Adaptive Partitioning for Irregular Applications on Heterogeneous CPU-GPU Chips *

Antonio Vilches[1], Rafael Asenjo[1], Angeles Navarro[1], Francisco Corbera[1], Rubén Gran[2], and María Garzarán[3]

[1] Universidad de Málaga, Spain. e-mail: {avilches, asenjo, angeles, corbera}@ac.uma.es
[2] Universidad de Zaragoza, Spain. e-mail: rgran@unizar.es
[3] Dept. Computer Science, UIUC, USA. e-mail: garzaran@illinois.edu

Abstract
Commodity processors are comprised of several CPU cores and one integrated GPU. To fully exploit this type of architectures, one needs to automatically determine how to partition the workload between both devices. This is specially challenging for irregular workloads, where each iteration's work is data dependent and shows control and memory divergence. In this paper, we present a novel adaptive partitioning strategy specially designed for irregular applications running on heterogeneous CPU-GPU chips. The main novelty of this work is that the size of the workload assigned to the GPU and CPU adapts dynamically to maximize the GPU and CPU utilization while balancing the workload among the devices. Our experimental results on an Intel Haswell architecture using a set of irregular benchmarks show that our approach outperforms exhaustive static and adaptive state-of-the-art approaches in terms of performance and energy consumption.

Keywords: Heterogeneous CPU-GPU chips, adaptive partitioning, dynamic scheduling, parallel for.

1 Introduction

Hardware vendors such as Intel, Qualcomm, AMD, or NVidia have recently released heterogeneous processors characterized by featuring several CPU cores and a GPU on the same die. However, the success of these systems will rely on the ability of the software to adapt the application level parallelism to exploit the underlying available hardware.

We consider the problem of efficiently executing the iterations of a parallel loop on heterogeneous CPU-GPU chips by executing on both, the CPU cores and the integrated GPU. This requires a carefully partitioning of the workload across the CPU cores and the GPU accelerator. Creating a dynamic and adaptive work distribution mechanism that is portable across processors is challenging and even more when the application exhibits irregularities. We propose a lazy partitioning of the loop iteration space into chunks (or blocks) of iterations. These chunks are greedily processed on the CPU cores or on the GPU and their sizes are adaptively computed

*This work has been supported by the following Spanish projects: TIN2010-16144, TIN 2013-42253-P, TIN2013-46957-C2-1-P, Consolider NoE TIN2014-52608-REDC, P11-TIC-08144 and gaZ: T48 research group.

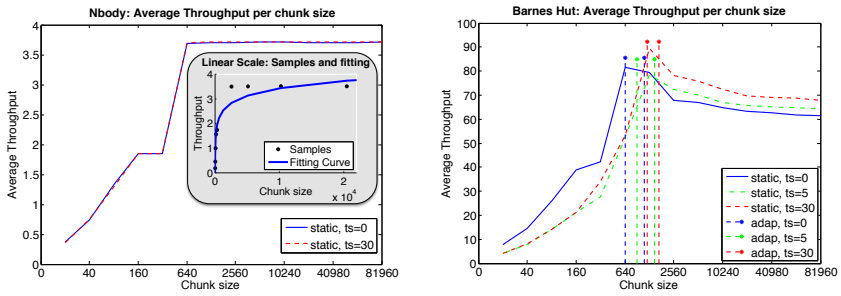
throughout the execution of the loop. To provide a productive programming interface we have extended the `parallel_for` template of the TBB task framework [17] to allow its exploitation on heterogeneous CPU-GPU chip processors. Previous task frameworks that offer support for heterogeneous CPU-GPU systems, like StarPU [1], OmpSs [3], XKaapi [13], and Concord [11] implement a variety of static and dynamic scheduling strategies, but do not adapt the size of the chunk of iterations assigned to the GPU. In contrast, our strategy monitors the throughput of each computing device (CPU cores and GPU) during the execution of the iteration space and uses this metric to adaptively resize chunks to optimize overall throughput and to prevent underutilization and load imbalance of GPU and CPUs due to small or large chunk sizes. We have found that correctly determining the chunk size for the GPU is highly important. For instance, if the GPU chunk size is too small, the GPU will not amortize the cost of offloading the work. If on the other hand, the GPU chunk size is too large, the GPU may suffer from too much memory contention due to uncoalesced memory accesses that can lead to suboptimal device performance. For these reasons, dynamically finding the appropriate chunk size for each device is critical to minimize execution time and energy consumption.

