Automatic Optimization of Thread-Coarsening for Graphics Processors

Alberto Magni
School of Informatics
University of Edinburgh
United Kingdom
a.magni@sms.ed.ac.uk

Christophe Dubach
School of Informatics
University of Edinburgh
United Kingdom
christophe.dubach@ed.ac.uk

Michael O'Boyle
School of Informatics
University of Edinburgh
United Kingdom
mob@inf.ed.ac.uk

ABSTRACT

OpenCL has been designed to achieve functional portability across multi-core devices from different vendors. However, the lack of a single cross-target optimizing compiler severely limits performance portability of OpenCL programs. Programmers need to manually tune applications for each specific device, preventing effective portability. We target a compiler transformation specific for data-parallel languages: *thread-coarsening* and show it can improve performance across different GPU devices. We then address the problem of selecting the best value for the coarsening factor parameter, *i.e.,* deciding how many threads to merge together. We experimentally show that this is a hard problem to solve: good configurations are difficult to find and naive coarsening in fact leads to substantial slowdowns. We propose a solution based on a machine-learning model that predicts the best coarsening factor using kernel-function static features. The model automatically specializes to the different architectures considered. We evaluate our approach on 17 benchmarks on four devices: two Nvidia GPUs and two different generations of AMD GPUs. Using our technique, we achieve speedups between 1.11× and 1.33× on average.

1. INTRODUCTION

Graphical Processing Units (GPUs) are widely used for high performance computing. They provide cost-effective parallelism for a wide range of applications. The success of these devices has lead to the introduction of a diverse range of architectures from many hardware manufacturers. This has created the need for a common programming language to harness the available parallelism in a portable way. OpenCL is an industry-standard language for GPUs that offers program portability across accelerators of different vendors: a single piece of OpenCL code is guaranteed to be executable on many diverse devices.

A uniform language specification, however, still requires programmers to manually optimize kernel code to improve performance on each target architecture. This is a tedious process, which requires knowledge of hardware behavior, and must be repeated each time the hardware is updated. This problem is particularly acute for GPUs which undergo rapid hardware evolution.

The solution to this problem is a cross-architectural optimizer capable of achieving performance portability. Current proposals for cross-architectural compiler support [21, 34] all involve working on source-to-source transformations. Compiler intermediate representations [6] and ISAs [5] that span across devices of different vendors have still to reach full support.

This paper studies the issue of performance portability focusing on the optimization of the *thread-coarsening* compiler transformation. Thread coarsening [21, 30, 31] merges together two or more parallel threads, increasing the amount of work performed by a single thread, and reducing the total number of threads instantiated. Selecting the best coarsening factor, *i.e.,* the number of threads to merge together, is a trade-off between exploiting thread-level parallelism and avoiding execution of redundant instructions. Making the correct choice leads to significant speedups on all our platforms. Our data show that picking the optimal coarsening factor is difficult since most configurations lead to performance downgrade and only careful selection of the coarsening factor gives improvements. Selecting the best parameter requires knowledge of the particular hardware platform, *i.e.,* different GPUs have different optimal factors

In this work we select the coarsening factor using an automated machine learning technique. We build our model based on a cascade of neural networks that decide whether it is beneficial to apply coarsening. The inputs to the model are static code features extracted from the parallel OpenCL code. These features include, among the others, branch divergence and instruction mix information. The technique is applied to four GPU architectures: *Fermi* and *Kepler* from Nvidia and *Cypress* and *Tahiti* from AMD. While naive coarsening misses optimization opportunities, our approach gives an average performance improvement of 1.16x, 1.11x, 1.33x, 1.30x respectively.

In summary the paper makes the following contributions:

- We provide a characterization of the optimization space across four architectures.
- We develop a machine learning technique based on a neural network to predict coarsening.
- We show significant performance improvements across 17 benchmarks

.

The remainder of the paper is organized as follows. Section 2.1 provides a brief overview of OpenCL and describes the thread coarsening transformation. A motivating example for the problem is provided in section 3. Section 4 describes the experimental setup and the compiler infrastructure. Section 5 presents a characterization of the optimization space. Section 6 describes the machine the machine learning model. The results are presented in Section 7. Section 8 presents the related work and section 9 concludes the paper.

2. BACKGROUND

2.1 OpenCL

OpenCL is a cross-vendor programming language used for massively parallel multi-core graphic processors. OpenCL kernel functions define the operations carried out by each data-parallel hardware-thread. Kernels are compiled at runtime on the *host* (a standard CPU) and sent to execution onto the *device* (in our case a GPU). It is the programmer's responsibility to decide how many threads to instantiate and how to arrange them into *work-groups*, defining the so-called *NDRange* of *thread-ids*. Each *thread-id* corresponds to thread executing within a work-group. Work-items in a work-group are guaranteed to be scheduled onto a single core and can share data in local memory and synchronize using barriers.

OpenCL programmers, usually, instantiate as many threads as they have data element to process. This results in hundreds of thousands of threads being scheduled on the target device, maximizing the degree of parallelism. However, there is no guarantee that this policy is the best for each possible target device. Consider for instance the presence of thread-invariant instructions [18, 33] which produces the same results independently from the thread executing them. Maximizing the number of threads might increase the total number of redundant operations executed by the GPU.

2.2 Thread Coarsening

One way to prevent the problems associated with having too fine-grained parallelism is to merge threads together. This reduces the redundancy in the number of integer operations and branches performed per floating point operation. This transformation, known as *thread coarsening* [21, 30, 31], works by fusing multiple work items together increasing the amount of work performed by a single thread and reducing the overall number of launched threads. While it appears to be an easy task for the programmer, it is preferable to apply this transformation automatically using a compiler for two reasons: (1) the merging is done by replicating instructions inside the kernel, which is error-prone to do by hand; (2) the *Coarsening Factor* (*i.e.,* the number of times the body of the thread is replicated) which gives the best performance depends on the program and on the hardware.

The coarsening transformation we implemented increases the amount of work performed by the kernel by replicating the instructions in its body. Leveraging divergence analysis [9, 15, 18] only instructions that depend on the *thread-id* (divergent instructions) are replicated. This reduces the overhead of executing uniform instructions across multiple threads. Divergent instructions are replicated and inserted right after the original ones updating the *thread-id* to work

```
1 kernel void square(global float* in, out) {
2   int gid = get_global_id(0);
3   out[gid] = in[gid]*in[gid]; }
```

(a) Original code

```
1 kernel void square2x(global float* in, out) {
2   int gid = get_global_id(0);
3   int tid0 = 2*gid+0;
4   int tid1 = 2*gid+1;
5   out[tid0] = in[tid0]*in[tid0];
6   out[tid1] = in[tid1]*in[tid1]; }
```

(b) After coarsening with factor 2

Figure 1. Vector square example in OpenCL. In the original code each thread takes the square of one element of the input array a. When coarsened by a factor two b , each thread now processes two elements of the input array.

on the correct data elements. Finally, the number of instantiated thread is reduced at runtime.

Figure 1 shows a kernel that calculates the square of the element of the input array. In the original code, each thread is responsible for squaring one element of the input array. After coarsening has been applied with factor two, each thread processes two element of the input array. As can be see on lines 3–4 of figure 1b, the index in the input array are calculated from the original thread iteration space as returned by the OpenCL *get_global_id(0)* function. Prior work on this compiler transformation ([21]) presents how to effectively manage branches and loops.

