Towards the Transparent Execution of Compound OpenCL Computations in Multi-CPU/Multi-GPU Environments*

Fábio Soldado, Fernando Alexandre, and Hervé Paulino

NOVA-LINCS / Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
herve.paulino@fct.unl.pt

Abstract. Current computational systems are heterogeneous by nature, featuring a combination of CPUs and GPUs. As the latter are becoming an established platform for high-performance computing, the focus is shifting towards the seamless programming of the heterogeneous systems as a whole. The distinct nature of the architectural and execution models in place raise several challenges, as the best hardware configuration is behavior and data-set dependent. In this paper, we focus the execution of compound computations in multi-CPU/multi-GPU environments, in the scope of Marrow algorithmic skeleton framework, the only, to the best of our knowledge, to support skeleton nesting in GPU computing. We address how these computations may be efficiently scheduled onto the target hardware, and how the system may adapt itself to changes in the CPU's load and in the input data-set.

1 Introduction

Most of the current computational systems are intrinsically heterogeneous, featuring a combination of multi-core CPUs and GPUs. However, the discrepancies of the programming and execution models in place make the programming of these hybrid systems a complex chore. Consequently, only experts with deep knowledge of parallel programming, and even computer architecture, are able to fully harness the available computing power.

The OpenCL specification has been designed with the purpose of enabling code portability across a wide range of architectures. However, the portability of performance is not guaranteed. In fact, it depends greatly on device-specific optimizations, which are cumbersome to implement, due to the low level nature of the programming model. Moreover, when targeting multiple devices, it is still up to the programmer to assume the parallel decomposition of the problem. Accordingly, the definition of high level programming constructs for heterogeneous computing has been the driver of a considerable amount of recent research both

* This work was partially funded by FCT-MEC in the framework of the PEst-OE/EEI/UI0527/2014 and PTDC/EIA-EIA/111518/2009 projects.

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part I, LNCS 8805, pp. 177–188, 2014.

at library and language level. A growing tendency is to build upon the notion of algorithmic skeleton [1–7].

We share this vision. Algorithmic skeletons render a template-based programming model that abstracts the complexity inherent to parallel computing by factorizing known solutions in the field into high level parameterizable structures. We claim that these characteristics can be used to, on one hand, hide the heterogeneity of the underlying hardware and, on the other, provide tools to cope with such heterogeneity, enabling device-specific parallel decompositions and optimizations. To that extent, we have been developing an algorithmic skeleton framework, entitled Marrow [8, 9], for the orchestration of OpenCL kernels. Marrow offers both data and task-parallel skeletons, and is the first on the GPU computing field to support skeleton composition, through nesting.

In this paper, we grow the Marrow framework to provide a skeletal programming model for the transparent programming of computational systems comprising multiple CPUs and GPUs. Our proposal distinguishes itself from the current state of the art by supporting the execution of compound computations, having in mind data locality requirements. Most of the current frameworks either expose the heterogeneity to the programmer [5] (and [10] when considering performance) or selectively direct the computations exclusively to one of their CPU and GPU back-ends [1–4, 10, 11]. In turn, the proposals that tackle the transparent conjoint use of both CPUs and GPUs either restrict their scope to the execution of single kernels [12,13] or require previous knowledge on the computation to run [14]. The contributions of this paper are thus on the definition of strategies to distribute the load of a Marrow compound computation across multiple CPUs and GPUs, and to adapt this distribution to different input data-sets and to the CPUs' load fluctuations.

2 The Marrow Framework

Marrow is a C++ algorithmic skeleton framework for the orchestration of OpenCL computations. It provides a set of data and task-parallel skeletons that can be combined, through nesting, to build compound computations. A Marrow computation may be interpreted as a tree of skeleton constructions that apply a particular behavior to their sub-tree, down to the leaf nodes, which represent the actual OpenCL kernels. The framework takes upon itself the entire host-side orchestration required to correctly execute these computational trees (CTs), including the proper ordering of the data-transfer and execution requests, and the communication between the tree nodes. The following skeletons are currently supported:

Pipeline - a pipeline of data-dependent CTs; **Loop** - *while* and *for* loops over a CT; **Map** - application of a CT upon independent partitions of a data-set; **MapReduce** - extension of *Map* with a subsequent reduction stage.

