Noname manuscript No.
(will be inserted by the editor)

Strategies for Maximizing Utilization in multi-CPU & multi-GPU Heterogeneous Architectures

Angeles Navarro · Antonio Vilches · Francisco Corbera · Rafael Asenjo

the date of receipt and acceptance should be inserted later

Abstract This paper explores the possibility of efficiently executing a single application using multicores simultaneously with multiple GPU accelerators under a parallel task programming paradigm. In particular, we address the challenge of extending a `parallel_for` template to allow its exploitation on heterogeneous architectures. Previous task frameworks that offer support for heterogeneous systems implement a variety of static and dynamic scheduling strategies, although the size of the chunk of iterations assigned to each device is always fixed. However, due to the asymmetry of the computing resources we propose in this work a dynamic scheduling strategy coupled with an adaptive partitioning scheme that resizes chunks to prevent underutilization and load unbalance of CPUs and GPUs. In this paper we also address the problem of the underutilization of the CPU core where a host thread operates. To solve it, we propose two different approaches: i) a collaborative host thread strategy, in which the host thread, instead of busy-waiting for the GPU to complete, it carries out useful chunk processing. To implement this strategy, we modify our partitioning scheme to provide a chunk to the host thread each time that a GPU device gets new work; and ii) a host thread blocking strategy combined with oversubscription, that delegates on the OS the duty of scheduling threads to available CPU cores in order to guarantee that all cores are doing useful work. Using two benchmarks we evaluate the overhead introduced by our scheduling and partitioning algorithms, finding that it is negligible. We also evaluate the efficiency of the strategies proposed finding that allowing oversubscription controlled by the OS can be beneficial under certain scenarios.

Dept. of Computer Architecture, University of Malaga, Spain.
E-mail: {angeles, avilches, corbera, asenjo,}@ac.uma.es

1 Introduction

Given the recent evolution of processor design, it is expected that future generations of processors will contain hundreds of cores. To increase performance per watt ratio, the cores will certainly be non-symmetric or heterogeneous with a few extremely powerful cores and numerous, but simpler, cores. In the context of current heterogeneous architectures, many HPC platforms include nodes that have a multicore and one (or more) decoupled or discrete GPUs. The success of these systems will rely on the ability to adapt application level parallelism to the hardware parallelism available.

We consider the problem of efficiently executing a single application in a heterogeneous environment by allowing the simultaneous execution of work on the accelerators and CPUs. In this context, a runtime system needs to offer a programming model that considers heterogeneity both in terms of computing power and possibly a disjoint address space. Therefore, the effective utilization of resources in GPU-accelerated systems requires careful partitioning of the workload across CPU cores and GPU accelerators. Creating an adaptive application level work distribution mechanism that is portable across systems with varying node configurations is challenging and is further complicated by applications that exhibit irregularity in the granularity of parallelism across computational tasks. In this paper we focus on the problem of efficiently partitioning and dynamically scheduling chunks of work on heterogeneous architectures comprised of multicores (or a multi-CPU) and multiple GPUs. In particular, we have extended the `parallel_for` template of the TBB task framework [12] to allow its exploitation in heterogeneous systems. We have selected TBB because its task scheduler implementation is the most efficient when compared with other task scheduler

that represent the state-of-the-art on heterogeneous environments, as we will show in section 2. Although we have used TBB as the runtime supporting system, our scheduling and partitioning strategies can also be applied to any other task framework.

Previous task frameworks that offer support for heterogeneous systems, like StarPU [1], OmpSs [4] and XKaapi [8] implement a variety of static and dynamic scheduling strategies, although the size of the chunk of iterations assigned to each device is fixed. On the contrary, our strategies dynamically resizes chunks to prevent underutilization and load unbalance of CPUs and GPUs due to small or large block sizes. Our partitioning strategies are adaptive and take into account the computational speed of the resources to fully utilize all available processing units and avoid load unbalance in the system during the application lifetime.

