Autotuning Stencils Codes with Algorithmic Skeletons

Chris Cummins

Master of Science by Research
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2018

Abstract

The physical limitations of microprocessor design have forced the industry towards increasingly heterogeneous architectures to extract performance. This trend has not been matched with software tools to cope with such parallelism, leading to a growing disparity between the levels of available performance and the ability for application developers to exploit it.

Algorithmic skeletons simplify parallel programming by providing high-level, reusable patterns of computation. Achieving performant skeleton implementations is a difficult task; developers must attempt to anticipate and tune for a wide range of architectures and use cases. This results in implementations that target the general case and cannot provide the performance advantages that are gained from tuning low level optimisation parameters.

To address this, I present OmniTune — an extensible and distributed framework for runtime autotuning of optimisation parameters. Targeting the workgroup size of OpenCL kernels, I demonstrate an implementation of OmniTune for stencil codes on CPUs and multi-GPU systems. I show in a comprehensive evaluation of 2.7×10^5 test cases that simple heuristics cannot provide portable performance across the range of architectures, kernels, and datasets which algorithmic skeletons must target.

OmniTune uses procedurally generated synthetic benchmarks and machine learning to predict workgroup sizes for unseen programs. In an evaluation of 429 combinations of programs, architectures, and datasets, with up to 7.3×10^3 parameter values for each, OmniTune is able to achieve a median 94% of the available performance, providing a 1.33× speedup over the values selected by human experts, without requiring any user intervention. This adaptive tuning provides a median speedup of 3.79× (max 74.0×) over the best possible performance which can be achieved without autotuning.

Acknowledgements

I would like to sincerely thank supervisors Hugh Leather and Pavlos Petoumenos for their excellent guidance, for suffering through early revisions of this work, and for making the beginning of my metamorphosis from fledgling undergraduate to cynical postgraduate an utter joy. My thanks to Michel Steuwer for his patient and enthusiastic expositions of the SkelCL internals, and to Murray Cole and the CDT staff for creating this great environment for learning. Finally, thank you to my fellow PPar colleagues who share my love of strong coffee and weak humour.

Contents

List of Figures

List of Tables

Chapter 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, leading to a growing disparity between the levels of available performance and the ability for application developers to exploit it.

The multicore processors of modern devices offer many opportunities for parallelism, but fully harnessing this processing power requires an intimate knowledge of both the parallel programming semantics of the language and performance characteristics of the underlying hardware. In recent years, general purpose programming with GPUs promises even greater data parallel throughput, but is a significantly greater challenge to tame, forcing developers to master an unfamiliar programming model (such as provided by CUDA or OpenCL) and architecture (SIMD with a multi-level memory hierarchy). As such, GPGPU programming is often considered beyond the realm of all but the most expert of programmers. If steps are not taken to increase the accessibility of such parallelism, this will only serve to widen the gap between available and utilised performance as the core counts of hardware continue to increase.

One possible solution for this *programmability challenge* comes in the form of algorithmic skeletons, which offer to simplify parallel programming by raising the level of abstraction so that developers can focus on solving problems, rather than coordinating parallel resources. They achieve this by providing robust parallel implementations of common patterns of computation which developers parameterise with their application-specific code. This greatly reduces the challenge of parallel programming, allowing users to structure their problem solving logic

sequentially, while offloading the cognitive cost of parallel coordination to the skeleton author.

1.1 Sacrificing Performance for Ease of Use

Unfortunately, the performance of parallel programs is often sensitive to low level parameter values, and when tuning these values, one size *cannot* fit all. The performance of parallel program parameters are sensitive to the underlying hardware, to the program being executed, and even to the *dataset* that is operated upon. This is especially problematic for algorithmic skeletons, as skeleton authors cannot tune the performance of an implementation across the breadth of these three dimensions, and this results in programs which forgo the performance advantages that can be achieved with the low level tuning of hand written parallel code.

If the performance of algorithmic skeletons is to be competitive with that of hand crafted parallel programs, then these skeletons must be capable of adapting to their environments. The development of such *autotuning* software is an entire research field itself — and understandably so: there is an irresistible appeal to the idea of software which is capable of improving its own efficiency without the need for human intervention. The unfortunate reality is that while these autotuning systems share the unified goal of improving the performance of their respective optimisation targets, the range of competing approaches and implementations has resulted in a fragmented state in which no one system has been able to gain the critical mass to achieve mainstream traction.

This is the first aim of this thesis: to tackle the issue of providing a unified interface for autotuning which reduces the amount of redundant and overlapping work needed to implement autotuning for different optimisation targets.

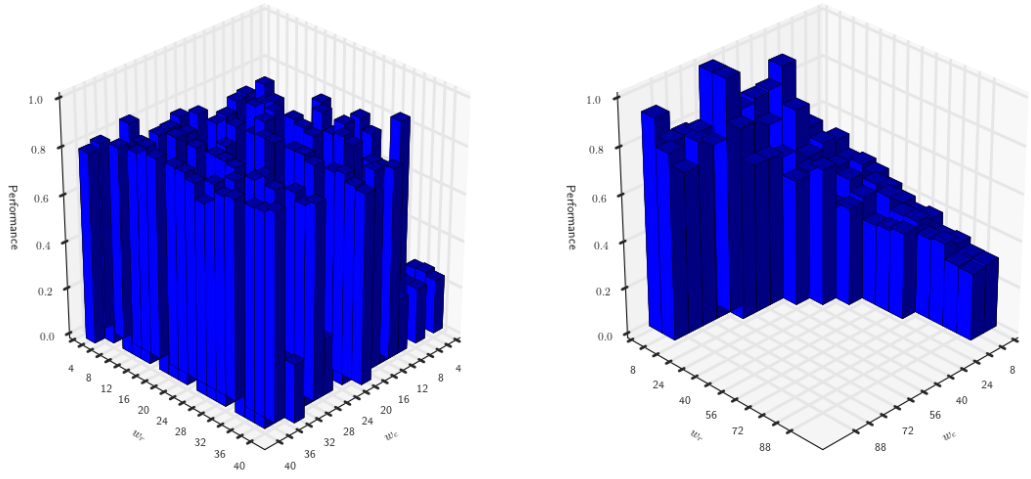
The second aim of this thesis is to explore techniques for predictive, machine learning-enabled autotuning for use in algorithmic skeletons. Typically, autotuning using iterative compilation requires enumerating some portion of the optimisation space for each program being tuned; however, the cost of such an exploration is prohibitively expensive when summed across the broad range of use cases targeted by algorithmic skeletons. As such, successfully autotuning algorithmic skeletons will require a method for *predicting* the values of parameters which will maximise performance, without the need for trial and error. To succeed, such a method of tuning does not need to provide perfectly accurate pre-

dictions, but simply sufficiently performant results so that, when combined with the vast accessibility improvements offered by algorithmic skeletons, it provides a convincing argument for algorithmic skeletons as the solution to the parallel programmability crisis.

1.2 Contributions

The key contributions of this thesis are:

- The development of *OmniTune* — a novel and extensible framework for the collaborative autotuning of optimisation parameters across the life cycle of programs.
- The application of OmniTune for tuning of the SkelCL algorithmic skeleton library. When tasked with predicting the workgroup sizes of stencil skeletons on both GPUs and CPUs, OmniTune achieves 94% of the available performance, providing a median speedup of 1.33× over values predicted by human experts, or 3.79× over the best possible statically chosen parameter values.
- The novel application of procedurally generated benchmark programs for training machine learning-enabled autotuners. This reduces the cost of training while increasing the size of the space which can be explored. The effectiveness of this approach is demonstrated by testing the OmniTune SkelCL autotuner trained using synthetic benchmarks against 68 configurations of real world stencil kernels, achieving comparable autotuning performance with that of cross-validation.
- An empirical evaluation of the performance of workgroup size to parameterise high-level parallel patterns. I enumerate the optimisation space of workgroup sizes for SkelCL stencil kernels across 269813 test cases, demonstrating an average performance loss of up to 15.14× if workgroup size is not correctly tuned.
- A comparison of multiple approaches for runtime autotuning: using classifiers to predict optimal parameter values, using regressors to predict the absolute runtime of programs, and using regressors to predict the relative performance of different parameter values.



(a) (b)

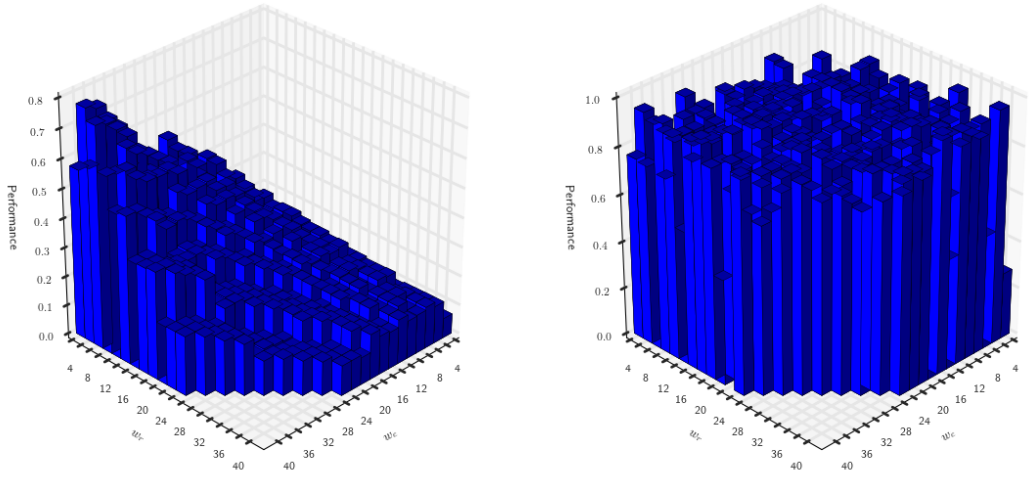
Figure 1.1: Workgroup size optimisation space of a stencil benchmark across devices: (a) Intel CPU, (b) NVIDIA GPU. This shows that stencil performance is dependent on properties of the underlying architecture, with different optimal workgroup sizes (56×20 vs. 64×4) for the two devices shown.

1.3 Motivation

In this section I present the case for autotuning the workgroup size of SkelCL stencil skeletons. Stencil workgroup sizes presents a two dimensional parameter space, consisting of a number of rows and columns. It is constrained by properties of both the stencil code and underlying architecture. For a detailed discussion of the parameter space and experimental methodology, see Chapters 2 and 6.

By comparing the mean runtime of a stencil program using different workgroup sizes while keeping all other conditions constant, we can assess the relative performance of different points in the optimisation space. Plotting this two dimensional optimisation space using a three dimensional bar chart provides a quick visual overview of the optimisation space. The two horizontal axes are used to represent the number of rows and columns in a workgroup, while the height of each bar shows the performance of a program at that point in the space (higher is better).

If the performance of workgroup sizes were not dependent on the execution device, we would expect the relative performance of points in the optimisation space to be consistent across devices. As shown in Figure 1.1, this is not the case,



(a) (b)

Figure 1.2: Workgroup size optimisation space of two stencils on the same device. Despite the underlying hardware being the same, the relative performance of workgroup sizes varies greatly between the two programs. The optimal workgroup sizes are 128×2 and 256×4 respectively.

with the optimisation space of the same benchmark on different devices being radically different. Not only does the optimal workgroup size change between devices, but the performance of suboptimal workgroup sizes is equally dissimilar.

The optimisation space of 1.1a has a grid-like structure, with clear performance advantages of workgroup sizes at multiples of 8 columns. A developer specifically targeting this device would learn to select workgroup sizes which follow this pattern. This domain specific knowledge does not transfer to the device shown in 1.1b, where the relatively simple optimisation space is more amenable to a stochastic hill climbing search.

Similarly, the optimisation space of two different stencils on the same device is shown in 1.2, demonstrating that the optimisation space is dependent on the program being executed.

The optimal workgroup size is different for each of the four examples, and the difference between the maximum and minimum performance workgroup sizes provides an average 37.0× speedup. The existing SkelCL stencil implementation uses a statically chosen workgroup size of 32×4, and this provides an average of only 63% of the available performance when compared to the best workgroup

size for these four examples. Even for this small set of examples, static values and simple heuristics cannot provide portable performance. The workgroup size parameter is sensitive to factors outside the influence of the developers control, such as the type of program, the data being operated on, and the execution device. This makes portable performance tuning a difficult task, and it has traditionally been the responsibility to domain specialists to laboriously hand tune individual programs to match the target problem and underlying hardware.

Given the important role that stencil codes play in many fields of computer science and simulation, and the difficulties in selecting workgroup sizes for portable performance, I believe that there is a compelling case for the development of an autotuner which can accommodate for these differences of workgroup size performance between devices and programs. It is my hypothesis that the performance of algorithmic skeletons will be improved by developing an autotuner which considers dynamic features which cannot be determined at compile time. The premise is that the optimisation spaces of algorithmic skeletons such as stencils are shaped by features which can only be determined at runtime. Effective searching of these spaces can only be performed by collecting empirical data rather than building predictive models.

The ambition of this thesis is to demonstrate that, using machine learning, we can develop predictive tuning systems which closely approach — and in some cases, outperform — the kinds of ad-hoc hand tuning which traditionally came at the cost of many man hours of work from expert programmers to develop.

1.4 Structure

The remainder of the document is structured as follows:

- Chapter 2 contains necessary background material, an introduction to the SkelCL framework, and a description of the techniques used throughout the thesis;
- Chapter 3 contains an exposition of relevant literature in the field of autotuning and heterogeneous parallelism, contrasting the related research with my own;
- Chapter 4 presents OmniTune, an extensible and distributed autotuner capable of predicting optimisation parameter values for unseen programs and

devices at runtime;

- Chapter 5 describes the application of OmniTune for selecting the workgroup size of SkelCL stencils;
- Chapter 6 describes a comprehensive exploration of the workgroup size optimisation space for stencil skeletons, including the methodology for obtaining performance data and experimental setup;
- Chapter 7 evaluates the effectiveness of OmniTune with respect to its accuracy, performance compared to human experts, and a cost benefit analysis of autotuning;
- Chapter 8 contains concluding remarks, a critical evaluation of the presented work, and plans for future research.

1.5 Summary

This introductory chapter has outlined the need for higher levels of abstraction for parallel programming and the difficulty that this provides for performance tuning. It advocates the use of adaptive tuning for algorithmic skeletons, and describes the contributions of this thesis towards this goal. In the next chapter, I provide an overview of the techniques and methodology used in this thesis.

Chapter 2

Background

2.1 Introduction

This chapter provides a detailed though non-exhaustive description of the theory and techniques used in this thesis. First is an overview of algorithmic skeletons, GPGPU programming, and the combination of the two in SkelCL. Second is an overview of the machine learning techniques used in this thesis, followed by a description of the Statistical tools tools and methodologies used in the evaluation.

2.2 Algorithmic Skeletons

Introduced by **Cole1989** in **Cole1989** algorithmic skeletons simplify the task of parallel programming by abstracting common patterns of communication, providing parallel implementations of higher order functions [**Cole1989**]. The interfaces to generic parallel algorithms exposed by algorithmic skeletons are parameterised by the user with *muscle functions* that implement problem specific logic. The idea is that this allows the user to focus on solving the problem at hand, affording greater ease of use by automating the coordination of parallel resources.

2.2.1 Abstracting Task and Data Parallelism

Algorithmic skeletons are categorised as either *data* parallel or *task* parallel. In data parallel skeletons, data are distributed across nodes for parallel processing, where each parallel node executes the same code on a unique subset of the data. Examples of data parallel algorithmic skeletons include *map*, *zip*, and *reduce*. The

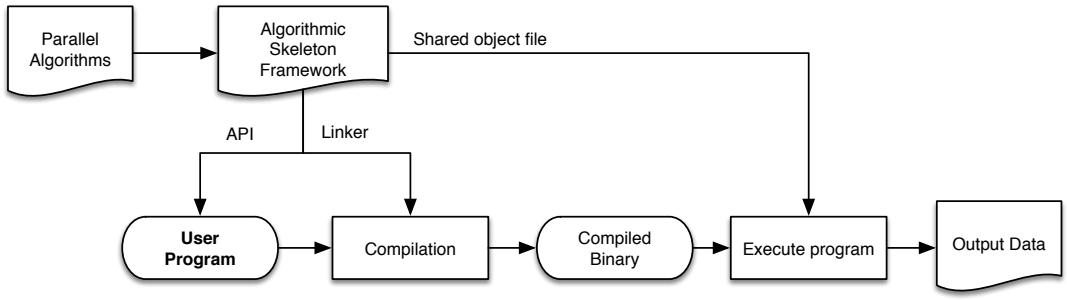


Figure 2.1: Typical usage of a library based Algorithmic Skeleton Framework. Other approaches to algorithmic skeletons involve embedding the API into the core language itself, or using template and macro substitution to remove the need for linking with a library.

data parallel operations provided by SkelCL are described in detail in Section 2.4.

Task parallel skeletons treat the data as a singular object and instead parallelise the execution of multiple tasks. Tasks are assigned to threads, which can communicate with each other by passing data between threads. Examples of task parallel algorithmic skeletons include *pipe*, *task farm*, and *for loops*.

2.2.2 Algorithmic Skeleton Frameworks

Algorithmic Skeleton Frameworks (ASkFs) provide concrete parallel implementations of parallel patterns, which are parameterised by the user to generate specific problem solving programs. The interfaces exposed by frameworks must be sufficiently generic to allow users to express a range of problems.

Implementations of algorithmic skeletons abound, targeting a range of different use cases and host languages. Notable examples include: eSkel [**Benoit2005a**], Skandium [**Leyton2010**], and FastFlow [**Aldinucci2011**]. The most prevalent form of ASkF is that of the standalone library which exposes a set of public APIs, shown in Figure 2.1. See [**Gonzalez2010**] for a more exhaustive review of ASkFs in the research literature.

In industry, Google's MapReduce [**Dean2008**] and Intel's Thread Building Blocks [**IntelTBB**] have utilised a similar approach to abstracting the coordination of parallel resources as in algorithmic skeletons, to great commercial success, although they do not advertise themselves as such.

2.3 GPGPU Programming

General purpose programming with graphics hardware is a nascent field, but has shown to enable massive data parallel throughput by re-purposing the hardware traditionally dedicated to the rendering of 3D graphics for generic computation. This was enabled by hardware support for programmable shaders replacing the fixed function graphics pipeline, and support for floating point operations in 2001. **Owens2006** provide a review of the first five years of general purpose computation on graphics hardware in [**Owens2006**].

In the ensuing progress towards increasingly programmable graphics hardware, two dominant programming models have emerged: CUDA and OpenCL, which both abstract the graphics primitives of GPU hardware and provide a platform for GPGPU programming. CUDA is a language developed by NVIDIA for programming their GPUs using a proprietary SDK and API [**Nvidia2007**], while OpenCL is a vendor-independent open standard based on a subset of the ISO C99 programming language, with implementations for devices from most major GPU manufactures [**Stone2010**]. Quantitative evaluations of the performance of CUDA and OpenCL programs suggest that performance is comparable between the two systems, although the wider range of target architectures for OpenCL means that appropriate optimisations must be made by hand or by the compiler [**Komatsu2010**; **Karimi2010**].

2.3.1 The OpenCL Programming Model

OpenCL is a parallel programming framework which targets CPUs, GPUs, and other parallel processors such as Field-Programmable Gate Arrays. It provides a set of APIs and a language (based on an extended subset of C) for controlling heterogeneous *compute devices* from a central host. Programs written for these devices are called *kernels*, and are compiled by platform-specific tool chains. At runtime, an OpenCL *platform* is selected and a *context* object is created which exposes access to each supported compute device through *command queues*. When a kernel is executed, each unit of computation is referred to as a *workitem*, and these workitems are grouped into *workgroups*. The sum of all workgroup dimensions defines the *global size*. For GPUs, workgroups execute on the Single Instruction Multiple Data (SIMD) processing units in lockstep. This is very different from the behaviour of traditional CPUs, and can cause severe performance penalties

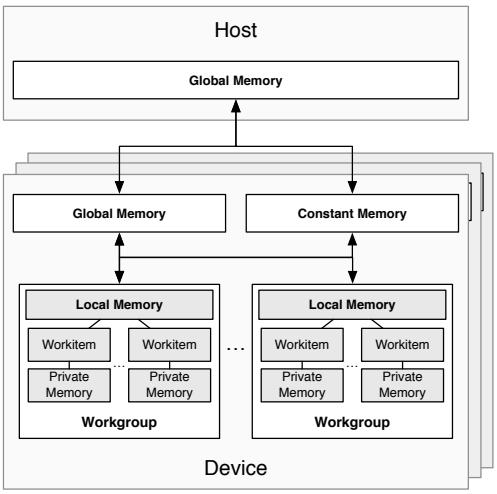


Figure 2.2: The OpenCL memory model. The host communicates with each device through transfers between global memory spaces. The capacity of each type of memory is dependent on the device hardware. In general, private memory is the fastest and smallest, and global memory is the largest and slowest.

in the presence of flow control, as workitems must be stalled across divering flow paths.

Memory Model

Unlike the flat model of CPUs, OpenCL uses a hierarchical memory model. The host and each OpenCL device has a single global memory address space. Each workgroup has a local memory space, and each workitem has a region of private memory.

Workgroups cannot access the memory of neighbouring workgroups, nor can workitems access the private memory of other workitems. OpenCL provides synchronisation barriers to allow for communication between workitems within a single workgroup via the local memory, but not global barriers. Memory transfers between the host and devices occurs between global memory regions. In the case of programming heterogeneous devices, these transfers must occur over the connection bus between the CPU and device (e.g. PCIe for discrete GPUs), which typically creates a performance bottleneck by introducing a performance overhead to transfer data to the device for processing, then back to the device afterwards. Direct transfers of data between devices is not supported, requiring an intermediate transfer to the host memory.

Performance Optimisations

The wide range of supported execution devices and differing standards-compliant implementations makes portable performance tuning of OpenCL programs a difficult task [**Rul2010**], and the interactions between optimisations and the hardware are complex and sometimes counter-intuitive [**Ryoo2008**].

The overhead introduced by memory transfers between host and compute devices further complicates comparisons of OpenCL performance on different devices. The conclusion of [**Gregg2011**] is that this overhead can account for a 2× to 50× difference of GPU program runtime. In [**Lee2010**], **Lee2010** present a performance analysis of optimised throughput computing applications for GPUs and CPUs. Of the 14 applications tested, they found GPU performance to be 0.7× to 14.9× that of multi-threaded CPU code, with an average of only 2.5×. This is much lower than the 100× to 1000× values reported by other studies, a fact that they attribute to uneven comparison of optimised GPU code to unoptimised CPU code, or vice versa. **Lee2010** found that multithreading, cache blocking, reordering of memory accesses and use of SIMD instructions to contribute most to CPU performance. For GPUs, the most effective optimisations are reducing synchronization costs, and exploiting local shared memory. In all cases, the programs were optimised and hand-tuned by programmers with expert knowledge of the target architectures. It is unclear whether their performance results still hold for subsequent generations of devices.

Despite the concerns of over-represented speedups, the potential for high performance coupled with the complexity and low levels of abstraction provided by OpenCL make it an ideal target for skeletal abstractions. SkelCL and SkePU are two such examples which add a layer of abstraction above OpenCL and CUDA respectively in order to simplify GPGPU programming [**Enmyren2010**].

