Dynamic Autotuning of Algorithmic Skeletons

Informatics Research Proposal
Chris Cummins

Abstract. The rapid transition towards multicore hardware has left application programmers requiring higher-level abstractions for coping with the complexity of parallel programming. Algorithmic Skeletons provide such abstractions but, without extensive tuning, typically cannot compete with the performance of hand optimised code. This paper proposes developing 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 be used to create a feedback loop of constant testing and evaluation of skeleton parameters across the lifespan of programs. It will use dynamic features extracted from muscle functions and input data to maximise runtime performance. 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 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 search the huge space of possible optimisation parameters by repeatedly evaluating different configurations to select the configuration which gives the best performance.

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. Online machine learning will enable the runtime exploration of the optimisation space while selecting configurations that maximise performance.

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. Contributions of a successful project will include:

- a first attempt to apply the principles of online machine learning to the runtime selection of Algorithmic Skeleton optimisation parameters;
- a dynamic autotuner which specifically targets features relevant to multi-GPU parallelism;
- experimental results comparing the performance of dynamically autotuned SkelCL against hand tuned OpenCL across multiple benchmarks.

2 Motivation

Consider a recursive merge sort algorithm. The algorithm takes an input list, and returns a sorted permutation. It checks the length of the input list to see if it is short enough to solve directly using a linear sorting method, or whether it should split it into multiple sub-lists and sort them recursively before combining the results. This computational pattern is abstracted by the Divide and Conquer skeleton, which can be effectively parallelised by executing each recursion as a new parallel task. The Divide and Conquer skeleton takes an input of type T_i and returns an output of type T_o, and is parameterised with definitions for four muscle functions:

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

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

$$n = k^d - 1$$

On real hardware, the number of available processing units limits the number of tasks which can be effectively executed in parallel. Since the Divide and Conquer pattern does not constrain the maximum depth that an algorithm may recurse to, the skeleton author must impose a maximum "parallelisation depth" to prevent the task switching costs skeletons which recurse deeply. Recursion above the parallelisation depth causes the creation of parallel tasks, below this depth, recursion occurs sequentially.

The remainder of this section describes experimental data that consider the effect of varying input conditions on the optimal parallelisation depth.

2.1 Experimental setup

I implemented a Divide and Conquer skeleton and parameterised it with muscle functions to implement merge sort. I parallelised the skeleton 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. I varied the parallelisation depth over the range 0 (sequential) through 10.

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 is used. Figure 1b shows the effect of varying the size and data type of the input vector.

We observe that changes in the input and muscle function definitions can have a significant impact when determining the optimal parallelisation depth parameter. Since the skeleton author cannot determine the types and values of inputs *a priori*, they must resort to picking a value which they expect to provide best average case performance, or devising a technique which sets this optimisation parameter at runtime as a response to different inputs. This proposal describes a solution using the latter approach.

3 Background

Relevant approaches to the problem of optimisation parameter tuning 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 has been generated beforehand. 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 select optimisation parameters based on empirical data gathered from prior evaluation of many different parameter configurations. In both cases, a performance function $f(c,p)$ models the relationship between a parameter configuration c, a program p, and some profitability goal. The purpose of the offline tuning phase is to select the configuration $c_{optimal}$ which maximises the output of the performance model:

$$c_{optimal} = \arg\max_{p} f(c,p)$$

The quality of predictive models 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 to interpolate between training data when faced with

