Fast Automatic Heuristic Construction Using Active Learning

William F. Ogilvie[1], Pavlos Petoumenos[1], Zheng Wang[2], and Hugh Leather[1]

[1] School of Informatics, University of Edinburgh, UK
{s0198982, ppetoume, hleather}@inf.ed.ac.uk
[2] School of Computing and Communications, Lancaster University, UK
z.wang@lancaster.ac.uk

Abstract. Building effective optimization heuristics is a challenging task which often takes developers several months if not years to complete. Predictive modelling has recently emerged as a promising solution, automatically constructing heuristics from training data. However, obtaining this data can take months per platform. This is becoming an ever more critical problem and if no solution is found we shall be left with out of date heuristics which cannot extract the best performance from modern machines.

In this work, we present a low-cost predictive modelling approach for automatic heuristic construction which significantly reduces this training overhead. Typically in supervised learning the training instances are randomly selected to evaluate regardless of how much useful information they carry. This wastes effort on parts of the space that contribute little to the quality of the produced heuristic. Our approach, on the other hand, uses active learning to select and only focus on the most useful training examples.

We demonstrate this technique by automatically constructing a model to determine on which device to execute four parallel programs at differing problem dimensions for a representative Cpu–Gpu based heterogeneous system. Our methodology is remarkably simple and yet effective, making it a strong candidate for wide adoption. At high levels of classification accuracy the average learning speed-up is 3x, as compared to the state-of-the-art.

Keywords: machine learning, workload scheduling

1 Introduction

Building effective program optimization heuristics is a daunting task because modern processors are complicated; they have a large number of components operating in parallel and each component is sensitive to the behaviour of the others. Creating analytical models on which optimization heuristics can be based has become harder as processor complexity has increased, and this trend is bound to continue as processor designs move further towards heterogeneous parallelism [1]. Compiler developers often have to spend months if not years to get a heuristic right for a targeted architecture, and these days compilers often support a wide range of disparate processors. Whenever a new processor comes out, even if derived from a previous one, the optimizing heuristics need to be re-tuned for it. This is typically too much effort and so, in fact, most compilers are out of date [2].

Machine Learning based predictive modelling has rapidly emerged as a viable means to automate heuristic construction; by running example programs (optimized in different ways) and observing how the variations affect program run-time automatic machine learning tools can predict good settings with which to compile new, as yet unseen, programs. There are many studies showing that machine learning outperforms human based approaches [2, 3]. Recent work also illustrates that it can be used to automatically port across architecture spaces [4] and can find more appropriate ways of mapping program parallelism to various platforms [5]. This new research area is promising, having the potential to fundamentally change the way compiler heuristics are designed; that is to say, compilers can be automatically tuned for new hardware without the need for months of compiler experts' time; however, before the potential of predictive modelling based heuristic construction can be realized there remain many hurdles which must be tackled. One major concern is the cost of collecting training examples. While machine learning allows us to automatically construct heuristics with little human involvement, the cost of generating training examples (that allow a learning algorithm to accumulate knowledge) is often very expensive.

This paper presents a novel, low-cost predictive modelling approach that can significantly reduce the overhead of collecting training examples without sacrificing prediction accuracy. Traditionally in predictive modelling training examples are randomly selected for labelling where, in the context of machine learning based compilers and run-time systems, labelling involves profiling code under varying conditions. This is inefficient because random selection often provides redundant data to the learner. In effect a cost is paid for training but little or no benefit is actually received. We tackle this problem by using *active learning* [6] to select and only focus on useful training instances, which greatly reduces the training overhead. Specifically, we build a number of initial distinct models with a small set of randomly selected training examples. We ask those models to make predictions on unseen data points, and the points for which the models 'disagree' the most are profiled. We then rebuild the models by re-running the learning algorithm with the new training example together with the existing ones, and repeat this process until a completion criterion is met after which a final heuristic is produced. In this way, we profile and collect training examples that provide the most information to the algorithm, thereby enabling it to improve prediction accuracy of the learned models more quickly.

We demonstrate the effectiveness of our approach by using active learning to automatically construct a heuristic to determine which processor will give the better performance on a Cpu–Gpu based heterogeneous platform at differing problem sizes for a given program. More specifically, our approach is evaluated by building heuristics to predict the better processor to use for 4 benchmarks which have equivalent OpenMp and OpenCl implementations; where OpenMp is used for the Cpu since it has a more mature implementation than OpenCl. Comparing our work to a typical random sampling technique, widely used in prior work, reveals that our methodology speeds up training by a factor of 3x on average: saving weeks of intensive compute time.

