Monash University

Electrical Engineering

Final Year Project (ECE4093)

Final Report

Author:
James Anastasiou
ID: 23438940

Supervisor:
Dr. David Boland

September 26, 2016

Contents

1 Significant Contribution **4**

2 Introduction **5**

2.1 Purpose and Scope 5

3 Project Description **6**

3.1 Compiler Analytics 6

3.1.1 OCLint Modification 7

3.2 GPU Benchmarking 7

3.2.1 Benchmarks 7

3.2.2 CUDA 8

4 Literature Review **9**

4.1 Introduction 9

4.2 Automatic Vectorisation 9

4.3 Parallelism 10

4.4 Static Analysis 12

4.5 Summary 12

5 Parallel Computation **13**

5.1 SIMD 13

5.2 Parallel Limitations 13

5.3 Parallel Primitives 14

5.3.1 Map 15

5.3.2 Reduce 15

5.3.3 Scan 16

5.3.4 Linear Algebra 17

5.4 GPU Parallelism 17

6 Clang Integration **18**

7 Project Components **19**

7.1 Static Analyser 19

7.1.1 Clang Integration 19

7.1.2 ASTMatcher Combinator Library 20
7.1.3 Pattern Recognition 23
7.1.3.1 Matching 23
7.1.3.2 Callback 23
7.1.4 Benchmark Integration 24
7.1.4.1 JSON Parsing 24
7.1.4.2 Internal Representation 24
7.1.5 Reporting 25
7.2 Benchmarks 25
7.2.1 Benchmark Structure 25
7.2.2 Tools 25
7.2.3 Implementations 25
7.2.3.1 Map 25
7.2.3.2 Reduce 25
7.2.3.3 Scan (Prefix Sum) 25
7.2.3.4 Dense Matrix Multiplication 25
7.2.4 Typesafe CUDA primatives 26

8 Case Study 28

9 Re-evaluation of Initial Goals 30
9.1 GPU Benchmark Development 31
9.1.1 Requirements 31
9.1.1.1 [FR.003] 31
9.1.1.2 [OA.001] 32
9.1.1.3 [OA.002] 32
9.1.1.4 [OA.003] 32
9.1.1.5 [OA.004] 33
9.1.1.6 [OA.005] 33
9.1.1.7 [OA.006] 33
9.1.1.8 Summary 34
9.2 Algorithm Analytics Development 34
9.2.1 Requirements 34
9.2.1.1 [FR.001] 34

9.2.1.2 [FR.002] 35
9.2.1.3 [FR.004] 35
9.2.1.4 [CA.001] 35
9.2.1.5 [CA.002] 35
9.2.1.6 [CA.003] 36
9.2.1.7 [CA.004] 36
9.2.1.8 [CA.005] 37
9.2.1.9 [CA.006] 37
9.2.1.10 [CA.007] 37
9.2.1.11 [CA.008] 37
9.3 Optimisation Analytics Development 37

10 Limitations and Extensions **39**
10.1 Limitations 39
10.2 Extensions 40

11 Appendix **41**

1 Significant Contribution

- Designed and implemented a Static Analysis tool for identifying potential parallelism in sequential C code.
- Designed and implemented an AST Matcher Combinator header only library.
- Designed and implemented a CPU and GPU benchmarking suite.
- Designed and implemented a Templated Typeclass interface for parallel primitives in CUDA C++.

2 Introduction

2.1 Purpose and Scope

This document is the complete final report of my final year project. It is the complete documentation of the entire project and as such it includes:

- Project Description
- Literature Review
- Evaluation of initial goals
- Final Design
- Project Limitations and Possible Extension

3 Project Description

My final year project aimed to create a set of static code/compiler analytics tools to help determine which algorithms within a code-base may be easily parallelized. The specific intention of it is to provide users with a report describing which parts of their code-base may see a potential speed-up from redeveloping them as CUDA (GPU) [?] kernels. In order to achieve this both benchmarking and theoretical analysis was required, as well as static analysis of the C description of the algorithm itself. My final project utilised a combined strategy of static AST analysis, via a hook into the Clang Libtooling, as well as integration with a benchmarking suite I created that utilised real high performance libraries for both Host and Device code. The use case of this software is for developers working primarily in modelling and mathematically intensive areas, as these tend to provide significant opportunity to provide improvement. The expectation of potential users is that they will note that the report alerts them to potential speedups, and then use the performance metrics to determine whether or not to hire a specialised engineer to redesign the relevant components of their system.

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 Libtooling 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 light 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 exposing the Clang Libtooling headers. 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 were lodged, and no modifications I have written have moved upstream. As I have substantially re-engineered and re-purposed the OCLint software I have elected to give it a new name, 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 involved 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 available 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

GPUs are exceptionally good at high throughput calculations, one particular example is SIMD, meaning *Same Instruction Multiple Data.* The performance of GPUs and the algorithms they are particularly useful for is well under continual research, however general problem classes that GPUs 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

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 GPUs that determine their performance in each of these categories. In order to satisfy time constraints and recognise real world concerns, I elected to use existing optimised libraries for the individual components of the benchmarking. I relied on the Eigen Library [?] for host side matrix operations. On the device side I relied on a combination of the thrust template library [?] in combination with CuBLAS [?]

3.2.2 CUDA

CUDA is Nvidia's proprietary 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 severely 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. This is not too great a concern however, as the benchmarking module is highly extensible and as a result can easily integrate with a variety of backends, including OpenCL or OpenMP.

4 Literature Review

4.1 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.