2.4 High-Level GPU Programming with SkelCL

Introduced in [**Steuwer2011**], SkelCL is an object oriented C++ library that provides OpenCL implementations of data parallel algorithmic skeletons for heterogeneous parallelism using CPUs or multi-GPUs. SkelCL addresses the parallel programmability challenge by allowing users to easily harness the power of GPUs and CPUs for data parallel computing.

The goal of SkelCL is to enable the transition towards higher-level programming of GPUS, without requiring users to be intimately knowledgeable of the concepts unique to OpenCL programming, such as the memory or execution model [**Steuwer2012**]. SkelCL has been shown to reduce programmer effort for developing real applications through the use of robust pattern implementations and automated memory management, maintaining performance within 5% of that of equivalent hand-written implementations in OpenCL [**Steuwer2013**].

SkelCL skeletons are parameterised with muscle functions by the user, which are compiled into OpenCL kernels for execution on device hardware. SkelCL supports operations on one or two dimensional arrays of data, with the Vector and Matrix container types transparently handling lazy transfers between host and device memory, and supporting partitioning for multi-GPU execution [**Steuwer2013a**]. SkelCL is freely available and distributed under dual GPL and academic licenses[1].

2.4.1 Pattern definitions

SkelCL provides six skeletons for data parallel operations: Map, Zip, Reduce, Scan, AllPairs, and Stencil. The focus of this thesis is on tuning the Stencil skeleton, but for the sake of completeness I provide here a brief overview of the behaviour of all six patterns.

Map

The Map operation is a basic building block of data parallel algorithms. Given a customising function f and a vector x of n elements, the Map operation applies the function f to each element $x_1, x_2, \ldots, x_n$, returning a vector of the same length:

$$\text{Map}\,(f, [x_1, x_2, \ldots, x_n]) \rightarrow [f(x_1), f(x_2), \ldots, f(x_n)] \tag{2.1}$$

The process is the same for an $n \times m$ matrix:

$$\text{Map}\left(f, \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{bmatrix}\right) \rightarrow \begin{bmatrix} f(x_{11}) & \cdots & f(x_{1m}) \\ \vdots & \ddots & \vdots \\ f(x_{n1}) & \cdots & f(x_{nm}) \end{bmatrix} \tag{2.2}$$

Execution of the customising function can be parallelised as each element is processed independently. There are no guarantees on execution ordering.

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

Zip

The Zip operation combines elements of containers pairwise. Given a binary operator $\oplus$ and two vectors x and y of n elements each:

$$\text{Zip}\,(\oplus, [x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_n]) \rightarrow [x_1 \oplus y_1, x_2 \oplus y_2, \ldots, x_n \oplus y_n] \tag{2.3}$$

For two matrices of $n \times m$ elements each:

$$\begin{aligned} \text{Zip}\left(\oplus, \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{bmatrix}, \begin{bmatrix} y_{11} & \cdots & y_{1m} \\ \vdots & \ddots & \vdots \\ y_{n1} & \cdots & y_{nm} \end{bmatrix}\right) \\ \rightarrow \begin{bmatrix} x_{11} \oplus y_{11} & \cdots & x_{1m} \oplus y_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} \oplus y_{n1} & \cdots & x_{nm} \oplus y_{nm} \end{bmatrix} \end{aligned} \tag{2.4}$$

The Zip operation is parallelised in the same manner as Map.

Reduce

The Reduce operator combines all elements of an input vector and returns a scalar. Given a binary operator $\oplus$, its identity $\boldsymbol{i}$ and a vector x of n elements:

$$\text{Reduce}\,(\oplus, \boldsymbol{i}, [x_1, x_2, \ldots, x_n]) \rightarrow x_1 \oplus x_2 \oplus \ldots \oplus x_n \tag{2.5}$$

For a $n \times m$ matrix:

$$\text{Reduce}\left(\oplus, i, \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{bmatrix}\right) \rightarrow x_{11} \oplus x_{12} \oplus \ldots \oplus x_{nm} \tag{2.6}$$

Reduction operates by recursively combining elements, storing intermediate results in fast local memory. No guarantee is made on execution ordering, so the binary operator must be associative.

Scan

The Scan operator applies a binary function $\oplus$ to each member of an input array x of n elements, such that element x_i at the ith position is calculated by applying the binary function to all elements $x_1, x_2, \ldots x_{i-1}$. For the first element x_i, the value is the identify $\boldsymbol{i}$ of the $\oplus$ operator:

$$\text{Scan}\,(\oplus, \boldsymbol{i}, [x_1, x_2, \ldots, x_n]) \rightarrow [\boldsymbol{i}, x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \ldots \oplus x_n] \tag{2.7}$$

The SkelCL scan implementation is heavily optimised to make use of local memory, based on the design from [**Harris2007a**].

AllPairs

The AllPairs skeleton is a specialised pattern for Matrix Matrix operations, introduced in [**Steuwer2014**]. Given two matrices of size $n \times d$ and $n \times m$, a binary operator $\oplus$:

$$\text{All Pairs}\left(\oplus, \begin{bmatrix} x_{11} & \cdots & x_{1d} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nd} \end{bmatrix}, \begin{bmatrix} y_{11} & \cdots & y_{1m} \\ \vdots & \ddots & \vdots \\ y_{n1} & \cdots & y_{nm} \end{bmatrix}\right) \rightarrow \begin{bmatrix} z_{11} & \cdots & z_{1m} \\ \vdots & \ddots & \vdots \\ z_{n1} & \cdots & z_{nm} \end{bmatrix} \tag{2.8}$$

where:

$$z_{ij} = [x_{i1}, x_{i2}, \ldots, x_{id}] \oplus \left[y_{j1}, y_{j2}, \ldots, y_{jd}\right] \tag{2.9}$$

An additional implementation is provided for when the $\oplus$ operator is known to match that of a zip pattern:

$$z_{ij} = \left[x_{i1}, \oplus y_{j1}, x_{i2} \oplus y_{j2}, \ldots, x_{id} \oplus y_{jd}\right] \tag{2.10}$$

This pattern has applications from bioinformatics to matrix multiplication, and uses fast local memory to reduce data access times for repeated reads.

Stencil

Stencils are patterns of computation which operate on uniform grids of data, where the value of each cell is updated based on its current value and the value of one or more neighbouring elements, called the *border region*. Introduced in [**Breuer2014**], SkelCL provides a 2D stencil skeleton which allows users to provide a function which updates a cell's value, while SkelCL orchestrates the parallel execution of this function across all cells.

The border region is described by a *stencil shape*, which defines an $i \times j$ rectangular region about each cell which is used to update the cell value. Stencil shapes may be asymmetrical, and are defined in terms of the number of cells in the border region to the north, east, south, and west of each cell, shown in Figure 2.3. Given a customising function f, a stencil shape S, and an $n \times m$ matrix:

$$\text{Stencil}\left(f, S, \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{bmatrix}\right) \rightarrow \begin{bmatrix} z_{11} & \cdots & z_{1m} \\ \vdots & \ddots & \vdots \\ z_{n1} & \cdots & z_{nm} \end{bmatrix} \tag{2.11}$$

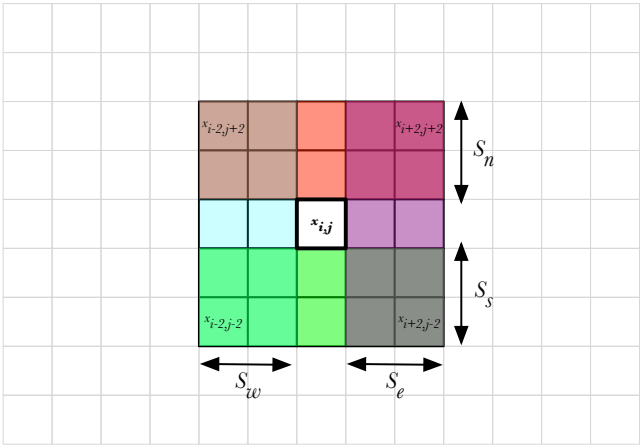


Figure 2.3: The border region of a stencil is defined using a stencil shape S, consisting of the four independent components describing the number of cells north S_n, east S_e, west S_w, and south S_s.

where:

$$z_{ij} = f\left(\begin{bmatrix} z_{i-S_n,j-S_w} & \cdots & z_{i-S_n,j+S_e} \\ \vdots & \ddots & \vdots \\ z_{i+S_s,j-S_w} & \cdots & z_{i+S_s,j+S_e} \end{bmatrix}\right) \tag{2.12}$$

Note that a stencil operation in which the size of the stencil shape S is zero in every direction is functionally equivalent to a Map operation. Where the border region includes elements outside of the matrix, values are substituted from either a predefined padding value, or the value of the nearest cell within the matrix, depending on user preference.

A popular usage of Stencil codes is for solving problems iteratively, whereby a stencil operation is repeated over a range of discrete time steps $0 \leq t \leq t_{max}$, and $t \in \mathbb{Z}$. An iterative stencil operation g accepts a customising function f, a Stencil shape S, and a matrix M with initial values M_{init}. The value of an iterative stencil can be defined recursively as:

$$g(f,S,M,t) = \begin{cases} \text{Stencil}\left(f,S,g(f,S,M,t-1)\right), & \text{if } t \geq 1 \\ M_{init}, & \text{otherwise} \end{cases} \tag{2.13}$$

Examples of iterative stencils are cellular automata. Another extension of the stencil operation accepts an ordered list of customising functions which are applied sequentially for each iteration. This has applications for multi-stage stencil operations such as Canny Edge Detection, in which four distinct stencil operations are performed as a sequence.

2.4.2 Implementation Details

Each skeleton is represented by a template class, declared in a header file detailing the public API, e.g. `SkelCL/Stencil.h`. A private header file contains the template class declaration, e.g. `SkelCL/detail/StencilDef.h`. Small OpenCL kernels are stored as inline strings inside the definition headers. Non-trivial OpenCL kernels are stored in separate source files, e.g. `SkelCL/detail/StencilKernel.cl`. Muscle functions are passed as OpenCL strings to the skeleton template classes, and the LLVM API is used to perform argument and function name substitution.

Stencil Implementation

For the stencil skeleton, each cell maps to a single work item; and this collection of work items is then divided into *workgroups* for execution on the target hardware. In a stencil code, each work-item reads the value of the corresponding grid elements, and the surrounding elements defined by the border region. Since the border regions of neighbouring elements overlap, the value of all elements within a workgroup are stored in a *tile*, which is a region of local memory. As local memory access times are much smaller than that of global memory, this greatly reduces the latency of the border region reads performed by each workitem. Changing the workgroup size thus affects the amount of local memory required for each workgroup, which in turn affects the number of workgroups which may be simultaneously active. So while the user defines the size, type, and border region of the of the grid being operated upon, it is the responsibility of the SkelCL stencil implementation to select an appropriate workgroup size to use.

2.5 Machine Learning

Multiple machine learning methods are used in this thesis to perform classification and regression tasks. Classification is the task of predicting the correct category — or class — for a new instance, based on "labelled" training data, i.e. instances whose categories are known. The properties describing instances are called *features*. The purpose of regression is to predict the relationship between a dependent variable and the value of one or more independent variables. Unlike with classification, the predicted dependent variable can be continuous, rather than categorical. This section briefly describes the properties of classifiers and

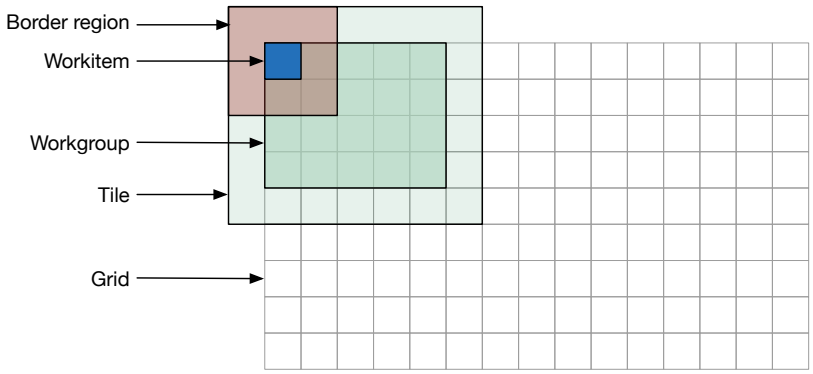


Figure 2.4: The components of a stencil: a grid of cells is decomposed into workgroups, consisting of multiple workitems. Each work item operates on a border region, and the size of the workgroup and outer border region defines a tile, which is a region of local memory allocated to each workgroup.

regressors used in this thesis.

ZeroR

A "ZeroR" classifier represents the simplest approach to classification, in that it ignores all features, and simply predicts the mode of the training data labels. This is useful for providing a baseline to evaluate the performance of more complex classifiers against, since a ZeroR classifier has no power of prediction.

Naive Bayes

Naive Bayes classifiers are probabilistic classifiers which assume strong independence between features. That is, the value of one feature is considered independently of another, and is what lends the *Naive* portion of the name. The goal of Naive Bayes is to predict the probability of a class y, given a set of d independent variables $x = x_1, x_2, \ldots x_d$. Naive Bayes applies Bayes theorem, which states that given a *prior* (i.e. unconditional) probability for each class $P(y)$, a class-*conditional* model $P(x|y)$, and a normaliser across all classes $P(x) = \sum_{y'} P(x|y')P(y')$: the probability of a class y given dependent variables x is equal to the probability of x given y, multiplied by the probability of y:

$$P(y|x) = \frac{P(x|y)P(y)}{\sum_{y'} P(x|y')P(y')} \tag{2.14}$$

The class which has the highest probability from the set of possible classes Y is the prediction. Note that the normaliser $P(x)$ does not affect the class which is

most likely. The class conditional model uses "counts" for each observation based on training data:

$$P(x|y) = \prod_{i=1}^{d} P(x_i|y) \tag{2.15}$$

The simplicity of Naive Bayes makes it attractive for various purposes such as text classification (using word frequency as features), but the assumption of independence makes it unsuitable for certain use cases.

Decision Trees

Decision trees are an intuitive form of classification in which a tree structure of decision nodes are used to predict the class for a given set of features. Decision trees are built using binary recursive partitioning: by creating a decision node for the feature which provides the highest gain, creating new child nodes for each possible outcome, splitting dividing the data amongst these child nodes, and recursing. To find the gain of an feature, we first compute the entropy of the data set. Given a set of data points D, and $p_{(+)}$ and $p_{(-)}$ are the number of positive and negative examples in D:

$$H(D) = -p_{(+)} \log_2 p_{(+)} - p_{(-)} \log_2 p_{(-)} \tag{2.16}$$

The gain of an feature x is then found using:

$$\text{Gain}(D, x) = H(D) - \sum_{V \in \text{Values}(x)} \frac{|D_V|}{|D|} H(D_V) \tag{2.17}$$

Decision trees are very popular and low overhead form of classification, as they can be implemented using a nested set of `if/else` statements. However, they can often overfit to the training data.

Random Forest

Random Forests are an ensemble technique which attempt to overcome the tendency of decision trees to overfit to training data by training multiple decision trees and using the mode of each tree to select predictions [**Breiman1999**]. Alternatively they can be trained using regression trees, in which case the predicted value is the mean output of all trees.

2.6 Statistics

This section describes a number of the statistical tools used throughout this thesis.

Interquartile Range

For a given set of ordered data values, the quartiles are the three points that divide the data set into four groups of equal size. The first quartile is the point which Q_1 divides the data equally between the lowest value and the median, Q_2. The third quartile Q_3 is the point which divides the data between the median and the highest value. The interquartile range (IQR) measures the dispersion of a data set, and is the difference between the third and first quartiles, $Q_3 - Q_1$.

Confidence Intervals

Confidence intervals estimate the interval for the true range of a population parameter. Given a set of samples of a population with estimates of a parameter value for each, the confidence interval (c_1, c_2) estimates the frequency that the true parameter value will fall between the per-sample confidence interval. Given a sample x with sample size n, the confidence interval for a confidence α is found using:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.18}$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}} \tag{2.19}$$

$$c_1 = \bar{x} - z_{1-\alpha/2}\frac{\sigma}{\sqrt{n}} \tag{2.20}$$

$$c_2 = \bar{x} + z_{1-\alpha/2}\frac{\sigma}{\sqrt{n}} \tag{2.21}$$

Where the value $z_{1-\alpha/2}$ assumes a Gaussian distribution of the underlying data, and the values for popular α values are typically found using pre-computed "Z-tables". To calculate confidence intervals for the ratio of two means, $\bar{x}_1$ and $\bar{x}_2$ with sample sizes n_1 and n_2, and respective standard deviations σ_1 and σ_2:

$$\sigma_x = \sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}} \tag{2.22}$$

$$c_1 = \bar{x}_1 - \bar{x}_2 - z_{1-\alpha/2}\sigma_x \tag{2.23}$$

$$c_2 = \bar{x}_1 - \bar{x}_2 + z_{1-\alpha/2}\sigma_x \tag{2.24}$$

The above calculations assumes that the sample variance (σ^2) is an accurate estimation of the true population variance. This relies on the assumption of an underlying Gaussian distribution, which, according to the central limit theorem, is true of large sample sizes (typically $n \geq 30$). In cases of smaller sample sizes, the sample and population variances can differ significantly, so one must instead use a Student's t-distribution, $t_{1-\alpha/2}$, in place of $z_{1-\alpha/2}$.

Histogram

Histograms are a widely used statistical visualisation which shows the distribution of numerical data. The data is first divided into a set of discrete, equally sized sub-intervals, or *bins*. The number of data points in each bin is used to show visualise the density distribution. The shortcoming of histograms is that their appearance is heavily influenced by three user-selected parameters: the number of bins, the width of bins (binwidth), and the endpoints chosen. As such, they may provide a misleading representation of the data if inappropriate values for any of these parameters are chosen. Kernel Density estimation is a technique for showing the distribution of data which circumvents some of these issues.

Kernel Density Estimation

A Kernel Density Estimate (KDE) is an approximation of the probability density function of a random variable. Given a random variable x, and bandwidth h and a kernel K, the Kernel Density Estimate $\hat{f}_h(x)$ can be found using:

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right) \tag{2.25}$$

Using a smooth kernel such as a Gaussian distribution for the kernel produces a smooth density estimated, unlike histograms. However, like histograms, the appearance of a Kernel Density Estimate plot is dependent on the value of the bandwidth parameter h (equivalent to binwidth in histograms), so care must be taken to select a value to minimise over or under smoothing. Grouped data can be shown by plotting multiple KDEs on the same axes, although if the number of groups is large, a box plot or violin plot may be more appropriate.

Box plot

Box plots are used to show the distribution of quartile ranges for grouped data. The contain the following features:

- Horizontal lines at the lower quartile, median and upper quartile.
- Vertical lines above and below the upper and lower quartiles to the most extreme data point within 1.5 IQR of the upper/lower quartile, with horizontal whiskers at the end of the vertical lines.
- Dots beyond the ends of the vertical lines to show outliers.

A variation of box plots used in this thesis is the violin plot, which extends the box plot with a fitted Kernel Density Estimate plot to show the probability *density* of data at different values. To construct a violin plot, KDEs for each group are rotated and mirrored to generate a smoothed visualisation of the distribution.

2.7 Summary

This chapter gave a brief introduction to general-purpose computing using graphics hardware, the OpenCL framework, and SkelCL. It was followed by a description of the machine learning techniques used throughout this thesis, and the statistical tools used for the evaluation. In the next chapter, I review the state of the art in the field of autotuning, and provide context for the contributions of this thesis.

Chapter 3

Related Work

This chapter begins with a brief survey of the broad field of literature that is relevant to algorithmic skeletons. This is followed by a review of the current state of the art in autotuning research, focusing on heterogeneous parallelism, algorithmic skeletons, and stencil codes. It presents the context and rationale for the research undertaken for this thesis.

3.1 Automating Parallelism

It is widely accepted that parallel programming is difficult, and the continued repetition of this claim has become something of a trite mantra for the parallelism research community. An interesting digression is to discuss some of the ways in which researchers have attempted to tackle this difficult problem, and why, despite years of research, it remains an ongoing challenge.

The most ambitious and perhaps daring field of parallelism research is that of automatic parallelisation, where the goal is to develop methods and systems to transform arbitrary sequential code into parallelised code. This is a well studied subject, with the typical approach being to perform these code transformations at the compilation stage. In **Banerjee1993**'s thorough review [**Banerjee1993**] on the subject, they outline the key challenges of automatic parallelisation:

- determining whether sequential code can be legally transformed for parallel execution; and
- identifying the transformation which will provide the highest performance improvement for a given piece of code.

Both of these challenges are extremely hard to tackle. For the former, the difficulties lie in performing accurate analysis of code behaviour. Obtaining accurate dynamic dependency analysis at compile time is an unsolved problem, as is resolving pointers and points-to analysis [**Atkin-granville2013**; **Hind2001**; **Ghiya2001**].

The result of these challenges is that reliably performant, automatic parallelisation of arbitrary programs remains a far from reached goal; however, there are many note worthy variations on the theme which have been able to achieve some measure of success.

One such example is speculative parallelism, which circumvents the issue of having incomplete dependency information by speculatively executing code regions in parallel while performing dependency tests at runtime, with the possibility to fall back to "safe" sequential execution if correctness guarantees are not met [**Prabhu2010**; **Trachsel2010**]. In [**Jimborean2014**], **Jimborean2014** present a system which combines polyhedral transformations of user code with binary algorithmic skeleton implementations for speculative parallelisation, reporting speedups over sequential code of up to 15.62× on a 24 core processor.

Another example is PetaBricks, which is a language and compiler enabling parallelism through "algorithmic choice" [**Ansel2009**; **Ansel2010**]. With PetaBricks, users provide multiple implementations of algorithms, optimised for different parameters or use cases. This creates a search space of possible execution paths for a given program. This has been combined with autotuning techniques for enabling optimised multigrid programs [**Chan2009**], with the wider ambition that these autotuning techniques may be applied to all algorithmic choice programs [**Ansel2014**]. While this helps produce efficient parallel programs, it places a great burden on the developer, requiring them to provide enough contrasting implementations to make a search of the optimisation space fruitful.

Annotation-driven parallelism takes a similar approach. The user annotates their code to provide "hints" to the compiler, which can then perform parallelising transformations. A popular example of this is OpenMP, which uses compiler pragmas to mark code sections for parallel or vectorised execution [**Dagum1998**]. Previous work has demonstrated code generators for translating OpenMP to OpenCL [**Grewe2013**] and CUDA [**Lee2009**]. Again, annotation-driven parallelism suffers from placing a burden on the developer to identify the potential areas for parallelism, and lacks the structure that algorithmic skeletons provide.

Algorithmic skeletons contrast the goals of automatic parallelisation by removing the challenge of identifying potential parallelism from compilers or users, instead allowing users to frame their problems in terms of well defined patterns of computation. This places the responsibility of providing performant, well tuned implementations for these patterns on the skeleton author.

3.2 Iterative Compilation & Machine Learning

Iterative compilation is the method of performance tuning in which a program is compiled and profiled using multiple different configurations of optimisations in order to find the configuration which maximises performance. One of the the first formalised publications of the technique appeared in **Bodin1998** by **Bodin1998** [**Bodin1998**]. Iterative compilation has since been demonstrated to be a highly effective form of empirical performance tuning for selecting compiler optimisations.

Given the huge number of possible compiler optimisations (there are 207 flags and parameters to control optimisations in GCC v4.9), it is often unfeasible to perform an exhaustive search of the entire optimisation space, leading to the development of methods for reducing the cost of evaluating configurations. These methods reduce evaluation costs either by shrinking the dimensionality or size of the optimisation space, or by guiding a directed search to traverse a subset of the space.

