Int J Parallel Prog (2019) 47:213–233
https://doi.org/10.1007/s10766-018-0555-0

Heterogeneous parallel_for Template for CPU–GPU Chips

Angeles Navarro[1] · Francisco Corbera[1] · Andres Rodriguez[1] · Antonio Vilches[1] · Rafael Asenjo[1]

Received: 11 September 2017 / Accepted: 8 January 2018 / Published online: 31 January 2018
© Springer Science+Business Media, LLC, part of Springer Nature 2018

Abstract Heterogeneous processors, comprising CPU cores and a GPU, are the de facto standard in desktop and mobile platforms. In many cases it is worthwhile to exploit both the CPU and GPU simultaneously. However, the workload distribution poses a challenge when running irregular applications. In this paper, we present LogFit, a novel adaptive partitioning strategy for parallel loops, specially designed for applications with irregular data accesses running on heterogeneous CPU–GPU architectures. Our algorithm dynamically finds the optimal chunk size that must be assigned to the GPU. Also, the number of iterations assigned to the CPU cores are adaptively computed to avoid load unbalance. In addition, we also strive to increase the programmer's productivity by providing a high level template that eases the coding of heterogeneous parallel loops. We evaluate LogFit's performance and energy consumption by using a set of irregular benchmarks running on a heterogeneous CPU–GPU processor, an Intel Haswell. Our experimental results show that we outperform Oracle-like static and other dynamic state-of-the-art approaches both in terms of performance, up to 57%, and energy saving, up to 31%.

✉ Rafael Asenjo
asenjo@ac.uma.es

Angeles Navarro
angeles@ac.uma.es

Francisco Corbera
corbera@ac.uma.es

Andres Rodriguez
andres@ac.uma.es

Antonio Vilches
avilches@ac.uma.es

1 Universidad de Málaga, Andalucía Tech, Málaga, Spain

Keywords Heterogeneous architecture · parallel_for · Dynamic scheduling · Adaptive partitioning

1 Introduction

We are currently seeing a growing variety of heterogeneous processors, characterized by featuring several CPU cores and an accelerator on the same die. The success of these systems will rely on the ability to map the application level parallelism to exploit the underlying available devices.

We consider the problem of efficiently executing parallel loops on heterogeneous CPU–GPU chips, by scheduling the work on both devices. This requires a careful partitioning of the iteration space into blocks of iterations (called chunks). The size of these chunks should be appropriately computed for each device to optimize the system throughput. To cater for programmer's productivity, we extend the `parallel_for()` function template provided by the task library TBB (Intel Threading Building Blocks) [22] to ease the implementation of parallel loops on heterogeneous CPU–GPU processors. We provide our template library with an adaptive partitioning strategy that is able to balance the workload among the available devices (CPU and GPU) while dynamically selecting the chunk size that ensures a near optimal throughput on each device.

Developing a dynamic and adaptive partitioning mechanism is challenging, and even more when the computational needs may vary at runtime, as it happens in irregular applications. There are several strategies that offer support for heterogeneous CPU–GPU systems, like StarPU [1], OmpSs [3], XKaapi [17], HDSS [2], Fluidic [21] and Concord [14]. These previous task frameworks implement a variety of dynamic scheduling and partitioning strategies which aim to balance the workload between the CPU and the GPU. In general, and assuming that the host-to-device and device-to-host times are not an issue, as it happens in chips with integrated accelerators that share the main memory, these strategies would consider that once the size of the chunk offloaded to the GPU is high enough to occupy all the GPU execution units, then the throughput of the accelerator will tend to stay fixed. In this paper, we demonstrate that not only a small chunk size, but also a large one can lead to a suboptimal throughput, especially when running irregular applications. In our strategy, we continuously resize the chunk of iterations offloaded to the GPU in order to prevent underutilization of the execution units, while also resizing the chunks assigned to the CPU cores to avoid load imbalance among the GPU and the CPU.

The contributions of the paper are: (i) We analyze the performance impact of executing irregular applications on integrated GPUs when the size of the chunk of iterations offloaded to the device varies (Sect. 2); (ii) We describe the proposed heterogeneous `parallel_for()` template and the API that simplifies movement of data among devices and heterogeneous execution orchestration (Sect. 3); (iii) We present a novel adaptive partitioning algorithm that can be applied to parallel loops to automatically find a near optimal chunk size for the GPU and the CPU cores (Sect. 4); and (iv) Using regular and irregular applications, we evaluate the efficiency of our approach, in terms of performance and energy consumption, and compare it with related work (Sect. 5).

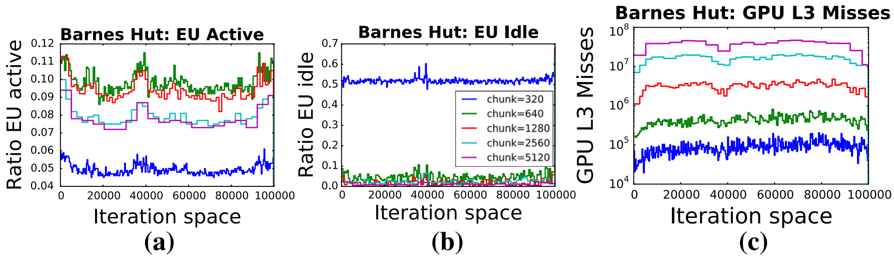

Fig. 1 Evolution of GPU hardware-based metrics while execution the first time-step of Barnes Hut on the Intel HD Graphics 4600 GPU.

2 Motivation

In this section, we motivate the need for varying the amount of iterations offloaded to the GPU (chunk size) while dynamically scheduling a parallel loop of irregular iterations.

As a running example, we introduce the Barnes Hut algorithm[1] for solving the n-body problem. This irregular benchmark performs a gravitational particle simulation for a number of time-steps. Each time-step can be implemented as a parallel loop in which, for each particle, its next position in a 3D space has to be computed. The irregularity comes from the fact that the amount of computations varies from particle to particle since the number of gravitational interactions depends on the relative distances of the particles. Therefore, the amount of work performed by each iteration of the `parallel_for` exhibits a high variability. To understand how the range of iterations offloaded to the GPU affects the performance in this benchmark we conduct some experiments on an i7-4770 Intel Haswell processor with four CPU cores and an integrated on-chip GPU (HD4600).

Figure 1 shows the evolution of three GPU hardware-based metrics for the first time-step and an input set of 100,000 particles (the iteration space, so 100,000 iterations in each time-step) in Barnes Hut. Each figure represents the evolution of the metric of interest when we offload chunks of fixed size (see chunk sizes legend in Fig. 1b) to the GPU. We use Intel VTune Amplifier 2015 [13] to collect the metrics. Figure 1a, b show the ratio of cycles for which all the GPU Execution Units are in the Active (EU Active) or Idle (EU idle) state, respectively, whereas Fig. 1c represents the Last-Level Cache (LLC) cache misses due to GPU memory requests.