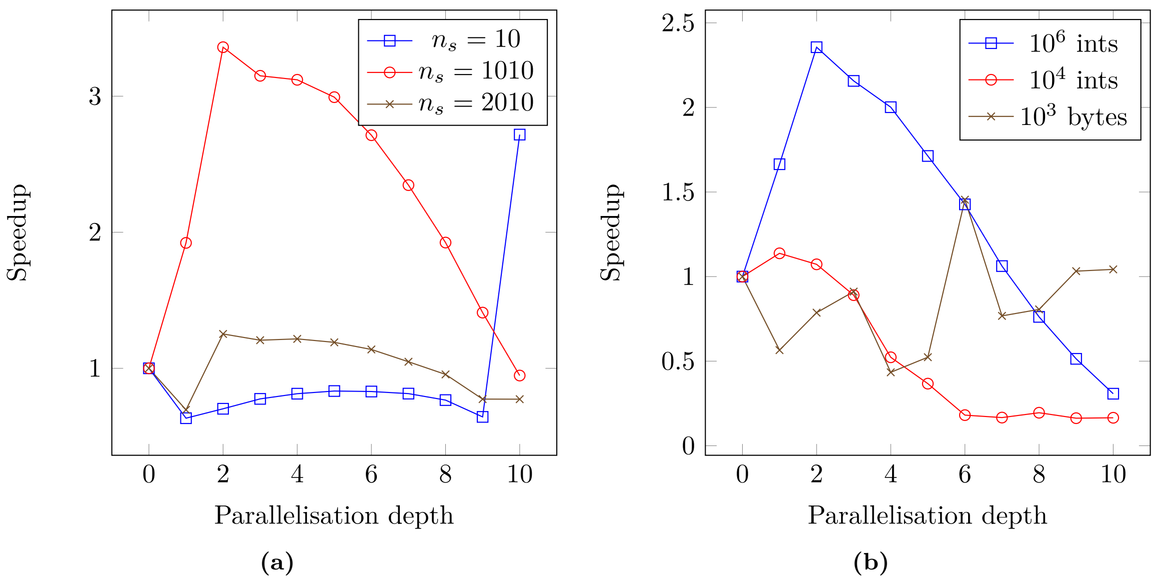


Figure 1: The performance impact of dynamic features on the optimisation parameter "parallelisation depth": in 1a, as a function of split size n_s; in 1b, as a function of input type and size. In both cases, no parallelisation depth value can provide optimal performance for all inputs.

