Dynamic Autotuning of Algorithmic Skeletons

Informatics Research Proposal
Chris Cummins
December 2014

Abstract.

1 Introduction

The computing world is increasingly finding parallelism as the only viable approach to maintaining continued performance improvements in the era of multicore hardware. 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 is a manual process which requires exhaustively searching the optimisation space to select optimal parameters.

The aim of this project is to demonstrate that the tuning of optimisation parameters can be successfully performed at runtime. This will enable self-tuning programs which adapt to their execution environment by selecting optimal parameters dynamically. Such configurations of parameters can be learned over time, allowing each successive iteration of a program to benefit from its predecessors.

The case for dynamically autotuning applications is strong. There are many factors which contribute to the performance of programs which cannot be determined by program developers. For example,

As a result, performance optimisation requires the programmer to either overfit the choice of parameters to optimise for a specific task and environment, or laboriously create heuristics which segment the optimisation space into regions of identical configurations. Both approaches have significant drawbacks: optimising for a specific task and environment creates brittle and non-portable optimisations that do not generalise to other architectures and inputs; and the task of creating heuristics which cover every possible combination of factors is prohibitively time consuming for developers.

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

- a dynamic autotuner will improve the performance of Algorithmic Skeletons in the general case, by selecting optimisations which target specific dynamic features;
- a dynamic autotuner will provide better average performance than a statically tuned equivalent implementation across different inputs, by adapting to features which can only be determined dynamically.

These hypotheses can be referred to respectively as the claims *specialisation* and *generalisation*. It can be inferred from these that a dynamic autotuner cannot provide better performance than a statically tuned equivalent implementation for a *fixed* input, since the extra instructions that implement the dynamic autotuning behaviour present a performance overhead. The reduction of this overhead is one of the greatest challenges facing the development of dynamic autotuners. The novelty of my solution is to reduce parameter convergence time by implementing a dynamic autotuner for Algorithmic Skeletons which can store the results of successive iterations persistently, across program runs. An evaluation of a successful implementation will contribute empirical evidence supporting the two hypotheses.

The rest of the document is structured as follows: Section 2 contains the motivation for this research; Section 3 briefly outlines related work; Sections 4 and 5 describe the methodology and evaluation plans for this research; Section 6 contains the work plan; followed by the conclusion in Section 7.

2 Motivation

TODO. This section will contain a brief outline the results of two tests on an example parallel merge sort implementation (DaC skeleton). The first test will show the strong and uneven relationship that TWO optimisation parameters have on performance; this is to demonstrate that these optimisation spaces are complex and require automated searching (i.e. it can't just be done by hand). The second test will show the difference in the optimal value of a SINGLE optimisation parameter as a function of different input types and sizes; this is to demonstrate that these optimisation spaces are influenced by dynamic features.

3 Background

This section briefly outlines some of the most closely related pieces of work that address the issue of improving software performance through the selection of optimal parameters. These can broadly be categorised as either offline tuning, or dynamic optimisation.

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 model $f(p, x)$ maps the relationship between a set of parameters p, a specific problem x, and some metric of profitability. The purpose of the offline tuning phase is to select the set of parameters $p_{optimal}$ which maximises 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 model 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. 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 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 programs with different combinations of compiler transforms in order to select the set which 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 static 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 the overhead of a round trip to a remote server is performed offline [5, 6]. A typical 150ms network round trip time in the performance critical path of a dynamic autotuner will cause a serious degradation of performance, given typical system performance in excess of 100 MIPS.

An offline tuning tool with particular relevance to this work is MaSiF [7], 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.

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 [8]. 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 [9]. 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 work to die along with the host process. This approach relies on the assumption that either that 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 [10].

PetaBricks attempts to capture higher level optimisation decisions than are available to dynamic optimisers [11]. 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 [12]. 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.

4 Methodology

This research will use the Algorithmic Skeleton library SkelCL[1] as the base platform upon which a dynamic autotuner will be developed. Michel Steuwer, a research associate at the University of Edinburgh, developed SkelCL as an approach to high-level programming for multi-GPU systems [13, 14]. Steuwer, Kegel, and Gorlatch demonstrated an 11× reduction in programmer effort compared to equivalent programs implemented with pure OpenCL, while suffering only a modest 5% overhead [15].

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 OpenCL code 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. SkelCL offers the unique advantage of being able to amortise many of the costs associated with dynamic compilation due to its JIT-like nature of compiling OpenCL kernels immediately before execution.

