Automatic Partitioning of Stencil Computations on Heterogeneous Systems

Alyson D. Pereira*, Rodrigo C. O. Rocha†, Luiz Ramos§, Márcio Castro* and Luís F. W. Góes‡
* Universidade Federal de Santa Catarina, Email: alyson.pereira@posgrad.ufsc.br, marcio.castro@ufsc.br
‡ Pontifícia Universidade Católica de Minas Gerais, Email: lfwgoes@pucminas.br
§ Universidade Estadual de Campinas, Email: luiz.ramos@ic.unicamp.br
† University of Edinburgh, Email: r.rocha@ed.ac.uk

***Abstract*—The stencil pattern is important in many scientific and engineering domains, spurring great interest from researchers and industry. In recent years, various optimizations have been proposed for parallel stencil applications running on GPUs. However, most of the runtime systems that execute those applications often fail to fully utilize the parallelism of modern heterogeneous systems. In this paper, we propose a mechanism based on machine learning that automatically partitions stencil computations across CPU and GPU. We implemented it into the PSkel framework and found that the mechanism can boost the performance of stencil applications on average by 17.9x compared to their sequential CPU-only counterparts, by 1.34x compared to a GPU-only version, and by 1.48x compared to a parallel CPU-only version.**

***Keywords*—Stencil, Work Partitioning, Decision Tree Learning**

I. Introduction

In recent years, Graphics Processing Units (GPUs) have been used in conjunction with general-purpose CPUs to enable High Performance Computing (HPC) with high energy efficiency. While modern CPUs use large caches and provide multiple out-of-order cores with branch prediction and speculation, GPUs are much richer in floating-point units and provide large amounts of simple processing cores.

Despite being commonly found in the same hardware platform or even on the same chip, CPUs and GPUs typically have different application programming interfaces. In fact, widespread programming approaches for GPUs provide very low-level programming abstractions. Furthermore, most of them require knowledge and careful use of the GPU memory model to maximize parallelism. Thus, programming parallel applications for CPU-GPU heterogeneous platforms can be challenging, tedious, and error-prone, even for experienced programmers [1].

A common approach to address such complexity is the use of algorithmic skeletons. Parallel skeletons model and abstract common parallel programming patterns (computation and coordination phases), thereby enabling the programmer to focus on algorithm design, rather than on runtime system details. Among existing parallel skeletons, the stencil pattern is critical in many scientific computing domains, including computational fluid dynamics and image processing [2].

In addition to reducing programming complexity in modern parallel architectures, it is critical to maximize their processing efficiency. However, most existing runtime systems for CPU-GPU platforms may not exploit the platform's potential. Specifically, they lack work partitioning mechanisms to split the computation of a parallel application across CPUs and GPUs, thereby improving the overall processing utilization. Based on those observations, previous research proposed frameworks with runtime systems that enable automatic work partitioning via static and/or dynamic strategies [3]–[5]. However, they commonly overlook specific application aspects that could help improving their partition optimization strategies.

In this paper, we propose and evaluate an automatic work partitioning mechanism, based on machine learning techniques, for heterogeneous systems. We implement our mechanism in the PSkel framework for stencil computations [6], which provides a single high-level programming abstraction across CPU and GPU. We evaluate the performance of the proposed work partitioning mechanism using three stencil applications. For those applications, our proposed mechanism has an average performance improvement of 17.9x compared to their sequential versions, and improves the average performance by 1.34x and 1.48x compared to their GPU-only and CPU-only parallel versions, respectively. In summary, our main contributions are as follows: (1) we propose an automatic work-partitioning mechanism, based on machine-learning, for iterative stencil applications, capable of predicting the best partitioning based on profiling information; (2) we show that the optimal partitioning scheme of a given application is hardware dependent and our mechanism is portable across systems; (3) we extend the PSkel framework with the proposed mechanism; and (4) we evaluate the mechanism and compare it against other approaches.

The remainder of this paper is organized as follows. Section II provides background on the stencil pattern and discusses related work. Section III describes the automatic work partitioning mechanism. Finally, Sections IV through VI present our evaluation method, results, and conclusions.

