A (ir)regularity-aware task scheduler for heterogeneous platforms

Artur Mariano[1], Ricardo Alves[1], Joao Barbosa[1,2], Luis Paulo Santos[1] and Alberto Proenca[1]

[1]Department of Informatics, University of Minho, Braga, Portugal
[2]Computer Sciences Department, University of Texas at Austin, Texas, USA

{amariano,ricardoa,jbarbosa,psantos,aproenca}@di.uminho.pt

Abstract. This paper addresses the design, implementation and validation of an effective scheduling scheme for both regular and irregular applications on heterogeneous platforms. The scheduler uses an empirical performance model to dynamically schedule the workload, organized into a given number of chunks, and follows the Heterogeneous Earliest Finish Time (HEFT) scheduling algorithm, which ranks the tasks based on both their computation and communication costs. The evaluation of the proposed approach is based on three case studies – the SAXPY, the FFT and the Barnes-Hut algorithms – two regular and one irregular application. The scheduler was evaluated on a heterogeneous platform with one quad-core CPU-chip accelerated by one or two GPU devices, embedded in the GAMA framework. The evaluation runs measured the effectiveness, the efficiency and the scalability of the proposed method. Results show that the proposed model was effective in addressing both regular and irregular applications, on heterogeneous platforms, while achieving ideal ($\geq 100\%$) levels of efficiency in the irregular Barnes-Hut algorithm.

Keywords

CPU+GPU, Scheduling, Irregularity, Heterogeneous, Performance-portability

1 Introduction

Current high-end computational platforms explore a diversity of multi/many core processors and co-processors that together have the potential to deliver a huge computational power. These platforms, commonly referred to as heterogeneous platforms (HetPlats), typically include multi-core CPU-chips and many-core, highly programmable, Graphical Processing Units (GPUs) and, to a lesser extent, specialized devices such as DSPs and/or FPGAs. Leveraging this latent computational power is mandatory, if high performance and energy efficient computing is to be achieved.

Efficiently exploiting HetPlats is, however, a very challenging task. Different devices exhibit diverse programming and computing models together with various computing capabilities, diverse development languages, tools and environments, and disjoint memory address spaces. This diversity results in increased programming and platform tuning complexity, which strongly impacts on programming productivity. The development of efficient applications for HetPlats requires time-consuming code optimization and often hinders performance portability if the configuration of the target heterogeneous system is modified.

To foster wider adoption of HetPlats, researchers have been proposing frameworks that aim to hide from the programmer some of the above cited challenges, thus increasing both programming productivity and performance portability. Even though applications developed using such frameworks will seldom achieve the performance levels of highly optimized codes, the tradeoff is that performance is still within acceptable bounds, while programming effort is strongly reduced and migration onto differently configuration systems is facilitated, if not automatic. These frameworks, including StarPU [4], Harmony [11], MDR [22] and GAMA [6], to name a few, address issues such as the management of the heterogeneous platform (e.g., launching, synchroninsing and terminating computing kernels), scheduling and data management.

Scheduling, understood as mapping the workload tasks onto devices and defining their order of execution [25], has been shown to be strictly related to the observed performance levels [15, 1]. A good schedule – whose goal

in the case of this paper is to minimize the application execution time – should take into account the different devices computing paradigms suitability to the tasks at hand, the devices relative computing capabilities and the costs of data movement within the system. Therefore, schedulers include, either explicitly or implicitly, a performance model to predict how a task will perform on a device, given the current system state, thus enhancing decision making. The accuracy of such predictions depends on many factors, such as the workload and computing system complexity, and the correctness of the available information.

The above cited frameworks for HetPlats include scheduling mechanisms devised for regular workloads. However, the scheduling problem is further aggravated if irregular workloads are considered. Whereas data parallel regular workloads exhibit similar computing requirements and memory access patterns per data element, irregular workloads present widely varying computing requirements and memory accesses across data elements. These variations strongly impact on the accuracy of the scheduler performance model and might result in poor overall performance if not explicitly taken into account on the scheduling policy. Although harder to schedule than regular applications, irregular applications are a substantial part of both scientific and technological applications [23]. In particular and due to the large Theoretical Peak Performance (TPP) of some accelerators, some irregular applications have been ported to GPUs. Good performance levels are usually achieved by re-designing the algorithm at the cost of very time consuming code tuning, in function of the underlying architecture [7, 21, 24].

This paper addresses the design and evaluation of an efficient scheduling policy for irregular workloads on HetPlats. Our goal is to demonstrate that such efficient scheduling is indeed possible for this most difficult irregular case, while maintaining the previously achieved successful results for regular workloads. The GAMA framework [6] was selected to validate the proposed scheduler policy, and compared with a static, manually parametrised, scheduler. Three different case studies are examined, including two regular algorithms (SAXPY and FFT) and one irregular algorithm (the Barnes-Hut n-Body solver).

This paper contributions include:

- the design of a scheduler and associated performance model aimed for both regular and irregular workloads on HetPlats;

- the validation of a scheduling policy, built upon the proposed performance model, that efficiently handles regular and irregular workloads.

The reminder of this paper is organized as follows. Section II presents the related work, including frameworks and scheduling strategies addressing HetPlats. Section III presents the proposed dynamic scheduler. Section IV introduces the evaluation environment and methodology, whereas Section V presents the validation of the proposed method. Section VI concludes the paper.

2 Related work

The development of applications to run on HetPlats is too time consuming, due to different programming models, workload distribution and data management. Several frameworks addressing HetPlats were released since 2008, aiming to increase programming productivity while simultaneously maintaining performance within acceptable bounds. These frameworks, which include StarPU, Harmony, MDR and GAMA, provide both a programming and an execution model, automatically orchestrating workloads and data movements in regular applications. The GAMA framework, however, also addresses irregular applications on HetPlats.

