AUTOMATIC PERFORMANCE TUNING OF_STENCIL COMPUTATIONS ON GRAPHICS PROCESSING UNITS

by

Joseph D. Garvey

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science Graduate Department of Electrical and Computer Engineering University of Toronto

© Copyright 2015 by Joseph D. Garvey
Abstract

Automatic Performance Tuning of Stencil Computations on Graphics Processing Units

Joseph D. Garvey
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2015

The focus of this work is the automatic performance tuning of stencil computations on Graphics Processing Units (GPUs). A strategy is presented that uses machine learning to determine the best way to use the GPU memory followed by a heuristic that divides the remaining optimizations into groups and exhaustively explores one group at a time. The strategy is evaluated using 104 synthetically generated OpenCL stencil kernels on an Nvidia GTX Titan GPU. The strategy is assessed both in terms of the number of configurations explored during auto-tuning and the quality of the best configuration obtained. Two alternative heuristics that use different groupings of the optimizations are explored. Relative to a random sampling of the space and an expert search, the strategy achieves a reduction in the number of configurations explored of up to 71% and 84% respectively while finding configurations that perform 12% and 6% better respectively.
Acknowledgements

I would like to thank my supervisor Tarek Abdelrahman for his wise and invaluable guidance throughout my Master’s degree and the development of this thesis. I am also grateful to him for having the patience to let me take the time to get this right.

I would also like to thank David Han. Our many conversations were essential in making this work a reality. Whenever I was stuck he helped me get going again and whenever I thought I had it all figured out he found a way to trip me up.

Finally, I owe a great debt of gratitude to my support network outside of the lab. My partner Kimberlea Patterson and my parents Robin Garvey and Dan Garvey have always believed in me and that confidence was what allowed me to push through the most difficult times.
# Contents

1 Introduction 1
   1.1 Research Overview ........................................... 2
   1.2 Contributions .................................................. 3
   1.3 Organization ................................................... 4

2 Background 5
   2.1 GPU Architecture ............................................... 5
       2.1.1 GPU Memory Hierarchy ...................................... 6
   2.2 OpenCL Programming Model .................................... 9
   2.3 Auto-tuning ..................................................... 10
   2.4 Supervised Machine Learning .................................. 11
       2.4.1 Decision Trees ............................................. 12
       2.4.2 Ensemble Methods ......................................... 13
       2.4.3 Cross-Validation ........................................... 13

3 Stencil Computations 15

4 Optimizations 18
   4.1 Work-Group Size ............................................... 18
   4.2 Block Merging ................................................... 19
   4.3 Cyclic Merging .................................................. 20
   4.4 Local Memory Caching ......................................... 21
   4.5 Vectorization ................................................... 21
   4.6 Image/Texture Memory ......................................... 24
   4.7 Optimization Space ............................................ 24
   4.8 Optimization Interactions .................................... 25

5 Interestingness 27
   5.1 Random Sampling ............................................... 27
5.2 Methodology ......................................................... 28
5.3 High-Performing Configurations Are Few ......................... 30
5.4 High-Performing Configurations Are Difficult to Find .......... 30
5.5 The Optimizations Interact ....................................... 31
5.6 Good Configurations Are Not the Same Across Kernels ......... 31
5.7 Summary ............................................................ 32

6 Auto-Tuning Strategy ................................................. 35
6.1 Predicting Data Loading Technique ............................... 35
6.2 Grouping and Search Heuristics ................................. 37
   6.2.1 Group-by-Dimension ......................................... 38
   6.2.2 Group-by-Optimization ...................................... 39
6.3 Overall Strategy .................................................. 40

7 Implementation ....................................................... 41
7.1 The Stencil Kernels ................................................. 41
7.2 Genesis Implementation ............................................ 42
   7.2.1 Limitations ................................................... 46
7.3 The Auto-Tuner .................................................... 47
7.4 Heuristic Specification Files ..................................... 48

8 Evaluation ............................................................. 51
8.1 Reference Approaches .............................................. 51
8.2 Evaluation Platform and Metrics .................................. 52
8.3 Experiments ......................................................... 52
8.4 Determining a Good Value for n ................................ 53
8.5 Auto-tuning Strategy Performance ................................ 54
8.6 Restricting the Optimization Space ............................... 56
8.7 Variable Work-Group Size ....................................... 57
8.8 Array Size and Shape ............................................. 59
8.9 Summary ............................................................ 61

9 Related Work .......................................................... 62

10 Conclusions ........................................................... 65
  10.1 Future Work ...................................................... 66
A Optimization Space Size

A.1 Full Space ......................................................... 68
A.2 Reduced Space ..................................................... 70

Bibliography ............................................................. 71
Chapter 1

Introduction

In recent years there has been a growing interest in the use of Graphics Processing Units (GPUs) to accelerate non-graphics applications, a process known as general-purpose GPU (GPGPU) computing [34]. As GPUs are ubiquitous in modern systems, their use is important in getting maximum performance out of these systems. Relative to CPUs, GPUs provide significantly more parallelism and arithmetic functional units. This makes them ideal for accelerating highly-parallel and computationally intensive applications.

Stencil computations appear in many domains such as image processing [35], partial differential equation solvers [23] and cellular automata [12]. They are often computationally intensive and have abundant parallelism. Thus, they are excellent candidates for acceleration on GPUs. This thesis focuses on accelerating stencil computations on GPUs.

Achieving high stencil code performance on GPUs is often challenging [37, 38]. Programmers must apply optimizations to their code in order to exploit the underlying GPU architecture [31]. The impact of each optimization is often difficult to assess, particularly when combined with other optimizations. As a result, analytically determining the ideal combination of optimizations to apply can be impossible, requiring programmers to run their code for each configuration of interest.

In essence, programmers must explore a large space of optimization configurations, i.e., combinations of optimizations and the way in which they are applied, to find good performing ones. A programmer might need to run a program with hundreds or even thousands of different optimization configurations. This process can be long both in terms of compile time and runtime. Indeed, previous work has shown that this can take months of compute time [11].

There has been considerable interest in performance auto-tuning, i.e., automatically exploring the space of possible configurations in efficient ways to determine a good com-
bination of optimizations to apply. A common approach is to use expert knowledge of the stencil computation and the underlying GPU architecture to limit the optimization search space, then to exhaustively search through this limited space; a strategy that we refer to as *expert search*. However, this approach requires new expert knowledge for every stencil computation and for every new GPU architecture. In addition, it can still require exploring a large number of configurations. Up until now, no good solution to this problem has been advanced. This thesis will address this problem by proposing a strategy for auto-tuning of stencil programs on GPUs that does not require expert knowledge of a given stencil program, finds better performing configurations than expert search, and does so in a fraction of the time.

1.1 Research Overview

This work presents an auto-tuning strategy for searching through the space of possible optimization configurations for good performing ones. The strategy uses machine learning and heuristics to prune the large optimization space, thereby allowing us to explore fewer configurations than expert search approaches while obtaining as good or better performing configurations.

Our focus is on stencil computations. That is, programs that compute an output array from an input array by computing each element of the output as a function of some elements of the input at fixed offsets relative to the address of the output element in question. We focus on stencils with distinct input and output arrays, i.e. those that do not make in-place updates to array data. Our OpenCL implementation of these stencils maps the loops that iterate over the arrays to the GPU threads and leaves any outer time loop on the host. We consider stencils with a variety of different sizes and shapes within these constraints.

We use a set of optimizations that are commonly applied to stencil computations. These consist of the values of generic OpenCL parameters (the work-group size in various dimensions) and thread merging factors (both block and cyclic), usage of various memory types (local and image), and vectorization width. Overall this results in an optimization space consisting of over 50 million configurations.

We first demonstrate that this optimization space is interesting. We do this by showing that most of the optimization configurations perform poorly and that the best values for each individual optimization vary across different stencil kernels. We further make the case for the space’s interestingness by showing that two naive strategies, tuning the optimizations independently and tuning for a given kernel and then using the result on
other kernels, are insufficient. Overall we use these results to make the case that auto-
tuning strategies such as the one proposed in this work are necessary to achieve good
performance.

Our approach tackles the large optimization configuration space in two ways. First, it
partitions the space based on memory optimization and uses a random forest [5] machine
learning model to predict the best partition (i.e. the optimal memory type) based on
static program features. Second, it groups the remaining optimizations so as to minimize
the dependencies among optimizations in different groups and auto-tunes the groups
independently. This is done with two alternative heuristics. The first groups the opti-
mizations along the dimensions of the grid in which the GPU threads are organized. The
second puts each optimization, applied to all dimensions of the grid, in a group by itself.

We evaluate the strategy using 104 synthetically generated OpenCL stencil programs
with a wide range of stencil patterns (dense, star, no-corner, diamond and thumbtack)
and radii (0 to 5) created using the program generation language Genesis [6]. We develop
infrastructure for running generic heuristics in order to perform this evaluation. Using
this infrastructure we run the strategy on the stencils using an Nvidia GTX Titan GPU.
In comparison to an expert search approach, the strategy with the better of the two
heuristics (the one based on dimensions) explores 84% fewer configurations while finding a
configuration that is on average 6% better. Compared to a random sampling of the space,
the strategy with the same heuristic explores 71% fewer configurations while finding one
that is 12% better on average.

1.2 Contributions

This thesis makes the following contributions:

- Proposes an auto-tuning strategy for stencil computations on GPUs that uses a
combination of machine learning and heuristics to find better performing optimiza-
tion configurations than expert search in less time.

- Demonstrates the interestingness of the optimization space on the stencil programs
examined, thereby making the case for auto-tuning.

- Develops infrastructure to execute generic auto-tuning heuristics so as to evaluate
the strategy presented.
1.3 Organization

The rest of this thesis is structured as follows. Chapter 2 provides background material on GPU hardware, OpenCL programming, auto-tuning, and machine learning. Chapter 3 defines stencil computations and the form of the stencil programs examined in this work. Chapter 4 describes the optimizations that are tuned and the size of the optimization space itself. Chapter 5 demonstrates that the optimization configuration space is interesting and makes the case that a sophisticated auto-tuning approach is necessary. Chapter 6 details the two main components of the auto-tuning strategy: the machine learning models and the heuristics. Chapter 7 discusses the design and generation of the synthetic stencils as well as the auto-tuning infrastructure. Chapter 8 evaluates the performance of the strategy as well as its sensitivity to parameters of the evaluation. Chapter 9 outlines related work and Chapter 10 concludes.
Chapter 2

Background

2.1 GPU Architecture

GPUs are highly parallel platforms consisting of many simple cores. There are a few major GPU vendors, each with their own GPU architectures. In this work an Nvidia GeForce GTX TITAN is used and as such we focus on its architecture, known as Kepler GK110.

Figure 2.1 shows a high-level overview of the Kepler GK110 architecture. In this architecture the cores are clustered into groups known as Streaming Multiprocessors (SMXs) [32]. Kepler GPUs can have up to 15 SMXs with the TITAN possessing 14 [30]. The number of functioning SMXs on a manufactured Kepler chip is one of the primary determinants in binning, with Nvidia selling parts possessing as few as 12 [28] working SMXs and as many as 15 [29].

Figure 2.2 shows a more detailed view of a Kepler SMX. Each Kepler SMX consists of 192 cores [32, 33]. SMXs do not possess sufficient instruction issuing ability to keep all of their cores busy executing different instructions and so, to keep the cores fully utilized, threads are launched in groups, known as warps, that execute in lock-step. Thus multiple cores within an SMX execute the same instruction simultaneously on possibly different data [32]. Specifically, each SMX in Kepler GK110 can dispatch four warps per cycle with two instructions being issued for each warp (i.e. 8 different instructions total per cycle). In Kepler GK110 each warp consists of 32 threads and each SMX can support a maximum of 64 warps simultaneously (i.e. a maximum of 2048 threads) [32]. To achieve good performance on this platform, as with most GPU platforms, it is essential that applications expose sufficient parallelism to allow a high utilization of the platform’s many cores.
2.1.1 GPU Memory Hierarchy

GPUs have a multi-tiered memory hierarchy whose characteristics have important performance implications. The various levels of this memory hierarchy are discussed below. Figure 2.3 shows this hierarchy for the Kepler GK110 architecture on which the GTX TITAN is based [32].

Global Memory

Global memory is the large, off-chip, DRAM memory of the GPU. The GTX TITAN possesses 6 GB of global memory [33]. This memory is very slow compared to other memories in the hierarchy. One of the most important factors affecting performance is how data is loaded from global memory. If multiple consecutive work-items all access a contiguous, aligned chunk of memory of a particular size these accesses are coalesced into a single memory transaction [31]. Coalesced accesses result in significantly fewer memory transactions, thereby reducing memory access time. Depending on the memory
access patterns of the application, some of the cost of global memory can be reduced by on-chip, hardware-managed L1 and L2 caches [32].