The Marrow programming model comprises two main stages: the construction of the CTs and the subsequent issuing of execution requests. The framework allows for the composition of arbitrary OpenCL kernels. Accordingly, for the sake of correctness and efficiency, setting up a CT leaf requires the specification of

the interface of the wrapped computational kernel, namely in what concerns its input and output parameters. For that purpose, the framework supplies a set of data-types to classify these parameters as vector or scalar values; mutable or immutable, and global or local. Moreover, the programmer may specify a kernel-specific local work-group size, for computations that are bound to particular sizes, and upon how many elements of the multi-dimensional range each computing thread (aka OpenCL work-unit) operates on. For instance, a thread may work upon multiple pixels of an image. This information will be used by the framework to compute the number of threads (OpenCL workspace) required to run the kernel.

```
1  vector<shared_ptr<IWorkData>> inData(2);
2  vector<shared_ptr<IWorkData>> outData(1);
3  // Gaussian noise kernel wrapper
4  outData[0] = inData[0] = shared_ptr<IWorkData> (new BufferData<cl_uchar4>());
5  inData[1] = shared_ptr<IWorkData> (new FinalData<int>(factor));
6  unique_ptr<IExecutable> gaussKernel (new KernelWrapper(gaussNoiseKernelFile, gaussNoiseFunction, inData, outData
       , 2)); // 2 is the number of elementary units computed per thread
7  // Solarise kernel wrapper
8  inData[1] = shared_ptr<IWorkData> (new FinalData<int>(threshold));
9  unique_ptr<IExecutable> solariseKernel (new KernelWrapper(solariseKernelFile, solariseFunction, inData, outData)
       );
10 // Mirror kernel wrapper
11 ...
12 // 3-stage pipeline
13 unique_ptr<IExecutable> pipeline (new Pipeline(gaussKernel, solariseKernel, mirrorKernel));
```

Listing 1. Image filter pipeline

Listing 1 presents a snippet of the construction of a pipeline with three stages. The setting up of the computation tree is a bottom-up process. It grows from the leafs, that take the form of `KernelWrapper` objects (line 8 and 11), which enclose a kernel's logic and domain in a single computational unit. The specification of the kernels' interface is expressed via the parameters of the associated wrapper. In this particular case, all kernels receive and output a single buffer (representing an image) and a final scalar value.

3 Cooperative Multi-CPU/Multi-GPU Execution

The main contribution of this paper is on the execution of Marrow CTs in hybrid multi-CPU/multi-GPU environments. To accomplish such enterprise we must address three key challenges: 1) how to efficiently decompose a CT among the multiple CPU and GPU devices; 2) how to efficiently distribute the work load among the available hardware resources, and; 3) how to adapt this distribution to different input data-sets and to the CPUs' load fluctuations. Due to space restrictions we will focus mainly on challenges 2 and 3, addressing the first lightly. For more details we refer the reader to a companion technical report [15].

Computational Tree Decomposition: To address this challenge we apply the same locality-aware approach that we devised for multi-GPU computing. We leverage the work-group based organization of OpenCL executions, namely the fact that work-groups (groups of threads) execute asynchronously and independently over data. Consequently, we opt to decompose the computation's data-set into partitions that can be adjusted to the best possible work-group size

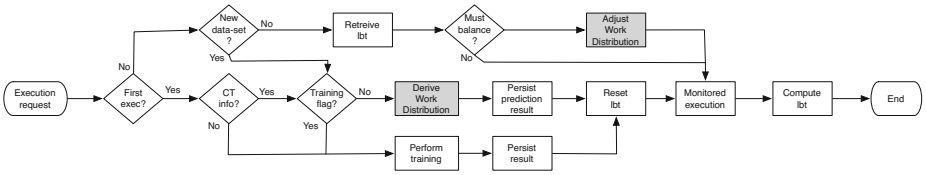

Fig. 1. The top-level work-distribution decision process

for each device. In that sense, we extend the scope of OpenCL's SPMD (Single Operation Multiple Data) based execution model to multiple devices, where each OpenCL work-group computes the CT over a partition of the input data-set.

The decomposition of the input data-set must be subjected to user-defined restrictions imposed in the CT's kernel specifications, but also be driven by the characteristics of the underlying hardware, such as the size of AMD wave-fronts or NVIDIA warps. Furthermore, in order to really leverage the aforementioned locality properties, the data communicated between two consecutive kernel executions must be partitioned in such a way that, each of such partitions may be communicated between the kernels simply by persisting in the device's memory across their execution. As a result, two kernel executions that communicate data-sets must expect an identical partitioning of such sets, with respect to their number and size(s), regardless the individual work-group size restrictions of either kernel. This approach induces a partitioning process with global vision of the computation, defining what we call a CT's *locality-aware domain decomposition*.