Machine learning has been successful applied to this problem, in [**Stephenson2003**], using "meta optimisation" to tune compiler heuristics through an evolutionary algorithm to automate the search of the optimisation space. **Fursin2011** continued this with Milepost GCC, the first machine learning-enabled self-tuning compiler [**Fursin2011**]. A recent survey of the use of machine learning to improve heuristics quality by **Burke2013** concludes that the automatic *generation* of these self-tuning heuristics but is an ongoing research challenge that offers the greatest generalisation benefits [**Burke2013**].

In [**Saclay2010**; **Memon2013**; **Fursin2014**], **Fursin2014** advocate a collaborative and "big data" driven approach to autotuning, arguing that the challenges facing the widespread adoption of autotuning and machine learning methodologies can be attributed to: a lack of common, diverse benchmarks and datasets; a lack of common experimental methodology; problems with continuously chang-

ing hardware and software stacks; and the difficulty to validate techniques due to a lack of sharing in publications. They propose a system for addressing these concerns, the Collective Mind knowledge system, which, while in early stages of ongoing development, is intended to provide a modular infrastructure for sharing autotuning performance data and related artifacts across the internet. In addition to sharing performance data, the approach taken in this thesis emphasises the collective *exploitation* of such performance data, so that data gathered from one device may be used to inform the autotuning decisions of another. This requires each device to maintain local caches of shared data to remove the network overhead that would be present from querying a single centralised data store during execution of a hot path. The current implementation of Collective Mind uses a NoSQL JSON format for storing performance data. The relational schema used in this thesis offers greater scaling performance and lower storage overhead as the amount of performance data grows.

Whereas iterative compilation requires an expensive offline training phase to search an optimisation space, 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 may result in 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 [**Bala2000**]. 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.

An interesting related tangent to iterative compilation is the development of so-called "superoptimisers". In [**Massalin1987**], the smallest possible program which performs a specific function is found through a brute force enumeration of the entire instruction set. Starting with a program of a single instruction, the superoptimiser tests to see if any possible instruction passes a set of conformity tests. If not, the program length is increased by a single instruction and the

process repeats. The exponential growth in the size of the search space is far too expensive for all but the smallest of hot paths, typically less than 13 instructions. The optimiser is limited to register to register memory transfers, with no support for pointers, a limitation which is addressed in [**Joshi2002**]. Denali is a superoptimiser which uses constraint satisfaction and rewrite rules to generate programs which are *provably* optimal, instead of searching for the optimal configuration through empirical testing. Denali first translates a low level machine code into guarded multi-assignment form, then uses a matching algorithm to build a graph of all of a set of logical axioms which match parts of the graph, before using boolean satisfiability to disprove the conjecture that a program cannot be written in n instructions. If the conjecture cannot be disproved, the size of n is increased and the process repeats.

3.2.1 Training with Synthetic Benchmarks

The use of synthetic benchmarks for providing empirical performance evaluations dates back to as early as 1974 [**Curnow1976**]. The *automatic generation* of such synthetic benchmarks is a more recent innovation, serving the purpose initially of stress-testing increasingly complex software systems for behaviour validation and automatic bug detection [**Verplaetse2000**; **Godefroid2008**]. A range of techniques have been developed for these purposes, ranging from applying random mutations to a known dataset to generate test stimuli, to so-called "whitebox fuzz testing" which analyses program traces to explore the space of a program's control flow. Csmith is one such tool which generates randomised C source programs for the purpose of automatically detecting compiler bugs [**Yang2012**].

A method for the automatic generation of synthetic benchmarks for the purpose of *performance* tuning is presented in [**Chiu2015**]. **Chiu2015** use template substitution over a user-defined range of values to generate training programs with a statistically controlled range of features. A Perl preprocessor generates output source codes from an input description using a custom input language Genesis. Genesis is more flexible than the system presented in this thesis, supporting substitution of arbitrary sources. The authors describe an application of their tool for generating OpenCL stencil kernels, but do not report any performance results.

3.3 Performance Tuning for Heterogeneous Parallelism

As briefly discussed in Section 2.3, the complex interactions between optimisations and heterogeneous hardware makes performance tuning for heterogeneous parallelism a difficult task. Performant GPGPU programs require careful attention from the developer to properly manage data layout in DRAM, caching, diverging control flow, and thread communication. The performance of programs depends heavily on fully utilising zero-overhead thread scheduling, memory bandwidth, and thread grouping. **Ryoo2008a** illustrate the importance of these factors by demonstrating speedups of up to 432× for matrix multiplication in CUDA by appropriate use of tiling and loop unrolling [**Ryoo2008a**]. The importance of proper exploitation of local shared memory and synchronisation costs is explored in [**Lee2010**].

In [**Chen2014**], data locality optimisations are automated using a description of the hardware and a memory-placement-agnostic compiler. The authors demonstrate impressive speedups of up to 2.08×, although at the cost of requiring accurate memory hierarchy descriptor files for all targeted hardware. The descriptor files must be hand generated, requiring expert knowledge of the underlying hardware in order to properly exploit memory locality.

Data locality for nested parallel patterns is explored in [**Lee**]. The authors use an automatic mapping strategy for nested parallel skeletons on GPUs, which uses a custom intermediate representation and a CUDA code generator, achieving 1.24× speedup over hand optimised code on 7 of 8 Rodinia benchmarks.

Reduction of the GPGPU optimisation space is demonstrated in [**Ryoo2008**], using the common subset of optimal configurations across a set of training examples. This technique reduces the search space by 98%, although it does not guarantee that for a new program, the reduced search space will include the optimal configuration.

Magni2014 demonstrated that thread coarsening of OpenCL kernels can lead to speedups in program performance between 1.11× and 1.33× in [**Magni2014**]. The authors achieve this using a machine learning model to predict optimal thread coarsening factors based on the static features of kernels, and an LLVM function pass to perform the required code transformations.

A framework for the automatic generation of OpenCL kernels from high-level

programming concepts is described in [**Steuwer2015**]. A set of rewrite rules is used to transform high-level expressions to OpenCL code, creating a space of possible implementations. This approach is ideologically similar to that of PetaBricks, in that optimisations are made through algorithmic choice, although in this case the transformations are performed automatically at the compiler level. The authors report performance on a par with that of hand written OpenCL kernels.

3.4 Autotuning Algorithmic Skeletons

An enumeration of the optimisation space of Intel Thread Building Blocks in [**Contreras2008**] shows that runtime knowledge of the available parallel hardware can have a significant impact on program performance. **Collins2012** exploit this in [**Collins2012**], first using Principle Components Analysis to reduce the dimensionality of the space of possible optimisation parameters, followed by a search of parameter values to optimise program performance by a factor of 1.6× over values chosen by a human expert. In [**Collins2013**], they extend this using static feature extraction and nearest neighbour classification to further prune the search space, achieving an average 89% of the oracle performance after evaluating 45 parameters.

Dastgeer2011 developed a machine learning based autotuner for the SkePU skeleton library in [**Dastgeer2011**]. Training data is used to predict the optimal execution device (i.e. CPU, GPU) for a given program by predicting execution time and memory copy overhead based on problem size. The autotuner only supports vector operations, and there is limited cross-architecture evaluation. In [**Dastgeer2015a**], the authors extend SkePU to improve the data consistency and transfer overhead of container types, reporting up to a 33.4× speedup over the previous implementation.

3.5 Code Generation and Autotuning for Stencils

Stencil codes have a variety of computationally expensive uses from fluid dynamics to quantum mechanics. Efficient, tuned stencil kernels are highly sought after, with early work in **Bolz2003** by **Bolz2003** demonstrating the capability of GPUs for massively parallel stencil operations [**Bolz2003**]. In the resulting years, stencil codes have received much attention from the performance tuning

research community.

Ganapathi2009 demonstrated early attempts at autotuning multicore stencil codes in [**Ganapathi2009**], drawing upon the successes of statistical machine learning techniques in the compiler community, as discussed in Section 3.2. They present an autotuner which can achieve performance up to 18% better than that of a human expert. From a space of 10 million configurations, they evaluate the performance of a randomly selected 1500 combinations, using Kernel Canonical Correlation Analysis to build correlations between tunable parameter values and measured performance targets. Performance targets are L1 cache misses, TLB misses, cycles per thread, and power consumption. The use of KCAA restricts the scalability of their system as the complexity of model building grows exponentially with the number of features. In their evaluation, the system requires two hours of compute time to build the KCAA model for only 400 seconds of benchmark data. They present a compelling argument for the use of energy efficiency as an optimisation target in addition to runtime, citing that it was the power wall that lead to the multicore revolution in the first place. Their choice of only 2 benchmarks and 2 platforms makes the evaluation of their autotuner somewhat limited.

Berkeley2009 targeted 3D stencils code performance in [**Berkeley2009**]. Stencils are decomposed into core blocks, sufficiently small to avoid last level cache capacity misses. These are then further decomposed to thread blocks, designed to exploit common locality threads may have within a shared cache or local memory. Thread blocks are divided into register blocks in order to take advantage of data level parallelism provided by the available registers. Data allocation is optimised on NUMA systems. The performance evaluation considers speedups of various optimisations with and without consideration for host/device transfer overhead.

Kamil2010 present an autotuning framework in [**Kamil2010**] which accepts as input a Fortran 95 stencil expression and generates tuned shared-memory parallel implementations in Fortan, C, or CUDA. The system uses an IR to explore autotuning transformations, enumerating a subset of the optimisation space and recording only a single execution time for each configuration, reporting the fastest. They demonstrate their system on 4 architectures using 3 benchmarks, with speedups of up to 22× compared to serial implementations. The CUDA code generator does not optimise for the GPU memory hierarchy, using only global memory. As demonstrated in this thesis, improper utilisation of local

memory can hinder program performance by two orders of magnitude. There is no directed search or cross-program learning.

In [**Zhang2013a**], **Zhang2013a** present a code generator and autotuner for 3D Jacobi stencil codes. Using a DSL to express kernel functions, the code generator performs substitution from one of two CUDA templates to create programs for execution on GPUs. GPU programs are parameterised and tuned for block size, block dimensions, and whether input data is stored in read only texture memory. This creates an optimisation space of up to 200 configurations. In an evaluation of 4 benchmarks, the authors report impressive performance that is comparable with previous implementations of iterative Jacobi stencils on GPUs [**Holewinski2012**; **Phillips2010**]. The dominating parameter is shown to be block dimensions, followed by block size, then read only memory. The DSL presented in the paper is limited to expressing only Jacobi Stencils applications. Critically, their autotuner requires a full enumeration of the parameter space for each program. Since there is no indication of the compute time required to gather this data, it gives the impression that the system would be impractical for the needs of general purpose stencil computing. The autotuner presented in this thesis overcomes this drawback by learning parameter values which transfer to unseen stencils, without the need for an expensive tuning phase for each program and architecture.

In [**Christen2011**], **Christen2011** presentf a DSL for expressing stencil codes, a C code generator, and an autotuner for exploring the optimisation space of blocking and vectorisation strategies. The DSL supports stencil operations on arbitrarily high-dimensional grids. The autotuner performs either an exhaustive, multi-run Powell search, Nelder Mead, or evolutionary search to find optimal parameter values. They evaluate their system on two CPUS and one GPU using 6 benchmarks. A comparison of tuning results between different GPU architectures would have been welcome, as the results presented in this thesis show that devices have different responses to optimisation parameters. The authors do not present a ratio of the available performance that their system achieves, or how the performance of optimisations vary across benchmarks or devices.

A stencil grid can be decomposed into smaller subsections so that multiple GPUs can operate on each subsection independently. This requires a small overlapping region where each subsection meets — the halo region — to be shared between devices. For iterative stencils, values in the halo region must be syn-

chronised between devices after each iteration, leading to costly communication overheads. One possible optimisation is to deliberately increase the size of the halo region, allowing each device to compute updated values for the halo region, instead of requiring a synchronisation of shared state. This reduces the communication costs between GPUs, at the expense of introducing redundant computation. Tuning the size of this halo region is the goal of PARTANS [**Lutz2013**], an autotuning framework for multi-GPU stencil computations. **Lutz2013** explore the effect of varying the size of the halo regions using six benchmark applications, finding that the optimal halo size depends on the size of the grid, the number of partitions, and the connection mechanism (i.e. PCI express). The authors present an autotuner which determines problem decomposition and swapping strategy offline, and performs an online search for the optimal halo size. The selection of overlapping halo region size compliments the selection of workgroup size which is the subject of this thesis. However, PARTANS uses a custom DSL rather than the generic interface provided by SkelCL, and PARTANS does not learn the results of tuning across programs, or across multiple runs of the same program.

3.6 Summary

There is already a wealth of research literature on the topic autotuning which begs the question, why isn't the majority of software autotuned? In this chapter I attempted to answer the question by reviewing the state of the art in the autotuning literature, with specific reference to auotuning for GPUs and stencil codes. The bulk of this research falls prey of one of two shortcomings. Either they identify and develop a methodology for tuning a particular optimisation space but then fail to develop a system which can properly exploit this (for example, by using machine learning to predict optimal values across programs), or they present an autotuner which targets too specific of a class of optimisations to be widely applicable. This project attempts to address both of those shortcomings by expending great effort to deliver a working implementation which users can download and use without any setup costs, and by providing a modular and extensible framework which allows rapid targeting of new autotuning platforms, enabled by a shared autotuning logic and distributed training data. The following chapter outlines the design of this system.

Chapter 4

OmniTune - an Extensible, Dynamic Autotuner

4.1 Introduction

In previous chapters I have advocated the use of machine learning for autotuning. In this chapter, I present OmniTune, a framework for extensible, distributed autotuning using machine learning. OmniTune provides a replacement for the kinds of ad-hoc tuning typically performed by expert programmers by providing runtime prediction of tunable parameters using collaborative, online learning of optimisation spaces. First I describe the high level overview of the approach to autotuning, then I describe the system architecture and set of interfaces exposed by Omnitune.

4.2 Predicting Optimisation Parameter Values

The goal of machine learning-enabled autotuning is to *predict* the values for optimisation parameters to maximise some measure of profit. These predictions are based on models built from prior observations. The prediction quality is influenced by the number of prior observations. OmniTune supports both prediction of parameters based on prior observations, and a method for collecting these observations. When a client program requests parameter values, it indicates whether the request is for training or performance purposes, and uses a different backend to select parameter values for each. New observations can then be added once parameters have been evaluated. Figure 5.1 shows this process.

4.3 System Architecture and Interface

Common implementations of autotuning in the literature either: embed the autotuning logic within the each target application, or take a standalone approach in which the autotuner is a program which must be invoked by the user to tune a target application. Embedding the autotuner within each target application has the advantage of providing "always-on" behaviour, but is infeasible for complex systems in which the cost of building machine learning models must be added to each program run. The standalone approach separates the autotuning logic, at the expense of adding one additional step to the build process. The approach taken in OmniTune aims to capture the advantages of both techniques by implementing a autotuning *as a service*, with only the lightweight communication logic embedded in the target applications.

Omnitune is built around a three tier client-server model. The applications which are to be autotuned are the *clients*. These clients commmunicate with a system-wide *server*, which handles autotuning requests. The server communicates and caches data sourced from a *remote*, which maintains a global store of all autotuning data. Figure 4.2 shows this structure.

There is a many to one relationship between clients, servers, and remotes, such that a single remote may handle connections to multiple servers, which in turn may accept connections from multiple clients. This design has two primary advantages: the first is that it decouples the autotuning logic from that of the client program, allowing developers to easily repurpose the autotuning framework to target additional optimisation parameters without a significant development

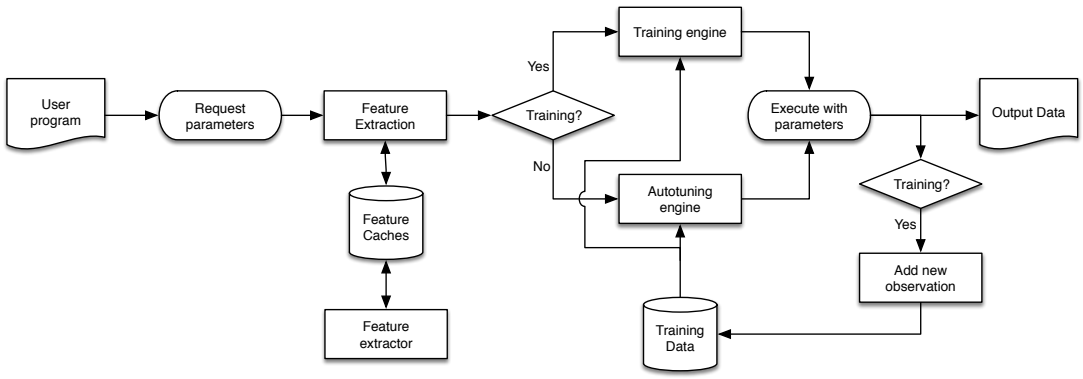


Figure 4.1: The process of selecting optimisation parameter values for a given user program with OmniTune.

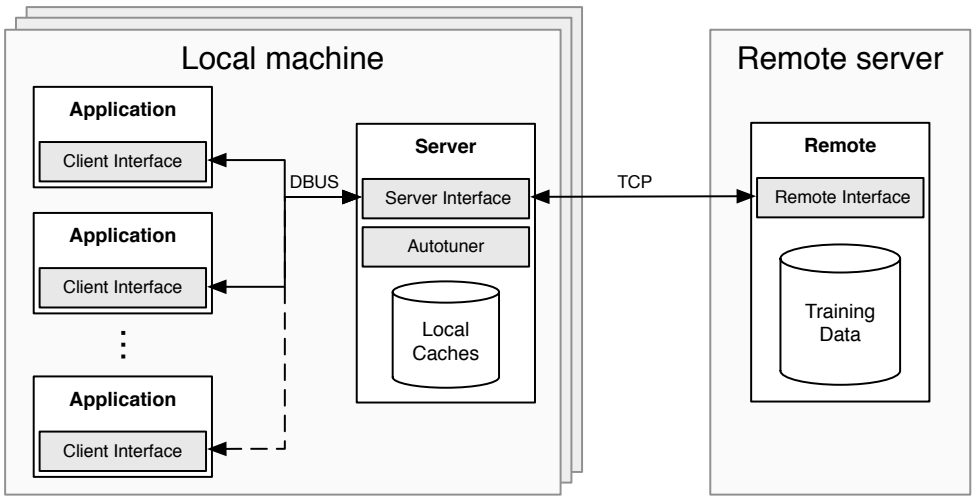


Figure 4.2: High level overview of OmniTune components.

overhead for the target applications; the second advantage is that this enables collective tuning, in which training data gathered from a range of devices can be accessed and added to by any OmniTune server.

OmniTune supports autotuning using a separate offline training phase, online training, or a mixture of both. Figure 4.3 shows an example pattern of communication between the three tiers of an OmniTune system, showing a distinct training phase. Note that this training phase is enforced only by the client. The following sections describe the interfaces between the three components.

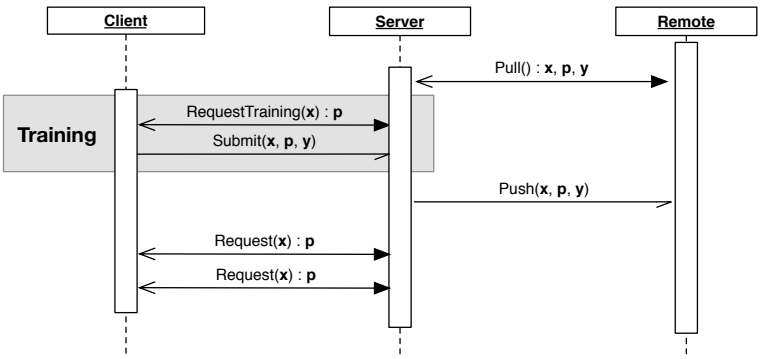


Figure 4.3: An example communication pattern between OmniTune components, showing an offline training phase.

4.3.1 Client Interface: Lightweight Communication

Client applications communicate with an OmniTune server through four operations:

- REQUEST(x) $\rightarrow p$ Given a set of explanatory variables x, request a set of parameter values p to maximise performance.
- REQUESTTRAINING(x) $\rightarrow p$ Given a set of explanatory variables x, allow the server to select a set of parameter values p for evaluating their fitness.
- SUBMIT(x, p, y) Submit an observed measurement of fitness y of parameters p, given explantory variables x.
- REFUSE(x, p) Refuse a set of parameters p given a set of explanatory variables x. Once refused, those parameters will not be returned by any subsequent calls to REQUEST() or REQUESTTRAINING().

This set of operations enables the core functionality of an autotuner, while providing flexibility for the client to control how and when training data is collected.

4.3.2 Server: Autotuning Engine

For each autotuning-capable machine, a system-level daemon hosts a DBus session bus which client processes communicate with. This daemon acts as an intermediate between the training data and the client applications, *serving* requests for optimisation parameter values. Servers operations are application-specific, so there is a set of operations to implement autotuning of each supported optimisation target.

The server is implemented as a standalone Python program, and contains a library of machine learning tools to perform parameter prediction, interfacing with Weka using the JNI. Weka is a suite of data mining software developed by the University of Waikato, freely available under the GNU GPL license [1]. OmniTune servers may perform additional feature extraction of explanatory variables supplied by incoming client requests. The reason for performing feature extraction on the server as opposed to on the client side is that this allows the results of expensive operations (for example, analysing source code of target applications) to be cached for use across the lifespan of client applications. The contents of

[1] http://www.cs.waikato.ac.nz/ml/weka/

these local caches are periodically and asynchronously synced with the remote, to maintain a global store of lookup tables for expensive operations.

On launch, the server requests the latest training data from the remote, which it uses to build the relevant models for performing prediction of optimisation parameter values. Servers communicate with remotes by submitting or requesting training data in batches, using two operations:

- PUSH($\mathbf{f}, \mathbf{c}$) Submit a set of labelled training points as pairs (f, c).
- PULL() $\rightarrow (\mathbf{f}, \mathbf{c})$ Request training data as a set of labelled (f, c) pairs.

4.3.3 Remote: Distributed Training Data

The role of the remote is to provide bookkeeping of training data for machine learning. Using the interface described in the previous section, remotes allow shared access to data from multiple servers using a transactional communication pattern.

4.4 Summary

This chapter has described the architecture of of OmniTune, a distributed autotuner which is capable of performing runtime prediction of optimal workgroup sizes using a variety of machine learning approaches. OmniTune uses a client-server model to decouple the autotuning logic from target programs and to maintain separation of concerns. It uses lightweight inter-process communication to achieve low latency autotuning, and uses caches and lookup tables to minimise the one-off costs of feature extraction.

Chapter 5

Autotuning SkelCL Stencils

5.1 Introduction

In this chapter I apply the OmniTune framework to SkelCL. The publicly available implementation [1] predicts workgroup sizes for OpenCL stencil skeleton kernels in order to minimise their runtime on CPUs and multi-GPU systems. The optimisation space presented by the workgroup size of OpenCL kernels is large, complex, and non-linear. Successfully applying machine learning to such a space requires plentiful training data, the careful selection of features, and classification approach. The following sections address these challenges.