The research presented in this paper makes the following contributions. It

– shows that the training overhead of machine learning based compiler heuristics design can be significantly reduced without sacrificing prediction accuracy;
– demonstrates how active learning can be used to automatically derive a heuristic to map OpenMp and OpenCl programs on a Cpu–Gpu based heterogeneous platform;
– provides detailed analysis of active learning based heuristic tuning.

The rest of this paper is organized as follows: in Sect. 2 we give a motivating example for this research, in Sect. 3 we discuss our approach and the implementation details of our system, in Sect. 4 we outline the methodology used to validate our technique, Sect. 5 provides our results and accompanying analysis, Sect. 6 references related work, and we conclude in Sect. 7.

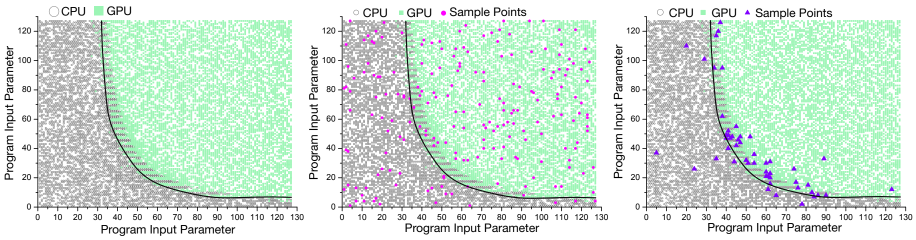
2 Motivation

To motivate our work, we demonstrate how much unnecessary effort is involved in the traditional random-sampling based learning techniques, and point out the extent to which a better strategy can improve matters. In Fig. 1(a) we show for `HotSpot`, from the Rodinia [7,8] suite, when it is better to run on the Cpu *versus* the Gpu for maximum performance. The benchmark accepts two independent program inputs, and these form the axes of the graph. The graph data itself was generated by randomly selecting 12,000 input combinations and running them on both the Cpu and Gpu enough times to make a statistically sound decision about which device is better for each, where a boundary line separates the regions at which either device should be chosen.

Machine learning has been shown to be a viable option for creating heuristics for this type of problem [9, 10]. To build such a heuristic, a machine learning algorithm typically requires a set of training examples to learn from. In our case, we need to use a set of profiled program inputs to find a model that is a good estimate of the boundary as shown in Fig. 1(a). The quality of the training examples will have a significant impact on the accuracy of the resultant model.

In Fig. 1(b) a random selection of 200 inputs to `HotSpot` is chosen, as might be typical in a standard 'passive' learning technique[3]. From this data a heuristic is created with the RandomCommittee machine learning algorithm from the

[3] In passive learning techniques, the training examples are selected without feedback as to the quality of the machine learned heuristic. Most usually, this will mean that all training examples are generated ahead of time and then a heuristic is learned once. In active learning, by contrast, the selection of training examples is an iterative process which is driven by feedback about the quality of the heuristic.



(a) The problem space (b) Random sample points (c) Ideal sample points

Fig. 1: Passive learning over randomly selected inputs versus learning from ideally selected inputs. Figure (a) shows the problem space of the Rodinia `HotSpot` benchmark. 12,000, 2-dimensional program inputs are run to discover which device (Cpu or Gpu) gives the better performance. A boundary line separates the parts of the space where Cpu and Gpu are better. Figure (b) shows a random selection of 200 inputs. Using RandomCommittee to learn a heuristic with these inputs achieves an accuracy of 95%. Figure (c) shows an ideal selection of 50 inputs near to the boundary line. Using RandomCommittee to learn a heuristic with these inputs achieves an accuracy of 97%, representing a 4x speed-up in training time.

Weka tool-kit [11], and the heuristic achieves a respectable 95% accuracy. Machine learning can clearly learn good heuristics in this case, but our intuition insists that the majority of the randomly selected inputs offer little useful information. In fact, we would expect that only those points near to the boundary line in Fig. 1(a) should be required to accurately define a model.

We prove this intuition in Fig. 1(c) where we have instead selected just 50 inputs close to the boundary line and once again asked the RandomCommittee algorithm to learn a heuristic. Using fewer than 15% as many observations as the standard passive learning technique we achieve an accuracy of 97%. There is, therefore, significant potential to reduce the training cost for the machine learned heuristics if we could only choose the right inputs to train over. Unfortunately, without already knowing the shape of the space it is impossible to tell what the best inputs should be, but nevertheless we will show that it is possible to approximate their location.

In this paper we present a simple active learning technique that maintains a set of training inputs, adding to the set incrementally by selecting inputs that look likely to improve the heuristic quality based on what has already been seen. For the `HotSpot` benchmark, our approach avoids nearly all of the unimportant inputs, quickly focussing in on the best inputs to choose. Our active learning method needs only 31 inputs to create heuristics as accurate as passive learning generates with 200 inputs; a reduction in training cost of 85%. The following section describes our methodology in detail.