4.2 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 [?] 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 [?]. 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 sub-optimal performance of the LLVM auto-vectoriser [?]. 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.

4.3 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 [?] 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 [?] 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.

4.4 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 [?], 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 [?] [?]. 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 [?] 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.

4.5 Summary

TODO:

5 Parallel Computation

Parallel computing is a type of computation whereby many operations are performed simultaneously, as opposed to serial computing where only one operation occurs at a time [?]. Parallel computing has been used as a high performance computing technique for some time, with recent physical limitations on serial processors forcing further development in the parallel world. Modern parallel computing focuses primarily on extracting maximum performance from data level parallelism, the process by which independent processors act on a distributed load of data, often performing SIMD (Signle Instruction Multiple Data) operations.

5.1 SIMD

SIMD describes a computation structure by which many processing units execute the same instruction on multiple data in parallel. SIMD allows for significant computation speed improvements over traditional serial data processing by operating on multiple data at once. The theoretical speed improvements a SIMD processor has over a traditional serial processor can be described by: $speedup \propto min\,(N_{processing\_units}, N_{data})$. In many computing environments N_{data} is significantly larger than $N_{processing\_units}$ simplifying the relationship to merely $speedup \propto N_{processing\_units}$. Thus it is clear that with increasing number of processing units the commputation speed increases. As a result of this relationship devices have been developed which can exploit this fact, most modern CPU's include some form of Vectorised SIMD instruction set. Advanced Vector Extensions (AVX) are an extension to the x86 instruction set which are supported by both AMD and Intel, which utilises SIMD to improve processor performance for highly parallel workloads. GPU's however are far more suited to the task of performing SIMD operations as they often have orders of magnitude more processing units than comporable CPU's.

5.2 Parallel Limitations

One of the largest limitations that concerns parallel computing is data dependency. A data dependency is a situation where operations in the algorithm require data from earlier in the algorithm in order to continue processing. An example situation would be:

```c
double mean = 0;

// Calculate Mean
for (size_t i = 0; i < ELEMS; ++i)
    mean += vec[i]/ELEMS;

// Data Dependency: Relies on Calculated Mean
for (size_t i = 0; i < ELEMS; ++i)
    vec[i] = abs(vec[i] - mean);

// Data Dependency: Relies on other value of vec
for (size_t i = 0; i < ELEMS; ++i)
    vec[i] = vec[vec[i]];
```

in this case we have two examples of data dependency, in the first case we are trying to normalise the vector by subtracting the mean. This is an example of a data dependency where a SIMD operation relies on prior information, in this case this will not impact our ability to perform the operations simultaneously, as at no point does the input information rely on potential changes to the output information as a side-effect of this computation. However in the second case, where we re-assign the vector values, there is a data dependency that prevents a Parallel Implimentation from being naivly implemented. `vec[vec[i]]` has a dependency on prior operations performed to `vec[]` which prevents us from processing all elements of this loop simultaneously. There are however classes of problems which may be simply parallelised, these are known as Parallel Primitives.

5.3 Parallel Primitives

Parallel Primitives, or Vector Primitives are operations over a collection of values, there are three operations which constitute these primitives:

- Map
- Reduce
- Scan (Prefix Networks)

These operations all rely on the ability to reformat the problem specification to utilise a computation graph for simultanous calculation.

5.3.1 Map

```
for (size_t i = 0; i < SIZE; ++i)
  out_vec[i] = in_vec[i] * 2;
```

Map operations are the simplest of the three Parallel Primitives, they are simply operations describing a one to one mapping from some input value to some output value, Mapped over a set of values. Map operations have some restrictions upon them about what is considered valid inputs. Operators must have an arity of 1 and the operator must be stateless. If these rules are held then the system will be by definition an LTI system, allowing for a trivial SIMD implementation of the resulting transformation. If however the input arguments do not satisfy these requirements, then the resulting operation will not be well formed, and in most implementations the result will be ill-defined. Map operations have a work complexity of $O(N)$ for CPU implementations and $O(N/k)$ for parallel implementations where k is the number of computational cores available.

5.3.2 Reduce

```
for (size_t i = 0; i < SIZE; ++i)
  k += in_vec[i];
```

Reductions are the next simplest of the Parallel Primitives, they are a mapping from many input values to one output value. Input argument restrictions on Reductions are more strict than on Map operations. Reductions require that the operator and data form a monoid; the Operator must be a binary associative operator, and the set of input values must be closed under that operator. A simple example would be addition, over the Reals. Like with the Map primitive, if the input restrictions are not met, then the operation will not be well formed. Reduce operations have a work complexity of $O(N)$ for CPU implementations and $O(N)$ for parallel implemetations, where the Step Complexity is $O(logN)$. Reductions form a work-efficient operation.

Additional optimisation can occur if the operator is not only associative but also commutative, whereby the gathering of values may occur out of typical oder providing potentially better constant factors.