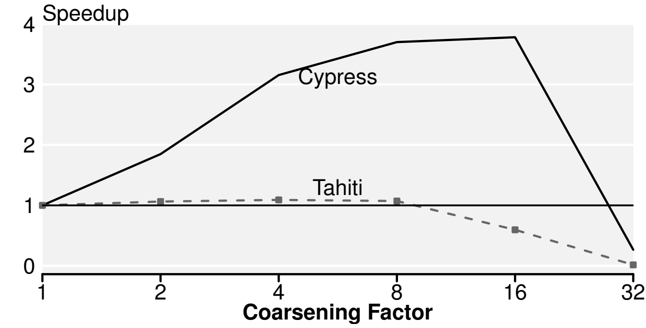
3. MOTIVATION

This section motivates the work explaining the importance of determining whether to coarsen, and by how much. Figure 2 shows the speedup given by thread-coarsening over the baseline (no coarsening) as a function of the corsening factor for two benchmarks on different GPUs.

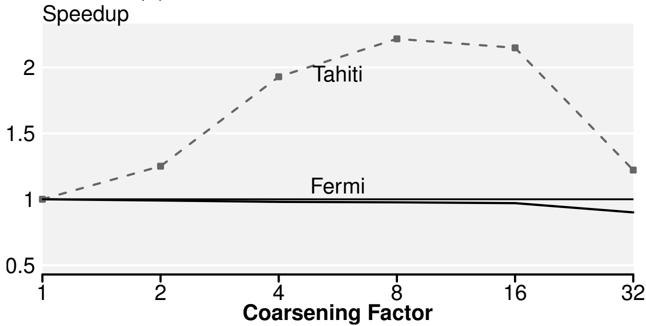
The first figure 2a shows the performance of *NBody* on two AMD architectures: *Cypress* and *Tahiti*. Even though these two devices are from the same vendor, they have significant architectural differences [3]. Cypress cores are VLIW while Tahiti cores have processing elements working in SIMD fashion. Such differences are reflected in the different reactions shown by the benchmark to the coarsening transformation. The higher degree of ILP made available by coarsening in the *NBody* benchmark is successfully exploited by the VLIW cores of *Cypress* ensuring a significant speedup. The same does not apply to *Tahiti* where no improvement can be obtained and we record a significant slowdown for higher factors.

Differences across architectures also emerge for *BinarySearch*. Here, we make a comparison between the AMD *Tahiti* and the Nvidia *Fermi* architectures. On the first device *BinarySearch* ensures an improvement of 2.3x over the default for coarsening factor 8. On Nvidia, however, no improvement can be obtained.

From these simple examples it is clear that optimizing the coarsening pass is an important problem. In addition, the coarsening factor must be determined on a per-program and per-platform basis. Classical hand-written heuristics



(a) *NBody* run on *Cypress* and *Tahiti*



(b) *BinarySearch* run on *Fermi* and *Tahiti*

Figure 2. Speedup achieved with thread coarsening as a function of the coarsening factor for *NBody* and *BinarySearch*

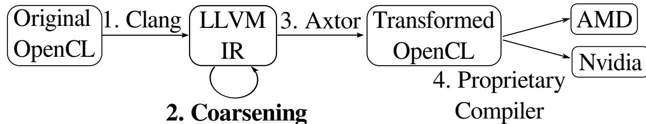


Figure 3. Compiler toolchain

are not only complex to develop, but are likely to fail due to the variety of programs and ever-changing OpenCL devices available.

4. EXPERIMENTAL SETUP

This section describes the compiler toolchain (4.1), the coarsening stride (4.2), the benchmarks (4.3) and the devices used in the experiments (4.4).

4.1 Compiler Toolchain

One way to automatically evaluate a compiler transformation in OpenCL on multiple devices is to use a source-to-source compiler. It avoids the need to have access to potentially properietary compiler internals. Alternative approaches based on SPIR [6] could also be used in the future, when it is going to reach wide adoption. Figure 3 shows our compilation infrastructure. The OpenCL C code of the target kernel is translated to LLVM bitcode using the *clang* open-source front-end (stage 1). The coarsening transformation is applied at the LLVM bitcode level (stage 2). The transformed LLVM program is then translated back to OpenCL C using the *axtor* [24] LLVM OpenCL-backend (stage 3). The transformed OpenCL program is then fed into the hardware vendor proprietary compiler (stage 4) for code generation.

	Program name	Source	Num. Threads
1)	binarySearch	AMD SDK	268M
2)	blackscholes	Nvidia SDK	16M
3)	convolution	AMD SDK	6K
4)	dwtHaar1D	AMD SDK	2M
5)	fastWalsh	AMD SDK	33M
6)	floydWarshall	AMD SDK	(8K x 8K)
7)	mriQ	Parboil	524K
8)	mt	AMD SDK	(4K x 4K)
9)	mtLocal	AMD SDK	(4K x 4K)
10)	mvCoal	Nvidia SDK	16K
11)	mvUncoal	Nvidia SDK	16K
12)	nbody	AMD SDK	131K
13)	reduce	AMD SDK	67M
14)	sgemm	Parboil	(3K x 3K)
15)	sobel	AMD SDK	(1K x 1K)
16)	spmv	Parboil	262K
17)	stencil	Parboil	(3K x 510 x 62)

Table 1. OpenCL applications with the reference number of threads.

4.2 Coarsening Factor

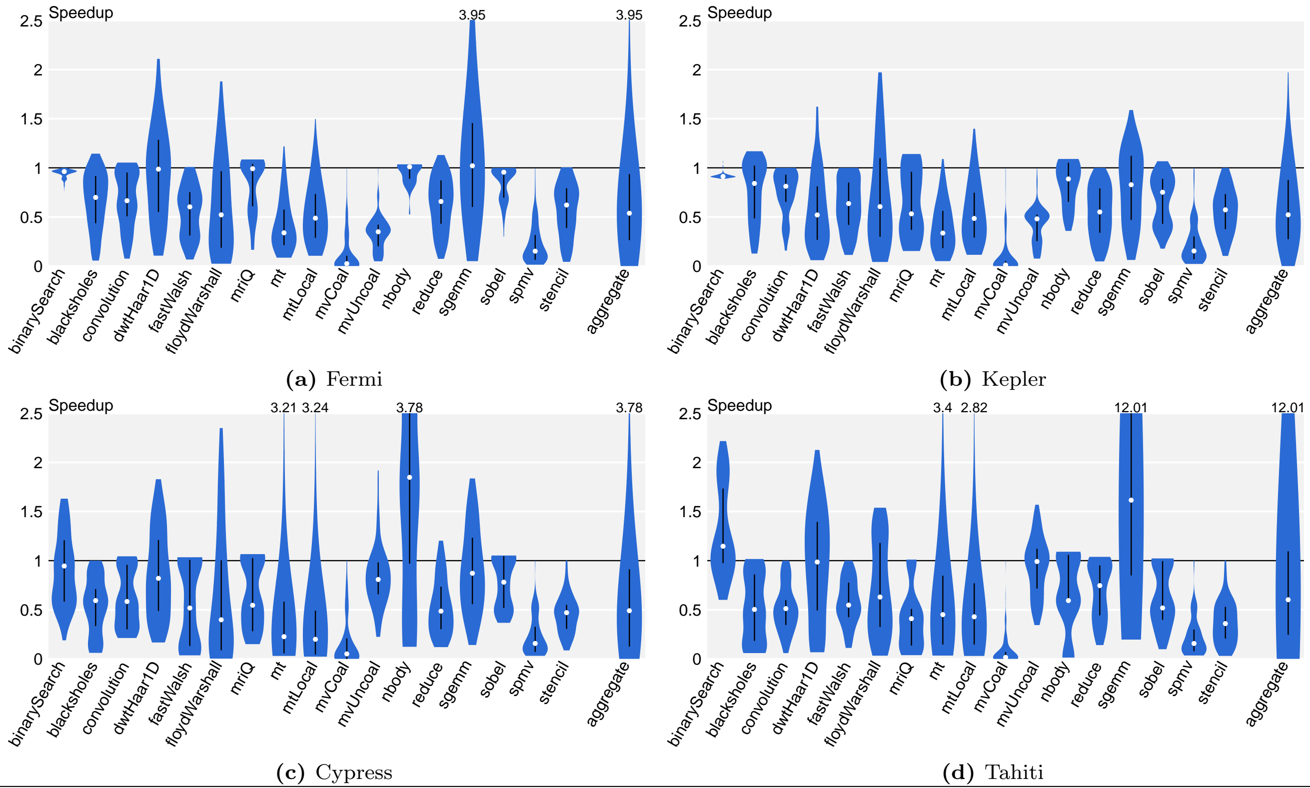
In this work we address the tuning of the *coarsening factor* parameter. The first one controls how many threads to merge together. For our experiments we allowed this parameter to have the following values: $[1, 2, 4, 8, 16, 32]$. Where coarsening factor of 1 corresponds to the original program. We limited our evaluations to powers of two since these are the most widely applicable parameter values, given the reference input sizes for the considered benchmarks. The tuning technique in the next chapters can be easily extended to support non powers of two.

