

Automatic mapping for OpenCL-Programs on CPU/GPU Heterogeneous Platforms

Konrad Moren¹ and Diana Göhringer²

¹ Fraunhofer Institute of Optronics, System Technologies and Image Exploitation
IOSB, Ettlingen 76275, Germany

`konrad.moren@iosb.fraunhofer.de`

² TU Dresden, Adaptive Dynamic Systems, 01062 Dresden, Germany

`diana.goehringer@tu-dresden.de`

Abstract. Heterogeneous computing systems with multiple CPUs and GPUs are increasingly popular. Today, heterogeneous platforms are deployed in many setups, ranging from low-power mobile systems to high performance computing systems. Such platforms are usually programmed using OpenCL which allows to execute the same program on different types of device. Nevertheless, programming such platforms is a challenging job for most non-expert programmers. To enable an efficient application runtime on heterogeneous platforms, programmers require an efficient workload distribution to the available compute devices. The decision how the application should be mapped is non-trivial. In this paper, we present a new approach to build accurate predictive-models for OpenCL programs. We use a machine learning-based predictive model to estimate which device allows best application speed-up. With the LLVM compiler framework we develop a tool for dynamic code-feature extraction. We demonstrate the effectiveness of our novel approach by applying it to different prediction schemes. Using our dynamic feature extraction techniques, we are able to build accurate predictive models, with accuracies varying between 77% and 90%, depending on the prediction mechanism and the scenario. We evaluated our method on an extensive set of parallel applications. One of our findings is that dynamically extracted code features improve the accuracy of the predictive-models by 6.1% on average (maximum 9.5%) as compared to the state of the art.

Keywords: OpenCL, heterogeneous computing, workload scheduling, machine learning, compilers, code analysis

1 Introduction

One of the grand challenges in efficient multi-device programming is the workload distribution among the available devices in order to maximize application performance. Such systems are usually programmed using OpenCL that allows executing the same program on different types of device. Task distribution-mapping defines how the total workload (all OpenCL-program kernels) is distributed among the available computational resources. Typically application developers solve this

problem experimentally, where they profile the execution time of kernel function for each available device and then decide how to map the application. This approach error prone and furthermore, it is very time consuming to analyze the application scaling for various inputs and execution setups. The best mapping is likely to change with different: input/output sizes, execution-setups and target hardware configurations [1, 2]. To solve this problem, researchers focus on three major performance-modeling techniques on which mapping-heuristic can be based: simulations, analytical and statistical modeling. Models created with analytical and simulation techniques are most accurate and robust[3], but they are also difficult to design and maintain in a portable way. Developers often have to spend huge amount of time to create a tuned-model even for a single target architecture. Since modern hardware architectures are rapidly changing those methods are likely to be out of the date. The last group, statistical modeling techniques overcome those drawbacks, where the model is created by extracting program parameters, running programs and observing how the parameters variation affects their execution times. This process is independent of the target platform and easily adaptable. Recent research studies [4, 5, 6, 7, 8, 9] have already proved that predictive models are very useful in wide range of applications. However, one major concern for accurate and robust model design is the selection of program features.

Efficient and portable workload mapping requires a model of corresponding platform. Previous work on predictive modeling [10, 11, 12, 13] restricted their attention to models based on features extracted statically, avoiding dynamic application analysis. However, performance related information, like the number of memory transactions between the caches and main memory, is known only during the runtime.

In this paper, we present a novel method to dynamically extract code features from the OpenCL programs which we use to build our predictive models. With the created model, we predict which device allows the best relative application speed-up. Furthermore, we developed code transformation and analysis passes to extract the dynamic code features. We measure and quantify the importance of extracted code-features. Finally, we analyze and show that dynamic code features increase the model accuracy as compared to the state of the art methods. Our goal is to explore and present an efficient method for code feature extraction to improve the predictive model performance. In summary:

- We present a method to extract OpenCL code features that leads to more accurate predictive models.
- Our method is portable to any OpenCL environment with an arbitrary number of devices. The experimental results demonstrate the capabilities of our approach on three different heterogeneous multi-device platforms.
- We show the impact of our newly introduced dynamic features in the context of predictive modeling.

This paper is structured as follows. Section 2 gives an overview of the related work. Section 3 presents our approach. In Section 4 we describe the experiments.

