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 1.11x and 1.33x 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.

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 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 coarsening 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 Binary-Search. 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
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 proprietary 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 LLVMBitcode using the clang open-source front-end (stage 1). The coarsening transformation is applied at the LLVMBitcode level (stage 2). The transformed LLVMBitcode 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.

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 openclbase 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.
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 programs 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 mCVoal 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
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). It is 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 5](image.png)

**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.
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 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 is 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).
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.

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.

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

\[
\frac{\text{feature} \_\text{After} - \text{feature} \_\text{Before}}{\text{feature} \_\text{Before}}, \quad \text{where feature} \_\text{Before} \quad \text{and feature} \_\text{After} \quad \text{are the values of the feature before and after coarsening. In the remainder of the paper delta features are identified by Delta added to the feature name.}
\]

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
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. Our machine learning based predictor is called NN Model. Our machine learning based predictor is called NN Model. Our machine learning based predictor is called NN Model. Our machine learning based predictor is called NN Model. Our machine learning based predictor is called NN 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 DivRegionInstrs 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).
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 mvec, 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. Ryyo 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
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 neighbour 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.

### Table 5.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Fermi</th>
<th>Kepler</th>
<th>Cypress</th>
<th>Tahiti</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Predicted</td>
<td>Best</td>
<td>Predicted</td>
<td>Best</td>
</tr>
<tr>
<td>binarySearch</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>blackscholes</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>convolution</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>dwtHaar1D</td>
<td>4</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>fastWalsh</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>floydWarshall</td>
<td>4</td>
<td>16</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>mriQ</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>mt</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>mtLocal</td>
<td>1</td>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>mvCoal</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>mvUncoal</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>nbody</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>reduce</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sgemm</td>
<td>8</td>
<td>32</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>sobel</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>spmv</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>stencil</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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