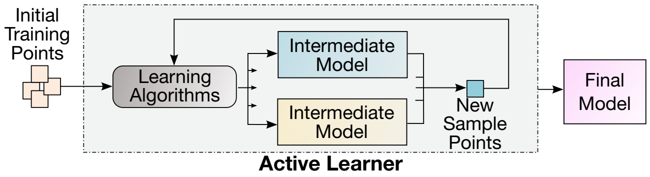


Fig. 2: An overview of our active learning approach. Initially, we use a few random samples to construct several intermediate models. Those models are utilized to choose which new data point is to be profiled. The new sampled data point is then used to update the models. We repeat this process until a certain termination criterion is met where a final model will be produced as the outcome.

3 Our Approach

As a case study, this work aims to learn a predictor to determine the best processor to use for a given program input. We wish to avoid profiling inputs that provide little or no information for the learning algorithm to train over so that we can minimize the overhead of collecting training examples. We achieve this by using active learning which carefully chooses each input to be profiled in turn. At each step, our algorithm attempts to choose a new input that will most improve the machine learned heuristic when it is added to the training set of examples.

Figure 2 provides an overview of how our approach can be applied to this case study problem. First, some number of program inputs are chosen at random to 'seed' the algorithm and these are then profiled to determine the better device for them – CPU or GPU. What follows is a number of steps which progressively add to the set of training inputs until some termination criterion are met. To select which program input to add to the training set for profiling, a number of different, intermediate models are created using the current training set and different machine learning algorithms. Our method then searches for an input for which the intermediate models or heuristics most disagree as to whether it should be run on the CPU or the GPU. The intuition is that the more these models agree on an input, the less likely it is able to improve the prediction accuracy of the learned heuristic.

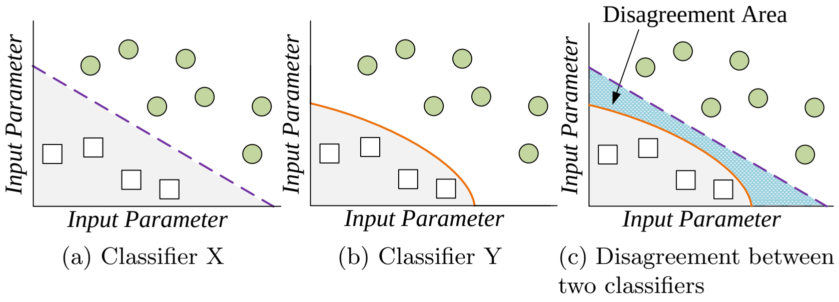
The technique for choosing new training inputs is called *Query by Committee* (QBC) [12] and is described in Sect. 3.1, whilst Sect. 3.3 details how the program training inputs are profiled: particularly, how the decision about whether the input should be run on the CPU or on the GPU is made statistically sound.

3.1 Query by Committee

The key idea behind active learning is that a machine learning algorithm can perform better with fewer training points if it is allowed to choose the data from which it learns. There are a number of approaches available [13] but we employ a heterogeneous implementation of the *Query by Committee* (QBC) algorithm, a widely utilized active learning technique, to select the most useful training examples from the input-space.

The QBC algorithm requires a group of distinct machine learning models (instead of just one) to be used. The 'committee' consists of a number of different learning algorithms that are initially trained with a small set of randomly collected training examples. In our case, those training examples are a set of profiled program inputs with a label indicating which processor gives better performance for each input. As those models are initially built from a small set of training examples, they are unlikely to be highly accurate. We will improve them with the following iterative steps using new training examples. The key point is how to only select the training examples (i.e. which program inputs to be profiled in our case) that are likely to improve the prediction accuracy. To do so, we ask each model in the committee to make predictions on a random *candidate set* of program inputs that are not present in the current training example set (and hence they haven't been profiled yet). As a result, different models may or may not reach consent for a particular program input. We then only profile those inputs for which the 'committee' *disagrees* the most to discover the true, best-performing processor, adding those new training examples into the training set, and re-running the learning algorithms to update the models. The justification for this is that we do not want to create new training instances from parts of the problem-space which are already understood by the committee of algorithms, but rather would like to sample those regions which are least well defined. The insight being that if we reduce the regions of disagreement between the committee members, by choosing training instances from within those regions, we incrementally get closer to the true boundary over which the processor choice should be altered and hence increase the accuracy of our final heuristic.