II. Background and related work

In this section, we provide background on the stencil parallel pattern and discuss past efforts in work partitioning for CPU-GPU systems.

A. The Stencil Pattern

The stencil pattern operates on n-dimensional data structures, using an input data value and its neighbors to compute the corresponding output data element [7]. Specifically, a sliding window (also called *mask*) scans the entire set of input data and produces output data using a stencil function. The mask size corresponds to a specific number of neighbors of each element of the input data. The stencil function performs computations using the mask and the neighbors of each element of the input data to produce a corresponding element in the output data. The stencil application repeats that process on every element of the input data.

Tiling is a common technique used to process large amounts of data on limited hardware resources. Tiling techniques for iterative stencil computations have been vastly studied in the past and remains relevant even in modern CPU and GPU architectures, where memory can be scarce in light of applications with large data sizes [2], [8]–[10]. In addition, in those systems, the memory bandwidth may be a bottleneck, especially given the data transfer overheads that tiling imposes. Overall, existing CPU and GPU approaches typically entail transforming iteration loops (time dimension) and stencil loops (space dimensions), provided that the stencil data is *enabled*; i.e., available at the memory level at which the processing elements can apply the loop transformations [11]. In CPUs, that level is typically the CPU cache, whereas in GPUs, the level is either the shared or the global memory of the device.

To illustrate and detail how tiling can be applied to stencil computations, we use a formal definition. Let A be a 2D data matrix, with dimensions $\dim(A) = (w, h)$, where w and h are, respectively, its width and height. Using tiles of dimensions (w', h') yields $\lceil \frac{w}{w'} \rceil \lceil \frac{h}{h'} \rceil$ possible tiles of A. Let $A_{i,j}$ be one such tile, where $0 \leq i < \lceil \frac{w}{w'} \rceil$ and $0 \leq j < \lceil \frac{h}{h'} \rceil$ $A_{i,j}$ has offset (iw', jh') relative to the top left corner of A, and $\dim(A_{i,j}) = (\min\{w', w - iw'\}, \min\{h', h - jh'\})$. The offset is an indexing displacement required for accessing the elements of the tile.

Applying a stencil on A, entails computing an element-wise neighborhood function (mask) containing the displacement of each neighbor of a given central element. Due to neighbor dependence, computing the stencil function along the boundary of a tile requires obtaining values from adjacent tiles. Let r be the range of the neighborhood mask, i.e., r is the most distant displacement required for the neighborhood defined by the mask. The area of range r comprising the neighborhood is often denominated *halo region*. If the stencil function is applied iteratively on A, for t iterations, the neighborhood dependence among tiles limits the number of iterations that can be consecutively computed without a proper memory synchronization in the computing device.

To enable performing $t' \in [1, t]$ consecutive iterations for each tile $A_{i,j}$, a tile must be enlarged to include a number of adjacent halo regions, collectively called a *ghost zone*. The number of adjacent halo regions that compose the ghost zone is proportional to t'. The overlap across neighboring tiles allows the GPU to generate its halo regions locally for a number of consecutive iterations proportional to the size of the ghost zone. In particular, the enlarged $A_{i,j}$ tile transferred to the GPU global memory has enlarged dimensions $\dim^*(A_{i,j}) = (\max\{\min\{w' + 2rt', w - iw' + rt'\}, 1\}, \max\{\min\{h' + 2rt', h - jh' + rt'\}, 1\})$ and has offsets $(\max\{iw' - rt', 0\}, \max\{jh' - rt', 0\})$, relative to A.

Because of neighborhood dependencies in stencil computations, each iteration on an enlarged tile adds noise to the ghost zone from the border towards the center of the tile. After t' consecutive iterations, only the logical tile region holds correct and non-overlapping values. The larger the value of t', the more redundant computations are required to correctly compute the logical tile region. Thus, sizing the ghost zones poses a trade-off between the cost of redundant computations and the reduction in communication when processing iterative stencil computations on GPUs. Despite the potential benefit of this trapezoidal tiling technique, an improper selection of the ghost zone size may negatively impact the overall performance [8], [10].