4.3 Benchmarks

We have used 17 benchmarks from various sources as shown in table 1. In the case of the *Parboil* benchmarks we used the *opencl_base* version. The table also shows the global size (total number of threads) used. The baseline performance reported in the results section is that of the best work group size as opposed to the default one chose by the programmer. This is necessary to give a fair comparison across platforms when the original benchmark is written specifically for one platform. Each experiment has been repeated 50 times aggregating the results using the median. Previous work ([22]) has shown that the choice for the optimal coarsening factor is consistent across input sizes. For this reason we restricted our analysis to the input size (*i.e.*, total number of threads running) for the uncoarsened kernel listed in table 1.

4.4 Devices

For our experiments we used four devices from two vendors. From Nvidia we used GTX480 and Tesla K20c. The first one is based on the Fermi architecture [2] with 15 streaming multiprocessors (SM) and 32 CUDA cores for each. The Tesla K20c has the latest Kepler architecture [4]. K20c has 13 SMs each with 192 CUDA cores.
From AMD we used the Radeon HD 5900 and Tahiti 7970. The first one, codenamed Cypress, has 20 SIMD cores each with 16 thread processors, a 5-way VLIW core. The Graphics Core Next architecture in Tahiti 7970 is a radical change



(a) Fermi **(b)** Kepler **(c)** Cypress **(d)** Tahiti

Figure 4. Violin plots showing the distribution of speedups for all the benchmarks on the four devices. The shape of the violin corresponds to the speedup distribution. They indicate on how hard it is to tune a program. The thick black line shows where 50% of the data lies. The white dot is the position of the median.

Name	Model	GPU Driver	OpenCL version	Linux kernel
Fermi	Nvidia GTX 480	304.54	1.1 CUDA 5.0.1	3.2.0
Kepler	Nvidia K20c	331.20	1.1 CUDA 5.0.1	3.7.10
Cypress	AMD HD5900	1124.2	1.2 SDK 1124.2	3.1.10
Tahiti	AMD HD7970	1084.4	1.2 SDK 1084.4	3.1.10

Table 2. OpenCL devices used for our experiments.

for AMD. Each of the 32 computing cores contains 4 vector units (of 16 lanes each) operating in SIMD mode. This exploits the advantages of dynamic scheduling as opposed to the static scheduling needed by VLIW cores.

5. OPTIMIZATION SPACE CHARACTERISTICS

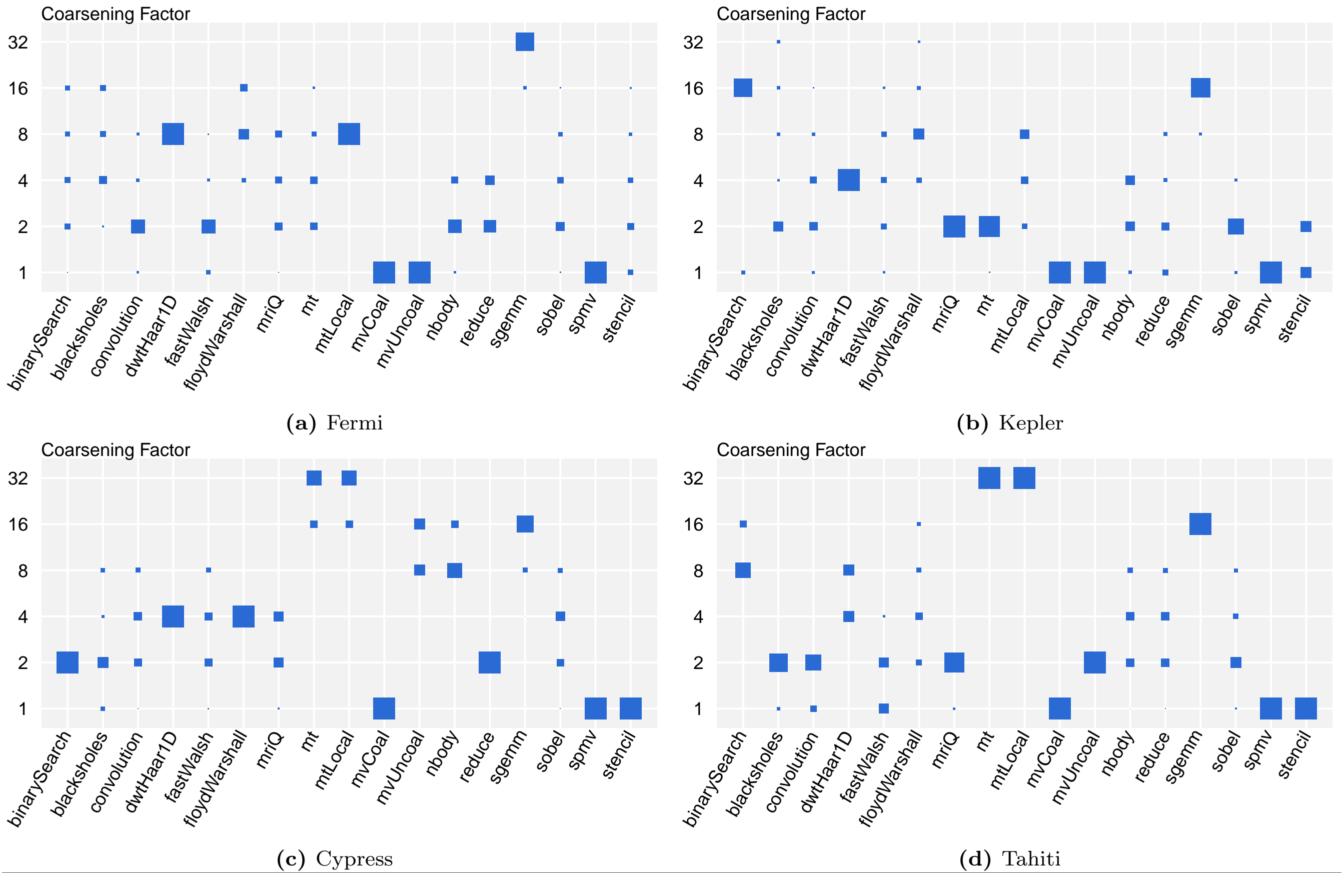
In the motivating example, in figure 2, we showed that different applications behave differently with respect to coarsening on different devices. In this section we expand on this topic and present the overall optimization space.