Work-Load Distribution: To efficiently execute computations composed by multiple (arbitrary) OpenCL kernels in hybrid multi-CPU/multi-GPU environments, we must engender a solution that is able to deliver good performances without requiring previous knowledge on the CT to execute.

In [9] we addressed this issue for heterogeneous multi-GPU environments. The workload is statically distributed among the devices, according to their relative performance. This static approach, although simple, delivers good performance results for GPU-accelerated executions, due to the specialized nature of the underlying execution model: one kernel execution at a time, with no preemption and no input/output operations. These premises are not valid for CPU executions. The execution time of a CPU computation is highly conditioned by the load of the processor, which is time-shared by multiple threads, and by hardware optimizations that cannot be fully controlled by the programmer, such as cache memory management. Therefore, to adequately balance the load of arbitrary computations between CPU and GPU of devices is still a challenge.

In this paper we are particularly interested in recurrent applications of CTs upon possibly different data-sets with different sizes. Therefore, we want to have a lightweight mechanism that is able to infer a suitable configuration for a CT's execution, given a particular parameterization. We address this challenge via profile-based self-adaptation. We still rely on static scheduling: the workload is distributed in advance between the available devices. However, this distribution

resorts to a CT-specific profile built from past runs. Furthermore, we refine this information for subsequent executions, so that the distribution may be adapted to alterations of the CPUs' load and/or of the input data-set's size. The first execution of a CT upon a particular workload is preceded by the inference of the best configuration (of the ones known to the system) to run such computation. From that point on, subsequent executions are monitored by a controlling process that identifies, and corrects, load unbalances, so that the ensuing executions may be better adapted to the system's current load.

The top-level work-flow of the work-distribution process is depicted in Fig. 1. The entire decision process builds on the availability of historic data about the target CT's execution. This knowledge-base (KB) stores information about the best configuration for a given input data-set. The primary source is a training process that is triggered whenever there is no information about the target CT, or the user explicitly demands it through the framework's configuration settings. If none of these conditions hold, the framework will try to determine a suitable workload distribution from the KB. Such enterprise is trivial if the necessary information is already available, otherwise it will have to be derived from the existing knowledge. Both the training and the derivation branches terminate with the persisting of the attained result, qualified with the process employed. As a result, the derivation process also contributes to populate the KB, serving as a cache for following executions.

Once a work-distribution configuration has been derived, the execution proceeds under the monitoring of the controller process. This process simply calculates the deviation (dev) between the execution times of each concurrent application of the CT over a partition of the original data-set, and computes a load balancing threshold $lbt(n) = isUnbalanced(dev) \times w + lbt(n-1) \times (1-w)$, for the given execution number n – $isUnbalanced(x)$ indicates if the deviation falls out of the allowed interval, and w denotes the weight assigned to the last execution relatively to the historic data. The use of a weighted historic data factor makes lbt less sensitive to sporadic unbalanced executions.

When a CT is applied recurrently, but over data-sets with different characteristics, the work-distribution process may configured to revert to (a) the training process or to (b) the derivation of a new workload distribution from the KB. Option (a) is tailored for applications that operate over the same type of data-sets for long periods of time. In such cases, it compensates to have the best possible configuration, even if it takes some time to obtain it. Option (b) is more directed to applications that operate over indiscriminate types of data-sets, and want to build on previous knowledge to adjust the framework to the particularities of each of such types.

Dimensions to consider in the training process: The Marrow framework is modular and extensible, in the sense that it allows for multiple OpenCL back-ends, each one specialized for a given type of device. These back-ends are exposed as *execution platforms* that abstract the OpenCL communication details to the remainder modules and encapsulate all the device-specific optimizations. They are also responsible for supplying an iterator over which of their configuration

parameters must be included in the training process. The current Marrow implementation features two execution platforms: CPU and GPU. The first supplies an iterator over the affinity fission configurations the device supports, a subset of: $(\bigcup_{i=1}^{4}\{L_i\_CACHE\})\cup\{NUMA\}\cup\{NO\_FISSION\}$, while the second offers an iterator over the number of overlapped executions to be performed in the GPU.

The training process: The process searches for the best parameterization of the existing execution platforms for the computation at hand. The algorithm performs an uniform search over the search space, spanning all possible combinations. This strategy is viable for the dimension of our current space. Nonetheless, in the future, scalability requirements may lead to the use of other techniques.