The contributions of the paper are: i) a novel adaptive partitioning algorithm that can be applied to parallel loops to automatically find the appropriate chunk size for the GPU and the CPU cores; ii) to demonstrate the positive impact of using adaptive GPU and CPU chunk sizes that dynamically vary based on changes in the throughput of the application; and iii) the evaluation, using regular and irregular applications, of our partitioning strategiy in terms of performance and energy consumption, showing that we outperform other state-of-the-art approaches.

2 Motivation

In this Section, we motivate the need for varying the chunk size when dynamically scheduling chunks of iterations of a parallel loop of an irregular application on an heterogeneous CPU-GPU chip. We discuss two different implementations of an n-body problem: a regular approach called Nbody [10] and an irregular one called Barnes Hut [12]. We have chosen these benchmarks because they use the same inputs and perform similar computations using two different approaches. These two benchmarks consist of a sequential outer loop that represents a sequence of time-steps. In each time-step all the body-body gravitational interactions are computed. To that end, each time-step contains a parallel loop that traverses all the bodies and an inner loop that, for each body of the parallel loop, computes the interactions with other bodies. Nbody is considered a brute force method that computes all the interactions that the n bodies in the system induce upon each other. On the other hand, Barnes Hut employs an octree data structure and the number of interactions of each body varies depending on its location. Thus, for each parallel loop iteration, the inner loop has a different number of iterations (or interactions), resulting in an irregular application.

To understand the impact that the chunk size has in the performance, we ran some experiments where Nbody and Barnes Hut were executed only on the GPU or only on the CPU cores. The experiments were conducted on a Core i7-4770, 3.4 GHz based on Haswell architecture with an integrated GPU HD 4600, for 75 time-steps and using 100,000 bodies as input. Figure 1 shows the average GPU throughput for different chunk sizes for Nbody (time-steps 0 and 30) and Barnes Hut (time-steps 0, 5, and 30) using log_2 scale in the x-axis. For each time-step, the data were collected by running the iteration space of the parallel loop with a fixed chunk size (the x-axis). The chunk sizes were set to 20×2^k, where k goes from 0 to 13, being 20 the number of execution units, nEU, on Haswell's GPU. Figure 1(a) shows the behavior of the regular Nbody application, where $chunk\_size = 640$ obtains an optimal aver-



(a) Nbody (b) Barnes Hut

Figure 1: Average GPU's throughput (iter./ms.) for different chunk sizes and different time-steps. Note the log_2 scale in x-axis.

age throughput and larger chunk sizes have a minimum impact on throughput. This result is the same across all time-steps. The inner subfigure in Figure 1(a) shows some samples of the throughput for different chunk sizes and the logarithmic curve that fits them using linear scale. We will come back to this topic in section 4. Figure 1(b) shows a very different scenario in the irregular Barnes Hut application. Here, in time-step 0, $chunk\_size = 640$ obtains the highest average throughput, while in time-steps 5 and 30 the highest average throughput is obtained with $chunk\_size = 1280$. For all time-steps, the application takes an important performance hit beyond those points. The figure also shows the average throughput when the chunk size is adaptively selected to the value that provides the highest throughput during the iteration space execution, for time step 0 (the blue marks '•'), time-step 5 (the green marks '•'), and time-step 30 (the red marks '•'). For time-step=0 the chunk sizes were selected between [620, 1320], while for time-step=5 and time-step=30 they were between [740, 1560] and [1280, 1760], respectively. In all cases, w.r.t. the maximum average throughput of the static experiments, an additional improvement of 5%-7% was observed thanks to the adaptive selection of the chunk size. These results illustrate that the GPU's chunk size has a significant impact in the GPU performance and that finding the best value for irregular applications can be challenging, because the appropriate chunk size might not be a constant, but rather change during the execution.

There are some reasons that explain the GPU throughput behavior shown in Figure 1. For both, Nbody and Barnes Hut, small chunk sizes result in low throughput due to: i) an insufficient amount of work offloaded to the GPU to exploit all execution units; and ii) the overhead of data transfer and kernel launch. On the other hand, for Nbody, large chunk sizes achieve higher utilization and amortize the cost of offloading work to the GPU, as shown in Figure 1(a). However, for Barnes Hut the chunk size has to be carefully and adaptively selected in order to make the most out of the GPU. As pointed out in a previous work [4], the access pattern in Barnes Hut can result in uncoalesced memory accessess, that may represent between 65% to 75% of the total number of issued instructions. Since many threads are trying to concurrently access uncoalesced memory addresses, contention for the shared memory bandwidth increases. Therefore, for this kind of codes, a large GPU chunk size might be counter-productive, as Figure 1(b) shows. Notice that previous works have not performed a sensitivity analysis for irregular applications as the one shown in Figure 1. Thus, those works assume a behavior similar to that of Nbody and assign large blocks to the GPU (to make sure the throughput is in the plateau). They do not impose any restriction on how large the GPU block size can be, as they do not consider that large GPU block sizes can harm performance, as they are not harmful for regular applications like Nbody.

