts);
    cudaMemcpy(d_a, d_b, sizeof(uint8_t) * size, cudaMemcpyDeviceToDevice);
    cudaMemcpy(d_b, d_a, sizeof(uint8_t) * size, cudaMemcpyDeviceToDevice);
    clock_gettime(CLOCK_REALTIME, &te);

    timespec diff = diffTime(ts, te);
    std::cout << diff.tv_sec << "s " << diff.tv_nsec << "ns "
              << ((float) size / diff.tv_nsec) << "GB/s" << std::endl;
}
cudaFree(d_a);
cudaFree(d_b);
free(h_a);
free(h_b);
```

```
}
```

Even though this code does not interface with the GPU benchmark object it is a trivial transformation to effect that change and bring this inital benchmark into a useable form.

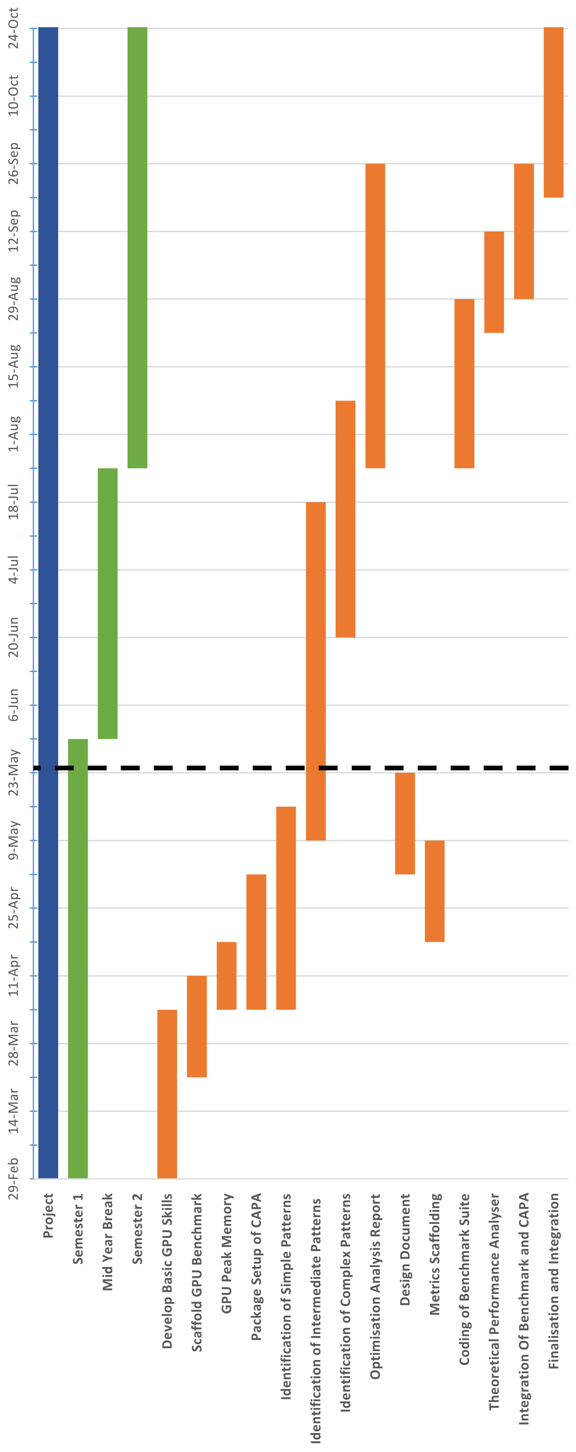
6.2.4 Scaffolding of Performance Metric Reporting

By utilising the OCLint tool, the rule and reporter interface has already been written. There are a few alterations that still need to be undertaken, as the nature of the reports is somewhat different between the original intention of OCLint, and what I am doing, however the overall scaffolding and relationship between analysis and report is finished, and quite robust.

6.2.5 Capable of Analysing Arbitrarily Large C Codebases

As the static code analyser uses the Clang libtooling, if the codebase being run on is capable of being built by clang, then it is possible for the codebase to be analysed CAPA. This is due to the heavy lifting being done by the clang compiler, and the tool hooking in only at the AST stage per compilation unit.

7 Timeline


29-Feb
14-Mar
28-Mar
11-Apr
25-Apr
9-May
23-May
6-Jun
20-Jun
4-Jul
18-Jul
1-Aug
15-Aug
29-Aug
12-Sep
26-Sep
10-Oct
24-Oct
Project
Semester 1
Mid Year Break
Semester 2
Develop Basic GPU Skills
Scaffold GPU Benchmark
GPU Peak Memory
Package Setup of CAPA
Identification of Simple Patterns
Identification of Intermediate Patterns
Identification of Complex Patterns
Optimisation Analysis Report
Design Document
Metrics Scaffolding
Coding of Benchmark Suite
Theoretical Performance Analyser
Integration Of Benchmark and CAPA
Finalisation and Integration

8 Appendix

All code can be found at `http://github.com/jhana1/CAPA`.

References

[1] R. Espasa and M. Valero, “Exploiting instruction-and data-level parallelism,” *IEEE micro*, vol. 17, no. 5, pp. 20–27, 1997.

[2] “Auto-vectorization in llvm.” [Online]. Available: http://llvm.org/docs/Vectorizers.html

[3] Y. Sui, X. Fan, H. Zhou, and J. Xue, “Loop-oriented array-and field-sensitive pointer analysis for automatic simd vectorization,” in *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems.* ACM, 2016, pp. 41–51.

[4] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis, “A new era in scientific computing: Domain decomposition methods in hybrid cpu–gpu architectures,” *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 13, pp. 1490–1508, 2011.

[5] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, “Scan primitives for gpu computing,” in *Graphics hardware*, vol. 2007, 2007, pp. 97–106.

[6] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, “A few billion lines of code later: using static analysis to find bugs in the real world,” *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[7] T. Ball and S. K. Rajamani, “Automatically validating temporal safety properties of interfaces,” in *Proceedings of the 8th international SPIN workshop on Model checking of software.* Springer-Verlag New York, Inc., 2001, pp. 103–122.

[8] W. R. Bush, J. D. Pincus, and D. J. Sielaff, “A static analyzer for finding dynamic programming errors,” *Software-Practice and Experience*, vol. 30, no. 7, pp. 775–802, 2000.

[9] C. Woolley, “High-productivity cuda programming,” 2013. [Online]. Available: http://on-demand.gputechconf.com/gtc/2013/presentations/S3008-High-Productivity-CUDA-Programming.pdf