The algorithm requires a stoppage criterion, in the form of a `precision` value, a quality factor, in the form of the number of executions to be performed in each possible configuration, and the `task` to execute. It also assumes the existence of the CPU and GPU execution platforms, as well as of the `scheduler` module responsible for performing the static distribution of the work among the available devices. Given these premises, for each CPU_fission/GPU_overlap configuration the algorithm behaves as follows: 1) reconfigure the existing execution platforms; 2) activate the `scheduler`'s profiling mode, providing it the precision value and the quality factor; 3) perform the loop to obtain the best workload distribution for the current configuration - the *scheduler* internally determines the next configuration to test and computes the average execution time for the `nTrainingExecutions`; 4) store the work distribution that yielded the best performance and the associated execution time; 5) select the best performing configuration and reconfigure the execution platforms accordingly, and; 6) decompose the input data-set in conformance to the current parallelism level.

Concerning the CPU/GPU work-distribution internally determined by the `scheduler`, we have formulated two strategies: **50/50 split** evens, as much as possible, the time that each device type (CPU or GPU) takes to carry out the computation. For that purpose, it continuously transfers increasingly smaller parts of the load from the worst to the better performing device type. The relative execution times are expected to converge after a small number of iterations, being the algorithm bound to the `precision` value passed to the scheduler. **CPU assisted GPU execution** lessens the CPUs' role to a mere assistant of the GPUs, contributing only to lighten the latter's load. To that end, the strategy incrementally assigns work to the CPU so that the CT's overall execution time is minimized, independently of the devices' relative execution times.

Configuration Derivation: The configuration derivation process is twofold, represented by the two darker boxes in Fig. 1. Box "*Derive work distribution*" represents the derivation required to kickoff the execution of a CT, whenever the training flag in not active. For that purpose, we infer, by the means of interpolation, a workload distribution for the given input data-set from the information currently stored in the KB. The function to interpolate may be multidimensional, bound to the CT's number of input arguments. Our current approach employs the *nearest-neighbor* method sustained by the Euclidean distance over a multidimensional space, with as many dimensions as number of input arguments. The

framework allows for the simple integration of new interpolation methods, and in the future other approaches may be implemented.

The second box "*Adjust work distribution*" is triggered whenever the value of $lbt \approx 1$. In order for the load balancing process to be as little intrusive as possible to the application's global execution, we employ a two-level approach, where the first level is considerable lighter, computationally, than the second. This **first** level tries to re-balance the load by simply transferring a percentage of the workload from the slowest performing device type to the other. The work transference is iteratively applied in the following executions of the CT, until the execution time of the parallel executions converge below the $maxDev$ upper bound. The **second** level reverts to a partial execution of the training process that only considers the current fission/overlap configuration.

4 Experimental Results

The purpose of this study is to quantify the performance gains that the conjoint use of the CPU and GPU may deliver relatively to GPU-only executions, and to assess the efficiency of the proposed work distribution and balancing strategies.

For the experiments, we resorted to five benchmarks that make use of the different skeletons available in Marrow. The first two make use of the *Pipeline* skeleton. **Filter Pipeline** composes three image filters: Gaussian Noise, Solarize and Mirror. All of them may be independently applied to distinct lines of the image to be processed. Accordingly, the line is the partition's elementary unit. **FFT** is a set of Fast-Fourier Transformations (FFTs) where FFT is pipelined with its inversion. The decomposition elementary unit is the size of each FFT which is 512 kBytes. Ergo, each device is assigned with a set of such FFTs. The third benchmark is the iterative **N-Body** simulation supported by the *Loop* skeleton. The kernel implements the direct-sum algorithm that, for each single body, computes its interaction with all the remainder. Therefore, there is a dependency on the whole data-set, requiring replication to all the data-set to all devices. The distribution is hence performed at body level, entailing a synchronization point in-between each iteration. The final two benchmarks are simple *Map* applications. The BLAS **Saxpy** routine computes a single-precision multiplication of a constant with a vector added to another vector. The computation is embarrassingly parallel and does not require any partitioning restrictions. **Segmentation** performs a transformation over a gray-scale three dimensional image. Although there is no algorithmic dependencies between pixel elements, the elementary size is set to the size of the first two dimensions so the partitioning is performed only over the last one.