**Shared Memory**

Shared memory is a software-managed cache present at the SMX level that is much faster than global memory. Since each SMX has its own shared memory cache, it can be used to allow the cores of an SMX to work on data cooperatively. However, the shared memory of one SMX cannot be accessed by programs running on another SMX. In the Kepler GK110 architecture, the L1 and shared memory share a 64 KB cache. CUDA provides an API that makes it possible to adjust how data is allocated between the two to follow a 16 KB/48 KB split (in either cache’s favour) or a 32 KB/32 KB split. OpenCL, however,
provides no such API; the L1 cache is always fixed to 16 KB while shared memory is fixed to 48 KB [32].

**Texture Memory**

Texture memory is a hardware-managed cache with multidimensional locality that is used to store certain data types [31]. Data that makes use of this cache is initially stored in global memory but is cached in the texture cache. This cache can exploit locality in non-innermost dimensions of array data. On the GTX Titan, this cache can be used to store any read-only data and consists of 48 KBs per SMX [32].

**Registers**

The fastest level of the memory hierarchy is the registers themselves. On the GTX Titan, each thread can access up to 255 registers [32] although the total register file, which is shared across the SMX, consists of only 65536 registers [32], meaning that it is not possible for all of the 192 cores to use the maximum number of registers at the same time. Despite the fact that the registers are physically present at the SMX level, they are thread-specific and usually cannot be shared across threads.
2.2 OpenCL Programming Model

The programs used in this work are all written using Open Computing Language (OpenCL) [40], an industry standard language for programming heterogeneous parallel systems. Its paradigm consists of a host from which programs, known as kernels, are launched and one or more devices, in this case a GPU, on which those kernels execute. Many instances of the kernel code are executed concurrently, each by a work-item, i.e., a thread. Work-items are grouped in a multidimensional grid and are identified by their indices, known as global IDs, in each dimension of this grid. Generally, to properly exploit the GPU architecture, OpenCL programs are written in such a way that all work-items execute the same instructions on different data. The global IDs are often used to determine on which data each work-item should work.

OpenCL also has the notion of a work-group, a multidimensional collection of work-items within the grid that execute on the device at the same time. The IDs of a work-item in each dimension within its work-group are known as its local IDs. Figure 2.4 shows a graphical representation of this two-tiered grid decomposition. Work-groups provide two major benefits. First, work-items in the same work-group are present simultaneously on the device, making it possible to synchronize across them. Most hardware implementations ensure that work-groups are executed together at some level in the hardware. In the case of GPUs, work-items from the same work-group are all mapped to the same SMX. Second, work-items in the same work-group share a software-managed cache known as local memory which in the case of GPUs is realized using shared memory.

As a result of the fact that all work-items in a work-group are mapped to one SMX, various resource requirements of the work-items can impose limitations on the maximum work-group size. For example, as mentioned earlier, each SMX has a limited number of registers as well as a limited amount of shared memory. Thus the register usage or local memory usage of each work-item could be the limiting factor that determines the maximum work-group size.

The sizes of the two grids mentioned, global and local, are provided by the host program at kernel launch time and together constitute the launch configuration of the kernel. OpenCL separates the compilation of a kernel from the launching of that kernel and as a result the total number of work-items that will be launched and the size of each work-group are not known at kernel compile time.

OpenCL defines a number of image data types that can be used to store array data. On GPUs like the GTX Titan, these images can be cached in the texture cache.
Finally, OpenCL also supports vector types as well as arithmetic on those vector types. In the context of GPUs these vector operations can be mapped to specialized vector hardware or distributed amongst cores. While one would expect that these vector operations could improve arithmetic performance, what may not be as obvious is that even reads and writes to global memory can be accelerated by acting on vectorized data [42].

2.3 Auto-tuning

The problem of finding a combination of optimizations that perform well in a large optimization space, although exacerbated on GPUs, is an issue on CPUs as well. Auto-tuning, defined here as running many configurations in order to find one that performs well, has been used to deal with this problem [8, 9, 21, 41]. In fact, the increased use of auto-tuners was one of the principle recommendations of the landmark technical report:
"The Landscape of Parallel Computing Research: A View from Berkeley," [2] which has guided the general direction of much of the research in parallel computing since its publication.

If the optimization space is reasonably small, as is often the case when dealing with a single or a small number of optimizations, exhaustive tuning of the space can potentially allow the best configuration to be discovered. However, once the optimization space reaches a certain size this becomes impractical. A common approach for dealing with this is to use expert knowledge of the application and the optimizations to restrict the optimization space and to exhaustively search through this subspace. This approach, which will be referred to as expert search, is quite popular for stencil programs on both CPUs and GPUs [9, 20, 25]. Unfortunately, determining the optimal subset of the optimization space to search requires knowledge of the application, the optimizations being tuned, and the hardware platform being targeted, a set of knowledge that any one person is unlikely to possess. Furthermore, even with expert knowledge, the optimal combination of optimizations can be difficult to predict, meaning that it is possible to miss out on good configurations even when using expert knowledge to restrict the search space.

2.4 Supervised Machine Learning

Supervised machine learning is concerned with solving the following general problem: given a training set of input values with known output values that are related by some unknown underlying function, determine a function (the machine learning model) that approximates the underlying function [36]. The performance of the model is then evaluated on a test set whose elements were not part of the training set. If the model can perform well on the test set, one would expect it to perform well on any future inputs for which the output values are not known. A model that has this property is said to generalize well. Conversely, if a model does well only on the training set and does not generalize it is said to have overfit the training set.

There are a number of different machine learning techniques currently in use to create machine learning models to solve this problem. They differ in the particular space of hypothesis functions from which they choose their model and thus the degree of complexity that they can express. As a consequence of this, techniques that explore a more complex hypothesis space generally require more time to do so and usually require a larger training set in order to create a model that generalizes well.
2.4.1 Decision Trees

The machine learning model that will be used in this work is the decision tree. A decision tree maps its inputs to an output via a sequence of questions (the nodes of the tree) that are evaluated on the input data. The answer to each question determines the next question to ask, and eventually leads to a final decision (the leaves of the tree). The questions usually take the form of inequalities on the input data. An example tree of this type is shown in Figure 2.5. Effectively, the decision tree divides the entire input space into regions corresponding to different output values. Each of these regions corresponds to a leaf of the tree [36]. One can limit the complexity of a decision tree by limiting its maximum depth, i.e. the maximum number of nodes before reaching a leaf. If a function can be expressed accurately with a shallow tree then it is well suited to representation as a decision tree. Conversely this model does poorly on functions that require a very deep tree to express [36], i.e. those whose input space requires many partitions to separate by output value.

![Decision Tree Diagram]

Figure 2.5: An example decision tree for predicting optimal memory type from stencil features
Determining the optimal decision tree for a given training set can take prohibitively long for large training sets. However, good performance can be achieved with a greedy algorithm which, at each stage, constructs the next node by choosing the dimension of the input that causes the maximum change in the output [36]. One would then continue to proceed in this way until either all of the training data was separated or the maximum depth was reached.

### 2.4.2 Ensemble Methods

Ensemble methods are a class of machine learning techniques that combine the predictions of many simple, weak models in order to achieve a single, high-performing model. For example, in such a method each model might get a single vote about the final output with the majority chosen as the prediction of the ensemble. The main advantage of this approach is that it can improve accuracy without the risk of overfitting that would be incurred by using a single, more complex model. As each individual model is simple, it is unlikely to overfit the training set. There are a variety of different ways in which the constituent models in an ensemble can be trained, such as random sampling of the training set (bagging) [4] or more intelligent selection so as to ensure that new models focus more on those elements of the training set on which other models perform poorly (boosting) [10].

The technique that will be used in this work is a variant of bagging specific to ensembles of decision trees, known as random forests [5]. This technique creates multiple decision trees, each from a randomly selected subset of the elements in the training set. Rather than constructing each tree in the greedy method described earlier, each node is constructed using a randomly chosen input or a combination of randomly chosen inputs [5]. This approach minimizes the correlation between the various decision trees in the forest, thereby improving generalization. When encountering a new input, that input is evaluated using all of the trees in the forest and each tree then casts a vote for the output value that it predicts. The value with the most votes is the output prediction of the forest.

### 2.4.3 Cross-Validation

As was mentioned previously, evaluating the performance of a machine learning model involves testing the model’s performance on a test set of data that was not used in training. Unfortunately, holding out some data from training can result in a lower quality model, particularly when only a small amount of data is available to begin with. A
technique for training and testing models known as k-fold cross-validation was developed to counter this problem [36]. When using this technique, multiple models are trained and tested on different splits of the total data set into training and testing subsets such that no data point is ever used to test a model which it was used to train. Specifically, in k-fold cross-validation, the total data set is split into k subsets of equal size. Then k models are constructed, each of which leaves out one of the k subsets and is trained on the other k-1. Then each of the k models is tested on the subset of the data that was left out of its training. The final performance is the average performance across the k models.

The choice of the number of folds has important implications. If k is small, then the partitions themselves will be large, and thus a large amount of data will be held out during the training of any given model, potentially weakening it. Conversely, if k is too large, the models will benefit from larger training sets but the training process itself can take too long as more models will have to be trained, each of them on more data. Generally, the smaller the original data set is, the larger the choice of k should be in order to provide the model with sufficient training inputs. In the extreme, k can be set to the number of elements in the data set; this is known as leave-one-out cross-validation as each training set is constructed by removing a single element from the data set.
A stencil computation produces one or more output arrays as a function of one or more input arrays in such a way that each output element is a function of some input elements at fixed offsets relative to the index of the output element. This fixed set of offsets defines the stencil. Figure 3.1 shows sequential C code for the general form of a stencil computation with one three-dimensional input array and one three-dimensional output array. This code skeleton subsumes all 3D stencil computations regardless of stencil size or shape. The loops that iterate over \( x \), \( y \), and \( z \) are known as the spatial loops and they sweep through the elements of the output array, \texttt{out}. The upper bounds of these loops, \texttt{X\_SIZE}, \texttt{Y\_SIZE}, \texttt{Z\_SIZE} are constants based on the size of the output array. In each inner loop iteration one element of the output array is computed as a function, \( f \), of the stencil defined by the set of constants \( C_{ij} \), the \( i \)th offset in the \( j \)th dimension. The total number of such constant sets, \( N + 1 \), is referred to as the size of the stencil and corresponds to the number of input array elements required to compute each output element. The outer \texttt{t} loop repeats the whole process, with the roles of the arrays reversed, and is known as the time loop. The number of iterations of this loop, \texttt{T\_MAX}, depends on the particular application. It is shown as a constant here, although for some stencil applications a convergence check is performed and iteration ends once convergence has been achieved. The values of the constants \( C_{ij} \), the stencil size, the choice of \( f \), and the value of \texttt{T\_MAX} are what separate 3D stencil applications.

It is possible for the input and output arrays of a stencil computation to be the same, e.g., in successive over-relaxation [43]. This introduces spatial loop-carried data dependencies that render such stencils less suited for GPUs. Thus, this work is restricted to stencil computations in which the input and output arrays are different. For simplicity, only stencil programs with one input and one output array of the same dimensionality are considered.
void stencil_computation(
  float (*array1)[Y_SIZE][X_SIZE],
  float (*array2)[Y_SIZE][X_SIZE])
{
  float (*in)[Y_SIZE][X_SIZE] = array1;
  float (*out)[Y_SIZE][X_SIZE] = array2;
  for (int t = 0; t < T_MAX; ++t) {
    for (int z = 0; z < Z_SIZE; ++z)
      for (int y = 0; y < Y_SIZE; ++y)
        for (int x = 0; x < X_SIZE; ++x)
          {
            float temp0 = in[z+C0z][y+C0y][x+C0x];
            float temp1 = in[z+C1z][y+C1y][x+C1x];
            ...
            float tempN = in[z+C_Nz][y+C_Ny][x+C_Nx];
            out[z][y][x] = f(temp0, temp1, ..., tempN);
          }
    // Swap in and out pointers
  }
}

Figure 3.1: Generic 3D Stencil Computation

Stencil computations can have a variety of different stencil patterns, as shown in Figure 3.2. The stencil radius is the maximum distance between any element of the stencil and the centre of the stencil. The examples in Figure 3.2 all have a stencil radius of two. The stencil size is the number of input elements used to compute each output element.

When implemented naively in OpenCL the spatial loops of a stencil become a kernel, as shown in Figure 3.3. The computation of each output element is mapped to a single work-item and the for loops are replaced with OpenCL API calls to get_global_id(). Thus, in this example, a three dimensional global grid is used. The letters $x$, $y$, and $z$ will be used to refer to the 0th, 1st, and 2nd dimensions of this grid respectively. The time loop is executed by the host, causing it to repeatedly launch the kernel. The focus of this work is optimizing this kernel and thus optimizations affecting the time loop (e.g., tiling [14, 26]) are not considered.
Chapter 3. Stencil Computations

17

(a) Dense

(b) No-Corners

(c) Diamond

(d) Thumbackt

(e) Star

Figure 3.2: Stencil patterns

```c
__kernel void stencil_computation(
    global float (*)[Y_SIZE][X_SIZE],
    global float (*)[Y_SIZE][X_SIZE])
{
    int x = get_global_id(0);
    int y = get_global_id(1);
    int z = get_global_id(2);

    float temp0 = in[z+C_0z][y+C_0y][x+C_0x];
    float temp1 = in[z+C_1z][y+C_1y][x+C_1x];
    ...
    float tempN = in[z+C_Nz][y+C_Ny][x+C_Nx];
    out[z][y][x] = f(temp0,temp1,...,tempN);
}
```

Figure 3.3: Generic 3D OpenCL Stencil Kernel
Chapter 4

Optimizations

This chapter describes the optimizations that constitute the optimization space used in this work. Each of the first six sections describes an optimization and how it can improve performance. These sections also demonstrate how the optimizations are implemented in the code, along with examples where appropriate. Lastly, each section explains how its respective optimization can degrade performance and how the impact of the optimization can thus be difficult to predict; this justifies the need for tuning. In the last two sections, the size of the entire optimization space is calculated and interactions amongst the optimizations are further discussed.