StarPU is a runtime system that provides a high-level, unified execution model for task scheduling on heterogeneous platforms [4]. StarPU's scheduler aims to minimize the cost of transfers between processing units and use the data transfer cost prediction to improve the task scheduler decisions [2]. The scheduling decisions are supported by an empirical, history-based performance model, which predicts the execution time of a task on each system device. The performance model is calibrated, either at runtime using linear regression models, or offline for non-linear models [3]. The system keeps track of each replicated data on the platform, determining whether accessing a particular data block requires a transfer or not.

Harmony aims to simplify parallelism management, dynamic scheduling and on-line monitoring-driven performance optimization, for heterogeneous many core systems [11]. Harmony's dynamic scheduler is based on mapping kernels to devices and variables to memory spaces as the program is being executed [10]. The scheduling operation lasts while the window of kernels fetched from the program, continuously updated, is not empty. The scheduler includes a performance model to predict the execution of kernels based on the used variables, the information about its PTX assembly code, and the history of previous execution of the same or similar kernels.

The Model Driven Framework (MDR) design is based on several performance models with impact on run-time decisions, including mapping and scheduling tasks to computing units (CU)s and copying data between memory spaces [22]. It thus models task execution, while orchestrating the data-movement within the platform. The workloads are represented as parallel-operator directed acyclic graphs (PO-DAGs). The scheduling decisions are based on four identified criteria: suitability, locality, availability and criticality (SLAC). Empirical performance models are used to estimate the execution time of each kernel, whereas analytical models are used to estimate communication costs.

The GPU and Multi-core Aware (GAMA) framework under development aims to bridge the programming model between the CPU-chip and accelerators such as GPUs [6]. CPU-cores and GPU Stream Multiprocessor (SM) units are referred to as CUs. Identical CUs that share a common level of memory are grouped together into devices following the hierarchical memory system in HetPlats. The grouping into devices enables the use of the platform specific programming and execution model allowing cooperation and synchronization among CUs. The system employs a global address space to tackle the distributed memory model of any particular host platform. It also employs a relaxed consistency memory model to favour performance.

An application in GAMA is a collection of data-parallel jobs submitted to a run-time system, for scheduling among the available computational resources, where the dependencies among jobs are solved with explicit synchronization barriers. The goal of the run-time scheduler is to reduce the time to solution of any given application. Since in most cases the granularity is too coarse to enable a balanced scheduling policy, the run-time system recursively employs a user-defined dice operation to adjust it to the device specification, resulting in the creation of execution tasks of the same job. The execution of these tasks must be assumed as out-of-order, unless a synchronization barrier is explicitly issued.

Other frameworks to address HetPlats were proposed, including Qilin [20], Merge [19], Anthill [12] and HyperFlow [9], but none of these addresses irregular applications or presents relevant scheduling differences from the previous presented frameworks. Anthill, for instance, employs both First Come First Served (FCFS) and Dynamic Weighted Round Robin (DWRR) scheduling policies, which are not particularly designed to schedule irregular workloads.

Other works have proposed scheduling models for HetPlats [17, 8]. These models are supported by empirical schedulers, based on historical data gathered in run-time. The closest study to the work in this paper has focused on irregular reduction applications arising from unstructured meshes [16]. The authors proposed a multi-level partitioning framework with a *work-stealing* based scheduler, which considers granularity issues when distributing the workload. However, the proposed scheme is mostly appropriate for irregular reductions – particularly relevant in the field of Computational Fluid Dynamics – and not to the most general case of irregular data parallel workloads.

3 A (ir)regularity-aware scheduler to HetPlats

This paper proposes a new scheduling strategy to address both regular and irregular applications in HetPlats. To test and validate the scheduler, the GAMA framework was used as a test-bed. Applications in the GAMA framework are organized as a set of jobs, each with a set of computation tasks and associated data domains, which form the workload.

Regular applications have been successfully scheduled by partitioning the workload according to either the computing capability of each CU or the performance model information, either under static or dynamic scheduling schemes. The efficient scheduling of irregular applications, on the other hand, necessarily requires dynamic scheduling schemes, capable of balancing the load on the system during the life-time of the application's execution [5, 18, 13].

The proposed scheduler is organized in two phases: (i) to build an initial empirical performance model and (ii) to dynamically schedule the workload, organized into a given number of chunks i.e., sets of tasks that belong to the same job. Phase (ii) aims to overcome the ineffectiveness of performance models to schedule irregular applications. Its assignment policy is inspired on the Heterogeneous Earliest Finish Time (HEFT) scheduling algorithm, which ranks the tasks based on both their computation and communication costs, assigning every unscheduled task t_i to the device d_j that minimizes the EFT value of the task t_i [26].

