W. Karl, J. Keller (Eds.): PARS 2017
Proc. 27th PARS-Workshop

Predicting Efficient Execution with Source Code Analysis in a Heterogeneous Environment

Markus Helwig[1] Thomas Becker[2]

Abstract: Finding a good schedule for the tasks of an application is a critical step for the efficient usage of heterogeneous systems. A good schedule can only be found with information about the tasks to be scheduled. In a dynamic system, this information is normally only available after each task is at least executed once, thereby creating an initial overhead until a good schedule can be created. Therefore, we introduce a method based on static code analysis and machine learning algorithms to predict the fastest processor of a given OpenCL task before runtime by classification which helps to reduce this initial overhead. We show how we used a static code analysis implementation based on Clang to generate training data on a set of 10 different heterogeneous processors including Intel, AMD and Nvidia GPUs, a Intel Xeon Phi and Intel CPUs. This training data was used to generate prediction models via several different machine learning algorithms including Random Forest and k-Nearest Neighbour and then evaluate the models by predicting the fastest processor out of two and more processors via classification.

Keywords: Heterogeneous Systems, Machine Learning, Static Code Analysis, Classification

1 Introduction

Due to technical limitations of transistors, satisfying the steadily increasing performance need by increasing the clock frequency is no longer a viable option. Therefore, hardware vendors started introducing processors with a growing number of cores. Additionally, with the rise of graphics processing units (GPU) for general purpose computations there is mostly at least one specialized accelerator included in a computer system. The current trend goes to even more specialized architectures including additional accelerators like FPGAs or the Intel Xeon Phi processors.

To free the application developer from having to use different programming models and languages, OpenCL [Khr08], which offers a uniform programming model for a wide range of processing units, was developed in 2008. What OpenCL does not offer, is the answer to the question on which processing unit an execution is the most efficient in terms of execution time. A solution to this problem are runtime systems like HALadapt [Kic14] which use scheduling algorithms to map tasks to processing units optimising execution time.

These scheduling algorithms need information about the tasks to be scheduled to find good solutions which in dynamic systems normally can only be collected by execution

[1] Karlsruhe Institute of Technology (KIT), CAPP, Haid-und-Neu-Str. 7, 76131 Karlsruhe Germany, markus@helwig.eu
[2] Karlsruhe Institute of Technology (KIT), CAPP, Haid-und-Neu-Str. 7, 76131 Karlsruhe Germany, thomas.becker@kit.edu

generating overhead before a good schedule can be found. Good scheduling decisions are important because the execution time of a task can vary greatly between heterogeneous processors and so a disadvantageous mapping can have a huge impact on the total application runtime.

Thus, the beforehand knowledge of which processor has the lowest execution time for a given OpenCL task can lead to a reduction in necessary profiling overhead and the creation of better schedules at the start of an execution cycle.

In this work, a methodology based on a source code analysis using Clang and LLVM and machine learning techniques to predict the fastest processor for a given OpenCL task by classification is developed. This means that no actual execution times are predicted but this work builds the basis for execution time prediction in future work. The source code analysis is restricted to code metrics which can be statically collected, thereby reducing the overhead at the start of an execution cycle. By using these at most static code metrics the fastest processor of an OpenCL task is determined by a classification via machine learning. In a following evaluation (Sect. 6) it is shown that the combination of code analysis and machine learning techniques is not only able to select the fastest of two processors but also the fastest of a set of various heterogeneous processors.

2 Related Work

One of the most well known runtime systems for heterogeneous systems and task scheduling is StarPU [ATN09] which also uses performance prediction models. The prediction models are build during runtime using a performance history database which is filled by measuring actual task executions and thereby creating an overhead. The goal of the work presented here is to find a way to reduce this start up overhead.

In [BFA14] a machine learning model is used to predict the expectable speed up for the port of CPU code to a GPU and the best device for a given OpenCL kernel out of a multi-core CPU und two GPUs. Baldini et al. collect dynamic code features by binary instrumentation using Ping and use these and the speed up of a 12-threaded OpenMP implementation as features for the prediction. Binary instrumentation usually alters the application execution and the additional instructions create overhead during runtime. As Baldini et al. state in their paper, the use of performance counters could be better suited for runtime contexts like scheduling. Additionally, this work evaluates bigger sets and with the Xeon Phi an additional accelerator.

