Divergence Aware Automated Partitioning of OpenCL Workloads

Anirban Ghose,
Indian Institute of Technology, Kharagpur
anirban.ghose@cse.iitkgp.ernet.in

Soumyajit Dey,
Indian Institute of Technology, Kharagpur
soumya@cse.iitkgp.ernet.in

Pabitra Mitra
Indian Institute of Technology, Kharagpur
pabitra@cse.iitkgp.ernet.in

Mainak Chaudhuri
Indian Institute of Technology, Kanpur
mainak.chaudhuri@cse.iitk.ac.in

ABSTRACT

Heterogeneous partitioning is a key step for efficient mapping and scheduling of data parallel applications on multi-core computing platforms involving both CPUs and GPUs. Over the last few years, several automated partitioning methodologies, both static as well as dynamic, have been proposed for this purpose. The present work provides an in-depth analysis of control flow divergence and its impact on the quality of such program partitions. We characterize the amount of divergence in a program as an important performance feature and train suitable Machine Learning (ML) based classifiers which statically decide the partitioning of an OpenCL workload for a heterogeneous platform involving a single CPU and a single GPU. Our approach reports improved partitioning results with respect to timing performance when compared with existing approaches for ML based static partitioning of data parallel workloads.

CCS Concepts

•Computing methodologies → Supervised learning; •Computer systems organization → Parallel architectures; *Single instruction, multiple data;* **•Software and its engineering → Parallel programming languages;**

Keywords

Control Flow Divergence, OpenCL, Feature Extraction

1. INTRODUCTION

Heterogeneous computing platforms strive to deliver high performance and energy efficient solutions for compute intensive applications. With the rise of GPGPUs(General Purpose Computation on Graphic Processing Units), heterogeneous platforms comprising GPU and CPU cores have become a necessity in diverse application domains ranging from molecular physics to biomedical sciences.

ISEC '16, February 18-20, 2016, Goa, India

DOI: http://dx.doi.org/10.1145/2856636.2856639

For heterogeneous computing environments, the problem of optimally mapping a given computational workload to the underlying processing elements leading to efficient resource utilization is non trivial in general [1]. The optimality under question is with respect to execution time of the workload. We provide a static partitioning method for data-parallel OpenCL workloads targeted to heterogeneous platforms. Our method can be best described as a source-to-source optimization framework built around the OpenCL runtime system. OpenCL is a standardized device independent parallel programming framework for heterogeneous platforms comprising computing hardware with widely varying computational characteristics e.g., general purpose (CPU), data parallel (GPU), task parallel (CELL/B.E.) [2] etc.

Several static and dynamic methodologies exist that address the problem of workload partitioning over platforms comprising CPUs and GPUs. Dynamic methods perform work-load balancing of computational resources at runtime [3, 4, 5]. From an application perspective, dynamic methods are suitable for situations like online scheduling where run time decisions can actually be effected to alter the platform mapping of workloads. Static methods, on the other hand are actually compilation strategies which are designed to work without the knowledge of dynamic execution scenarios. Static frameworks are based on prior off-line profiling of the target application [1, 6] or using mechanisms like machine learning (ML) to predict near optimal program partitions based on static program structure [7, 8, 9, 10, 11]. Our approach is along the lines of machine learning based methodologies for static partitioning of OpenCL workloads on heterogeneous systems comprising a single CPU and a single GPU. We focus on the use of control-flow divergence as an important deciding factor for program partitions. We motivate the impact of divergence on program execution times and the resulting optimal partitions as follows.

