Received: 21 September 2015 | Revised: 20 September 2016 | Accepted: 20 September 2016

DOI 10.1002/cpe.4029

WILEY

RESEARCH ARTICLE

Machine learning-based auto-tuning for enhanced performance portability of OpenCL applications

Thomas L. Falch | **Anne C. Elster**

Department of Computer and Information Science, Norwegian University of Science and Technology (NTNU), Trondheim, Norway

Correspondence
Thomas L. Falch, Department of Computer and Information Science, Norwegian University of Science and Technology, Trondheim, Norway.
Email: thomafal@idi.ntnu.no

Funding Information
CloudLightning EU H2020 project., Grant/Award Number: 643946

Summary

Heterogeneous computing, combining devices with different architectures such as CPUs and GPUs, is rising in popularity and promises increased performance combined with reduced energy consumption. OpenCL has been proposed as a standard for programming such systems and offers functional portability. However, it suffers from poor performance portability, because applications must be retuned for every new device. In this paper, we use machine learning-based auto-tuning to address this problem. Benchmarks are run on a random subset of the tuning parameter spaces, and the results are used to build a machine learning-based performance model. The model can then be used to find interesting subspaces for further search. We evaluate our method using five image processing benchmarks, with tuning parameter space sizes up to 2.3 M, using different input sizes, on several devices, including an Intel i7 4771 (Haswell) CPU, an Nvidia Tesla K40 GPU, and an AMD Radeon HD 7970 GPU. We compare different machine learning algorithms for the performance model. Our model achieves a mean relative error as low as 3.8% and is able to find solutions on average only 0.29% slower than the best configuration in some cases, evaluating less than 1.1% of the search space. The source code of our framework is available at `https://github.com/acelster/ML-autotuning`.

KEYWORDS

artificial neural networks, auto-tuning, heterogeneous computing, machine learning, OpenCL, performance portability

1 | INTRODUCTION

After the end of single core frequency scaling, parallel and heterogeneous systems have become increasingly popular. One of the most popular heterogeneous platforms today is a latency optimized CPU with a few, high single-thread performance cores, combined with 1 or more throughput optimized GPUs with many, slower, and simpler cores, for high parallel performance.

While such systems are highly capable in theory, programming them remains challenging. One notable issue is portability, making code written for one device be executed on another. OpenCL[1] has been proposed as a solution to this problem. Programs written in OpenCL can be executed on any device supporting the standard. Currently, this includes CPUs and GPUs from AMD, Intel, and Nvidia as well as devices from other vendors.

Although OpenCL offers functional portability—that is, OpenCL code will run correctly on different devices—it does not offer *performance portability*. Therefore, if code tuned to achieve high performance on one device is executed unmodified on another, it often performs badly. For this reason, code must be retuned for each new device it is executed on. This is clearly undesirable, as the retuning process is time-consuming, may introduce errors, and requires expert programmer knowledge of both the software and the hardware. The problem of performance portability is neither new nor tied to OpenCL. It is almost always present when moving code between CPUs of different generations or vendors. However, this problem is exacerbated with OpenCL, because it is designed for a larger variety of devices, with more diverse architectures, so that even small code changes can have a significant impact. Furthermore, it provides no facilities for automatic tuning of code when moving between devices.

In this article, we explore how *auto-tuning*[2–4] may be used to overcome this issue. In its simplest form, auto-tuning involves automatically generating and measuring the performance of several

*Present address: Institute for Computational and Engineering Sciences (ICES), University of Texas at Austin, USA.

candidate implementations and then picking the best one. Auto-tuning can be divided into two types: empirical and model driven.[5] In empirical auto-tuning, all possible candidate implementations are evaluated to find the best one. While this guarantees that the optimal implementation is found, it can be highly time-consuming if there are a large number of candidates. Model-driven auto-tuning attempts to solve this problem by introducing a performance model. The model is used to find a subset of promising candidates, which are then evaluated. While this reduces the time required for the auto-tuning, the results depend heavily on the quality of the performance model, which furthermore can be difficult and time costly to develop.

There is also a third approach[6,7]: Instead of manually deriving an analytical performance model, it can be built automatically, using machine learning methods. In this case, a random set of candidate implementations are executed, and the measured execution times are used to learn a statistical model. This model is then used to pick promising candidates for evaluation, as with traditional model driven auto-tuning.

In the following sections, we show how to use machine learning-based auto-tuning to retune OpenCL code to different

TABLE 1 Parameters used for the benchmarks and their possible values

Benchmark	Parameter	Possible values
common	Workgroup size in x dimension	1, 2, 4, 8, 16, 32, 64, 128
	Workgroup size in y dimension	1, 2, 4, 8, 16, 32, 64, 128
	Output pixels per thread in x dimension	1, 2, 4, 8, 16, 32, 64, 128
	Output pixels per thread in y dimension	1, 2, 4, 8, 16, 32, 64, 128
`convolution`	Use image memory	0, 1
	Use local memory	0, 1
	Add padding to image	0, 1
	Interleaved memory reads	0, 1
	Unroll loops	0, 1
`raycasting`	Use image memory for data	0, 1
	Use image memory for transfer function	0, 1
	Use local memory for transfer function	0, 1
	Use constant memory for transfer function	0, 1
	Interleaved memory reads	0, 1
	Unroll factor for ray traversal loop	1, 2, 4, 8, 16
`stereo`	Use image memory for left image	0, 1
	Use image memory for right image	0, 1
	Use local memory for left image	0, 1
	Use local memory for right image	0, 1
	Unroll factor for disparity loop	1, 2, 4, 8
	Unroll factor for difference loop in x direction	1, 2, 4
	Unroll factor for difference loop in x direction	1, 2, 4
`bilateral`	Workgroup size in x dimension	1, 2, 4, 8, 16, 32
	Workgroup size in y dimension	1, 2, 4, 8, 16, 32
	Workgroup size in z dimension	1, 2, 4, 8, 16, 32
	Output pixels per thread in x dimension	1, 2, 4, 8, 16, 32
	Output pixels per thread in y dimension	1, 2, 4, 8, 16, 32
	Output pixels per thread in z dimension	1, 2, 4, 8, 16, 32
	Use image memory	0, 1
	Use local memory	0, 1
	Precompute color filter	0, 1
	Precompute distance filter	0, 1
`median`	Workgroup size in x dimension	1, 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 128
	Workgroup size in y dimension	1, 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 128
	Pixels per thread in x dimension	1, 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 128
	Pixels per thread in y dimension	1, 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 128
	Use image memory for input	0, 1
	Use local memory for input	0, 1
	Use sorting (0) or histogram (1) based algorithm	0, 1
	Use local memory for sorting/histogram array	0, 1

The parameters listed under *common* are present in the `convolution`, `raycasting` and `stereo` benchmarks.

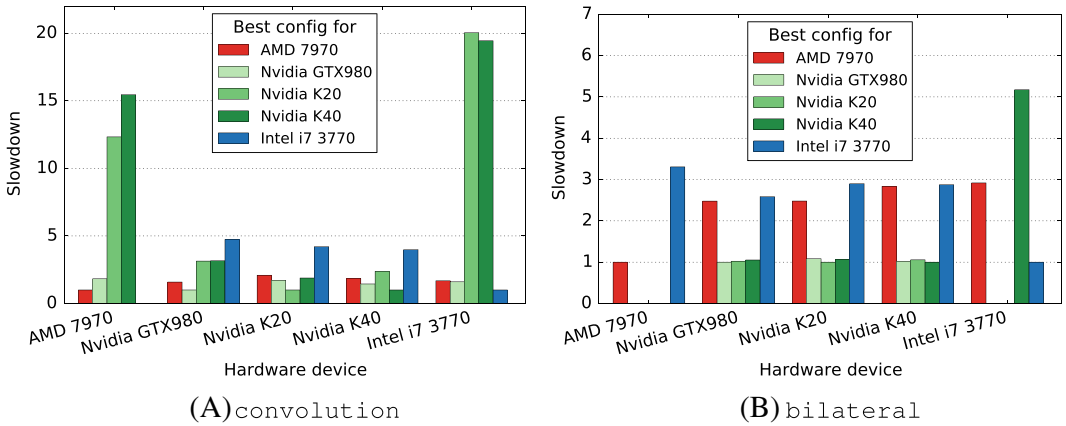


FIGURE 1 *Slowdown* using optimal configurations from other devices compared with the optimal configuration for a given device. Missing bars indicate that the configuration was invalid on that device

devices. We show how we achieve good performance without having to evaluate a large number of configurations or manually build a performance model. The work presented here is an extended version of Falch and Elster.[8] Here, we additionally include

- Improved background and related work sections, including tables classifying the machine learning methods used for performance modeling in related work.
- An analysis of the effect of including invalid configurations during training.
- A method to determine the size of the subspace explored exhaustively as the second step of our auto-tuner, based on a probabilistic calculation.
- A comparison of the relative accuracy of different machine learning methods for our performance model.
- An extension showing how our model can incorporate multiple different problem sizes, with promising results.
- A more thorough evaluation, using additional devices, benchmarks, and parameter settings as well as a comparison with naive, search-based auto-tuning.

The remainder of this article is structured as follows: The next section provides an example demonstrating the need for solutions to the problem of poor OpenCL performance portability. Section 3 contains background information on heterogeneous computing and OpenCL as well as an introduction to machine learning and how it can be used for performance modeling. Section 4 provides an overview of related work and classifies the machine learning methods found there. Our auto-tuning method is described in Section 5. Results are presented in Section 6 and discussed in Section 7. Finally, Section 8 concludes and outlines possible future work.

2 | MOTIVATIONAL EXAMPLE

To illustrate the poor performance portability of OpenCL, we used 2 basic image processing benchmarks, convolution and bilateral filtering, further described in Table 4 using the default problem sizes listed in Table 5. We ran the 2 benchmarks on 5 common devices, an Intel i7 3770 (Ivy Bridge) CPU, an Nvidia Tesla K20 GPU, an Nvida Tesla K40 GPU, an Nvidia GTX980 GPU, and an AMD Radeon HD 7970 GPU. The benchmarks have a number of tuning parameters, such as the workgroup size, and whether or not to apply various potential optimizations (all the parameters are described in Table 1). By giving different values to the different tuning parameters, we can generate 131 072 and 655 360 different parameter configurations for `convolution` and `bilateral`, respectively. Because the architectures of the devices are different, we expect that the best tuning parameter configurations for each device will be different as well. We confirmed this by exhaustively trying all possible configurations for all 5 devices, thereby finding the best configuration for each device, which all differed from each other. We then measured the performance of these 5 parameter configurations on all the devices. This was repeated for both benchmarks.

Figure 1 shows the results. As we can see, using the "wrong" configuration can seriously degrade performance, even when that configuration is optimal for some other device. For instance, using the best Nvidia K40 configuration on the Intel i7 resulted in slowdowns of 19.4 and 5.2 for `convolution` and `bilateral`, respectively, compared with the best Intel configuration. While this result is unsurprising given the large architectural differences between CPUs and GPUs, the problem persists between the GPUs, although to a lesser degree. Using the best Nvidia K40 configuration on the AMD card for `convolution` or the best AMD configuration on the K40 for `bilateral` results in slowdowns of 15.5 and 2.9, respectively. Even between GPUs from the same vendor, we see the same issue. For `convolution`, using the best K40 configuration on the GTX980 gives a slowdown of 3.2, and using the best K20 configuration on the K40, two devices with the same microarchitecture, gives a slowdown of 2.4. Finally, as the lack of bars in certain cases indicates, the best configuration on 1 device might simply not compile or run on another, as described in further detail in Section 5.2.1.

For clarity, we only include results from 2 benchmarks here; however, the same trend of poor performance portability can be found in other benchmarks as well. This plainly demonstrates the need for OpenCL code to be retuned before executing it on a new device. Furthermore, resorting to exhaustive search, as we did in this example, is in general not practical because of the potentially very large size of the configuration spaces.

3 | BACKGROUND

In this section, we will provide background information on heterogeneous computing with OpenCL. We will also give an introduction to machine learning and how it has been used for performance modeling.

3.1 | Heterogeneous computing and OpenCL

Combining CPUs and GPUs is currently one of the most popular ways to create a heterogeneous platform. While GPUs originally were developed for graphics rendering, they have evolved into general purpose programmable, highly parallel accelerators. Here, we will only present a brief overview of their architecture and programming model; for further details, the reader is referred to Owens et al,[9] Brodtkorb et al,[10] and Smistad et al.[11]

A number of compute units compose GPUs; each of which consists of several processing elements.† The processing elements of a compute unit work in an SIMD fashion, executing instructions in lock step. The largest memory space available is global memory, which resides in slow, off-chip DRAM (but is separate from the system's main memory). While the global memory is cached on newer GPUs, they also have a fast, on-chip, scratch pad memory that can be used as a user-managed cache. In addition, they have texture memory, which is optimized for access patterns with 2D and 3D spatial locality, and constant memory designed to allow for high performance when accessed by many threads concurrently.