In Section 5 we present results and discuss the limitations of our method. In the last section, we draw our conclusion and show directions for the future work.

2 Background and Existing Approaches

Several related studies have tackled the problem of feature extraction from OpenCL programs, followed by the predictive model building.

Grewe[10] et al. proposed a predictive model based on static OpenCL code features to estimate the optimal split kernel-size. Authors present that the estimated split-factor can be used to efficiently distribute the workload between the CPU and the GPU in a heterogeneous system.

Magni[11] et al. presented the use of predictive modeling to train and build a model based on Artificial Neural Network algorithms. They predict the correct coarsening factor to drive their own compiler tool-chain. Similarly to Grewe they target almost identical code features to build the model.

Kofler[12] et al. build the predictive-model based on Artificial Neural Networks that incorporates static program features as well as dynamic, input sensitive features. With the created model, they automatically optimize task partitioning for different problem sizes and different heterogeneous architectures.

Wen[13] et al. described the use of machine learning to predict the proper target device in context of a multi-application workload distribution system. They build the model based on the static OpenCL code features with few runtime features. They included environment related features, which provide only information about the computing-platform capabilities. This approach is most related to our work. They also study building of the predictive model to distribute the workloads in a context of the heterogeneous platform.

One observation is that all these methods extract code features statically during the JIT compilation phase. We believe, that our novel dynamic code analysis, can provide more meaningful and valuable code features. We justify our statement by profiling the Listing 1.1.

```
1 kernel
2 void floydWarshall( global uint * pathDist, global uint * path,
3                   const uint numNodes, const uint pass)
4 {
5     const int xValue = get_global_id(0);
6     const int yValue = get_global_id(1);
7     const int oldWeight = pathDist[yValue * numNodes + xValue];
8     const int tempWeight = (pathDist[yValue * numNodes + pass] +
9     pathDist[pass * numNodes + xValue]);
10    if (tempWeight < oldWeight){
11        pathDist[yValue * numNodes + xValue] = tempWeight;
12        path[yValue * numNodes + xValue] = pass;
13    }
```

Listing 1.1. AMD-SDK FloydWarshall kernel

The results are shown in Fig.1. These experiments demonstrate the execution times of the Listing 1.1 executed with varying input values (*numNodes*, *pass*) and execution-configurations on our experimental platforms. We can observe that even for a single kernel function, the optimal mapping considerably depends

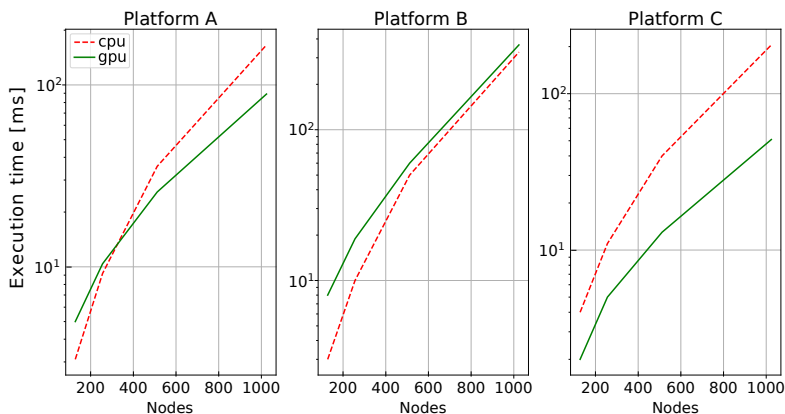


Fig. 1. Profiling results for an AMD-SDK FloydWarshall kernel function on test platforms. The target architectures are detailed in the Section 4.1. The Y-Axis presents the execution time in milliseconds, the X-Axis shows the varying number of nodes.

on the input/output sizes and the capabilities of the platform. In Listing 1.1 the arguments *numNodes* and *pass* control effectively the number of requested cache lines. According to our observations, many of the OpenCL programs rely on kernel input arguments, known only at the enqueueing time. In general, input values of OpenCL-function arguments are unknown at the compilation time. Many performance related information, like the memory access pattern, number of executed statements, could possibly be dependent on these parameters. This is a crucial shortcoming in previous approaches. The code-statements dependent on values known during the program execution are undefined and could not provide quantitative information. Since current state of the art methods analyze and extract code features only statically, new methods are needed. In the next section, we present our framework that addresses this problem.

