Dynamic Autotuning of Algorithmic Skeletons

Informatics Research Proposal
Chris Cummins

Abstract. The rapid transition towards multicore hardware which has left application programmers requiring higher-level abstractions for dealing with the complexity of parallel programming. Algorithmic Skeletons provide such abstractions, but typically cannot compete with the performance of hand optimised parallel algorithms without extensive tuning. This paper proposes the development of a dynamic, "always-on" autotuner for SkelCL, an Algorithmic Skeleton library which enables high-level programming of multi-GPU systems. An online machine learning system will create a feedback loop of constant testing and evaluation of skeleton parameters across the lifespan of programs. Dynamically extracted features will characterise muscle functions and input data, and runtime performance data will be used to select optimisation parameters. Such a system will extend the state of the art by enabling empirical optimisation without the huge offline training phases associated with iterative compilation.

1 Introduction

Parallelism is increasingly being seen as the only viable approach to maintaining continued performance improvements in a multicore world. Despite this, the adoption of parallel programming practises has been slow and awkward, due to the prohibitive complexity and low level of abstractions available to programmers.

Algorithmic Skeletons address this issue by providing reusable patterns for parallel programming, offering higher-level abstractions and reducing programmer effort [1, 2]. Tuning the performance of these Algorithmic Skeletons requires programmers to either manually set optimisation parameters based on intuition, or to use an offline training technique to search the space of optimisation parameters and select the optimal configuration.

The aim of this project is to demonstrate that the tuning of optimisation parameters can be successfully performed at runtime without needing offline training. This will enable self-tuning programs which adapt to their execution environment by selecting optimal parameters dynamically. Dynamic optimisation training will be achieved using online machine learning, allowing successive program runs to improve upon its predecessors.

1.1 Hypotheses

This project proposes two hypotheses about the performance of Algorithmic Skeletons:

- a dynamic autotuner will select optimisations that provide improved performance over a baseline Algorithmic Skeleton implementation;
- a dynamic autotuner will provide improved performance over a hand-tuned OpenCL implementation across a range of different inputs, by adapting to changes in the inputs dynamically.

These hypotheses can be referred to respectively as the claims *specialisation* and *generalisation*. We can infer from these that a dynamic autotuner cannot provide better performance than an equivalent OpenCL implementation which has been tuned for a *fixed* input, since the extra instructions required to implement the dynamic autotuner present an unavoidable performance overhead. The reduction of this overhead is one of the greatest challenges facing the development of dynamic autotuners.

1.2 Contributions

The novelty of my solution is to apply online machine learning techniques to the problem of optimisation parameter selection for Algorithmic Skeletons. Targeting the high-level abstractions of Algorithmic Skeletons at runtime exposes many features which cannot be statically determined, such as:

- properties of the input data, e.g. the size, data type, and dimensionality;
- copy-up and copy-down times for transferring data to and from device memory;
- the available devices and number of cores;
- OpenCL compiler settings and optimisation flags;
- system state information, e.g. system load.

2 Motivation

Consider a recursive merge sort algorithm. The algorithm takes an input list, and returns a sorted permutation by determining whether the input list is short enough to be solved directly using a linear sorting method, or whether it should split it into multiple sublists and sort them recursively before combining the results. This computational pattern is abstracted by the Divide and Conquer skeleton, which, when parameterised with four muscle functions, can effectively parallelise this task by considering each recursion as a new task to be executed in parallel. The four muscle functions operate on an input type T_i, return an object of type T_o, and can be formally defined as:

$$\begin{aligned} \mathit{should\_divide} : T_i &\rightarrow \mathit{boolean} \\ \mathit{divide} : T_i &\rightarrow [T_i] \\ \mathit{conquer} : T_i &\rightarrow T_o \\ \mathit{combine} : [T_o] &\rightarrow T_o \end{aligned}$$

The degree of a Divide and Conquer skeleton is the number of sub-problems that a problem is divided into by the *divide* function. For a given degree k, the number of tasks n grows exponentially with depth d:

$$n = k^d - 1$$

On a given machine, the number of tasks which can be effectively executed in parallel is limited by the number of available cores. Since the Divide and Conquer pattern does not constrain the maximum depth that an algorithm may expand to, the negative performance of task switching cost can be avoided by imposing a maximum "parallelisation depth". Recursion above this depth causes the creation of parallel tasks, below this depth, recursion occurs sequentially.