In [WWO14] the goal is to support the scheduling of OpenCL kernels in a system consisting of a CPU and a GPU. This is achieved by predicting speed up categories for OpenCL source code to differentiate between an execution on a GPU or a CPU by classifying a kernel to low or high speed up on the GPU. Beside some static code metrics the approach also uses the input and output sizes and thread numbers.

The goal of [HIKS15] is to automatically select the fastest OpenCL device for the usage of Java's parallel stream API. This is done by implementing a classification directly in

the Java JIT compiler. This classification distinguishes between an execution on GPUs or CPUs. In the presented work, the predictors have to differentiate not only two processors but sets with up to ten processors making the prediction more complex.

Wu et al. [WGL+15] use machine learning to predict the performance of a GPU kernel on different target architectures belonging to the training set based on a previous execution of the kernel on a base hardware configuration and collected performance counter values. This is done by a classifier which maps the kernel to clusters representing different scaling behaviours. In contrast to the work presented in this paper, Wu et al. are only focused on the performance of kernels on GPUs and additionally need an execution of every kernel to predict its performance.

Amarís et al. [AdCD+16] predict the execution time of applications using vector operations on NVIDIA GPUs via different machine learning algorithms. The models use dynamic code metrics and architecture characteristics like the number of cores and the maximum GPU clock rate as features. As the work of Wu et al., Amarís et al. also only focus on GPUs and do not consider CPUs or other accelerators. Additionally, they only consider a very specific set of applications.

In the two following works an optimal task partitioning for the simultaneous execution of OpenCL on multiple processors is predicted. In both works, mainly static code metrics are used as features for the machine learning algorithms, complemented by some runtime features like the number of threads or transfer sizes between host and device memory. In [GO11] the prediction distinguishes between an execution on either the CPU or the GPU or a mixed execution. The prediction of the optimal distribution on the multiple processors is then predicted in a second step. Kofler et al. [KGCF13] use Support Vector Machines aswell as Artificial Neural Networks to find the optimal task partition in a single step.

3 Fundamentals

3.1 Source Code Analysis

Source code analysis is used in this work to extract metrics from the source code of OpenCL programs and can be fielded into static and dynamic techniques.

Dynamic methods try to extract characteristics of a given source code during runtime by instrumenting the code which can influence and prolong the actual execution.

Static code analysis on the other hand limits the observation to the source code itself. The analysis can happen before or after the compilation of the program. Also, a possible intermediate representation could be the objective of the analysis. But since there are no information about the actual execution of the program, the analysis can deliver, especially for loops and branches, only an abstract view on the execution. Another disadvantage is, that optimizations by compilers cannot, or in case of the analysis of an intermediate representation can only partially, be taken into account.

Because the goal is to reduce the necessary profiling overhead at the start of an execution cycle, in this work only a static code analysis is used complemented by some dynamic metrics which are known before the actual execution.

3.2 Machine Learning

Machine learning can be split into supervised and unsupervised algorithms. Supervised machine learning tries to infer a function from labelled training data. Algorithms generate this inferred function by analysing the training data. This function can then be used to map new examples.

The training data consists of pairs of feature vectors and a label which are also known as training examples. This label is predicted by the inferred function for future examples and can be a category or a real number. In the first case the problem is called classification and in the latter regression. The inferred function is called machine learning model.

Since we are trying to predict the fastest of a set of processors, we are using a category as a label. Therefore, we are going to use classification algorithms of supervised machine learning.

In this work the ***k*-Nearest Neighbour** [Run16], **Random Forests** [Ho95] and **Support Vector Machines** [Run16] are used.

***k*-Nearest Neighbour** tries to find the k nearest points of an example to be predicted in the space of the feature vectors by computing the distance to all points via a distance metric. The most frequently represented label among the k nearest points is then predicted, so explained in [Run16].

Decision Trees are machine learning models which make predictions by learning simple decision rules inferred from the training data. Starting at the root of a tree, in each step the training examples are split into two parts by a threshold of a single feature.

Random Forests are a combination of multiple Decision Trees and were first mentioned in [Ho95]. To make a prediction, each Decision Tree predicts a value. The value which gets predicted the most is the prediction result.