3 Proposed Approach

This section describes the design and the implementation of our dynamic feature extraction method. We present all the parts of our extraction approach: transformation and feature building. We describe which code parameters we extract and how we build the code features from them. Finally, we present our methodology to train and build the statistical performance model based on the extracted features.

3.1 Architecture Overview

Fig.2 shows the architecture of our approach. We modify and extend the default OpenCL-driver to integrate our method. First, we use the binary LLVM-IR rep-

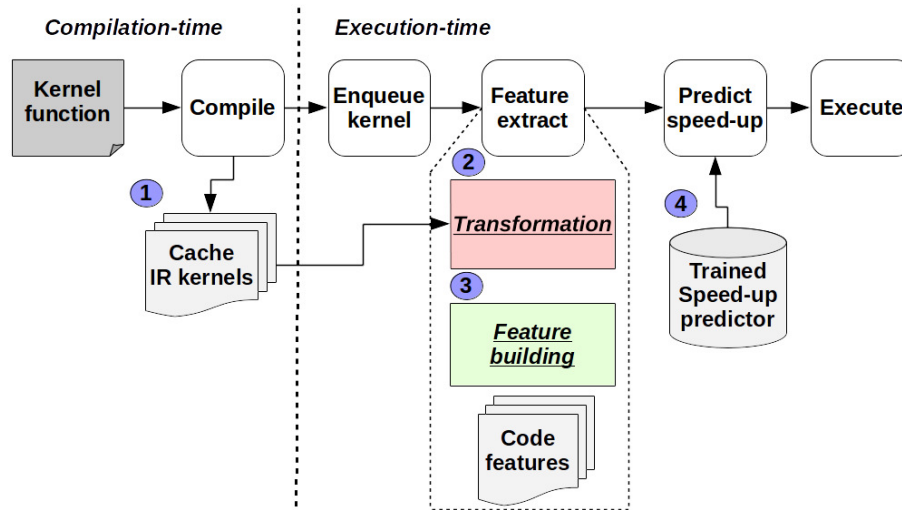


Fig. 2. Architecture of the proposed approach.

representation of the kernel function and cache it in the driver memory ❶. We reuse IR functions during enqueueing to the compute-device. During the enqueueing phase, cached IR functions with known parameters are used as inputs to the transformation engine. At the time of enqueueing, the values of input arguments, the kernel code and the NDRange sizes are known and remain constant. A semantically correct OpenCL program always needs this information to properly execute [14]. Based on this observation, our transform module ❷ rewrites the input OpenCL-C kernel code to a simplified version. This kernel-IR version is analyzed to build the code features ❸. Finally we deploy our trained predictive model and embed it as a last stage in our modified OpenCL driver ❹. Following sections describe steps ❶-❹ in more details.

3.2 Dynamic code feature analysis and extraction

The modified driver extends the default OpenCL driver by three additional modules. First, we extend and modify the *clBuildProgram* function in OpenCL API. Our implementation adds a caching system ❶ to reduce the overhead of invoking transformation and feature-building modules. We store internal LLVM-IR representations in the driver memory to efficiently reuse it in the transformation module ❷. Building the LLVM-IR module is done only once, usually at the application beginning. The transformation module ❷ is implemented within the *clEnqueueNdRangeKernel* OpenCL API function. This module rewrites the input OpenCL-C kernel code to a simplified version. The Fig.3 shows the transformation architecture. The module includes two cache objects, which store original and pre-transformed IR kernel functions. We apply transformations in

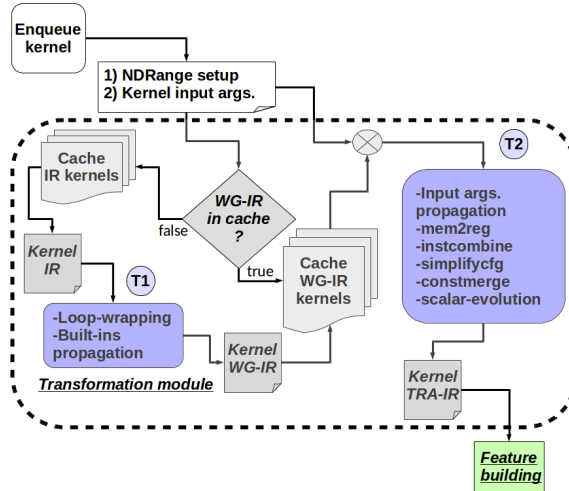


Fig. 3. Detailed view on our feature extraction module.