In this work, to enable the work-partitioning of iterative stencil computations between CPU and GPU, we adapt the trapezoidal tiling technique available in PSkel [10] to generate two enlarged data tiles, one for the GPU and other for the CPU. The size of each tile is transparently calculated according to the work distribution between CPU and GPU.

B. Previous Efforts on Automatic Work Partitioning on CPU-GPU Systems

Several researchers have studied work partitioning mechanisms for modern multi-GPU and CPU-GPU systems seeking better utilization of processing elements and application response times. For example, the compiler and runtime framework proposed by Ravi et al. maps annotated reduction patterns to CPU-GPU systems and dynamically assigns chunks of computation across devices [5]. However, the performance of the applications relies on a good selection of the chunk sizes, and, unlike our work, they do not take any application runtime characteristics into account for work distribution.

Grewe and O'Boyle [3] adopt a ML-based compiler model that performs static partitioning of data-parallel tasks based on program code and runtime features. Unlike in our approach, their work uses SVM, and does not focus on any parallel pattern. Instead, they use OpenCL code to generate a single kernel for multiple devices. Moreover, they extract the machine learning features from a compiler analysis of the program source code. While their approach is a good option to characterize an application and avoid online profiling, the obtained features can change since the compiler may perform code optimizations after the extraction of features, for example by the OpenCL JIT compiler, and it is also not possible to obtain features that describes the runtime behavior of the application.

Ganapathi et al. [12] prune the search space using statistical machine learning to boost the prediction accuracy of optimizations parameters for multicore stencil codes focusing on performance and energy efficiency. With a similar goal, Luo et al.

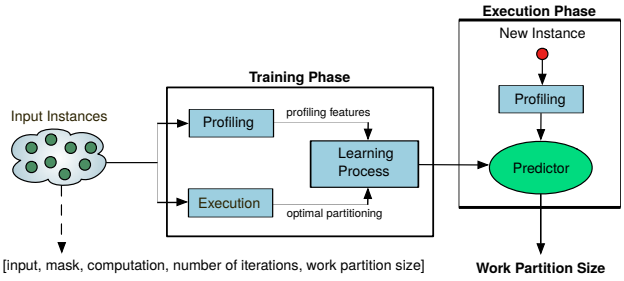

Fig. 1: Overview of the proposed approach.

[13] prune the optimization space by feature similarity, which their system obtains from its self-learning optimal solution-space database. However, their approach requires a domain-specific language and, as the work by Ganapathi, does not consider work partitioning across CPU and GPU.

Shen et al. [14] focus on imbalanced applications by finding the best match between a qualitative model of the application and the hardware capabilities of the execution environment. In a follow-up work focused on multi-kernel applications, the authors exploit inter-kernel parallelism and select a partitioning strategy based on the execution flow of the application [15]. To eliminate the need for a training phase, Kim et al. [4] assume homogeneous device performance in multi-GPU systems and partition work homogeneously across devices.

Luk et al. [16] perform curve-fitting to map the partitioned computations to processing elements based on an analytical model that take into account the application, problem size, and platform configurations. The approach uses dynamic compilation to adapt to runtime changes. Lee et al. [17] propose a greedy algorithm to partition work across multiple processing elements of varying computational power. Their algorithm uses a top-down induction tree where a node represents a distribution of work across devices and taking an edge entails estimating response time based on the data transfer cost and performance variation in the work execution times.

III. Automatic Work Partitioning Mechanism

Our solution relies on the ML-based framework proposed in [18], which is comprised of an offline training phase and an online execution phase. Figure 1 depicts the framework used in our approach. The offline phase entails: (1) producing a training set composed of *input instances* (i.e., features and an outcome variable), generated by a synthetic stencil benchmark; (2) selecting a set of performance-impacting *features* of the stencil computations; (3) creating a predictor based on the selected features to infer the best work partition size for a given scenario; (4) evaluating the predictor's accuracy; and (5) adjusting the prediction model. The online phase entails: (1) online profiling of the target application on the target execution environment; (2) applying the prediction model to select the best partition size of data and computation; and (3) performing work assignment to the processing units using a work partitioning mechanism.