This section describes an experiment to collect empirical data on the effect of varying this parallelisation under varying input conditions.

2.1 Experimental setup

A Divide and Conquer skeleton was implemented and parameterised with muscle functions to implement merge sort. The skeleton was parallelised using the C++11 Thread Support Library, and a testbench recorded the mean time to sort a vector of random unsorted data over 30 iterations.

2.2 Results

Figure 1 shows the mean performance speedup of different parallelisation depths over sequential execution. Figure 1a shows the effect of varying the split size, which is a property of the *should_divide* muscle function that determines the maximum list size at which recursive sort should bottom out and insertion sort should be used. Figure 1b shows the effect of varying the size and data type of the input vector.

In both cases we observe that changing properties of the input and muscle functions has a significant impact on the optimal parallelisation depth parameter. Since neither of those properties can be known *a priori*, the skeleton author must resort to picking a value which they expect to provide best average case performance, which will perform suboptimally for many different input conditions. A dynamic autotuner could adapt to these different inputs by setting the parallelisation depth at runtime.

3 Background

Relevant approaches to the problem of selecting optimisation property values can be broadly categorised as either offline tuning or dynamic optimisation. This section outlines some of the most important works in each category, followed by an introduction to the SkelCL library.

3.1 Offline tuning

Offline tuning involves selecting the set of parameters that provides the best performance for a given input, based on some model of performance that is generated offline. Performance models can either be predictive, in that they attempt to characterise performance as a function of the optimisation parameters and input, or empirical, in that they predict performance based on empirical data gathered by evaluating many different parameter configurations. In both cases, a performance function $f(p, x)$ models the relationship between a set of parameters p, a specific problem x, and some profitability goal. The purpose of the offline tuning phase is to select the set of parameters $p_{optimal}$ which maximises the output of the performance model:

$$p_{optimal} = \arg\max_{p} f(p, x)$$

For predictive models, the quality of results is limited by the ability of the prediction function to accurately capture the behaviour of a real world system. Given the complexities of modern architectures and software stacks, such models have become increasingly hard to develop, although Yotov et al. demonstrated in [7] that under certain scenarios, the performance of accurately generated hand-tuned models can approach that of empirical optimisations.

The quality of empirical models is limited by the amount of training data available to it, and the ability

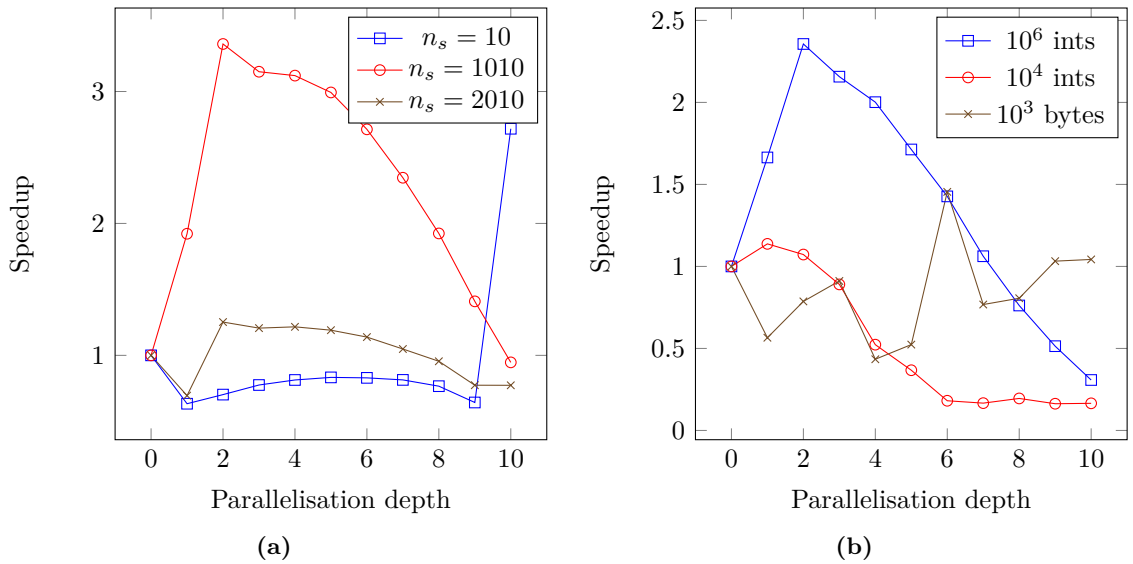

Figure 1: The performance impact of dynamic features on the parallelisation depth parameter: in 1a, as a function of split size n_s; in 1b, as a function of input type and size. In both cases, the optimal parameter value is highly dependent on the dynamic feature.