OpenCL[1] has been proposed as a standard for heterogeneous computing and makes it possible to write code once and execute it on different devices, including CPUs and GPUs. The code is organized into host code and kernels. The host code executes as a normal CPU program and sets up and launches the kernels on a device, like a GPU or the same CPU the host code is executing on, using calls to the OpenCL library. Kernels are written in a variant of C and are executed in parallel by multiple threads known as work items, which are organized into work groups. If executed on a GPU, the work groups are typically mapped to compute units and the work items to processing elements, on the CPU they are mapped to the CPU cores. A number of logical memory spaces exist: local memory (mapped to the fast on-chip memory on GPUs), image memory (mapped to the GPU texture memory), and constant memory (mapped to the hardware constant memory on the GPU). On the CPU, all of these memory spaces are typically mapped to main memory. Moving data between these memory spaces is explicitly done by the user from the host or kernel code. Because they may be mapped to different kinds of hardware memory, they have different performance characteristics.

3.2 | Machine learning for performance modeling

This section will provide a basic introduction to machine learning with a focus on how it can be used for performance modeling.‡ Machine learning-based algorithms use example data to automatically build, or *learn*, models. These models can later be used to make predictions or decisions based on patterns found in the data, as opposed to writing an explicit program to do the equivalent task.

Machine learning algorithms can broadly be divided into unsupervised and supervised learning. In unsupervised learning, the input is unlabeled data and the algorithm will attempt to automatically discover some structure in these data. Clustering algorithms will, for instance, attempt to cluster the input data points into (potentially hierarchical) groups where the members of the same group have similar properties.

In supervised learning, on the other hand, the algorithm is provided with data in the form of input and output pairs, which are used to build a model of the mapping between the input and the output. The model can then later be used to predict the output for unseen input. If the outputs are categories, this is known as classification and , if they are real numbers, as regression. In both cases, it can be viewed as a form of function approximation. As such, it overlaps with statistical regression analysis. A large number of algorithms exist; many of which can (sometimes with small modifications) do both regression and classification. Examples include artificial neural networks (ANNs), support vector machines (SVM) and support vector regression (SVR), decision trees and regression trees as well as various forms of linear regression. All of these algorithms can be used with ensemble methods such as boosting and bagging. Such methods attempt to improve the quality of the model by outputting some kind of average of several models, built with different initial parameters or different training data.

Performance models attempt to predict, or explain, the performance of computer programs or the hardware upon which they run. There exists a long tradition for analytical performance modeling, where models in the form of mathematical equations are devised by hand, based on expert knowledge of the hardware, software, and their interaction. However, the increasing complexity and diversity of modern computer systems have made this task more difficult. For this reason, using machine learning to automatically build performance models has received a lot of attention, as shown in Table 2. The table contains references to works using machine learning for performance modeling, especially in the context of auto-tuning. The references are further described in Section 4.

Machine learning performance modeling requires performance data from which the models can be learnt. The data are typically gathered by running a large number of programs, or the same program with different inputs or under different conditions, and recording the relationship

TABLE 2 Overview of related work using machine learning for performance modeling, classified by machine learning method, and hardware system used

	Hardware		
Method	CPU only	GPU, GPU + CPU	Cluster
Artificial neural networks	13,14	15	16–19
Regression/classification trees	20	6,12,22,23–26	18,27
K-nearest neighbour	20,28,29	30	
SVR/SVM	28,31		18
Other	7,32	22,33,34	18

SVR/SVM, support vector regression and support vector machines.

†Here, we are adopting the terminology of OpenCL. On Nvidia GPUs, these are known as streaming multiprocessors and CUDA cores, on AMD GPUs as compute units and stream processors.

‡For a general introduction, see e.g..[12]

TABLE 3 Overview of related work using machine learning for performance modeling, classified by machine learning method, and hardware system used

	Input				
Output	**Static code features**	**Dynamic code features**	**Problem size**	**Tuning parameters**	**Hardware parameters**
Execution time	20,22		6,16–19,22,	6,18,20,23,27	16,19,22
Speedup	26,22,14	14	22	14	22
Best tuning params.	13,15,20,21,28,29	7,29	24,30,31		
Work distribution	25,34		33,34		
Power				23,27	
Output size			17		

between the input and output variables of interest. These data are then given to a machine learning algorithm that builds a model. The model can later be used to predict the output variable for new, unseen, input variables. While the models of some machine learning algorithms are are hard to interpret, others create models that can be inspected to understand how the different input variables contribute to the output.

The input and output variables used depend on the purpose of the model. Examples include predicting execution time from the values of tuning parameters or static code features. Tuning parameters can also be the output of a model, by predicting the best values of tuning parameters from the problem size, or hardware parameters. Table 3 shows the input and output variables used for the machine learning models found in the related work of this article.

3.2.1 | Analytical and machine learning modeling

An important distinction between analytical and machine learning-based modeling is the treatment of the environment of the model, such as the hardware or software stack used. In analytical modeling, the effects of the environment can be taken explicitly into account by the model developer, such that those parameters of the environment, which matter can be included in the model. This means that the model is still valid if the environment changes, and it is simple to reason about. For machine learning-based modeling, on the other hand, the environment used will affect the training data and therefore be incorporated into the model directly. Thus, even if the model is amenable to interpretation, it can be hard to distinguish between the effects of the environment and the input variables. Furthermore, the model will only be valid for the environment where the training data were gathered; if it is changed, new data must be gathered and the model retrained. It is possible to circumvent this problem by letting the parameters of the environment that is believed to be relevant and be included as inputs to the model, but this will require more training data, gathered while changing those parameters.

After surveying the related work of this paper, we did not find any consensus on which machine learning algorithms are best suited for performance modeling, as this choice depends heavily upon the exact input and output variables used, their relationship, the programs used, and the software and hardware environment.

However, at the end of the next section, we include a summary of the algorithms found in the related work for this article in Table 2, giving some indication of which methods are currently popular. We also classify the input and output variables used in Table 3.

4 | RELATED WORK

Auto-tuning[4] is a well-established technique that has been successfully applied in a number of widely used high-performance libraries, including Fastest Fourier Transform in the West (FFTW)[35] for fast Fourier transforms,[36] Optimized Sparse Kernel Interface (OSKI)[3] for sparse matrices, and Automatically Tuned Linear Algebra Software (ATLAS)[2] for linear algebra.[37]

There are also examples of application specific empirical auto tuning on GPUs, eg, for stencil computations,[38] matrix multiplication,[39] and fast Fourier transforms.[40] Khan et al[41] used a script-based auto-tuning compiler to translate sequential C loop nests to parallel CUDA code. Furthermore, analytical performance models for GPUs and heterogeneous systems have been developed[42–45] and used for auto-tuning.[5]

Much work has been done on creating machine learning-based models and using them for tuning and auto-tuning, eg, to determine loop unroll factors,[28,46] which optimizations to apply for parallel stencil computations,[7,47] Message Passing Interface (MPI) parameters,[14] and general compiler optimizations.[20–32] Kulkarni et al[13] developed a method to determine a good ordering of compiler optimization phases, on a per function basis. Their method uses a neural network to determine the best optimization phase to apply next, given characteristics of the current, partially optimized code. They evaluated their method in a dynamic compilation setting, using Java. Singh et al[19] and Ipek et al[16] used a method similar to ours, based on ANN performance models. However, their focus was on large-scale parallel platforms such as the BlueGene/L, and they did not use their model as part of an auto-tuner. Vuduc et al [31] developed a statistical model to stop empirical search early if a near-optimal solution has been found, as well as a machine learning model to pick the best implementation of an algorithm based on the input size. Yigitbasi et al[18] also adopted an approach similar to ours, by building a machine learning-based performance model for MapReduce with Hadoop[48] and using it in an auto-tuner. In contrast to these works, our method uses values of tuning parameters to directly predict execution time, as part of an auto-tuner, using OpenCL on heterogeneous hardware.

Machine learning approaches have also been used for performance modeling and auto-tuning in a heterogeneous setting. Several works deal with developing machine learning methods to determine whether to execute a kernel on the GPU or the CPU[33,25] and how to balance load between the devices. [34,17] Furthermore, machine learning models have been developed to predict the best OpenCL memory space to use for stencil computations[26,21] and the best workgroup size. [22] Magni et al[15]

used an ANN model to determine the correct thread coarsening factor (the amount of work per thread) based on static code features, for OpenCL on different platforms. In another study, they used a nearest neighbor approach to determine the best way to parallelize sequential loops for OpenACC.[30] Liu et al[24] focused on how the properties of the program inputs affect the performance of CUDA programs and developed a method where a machine learning-based algorithm was used to determine the best optimization parameters for a kernel based on the input. In contrast to these works, we develop a performance model that predicts the execution time based on multiple different tuning parameters and use it in an auto-tuner.

The works by Bergstra et al,[6] Jia et al,[23] and Nugteren et al[49] are those most closely related to ours. In Bergstra et al,[6] a model based on boosted regression trees was used to build an auto-tuner, evaluated with a single GPU benchmark, filterbank correlation. The Starchart system[23] was used to build a regression tree model that could be used to partition the design space of an application, discover its structure, and find optimal parameter values within the different regions. It was then used to develop an auto-tuner for several GPU benchmarks. In contrast to these, our work compares different machine learning models, has more parameters for each kernel, and uses OpenCL to tune applications for both CPUs and GPUs. Nugteren et al introduced CLTune,[49] which, similarly to our system, uses auto-tuning to achieve performance portability for OpenCL kernels. Their system does, however, use search-based methods like simulated annealing and particle swarm optimization to find the best configuration, rather than machine learning. Furthermore, it was evaluated using fewer benchmarks.

Methods to achieve OpenCL performance portability without auto-tuning have also been explored. Zhang et al[50] identified a number of parameters that affect the performance of OpenCL codes on different platforms and showed how setting the appropriate values can improve performance. Faberio et al[51] used iterative optimization to adapt OpenCL kernels to different hardware by picking the optimal tiling sizes. A different approach was taken by Pennycook et al,[52] who attempted to determine application settings that will achieve good performance on different devices, rather than optimal performance on any single device.

We summarize this section in Tables 2 and 3. Table 2 classifies the machine learning approaches of the works in this section by the machine learning algorithm used, as well as the hardware system used. Table 3 classifies the works based on the types of input and output variables of the models. Works presenting multiple models or using multiple classes of input or output variables or hardware systems are listed multiple times. The tables are not based on an exhaustive literature survey but are intended to summarize this section and give some indication of currently popular machine learning methods and how they are used.

5 | MACHINE LEARNING-BASED AUTO-TUNING

Our auto-tuning scheme starts with parameterized benchmarks. The parameters form a space of possible implementations, with 1 possible implementation for each parameter configuration. From this space, we randomly select a subset of samples that are used to build a machine learning-based model. The model is then used to predict the execution time for all the possible configurations. In a second stage, a subset of configurations with the lowest predicted execution times are found, and their actual execution times are measured. Finally, the best of these configurations is found, and returned by the auto-tuner. If the model is sufficiently accurate, the optimal configuration will be among those found in the second stage and therefore returned by the auto-tuner. The entire method is illustrated in Figure 2.

5.1 | Code parameterization and candidate generation

Our method requires benchmarks with tuning parameters from which one can generate candidate implementations, with 1 candidate for each tuning parameter configuration. These candidates are all functionally equivalent, but using different values for the tuning parameters causes their performance to vary.

Our auto-tuning approach is general in the sense that it is able to take any application where the performance depends on the values of some parameters, and search for the values of those parameters that minimize the execution time. The method is oblivious to the exact nature of the parameters, their possible values, and the mechanics through which they affect performance. However, because our objective is trying to improve OpenCL performance portability, the intended input

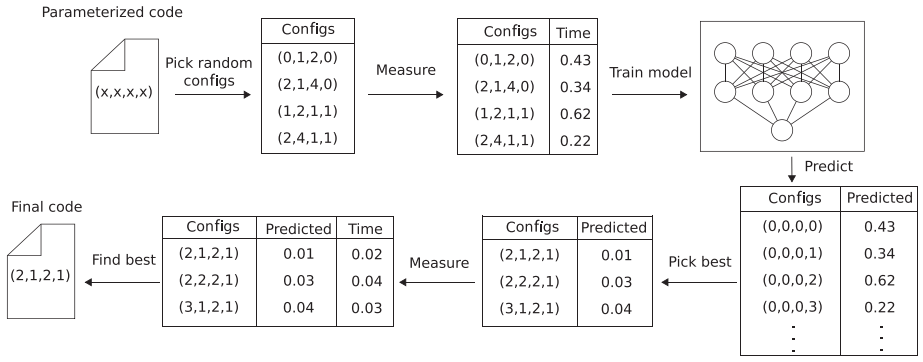


FIGURE 2 Overview of our auto-tuning approach. Execution times for a random set of configurations are measured and used to train a model, which then finds interesting subspaces for exhaustive search

TABLE 4 Description of image processing benchmarks used, including data type and dimensionality of input and output images and tuning parameter space size

Benchmark	Description	Input	Output	Param. space size
`convolution`	Convolution of image with box filter, example of stencil computation.	2D, `float`	2D, `float`	131 072
`raycasting`	Volume visualization generating a 2D image from 3D volume data.	3D, `float`	2D, `uchar4`	655 360
`stereo`	Computing disparity between 2 stereo images to determine distances to objects.	2 x 2D, `int`	2D, `int`	2 359 296
`bilateral`	Bilateral filtering of volume data with 2 filters based on physical and color distance.	3D, `uchar`	3D, `uchar`	746 496
`median`	Median filtering of image for noise reduction. More computationally heavy than `convolution`.	2D, `uchar`	2D, `uchar`	331 776

to our auto-tuner is a parameterized OpenCL kernel. This is a kernel that includes various optimizations and transformations thought to be beneficial for performance on different devices, as well as an interface to pick which optimizations and transformations to apply. In the following, we will describe the benchmarks we used and how we parameterized them. However, the general nature of the method should be kept in mind.