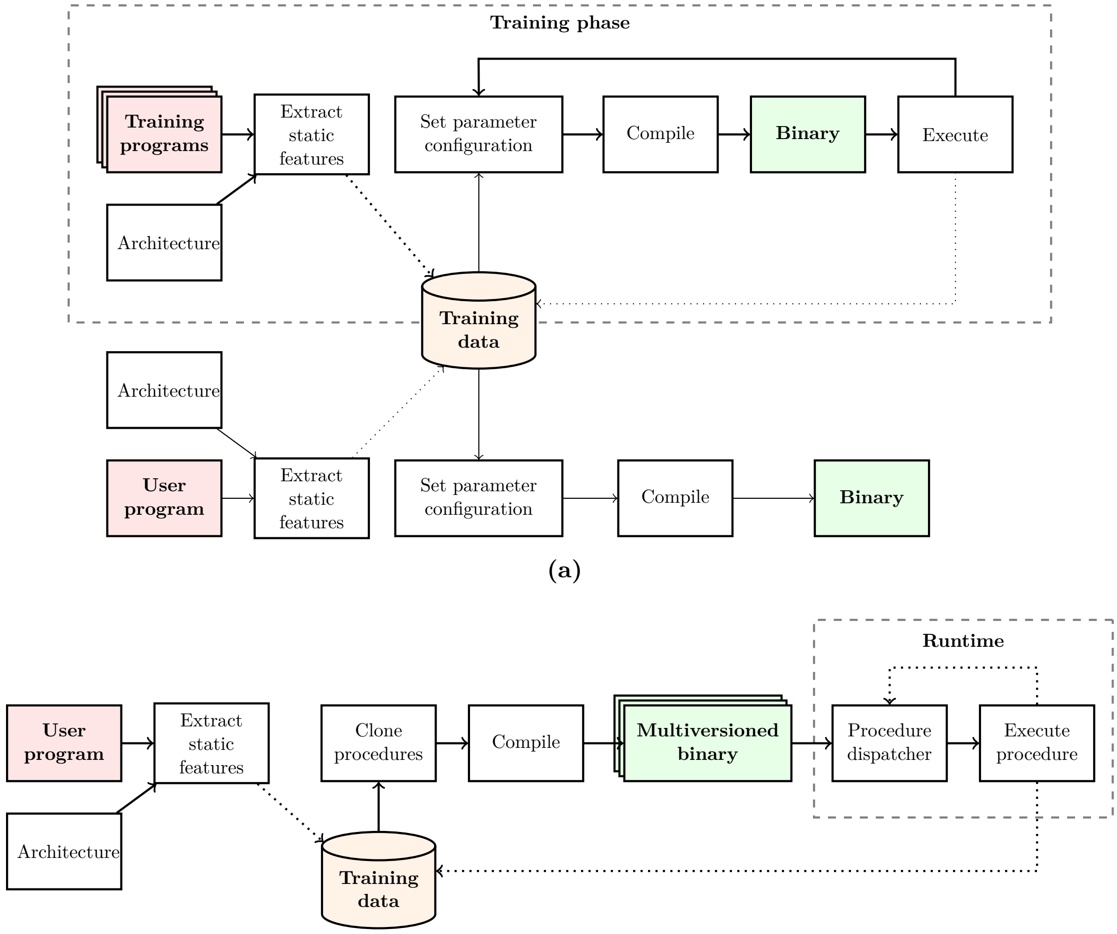


Figure 2: Two approaches to static autotuning: in 2a, offline autotuning using a separate training phase, as used in [3–5]; in 2b, online autotuning using procedure multiversioning, as used in [6]. In offline autotuning, training programs are used to populate the training dataset. In online autotuning,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

new unknown inputs. Offline machine learning techniques have proven popular as an approach to reducing the number of evaluations of training programs which are required. In [3], Agakov et al. use Markov Chains to learn the most profitable areas of the optimisation space of source to source transformations.

In [4], Fursin et al. present Milepost GCC, a self-tuning research compiler that selects optimisations based on static program features. The approach proposed in this paper differs by performing this search of the optimisation space during normal program runs, instead of requiring costly offline training.

The task of collecting training data for offline autotuners has been effectively distributed in [8, 9]. A remote server contains a central store of training data which is retrieved and contributed to by distributed clients; this allows multiple clients to share the results of optimisations. The overhead of communicating with a remote server would be too great to use dynamically, a typical 150ms network round trip time in the critical path of a program would cause a serious performance degradation.

Collins et al. presented the offline autotuner MaSiF in [5]. Principle Component Analysis was used to reduce the search size of the optimisation space for FastFlow and Intel Thread Building Blocks, two popular Algorithmic Skeleton libraries. They achieved 89% of the oracle performance by searching 0.05% of the optimisation space. This paper differs by targeting the feature space of heterogeneous parallelism and using online machine learning instead of offline training.

A system-level overview of offline autotuning is shown in Figure 2a.

3.2 Dynamic optimisation

Dynamic optimisers improve the performance of programs by exploring the optimisation space at runtime. Implementing an effective dynamic optimiser is a challenging task, as the need to search the optimisation space must be balanced against the need to provide quality of service by avoiding suboptimal configurations. 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 must be minimised.

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 such as function inlining, and cannot offer the performance gains that higher-level parameter tuning such as setting the size of thread pools provides.

Fursin et al. negated the cost of dynamic compilation in [6] by compiling multiple versions of target subroutines ahead of time. At runtime, execution is switched between the available versions which are ranked by performance. Figure 2b shows a system-level overview of this approach. In practice, this technique massively reduces the size of 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. The approach proposed in this paper enables online searching of the entire optimisation space by compiling OpenCL kernels at runtime.

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 cost, or that the run time of the 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 in a persistent database [11].

In [12], Ansel et al. attempts to capture high-level algorithmic choices using PetaBricks, 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 has the disadvantage of increasing programmer effort by requiring them to implement multiple versions of an algorithm tailored to different optimisation parameters. A dynamic autotuner for Algorithmic Skeletons will be able to exploit these high-level optimisations without increasing programmer effort, by hiding the complexity of optimisations within the SkelCL library.

SiblingRivalry [13] is a dynamic optimiser that provides sustained quality of service by dividing the available processing units in half. When invoked, 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. This allows for the low cost evaluation of suboptimal configurations, but incurs a large runtime penalty 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% performance overhead [16].

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 by parameterising Algorithmic Skeletons 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 and the extraction of dynamic features.
2. Evaluate the significance of optimisation parameters and dynamic features. Use the results of this evaluation to select an online machine learning algorithm for parameter tuning.
3. Implement a low overhead dynamic autotuner which uses this online machine learning model to select optimisation parameters at runtime.

This section outlines the work required for each stage, listing some of the possible challenges and approaches to overcoming them.