TABLE I: Input parameters and their values used in the synthetic application to generate the training set.

Input parameter	Value
Input sizes	128^2, 256^2, 512^2,1024^2, 2048^2, 4096^2, 8192^2
Mask sizes	4, 9, 12, 24, 25, and 49
Number of iterations	10, 20, 30, 40, 50 and 60
Arithmetic operation	Addition, multiplication or both

A. Training Set and Profiling Features

To create a training set for our predictive model, we adopt a synthetic stencil benchmark capable of exercising different characteristics of stencil applications, as described in Table I. In particular, we execute and profile the benchmark on the target execution environment using multiple combinations of input parameters and work partition sizes (e.g., assign 40% work to the GPU and 60% to the CPU).

We instrument the synthetic benchmark using the PAPI library [19]. PAPI provides an interface for accessing hardware performance counters available on modern microprocessors, with a neglible overhead on performance. These counters exist as a small set of registers that count events, i.e., occurrences of specific signals related to the processor's function. From these counters it is possible to define metrics that corresponds to the behavior of the application on the underlying hardware, e.g., cache-miss ratio.

Each combination of input parameters compose an input instance. For each instance, we capture all possible profiling features available on CPU and GPU (only for a single iteration of the input instance) and we also track the work-partitioning scheme that results in the best execution time. These profiling features compose the features of the input instances and the best work-partitioning is its outcome variable. This enables our approach to be applied to any CPU-GPU platform configuration.

We obtained a total of 18 features for the CPU, including: cache miss ratio on all cache levels; TLB misses; floating-point, branch, load, and store instructions; total instructions per cycle; and resource stalls per cycle. For the GPU we obtained a total of 104 features, including: cache hit ratio on all cache levels, data read, and write throughput; data read, and write transactions; floating-point and integer instructions; stalls; utilization of the load/store, arithmetic logic, control flow and memory GPU units; and the efficiency of the stream multiprocessors, floating-point units, branches, global memory load and stores.

B. Decision Tree Learning

Using the training set of the previous step, we create a predictor capable of inferring work partition sizes for new unobserved instances. Our predictor expresses partition sizes as a percentage of the work assigned to the GPU, which is our outcome variable. Since optimal work partition sizes typically translate to optimal application performance in each scenario, we aim for a highly-accurate predictor.

TABLE II: Profiling features with highest impact on performance.

Source	Acronym	Description
CPU	CPU_L3_TCM	Ratio of the number of L3 cache misses to the number of L3 cache accesses
	CPU_L2_DCM	Ratio of the number of L2 data cache misses to the number of L2 data cache accesses
GPU	GPU_L1_TCH	Hit rate in L1 cache for global loads
	CPU_L2_TCH	Hit rate at L2 cache for all read requests from L1 cache
Application	ITERATIONS	Total number of stencil iterations

To build our predictor, we adopt a decision tree learning algorithm, namely the Classification and Regression Tree (CART) algorithm [20]. CART outputs a decision tree based on a learning sample, composed of a set of historical data with pre-assigned classes for all observations. In CART, a decision tree node represents a binary (yes/no) question that splits the learning sample into smaller subparts. In particular, the algorithm searches all possible variables and values for the best split, i.e., the question that splits the data into two parts with maximum homogeneity. This process is successively repeated for the data fragments resulting from each step to produce a classification tree.

To create our predictor, we initially generate a decision tree using all the profiling features obtained. The generated decision tree will contain just a subset of features that are capable to define the best work-partitioning. However, it is possible to obtain a better accurate decision tree by swapping some of the features from the previous decision tree and perform a cross-validation test. We empirically optimize the prediction model to obtain the profiling features that most impact on performance. Table II summarizes the details of the selected features. The resulting predictor can infer a work partition size based on the profiling information of a new unobserved instance.

C. Work Partitioning Mechanism

For the online execution phase, we leverage the work partitioning and tiling mechanisms available in PSkel framework [6], [10]. Its runtime system is able to automatically partition the input data into two tiles for the CPU and GPU device, according to a user-specified argument designating what percentage of the input data should be assigned to the GPU. The tiling mechanism replicates the borders around the partition point in the input data so that each processing unit works on its private copy of the borders. That redundant computation eliminates the need for memory data-transfer, synchronization, and book-keeping for border exchange between processing units across different iterations of the application [10].