We chose the following 5 benchmarks for our experiments: `convolution`, `raycasting`, `stereo`, `bilateral`, and `median`, described in Table 4. These benchmarks are all from the domain of image processing and visualization. They were selected because they are computationally demanding, well suited for parallel implementation, and widely used for a broad range of applications. Furthermore, improving the performance of these particular algorithms is of great interest to our medical image processing and visualization collaborators.[11,53] To ease the process of parameterizing the kernels, we picked benchmarks consisting of a single, data-parallel kernel. Several of the optimizations we apply are related to the OpenCL memory hierarchy. For this reason, we carefully picked benchmarks such that the number and dimensionality of the input and output images and other data structures, as well as the data types of the pixels were varied. An overview of this can be found in Table 4. Because most of the optimizations we apply relate to the memory hierarchy, each of these benchmarks serves as a proxy for other benchmarks with similar input/output patterns, which we believe will interact in a similar fashion with out auto-tuner. Thus, although we only have a handful of benchmarks, they are representative of a much broader range of applications.

In general, the process of creating a parameterized benchmark is hard to formalize, because it depends strongly on the benchmark and requires knowledge of the application, programming model, and the hardware on which it will run. However, it broadly consists of 2 steps: (1) identifying locations with multiple potential implementations (like whether or not to apply a potential optimization), but where the best implementation is unknown, or device dependent, and (2) implementing all the variants, as well as code that allows 1 variant to be selected through some interface.

Because we restricted ourselves to OpenCL image processing codes, we were able to formalize the way to perform the parameterization in more detail as follows:

1. The OpenCL workgroup size is already a parameter.
2. The kernel is written to process a single (output) pixel. It is then wrapped in for-loops so that it can process multiple pixels. The amount of work for each thread (adjusted together with the total number of threads becomes a parameter).
3. Data structures that are only read from are identified. These can potentially benefit from being placed in image memory. Whether or not to do this is added as a parameter.
4. Data structures where data are reused by the same or different threads are identified. These can potentially benefit from being placed in local memory. Whether or not to do this is added as a parameter.
5. Small data structures that are only read from are identified. These can potentially benefit from being placed in constant memory. Whether or not to do this is added as a parameter.
6. Loops with constant trip counts are identified. These can potentially be unrolled.
7. Application-specific optimizations (like which sorting algorithm to use in `median`) are identified and added as parameters.

The parameters are implemented using either preprocessor macros in the kernel code or using variables set on the host prior to kernel launch. For instance, the local memory optimization is implemented by writing two versions of the code accessing memory, one using local memory, and one not, and using a `#if #else #endif` macro to pick the right version at kernel compile time. The memory space parameters can in general be combined in any way; for instance, if both image and local memory are used, the data structure will first be stored in image memory and then be manually cached in local memory.

The loop unrolling in `convolution` and `stereo` is implemented using OpenCL compiler pragmas. In `convolution`, the parameter simply determines if the unroll pragma is included or not, resulting in either full or no unrolling, while the parameter in `stereo` specifies several possible unroll factors. The unroll pragma is an optional extension and may be ignored by the OpenCL compiler; the compiler may also perform unrolling if this pragma is not present. In `raycasting`, the unrolled loop is the main ray traversal loop. This is a while loop where the number of iterations depends on the input data and can therefore not be unrolled by the OpenCL compiler. For this loop, we have implemented partial loop unrolling manually, using preprocessor macros, and the parameter is the loop unroll factor. The scheme may lead to unnecessary memory accesses and computations for unroll factors larger than 1 but will not affect correctness.

An overview and description of all the tuning parameters used for all the benchmarks, and their possible values, can be found in Table 1 in Section 2. The parameter space sizes can be found in Table 4.

While we followed the procedure outlined above, we did not implement all possible optimizations identified and have, in some cases, notably for the loop unrolling, workgroup size and work per thread, deliberately limited the potential parameters values to limit the

search space. In our evaluation, we compare the results found by our auto-tuner to the true best configuration, found by exhaustive search. This exhaustive search is highly expensive for large search spaces, and we limited the parameter space of the benchmarks to increase the practicality of the evaluation. We have also attempted to create benchmarks with different parameter space sizes and with different parameters and parameter ranges to evaluate the effect of these changes on the performance of the auto-tuner.

5.1.1 | Parameter space

In this section, we will describe the parameter spaces formed by our parameterization of the benchmarks.

Figure 3 shows the distribution of slowdowns for all the benchmarks described in the previous subsection and devices described in Section 6. Specifically, for each device and benchmark combination, it shows the distribution of the slowdowns of all the valid configurations compared with the best configuration for that device/benchmark combination. Despite variation between the devices and the benchmarks, some trends are clear. Firstly, there are few very good configurations: Configurations no more than 10% slower than the best constitute less than 4.74% of the valid configurations in all cases and less than 0.01% for several benchmark/device combinations. Secondly, the range of execution times is large: There are a large number of configurations more than 100 times slower than the best, and in some cases, there are several more than 1000 times slower than the best. Finally, there is a notable difference between the CPUs and the GPUs, the 2 previously mentioned trends are stronger for the GPUs. On the CPUs, there are a larger fraction of good configurations, and fewer very bad

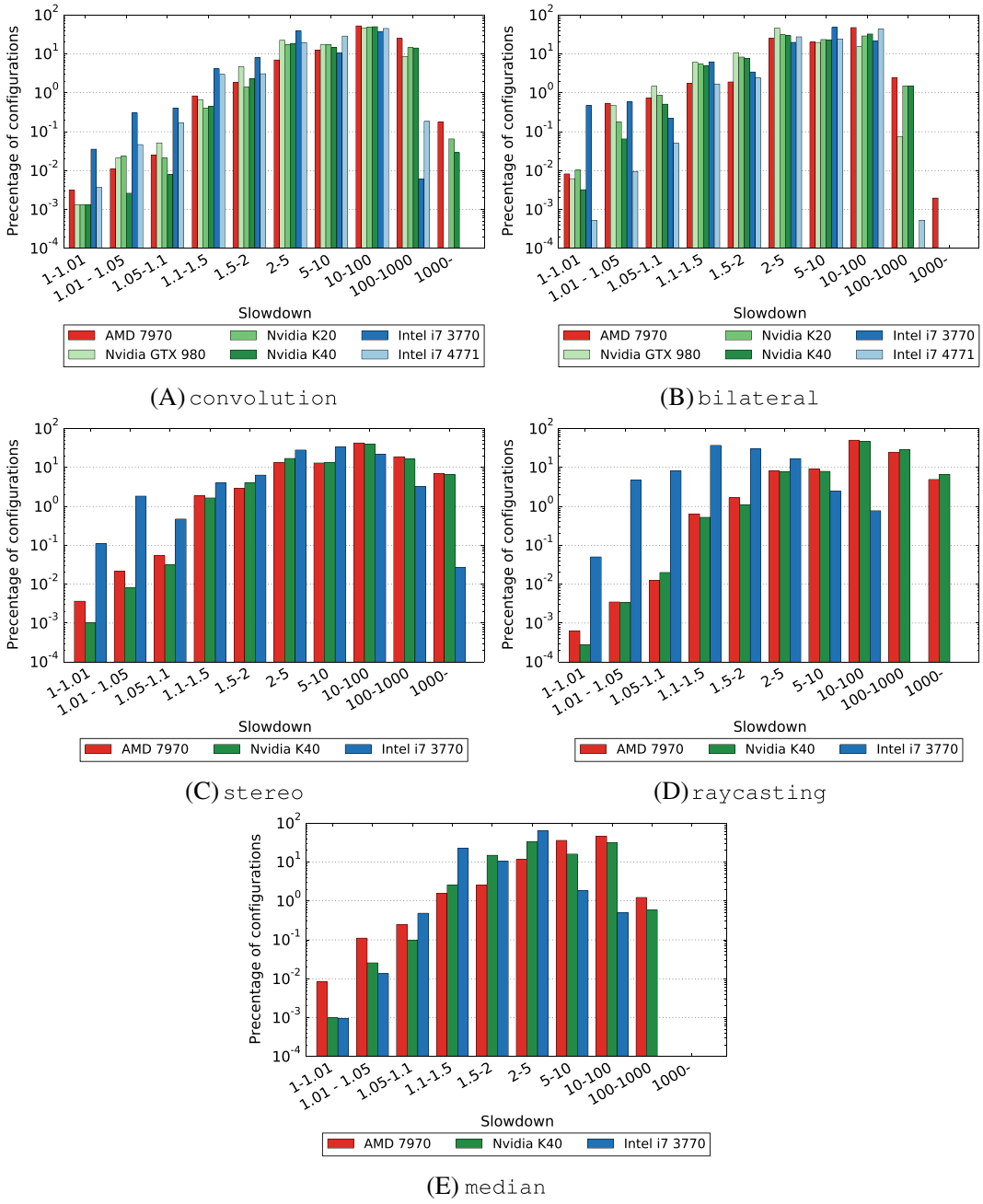
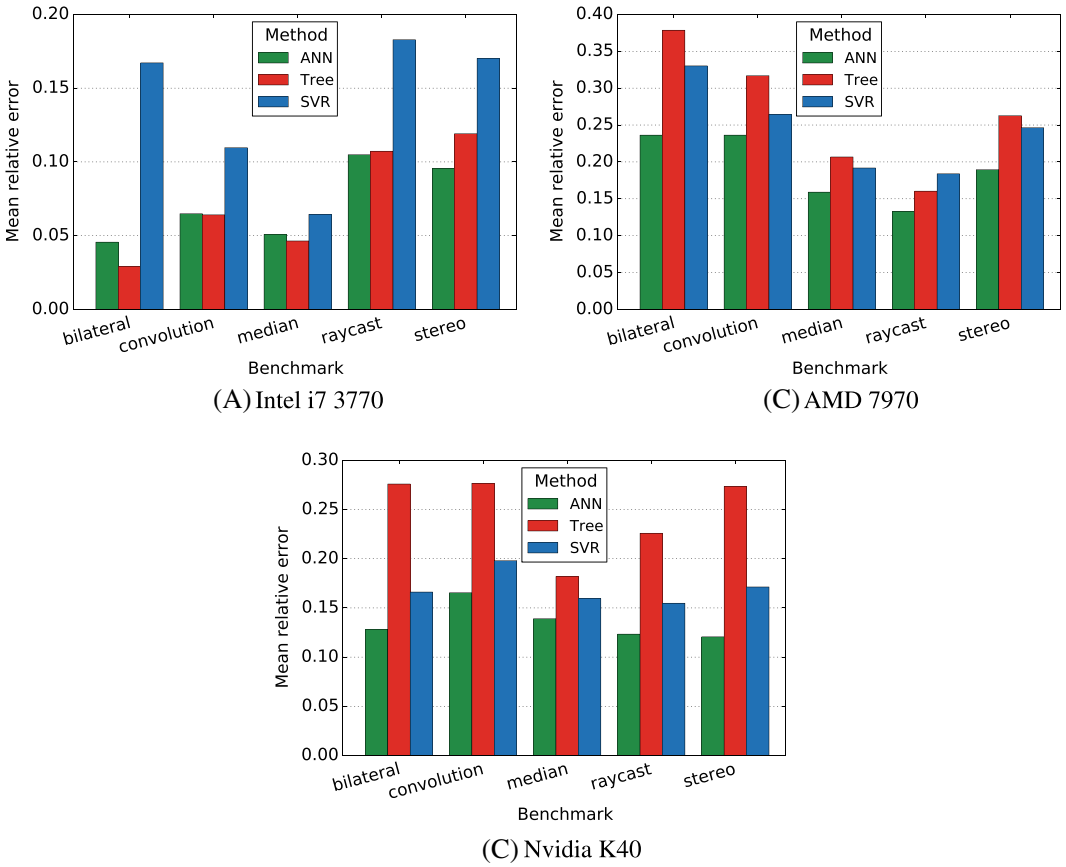


FIGURE 3 Distribution of slowdowns of configurations compared with the best configuration. Note the logarithmic scale



(A) Intel i7 3770

(C) AMD 7970

(C) Nvidia K40

FIGURE 4 Accuracy of different machine learning methods, for 2000 training samples, for different benchmarks and devices

configurations, indicating that the performance on the CPUs is less sensitive to the tuning parameters.