Regarding the CPU cores, a similar chunk size study for Nbody and Barnes Hut showed that chunk sizes of 10 or more iterations always obtained the maximum constant throughput,

independently of the benchmark regularity. Moreover, our experimental data also show that large chunk sizes may result in load imbalance between CPU and GPU when they are collaboratively processing the parallel loop, especially at the end of the iteration space. Our scheduler addresses the described issues: adaptive chunk size selection and load imbalance.

3 Scheduler description

Our scheduler considers loops with independent iterations (parallel for) and features a dynamic work scheduling policy and an adaptive GPU and CPU chunk partitioning. Our approach adaptively partitions the whole iteration space into chunks or blocks of iterations. The goal of the partitioning strategy is to evenly balance the workload of the loop among the compute resources (GPU and CPU cores) as well as to assign to each device the chunk size that maximizes its throughput during computation. Our scheduler builds on top of an extension of the TBB `parallel_for` template for heterogenous CPU-GPU systems by Navarro et al. [15]. The developer can invoke our `parallel_for` function, which has the three following arguments: the iteration space, the body object of the loop, and the partitioner object. The latter implements the adaptive partitioning strategy that computes the appropriate chunk size for each compute device (CPU cores or GPU). This partitioning strategy is the novelty of this paper and is described in Section 4. The user is also required to write a `class` that processes the chunk on a CPU core or on the GPU. Our template can work on different types of heterogeneous systems, but our focus in this paper is on architectures with an on-chip GPU. Internally, our scheduler first selects an idle device (GPU or CPU core) and our partitioner computes the number of iterations (chunk size) that will be extracted from the range of the remaining iterations and will be assigned to that device. The time to compute the chunk on the CPU or GPU is recorded[1]. This is necessary to compute the device's throughput, which is used by the partitioner described next. More details of the scheduler can be found in [15].

4 Partitioning strategy

In this Section we explain how our partitioning approach, that we call LogFit, works. We assume that the execution time can be seen as a sequence of scheduling intervals $\{t_{G_0}, t_{G_1} \dots t_{G_{i-1}}, t_{G_i}, t_{G_{i+1}} \dots\}$ for the GPU and $\{t_{C_0}, t_{C_1} \dots t_{C_{i-1}}, t_{C_i}, t_{C_{i+1}} \dots\}$ for each CPU core. Each computing device at its *ith* interval, t_{G_i} or t_{C_i}, gets a chunk of iterations of size $G(t_{G_i})$ and $C(t_{C_i})$, respectively. The running time for the assigned GPU chunk, $T(t_{G_i})$, or CPU chunk $T(t_{C_i})$, is recorded. This time is used to compute the throughput in the corresponding interval, $\lambda_G(t_{G_i}) = G(t_{G_i})/T(t_{G_i})$ for the GPU or $\lambda_C(t_{C_i}) = C(t_{C_i})/T(t_{C_i})$ for a CPU core.

First we explain how LogFit computes the recommended chunk size, $G(t_{G_i})$, for the GPU. A previous work [2] have found that for regular applications, the GPU throughput follows a logarithmic curve with respect to the size of the chunk: $y = a\ln(x) + b$, being x the chunk size and y the estimated throughput. We corroborated that finding for the Nbody benchmark, as depicted in the inner subfigure of Figure 1(a). For this case, we can accurately compute the recommended chunk size by first collecting some points of the curve (by sampling at runtime the GPU throughput for different chunk sizes) and then applying the least squares fitting of the collected data to calculate the parameters a and b of the expression $a\ln(x)+b$. A threshold value, *thld*, is used to determine the point at which the estimated throughput is stabilized. The threshold allows us to compute the recommended chunk size as $a/thld$. This value represents the size at which the slope of the estimated curve varies less than 1%, and where the throughput can be considered as near-optimal. In our experiments we use 4 samples and $thld = 0.01$ (also

[1]Note that for the GPU, the time includes the computation time as well as the hostToDevice and deviceToHost times.

suggested in [2]). Using this procedure we found that the recommended chunk size for Nbody is 700, which is corroborated by inspecting Figure 1(a).