In order to obtain the features needed for the decision tree classifier, when a new iterative stencil application is executed, the framework profiles its execution during the first stencil iteration, simultaneously on both CPU and GPU devices. Then, based on the rules of the decision tree, the framework predicts the work-partitioning that should be applied to the execution of the remaining iterations of the application. During the simultaneous execution, the framework dedicates one CPU core to manage the GPU execution, while the remaining CPU cores perform the stencil computation on separate data tiles.

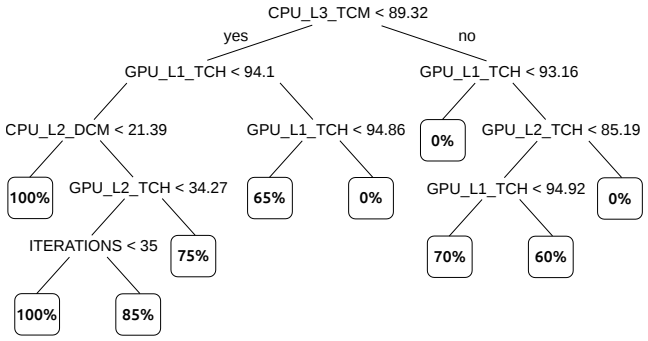

Fig. 2: Decision tree generated by RPART.

IV. Experimental setup

In this section, we describe our evaluation method. We carried out our experiments on a NUMA machine with two six-core Intel Xeon E-5645 processor with 32GB of RAM and a NVIDIA Quadro 2000 GPU. To build the applications, we used NVIDIA CUDA version 8.0 and GCC version 4.9.2.

Since our work partitioning approach adopts supervised learning, it requires input instances. To avoid exhaustive search, we aim at covering a significant space of the stencil spectrum by using a set of synthetic benchmarks to generate a training set. We then use that training set to build a decision tree for the predictive model of our automatic work partitioning approach.

We pragmatically selected a wide range of settings for the synthetic benchmarks, aiming to cover the main features of the stencil spectrum. In particular, we varied the input size, the mask size, the number of iterations, and the types of arithmetic operations performed on the stencil mask. Moreover, we vary the GPU work-partitioning size for each of the previous combinations in order to obtain its best work-partitioning scheme. We choose a total of 20 partitioning schemes, ranging from 0% to 100%, in multiples of 5%. It may be possible that a partition between these ranges has the best execution time, thus our best work-partitioning is an approximation.

Table I describes the values we considered for each feature. From these combinations, we profile a total of 162 synthetic applications in order to obtain the profiling features needed for our predictive model, since we only need to capture the behavior of the stencil applications during its first iteration. Then, we execute the remaining combinations with the selected work-partitioning schemes in order to obtain the best execution time for each of them. In total, we obtain 972 input instances to the decision tree predictor.

To generate the decision tree we used *RPART*[1], an open-source implementation of the CART algorithm. RPART specifically generates a decision tree given a training set with a group of features and a particular class for each input

[1]https://cran.r-project.org/package=rpart

TABLE III: Profiling features of the stencil applications.

App	Profiling Features (%)			
	CPU_L3_TCM	CPU_L2_DCM	GPU_L1_TCH	GPU_L2_TCH
Cloudsim	98.61	44.75	50.88	66.32
FAST	4.85	10.82	91.03	89.83
GoL	98.52	6.01	94.5	75.92

instance. Figure 2 shows the generated decision tree using the synthetic applications as input instances. Questions on the split nodes represent the profiling features we previous identified as the features with highest impact on performance. Leaf nodes indicates the amount of work-partitioning that must be assigned to GPU. Table II gives a description of each of feature.