All experiments were conducted on a system featuring one hyper-threaded six-core Intel(R) Core(TM) i7-3930K CPU @ 3.20GHz, with 6 L1 and L2 caches (one per core) and a single L3 cache, shared by all cores; two AMD HD 7950 GPU devices attached to two dedicated PCIe x16 lanes; and 32 GBytes of RAM.

For each benchmark we have established three parameterization classes and two baselines, that report the time for the GPU-only accelerated execution of the

Table 1. Benchmark characterization

Benchmark	Input type	Input argument	1 GPU								2 GPUs							
			Baseline	50/50 Training			CPU assisted GPU Training				Baseline	50/50 Training			CPU assisted GPU Training			
			execution time	Configuration (fission/overlap)	Level of parallelism	Distribution (GPU/CPU)	Configuration (fission/overlap)	Level of parallelism	Distribution (GPU/CPU)		execution time	Configuration (fission/overlap)	Level of parallelism	Distribution (GPU/CPU)	Configuration (fission/overlap)	Level of parallelism	Distribution (GPU/CPU)	
Filter pipeline	Image size (pixels)	1024x1024	1.97	L2/3	9	91.8/8.2	L1/3	9	92.5/7.5		1.12	L3/2	5	94.0/5.4	L1/3	12	98.8/1.2	
		2048x2048	5.10	L3/4	5	92.9/7.1	none/4	5	93.8/6.3		3.84	L3/4	9	96.1/3.9	none/3	7	98.8/1.2	
		4096x4096	16.80	none/4	5	93.8/6.3	none/4	5	93.8/6.3		11.76	none/4	9	96.9/3.1	none/4	9	97.5/2.5	
FFT	Size of data-set	128MB	35.28	L2/3	9	32.8/67.2	L2/4	10	37.5/62.5		23.76	L1/4	14	59.8/40.2	L2/3	12	52.5/47.5	
		256MB	67.83	L2/4	10	31.3/68.7	L1/4	10	30.0/70.0		43.12	L1/4	14	58.6/41.4	L2/4	14	55.0/45.0	
		512MB	88.93	L1/3	9	37.1/62.9	L2/1	7	15.0/85.0		77.21	L1/4	14	56.3/43.8	L1/4	14	57.5/42.5	
NBody	Number of bodies	16384	37.17	-	-	-	L1/1	7	95.0/5.0		29.87	-	-	-	L3/1	3	98.8/1.2	
		32768	101.56	-	-	-	L2/1	7	97.5/2.5		69.63	-	-	-	L2/1	8	98.8/1.2	
		65536	356.85	-	-	-	L2/1	7	98.8/1.2		200.76	-	-	-	L2/1	8	98.8/1.2	
Saxpy	Number of elements	1×10^8	2.56	L1/2	8	41.4/58.6	L2/2	8	37.5/62.5		1.59	none/2	5	75.0/25.0	L1/2	10	67.5/32.5	
		10×10^8	14.91	L1/2	8	45.3/54.7	none/4	5	67.5/32.5		10.97	L3/4	9	87.5/12.5	none/4	9	88.8/11.2	
		50×10^8	72.86	L1/3	9	43.8/56.3	L1/3	9	47.5/52.5		46.84	L3/4	9	85.2/14.8	L1/4	14	77.5/22.5	
Segmentation	Number of elements	1MB	0.79	none/2	3	59.9/40.1	none/1	2	55.0/45.0		0.72	none/1	3	69.5/30.5	none/2	5	85.6/14.4	
		8MB	2.88	none/4	5	81.3/18.7	L3/4	5	78.8/21.2		1.87	none/3	7	88.3/11.7	none/2	5	88.8/11.2	
		60MB	16.70	none/4	5	82.6/17.4	L1/4	10	78.8/21.2		10.75	none/4	9	93.0/7.0	L1/4	14	88.8/11.2	

benchmarks using just one or the two available GPUs. Table 1 presents, for each of the determined parameterization classes, the baseline execution times and the results of applying the two training strategies to both the single and dual GPU setups. We do not present the result of the 50/50 split training for the NBody benchmark because the results are not usable. From this table we may observe that the best fission/overlap configuration depends on several factors: the actual computation, the input data-set's size, the number of devices, and the training strategy. Nonetheless, there is an aspect that spans most of the results: the correlation between the data-set's size and the level of fission and overlap. The advantages of the overlap tendentiously increase with the size of the data-set, leveraging the scalability lend by the two PCIe buses. In turn, the advantages of fission seem to decrease, implying that the fission support in the AMD OpenCL implementation (version 1348.5) is particularly beneficial when a data-set's partition fits the fission cache level.