Support Vector Machines (SVMs) try to divide the training examples in the space of the feature vectors by a hyperplane. As stated in [Run16], the optimal hyperplane is determined by maximizing the distance of a to be selected distance metric to the feature vectors. regression kannst du weglassen, da die Laufzeitvorhersage ja nicht betrachtet wird

Cross Validation is used to find the optimal parameters for the algorithms as described in Sec. 5. Thereby, the training examples are split in several sets. A model is generated for each set, which is used as a validation set. The remaining sets are used as training examples to generate the model. To estimate the performance, each model predicts values for the examples of the validation set. The prediction performance for the whole set is then determined by taking the average. This allows us to not further divide the training set into

a training set and an evaluation set used to evaluate the parameters. Instead, we can use the whole training set and use **Cross Validation** to evaluate the different models and find the best parameters.

The machine learning parts of this work were implemented with the Python programming language making usage of the scikit-learn [PVea11] library.

4 Source Code Analysis

The source code analysis is implemented by using the Clang tooling from the LLVM compiler suite [LA04]. With Clang tooling the built up abstract syntax tree can be traversed. The different source code constructs can then be observed by visitor functions which get called if a specific construct is found. In this way, metrics like binary operations or memory accesses can be counted. The binary operations are counted for each data type separately. If a construct occurs within a loop or a branch the metrics are multiplied or divided by a factor which is a parameter of the analysis and can be set before the analysis starts.

Beside these counted occurrences of different kinds of operations, two complexity metrics are also extracted. The used complexity metrics are the **NPath Complexity** from [Nej88] and the **Cyclomatic Complexity** from [McC76]. The implementation calculating these complexity metrics is taken from [OCL12].

Since a static code analysis can only provide an abstract view, the analysis is improved by a branch prediction and a loop detection. The prediction is based upon variables whose values are known by e.g. visited defines, given kernel arguments or being the return value of functions returning a fixed value like work group functions. By calculating the visited operations which use known variables, estimations of the probability if a branch is taken or not and the number of loop iterations can be made. With these predictions, the multiplier for the metrics occurring within loops or branches is adjusted.

Because not every loop or branch condition depends on variables whose values are fixed for all work items, the prediction is enhanced by the value range propagation from [Pat95]. With the value range propagation, for each variable a value range and a probability is saved. Since there is no guarantee of being correct while calculating with these *probabilistic variables*, the range distribution of the values is always considered as *one*. With these *probabilistic variables*, predictions could be made for more loops and branches.

Following is an overview of the extracted metrics:

Number of binary operations		
• char • short • int	• long • half • float	• double • bool
Memory accesses		
• number of global read accesses	• read amount from global memory	• number of global write accesses
• written amount to global memory	• number of local read accesses	• read amount from local memory
• number of local write accesses	• written amount to local memory	
• proportion of global memory accesses to the number of total binary operations		
Loops and branches		
• number of loops	• number of branches	• average depth of nested loops
• average depth of nested branches	• max. depth of nested loops	• max. depth of nested branches
• number of nested loops	• number of nested branches	
Number of called OpenCL functions		
• atomic und asynchronous	• mathemtical	• other
• number of kernel arguments	• number of buffer kernel arguments	• number of array accesses
• synchronization points	• NPath Complexity	• Cyclomatic Complexity
• declarations of variables	• problem sizes	

5 Methodology

To train the machine learning models, training data has to be created. As training data the runtime of about 270 OpenCL kernels were measured and the kernels afterwards analysed. The OpenCL kernels were taken from the following software development kits and benchmarks:

- AMD APP SDK
- Intel OpenCL Samples
- Hetero-Mark [SGZ+16]
- Parboil Benchmark [SRS+12]
- PolyBench/GPU [GGXS+12, GGP12]
- Rodinia Benchmark Suite
- SHOC Benchmark Suite
- Hydro2de [BC01]

By using the OpenCL wrapper from [Kic14], the runtime of the OpenCL kernels can be measured. The OpenCL wrapper acts as an OpenCL device and passes the OpenCL function calls to a real OpenCL device. Thus, the wrapper has access to detailed information and parameters of the OpenCL kernel and besides measuring the runtimes, the values of committed kernel parameters, problem sizes and build parameters can also be logged which only creates overhead for the host part of the executed program but is not influencing the execution on the device itself.