These hardware metrics indicate that small chunk sizes (i.e. 320 iterations—or particles—) do not effectively keep active all the available EUs, as the ratio EU Idle indicates in Fig. 1b (see blue line). In contrast, when the chunk size is large enough to feed all the EUs (EU Idle < 0.1 for chunks > 320), then the EUs utilization improves. However, looking at Fig. 1a, we find out that chunk sizes higher than 640 decrease the ratio EU Active. Irregular benchmarks like Barnes Hut usually exhibit uncoalesced memory accesses (memory divergences) that can lead to scenarios where most of EUs are stalled due to memory contention. As pointed out in a previous work [4], Barnes

[1] https://github.com/avilchess/barneshut

Hut exhibits an uncoalesced memory access pattern that may represent between 65 and 75% of the total number of issued instructions. Such a pattern is responsible for the increment in the ratio EU Stalled when the chunk size increases. This is corroborated by the increment in L3 cache misses when the chunk size increases (see cyan and pink lines in Fig. 1c), which clearly increases the pressure in the memory interconnect. Similar behavior is also exhibited by other benchmarks, as has been reported previously [16, 23]. In our case, chunk sizes larger than 1280 dramatically increase L3 misses, which in turn increases the ratio of EU Stalled (> 0.9) and reduces the ratio of EU Active (< 0.08). All this results in that, for this time-step, the highest average GPU throughput is obtained when chunks of only 640 iterations are offloaded to the GPU. However, in other time-steps, for example 5 and 30, the highest average throughput is obtained with chunk size = 1280. In any case, for all time-steps, the application takes an important performance hit for larger chunk sizes.

We have validated that if the chunk size is adaptively selected to the value that provides the highest throughput during the execution, then an additional improvement of 5–7% is observed. This requires to vary the chunk sizes between [620–1320] for time-step 0, and between [740–1560] and [1280–1760] for time-steps 5 and 30, respectively.

On the other hand, we have observed that the effective throughput for the CPU cores is not that sensitive to the chunk size. This is because the CPU cores are provided with other architectural features that hide more effectively memory divergencies than GPUs. In fact, for our Barnes Hut example, as long as the chunk size is bigger than a certain threshold value[2], the average CPU throughput tends to be constant independently of the chunk size.

Clearly, irregular application can benefit from an adaptive mechanism to compute the optimal GPU's chunk size throughout the whole iteration space and all the time-steps. As we are interested in the collaborative execution between GPU and CPU cores, we also consider that offloading large chunk sizes to GPU may result in load imbalance, especially at the end of the iteration space. In any case, our partition strategy, LogFit, which we describe in Sect. 4, tackles the described issues: finding an optimal chunk size for the GPU and the CPU while balancing the load among the devices.

3 Programming Interface

In addition to performance and energy efficiency, we also care about the productivity and ease of use of heterogeneous architectures. To that end, we introduce the Heterogeneous Building Blocks (**HBB**) library. It is a C++ template library that builds on top of OpenCL and TBB libraries, and it offers a `parallel_for()` function template to run on heterogeneous CPU–GPU systems, as depicted in Fig. 2.

The left part of Fig. 2 shows the proposed software stack. Our library (HBB) offers an abstraction layer that hides the initialization and management details of TBB and OpenCL constructs (contexts, command queues, device_ids, etc.) [24]. The right part

[2] For instance, Threading Building Blocks library (TBB) [22], recommends to have CPU chunk sizes that take 100,000 clock cycles at least.

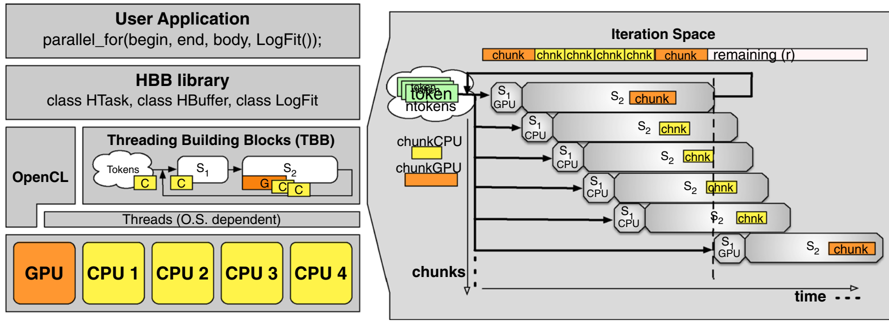

Fig. 2 Software stack and scheduling approach

of Fig. 2 shows that the internal engine that moves the `parallel_for()` function is a two-stage TBB pipeline. At the top of this part we can see the iteration space with the chunks that have already been assigned to a processing unit (in orange for GPU and yellow for CPU cores) and the remaining iterations, r, that have not been assigned yet (in white). The example shows an execution of the pipeline with 5 `tokens`. The tokens represent the number of items that can be in-flight traversing the pipeline. The tokens are used to control the number of computing devices that we want to use in the system (e.g., 5 tokens = 4 CPUs + 1 GPU). Once a token enters in the stage S_1, it checks if the GPU device is idle and that there is work that can be offloaded to the GPU. In that case, the GPU is acquired and initialized with the current GPU chunk. If there is no idle GPU device, then a CPU core is idle and the token is initialized with the current CPU chunk. In both cases, the partitioner extracts the corresponding chunk of iterations from the set of remaining iterations. Next, stage S_2 executes the selected chunk in the corresponding device (GPU or CPU core). Once a token has finished the work of stage S_2, it goes back and enters again stage S_1.

One of the biggest advantages of this `parallel_for()` implementation [19] is the asynchronous mode of computing, because each computing device (GPU or CPU core) will host one of the tokens that traverses the pipeline, and will carry out the corresponding chunk computations independently of the other devices. Thus, we avoid unnecessary synchronization points between computing devices with different computing power. In contrast, other state of the art approaches [18,26] suffer from load imbalance due to the usage of `fork-join` patterns with implicit synchronization points between CPU and GPU. In the rest of this section, we explain the functionality and implementation details of the main HBB components.

3.1 `HBuffer` and `HTask` Classes

The HBB library provides a `HBuffer` template class that offers an abstraction to avoid the explicit management of memory buffers. Each `HBuffer`<T> instance represents a logical shared buffer that can be accessed either by the CPU and GPU. As we can see in Fig. 3, this class hides the data buffer management and the user just needs to call the methods `getHostPtr()` (line 10) and `getDevicePtr()` (line 17) to get access

```
1  #include ''HTask.h''
2  #include ''HBuffer.h''
3  using namespace hbb;
4
5  class Body : public HTask{
6     HBuffer<int> * buf_a;
7  public:
8    Body(KernelInfo k, HBuffer * _buf_a) : HTask(k){...}
9    void operatorCPU(int begin, int end) {
10      int *a = buf_a->getHostPtr(BUF_RW);
11      for(i=begin; i!=end; i++) a[i] = a[i] + 1;
12   }
13   void operatorGPU (int begin, int end){
14      //Setting kernel arguments
15      setKernelArg(0, sizeof(int), &begin);
16      setkernelArg(1, sizeof(int), &end);
17      setKernelBuf(2, sizeof(BUF), buf_a->getDevicePtr(BUF_RW));
18      //Launching kernel
19      launchKernel(begin, end);
20   }
21 };
```

Fig. 3 Definition of class body

to the CPU memory buffer and the GPU memory buffer, respectively. These methods receive an additional parameter that sets the buffer access mode (`BUF_RO`, `BUF_WO` and `BUF_RW`)[3], which is used to avoid unnecessary data transfers. The allocation of the `HBuffer` instance is shown in Fig. 4 (line 8). The default constructor of the class takes as argument just the buffer size, but optionally, a second argument can be used to set the buffer in Zero-Copy mode (USE_ZCB, line 8 in Fig. 4).