In a data-parallel program having multiple computation threads executing in parallel, it may often be the case that the control path followed by two threads actually differ post some program point. The threads follow different execution paths if some branch condition is a function of the thread-id and the condition evaluates to true for one thread and false for the other. In the OpenCL programming model, such groups of *work-items* (i.e. OpenCL threads) concurrently execute the kernel (device specific code) on a device which has support for vector processing. When a group of work-items encounter a conditional statement, work-items taking different paths execute sequentially. Such instances of control flow divergence may cause severe performance degradation as observed for the code

snippet below.

```
__kernel void divergent(__global float*A,int N){
  int tid= get_global_id (0);
  float sum=0.0;
  int U=(tid+1)*N;
  if(tid<N)
  for(int i=tid;i<U;i++)
  //  if(tid%2==0)
      sum+=pow(sin(A[i]),i);
}
```

The above code snippet is a slightly modified version of an artificial program used in [12]. Work-item `tid` will execute the loop a variable number of times. The variable `sum` is updated with the data value at position i which is calculated as the i^{th} power of the sine of the original data value in `A`. Let us now introduce a divergent branch, say `if(tid%2==0)` inside the `for` loop. The presence of this branch condition has a huge impact on the execution time. Since, the branch condition is dependent on the thread-id, work-items with even thread-ids will execute the mathematical function, while the ones with odd ids will be stalled. The effect of stalling is compounded as this check will be executed per iteration.

The generic device independent programming framework of OpenCL provides us with the flexibility of engaging the CPU and GPU devices in parallel by partitioning the set of global work-items. The effect of control flow divergence on the execution performance of such partitioned OpenCL programs is highlighted in Figure 1 for the code segment we discussed earlier. A partition option represents the split ratio of the global work-item set across the CPU and GPU devices in parallel. The X-axis depicts the split ratio with the value (100:0) signifying full CPU and the value (0:100) signifying full GPU execution.

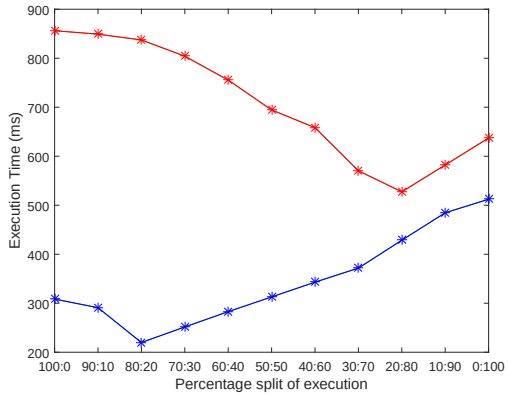

Figure 1: Execution profile for different partition options.

The blue line (below) depicts the execution times for various *partition options* of the program version without `if(tid%2==0)` inside the `for` loop (i.e. the non-divergent program). The red line (above) depicts the execution time profile for the divergent program. We can observe that there is a considerable increase in execution time for the divergent program when compared with the non-divergent version. This is expected because of the sequential execution forced by control flow divergence. The programs are similar in all other aspects (like arithmetic operations, barriers, memory access etc) except for the conditional branch. What is interesting is that the execution time for the divergent program does not vary in a similar pattern when compared with the non-divergent version across different partition options.

The non-divergent program will execute with minimum time for the partition option (80:20) i.e. 80% of the computation is mapped to the CPU while 20% of the computation is mapped to the GPU. The divergent program executes efficiently for the partition (20:80). This synthetic example perfectly illustrates the fact that the divergent nature of control flow behavior plays an important role while deciding the optimal partitioning of an OpenCL workload. The exact reason behind this can be attributed to the difference in handling of divergence by CPU and GPU architectures as well as the complex inter-play of program features which actually decide what is an optimal partition for a given data-parallel program.

The objective of this communication is to establish control-flow divergence as a deciding factor in static partitioning of data-parallel workloads and providing improved source-level program partition optimizers. In recent times, there has been a lot of interest in applying machine learning based classification techniques for static program partitioning. In such static methods, program features characterizing execution time performance are extracted at compilation time. Based on the derived feature vector, an optimal partition of the program is predicted. With the same objective as well as workflow, we emphasize the inclusion of control flow divergence as an important program feature. We extend the feature set reported in [9] with control flow divergence and analyze the resulting improvements in classification accuracy. We build upon an existing static analysis method for divergence detection [13] and characterize control flow divergence as a performance feature in our ML based partitioning framework. The salient features of the contribution are as follows.