To generate the training examples, the OpenCL kernels were executed with a variety of problem sizes on all of the ten different processors. With each problem size the kernel was run at least ten times.

After measuring the kernel runtimes, the source code of each OpenCL kernel was analysed by the code analysis developed in Sec. 4. The extracted code metrics were then combined with the problem size to a feature vector. As label, the device number of the OpenCL device which executed the related kernel the fastest was used. So, each feature vector contains the collected code metrics of the associated OpenCL kernel and is labeled with the device number of the OpenCL device which executed the kernel the fastest in the considered evaluation scenario. This means the labels can change over different evaluation scenarios and the classification has to predict the correct label for a given OpenCL kernel.

To calculate an average over the multiple executions of each kernel and also to minimise a bias towards a specific kernel which results in many slightly different code metrics, vectors with *similar* code metrics are combined. Two vectors are defined as being *similar*, if all features at most differ by *five* percent. Afterwards, the vectors are normed.

Out of the resulting vectors two sets are created. 75 percent are forming the training set, the other 25 percent the validation set. The validation set is exclusively used to evaluate the final machine learning models and is not used to train the models so the evaluation can be done with apriori unknown data.

The training set is then used to train the machine learning models. To determine the optimal parameters for the model generating algorithms, a grid search is used to train multiple models. Cross Validation (see sec. 3.2) is used to determine the performance because thereby it is not necessary to use a subset of the training set exclusively for evaluation. The best performing model is taken for further experiments and is then, as mentioned previously, validated with the validation set.

The impact of the different features are determined by the ANOVA F-Test [CL06] (1) which is defined as follows for a set of training vectors x_k with n_+ positive and n_- negative instances:

$$F(i) = \frac{\left(\bar{x}_i^{(+)} - \bar{x}_i\right)^2 + \left(\bar{x}_i^{(-)} - \bar{x}_i\right)^2}{\frac{1}{n_+-1}\sum_{k=1}^{n_+}\left(x_{k,i}^{(+)} - \bar{x}_i^{(+)}\right)^2 + \frac{1}{n_--1}\sum_{k=1}^{n_-}\left(x_{k,i}^{(-)} - \bar{x}_i^{(-)}\right)^2}, \tag{1}$$

where $\bar{x}_i, \bar{x}_i^{(+)}, \bar{x}_i^{(-)}$ are the average of the $i-$th feature of the whole, positive and negative data sets, respectively; $x_{k,i}^{(+)}$ is the $i-$th feature of the $k-$th positive instance, and $x_{k,i}^{(-)}$ is the i-th feature of the $k-$th negative instance. The calculated impact is then validated by training models with a reduced feature set. This is done by halving the number of used features until only one feature is used for the training. Every time the lower scored features are removed.

6 Experiments

The experiments were run on a variety of different processors. Among them are three central processing units (CPUs) from Intel, three graphics processing units (GPUs) from AMD and two from NVIDIA, the Intel Xeon Phi accelerator and an integrated GPU from Intel. On the side of the CPUs there were the Intel I7-6700k, the Intel i7-5820k and a system with two Intel Xeon E5-2670v2 processors. The GPUs are covered by the NVIDIA GTX 980 Ti and the NVIDIA GTX 650 Ti Boost. AMD is represented by the Radeon HD 7970, Radeon HD 7750 and the Radeon RX 470. Beside these models the Intel Xeon Phi 7120 and the Intel HD Graphics 530 are being used.

The OpenCL kernels were executed on an Arch Linux with the Linux Kernel in version 4.7. For the AMD devices the AMD Catalyst driver was used in version 15.9, for the NVIDIA

devices the GeForce 367.27 driver. The Intel devices were driven by three different drivers. The Intel CPUs were used with the Intel OpenCL Runtime 16.1.1 and the Intel Xeon Phi accelerator with Version 14.2. On the Intel GPU OpenCL was executed with the Intel OpenCL driver 3.0.