Unfortunately, as Figure 1(b) shows, irregular applications do not follow this logarithmic behavior. However, LogFit heuristic assumes that the near-optimal throughput for irregular applications can be approximated as the near-optimal throughput computed for each one of the intervals, $t_{G_0}, t_{G_1}, \ldots t_{G_{i-1}}, t_{G_i}, t_{G_{i+1}}, \ldots$. The near-optimal throughput for each interval can be estimated now from a logarithmic expression, and from that expression we get the new recommended GPU chunk size as explained above. The procedure is the following:

- Initially, to find the first GPU chunk size, $G(t_{G_0})$, for the first scheduling interval, t_{G_0}, we sample 4 block sizes, $x_i = nEU \times 2^{k-1}, k = 1:4$, being nEU[2] the number of Execution Units -EU- on the target GPU, and measure their corresponding throughput, y_i, getting a set $setSamples = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$. Next, we apply the logarithmic fitting to that set, as previously explained, to get $G(t_{G_0}) = a/thld$.

- After executing the chunk of iterations in the GPU, we get the time, $T(t_{G_0})$ and measure the throughput $\lambda_G(t_{G_0}) = G(t_{G_0})/T(t_{G_0})$. This measured throughput and the corresponding chunk size represent a new updated sample that can be used by our fitting procedure to find the recommended GPU chunk size for the next scheduling interval t_{G_1}. To avoid re-taking four samples in the next scheduling interval, we use the previous set of samples and just replace the fourth sample with the new one, so that the set of samples that we use to compute $G(t_{G_1})$ is now $setSamples = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), (G(t_{G_0}), \lambda_G(t_{G_0}))\}$. In general, for a given time interval t_{G_i} for which we want to compute its recommended chunk size $G(t_{G_i})$, we use the set of samples given by $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), (G(t_{G_{i-1}}), \lambda_G(t_{G_{i-1}}))\}$. We have analytically and experimentally validated that re-sampling for each time interval using small chunk sizes ($\{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)\}$) is not worthwhile: the penalty incurred by processing small chunk sizes at sub-optimal throughput is not compensated by the throughput improvement obtained due to a more accurate measurements in the scheduling interval. In any case, $(G(t_{G_{i-1}}), \lambda_G(t_{G_{i-1}}))$ already represents an accurate measurement for the next interval.

The rationale for monitoring the throughput and using this metric to adjust the chunk size is that if there is a change in the throughput, it is because the workload regime has changed and we need to adapt the GPU chunk size accordingly. If the throughput has increased with respect to the previous time interval, this is due to a decrement in the time per iteration to compute this last assigned chunk, or in other words, either the workload per iteration has decreased or the number of cycles that the EU have been stalled has been reduced due to an increment in the number of coalesced memory accesses. In any case, a problem of GPU under-utilization may appear. Thus, we increase the chunk size in order to increase Thread Level Parallelism (TLP). If there is not GPU under-utilization, enlarging the chunk size will not worsen the throughput, otherwise the throughput will likely increase in these scenarios. On the contrary, if the throughput has fallen, then there has been an increment in the time per iteration to compute the last chunk, which can be due to either an increment in the workload per iteration or an increment in the number of cycles that the EU have been stalled that can be produced by a higher number of uncoalesced memory accesses. In these cases, we reduce the chunk size in order to alleviate a probable problem of over-provisioned GPU and for the sake of better load sharing with the CPU.

Regarding the policy to set the CPU core chunk size, LogFit follows the heuristic by Navarro et al. [15]. Basically, this heuristic is based on a strategy to minimize the load imbalance among

[2] nEU = clGetDeviceInfo(deviceId, CL_DEVICE_MAX_COMPUTE_UNITS)

the CPU cores and the GPU while optimizing the throughput of the system. To that end, the optimization model described there recommends that: *each time that a chunk is partitioned and assigned to a device, its size should be selected such that it is proportional to the device's effective throughput.* Therefore, in our partitioning algorithm the chunk size assigned to a CPU core, $C(t_{C_i})$ verifies: $C(t_{C_i})/\lambda_C(t_{C_{i-1}}) = G(t_{G_i})/\lambda_G(t_{G_{i-1}})$, where $\lambda_C(t_{C_{i-1}})$ and $\lambda_G(t_{G_{i-1}})$ are the CPU and GPU throughputs of the previously computed scheduling interval, respectively. In other words, we have: $C(t_{C_i}) = G(t_{G_i}) \cdot \lambda_C(t_{C_{i-1}})/\lambda_G(t_{G_{i-1}})$.