5.1 Distribution of Speedup

The violin plots in figure 4 show the distribution of the speedups (over no coarsening) achievable by changing the coarsening factor and the local work group size. The width of each violin corresponds to the proportions of configurations with a certain speedup. The white dot denotes the median speedup, while the thick black line shows where 50% of the data lies. These plots are effective in highlighting differences and similarities in spaces of different applications. Intuitively, violins with a pointy top correspond to benchmarks difficult to optimize: configurations giving the maximum performance are few. For example consider *mt* and *mtLocal* (matrix multiplication and matrix multiplication with local memory). These two program show similar shapes among the two Nvidia and AMD devices. On the other hand *nbody* shows different shapes among the four architectures: a large improvement is available on *Cypress* while no performance gain can be obtained on the other three devices. Interesting cases for analysis are benchmarks such as *mvCoal* and *spmv*. They have an extremely pointy top with the whole violin lying below 1, meaning no performance improvement is possible on these benchmarks with coarsening. Consider now the aggregate violin in the final column of each subfigure. This shows the distribution of speedups across all programs. In all the four cases the white dots lies well below the 1, near 0.5 in most cases. This means that on average, the coarsening transformation will slow programs down. This fact coupled with the different optimization distributions shows how difficult choosing the right coarsening factor is. This is probably the reason why there is little prior work in determining the right coarsening factor. The model presented in section 6 tackles this problem of selecting an appropriate factor for each program.

5.2 Coarsening Factor Characterization

The objective of this work is the determination of the best thread-coarsening factor. In this section we quantify how



(a) Fermi **(b)** Kepler **(c)** Cypress **(d)** Tahiti

Figure 5. Hinton diagram showing the percentage of configurations with the given coarsening factor in the performing top 5% configurations. Intuitively, the larger the square is the more likely it is for the given coarsening factor to perform close to the optimum.

this parameter varies across program and platform. Figure 5 shows a Hinton diagram for each target device. For each benchmark and coarsening factor it plots a square. The size of the square is proportional to the percentage of configurations having the given coarsening factor among the top 5% performing ones. Intuitively: the larger the square is, the more likely it is that the corresponding coarsening factor will perform well. From these plots it is immediately clear that no single factor can give good performance for all the programs. On the two Nvidia devices (*Fermi* and *Kepler*) the optimal coarsening factor tends to be among 1, 2, or 4, with few exceptions. On the two AMD architectures (*Tahiti* and *Cypress*) instead the best coarsening factor are often larger (4 or 8).
In section 7 we provide an evaluation of the improvement that can be obtained using a single coarsening factor for all the programs. We compare our prediction model against this baseline.

5.3 Cross-architecture optimization portability

In this section we investigate if the knowledge of the best coarsening factor for one architecture can be successfully exploited to a new platform. This study fulfills two purposes: (1) it characterizes the diversity of the devices that are evaluated in this work, (2) it studies how successful would an optimizer tuned for a given platform be when ported to another one. For this task we explored all coarsening factors on all the kernel and devices selecting the best for each. Given two devices A and B we then evaluate for each kernel the performance of the best coarsening factor of A (*Search-Device*) on B (*Target-Device*). Figure 6 reports the results of this analysis for all combinations of A and B.

From these plots we can see, for example, that performance transfers effectively from *Kepler* to *Fermi* (where we reach 77% of the maximum speedup) but, interestingly, the contrary is not true (where only 47% is attainable). Its important to notice that porting performance across Nvidia and AMD leads to bad performance on average. For example the best configurations from *Fermi* and *Kepler* give 28% and 41% of the maximum on *Cypress*. Similarly bad performance is given when porting from *Cypress* and *Tahiti* to *Kepler*.

Consider that these results represent the upper bound of how an optimizer specialized for one architecture would behave when applied to another one. Any realistic compiler heuristic when applied to an unseen program, would perform considerably worse than this. This shows that performance portability is not guaranteed when going across vendors, even if we have optimized the same program on a previous device This justifies the need to specialize the choice of the coarsening configuration on a per-platform basis relying on a flexible optimizer.

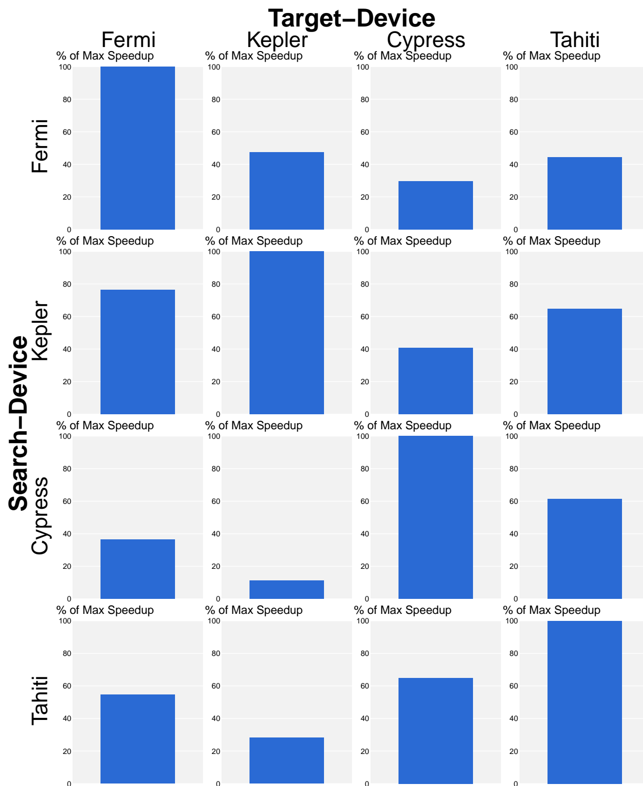


Figure 6. Each barplot reports the percentage of the maximum performance of *Target-Device* (columns) is obtained when evaluating the best configuration found searching on *Search-Device* (rows). When *Target-Device* and *Search-Device* are the same the bar reached 100% by construction. For example, the *Fermi-Cypress* subplot (on the top-left) shows that 37% of the maximum performance of *Cypress* is achievable when using the best configuration coming from *Fermi*.

6. PREDICTIVE MODEL

The goal of the model is to decide the optimal coarsening factor for each kernel on a specific architecture. The model works using static program features extracted at compile time. Relying on these features, the model makes a binary decision on whether to coarsen or not. By iteratively querying the model we can decide when to stop coarsening. We train a new model for each architecture: four in total.

6.1 Training

This section describes how we generate training data.

We use Leave One Out Cross Validation. This standard machine learning technique works selecting one benchmark for testing and using the remaining ones for training. This means we train our model on 16 benchmarks and apply it to the testing program. We repeat this process 17 times, one for each test case.

Figure 7 gives a visual representation of the training process. OpenCL kernels from the training set are compiled and run with the six different coarsening factors. During the compilation step (Phase 1) a compiler-analysis pass collects static features of each different version of the code. The full list of these features is given in section 6.3. In Phase 2 we run all the different versions of the program arranging the results

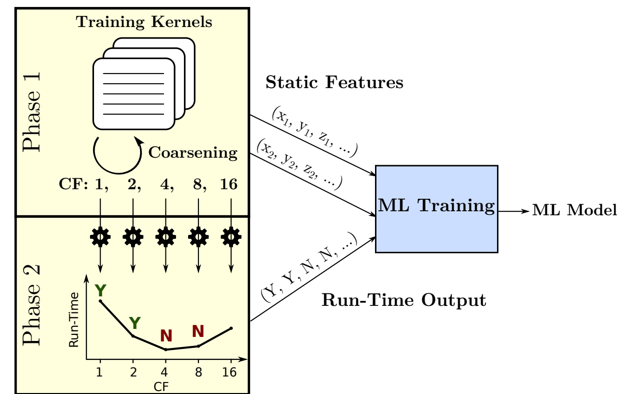


Figure 7. Model training overview. In Phase 1) we collect static features of the kernel functions. In Phase 2) we run the testing programs collecting the binary output variables.

by increasing coarsening factor. The binary training output variable is computed answering the following question: *Is further coarsening going to improve the performance of the current version?* Asking this question on each version of the kernel function will generate a tuple of binary values. Then, for each kernel the 5 sets of features and the 5-tuple of binary variables are used for training the machine learning model. This process is repeated for each target architecture.