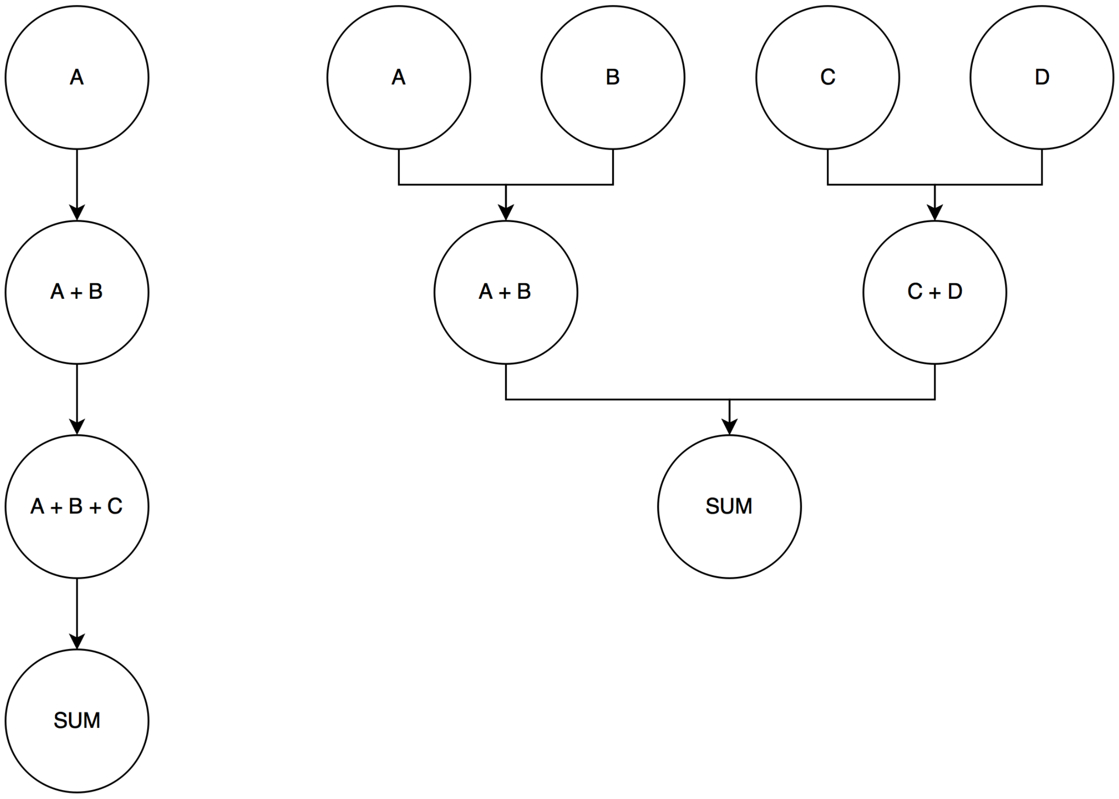


Figure 1: Serial vs Parallel Sum Reduction Tree

5.3.3 Scan

```
out_vec[0] = in_vec[0];
for (size_t i = 1; i < SIZE; ++i)
  out_vec[i] = in_vec[i] + out_vec[i-1];
```

Scans or Prefix Networks are the most complext of the Parallel Primitives. Scans are a mapping from many input values to many output values. They are a direct generelisation of a Reduction, where the cumulative intermediate values are maintained. Scan operations require the same restrictions for Reductions hold. Scans are not trivially parallelisable, as there is a data dependency on prior calculated values, however there are algorithms for performing parallel Scan operations whilst still retaining work efficiency (a work complexity of $O(N)$) [?].

5.3.4 Linear Algebra

TODO: THIS (NOT A PRIMITIVE, BLAS HOWEVER IS VERY PARALLELIS-ABLE)

5.4 GPU Parallelism

GPUs typically consist of thousands of processing cores, this is significantly greater than the common 4-8 cores found on modern CPUs. GPU architecture relies on numerous simple processing units which engage in SIMD, leveraging the relationship between computational cores and work efficiency.

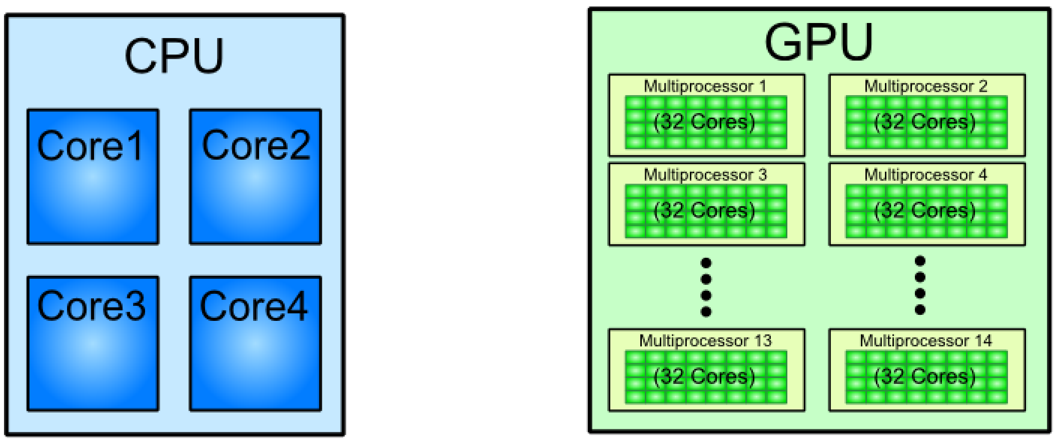


Figure 2: CPU vs GPU Architecture [?]

Efficient GPU programming like efficient CPU programming requires specialised knowledge, both of the target hardware, and of the paradigm. GPU programming is able to exploit the massively parallel compute archictecture on these devices. Coupling fast global memory, high performance shared memory and numerous local registers GPUs provide all the requirements for exploiting SIMD in a massively parallel space.