5.2 | Model building

The goal of our performance model is to be able to predict the execution time of a benchmark given a tuning parameter configuration. To do this, we first run the benchmark with several randomly chosen parameter configurations and record the corresponding execution times. These input-output pairs, or training samples, are then used to build, or train, the model. Using machine learning terminology, we are using *supervised learning*.

Multiple supervised learning algorithms exist. We compared ANNs, SVR, and regression trees, based on the popularity of these methods in the literature, as shown in Section 4, and their good predictive power, ability to handle arbitrary functions, and ability to handle noisy input. The results can be found in Figure 4.

Here, we use all 5 benchmarks described in the previous subsection and our 3 main computing devices, described in Section 6. For brevity, we only report results for 2000 training configurations, but they reflect the results for other values as well. As we can see, ANNs always perform best on the 2 GPU devices and are the best or close to the best on the CPU. The regression trees perform well on the CPU, being close to, and in some cases better than, ANNs. They do, however, perform poorly on the GPUs. Support vector regression performs better on the GPUs than the CPU but are still outperformed by ANNs.

Because of their good overall performance, and especially their superior performance on GPUs, we choose to use ANNs as our main machine learning methods, and all the results in the rest of this paper use ANNs. A significant drawback of ANNs is, however, the opaqueness of the resulting model, which is difficult to interpret and makes it hard to gain deeper insight into how the parameters interact and contribute to the final performance.

For all the methods, we experimented extensively to find the best hyperparameters, that is, the best parameters to the machine learning method. For ANN, we used a single hidden layer and evaluated all possible sizes for this layer between 5 and 100 neurons in increments of 5. We found that the optimal value depended upon both the benchmark and device used, and do therefore use between 40 and 70 neurons, depending upon the device/benchmark combination, in our experiments. Furthermore, we found that sigmoid activation functions gave good performance.

We also used a technique known as *bagging*[54] to further increase the performance of the model. Rather than using all the training data to build a single neural network, we split it into k parts and build k networks, each trained using all the data except one of the parts. During prediction, we feed the input to all the networks and then take the mean of their outputs as the final output. We found that this increased the accuracy of the predictions. For k, we have used a value of 11. For SVR and the regression tree model, we used the related technique of boosting.[55]

We also experimented with active learning,[56] where the partially complete model is used to determine what training data to use next to achieve the best accuracy, but without improvements.

During ANN training, the weights are adjusted to minimize the mean squared error between the predictions and the actual output. In our case, this would cause problems because we use the ANN to predict the execution time directly and are therefore interested in minimizing the relative error, rather than the absolute error. We resolved this problem by introducing an additional step where we take the logarithm of the execution times before training the neural network. The neural network then predicts the logarithm of the execution time and attempts to minimize the mean squared error when comparing with the logarithm of the actual execution time. This achieves the desired effect because reducing the absolute error of the logarithm of 2 values is equivalent to reducing the relative error of the values directly.

In this work, we create a machine learning-based performance model to find promising parts of an application optimization search space. An alternative approach is to use stochastic search, such as simulated annealing, particle swarm optimization,[57] or the Firefly algorithm[58] in this space directly, as done by, eg, Nugteren and Codreanu.[49] We chose to use machine learning-based methods to evaluate if the success and attention they have enjoyed recently [59,60] could be leveraged for auto-tuning. Furthermore, because of the complex search space formed by the benchmarks under consideration, we suspect that search based methods might more easily get stuck in local minima.

5.2.1 | Invalid configurations

A challenge specific for this kind of data is invalid parameter configurations, that is, configurations for which the corresponding code cannot be executed. This is typically because the resulting code requires more resources than are available on a given device. For instance, some devices place restrictions on how large workgroups can be, or how much local memory is available. Thus, invalidity is a function of both the configuration and device; if a configuration is invalid on 1 device, it might not be on another, more capable one. Furthermore, none of the configurations are inherently invalid; given a sufficiently capable device, they could all execute in theory.

Determining if a particular configuration is invalid for some device can often be done statically, but in some cases, it is necessary to attempt to compile and run the kernels.

If the model is trained for a specific device using a number of randomly selected configurations, some fraction of these will be invalid. These can be handled by (1) ignoring them and training the model with only valid configurations or (2) including the invalid configurations with artificial dummy timings. For the latter alternative, it is also necessary to come up with some scheme for inventing the dummy execution times. Possibilities include using the best, worst, and median of the valid execution times measured. Furthermore, for each invalid configuration, one can find its closest valid neighbor and used the execution time of that neighbor.

We evaluated these alternatives, comparing their accuracy when predicting the execution time of valid configurations. While optimizing for this metric may cause the model to predict low execution times for invalid configurations, this is of lesser importance, because we are able to filter out invalid configurations in a separate step later.

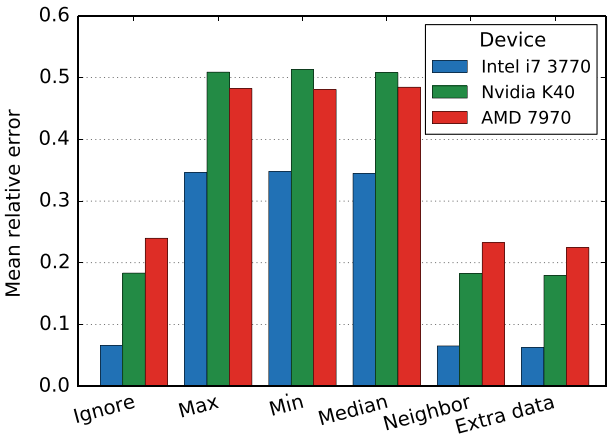


FIGURE 5 Mean relative prediction error for valid configurations for different ways of handling invalid configurations

Figure 5 shows the results. As we can see, using the minimum, maximum, or median yields much poorer results than simply ignoring the invalid configurations. On the other and, using the neighbor scheme outlined above slightly increases accuracy. However, one needs to keep in mind that this scheme in practice results in gathering more training data, because the neighbors of the invalid configurations typically are not among the original valid configurations. If one instead included this extra data directly, the accuracy becomes slightly better. That is, it is better to use the additionally measured execution times with their actual configurations rather than with the invalid neighbor configurations.

For this reason, we chose to simply ignore invalid configurations during training. This does cause the model to sometimes predict low execution times for invalid configurations, thereby including them in the second step. We explain how we deal with this in the next section.

5.3 | Prediction and evaluation

After the model is built, an estimate of the optimal parameter configuration can be found by simply predicting the execution time for all possible configurations and picking the best one. This remains feasible, despite large parameter spaces, because evaluating the model is orders of magnitude faster than executing the actual benchmarks.

However, because the model is not perfectly accurate, it is unlikely that the configuration with the lowest predicted execution time is the one with the lowest actual execution time. For this reason, our auto-tuning process includes a second stage, where the model is used to find promising subspaces of the parameter space. These subspaces are likely to contain the actual optimal configuration and are kept small enough so that they can be searched exhaustively.

In practice, we do this by picking the M configurations with the lowest predicted execution times. We then measure the actual execution times of these configurations and find the best among them. This process is still not guaranteed to find the globally best configuration, because the model may be so inaccurate that the globally optimal configuration is not included among the M configurations in the second stage. For this reason, M should be adjusted to fit the accuracy of the model.

For a highly accurate model, M can be small, while for a more inaccurate model, M should be larger, to achieve the same confidence that the globally best solution is among the M best predictions.

Rather than using exhaustive search to find the best of the M configurations in the second stage, some stochastic search method, like those described in Section 5.2, could be used to speed up this stage. However, while the model in the second stage conceptually finds a small number of interesting contiguous subspaces, the procedure outlined in the previous paragraph typically results in a list of discrete points in the space. Stochastic search methods are poorly suited for this kind of data because they assume a contiguous search space. Finally, the number of configurations in the second step is typically small compared with the amount of training data, which is the bottleneck.

While M can be set by hand in an ad hoc manner, based on experience, or after estimating the accuracy of the model, we will here present a more principled way to determine its value. To do this, we assume we have N possible configurations in total, $C_1,...,C_N$; all of which are valid. Each configuration has a true execution time, $t_1,...,t_N$, as well as a predicted execution time, $p_1,...,p_N$. The true execution times are (except for the training data) unknown. The predicted execution times come from the model, which in general is inaccurate; hence, the predicted and true times will not match in general, that is, $t_i \neq p_i$.

Assuming that we, from the predicted values and knowledge of the error distribution, can obtain probability density functions for the true execution times $f_1,...,f_N$, as well as corresponding cumulative probability density functions $F_1,...,F_N$, the probability r_j that the true execution time of an arbitrary configuration C_j is the minimum will be

$$r_j = \int_{-\infty}^{\infty} (f_j(x) \prod_{i \neq j} (1 - F_i(x)))dx \quad (1)$$

To understand this equation, remember that $f_j(x)$ is the probability that the true execution time of configuration j is x and that $1 - F_i(x)$ is the probability that the true execution time of configuration i is larger than x. Thus, we sum together, for each possible value of x, the probability that configuration j has that execution time and that all the other configurations have larger execution times. We let the integration go from $-\infty$ rather than 0 because the definitions of the fs will disallow nonzero probabilities for negative execution times. Thus, if we want to be certain that we have the best result with probability s, we simply pick the M configurations with the highest probabilities computed with Eq. (1) such that the sum of their probabilities is larger than the threshold s.

Even if we ignore some of the problems with the assumptions, this scheme is impractical because the numerical evaluation of Eq. (1) is too demanding. Instead, we adopt an alternative approach where we only look at a single configuration, as well as the currently best found configuration. In particular, if we only consider the currently best found configuration with execution time t_{min}, as well as an arbitrary other configuration C_i, the probability q_i that the actual execution time of C_i is smaller than t_{min} is

$$q_i = F_i(t_{min}) \quad (2)$$

In this way, we can calculate an estimate of the probability that a given configuration is better than the currently best found configuration. We can then include in the second stage all configurations where this estimate is above some threshold. In particular, to find the M second-stage configurations, we can start with the one with the best predicted execution time and, going from low to high predicted execution times, continue to include configurations as long as the estimated probability calculated with Eq. (2) remains above some threshold T. We initially use the best configuration in the training data as the best configuration found and update it as we go along. In this way, we keep including configurations as long as the probability that they are better than the currently best found configuration is above the threshold T. Thus, rather than having to specify M directly in some ad hoc manner, the user specifies the threshold T, which has a more meaningful interpretation.

This method also handles invalid configurations gracefully: as we go from configurations with low to high predicted execution times, before we evaluate Eq. (2) we attempt to compile the corresponding code, thus determining its validity. In this way, we always end up with M valid configurations, even if the model erroneously predicts that invalid configurations have low execution times.

For the method to work, we must still be able to determine the fs. For simplicity, we assume that the distributions f are Gaussian, making Eq. (2) straightforward to evaluate. While this greatly simplifies the calculation, it leads to inaccuracies because it implies that there are nonzero probabilities for negative execution times. As estimates for the means, we simply use the predicted execution times. The variance is estimated from the errors of the model, so that a more inaccurate model will have a larger variance. In particular, as we include

TABLE 5 Problem sizes for the benchmarks and their possible values

Benchmark	Problem size parameter	Possible values
`convolution`	Image width	2048, *3020*, **4096**
	Image height	*2048*, 3020, **4096**
	Filter width	*7*, **9**, 11
	Filter height	7, **9**, *11*
`raycasting`	Image width	*256*, **512**
	Image height	*256*, **512**
	Volume dimension	**128**, *256*
	Transfer function size	**128**,*256*
	Use trilinear interpolation	*0*,**1**
`stereo`	Image width	**256**
	Image height	**256**
	Disparity	**8**
	Radius	**2**
`bilateral`	Volume width	*128*, **256**
	Volume height	**128**, *256*
	Volume depth	**128**, *256*
	Filter width	*3*, **5**, 7
	Filter height	3, ***5***, 7
	Filter depth	3, **5**, *7*
`median`	Image width	**3072**, 3840, *4608*
	Image height	**3072**, 3840, *4608*
	Filter height	*3*, **5**, 7
	Filter width	*3*, **5**, 7

The default problem size is highlighted with bold and the alternative with italics. For `stereo`, we only considered a single problem size.

new configurations in the second stage, we use the difference between their actual and predicted execution time to update a running estimate of the variance.