two phases $T1$ and $T2$. First phase $T1$, we load for a specific kernel name the IR-code created during ❶ and then wrap the code region with work-item loops. The wrapping technique is a known method described by Lee [15] and already applied in other studies [16, 17]. The work-group IR-function generation is performed at kernel enqueue time, when the group size is known. The known work-group size makes it possible to set constant values to the work-item loops. In a second phase $T2$, we load the transformed work-group IR and propagate constant input values. After this step, the IR includes all specific values not only the symbolic expressions. The remaining passes of $T2$ further simplifies the code. The Listing 1.2 presents the intermediate code after the transformation $T1$ and input argument values propagation. Due to the space limitation, we do not present the original LLVM-IR code but a readable-intermediate representation.

```

1 kernel
2 void floydWarshall( global uint * pathDist, global uint * path)
3 {
4     for(int yValue=0; yValue<1024; yValue++){
5         for(int xValue=0; xValue<1024; xValue++){
6             const int oldWeight = pathDist[yValue * 1024 + xValue];
7             const int tempWeight = (pathDist[yValue * 1024 + 16] +
8                 pathDist[16 * 1024 + xValue]);
9             if (tempWeight < oldWeight){
10                pathDist[yValue * 1024 + xValue] = tempWeight;
11                path[yValue * 1024 + xValue] = 16;
12            }
13        }
14    }
15 }

```

Listing 1.2. The readable-intermediate representation of Listing 1.1 after input and built-in constants propagation. The execution parameters are: numNodes=1024, pass=16, work-group sizes=(1,1), global sizes=(1024,1024)

We can observe that the constant propagation pass, enables to determine how the memory accesses are distributed. Now the system can extract not only how many load and stores are requested, but also how are they distributed. With pure static code analysis, this information is not available. Additionally, compared to the pure static methods we analyze more accurately the instructions. Our method simplifies the control flow graph and analyzes only the executable instructions. In contrast, the static code analysis scans all basic blocks also these that are not used. Furthermore, we extract for each load and store instructions the Scalar Evolution (SCEV) expressions. The extracted SCEV expressions represent the evolution of loop variables in a closed form [18, 19]. A SCEV consist of a starting value, an operator and an increment value. They have the format $\{ \langle base \rangle, +, \langle step \rangle \}$. The base of an SCEV defines its value at loop iteration zero and the step of an SCEV defines the values added on every subsequent loop iteration [20]. For example, the SCEV expression for the load instruction in Listing 1.2 on line 6 has the form $\{ \{ \%pathDist, +, 4096 \}, +, 4 \}$. We can see that this compact representation describes the memory access of the kernel input argument `%pathDist`. With this information, we analyze the SCEVs for existing loads and stores to infer the memory access. We group the extracted memory accesses in four groups. First invariant accesses with the stride zero. Stride zero accesses(i.e., invariant) means that the memory access index is the same for all loop iterations in a work-group. The second group, consecutive accesses with stride one. Stride one means that the memory access index increases by one for consecutive loop iterations. The third group, non-consecutive accesses with the stride N , where N means that the memory access index is neither invariant nor stride one. Finally, the last group, the unknown accesses with the stride X . In general, SCEV expression can have an unknown value due to a dependence on the results calculated during the code execution. Table 1 presents all extracted information about the kernel function.

	Features	Description
F1	$(arithmetic_inst)/(all_inst)$	computational intensity ratio
F2	$(memory_inst)/(all_inst)$	memory intensity ratio
F3	$(control_inst)/(all_inst)$	control intensity ratio
F4	<i>datasize</i>	global memory allocated
F5	<i>globalWorkSize</i>	number of global threads
F6	<i>localWorkSize</i>	number of local threads
F7	<i>workGroups</i>	number of work-groups
F8	<i>Stride0</i>	invariant memory accesses
F9	<i>Stride1</i>	consecutive memory accesses
F10	<i>StrideN</i>	scatter/gather memory accesses
F11	<i>StrideX</i>	unknown memory accesses

Table 1. Features extracted with our dynamic analysis method. These features are used to build the predictive model.