To predict the best partitioning for a new unobserved stencil application, we profile the first stencil iteration of the application. Based on the value of each profiling feature, the decision tree algorithm outputs the best partitioning for the remaining iterations. We evaluated the generalization capability of the decision tree using a 2-fold cross-validation method with stratified sampling. In stratified sampling, each sample set has the same proportion of classes as the whole training set. For each fold, we have two sets s_1 and s_2. For the first fold we train on s_1 and test on s_2, followed by training on s_2 and testing on s_1 for the second fold. According to this method, our ML-based approach is 85% accurate when used to infer the best work partition size for unobserved instances.

To evaluate our proposed work partitioning mechanism, we considered three 2D stencil applications: **CloudSim**, which simulates cloud dynamics based on a cellular automaton [21]; **Game of Life (GoL)**, which is a stencil implementation of the cellular automaton Conway's "Game of Life" [22]; **FAST**, which implements corner detection in images [23].

Table III shows the profiling feature values obtained of each one of the stencil applications when running with input sizes of 8192^2. All applications present different feature values, which makes hard to predict the best work partition size for each case. It is important to mention that these feature values change considerably with different input sizes.

V. Experimental results

This section evaluates the performance of our work partitioning mechanism using the three applications discussed in Section IV. In particular, we fully utilize the available GPU memory and execute the applications iteratively with 8192^2 input matrices and we vary the number of iterations from 10 to 60 in multiples of 10.

We evaluate each application using different partitioning approaches, namely: automatic work partitioning (**AWP**); a sequential baseline version running on a single CPU core (**Sequential**); an OpenMP parallel version running on the CPU (**CPU-only**); a CUDA version running on the GPU (**GPU-only**); a naive partitioning strategy based on the ratio of the computing time of first stencil iteration on CPU and GPU, also considering the memory transfers to the GPU and the number of stencil iterations (**Naive**); and an **oracle**. The oracle always selects the work partition that provides the best execution time

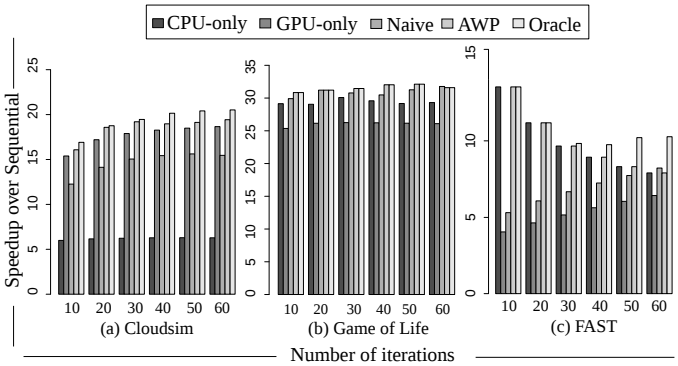

Fig. 3: Speedup over the sequential execution.

for each application and configuration. To provide a common basis for performance comparison, we compute the speedup of all variants relative to Sequential. Figure 3 shows the speedup of the stencil applications. The speedup takes into account the time spent by the GPU memory transfers. However, for the partitioned executions, those memory transfers are overlapped with computations on the CPU.

Figure 3a shows the results for CloudSim. Since the application is rich in floating-point operations, it tends to perform better on the GPU, where floating-point units are much more abundant than in the CPU. On average, GPU-only boosts the performance of CloudSim by 17.6x compared to Sequential, leaving little room for improvement towards the oracle's performance. The Naive partitioning picks a work-partitioning of 60%, however, while this partitioning schemes improves the performance over a CPU-only execution it degrades the performance in relation to a GPU-only execution. The AWP method predicts a GPU work partition of 75% and achieves a speedup of 18.53x on average for executions with multiple iterations (at least 94% of the oracle's performance, which has a optimal work-partitioning of 85%). In those cases, AWP improves the average performance of CloudSim by 3x and by 5% compared to CPU-only and GPU-only, respectively.

Figure 3b presents the speedup of GoL. The application features conditional branches that may reduce the potential performance gains on the GPU due to divergence [24]. In fact, CPU-only holds an 11% speedup over GPU-only and a 29.4x speedup compared to Sequential. With AWP, GoL performs up to 23% better than on GPU-only, achieving a overall speedup of 31.5x compared to Sequential, matching the oracle partition of 65%. The Naive partitioning also benefits GoL, performing at 98% of the oracle average performance.