5.2 Training

One challenge of performing empirical performance evaluations is gathering enough applications to ensure meaningful comparisons. Synthetic benchmarks are one technique for circumventing this problem. The automatic generation of such benchmarks has clear benefits for reducing evaluation costs; however, creating meaningful benchmark programs is a difficult problem if we are to avoid the problems of redundant computation and produce provable halting benchmarks.

In practise, stencil codes exhibit many common traits: they have a tightly constrained interface, predictable memory access patterns, and well defined numerical input and output data types. This can be used to create a confined space of possible stencil codes by enforcing upper and lower bounds on properties of the codes which can not normally be guaranteed for general-purpose programs,

[1] `https://github.com/ChrisCummins/omnitune`

e.g. the number of floating point operations. In doing so, it is possible to programatically generate stencil workloads which share similar properties to those which we intend to target.

Based on observations of real world stencil codes from the fields of cellular automata, image processing, and PDE solvers, I implemented a stencil generator which uses parameterised kernel templates to produce source codes for collecting training data. The stencil codes are parameterised by stencil shape (one parameter for each of the four directions), input and output data types (either integers, or single or double precision floating points), and *complexity* — a simple boolean metric for indicating the desired number of memory accesses and instructions per iteration, reflecting the relatively bi-modal nature of the reference stencil codes, either compute intensive (e.g. FDTD simulations), or lightweight (e.g. Game of Life).

Using a large number of synthetic benchmarks helps adress the "small n, large P" problem, which describes the difficulty of statistical inference in spaces for which the set of possible hypotheses P is significantly larger than the number of observations n. By creating parameterised, synthetic benchmarks, it is possible to explore a much larger set of the space of possible stencil codes than if relying solely on reference applications, reducing the risk of overfitting to particular program features.

5.3 Stencil Features

Properties of the architecture, program, and dataset all contribute to the performance of a workgroup size. The success of a machine learning system depends on the ability to translate these properties into meaningful explanatory variables — *features*. To capture this in OmniTune, parameter requests are packed with a copy of the OpenCL kernel and attributes of the dataset and device. The OmniTune server extracts 102 features describing hte architecture, kernel, and dataset from the message:

- **Device** — OmniTune uses the OpenCL `clGetDeviceInfo()` API to query a number of properties about the target execution device. Examples include the size of local memory, maximum work group size, number of compute units, etc.

Dataset Features	Type
Number of columns in matrix	numeric
Number of rows in matrix	numeric
Input data type	categorical
Output data type	categorical

Kernel Features	Type
Border region north	numeric
Border region south	numeric
Border region east	numeric
Border region west	numeric
Static instruction count	numeric
`AShr` instruction density	numeric
`Add` instruction density	numeric
`Alloca` instruction density	numeric
`And` instruction density	numeric
`Br` instruction density	numeric
`Call` instruction density	numeric
`FAdd` instruction density	numeric
`FCmp` instruction density	numeric
`FDiv` instruction density	numeric
`FMul` instruction density	numeric
`FPExt` instruction density	numeric
`FPToSI` instruction density	numeric
`FSub` instruction density	numeric
`GetElementPtr` instruction density	numeric
`ICmp` instruction density	numeric
`InsertValue` instruction density	numeric
`Load` instruction density	numeric
`Mul` instruction density	numeric
`Or` instruction density	numeric
`PHI` instruction density	numeric
`Ret` instruction density	numeric
`SDiv` instruction density	numeric
`SExt` instruction density	numeric
`SIToFP` instruction density	numeric
`SRem` instruction density	numeric
`Select` instruction density	numeric
`Shl` instruction density	numeric
`Store` instruction density	numeric
`Sub` instruction density	numeric
`Trunc` instruction density	numeric
`UDiv` instruction density	numeric
`Xor` instruction density	numeric
`ZExt` instruction density	numeric
Basic block density	numeric
Memory instruction density	numeric
Non external functions density	numeric
Kernel max workgroup size	numeric

Device Features	Type
SkelCL device count	numeric
Device address width	categorical
Double precision fp. configuration	categorical
Big endian?	categorical
Execution capabilities	categorical
Supported extensions	categorical
Global memory cache size	numeric
Global memory cache size	categorical
Global memory cacheline size	numeric
Global memory size	numeric
Host unified memory?	categorical
2D image max height	numeric
2D image max width	numeric
3D image max depth	numeric
3D image max height	numeric
3D image max width	numeric
Image support	categorical
Local memory size	numeric
Local memory type	categorical
Max clock frequency	numeric
Number of compute units	numeric
Max kernel constant args	numeric
Max constant buffer size	numeric
Max memory allocation size	numeric
Max parameter size	numeric
Max read image arguments	numeric
Max samplers	numeric
Max device workgroup size	numeric
Max workitem dimensions	numeric
Max work item sizes width	numeric
Max work item sizes height	numeric
Max work item sizes depth	numeric
Max write image arguments	numeric
Mem base address align	numeric
Min data type align size	numeric
Native vector width `char`	numeric
Native vector width `double`	numeric
Native vector width `float`	numeric
Native vector width `half`	numeric
Native vector width `int`	numeric
Native vector width `long`	numeric
Native vector width `short`	numeric
Preferred vector width `char`	numeric
Preferred vector width `double`	numeric
Preferred vector width `float`	numeric
Preferred vector width `half`	numeric
Preferred vector width `int`	numeric
Preferred vector width `long`	numeric
Preferred vector width `short`	categorical
Queue properties	categorical
Single precision fp. configuration	categorical
Device type	categorical
OpenCL vendor	categorical
OpenCL vendor ID	categorical
OpenCL version	categorical

Table 5.1: OmniTune SkelCL Stencil features for dataset, kernel, and device.

- **Kernel** — The user code for a stencil is passed to the OmniTune server, which compiles the OpenCL kernel to LLVM IR bitcode. The `opt InstCount` statistics pass is used to obtain static counts for each type of instruction present in the kernel, as well as the total number of instructions. The instruction counts for each type are divided by the total number of instructions to produce a *density* of instruction for that type. Examples include total static instruction count, ratio of instructions per type, ratio of basic blocks per instruction, etc.
- **Dataset** — The SkelCL container type is used to extract the input and output data types, and the 2D grid size.

See Table 5.1 for a list of feature names and types.

5.3.1 Reducing Feature Extraction Overhead

Feature extraction (particularlly compilation to LLVM IR) introduces a runtime overhead to the classification process. To minimise this, lookup tables for device and dataset features are used, and cached locally in the OmniTune server and pushed to the remote data store. The device ID is used to index the devices table, and the checksum of an OpenCL source is used to index the kernel features table. Before feature extraction for either occurs, a lookup is performed in the relevant table, meaning that the cost of feature extraction is amortised over time.

5.4 Optimisation Parameters

SkelCL stencil kernels are parameterised by a workgroup size w, which consists of two integer values to denote the number of rows and columns (where we need to distinguish the individual components, we will use symbols w_r and w_c to denote rows and columns, respectively).

5.4.1 Constraints

Unlike in many autotuning applications, the space of optimisation parameter values is subject to hard constraints, and these constraints cannot conviently be statically determined. Contributing factors are architectural limitations, kernel constraints, and refused parameters.

Architectural constraints

Each OpenCL device imposes a maximum workgroup size which can be statically checked by querying the `clGetDeviceInfo()` API for that device. These are defined by archiectural limitations of how code is mapped to the underlying execution hardware. Typical values are powers of two, e.g. 1024, 4096, 8192.

Kernel constraints

At runtime, once an OpenCL program has been compiled to a kernel, users can query the maximum workgroup size supported by that kernel using the `clGetKernelInfo()` API. This value cannot easily be obtained statically as there is no mechanism to determine the maximum workgroup size for a given source code and device without first compiling it, which in OpenCL does not occur until runtime. Factors which affect a kernel's maximum workgroup size include the number registers required for a kernel, and the available number of SIMD execution units for each type of instructions in a kernel.

Refused parameters

In addition to satisfying the constraints of the device and kernel, not all points in the workgroup size optimisation space are guaranteed to provide working programs. A refused parameter is a workgroup size which satisfies the kernel and architectural constraints, yet causes a `CL_OUT_OF_RESOURCES` error to be thrown when the kernel is enqueued. Note that in many OpenCL implementations, this error type acts as a generic placeholder and may not necessarily indicate that the underlying cause of the error was due to finite resources constraints.

Legality

We define a *legal* workgroup size as one which, for a given *scenario* (a combination of program, device, and dataset), satisfies the architectural and kernel constraints, and is not refused. The subset of all possible workgroup sizes $W_{legal}(s) \subset W$ that are legal for a given sceanario s is then:

$$W_{legal}(s) = \{w | w \in W, w < W_{\max}(s)\} - W_{refused}(s) \tag{5.1}$$

Where $W_{\max}(s)$ can be determined at runtime prior to the kernels execution, but the set $W_{refused}(s)$ can only be determined experimentally.

5.4.2 Assessing Relative Performance

Given a set of observations, where an observation is a scenario, workgroup size tuple (s, w); a function $t(s, w)$ which returns the arithmetic mean of the runtimes for a set of observations; we can calculate the speedup $r(s, w_1, w_2)$ of competing workgroup sizes w_1 over w_2 using:

$$r(s, w_1, w_2) = \frac{t(s, w_2)}{t(s, w_1)} \tag{5.2}$$

Oracle Workgroup Size

The *oracle* workgroup size $\Omega(s) \in W_{legal}(s)$ of a sceanrio s is the w value which provides the lowest mean runtime. This allows for comparing the performance $p(s, w)$ of a particular workgroup against the maximum available performance for that scenario, within the range $0 \leq p(s, w) \leq 1$:

$$\Omega(s) = \underset{w \in W_{legal}(s)}{\arg\min}\ t(s, w) \tag{5.3}$$

$$p(s, w) = r(s, w, \Omega(s)) \tag{5.4}$$

Establishing a Baseline

The geometric mean is used to aggregate normalised relative performances due to its multiplicative property [**Fleming1986**]. For a given workgroup size, the average performance $\bar{p}(w)$ across the set of all scenarios S can be found using the geometric mean of performance relative to the oracle:

$$\bar{p}(w) = \left(\prod_{s \in S} r(s, w, \Omega(s)) \right)^{1/|S|} \tag{5.5}$$

The *baseline* workgroup size $\bar{w}$ is the value which provides the best average case performance across all scenarios. Such a baseline value represents the *best* possible performance which can be achieved using a single, statically chosen workgroup size. By defining $W_{safe} \in W$ as the intersection of legal workgroup sizes, the baseline can be found using:

$$W_{safe} = \cap \left\{ W_{legal}(s) | s \in S \right\} \tag{5.6}$$

$$\bar{w} = \underset{w \in W_{safe}}{\arg\max}\, \bar{p}(w) \tag{5.7}$$

5.5 Machine Learning

The challenge is to design a system which, given a set of prior observations of the empirical performance of stencil codes with different workgroup sizes, predict workgroup sizes for *unseen* stencils which will maximise the performance. The OmniTune server supports three methods for achieving this.

5.5.1 Predicting Oracle Workgroup Size

The first approach to predicting workgroup sizes is to consider the set of possible workgroup sizes as a hypothesis space, and to use a classifier to predict, for a given set of features, the workgroup size which will provide the best performance. The classifier takes a set of training scenarios $S_{training}$, and generates labelled training data as pairs of scenario features $f(s)$ and observed oracle workgroup sizes:

$$T = \{(f(s), \Omega(s)) \,|s \in S_{training}\} \tag{5.8}$$

During testing, the classifier predicts workgroup sizes from the set of oracle workgroup sizes from the training set:

$$W_{training} = \{\Omega(s) | s \in S_{training}\} \tag{5.9}$$

This approach presents the problem that after training, there is no guarantee that the set of workgroup sizes which may be predicted is within the set of legal workgroup sizes for future scenarios:

$$\bigcup_{\forall s \in S_{testing}} W_{legal}(s) \nsubseteq W_{training} \tag{5.10}$$

This may result in a classifier predicting a workgroup size which is not legal for a scenario, $w \notin W_{legal}(s)$, either because it exceeds $W_{\max}(s)$, or because the parameter is refused. For these cases, I evaluate the effectiveness of three fallback strategies to select a legal workgroup size:

1. *Baseline* — select the workgroup size which is known to be safe $w < W_{safe}$, and provides the highest average case performance on training data.

2. *Random* — select a random workgroup size which is known from prior observations to be legal $w \in W_{legal}(s)$.

3. *Nearest Neighbour* — select the workgroup size which from prior observations is known to be legal, and has the lowest Euclidian distance to the prediction.

See Algorithm 1 for definitions.

5.5.2 Predicting Stencil Code Runtime

A problem of predicting oracle workgroup sizes is that it requires each training point to be labelled with the oracle workgroup size which can be only be evaluated using an exhaustive search. An alternative approach is to build a model to attempt to predict the *runtime* of a stencil given a specific workgroup size. This allows for training on data for which the oracle workgroup size is unknown, and has the secondary advantage that this allows for an additional training data point to be gathered each time a stencil is evaluated. Given training data consisting of $(f(s), w, t)$ tuples, where s is a scenario, w is the workgroup size, and t is the observed mean runtime, we can train a regressor $g(f(s), w)$ which predicts the mean runtime of an unseen scenario. The predicted oracle workgroup size $\bar{\Omega}(s)$ is then the w value which minimises the output of the regressor:

$$\bar{\Omega}(s) = \underset{w \in W_{legal}(s)}{\arg\min}\ g(f(s), w) \tag{5.11}$$

Note that since we cannot know in advance which workgroup sizes will be refused, that is, $W_{refused}(s)$ cannot be statically determined, this process must be iterated until a workgroup size which not refused is selected. Algorithm 2 shows this process.

5.5.3 Predicting Relative Performance of Workgroup Sizes

Accurately predicting the runtime of an arbitrary program is a difficult problem due to the impacts of flow control. In such cases, it may be more effective to instead predict the *relative* performance of two different workgroup sizes for the same program. To do this, we select a baseline workgroup size $w_b \in W_{safe}$, and train a regressor $g(f(s), w, w_b)$ with training data labelled with the relative performance over the baseline $r(w, w_b)$. Predicting the optimal workgroup requires maximising the output of the regressor:

$$\bar{\Omega}(s) = \underset{w \in W_{legal}(s)}{\arg\max}\ g(f(s), w, w_b) \tag{5.12}$$

Algorithm 1 Selecting optimal workgroup sizes using classification

Require: scenario s d.

Ensure: workgroup size w

procedure BASELINE(s) ▷ Select the best w from W_{safe}.
  $w \leftarrow \text{classify}(f(s))$
  if $w \in W_{legal}(s)$ **then**
    return w
  else
    return $\underset{w \in W_{safe}}{\arg\max} \left(\prod_{s \in S_{training}} p(s,w) \right)^{1/|S_{training}|}$
  end if
end procedure

procedure RANDOM(s) ▷ Select a random workgroup size.
  $w \leftarrow \text{classify}(f(s))$
  while $w \notin W_{legal}(s)$ **do**
    $w \leftarrow$ random choice $w \in \{w | w < W_{max}(s), w \notin W_{refused}(s)\}$
  end while
  return w
end procedure

procedure NEARESTNEIGHBOUR(s) ▷ Select the closest workgroup size to prediction.
  $w \leftarrow \text{classify}(f(s))$
  while $w \notin W_{legal}(s)$ **do**
    $d_{min} \leftarrow \infty$
    $w_{closest} \leftarrow$ null
    for $c \in \{w | w < W_{\max}(s), w \notin W_{refused}(s)\}$ **do**
      $d \leftarrow \sqrt{(c_r - w_r)^2 + (c_c - w_c)^2}$
      if $d < d_{min}$ **then**
        $d_{min} \leftarrow d$
        $w_{closest} \leftarrow c$
      end if
    end for
    $w \leftarrow w_{closest}$
  end while
  return w
end procedure

Algorithm 2 Selecting workgroup sizes by predicting program runtimes

Require: kernel features k, hardware features h, dataset features d.

Ensure: workgroup size w

1: $W \leftarrow \{w | w < W_{\max}(s)\} - W_{refused}(s)$ ▷ Set of possible workgroup sizes.
2: $w \leftarrow \arg\min_{w \in W} g(f(s), w)$ ▷ Predict candidate workgroup size.
3: **while** $w \notin W_{legal}(s)$ **do**
4: $W \leftarrow W - \{w\}$ ▷ Remove candidate from set.
5: $w \leftarrow \arg\min_{w \in W} g(f(s), w)$ ▷ Select next candidate workgroup size.
6: **end while**
7: **return** w

Algorithm 3 Selecting workgroup sizes by predicting relative performance

Require: kernel features k, hardware features h, dataset features d, baseline w_b.

Ensure: workgroup size w

1: $W \leftarrow \{w | w < W_{\max}(s)\} - W_{refused}(s)$ ▷ Set of possible workgroup sizes.
2: $w \leftarrow \arg\max_{w \in W} g(f(s), w, w_b)$ ▷ Predict candidate workgroup size.
3: **while** $w \notin W_{legal}(s)$ **do**
4: $W \leftarrow W - \{w\}$ ▷ Remove candidate from set.
5: $w \leftarrow \arg\max_{w \in W} g(f(s), w, w_b)$ ▷ Select next candidate workgroup size.
6: **end while**
7: **return** w

As with predicting runtimes, this process must be iterated to accommodate for the emergent properties of $W_{legal}(s)$. See Algorithm 3 for a description of this process.

5.6 Implementation

The OmniTune framework is implemented as a set of Python classes and interfaces, which are inherited from or implemented to target specific autotuning cases. The entire framework consists of 8987 lines of Python code, of which 976 is dedicated to the SkelCL frontend. By design, the client-server model minimises the impact of number of modifications that are required to enable autotuning in client applications. The only modification required is to replace the hardcoded values for workgroup size with a subroutine to request a workgroup size from the OmniTune server over a DBUS connection. The server is implemented as a standalone Python program, and uses sqlite to maintain local data caches. The OmniTune remote is an Amazon Web Services virtual machine instance, using

MySQL as the relational data store. Figure 5.1 shows the relational database schema used to store stencil runtime information. Additional tables store data in coalesced forms for use as machine learning training data.

For classification, five classifiers are supported, chosen for their contrasting properties: Naive Bayes, SMO, Logistic Regression, J48 Decision tree, and Random Forest. For regression, a Random Forest with regression trees is used, chosen because of its efficient handling of large feature sets, compared to linear models.

5.7 Summary

This section has described has the application of OmniTune for predicting workgroup sizes of SkelCL stencil programs, using three different machine learning approaches. The first approach is to predict the optimal workgroup size for a given program based on training data which included the optimal workgroup sizes for other stencils. The second approach is to select a workgroup sizex by sweeping the space of possible workgroup sizes, predicting the runtime a program with each. The third approach is to select a workgroup size by sweeping the space of possible workgroup sizes, predicting the relative gain of each compared to a known baseline. In the next section, we will describe the process for collecting empirical performance data.

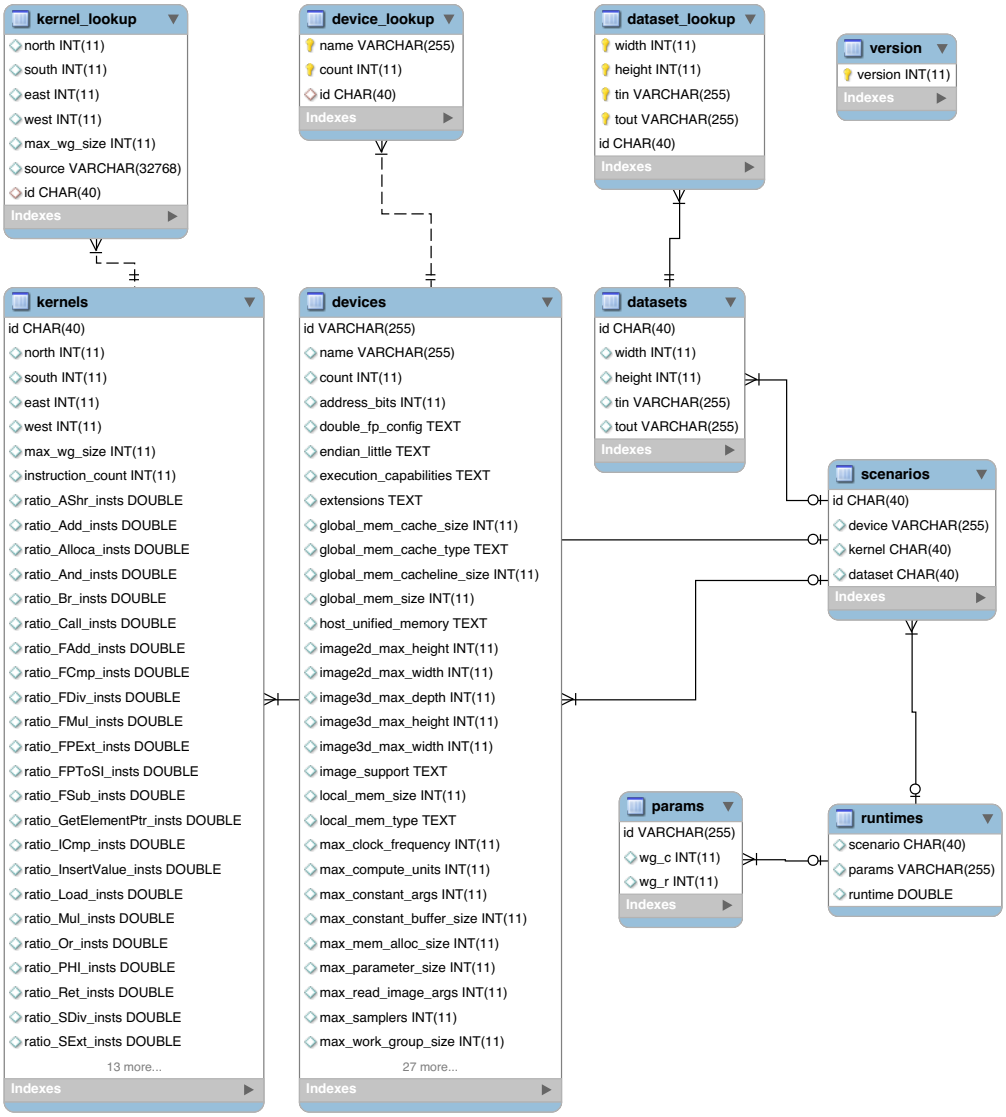


Figure 5.1: Database schema for storing SkelCL stencil runtimes. Feature lookup tables and normalisation are used to provide extremely compact storage, requiring only 56 bytes for each additional runtime of a known stencil program.

Chapter 6

Exploring the Workgroup Size Space

6.1 Introduction

This chapter describes an exhaustive enumeration of the workgroup size optimisation space for 429 combinations of architecture, program, and dataset. It contains the methodology used to collect empirical performance data on which to base performance comparisons of different workgroup sizes, and the steps necessary to obtain repeatable results.

6.2 Experimental Setup

Table 6.1 describes the experimental platforms used. All runtimes were recorded with millisecond precision using either the system clock or OpenCL's Profiling API. Measurement noise was minimised by reducing system load through dis-

Processor	Memory	OpenCL Devices	Driver
Intel i7-2600	8 GB	Nvidia GTX TITAN	OpenCL 1.1
Intel i7-2600K	16 GB	Nvidia GTX 690	OpenCL 1.1
Intel i7-3820	8 GB	4× Nvidia GTX 590	OpenCL 1.1
Intel i7-3820	8 GB	CPU, 2× AMD Tahiti 7970	OpenCL 1.2
Intel i5-4570	8 GB	CPU	OpenCL 1.2
Intel i5-2430M	8 GB	CPU	OpenCL 1.2

Table 6.1: Specification of experimental platforms.