5 Experimental Results

We run our experiments on a quad-core Intel Core i7-4770, 3.4GHz, based on Haswell architecture and featuring the HD-4600 on-chip GPU. We rely on Intel Performance Counter Monitor (PCM) tool [7] to access the HW counters (which also provide energy consumption in Joules). Intel TBB 4.2 provides the core template of the heterogenous `parallel_for`. The GPU kernels are implemented with Intel OpenCL SDK 2014. The benchmarks are compiled using Intel C++ Compiler 14.0 with -O3 optimization flag. We measured time and energy in 10 runs of the applications and report the average.

We have selected five benchmarks: Nbody [10] and Barnes Hut [12] presented in Section 2, plus three other irregular applications, PEPC[3] [8], CFD from the Rodinia suite [5], and SpMV from the SHOC suite [6]. For the Nbody benchmark an input set of 10,000 bodies and 50 time-steps were simulated. For Barnes Hut an input set of 100,000 bodies and 75 time-steps were computed. PEPC is fed with 100,000 point charges and 75 time-steps. CFD uses the missile.0.2M input data (which considers 2,000 time-steps) and SpMV processes the GL7d13 sparse matrix from the University of Florida Sparse Matrix Collection that exhibits a triangular-like profile. Barnes Hut and PEPC are examples of coarse grained application (each iteration executes in 100 and 110 microseconds, respectively, on one Haswell core) while CFD and SpMV are fine grained ones (0.3 and 7 microseconds per iteration, respectively, on one Haswell core).

Initially, we performed a sensitivity study of the main parameters of the LogFit partitioning strategy using our benchmarks, finding that by increasing (decreasing) *thld*, our approach tends to find smaller (greater) chunk sizes. While smaller chunk sizes quickly increase times, greater chunks also degrade times but the variation is smaller. We also considered more samples during the fitting, finding that by doubling the number of samples the sizes tend to slightly increase (less than 5%) which slightly degrades time (less than 2%).

5.1 Alternative partitioning strategies

To validate our LogFit approach we compare with three other related strategies: Static, Concord [11] and HDSS [2]. As a baseline, we use a Static partitioner (Oracle-like) that assigns one big chunk to the GPU and the rest of the iterations to the CPU. The size of this single GPU chunk is computed by a previous offline search phase that exhaustively looks for a partitioning of the iteration space between CPU and GPU that minimizes the execution time. This profiling step runs the application 11 times. For each run, the percentage of the iteration space offloaded as a single chunk to the GPU varies (between 0% –only CPU execution– and 100% –only GPU execution– using 10% steps). With this approach, profiling results for Nbody and Barnes Hut show that 50% and 70% of the iteration space should be computed on the GPU, respectively. Notice that this Static approach does not have runtime scheduling overheads, but the profiling step requires 11 runs for each value of $nThreads > 1$.

[3]PEPC is a Barnes Hut like problem but it computes electrical forces instead of gravitational ones. Another difference is that our Barnes Hut implementation sorts the particles to better exploit spatial locality.

The other two approaches, Concord and HDSS have a profiling phase where the relative speed of the GPU and the CPU is computed. Concord computes the relative speed with an online profiling phase where the GPU is assigned a fixed chunk size and the CPU cores pick chunks until the GPU finishes its chunk. Then, Concord switches to the execution phase where the relative speed computed during the profiling phase is used to partition once the remaining iterations between the CPU and GPU. Concord also considers re-profiling until the relative speed stabilizes or until a given percentage of the iteration space (50%) is processed. HDSS uses a training phase to compute the relative speed for both, the GPU and the CPU. It starts with a small chunk size and increases it gradually while recording the corresponding throughput of each sample. With the first four samples, HDSS fits a logarithmic curve and computes a first recommended GPU chunk size. HDSS keeps iterating in the training phase by adding samples and recomputing the logarithmic fitting and the relative speed until the difference between two consecutive speeds is less than a given threshold (1%) or a fixed percentage (20%) of the iteration space has been processed. Then, it moves to a completion phase where block sizes computed in the adaptive phase are no longer used. Instead, HDSS starts assigning chunks to the CPU and GPU relying on a Modified Guided Self-Scheduling (MGSS): it first assigns the largest possible chunk size to each device considering its relative speed and gradually reduces the chunk sizes towards the end of the iteration space to avoid load imbalance.