Before using the `parallel_for()` function, the user must extend the `HTask` abstract class in order to define the body of the parallel loop (line 5 in Fig. 3). In this example we define the class `Body` that implements two methods: `operatorCPU()` (line 9) defines the CPU code for a single core in C++; and `operatorGPU()` (line 13) that takes care of the argument setting and kernel launching on the GPU. Note that kernel loading and compiling is automatically done by the `HTask` constructor when it receives the `KernelInfo` parameter (line 8) that comprises the kernel file path (`KernelFile`) and the kernel function name (`KernelName`). There are two methods to set the kernel arguments: the `setKernelArg()` method for variables of basic types (lines 15–16), and the `setKernelBuf()` method for instances of the class `HBuffer<T>` (line 17). Additionally, the `HTask` class provides a `launchKernel()` method (line 19) so the user does not have to manage the `command_queue` or the kernel id.

3.2 Function Template: `parallel_for()`

Figure 4 shows a main function with all the required component initialization and allocation to use the `parallel_for()`. The constructor of the `HInit` class receives

3 RO = Read-Only; WO = Write-Only; RW = Read–Write

```
1  #include ''parallel_for.h''
2  using namespace hbb;
3  int main(int argc, char* argv[]){
4    // Start task scheduler
5    HInit HInit(numcpus, true);
6    ...
7    KernelInfo k(kernelFile, kernelName);
8    HBuffer<int> * a = new HBuffer<int>(N,USE_ZCB);
9    Body body(k, a);
10   ...
11   parallel_for(begin, end, body, new LogFit());
12   ...
13 }
```

Fig. 4 Using the `parallel_for()` function template

two arguments: the first one indicates the number of active CPU cores and the second one is a `bool` that represents if the GPU must be initialized or not. Once the library has been initialized, the user can create the `KernelInfo`, `HBuffer` and the `Body` instances (lines 7–9) required to run the `parallel_for()` function.

The `parallel_for()` function template receives four parameters (line 11): the lower and upper bounds of the iteration space, `begin` and `end`, the `Body` instance which contains the implementation of the CPU and GPU body loop, and an instance of a partitioner, `LogFit`, that is explained next.

4 Partitioning Strategy

In this section, we describe the details of LogFit, our adaptive partitioning strategy. LogFit targets *parallel_for* loops that run onto heterogeneous processors. Our approach is designed as a three phases strategy consisting on: the Exploration Phase (EP), the Stable Phase (SP) and the Final Phase (FP). In the Exploration Phase we look for an initial GPU and a CPU chunk sizes that maximize the throughput in both devices. For it, we carry out an exploration in which we sample the throughput for different GPU chunk sizes and from them compute the point (chunk size) in which the throughput is stabilized. During this phase, for each sampled GPU chunk size, we compute a CPU chunk size that balances the load for all the devices. Then, in the Stable Phase we continuously re-adjust the GPU chunk size to cope with the application irregularities, and at the same time we re-compute a CPU chunk size that balances the load for all the devices while ensuring optimal throughput. The Final Phase is activated when there are few remaining iterations and we have to pay particular attention to load balance between the CPU cores and the GPU. In all these phases, our heuristic is based on approximating the GPU throughput as a logarithmic function of the GPU chunk size [2]. Thus, LogFit name stands for the logarithmic curve fitting in which the method relies to estimate future throughputs based on previous ones, as we explain next.

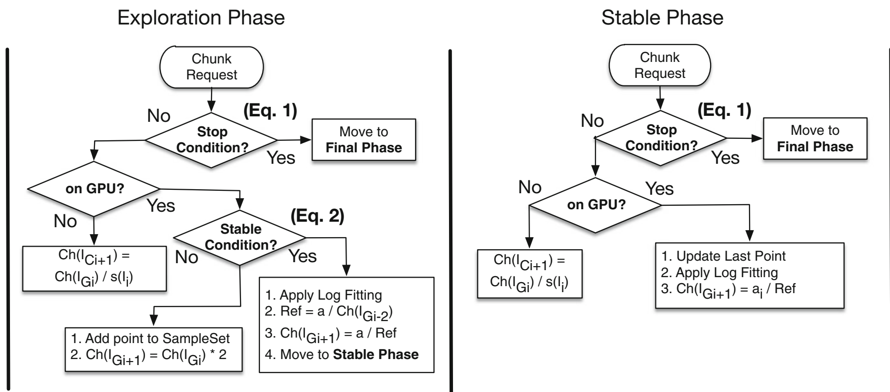

Fig. 5 LogFit's partition strategy flow chart

4.1 Overview of the Partition Problem

We assume that the execution of a *parallel_for* loop can be seen as a sequence of scheduling intervals $\{I_{G_0}, I_{G_1} \ldots I_{G_{i-1}}, I_{G_i}, \ldots\}$ for the GPU and $\{I_{C_0}, I_{C_1} \ldots I_{C_{i-1}}, I_{C_i}, \ldots\}$ for each CPU core. Each computing device at the ith interval computes a chunk of iterations of size $Ch(I_{G_i})$ (GPU chunk size) and $Ch(I_{C_i})$ (CPU chunk size), respectively. The running time for the assigned GPU chunk, $T(I_{G_i})$, or CPU chunk $T(I_{C_i})$, is recorded. This time is used to compute the throughput in the corresponding interval, $\lambda(I_{G_i}) = Ch(I_{G_i})/T(I_{G_i})$ for the GPU or $\lambda(I_{C_i}) = Ch(I_{C_i})/T(I_{C_i})$ for the CPU core. In particular, this throughput is the parameter that we keep monitoring to adjust the chunk size for each device to ensure optimal resource utilization and to avoid load imbalance. But let's explore in more detail how each phase of our partitioning strategy works.