An Example: Figure 3 provides a hypothetical example to demonstrate how new training points are selected by QBC in our case. In Fig. 3(a) we are presented with an input-space which is fully described by two input parameters and has some training samples already shown. In this example, our committee consists of two different classification algorithms which will result in two classifiers that we will call `X` and `Y`. Based upon the location of these training examples in the space, and which device is faster under these conditions (represented by different shapes), the two different algorithms may give different models as illustrated in Fig. 3(a) and Fig. 3(b). If we overlap these classification boundaries of the two models, as in Fig. 3(c), we can see that there are parts of the space that classifiers X and Y are in agreement about and a region of disagreement. Knowing the disagreement regions, we then only select a new program input that both model disagree with to be profiled as our new training example. The question is which program input to choose? This will certainly require a metric to access the disagreement, which will be described in the next section.



(a) Classifier X (b) Classifier Y (c) Disagreement between two classifiers

Fig. 3: A simplified input space with two input parameters and the locations of profiled training examples. We use two different learning algorithms to build two different classifiers – (a) and (b). We then combine these models, as in (c), to find the region of disagreement between them and use this information to better choose where future training samples should be drawn from.

3.2 Assessing Disagreement

We use information entropy (1) [14] to evaluate the level of disagreement for each point that have not been profiled so far, where $p\left(x_i\right)$ is the proportion of committee members that predict that instance X is fastest on device i of n. The candidates with the maximum entropy value seen in each iteration of the learning loop are collected and a random candidate is chosen from within this high entropy subset as the next training example. This means that the inputs associated with the chosen candidate are run on the Cpu and Gpu kernels and it is determined which processor is faster under those input conditions. This new training example is added to the current training set and its inputs are removed form the candidate set to ensure the two remain disjoint. The learning loop begins another iteration with the models being formed with the addition of the new data.

$$H\left(X\right) = -\sum_{i=1}^{n} p(x_i) \log p(x_i) \tag{1}$$

3.3 Statistical Sounded Profiling

Since computer timings are inherently noisy we use statistics to increase the reliability of our models. In particular, we record a minimum number of timings from each device, as specified by the user. We use Interquartile Range [15] outlier removal then apply Welch's t-test [16] to discover if one hardware device is indeed faster than the other. If we cannot conclude from the t-test that this is the case, then we perform an equivalence test. Both devices are said to be 'equivalent' if the difference between the higher mean plus its 95% confidence interval minus the lower mean minus its confidence is within some threshold of indifference. In our system this threshold was set to be within 1% of the minimum of the two means. If the fastest device cannot be determined and they are not equivalent an extra set of observations are obtained and the tests applied again, up until some user defined number of tries. In the case of equivalence or of no determination being made within this threshold of attempts the Cpu is chosen as the preferred device since it is more energy-efficient.

4 Experimental Setup

This section describes the details of the experimental case studies that we undertook, starting with the platform and benchmarks used, moving on to the particular Qbc settings, and finally discussing the evaluation methodology.

4.1 Platform and Benchmarks

We evaluated our approach on a Cpu–Gpu based heterogeneous platform with a Intel Core i7 7770 4-core CPU (8 Hardware threads) @ 3.4GHz and a NVIDIA Geforce GTX Titan GPU (6 GB memory). The machine runs OpenSuse V12.3 Linux and we use gcc V4.7.2 and the NVIDIA CUDA Toolkit v5.5 for compilation. We used 3 benchmarks from the Rodinia suite, `HotSpot`, `PathFinder`, and `SRAD`, and we also included a simple matrix multiplication application. These benchmarks were specifically chosen because they had equivalent OpenCl and OpenMp versions and each has multiple program inputs which affect the dimensions of their respective problem-spaces.

Table 1: The sizes of the input-space for each benchmark. Each dimension has a value of between *Min* and *Max*, inclusive, and a step value of *Stride*. *Size* gives the total number of points in each input-space, and *Cand* is the number of points in the candidate set for each benchmark.

Benchmark	**#Dimentions**	**Min**	**Max**	**Stride**	**Size**	**Cand**
HotSpot	2	1	128	1	$16,384$	10,000
MatMul	3	1	256	1	$1.6x10^7$	10,000
Pathfinder	2	2	1024	1	$1.0x10^6$	10,000
SRAD	2	128	1024	16	$3,136$	2,636

4.2 Active Learning Settings

Machine Learning Models: Our active learning framework uses 12 unique algorithms from the Weka tool-kit to form the committee, each executed with default parameter values. They are `Logistic`, `MultilayerPerceptron`, `IB1`, `IBk`, `KStar`, `LogitBoost`, `MultiClassClassifier`, `RandomCommittee`, `NNge`, `ADTree`, `RandomForest`, and `RandomTree`. These were selected because they can produce a binary predictor from numeric inputs and have been widely used in prior work.