6 Clang Integration

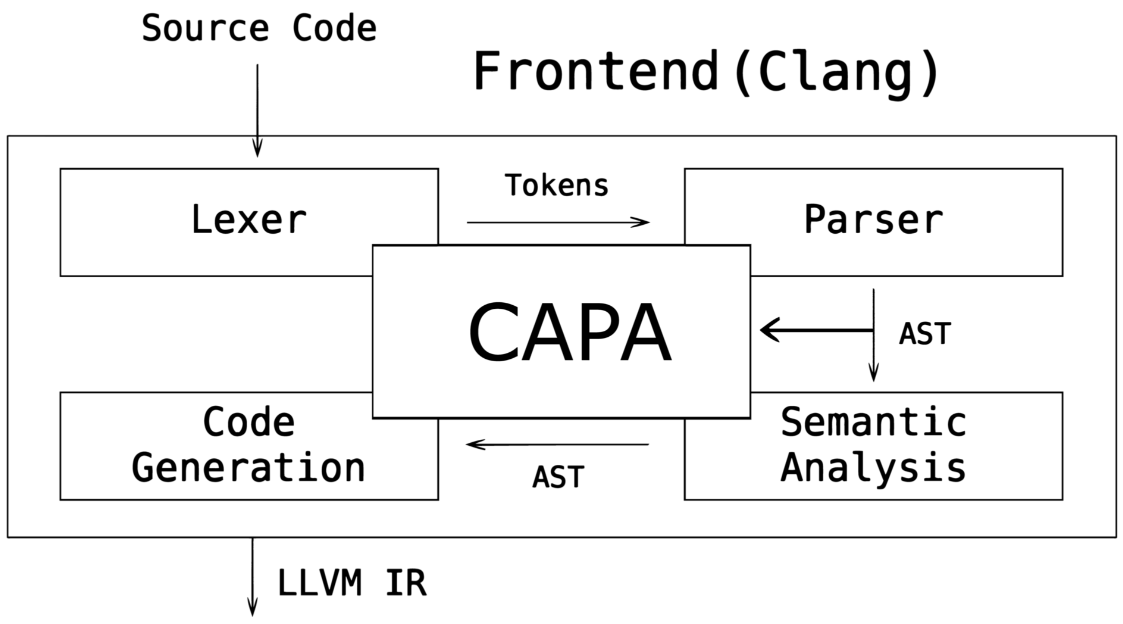


Figure 3: CAPA Hook-In

TODO: THIS

7 Project Components

This section aims to provide an in depth look at what has been achieved over the life of the project.

7.1 Static Analyser

7.1.1 Clang Integration

CAPA utilises the Clang Libtooling library in order to perform semantic static analysis of any C codebase. The Clang compiler exposes a public library for interfacing with their intermediate compilation stage representations of the original source. For the purpose of this project it was decided to use the exposed AST interface in order to perform the static analysis. Clang provides a number of methods for working with the AST, namely the Visitor and Matcher interfaces. The Visitor library utilises the visitor pattern, and a callback is undertaken upon visiting any node which meets the requirements set forth in the visitor module. This is a useful tool, however it is not as powerful as the Matcher interface, which allows complex grammar's to be generated for highly specific, tailored matches. CAPA utilises the ASTMatcher callback interface in order to provide complex generic and extensible traversals of the AST. The matcher interface provides a declarative API by which the program searches for regions which satisfy known static requirements. The interface itself is rather unwieldly to use directly.

```
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);
```

As a result I created a matcher combinator library for simplification purposes.

7.1.2 ASTMatcher Combinator Library

In order to simplify the matchers and prevent them from becoming unmanagebly large, I designed a lambda based combinator library for creating complex AST-Matchers. Utilising C++14 auto lambda return type declarations I was able to construct a number of higher order combinators which can be combined together to construct more complex matchers with less ambiguity. A simple example:

```
namespace CAPA {

// Variable Binding Combinators
auto VarBind = [](std::string binding)
{
    return declRefExpr(to(varDecl().bind(binding)));
};

auto DVarBind = [](std::string binding)
{
    return hasDescendant(VarBind(binding));
};
```

```cpp

auto VectorBind = [](std::string binding)
{
    return arraySubscriptExpr(
            hasBase(DVarBind(binding + "Base")),
            hasIndex(DVarBind(binding + "Index")));
};

auto MatrixBind = [](std::string binding)
{
    return arraySubscriptExpr(
            hasBase(hasDescendant(arraySubscriptExpr(
                hasBase(DVarBind(binding + "Base")),
                hasIndex(DVarBind(binding + "Row"))))),
            hasIndex(DVarBind(binding + "Column")));
};

// Loop Binding Combinators
auto LoopInit = [](std::string level)
{
    return anyOf(
        declStmt(hasSingleDecl(varDecl(hasInitializer(
            integerLiteral(anything()))).bind("InitVar" + level))),
        binaryOperator(
            hasOperatorName("="),
            hasLHS(VarBind("InitVar" + level))));
};

auto LoopIncrement = [](std::string level)
{
    return anyOf(
            unaryOperator(anyOf(
                    hasOperatorName("++"),
                    hasOperatorName("--")),
                hasUnaryOperand(VarBind("IncVar" + level))),
            binaryOperator(anyOf(
                    hasOperatorName("+="),
                    hasOperatorName("-=")),
                hasLHS(VarBind("IncVar" + level)),
                hasRHS(expr().bind("Stride" + level))));
};
```

```

auto ForLoop = [](std::string binding, std::string level, auto injectBody)
{
    return forStmt(
            hasLoopInit(LoopInit(level)),
            hasCondition(binaryOperator(hasRHS(expr().bind(binding +
                "CondRHS")))),
            hasIncrement(LoopIncrement(level)),
            hasBody(injectBody)).bind(binding);
};

} // End Namespace CAPA
```