4.2 The Algorithm Design: LogFit

LogFit is designed as a three-phase partition strategy: the Exploration Phase (EP), the Stable Phase (SP) and the Final Phase (FP). Figure 5 shows the flow chart for the Exploration and Stable Phases. In both phases, the **Stop Condition** is firstly checked. This condition queries for the corresponding scheduling interval, $i + 1$, if there are enough iterations to feed all devices with at least one chunk. If it is not the case, then the algorithm moves to the Final Phase, where a final partition must be performed to distribute all remaining iterations, r, among the available computing devices. The **Stop Condition** is satisfied when,

$$r < Ch(I_{G_i}) + Ch(I_{C_i}) \cdot ncores \tag{1}$$

Exploration Phase (EP) If there are remaining iterations and the GPU is idle, we check if we have to stay in the Exploration Phase sampling new chunk sizes or if the throughput of the previous samples has stabilized. For it, we check if the **Stable Condition** is satisfied:

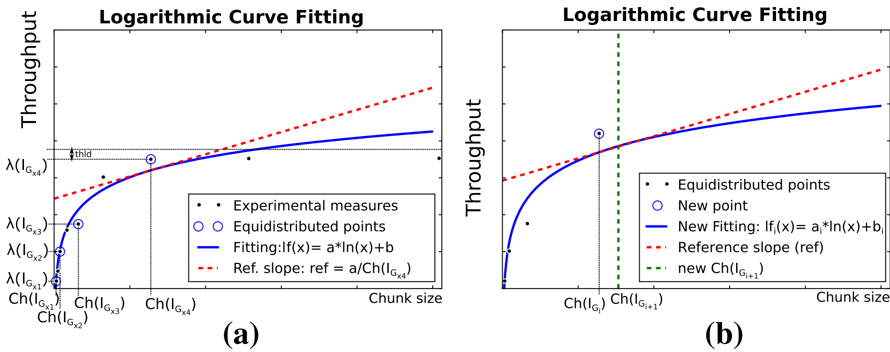

Fig. 6 Logarithmic fitting to compute GPU chunk sizes. **a** Fitting in EP, **b** fitting in SP

$$\left[\lambda(I_{G_{i-2}}) * (1+\theta) > \lambda(I_{G_i})\right] \wedge \left[\lambda(I_{G_{i-2}}) * (1+\theta) > \lambda(I_{G_{i-1}})\right] \quad (2)$$

which returns true when we have already found a sample (chunk size $Ch(I_{G_{i-2}})$) for which its throughput is similar enough (using a threshold value, θ) to the throughput of the two subsequent samples. As illustrated in Fig. 6a, during the EP we keep sampling the GPU throughput for different GPU chunk sizes (doubling the size each time) until the Eq. 2 returns true. When this happens, we select four (n = 4) equidistributed samples from the collected set of samples, being the last sample $(Ch(I_{G_{i-2}}), \lambda(I_{G_{i-2}}))$. With these selected samples we compute the Least Squares fitting method to obtain the value of the coefficients a and b of a logarithmic function, $lf(x) = a \cdot ln(x) + b$. Finally, we compute the reference value, $Ref = a/Ch(I_{G_{i-2}})$, which allows us to establish a reference ratio between throughput and chunk-size for the benchmark data-set. More precisely, Ref is the slope of the tangent line to $lf(x)$ at the point $Ch(I_{G_{i-2}})$ (see red dotted line in Fig. 6a). Note that this reference value is calculated only once, as you can see in Fig. 5.

In order to start with a big enough $Ch(I_{G_0})$, for the first GPU's scheduling interval, I_{G_0}, LogFit follows the expression $Ch(I_{G_0}) = nEU$, being nEU the number of Execution Units (EUs) on the target GPU.[4] Then, the chunk size is doubled at each interval until the Stable Condition is satisfied, $Ch(I_{G_i}) = nEU \times 2^i, i = 0 : t$.

During this phase, the CPU cores are also computing chunks of the iteration space. The first CPU chunk size is $Ch(I_{C_0}) = nEU$, but for next chunks, we adapt the size dynamically. To this end, we also monitor the CPU throughput for each CPU chunk, which allows us to compute the *relative speed* of the GPU over the CPU, $s(I_i) = \lambda(I_{G_i})/\lambda(I_{C_i})$. The factor $s(I_i)$ is used to adaptively adjust the size of the next chunk assigned to a CPU core using the expression: $Ch(I_{C_{i+1}}) = Ch(I_{G_i})/s(I_i)$. For example, if at a given scheduling interval the GPU has processed a chunk twice as fast as the CPU ($s(I_i) = 2$), CPU chunk size for the next interval will be half the size of the GPU chunk, so that we can keep the CPU and the GPU workload balanced at each scheduling interval, in spite of the different workload phases that irregular applications exhibit. Note that the CPU throughput is not as sensitive as the GPU to

[4] nEU = clGetDeviceInfo(deviceId, CL_DEVICE_MAX_COMPUTE_UNITS)

the chunk size, but this frequent adaptation will help to ensure workload balance at the end of the iteration space.

Stable Phase (SP) This phase is activated once the previous phase finds a chunk that optimally occupies the computational resources of the GPU. In this phase, LogFit adapts the size of new GPU chunks depending on the throughput variations. For regular applications that exhibit a constant throughput across the whole iteration space, the GPU chunk will stay almost constant. But, for irregular applications that may exhibit big throughput variations LogFit will propose bigger/smaller chunk sizes depending on throughput rise/drop. In general, the equation to compute the next GPU chunk size is $Ch(I_{G_{i+1}}) = a_i/Ref$, were a_i is obtained from re-computing the logarithmic fitting taking into account the GPU throughput of the last computed GPU chunk, $\lambda(I_{G_i})$ and $Ch(I_{G_i})$, respectively. This is illustrated in Fig. 6b, where the four samples used to compute a_i and b_i (from $lf_i(x) = a_i \cdot ln(x) + b_i$) are $\{(Ch(I_{G_{x1}}), \lambda(I_{G_{x1}})), (Ch(I_{G_{x2}}), \lambda(I_{G_{x2}})), (Ch(I_{G_{x3}}), \lambda(I_{G_{x3}})), (Ch(I_{G_i}), \lambda(I_{G_i}))\}$. This is, we use the first 3 samples used for the fitting in EP plus the forth sample that informs about the GPU chunk size and throughput just from the previous scheduling interval. The net effect of this heuristic is that we slightly increase the GPU chunk size when the throughput increases, and the contrary when the throughput decreases. That way we try to avoid both over provisioning or underutilization of the GPU EUs.

Regarding the CPU, in this phase we keep adapting the CPU chunk size to the GPU one by following the same equation, $Ch(I_{C_{i+1}}) = Ch(I_{G_i})/s(I_i)$, that we already used in the previous EP.

Final Phase (FP) As mentioned before, the execution of the Final Phase is activated by the **Stop Condition** shown in the Eq. 1. At this point, we need to find out the best possible partitioning of the remaining iterations. The goal is to minimize the time to compute the remaining iterations by finding $T_{min} = min(T_{CPU}, T_{GPU}, T_{HET})$. Thus, we devise three possible scenarios: (i) **CPU case**; (ii) **GPU case**; and (iii) **HET case**.

In the **CPU case**, the entire set of remaining iterations should be executed on the CPU cores. In this case, the estimated CPU execution time, $T_{CPU} = \frac{r}{n_{cores} \cdot \lambda(I_{C_i})}$, is the shorter one. GPU is therefore banned from doing any additional computation.

In the **GPU case**, the whole set of remaining iterations is assigned to the GPU. The estimated GPU execution time is computed in the same way as the CPU execution time. However, the GPU throughput depends on the size of the GPU chunk. Thus, we define a function ($thr_G(r)$) that accurately approximates the GPU logarithmic behavior by using the last set of four equidistributed samples. To simplify the maths, we assume that the GPU throughput exhibits a linear behavior between each pair of equidistributed samples. This is, we approximate the logarithmic curve by a piecewise linear with four pieces (that we call GPUthr segments). This allows us to estimate $thr_G(r)$ and then to estimate $T_{GPU} = \frac{r}{thr_G(r)}$. If this is the minimum time, the CPU cores are power gated.