Program Input Space: The dimensions of the input-space for each benchmark were chosen to give realistic values to learn over – see Table 1.

Initial Training Set and Candidate Set Sizes: For all experiments the training set was initialised with a single randomly chosen instance – the minimum possible. The effect of changing this parameter is discussed in Sect. 5.3. The candidate set size was either 10,000 inputs not already present in the training and test sets or the maximum number of points not in the training and test sets, whichever was smaller – see Table 1.

Termination Criterion: The learning iterations were halted at 200 steps since it was found experimentally that the learning improvement had plateaued by that time.

4.3 Evaluation Methodology

Runtime Measurement and Device Comparison To determine if a benchmark is better suited to the CPU or GPU for a given input it is run on each device at least 10 times and at most 200 times. As mentioned in Sect. 3, we employ interquartile-range outlier removal, Welch's t-test, and equivalence testing to ensure the statistical soundness of the gathered program execution times.

Testing For testing purposes, a set of 500 inputs were excluded from any training and candidate sets. Both our active and passive learning experiments were run 10 times for each benchmark and the arithmetic mean of the accuracy (or other metrics) were recorded. For both active and passive learning, the accuracy was taken as the average accuracy of all 12 models. That is to say, we compared the average accuracy achieved using a 12-member QBC algorithm *versus* the same 12 algorithms trained using random data as the number of QBC-chosen or randomly-chosen training sets increased in size.

5 Experimental Results

In this section we begin by presenting the overall results of our experiments, showing that our active learning approach can significantly reduce the training time by a factor of 3 when compared to the random sampling technique. We then move on to examine the performance exhibited by our system for each benchmark in turn. Finally, we discuss how the change in two user supplied parameters (i.e. initial training set and candidate set sizes) can affect the performance of our methodology.

5.1 Overall Learning Costs

Figure 4 shows the average learning speed-up of our approach over the passive, random-sampling technique traditionally used in heuristic construction.

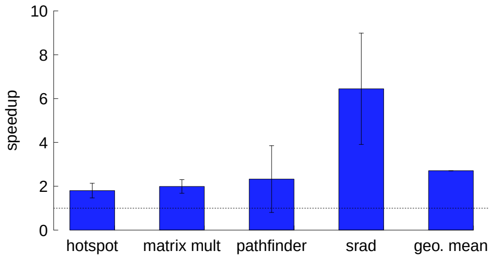


Fig. 4: On average our methodology requires 3x fewer training examples to create a high quality heuristic than the traditional random-sampling technique, proving that this simple algorithm can save weeks, and potentially months, of compute time.

The speed-up values are based on the number of inputs which need to be profiled in order to train a predictor to an accuracy of at least 85%. As can be seen from this figure, our approach constantly outperforms the classical random-sampling technique for all benchmarks, which in real terms means a saving of weeks to train these heuristics.

5.2 Analysis of Training Point Selection

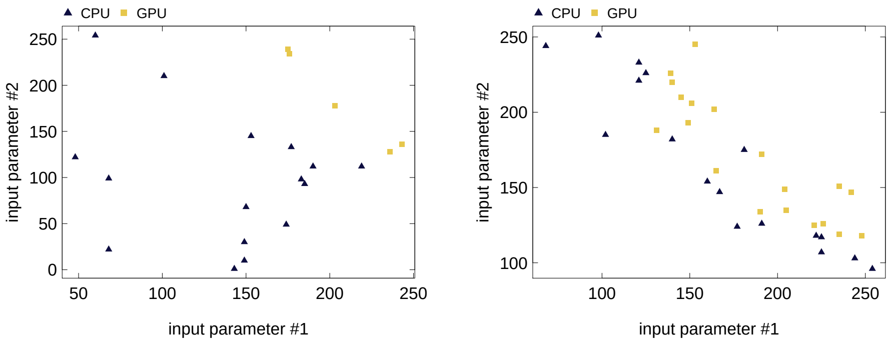
If we look at Figs. 5–8 we can see clearly where the cost savings associated with QBC are coming from. That is, in all cases the algorithm quickly chooses points surrounding the boundary between the CPU and the GPU optimum regions, giving it the ability to more accurately approximate its shape in less time.

5.3 Sensitivity to Parameters

As well as confirming the validity of our approach we also conducted two further experiments to determine the impact that some user defined parameters might have on the effectiveness of the system. The first experiment involved altering how many randomly selected training examples were initially supplied to the QBC algorithm to get it started. The second experiment investigated the extent to which changing the candidate set size would have an effect on the speed of heuristic construction. Results for both examinations are shown in Fig. 9 and Fig. 10, respectively.