The selected features are not specific for any micro-architecture or device type. We extract the existing OpenCL-C arithmetic, control and memory instructions. Additionally in contrast to other approaches, we extract the memory access pattern. The selection of the features is a design specific decision. We analyze in more detail the importance of selected features in Section 4.2. In the next section, we use our extracted features to create the training data and describe how we train our predictive model.

3.3 Building the prediction model

Building machine-learning based models involves the collection of data that is used in the model training and evaluation. To retrieve the data we execute, extract features and measure the execution time for various test applications. We use different applications implemented in: the NVIDIA OpenCL SDK [21], the AMD APP SDK [14], and the Polybench OpenCL v2.5 [22]. We execute the applications with different input data sizes. The purpose of this is twofold. First, the variable sizes of input data let us collect more training data and second, the data is more diverse due to the implicit change in work-group sizes. Many of these applications adapt the number of work-groups with the change of input/output data sizes. By varying the input variables of applications, we create the data set with 5887 samples. The list of application is shown in Table 2.


Suite	Application	Input sizes	Application	Input sizes
AMD SDK	Binary Search	80K-1M	Bitonic Sort	8K-64K
	Binomial Option	1K-64K	Black Scholes	34M
	DCT	130K-20M	Fast Walsh Transform	2K-32K
	Floyd Warshall	1K-64K	LU Decomposition	8M
	Monte Carlo Asian	4M-8M	Matrix Multiplication	130K-52M
	Matrix Transpose	130K-50M	Quasi Random Sequence	4K
	Reduction	8K	RadixSort	8K-64K
	Simple Convolution	130K-1M	Scan Large Arrays	4K-64K
	Nvidia SDK	DXT Compression	2M-6M	Median Filter
Dot Product		9K-294K	FDTD3d	8M-260M
HMM		2M-4M	Tridiagonal	320K-20M
Polybench	Atax	66K-2M	Bicg	66K-2M
	Gramschmidt	15K-1M	Gesummv	130K-5M
	Correlation	130K-5M	Covariance	130K-52M
	Syrk	190K-5M	Syr2k	190K-5M

Table 2. The applications used to train and evaluate our predictive model.

In our approach, we execute presented OpenCL programs on the CPU and the GPU to measure the speedup of the GPU execution for each individual kernel over the CPU. Furthermore, to consider various costs of data transfers on architectures with discrete and integrated GPUs, we measure the transfer times between the CPU and GPU. We define it as DT . To model the real cost of the execution on the GPUs, we add the DT to the GPU execution time. Finally, in a last step we combine the CPU/GPU execution times and label the kernel-code

to one of five speed-up classes. The Equation 1 defines the speed-up categories for our predictive model.

$$Speedup_class = \begin{cases} Class1 & \frac{CPU}{GPU+DT} \leq 1x(no\ speedup) \\ Class2 & 1x < \frac{CPU}{GPU+DT} \leq 3x \\ Class3 & 3x < \frac{CPU}{GPU+DT} \leq 5x \\ Class4 & 5x < \frac{CPU}{GPU+DT} \leq 7x \\ Class5 & \frac{CPU}{GPU+DT} \geq 7x \end{cases} \quad (1)$$

In our experiments, we use the Random Forest (RF) classifier. The reason for this is twofold. First, the RF classifier enables to build the relative feature importance ranking. In Section 5 we use this metric to explore the relative feature importance on the classification accuracy. The second one is that, the classifiers based on decision trees are usually fast. We also investigated other machine learning algorithms but due to the space limitations, we will not show a detailed comparison of these classifiers. Finally, once the model is trained we use the trained model during the runtime  to determine the kernel scheduling.

4 Experimental Evaluation

4.1 Hardware and Software Platforms

We evaluate on three CPU+GPU platforms. The details are shown in Table 3. All platforms have Intel CPUs, two platforms include discrete GPUs. The third platform is an Intel SoC (System on Chip) with integrated CPU/GPU. We use LLVM 3.8 with Ubuntu-Linux 16.04 LTS to drive our feature extraction tool. The host-side compiler is GCC 5.4.0 with -O3 option. On the device-side Intel OpenCL SDK 2.0, NVIDIA Cuda SDK 8.0 and AMD OpenCL SDK 2.0 provide compilers.