4.1 Work-Group Size

The work-group size is the number of work-items in a work-group in each dimension. This size can have a significant impact on performance because it affects available parallelism. It needs to be large enough to utilize the hardware, but not so large as to reduce parallelism by reducing the number of work-groups that can execute at once. A large work-group size can also exhaust other resources such as registers or local memory, reducing performance or rendering a configuration unexecutable. The work-group size is normally set at kernel launch time in the host code and thus requires no change to the code of the kernel. With that said, in the kernels used in this work, the work-group size is hardcoded in the kernels for performance reasons; as will be detailed in Section 7.3.
4.2 Block Merging

Block merging combines the work done by adjacent work-items in a given dimension into a single work-item in the same dimension. The number of work-items merged together is referred to as the block merge factor. Figure 4.1 shows an example of applying this optimization to the 0th dimension of the OpenCL grid. A loop is introduced so that instead of computing a single element, each work-item computes multiple elements (two in the example). In order for the behaviour of the original kernel to be preserved, an accompanying decrease in the global grid size by the same factor is required such that half as many total work-items are launched.

Block merging increases thread work granularity thereby mitigating some of the overhead of launching threads. It also allows duplicate reads to be removed if the introduced loop is unrolled. Specifically, it allows global memory accesses to be replaced by register reads if the merged work-items require the same input element in their computation. The removal of these duplicate reads requires a compiler that is able to unroll the loop created by merging, identify duplicate reads and remove them.

However, block merging increases the register use of each work-item. This in turn can decrease the maximum work-group size and the number of concurrent work-groups executing on an SMX. Further, block merging in the inner-most dimension of the global grid can uncoalesce already coalesced accesses. These factors negatively impact performance and make it hard to predict the benefit of block merging.

Figure 4.1: An example of the block merge optimization with a merging factor of 2 in the x dimension.
Chapter 4. Optimizations

Figure 4.2: An example of the cyclic merge optimization with a merging factor of 4 in the y dimension.

4.3 Cyclic Merging

Cyclic merging combines non-adjacent work-items in a given dimension such that work is assigned to work-items in a round-robin fashion. The number of work-items merged in this manner is referred to as the cyclic merge factor. Cyclic merging introduces a loop into the kernel code for each dimension merged and also requires some changes to the array index calculations, as shown in Figure 4.2. As with block merging, an appropriate change in the launch configuration is required to preserve kernel behaviour. Specifically, the size of the global grid must decrease by cyclic merge factor.

Cyclic merging increases thread work granularity and has the benefit that when applied to the inner-most dimension of the grid it does not disrupt existing memory coalescing. However, it is unlikely to remove duplicate reads since the elements accessed by the merged work-items are likely far apart. Similar to block merging, it can limit parallelism by increasing register usage and decreasing the maximum work-group size and the maximum number of concurrent work-groups. Thus, it may or may not benefit performance.
4.4 Local Memory Caching

This optimization refers to caching the input data into local memory before reading it. This caching is done cooperatively by the threads in a way that ensures memory coalescing when possible. Only the minimum amount of data needed to avoid storing partial results is loaded from the z dimension at any one time so as to reduce the total amount of local memory needed.

Figure 4.3 shows an example of the changes to the kernel to implement this optimization. Reads from global to local memory are introduced at the beginning of the kernel, the existing reads from global memory are replaced with reads to local memory, and a barrier is added between the two types of reads for synchronization. The shape of the work-group influences this code; the example shown here uses a work-group of 32x1x1 work-items in the x, y, and z dimensions respectively. As can be seen in this example, the z dimension is treated differently than y and x in its use of a modulo operation to map z values from the input array to z values in local memory. This is done to simplify the case when cyclic merging is implemented in the z dimension. When this is the case, an extra load to local memory is added inside of the blocking loop along with appropriate barriers. This load causes the z slices to be cycled through local memory, with old ones over-written when no longer needed. One may also notice that the boundaries in the x dimension are handled differently than those in y and z. This is to ensure that the reads are properly aligned to the correct memory addresses to ensure full memory coalescing. Note that this code still can exhibit memory non-coalescing when the work-group size in the x dimension is very small.

Use of local memory can improve kernel performance when there are repeated accesses to the same data within a work-group because global memory transactions are replaced by faster local memory accesses. It can also improve memory coalescing when memory transactions are otherwise uncoalesced. However, large local memory requirements can unnecessarily restrict the maximum work-group size, limiting available parallelism. Furthermore, use of local memory adds the overhead of extra memory accesses and synchronization. Thus, it is not always beneficial [16].

4.5 Vectorization

This optimization converts reads from the input array, arithmetic operations on the intermediate result, and writes to the output array into vector operations to exploit specialized vector hardware. It only applies to kernels that are block merged in the x
Chapter 4. Optimizations

Figure 4.3: An example of the local memory optimization. Here local memory is used to cache the input array.
Chapter 4. Optimizations

---

```
-kernel void stencil_computation(
    global int (*in)[Y_SIZE][X_SIZE],
    global int (*out)[Y_SIZE][X_SIZE])
{
    int x = get_global_id(0);
    int y = get_global_id(1);
    int z = get_global_id(2);

    out[z][y][x] = in[z][y][x-2]+in[z][y][x]+in[z][y][x+1];
}
```

(a) Unoptimized stencil kernel

```
-kernel void stencil_computation(
    global int (*in)[Y_SIZE][X_SIZE/2],
    global int (*out)[Y_SIZE][X_SIZE/2])
{
    int x = get_global_id(0);
    int y = get_global_id(1);
    int z = get_global_id(2);

    int2 temp, temp1, temp2;
    // Aligned reads are performed normally
    temp += in[z][y][x-2];
    temp += in[z][y][x];

    // Misaligned reads must be broken up
    temp1 = in[z][y][x];
    temp2 = in[z][y][x+1]
    temp.s1 += temp1.s2;
    temp.s2 += temp2.s1;

    out[z][y][x] = temp;
}
```

(b) Optimized stencil kernel with block merging by a factor of 2 in the x dimension and vectorization by a factor of 2

Figure 4.4: An example of the vectorization optimization

dimension and only to those operations that were formerly done by multiple work-items that, post-merging, are accomplished by a single work-item. Thus the two optimizations in conjunction replace parallelism via work-items with parallelism via vectorization.

Figure 4.4 shows an example of the vectorization optimization. Reads that align to vector boundaries only require changing the data types of relevant variables to vector types. However, for input reads that do not align to a vector boundary, two vector reads need to be made and an interim vector result must be composed. Output writes are always aligned to vector boundaries.
Chapter 4. Optimizations

```c
__constant sampler_t sampler =
  CLK_FILTER_NEAREST |
  CLK_NORMALIZED_COORDS_FALSE |
  CLK_ADDRESS_NONE;

__kernel void stencil_computation(
  __read_only image3d_t in,
  global int (*out)[Y_SIZE][X_SIZE])
{
  int x = get_global_id(0);
  int y = get_global_id(1);
  int z = get_global_id(2);

  int temp[N+1];
  temp[0] = read_imagef(in, sampler, (int4)(x+C0x, y+C0y, z+C0z, 0)).s0;
  temp[1] = read_imagef(in, sampler, (int4)(x+C1x, y+C1y, z+C1z, 0)).s0;
  ...
  temp[N-1] = read_imagef(in, sampler, (int4)(x+CNx, y+CNy, z+C Nz, 0)).s0;
  out[z][y][x] = f(temp);
}
```

Figure 4.5: An example of the texture memory optimization.

### 4.6 Image/Texture Memory

This optimization stores the input array as an OpenCL image data type rather than a standard OpenCL memory buffer, allowing the kernel to exploit the texture cache. It requires changing the data type of the input array and its allocation on the host. Further, OpenCL API calls must be used to access the elements rather than normal array subscripts. The texture cache is hardware managed and thus requires no additional changes to the kernel. Example code demonstrating this optimization is shown in Figure 4.5.

This optimization can improve performance by replacing expensive global memory accesses with accesses to the texture cache. However, local memory can accomplish the same and using both simultaneously adds overhead for no benefit. Determining which to use for a given kernel is non-trivial.

### 4.7 Optimization Space

All of the optimizations are summarized in Table 4.1 along with abbreviations used throughout the rest of the document to refer to them. Each optimization takes on a value from the given range. Thus, an optimization configuration is the full set of values, one for each optimization. We restrict optimization values to powers of two to make the
Chapter 4. Optimizations

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Abbreviation</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work-group size</td>
<td>WX, WY, WZ</td>
<td>1</td>
<td>NX, NY, NZ</td>
</tr>
<tr>
<td>Block merge factor</td>
<td>BX, BY, BZ</td>
<td>1</td>
<td>NX, NY, NZ</td>
</tr>
<tr>
<td>Cyclic merge factor</td>
<td>CX, CY, CZ</td>
<td>1</td>
<td>NX, NY, NZ</td>
</tr>
<tr>
<td>Vector width</td>
<td>VX</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>Local memory</td>
<td>N/A</td>
<td>0 (no)</td>
<td>1 (yes)</td>
</tr>
<tr>
<td>Image memory</td>
<td>N/A</td>
<td>0 (no)</td>
<td>1 (yes)</td>
</tr>
</tbody>
</table>

Table 4.1: Optimization abbreviations and allowable values. N denotes the input/output array size in each dimension. X, Y, and Z correspond to the x, y, and z dimensions of the work-item grid respectively.

space more tractable and because it significantly simplifies the implementation of the optimizations. Further, the possible values for many of the optimizations depend on the other optimizations’ values. The product of W, B, and C for a given dimension must be less than or equal to N in that dimension. In addition, VX must be less than or equal to BX. When using a size of 256 for NX, NY, and NZ as will be done in this work, these restrictions result in a total optimization space of over 50 million configurations (see Appendix A for the math behind this calculation). Many of these configurations are not actually executable because they violate some limitation of the hardware, e.g., they use more local memory than is available on the device. Nonetheless, the space is prohibitively large to explore exhaustively.

4.8 Optimization Interactions

The preceeding optimizations interact with one another in non-trivial ways.

For example, block and cyclic merging can increase register usage of work-items and thus impact the optimal work-group size. Similarly, using local memory adds an additional resource constraint that can limit the work-group size.

Local memory also impacts the optimal work-group shape. When local memory is not used, a rectangular work-group shape, with a large size in the x dimension and small size in y and z, is usually favoured because of better cache performance. However, when local memory is used, more cubic work-group shapes are favoured because they result in more memory reuse for the same size of local memory.

Another example is how cyclic merging interacts with local memory due to the way in which the local memory optimization is coded. Although cyclic merging in y and z might seem similar since neither is in the dimension of coalescing, they can have dramatically
different effects on performance when used in conjunction with local memory due to the fact that cyclic merging in $y$ increases local memory use per work-group while cyclic merging in $z$ does not but instead adds more synchronization operations.

All of these interactions mean that finding the best optimization configuration is not simply a matter of determining the best configuration for each optimization in isolation. This will be explored further in the next chapter.
Chapter 5

Interestingness

This chapter demonstrates the need for auto-tuning by showing that the optimization space for the set of 104 kernels used throughout this work (defined in Section 7.1) is “interesting”. That is, it is sufficiently complex that simple approaches to determining a high-performing configuration fail. To this end, it will be shown that:

1. The majority of configurations in the space perform poorly.
2. The optimal value for each optimization varies from kernel to kernel.
3. The optimizations interact, requiring that they be explored together and not individually.
4. The best optimization configuration of one kernel does not necessarily perform well for other kernels.

5.1 Random Sampling

Ideally, the entire optimization space would be examined to demonstrate the above four points. However, this is not feasible in a reasonable amount of time for a single kernel, let alone for a large number of them. Consequently, the exploration of the optimization space is performed by randomly sampling configurations. These configurations are used as a proxy for the full optimization space.

To improve the quality of the randomly sampled configurations, some obviously bad configurations are omitted: (1) ones in which the block merging factor in the $x$ dimension is greater than the vectorization factor, since this reduces memory coalescing and experience has shown that this always results in poor performance and (2) ones with block merging in the $y$ and $z$ dimensions because the Nvidia compiler is not able to remove the
duplicate reads produced by block merging (see Section 4.2). Further, only one of four forms of data loading is allowed: global memory without vectorization, global memory with vectorization, local memory, or image memory. This is done because vectorizations, local memory, and texture memory all effectively target the same thing: the way in which data is loaded from global memory. This choice will be referred to as the data loading technique. These restrictions reduce the optimization space to 475,875 possible configurations. Although this a huge reduction in the number of configurations, it would still be impractical to examine the optimization space exhaustively. Extrapolating from the results in Section 8.5 indicates that it would take over four days to compile every configuration for a single kernel (the runtime would not be significant). Thus compiling all 104 that will be used later would take over a year.