LogFit departures from these two previously described approaches in two ways. First, instead of looking for an stable relative speed that may never be found for irregular codes, LogFit's main goal is to adaptively select the recommended GPU chunk size that is big enough to fully occupy the GPU execution units while controlling the number of cycles that the execution units get stalled due to uncoalesced memory accesses. LogFit considers that the relative speed varies throughout the irregular code execution. Thus, for each scheduling interval, it also recomputes the CPU chunk sizes to ensure load balance between the CPU and GPU; Second, instead of having a steady phase following the adaptive one, LogFit keeps monitoring and adapting the GPU and CPU chunk sizes, while trying to minimize the overheads of the adaptive mechanisms, as we demonstrate next. Notice that LogFit and Concord remember information from one time-step to the next one: LogFit remembers $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$ samples, whereas Concord remembers the GPU relative speed. HDSS does not re-use information.

5.2 Performance and energy comparison

Figure 2 shows, for our benchmarks and the four different partitioning approaches: left - execution time (ms.), center - energy breakdown (Joules), and right - energy vs. performance (Joules vs. iterations per ms.) on Haswell for different number of threads. The energy breakdown distinguishes the energy consumed on the cores, E_CPU, on the GPU, E_GPU, and on the uncore components of the chip, E_Un. Note that when using only 1 thread, we get the only-GPU execution. In the energy vs. performance plots, each mark in the lines represents the number of threads from 1 to 5. In these plots, note that the closer to the right-bottom zone, the better the tradeoff between energy consumption and throughput. Typically, when increasing the number of threads, the curves move towards the upper right corner (higher performance and energy consumption), although there are exceptions, as explained next.

The study with the regular Nbody application aims at assessing the overhead of the adaptive engine in the three adaptive schedulers w.r.t. the Static approach. As the figure shows, Concord and LogFit perform similarly and execution times and energy consumption are close to those of the Oracle-like Static implementation. Even for this regular benchmark, Concord and LogFit can be faster than Static. In the figure this occurs for 3 threads because Static only evaluates 11 different partitions, while the adaptive approaches may find a finer tuned distribution of work between CPU and GPU. HDSS pays an additional overhead because it trains at the beginning