5.4 | Varying problem size

In the presentation so far, we have assumed a fixed problem size for the benchmarks or that the best configuration is independent of the problem size. This is an unrealistic assumption; in practice, the problem size is typically varied, eg, by using filters of different sizes for the `convolution` benchmark. Furthermore, the best configuration for 1 problem size might not be the best one for another problem size. We demonstrated this using our convolution benchmark, with the default and alternative problem size from Table 5. For the default problem size on the Nvidia K40, the best configuration for the alternative problem size is 1.42 times slower than the best configuration for the default problem size.

The naive solution to this problem would be to redo the auto-tuning every time the problem size is changed. However, despite the fact that our machine learning-based auto-tuning is faster than exhaustive search, it is still a fairly costly operation. Instead, as an extension to our previous work,[8] we propose a different solution, where the problem size is integrated into the performance model, thus making it capable of predicting the execution time given both problem size and tuning parameter configuration. We do this by modifying the auto-tuning pipeline outlined above as follows.

- The benchmarks are changed to include parameters for problem size, if not already present.

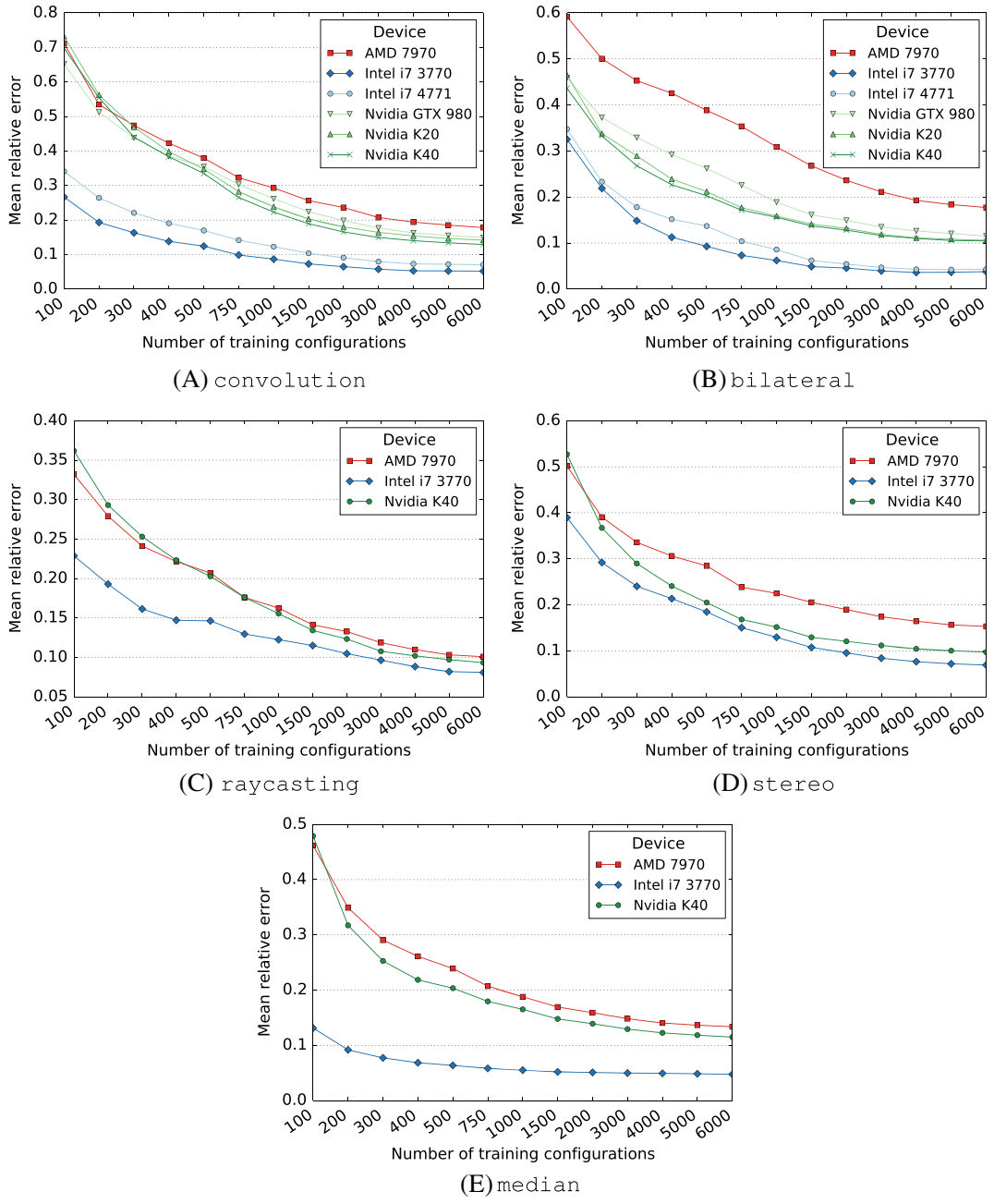


FIGURE 6 Mean relative prediction error for increasing number of training configurations, for different devices and benchmarks

- During training, we include training configurations with different problem sizes.
- The machine learning performance model is adapted to have additional inputs for the problem size.
- During prediction, we do not predict for the entire input space of the model but keep the problem size fixed at the size of interest and only predict for all the different tuning parameter configurations.

In this way, we obtain a much more general model, requiring less training data, by allowing knowledge gathered for 1 problem size to be applied to others. However, it works best if the behaviour of the benchmark is fairly similar and varies in a predictable way as the problem size changes. Furthermore, it requires that the relevant features of the problem size can be encoded in some reasonable way, with a small number of parameters.

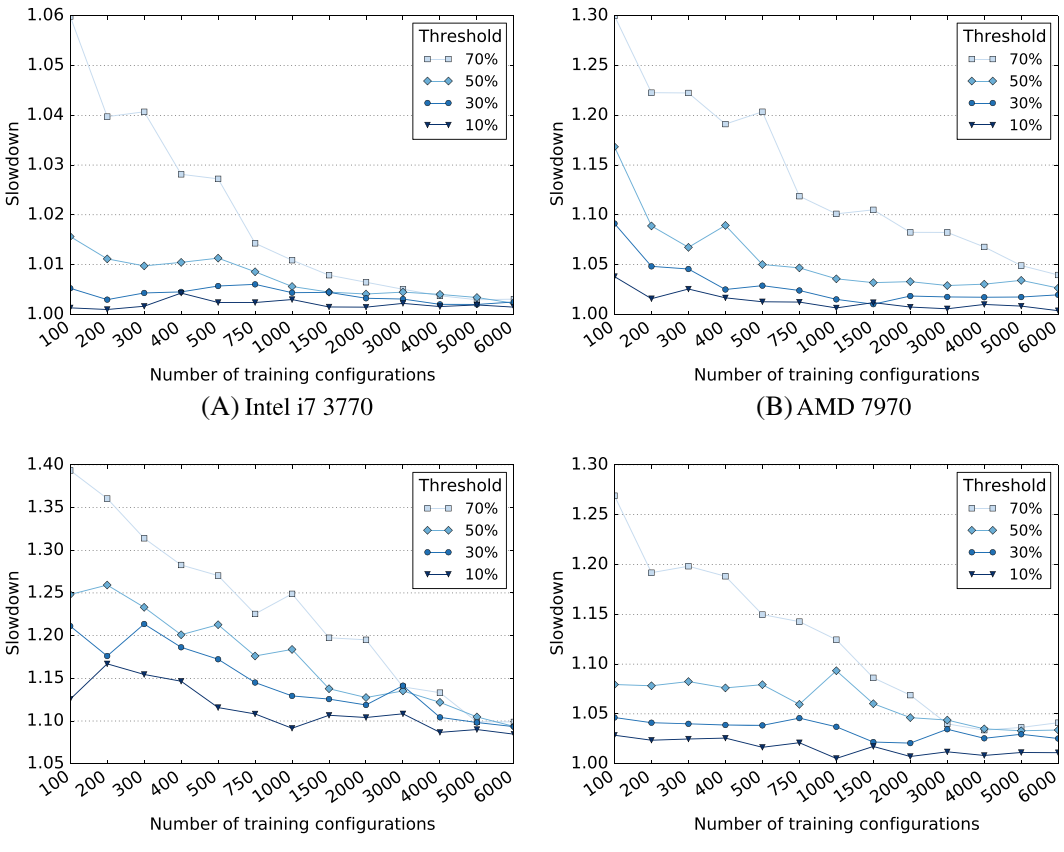
6 | RESULTS

To evaluate our method, we implemented 5 parameterized benchmarks in OpenCL, `stereo`, `convolution`, `bilateral`, `raycasting`, and `median`, with parameter space sizes ranging from 131K to 2.3M. The benchmarks, and the motivation for choosing them, is described in Section 5.1; a summary, and details about inputs and outputs, as well as parameter space sizes can be found in Table 4. The parameters used for the benchmarks can be found in Table 1 in Section 2. Unless otherwise noted, we used the default problem sizes from Table 5. For the experiments, we primarily used 3 devices: an Nvidia Tesla K40 GPU, an AMD Radeon HD 7970 GPU, and an Intel i7 3770 (Ivy Bridge) CPU. For some experiments, we also included additional devices: an Intel i7 4771 (Haswell) CPU and Nvidia Tesla K20 and GTX980 GPUs.

6.1 | Model accuracy

To evaluate the accuracy of the models created, we compared predicted and actual execution times. To study the effect of the amount of training data on the accuracy, we repeated this for neural networks trained using an increasing number of configurations. In particular, for each training set size n, we randomly selected n configurations from the entire parameter space and used them for training. We then used 2000 configurations, randomly selected from the entire parameter space (excluding those used for training) as validation, computing the relative prediction error of the model. Because the output of the model depends on the particular configurations used during training, as well as the random initial weights of the neural network, we repeated this process 20 times for each training set size, each time building a new neural network using new randomly selected configurations for training and validation. For each training set size, we report the mean of the relative error of these 20 runs. The results are shown in Figure 6.

The figures show that the mean relative error decreases as more samples are used to train the models but stabilizes or decreases much more slowly after around 2000 to 3000 samples, for all devices and benchmarks. Comparing the different devices, the CPUs perform



(A) Intel i7 3770 (B) AMD 7970

(C) Nvidia K40 (D) Nvidia GTX 980

FIGURE 7 Slowdown of configuration found by auto-tuner, compared with globally best configuration, for the `convolution` benchmark

significantly better than the GPUs, with a mean relative error between 3.8% and 8.1% for 6000 training configurations on the CPUs. The corresponding numbers are 9.3%-14.9% for the Nvidia GPUs and 10.0%-17.8% for the AMD GPU, with the AMD GPU consistently performing worst, notably for the `stereo` and `bilateral` benchmarks. If we compare the different benchmarks, the GPUs perform best on `raycasting` and worst on `convolution`. The CPUs, on the other hand, perform worst on `raycasting` and best on `bilateral`.

6.2 | Auto-tuning

To evaluate our auto-tuners ability to find good configurations, we measured the actual execution times for all possible configurations, for all combinations of devices and benchmarks. While this process is too time-consuming to perform as a part of a normal auto-tuning procedure, it allows us to assess the quality of the output of our auto-tuner by comparing it with the ground truth, that is, to the known globally optimal configurations.

As explained in the previous section, the model, and therefore the best configuration found, depends upon the configurations used for training, and the random initial weights of the neural network. All the results presented in this section is therefore the average of 30 runs.

The quality of the output depends on the number of samples used for training, as well as the threshold used in the second stage, which controls the size of the subspace of interesting configurations that are searched exhaustively. Figure 7 shows the results for the `convolution` benchmark, for the different devices. The figure shows the slowdown of the best configuration found by the auto-tuner compared with the known globally best configuration, as the number of training samples, and second-stage thresholds are varied. As we can see, our auto-tuner is able to find good configurations. For example, when 1000 configurations are used to train the model and we use a second-stage threshold of 10%, we find solutions that on average are 0.65%, 9.15%, 0.53%, and 0.29% slower than the global minimum for the AMD 7970, K40, GTX 980, and i7, respectively. The 10% threshold corresponds to using on average 305.7, 436.2, 283.9, and 111.4 samples in the second stage for the AMD 7970, K40, GTX 980, and i7, respectively; that is, we never evaluate more than 1.1% of the total number of possible configurations.

Ideally, using a fixed threshold for the second stage should result in the same average slowdown regardless of the number of samples used to train the model. If a less accurate model is used, this should be compensated for by including more configurations in the second stage. While this holds in some cases, eg, for the i7, GTX 980, and AMD 7970 for the 10%, 30% and 50% thresholds, respectively, we more generally see that the slowdown decreases as the number of training samples increases. On the other hand, it is generally the case that using a lower threshold improves performance for a given number of training samples, as expected.