In the **HET case**, we estimate if the remaining iterations should be computed on the GPU and the CPU cores. In order to find an optimal distribution, we need to partition the remaining iterations, r, so that $T_{HET} = T_{GPU} = T_{CPU}$. This is equivalent to find the number of iterations, x, to offload to the GPU such that: $\frac{x}{thr_G(x)} = \frac{r-x}{ncores \cdot \lambda(I_{C_i})}$,

where $thr_G(x)$ can be computed using the piecewise linear approximation of the last logarithmic function, as we did in the GPU case.

5 Experimental Results

In this section we first describe the architecture on which we conduct the experiments with the selected benchmarks. Next, we analyze how our partitioning strategy follows the throughput of the computational devices and adapts the size of the chunks accordingly. Finally, we compare our proposal with other related state-of-the-art dynamic alternatives to assess the performance and energy efficiency of our approach.

5.1 Experimental Setup

We run our experiments on an Intel Quad-Core processor: a Core i7-4770, 3.4GHz, based on the Haswell micro-architecture. This processor features an on-chip GPU, HD Graphics 4600. We rely on the Intel Performance Counter Monitor (PCM) library [9] to access the hardware counters (which also provide energy consumption in Joules). Intel Threading Building Blocks (TBB 4.2) provides the core task engine of our heterogeneous `parallel_for()`. The GPU kernels are implemented in OpenCL language and compiled by using the Intel OpenCL SDK 2014. Since there is no OpenCL drivers for Linux, Windows 7 is the installed OS. The host code part of the benchmarks is compiled with Intel C++ Compiler 15.0 and -O3 optimization flag. We measured time and energy in 10 runs of the applications and report the average.

We use five benchmarks whose details can be seen in Table 1. It shows the data input, number of invocations and the iteration space for each benchmark. These applications come from several domains and exhibit different behavior: regular versus irregular, coarse grained versus fine grained. Nbody [12] and Barnes Hut [15] are two different approaches to the n-body problem: the first one computes all the interactions between particles (regular), while the second one use an octree to traverse particles (irregular). PEPC [10] is similar to Barnes Hut but computes electrical forces instead of gravitational ones. Another important difference is that Barnes Hut sorts the particles to better exploit spatial locality, but PEPC does not. CFD comes from Rodinia Benchmark suite [6] and performs a Fluid Dynamic simulation. SpMV comes from the SHOC Benchmark suite [7], and performs a sparse matrix-vector multiplication. The

Table 1 Benchmarks description

Name	Suite	Description	Input|invocations	It. space
NBody	Intel OpenCL [12]	Particle simulation	100,000 bodies|1	100,000
Barnes Hut	Lonestar [15]	Particle simulation	runC|75	100,000
PEPC	PEPC [10]	Coulombs simulation	100,000 Q_i| 50	100,000
CFD	Rodinia [6]	Fluid dynamics	missile.0.2M|6000	232,536
SPMV	SHOC [7]	Linear algebra	GL7d16.mtx [8]|200	955,128

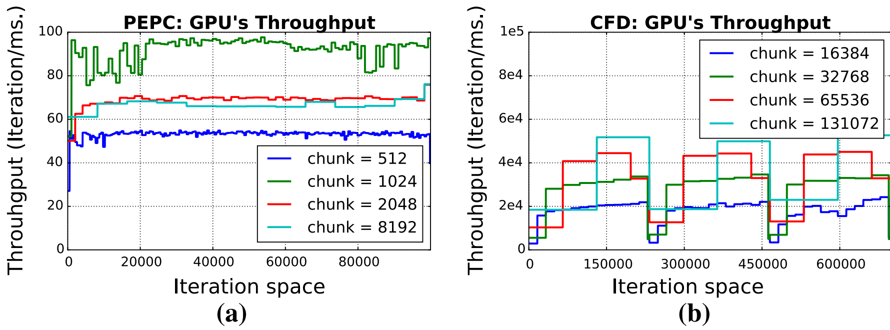

Fig. 7 Throughput evolution: PEPC and CFD on the Intel integrated GPU. **a** PEPC, **b** CFD

last four benchmarks are irregular. Considering these irregular benchmarks, Barnes Hut and PEPC are examples of coarse grained application (each iteration executes in approximately 32 and 69 µs, respectively, on one Haswell core) while CFD and SpMV are fine grained ones (0.13 and 0.1 µs per iteration, respectively, on one Haswell CPU core).

Figure 7 shows the GPU's throughput across the iteration space of the first time step for two irregular applications: PEPC and CFD. We can observe big differences between the two benchmarks plots, mainly because PEPC exhibits coarse grained parallelism but CFD presents with fine grained one. Actually, PEPC has a quite stable throughput, that is maximized for GPU chunk sizes of 1024 iterations. However, CFD exhibits higher throughput irregularities, as the most performing chunk size varies between 32, 768 and 131, 072, depending on the application phase.

5.2 Analysis of GPU's Chunk Size Adaptation to Throughput Variations

To graphically assess how well LogFit adapts the GPU chunk size to each throughput phase, Fig. 8a shows the evolution of the throughput (blue) and the GPU chunk size (green) throughout the iteration space for the first time-step of Barnes Hut. It is

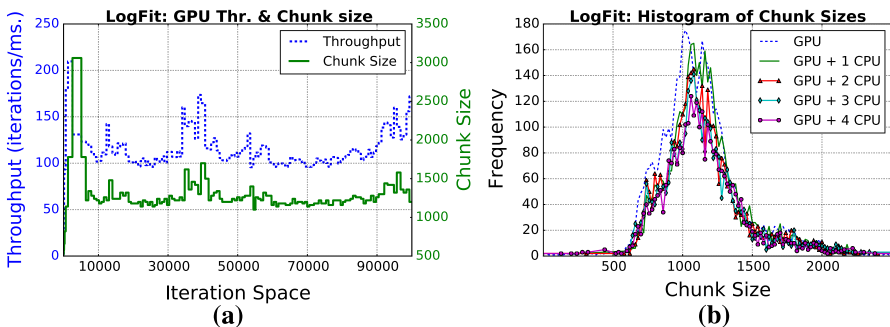

Fig. 8 GPU throughput and chunk study for Barnes Hut. **a** Throughput and chunk size evolution, **b** histogram

noticeable that the chunk size curve closely follow the throughput with just a small delay, since we use the previous throughput to compute the next GPU chunk size.

The Fig. 8b shows an analysis of the GPU chunk size adaptation throughout all time-steps for Barnes Hut. It shows the chunk size histogram that LogFit selects during several executions where the number of threads varies between 1 (only GPU execution) and 5 (four CPU cores plus the GPU host thread). Notice that for all executions the most frequent chunk sizes are within the range 1000 and 1300. We also see that the frequency of chunk sizes decreases as the number of threads increases, because, as more chunks are assigned to the CPU cores less chunks are computed on GPU. In any case, the increment in the number of threads does not affect the selection of the chunk size for the GPU, which demonstrates that our partitioning method works properly independently of the number of cores.