Speedup Results: Figures 2 and 3 present the speedups obtained by the CPU+GPU ensemble when compared to the GPU-only baselines. The results show that the use of the hybrid infrastructure is beneficial in almost every conducted experiments - the NBody benchmark is the exception. The impact is, naturally, more visible in the single GPU configuration, where the gains range from 1.3 to 3, whilst in the 2 GPU configuration they range from 1.4 to 2.6. The speedups are particularly noticeable in the communication-bound computations. Paradigmatic examples are the smaller parameterization classes of Saxpy and Segmentation, where the CPU boosts the overall performance more than twice for both the 1 and 2 GPUs configurations. In both these benchmarks, as the data-set's size increases so does the computational weight, mitigating the benefits of the CPU. This behavior may also be observed in the FFT benchmark. The FFT kernels are computationally heavy but also operate upon large data-sets: 128 to 512 MBytes. It is the ever-present trade-off between the overhead of data-transfers and the computational complexity of the computation.

Filter Pipeline is more computation bound. Three different computations are applied over a single image transferred to the GPU. Nonetheless, the CPU's utility is still more visible with smaller images, where less parallelism is required. In NBody the advantages of using the CPU are minimum due to the large amount of work assigned to the GPU. Loop's execution model is more complex than the remainder skeletons, using the GPU for executing the loop's body,

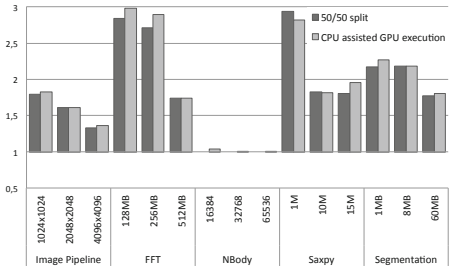

Fig. 2. Speedup: CPU+1GPU vs 1GPU

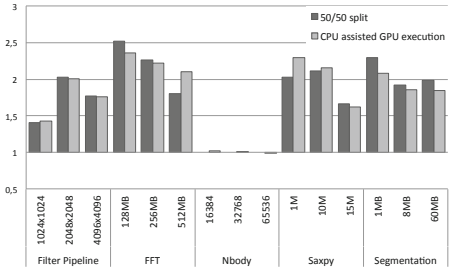

Fig. 3. Speedup: CPU+2GPUs vs 2GPUs

Table 2. Filter Pipeline: training's results versus interpolation from past executions

Image id	Image size	Training result					Derivation	Balancing		Resulting distribution		Exec.	Relative
		Fission	Overlap	GPU (%)	CPU (%)	Exec. time	Nearest neighbour	Level 1	Level 2	GPU (%)	CPU (%)	time	perf.
Image 1	1024x1024	L3	3	90.8	9.2	1.10							
Image 2	512x512	L3	2	87.5	12.5	0.54	Image 1	6	1	81.0	19.0	0.64	84%
Image 3	1024x2048	L1	4	91.5	8.5	1.74	Image 1	2	0	90.7	9.3	1.87	93%
Image 4	2048x512	L2	3	89.8	10.2	1.06	Image 1	4	0	90.6	9.4	1.07	100%
Image 5	2048x2048	none	4	92.9	7.1	3.17	Image 3	1	0	90.6	9.4	3.48	91%
Image 6	4096x4096	L3	4	93.8	6.3	12.59	Image 5	0	0	91.8	8.2	13.41	94%

and the CPU for iteration synchronization and evaluating the loop's condition. Our current profiling process is not able to differentiate the work performed on each type of processor, an issue that opens the door to more fine-grained profiling approaches. Nonetheless, in this particular benchmark, delegating the execution almost entirely to the GPUs is the best option. We artificially forced other distributions with worse results.

Regarding the two training strategies, the 50/50 split seems to be more reliable, as it tries to balance the load among the device types, while also taking into consideration the overall execution time. However, the results are quite close and even converge when the precision is very high.

Efficiency of the Work Distribution and Load Balancing Strategies: We selected the Filter Image benchmark to evaluate how does our configuration derivation behaves in the presence of different input data-sets. We begin with an empty KB, and populate it as the benchmark is successively applied to images *Image 1* to *Image 6*. Thus, when *Image i* is executed the KB contains knowledge about images *Image 1* to *Image* $i-1$.