1. We implement a tool-flow which comprises three modules - a) a static analysis front-end for automatically extracting static program features, b) a module for determining the optimal partition of an OpenCL workload and c) a machine learning back-end which generates a classifier (i.e. the partitioner) model using supervised learning techniques. Details of the work-flow are described in Section 2.
2. We introduce the feature of control-flow divergence. We develop an analysis pass on top of the Ocelot framework [14] which computes the percentage of divergence of an OpenCL kernel. Details are depicted in Section 3.
3. We train classifier models with and without the feature of divergence and establish the fact that incorporating this feature benefits the accuracy of the partition function learned by the classifier. Experimental results are illustrated in Section 4.

In order to automate the tool-flow and accelerate the training process, we additionally implement a partition generator that can synthesize partitioned variants of OpenCL programs when a partition ratio is specified. With respect to some given OpenCL program, a partition variant is a functionally equivalent program with a different ratio of workload sharing among GPU and CPU.

Our experiments convincingly establish that existing classifier based static partitioning methods are not really effective for a large class of control-flow intensive workloads and the inclusion of the divergence feature provides real improvement in performance of the predicted and synthesized partitioned program variants. The non-triviality of the inclusion of the analysis can be further comprehended from our observations on program specific partition results as discussed in Section 4. We conclude our paper with a summary and scope for future work in Section 5.

2. WORKFLOW

A typical OpenCL program comprises two parts - a single threaded host program which executes on the CPU and the kernel which is data-parallel code to be executed on devices with support for vector processing. Data initialized in the host memory is first copied to the different devices of the system. The host program can either *Just-in-Time*-compile (JIT compile) the kernels or load the kernel binaries directly on the devices. Since we perform static analysis of kernel source, we adopt JIT mode of execution. The host generated kernel binaries process the data copied on the respective devices.

Once the kernel processing is finished, the results are again copied back to the host memory for further post-processing. The partitioning problem in the context of OpenCL programs entails splitting the data defined in the host memory space in a fixed ratio into segments and distributing it across the devices of the heterogeneous system. Our work is based on a single CPU-GPU system and so the data space is split into two distinct segments in a fixed ratio. We fix a resolution of 10 for classifying the different possible split ratios into partition classes which are defined as follows.

DEFINITION 1. *The **partition class** of a program P is given by an integer from the set $D = \{0, 1, 2, 3..., 10\}$ which indicates the split-ratio of the N-dimensional host data space of P across the CPU and the GPU. If the partition class of P is considered as i $(0 \leq i \leq 10)$ then it signifies that the program P executes in CPU for $(100-i\times10)\%$ of its computation domain. The program executes in GPU for the rest of the data space.*

Allowing program partitions at too fine granularities do not make much sense unless the input size for the programs are very large so that the CPU-GPU data transfer overhead do not make nearby partitions very difficult to disambiguate. Based on the notion of partition classes, the formal problem statement is as follows.

Let $F = \langle f_1, \cdots, f_n \rangle$ denote a set of static features of a program P to be mapped on a heterogeneous architecture comprising a CPU and a GPU. The static program features of P are denoted by $F(P) = \langle f_1(P), \cdots, f_n(P) \rangle$. We intend to learn a classifier function C such that $C(F(P)) \in D$ where $C(F(P))$ is the predicted optimal partition class of the program P.

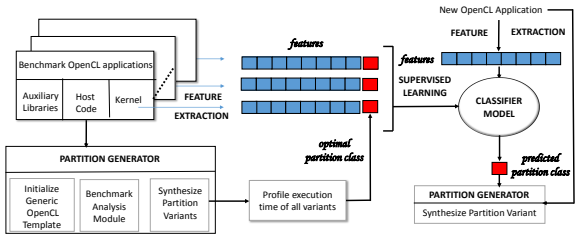

Figure 2: Flowchart for training and testing phases

The overall workflow for training a classifier model followed by evaluating its quality is depicted in Figure 2. The individual components of the figure are described as follows.