In Fig. 9 it is clear that increasing the number of random training instances used to seed the QBC algorithm for `HotSpot` has no significant affect in the long-term performance but is detrimental in the short term, however, one can imagine a case where a complex space with many localized features may be better explored through an initially random approach followed-up by active learning.

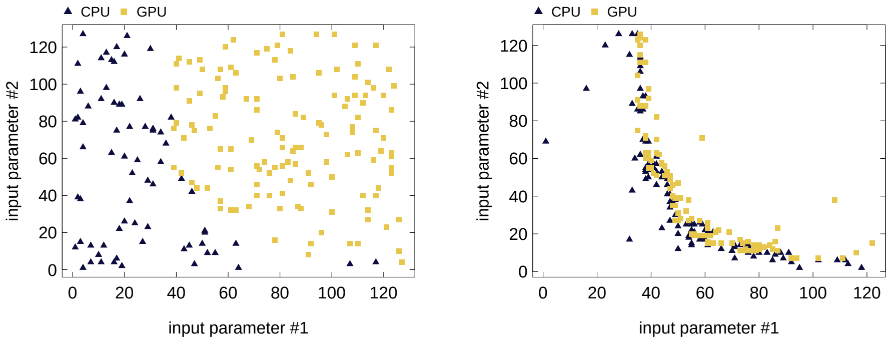
Figure 10 shows how changing the size of the candidate set for the `HotSpot` benchmark affects the performance of the system. In particular, the data indicates a lower candidate set size may be more beneficial. Presumably this is because a high candidate set increases the likelihood or the learner receiving redundant information from neighbouring high entropy points.



(a) Random Selection

(b) QBC Selection

Fig. 5: Since the `Matrix Multiplication` input-space is three-dimensional and not as simply defined as the other benchmarks it is difficult for a human to visualise the separation between CPU and GPU regions; to make it a little easier the graph above was flattened so that the z-axis has values $122 \leq z \leq 144$. However, active learning was over six times faster than random sampling at producing a high quality model for this code, quicker than the other programs tested and likely due to the additional dimension reducing the effectiveness of random selection.



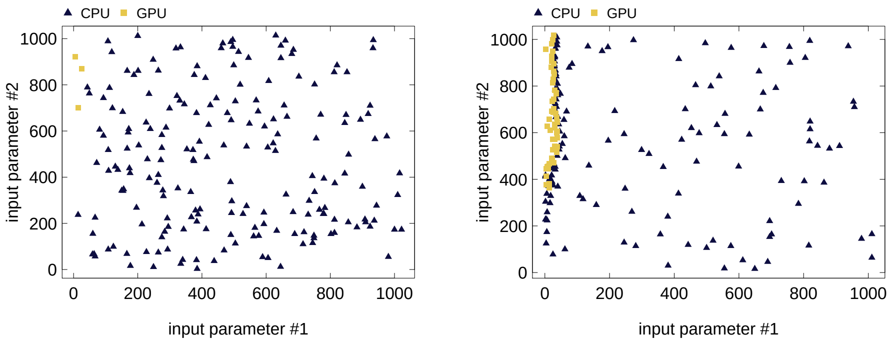
(a) Random Selection

(b) QBC Selection

Fig. 6: The difference between QBC and random sampling is stark for the `HotSpot` code. In particular, the QBC algorithm is able to quickly converge and define the boundary between the two devices whilst random selection trains on redundant or less informative points, proved by the fact it takes twice as long as QBC.

6 Related Work

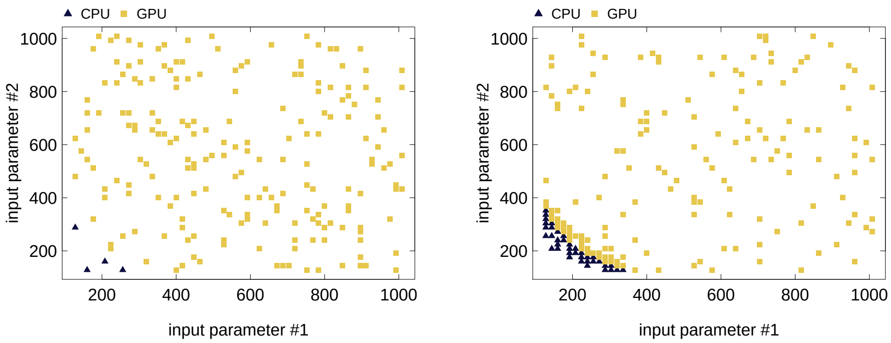
Analytic Modelling Analytic models have been widely used to tackle complex optimization problems, such as auto-parallelization [17, 18], runtime estimation [19–21], and task mappings [22]. A particular problem with them, however, is the model has to be re-tuned whenever it is targeted at new hardware [23].