As evaluation metric for the different experiments *accuracy* (2) was used which is defined as the ratio of correct to all predictions.

$$accuracy = \frac{correct\ predictions}{total\ number\ of\ predictions} \tag{2}$$

The OpenCL source code was analysed with seven different settings of the code analysis to enable the evaluation of these optimization steps. At first, the branch prediction and loop detection were disabled and the problem size and values of kernel arguments not taken into account (step 1). Metrics in loops and branches were multiplied with *one*. In a next step, the problem sizes were taken into account during the analysis and metrics in branches were divided by *two* (step 2). After this, the branch prediction was enabled without probabilistic variables taken into account (step 3). In the following step the values of kernel arguments were considered (step 4) and afterwards the branch prediction was enabled with probabilistic variables (see Sec. 4) (step 5). In the two following steps, the multiplier in loops is set to *two* (step 6) and after this the loop detection was enabled (step 7). In a last step, the multiplier in loops was set to *16* (step 8). It should be noted that these optimizations steps are independent of each other and can be used in various different combinations and in any order.

6.1 Selecting the Fastest of two Processors

In the first experiment, we consider an OpenCL kernel which is given but unknown to the machine learning model. The processor which runs this kernel faster has to be predicted out of a pair of processors. Overall, the combination of the 10 processors resulted in a total of 45 pairs. For every pair a different machine learning model was trained and the labels of the feature vectors changed. In this paper, two pairs, the AMD RX 470 combined with the Intel i7-5820k and the combination of the Intel i7-5820k and Intel i7-6700k CPUs, are showcased representatively.

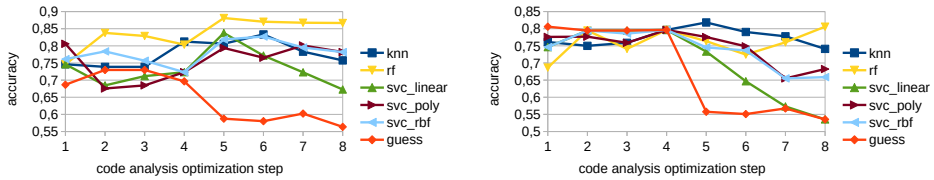
In Fig. 1 the prediction performance is shown for five different models generated by Random Forests (rf), *k*-Nearest Neighbour (knn) and Support Vector Machines (SMVs). Three different kernels were used for SMVs, namely a linear (svc_linear), a polynomial (svc_poly) and a radial basis function (svc_rbf). These five models were trained for the eight different settings of the code analysis, explained in Sec. 6.

A guessed (guess) prediction acted as a baseline which always predicts the processor which is the fastest in most cases as being the fastest for every OpenCL kernel, resulting in a much higher baseline for validation sets being dominated by one device than a totally random guess. This means that the processor that executes more OpenCL kernels faster is always

chosen as prediction and thereby, for this experiment the accuracy is always at least slightly over 50%.

As can be seen in Fig. 1, especially the Random Forest models could distinguish between the two processors, in particular after the activation of the branch prediction with probabilistic variables with an accuracy of 0.88. This is a great improvement over the guessed prediction with an accuracy of 0.58. *k*-Nearest Neighbour can also provide a good prediction performance and in some cases achieves better results than the Random Forests.

By activating the branch prediction with probabilistic variables, the code analysis can provide a more realistic view on the actual execution and differentiate better between multiple executions in which different processors are faster. This results in more and better distinguished training examples. The different amounts of training examples created by varying the code analysis also lead to different baseline values.



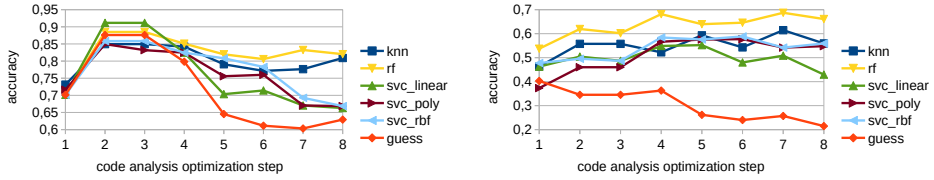
(a) Pair: AMD RX 470 and Intel i7-5820k

(b) Pair: Intel i7-5820k and Intel i7-6700k

Fig. 1: The accuracy for the trained models with different settings of the code analysis, selecting the optimal between two processors.