5.3 Sources of Overhead in Dynamic Partitioning

As we explained before, the host thread is responsible of feeding the GPU by executing the usual steps needed to offload work to the GPU: `hostToDevice()`, `launchKernel()`, `deviceToHost()`, and `clFinish()`. The first three calls asynchronously enqueue the memory transfers and the kernel launch on the GPU's command queue, whereas the latter is a synchronous wait. Figure 9 shows all the operations that take place each time a chunk of iterations is offloaded to the GPU. After the `deviceToHost()` operation is completed, the host thread is notified but some time may be taken by the OS to re-schedule the host thread. This time is illustrated in the Fig. 9 with the label "Thread dispatch".

In order to measure the relevant overheads involved in the execution on GPU's, some time stamps are taken on the CPU (Tc1, Tc2 and Tc3) and on the GPU (Tg1–Tg5) as depicted in Fig. 9. To get the CPU time stamps we rely on Intel TBB's *tick_count* class, whereas for the GPU we set the OpenCL command queue in profiling mode so that we can read the "start" and "complete" time stamps of each of the enqueued commands.

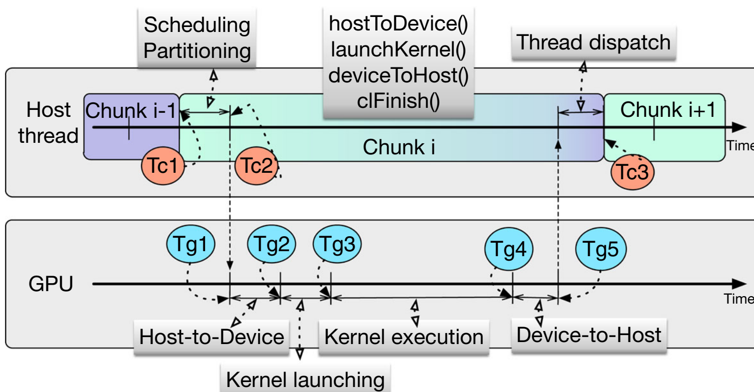

Fig. 9 Steps required to offload a chunk to the GPU

We use the previous times to compute the overhead of Scheduling and Partitioning, O_{sp}, Host-to-Device operation, O_{hd}, Kernel Launching, O_{kl}, Device-to-Host, O_{dh}, and Thread Dispatch, O_{td}, as follows:

$$O_{sp} = \frac{\sum_{\#GPUchunks} (Tc2 - Tc1)}{TotalExecutionTime} \qquad O_{hd} = \frac{\sum_{\#GPUchunks} (Tg2 - Tg1)}{TotalExecutionTime}$$
$$O_{kl} = \frac{\sum_{\#GPUchunks} (Tg3 - Tg2)}{TotalExecutionTime} \qquad O_{dh} = \frac{\sum_{\#GPUchunks} (Tg5 - Tg4)}{TotalExecutionTime}$$
$$O_{td} = \frac{\sum_{\#GPUchunks} \Big((Tc3 - Tc2) - (Tg5 - Tg1)\Big)}{TotalExecutionTime} \tag{3}$$

After identifying the main sources of overhead of our dynamic approach, we discuss the optimizations that can be implemented to tackle them. The first optimization technique is the zero-copy-buffer (ZCB) capability of heterogeneous chips that reduces data movements between CPU and GPU since they can share a common region of main memory. The second optimization rises the priority of the GPU host thread (called PRIO), so the GPU host thread has higher priority than any other thread and can be immediately dispatched. This is key when the GPU processes chunks more efficiently than the CPU, as it happens in our benchmarks. To boost the host thread priority we rely on the function *SetThreadPriority()* from Windows API. Finally, the third optimization is a combination of the previous ones, and we call it ZCB + PRIO.

In order to study the effects of those overheads and the impact of these three optimizations, we run the five benchmarks activating each one of the optimizations and measuring the overheads, total execution time and energy consumption. This measurements are shown in Fig. 10 and are obtained using 4 threads to process chunks on the 4 core CPU and one extra thread (host thread) to offload chunks to the integrated GPU. The left graph shows the overheads with four bars per benchmark: (i) the non-optimized reference version (Base, first bar); (ii) zero-copy-buffer optimization (ZCB, second bar); (iii) high-priority host thread optimization (PRIO, third bar), and (iv) the two previous optimizations combined (PRIO + ZCB, fourth bar). In the overhead graph on the left, each bar shows the breakdown of each particular overhead, being the Thread Dispatch overhead, O_{td} the lighter part of the bar and the Device-to-Host overhead, O_{dh} the darkest one.

The base version of coarse grain benchmarks like BarnesHut and PEPC suffer from a high Thread Dispatch overhead, 36 and 32%, respectively. In this case PRIO and

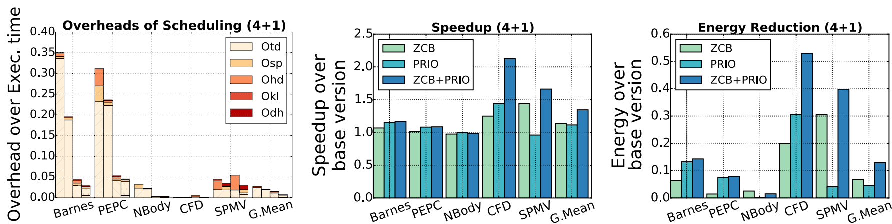

Fig. 10 Impact of optimizations on overhead, execution time and energy

ZCB + PRIO optimizations reduce total overhead to 5%. Scheduling and Partitioning, Kernel Launching and Device-to-Host overheads are negligible, but Host-to-Device Overhead, O_{hd}, can be up to 2% of the total execution time for a fine-grained benchmark like SPMV. This overhead can be reduced to 0.2% thanks to the ZCB optimization. This optimization also have a significant impact in the other fine-grained benchmark CFD.

The central graph shows the speedup achieved by each optimization over the base version. Coarse grain benchmarks BarnesHut, PEPC and NBody, do not benefit too much from the optimizations as the speedup is below 1.2×. However, ZCB + PRIO has quite an impact over fine-grained benchmarks, CFD and SPMV, yielding a speedup of 2.1× and 1.6× respectively. The left graph shows the energy consumption reduction, that strongly follows the speedup. All in all, applying ZCB + PRIO optimizations, the geometric mean of total overhead is below 1%, speedup is 1.3× and 12% less energy consumption.

5.4 Performance and Energy Comparison

To validate our LogFit partition strategy, we compare with three other related partition strategies: Static, Concord [14] 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 the 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, by using 10% steps).

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 within an online profiling phase where the GPU computes a fixed chunk size and the CPU cores compute small 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 distribute all the remaining iterations between the CPU and GPU at once.

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 is computed. 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 imbalances.

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, its main goal is to adaptively select the recommended GPU chunk size that is big enough to fully feed the GPU's EUs while controlling the number of cycles that the execution units get stalled due to memory contention. LogFit is the only alternative that considers variations in the relative speed throughout the complete code execution. Thus, for each scheduling interval, it recomputes the GPU and 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 three points $\{(Ch(I_{G_{x1}}), \lambda_G(I_{G_{x1}})),\ (Ch(I_{G_{x2}}), \lambda_G(I_{G_{x2}})),\ \{(Ch(I_{G_{x3}}), \lambda_G(I_{G_{x3}}))\}$ samples, whereas Concord remembers the GPU relative speed. On the other hand, HDSS does not re-use previous information.