When choosing the sample size it is important to ensure that enough of the space is searched to find a good performing configuration without taking an inordinate amount of time. In order to determine the appropriate sample size, samples of various sizes are taken on five of the 104 kernels. For each of the selected kernels, samples are taken ranging in size from 200 to 2000 optimization configurations in increments of 200 and the best runtime of each sample is determined. To make the results more statistically sound, each sampling size is sampled 10 times and an average and standard deviation of the best runtime is determined. The results of this process for the five chosen kernels, one of each of the stencil types mentioned in Chapter 3, are shown in Figure 5.1. The five kernels chosen all had a dimensionality of three and a radius of two as these parameters result in a significant amount of variability across the five kernels while running in a reasonable amount of time.

Most of these graphs show a general trend where performance initially improves (best runtime decreases) as sample size increases and then flattens off. The point at which this occurs varies from one kernel to the next, but a sample size of 1000 is sufficiently conservative to be within the plateau for any of these kernels. Consequently, a sample size of 1000 is used for the rest of the experiments.

5.2 Methodology

The rest of the experiments in this chapter are conducted on the full set of 104 kernels. For each sampled configuration, \( i \), in the upcoming results, speedup is calculated according to Equation 5.1 below, where runtime\(_b\) is the runtime of the best-performing configuration for that kernel from the random sample and runtime\(_i\) is the runtime of the configuration
of interest. Therefore, the speedup of a given configuration from the random sample is always less than or equal to one, with higher values being better.

\[
speedup_i = \frac{\text{runtime}_b}{\text{runtime}_i}
\]  

(5.1)

Sections 5.3 and 5.4 directly use the data obtained from performing a sample of 1000 configurations on each of the 104 kernels. Sections 5.5 and 5.6 perform additional experiments that require executing additional configurations.
5.3 High-Performing Configurations Are Few

Figure 5.2 shows a histogram of speedup values across all of the configurations for all of the kernels, that is, each bar indicates how many optimization configurations fell in the given speedup bin. The majority of the configurations perform poorly, with only 1.5% of them achieving a speedup within 10% of the best performing configuration. Conversely, a very large fraction of the configurations, 39.7%, experienced more than a 10x slowdown relative to the best configuration. This shows that the optimization space is skewed towards poorly performing configurations. This skew makes the space interesting, as it makes techniques such as random sampling less likely to be successful at determining a good optimization configuration.

![Figure 5.2: Distribution of configuration speedups](image)

5.4 High-Performing Configurations Are Difficult to Find

Next, it is shown that the best value for each optimization varies across different kernels. If a single value is always best for each optimization, then it doesn’t matter that most of the configurations perform poorly, one can always set each optimization to its best value.
Figure 5.3 shows the number of times each possible value for each optimization was used in the best-performing configuration for a kernel. The graphs show that there is significant variation in the optimization values that give the best performance, particularly for WX, WY, WZ, CZ, and data loading technique, where no individual value is best for more than 36 out of the 104 kernels. BX/VX shows less variation, likely because it has a fixed value (of one) whenever vectorization is not used.

5.5 The Optimizations Interact

Next, it is demonstrated that the optimizations interact by showing that tuning the optimizations individually is not sufficient to find a good optimization configuration. For each kernel, the best value for each optimization is determined while keeping the other optimizations held to the values used by the best configuration found by random sampling for that kernel. If the optimizations truly do not interact, then the held values for the optimizations that are not being tuned should not matter. Once all optimizations have been tuned independently, they are combined to arrive at a final configuration.

A histogram of the speedup values of this approach across all 104 kernels is shown in Figure 5.4. The leftmost bar consists of configurations that are not even executable. On average, this approach achieves only 88% of the performance of the best sampled configuration. That is, on average, it performs worse than its starting point. Thus, determining the best value for each optimization individually does not result in good performance; the optimizations must be examined together.

5.6 Good Configurations Are Not the Same Across Kernels

Finally, it is shown that the best performing configuration for a kernel does not, on average, perform well across all kernels, thus demanding the auto-tuning of each kernel individually. This is done by running the best performing configuration of each kernel on the 103 other kernels. Figure 5.5 shows a histogram of the resulting speedups. If the optimal configuration of each kernel worked best for all other kernels, all configurations would achieve a speedup of 1. The left-most bar, once again, corresponds to configurations that are not even executable. Although many of the configurations perform well, and a few even exceed a speedup of 1 on a particular kernel, the vast majority of configurations result in a slowdown, often a large one. On average, these configurations
achieve 64% of the performance of the best configuration. This indicates that the best performing configuration cannot be learned for one kernel and used for other kernels.

5.7 Summary

In conclusion, the four points made in this section show that the optimization space is interesting; it consists mostly of poorly-performing configurations and the best configurations vary significantly from one kernel to another. Furthermore, simple tuning approaches, such as tuning the optimizations independently or applying the best configuration found on one kernel to others, do not result in good performance. This indicates that to achieve good performance a strategy is required that accounts for the interactions amongst optimizations and also for the differences amongst input kernels.
Figure 5.3: Distribution of best values for each optimization
Chapter 5. Interestingness

Figure 5.4: Speedup distribution when each optimization is explored independently

Figure 5.5: Speedup distribution of the best configuration for each kernel on all other kernels
Chapter 6

Auto-Tuning Strategy

This chapter outlines the auto-tuning strategy. It consists of two components: a machine learning model to predict the optimal data loading technique and heuristics for tuning the remaining optimizations.

6.1 Predicting Data Loading Technique

The use of local and texture memory, as well as vectorization, all optimize the way in which data is loaded from the GPU’s memory hierarchy. To simplify the optimization space, data is only allowed to be loaded in one of four ways: from global memory without vectorization, from global memory with vectorization, from local memory, or from texture memory. Although some combination of these could be used in theory, our previous experience indicates that due to the extra overhead this is rarely beneficial in practice and thus this is a reasonable simplification to make. As mentioned previously, this choice is referred to as the data loading technique. This technique is determined first because it has the largest impact on performance.

The impact of the data loading technique depends to a large extent on the memory access patterns of the stencil computation. Thus, we hypothesize that the optimal data loading technique can be predicted using static program features, such as stencil size.

Figure 6.1 shows a histogram of the stencil sizes for which each data loading technique is best for the 104 kernels. There is a general transition from using global memory with vectorization to image/texture memory to global memory without vectorization and then finally to local memory as the stencil size (i.e., the number of input elements in the stencil) increases. This suggests that our hypothesis is valid; this (and possibly other) static features can be used to predict the optimal data loading technique.
Since the mapping of static program features to optimal data loading technique is likely complex and platform-specific it would be difficult to model it analytically. Instead, supervised machine learning [36] is used to predict the optimal data loading technique. Using machine learning for this part of the strategy has a significant benefit when changing platforms. With an analytical model, when changing platforms, the model would have to be completely reconstructed by an expert for the new platform. This machine learning approach, on the other hand, can easily be applied to a new platform by simply retraining the model on that platform. The model effectively learns the requisite expert knowledge.

A random forest [5] machine learning model is trained to this end. The inputs to this model are stencil size and dimensionality (1D, 2D, or 3D) of a kernel as well as which dimension, if any, differs from the rest if the kernel is asymmetric. For 1D kernels this corresponds to the dimension in which the stencil has a radius greater than zero, for 2D kernels this is the dimension in which the stencil has a radius of zero, and for 3D kernels this value is set to 0 except for the thumbtack kernels for which the value corresponds to the dimension containing the normal line. The stencil type is explicitly not used as a model input so that the model does not become unusable when encountering a stencil type not used in this work. The output of the model is the predicted optimal data loading technique for the kernel. The random forest consists of 100 trees, each node of each tree is constructed based on the best feature out of two randomly selected features (rather
than the best out of all three features), and there is no specified maximum depth to each tree.

A decision tree based model was chosen over other models due to the way in which the inputs interact. The meaning of a given value for one input can be dependent on the value of another input and as such linear techniques would likely fare poorly. They would attempt to extract the impact of a feature from the values of other features which is impossible with this set of features. A decision tree however can handle these types of interactions well by fracturing the input space appropriately. A random forest approach was chosen over a single, monolithic decision tree in order to reduce overfitting [5]. The small data size that will be used in this work in comparison to typical machine learning problems (only 104 kernels) means that overfitting would be a real danger with a complex model such as a single decision tree. In particular, the stencil size input feature ranges in value from 1 to 1331. With only 104 data points, this range will of necessity not be populated fully. An overfitted model might exploit this to tailor the model to individual kernels too closely. A random forest on the other hand resists overfitting as was described in Section 2.4.2 and thus helps to mitigate this problem. The specific parameters of the random forest, 100 trees, 2 random features per node, and no maximum tree depth, are simply the default parameters for the random forest implementation used in this work. Further exploration of these parameters is left for future work. In particular, limiting the maximum tree depth might improve generalization of the model by further reducing overfitting.

6.2 Grouping and Search Heuristics

The auto-tuning heuristics further reduce the optimization space by breaking up the remaining optimizations into groups that are explored independently of each other. The values of the optimizations in each group are exhaustively explored, that is, the kernel is run for every possible value, while keeping the values of the other optimizations fixed. This process will be referred to as tuning of the optimizations in question. Once the best values are found for these optimizations, they are fixed and the optimizations in the next group are tuned.

This approach has two key challenges. The first is how to group the optimizations, given that they are not independent of one another, as shown in Section 5.5. The groups are chosen so as to prioritize certain interactions amongst the optimizations over other interactions. Different groupings are explored in order to determine which interactions are most important. The second challenge is how to order the tuning of the groups. The
groups are tuned in order of their expected performance impact. The impact of group order is then mitigated by repeating the entire sequence of groups \( n \) times. The choice of \( n \) is important. If too small, then the heuristics will not execute enough iterations to reach a local minima. If \( n \) is too large, the heuristics may evaluate extra configurations needlessly and take longer to run. An exploration of the optimal value of \( n \) is conducted in Section 8.4.

Two complementary ways of grouping the optimizations are explored, by \textit{dimension} and by \textit{optimization}, resulting in two possible search heuristics. In the first, all optimizations are explored for each dimension independently. In the second, each optimization is explored independently for all dimensions.

### 6.2.1 Group-by-Dimension

This heuristic considers all of the optimizations that correspond to a particular dimension at once, regardless of the optimization they perform. Thus it prioritizes the interactions of the optimizations in each dimension above interactions between optimizations affecting different dimensions. This can result in optimal ratios of \( W \) to \( B \) to \( C \) in each dimension but will likely lead to suboptimal allocation of global constraints such as the optimal three dimensional work-group shape. Specifically, this heuristic follows these steps:

1. Set \( WY=1, WZ=1, BY=1, BZ=1, CY=1, \) and \( CZ=1 \)
2. Tune \( WX, BX, \) and \( CX \).
3. Tune \( WY, BY, \) and \( CY \).
4. Tune \( WZ, BZ, \) and \( CZ \).
5. Repeat steps 2-4 \( n \) times.

By tuning those optimizations affecting the \( x \) dimension first, it is given the highest priority as it has the largest impact. This is because the innermost dimension is the one that impacts memory coalescing and most significantly benefits from caching. Since larger values are usually better for the work-group size in the dimension of coalescing, the initial values of \( WY \) and \( WZ \) are set to one so as to allow \( WX \) to be as large as possible. Similarly, \( BY, BZ, CY, \) and \( CZ \) are set to one initially to allow \( BX \) and \( CX \) to explore as large a range as possible before encountering significant register pressure.

This ordering and these initial values risks finding a final work-group shape that is too large in \( x \) and too small in \( y \) and \( z \). This is an unavoidable but acceptable risk with
this technique due to the fact that larger values in $x$ are generally better. Furthermore, 
this risk is mitigated somewhat by the additional heuristic iterations (although those 
iterations cannot escape all bad local minima).

### 6.2.2 Group-by-Optimization

This heuristic considers all of the optimizations that correspond to a particular optimization at once, regardless of which dimension they affect. Thus the interactions it prioritizes are the opposite of the previous heuristic. It will likely result in ideal values for constraints that span the dimensions such as the overall work-group shape, but will find suboptimal ratios of work-group size to thread merging factors. It follows these steps:

1. Set $BX=1$, $BY=1$, $BZ=1$, $CX=1$, $CY=1$, and $CZ=1$.

2. Tune $WX$, $WY$, and $WZ$. On all passes after the first, rather than keeping $CX$, $CY$, and $CZ$ constant, keep the following products constant: $WXxCX$, $WYxCY$, $WZxCZ$.