Name	Compute units	Frequency	Local Memory	Global Cache	Global Memory
AMD Tahiti 7970	32	1000 Hz	32 KB	16 KB	2959 MB
Intel i5-2430M	4	2400 Hz	32 KB	256 KB	7937 MB
Intel i5-4570	4	3200 Hz	32 KB	256 KB	7901 MB
Intel i7-3820	8	1200 Hz	32 KB	32 KB	7944 MB
Nvidia GTX 590	1	1215 Hz	48 KB	256 KB	1536 MB
Nvidia GTX 690	8	1019 Hz	48 KB	128 KB	2048 MB
Nvidia GTX TITAN	14	980 Hz	48 KB	224 KB	6144 MB

Table 6.2: Specification of experimental OpenCL devices.

abling all unwanted services and graphical environments, and exclusive single-user access was ensured for each platform. Frequency governors for each CPU were disabled, and the benchmark processes were set to the highest priority available to the task scheduler. Datasets and programs were stored in an in-memory file system.

6.2.1 Devices

Table 6.2 describes the OpenCL devices used for testing, as available on the experimental platforms.

6.2.2 Benchmark Applications

In addition to the synthetic stencil benchmarks described in Section 5.2, six stencil kernels taken from four reference implementations of standard stencil applications from the fields of image processing, cellular automata, and partial differential equation solvers are used:

- **Game of Life** Conway's Game of Life [**Conway1970**] is a cellular automaton which models the evolution of a regular grid of cells over discrete time steps. At each time step, each cell value is updated to be either *live* or *dead* based on it's current state and the state of the one immediately neighbouring cell to the north, south, east, and west.

- **Heat Equation** The heat equation is a partial differential equation which describes the distribution of heat in a given region over time. Each iteration

of the stencil represents a discrete time step, and the value of each cell (i.e. the temperature) is smoothed based on the temperatures of surrounding cells and the thermal conductivity of the material being simulated.

- **Gaussian Blur** The Gaussian blur is a common image processing algorithm, used to reduce noise and detail in an image. A two dimensional Gaussian blur defines a function to compute a pixel value based on the value of neighbouring pixels. Gaussian blurs are parameterised by a radius which define symmetric, square stencil regions about the centre pixel. Unlike the previous two applications, the Gaussian blur is not an iterative stencil.

- **Canny Edge Detection** The Canny edge detection algorithm is a multi-stage approach to detecting edges in images [**Canny1986**]. It consists of four distinct stages: a noise reduction operation, an edge detection operation, a non-maximum suppression, and a threshold operation. Each step is implemented as a separate SkelCL stencil and combined into a SkelCL StencilSequence.

Table 6.3 shows details of the stencils kernels for these reference applications, and the synthetic training benchmarks used.

6.2.3 Datasets

For each benchmark, multiple dataset sizes were used, as shown in Table 6.4.

6.2.4 Sampling Strategy

The number of "moving parts" in the modern software stack provides multiple sources of noise when measuring program execution times. As such, evaluating the relative performance of different versions of programs requires a judicious approach to isolate the appropriate performance metrics and to take a statistically rigorous approach to collecting data.

Isolating the Impact of Workgroup Size

The execution of a SkelCL stencil application can be divided into 6 distinct phases, shown in Table 6.5.

Name	North	South	East	West	Instruction Count
complex	30	30	30	30	161
complex	1	10	30	30	681
complex	20	10	20	10	161
complex	5	5	5	5	734
complex	5	5	5	5	161
complex	1	10	30	30	154
complex	10	10	10	10	161
complex	20	20	20	20	706
complex	1	1	1	1	137
complex	20	10	20	10	706
complex	1	1	1	1	661
complex	10	10	10	10	794
complex	30	30	30	30	706
complex	20	20	20	20	161
simple	5	5	5	5	67
simple	20	10	20	10	612
simple	20	20	20	20	612
simple	1	10	30	30	592
simple	10	10	10	10	700
simple	30	30	30	30	612
simple	1	1	1	1	93
simple	30	30	30	30	67
simple	20	10	20	10	67
simple	5	5	5	5	640
simple	20	20	20	20	67
simple	10	10	10	10	67
simple	0	0	0	0	40
simple	1	10	30	30	65
simple	1	1	1	1	617
gaussian	5	5	5	5	83
gaussian	5	5	5	5	655
gaussian	5	5	5	5	657
gaussian	0	0	0	0	46
gaussian	5	5	5	5	82
gol	1	1	1	1	714
gol	1	1	1	1	190
he	1	1	1	1	113
he	1	1	1	1	637
nms	1	1	1	1	748

Width	Height	Type in	Type out
512	512	float	float
1024	1024	float	float
2048	2048	float	float
4096	4096	float	float
4096	4096	int	int

Table 6.4: Description of experimental datasets.

	Description	Type	Cost
$\boldsymbol{c}$	Kernel compilation times	Host	Fixed
$\boldsymbol{p}$	Skeleton prepare times	Host	Fixed
$\boldsymbol{u}$	Host $\rightarrow$ Device transfers	Device	Fixed
$\boldsymbol{k}$	Kernel execution times	Device	Iterative
$\boldsymbol{d}$	Device $\rightarrow$ Host transfers	Device	Fixed
$\boldsymbol{s}$	Devices $\leftrightarrow$ Host (sync) transfers	Host	Iterative

Table 6.5: Execution phases of a SkelCL stencil skeleton. "Fixed" costs are those which occur up to once per stencil invocation. "Iterative" costs are those which scale with the number of iterations of a stencil.

- **Kernel compilation times** Upon invocation, template substitution is performed of the user code into the stencil skeleton implementation, then compiled into an OpenCL program. Once compiled, the program binary is cached for the lifetime of the host program.

- **Skeleton preparation times** Before a kernel is executed, a preparation phase is required to allocate buffers for the input and output data on each execution device.

- **Host → Device and Device → Host transfer times** Data must be copied to and from the execution devices before and after execution of the stencils, respectively. Note that this is performed lazily, so iterative stencils do not require repeated transfers between host and device memory.

- **Kernel execution times** This is the time elapsed executing the stencil kernel, and is representative of "work done".

- **Devices ↔ Host (sync) transfer times** For iterative stencils on multiple execution devices, an overlapping halo region is shared at the border between the devices' grids. This must be synchronised between iterations, requiring an intermediate transfer to host memory, since device to device memory is not currently supported by OpenCL.

For each of the six distinct phases of execution, accurate runtime information can be gathered either through timers embedded in the host code, or using the OpenCL `clGetEventProfilingInfo()` API for operations on the execution devices. For single-device stencils, the total time t of a SkelCL stencil application is simply the sum of all times recorded for each distinct phase:

$$t = \mathbf{1}\boldsymbol{c}^{\boldsymbol{T}} + \mathbf{1}\boldsymbol{p}^{\boldsymbol{T}} + \mathbf{1}\boldsymbol{u}^{\boldsymbol{T}} + \mathbf{1}\boldsymbol{k}^{\boldsymbol{T}} + \mathbf{1}\boldsymbol{d}^{\boldsymbol{T}} \tag{6.1}$$

Note that there are no synchronisation costs s. For applications with n execution devices, the runtime can be approximate as the sum of the sequential host-side phases, and the sum of the device-side phases divided by the number of devices:

$$t \approx \sum_{i=1}^{n} (\mathbf{1}\boldsymbol{c}_i^{\boldsymbol{T}}) + \mathbf{1}\boldsymbol{p}^{\boldsymbol{T}} + \mathbf{1}\boldsymbol{s}^{\boldsymbol{T}} + \frac{\sum_{i=1}^{n} \mathbf{1}\boldsymbol{u}_i^{\boldsymbol{T}} + \mathbf{1}\boldsymbol{k}_i^{\boldsymbol{T}} + \mathbf{1}\boldsymbol{d}_i^{\boldsymbol{T}}}{n} \tag{6.2}$$

The purpose of tuning workgroup size is to maximise the throughput of stencil kernels. For this reason, isolating the kernel execution times $\boldsymbol{k}$ produces the

most accurate performance comparisons, as it removes the impact of constant overheads introduced by memory transfers between host and device memory, for which the selection of workgroup size has no influence. Note that as demonstrated in [**Gregg2011**], care must be taken to ensure that isolating device compute time does not cause misleading comparisons to be made between devices. For example, if using an autotuner to determine whether execution of a given stencil is faster on a CPU or GPU, the device transfer times $\boldsymbol{u}$, $\boldsymbol{d}$, and $\boldsymbol{s}$ would need to be considered. For our purposes, we do not need to consider the location of the data in the system's memory as it is has no bearing on the execution time of a stencil kernel.

Validating Program Behaviour

Gold standard output was recorded by executing each of the real-world benchmarks programs using the baseline workgroup size. The output of real-world benchmarks with other workgroup sizes was compared to this gold standard output to guarantee correct program execution.

6.3 Summary

This section describes the methodology for collecting relative performance data of SkelCL stencil benchmarks under different combinations of architecture, program, dataset, and workgroup size. The next chapter evaluates these performance results, and analyses the performance of OmniTune at predicting workgroup sizes.

Chapter 7

Evaluation

7.1 Introduction

This chapter evaluates the performance of OmniTune when tasked with selecting workgroup sizes for SkelCL stencil codes. First I discuss measurement noise present in the experimental results, and the methods used to accommodate for it. Then I examine the observed effect that workgroup size has on the performance of SkelCL stencils. The effectiveness of each of the autotuning techniques described in the previous chapters is evaluated using multiple different machine learning algorithms. The prediction quality of OmniTune is scrutinised for portability across programs, devices, and datasets.

Overview of Experimental Results

The experimental results consist of measured runtimes for a set of *test cases*, collected using the methodology explained in the previous chapter. Each test case τ_i consists of a scenario, workgroup size pair $\tau_i = (s_i, w_i)$, and is associated with a *sample* of observed runtimes from multiple runs of the program. A total of 269813 test cases were evaluated, which represents an exhaustive enumeration of the workgroup size optimisation space for 429 scenarios. For each scenario, runtimes for an average of 629 (max 7260) unique workgroup sizes were measured. The average sample size of runtimes for each test case is 83 (min 33, total 16917118).

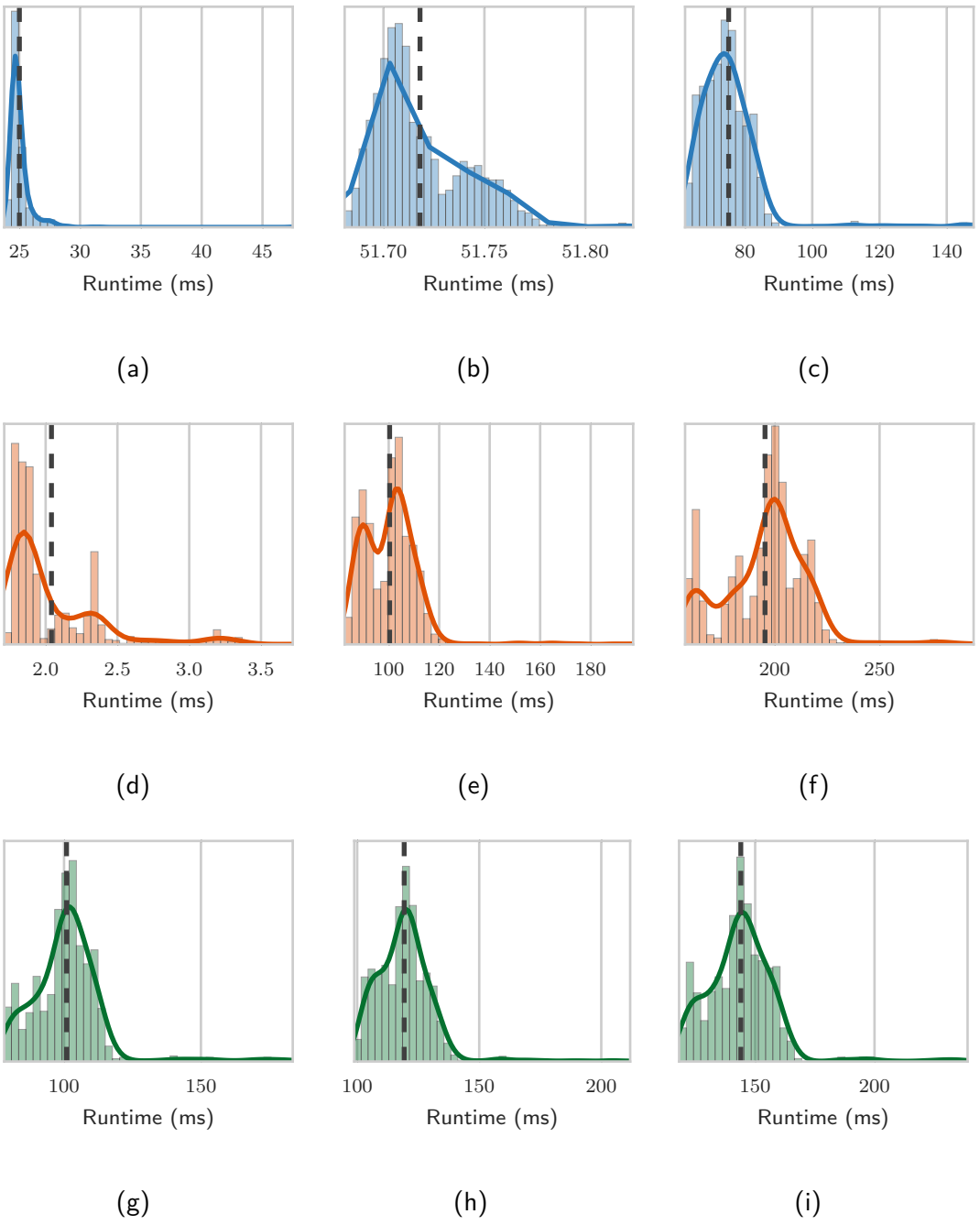


Figure 7.1: Distribution of runtime samples for test cases from three devices. Each plot contains a 35-bin histogram of 1000 samples, and a fitted kernel density estimate with bandwidth 0.3. The sample mean is shown as a vertical dashed line. The top row are from the Intel i5-4570, the second row from the Nvidia GTX 590, and the third row from the AMD Tahiti 7970. In some of the plots, the distribution of runtimes is bimodal, and skewed to the lower end of the runtimes range.

7.2 Statistical Soundness

The complex interaction between processes competing for the finite resources of a system introduces many sources for noise in program runtime measurements. Before making any judgements about the relative performance of optimisation configurations, we must establish the level of noise present in these measurements. To do this, we evaluate the distribution of runtimes for a randomly selected 1000 test cases, recording 1000 runtime observations for each. We can then produce fine-grained histograms of runtimes for individual test cases. Figure 7.1 shows an example nine of these, for test cases from three devices. The plots show that the distribution of runtimes is not always Gaussian; rather, it is sometimes bimodal, and generally skewed to the lower end of the runtime range, with a long "tail" to the right. This fits our intuition that programs have a hard *minimum* runtime enforced by the time taken to execute the instructions of a program, and that noise introduced to the system extends this runtime. For example, preempting an OpenCL process on a CPU so that another process may run may cause the very long tail visible in Figure 7.1a.

The central limit theorem allows the assumption of an underlying Gaussian distribution for samples of size ≥ 30 [**Georges2007**]. Given our minimum sample size of 33, we can use 95% confidence intervals to provide statistical confidence that the arithmetic mean of observed runtimes with respect to the true mean. As the number or samples increases, we should expect the size of the confidence interval to shrink. This is illustrated in Figure 7.2, which plots the average size of 95% confidence intervals across the 1000 test cases, normalised to their respective means, as a function of sample size. It shows the diminishing returns that increasing sample size provides. For example, increasing the sample count from 10 to 30 results in an approximate 50% reduction in confidence interval size. Increasing the sample size from 30 to 50 results in only a 25% reduction.

By comparing the average confidence interval at different sample counts against the full experiment results of 269813 test cases, we can assert with 95% confidence that the true mean for each test case is within 2.5% of the sample mean (given the average number of samples per test case), or 3.7% in the worst case (at the minimum number of samples). Since the differences between baseline and optimal workgroup sizes is often well in excess of 100%, there is no overlap of confidence intervals between competing workgroup sizes.

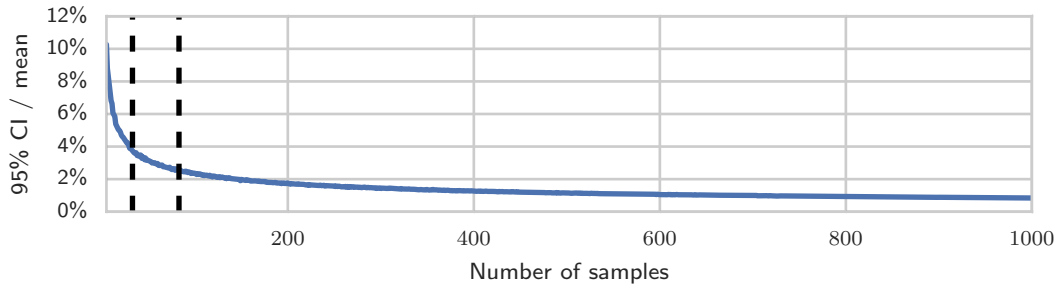


Figure 7.2: Ratio of confidence interval to mean as a function of sample count. Two dashed lines indicate the confidence intervals at the minimum (3.7%) and mean (2.5%) number of samples used in the experimental dataset.

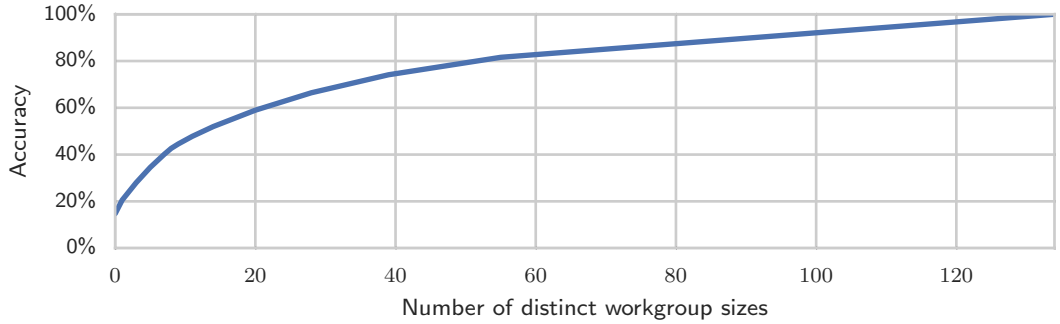


Figure 7.3: Accuracy compared to the oracle as a function of the number of workgroup sizes used. The best accuracy that can be achieved using a single statically chosen workgroup size is 15%. Achieving 50% oracle accuracy requires a minimum of 14 distinct workgroup sizes.

7.3 Workgroup Size Optimisation Space

In this section we explore the impact that the workgroup size optimisation space has on the performance of stencil codes.

7.3.1 Oracle Workgroup Sizes

For each scenario s, the oracle workgroup size $\Omega(s)$ is the workgroup size which resulted in the lowest mean runtime. If the performance of stencils were independent of workgroup size, we would expect that the oracle workgroup size would remain constant across all scenarios $s \in S$. Instead, we find that there are 135 unique oracle workgroup sizes, with 31.5% of scenarios having a unique work-

group size. This demonstrates the difficult in attempting to tune for *optimal* parameter values, since 14 distinct workgroup sizes are needed to achieve just 50% of the oracle accuracy (Figure 7.3), although it is important to make the distinction that oracle *accuracy* and *performance* are not equivalent.

Figure 7.4a shows the distribution of oracle workgroup sizes, demonstrating that there is clearly no "silver bullet" workgroup size which is optimal for all scenarios, and that the space of oracle workgroup sizes is non linear and complex. The workgroup size which is most frequently optimal is $w_{(64\times4)}$, which is optimal for 15% of scenarios. Note that this is not adequate to use as a baseline for static tuning, as it does not respect legality constraints, that is $w_{(64\times4)} \notin W_{safe}$.

7.3.2 Workgroup Size Legality

As explained in Section 5.4, the space of legal workgroup sizes $W_{legal}(s)$ for a given scenario s comprises all workgroup sizes which: do not exceed the maximum allowed by the OpenCL device and kernel $W_{\max}(s)$, and are not refused by the OpenCL runtime.

Maximum workgroup sizes

We define the *coverage* of a workgroup size to be the ratio $0 \leq x \leq 1$ between the number of scenarios for which the workgroup size was less than $W_{\max}(s)$, normalised to the total number of workgroup sizes. A coverage of 1 implies a workgroup size which is always legal for all combinations of stencil and architecture. A workgroup size with a coverage of 0 is never legal. Figure 7.5 plots the coverage of a subset of the workgroup size optimisation space.

Note that since $W_{\max}(s)$ defines a hard limit for a given s, if statically selecting a workgroup size, one must limit the optimisation space to the smallest $W_{\max}(s)$ value, i.e. only the workgroup sizes with a coverage of 1. The observed $W_{\max}(s)$ values range from 256–8192, which results in up to a 97% reduction in the size of the optimisation space when $W_{\max}(s) = 8192$, even though only 14% of scenarios have the minimum value of $W_{\max}(s) = 256$.