In Fig. 11, we compare performance and energy consumption while executing the previously introduced benchmarks with the four different partitioning approaches. This figure shows three different plots for each benchmark: the left hand side plots show the execution time (in milliseconds) while executing the experiments from 1 threads to 5. In general, as we increase the number of threads, and use more computing devices, we reduce the total execution time. The middle plots show the total energy consumption (Joules) while increasing the number of threads, we show an energy breakdown which 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. However, from 2 to 5 threads, we add one CPU core until 4, plus the GPU. In the energy versus performance plots, right hand side on the figure, 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 higher 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 performs similarly to the Oracle-like Static implementation, while HDSS and LogFit tend to provide lower throughput and higher energy consumption from 1 to 4 threads. HDSS and LogFit pay an additional overhead due to longer training phases. Interestingly for 5 threads, LogFit can be faster and consumes slightly less energy than Static. The reason is due to the fact that Static only evaluates 11 different partitions, while our adaptive approach rightly finds a finer tuned distribution of work between the CPU and GPU. Anyway, the 1 thread execution (only GPU execution) shows the maximum overhead of dynamic strategies: Concord, HDSS and LogFit are 1, 2, and 5% slower than Static, respectively. These results show that LogFit has an acceptable overhead in comparison with the Static approach. However, LogFit does not need the offline profiling that Static requires and performs better than any other partition strategy with 5 threads, because performs a finer load balance strategy thanks to the final phase.

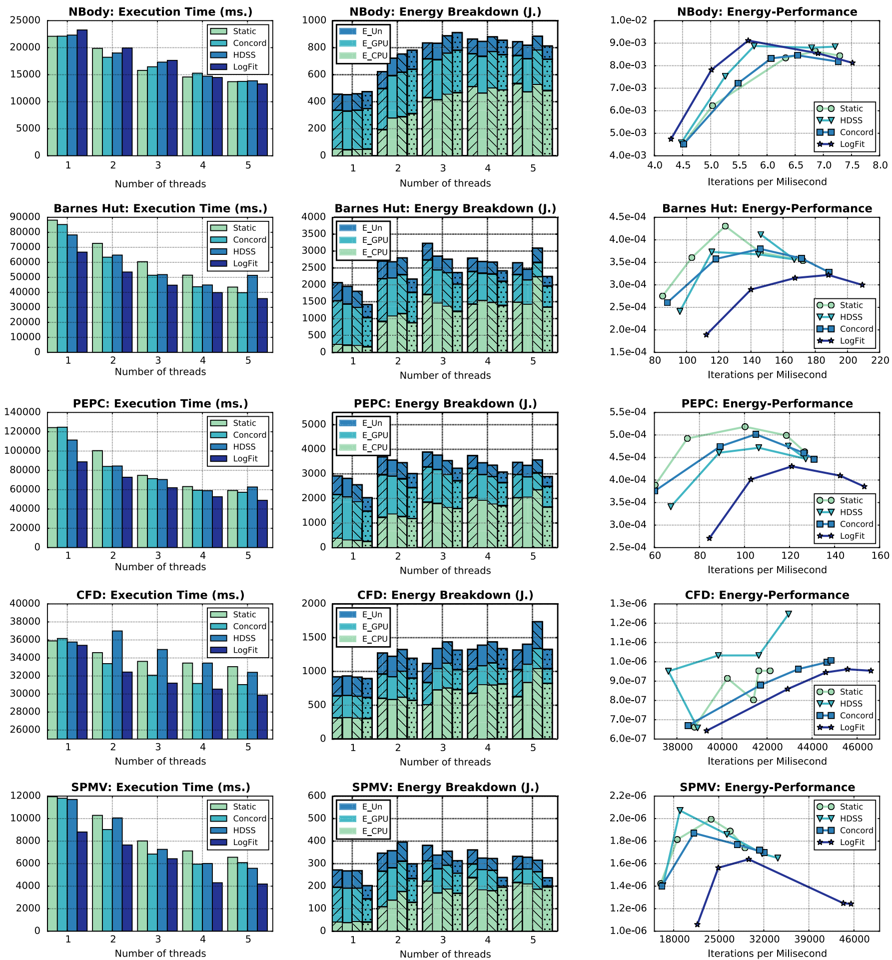

Fig. 11 Results for Nbody, Barnes Hut, PEPC, CFD and SPMV benchmarks. Time and Energy graphs, left to right: Static, Concord, HDSS and LogFit

By looking at the execution time plots of the irregular applications, it is noticeable that LogFit always outperforms the Static and the other dynamic alternatives, and it also consumes less energy.

Regarding the irregular coarse grained Barnes Hut and PEPC benchmarks, all dynamic approaches outperform the Static one. In terms of execution time, for 1 thread (only GPU), LogFit runs 32 and 40% faster than Static for Barnes Hut and PEPC, respectively. It keeps outperforming all other alternatives with any other number of threads, e.g., for 5 threads LogFit runs 21% faster than Static for Barnes Hut and PEPC. However, for 5 threads, Concord just runs 9% (Barnes Hut) and 7% (PEPC) faster, whereas HDSS gets poorer result than Static with 5 threads. According to the Energy plots, LogFit achieves the minimum energy for 1 thread (GPU execution). For Barnes Hut, LogFit consumes 31, 27, and 22% less energy than Static, Concord and

HDSS, respectively. Again, LogFit is more energy efficient than the other partition strategies when executing PEPC: for 1 thread, it consumes 30, 28 and 21% less energy than Static, Concord and HDSS, respectively. For both, Barnes Hut and PEPC, for any given number of threads, LogFit always delivers the best performance with the minimum energy consumption.

For fine grained applications, CFD and SpMV, the profiling phases implemented in Concord and HDSS behave worse than the LogFit's exploration phase. Let's recall that in both approaches, the relative speed resulting at the end of the profiling phase is used during the rest of the execution. However, in these two benchmarks, to find the right relative speed we need to perform an exhaustive search by sampling bigger chunk sizes. These two fine grained benchmarks also present and additional challenge as the GPU chunk size required to yield a near optimal GPU throughput is comparable to the whole iteration space. Thus, an exploration phase that allows to profile the whole iteration space will find a better chunk size that better exploits the GPU, as LogFit does.

HDSS performs poorly when executing CFD, as this approach executes the profiling (training) phase for each one of the 6000 time-steps of this application. During each run of the profiling phase, HDSS offloads small chunks to the GPU, with sub-optimal chunk sizes. When finally HDSS finishes the profiling phase there are not enough remaining iterations to assign an optimal chunk size to the GPU. In addition, this results in load imbalance with the CPU. LogFit successfully detects that the number of available iterations is not enough to stay in the Stable Phase, so it moves on to the Final Phase that successfully balances the work between the CPU and GPU.