To avoid getting lost in the details, we summarize the corresponding results for the other benchmarks. The full results are provided on-line,

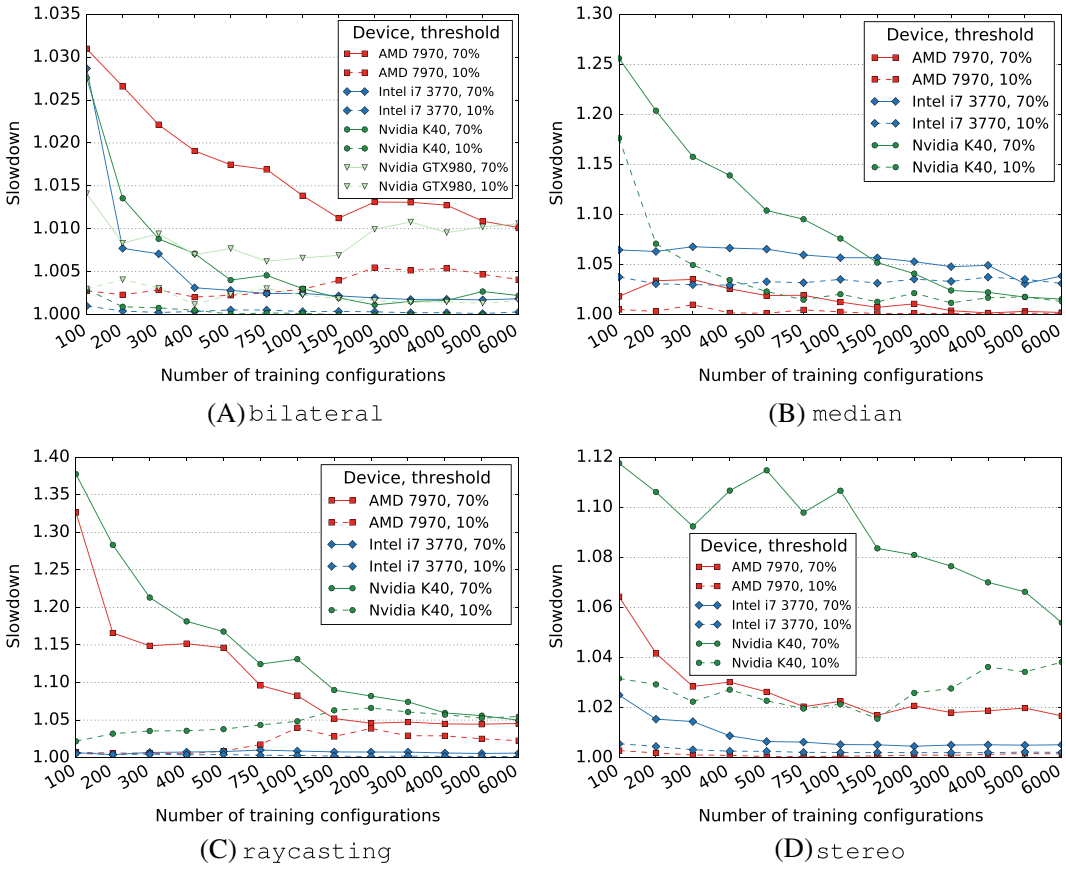


FIGURE 8 Slowdown of configuration found by auto-tuner, compared with globally best configuration, for different benchmarks

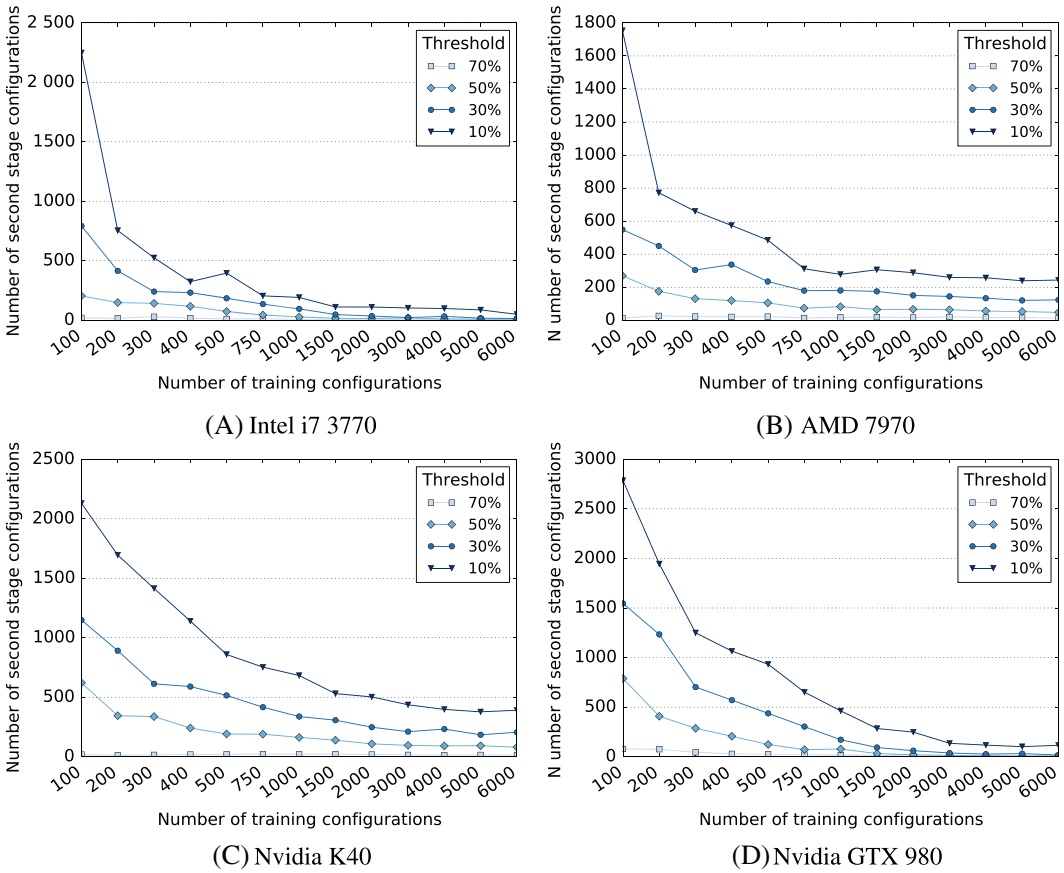


FIGURE 9 Number of configurations used in second stage

together with the underlying code.[§] Figure 8A-D shows the results for the different devices on the `bilateral`, `median`, `raycasting`, and `stereo` benchmarks, respectively, showing only the 70% and 10% thresholds. As we can see, if we use 1000 samples for training, and a 10% threshold, we are in all cases able to find solutions on average no more than 5% slower than the global minimum. The Intel CPU generally performs best, with the exception of the `median` benchmark. The performance on the `bilateral` benchmark is better than the other benchmarks, in particular for the 70% threshold.

The number of configurations used in the second stage for the `convolution` benchmark for the different devices are shown in Figure 9. As mentioned, we see how the less accurate models, trained with fewer configurations, use a larger number of configurations in the second stage to compensate for their lack of accuracy. In particular, the number of configurations drops rapidly initially, before leveling of as more training data are used, similar to how the model accuracy behaves, as shown in Figure 6. When the highly inclusive 10% threshold is used, hundreds of configurations are included, to increase the probability of finding the true minimum, while for the 70% threshold, there are never more than 10 configurations included.

6.2.1 | Naive search

While the results so far demonstrate the ability of our auto-tuner to find good configurations, we also want to ensure that the time required to build the model pays of. To do this, we compare our method with a naive search of a random set of configurations. In particular, our method returns the best configuration found of m random training configurations and n second-stage configurations picked by our model. We compare this against simply picking the best of $m + n$ random configurations. Our method should outperform the naive approach, because the n configurations picked by our model should on average be better than n random configurations. The results are shown in Figure 10, as slowdown compared with the globally best configuration, for the `convolution` benchmark, for different devices. As mentioned previously, the result of both our method and the naive approach depends upon the configurations used for training and search. Figure 10 therefore shows the average result of 30 runs, as well as error bars indicating the *range*, that is, the best and worst result obtained. To enable a fair comparison, we fix the number of configurations used in the second step and include results for using 30 and 1000 configurations.

As we can see, our method does perform significantly better than the naive approach. In particular, while the best case results for the naive approach can be quite good, both the average and worst-case results are worse than our auto-tuning. The difference does, however, diminish as more training data are used. This is expected, because it increases the chances for both our auto-tuning and the naive approach to find a good configuration among the first m configurations, decreasing the significance of the n configurations in the second step. For the same reason, our method outperforms the naive approach for longer when a larger second stage is used, as this places less emphasis on the first stage.

[§] Available at `https://github.com/acelster/ML-autotuning`.

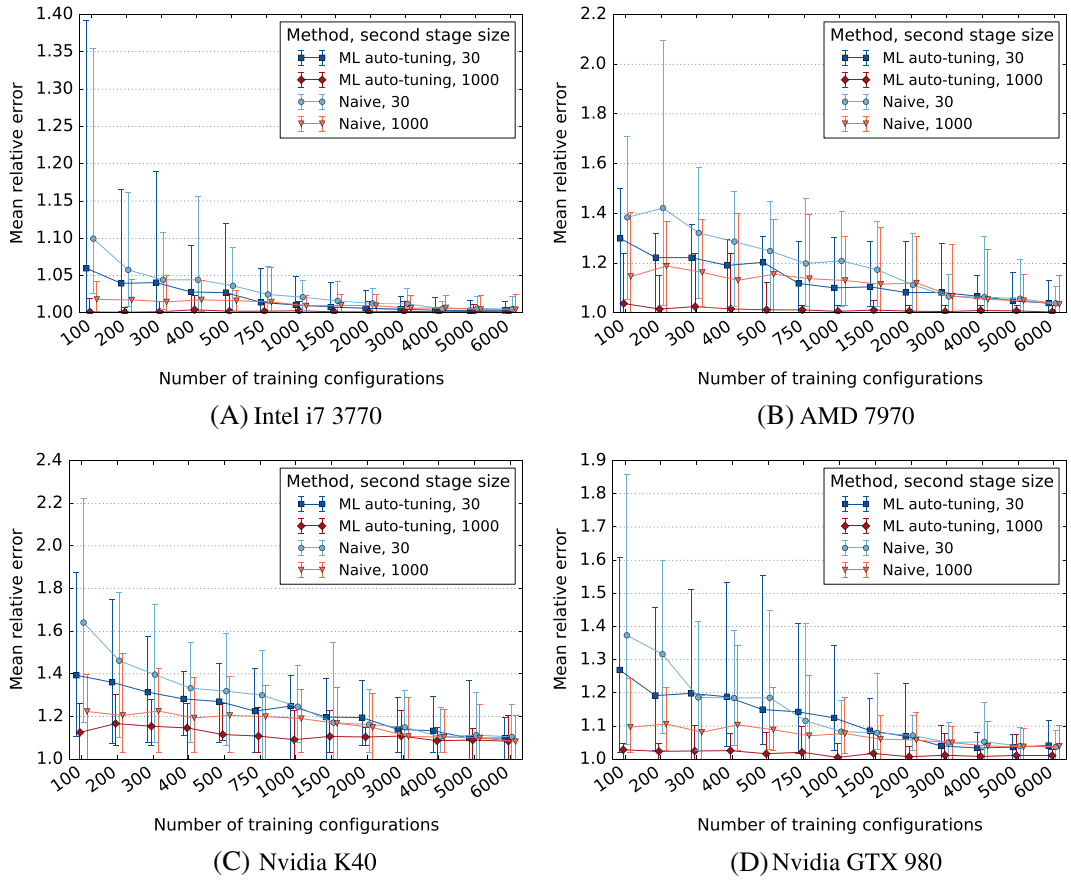


FIGURE 10 Comparison with naive auto-tuning. Average slowdown of solution found compared with global minimum. Error bars indicate range

6.3 | Varying problem sizes

To compare the accuracy of a model trained for multiple problem sizes with that of dedicated models for each problem size, we modified the benchmarks so that the problem sizes could be varied and added as parameters. For each benchmark, we trained a single model for all the different problem sizes and compared the accuracy of this combined model against models trained for a single, fixed problem size. In particular, for each benchmark, we picked 2 different problem sizes and compared the accuracy of the combined model, against the accuracy of models trained only for those problem sizes. The 2 problem sizes used are shown in Table 5

Figure 11 shows results for the `convolution` and `bilateral` benchmarks, while Figure 12 shows results for `median` and `raycasting`. For each benchmark, we show 2 figures, comparing (the same) combined model against dedicated models for the 2 different problem sizes. We do not include results for the `stereo` benchmark because of time constraints and its large parameter space.

As these figures show, the combined model achieves good results but is still worse than the dedicated models. We also notice that the difference is larger with few training configurations but shrinks as more training data are used, so their accuracies becomes more equal. The most notable exception is for the AMD GPU on the `bilateral` benchmark, where the dedicated model performs much better than the combined. These results suggest that using a single model for multiple problem sizes is a viable approach. While more training data might be required for sufficient accuracy compared with a single dedicated model, this will be compensated for by the models' ability to replace multiple dedicated models.