Refused Parameters

In addition to the hard constraints imposed by the maximum workgroup size, there are also refused parameters, which are workgroup sizes which are rejected

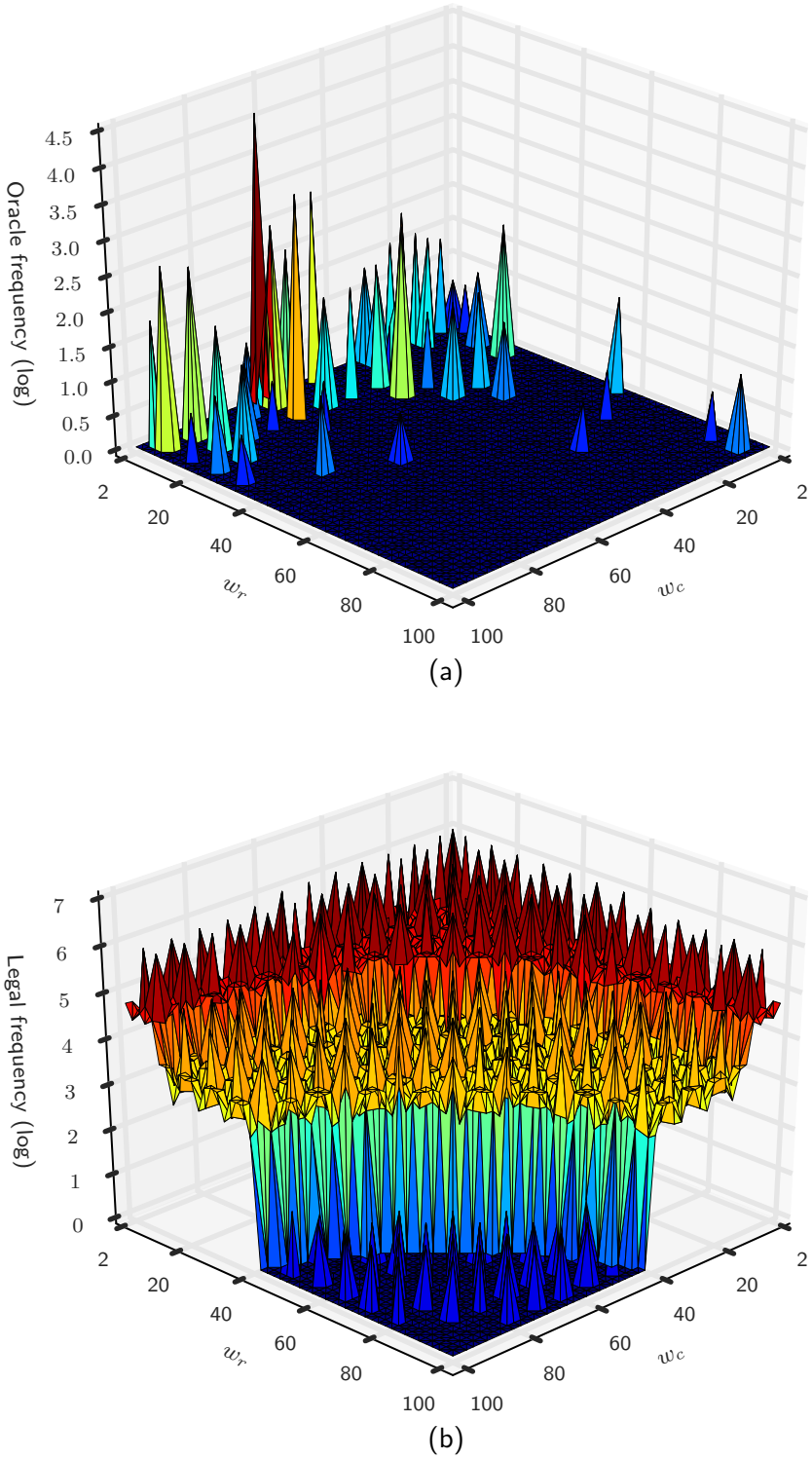


Figure 7.4: Log frequency counts for: (a) optimality, and (b) legality for a subset of the aggregated workgroup size optimisation space, $w_c \leq 100, w_r \leq 100$. The space of oracle workgroup size frequencies is highly irregular and uneven, with a peak frequency of $w_{(64\times4)}$. Legality frequencies are highest for smaller row and column counts (where $w < W_{\max}(s) \forall s \in S$), and w_c and w_r values which are multiples of 8.

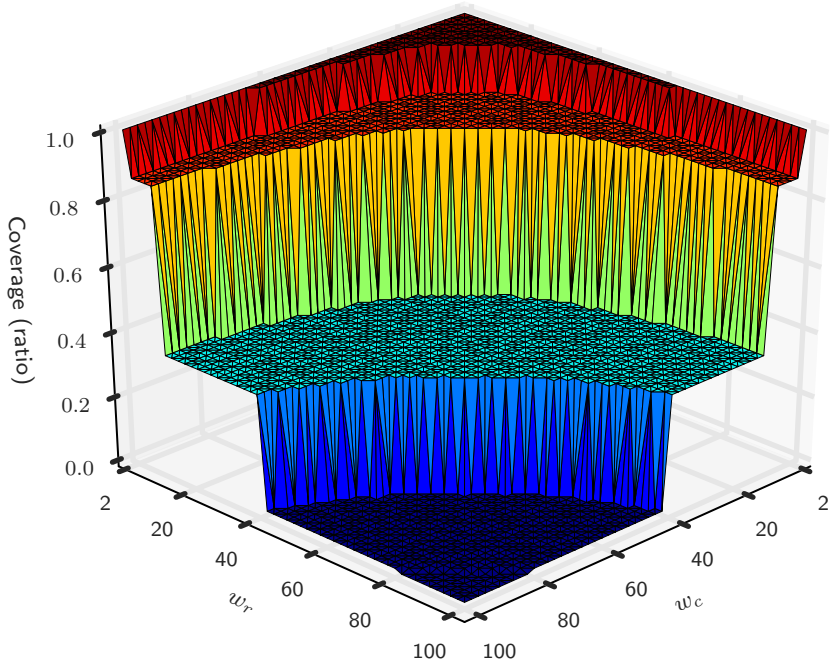


Figure 7.5: A subset of the aggregated workgroup size optimisation space, $w_c \leq 100, w_r \leq 100$, showing the *coverage* of each workgroup size, i.e. the ratio of scenarios for which a workgroup size satisfies architecture and kernel enforced constraints ($W_{\max}(s)$). Workgroup sizes with a coverage of < 1 fail to satisfy these constraints for one or more scenarios. Only workgroup sizes with a coverage of 1 may be used for static tuning, which greatly reduces the size of the optimisation space. Observed $W_{\max}(s)$ values are multiples of 256, hence the abrupt "steps" in coverage.

Parameter	Refused (%)
18×24	0.32
26×16	0.31
28×32	0.31
4×2	0.31
16×14	0.30
14×40	0.30
24×36	0.30
36×24	0.30
4×26	0.30
4×6	0.30

Parameter	Refused (%)
8×18	0.30
8×22	0.30
14×48	0.29
28×24	0.29
30×8	0.29
48×14	0.29
8×10	0.29
10×8	0.28
24×18	0.28
26×32	0.28

Parameter	Refused (%)
26×8	0.28
2×32	0.28
2×48	0.28
4×42	0.28
50×4	0.28
10×48	0.28
12×24	0.28
16×18	0.28
24×2	0.28
48×18	0.28

Table 7.1: The thirty most refused parameters, ranked in descending order. There is little correlation between the size of workgroup and the likelihood that it is refused, suggesting that the cause of refused parameters is not a resource constraint, but a behavioural issue.

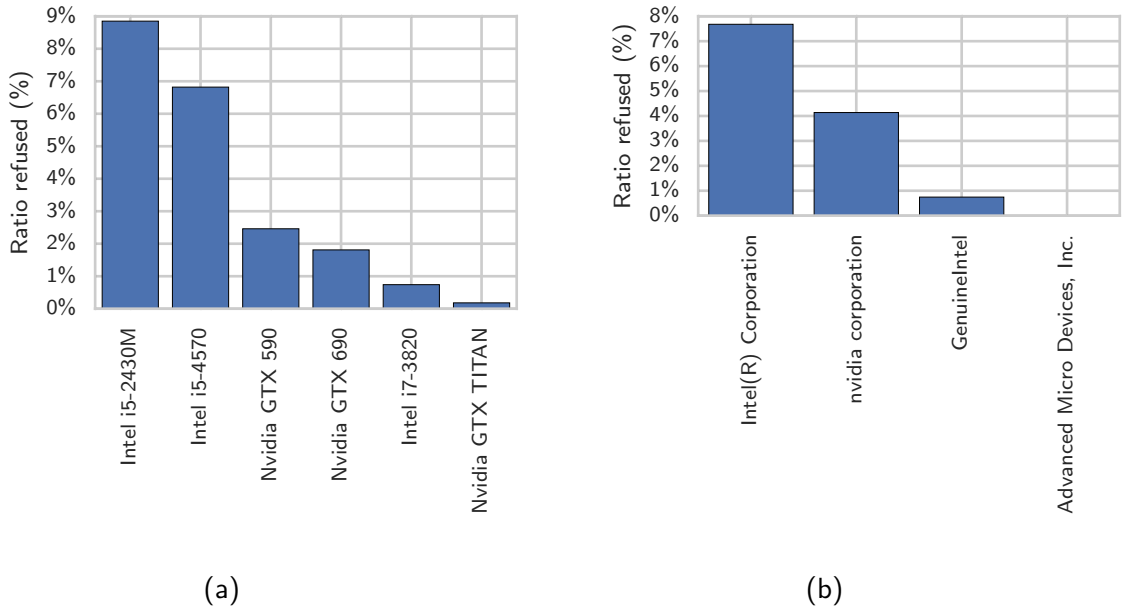


Figure 7.6: The ratio of test cases with refused workgroup sizes, grouped by: (a) OpenCL device ID; (b) device vendor ID. Parameters were refused most frequently by Intel i5 CPUs, then by previous-generation NVIDIA GPUs. No parameters were refused by AMD devices.

by the OpenCL runtime and do not provide a functioning program. Of the 8504 unique workgroup sizes tested, 11.4% were refused in one or more test cases. An average of 5.5% of all test cases lead to refused parameters. For a workgroup size to be refused, it must satisfy the architectural and program-specific constraints which are exposed by OpenCL, but still lead to a `CL_OUT_OF_RESOURCES` error when the kernel is enqueued. Table 7.1 lists the most frequently refused parameters, and the percentage of test cases for which they were refused. While uncommon, a refused parameter is an obvious inconvenience to the user, as one would expect that any workgroup size within the specified maximum should behave *correctly*, if not efficiently. Figure 7.4b visualises the space of legal workgroup sizes by showing the frequency counts that a workgroup size is legal. Smaller workgroup sizes are legal most frequently due to the $W_{\max}(s)$ constraints. Beyond that, workgroup sizes which contain w_c and w_r values which are multiples of eight are more frequently legal, which is a common width of SIMD vector operations [**IntelCorporation2012**].

Experimental results suggest that the problem is vendor — or at least device — specific. By grouping the refused test cases by device and vendor, we see a

much greater quantity of refused parameters for test cases on Intel CPU devices than any other type, while no workgroup sizes were refused by the AMD GPU. Figure 7.6 shows these groupings. The exact underlying cause for these refused parameters is unknown, but can likely by explained by inconsistencies or errors in specific OpenCL driver implementations.

As these OpenCL implementations are still in active development, it is anticipated that errors caused by unexpected behaviour will become more infrequent as the technology matures. Figure 7.6a shows that the ratio of refused parameters decreases across the three generations of Nvidia GPUs: GTX 590 (2011), GTX 690 (2012), and GTX TITAN (2013). The same trend is apparent for the two Intel i5s: i5-2430M (2011), and i5-4570 (2013), although not for the i7-3820 (2012). For now, it is imperative that any autotuning system is capable of adapting to these refused parameters by suggesting alternatives when they occur.

7.3.3 Baseline Parameter

The baseline parameter $\bar{w}$ is the workgroup size which provides the best overall performance while being legal for all scenarios. It is the workgroup size $w \in W_{safe}$ which maximises the output of the performance function $\bar{p}(w)$. As shown in Table 7.2, only a *single* workgroup size $w_{(4\times4)}$ from the set of experimental results is found to have a legality of 100%, suggesting that an adaptive approach to setting workgroup size is necessary not just for the sake of maximising performance, but also for guaranteeing program correctness.

The utility of the baseline parameter is that it represents the best performance that can be achieved through static tuning of the workgroup size parameter. We can evaluate the performance of suboptimal workgroup sizes by calculating the geometric mean of their *performance* for a particular scenario $p(s, w)$ across all scenarios, $\bar{p}(w)$. The baseline parameter $\bar{p}(\bar{w})$ achieves only 24% of the available performance. Figure 7.7 plots workgroup size *legality* and *performance*, showing that there is no clear correlation between the two. In fact, the workgroup sizes with the highest mean performance are valid only for scenarios with the largest $W_{\max}(s)$ value, which account for less than 1% of all scenarios, further reinforcing the case for adaptive tuning. The workgroup sizes with the highest legality are listed in Table 7.2, and the workgroup sizes with the highest performance are listed in Table 7.3.

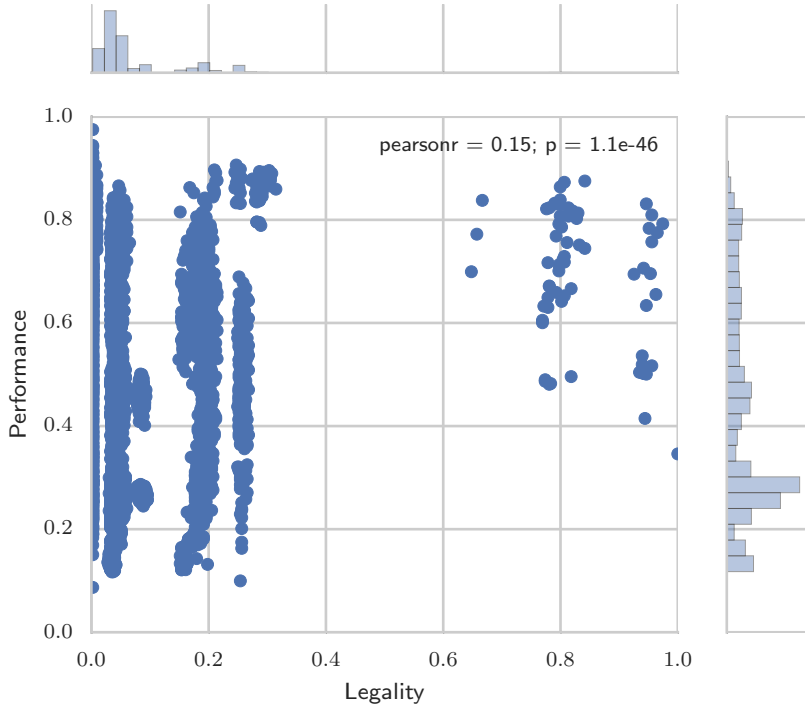


Figure 7.7: Average legality and performance relative to the oracle of all workgroup sizes. Clearly, the relationship between legality and performance is not linear. Distinct vertical "bands" appear between regions of legality caused by the different $W_{\max}(s)$ values of devices. The irregular jitter between these vertical bands is caused by refused parameters.

Figure 7.8 shows the speedup of the oracle workgroup size over the baseline parameter $w_{(4\times4)}$ for all scenarios. If we assume that sufficiently pragmatic developer with enough time would eventually find this static optimal, then this provides a reasonable comparison for calculating speedups of an autotuner for workgroup size. Comparing the runtime of workgroup sizes relative to the oracle allows us to calculate upper bounds on the possible performance which can be expected from autotuning.

7.3.4 Speedup Upper Bounds

For a given scenario s, the ratio of the workgroups sizes from $W_{legal}(s)$ which provide the longest and shortest mean runtimes is used to calculate an upper

Parameter	Legality (%)	Performance (%)
4×4	100.0	34.6
32×4	97.4	79.2
40×4	96.5	77.5
16×4	96.3	65.5
56×4	95.6	81.0
16×8	95.6	75.7
8×4	95.6	51.7
24×4	95.3	69.6
24×8	95.1	78.4
48×4	94.6	83.1
8×8	94.6	63.4
4×16	94.6	50.1
4×8	94.4	41.5
8×16	94.2	70.6
4×40	94.2	50.2
4×32	93.9	53.6
4×48	93.9	51.9
4×56	93.9	50.3
4×24	93.5	50.4
8×24	92.5	69.5
64×4	84.1	87.5
16×16	84.1	74.5
24×16	83.2	75.1
32×8	83.0	81.3
40×16	82.8	80.3

Table 7.2: The 25 workgroup sizes with the greatest legality.

Parameter	Legality (%)	Performance (%)
270×24	0.2	97.5
174×38	0.2	94.4
310×20	0.2	94.2
546×10	0.2	93.0
282×16	0.2	93.0
520×10	0.2	92.9
520×12	0.2	92.3
746×8	0.2	92.0
38×140	0.2	91.9
300×18	0.2	91.3
700×6	0.2	91.1
96×24	24.7	90.6
88×48	0.7	90.6
820×6	0.2	90.4
88×32	24.9	90.0
96×32	25.4	89.8
88×40	21.0	89.7
80×16	30.3	89.6
722×10	0.2	89.5
280×24	0.2	89.5
910×6	0.2	89.4
88×24	28.7	89.4
64×24	29.4	89.2
80×32	24.5	89.2
72×16	29.4	89.2

Table 7.3: The 25 workgroup sizes with the greatest mean performance.

bound for the possible performance influence of workgroup size:

$$r_{max}(s) = r(s, \underset{w \in W_{legal}(s)}{\arg\max}\ t(s,w), \Omega(s)) \tag{7.1}$$

When applied to each scenario $s \in S$ of the experimental results, we find the average of speedup upper bounds to be 15.14× (min 1.03×, max 207.72×). This demonstrates the importance of tuning stencil workgroup sizes — if chosen incorrectly, the runtime of stencil programs can be extended by up to 207.72×. Note too that for 5 of the scenarios, the speedup of the best over worst workgroup sizes is $\leq 5\%$. For these scenarios, there is little benefit to autotuning; however, this represents only 1.1% of the tested scenarios. For 50% of the scenarios, the speedup of the best over worst workgroup sizes is $\geq 6.19\times$.

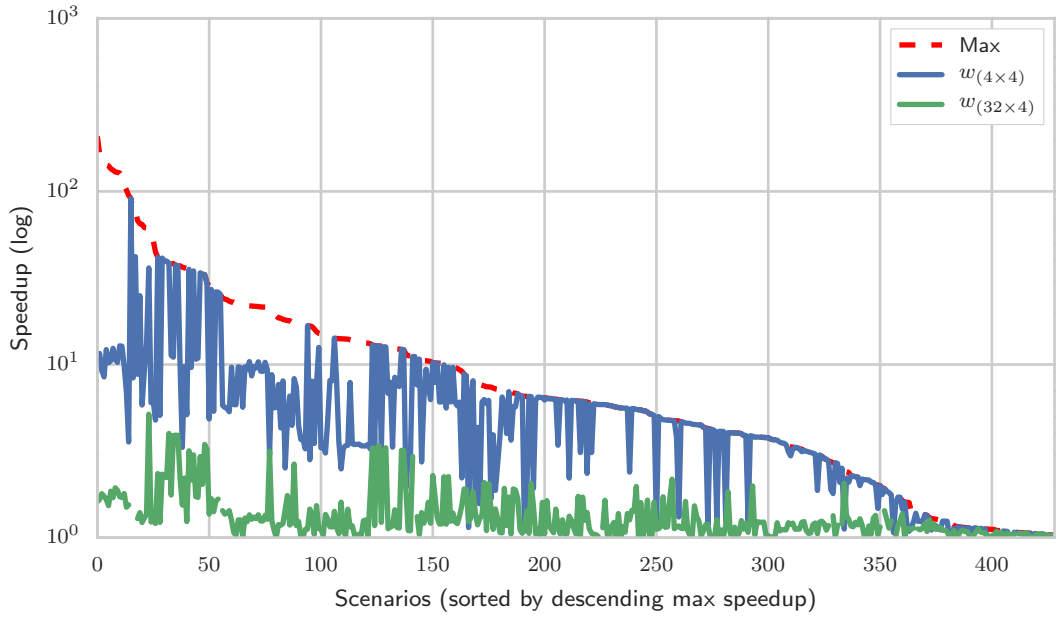


Figure 7.8: Speedup of oracle workgroup size over: the worst performing workgroup size for each scenario (*Max*), the statically chosen workgroup size that provides the best overall performance ($w_{(4\times4)}$), and the human expert selected parameter ($w_{(32\times4)}$). Note that the human expert parameter is not legal for all scenarios.

7.3.5 Human Expert

In the original implementation of the SkelCL stencil skeleton [**Breuer2013**], **Breuer2013** selected a workgroup size of $w_{(32\times4)}$ in an evaluation of 4 stencil operations on a Tesla S1070 system. We can use this as an additional parameter to compare the relative performance of workgroup sizes against. However, the $w_{(32\times4)}$ workgroup size is invalid for 2.6% of scenarios, as it is refused in 11 test cases. By device, those are: 3 on the GTX 690, 6 on the i5-2430M, and 2 on the i5-4570. For the scenarios where $w_{(32\times4)}$ *is* legal, the human expert chosen workgroup size achieves an impressive geometric mean of 79.2% of the oracle performance. The average speedup of oracle workgroup sizes over the workgroup size selected by a human expert is 1.37× (min 1.0×, max 5.17×). Since the workgroup size selected by the human expert is not legal for all scenarios, we will examine the effectiveness of heuristics for tuning workgroup size.

7.3.6 Heuristics

In this section we will consider the effectiveness of instead selecting workgroup size using two types of heuristics. The first, using the maximum workgroup size returned by the OpenCL device and kernel APIs to select the workgroup size adaptively. The second, using per-device heuristics, in which the workgroup size is selected based on the specific architecture that a stencil is operating on.

Using maximum legal size

A common approach taken by OpenCL developers is to set the workgroup size for a kernel based on the maximum legal workgroup size queried from the OpenCL APIs. For example, to set the size of 2D workgroup, a developer the square root of the (scalar) maximum wgsize to set the number of columns and rows (since $w_c \cdot w_r$ must be $< W_{\max}(s)$). To consider the effectiveness of this approach, we group the workgroup size performances based on the ratio of the maximum allowed for each scenario. We can also perform this for each of the two dimensions — rows and columns — of the stencil workgroup size.

Figure 7.9 shows the distribution of runtimes when grouped this way, demonstrating that the performance of (legal) workgroup sizes are not correlated with the maximum workgroup sizes $W_{\max}(s)$. However, when considering individual components, we observe that the best median workgroup size performances are achieved with a number of columns that is between 10% and 20% of the maximum, and a number of rows that is between 0% and 10% of the maximum.

Per-device workgroup sizes

One possible technique to selecting workgroup size is to tune particular values for each targeted execution device. This approach is sometimes adopted for cases with particularly high requirements for performance on a single platform, so it produces an interesting contrast to evaluating a machine learning approach, which attempts to predict workgroup sizes for unseen platforms without the need for an expensive exploration phase on the new platform.

Figure 7.10 shows the performance of workgroup sizes relative to the oracle across scenarios grouped by: kernel, device, and dataset. When grouped like this, a number of observations can made. First is that not all of the kernels are sensitive to tuning workgroup size to the same degree. The *sobel* kernel has the

Device	Oracle	Legality	Perf. min	Perf. avg.
AMD Tahiti 7970	48×4	1.0	0.54	0.91
Intel i5-2430M	64×16	0.8	0.37	0.91
Intel i5-4570	88×8	0.89	0.33	0.89
Intel i7-3820	40×24	0.95	0.76	0.97
NVIDIA GTX 590	$12 \times x2$	0.8	0.2	0.9
NVIDIA GTX 690	64×4	0.93	0.32	0.84
NVIDIA GTX TITAN	64×4	1.0	0.26	0.81
GPUs	64×4	0.76	0.26	0.86
CPUs	88×8	0.88	0.33	0.91

Table 7.4: Selecting workgroup size using a per-device heuristic. The mode optimal workgroup size for each device type $\bar{w}$ is evaluated based on legality, and relative performance to the oracle (minimum and average) when legal.

lowest median performance, indicating that it is the most profitable to tune, while the *threshold* kernel is the least profitable. Similarly, the Intel i7-3820 is far less amenable to tuning than the other devices, while the Intel i5-4570 is the most sensitive to the workgroup size parameter. Such variances in the distributions of workgroup sizes suggest that properties underlying the architecture, kernel, and dataset all contribute towards the proper selection of workgroup size.

To test the performance of a per-device heuristic for selecting workgroup size, we group the scenarios by device, and compare the relative performance of all workgroup sizes for each group of scenarios. The most frequently optimal workgroup size $\bar{w}$ for each device is selected, and the legality and performance of each scenario using that workgroup size is evaluated. Table 7.4 shows the results of this evaluation. The GTX 690 and GTX TITAN share the same $\bar{w}_{(64\times4)}$ value, while every other device has a unique optimum. The general case performance of these per-device parameters is very good, although legality is only assured for the GTX TITAN and AMD 7970 (which did not refuse any parameters). However, the worst case performance of per-device workgroup sizes is poor for all except the i7-3820 (which is least sensitive to tuning), suggesting that device alone is not capable of reliably informing the choice of workgroup size.