The scheduler uses a hierarchic queuing system where each CU has its own private Local Queue (LQ) and share a High Level Queue (HLQ) with all the other CUs. At the beginning of the application, its tasks are stored in the High Level Queue (HLQ). The scheduler dynamically assigns these tasks to each LQ at runtime, which worker threads later pop and execute.
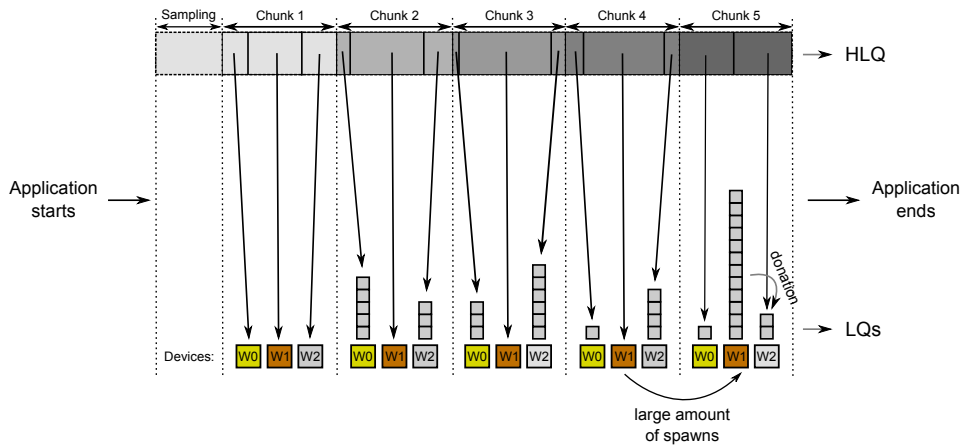
Figure 1. A scheduling behaviour during the application's lifetime. The figure shows a snapshot of the local queues before scheduling each chunk.

The empirical performance model is designed to estimate the execution time of a pair $(task(size), device)$, and is built and fed with a sampling process. This process equally divides a workload sample (10% by default) and assigns the resultant parts among the available workers. The execution times of these sampling tasks are stored on the performance model. The rest of the workload sample remains in the HLQ, to be scheduled afterwards. The estimated execution time for a task on a device is computed as the median of measured trials for that task on that particular device.

As shown in Figure 1, the workload is continuously assigned in chunks (5 in this case), after the sampling process. The scheduler assigns a chunk of tasks whenever it is signaled by a worker thread, in a demand-driven fashion. This happens when the number of tasks in one particular LQ is lower than a fixed parameter (4 by default). This scheme differs from the original HEFT, since for an application with N tasks, the proposed scheduler dynamically assigns $\frac{N}{x}$ tasks for each run, where x is the number of runs. A possible refinement for later work is to balance the workload during the execution of the last chunk, either with work-donation or work-stealing (shown in Figure 1), especially relevant to address dynamic task spawn, that is likely to become implemented in GAMA in the future.
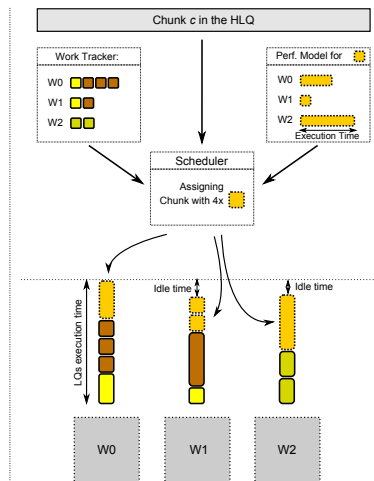

Figure 2. Illustration of the default assignment policy, for a given chunk c. Rectangles represent tasks, each color represent a different job (■, ■, ■ and ■ represent tasks from different jobs) and the width of each rectangle represents the estimated execution time of that task.

The assignment of each chunk of tasks follows a pre-configured assignment policy. In this policy, the number of tasks assigned to each device is defined according to the load of each device at each instant and the suitability of each device to the task under assignment, as estimated by the performance model. Figure 2 further details the default assignment policy, for a given case: the scheduler assigns a given chunk c, in this case of 4 tasks, among the available workers. It aims to minimize the execution time of the LQs and also to minimize each worker idle time, favouring the application time-to-solution (TTS).

The performance model is implemented as a dual-level Hash Table to favour efficient accesses, as shown in Figure 3. The first level Hash Table contains the jobs to which tasks belong, whereas the second level Hash Table contains the measured execution times for different sized tasks of the same job. With this dual level scheme, access to both data elements is expected to be done in constant time.
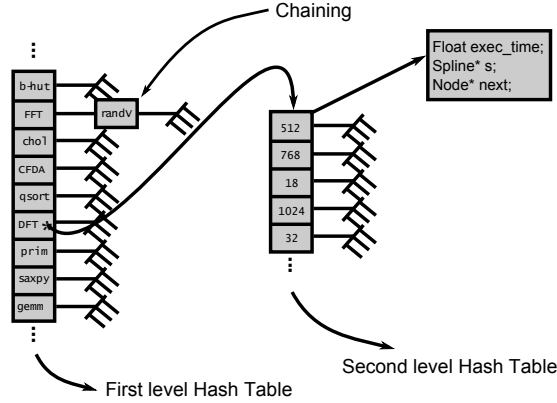

Figure 3. Dual-level Hash Table Performance Model.

The scheduler includes a work tracking facility, which keeps track of each task on the system. The estimated execution time of an LQ on each moment, referred to as TLQ, for an application with m jobs, is given by equation 1. $\lambda_{wt}(j, w)$ represents the number of tasks associated with job j on the local queue of the worker w, according to the Work Tracker module; $\epsilon_{pm}(j(s), w)$ represents the performance model estimation for the execution time of a task i, with data domain size s_i, of job j and to be executed on the worker w.

$$TLQ_w = \sum_{j=0}^{m-1} \sum_{i=0}^{\lambda_{wt}(j,w)-1} \epsilon_{pm}(j(s_i), w) \tag{1}$$