6.2 Cascade NN Model

Our model is built using Neural Networks for classification. Neural Networks [7] are a supervised learning algorithm that classify data-points assigning to each of them the expected probability to belong to a given class.

Given a previously unseen program the model determines whether coarsening should be applied or not. If the answer is yes, the model is applied again to the coarsened version of the program to determine if we should keep going or stop coarsening. In the latter case, the model is called a third time deciding if the program should be coarsened one more time and so on. Figure 8 gives an overview of how the model is applied. Such model relies on the peculiar shape of the coarsening performance curve, which shows improvement up until the optimal coarsening factor.

Further coarsening leads to performance degradation. This common behavior is shown with figure 2. Note that no actual recompilation of the program is needed. The values of the features for the coarsened versions can be statically computed from the values of the uncoarsened version. This it true because the coarsening transformation, from the source-code point of view, behaves in a predictable manner. The model simply takes as input the static features of the original OpenCL kernel and gives as output the predicted coarsening factor. This makes extremely easy to integrate such model in a real optimizer compiler.

We choose the cascading predictor to implement a conservative approach. By iteratively querying the model in multiple intermediate steps we make sure to limit the application of thread coarsening only to those applications which truly benefit from it. This is a beneficial property, since large coarsening factors are rarely optimal (consider the Hinton diagrams in figure 5).

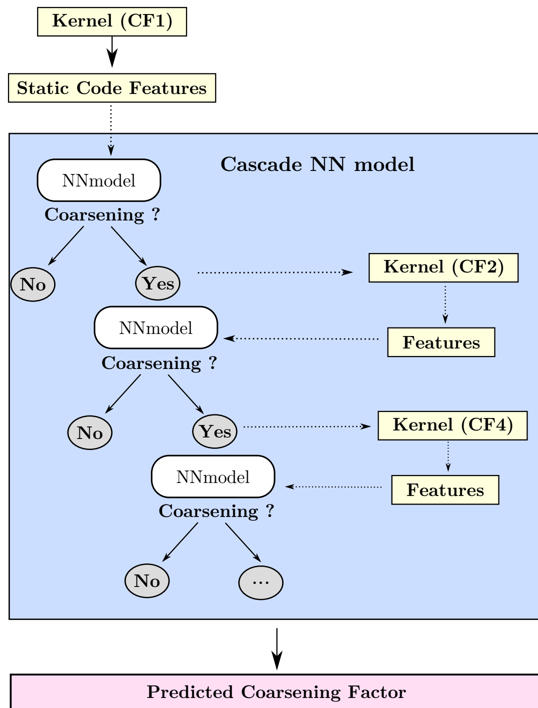


Figure 8. High-level view the use of our Neural Network cascade model. Static source code features of the program are extracted from the original kernel code and fed into the model which decides whether to coarsen or not. If yes, new features are computed and the model is queried again. From a high level point of view the model simply computes the best coarsening factor from static program characteristics. At the end of the process the program is compiled using the predicted factor.

6.3 Program Features

This section describes the static feature selection process. The model is based exclusively on static characteristics of the target OpenCL kernel function. These characteristics, called features, are extracted using a function pass working on the LLVM intermediate representation. Note that, since our goal is to develop a *portable* optimizer, we don't use any architecture-specific feature. We apply our function analysis at the LLVM-bitcode level because this is the lowest level of abstraction that is portable across architectures. In particular, we do not collect any information after instruction scheduling or register allocation, since these are target-specific phases. We first describe the candidate features. This is followed by the process used to reduce these to a smaller useful subset.

6.3.1 Candidate Feature

The full list of candidate features is given in table 3. We selected these based on the results of previous work [21]; where we analyzed which hardware performance counters are affected by the coarsening transformation. These are the number of executed branches, memory utilization (in terms of load and cache utilization) and instruction level parallelism (ILP). We approximate these *dynamic* counters using *static* code characteristics counterparts, such as the total number of kernel instructions, static number of branches and divergent regions and static ILP. There are 17 candidate static features that approximate the six most important dynamic counters.

Feature Name	**Description**
BasicBlocks	Number of Basic Blocks
Branches	Number of Branches
DivInsts	Number of Divergent Instructions
DivRegionInsts	Number of Instructions in Divergent Regions
DivRegionInstsRatio	Ratio between the Number of instructions inside Divergent Regions and the Total number of instructions
DivRegions	Number of Divergent Regions
TotInsts	Number of Instructions
FPInsts	Number of Floating point Instructions
ILP	Average ILP per Basic Block
Int/FP Inst Ratio	Ration between Integer and Floating Point Instructions
IntInsts	Number of Integer Instructions
MathFunctions	Number of Math Builtin Functions
MLP	Average MLP per Basic Block
Loads	Number of Loads
Stores	Number of Stores
UniformLoads	Number of Loads that do not depend on the Coarsening Direction
Barriers	Number of Barriers

Table 3. Candidate static features used by the machine learning model.

Absolute Features.

Divergent control flow has a strong impact on performance for graphics processors because it forces the serialization of instructions that would normally be executed concurrently on the different lanes of a warp [1]. The degree of a kernel thread divergence is measured using the results of *divergence analysis*. We count the total number of divergent instructions (instructions that depend on the *thread-id*, *i.e.,* the position of the work-item in the NDRange space), the total number of divergent regions (CFG regions controlled by a divergent branch) and their relative size with respect to the overall number of instructions in the kernel. Other counters related to the complexity of the control flow of the kernel are the total number of blocks. Finally we also compute the average per-block theoretical ILP and Memory Level Parallelism (MLP) at the LLVM instruction level. To compute the static ILP and MLP we followed the methodology presented in prior work [27].

Relative Features.

As described in section 2.2 the effects of thread coarsening on the shape of the code are highly predictable. Thanks to this property we can statically compute the value that features will assume after the application of the coarsening transformation. This information is used during training and prediction. In particular, for each feature, we also calculate its relative increase caused by thread coarsening. This value is computed according to the following formula:

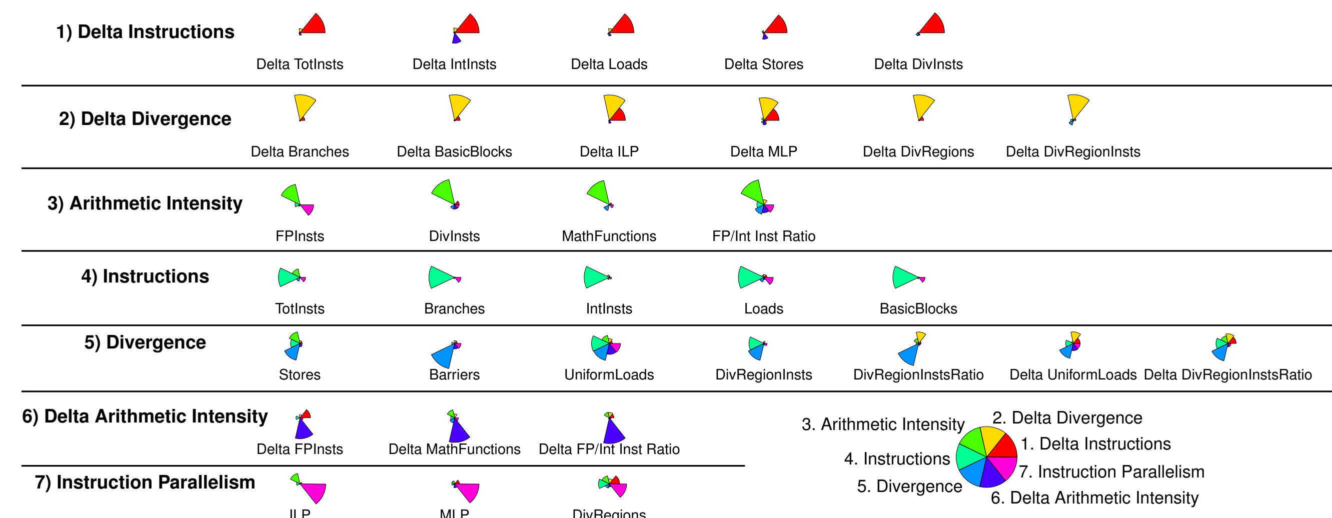


Figure 9. Result of the PCA analysis with Varimax rotation. The seven resulting components are labeled on the left and sorted according to their relative importance (the amount of variance of the feature space covered by each of them). On the right-end side segment-plots depicts the contribution of each original feature to a given output principal component. Features are laid out in rows according to the component they contribute to.

1) Delta instructions
2) Delta divergence
3) Arithmetic Intensity
4) Instructions
5) Divergence
6) Delta Arithmetic Intensity
7) Instruction Parallelism

Table 4. Final features selected after the application of Principal Component Analysis.

$\frac{(feature_{After} - feature_{Before})}{feature_{Before}}$, where $feature_{Before}$ and $feature_{After}$ are the values of the feature before and after coarsening. In the remainder of the paper delta features are identified by *Delta* added to the feature name.

6.3.2 Feature Selection

This section describes how we select features starting from the candidate ones.

Principal Component Analysis.

The large number of features (absolute plus deltas) is automatically reduced using the standard technique of Principal Component Analysis.

It is important to notice that we use the same set of features in table 3 for all devices without manual intervention. Principal Components Analysis minimizes the cross-correlation of the input features by linearly aggregating those features that are highly correlated and therefore redundant. After application of PCA we have the 7 final selected features in table 4 which describe 95% of the variance of the original space.

With the goal of providing an understanding of the importance of each feature we applied to the results of naive PCA a space transformation called Varimax rotation [23]. This transformation makes sure that each feature of the original space contributes either very strongly or very weakly to each new principal component. Such a property makes easier to interpret the output of PCA and allows one to label each principal component with a meaningful name based on the features that contribute to it. A similar approach to PCA has been followed by Kerr *et al.* [16] for prediction of execution time of parallel applications in heterogeneous environments.

Figure 9 shows the result of the PCA with Varimax rotation when reducing the space from 34 candidate features to the 7 components. The new principal components are organized by rows and labeled on the left. Each segment plot shows which component a given original feature contributes to. Components are sorted by rows according to their relative importance: how much of the feature variance they cover. So the most important component is labeled *Delta Instructions*. This component groups together features describing by how much the number of instructions in the kernel body increases while coarsening.

The second most important component (*Delta Divergence*) describes the increase in the amount of divergence. It is interesting to notice that the two most important components describe the *difference* of characteristics in the feature space rather then their absolute value. Absolute features account for a smaller amount of variance of the feature space. Section 7.4 describes how two representative features of the first two components are related to the results of the coarsening transformation and how they are used for prediction.

7. RESULTS

This section presents the evaluation of our machine learning predictor. All the speedups reported in this section are relative to the execution time of the original application executed without applying the coarsening transformation, *i.e.*, coarsening factor 1.

7.1 Unique Factor Model Description

To offer a rigorous evaluation of our machine learning model we compare its performance against a simple heuristic. Given a set of training programs with the corresponding

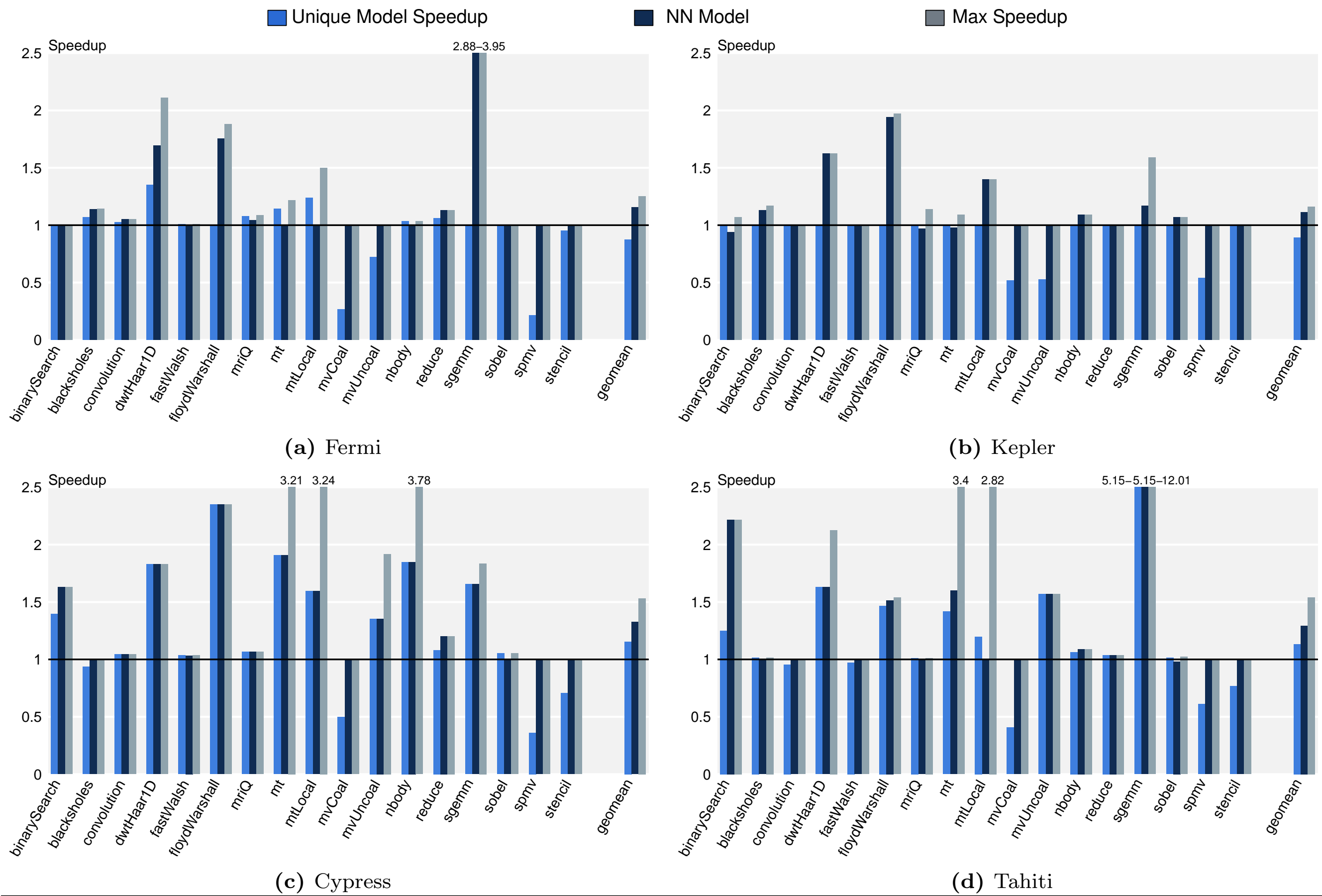


Figure 10. Bar-plots summarizing the results. The first bar (■) represents the speedup given by *Unique Model*. The second bar (■) is the speedup given by the *NN Model*. The third bar (■) is the maximum speedup attainable with thread coarsening.

speedups for each coarsening factor we compute the single best factor on average for that platform. The chosen value will then subsequently be applied on the testing program (not present in the original training set). This optimization policy makes the assumption (often wrong) that programs behave similarly to one-another and, therefore, similar optimization parameters give comparable performance. We employed this simple average-based policy because of lack of similar previous work against which to compare our model. In the following sections we call this simple model *Unique Model*. Our machine learning based predictor is called *NN Model*.