7.3.7 Summary

In this section we have explored the performance impact of the workgroup size optimisation space. By comparing the relative performance of an average of 629 workgroup sizes for each of 429 scenarios, the following conclusions can be drawn:

1. The performance of a workgroup size for a particular scenario depends properties of the hardware, software, and dataset.

2. The performance gap between the best and workgroup sizes for a particular combination of hardware, software, and dataset is up to 207.72×.

3. Not all workgroup sizes are legal, and the space of legal workgroup sizes cannot statically be determined. Adaptive tuning of workgroup size is required to ensure reliable performance.

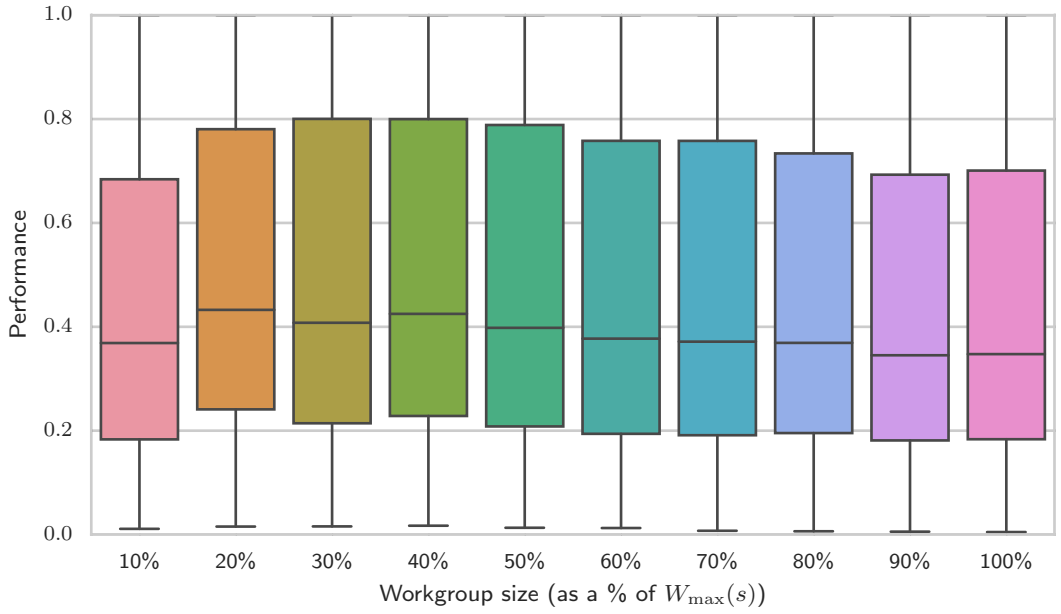
4. Differing scenarios have wildly different optimal workgroup sizes, and the best performance can be achieved using static tuning is optimal for only 15% of scenarios.

In the following section, we will evaluate the performance of OmniTune for selecting workgroup sizes.

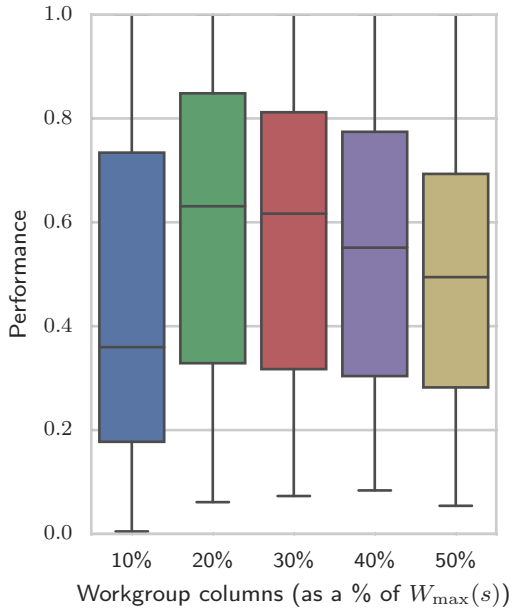
7.4 Autotuning Workgroup Sizes

In this section, we evaluate the performance of OmniTune for predicting workgroup sizes of SkelCL skeletons using the prediction techniques described in Section 5.5. For each technique, we partition the experimental data into training and testing sets, $S_{training} \subset S$ and $S_{testing} = S - S_{training}$. A set of labelled training data $D_{training}$ is derived from $S_{training}$, and the prediction quality is testing using the validation set $D_{testing}$ derived from $S_{training}$. We use multiple approaches to partitioning test data to evaluate the prediction quality under different scenarios. The processes for generating validation sets are:

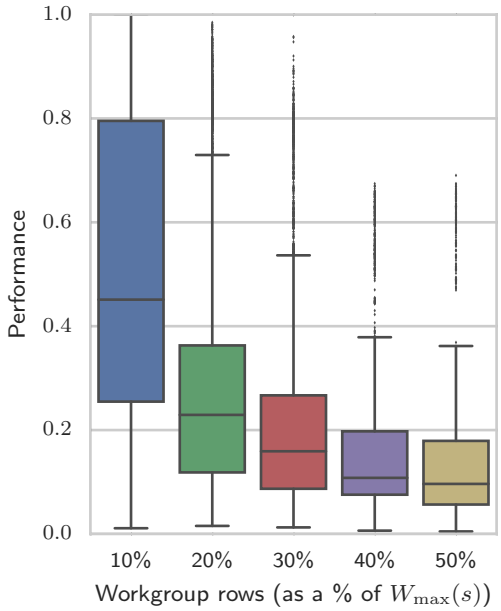
- 10-fold — shuffle the set of all data and divide into 10 validation sets, each containing 10% of the data. This process is repeated for 10 rounds, resulting in 100 validations of 10 permutations of the data.



(a)

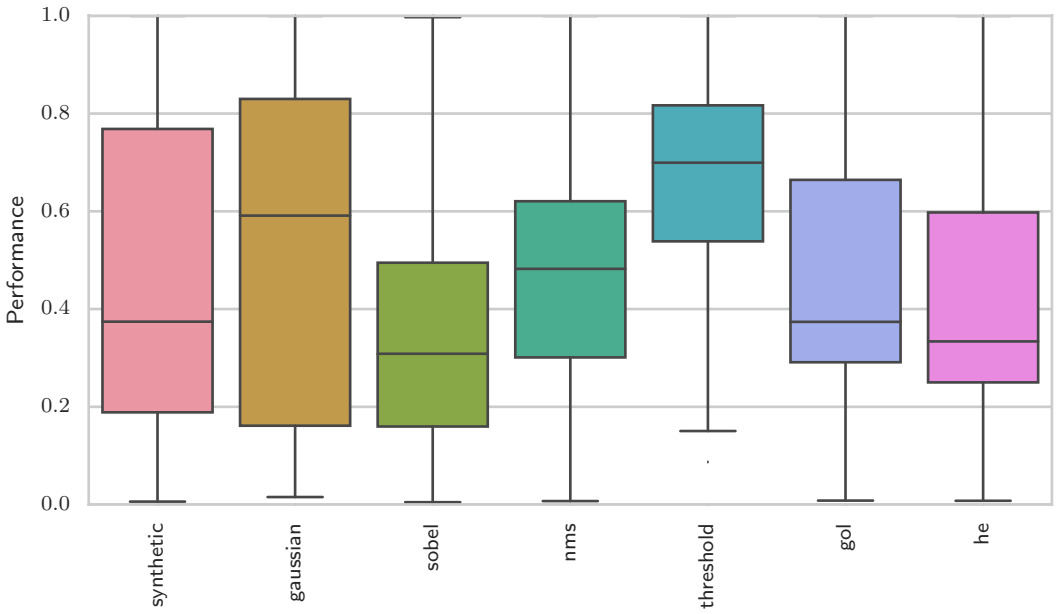


(b)

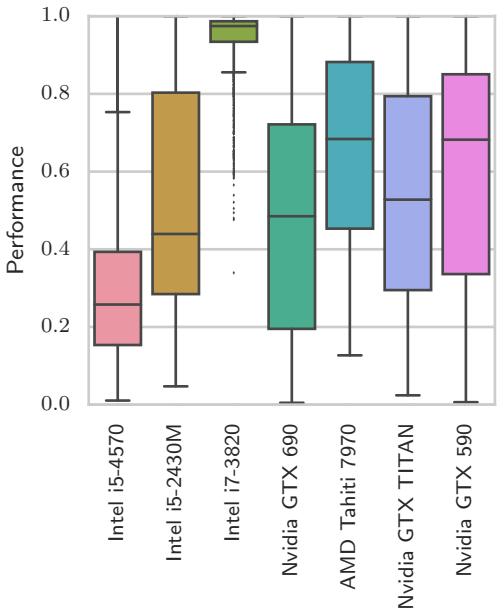


(c)

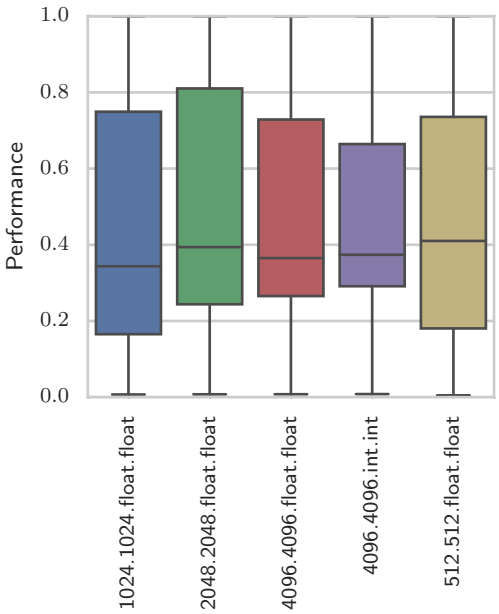
Figure 7.9: Comparing workgroup performance relative to the oracle as function of: (a) maximum legal size, (b) number of columns, and (c) number of rows. Each workgroup size is normalised to the maximum allowed for that scenario, $W_{\max}(s)$.



(a)



(b)



(c)

Figure 7.10: Performance relative to the oracle of workgroup sizes across all test cases, grouped by: (a) kernels, (b) devices, and (c) datasets.

- Synthetic — divide the training data such that it consists solely of data collected from synthetic benchmarks, and use data collected from real-world benchmarks to test.
- Device — partition the training data into n sets, one for each device. Use $n-1$ sets for training, repeating until every partition has been used for testing once.
- Kernel — partition the training data into n sets, one for each kernel. Use $n-1$ sets for training, repeating until every partition has been used for testing once.
- Dataset — partition the training data into n sets, one for each type of dataset. Use $n-1$ sets for training, repeating until every partition has been used for testing once.

For each autotuning technique, the results of testing using the different validation sets are reported separately. The autotuning techniques evaluated are: using classifiers to predict the optimal workgroup size of a stencil, with fallback strategies to handle illegal predictions; using regressors to predict the runtime of a stencil using different workgroup sizes, and selecting the legal workgroup size which has the lowest predicted runtime; and using regressors to predict the relative performance of workgroup sizes over a baseline, and selecting the workgroup size which has the highest predicted relative performance. We first describe the evaluation strategies for each technique, before presenting experimental results and analysis.

7.4.1 Evaluating Classifiers

The methodology for selecting workgroup size using classifiers is described in Section 5.5.1. Training data consists of pairs of feature vectors $f(s)$ and oracle workgroup sizes $\Omega(s)$:

$$D_{training} = \{(f(s), \Omega(s)) | s \in S_{training}\} \tag{7.2}$$

Testing data are not labelled with oracle workgroup sizes:

$$D_{testing} = \{f(s) | s \in S_{testing}\} \tag{7.3}$$

Each classifier is evaluated using the three different classification techniques: Baseline, Random, and NearestNeighbour, which differ in the way in

which they handle illegal predictions. Illegal predictions occur either because the classifier has suggested a parameter which does not satisfy the maximum workgroup size constraints $w < W_{\max}(s)$, or because the workgroup size is refused by OpenCL $w \in W_{refused}(s)$. Workgroup sizes are predicted for each scenario in the testing set, and the quality of the predicted workgroup size is evaluated using the following metrics:

- accuracy (binary) — whether the predicted workgroup size is the true oracle, $p(f(s)) = \Omega(s)$.
- validity (binary) — whether the classifier predicted a workgroup size which satisfies the workgroup size constraints constraints, $p(f(s)) < W_{\max}(s)$.
- refused (binary) — whether the classifier predicted a workgroup size which is refused, $p(f(s)) \in W_{refused}(s)$.
- performance (real) — the relative performance of the predicted workgroup size relative to the oracle, $0 \leq r(p(f(s)), \Omega(s)) \leq 1$.
- speedups (real) — the relative performance of the predicted workgroup size relative to the baseline workgroup size $w_{(4\times4)}$, and human expert workgroup size $w_{(32\times4)}$ (where applicable).
- time (real) — the round trip time required to make the prediction, as measured by the OmniTune client. This includes classification time and interprocess communication overheads between the client and server.

The *validty* and *refused* metrics measure how often fallback strategies are required to select a legal workgroup size $w \in W_{legal}(s)$.

7.4.2 Evaluating Regressors

Sections 5.5.2 and 5.5.3 describe methodologies for selecting workgroup sizes by predicting program runtimes or relative performance, respectively. The evaluation approach for both methodologies is the same, only the training data differs. For predicting runtimes, training data consists of feature vectors, labelled with the mean observed runtime $t(s, w)$ for all legal workgroup sizes:

$$D_{training} = \bigcup_{\forall s \in S_{training}} \Big\{ (f(s), t(s,w)) | w \in W_{legal}(s) \Big\} \tag{7.4}$$

For predicting speedups, the features vectors are labelled with observed speedup over the baseline parameter $\bar{w}$ (see Section 7.3.3) for all legal workgroup sizes:

$$D_{training} = \cup \Big\{ (f(s), r(s, w, \bar{w})) | w \in W_{legal}(s) \Big\} \forall s \in S_{training} \tag{7.5}$$

Test data consists of unlabelled feature vectors:

$$D_{testing} = \{ f(s) | s \in S_{testing} \} \tag{7.6}$$

The quality of predicted workgroup sizes is evaluated using the following metrics:

- accuracy (binary) — whether the predicted workgroup size is the true oracle, $p(f(s)) = \Omega(s)$.
- performance (real) — the relative performance of the predicted workgroup size relative to the oracle, $0 \leq r(p(f(s)), \Omega(s)) \leq 1$.
- speedups (real) — the relative performance of the predicted workgroup size relative to the baseline workgroup size $w_{(4\times4)}$, and human expert workgroup size $w_{(32\times4)}$ (where applicable).
- time (real) — the round trip time required to make the prediction, as measured by the OmniTune client. This includes classification time and inter-process communication overheads between the client and server.

Unlike with classifiers, the process of selecting workgroup sizes using regressors is resistant to refused parameters, so no fallback strategies are required, and the *validity* and *refused* metrics are not used.

7.4.3 Results and Analysis

The purpose of this evaluation is to test the effectiveness of machine learning-enabled autotuning for predicting workgroup sizes of SkelCL stencils codes. First, we consider the prediction performance of classifiers.

With the exception of the ZeroR, which predicts *only* the baseline workgroup size $w_{(4\times4)}$, the classifiers achieve good speedups over the baseline. Average classification speedups across all validation sets range between 4.61× and 5.05×. Figures 7.11 and 7.12 show a summary of results using 10-fold cross-validation and cross-device validation, respectively. The highest average speedup is achieved by SMO, and the lowest by Naive Bayes. The difference between average speedups is not significant between the types of classifier, with the exception of SimpleLogistic, which performs poorly when trained with synthetic benchmarks and tested against real-world programs. This suggests the model over-fitting to features of the synthetic benchmarks which are not shared by the real-world tests.

By isolating the test cases where classifiers predicted an illegal or refused parameter, we can directly compare the relative effectiveness of each fallback handler. The fallback handler with the best average case performance is NearestNeighbour, with an average speedup across all classifiers and validation sets of 5.26×. The speedup of Random fallback handler is 3.69×, and 1.0× for Baseline. Figure 7.14 plots the speedups of each fallback handler for all of these isolated test cases. Interestingly, both the lowest and highest speedups are achieved by the Random fallback handler, since it essentially performs a random exploration of the optimisation space. However, the NearestNeighbour fallback handler provides consistently greater speedups for the majority of test cases, indicating that it successfully exploits the structure of the optimisation spaces.

Figures 7.13a and 7.13b show a summary of results for classification using regressors to predict program runtimes and speedups, respectively. Of the two regression techniques, predicting the *speedup* of workgroup sizes is much more successful than predicting the *runtime*. This is most likely caused by the inherent difficulty in predicting the runtime of arbitrary programs, where dynamic factors such as flow control and loop bounds are not captured by the kernel features used in OmniTune, which instead use simple static static instruction counts and densities. The average speedup achieved by predicting runtimes is 4.14×. For predicting speedups, the average is 5.57×. Tables 7.5, 7.6, and 7.7 show mean performances and speedups for: J48 classifier using the NearestNeighour fallback strategy, classification using runtime regression, and classification using speedup regression, respectively.

If we eliminate the 2.6% of test cases for which the workgroup size of $w_{(32\times4)}$ is illegal, we can compare the performance of OmniTune directly against the hu-

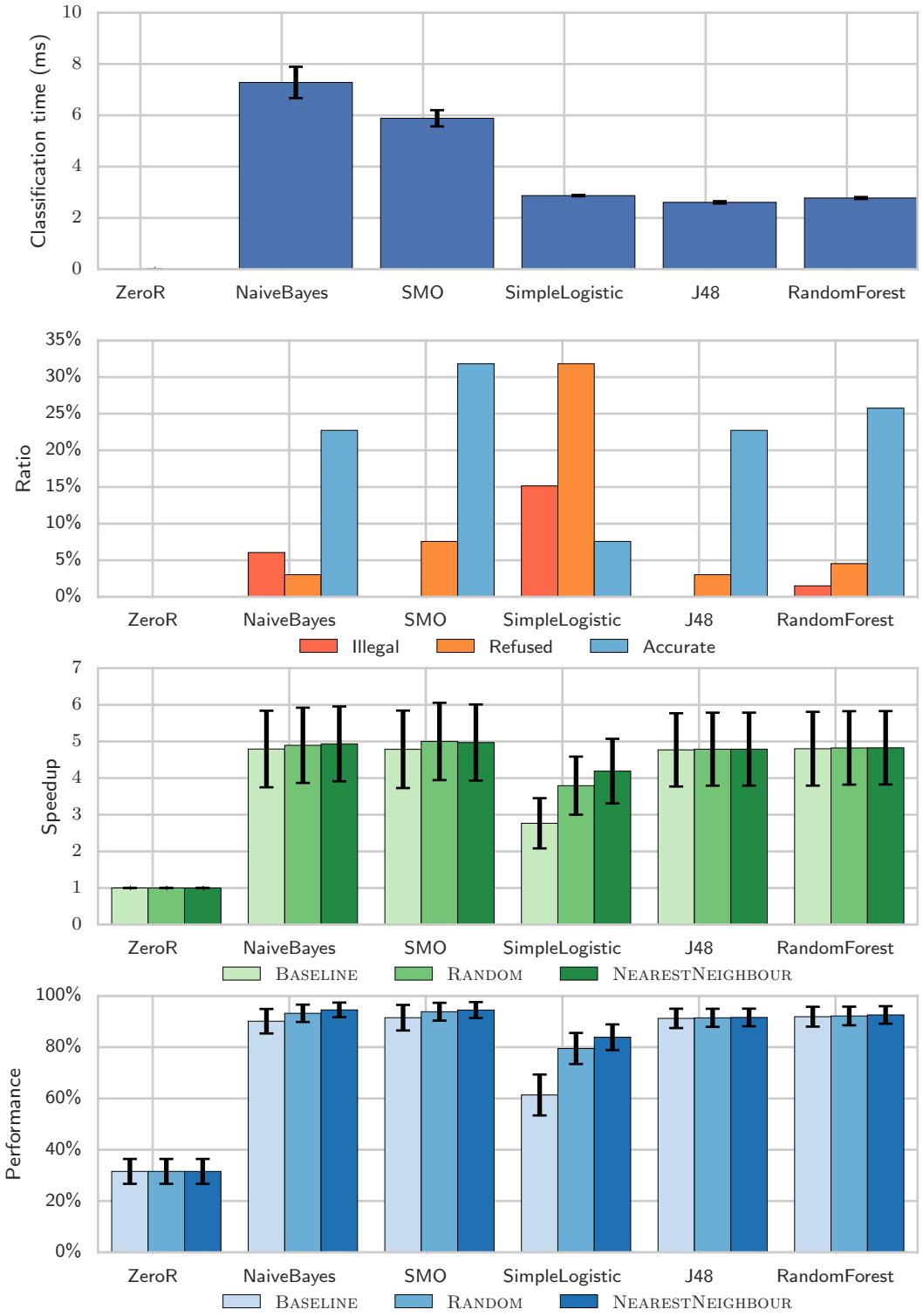


Figure 7.11: Classification results for synthetic benchmarks. Each classifier is trained on data from synthetic stencils, and tested for prediction quality using data from 6 real world benchmarks.

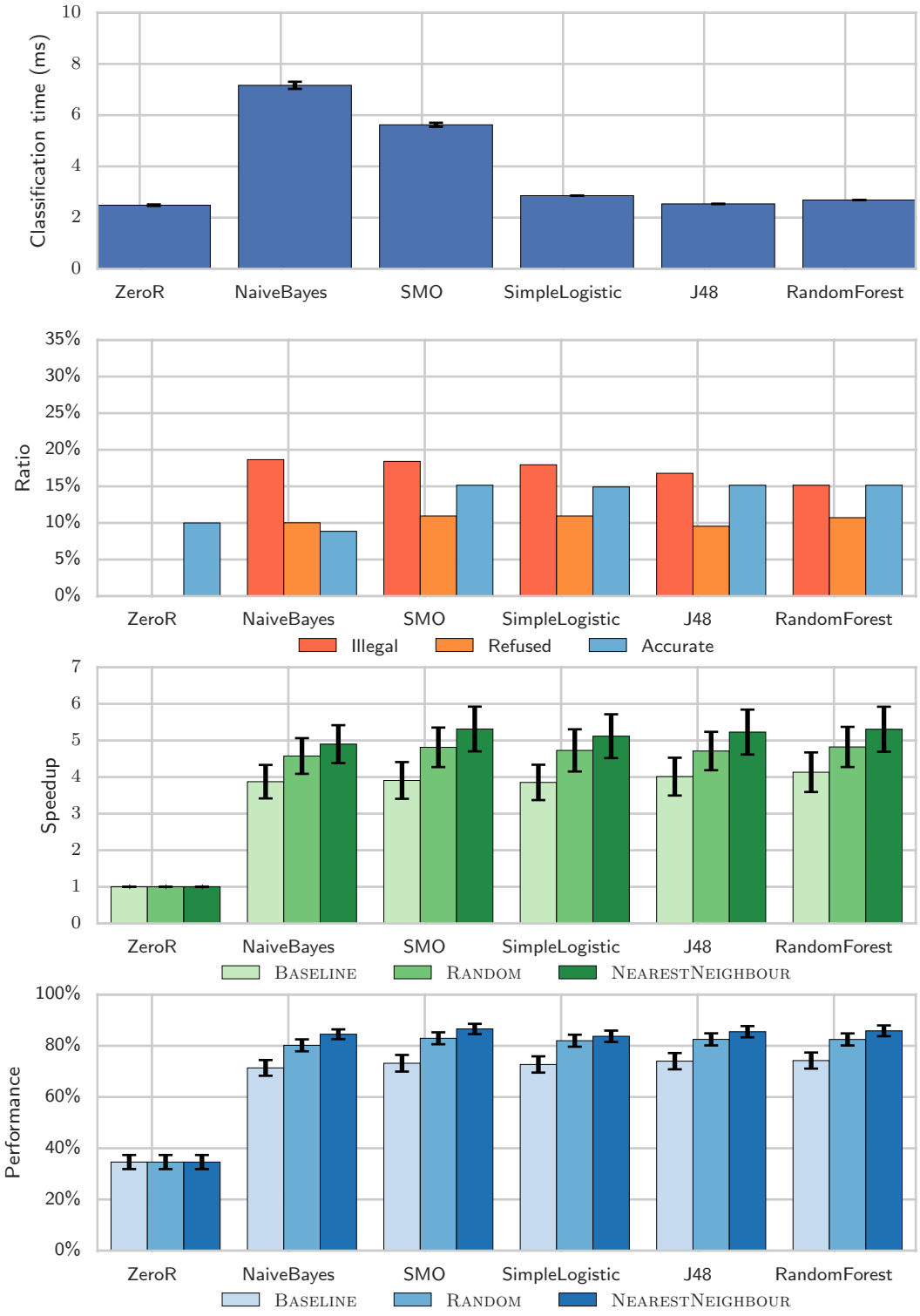


Figure 7.12: Classification results of cross-device evaluation. Each classifier is trained using data from $n-1$ devices, and tested for prediction quality using data for the n^{th} device.