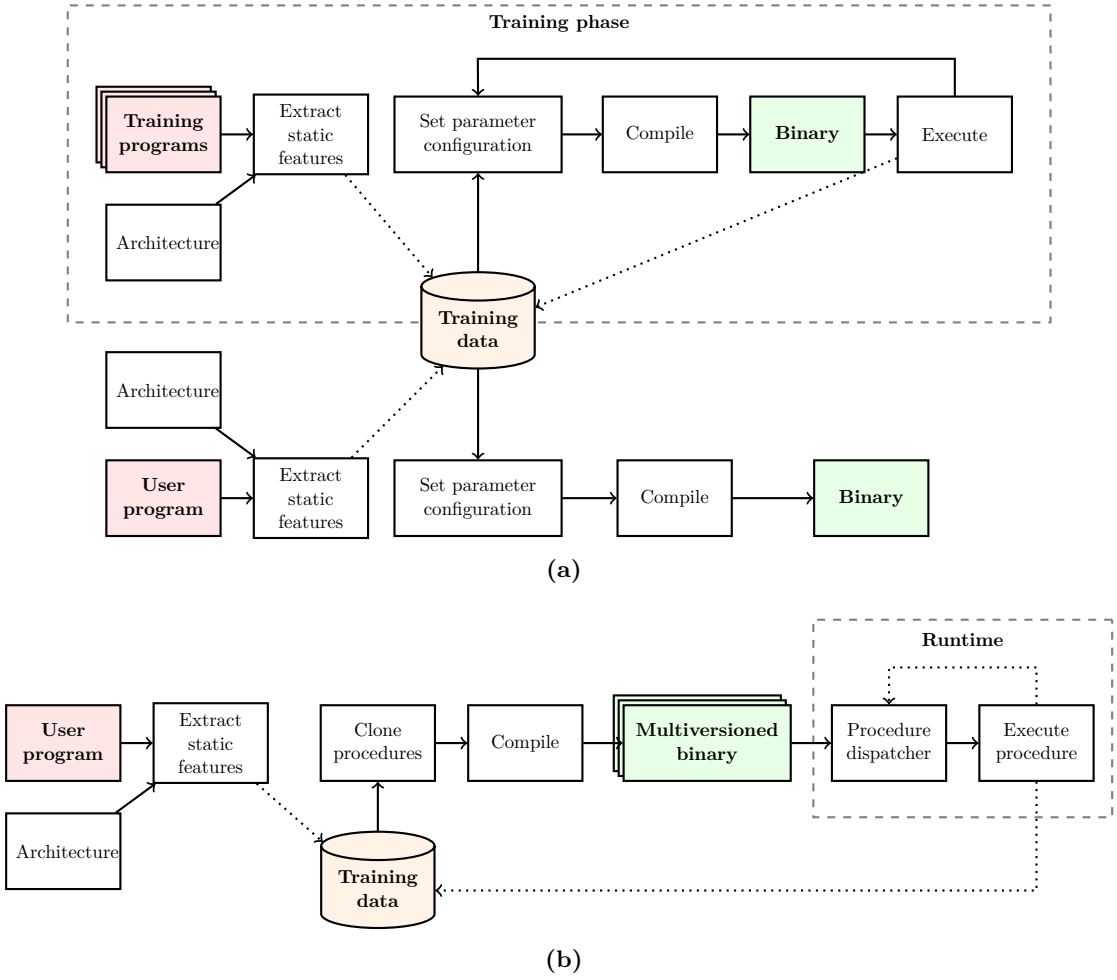

Figure 2: Two approaches to static autotuning: 2a. offline autotuning using a separate training phase, as used in [3–5]; 2b. online autotuning using procedure multiversioning, as used in [6]. In offline autotuning, training programs are used to populate the training data. In online autotuning, multiple versions of procedures and compiled and switched between using a procedure dispatcher at runtime.

to interpolate between training data when faced with new unknown inputs.