In almost every case the best predictions were made by models trained with all features. But acceptable predictions could also be made by models trained with a fourth. Among the most important features are the proportion between global memory accesses and binary operations, the number of array accesses and binary operations, the maximal depth of loops and reading memory access to the global memory.

6.2 Selecting the Fastest Processor of a Set



(a) Set 1, all processors.

(b) Set 2, all processors except AMD HD 7970, NVIDIA GTX 650 Ti and NVIDIA GTX 980 Ti.

Fig. 2: The accuracy for the trained models with different settings of the code analysis, selecting the optimal processor of a set.

In the second experiment, the task is to select the fastest processor out of a set of heterogeneous processors for a given but to the machine learning model unknown OpenCL kernel.

For this experiment, two sets were generated. The first set consists of every available processor. For the second set the three fastest and so most dominating processors, the AMD HD 7970, the NVIDIA GTX 650 Ti Boost and the NVIDIA GTX 980 Ti, are taken out to make the set more even and thereby the classification harder. With these sets the same experiments as in Sec. 6.1 are made. As in the first experiment, the baseline (guess) is always predicting the most dominating processor.

As the first set is strongly dominated by the NVIDIA GTX 980 Ti, the machine learning models could hardly deliver a better prediction as a guessed result. This was especially the case for the first settings of the code analysis, as can be seen in Fig. 2a. By activating the branch prediction with probabilistic variables not only the executions are represented better by the training examples but also the models deliver better results. The best predictions are made by the Random Forest model with the optimization step seven, described in Sec. 6. This model achieves an accuracy of 0.83 compared to 0.6 of the guessed result.

The second set is clearly more balanced, as one device is only the fastest for about 25 percent of the kernels. Although multiple devices are represented by a similar amount of training examples, the Random Forest models can deliver an accuracy of up to 0.69 in contrast to a guessed result with an accuracy of 0.25. This was accomplished by activating the branch prediction with probabilistic variables and the loop detection.

	Set 1	Set 2
#1	array accesses	global memory writes
#2	binary operations	global written size
#3	binary operations (int)	global memory reads
#4	global memory reads	array accesses
#5	global read size	binary operations
#6	number of loops	binary operations (int)
#7	number of nested loops	global read size
#8	binary operations (double)	Cyclomatic Complexity
#9	binary operations (float)	binary operations (float)

Tab. 1: The most important features for the experiments with sets of processors.

For these experiments, the best results are also achieved with the models being trained with all extracted features. Despite this, adequate results could be delivered with a reduced number of features. With a fourth of the features the models are still able to distinguish themselves from a guessed prediction. Especially for the second set, the achieved accuracy of 0.62 was only slightly lower.

Table 1 shows that very similar features to the first experiment have the most impact. Among the most important ones are numbers of binary operations, array accesses and numbers describing the global memory access. For the second set the *Cyclomatic Complexity* also has a relatively high impact.

7 Conclusion

Finding a good schedule is important for high performance in heterogeneous systems. To be able to generate a good schedule, information about the tasks to be scheduled is

necessary. This information is normally generated by monitoring the execution generating overhead during runtime.

Therefore in this work, we showed that static code analysis and machine learning algorithms can be used to create models which are able to predict the fastest processor out of a set by classification finding a way to reduce the time to settle to a good schedule. The static code analysis was implemented using Clang tooling and collects a set of different code metrics. The analysis was extended by complexity metrics, branch and loop prediction and the collection of kernel arguments improving the representation of the actual execution. The importance of the different metrics was evaluated by the ANOVA F-Test and validated by models trained with a subset of features.

We analysed about 270 OpenCL kernels generating training data which was used to evaluate models for the prediction of the fastest processor generated by several different machine learning algorithms like k-Nearest Neighbour, SVMs and Random Forest. The models were evaluated in two scenarios.

In the first scenario, two processors had to be compared. The experiments showed that the models could improve the baseline of always predicting the processor which was faster on most OpenCL kernels. Especially in scenarios where the kernels were evenly distributed between the processors like AMD RX 470 against Intel i7-5820k with a Random Forest model achieving an accuracy of 0.88 against 0.58 of the baseline.