The methodology used in this research will approach the problem in three stages:

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

In the first stage, the SkelCL library will be changed to support dynamic parameter setting. This will involve a number of modifications to replace compile-time constant parameters such as the mapping of threads to work groups and the OpenCL compiler configuration with equivalent variable parameters which can be set at runtime.

Pilot experiments will then be used to evaluate the effect of different parameters on performance, by manually varying these runtime parameters across a range of different inputs and measuring their impact on performance. Statistical methods will be used to analyse these results in order to isolate the parameters with the greatest performance impact. In previous research, Principle Component Analysis has been used effectively to reduce the dimensionality of optimisation spaces. This exploratory phase provides opportunities for the novel use of dynamic features for the purpose of autotuning Algorithmic Skeletons. 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. A dynamic autotuner exposes many new features which cannot be statically determined, such as:

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

The final stage will construct a dynamic autotuner that uses the features 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

[1] `http://skelcl.uni-muenster.de/`

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 implementation challenges. The primary challenge is to minimise the runtime overhead of the dynamic autotuner 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. Additionally, the placement of profiling counters can be optimised manually.

The convergence time of autotuning can be improved by saving the results of trial configurations persistently in a central database. This provides two primary advantages: first, it allows the results of autotuning to be used by future invocations of the program; second, it allows the result of autotuning to be shared amongst different programs. The challenge of implementing this persistent data storage is that results must be stored efficiently and compactly, this is 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 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 range of representative benchmarks.

Experimental performance results will be compared against: baseline performance provided by an unmodified SkelCL implementation; and a hand-tuned "oracle" implementation using an optimal configuration discovered through an offline exhaustive search of the optimisation space. Performance could also be compared against a hand tuned OpenCL implementation, in order to compare the performance cost of using the high-level Algorithmic Skeleton abstractions against the 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 so that they are representative of a range of real world use cases. These benchmarks should be chosen at an early stage in the project, as there will be a tight feedback loop between implementation and evaluation during the prototype development.

The stochastic nature of autotuning and machine learning techniques, the performance evaluation of these representative benchmarks must be performed with strict statistical rigour, using appropriate techniques for detecting outliers and inferring confidence intervals of performance results [16].

Other measurable success metrics include: the overhead introduced by the runtime; the amount of time required to converge to a sufficiently good configuration; and the ability of the dynamic optimiser to adapt to changes in dynamic features (e.g. system load). All of these metrics will be evaluated by profiling performance benchmarks.

There will be a tight feedback loop between implementation and evaluation throughout the lifetime of the project.

6 Work plan

The schedule for this project is shown in Figure 1. 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 implementation code base is frozen in order to facilitate the evaluation. All source code and experimental results will be tracked using the git version control system[1], and GitHub[2] will be used to track issues and milestone progress.