3. Tune $BX$, $BY$ and $BZ$.

4. Tune $CX$, $CY$, and $CZ$.

5. Repeat steps 2-4 n times.

By tuning the work-group size first over the thread merging factors, overall parallelism level is prioritized over register usage. Similarly, after the first iteration, rather than holding $CX$, $CY$, and $CZ$ constant in step 2, $W_ixC_i$ for $i=X$, $Y$, and $Z$ is kept constant, thereby allowing the $Cs$ to vary in a controlled manner. This product corresponds to the amount of work assigned to each work-group, while the $Cs$ correspond to the amount assigned to each work-item. It is likely that the amount of work assigned to each work-group has a bigger impact on the total amount of available parallelism and thus this heuristic once again prioritizes overall parallelism level over register usage.

Finally, the block merging factor is tuned before the cyclic merge factor to ensure that there is as much register availability for block merging as possible because it is usually the more efficient way of merging threads as it is more likely to allow for removal of duplicate reads through register sharing.

Overall this ordering and these initial values risk finding overly large values for the work-group size and overly small values for the thread merging factors. As mentioned
previously this is considered an acceptable risk since overall parallelism level is believed to be the larger determinant of performance. As with the group-by-dimension heuristic, some of this risk is mitigated via the additional heuristic iterations.

6.3 Overall Strategy

The overall strategy when auto-tuning a kernel is: first use the machine learning model to predict the optimal data loading technique for the kernel then apply a heuristic while fixing the data loading technique to the value predicted by the model. Since the heuristics can results in different final configurations, even for the same data loading technique, the optimal data loading technique can vary between the two alternative heuristics. As such a separate model is trained for each heuristic.
Chapter 7

Implementation

7.1 The Stencil Kernels

In order to evaluate the heuristics and machine learning models, a large, representative set of stencil kernels is needed. Since there is no publicly available, large benchmark suite of such kernels, a set of synthetic kernels is generated instead. These kernels are created in such a way that their variation is controlled (see Section 7.2) and they are representative of a wide range of stencil computations.

Most, if not all, work on stencil auto-tuning in the literature has focused on only a handful of stencil programs. The set of generated programs in this work covers the shapes and radii of almost every stencil used in those previous papers. Thus, it is reasonable to assume that this set of synthetic kernels subsumes real stencils.

Five stencil subtypes are generated: dense, star, diamond, no-corners, and thumbtack, as depicted in Figure 3.2. For each subtype, 1D, 2D, and 3D stencils variants are considered in every possible orientation. For example, for 2D stencils, a variant in each possible plane (xy, xz, and yz) is created. All the stencils act on 3D input and output arrays consisting of 256 elements in each dimension. Their radii range from 0 to 5. Each stencil has the same radius in every one of its dimensions as it was found that assymetric kernels were very rare in practice. Nonetheless, we explore the impact of asymmetry in Section 8.8. Generating every combination of these parameters results in 104 unique kernels, as summarized in Table 7.1.

As can be seen from the table, there is a good spread of values across the different dimensionalities, radii, and stencil patterns. Any unevenness is a result of how duplicated configurations are handled. If duplicates were allowed, then every stencil type would have 15, 15, and 5 1D, 2D, and 3D variants respectively and would have 7 variants of each possible radius (with the exception of the thumbtack type). However, many of these
variants are actually the same and so they are removed from the set. For example, diamond kernels of radius two are identical to star kernels of radius two. These kernels are, arbitrarily, classified as star kernels for the purposes of the table, which is why the diamond kernel entries for 2D, 3D, and radius one are smaller than those of the star kernels. The thumbtack kernels are a special case because there is no thumbtack variant of one or two dimensions. On the other hand, while there exists only one orientation of each of the other types in three dimensions, the thumbtack has three possible orientations since it is inherently asymmetric. Thus there are three times as many 3D thumbtack kernels as there are 3D dense kernels.

All of the kernels perform a weighted sum of their input elements using weights that are randomly chosen during program generation. These weights are thus known at kernel compile time. It is assumed that boundary data is copied into place beforehand. This assumption results in no loss of generality because any boundary handling technique could first be implemented by copying the desired boundary into place beforehand. Finally, all the kernels are run with single-precision floating point input data that is randomly generated.

### 7.2 Genesis Implementation

The kernels are written and generated using Genesis [6], a language and accompanying preprocessor for generating synthetic programs.

Figure 7.1 shows a simple Genesis example that will be used to illustrate some relevant Genesis constructs. This snippet of code is used to produce the reads in the loop body of the dense kernels, i.e. the stencil itself. The first three lines use the `genloop` construct to cause the indicies i, j, and k, to iterate between the min and max values for the
z, y, and x dimensions respectively. These min and max values are set earlier in the code based on the radius, dimensionality, and orientation of the stencil in question. The difference between the genloop construct and a normal loop present in the code is that genloop executes at program generation time—when the Genesis compiler itself is run. Thus this snippet of code does not result in any loops in the generated kernel file. Instead whatever code is produced inside of these loops will be replicated many times. The next line samples a value weight_cube from a distribution weight_cube_dist that is also defined elsewhere. This is used to choose a random value for the weight of this particular stencil element. Finally, the fifth line takes the three indices along with the weight and calls a Genesis feature, the Genesis equivalent of a function, with them as arguments. This feature, named read_access, generates a read from the input array using the given indices as offsets as well as a multiplication by the given weight. The code it generates adds the result of this operation to a temporary variable that is later written to the output array. Note that as with genloop, Genesis features differ from normal functions in that they are executed at program generation. Thus the read_access feature in this example does not result in a function call in the generated code, instead it is replaced with a single line of code.

```
genloop i : −1*$\{z_{\text{min}}\}:*\{z_{\text{max}}\}
genloop j : −1*$\{y_{\text{min}}\}:*\{y_{\text{max}}\}
genloop k : −1*$\{x_{\text{min}}\}:*\{x_{\text{max}}\}
  value weight_cube sample weight_dist
  ${\text{read_access}({i},{j},{k},{weight_cube},{id})}$
end
end
end
```

Figure 7.1: Genesis code for the loop body of the dense kernels

An example output of the read_access feature with inputs of 1, 0, -1, and 3 is shown in Figure 7.2a. The code has been simplified to remove code for vectorization, local memory, and texture memory, thus only the global memory without vectorization variant of the code remains. If the entire block of Genesis code from Figure 7.1 is run, output such as that in Figure 7.2b is created. In this particular example, z_min, z_max, x_min, and x_max were set to one while y_min and y_max were set to 0. This corresponds to the 2D dense kernel in the xz plane with radius one. As with the previous example, only the code for global memory without vectorization is shown.

The elements of the kernel code that vary across kernels are implemented using Genesis constructs such as genloop while the optimizations are implemented using C preprocessor directives. Thus, before compiling a kernel, it is possible to select what
temp += 10 * input [z+1][y+0][x+1];

(a) Read_access feature - simplified

(b) Dense kernel loop body - simplified

(c) Read_access feature - less simplified

Figure 7.2: Example Genesis output

optimizations to apply by setting various define statements. This allows optimization-related parameters to be known at compile time, allowing the compiler to perform as much simplification as possible. Figure 7.3 demonstrates this with the definition of the read_access feature. As can be seen, preprocessor #if statements are used to select the desired version of the code based on the constants USE_LOCAL_MEMORY_INPUT and STORE_ARRAYS_AS_IMAGES which are defined elsewhere. Since these preprocessor statements are not evaluated until after the Genesis preprocessor is run, the generated code contains all three variants of the feature read. The generated output of this feature with the local and texture memory variants included is shown in Figure 7.2c. Even this variant of the code has been simplified as the vectorization code was removed for brevity. Although the generated code is complicated, the OpenCL preprocessor simplifies it considerably. For example, when USE_LOCAL_MEMORY_INPUT and STORE_ARRAYS_AS_IMAGES are both false, the code in Figure 7.2c reduces to that shown in Figure 7.2a.

A single Genesis input file is used to generate all of the kernels used in this work and each kernel, once generated, contains every possible optimization configuration. As a result, each optimization, and in fact all code that is common across kernels, needs to
be coded only once, albeit in a generic way. An example of this is shown in Figure 7.4 in which the Genesis code for the dense and star stencils are shown to reuse the read_access feature. The Genesis construct genif is used to distinguish between the two. If the kernel code itself were written in such a way, with ifs, loops, and function calls, then there would likely be a drop in performance due to the added overhead, depending on how aggressively the compiler inlines functions and unroll loops. But since the Genesis preprocessor is run before the kernel is ever compiled, the generated code retains none of these artifacts. Overall the combination of Genesis preprocessor and OpenCL preprocessor results in final code that is very similar to what a programmer targeting the desired optimization configuration from the beginning would write.

As a result of limitations in the OpenCL vectorization syntax, some of the generated files can be very long in order to contain every possible vectorization variation. This length is then dramatically reduced when the OpenCL preprocessor is run and chooses the desired optimization configuration. As a result this large file size has no impact on kernel runtime but it may be a significant factor in kernel compile time.

Keeping the kernel parameters as Genesis values also makes it easy and convenient to generate any subset of the kernels. The section of Genesis code that performs initialization is known as the generate block; our generate block is shown in Figure 7.5. Each line in the body of this block defines a distribution that will be enumerated later in the Genesis code. The ranges of all these distributions can be set as desired to modify the set of kernels generated. The specifications shown result in far more combinations than the number of kernels used in our experiments due to duplicated stencils. To prevent generation of duplicate kernels, a validity check feature is executed in Genesis that prevents certain duplicated configurations from generating files by killing them using genassert statements. Thus when Genesis is run on the input file as shown in Figure 7.5, it attempts to generate every combination of the listed parameters, i.e. 270 files, but of these attempts, only 104 are successful.
There are two main complications that this kernel specification leaves out that could be the subject of future study.

First, only kernels for which the weights are known at compile time are considered. This allows the weights to be encoded in the instructions themselves. There are some real applications for which this is not the case, and this gives rise to a series of additional optimizations that could be applied to the manner in which the weights themselves are loaded from memory. In particular, if the stencil were large, and thus all the weights couldn’t be stored in registers at one time, the loading of those weights could have a
significant impact on performance. Constant memory seems well suited to this task and could prove beneficial.

Second, all of the stencils perform a weighted sum of their inputs to compute their output. As a result the amount of computation per input element is fixed. Since this affects the computation to communication balance of the kernels it can have an impact on the benefit of many optimizations. While the heuristics would likely be applicable to stencils with different amounts of computation per input element, it would probably be necessary to include the computation per element as an input to the machine learning model in order to properly predict the optimal memory type for such kernels.

### 7.3 The Auto-Tuner

In order to evaluate the performance of the auto-tuning strategy as well as the various baselines that it will be compared against, some auto-tuning infrastructure was developed. The infrastructure enables the repeated compilation and execution of different optimization configurations for a kernel so as to facilitate result collection.

Figure 7.6 shows the auto-tuner’s flow. It takes as input a set of heuristic specification files and an input kernel. Each specification file describes what optimizations to vary and which to keep constant in each step of a heuristic.

The auto-tuner copies the input data from the host to the GPU before the first configuration is tested. If a desired configuration has already been run its data is retrieved from a cache of past results, otherwise the auto-tuner sets various optimization-related parameters in the code and compiles the kernel using the OpenCL API. If the kernel can be launched on the device, the OpenCL binary is run on the GPU four times and the average runtime of the last three runs is recorded. Finally, the output of the kernel is checked for correctness. This final check was successful in all results presented in this work.
This approach requires a recompile for every unique optimization configuration, even if configurations differ only in runtime parameters such as the work-group size. This allows these values to be compile-time constants rather than being queried at runtime using OpenCL API functions. This maximizes the compiler’s ability to optimize by constant folding, constant propagation and dead code elimination.

In order to train the machine learning model, each heuristic is run with each data loading technique. As such there is a separate heuristic description file for each heuristic-data loading technique combination (eight in all). A model is then trained and tested for each heuristic using leave-one-out cross-validation [36] to ensure that no kernel was part of the training set used to construct the model used to test it. There is no need for automated feature extraction of the source-level features because these features are known for a given synthetic kernel when it is generated. However, in order to implement the strategy in a compiler for arbitrary stencil programs, such support would be required.

### 7.4 Heuristic Specification Files

The auto-tuner supports a generic heuristic specification file syntax that can be used to describe a wide-range of heuristics. Each line of a heuristic specification file corresponds to a group of kernel runs and consists of a series of statements that execute in order. Each statement assigns a value to a variable. If the name of that variable corresponds to an optimization then this assignment sets the value of that optimization, otherwise the variable has no direct impact on the optimization configuration that is run but can be used to store temporary values. This is necessary to express some more complicated
heuristics. One can also specify ranges of values and the stride that should be used to cover that range. Furthermore, every value can possibly be a mathematical expression containing constants as well as other variables. If the value for a particular optimization is not assigned on a given line then it takes on the value of that optimization from the best configuration that has been found so far by the heuristic while executing previous lines. Finally, random sampling of values within a range is also supported. This last feature makes it possible to represent the random sampling done in Chapter 5 as a heuristic. In fact all of the experiments from that section were conducted using heuristic specification files.