LogFit achieves the highest improvement when executing SpMV. For instance, with 1 thread, LogFit runs 36, 34, and 33% faster than Static, Concord and HDSS, respectively, whereas with 5 threads it runs 57, 45 and 33% faster than Static, Concord and HDSS. According to the energy plot, the most energy efficient configuration is LogFit with 1 thread, consuming 26% less energy than Static, and less than 24 and 25% than the other dynamic strategies. 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 27, 19 and 28% and consumes 15, 14 and 24% less energy, respectively.

6 Related Works

Approaches such as CUDA [20], OpenCL [24] and OpenACC [11] facilitate the programming of heterogeneous systems composed of a multicore and a GPU. 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 [1–3,14,18]. Among these works, the ones closer to ours are HDDS [2] and Concord [14]. As commented in the previous section, 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. None of the mentioned related approaches dynamically change the chunk size 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. However, Concord also suffers from imbalance when the application is irregular, as they usually stop profiling after 50% of the items have been processed. Also, none of the previous works evaluate energy consumption.

StarPU [1] and XKaapi [17] provide a runtime for scheduling a DAG of tasks on heterogeneous CPU–GPU architectures, and a programming model with an API to select the scheduling policy, while OmpSs [3] provides a set of OpenMP-like pragmas and a run-time system to schedule tasks while preserving dependencies. These systems show performance results for heterogeneous execution, but the granularity of the workload that is offloaded to the GPU is determined by the programmer or the compiler, and not automatically determined by the run-time as in LogFit.

Some proposals extend work stealing for heterogeneous CPU–GPU architectures [5,25] and use a host thread to steal work for the GPU. These approaches partition the iteration space eagerly, as most work stealing approaches do. We perform a lazy partitioning to better determine the most appropriate chunks sizes to be assigned to the GPU and the CPU. This is a distinguishing feature of LogFit when compared to the above mentioned frameworks: we explore adaptive GPU and CPU chunk resizing to optimize throughput while maintaining load balance.

7 Conclusions and Future Work

Heterogeneous CPU–GPU chips enable the possibility of more coupled work distribution strategies between the integrated devices. In this paper we demonstrate that finding the appropriate chunk size for GPU and CPU cores in the context of parallel loops in irregular applications is critical for performance and energy efficiency. To tackle this issue, our heterogeneous `paralle_for()` template manages the orchestration of work and simplifies movement of data among devices. The template is supported by LogFit, a novel adaptive partitioning strategy that we also describe in the paper. This strategy 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, taking special care of tuning the chunk sizes during the final phase when the number of remaining iterations is not enough to completely feed all the computing devices.

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 57% (27% 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 45 and 43% of speedup improvement (19 and 28% on average), as well as an average energy saving of 14 and 24%, respectively. Among all the approaches, LogFit is 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.

Acknowledgements Funding was provided by Ministerio de Economía y Competitividad (Grant No. TIN2016-80920-R) and Consejería de Economía, Innovación, Ciencia y Empleo, Junta de Andalucía P11-(Grant No. TIC-08144).

References

1. Augonnet, C., Clet-Ortega, J., Thibault, S., Namyst, R.: Data-aware task scheduling on multi-accelerator based platforms. In: Proceedings of ICPADS, pp. 291–298 (2010)
2. Belviranli, M., Bhuyan, L., Gupta, R.: A dynamic self-scheduling scheme for heterogeneous multi-processor architectures. ACM Trans. Archit. Code Optim. **9**(4), 57 (2013)
3. Bueno, J., Planas, J., Duran, A., Badia, R., Martorell, X., Ayguade, E., Labarta, J.: Productive programming of GPU clusters with OmpSs. In: Proceedings of IPDPS (2012)
4. Burtscher, M., Nasre, R., Pingali, K.: A quantitative study of irregular programs on GPUs. In: Proceedings of IISWC, pp. 141–151 (2012)
5. Chatterjee, S., Grossman, M., Sbirlea, A., Sarkar, V.: Dynamic task parallelism with a GPU work-stealing runtime system. In: LNCS Series, vol. 7146, pp. 203–217 (2011)
6. Che, S., et al.: A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In: IISWC, pp. 1–11 (2010)
7. Danalis, A., Marin, G., McCurdy, C., et al.: The scalable heterogeneous computing (SHOC) benchmark suite. In: GPGPU, pp. 63–74 (2010)
8. Davis, T.A., Hu, Y.: The University of Florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1–25 (2011)
9. Dementiev, R., Willhalm, T., Bruggeman, O., et al.: Intel Performance Counter Monitor (2012). www.intel.com/software/pcm
10. Gibbon, P., Frings, W., Mohr, B.: Performance analysis and visualization of the n-body tree code PEPC on massively parallel computers. In: PARCO, pp. 367–374 (2005)
11. Hart, A.: The OpenACC programming model. Technical Report, Cray Exascale Research Initiative Europe (2012)
12. Intel: Intel OpenCL N-Body Sample (2014)
13. Intel VTune Amplifier 2015 (2014). https://software.intel.com/en-us/intel-vtune-amplifier-xe
14. Kaleem, R., et al.: Adaptive heterogeneous scheduling for integrated GPUs. In: International Conference on Parallel Architectures and Compilation, PACT '14, pp. 151–162 (2014)
15. Kulkarni, M., Burtscher, M., Cascaval, C., Pingali, K.: Lonestar: a suite of parallel irregular programs. In: ISPASS, pp. 65–76 (2009)
16. Li, D., Rhu, M., et al.: Priority-based cache allocation in throughput processors. In: International Symposium on High Performance Computer Architecture (HPCA) (2015)
17. Lima, J., Gautier, T., Maillard, N., Danjean, V.: Exploiting concurrent GPU operations for efficient work stealing on multi-GPUs. In: SBAC-PAD'12, pp. 75–82 (2012)
18. Luk, C.K., Hong, S., Kim, H.: Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: Proceedings of Microarchitecture, pp. 45–55 (2009)
19. Navarro, A., Vilches, A., Corbera, F., Asenjo, R.: Strategies for maximizing utilization on multi-CPU and multi-GPU heterogeneous architectures. J. Supercomput. **70**, 756–771 (2014)
20. NVidia: CUDA Toolkit 5.0 Performance Report (2013)
21. Pandit, P., Govindarajan, R.: Fluidic kernels: cooperative execution of OpenCL programs on multiple heterogeneous devices. In: CGO (2014)
22. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media, Inc. (2007)
23. Rogers, T.G., O'Connor, M., Aamodt, T.M.: Cache-conscious wavefront scheduling. In: IEEE/ACM International Symposium on Microarchitecture, MICRO-45 (2012)
24. Russel, S.: Levering GPGPU and OpenCL technologies for natural user interaces. Technical Report, You i Labs inc (2012)

25. Sbirlea, A., Zou, Y., Budimlic, Z., Cong, J., Sarkar, V.: Mapping a data-flow programming model onto heterogeneous platforms. In: Proceedings of LCTES, pp. 61–70 (2012)
26. Wang, Z., Zheng, L., Chen, Q., Guo, M.: CPU + GPU scheduling with asymptotic profiling. Parallel Comput. **40**(2), 107–115 (2014)