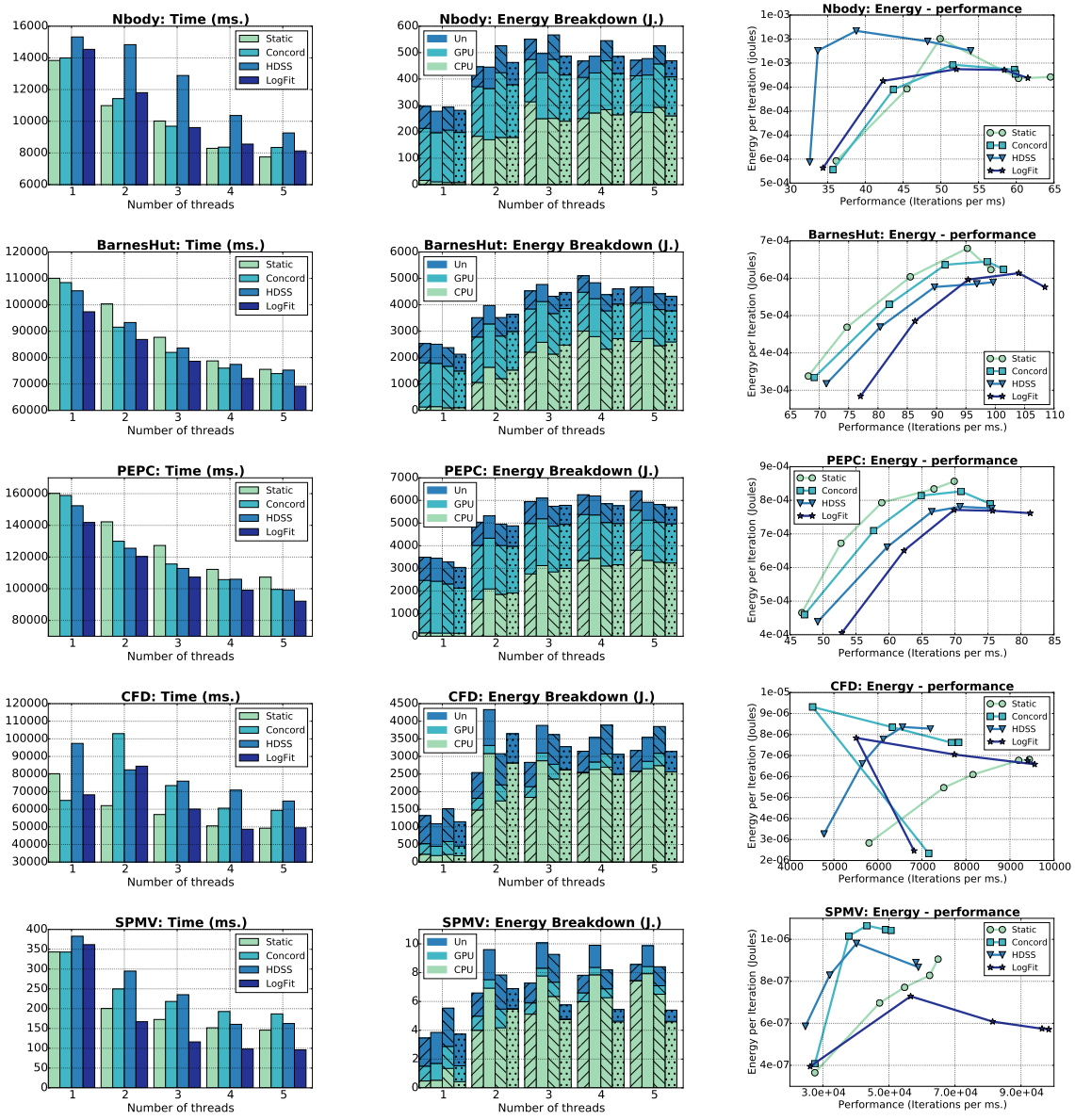


Figure 2: Results for Nbody, Barnes Hut, PEPC, CFD and SPMV benchmarks for Haswell. Time and Energy graphs, left to right: Static, Concord, HDSS and LogFit.

of each time-step, while Concord and LogFit can use previously computed information. For 5 threads, Concord, HDSS and LogFit are 8%, 20%, and 4.9% slower than Static, respectively. These results show that LogFit has an acceptable overhead in comparison with the Static approach. Moreover, LogFit does not need the offline profiling that Static requires.

Regarding the irregular coarse grained Barnes Hut and PEPC benchmarks, all adaptive approaches outperform the Static one. In terms of execution time, for 5 threads LogFit runs 10% and 17% faster than Static for Barnes Hut and PEPC, respectively. However, Concord runs just 2% (Barnes Hut) and 7% (PEPC) faster, whereas HDSS is 1% (Barnes Hut) and 8% (PEPC) faster. As the Energy and Energy vs. Performance plots show, HDSS exhibits similar performance as Concord, but the former is more energy efficient because it tends to offload

a larger portion of the iteration space to the GPU, and therefore, the E_CPU diminishes. In any case, LogFit achieves the minimum energy (for 1 thread) and the maximum performance (for 5 threads) as illustrated in the corresponding Energy-Performance plot. For PEPC and any given number of threads, LogFit always delivers the best performance with the minimum energy consumption.

For fine grained applications, CFD and SpMV, the adaptive approaches implemented in Concord and HDSS behave worse than LogFit. Let's recall that in both approaches, the relative speed resulting at the end of the profiling/training phase is used during the rest of the execution. However, in these benchmarks, this speed does not really fit the actual speed of the execution/completion phase. These two fine grained benchmarks also present and additional challenge because the GPU chunk size required to achieve a good enough GPU throughput is comparable to the whole iteration space. HDSS performs poorly for CFD because this approach enters the training phase for each of the 2000 time-steps of this application. During each run of the training phase, HDSS feeds the GPU with sub-optimal chunk sizes. When finally HDSS enters the completion phase there are not enough remaining iterations to assign to the GPU a chunk size big enough to obtain the predicted GPU throughput (the estimated relative speed can not be guaranteed). In addition, this leads to load imbalance with the CPU. By looking at the energy graphs we can note that for Concord and HDSS, the GPU consumes more energy (E_GPU) than for LogFit. This is explained because Concord and HDSS always assign the largest possible chunk size to the GPU in their completion phase. However, LogFit successfully detects that, for the exhibited granularity, the number of available iterations is not enough to assign to the GPU a chunk sufficiently large to obtain the predicted GPU throughput, so it's not scheduled on the GPU. All in all, for CFD and 5 threads, LogFit is 20% and 30% faster than Concord and HDSS, respectively. Also for CFD, Concord and HDSS consume 13% and 23% more energy, respectively. The advantage of LogFit is even bigger for SpMV and 5 threads: 94%, 69% and 52% faster and 82%, 55% and 58% more energy efficient than Concord, HDSS and Static, respectively. The Energy-Performance plots show that again LogFit is the approach that consumes the least energy (for 1 thread) and the fastest one (for 5 threads). On the average, considering the four irregular benchmarks and 5 threads, LogFit runs faster than Static, Concord and HDSS by 18%, 28% and 27% and consumes 18%, 23% and 19% less energy.