4.2 Evaluation of the model

We train and evaluate two speed-up models with different features to compare our approach with the state of the art. The first model, is based on our dynamic feature extraction method. Table 1 shows the features applied to build the model. To train and build the second model, we extract statically only the code features $F1-F7$ from the kernel function (i.e. during the JIT-compilation). The memory access features $F8-F11$ known only during the runtime are not included. For both models, we apply the following train and evaluation method. We split 10 times our dataset into train and test sets. Each time we randomly select 33% of dataset samples for the evaluation process. The remaining 67% are used to train the model. Figure 4 presents the confusion matrix for the evaluation scenario.

We observe that the prediction accuracy for the model created with dynamic features is higher than for the model based on static features. On the Platform

Platform A	CPU	GPU
	I7-4930K	Radeon R9-290
Architecture	Ivy Bridge	Hawaii
Core Count	6 (12 w/ HT)	2560
Core Clock	3.9 GHz	0.9 GHz
Memory bandwidth	59.7 GB/s	320 GB/s
Platform B	CPU	GPU
	I7-6600U	HD-520
Architecture	Skylake	Skylake
Core Count	2 (4 w/ HT)	192
Core Clock	3.4 GHz	1.0 GHz
Memory bandwidth	34.1 GB/s	25.6 GB/s
Platform C	CPU	GPU
	Xeon E5-2667	Geforce GTX 780 Ti
Architecture	Sandy Bridge	Kepler
Core Count	6 (12 w/ HT)	2880
Core Clock	3.5 GHz	0.9 GHz
Memory bandwidth	51.2 GB/s	288.4 GB/s

Table 3. Hardware Platforms

A the model based on dynamic features have a 90.1% mean accuracy. The accuracy values is an average over testing scenarios. We calculate the accuracy as the ratio between sum of values on the diagonal in Figure 4 to all values. We observe similar results for two other Platforms B and C. The mean accuracies for the remaining platforms are 77% and 84% for Platforms B and C respectively. Overall, we can report increase of the prediction accuracy with dynamically extracted features by 9,5%, 4,9% and 4,1% for the tested Platforms. We observe also that, the model based on dynamic features leads to lower slowdowns. We can observe from Figure 4 that the model with static features predicts less accurate, the error rate is 19,4%, for the dynamic model only 9,9%. More importantly, we can see that the distribution of errors is different. Overall, we can observe that the number of miss-predictions, values below and above the diagonal, is higher for the model created with static features. In the worst case, the model based on statically extracted features predicts only 36 times correctly the $7x$ speed-up on the GPU. This point corresponds to the lowest row in the confusion matrix presented in Figure 4.

5 Discussion

We find out in our experiments that the predictive models designed with the dynamic code features are more accurate and lead to lower performance degradation in context of workload distribution. To further explore the impact of dynamic features on the classification, we analyze the relative feature importance. The selected RF classifier enables to build the relative feature importance ranking. The relative feature importance metric is based on two statistical methods Gini-impurity and the Information gain. More details about the RF classifier and the feature importance metric are included in the [23]. Figure 5 presents

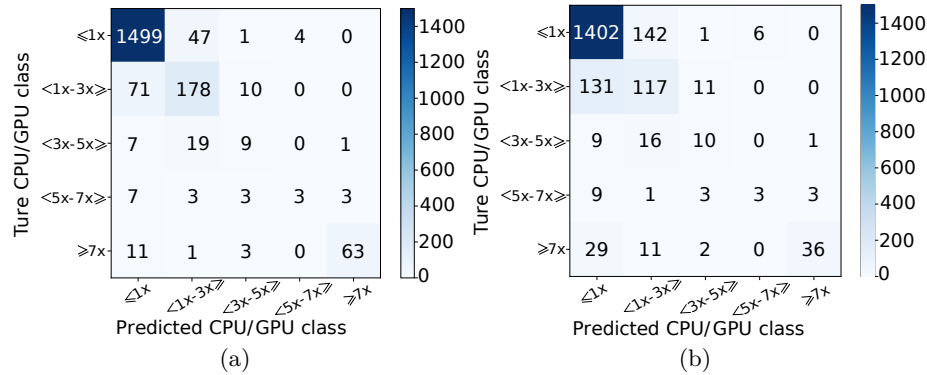


Fig. 4. The confusion matrix for platform A, (a) results for the model with dynamic features (b) results without dynamic features.

the relative feature importance for the both models presented in the previous section.