1. Benchmark OpenCL applications from various vendors are chosen. These benchmarks typically contain OpenCL host codes with problem specific auxiliary library routines and a set of OpenCL kernels.
2. We create a *Partition Generation* framework for automatically synthesizing the partition variants of OpenCL kernels. For a given kernel P and a target partition class $k \in D$, the partition generator synthesizes the desired program variant belonging to the partition class k using automated source level transformations. A similar automated single OpenCL kernel partitioning framework has been reported in [10].
3. Static information pertaining to device code is extracted using *Feature Extraction*. We implement LLVM compiler [15] passes in order to obtain the static program features characterizing computation power, memory usage, problem size and control flow behavior. We consider the entire feature set reported in [8] which characterize the first three of the listed properties. Additionally, we quantify the performance aspect of control flow behavior by evaluating the percentage of divergence and including it in the feature set.
4. Given a set B of popular OpenCL benchmarks, we create all possible partitioned variants using the tool-flow described in step 2. All the variants are executed in the target heterogeneous platform and profiled. The variant with the minimum execution time is marked for every test program. In that way we create a data-set comprising entries given by $\{\langle F(P), opt(P)\rangle \mid P \in B\}$ where $opt(P)$ is the optimal partition of program P as found by execution time profiling of all partition variants of P and $F(P)$ is the feature vector of P as computed using our front-end static analysis (Step 3). We use this labelled training dataset and use Supervised Learning methods to generate a classifier model.
5. To evaluate the quality of the learned classifier model, partition classes predicted for unseen program instances are compared with the optimal partition class values.
6. For a new program ($\notin B$), when the classifier predicts a partition class, we use the *Partition Generation* Framework to synthesize the desired partitioned variant.

In that way we have an automated source-level transformation tool for optimizing OpenCL programs for execution in heterogeneous platforms. As discussed earlier, in our tool, control-flow divergence plays an important role in deciding the partition class and the subsequent task of partition variant synthesis. We discuss the properties of this program feature in detail as follows.

3. CONTROL FLOW DIVERGENCE

Procedures for handling divergence differ between CPU and GPU architectures. In the performance perspective, this actually transpires to optimal partitions being different for two programs having similar static characteristics except for the amount of possible divergence. While GPU architectures have dedicated hardware support for predicated execution, CPU architectures employ software predication based approaches [16] for handling divergence. We exemplify the difference in divergence handling by comparing the respective assembly codes of the following code snippet.

```
__kernel void divergent(__global int *A,int N){
  int tid=get_local_id(0);
  int gid=get_global_id(0);
  float x=A[gid];
  if(tid<11){A[gid]+=sqrt(x);A[gid]*=3;}
  else{A[gid]-=sqrt(x);A[gid]+=11;}
}
```

```
.LBB0_2: /*CPU*/
vpcmpgtd %ymm0,%ymm1,%
     ymm5
vmovups (%edx),%ymm6
vsqrtps %ymm6,%ymm7
vaddps  %ymm7,%ymm6,%ymm1
vmulps  %ymm2,%ymm1,%ymm1
vsubps  %ymm7,%ymm6,%ymm6
vaddps  %ymm3,%ymm6,%ymm6
vblendvps %ymm5,%ymm1,%
     ymm6,%ymm1
vmovups %ymm1, (%edx)
```

```
BB0_1: /*GPU*/
ld.global.f32   %f1,[%rd4];
sqrt.approx.f32 %f2,%f1;
mov.u32         %r4,10;
setp.gt.s32     %p1,%r3,%r4;
@%p1 bra        BB0_2;
add.f32         %f3,%f2,%f1;
st.global.f32   [%rd4],%f3;
BB0_2:
sub.f32         %f6,%f1,%f2;
st.global.f32   [%rd4],%f6;
ret;
```