To establish individual baselines, we independently ran the training process for each image. The left-side of Table 2 presents such results. Then, beginning with the empty KB, we resorted to our nearest neighbor interpolation to successively apply the benchmark to images *1* to *6*. The benchmark was configured to run 500 times, so that we could count how many times the load balancing process was triggered, and its level — the framework was configured to considered balanced all runs whose partial execution times say within 88% of each other. We also measured the final workload distribution, and the performance loss relatively to the one obtained from the training process. All these values are

presented on the right-side of Table 2. We chose this benchmark because, as may be observed in the table, the best fission/overlap configuration is very dependent on the data-set. Thus, we wanted to asses if the simple balancing of load within the configuration derived from the interpolation was enough to obtain good results. What may be initially concluded is that the second balancing level is rarely used. Moreover, as should be expected, the derivation process is highly dependent of the affinity of the current data-set in regard to the ones stored in the KB. Naturally, the probability of finding a good candidate increases in time, as the KB grows. Nonetheless, as the number of images rises we are able to deliver performances within less than 10% of the performance obtainable from the training process.

For our last experiment, we selected the 50 million parameterization class of the Saxpy benchmark to study how the framework adapts to load fluctuations in the CPU. From Table 1 we have that the initial workload distribution is GPU $\leftarrow$ 49.61% and CPU $\leftarrow$ 50.39%. To introduce the load fluctuation on the CPU we implemented an application that spawns as many software threads as available hardware threads, each running a computationally heavy algebraic problem. Fig. 4 depicts the framework's adaptation to the sudden increase in the CPU's load. The process begins with the successive application of the level 1 load balancing strategy (L1 in the chart). However, the solution is not sufficiently aggressive to quickly shift the work to the GPU and, thus, the second level (L2 in the chart) is forced to kick in. The distribution stabilizes at GPU $\leftarrow$ 58.18% and CPU $\leftarrow$ 40.82%. As we terminate the load inducing application, the system becomes unbalanced once again. The balancing process restarts and, upon a first level 2 execution, the system is mostly balanced. Yet, in order for it to stay within the pre-determined 88% mark for balanced executions, a second burst of load balancing operations is required. The final distribution is GPU $\leftarrow$ 49.22% and CPU $\leftarrow$ 50.78%.

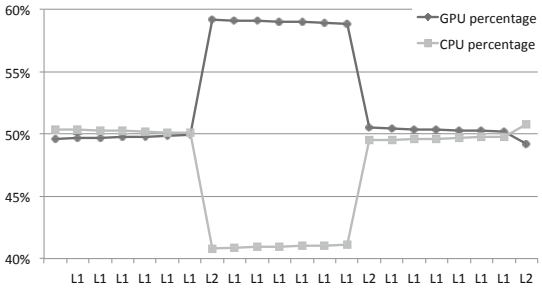

Fig. 4. Saxpy subjected to load fluctuations

5 Related Work

There are several skeleton/template frameworks that address GPU computing. However, their focus in solely on data-parallel skeletons and none of them support skeleton nesting, thus no compound behaviors can be offloaded to the GPU. Heterogeneity support in such frameworks comes in two flavors. The first approach is to include, at language level, constructions to determine where the computation

must take place, such is the case of Muesli [5]. The second approach obliges the programmer to direct the compilation at either CPUs or GPUs. This category includes SkePU [1] and Thrust [4], which feature multiple mutual exclusive backends, for either GPGPU or shared-memory parallel programming. SkePU can be combined with the StarPU scheduler to provide computation locality abstraction. Although StarPU has the capability to schedule tasks on both multi-core CPUs and GPUs simultaneously, when a task is submitted to SkePU, only the best performing device for the given input size is selected.

In [12], the authors propose a run-time infrastructure for executing MapReduce computations on hybrid CPU/GPU systems. They propose different methods for work distribution, from the scheduling of map tasks across the devices, to the placing of the map stage on a device and of the reduction stage on the other, much like what we allow in our MapReduce skeleton. The dynamic work distribution tries to adjust the task block sizes for subsequent executions of tasks in the same application.

Dandelion [10] offers a LINQ-based programming model for heterogeneous systems, extended with other constructions, such as loops. A specialized compiler generates code for both CPU and GPU from the source code, while the run-time system transparently offloads computations to the GPU, when the workload so justifies. Good performance requires the programmer to convey some GPU-related information, and no CPU+GPU support is provided.