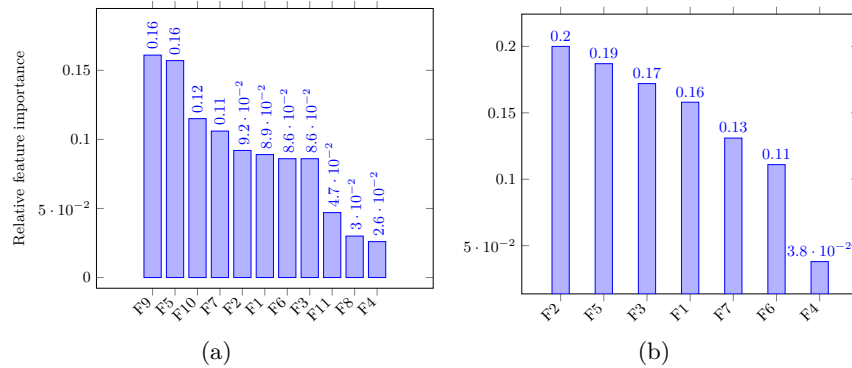


Fig. 5. Relative feature importance for the classifier (a) trained with dynamic features and (b) with statically extracted features. The values on X-Axis are features presented in Table 1, the Y-Axis represents the ranking of relative feature importance.

We can observe for the model created with the dynamic code features that the most informative features (i.e. mostly reducing the model variance), are consecutive memory access $F9$ and the $F5$ number of global work-items. For the second model created with statically extracted features, most informative are number of loads and stores the $F2$ and again the $F5$ count of global work-items. The high position in the ranking for loads and stores confirms the importance of memory accesses extracted with our dynamic approach. One intuitive and

reasonable explanation for the importance of dynamic code features (memory accesses) would be that many of the analyzed workloads are memory-bound.

5.1 Limitations

Our dynamic approach described in previous sections increases a classification accuracy. However, the proposed and described method in this paper has also several limitations. Our memory access analysis is limited to a sub-set of all possible code variants. The Scalar Evolution pass computes only the symbolic expressions for combinations of constants, loop variables and static variables. It supports only a common integer arithmetic like addition, subtraction, multiplication or unsigned division [20]. Other possible code variants and resulting statements lead to unknown values. Another aspect is the feature extraction time. Compared to the pure static methods our dynamic method generates an overhead during the runtime. We can observe the variable overhead between 0.3 and 4 ms, dependent on the platform capabilities and the code complexity.

6 Conclusion and Outlook

Deploying data parallel applications using the right hardware is essential for improving application performance on heterogeneous platforms. A wrong device selection and as a result not efficient workload distribution may lead to a significant performance loss. In this paper, we propose a novel systematic approach to build the predictive model that estimates the compute device with an optimal application speed-up. Our approach uses dynamic features available only during the runtime. This improves the prediction accuracy independently of the applications and hardware setups. Therefore, we believe that our work provides an effective and adaptive approach for users who are looking for high performance and efficiency on heterogeneous platforms. The performed experiments and results encourage us to extend and improve our methodology in the future. We will extract and experiment with other code features and classifiers. Additionally, we will improve our feature extraction method to further increase the model accuracy and reduce the overall runtime.

References

- [1] Calotoiu, A., Hoefler, T., Poke, M., Wolf, F.: Using automated performance modeling to find scalability bugs in complex codes. In Gropp, W., Matsuoka, S., eds.: International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013, New York, NY, USA, ACM (2013) 45:1–45:12
- [2] Hoefler, T., Gropp, W., Kramer, W., Snir, M.: Performance modeling for systematic performance tuning. In: State of the Practice Reports. SC '11, New York, NY, USA, ACM (2011) 6:1–6:12