Figure 3: CPU and GPU Assembly Code

The corresponding CPU assembly and GPU PTX (Parallel Thread Execution) codes are illustrated in Figure 3. In the CPU assembly code, the square root, addition, subtraction and multiplication instructions (vpsqrtps, vpaddps, vpsubps and vpmulls) in both the `if` and `else` blocks are executed and stored in registers `%ymm1` and `%ymm6` respectively. The `vpcmpgtd` instruction computes the mask vector. The function `vblendvps` selects the register value depending on the value of the mask vector and subsequently stores it in the global array. This procedure forces the computations in both the blocks to be executed. However, in the GPU assembly code the respective computations in the blocks are performed by the corresponding threads only. The predicate %p1 is computed using the `setp` instruction. The PTX branch instruction depends on this

predicate. The assembler sets a branch synchronization marker that pushes the current mask vector on a stack inside each thread. Inactive work items for the `if` block are masked out and are not executed. At the end of execution of a conditional block, the mask vector is popped, its bits are flipped and again pushed into the stack. The remaining work items execute the other conditional block in sequence. In contrast to the CPU, the SIMD threads in the GPU execute the respective instructions that they are supposed to in the conditional blocks. The CPU executes instructions in both the conditional blocks for each thread.

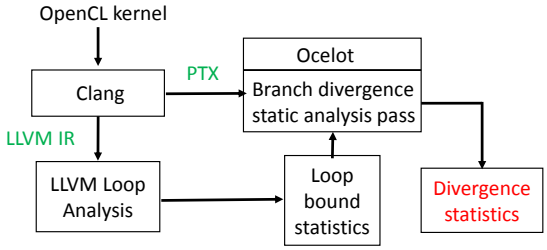

Figure 4: Framework for obtaining control flow features.

We now discuss how the features related to control-flow are computed. The flowchart for obtaining control features is shown in Fig. 4. We use clang[17] to compile OpenCL kernels to LLVM IR. An LLVM compiler pass converts the LLVM IR to PTX (Parallel Thread Execution) which is a virtual instruction set for general purpose parallel programming. There exists a unique mapping between LLVM IR and PTX backend [18]. The LLVM IR is passed to an LLVM loop analysis module which computes the iteration count of each loop. For loops dependent on thread-ids, this count is actually an upper bound. The PTX code generated is passed to the Ocelot compiler framework. Ocelot includes a static analysis proposed by [13] which iterates through all the branch instructions in order to identify which of them are divergent. By invoking the loop as well as the divergence analysis over a program, we identify the set L of loops, the bound n_i for every loop $l_i \in L$, the set B of branches. Also, we identify the set $B_l = \{(b_i, l_i) \mid l_i \in L\} \subseteq B$ of branches inside loops and the set $B_{div} \subseteq B$ of divergent branches. Hence, the worst-case static estimate of branch instructions which shall execute becomes $\sum_{b_i \in B_l} n_i + |B \setminus B_l|$. The static estimate of diverging branch instructions which shall execute becomes $\sum_{b_i \in B_l \cap B_{div}} n_i + |B_{div} \cap (B \setminus B_l)|$. We thus obtain the total number of potential branches as well as the percentage of divergence.

There exists other static analysis techniques for divergence detection in data-parallel programs [19, 12, 16]. We employ the analysis reported in [13] because of improved precision and support of the formal semantics of an SIMD execution model [20]. The notion of processing elements operating in parallel and associated operations on them constitute the basic foundation of the SIMD execution model. The PTX representation provides the relevant instructions and constructs for the same making it the ideal choice for static analysis pertaining to divergence.

Static estimates of divergence are naturally conservative. Such conservative approximations can surely be refined using hybrid techniques involving dynamic feature extraction. However, the present work is only confined to a purely static method and in majority scenarios, static estimates work fine for our approach as will be revealed in the subsequent section. After computing the feature of control flow divergence we move on to the training stage where we train classifier models and evaluate their quality.