The works presented in [11] and [16] address the paralellization of nested loops in heterogeneous environments. They both resort to polyhedral optimization techniques which restrict their scope to loops with affine expressions. The first selects the best device to run the computation but never resorts to CPU+GPU wide computations. The second generates OpenCL code for CPU, GPU and CPU+GPU environments, selecting the best hardware depending on the data flow requirements. Very few results are given for CPU+GPU environments.

Finally, in [14], the presented solution allows for work-partitioning among devices based on a performance model updated at run-time. This solution, however, requires the existence of multiple versions of the program, one for each different device present in the system.

6 Concluding Remarks

In this paper we have presented a systematic approach for the cooperative multi-CPU/multi-GPU execution of Marrow computations. We have proposed a locality-aware domain decomposition of Marrow skeleton trees that promotes data-locality but, simultaneously, allows for the multiple OpenCL kernels to be executed under different work-group configurations, as long as communication compatibility is assured. We have also proposed profile-based workload distribution and load balancing strategies, that build from past runs of a given computational tree to derive suitable configurations for subsequent executions. The experimental results show that our approach brings speedups up to 300% over GPU-only executions. Moreover, they also provide some insight on the framework's ability to adapt to data-sets of different sizes and to fluctuations on the

CPU's load. Regarding future work, the focus is on the improvement of our configuration derivation. We want to refine our profile-based approach with more sophisticated interpolation techniques, and combine these with static analysis of the kernel's code, in order to reduce the weight of the training stage.

References

1. Dastgeer, U., Li, L., Kessler, C.: Adaptive implementation selection in the skePU skeleton programming library. In: Wu, C., Cohen, A. (eds.) APPT 2013. LNCS, vol. 8299, pp. 170–183. Springer, Heidelberg (2013)
2. Steuwer, M., Gorlatch, S.: SkelCL: Enhancing openCL for high-level programming of multi-GPU systems. In: Malyshkin, V. (ed.) PaCT 2013. LNCS, vol. 7979, pp. 258–272. Springer, Heidelberg (2013)
3. AMD Corporation: Bolt C++ Template Library, `http://developer.amd.com/tools/heterogeneous-computing/`
4. Hoberock, J., Bell, N.: Thrust: A parallel template library, `http://thrust.github.io/`
5. Ernsting, S., Kuchen, H.: Algorithmic skeletons for multi-core, multi-GPU systems and clusters. Int. J. High Perform. Comput. Netw. 7(2), 129–138 (2012)
6. Huynh, H.P., et al.: Scalable framework for mapping streaming applications onto multi-GPU systems. In: PPoPP 2012, pp. 1–10. ACM (2012)
7. Dubach, C., others: Compiling a high-level language for GPUs (via language support for architectures and compilers). In: PLDI 2012, pp. 1–12. ACM (2012)
8. Marques, R., Paulino, H., Alexandre, F., Medeiros, P.D.: Algorithmic Skeleton Framework for the Orchestration of GPU Computations. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 874–885. Springer, Heidelberg (2013)
9. Alexandre, F., Marques, R., Paulino, H.: On the support of task-parallel algorithmic skeletons for multi-GPU computing. In: SAC 2014, pp. 880–885. ACM (2014)
10. Rossbach, C.J., Yu, Y., Currey, J., Martin, J.-P., Fetterly, D.: Dandelion: a compiler and runtime for heterogeneous systems. In: SOSP 2013, pp. 49–68. ACM (2013)
11. Dollinger, J.F., Loechner, V.: Adaptive runtime selection for GPU. In: ICPP 2013, pp. 70–79. IEEE Computer Society Press (2013)
12. Chen, L., Huo, X., Agrawal, G.: Accelerating MapReduce on a coupled CPU-GPU architecture. In: SC 2012, pp. 25:1–25:11. IEEE Computer Society Press (2012)
13. Lee, J., et al.: Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems. In: PaCT 2013, pp. 245–255. IEEE (2013)
14. Colaço, J., Matoga, A., Ilic, A., Roma, N., Tomás, P., Chaves, R.: Transparent application acceleration by intelligent scheduling of shared library calls on heterogeneous systems. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2013, Part I. LNCS, vol. 8384, pp. 693–703. Springer, Heidelberg (2014)
15. Soldado, F., Alexandre, F., Paulino, H.: Transparent execution of compound OpenCL computations in multi-CPU/multi-GPU environments. Technical report, CITI/DI, Universidade NOVA de Lisboa (2014)
16. Dathathri, R., et al.: Generating efficient data movement code for heterogeneous architectures with distributed-memory. In: PaCT 2013, pp. 375–386. IEEE (2013)