Finally, Figure 3c presents the performance for FAST. Unlike Cloudsim, FAST is rich in control flow instructions. The resulting execution is imbalanced, and therefore less suitable for GPUs. In fact, the imbalanced execution can benefit significantly from the high-accuracy branch prediction in the CPU. For that reason, FAST performs 86% better on CPU-only than on GPU-only. The Naive partitioning picks a GPU work-partitioning of 75%, which performs less than a CPU-only execution except for a 60 iterations scenario where it slightly improves the performance by 4%. AWP predicts a

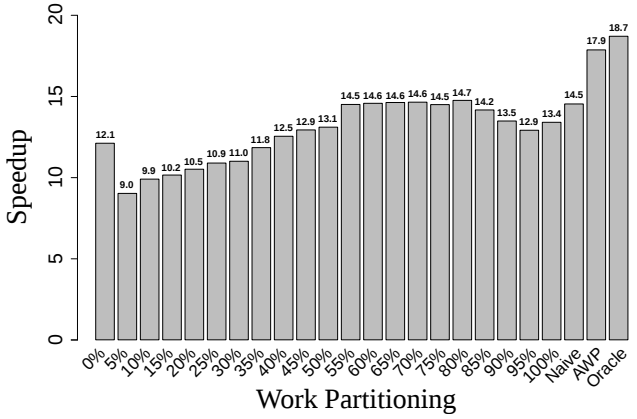

Fig. 4: Average speedup over all applications.

CPU-only execution for all iterations, corresponding to 91% of the oracle performance, which has 65% for GPU work-partitioning.

From the results above we conclude that harnessing the processing power of both CPU and GPU can significantly improve the performance of stencil applications. Figure 4 shows the average speedup (geometric mean) for all applications when selecting a fixed GPU work partition (from 0% to 100%), using the Naive and AWP approaches and the best partitioning (oracle) for each application. The oracle can improve the average performance of the applications by 18.7x compared to Sequential, by 1.39x (maximum of 3.35x) compared to GPU-only, and by 1.54x (maximum of 3.27x) compared to CPU-only. Our predictive model (AWP) outperforms any fixed work-partitioning and even the Naive approach for partition selection, and attain 96% of the oracle's performance on average. This represents an average performance improvement of 17.9x over Sequential, 1.48x over CPU-only (maximum of 3.09x), and 1.34x over GPU-only (maximum of 3.35x).

VI. Conclusion

In this work, we presented an automatic work-partitioning approach for stencil applications. We implemented it into the PSkel stencil framework, which provides a high-level programming interface for executing stencil computations on heterogeneous systems. We showed that a high-level programming interface integrated with an automatic work-partitioning mechanism can exploit the high performance offered by heterogeneous systems, while improving their ease of programming. Our results showed that the ML-based strategy improves the performance of parallel stencil applications on heterogeneous systems by up to 32x over the single-core implementation and by up to 3.35x compared to a GPU-only version, reaching more than 77% of the oracle's performance.

Future work includes the development of work-partitioning mechanism for more stencil applications and platforms; the improvement of the predictive model for imbalanced stencils; and the study of other automatic optimizations and adaptive optimization algorithms.

References

[1] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers," in *Proc. of 2011 Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, 2011.

[2] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in *Proc. 26th ACM Int. Conf. on Supercomputing*, 2012, pp. 311–320.

[3] D. Grewe and M. F. P. O'Boyle, "A static task partitioning approach for heterogeneous systems using OpenCL," in *Proc. 20th Int. Conf. on Compiler Construction*, 2011, pp. 286–305.

[4] J. Kim, H. Kim, J. H. Lee, and J. Lee, "Achieving a single compute device image in OpenCL for multiple GPUs," in *Proc. 16th ACM Symp. on Principles and Practice of Parallel Programming*, 2011, pp. 277–288.

[5] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal, "Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations," in *Proc. 24th ACM Int. Conf. on Supercomputing*, 2010, pp. 137–146.