(a) Random Selection

(b) QBC Selection

Fig. 7: The `PathFinder` QBC graph displays more randomness than the previous two. The probable reason for this, judging by the location of the boundary line, is that the active learner cannot initially locate the GPU region. Nevertheless, active learning is still twice as fast at generating a good quality heuristic compared with the random sampling technique.



(a) Random Selection

(b) QBC Selection

Fig. 8: Similarly, the `SRAD` input-space appears to show that QBC initially searches randomly because it has difficulty approximating the location of the comparitively small CPU region. However, once the algorithm has an idea of where this region is located it quickly concentrates on the boundary and forms a high-quality heuristic in half the time of the passive learning methodology.

Predictive Modeling Predictive modeling has been shown to be useful in the optimization of both sequential and parallel programs [9,10,24,25]. Its great advantage is that it can adapt to changing platforms as it has no *a priori* assumptions about their behaviour but it is expensive to train. There are many studies showing it outperforms human based approaches [2,3,26–30]. Prior work for machine learning in compilers, as being exemplified by MilePost GCC project [31], often uses random sampling or exhaustive search to collect training examples. The process of collecting training examples could be expensive, taking several

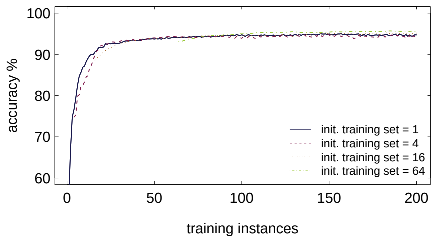


Fig. 9: This graph shows that increasing the number of random examples given initially to the QBC algorithm for `HotSpot` is at first detrimental to its performance, however, in a complex space increased randomness may help discover complex localized features.

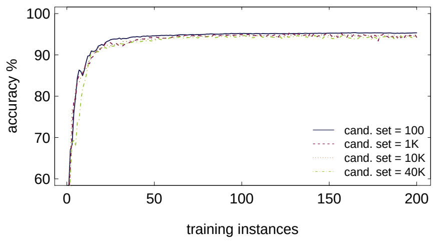


Fig. 10: This graph shows that choosing a lower candidate set size may be more beneficial than a larger one.

weeks if not months. Using active learning, our approach can significantly reduce overhead of collecting training examples. This accelerates the process of tuning optimization heuristics using machine learning. The Qilin compiler [32] uses runtime profiling to predict a parallel program's execution time and map work across the CPU and GPU accordingly. Our approach does not require run-time profiling and therefore avoids program slow-downs resulted from the potentially expensive runtime profiling.

Active Learning for Systems Optimization A recent paper by Zuluaga *et al.* [33] proposed an active learning algorithm to select parameters in a multi-objective problem. Their work is not concerned with single-objective workload scheduling and does not consider statistical soundness of raw data. Balaprakash *et al.* [34,35] used active learning to reduce execution time of scientific codes but they only consider code variants and OpenCl parameters as inputs; they do not discuss the impact of problem size on performance.

Problem Size Optimization Optimizing code for different problem sizes in heterogeneous systems is discussed by Liu *et al.* [36] where they give an implementation of a compiler which uses a combination of regression trees and representative Gpu kernels, but their approach uses exhaustive search. Adaptic is a compilation system for Gpus [37] and uses analytical models to map an input stream onto the Gpu at runtime but their technique is not easily portable, where ours tackles that problem directly by making learning cheaper.

7 Conclusions

We have presented a novel, low-cost predictive modelling approach for machine learning based automatic heuristic construction. Instead of building heuristics based on randomly chosen training examples we use active learning to focus on those instances that improve the quality of the resultant models the most. Using QBC to construct a heuristic to predict which processor to use for a given program input our approach speeds up training by a factor of 3x, saving weeks of compute time.

Acknowledgements

This work was funded under the EPSRC grant, ALEA (EP/H044752/1).

References

[1] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous System Coherence for Integrated CPU–GPU Systems," in *Proc. MICRO'13*.

[2] S. Kulkarni and J. Cavazos, "Mitigating the Compiler Optimization Phase-Ordering Problem using Machine Learning," in *Proc. OOPSLA'12*.

[3] C. Dubach, T. Jones, E. Bonilla, G. Fursin, and M. F. P. O'Boyle, "Portable Compiler Optimisation Across Embedded Programs and Microarchitectures using Machine Learning," in *Proc. MICRO'09)*.

[4] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam, "Rapidly Selecting Good Compiler Optimizations using Performance Counters," in *Proc. CGO'07*.

[5] D. Grewe, Z. Wang, and M. F. O'Boyle, "Portable Mapping of Data Parallel Programs to OpenCL for Heterogeneous Systems," in *Proc. CGO'13*.