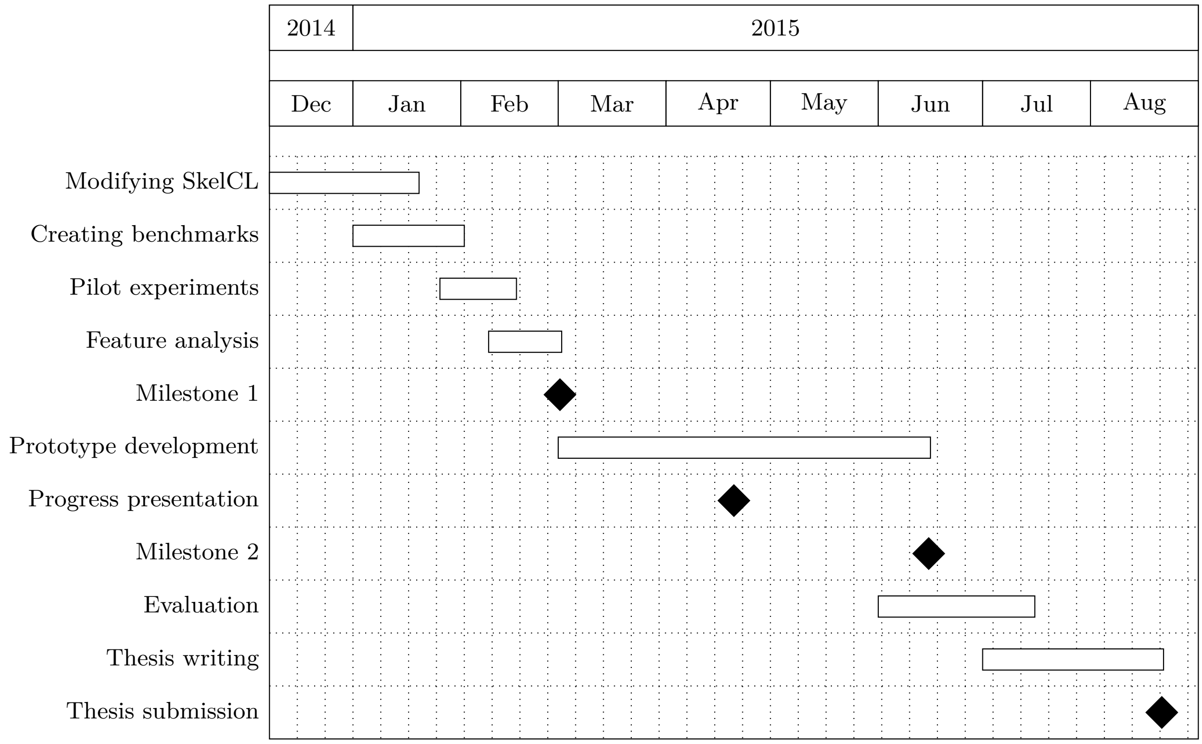

Figure 1: Project schedule Gantt chart.

7 Conclusion

Existing research has shown that Algorithmic Skeletons improve programmer effectiveness for a range of tasks from general purpose computing, to bioinformatics, and complex simulations. For example, the SkelCL library has been used to implement high performance medical imaging applications. A dynamic autotuner for SkelCL will improve the performance of these applications, and provide a starting point for future research into the online autotuning of Algorithmic Skeletons.

We are ideally suited for tackling this difficult problem at University of Edinburgh. Not only have academic members been responsible for introducing and developing Algorithmic Skeletons [1, 2, 17], but there is a large and active research interest in iterative compilation and machine learning based optimisation [3, 4, 9]. Previous research at the University of Edinburgh has also approach the static autotuning of Algorithmic Skeletons [7, 18], which will provide a solid source of inspiration and an interesting counterpoint for evaluating the performance of a dynamic autotuning approach.

While iterative compilation is a very well studied field, fewer papers have been published about dynamic optimisation. Therefore work in this field has a greater chance of influencing

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

future research, besides the primary benefit of improving the performance of algorithmic skeletons.

References

[1] Murray I Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman London, 1989. URL: `http://homepages.inf.ed.ac.uk/mic/Pubs/skeletonbook.pdf`.

[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. URL: `http://linkinghub.elsevier.com/retrieve/pii/S0167819104000080`.

[3] F. 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. URL: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1611549`.

[4] Grigori Fursin et al. "MILEPOST GCC: machine learning based research compiler". In: *GCC Summit*. 2008.

[5] Grigori Fursin et al. "Collective Mind: towards practical and collaborative auto-tuning". In: *Scientific Programming* 22.4 (2014), pp. 209–329.

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

[7] Alexander Collins et al. "MaSiF: Machine learning guided auto-tuning of parallel skeletons". In: *20th Annual International Conference on High Performance Computing* (Dec. 2013), pp. 186–195. URL: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6799098`.

[8] Vasanth Bala and Evelyn Duesterwald. "Dynamo: A Transparent Dynamic Optimization System". In: *ACM SIGPLAN Notices*. ACM, 2000, pp. 1–12.

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

[10] Grigori Fursin and Olivier Temam. "Collective optimization". In: *High Performance Embedded Architectures and Compilers* (Dec. 2009), pp. 34–49. URL: `http://portal.acm.org/citation.cfm?doid=1880043.1880047`.

[11] Jason Ansel et al. "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. Vol. 44. 6. ACM, 2009, pp. 38–49.

[12] Jason Ansel and Una-may O Reilly. "SiblingRivalry: Online Autotuning Through Local Competitions". In: *International Conference on Compilers Architecture and Synthesis for Embedded Systems*. 2012.

[13] 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. URL: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6008967`.

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

[15] 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. URL: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6270864`.

[16] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically rigorous java performance evaluation". In: *ACM SIGPLAN Notices* 42.10 (Oct. 2007), p. 57. URL: `http://portal.acm.org/citation.cfm?doid=1297105.1297033`.

[17] Anne Benoit et al. "Flexible Skeletal Programming with eSkel". In: *Euro-Par 2005 Parallel Processing*. Springer, 2005, pp. 761–770.

[18] Alexander Collins, Christian Fensch, and Hugh Leather. "Auto-Tuning Parallel Skeletons". In: *Parallel Processing Letters* 22.02 (June 2012), p. 1240005. URL: `http://www.worldscientific.com/doi/abs/10.1142/S0129626412400051`.