In the second scenario, the fastest processor out of a set of at most 10 processors including Intel CPUs and a GPU, AMD and Nvidia GPUs and an Intel Xeon Phi had to be found. Here, the models could also achieve better results than the baseline which is again best seen by an even set including all processors but the three fastest. The baseline only achieved an accuracy of 0.25 while the best model generated by Random Forest delivered an accuracy of 0.69. In total, we showed that it is possible to reliably predict the fastest processor out of a given set and that models created by Random Forest and k-Nearest Neighbour achieve the best results.

In the future, we want to extend this work to not only predict the fastest processor but to be able to predict the actual execution time of an OpenCL kernel for a given processor. In order to do this reliably, we need to examine which applications and application characteristics are and are not well suited for the classification and thereby for an actual runtime prediction.

References

[AdCD+16] M. Amarís, R. Y. de Camargo, M. Dyab, A. Goldman, and D. Trystram. A comparison of GPU execution time prediction using machine learning and analytical modeling. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 326–333, Oct 2016.

[ATN09] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009)*, Delft, Netherlands, August 2009.

[BC01] C. Beffa and R. J Connell. Two-dimensional flood plain flow. I: Model description. *Journal of Hydrologic Engineering*, 2001.

[BFA14] I. Baldini, S. J. Fink, and E. R. Altman. Predicting GPU Performance from CPU Runs Using Machine Learning. In *SBAC-PAD*, pages 254–261. IEEE Computer Society, 2014.

[CL06] Yi-Wei Chen and Chih-Jen Lin. *Combining SVMs with Various Feature Selection Strategies*, pages 315–324. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[GGP12] Scott Grauer-Gray and Louis-Noel Pouchet. PolyBench/GPU - Implementation of PolyBench codes for GPU processing. `http://web.cse.ohio-state.edu/~pouchet/software/polybench/GPU/`, March 2012. Accessed: 2016-09-30.

[GGXS+12] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10, May 2012.

[GO11] D. Grewe and M. F. P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 286–305, Berlin, Heidelberg, 2011. Springer-Verlag.

[HIKS15] A. Hayashi, K. Ishizaki, G. Koblents, and V. Sarkar. Machine-Learning-based Performance Heuristics for Runtime CPU/GPU Selection. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, pages 27–36, New York, NY, USA, 2015. ACM.

[Ho95] Tin Kam Ho. Random Decision Forests. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 1) - Volume 1*, ICDAR '95, pages 278–, Washington, DC, USA, 1995. IEEE Computer Society.

[KGCF13] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer. An Automatic Input-sensitive Approach for Heterogeneous Task Partitioning. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, 2013.

[Khr08] Khronos. OpenCL. *https://www.khronos.org/news/press/2008/06*, 2008.

[Kic14] Mario Kicherer. *Reducing the Complexity of Heterogeneous Computing: A Unified Approach for Application Development and Runtime Optimization*. PhD thesis, Karlsruhe Institute of Technology, 2014.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[McC76] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[Nej88] Brian A. Nejmeh. NPATH: A Measure of Execution Path Complexity and Its Applications. *Commun. ACM*, 31(2):188–200, February 1988.

[OCL12] OCLint. OCLint. `http://oclint.org/`, 2012.

[Pat95] Jason R. C. Patterson. Accurate Static Branch Prediction by Value Range Propagation. *SIGPLAN Not.*, 30(6):67–78, June 1995.

[PVea11] F. Pedregosa, G. Varoquaux, and A. Gramfort et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 2011.

[Run16] T. A. Runkler. *Data Analytics - Models and Algorithms for Intelligent Data Analysis, Second Edition*. Springer Wiesbaden, 2016.

[SGZ+16] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. Mccardwell, A. Villegas, and D. Kaeli. Hetero-mark, a benchmark suite for CPU-GPU collaborative computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2016.

[SRS+12] J. A. Stratton, C. Rodrigues, I-Jui Sung, N. Obeid, Li-Wen Chang, N. Anssari, G. D. Liu, and Wen-mei W. Hwu. Parboil: A Revised Benchmark Suite for ScientiïňĄc and Commercial Throughput Computing. Technical Report IMPACT-12-01, University of Illinois at Urbana-Champaign, March 2012.

[WGL+15] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. GPGPU performance and power estimation using machine learning. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 564–576, Feb 2015.

[WWO14] Y. Wen, Z. Wang, and M. F. P. O'Boyle. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, Dec 2014.