7 | DISCUSSION

Some aspects of the results warrant further discussion. Firstly, the model accuracy varies significantly between the different devices, with the CPUs outperforming the GPUs. For the CPUs, the mean relative errors lie between 3.8% and 8.1%, while the corresponding numbers are 9.3%-14.9% for the Nvidia GPUs, which again outperforms the AMD GPU, where the errors are between 10.0% and 17.8%. One possible explanation is the layout of the tuning parameter search space, shown in Figure 3 in Section 5.1.1. The execution times vary less for the CPUs than GPUs, possibly because the memory-related parameters have less effect on the CPUs, where all the logical memory spaces are mapped to the same physical memory, as described in Section 3. This might make prediction easier. Furthermore, again because local memory is physically located in the systems' main memory, which is typically larger than on-chip memory on GPUs, there are fewer invalid configurations on the CPU, which may contribute towards increased accuracy. Finally, the execution times on the CPU are generally longer, potentially making the timing measurements more reliable. As for the difference between Nvidia and AMD, this may also be caused by the differences in search space layout, because the search spaces for AMD are more varied, with more very fast and very slow configurations. The optimization parameters may therefore have larger effect and be harder to predict.

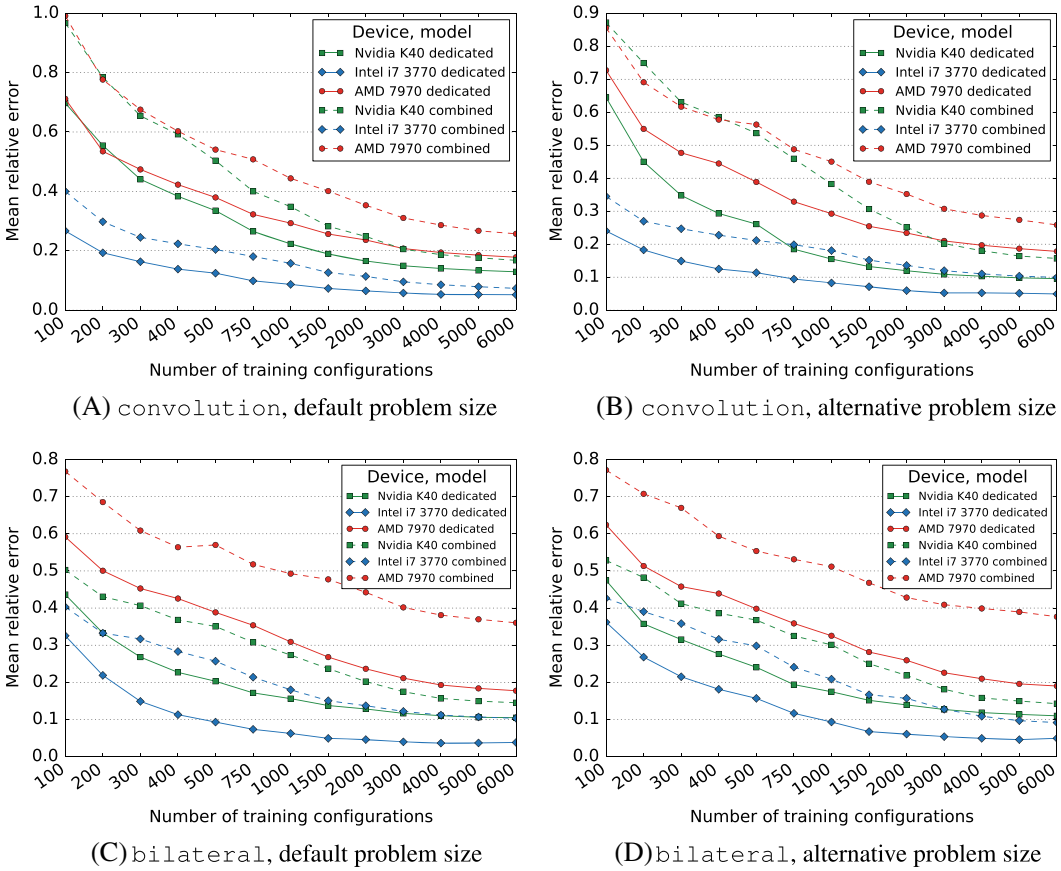


FIGURE 11 Mean error of combined model for several problem sizes compared to dedicated model for each problem size, for convolution and bilateral benchmarks

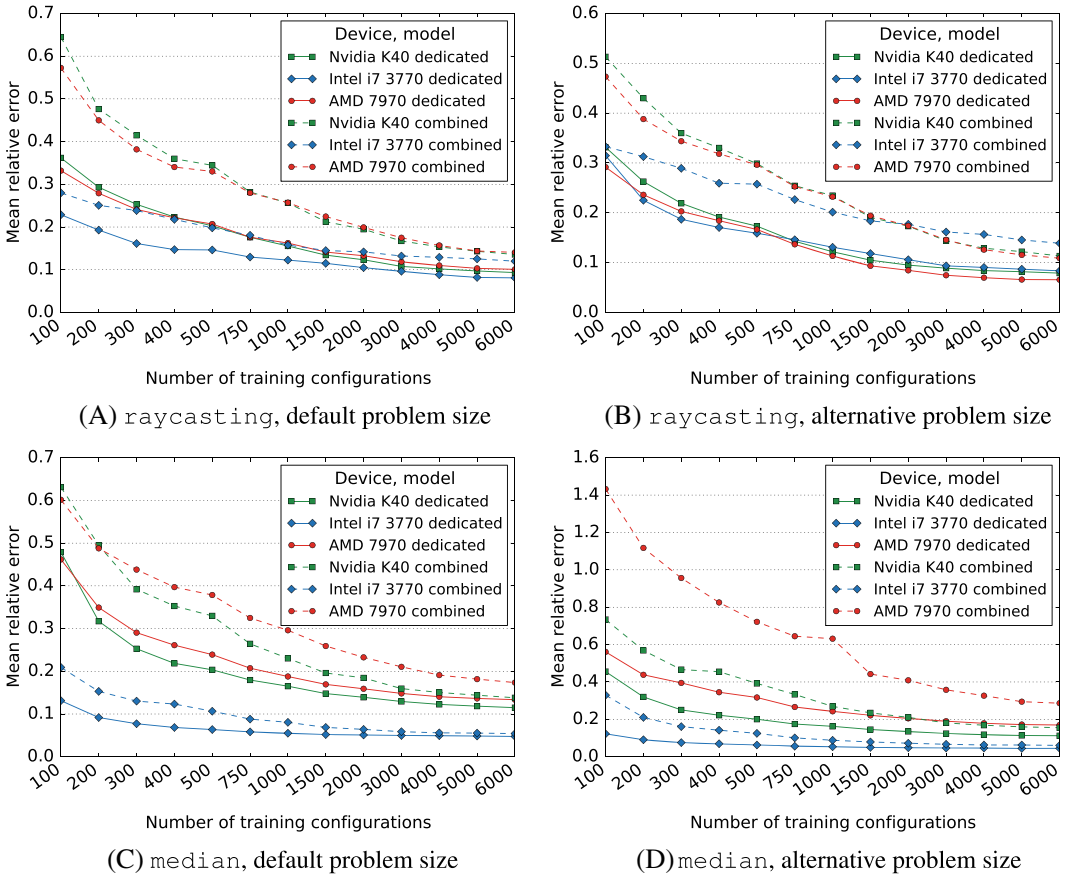
As for the variation in accuracy between the different benchmarks, there are no clear patterns. In particular, while the GPUs perform best on the `raycasting` benchmark, making this the benchmark with the best average accuracy across the different devices, this is also the benchmark where the CPU performs worst. On the other hand, the GPUs perform worst on the `convolution` benchmark, making this the benchmark with the worst average accuracy across the different devices. However, here, the CPU performs average. Furthermore, there does not appear to be any clear and direct relationship between the number of tuning parameters, or the tuning parameter size and the accuracy achieved.

Concerning the auto-tuning results, one thing that stands out is the good performance of the `bilateral` benchmark, where the best solution found; even using the 70% threshold is no more than 1.5% slower than the globally best configuration for all devices, with only 1000 configurations as training data. This may be because `bilateral` has a relatively high fraction of good performing configurations, as shown in Figure 3 in Section 5.1.1. Furthermore, a comparatively large number of second stage configurations are used for `bilateral`. This may be because the assumptions of the second-stage calculations are particularly bad in this case, so that more configurations are included than the accuracy of the model really calls for.

In general, the relationship between the best results is found by our auto-tuner, and the second stage threshold is currently more unpredictable than desired. Ideally, a fixed threshold should give results of the same quality, independent of the accuracy of the model, by including more configurations in the second stage for inaccurate models. The reason why we only partially see this may be caused by the many assumptions that go into calculating which configurations should be included for a given threshold, particularly related to estimating the error, and error distribution. Furthermore, the layout of the tuning parameter space is likely to have a large effect, which should also be taken into consideration.

Another issue relates to the software stack. In particular, we observed that upgrading the driver and CUDA toolkit version for the Nvidia K40 caused two changes: (1) execution times generally improved and (2) the best configuration changed. That is, upgrading the software would require a new round of auto-tuning to be performed. We suspect that this applies to the other devices as well. This is not surprising, because different compiler versions might perform different kinds of optimizations on the source code. As OpenCL compilers mature, and the changes between versions become more incremental, this problem will likely be reduced but should be kept in mind by GPU developers for the time being.

Finally, the time required to train the ANN models is significant, but small compared with the cost of gathering the training data. As an example, for the `convolution` benchmark on the Nvidia K40 GPU, training the model with 2000 samples takes about 1 minute and gathering the data takes about 30 minutes. The time required to gather data is so high because it does include not only the execution time of the kernels themselves but also the overhead of compiling the kernels, as well as the time wasted in attempting to compile and launch the kernels with



(A) `raycasting`, default problem size

(B) `raycasting`, alternative problem size

(C) `median`, default problem size

(D) `median`, alternative problem size

FIGURE 12 Mean error of combined model for several problem sizes compared to dedicated model for each problem size, for raycasting and median benchmarks

invalid configurations. Thus, like any auto-tuning scheme, we add some overhead. However, for code executed many times, this can easily be justified because of the good configurations found.

Despite the limitations and issues discussed in this section, our auto-tuning framework is able to overcome the problem of poor OpenCL performance portability. We are able to take a single OpenCL program and make it achieve good performance on different devices. In particular, we are able to find tuning parameter configurations that are close in performance to the globally optimal ones, without time-consuming exhaustive search or manually derived performance models.

8 | CONCLUSION AND FUTURE WORK

OpenCL has emerged as a programming standard for heterogeneous systems. While it offers functional portability, performance portability remains problematic, and code must almost always be retuned when moved from one device to another to achieve good performance. To address this problem, we have developed and tested a machine learning-based auto-tuning framework for OpenCL. The framework measures the execution time of several candidate implementations from a tuning parameter configuration space and uses the results to build an ANN, which works as a performance model. This model is then used to find interesting parts of the configuration space, which are explored exhaustively to find the best implementation. The size of this subspace is determined using a probabilistic model. Our approach can be used to handle different problem sizes with a single performance model. Our neural network model achieves a mean relative error as low as 3.8% for 5 different image processing benchmarks executed on different devices, including an Intel i7 4771 CPU (Haswell), an Nvidia Tesla K40 GPU, and an AMD Radeon HD 7970 GPU. The auto-tuner is able to find good configurations, in some cases, on average only 0.29% slower than the globally best configuration, after evaluating less than 1.1% of the search space. These results indicate that machine learning-based auto-tuning has promise as a way to overcome OpenCL performance portability.

Future work may include enhancing the accuracy of the model, improving and getting a better understanding of how to control the size of the second stage, and evaluating the method on novel hardware architectures, beyond just CPUs and GPUs. We have also developed, and may continue to improve a high level language from which one can automatically generate parameterized OpenCL code.[61] How to use some of these techniques for enhancing one or more of the use cases in the EU Horizon 2020 CloudLightning project we are a part of will also be investigated.[62] Furthermore, adding more parameters, including advanced new features specific to a given architecture,[63] may be an interesting possibility.

ACKNOWLEDGEMENTS

The authors would like to thank NTNU through IME and MedTech for their support of the Medical Computing & Visualization project,

Nvidias GPU Research Center program and NTNU for hardware donations to our NTNU/IDI HPC-Lab, and Drs. Malik M. Z. Khan and Frank Lindseth for their helpful discussions. The latter stages of this work was also partially funded by the European Union's Horizon 2020 Research and Innovation Programme through the CloudLightnig project (http://www.cloudlightning.eu) under Grant Agreement Number 643946. The authors would also like to thank Dr. Keshav Pingali and ICES at the University of Texas at Austin for hosting us during parts of the work for this article.