[6] B. Settles, "Active Learning Literature Survey," University of Wisconsin–Madison, Computer Sciences Technical Report 1648, 2009.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. IISWC'09*.

[8] S. Che, J. Sheaffer, M. Boyer, L. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *Proc. IISWC'10*.

[9] K. D. Cooper, P. J. Schielke, and D. Subramanian, "Optimizing for Reduced Code Space using Genetic Algorithms," in *Proc. LCTES'99*.

[10] Z. Wang and M. F. O'Boyle, "Mapping Parallelism to Multi-cores: A Machine Learning Based Approach," in *Proc. PPoPP'09*.

[11] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," *SIGKDD Explorations*, 2009.

[12] H. S. Seung, M. Opper, and H. Sompolinsky, "Query by Committee," in *Proc. COLT'92*.

[13] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

[14] I. Dagan and S. P. Engelson, "Committee-Based Sampling For Training Probabilistic Classifiers," in *Proc. ICML'95*.

[15] D. S. Moore and G. P. McCabe, *Introduction to the Practice of Statistics*. W. H. Freeman, 2002.

[16] B. L. Welch, “The Generalization of “Student's” Problem when Several Different Population Variances are Involved,” *Biometrika*, 1947.

[17] C. Bastoul, “Code Generation in the Polyhedral Model Is Easier Than You Think,” in *Proc. PACT'04*.

[18] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos, “Iterative Optimization in the Polyhedral Model: Part II, Multidimensional Time,” in *Proc. PLDI'08*.

[19] M. Clement and M. Quinn, “Analytical Performance Prediction on Multicomputers,” in *Proc. SC'93*.

[20] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, “The worst-case execution-time problem – overview of methods and survey of tools,” *ACM TECS*, 2008.

[21] S. Hong and H. Kim, “An Analytical Model for a GPU Architecture with Memory-level and Thread–level Parallelism Awareness,” in *Proc. ISCA'09*.

[22] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, “Flextream: Adaptive Compilation of Streaming Applications for Heterogeneous Architectures,” in *Proc. PACT'09*.

[23] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, “Meta Optimization: Improving Compiler Heuristics with Machine Learning,” in *Proc. PLDI'03*.

[24] Z. Wang and M. F. O'Boyle, “Partitioning streaming parallelism for multi-cores: a machine learning based approach,” in *PACT '10*.

[25] D. Grewe, Z. Wang, and M. F. O'Boyle, “Opencl task partitioning in the presence of gpu contention,” in *LCPC '13*.

[26] D. Grewe, Z. Wang, and M. O'Boyle, “A workload-aware mapping approach for data-parallel programs,” in *HiPEAC '11*.

[27] M. Zuluaga, A. Krause, P. Milder, and M. Püschel, ““Smart” Design Space Sampling to Predict Pareto–Optimal Solutions,” in *Proc. LCTES'12*.

[28] D. Grewe, Z. Wang, and M. O'Boyle, “Portable mapping of data parallel programs to opencl for heterogeneous systems,” in *CGO '13*.

[29] M. K. Emani, Z. Wang, and M. F. P. O'Boyle, “Smart, adaptive mapping of parallelism in the presence of external workload,” in *CGO '13*.

[30] Z. Wang and M. F. P. O'boyle, “Using machine learning to partition streaming programs,” *ACM TACO*, 2013.

[31] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather, C. Williams, M. O'Boyle, P. Barnard, E. Ashton, E. Courtois, and F. Bodin, in *Proceedings of the GCC Developers' Summit*.

[32] C.-k. Luk, S. Hong, and H. Kim, “Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping,” in *Proc. MICRO'09*.

[33] M. Zuluaga, A. Krause, G. Sergent, and M. Püschel, “Active Learning for Multi–Objective Optimization,” in *Proc. ICML'13)*.

[34] P. Balaprakash, R. B. Gramacy, and S. M. Wild, “Active-Learning-Based Surrogate Models for Empirical Performance Tuning,” in *Proc. CLUSTER'13*.

[35] P. Balaprakash, K. Rupp, A. Mametjanov, R. B. Gramacy, P. D. Hovland, and S. M. Wild, “Empirical Performance Modeling of GPU Kernels Using Active Learning,” in *Proc. ParCo'13*.

[36] Y. Liu, E. Z. Zhang, and X. Shen, “A Cross-Input Adaptive Framework for GPU Program Optimizations,” in *Proc. IPDPS'09*.

[37] M. Samadi, A. Hormati, M. Mehrara, J. Lee, and S. Mahlke, “Adaptive Input-aware Compilation for Graphics Engines,” in *Proc. PLDI'12*.