4. EXPERIMENTAL RESULTS

We evaluate the approach discussed so far on a heterogeneous system consisting of an NVIDIA Tesla C2050 GPU clocked at 0.95 GHz and an Intel Xeon E5620 CPU clocked at 2.4 GHz with Hyperthreading technology. We have used a total of 56 kernels from various OpenCL benchmarks such as NVIDIA SDK[21], AMD-APP SDK[22], Parboil[23] and Rodinia [24]. We generate 206 training data points for our dataset by varying the number of work items and problem size. The work can be validated on any other CPU-GPU system, only by repeating the training phase for that platform.

We use the Weka toolkit [25] to train and evaluate our machine learning models. For our evaluation method, we use the popular *leave one out* cross-validation method to compute the accuracy. We have trained a two-level hierarchical classifier using the entire feature set with both Decision Trees (DT) and Radial Basis Function (RBF) networks initially. The first level of the hierarchical classifier focuses on the two extreme ends of execution i.e., purely GPU and purely CPU. If the predicted class is inconclusive, the program is sent to the second level classifier. The second level classifier is relatively complex and predicts one out of the eleven partition classes for the corresponding program.

The leave one out cross-validation accuracies for the first level DT and RBF classifiers are 83.71% & 89.23% respectively. For the second level DT and RBF classifiers, these valuse are 75.68% & 80.84% respectively. The difference in the accuracy values of DT and RBF may be attributed to the fact that the problem of finding the optimal partition is hard with non-linearity of class boundaries. In this regard, RBF networks are able to predict the class boundaries more efficiently than DT.

In order to establish the impact of divergence on the optimal program partition, we now train two RBF classifiers, *DIV* and *NDIV* which include and exclude the branch divergence feature respectively. It may be observed from Table 1 that there is a considerable difference between the accuracies of *DIV* and *NDIV* for both the individual levels as well as the combined model.

Table 1: Classifier accuracy of various models

Models ↓	RBF network accuracy	
	Without Divergence	With Divergence
CPU-GPU-Inconclusive	84.12%	89.23%
Partition CPU-GPU	66.37%	80.84%
Combined Model	64.31%	81.23%

A selected list of popular benchmark programs with varying degrees of divergence and different work-item sizes from our experimental dataset is provided in Table 2 . Partition classes predicted by *NDIV* and *DIV*, and the class representing the optimal partition along with associated execution times for the benchmark programs are also presented. Based on these experimental results, we discuss the impact of branch divergence on the decision of the classifiers.

The kernels *Histogram256* (100% divergence) and *Histogram64* (75% divergence) present complex branching behaviour and a high level of branch divergence. *Histogram256* contains three distinct `for` loops whose branch conditions are dependent on the thread-id and are divergent. Histogram64 contains a relatively simpler structure with two divergent `for` loops and a divergent `if` statement containing a non-divergent `for` loop. The classifier *DIV* correctly predicts the optimal partition class value for both the programs. In contrast, classifier $N-DIV$ fails to understand the overhead caused by the divergent branches in the programs and assumes that uniform computation is being done inside the loops, thereby mispredicting it and leading to an increase in execution time.

For kernels with lower levels of divergence, the difference in partition classes and execution times are considerably less among *DIV* and *NDIV*. In such cases, the chances of misclassification due to *DIV* is possibly as high as it is due to *NDIV*. The kernel BoxFilter-Horizontal (16.67% divergence) is correctly mapped to the optimal