This combinator library again is easily extensible and as further needs arose I increased its complexity and breadth. Ultimately it provides a simpler way of interfacing with the Clang AST Matcher library through safer higher order functions. This combinator library would not be possible with earlier versions of C++, as it has a reliance on auto return types to prevent template instantiation stack errors. The AST Matcher interface heavily relies on template meta-programming in order to ensure a clean typesafe interface, the combinator library I designed extends that, yet still maintains complete statically verifiable interfaces that are type correct.

```
auto left = VectorBind("Out");
auto right = hasDescendant(VectorBind("In"));
auto unless =
    hasDescendant(arraySubscriptExpr(hasDescendant(binaryOperator())));

auto body = hasDescendant(BinaryOperatorBindUnless("=", "Assign", left, right,
    unless));

auto MapMatcher = ForLoop("Map", "", body);

addMatcher(MapMatcher);
```

This version is clearly far simpler to understand and to modify, whilst still achieving the same callback results as the original version. This demonstrates the power of the Matcher Combinator interface, as well as the ease of use.

7.1.3 Pattern Recognition

7.1.3.1 Matching By utilising the OCLint tool the scaffoling around the libtooling had already been provided, simplifying the interface between pattern matching and reporting. There was still a significant rewrite of most of the interface, however the heirachy and design philosophy was clear. The actual matching of potentially parallel sections of code relies on a few assumptions about the nautre of algorithms which may be efficiently implemented on a GPU.

- Little prior data dependency
- A large number of elements require processing
- Minimal control flow is required

Given these assumptions, in combination with the knowledge of problem spaces that the GPU is highly performant in:

- Map Operations
- Reduce Operations
- Scan (Prefix Sum) Operations
- Matrix Operations

it became clear that identifying components of the codebase that exhibited similarities to these cases would be important.

7.1.3.2 Callback Even though the matcher can be highly specific, there is still the requirement of the callback. The callback is responsible for identifying whether the matched pattern is actually representative of the expected pattern, or whether it is infact a potential false positive. Additionally the callback is responsible for logging the detected pattern as well as relevant information, such as the number of elements being manipulated, for use by the reporter module. This will be explained in more detail in section **??**.

7.1.4 Benchmark Integration

A key aspect of CAPA is the integration of the benchmarking component with the detected outcomes to provide more detailed performance statistics during the reporting phase.

7.1.4.1 JSON Parsing CAPA relies on performance benchmarks being generated by the benchmarking tool, this reports back information in a JSON form which is read and parsed by CAPA into a useable form. The JSON for Modern C++ library was used [**?**]. JSON was chosen as it is a human readable format which is supported by a large number of tools, additionally the benchmarking libary utilised provided a JSON exporter, simplifying the integration of the two tools.

7.1.4.2 Internal Representation Internally the benchmarking information relies on this construct:

```
class BenchmarkSet
{
public:
    BenchmarkSet(std::string benchmarkLocation);
    BenchmarkSet();
    bool Exists(std::string operation) const;
    double Speedup(std::string operation, std::size_t dimension) const;
    double Speedup(std::string operation) const;
    std::tuple<double, double> GetResult(std::string operation, std::size_t
        dimension) const;
    std::tuple<double, double> LowerUpper(std::string operation, std::size_t
        dimension) const;

private:
    std::map<std::string, std::map<std::size_t, std::tuple<double, double>>>
        benchmarks;
    static const std::map<const std::string, const std::size_t> VectorFixtures;
    static const std::map<const std::string, const std::size_t> MatrixFixtures;
};
```

which simply provides a simple concise manner in which to interface with STL containers for the purpose of passing around the parsed benchmarking information.

The BenchmarkSet object is responsible for handling all requests for benchmark information, including providing theoreticals if no benchmark JSON file exists. This information is then used by the reporter module to provide the end user with further information about potential optimisations within their analysed codebase.