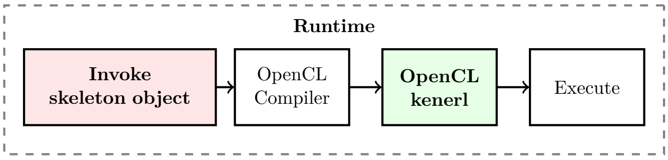
4.1 Model features and parameters

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. This will provide the set of actions that can be taken based on performance predictions of the machine learning model. Examples of parameters which can be set dynamically include the mapping of work items to threads and the OpenCL compiler configuration. I will then modify the container types of SkelCL so that properties of input data structures can be extracted at runtime. These will provide the input features to the machine learning model. Examples of dynamic features include the dimensionality and types of data.

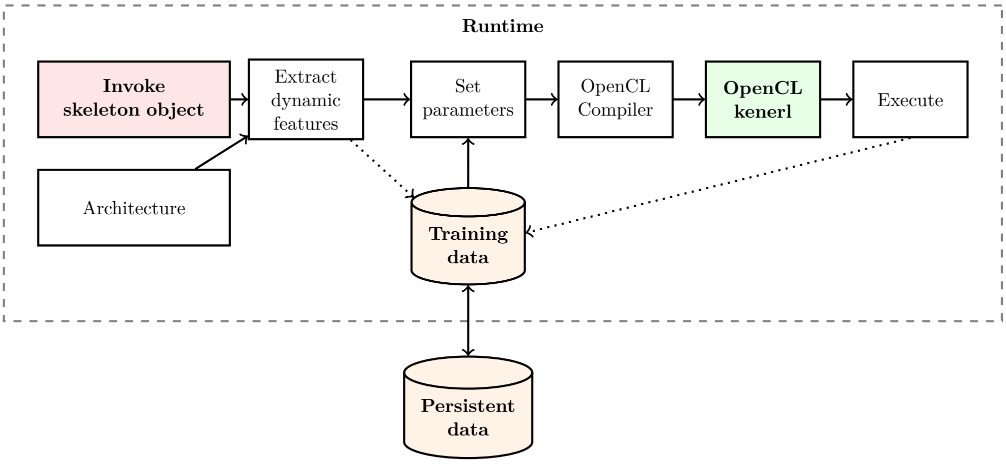
4.2 Online machine learning

Exploratory experiments will then be used to evaluate the effect of different parameters and features by varying test stimuli across a range of different inputs and measuring their impact on performance. Statistical methods will be used to analyse these results and isolate the parameters and features with the greatest performance impact. Principle Component Analysis can be used to reduce the dimensionality of this optimisation space by orientating the space along the directions of greatest variance.

The purpose of this exploratory phase is to identify the parameters and features which can be used to most effectively search the optimisation space, and to guide the choice of an online machine learning algorithm. The goal of the online machine learning algorithm is to generate parameter configurations which will provide the best performance for a given skeleton and input dataset. Every time the user invokes a skeleton object, the machine learning algorithm must:



(a)



(b)

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. Profiling information is gathered during execution and added to the training dataset.

1. Predict the parameter configuration which will provide the best performance based on the features.
2. Compile and execute the skeleton with this configuration.
3. Measure the true performance of the skeleton and use this result to refine future predictions.

The primary challenge in developing the machine learning algorithm is to balance the potentially conflicting requirements to:

- offer the best performance configurations to maximise performance;
- search the large optimisation space to avoid becoming trapped in local minima;
- build statistical confidence in training data through repeated invocations of identical configurations.

4.3 Dynamic autotuner implementation

In the final stage, I will implement a dynamic autotuner which uses the online machine learning algorithm, 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 using online machine learning for Algorithmic Skeletons. The goal 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. Figure 3b shows a system-level overview of dynamically autotuned SkelCL.

A major challenge when implementing online autotuning 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 a significant overhead associated with dynamic optimising: that of instrumenting the code to enable profiling and tracing. Since Algorithmic Skeletons coordinate muscle functions, it is possible to forgo many of the counters required for performance profiling by making assumptions about the execution frequency of certain code paths given the nature of the skeleton. I will place profiling counters by hand at critical points in the SkelCL library to minimise the frequency of counter increments.

The convergence time of autotuning can be improved by using a central database to store optimisation results. 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 across 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.