7.2 Performance Evaluation

Figure 10 shows the results of our experiments for the four target devices. We report three bars: one for the performance of *Unique Model*, the second one for *NN Model* and the final one shows the maximum speedup attainable with coarsening.

Performance of Unique Model.

We first observe that *Unique Model* performs poorly on all the devices. On the two Nvidia it results in a slowdown on average, while on AMD its gives an improvement over the baseline but it is far from the optimal. This difference of *Unique Model* across the two vendors can be explained considering that on Nvidia the performance profile is quite irregular. Either a benchmark greatly benefit from coarsening, such as *sgemm* and *floydWarshall* or it does not at all, as in the case of *spmv* and *stencil*. These two types of behavior make hard to estimate performance based on the average trend without considering the inherent properties of each benchmark. On AMD, *Unique Model* records a speedup since coarsening given on average higher improvements (1.53x and 1.54x) and a larger number of kernels benefit from it.

Divergent Kernels.

Applications such as *mvCoal*, *spmv* and *stencil* are penalized by coarsening on all the devices, *i.e.,* the best factor is 1. The reason for this lies in the divergent behavior of these applications. These kernels are enclosed into divergent regions checking for the bounds of the iteration space. Such a pattern is particular detrimental to the baseline performance on GPUs and coarsening makes this problem worse. *Unique Model* leads to significant slowdowns on these kernels since they deviate from the norm. The features *DivRegions* and *DivRegionInsts* model the degree of divergence in the kernel body. Leveraging this information *NN Model* successfully predicts coarsening factor 1 for all the benchmarks for which *Unique Model* leads to slowdowns. This positive result is a consequence of the conservative design policy of our cascade model (section 6.2).

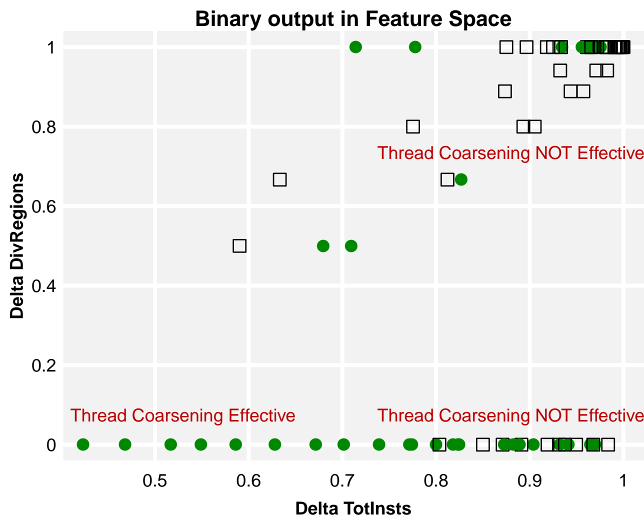


Figure 11. All our training data-points laid out according to two of the most relevant features: *Delta TotInsts* and *Delta DivRegions*. A dot identifies that coarsening is successful, a square that is not. The red labels highlights different regions of the feature space.

Uniform Kernels.

On the other hand *NN Model* enables coarsening effectively for benchmarks such as *sgemm* and *floydWarshall* on all architectures. These two applications show high speedups with coarsening. This is because they benefit from the reduction in redundant memory operations that coarsening ensures. Such a characteristic is expressed in the feature *UniformLoads*. In [21] we propose an explanation for this behavior.

The two applications for which *NN Model* fails to achieve significant improvements are *mt* and *mtLocal*. On AMD in particular these two kernels benefit from coarsening but the model does not take advantage of the possible gain. The reason lies in the particular shape of the code of these two programs. They contain few instructions, just loads and stores. Thus coarsening leads to the replication of all the instructions in the function body. Consequently this makes the feature *Delta TotInst* and *Delta Divergent Insts* have high values. Usually this signifies that coarsening should be disabled.

In summary, *NN Model* gives an average performance improvement of 1.16x, 1.11x, 1.33x, 1.30x on *Fermi*, *Kepler*, *Cypress* and *Tahiti* respectively. Our model never spoils performance significantly in any of the 68 cases. The largest performance penalty happens on *binarySearch* run on *Kepler* and it is about 5%.

7.3 Accuracy of prediction

Table 5 shows the predicted coarsening factor against the optimal one for all programs and devices. From this table we confirm the diversity of the best optimization factor as described by the Hinton diagrams in figure 5. We also see that *NN Model* seldom overshoots the factor and always predicts *no coarsening* (*i.e.,* factor 1) when the optimal is one.

7.4 Model Analysis

This section provides an understanding of how the *NN Model* predicts the coarsening factor relying on the distribution of the training points in the feature space. Figure 11 shows all the points in our data set, *i.e.,* all kernels for all the coarsening factors, arranged in the two dimensional space of *Delta TotInsts* and *Delta DivRegions* for AMD *Cypress*. The plot shows a dot for each benchmark for which coarsening is effective and a square when it is not, these are our training data. From this plot we clearly see that *Delta DivRegions* (on the y-axis) splits the data points in two clusters: one for which its value is 0 and one for which its value is positive. For the first cluster other features will discriminate these points in a higher-dimensional space. In the first cluster fall kernels such as *mt*, *sgemm* or *mri-q* which do not have any divergent region. The second cluster instead contains *mvCoal*, *stencil*, *spmv*, kernels whose body is enclosed by divergent branches. The feature on the x-axis instead represent the overall difference in instructions in the kernel body before and after coarsening.

From the distribution of dots and square we notice the following trends. (1) Configurations at the bottom of the graph (with low *Delta DivRegions*) tend to benefit from coarsening. Not having divergent regions in the original code means that coarsening is likely to be effective. The opposite also holds, since configurations in the top half of the plot usually do not benefit from the transformation. (2) Configurations on the right-hand side of the plot are likely not to benefit from coarsening. A sharp increase in the instruction number signifies a high level of divergence leading to poor performance.

8. RELATED WORK

Thread coarsening.

Volkov *et al.* [31] were the first to introduce thread coarsening as a compiler transformation for GPGPU computation, they focused on linear algebra applications. Unkule *et al.* [30] proposed an evaluation of coarsening as a compiler pass. Coarsening has only been applied by hand to a limited set of CUDA applications. Experiments have been deployed only on only one device with no explanation of the performance results. Coarsening is one of transformations used by Yang *et al.* [34] in their cross-architectural compiler for OpenCL. Their source-to-source transformations have been implemented in the Cetus compiler working on the AST. This limits the applicability of their methodology to simple linear algebra programs, thus missing the possibility of studying the effects of transformations on complex benchmarks. None of the mentioned works proposed a heuristic or a model to automatically determine the coarsening factor, they are mere evaluations of the attainable performance.

GPU optimizations.