Figure 7.7a shows the heuristic specification file for the group-by-dimensions heuristic in the case where local memory was chosen as the data loading technique. In this case, n=1 (i.e. one additional iteration over the groups is performed). The lines of this file correspond to the steps from the description of the heuristic provided in Section 6.2.2. The first line consists of steps 1 and 2 as it sets the initial values and iterates over those optimizations affecting the x dimension. Lines 2 and 3 correspond to steps 3 and 4 respectively as the heuristic iterates through those optimizations affecting y and then z. Finally, lines 4-6 repeat steps 2 through 4 again. Note that many of the expressions make use of NX, NY, or NZ. These are special constants set by the auto-tuner program to be equal to the data size in each of the three dimensions. The "*2" in the range expressions indicates that the value in question grows by a factor of two with each iteration, consequently the values of W and C explore all allowable powers of two. It is also worth noting that BX, BY, and BZ are not explored by this heuristic specification file due to the limitations of the Nvidia compiler mentioned in Section 5.1. If B was explored the file would change to be as shown in Figure 7.7b.
Chapter 7. Implementation

(a) No BX, BY, or BZ

(b) With BX, BY, and BZ

Figure 7.7: The heuristic specification file for the group-by-dimensions heuristic using local memory
Chapter 8

Evaluation

This chapter outlines the evaluation methodology and presents performance results for the auto-tuning strategy discussed in Chapter 6.

8.1 Reference Approaches

The strategy is compared to 3 reference approaches: random sample (Rand), expert search (ExpS), and an oracle (Oracle).

The Rand approach corresponds to running the first stage of the interestingness experiment described in Chapter 5, i.e. running 1000 configurations, and selecting the best performing one.

The ExpS approach represents the one used by most existing stencil auto-tuning work in the literature (described in Chapter 9). It performs exhaustive, empirical auto-tuning but only on a restricted subset of the space chosen by an expert. We use our experience to restrict the space as follows: VX must be less than or equal to 4, WX must be greater than or equal to 32, WY*CY must be less than or equal to 4, and WZ*CZ must be less than or equal to 4.

There is an oracle-based approach for each of the two heuristics. Each consists of running the heuristic with every data loading technique and then taking the best configuration found across all of them. These oracles effectively give upper bounds on the speedups that could be achieved with perfect machine learning models.
8.2 Evaluation Platform and Metrics

The results are gathered on an Nvidia GTX Titan GPU forced to its highest performance state to ensure that the results are not skewed by power saving features. The host is an Intel i7 960 with 6GB of RAM running Ubuntu 12.04 and Nvidia driver version 325.15. This version of the Nvidia OpenCL compiler does not remove duplicate reads introduced by block merging. Thus, BY and BZ are fixed to one for this evaluation (see Section 5.1). The machine learning models are created using Weka 3.6 [15] with its default random forest parameters: 100 trees, each of unlimited depth, and each node created from one of two randomly chosen input features.

Each of the machine learning models is assessed using two metrics: its absolute accuracy, i.e., on what fraction of the kernels it predicts the correct data loading technique, and the average speedup of the heuristic using the model relative to its corresponding oracle, averaged over all kernels. This speedup reflects the penalty of a mispredicted loading technique on performance. This second metric is more important than absolute accuracy in practice, because it is what the end user of the strategy would care about. Even if the machine learning model never predicted the optimal memory type, if it always predicted one that was very close to it in performance, the model would still be useful.

The performance of the heuristics is also assessed with two metrics. The first is the runtime of the best configuration discovered by the heuristic. This time is normalized by calculating the speedup with respect to the best configuration found by \texttt{Rand} according to Equation 5.1. The geometric mean of speedup is used to report average speedup across the kernels.

The second metric is the time the heuristic takes to get to that configuration, measured by the number of configurations examined, the compile time on the host and the runtime of these configurations on the GPU. The runtime of each configuration corresponds to a single run of a kernel, i.e. a single iteration of the time loop. The compile time and runtime averages are also calculated as geometric means so that longer running kernels do not have a disproportionately large impact on the results.

8.3 Experiments

The rest of this Chapter is composed of a number of experiments and their results. First, an experiment is performed to determine the optimal number of iterations of the heuristics. Next the main results themselves are presented, including the performance of the strategy with each of the heuristics as well as the performance of the baselines. After
that three interesting variations are considered: using expert knowledge to restrict the
heuristic search space, allowing the work-group size to be a variable, and changing the
input and output array sizes and shapes.

8.4 Determining a Good Value for $n$

As was mentioned in Section 6.2, the heuristics have a parameter, denoted $n$, that deter-
mines how many times they iterate over their optimization groups. Increasing $n$ allows
a heuristic to potentially find a better performing optimization configuration at the cost
of more compiles and more runtime. This tradeoff is only possible up until the point
that the heuristic reaches a local minima and no longer examines any further unique
configurations.

In order to determine a good value for $n$, the group-by-dimensions heuristic is run
on a subset of the kernels for various values of $n$. The seven kernels of dimensionality
three and radius two were chosen for this experiment as they represent a reasonable
cross-section of the overall kernel set while running in a reasonable amount of time. For
each value of $n$, the geometric mean of the runtime of the best configuration found by
the heuristic across each of the seven kernels is calculated. Plotting these values as $n$ is
varied produces the graph shown in Figure 8.1.

![Figure 8.1: Performance for various values of $n$](image)

As can be seen there is a large benefit from adding one iteration to the heuristic,
i.e. going from $n=0$ to $n=1$. Although it is not visible from the figure, there is also a
very slight improvement from n=1 to n=2. After that there is no benefit to additional iterations. In terms of the results on the individual kernels, it took no additional iterations to reach a local minima on four of the seven, one iteration for two of them, and two iterations for only one. The fact that only one kernel needed the second iteration explains why the average shows so little benefit for that iteration. As one of the test kernels did require it to achieve a local minima, in order to be conservative in the rest of the results presented, a value of n=2 will be used.

8.5 Auto-tuning Strategy Performance

The speedups, number of unique configurations, compile times and GPU runtimes (in seconds) for all heuristics and reference approaches, averaged over all kernels, are shown in Table 8.1. The table also shows “best count”, the number of times each strategy found the best performing configuration amongst those explored. Note that the best counts for the oracles are necessarily greater than or equal to that of the heuristics as they look at a superset of the configurations of those strategies. Further, the best count column total for the baselines and the oracles is more than 104 because for some kernels more than one strategy finds the best configuration.

<table>
<thead>
<tr>
<th>Speedup</th>
<th>Configs</th>
<th>Compile Time (s)</th>
<th>Runtime (s)</th>
<th>Best Count</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Baselines</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expert search</td>
<td>1.06</td>
<td>1800</td>
<td>1249</td>
<td>13.53</td>
</tr>
<tr>
<td>Random sample</td>
<td>1.00</td>
<td>995</td>
<td>664</td>
<td>14.24</td>
</tr>
<tr>
<td><strong>Oracles</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dimensions</td>
<td>1.14</td>
<td>1122</td>
<td>741</td>
<td>29.94</td>
</tr>
<tr>
<td>Optimizations</td>
<td>1.19</td>
<td>3958</td>
<td>2474</td>
<td>45.01</td>
</tr>
<tr>
<td><strong>Our Heuristics</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dimensions</td>
<td>1.12</td>
<td>293</td>
<td>166</td>
<td>5.70</td>
</tr>
<tr>
<td>Optimizations</td>
<td>1.16</td>
<td>1019</td>
<td>610</td>
<td>9.07</td>
</tr>
</tbody>
</table>

Table 8.1: Performance of heuristics and reference approaches

A number of observations can be made from this table. First ExpS outperforms Rand both in speedup and in best count. It explores significantly more configurations in order to do so and thus requires more compile time, however it requires slightly less device-side runtime as these configurations are generally better performing. Secondly, the machine-learning-based heuristics are able to achieve significantly higher speedups than both of those reference approaches with the same number or fewer configurations considered.
and less device-side runtime. In particular the dimensions heuristic considers 71% fewer configurations than Rand (293 vs. 995) while finding ones that are 12% better on average and considers 84% fewer configurations than ExpS (293 vs. 1800) while finding ones that are 6% better on average. Similarly, it takes 60% and 58% less device-side runtime than Rand and ExpS respectively.

Table 8.2 gives some insight into why the heuristics outperform the baselines so thoroughly. It summarizes the speedups and best counts achieved by the heuristics for each data loading technique (i.e., without the use of the machine learning models). None of the individual loading techniques is best all of the time, and none of them has a speedup as large as Rand on average. However, Rand never finds the best configuration of all of the strategies on any given kernel as evidenced by its best count of 0. Conversely, the individual data loading techniques, although worse on average, have much higher best count values; when they do well, they do very well. By combining the strengths of all of these data loading techniques, the machine-learning-based strategies are able to achieve best counts of 23 and 55 for the dimensions and optimizations heuristics respectively. As a result, they are able to achieve greater average speedups than Rand and ExpS.

<table>
<thead>
<tr>
<th>Data Loading Technique</th>
<th>Speedup</th>
<th>Best Count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Dims Opts</td>
</tr>
<tr>
<td>Global memory without vectorization</td>
<td>0.91</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 7</td>
</tr>
<tr>
<td>Global memory with vectorization</td>
<td>0.84</td>
<td>0.90</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 19</td>
</tr>
<tr>
<td>Image/Texture memory</td>
<td>0.95</td>
<td>0.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7 14</td>
</tr>
<tr>
<td>Local memory</td>
<td>0.95</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13 32</td>
</tr>
</tbody>
</table>

Table 8.2: Breakdown by data loading technique

Comparing the dimensions and optimizations heuristics, the optimizations approach is able to achieve a substantially higher speedup (16% vs. 12%) but at the cost of more than three times the number of configurations and almost double the runtime. Thus, the group-by-dimensions heuristic seems to provide the better tradeoff between the various metrics. With that said, which of these approaches is best could vary from one user to the next depending on his or her particular requirements for speedup and tuning time.

The absolute accuracies of the machine learning models are 83% and 76% for the dimensions and optimizations heuristics respectively. Relative to the oracles, these strategies achieve 98% and 97% of the maximum possible performance. These values indicate that although the models do not always predict correctly, when they are incorrect, their mistakes have little impact on performance.
In general compile time per configuration is fairly consistent so trends in configurations considered tend to be mirrored in compile time. The heuristics do have slightly lower average compile times per configuration due to them disproportionately looking at texture and local memory configurations which tend to be slightly faster to compile than global memory configurations. Compile time is not necessarily correlated with runtime though, as was shown by the two baselines. It is also important to note that compile time dramatically dominates runtime for all of the strategies. This might lead one to believe that runtime is irrelevant, but this is a dangerous assumption to make. While compile time was dominant for these kernels, longer running kernels might show the opposite behaviour, so it is important to consider both time metrics.

8.6 Restricting the Optimization Space

One can use expert knowledge of the optimization space to further limit the exploration performed by the heuristics and thus reduce the number of configurations they consider. For this experiment, the work-group size in the $x$ dimension was forced to be at least 16 and the work group sizes in the $y$ and $z$ dimensions to be at most 32. This ensures that memory coalescing is exploited and prohibits a total work-group size over 1024, which is rarely beneficial. These restrictions can be looser than those used by ExpS because the heuristics and machine learning model already significantly reduce the optimization space size.

The results of applying these restrictions to the dimensions heuristic are shown in Table 8.3. There is a significant reduction in the number of configurations considered and compile time and an even more significant reduction in runtime. This is because the excluded configurations are generally the worst performing ones. Speedup drops only slightly because the new heuristic rarely misses any of the best configurations.

This shows that this strategy complements expert search approaches as the two strategies can effectively be used together.

<table>
<thead>
<tr>
<th>Speedup</th>
<th>Configs</th>
<th>Compile Time (s)</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensions</td>
<td>1.12</td>
<td>293</td>
<td>166</td>
</tr>
<tr>
<td>Dimensions Restricted</td>
<td>1.11</td>
<td>172</td>
<td>96</td>
</tr>
</tbody>
</table>

Table 8.3: Results for the restricted heuristics
8.7 Variable Work-Group Size

As mentioned in Section 7.3, in order to allow the compiler to optimize as much as possible, the work-group size is fixed at compile time. However, the OpenCL standard allows the work-group size to be set at kernel launch time and to be queried by the kernel at runtime using an API call (as mentioned in Section 2.2).

Fixing the work-group size dramatically increases the number of compilations that must be made and also affects the design of the heuristics themselves. For example, if changing the work-group size did not require a recompile but only a (possibly cheaper) rerun of a kernel, then the heuristics would ideally explore any compiled configuration for multiple, if not all, work-group sizes.

Even if leaving the work-group size as a variable has a large impact on performance, if that impact is consistent across all configurations then its benefit can still be gained, without the accompanying costs, by running the heuristics with variable work-group sizes and then fixing the work-group size for the final configuration. This would still result in the same final configuration. Thus fixing the work-group size during heuristic execution is only justified if it results in a large increase in performance that varies significantly from one configuration to another such that not including this optimization during heuristic execution would result in a substantially suboptimal final configuration.