6 Related works

Approaches such as CUDA [16], OpenCL [18] and OpenACC [9] facilitate the programming of heterogeneous systems of a multicore and an associated GPU by supporting code portability across the two devices. However, they rely on the programmer to specify how the workload should be distributed between CPU and GPU.

Previous works consider the problem of automatically scheduling on heterogeneous platforms with a multicore CPU and an integrated or discrete GPU [14, 1, 3, 2, 11]. Among these works, the ones closer to ours are HDDS [2] and Concord [11]. The main difference between these works and ours is that they do not take into account the irregularity of the workload. Their main focus is to determine the computational speed of each device and with this information to assign the maximum chunk size to the GPU (and the multicore CPU) to avoid load imbalance. HDSS is introduced in [2] where the authors compare its proposal with Qilin [14], finding that HDSS always outperforms the later. In Section 5 we have compared LogFit against HDSS [2] and also Concord [11]. None of the mentioned related approaches change the block size dynamically based on the throughput of the application. Among them, Concord is the only one that evaluates irregular applications. It is also the only one that, like us, focus on heterogeneous CPU-GPU chips. All other approaches use the more powerful discrete GPUs.

StarPU [1] and OmpSs [3] have runtime systems that support heterogeneous executions, but the granularity of the workload that is offloaded to the GPU is determined by the programmer or the compiler, and not automatically determined as LogFit does.

7 Conclusions and future works

In this paper, we address the problem of finding the appropriate chunk size for GPU and CPU cores in the context of parallel loops in irregular applications running on heterogeneous CPU-GPU chips. We propose LogFit, a novel adaptive partitioning strategy that dynamically finds the chunk size that gets near optimal performance for the GPU at any point of the execution, while balancing the workload among the GPU and the CPU cores. Using a regular and a set of irregular benchmarks, we have assessed the performance and energy consumption of our partitioner with respect to a Static approach and other adaptive state of the art partitioners. For the studied irregular benchmarks on Haswell and 5 threads, we outperform the Oracle-like Static approach by up to 52% (18% on average) and avoid the exhaustive offline profiling. With respect to the state-of-the-art Concord and HDSS approaches and for 5 threads, we obtain up to 94% and 69% of speedup improvement (28% and 27% on average), respectively. Among all the approaches, LogFit is almost always the solution that results in the minimum energy consumption or the maximum performance. As future work, we will consider the parameter of energy consumption as part of the scheduling decisions.

References

[1] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *Proc of ICPADS*, 2010.

[2] M.E. Belviranli, L.N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013.

[3] J. Bueno, J. Planas, A. Duran, R.M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of GPU clusters with OmpSs. In *Proc. of IPDPS*, 2012.

[4] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *Proc. of IISWC*, 2012.

[5] Shuai Che et al. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *IISWC*, 2010.

[6] A. Danalis, G. Marin, C. McCurdy, J.S. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, and J.S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. In *GPGPU*, 2010.

[7] R. Dementiev, T. Willhalm, O. Bruggeman, P. Fay, P. Ungerer, A. Ott, P. Lu, J. Harris, P. Kerly, and P. Konsor. Intel performance counter monitor. `www.intel.com/software/pcm`, 2014.

[8] Paul Gibbon, Wolfgang Frings, and Bernd Mohr. Performance analysis and visualization of the n-body tree code pepc on massively parallel computers. In *PARCO*, pages 367–374. Citeseer, 2005.

[9] Alistair Hart. The OpenACC programming model. Technical report, Cray Exascale Research, 2012.

[10] Intel. *Intel OpenCL N-Body Sample*, 2014.

[11] Rashid Kaleem et al. Adaptive heterogeneous scheduling for integrated GPUs. In *Intl. Conf. on Parallel Architectures and Compilation*, PACT '14, pages 151–162, 2014.

[12] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *ISPASS*, Apr 2009.

[13] J.V.F. Lima, T. Gautier, N. Maillard, and V. Danjean. Exploiting concurrent GPU operations for efficient work stealing on multi-GPUs. In *SBAC-PAD'12*, pages 75–82, 2012.

[14] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proc. Micro*, pages 45–55, 2009.

[15] A. Navarro, A. Vilches, F. Corbera, and R. Asenjo. Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. *The Journal of Supercomputing*, 2014.

[16] NVidia. *CUDA Toolkit 5.0 Performance Report*, Jan. 2013.

[17] James Reinders. *Intel TBB: Multi-core parallelism for C++ programming*. O'Reilly, 2007.

[18] S.A. Russel. Levering GPGPU and OpenCL technologies for natural user interaces. Technical report, You i Labs inc., 2012.