The number of tasks to assign to each worker at each moment, when scheduling a chunk of T tasks belonging to job j, is computed through a system of n linear equations and n unknowns, n being the number of workers. It is based on balancing and minimizing the execution time of all the LQs in the system. The equation system is given by:

$$\begin{cases} TLQ_0 + t_0 \times \epsilon_{pm}(j, 0) = TLQ_1 + t_1 \times \epsilon_{pm}(j, 1) \\ TLQ_1 + t_1 \times \epsilon_{pm}(j, 1) = TLQ_2 + t_2 \times \epsilon_{pm}(j, 2) \\ ... \\ TLQ_{n-1} + t_{n-1} \times \epsilon_{pm}(j, n\text{-}1) = TLQ_n + t_n \times \epsilon_{pm}(j, n) \\ t_0 + t_1 + t_2 + ... + t_n = T \end{cases} \tag{2}$$

where t_w represents the number of tasks to assign to the worker w. This system is always possible and determined. Its solution may include negative values, which are set to 0, and positive values are proportionally adjusted to the number of tasks to assign, T. In this context, negative values represent overloaded queues. This is corrected by further (chunk) assignments, which attempt to correct the system load imbalance, as an alternative to migrating (stealing/donation) tasks from those queues.

4 Evaluation environment and methodology

The scheduler evaluation runs were performed on a computational heterogeneous platform, whose devices, a CPU-chip and two GPU boards, are specified in Table 1. The system runs Linux Ubuntu 11.10 (Kernel 3.0) with CUDA 5.0 (beta release). The test code was compiled with GCC 4.6 and NVCC 5.0, with -O2 optimizations in both compilers. All trials were executed and measured 25 times, filtered by the k-best algorithm, for $k = 3$.

Device type	CPU-chip	GPU board
Number	1	2
Manufacturer	Intel	NVIDIA
Code	Core i7-960	GTX 580
Code Name	Bloomfield	GF110
Year	2009	2010
Architecture	Nehalem	Fermi
#CUs	4 cores	16 MT-SIMD
CU frequency	3.20 GHz	772 MHz
SMT	2x	48x
Vector Support	SSE 4.2	-
Compute Capability	-	2.0
L1 Cache	32KB iC + 32KB dC	64KB per CU
L2 Cache	256KB per core	768KB, shared
L3 Cache	8MB, shared	-
Single Precision TPP	102 GFLOPS	1581 GFLOPS
TDP	130 Watt	244 Watt
Main Memory	8GB	1.5GB

Table 1. Target hardware platform.

The scheduler was submitted to performance trials with three different case studies: two regular algorithms, the SAXPY and the FFT, and one irregular algorithm, the Barnes-Hut n-Body solver. The SAXPY algorithm has low computation requirements per memory access, which suggests that may be memory bound. The FFT implementation, based on the Cooley-Tukey algorithm, is particularly suited for the CPU and is a compute-bounded case. The Barnes-Hut algorithm is irregular since not only (i) the computation involved when calculating the net force for a given body is different for different bodies but also (ii) data accesses for each body do not follow any predictable pattern.

The evaluation runs measured (i) the execution times of both the dynamic and the best static scheduling decisions, (ii) the CPU/GPU workload distribution, (iii) the efficiency η of the pair (framework,scheduler) and (iv) the scalability of accelerators.

The first set of evaluation runs measured the execution time of the case studies, both with the proposed scheduler and the static scheduler. The results of both schedulers were compared by presenting the execution time of the application when statically scheduling the workload among the quad-core CPU-chip and one of the GPUs, in all possible variations on multiples of 10%. This methodology was also the base of similar studies that evaluated the performance other scheduling mechanisms [20, 15]. Both schedulers were also compared with CPU and GPU libraries of the algorithms under study.

The second set of evaluation runs measured the workload distribution delivered by the proposed scheduler, among the CPU-chip and one GPU. These trials measure the relation of the GPU usage in function of the input data set size.

The third set of evaluation runs measured the efficiency η of the GAMA framework, when supported by the proposed dynamic scheduler, according to a well defined formula defined in [4], shown in equation 3. It expresses how well the framework takes advantage of the multiplicity of architectures on the platform, based on the computational power Ψ of the platform and each device individually.

$$\eta = \frac{\Psi_{HetPlat}}{\Psi_{D_0} + \Psi_{D_1} + ... + \Psi_{D_n}} \tag{3}$$

where $\Psi_{HetPlat}$ represents the computational power associated associated to the whole platform and Ψ_{D_i} represents the computational power associated to the device i, when the framework forces the whole workload to run exclusively on that device. With regard to a particular algorithm, the computational power of a device

i (or a platform p) is given by the ratio between its input size and the execution time that device D_i delivers, as shown in equation 4.

$$\Psi_{D_i} = \frac{input\ size}{execution\ time} \tag{4}$$

Finally, the scalability of the dynamic scheduler was assessed, when adding one or more GPU boards accelerators to a system with a single CPU-chip.
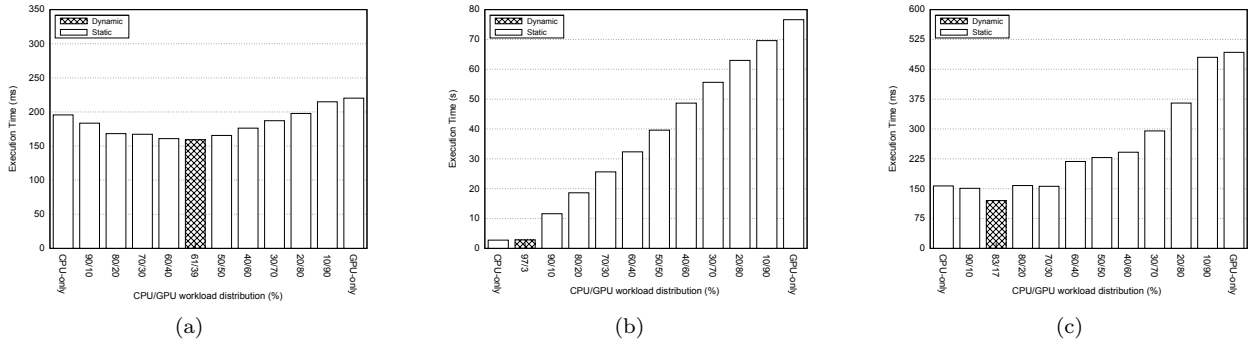