Table 2: Predicted execution time of divergent programs

Benchmark OpenCL kernels	Percentage of branch divergence	No. of work items	Prediction by NDIV		Prediction by DIV		Optimal partition	
			Execution time (ms)	Partition class	Execution time (ms)	Partition class	Execution time (ms)	Partition class
nw_kernel2 (RODINIA)	34.20%	2048	7.33	5	5.6	8	5.6	8
splitSort (PARBOIL)	44.44%	664064	6.97	6	5.43	10	5.43	10
mb_sad_calc (RODINIA)	66.67%	2048	265.68	0	127.63	10	127.63	10
		4096	290.35	6	133.45	10	133.45	10
lud_perimeter (RODINIA)	20%	2048	14.27	10	14.27	10	10.33	8
uniformAdd (PARBOIL)	100%	8389120	0.28	7	0.412	10	0.0.28	7
larger_sad_calc16 (PARBOIL)	100%	38404	7.74	10	7.74	10	7.74	10
		20482	3.87	10	3.87	10	3.87	10
Histogram256 (NVIDIA)	100%	279680	2.05	0	0.33	10	0.33	10
Histogram64 (NVIDIA)	75%	279680	1.52	0	0.22	10	0.22	10
Reduction4 (NVIDIA)	62.5%	32768	0.018	10	0.018	10	0.018	10
		65536	0.023	10	0.023	10	0.023	10
Reduction0 (NVIDIA)	50%	32768	0.029	10	0.029	10	0.029	10
		65536	0.042	10	0.042	10	0.042	10
MatVecMulCoalesced1 (NVIDIA)	50%	157987120	103	0	45	10	45	10
SimpleConvolution (AMD)	50%	2097152	2.74	10	2.74	10	2.58	8
MatVecMulCoalesced0 (NVIDIA)	37%	157987120	62.16	0	49.13	10	49.13	10
HorizontalSAT0 (AMD)	50%	16777216	30.31	7	17.32	8	17.32	8
BoxFilterHorizontal (AMD)	16.67%	8388608	15.57	8	14.49	7	14.49	7
		2097152	5.02	6	3.68	7	3.68	7
BoxFilterVertical (AMD)	16.67%	8388608	16.73	8	15.68	7	15.68	7
		2097152	3.96	7	3.96	7	2.99	8

partition class for both the work-item sets by *DIV* while it is incorrectly predicted by *NDIV*. The kernel BoxFilterVertical (16.67%) is correctly mapped to the optimal partition class for the first work-item set by *DIV*, while it is incorrectly predicted by *NDIV*. For the second work-item set, both the classifiers fail to classify correctly.

We now list cases where the feature of divergence fails to add any insight or misguide the decision of the classifier. The decisions inferred by both the classifiers for the reduction kernels - Reduction0 (62.5% divergence) and Reduction4 (50% divergence) are similar. These kernels contain `for` loop structures with nested `if` statements whose branch conditions depend on the thread-ids and the induction variables of the `for` loop suggesting high levels of divergence. However, the execution time for these programs are determined primarily by the overhead due to use of synchronization primitives (barriers) and the possibility of shared memory bank conflicts. The feature of divergence does not provide any additional insight here and both the classifiers for the kernels predict the optimal partition class values. The kernel *large_sad_calc_16* (100% divergence) contains one divergent `if` statement and one divergent `for` statement. The branch condition associated with the `for` loop is dependent on the thread-id and a variable which is computed during runtime. The value of this variable evaluates to the size of the work-group in this program. This scenario ensures that all the threads in a work-group are active and there is no control flow divergence. However the static analysis fails to detect this and labels the branch as divergent. We observe that there is no change in the decisions made by the two classifiers. Both the classifiers predict the optimal partition class value 10 for this program, owing to the high amount of regular computation present in the kernel favouring GPU friendly execution. The kernel *uniformAdd* contains `if` statements which are labelled as divergent but are actually not during runtime. The classifier *NDIV* correctly assigns the program with the optimal partition class value here, but the classifier model *DIV* mispredicts as it is under the impression that the program is divergent according to the static analysis. However, the computation involved is much less and so the difference between program execution times for different partition classes are not significant. As suggested by the aforementioned accuracy measures, the classifier *DIV* is incorrect in 18.77% cases for our experimental dataset and this is primarily because of these specific class of test cases. The classifier *DIV* is considerably accurate for other benchmark programs, especially highly divergent ones.