The search for the maximum performance on graphics processors has lead to the development of application-specific auto-tuners. Examples of these are [8] and [10]. Much research has been put also in more widely applicable transformations for GPUs. Ryoo *et al.* [25] have been the first to propose compiler transformations for GPGPU computing. They address memory-related optimizations with the goal of improving access patterns. Liu *et al.* [19] perform

Kernel	Fermi		Kepler		Cypress		Tahiti	
	Predicted	**Best**	**Predicted**	**Best**	**Predicted**	**Best**	**Predicted**	**Best**
binarySearch	1	1	4	16	2	2	8	8
blackscholes	4	8	2	4	1	1	1	2
convolution	4	4	1	1	4	4	1	1
dwtHaar1D	4	8	4	4	4	4	2	4
fastWalsh	1	2	1	1	8	4	1	1
floydWarshall	4	16	4	8	4	4	4	16
mriQ	8	4	4	2	4	4	1	2
mt	1	8	4	2	4	32	4	32
mtLocal	1	8	4	4	4	32	1	32
mvCoal	1	1	1	1	1	1	1	1
mvUncoal	1	1	1	1	4	8	2	2
nbody	1	2	2	2	2	16	4	4
reduce	4	4	1	1	2	2	2	4
sgemm	8	32	2	16	4	8	2	16
sobel	1	1	2	2	1	4	8	4
spmv	1	1	1	1	1	1	1	1
stencil	1	8	1	1	1	1	1	1

Table 5. The table reports the best coarsening factors compared against the ones predicted by *NN Model* for all programs and devices.

an analysis of input sensitivity of CUDA programs choosing the best local work size according the input size. Zhang *et al.* [35] propose a dynamic thread remapping policy to reduce the impact on performance of branch divergence and non-coalesced accesses. High level program tuning for GPUs has been the target of extensive research too. Sponge [13] is a compiler toolchain for the optimization of streaming programs for GPUs. Dubach *et al.* [12] propose a set of compiler optimizations to improve performance of OpenCL programs generated by the Lime Java compiler.

Compiler-transformation tuning using machine learning.

Much research as been focused on the tuning of compiler transformations using machine learning. Stephenson *et al.* [28] have introduced two classification models to determine the best unrolling factor the loops of the SPEC 2000 benchmarks. In [20], a nearest neigbour model is used to select a set of optimizations for numerical Java programs. In [11] Dubach *et al.* predict the performance of an optimized version of a program given static features and profile runs of different versions of the same single-threaded program using a Neural Network for modelling. Sanchez *et al.* [26] employ Support Vector Machines to improve compilation time and start-up performance of Just-In-Time compilation. SVMs are also used in [32] to map streaming programs to multi-cores. Machine learning has been used by Stock *et al.* [29] use an ensemble of various machine learning models for tuning the parameters of automatic loop vectorization for application-specific polyhedral programs. More recently Jia *et al.* have employed regression trees to explore the space of algorithmic and hardware parameters for OpenCL GPU applications [14]. Finally, Lee *et al.* [17] employ a variety of machine learning methods for performance prediction of parallel applications.

9. CONCLUSION

This paper addresses the problem of performance portability for general purpose computing on GPUs by optimizing the coarsening factor for the thread-coarsening transformation pass. We first presented an evaluation of the complexity of the problem characterizing the optimization space. On top of the insights gained with this analysis we built a machine learning model based on a Neural Network to determine the best coarsening factor on four GPUs from two different vendors. The proposed technique achieves a average performance improvement of 1.16x, 1.11x, 1.33x, 1.30x on *Fermi*, *Kepler*, *Cypress* and *Tahiti* respectively, without penalizing performance.

10. REFERENCES

[1] Nvidia Corporation *The Cuda specification.*

[2] Nvidia's Next Generation CUDA Compute Architecture: Fermi `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2009.

[3] *AMD Accelerated parallel processing OpenCL*, 2012.

[4] Nvidia's Next Generation CUDA Compute Architecture: Kepler `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012.

[5] HSA Programmer's Reference Manual: HSAIL Virtual ISA and Programming Model, Compiler Writer's Guide, and Object Format (BRIG), 2013.

[6] The SPIR Specification, Standard Portable Intermediate Representation, Version 1.2, Jan. 2014.

[7] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, Inc., New York, NY, USA, 1995.

[8] J. W. Choi, A. Singh, and R. W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus.

PPoPP '10, pages 115–126, New York, NY, USA, 2010. ACM.

[9] B. Coutinho, D. Sampaio, F. Pereira, and W. Meira. Divergence analysis and optimizations. PACT, pages 320 –329, oct. 2011.

[10] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju. Auto-tuning of fast fourier transform on graphics processors. *SIGPLAN Not.*, 46(8):257–266, Feb. 2011.

[11] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. CF '07, pages 131–142, New York, NY, USA, 2007. ACM.

[12] C. Dubach, P. Cheng, R. M. Rabbah, D. F. Bacon, and S. J. Fink. Compiling a high-level language for gpus: (via language support for architectures and compilers). In *PLDI*, pages 1–12, 2012.

[13] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. ASPLOS '11, pages 381–392, New York, NY, USA, 2011. ACM.

[14] W. Jia, K. Shaw, and M. Martonosi. Starchart: Hardware and software optimization using recursive partitioning regression trees. PACT '13, 2013.

[15] R. Karrenberg and S. Hack. Improving performance of opencl on cpus. CC, pages 1–20, 2012.

[16] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling gpu-cpu workloads and systems. GPGPU '10, pages 31–42, New York, NY, USA, 2010. ACM.

[17] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, and S. A. McKee. Methods of inference and learning for performance modeling of parallel applications. PPoPP '07, pages 249–258, New York, NY, USA, 2007. ACM.

[18] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. AsanoviÄĞ. Convergence and scalarization for data-parallel architectures, 2013.

[19] Y. Liu, E. Zhang, and X. Shen. A cross-input adaptive framework for gpu program optimizations. IPDPS '09, pages 1 –10, may 2009.

[20] S. Long and M. F. O'Boyle. Adptive java optimisation using instance-based learning. ICS, pages 237–246, 2004.

[21] A. Magni, C. Dubach, and M. F. O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. SC '13. ACM, 2013.

[22] A. Magni, C. Dubach, and M. F. P. O'Boyle. Exploiting gpu hardware saturation for fast compiler optimization. GPGPU-7, 2014.

[23] B. Manly. *Multivariate Statistical Methods: A Primer, Third Edition.* Taylor & Francis, 2004.

[24] S. Moll. Decompilation of LLVM IR, 2011.

[25] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[26] R. Sanchez, J. Amaral, D. Szafron, M. Pirvu, and M. Stoodley. Using machines to learn method-specific compilation strategies. CGO '11, pages 257 –266, april 2011.

[27] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc. A performance analysis framework for identifying potential benefits in gpgpu applications. PPoPP '12, pages 11–22, New York, NY, USA, 2012. ACM.

[28] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. CGO '05, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

[29] K. Stock, L.-N. Pouchet, and P. Sadayappan. Using machine learning to improve automatic vectorization. *ACM Trans. Archit. Code Optim.*, 8(4):50:1–50:23, Jan. 2012.

[30] S. Unkule, C. Shaltz, and A. Qasem. Automatic restructuring of gpu kernels for exploiting inter-thread data locality. CC, pages 21–40, 2012.

[31] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[32] Z. Wang and M. F. O'Boyle. Partitioning streaming parallelism for multi-cores: A machine learning based approach. PACT, 2010.

[33] P. Xiang, Y. Yang, M. Mantor, N. Rubin, L. R. Hsu, and H. Zhou. Exploiting uniform vector instructions for gpgpu performance, energy efficiency, and opportunistic reliability enhancement. ICS '13, pages 433–442, New York, NY, USA, 2013. ACM.

[34] Y. Yang, P. Xiang, J. Kong, M. Mantor, and H. Zhou. A unified optimizing compiler framework for different gpgpu architectures. *TACO*, 9(2):9, 2012.

[35] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for gpu computing. ASPLOS '11, pages 369–380, New York, NY, USA, 2011. ACM.