Figure 4. Execution time of several workload distributions for the SAXPY, FFT and Barnes-Hut algorithms in GAMA, with dynamic and static schedulers, on a CPU+GPU setup configuration. SAXPY was executed with an input-set of 2^{27} elements in each vector, FFT with 2^{25} double precision elements and the Barnes-Hut algorithm with a 2^{15} particles system.

5 Validation

Figure 4 shows the execution times of the three algorithms under study, in the GAMA framework, comparing several workload distributions for the static and the dynamic schedulers. The results are obtained using the quad-core CPU and a single GPU.

Figure 4(a) shows the results of the SAXPY algorithm where the best workload distribution given by the static scheduler lied between assigning 60%+40% and 50%+50% of the workload to the CPU+GPU. The dynamic scheduler lied in this band, delivering the best performance among the entire set of trials.

Figure 4(b) shows that scheduling the whole FFT workload on the CPU-chip was the most efficient static solution, since the devised FFT implementation is particularly suited for the CPU. The performance of the dynamic scheduler delivered slightly worst levels of performance, since small parts of the workload were scheduled to the GPU, due to the sampling process. These results do not match the vision that every CPU/GPU-only algorithm is necessarily severely affected when small parts of the workload are assigned to the less appropriate device [15].

Figure 4(c) shows the static and dynamic workload distributions for the net force calculation of the Barnes-Hut implementation in GAMA. Once again, the dynamic scheduler automatically reached the fastest workload distribution. The remaining jobs of the algorithm were calculated in OpenMP, and they were not considered in measurements.

These results show that the devised scheduler delivered the highest levels of performance when compared with all the possible static workload distributions in both the regular SAXPY and in the irregular Barnes-Hut algorithm. In particular, the dynamic scheduler was able to automatically find an efficient workload distribution, relieving the programmer from parameterize the static scheduler to achieve efficient workload distributions.. With respect to the FFT, based on a CPU-tailored Cooley-Tukey implementation, the small gap to the best performance result was due to assigning some workload to the GPU during the sampling process.

Memory transfers are another critical issue in dynamic scheduling in HetPlats, were accelerators may incur in substantial memory access latencies [14]. This is also relevant in GAMA, which currently allocates pinned

memory, accessed by accelerators through PCIexpress channels. Figure 5 shows the relation of both CPU's and GPU's usage and the input set size for the case studies.

These experimental measurements suggest that, except for the FFT, larger input data-sets lead to higher computational-communication ratios and/or higher overlapping of computation with memory transfers and higher GPU usage, as expected. These results do not apply to the FFT implementation, since the CPU-chip is the most adequate for the Cooley-Tukey algorithm.
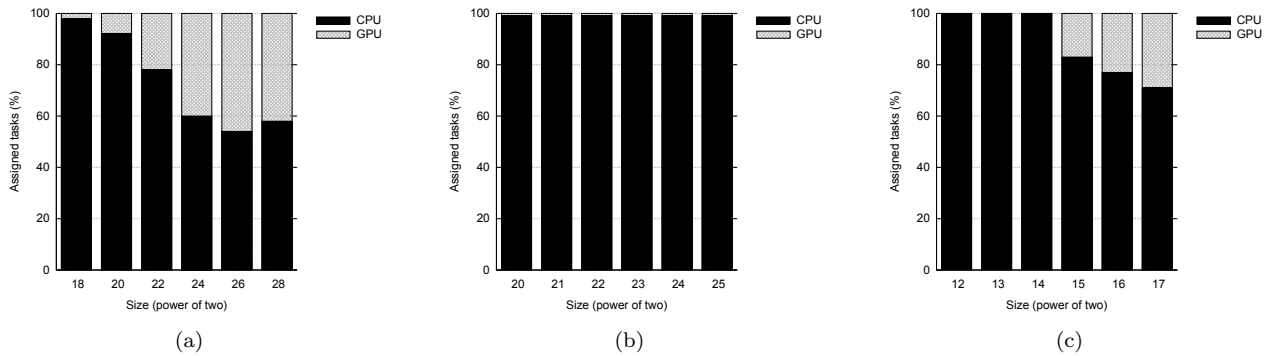

Figure 5. CPU and GPU workload distribution under dynamic scheduling, for SAXPY, FFT and Barnes-Hut. SAXPY experiments ranging from 2^{18} to 2^{28} elements in each vector. FFTs performed with 2^{20} to 2^{25} double precision elements and the Barnes-Hut algorithm with 2^{12} to 2^{17} particles systems.

The impact of the input data size on memory transfer efficiency is especially noticeable in the Barnes-Hut algorithm. As shown in Figure 5(c), the scheduler found no benefit in assigning workload to the GPU, to systems with 2^{14} or less particles.