In order to test this idea, variants of the kernels are created in which the work-group size is not a fixed constant but rather is queried using the OpenCL API call `get_local_size`. The group-by-dimensions heuristic is then run on these modified kernels. Once the heuristic has determined a final, best configuration, that configuration is run using the variant of the kernel with a fixed work-group size.

The results for this experiment are shown in the second row of Table 8.4. The first row of this table is the performance on the original kernels, copied from Table 8.1 above. Relative to the same heuristic applied to the fixed work-group size kernels, this approach results in a dramatically lower average speedup of only 1.01, i.e. barely better than `Rand`. The two approaches consider the same number of configurations, as is to be expected given that they apply the exact same heuristic. The variable work-group size variant, however, takes dramatically less compile time to do so as it does not require a recompile for each configuration. This results in a 79% reduction in compile time relative to the same heuristic working on fixed work-group size kernels. There is, of course, an increase in runtime due to the fact that many of the explored configurations perform worse. On average, runtime increased by 26%. For kernels that are compile-time dominated, such as those in this experiment, this compile-time/runtime trade-off is very desirable. Overall
though, the huge degradation in speedup renders this approach undesirable and thus justifies the decision to fix the work-group size at compile-time.

<table>
<thead>
<tr>
<th></th>
<th>Speedup</th>
<th>Configs</th>
<th>Compile Time (s)</th>
<th>Runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed work-group size</td>
<td>1.12</td>
<td>293</td>
<td>166</td>
<td>5.70</td>
</tr>
<tr>
<td>Variable work-group size</td>
<td>1.01</td>
<td>293</td>
<td>35</td>
<td>7.18</td>
</tr>
<tr>
<td>Combination</td>
<td>1.12</td>
<td>303</td>
<td>122</td>
<td>5.95</td>
</tr>
</tbody>
</table>

Table 8.4: Results for Variable Work-Group Size Kernels

Table 8.5 shows a breakdown of the speedup results by data loading technique. That is, each row in this table shows the performance of the group-by-dimensions heuristic if the listed data loading technique was used for all kernels rather than the choice of the machine learning model. The first column shows the performance on the fixed work-group kernels copied from Table 8.2 above. The second column shows the results on the kernels with work-group size as a variable. Finally, the third column shows the results on the kernels with work-group size as a variable where the final configuration is reran on the kernels with work-group size as a constant. This last step isolates the direct runtime impact of making the work-group size a variable from its impact on the heuristic. As can be seen, while there is a huge loss in performance for the kernels that use local memory and texture memory, there is almost no impact for those kernels that use global memory. Furthermore, a large fraction of the performance loss on the local memory configurations is recaptured by using a constant work-group size for the final configuration.

<table>
<thead>
<tr>
<th>Data Loading Technique</th>
<th>Speedup</th>
<th>Constant</th>
<th>Variable</th>
<th>Constant Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global memory without vectorization</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td></td>
</tr>
<tr>
<td>Global memory with vectorization</td>
<td>0.84</td>
<td>0.83</td>
<td>0.80</td>
<td></td>
</tr>
<tr>
<td>Image/Texture memory</td>
<td>0.95</td>
<td>0.77</td>
<td>0.77*</td>
<td></td>
</tr>
<tr>
<td>Local memory</td>
<td>0.95</td>
<td>0.66</td>
<td>0.87</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.5: Breakdown of variable work-group size kernel results by data loading technique. * One of the final configurations that used texture memory was unexecutable on the fixed work-group size variant of the kernel. The variable work-group size version was used for this kernel.

To understand why a fixed work-group size provides such a significant benefit to optimization configurations that use local memory, one has to analyze the code itself. In order to map the much larger global memory space into the small section of local memory used by the kernel, a modulo operation is required. The divisor of this modulo
is a function of the work-group size and some compile-time constants. By fixing the work-
group size at compile time, this divisor can be calculated ahead of time using constant
propagation and constant folding and the integer division necessitated by the modulo
operation can be replaced via strength reduction. However, if this divisor is not known
at compile-time then a costly integer division operation must be present in the compiled
program, potentially causing a dramatic increase in runtime.

Given that the speedup did not degrade on the kernels that preferred global memory,
one would expect that a combination of the fixed and variable work-group size approaches
could be beneficial. By using the fixed work-group size variant of a kernel when the ma-
chine learning model predicts local or texture memory and the variable work-group size
variant otherwise, one can achieve some of the compile time reduction of the variable
work-group size approach without any of the speedup penalty. The results for this ap-
proach are shown in the third row of Table 8.4. As expected, the average speedup is
the same as the group-by-dimensions heuristic run exclusively on kernels with a fixed
work-group size, but the compile time is reduced by 27% while the runtime increase is
fairly small at only 4%. For compile-time dominated kernels this is a huge improvement,
and this approach should be used in place of the original version of the strategy.

8.8 Array Size and Shape

As mentioned in Section 7.1, all of the kernels use three dimensional input and output
arrays of 256 elements in each dimension. The size of these arrays likely has an effect
on the interestingness of the space and thus the need for the strategy presented here. In
particular, one would expect that as the input becomes sufficiently small to fit in certain
levels of the cache hierarchy, one of the data loading techniques would become best for
all kernels (i.e. the optimization space would become less interesting). To determine if
and when this is the case, the group-by-dimensions strategy was run on arrays of various
sizes ranging from 32x32x32 to 512x512x512. To save time, this experiment was only run
on a subset of the kernels. As in Section 8.4, the seven three dimensional stencils with
radius two were chosen. Table 8.6 shows the performance of the strategy, the relevant
oracle, and random sampling on these different sizes. The first three rows of data in
this table are speedups for the three techniques mentioned. The last row shows the
absolute number of correct predictions made by the machine learning model (out of the
seven kernels) for each of the array sizes. As can be seen, as the input size gets smaller,
the machine learning model becomes less accurate and the performance of the strategy
decreases. For very small array sizes (32x32x32), even the oracle performance is very
low, indicating that the heuristic provides little benefit over random sampling. This is likely because the space itself has become less interesting and thus random sampling is sufficient to find a good-performing configuration as most configurations have similar performance. Investigating why the model mispredicts for these small input sizes shows that global memory becomes more and more favoured as the input size decreases. This is consistent with the hypothesis that, at these smaller array sizes, the L1 and L2 caches of the GPU are able to better address the memory needs of the kernel, and, as a result, caching in the texture cache or in local memory is unneeded.

<table>
<thead>
<tr>
<th>Data Size per dimension</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedups</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dimensions oracle</td>
<td>1.03</td>
<td>1.10</td>
<td>1.10</td>
<td>1.07</td>
<td>1.16</td>
</tr>
<tr>
<td>Dimensions predicted</td>
<td>0.80</td>
<td>0.97</td>
<td>1.07</td>
<td>1.07</td>
<td>1.16</td>
</tr>
<tr>
<td>Random Sample</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Correct Predictions</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 8.6: Heuristic performance on various input sizes

In all of the previous experiments, the input and output arrays had symmetric shapes. In many applications though, particularly in the field of image processing, asymmetric arrays are more common. For example, if $x$ and $y$ represent the spatial dimensions of an image and $z$ represents the time axis, there are many applications in which $z$ will be much smaller than $x$ and $y$. In order to determine how the strategy performs on these input sizes, the group-by-dimensions strategy was run on kernels with input sizes of $1024 \times 1024$ in $x$ and $y$ and ranging in size from 8 to 128 in the $z$ dimension. The results of this experiment are shown in Table 8.7 below. As with the previous experiment, the speedup for the strategy, its oracle, and random sampling are shown, along with the number of correct predictions (out of seven) made by the machine learning model. While the strategy does not perform quite as well as it does on the original data size of $256 \times 256 \times 256$, it does almost as well. What is most noticeable about these results is the consistency in the strategy’s performance. Regardless of the size in the $z$ dimension, the machine learning model predicts the correct data loading technique for five of the seven kernels and the heuristic achieves a speedup of 5-6%. Of note is that none of the sizes considered in this experiment result in a total number of array elements less than that considered by the $128 \times 128 \times 128$ test in the previous experiment, thus this experiment removes the impact of input size in its investigation.
### Table 8.7: Heuristic performance on various input sizes

<table>
<thead>
<tr>
<th>Z Size</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speedups</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dimensions oracle</td>
<td>1.11</td>
<td>1.11</td>
<td>1.09</td>
<td>1.11</td>
<td>1.10</td>
</tr>
<tr>
<td>Dimensions predicted</td>
<td>1.06</td>
<td>1.05</td>
<td>1.06</td>
<td>1.05</td>
<td>1.05</td>
</tr>
<tr>
<td>Random Sample</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td><strong>Correct Predictions</strong></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Overall, these two experiments show that while the strategy breaks down at small input sizes, it is insensitive to input size once a certain threshold has been reached, and furthermore, it is insensitive to input shape. It is also important to point out that when the array sizes are small, the runtime of the kernels is also small and there is thus less need for an auto-tuning strategy.

### 8.9 Summary

In summary, of the two heuristics presented in Chapter 6, the group-by-dimensions heuristic generally has the most favourable trade-off between performance and time. It provides a significant improvement over both **ExpS** and **Rand** in all metrics considered. The machine learning model has reasonable absolute accuracy and achieves most of the speedup of the oracle, indicating that when it does mispredict, those mispredictions are not costly. Although the strategy does not require expert knowledge, it can be improved by incorporating such knowledge to impose further restrictions on the heuristic’s search space. It can be further improved by allowing the work-group size to be left as a variable, queried using the OpenCL API, when the machine learning model predicts that either of the global memory techniques is best. Finally, the strategy also works on larger array sizes and on asymmetric array shapes.
Chapter 9

Related Work

There is a large body of work dealing with automatic performance tuning of stencil codes on both multicores and GPUs. In this chapter we review the more salient work.

Datta et al. [9] tune 7-point (star with radius 1) and 27-point (dense with radius 1) 3D stencils on many platforms, including GPUs, and consider a similar optimization space to that used in this work. Specifically, they tune the parameters of a multi-tiered data decomposition which is analogous to the work-group size and the thread merging factors studied here.

Kamil et al. [19, 20] use mostly the same optimization space as Datta et al. while focusing on four different stencils ranging from 1D to 3D from the fields of PDE solvers and image processing. They do this on a variety of CPU platforms.

Mametjanov et al. [25] work with seven different stencil kernels from the PETSc parallel numerical toolkit that perform matrix and vector operations necessary for Newton-Krylov methods. They tune CUDA implementations of these kernels for two different GPUs. The optimizations they consider consist of the launch configuration, the number of kernel streams launched (so as to allow overlap of asynchronous kernel launches and asynchronous data transfer- an optimization not considered here), the L1/shared memory cache split (not an option in OpenCL, see Section 2.1.1), and various compiler flags.

Zhang and Mueller [44] tune four stencil kernels: 7-point, 27-point, 13-point (star with radius 2), and 19-point (no-corners with radius 1) for execution on GPU clusters. Their multi-node optimizations are not relevant to this work, but they do consider some single-node optimizations: specifically the launch configuration and whether or not to use texture memory.

Finally Holewinski, Pouchet, and Sadayappan [17] tune the work done per thread, the launch configuration, and the time-tile size of a variety of kernels, some synthetic and some from the domains of PDE solvers, image processing, and gradient descent.
Some of the above authors created their kernels by hand or focused on applications for which kernels already existed [9, 25] while others started from a higher-level description or from sequential code and included compiler support to produce the kernels [19, 20, 44].

Common to all of the work described above is how they search their optimization space. They either perform exhaustive search directly [17] or use expert knowledge to first limit the optimization space and then perform exhaustive search on that subset of the space [9, 19, 20, 25, 44].

In contrast, the work shown here does not require expert knowledge of the platform or the application as it does not limit which values are searched for each optimization, but rather which optimizations are searched concurrently, a limitation that is more likely to work from application to application or from platform to platform. Furthermore, it was shown in Section 8.6 that incorporating expert knowledge into the strategy can improve performance even further.

There has been other work that used more intelligent auto-tuning strategies than the expert search approach. Tang et al. [39] develop an analytical model that approximates the performance impact of different launch configurations and thread merge factors on stencil kernels that have been optimized to use local memory using their in-plane loading technique. Meng and Skadron [26] work on stencils that have been tiled in the time loop and develop an analytic model that maps ghost zone size (zones of replicated computation used to minimize synchronization) to performance. They then use gradient descent on this model to determine the optimal size for a given stencil. Grosser et al. [14] develop an algorithm to determine the best way to split tile time-loop-containing stencils. Finally, Han and Abdelrahman [16] develop a machine learning model that predicts whether or not an array should be cached in local memory for an arbitrary input kernel. However, all of this work (with the exception of Tang’s [39]) focuses exclusively on a single optimization. In contrast, we consider multiple optimizations concurrently and the strategy developed takes into account the interactions amongst the optimizations.

One common theme with all of the works mentioned so far is the focus on a handful of applications. In most of the above studies only 2-12 stencil kernels were considered. In contrast, we use a large number of programs that were synthetically generated to have a wide range of stencil patterns and sizes. In fact, the synthetic kernels looked at in this work contain almost every stencil pattern analyzed in previous stencil auto-tuning work. The exception to this is Han’s work that also makes use of synthetic kernels in order to explore a large number of programs.