REFERENCES

1. KHRONOS Group. OpenCL – the open standard for parallel programming of heterogeneous systems. Available from: http://www.khronos.org/opencl/ [Acessed 15.01.2015]; 2015.
2. Whaley RC, Petitet A. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Softw Pract Exp*. 2005;35(2):101–121.
3. Vuduc R, Demmel JW, Yelick KA. OSKI: a library of automatically tuned sparse matrix kernels. *J Phys Conf Ser*. 2005;16(1):521–530.
4. Suda R, Naono K, Teranishi K, Cavazos J. *Software Automatic Tuning: Concepts and State-of-the-Art Results*. Berlin Heidelberg: Springer; 2011.
5. Yotov K, Li X, Ren G, Cibulskis M, DeJong G, Garzaran M, Pardua D, Pingali K, Stodghill P, Wu P. A comparison of empirical and model-driven optimization. *SIGPLAN Not*. 2003;38(5):63–76.
6. Bergstra J, Pinto N, Cox D. Machine learning for predictive auto-tuning with boosted regression trees. *Innovative Parallel Computing (INPAR), 2012*, San Jose, CA; 2012 May:1–9.
7. Ganapathi A, Datta K, Fox A, Patterson D. A case for machine learning to optimize multicore performance. *First Usenix Workshop on Hot Topics in Parallelism (HotPar09)*, USENIX Association, Berkeley, CA, USA; 2009:1–1.
8. Falch TL, Elster AC. Machine learning based auto-tuning for enhanced OpenCL performance portability. *International Parallel and Distributed Processing Symposium Workshops, IWAPT 2015, 2015 IEEE*, Hyderabad; 2015:1231–1240.
9. Owens J, Houston M, Luebke D, Green S, Stone JE, Phillips JC. GPU computing. *Proc of the IEEE*. 2008;96(5):879–899.
10. Brodtkorb AR, Dyken C, Hagen TR, Hjelmervik JM, Storaasli OO. State-of-the-art in heterogeneous computing. *Sci Program*. 2010;18(1):1–33.
11. Smistad E, Falch TL, Bozorgi M, Elster AC, Lindseth F. Medical image segmentation on GPUs–a comprehensive review. *Med Image Anal*. 2015;20(1):1–18.
12. Alpaydin E. *Introduction to Machine Learning*. Cambridge: MIT Press; 2010.
13. Kulkarni S, Cavazos J. Mitigating the compiler optimization phase-ordering problem using machine learning. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12. New York, NY, USA: ACM; 2012:147–162.
14. Pellegrini S, Wang J, Fahringer T, Moritsch H. Optimizing MPI runtime parameter settings by using machine learning. In: Ropo Matti, Westerholm Jan, Dongarra Jack, eds. *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg; 2009:196–206.
15. Magni A, Dubach C, O'Boyle M. Automatic optimization of thread-coarsening for graphics processors. *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14. New York, NY, USA: ACM, 2014:455–466.
16. Ipek E, de Supinski BR, Schulz M, McKee SA. An approach to performance prediction for parallel applications. In: Cunha J, Medeiros P, eds. *Euro-par 2005 Parallel Processing*, Lecture Notes in Computer Science, vol. 3648. Berlin Heidelberg: Springer; 2005:196–205.
17. Li J, Ma X, Singh K, Schulz M, de Supinski BR, McKee SA. Machine learning based online performance prediction for runtime parallelization and task scheduling. *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*; 2009 April:89–100.
18. Yigitbasi N, Willke TL, Liao G, Epema D. Towards machine learning-based auto-tuning of MapReduce. *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on*; 2013 Aug:11–20.
19. Singh K, Ipek E, McKee SA, de Supinski BR, Schulz M, Caruana R. Predicting parallel application performance via machine learning approaches. *Concurr Comp Pract E*. 2007;19(17):2219–2235.
20. Fursin G, Kashnikov Y, Memon AW, Chamski Z, Temam O, Namolaru M, Yom-Tov E, Mendelson B, Zaks A, Courtois E, Bodin F, Barnard P, Ashton E, Bonilla E, Thomson J, Williams CKI, O'Boyle M. Milepost GCC: machine learning enabled self-tuning compiler. *Int J Parallel Prog*., Berlin Heidelberg: Springer; 2011;39(3):296–327.
21. Garvey JD, Abdelrahman TS. Automatic performance tuning of stencil computations on GPUs. *Parallel Processing (ICPP), 2015 44th International Conference on*, Beijing; 2015 Sept:300–309.
22. Cummins C, Petoumenos P, Steuwer M, Leather H. Autotuning OpenCL workgroup size for stencil patterns. *arXiv preprint arXiv:1511.02490*. 2015.
23. Jia W, Shaw KA, Martonosi M. Starchart: hardware and software optimization using recursive partitioning regression trees. *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13. Piscataway, NJ, USA: IEEE Press; 2013:257–268.
24. Liu Y, Zhang EZ, Shen X. A cross-input adaptive framework for GPU program optimizations. *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, Rome; 2009 May:1–10.
25. Grewe D, Wang Z, O'Boyle MFP. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*; 2013:1–10.
26. Han TD, Abdelrahman TS. Automatic tuning of local memory use on GPGPUs. *arXiv preprint arXiv:1412.6986*. 2014.
27. Balaprakash P, Gramacy RB, Wild SM. Active-learning-based surrogate models for empirical performance tuning. *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*; 2013Sept:1–8.
28. Stephenson M, Amarasinghe S. Predicting unroll factors using supervised classification. *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, IEEE; 2005:123–134.
29. Agakov F, Bonilla E, Cavazos J, Franke B, Fursin G, O'Boyle MFP, Thomson J, Toussaint M, Williams CKI. Using machine learning to focus iterative optimization. *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06. Washington, DC, USA: IEEE Computer Society; 2006:295–305.
30. Magni A, Grewe D, Johnson N. Input-aware auto-tuning for directive-based GPU programming. *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6. Houston, Texas, USA: ACM, 2013:66–75.
31. Vuduc R, Demmel JW, Bilmes J. Statistical models for automatic performance tuning. In: Alexandrov VN., Dongarra JJ, Juliano BA, Renner RS, Tan CJK, eds. *Computational Science ICCS 2001*, Lecture Notes in Computer Science, vol. 2073. Berlin Heidelberg: Springer; 2001:117–126.
32. Cavazos J, Fursin G, Agakov F, Bonilla E, O'Boyle MFP, Temam O. Rapidly selecting good compiler optimizations using performance counters. *International symposium on code generation and optimization (cgo'07)*; 2007 March:185–197.
33. Ogilvie WF, Petoumenos P, Wang Z. Active learning accelerated automatic heuristic construction for parallel program mapping. *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14. ACM; 2014:481–482.

34. Grasso I, Kofler K, Cosenza B, Fahringer T. Automatic problem size sensitive task partitioning on heterogeneous parallel systems. *SIGPLAN Not*. 2013;48(8):281–282.
35. Frigo M, Johnson SG. The design and implementation of FFTW3. *Proc of the IEEE*. Piscataway, NJ, USA: IEEE 2005;93(2):216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".
36. Elster AC, Meyer JC. A super-efficient adaptable bit-reversal algorithm for multithreaded architectures. *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*; 2009 May:1–8.
37. Jensen RE, Karlin I, Elster AC. Auto-tuning a matrix routine for high performance. *Norsk Informatikkonferanse*, Tromso, Norway, 2011;2011.
38. Zhang Y, Mueller F. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12. ACM, San Jose, CA, USA; 2012:155–164.
39. Li Y, Dongarra J, Tomov S. A note on auto-tuning GEMM for GPUs. *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09. Springer-Verlag, Berlin, Heidelberg; 2009:884–892.
40. Nukada A, Matsuoka S. Auto-tuning 3-D FFT library for CUDA GPUs. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09. Portland, OR, USA: ACM; 2009:30:1–30:10.
41. Khan M, Basu P, Rudy G, Hall M, Chen C, Chame J. A script-based auto-tuning compiler system to generate high-performance CUDA code. *ACM Trans Archit Code Optim*. 2013;9(4):31:1–31:25.
42. Meyer JC. Performance modeling of heterogeneous systems. *Ph.D. Thesis*, Norwegian University of Science and Technology, Trondheim, Norway; 2012.
43. Meyer JC, Elster AC. Performance modeling of heterogeneous systems. *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*; 2010 April:1–4.
44. Meyer JC, Elster AC. BSP as a performance predictor on heterogeneous interconnect platforms. *Proceedings of 9th Intl. Workshop on State of the Art in Scientific Computing (PARA 2008)*, Berlin Heidelberg: Springer LNCS; 2008.
45. Hong S, Kim H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Computer Architecture News*, vol. 37, New York, NY, USA: ACM; 2009:152–163.
46. Monsifrot A, Bodin F, Quiniou R. A machine learning approach to automatic production of compiler heuristics. *Artificial Intelligence: Methodology, Systems, and Applications: 10th International Conference, AIMSA 2002 Varna, Bulgaria, September 4–6, 2002 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg; 2002:41–50.
47. Datta K, Murphy M, Volkov V, Williams S, Carter J, Oliker L, Patterson D, Shalf J, Yelick K. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Austin, TX, USA: IEEE Press; 2008:4.
48. Apache Software Foundation. Welcome to Apache Hadoop. Available from: http://hadoop.apache.org [Acessed 24.04.2016]; 2016.
49. Nugteren C, Codreanu V. CLTune: a generic auto-tuner for OpenCL kernels. *Embedded Multicore/Manycore Systems-on-Chip (MCSoC-15), IEEE 9th International Symposium on*, IEEE; 2015:195–202.
50. Zhang Y, Sinclair II M, Chien AA. Improving performance portability in OpenCL programs. In: Kunkel JM, Ludwig T, Meuer HW, eds. *Supercomputing*, Lecture Notes in Computer Science, vol. 7905. Berlin Heidelberg: Springer; 2013:136–150.
51. Fabeiro JF, Andrade D, Fraguela BB. OCLoptimizer: an iterative optimization tool for OpenCL. *Procedia Comput Sci*. 2013;18(0):1322–1331. 2013 International Conference on Computational Science.
52. Pennycook SJ, Hammond SD, Wright SA, Herdman JA, Miller I, Jarvis SA. An investigation of the performance portability of OpenCL. *J Parallel Distrib Comput*. 2013;73(11):1439–1450. Novel architectures for high-performance computing.
53. Smistad E, Bozorgi M, Lindseth F. FAST: framework for heterogeneous medical image computing and visualization. *Int J Comput Assist Radiol Surg*. 2015;10(11):1–12.
54. Breiman L. Bagging predictors. *Mach Learn*. 1996;24(2):123–140.
55. Freund Y, Schapire RE. Experiments with a new boosting algorithm. *ICML*, vol. 96; 1996:148–156.
56. Cohn D, Atlas L, Ladner R. Improving generalization with active learning. *Mach Learn*. 1994;15(2):201–221.
57. Kennedy J. *Encyclopedia of Machine Learning*. Boston, MA: Springer US; 2010;760–766.
58. Yang XS. *Stochastic Algorithms: Foundations and Applications: 5th International Symposium, SAGA 2009, Sapporo, Japan, October 26-28, 2009. Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg; 2009;169–178.
59. Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional neural networks.. In: Pereira F, Burges CJC, Bottou L, Weinberger KQ, eds. *Advances in Neural Information Processing Systems 25*. Red Hook, NY, USA: Curran Associates, Inc.; 2012:1097–1105.
60. Silver D, Huang A, Maddison CJ, Guez A, Sifre L, van den Driessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M, Dieleman S, Grewe D, Nham J, Kalchbrenner N, Sutskever I, Lillicrap T, Leach M, Kavukcuoglu K, Graepel T, Hassabis D. Mastering the game of go with deep neural networks and tree search. *Nature*. 2016;529(7587):484–489.
61. Falch TL, Elster AC. ImageCL: An image processing language for performance portability on heterogeneous systems. *2016 International Conference on High Performance Computing & Simulation (HPCS)*, Innsbruck, Austria: IEEE; 2016: 562–569.
62. Lynn T, Xiong H, Dong D, Momani B, Gravvanis G, Filelis-Papadopoulos C, Elster A, Khan M, Tzovaras D, Giannoutakis K, Petcu D, Neagul M, Dragon I, Kuppudayar P, Natarajan S, McGrath M, Gaydadjiev G, Becker T, Gourinovitch A, Kenny D, Morrison J. CLOUDLIGHTNING: A framework for a self-organising and self-managing heterogeneous cloud. *In Proceedings of the 6th International Conference on Cloud Computing and Services Science*; 2016: 333–338.
63. Falch TL, Elster AC. Register caching for stencil computations on GPUs. *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2014 16th International Symposium on*, Timisoara, Romania: IEEE; 2014:479–486.

How to cite this article: Falch TL, Elster AC. Machine learning based auto-tuning for enhanced performance portability of OpenCL applications. *Concurrency Computat: Pract Exper*. 2017;29:e4029. https://doi.org/10.1002/cpe.4029