Plots in Figures 6, 7, 8 capture alternative implementations: static and dynamic versions with GAMA versus CPU and GPU-only highly efficient versions of the presented case studies. For SAXPY, GAMA is compared with Lapack++ (CPU-only) and cuBLAS (GPU-only), as shown in Figure 6.
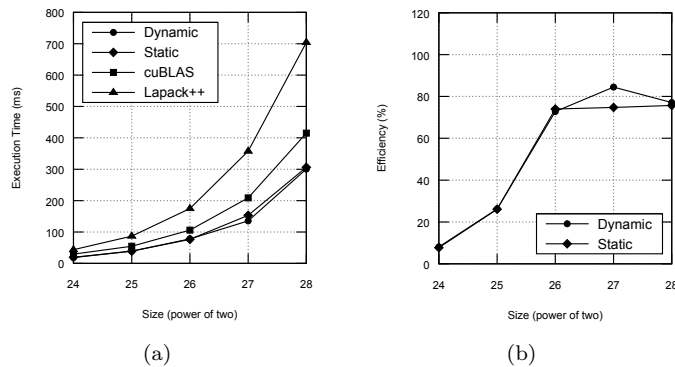

Figure 6. (a) Execution time in milliseconds for the SAXPY algorithm in GAMA, with dynamic and static schedulers, cuBLAS and Lapack++. (b) Efficiency η in percentage for GAMA, with dynamic and static schedulers.

The results shown in Figure 6(a) evidence that the proposed dynamic scheduler is equally or more efficient than the best static workload distribution, across a wide range of input data-sets. GAMA also overcame both cuBLAS and Lapack++, suggesting that the evaluated SAXPY implementation is able to efficiently leverage both architectures in the test-bed platform. The efficiency η, for both the dynamic and the best static workload distributions, is shown in Figure 6(b). The results show that these schedulers are similar with regard to efficiency, except for the SAXPY executing with 2^{27} elements in each vector, where the dynamic scheduler obtained better results.

For the FFT, GAMA is compared with both cuFFT (GPU) and the parallel FFTW (CPU) libraries, as Figure 7(a) shows.
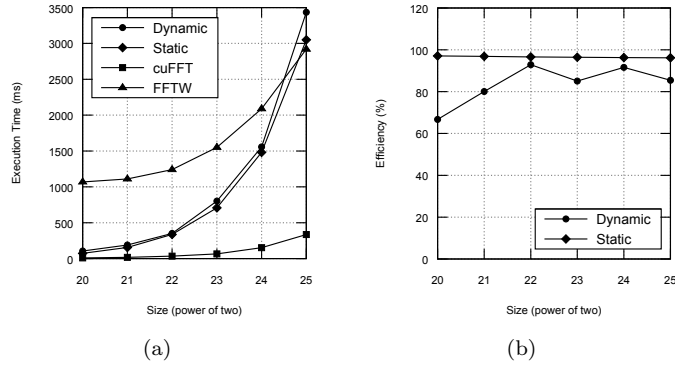

Figure 7. (a) Execution time in milliseconds for the FFT algorithm in GAMA, with dynamic and static schedulers, cuFFT and the FFTW's parallel version. (b) Efficiency η in percentage for GAMA, with dynamic and static schedulers.

cuFFT is considerably faster than both GAMA versions and the FFTW library, set to work with four threads, the same number of x86 workers running on GAMA. The static scheduler, which was set at assigning the whole workload to the CPU, obtained faster executions than the dynamic scheduler, due to the sampling process to build the performance model.

GAMA achieves the best efficiency η levels with the static scheduler, as Figure 7(b) shows. These efficiency levels went over 90%, whereas the dynamic scheduler delivered efficiency levels from $\approx$65% to $\approx$90%. Even though the execution times of the dynamic scheduler are not significantly different from the static scheduler, the levels of efficiency η are considerably smaller when compared with the static scheduler.

With respect to the Barnes-Hut algorithm, GAMA is compared with a developed OpenMP version (CPU) and an implementation presented by Martin Burtscher et al., highly tuned to deliver major performance levels on the GPU Fermi architecture [7], as shown in Figure 8(a). Moreover, its implementation is very hard to port to other architectures, and the used algorithm does not work on multiple devices.

Martin Burtscher's version ran substantially faster than GAMA, especially for larger input set sizes. On the other hand, the OpenMP version was considerably slower than GAMA, either when using the static or the dynamic scheduler. While Martin Burtscher's version was more than 12 times faster than GAMA with the dynamic scheduler, GAMA was $\approx$4 times faster than the OpenMP version.
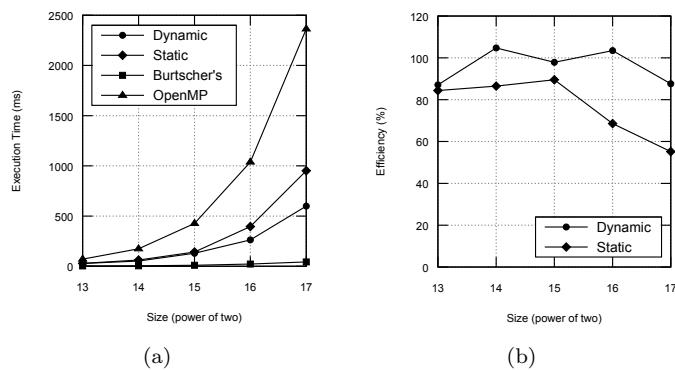

Figure 8. (a) Execution time in milliseconds of the Barnes-Hut algorithm running in GAMA, with both dynamic and static scheduling, along with the devised OpenMP and Martin Burtscher's implementations. (b) Efficiency η of GAMA, when using the dynamic and the static schedulers.