An example of an empirical approach to offline tuning is iterative compilation, which uses an offline training phase to perform an extensive search of the optimisation space of a program by compiling and evaluating the same program with different combinations of compiler transforms in order to select the set that provides the best performance.

Iterative compilation techniques has been successfully applied to a range of optimisation challenges, particularly when combined with machine learning techniques to focus the number of evaluations of training programs which are required [3]. Milepost GCC is a research compiler that uses iterative compilation to adjust compiler heuristics for optimising programs on different architectures [4]. Machine learning techniques are applied to model the large optimisation space based on static features extracted from the source code of training programs. A classifier predicts optimisation parameters for new programs by comparing the extracted program features against the training data.

An alternative approach to the problem of gathering sufficient training data is to distribute the task of collecting it by enabling the results of different program evaluations to be shared with a central remote database. This has been successfully applied to offline autotuners, in which a remote server is used to store and retrieve training data [8, 9]. The overhead of communicating with this remote server would be too great to use dynamically, a typical 150ms network round trip time in the performance critical path of a dynamic autotuner will cause a serious degradation of performance.

An offline tuning tool with particular relevance to this work is MaSiF [5], a static autotuner which uses iterative compilation techniques to perform a focused search of the optimisation space of FastFlow and Intel Thread Building Blocks, two popular Algorithmic Skeleton libraries. While sharing the same goal as MaSiF, the approach of this project focuses on performing optimisation space searching at runtime, and targets multi-GPU programming.

3.2 Dynamic optimisation

Dynamic optimisation is a very different approach to the problem of performance optimisation than offline

tuning. Whereas offline tuning typically requires an expensive training phase to search the space of possible optimisations, dynamic optimisers perform this optimisation space exploration at runtime, allowing programs to respond to dynamic features "online". This is a challenging task, as a random search of an optimisation space will typically result in many configurations with vastly suboptimal performance. In a real world system, evaluating many suboptimal configurations will cause a significant slowdown of the program. Thus a requirement of dynamic optimisers is that convergence time towards optimal parameters is minimised.

Existing dynamic optimisation research has typically taken a low level approach to performing optimisations. Dynamo is a dynamic optimiser which performs binary level transformations of programs using information gathered from runtime profiling and tracing [10]. While this provides the ability to respond to dynamic features, it restricts the range of optimisations that can be applied to binary transformations. These low level transformations cannot match the performance gains that higher-level parameter tuning produces.

One of the biggest challenges facing the implementation of dynamic optimisers is to minimise the runtime overhead so that it does not outweigh the performance advantages of the optimisations. A significant contributor to this runtime overhead is the requirement to compile code dynamically. Fursin et al. negated this cost by compiling multiple versions of a target subroutine ahead of time [6]. At runtime, execution switches between the available versions, selecting the version with the best performance. In practice, this technique massively reduces the optimisation space which can be searched as it is unfeasible to insert the thousands of different versions of a subroutine that are tested using offline tuning.

Many existing dynamic optimisation systems do not store the results of their efforts persistently, allowing the training data to be lost when the host process terminates. This approach relies on the assumption that either the convergence time to reach an optimal set of parameters is short enough to have negligible performance impact, or that the runtime of the host process is sufficiently long to reach an optimal set of parameters in good time. Neither assumption can be shown to fit the general case. This has led to the development of collective compilation techniques, which involve persistently storing the results of successive optimisation runs using a database [11].

PetaBricks attempts to capture higher-level optimisation decisions than are available to dynamic optimisers [12]. It consists of a language and compiler which allows programmers to express algorithms that target specific dynamic features, and to select which algorithm to execute at runtime. This provides a promising optimisation space but has the drawback of increasing programmer effort by requiring them to implement multiple versions of an algorithm tailored to different optimisation parameters.