Some auto-tuning works have made up for this deficiency though by considering different inputs, which essentially expands the program space in an alternative manner. For
example, Bergstra, Pinto, and Cox [3] do this while developing machine learning models for auto-tuning of filterbank correlation kernels while Magni, Grewe, and Johnson [24] do it while creating a model to determine the best way to map sequential OpenACC loops to a launch configuration.

There are also some auto-tuning approaches that use statistical and machine learning models to partition and prune a large optimization configuration space similar to our approach. Jia, Shaw, and Martonosi use recursive partitioning regression trees to partition GPU hardware/software design spaces [18] while Ganapathi et al. use kernel canonical correlation analysis to partition the optimization configuration search space for 7-point and 27-point stencils [11]. These approaches both require sampling the optimization configuration space in order to perform their pruning. Thus their data sets come from different optimization configurations tested on the program of interest. In contrast, in our work the machine learning model is trained offline on kernels other than the one being tuned. Only static program features are used as inputs to the machine learning model and thus no runs of the program are needed to use the machine learning model and to accomplish the initial optimization space pruning. We also restrict the machine learning model to predict the data loading technique rather than our entire set of optimizations.

There is also a significant amount of prior work developing auto-tuning frameworks that can be used to auto-tune stencil computations as well as other GPU kernels. Christen, Schenk, and Burkhart [8] present a framework that provides compiler support to convert a high-level representation of a stencil into optimized and auto-tuned code. They don’t go into great detail about their tuning techniques, as their focus is on the code generation, but they mention that their auto-tuner is easily extendable with new search strategies. Ansel et al. [1] on the other hand developed a generic framework for creating auto-tuners, known as OpenTuner, whose emphasis is on having strong tuning algorithms in the back-end. The work presented in this thesis complements these frameworks. Indeed, the heuristics presented here could be implemented as strategies within them. OpenTuner is particularly well suited for this as it allows for the evaluation of multiple auto-tuning strategies simultaneously with priority dynamically reallocated during auto-tuning to those strategies that are finding better configurations.

Finally, due to the difficult-to-optimize nature of GPGPU kernels, there has been extensive work on auto-tuning of kernels other than stencils [7, 13, 22, 27]. These works are too numerous to list exhaustively, but the most relevant ones have been highlighted in the earlier paragraphs. In general, the size of this field demonstrates the importance of auto-tuning in extracting performance on GPU platforms.
Chapter 10

Conclusions

This thesis described an optimization space for stencil kernels running on GPUs. It showed the interestingness of that optimization space and demonstrated that simple techniques were not sufficient to find high-performing optimization configurations in that space.

It presented a strategy for automatic performance tuning of stencil kernels. This strategy uses machine learning to determine the best approach to using memory and then explores the remaining optimizations in groups. Two possible ways of grouping the optimizations so as to reduce inter-group dependencies were presented.

A total of 104 OpenCL kernels were synthetically generated and auto-tuning infrastructure was developed to evaluate the strategy. This evaluation considered both the duration of the auto-tuning and the quality of the best configuration obtained. The best heuristic was shown to achieve a reduction in the number of configurations explored relative to random sampling (by 71%) and expert search (by 84%) strategies while also finding better performing configurations, by 12% and 6% respectively.

While the strategy presented does not require expert knowledge of the optimizations or the kernel on the part of the user, it was shown that by using such knowledge, these results could be improved even further. The results could further be improved by allowing the work-group size to be a variable when the machine learning model predicts that global memory is optimal. Finally, the strategy was shown to be relatively indifferent to the input and output array sizes and shapes given that those sizes remain above a certain threshold below which the optimization space loses its interestingness.
10.1 Future Work

There are a number of possible directions for future work in this area. First, from a practical perspective, incorporating the strategy presented into an existing auto-tuning framework, as discussed in Chapter 9, will allow it to be easily used on new stencil kernels. Such a framework requires compiler support for the optimizations used in this work so, if not available, that compiler support is also a necessary extension.

One of the strengths of a machine learning approach to auto-tuning is that it requires little work to port to new platforms. As such, one obvious extension to this work is to test the performance of the strategy on other GPUs. Both other Nvidia GPUs with similar architectures as well as GPUs from other vendors with significantly different architectures will be interesting to explore.

Another direction of possible future work is to address some of the limitations of this work. One main limitation was the omission of optimizations that target the time loop. While the intuitive way to program stencils in OpenCL is to leave the time loop on the host due to the fact that it carries data dependencies, there has been work on tiling this loop [14, 26], which could potentially be beneficial. Another extension to this work that uses tiling of dependence-carrying loops would be to remove the restriction that the input and output arrays are distinct. Such an extension would allow the strategy to be used on a broader range of stencil kernels including ones that don’t seem easily amenable to GPUs. Another limitation of this work is the fact that all of the stencils performed the same amount of computation per input element. Specifically, every kernel performed a load, a multiplication, and an addition for each input element. Varying the computational intensity of the stencils would result in a broader range of kernels that could be more representative of stencil codes used in practice. This could result in different optimization performance. While one would expect that the heuristics would still perform well on such kernels due to their online tuning, the machine learning models would almost certainly require improvement through the addition of one or more new features to capture the amount of computation in a kernel.

This work can also be extended via improvements to the auto-tuning strategy. Different random forest parameters and model inputs can be considered or even different machine learning models entirely. New heuristics can also be explored to group the optimizations in different ways. Finally, techniques from the broader field of global optimization, of which this problem is a member, can be applied to this problem. For example, simulated annealing might be well suited to this challenge.
Finally, the results of Section 8.7 suggest a tangential line of future work. That section demonstrated that for some kernels a huge performance increase could be realized if the work-group size is a constant at compile-time. This suggests that for some OpenCL kernels there might be value in a just-in-time compilation at kernel launch that implements constant propagation and constant folding of launch-time constants such as the launch configuration.
Appendix A

Optimization Space Size

This appendix outlines the calculations to determine the optimization space size. There are two optimization space sizes mentioned in this document: the full space of 50,094,000 and the reduced space of 475,875. The calculations for these two numbers are presented in Sections A.1 and A.2 respectively.

A.1 Full Space

The optimizations and their possible values are reproduced in Table A.1. As was mentioned in Section 4.7, there are a number of restrictions placed on the possible values for each optimization that depend on the values of other optimizations. Namely, the product of W, B, and C for a given dimension must be less than or equal to N in that dimension. In addition, VX must be less than or equal to BX.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Abbreviation</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Work-group size</td>
<td>WX, WY, WZ</td>
<td>1 NX, NY, NZ</td>
</tr>
<tr>
<td>Block merge factor</td>
<td>BX, BY, BZ</td>
<td>1 NX, NY, NZ</td>
</tr>
<tr>
<td>Cyclic merge factor</td>
<td>CX, CY, CZ</td>
<td>1 NX, NY, NZ</td>
</tr>
<tr>
<td>Vector width</td>
<td>VX</td>
<td>1 16</td>
</tr>
<tr>
<td>Local memory</td>
<td>N/A</td>
<td>0 (no) 1 (yes)</td>
</tr>
<tr>
<td>Image memory</td>
<td>N/A</td>
<td>0 (no) 1 (yes)</td>
</tr>
</tbody>
</table>

Table A.1: Optimization abbreviations and allowable values. N denotes the input/output array size in each dimension. X, Y, and Z correspond to the x, y, and z dimensions of the work-item grid respectively.
To determine the number of possible optimization configurations we first look at each dimension independently as the values for the optimizations in one dimension have no impact on the possible values for other dimensions. To make the math easier, rather than working with the three optimizations W, B, and C, we map these three variables to three new variables $T_1$, $T_2$, and $T_3$ according to Equations A.1, A.2, and A.3.

\[ T_1 = B \]  \hspace{1cm} (A.1)

\[ T_2 = BW \]  \hspace{1cm} (A.2)

\[ T_3 = BWC \]  \hspace{1cm} (A.3)

For this new representation to be useful to us, we require that every unique configuration in one space corresponds to a unique configuration in the other. In other words, the transformation from the old representation to the new representation must be a one-to-one function. This can be demonstrated with some simple algebra by showing that these three equations allow us to uniquely solve for the original variables as shown in Equations A.4, A.5, and A.6.

\[ B = T_1 \]  \hspace{1cm} (A.4)

\[ W = \frac{T_2}{T_1} \]  \hspace{1cm} (A.5)

\[ C = \frac{T_3}{T_2} \]  \hspace{1cm} (A.6)

This new representation has a useful property that the previous one did not; the three variables obey the inequality shown in Equation A.7 (since $B$, $W$, and $C \geq 1$).

\[ T_1 \leq T_2 \leq T_3 \]  \hspace{1cm} (A.7)

With this useful property, the number of configurations of the three variables can be calculated as a simple application of the formula for combinations with repetition shown in Equation A.8 where $n$ is the number of possible values and $r$ is the number of selections (in our case three). Effectively one can imagine that three values are chosen from the full range of allowable values and they are then ordered in whatever way necessary so as to satisfy Equation A.7. As such, the order of the selections does not matter.
\[ \binom{n + r - 1}{r} \]  \hspace{1cm} (A.8)

This formula is sufficient for calculating the possible configurations in the \(y\) and \(z\) dimensions. The \(x\) dimension is a little more complicated due to \(V_X\). We know though, by Equation A.1, that since \(B_X\) must be greater than or equal to \(V_X\) then \(T_1\) must be greater than or equal to \(V_X\) as well. And by Equation A.7, \(T_2\) and \(T_3\) must be greater than or equal to \(V_X\) as well. This makes it easy to calculate all possible values of \(T_1\), \(T_2\), and \(T_3\) as a function of the value of \(V_X\). Namely, as \(V_X\) increases, the value of \(n\) in Equation A.8 decreases. Summing these over the five possible values of \(V_X\) (1, 2, 4, 8, and 16) gives us Equation A.9 where \(M\) is the number of possible values for each of \(B\), \(W\), and \(C\). For a value of \(N = 256\) as was used in this work, \(M = 9\) as there are 9 powers-of-two between one and 256 ((\(\log_2 256\)) + 1 = 9).

\[ \sum_{i=0}^{4} \binom{M + 2 - i}{3} \]  \hspace{1cm} (A.9)

If we combine these expressions for the three dimensions as well as multiplying by a factor of 4 for every combination of local and image memory we get the final result shown in Equation A.10. Evaluating this formula for \(M = 9\) results in a value of 50,094,000 configurations as mentioned in Section 4.7.

\[ 4 \binom{M + 2}{3}^2 \sum_{i=0}^{4} \binom{M + 2 - i}{3} \]  \hspace{1cm} (A.10)

### A.2 Reduced Space

The actual optimization space used in the evaluation includes additional restrictions listed in Section 5.1. Specifically, the following three restrictions are applied:

1. The block merging factor in the \(x\) dimension is restricted to be equal to the vectorization factor.
2. Block merging in the \(y\) and \(z\) dimensions is not allowed.
3. Only one of the following four forms of data loading is allowed: global memory without vectorization, global memory with vectorization, local memory, or image memory.

To calculate the number of optimization configurations in this reduced space, the number of optimizations for some of the data loading techniques are calculated separately.
The global memory without vectorization, local memory, and image memory techniques each account for the same number of optimization configurations. Global memory with vectorization on the other hand accounts for more configurations due to the additional parameter VX/BX.

As with the previous section, we use the new space represented by the T variables to simplify the math. For the non-vectorization data loading techniques, this corresponds to $T_1$ being fixed to 1 and then choosing $T_2$ and $T_3$ as before. Thus we can use Equation A.8 with $r = 2$ instead of 3. Since all dimensions are equivalent for these data loading techniques, the number of configurations for each is thus simply the cube of this value as shown in Equation A.11.

$$\left( \frac{n + 2 - 1}{2} \right)^3$$

(A.11)

Computing the vectorization configurations is a little more complicated. BX and VX are forced to be equal and VX must be between 2 and 16. We can modify Equation A.9 to remove the term corresponding to $VX = 1$ and to reduce the number of variables as we did to Equation A.8. This results in Equation A.12 for the number of possible configurations in the $x$ dimension.

$$\sum_{i=1}^{4} \left( \frac{M + 1 - i}{2} \right)$$

(A.12)

Multiplying this result by the possible configurations for the $y$ and $z$ dimensions and adding in the configurations from the non-vectorization data loading techniques from Equation A.11 results in Equation A.13.

$$\left( \frac{M + 2 - 1}{2} \right)^2 \sum_{i=1}^{4} \left( \frac{M + 1 - i}{2} \right) + 3 \left( \frac{M + 2 - 1}{2} \right)^3$$

(A.13)

Computing this for $M = 9$ results in 475,875 configurations as stated in Section 5.1.
Bibliography


International Parallel & Distributed Processing Symposium, pages 1–12, 2009.

[42] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU Compiler for Memory Opti-
Conference on Programming Language Design and Implementation, pages 86–97,
2010.


on GPU Clusters. In Proceedings of the Tenth International Symposium on Code