Monash University

Electrical Engineering

Final Year Project (ECE4093)

Design Document

Author:
James Anastasiou
ID: 23438940

Supervisor:
Dr. David Boland

May 10, 2016

Contents

1 Introduction

1.1 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 Analysis
- Analysis of Outstanding Requirements

2 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.

2.1 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.

2.1.1 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 de-

sign, 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).

2.2 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.

2.2.1 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.

2.2.2 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.

2.2.3 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.

3 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.

3.1 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.

3.1.1 Requirements

3.1.1.1 [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.

3.1.1.2 [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.

3.1.1.3 [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.

3.1.1.4 [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.

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

This requirement was covered and expanded under sections **??** and **??**

3.1.1.6 [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.

3.1.1.7 [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.

3.2 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.

3.2.1 Requirements

3.2.1.1 [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.

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

This relates back to the discussion about theoretical performance metrics in **??**. 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.

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

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

3.2.1.4 [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.

3.2.1.5 [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.

3.2.1.6 [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.

3.2.1.7 [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.

3.2.1.8 [CA.005] Provide Theoretical improvement information

Has been covered extensively here **??** and here **??**.

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

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

3.2.1.10 [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.

3.2.1.11 [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.

3.3 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.

4 Project Status

This section aims to look at what has been completed so far, and what still remains to be done, in order to complete the FYP.

4.1 Completed Tasks

4.1.1