7.1.5 Reporting

7.2 Benchmarks

7.2.1 Benchmark Structure

7.2.2 Tools

7.2.3 Implementations

7.2.3.1 Map

Host

Device

7.2.3.2 Reduce

Host

Device

7.2.3.3 Scan (Prefix Sum)

Host

Device

7.2.3.4 Dense Matrix Multiplication

Host

Device

7.2.4 Typesafe CUDA primatives

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]);
}

```

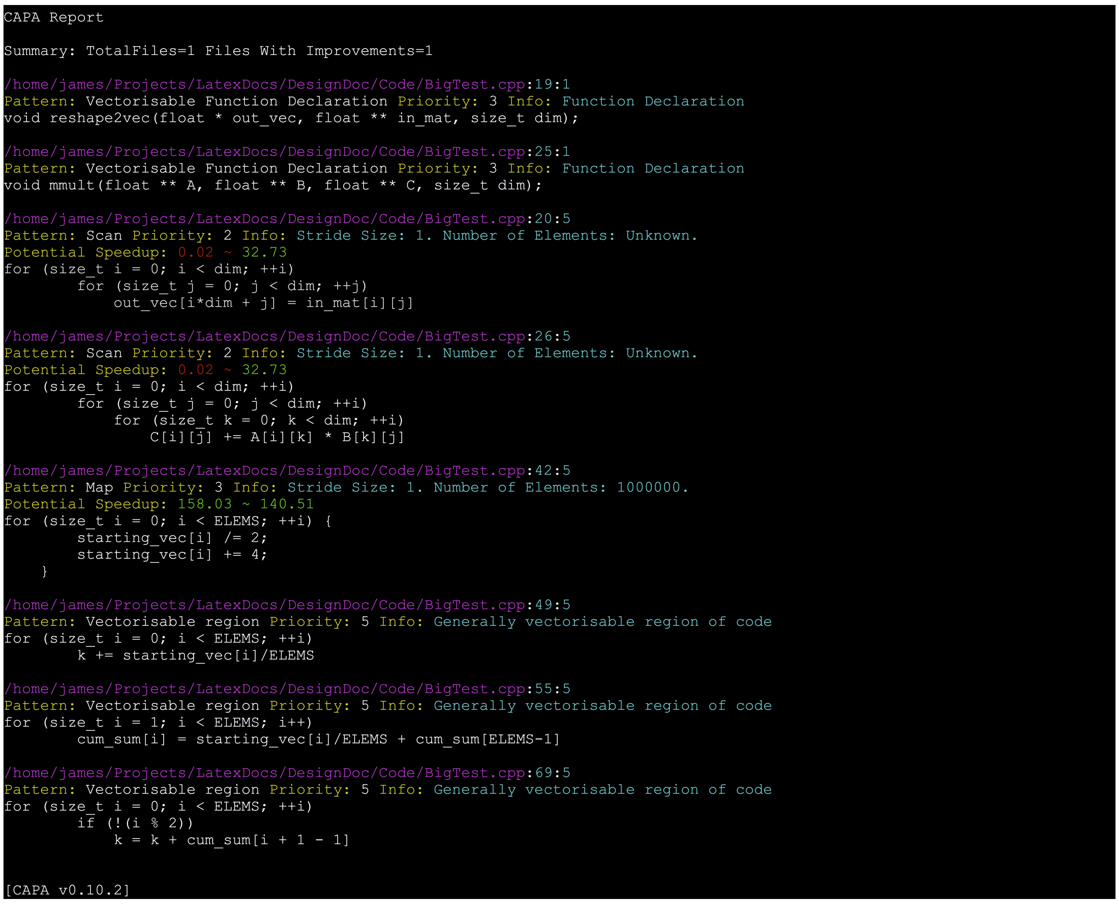
```
template <typename T, uCat<T> func>
__global__ void MapKernel(T *a, T *c, size_t size)
{
    size_t i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < size)
        c[i] = func(a[i]);
}
```

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.

8 Case Study

Code Under Test

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

/// CAPA:IGNORE
void random_fill(float *starting_vec, size_t size) {
    for (size_t i = 0; i < size; ++i)
        starting_vec[i] = (float) rand();
}

/// CAPA:IGNORE
void reshape2mat(float *in_vec, float *out_vec[], size_t dim) {
    for (size_t i = 0; i < dim; ++i)
        for (size_t j = 0; j < dim; ++j)
            out_vec[i][j] = in_vec[i*dim + j];
}

void reshape2vec(float *out_vec, float *in_mat[], size_t dim) {
    for (size_t i = 0; i < dim; ++i)
        for (size_t j = 0; j < dim; ++j)
            out_vec[i*dim + j] = in_mat[i][j];
}

void mmult(float **A, float **B, float **C, size_t dim) {
    for (size_t i = 0; i < dim; ++i)
        for (size_t j = 0; j < dim; ++i)
            for (size_t k = 0; k < dim; ++i)
                C[i][j] += A[i][k] * B[k][j];
}

/// Test example Case.
int main() {
    const size_t ELEMS = 1000*1000;
    float starting_vec[ELEMS];
    time_t t;

    srand((unsigned) time(&t));
    random_fill(starting_vec, ELEMS);

    // Divide all points by 2 and add 4 for later stage
    for (size_t i = 0; i < ELEMS; ++i) {
        starting_vec[i] /= 2;
        starting_vec[i] += 4;
    }

    // Calculate mean
    float k = 0;
    for (size_t i = 0; i < ELEMS; ++i)
        k += starting_vec[i]/ELEMS;

    // Renormalise all values and compute cumulative sum
    float cum_sum[ELEMS];
    cum_sum[0] = starting_vec[0]/k;
    for (size_t i = 1; i < ELEMS; i++)
        cum_sum[i] = starting_vec[i]/ELEMS + cum_sum[ELEMS-1];

    // Prepare for and perform Matrix Mult
    const size_t dim = sqrt(ELEMS);
    float cum_sum_mat[dim][dim];

    reshape2mat(cum_sum, (float **) cum_sum_mat, dim);
    mmult((float **) cum_sum_mat, (float **) cum_sum_mat, (float **)
        cum_sum_mat, dim);
    reshape2vec(cum_sum, (float **) cum_sum_mat, dim);

    // reduce only even values, but do it with conditional
    // and deliberately prevent it being caught
    k = 0;
    for (size_t i = 0; i < ELEMS; ++i)
        if (!(i % 2))
            k = k + cum_sum[i + 1 - 1];

    return k;
}
```



```
CAPA Report

Summary: TotalFiles=1 Files With Improvements=1

/home/james/Projects/LatexDocs/DesignDoc/Code/BigTest.cpp:19:1
Pattern: Vectorisable Function Declaration Priority: 3 Info: Function Declaration
void reshape2vec(float * out_vec, float ** in_mat, size_t dim);

/home/james/Projects/LatexDocs/DesignDoc/Code/BigTest.cpp:25:1
Pattern: Vectorisable Function Declaration Priority: 3 Info: Function Declaration
void mmult(float ** A, float ** B, float ** C, size_t dim);

/home/james/Projects/LatexDocs/DesignDoc/Code/BigTest.cpp:20:5
Pattern: Scan Priority: 2 Info: Stride Size: 1. Number of Elements: Unknown.
Potential Speedup: 0.02 ~ 32.73
for (size_t i = 0; i < dim; ++i)
        for (size_t j = 0; j < dim; ++j)
            out_vec[i*dim + j] = in_mat[i][j]

/home/james/Projects/LatexDocs/DesignDoc/Code/BigTest.cpp:26:5
Pattern: Scan Priority: 2 Info: Stride Size: 1. Number of Elements: Unknown.
Potential Speedup: 0.02 ~ 32.73
for (size_t i = 0; i < dim; ++i)
        for (size_t j = 0; j < dim; ++i)
            for (size_t k = 0; k < dim; ++i)
                C[i][j] += A[i][k] * B[k][j]

/home/james/Projects/LatexDocs/DesignDoc/Code/BigTest.cpp:42:5
Pattern: Map Priority: 3 Info: Stride Size: 1. Number of Elements: 1000000.
Potential Speedup: 158.03 ~ 140.51
for (size_t i = 0; i < ELEMS; ++i) {
        starting_vec[i] /= 2;
        starting_vec[i] += 4;
    }

/home/james/Projects/LatexDocs/DesignDoc/Code/BigTest.cpp:49:5
Pattern: Vectorisable region Priority: 5 Info: Generally vectorisable region of code
for (size_t i = 0; i < ELEMS; ++i)
        k += starting_vec[i]/ELEMS

/home/james/Projects/LatexDocs/DesignDoc/Code/BigTest.cpp:55:5
Pattern: Vectorisable region Priority: 5 Info: Generally vectorisable region of code
for (size_t i = 1; i < ELEMS; i++)
        cum_sum[i] = starting_vec[i]/ELEMS + cum_sum[ELEMS-1]

/home/james/Projects/LatexDocs/DesignDoc/Code/BigTest.cpp:69:5
Pattern: Vectorisable region Priority: 5 Info: Generally vectorisable region of code
for (size_t i = 0; i < ELEMS; ++i)
        if (!(i % 2))
            k = k + cum_sum[i + 1 - 1]


[CAPA v0.10.2]
```