- [3] Lopez-Novoa, U., Mendiburu, A., Miguel-Alonso, J.: A survey of performance modeling and simulation techniques for accelerator-based computing. *IEEE Trans. Parallel Distrib. Syst.* **26**(1) (2015) 272–281
- [4] Bailey, D.H., Snively, A. In: *Performance Modeling: Understanding the Past and Predicting the Future*. Springer Berlin Heidelberg, Berlin, Heidelberg (2005) 185–195
- [5] Nagasaka, H., Maruyama, N., Nukada, A., Endo, T., Matsuoka, S.: Statistical power modeling of GPU kernels using performance counters. In: *Green Computing Conference*, IEEE Computer Society (2010) 115–122
- [6] Kerr, A., Diamos, G.F., Yalamanchili, S.: Modeling GPU-CPU workloads and systems. In Kaeli, D.R., Leeser, M., eds.: *Proceedings of 3rd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2010*, Pittsburgh, Pennsylvania, USA, March 14, 2010. Volume 425 of *ACM International Conference Proceeding Series.*, ACM (2010) 31–42
- [7] Dao, T.T., Kim, J., Seo, S., Egger, B., Lee, J.: A performance model for gpus with caches. *IEEE Trans. Parallel Distrib. Syst.* **26**(7) (2015) 1800–1813
- [8] Baldini, I., Fink, S.J., Altman, E.R.: Predicting GPU performance from CPU runs using machine learning. In: *SBAC-PAD*, Washington, DC, USA, IEEE Computer Society (2014) 254–261
- [9] Tripathy, B., Dash, S., Padhy, S.K.: Multiprocessor scheduling and neural network training methods using shuffled frog-leaping algorithm. *Computers & Industrial Engineering* **80** (2015) 154–158
- [10] Grewe, D., O’Boyle, M.F.P.: A static task partitioning approach for heterogeneous systems using openCL. In Knoop, J., ed.: *Compiler Construction – (20th CC’11 (Part of 14th ETAPS’11))*. Volume 6601 of *Lecture Notes in Computer Science (LNCS)*. Springer-Verlag (NY), Saarbrücken, Germany (March-April 2011) 286–305
- [11] Magni, A., Dubach, C., O’Boyle, M.F.P.: Automatic optimization of thread-coarsening for graphics processors. In Amaral, J.N., Torrellas, J., eds.: *PACT*, ACM (2014) 455–466
- [12] Kofler, K., Grasso, I., Cosenza, B., Fahringer, T.: An automatic input-sensitive approach for heterogeneous task partitioning. In Malony, A.D., Nemirovsky, M., Midkiff, S.P., eds.: *ICS*, ACM (2013) 149–160
- [13] Wen, Y., Wang, Z., O’Boyle, M.F.P.: Smart multi-task scheduling for opencl programs on CPU/GPU heterogeneous platforms. In: *21st International Conference on High Performance Computing, HiPC 2014*, Goa, India, December 17-20, 2014. (2014) 1–10
- [14] AMD: *AMD APP SDK v2.9* (2014)
- [15] Lee, J., Kim, J., Seo, S., Kim, S., Park, J., Kim, H., Dao, T.T., Cho, Y., Seo, S.J., Lee, S.H., Cho, S.M., Song, H.J., Suh, S., Choi, J.: An opencl framework for heterogeneous multicores with local memory. In Salapura, V., Gschwind, M., Knoop, J., eds.: *19th International Conference on Parallel Architecture and Compilation Techniques (PACT 2010)*, Vienna, Austria, September 11-15, 2010, ACM (2010) 193–204
- [16] Kim, H.S., Hajj, I.E., Stratton, J.A., Lumetta, S.S., mei W. Hwu, W.: Locality-centric thread scheduling for bulk-synchronous programming models on CPU architectures. In Olukotun, K., Smith, A., Hundt, R., Mars, J., eds.: *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015*, San Francisco, CA, USA, February 07 - 11, 2015, IEEE Computer Society (2015) 257–268

- [17] Jo, G., Jeon, W.J., Jung, W., Taft, G., Lee, J.: Opencl framework for arm processors with neon support. In: Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing. WPMVP '14, New York, NY, USA, ACM (2014) 33–40
- [18] Zima, E.V.: On computational properties of chains of recurrences. In: Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation. ISSAC '01, New York, NY, USA, ACM (2001) 345–
- [19] Engelen, R.A.V.: Efficient symbolic analysis for optimizing compilers. In: In Proceedings of the International Conference on Compiler Construction (ETAPS CC'01. (2001) 118–132
- [20] Grosser, T., Größlinger, A., Lengauer, C.: Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* **22**(4) (2012)
- [21] Nvidia: Nvidia opencl sdk code samples (2014)
- [22] Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Auto-tuning a high-level language targeted to GPU codes. In: Innovative Parallel Computing (InPar), 2012. (May 2012) 1–10
- [23] Breiman, L.: Random forests. *Machine Learning* **45**(1) (2001) 5–32