5 Evaluation

My hypothesis is that the performance of Algorithmic Skeletons can be improved using dynamic autotuning. This hypothesis will be supported or rejected by empirical

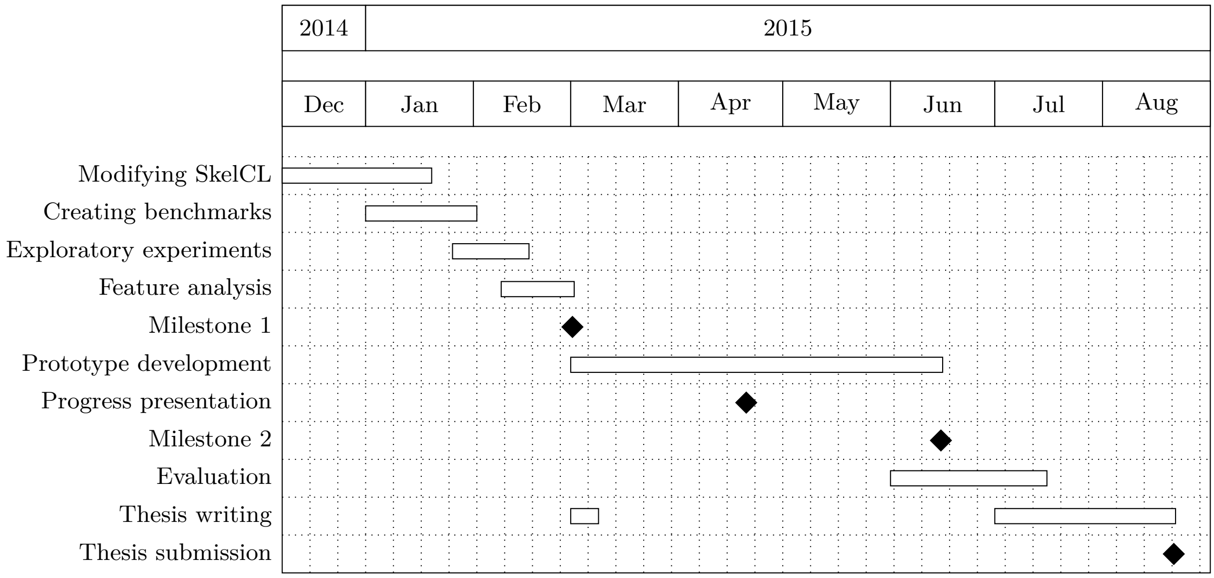


Figure 4: Project schedule Gantt chart.

evidence collected from an evaluation of the implemented prototype. I will use experimental evidence and standard empirical methods to evaluate the performance of SkelCL across a range of representative benchmarks.

I will compare experimental performance results 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 exhaustive search of the optimisation space. This will measure the ability of the autotuner 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] could 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 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 ability of the autotuner to adapt to different environments.

6 Work plan

Figure 4 shows the schedule for this project. In addition to the Intermediate Progress Presentation in April, I have created two personal milestones to provide progress checks. The first milestone corresponds with the end of the exploratory phase of development. The second milestone is at the end of the implementation stage. It marks the point at which development of the code base will freeze so as to enable an extended evaluation. I will use the git version control system[1] to track all source code and experimental data. I will use GitHub[2] to track issues and milestone progress.

7 Conclusion

This paper proposes the development of a dynamic autotuner for SkelCL which uses online machine learning techniques to explore the space of optimisation parameters and recommend optimal configurations based on dynamic features. This will be the first attempt to implement a dynamic autotuner using online machine learning for Algorithmic Skeletons, and will enable runtime performance tuning without the requirement of long offline training periods associated with state of the art Algorithmic Skeletons autotuners.

Our approach will be to first modify SkelCL so that it enables runtime configuration of optimisation parameters and dynamic extraction of features. Then we will

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

evaluate the significance of dynamic features and optimisation parameters to develop an effective online machine learning algorithm. We will implement this as a dynamic autotuner which searches and builds a persistent model of this optimisation space at runtime. We will compare the performance of this dynamic autotuner across a number of benchmarks against a baseline unmodified SkelCL and a gold standard hand-tuned OpenCL implementation.

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 OpenCL implementation [16]. As the trend towards increasingly parallel hardware continues, the demand for high-performance parallel programming abstractions will continue.

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.