SiblingRivalry poses an interesting solution to the challenge of providing sustained quality of service [13]. The available processing units are divided in half, and two copies of a target subroutine are executed simultaneously, one using the current best known configuration, and the other using a trial configuration which is to be evaluated. If the trial configuration outperforms the current best configuration, then it replaces it as the new best configuration. By doing this, the tuning framework has the freedom to evaluate vastly suboptimal configurations while still providing adequate performance for the user. However, a large runtime penalty is incurred by dividing the available resources in half.

3.3 SkelCL

Michel Steuwer, a research associate at the University of Edinburgh, developed SkelCL as an approach to high-level programming for multi-GPU systems [14, 15]. Steuwer, Kegel, and Gorlatch demonstrated an 11× reduction in programmer effort compared to equivalent programs implemented using pure OpenCL, while suffering only a modest 5% overhead [16].

The core of SkelCL comprises a set of parallel container data types for vectors and matrices, and an automatic distribution mechanism that performs implicit transfer of these data structures between the host and device memory. Application programmers express computations on these data structures using Algorithmic Skeletons that are parameterised with small sections of OpenCL code. At runtime, SkelCL compiles the Algorithmic Skeletons into compute kernels for execution on GPUs. This makes SkelCL an excellent candidate for dynamic autotuning, as it exposes both the optimisation space of the OpenCL compiler, and the high-level tunable parameters provided by the structure of Algorithmic Skeletons.

4 Methodology

The work required to complete this research has been broadly divided into three stages:

1. Modify SkelCL to enable the runtime configuration of optimisation parameters.
2. Evaluate the significance of dynamic features and different optimisation parameters to select the parameters which provide the most profitable optimisation space.
3. Implement a dynamic autotuner which searches and builds a persistent model of this optimisation space at runtime.

This section outlines the work required for each stage, listing some of the possible challenges and reasons that these will be overcome.

4.1 SkelCL Optimisation parameters

In the first stage, I will replace compile-time constant parameters in the SkelCL library with variable parameters, and add an API to support dynamically setting these parameters. Examples of parameters which can be set dynamically include the mapping of work items to threads and the OpenCL compiler configuration. I will also modify the container types of SkelCL so that dynamic features of input data structures can be extracted at runtime. Examples of dynamic features include the distribution of elements within a container and the data type of elements.

4.2 Feature and parameter evaluation

Pilot experiments will then be used to evaluate the effect of different parameters and features on performance by varying manually controlled stimuli across a range of