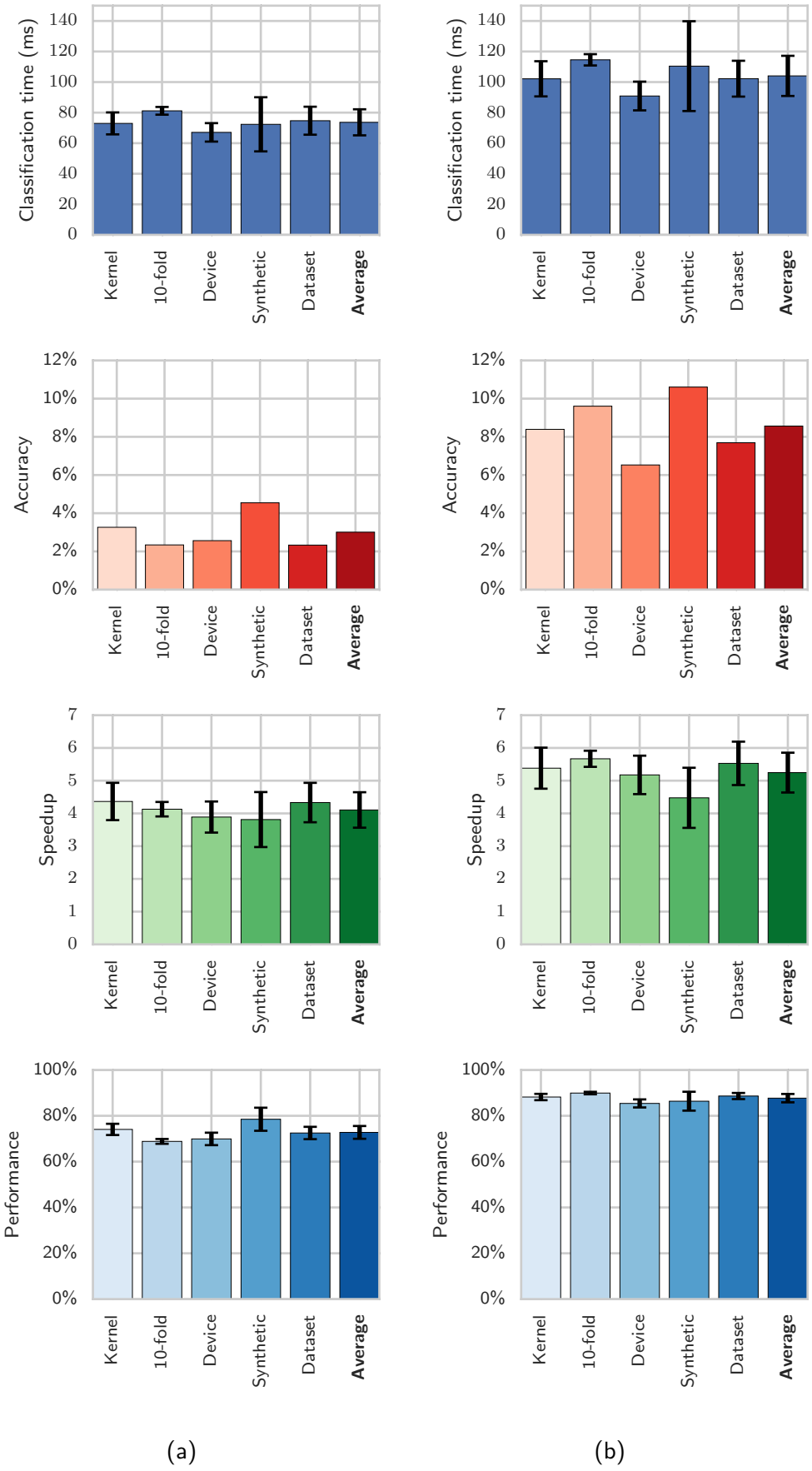


Figure 7.13: Evaluating the effectiveness of classification using regressors, by predicting: (a) the workgroup size with the minimal runtime, and (b) the workgroup size with the greatest speedup over a baseline.

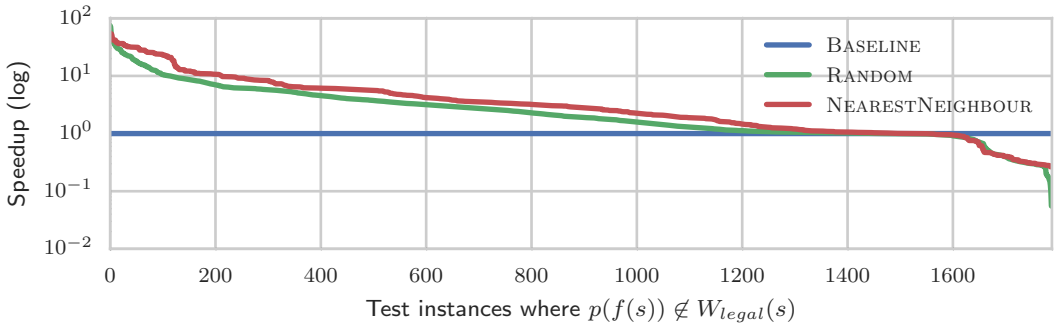


Figure 7.14: Comparison of fallback handlers, showing the speedup over baseline parameter for all test cases where a classifier predicted an illegal workgroup size.

Job	Performance	Speedup	Human Expert
10-fold	92%	5.65×	1.26×
Synthetic	92%	4.79×	1.13×
Device	85%	5.23×	1.17×
Kernel	89%	5.43×	1.21×
Dataset	91%	5.63×	1.25×
Average	**90%**	**5.45×**	**1.22×**

Table 7.5: Validation results for J48 and NearestNeighbour classification.

man expert chosen workgroup size. Figure 7.15 compares the speedups of all such validation instances over the human expert parameter, for each autotuning technique. The speedup distributions show consistent classification results for the five classification techniques, with the speedup at the lower quartile for all classifiers being $\geq 1.0\times$. The IQR for all classifiers is < 0.5, but there are outliers with speedups both well below $1.0\times$ and well above $2.0\times$. In contrast, the speedups achieved using runtime regression have a lower range, but also a lower median and a larger IQR. Clearly, runtime regression is the least effective of the evaluated autotuning techniques. Speedup regression is more successful, with the highest median speedup of all the techniques. However, it also has a large IQR and the lower quartile has a speedup value well below 1, meaning that for more than 25% of test instances, the workgroup size selected did not perform as well as the human expert selected workgroup size.

The prediction costs using regression are significantly greater than using classifiers. This is because, while a classifier makes a single prediction, the number of predictions required of a regressor grows with the size of $W_{\max}(s)$, since classifi-

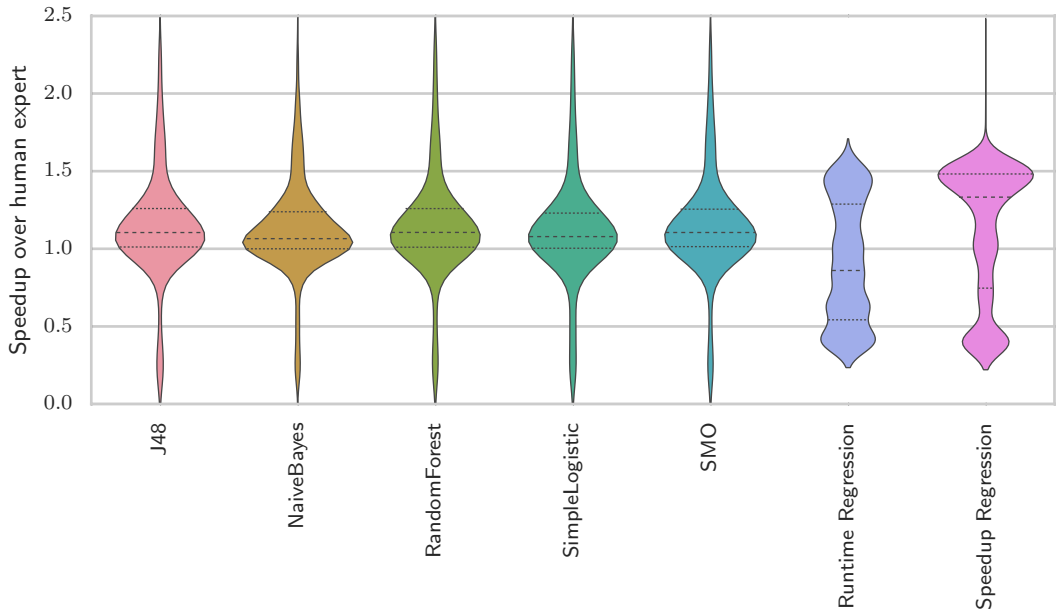


Figure 7.15: Distributions of speedups over *human expert*, ignoring cases where human expert prediction is invalid. Classifiers are using NEARESTNEIGHBOUR fallback handlers. The speedup axis is fixed to the range 0–2.5 to highlight the IQRs, which results in some outliers > 2.5 being clipped.

Job	Performance	Speedup	Human Expert
10-fold	68%	4.13×	0.88×
Synthetic	78%	3.81×	1.06×
Device	69%	3.89×	0.97×
Kernel	74%	4.36×	1.04×
Dataset	72%	4.33×	0.98×
Average	**70%**	**4.14×**	**0.92×**

Table 7.6: Validation results for runtime regression.

Job	Performance	Speedup	Human Expert
10-fold	89%	5.67×	1.10×
Synthetic	86%	4.48×	1.19×
Device	85%	5.18×	1.15×
Kernel	88%	5.38×	1.15×
Dataset	88%	5.53×	1.13×
Average	**89%**	**5.57×**	**1.12×**

Table 7.7: Validation results for speedup regression.

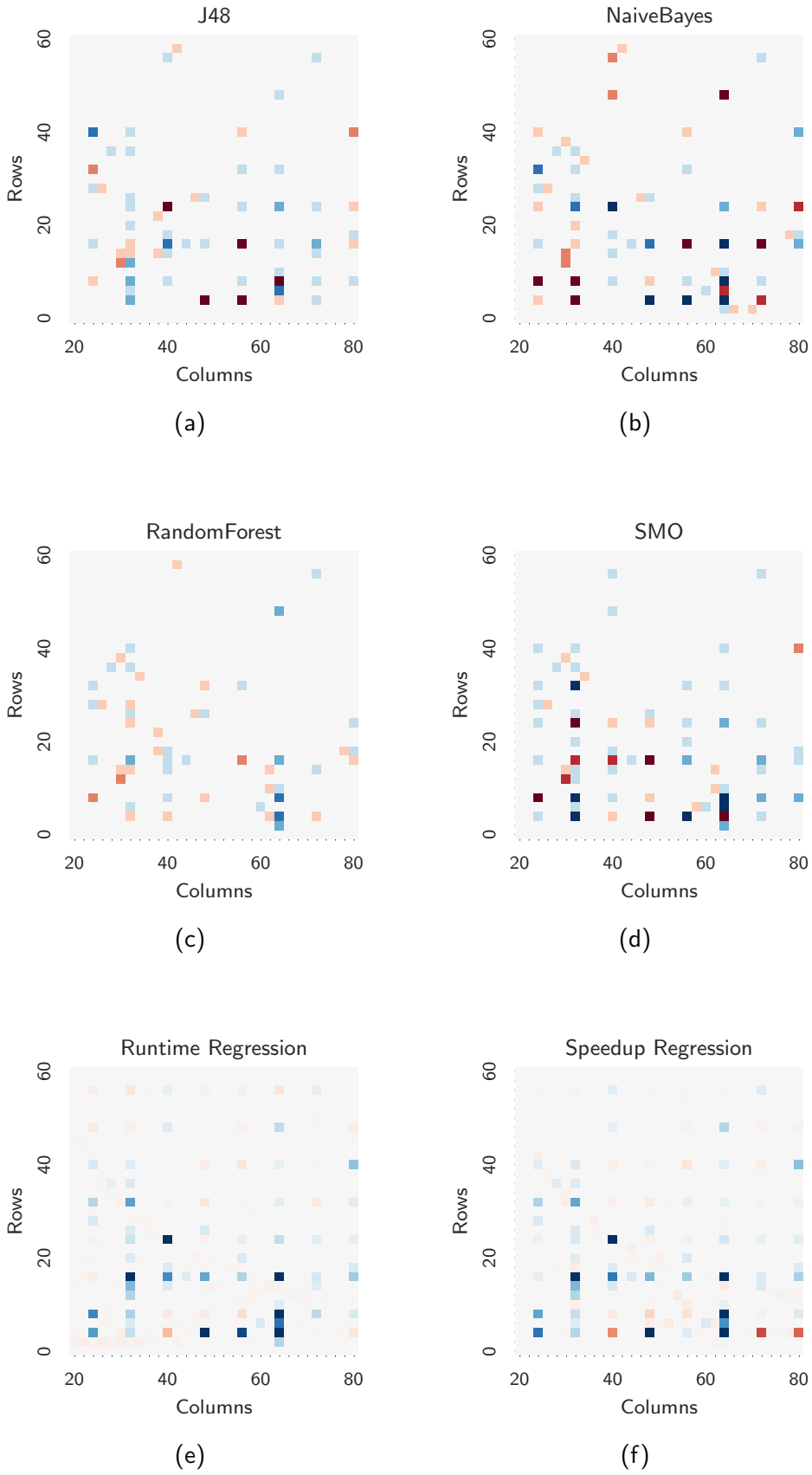


Figure 7.16: Heatmaps of classification errors for 10-fold cross-validation, showing a subset of the optimisation space. The shading in each cells indicates if it is predicted less frequently (blue), ore more frequently (red) than it is optimal. Colour gradients are normalised across plots.

cation with regression requires making predictions for all $w \in \{w | w < W_{\max}(s)\}$. The fastest classifier is J48, due to the it's simplicity (it can be implemented as a sequence of nested `if/else` statements).

Figure 7.16 visualises the classification errors of each of the autotuning techniques. It shows that while the performance of all of the classifiers is comparable, the distribution of predictions is not. Only the NaiveBayes and RandomForest classifiers predicted the human expert selected workgroup size of $w_{(32\times4)}$ as frequently, or more frequently, than it was optimal. The two regression techniques were the least accurate of all of the autotuning techniques.

7.4.4 Summary

From an evaluation of 17 different autotuning techniques using 5 different types of validation sets, the following conclusions about autotuning performance can be drawn:

- In the case of classifiers predicting illegal workgroup sizes, the best fallback strategy is to select the closest legal workgroup size.
- The performance of predicted workgroup sizes for unseen devices is within 8% of the performance for known devices.
- Predicting the *runtime* of stencils is the least effective of the evaluated autotuning techniques, achieving an average of only 68% of the available performance.
- Predicting the *speedup* of workgroup sizes provides the highest median speedup, but more frequently predicts a poorly performing workgroup size then the classifiers.
- Classification using regression costs an order of magnitude more time than using classifiers. The J48 classifier has the lowest overhead.

Chapter 8

Conclusions

As the trend towards higher core counts and increasing parallelism continues, the need for high level, accessible abstractions to manage such parallelism will continue to go. Autotuning proves a valuable aid for achieving these goals, providing the benefits of low level performance tuning while maintaining ease of use, without burdening developers with optimisation concerns. As the need for autotuned parallelism rises, the desire for collaborative techniques for sharing performance data must be met with systems capable of supporting this cross-platform learning.

In this thesis, I have presented my attempt at providing such a system, by designing a novel framework which has the benefits of fast, "always-on" autotuning, while being able to synchronise data with global repositories of knowledge which others may contribute to. The framework provides an interface for autotuning which is sufficiently generic to be easily re-purposed to target a range of optimisation parameters.

To demonstrate the utility of this framework, I implemented a frontend for predicting the workgroup size of OpenCL kernels for SkelCL stencil codes. This optimisation space is complex, non linear, and critical for the performance of stencil kernels, with up to a 207.72× slowdown if an improper value is picked. Selecting the correct workgroup size is difficult — requiring a knowledge of the kernel, dataset, and underlying architecture. The challenge is increased even more so by inconsistencies in the underlying system which cause some workgroup sizes to fail completely. Of the 269813 combinations of workgroup size, device, program, and dataset tested; only a *single* workgroup size was valid for all test cases, and achieved only 24% of the available performance. The value selected by human experts was invalid for 2.6% of test cases. Autotuning in this space

requires a system which is resilient these challenges, and several techniques were implemented to address them.

Runtime performance of autotuned stencil kernels is very promising, achieving an average 90% of the available performance with only a 3ms autotuning overhead. Even ignoring the cases for which the human expert selected workgroup size is invalid, this provides a 1.33× speedup, or a 5.57× speedup over the best performance that can be achieved using static tuning. Classification performance is comparable when predicting workgroup sizes for both unseen programs and unseen devices. I believe that the combination of performance improvements and the collaborative nature of OmniTune makes for a compelling case for the use of autotuning as a key component for enabling performant, high level parallel programming.

8.1 Critical Analysis

This section contains a critical analysis of the work presented in previous chapters.

OmniTune Framework

The purpose of the OmniTune framework is to provide a generic interface for runtime autotuning. This is demonstrated through the implementation of a SkelCL frontend; however, to truly evaluate the ease of use of this framework, it would have been preferable to implement one or more additional autotuning frontends, to target different optimisation spaces. This could expose any leakages in the abstractions between the SkelCL-specific and generic autotuning components.

Synthetic Benchmarks

The OmniTune SkelCL frontend provides a template substitution engine for generating synthetic stencil benchmarks. The implementation of this generator is rigidly tied to the SkelCL stencil format. It would be preferred if this template engine was made more flexible, to support generation of arbitrary test programs. Additionally, due to time constraints, I did not have the opportunity to explore how the number of synthetic benchmarks in machine learning test data sets affects classification performance.

One possible use of the synthetic stencil benchmark generator could be for

creating minimal test cases of refused OpenCL parameters so that bug reports could be filed with the relevant implementation vendor. However, this would have added a great level of complexity to the the generator, as it would have to isolate and remove the dependency on SkelCL to generate minimal programs, requiring significant implementation work.

Use of Machine Learning

The evaluation of OmniTune in this thesis uses multiple classifiers and regressors to predict workgroup sizes. The behaviour of these classifiers and regressors is provided by the Weka data mining suite. Many of these classifiers have parameters which affect their prediction behaviour. The quality of the evaluation could have been improved by exploring the effects that changing the values of these parameters has on the OmniTune classification performance. It would also have been informative to dedicate a portion of the evaluation to feature engineering, evaluating the information gain of each feature and exploring the effects of feature transformations on classification performance.

Evaluation Methodology

The evaluation compares autotuning performance against the best possible performance that can be achieved using static tuning, a simple heuristic to tune workgroup size on a per-device basis, and against the workgroup size chosen by human experts. It would have been beneficial to also include a comparison of the performance of these autotuned stencils against hand-crafted equivalent programs in pure OpenCL, without using the SkelCL framework. This would allow a direct comparison between the performance of stencil kernels using high level and low level abstractions, but could not be completed due to time constraints and difficulties in acquiring suitable comparison benchmarks and datasets.

8.2 Future Work

Future work can be divided into two categories: continued development of OmniTune, and extending the behaviour of the SkelCL autotuner.

The cost of offline training with OmniTune could be reduced by exploring the use of adaptive sampling plans, such as presented in [**Leather2009**]. This

could reduce the number of runtime samples required to distinguish good from bad optimisation parameter values.

Algorithm 4 proposes the behaviour of a hybrid approach to selecting the workgroup size of iterative SkelCL stencils. This approach attempts to exploit the advantages of all of the techniques presented in this thesis. First, runtime regression is used to predict the minimum runtime and a candidate workgroup size. If, after evaluating this workgroup size, the predicted runtime turned out to be inaccurate, then a prediction is made using speedup regression. Such a hybrid approach would enable online tuning through the continued acquisition of runtime and speedup performance, which would compliment the collaborative aspirations of OmniTune, and the existing server-remote infrastructure.

Other skeleton optimisation parameters could be autotuned by SkelCL, including higher level optimisations such as the selection of border region loading strategy, or selecting the optimal execution device(s) for multi-device systems. Optimisation parameters of additional skeletons could be autotuned, or the interaction of multiple related optimisation parameters could be explored. Power consumption could be used as an additional optimisation cotarget.

Algorithm 4 Selecting workgroup size using a combination of classifiers and regressors.

Require: kernel features k, hardware features h, dataset features d.

Ensure: workgroup size w

$r \leftarrow \min_{w \in W_{legal}(s)} f(k,h,d,w)$ ▷ Predict minimum runtime.
$w \leftarrow \arg\min_{w \in W_{legal}(s)} f(k,h,d,w)$ ▷ Workgroup size for r.
$t_r \leftarrow$ measure runtime of program with w
SUBMIT$(f(s), w, t_r)$ ▷ Submit observed runtime
if $t_r \approx r$ **then**
    return w ▷ Predicted runtime is accurate.
else
    $W \leftarrow \{w | w < W_{\max}(s)\}$
    converged $\leftarrow$ false
    $w_b \leftarrow$ baseline parameter
    $t_b \leftarrow$ measure runtime of runtime of program with w_b
    SUBMIT$(f(s), w_b, t_b)$ ▷ Submit observed runtime
    while not converged **do**
        $s \leftarrow \max_{w \in W} g(k,h,d,w)$ ▷ Predict best speedup.
        $w \leftarrow \arg\max_{w \in W} g(k,h,d,w)$ ▷ Workgroup size for s.
        $t \leftarrow$ measure runtime of program with s
        SUBMIT$(f(s), w, t)$ ▷ Submit observed runtime
        $s_w \leftarrow t_t / t$
        SUBMIT$(f(s), w, s_w)$ ▷ Submit observed speedup
        if $s_w \approx s$ **then**
            converged = true ▷ Predicted speedup is accurate.
        else
            $W = W - \{w\}$
        end if
    end while
    return w
end if