5. CONCLUSION

The work proposes a ML based static approach for partitioning OpenCL programs and emphasizes the importance of incorporating control flow divergence in the feature set. The classifier produces considerably better partitioning results when divergence is considered as a program feature. Future work includes refining the feature of divergence by improving its precision over existing methods. An interesting extension would be to investigate how the partitioning will be influenced using modern CPU architectures which have support for AVX instructions. We shall further inspect the impact that divergence would have while scheduling multiple OpenCL tasks in complex heterogeneous systems and design efficient heuristics for the same.

6. REFERENCES

[1] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping. In *Micro*, pages 45–55. IEEE, 2009.

[2] Khronos OpenCL Working Group et al. OpenCL-The Open standard for Parallel Programming of Heterogeneous Systems. *http://www.khronos.org/opencl*, 2011.

[3] Vignesh T Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. Compiler and Runtime Support for enabling Generalized Reduction Computations on Heterogeneous Parallel Configurations. In *ICS*, pages 137–146, 2010.

[4] Prasanna Pandit and R Govindarajan. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *CGO*, page 273, 2014.

[5] Rashid Kaleem, Rajkishore Barik, Tatiana Shpeisman, Brian T. Lewis, Chunling Hu, and Keshav Pingali. Adaptive Heterogeneous Scheduling for Integrated GPUs. In *PACT*, pages 151–162, 2014.

[6] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: A Programming Model for Heterogeneous Multi-core Systems. In *ASPLOS*, pages 287–296, 2008.

[7] Zheng Wang and Michael FP O'Boyle. Partitioning Streaming Parallelism for Multi-cores: A Machine Learning Based Approach. In *PACT*, pages 307–318, 2010.

[8] Dominik Grewe and Michael FP O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems using OpenCL. In *CC*, pages 286–305, 2011.

[9] Dominik Grewe, Zheng Wang, and Michael FP O'Boyle. OpenCL Task Partitioning in the Presence of GPU Contention. In *LCPC*, pages 87–101, 2011.

[10] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. An Automatic Input-Sensitive Approach for Heterogeneous Task Partitioning. In *ICS*, pages 149–160, 2013.

[11] Yuan Wen, Zheng Wang, and Michael O'Boyle. Smart Multi-task Scheduling for OpenCL programs on CPU/GPU Heterogeneous Platforms. In *HIPC*, 2014.

[12] Bruno Coutinho, Diogo Sampaio, Fernando Magno Quintao Pereira, and W Meira. Divergence Analysis and Optimizations. In *PACT*, pages 320–329, 2011.

[13] Diogo Sampaio, Rafael Martins, Sylvain Collange, and Fernando Magno Quintao Pereira. Divergence Analysis with Affine Constraints. In *SBAC-PAD*, pages 67–74, 2012.

[14] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: A Dynamic Optimization Framework for Bulk-Synchronous Applications in Heterogeneous Systems. In *PACT*, pages 353–364, 2010.

[15] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, pages 75–86, 2004.

[16] Ralf Karrenberg and Sebastian Hack. Improving Performance of OpenCL on CPUs. In *CC*, pages 1–20, 2012.

[17] Clang: A C Language Family Frontend for LLVM. http://clang.llvm.org.

[18] Helge Rhodin. A PTX Code Generator for LLVM. Master's thesis, Saarland University, 2010.

[19] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs. In *CGO*, pages 111–119, 2010.

[20] Craig A Farrell and Dorota H Kieronska. Formal Specification of Parallel SIMD Execution. *Theoretical Computer Science*, 169(1):39–65, 1996.

[21] NVIDIA OpenCL SDK. http://developer.nvidia.com/opencl.

[22] AMD-APP SDK. http://developer.amd.com/tools-and-sdks/.

[23] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. *Center for Reliable and High-Performance Computing*, 2012.

[24] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IISWC*, 2009.

[25] Ian H Witten, Eibe Frank, Leonard E Trigg, Mark A Hall, Geoffrey Holmes, and Sally Jo Cunningham. Weka: Practical Machine Learning Tools and Techniques with Java Implementations. 1999.