The dynamic scheduler in GAMA achieved more than 100% of efficiency twice, for systems with 2^{14} ($\eta \simeq$ 104%) and 2^{16} ($\eta \simeq$ 105%) particles, as shown in Figure 8(b). These results show that both the GAMA

framework and the dynamic scheduler can properly take advantage of the resources on the platform when running an irregular algorithm such as Barnes-Hut, in such a way that their cooperation is extremely effective.
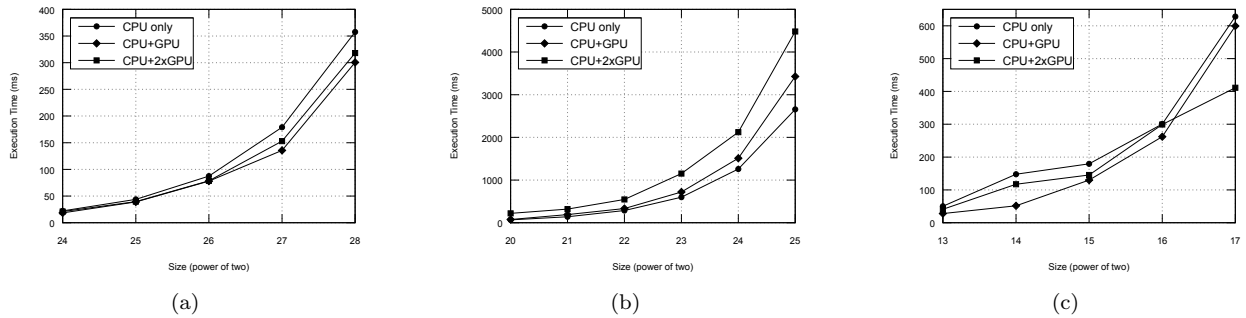

Figure 9. Scalability tests for 3 setup configurations: CPU-only, CPU+GPU and CPU+2xGPU, for SAXPY, FFT and Barnes-Hut. SAXPY experiments ranging from 2^{18} to 2^{28} elements in each vector. FFTs performed with 2^{20} to 2^{25} double precision elements and the Barnes-Hut algorithm with 2^{12} to 2^{17} particles systems.

The results in Figure 9 show the scalability tests for SAXPY, FFT and Barnes-Hut algorithms with different configurations (CPU-only, CPU+GPU and CPU+2xGPU). Figure 9(a) shows the SAXPY algorithm results: the CPU+GPU and the CPU+2xGPU configurations are up to 1.3x and 1.17x faster than the CPU only configuration, respectively. Despite having a slight increase in performance when adding a GPU, the algorithm decreases in performance when adding a second GPU. These results can be justified by the fact that the SAXPY algorithm is memory bound which can be troublesome on heterogeneous platforms since it is not possible to hide memory transfer latencies between devices with computation. This not only limits scalability, but also impairs performance.

Figure 9(b) shows the scalability results for the FFT algorithm. Adding a GPU in the FFT algorithm cripples performance and a second GPU increases the problem. This happens both due to the sampling process and the high tailorness of this particular FFT algorithm to the CPU-chip. Due to the performance modeling empirical scheme, each GPU executes part of the workload, thus creating a performance bottleneck. This problem is increased with every new GPU added to the system, sinte they all execute work parts.

Figure 9(c) shows the scalability results for the Barnes-Hut algorithm. The CPU+GPU configuration is up to 2.8x (2^{14} particles system) and the CPU+2xGPU is up to 1.5x (2^{17} particles system) better than the CPU-only configuration. Depending on the input data set size, it may be appropriate to use either one or two GPUs. However, the scalability is reduced among the plotted input data set sizes, since the shared PCIExpress channels become saturated at some point.

6 Conclusions

This paper presented a new dynamic scheduling model to simultaneously address regular and irregular applications on heterogeneous platforms. This scheduling scheme is based on an empirical performance model and on dynamically assigning chunks of tasks according to the load of each device at each instant, on a similar fashion to the HEFT policy. This model has been shown to be effective in addressing (ir)regular applications, in the presented evaluation runs with the SAXPY, the FFT and the Barnes-Hut n-Body solver, two regular and one irregular algorithms.

The results showed that the proposed scheduler distributed the workload according the best band of workload static distributions, for the three case studies. The performance of the FFT was hurt when building the initial performance model, running the sampling process that forces every CU on the system to run some tasks. However, the performance impair that arises due to the sampling process has shown to be nearly negligible.

In the GAMA framework, GPUs have direct access to the host memory (pinned memory), using PCIExpress channels. These channels have long latencies, that GAMA hides by overlapping data communication with computation: the larger the input data set size, the higher the assigned workload to the GPU, as seen for both the SAXPY and the Barnes-Hut algorithms.

With the devised scheduler, GAMA did beat open-source libraries such as Lapack++ (CPU) and cuBLAS (GPU) for SAXPY and FFTW (CPU) for the FFT. On the other hand, the performance of the proposed scheduler on a CPU+GPU configuration was substantially lower than cuFFT (GPU) and than Martin Burtscher's Barnes-Hut implementation (GPU). These latter implementations are faster, particularly due to the use of the device main and scratch-pad memories, while GAMA uses exclusively host memory.

The efficiency of GAMA for both the SAXPY and the Barnes-Hut algorithms achieved higher levels with the dynamic scheduler than with the best static solution. On the other hand, the best static workload distribution of the FFT was better than the dynamic scheduler, achieving more than 90% of efficiency across the range of tested input data set sizes. In the Barnes-Hut, the dynamic scheduler achieved ideal cases, by delivering more than 100% of efficiency in two particular input data set sizes.

The scalability of the proposed scheduler was noticeable for one and two GPUs, depending on the workload size and algorithm. However, the tested case studies are not especially scalable, which motivates the implementation of additional case studies in the future.