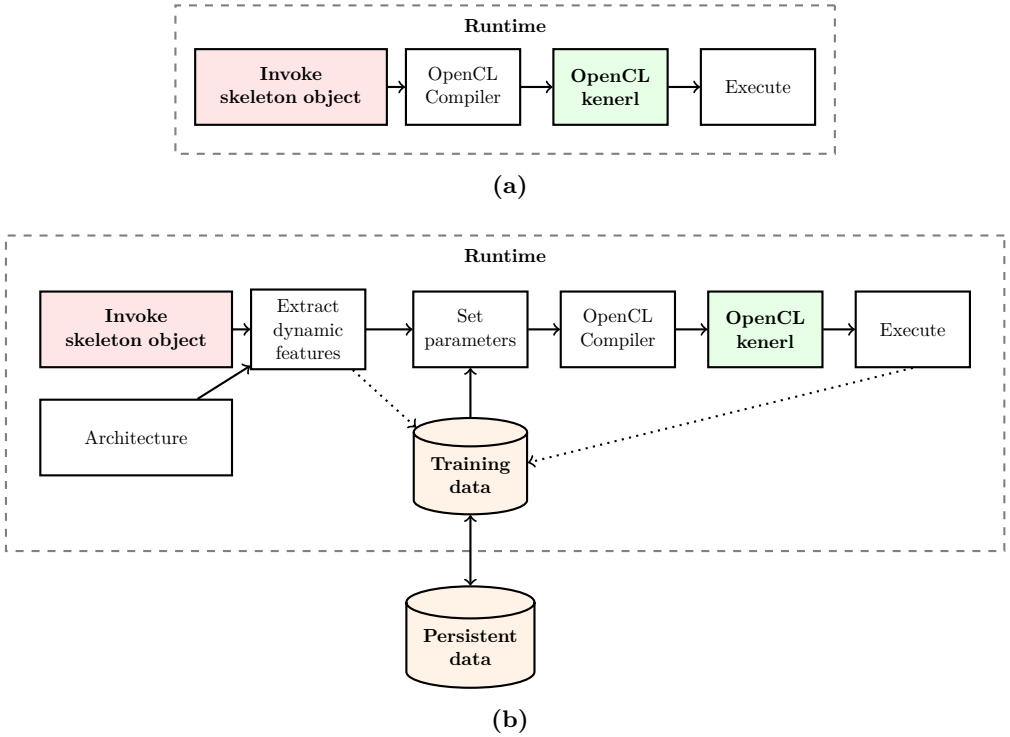

Figure 3: The skeleton invocation behaviour of current SkelCL (3a), and with dynamic autotuning (3b). When invoked, the dynamic features of a skeleton object are extracted and an online machine learning model recommends optimal parameters. The OpenCL compiler is invoked on this parameterised skeleton to generate an OpenCL kernel for execution on device. Profiler information is added to the training dataset.

different inputs and measuring their impact on performance. Statistical methods will be used to analyse these results in order to isolate the parameters and features with the greatest performance impact, and generate an optimisation space. For example, Principle Component Analysis can be used to reduce the dimensionality of this optimisation space by orientating the space along the directions of greatest variance.

This exploratory phase provides opportunities for the novel use of dynamic features for the purpose of autotuning Algorithmic Skeletons, since previous research has focused on offline tuning and so has been restricted to the set of features which can be either statically determined or approximated. In addition to dynamic features, SkelCL compiles OpenCL kernels at runtime immediately before execution. This enables the setting of arbitrary optimisation parameters without having to use the multi-versioning techniques of many state of the art dynamic optimisers, in much multiple versions of a subroutine are included in a compiled binary.

4.3 Dynamic autotuner implementation

In the final stage, I will construct a dynamic autotuner that uses the features and parameters selected in the exploratory phase. To the best of our knowledge, this will be the first attempt to develop a dynamic autotuner for Algorithmic Skeletons. The focus of the implementation will be to exploit the advantages of dynamic features to provide improved performance over existing static Algorithmic Skeleton autotuners, and to exploit the high-level abstractions of Algorithmic Skeletons to provide improved performance over existing dynamic optimisers.

Implementing a dynamic autotuner poses a number of difficult challenges. The primary challenge is to minimise the runtime overhead so that it does not outweigh the performance gains of the optimisations themselves. The proposed approach to dynamically autotune SkelCL will overcome one of the most significant overheads associated with dynamic optimising: that of instrumenting the code for the purpose of profiling and tracing. Since Algorithmic Skeletons coordinate muscle functions, it is possible to forgo many of the profiling counters that dynamic optimisers require by making assumptions about the execution frequency of certain code paths, given the nature of the skeleton. Profiling counters will be placed by hand at critical points in the code, allowing the frequency of counter increments to be minimised.

The convergence time of autotuning can be improved by saving the results of trial configurations persistently in a central database. This provides two advantages: first, it allows the results of autotuning to be used by future program runs; second, it allows the result of autotuning to be shared amongst any program which uses the SkelCL library. The challenge of implementing this persistent data storage is that results must be stored efficiently and compactly, to allow for indefinite scaling of the dataset as future results are added. Increasing the size of the training dataset also increases the time required to compute new results, and there is additional latencies associated with reading and writing data to and from disk.