Figure 4: Generated CAPA Report

9 Re-evaluation of Initial Goals

Considering the project has reached completion, it is important to review the requirements initially set forth in order to ascertain whether the objectives of the project have been met. My final year project involved a large amount of exploratory work, and as the project developed different area's became more important to the final deliverable than were anticipated at the beginning. For the design document submitted in week 12 of semester one, a review of my initial requirements was undertaken, and a re-evaluation of their status within the project was conducted. Since then further developments of the project demanded more time and focus be spent on other aspects, as such it is worth re-evaluating the midway evaluation, and where the project stands at completion. The key area's of the project were.

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

which then allows the following breakdown..

9.1 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 was a key aspect of satisfactorily completing the project. The GPU benchmark development had a number of requirements that describe what the project necessitates.

9.1.1 Requirements

9.1.1.1 [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 a completed benchmark suite covering all of the GPU primatives in a highly extensible and meta-programmable fashion. Each of the following benchmarks has been implemented.

- Peak Map
- Peak Fold/Reduce
- Peak Scan
- Peak Matrix Multiplication

Additionally it is trivial to extend the benchmark suite to cover other aspects of GPU performance.

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

This requirement was met. In order to satisfy this requirement I produced benchmarking code for GPU performance in problem sets that are known to be performant on a GPU. It was my original intention that as well as benchmarks that are known to be performant, that I would also write benchmarks that may naively appear to be performant, yet further inspection demonstrates that they are not in fact performant. This was a rather large task that did not relate back to the key intentions of the project. It was disappointing that constraints prevented the full exploration of this concept, however as the project developed it was clear that focusing on the positive performance aspects would provide a significantly better final product.

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

After careful consideration this requirement was relaxedi at the midpoint of the project. It was determined to be far too strict. When writing the original requirements analysis I was not as familiar with the CUDA toolchain as I was at the midpoint of the 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 decided to limit 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.

The revision of the initial requirement saved significant time and effort from being wasted in localisation and highly technical activities that would have had very limited benefits for the project.

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

This requirement was met. The final implementation of the benchmark suite contains both CPU and GPU benchmarks for identical operations. This is then used to provide comparitive performance information to the CAPA reporter in order to make predictions about performance improvements within the codebase.

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

This requirement was covered and expanded under sections 9.1.1.1 and 9.1.1.2

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

This requirement was met. 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. The implementation of this utilised work complexity comparisons between the sequential and parallel versions of operations. Some aspects of the parallel code matched do not provide these theoreticals, as in many cases the final semantics cannot be gathered from the AST representation without runtime information.

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

This requirement was not met. During the midpoint review of the project it was clear that the original intention of benchmarking CPU, GPU and FPGA implementations of algorithms was an unattainable goal, this was mainly due to the increased focus on the compiler analytics aspect of the projct. Between the original requirements analysis and the midpoint review the emphasis became more focused upon 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 elected early to not satisfy the requirement, and to remove it from what this project intends to achieve. Removing this requirement provided more time for solving problems which were more relevant to final product.