The usage of GPU accelerators by the dynamic scheduler was noticeable lower than expected, which happens due to the current structure of GAMA. As GAMA allocates the algorithm data-structures on the host memory, they are accessed by GPUs through long latency, shared PCIExpress channels, which impairs performance. GAMA project is however planned to include a software cache mechanism, which enables the accelerators memories to be used as data storage banks. As a result, this mechanism will very likely mitigate memory access latencies, especially in algorithms with significant data reuse, thus increasing algorithms performance.

Next planned work to improve this scheduler includes an implementation and assessment of the stealing and/or donation methods, which may help to mitigate or even correct eventual workload imbalance. This may especially arise from dynamic task spawn, a feature GAMA is expected to include soon, and from irregular applications in general.

Acknowledgment

References

[1] Ishfaq Ahmad, Yu kwong Kwok, and Min-You Wu. Performance Comparison of Algorithms for Static Scheduling of DAGs to Multiprocessors. In in Second Australasian Conference on Parallel and Real-time Systems, pages 185–192, 1995.

[2] Cédric Augonnet, Jérôme Clet-Ortega, Samuel Thibault, and Raymond Namyst. Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In The 16th International Conference on Parallel and Distributed Systems (ICPADS), Shangai, China, December 2010.

[3] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In Proceedings of the International Euro-Par Workshops 2009, HPPC'09, Lecture Notes in Computer Science, Delft, The Netherlands, August 2009. Springer.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation: Practice and Experience, Euro-Par 2009 best papers issue, 2010.

[5] Ioana Banicescu and Vijay Velusamy. Load Balancing Highly Irregular Computations with the Adaptive Factoring. In Proceedings of the 16th International Parallel and Distributed Processing Symposium, IPDPS '02, Washington, DC, USA, 2002. IEEE Computer Society.

[6] João Barbosa. GAMA framework: Hardware Aware Scheduling in Heterogeneous Environments. Technical Report, Informatics Department, University of Minho, September, 2012.

[7] Martin Burtscher and Keshav Pingali. GPU Computing Gems Emerald Edition: An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[8] K. Hazelwood K. Skadron C. Gregg, M. Boyer. Dynamic Heterogeneous Scheduling Decisions Using Historical Runtime Data. In Proceedings of the 2nd Workshop on Applications for Multi- and Many-Core Processors, San Jose, CA, June 2011.

[9] D. DeForest, A. Faustini, and R. Lee. Hyperflow. In Proceedings of the third conference on Hypercube concurrent computers and applications: Architecture, software, computer systems, and general issues - Volume 1, C3P, pages 482–488, New York, NY, USA, 1988. ACM.

[10] Gregory Diamos. Harmony: An Execution Model For Heterogeneous Systems. PhD thesis, Georgia Institute of Technology, December, 2011.

[11] Gregory Diamos and Sudhakar Yalamanchili. Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems. In HPDC'08, Boston, Massachusetts, USA, June 2008. ACM.

[12] Renato Ferreira, Wagner Meira Jr., Dorgival Olavo Guedes Neto, Lúcia Maria de A. Drummond, Bruno Coutinho, George Teodoro, Tulio Tavares, Renata Braga Araújo, and Guilherme T. Ferreira. Anthill: A scalable run-time environment for data mining applications. In 17th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2005), 24-27 October 2005, Rio de Janeiro, Brazil, pages 159–167. IEEE Computer Society, 2005.

[13] T. Gautier, J.L. Roch, and G. Villard. Regular Versus Irregular Problems and Algorithms. In In Proc. of IRREGULAR'95, pages 1–25. Springer, 1995.

[14] Chris Gregg and Kim Hazelwood. Where is the Data? Why You Cannot Debate GPU vs. CPU Performance Without the Answer. In International Symposium on Performance Analysis of Systems and Software, ISPASS, Austin, TX, April 2011.

[15] Dominik Grewe and Michael O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL, volume 6601, pages 286–305. Springer Berlin Heidelberg, 2011.

[16] Xin Huo, Vignesh T. Ravi, and Gagan Agrawal. Porting irregular reductions on heterogeneous cpu-gpu configurations. In HiPC'11, pages 1–10, 2011.

[17] Víctor J. Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers, HiPEAC '09, pages 19–33, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] Matthias Korch and Thomas Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms: Research articles. Concurr. Comput. : Pract. Exper., 16(1):1–47, December 2003.

[19] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Y. Meng. Merge: a programming model for heterogeneous multi-core systems. In ASPLOS, pages 287–296, 2008.

[20] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In MICRO, pages 45–55, 2009.

[21] Mario Mendez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. In Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, PPoPP '12, pages 107–116, New York, NY, USA, 2012. ACM.

[22] Jacques A. Pienaar, Anand Raghunathan, and Srimat Chakradhar. MDR: performance model driven runtime for heterogeneous parallel platforms. In Proceedings of the international conference on Supercomputing, ICS '11, pages 225–234, New York, NY, USA, 2011. ACM.

[23] Keshav Pingali, Milind Kulkarni, Donald Nguyen, Martin Burtscher, Mario Mendez-Lojo, Dimitrios Prountzos, Xin Sui, and Zifei Zhong. Amorphous data-parallelism in irregular algorithms. Technical Report TR-09-05, Department of Computer Science, The University of Texas at Austin, February 2009.

[24] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary Hall. EigenCFA: accelerating flow analysis with GPUs. SIGPLAN Not., 46:511–522, January 2011.

[25] Oliver Sinnen. Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing). Wiley-Interscience, 2007.

[26] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Trans. Parallel Distrib. Syst., 13(3):260–274, 2002.