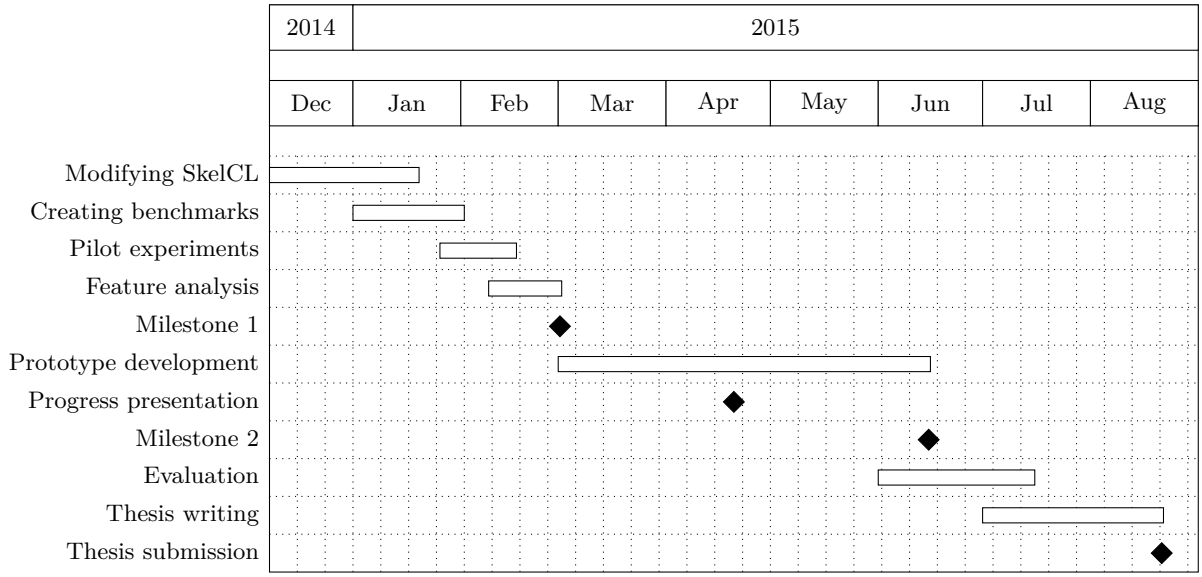

Figure 4: Project schedule Gantt chart.

5 Evaluation

My hypothesis is that the performance of Algorithmic Skeletons will be improved by using dynamic autotuning. This hypothesis will be supported or rejected by empirical evidence collected from an evaluation of the prototype implementation. Experimental evidence and standard empirical methods will be used to evaluate the performance of SkelCL across a range of representative benchmarks.

Experimental performance results will be compared against:

- a baseline implementation provided by an unmodified SkelCL implementation. This will compare the speedup of the autotuned version over the baseline;
- a hand-tuned "oracle" implementation using an optimal configuration discovered through an offline exhaustive search of the optimisation space. This will measure the autotuner's ability to converge towards optimal parameters over time;
- a "gold standard" implementation using hand tuned OpenCL without the SkelCL abstractions. This will compare the performance cost of using the high-level Algorithmic Skeleton abstractions against the reduction in programmer effort required to implement the equivalent program in pure OpenCL.

An important factor in the quality of the evaluation will be selecting performance benchmarks that are representative of a range of real world use cases. For this purpose, I will use existing SkelCL benchmarks which have been used in previous research: Mandelbrot sets [14], Sobel Edge Detection [15], and List-mode Ordered Subset Expectation Maximisation [17]. Additionally, a standard benchmark suite for heterogeneous computing such as Rodinia [18] can be used by first porting the implementations to use the SkelCL library.

The stochastic nature of autotuning and machine learning techniques means that the performance evaluation of representative benchmarks must be performed with strict statistical rigour, using appropriate techniques for profiling benchmark performance over multiple iterations [19]. The evaluation approach must carefully isolate independent variables and provide a controlled environment for testing the effects of altering them.

In addition to the overall performance evaluation of the dynamic autotuner, additional measurements can be made to isolate and record the overhead introduced by the runtime, the amount of time required to converge on optimal configurations, and the ability of the dynamic optimiser to adapt to changes in the runtime environment. This last measurement may require execution of the benchmarks on multiple different hardware configurations so as to measure the system's ability to adapt to different environments.

6 Work plan

The schedule for this project is shown in Figure 4. In addition to the Intermediate Progress Presentation in April, two personal milestones will be used to provide ongoing progress checks. The first milestone corresponds with the end of the exploratory phase of development. The second milestone is placed at the end of the implementation stage, marking the point at which the code base is frozen so as to focus on an extended evaluation. All source code and experimental results will be tracked using the git version control system[1]. GitHub[2] will be used to track issues and milestone progress.

7 Conclusion

Algorithmic Skeletons have been shown to improve programmer effectiveness by providing the necessary high-level abstractions for parallel programming. The SkelCL library has been used to implement high performance medical imaging applications using shorter, better structured programs that perform within 5% of a hand tuned

[1] `http://git-scm.com/`
[2] `https://github.com/`

OpenCL implementation [16].