9.1.1.8 Summary The original design of this project was to primarily provide a benchmarking and comparison suite to assist programmers and engineers in decision making about how to best optimise their software. Very quickly though the project was refocused on analysing source code rather than concepts. This decision increased the challenge of the project, but also provides more utility, as it has the capability to be integrated far deeper in the optimisation decision process. As a result however many aspects of the benchmarking side of the project were compromised. These compromises were recognised before the midpoint of the project and were thoroughly documented in the Design Document.

9.2 Algorithm Analytics Development

This is the crux of my final year project. The core objective was to produce software that analyses a C source file and identify whether the algorithms 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 was 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.

9.2.1 Requirements

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

This requirement was met. This was the key requirement of the entire project. This requirement could not be compromised on, and as such all other requirements had to relate to ensuring this requirement was met. Analysis of the algorithm was defined as static code analysis, and optimisation analysis was defined as the recognition of potentially parallelizable algorithms within the code. This was 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.

At the midpoint of the project most of this requirement had been met. However

I was not pleased with the manner in which the analysis was undertaken. It was quite fragile and difficult to parse. A redevelopment of the entire matching interface was undertaken and as a result I created functional combinator matcher library to facilitate the generation of generic extensible AST Matcher constructs. This allowed for the rapid redevelopment and extension of the original analysers, providing and extensible framework from which further matchers could easily be developed.

The fundamental intention of the requirement was to provide a list of potential improvements from within the code, similar to runtime profiling, by using semantics as described by the designer, rather than performance outcomes determined by a profiler. This was achieved.

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

This requirement was met. This relates back to the discussion about theoretical performance metrics in 9.1.1.6. This is the extension of that requirement. Where actual benchmark information is not available the tool provides theoretical performance based on time and work complexities of the relevant algorithms.

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

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

9.2.1.4 [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 CAPA is built.

9.2.1.5 [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.

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

This requirement was met. 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 meant 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 was a sizeable task. The matrix matcher however was completed, and identifies matrix-matrix multiplication, as well as matrix-vector multiplication.

9.2.1.7 [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. The original intention of this requirement was to identify graph traversals, this however proved to be a particularly difficult task. Whilst graph traversals are not identified by CAPA, we are still able to identify non-trivial potentially parallel sections of code. Primarily CAPA looks for loops with minimal control structures which may be potentially unrolled or vectorised. Additionally CAPA inspects function declarations and types to determine whether they contain types which are to be expected in highly parallel computation, typically pointers, arrays and unsigned integers defining size. This coupled with a low cyclometic complexity within the definition suggests that the included code may be potentially parallelisable, and as such it is often reported by the software.

This requirement has been met, with a slight caveat.

9.2.1.8 [CA.005] Provide Theoretical improvement information

Has been covered extensively here 9.2.1.2 and here 9.1.1.6.

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

This requirement has been met. Expanding on 9.2.1.2 and 9.1.1.6, if the user decides to provide additional information, then the theoretical performance metrics will take this information into consideration when calculating potential performance improvements. The user is capable of providing information by way of a configuration file which CAPA reads to provide more accurate theoretical calculations.

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

This requirement has been met. The project is capable of working on any Clang compilable C codebase. As such the analyser is capable of analysing general C code algorithm descriptions from a functional point of view.

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

This requirement has been conditionally met. Tags are not always visible at every node within the clang AST and as such I was not able to reliably analyse only subsections of a codebase, rather the user has to select which functions within the codebase they do not want to be reported. The program still analyses these regions of the codebase, however upon detecting an ignore clause in the parent function comment declaration, the analysis for that match terminates and the next matcher begins. This allows the developer to selectively remove false positives or known regions where more computational speed is not necessary. So whilst this condition was originally to only analyse within tagged regions, it has been changed to tag only regions which have not been tagged as: `///CAPA:IGNORE`

9.3 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, and it was achieved.

10 Limitations and Extensions

10.1 Limitations

10.2 Extensions

11 Appendix

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

References

[1] "Cuda," 2016, license: http://docs.nvidia/com/cuda/eula. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[2] "Oclint," 2016, license: Modified BSD. [Online]. Available: http://oclint.org/

[3] "Eigen," 2016, license: MPL2. [Online]. Available: http://eigen.tuxfamily.org

[4] "Thrust," 2016, license: Apache 2.0. [Online]. Available: http://thrust.github.io

[5] "Cublas," 2016, license: http://docs.nvidia.com/cuda/eula/. [Online]. Available: https://developer.nvidia.com/cublas

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

[7] "Auto-vectorization in llvm." [Online]. Available: http://llvm.org/docs/Vectorizers.html

[8] 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.

[9] 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.

[10] K. Nagarajan, B. Holland, A. D. George, K. C. Slatton, and H. Lam, "Accelerating machine-learning algorithms on fpgas using pattern-based decomposition," *Journal of Signal Processing Systems*, vol. 62, no. 1, pp. 43–63, 2011.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] G. S. Almasi, *Highly Parallel Processing (The Benjamin/Cummings series in computer science and engineering).* Benjamin-Cummings Publishing Co.,Subs. of Addison Wesley Longman,US, 1987. [Online]. Available: http://www.amazon.com/Parallel-Processing-Benjamin-Cummings-engineering/dp/0805301771%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0805301771

[17] M. Harris, “Parallel prefix sum (scan) with cuda,” 2007. [Online]. Available: http://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf

[18] O. Inzunza-Monreal, “Performance of parallel processing on processing units.”

[19] “Json for modern c++,” 2016, license: MIT. [Online]. Available: https://nlohmann.github.io/json/