Monash University

Electrical Engineering

Final Year Project (ECE4093)

Design Document

Author:
James Anastasiou
ID: 23438940

Supervisor:
Dr. David Boland

May 6, 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 3.1.1.1 and 3.1.1.2