A dynamic autotuner for SkelCL will improve the performance of these applications by providing optimisations which, while targeting specific applications and inputs, are transferable across problem domains. Through the novel combination of persistent training data with the dynamic compilation of OpenCL kernels and runtime features, it will be possible to implement a dynamic autotuner for SkelCL with minimal runtime overhead.

Our approach will be to first modify SkelCL so that it enables runtime configuration of optimisation parameters, then to evaluate the significance of dynamic features and optimisation parameters, before implementing a dynamic autotuner which searches and builds a persistent model of this optimisation space at run.

We are ideally suited for tackling this difficult problem at University of Edinburgh, with expert researchers in the fields of Algorithmic Skeletons, iterative compilation, and machine learning based optimisation. Previous research at the University of Edinburgh has addressed the static autotuning of Algorithmic Skeletons [5, 20], which will provide a point of reference for comparing a dynamic autotuning approach.

References

[1] Murray I Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman London, 1989.

[2] Murray I Cole. "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming". In: *Parallel Computing* 30.3 (Mar. 2004), pp. 389–406.

[3] Felix Agakov et al. "Using Machine Learning to Focus Iterative Optimization". In: *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2006, pp. 295–305.

[4] Grigori Fursin et al. "Milepost GCC: Machine Learning Enabled Self-tuning Compiler". In: *International Journal of Parallel Programming* 39.3 (Jan. 2011), pp. 296–327.

[5] Alexander Collins et al. "MaSiF: Machine Learning Guided Auto-tuning of Parallel Skeletons". In: *20th Annual International Conference on High Performance Computing - HiPC* (Dec. 2013), pp. 186–195.

[6] Grigori Fursin et al. "A Practical Method for Quickly Evaluating Program Optimizations". In: *High Performance Embedded Architectures and Compilers* 3793 (2005), pp. 29–46.

[7] Kamen Yotov et al. "A Comparison of Empirical and Model-driven Optimization". In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation - PLDI '03*. New York, New York, USA: ACM Press, 2003, pp. 63–76.

[8] Grigori Fursin et al. "Collective Mind: Towards practical and collaborative auto-tuning". In: *Scientific Programming* 22.4 (2014), pp. 309–329.

[9] Rafael Auler et al. "Addressing JavaScript JIT engines performance quirks: A crowdsourced adaptive compiler". In: *Compiler Construction*. Springer, 2014, pp. 218–237.

[10] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. "Dynamo: A Transparent Dynamic Optimization System". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation - PLDI '00*. Vol. 35. 5. New York, NY, USA: ACM, 2000, pp. 1–12.

[11] Grigori Fursin and Olivier Temam. "Collective Optimization". In: *High Performance Embedded Architectures and Compilers* 5409 (Dec. 2009), pp. 34–49.

[12] Jason Ansel et al. "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Vol. 44. 6. New York, NY, USA: ACM, 2009, pp. 38–49.

[13] Jason Ansel and Una-may O Reilly. "SiblingRivalry: Online Autotuning Through Local Competitions". In: *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. ACM, 2012, pp. 91–100.

[14] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. "SkelCL - A Portable Skeleton Library for High-Level GPU Programming". In: *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. IEEE, May 2011, pp. 1176–1182.

[15] Michel Steuwer and Sergei Gorlatch. "SkelCL: Enhancing OpenCL for High-Level Programming of Multi-GPU Systems". In: *Parallel Computing Technologies* 7979 (2013), pp. 258–272.

[16] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. "Towards High-Level Programming of Multi-GPU Systems Using the SkelCL Library". In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. Ieee, May 2012, pp. 1858–1865.

[17] Michel Steuwer and Sergei Gorlatch. "High-level Programming for Medical Imaging on Multi-GPU Systems Using the SkelCL Library". In: *Procedia Computer Science* 18 (Jan. 2013), pp. 749–758.

[18] Shuai Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Vol. 2009. c. IEEE, Oct. 2009, pp. 44–54.

[19] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. Vol. 42. 10. New York, NY, USA: ACM, Oct. 2007, p. 57.

[20] Alexander Collins, Christian Fensch, and Hugh Leather. "Auto-Tuning Parallel Skeletons". In: *Parallel Processing Letters* 22.02 (June 2012), p. 1240005.