[6] A. D. Pereira, L. Ramos, and L. F. W. Góes, "PSkel: A stencil programming framework for CPU-GPU systems," *Concurrency and Computat. Pract. and Exper.*, vol. 27, no. 17, pp. 4938–4953, 2015.

[7] M. D. McCool, "Structured parallel programming with deterministic patterns," in *Proc. 2nd USENIX Conf. on Hot Topics in Parallelism*, 2010, pp. 5–5.

[8] J. Meng and K. Skadron, "A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations," *Int. Journal of Parallel Programming*, vol. 39, no. 1, pp. 115–142, 2011.

[9] T. Lutz, C. Fensch, and M. Cole, "PARTANS: An Autotuning Framework for Stencil Computation on multi-GPU Systems," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 59:1–59:24, Jan. 2013.

[10] R. C. O. Rocha, A. D. Pereira, L. Ramos, and L. F. W. Góes, "TOAST: Automatic tiling for iterative stencil computations on GPUs," *Concurrency and Comput. Pract. and Exper.*, p. e4053, 2017.

[11] D. Orozco, E. Garcia, and G. Gao, "Locality optimization of stencil applications using data dependency graphs," in *Proc. 23rd Int. Conf. on Languages and Compilers for Parallel Computing*, 2011, pp. 77–91.

[12] A. Ganapathi, K. Datta, A. Fox, and D. Patterson, "A case for machine learning to optimize multicore performance," in *Proc. 1st USENIX Conf. on Hot Topics in Parallelism*, 2009, pp. 1–1.

[13] Y. Luo, G. Tan, Z. Mo, and N. Sun, "FAST: A Fast Stencil Autotuning Framework Based On An Optimal-solution Space Model," in *Proc. 29th ACM Int. Conf. on Supercomputing*, 2015, pp. 187–196.

[14] J. Shen, A. L. Varbanescu, P. Zou, Y. Lu, and H. Sips, "Improving performance by matching imbalanced workloads with heterogeneous platforms," in *Proc. 28th ACM Int. Conf. on Supercomputing*, 2014, pp. 241–250.

[15] J. Shen, A. L. Varbanescu, X. Martorell, and H. J. Sips, "Matchmaking Applications and Partitioning Strategies for Efficient Execution on Heterogeneous Platforms," in *44th Int. Conf. on Parallel Processing*, 2015, pp. 560–569.

[16] C.-K. Luk, S. Hong, and H. Kim, "Qilin: Exploiting Parallelism on Heterogeneous Multiprocessors with Adaptive Mapping," in *Proc. 42nd IEEE/ACM Int. Symp. on Microarchitecture*, 2009, pp. 45–55.

[17] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems," in *Proc. 22nd Int. Conf. on Parallel Archit. and Compilation Techniques*, 2013, pp. 245–256.

[18] M. Castro, L. F. W. Góes, C. P. Ribeiro, M. Cole, M. Cintra, and J.-F. Méhaut, "A machine learning-based approach for thread mapping on transactional memory applications," in *Proc. 18th Int. Conf. on High Performance Computing*, 2011, pp. 1–10.

[19] A. D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb, "Parallel performance measurement of heterogeneous parallel systems with GPUs," in *Proc. 2011 Int. Conf. on Parallel Processing*, 2011, pp. 176–185.

[20] L. Breiman, F. J. H., O. R. A., and S. C. J., *Classification and Regression Tress*, 1984.

[21] A. R. da Silva and M. M. Gouvêa, Jr., "Cloud dynamics simulation with cellular automata," in *Proc. Summer Computer Simulation Conf.*, 2010, pp. 278–283.

[22] M. Gardner, "Mathematical Games - The Fantastic Combinations of John Conway's New Solitaire Game 'Life'," *Scientific American*, vol. 223, no. 3, pp. 120–123, 1970.

[23] E. Rosten and T. Drummond, "Fusing Points and Lines for High Performance Tracking," in *Proc. 10th IEEE Int. Conf. on Computer Vision*, 2005, pp. 1508–1515.

[24] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in GPU programs," in *Proc. 4th Workshop on General Purpose Processing on Graphics Processing Units*, 2011, pp. 3:1–3:8.