Another departure from those state-of-the-art heterogeneous frameworks is related to the use of the host thread. In all accelerators-based architectures, the general convention is to dedicate one thread per accelerator (the *host thread*) to carry out the host-to-device, kernel launching and device-to-host chores. Depending on the synchronization mechanism of the accelerator's driver, the host thread may wait for device completion either busy-waiting or blocking or yielding. StarPU, OmpSs and XKaapi implicitly assume that each host thread runs in a dedicated CPU core typically using a busy-waiting synchronization, and that this setting will provide the best performance. This means that in a heterogeneous architecture with multiple GPUs, some of the CPU cores of the multicore will be exclusively dedicated to hosting a thread (the host thread). For applications where CPUs outperform GPUs significantly or where the number of GPUs is high, dedicating one or more CPU cores to just host GPUs can be a significant waste of resources. In this paper we study different approaches to avoid the underutilization of the core where a host thread operates: i) a collaborative host thread strategy, in which that host thread, instead of busy-waiting, carries out useful chunk processing; and ii) a host thread blocking strategy combined with oversubscription, that delegates on the OS the duty of scheduling threads to available cores.

The contributions of the paper can be summarized as follows:

- We extend the high level `parallel_for` TBB template to allow its execution in heterogeneous systems consisting of multi-CPU & multi-GPU.
- In order to perform the partition of work in the `parallel_for` template, an adaptive partitioning strategy derived from an analytical model that minimizes the load unbalance in the system, is proposed: *Non-Collaborative Host Thread* ($NCHT$).
- We address the problem of the underutilization of the CPU core where a host thread operates, by proposing two different approaches: i) collaborative host thread strategy, in which the host thread, instead of busy-waiting for the GPU to complete, carries out useful chunk processing. To that end, we design a second adaptive chunk partitioning strategy: *Collaborative Host Thread* (CHT); and ii) host thread blocking strategy combined with oversubscription, that delegates on the OS, the duty of scheduling threads to available CPU cores in order to guarantee that all cores are doing useful work. Let's note that this second approach is orthogonal to any partitioning strategy.
- Using regular and irregular applications, we evaluate the efficiency of our partitioning strategies, as well as the behavior of oversubscription under different synchronization mechanisms, finding that allowing moderate oversubscription controlled by the OS under a blocking or yielding synchronization can improve CPU core utilization, especially when the number of CPU cores is small and the number of GPU devices is high.

Next, to motivate our work, we present a comparative study of our basic adaptive partitioning strategy with StarPU, as well as a first evaluation of our approaches to fully utilize the CPU core where a host thread operates.

2 Motivation

For heterogeneous architectures, three factors are critical to achieve ideal performance: i) the computational speed of each computing resource should be accurately measured; ii) the assignment of chunks to the computational resources -the CPU cores and GPUs- must guarantee minimum load unbalance; and iii) the ideal chunk size that is sent to each resource for a task computation should be carefully identified and adaptively tuned during execution. The current state-of-the-art heterogeneous frameworks, do not consider factor iii) whereas in our approach we consider all of them. Ideal chunk size for GPU accelerators might be fixed to amortize data transfers (host-to-device and device-to-host) and to ensure that all theirs execution units are fully utilized. However, the CPU chunk size is not subjected to these constraints and it can be adaptively re-sized to guarantee that all computational resources finish at the same time.

In order to asses the relevance of the dynamic resizing of the chunk assigned to CPUs during computation, we conduct a first experiment in which we com-

pare our basic adaptive partitioning strategy that performs chunk re-sizing, *Non-Collaborative Host Thread*, *NCHT*, with StarPU fixed chunk size partitioning strategy. Our *NCHT* strategy has been implemented on top of the TBB task library and follows a greedy scheduling policy (more details about our strategy are found in section 4.1). Our partitioning heuristic follows this principle when computing the optimal chunk size for a resource that requests new work: if there are enough remaining iterations to keep all the resources busy then the chunk size selected is proportional to the resource's effective throughput; otherwise the size is computed from a weighted guided self-scheduling formula (more details in section 4.3.1).

The experiments were carried out in a platform described in section 5.1. Fig. 1 shows the execution time (y-axis) for a `MxV` benchmark (described in section 5.2) in a system configuration with 8 CPUs and 0, 1, 2 and 4 GPUs (x-axis), and 8 OS threads. The first bar, `NCHT`, represents our adaptive partitioning strategy. The next bars represent the times for the StarPU fixed chunk size partitioning strategy. This strategy has been evaluated with the three best available schedulers in StarPU: `SPU_greedy` that uses a central task queue from which available workers draw tasks to work on; `SPU_ws` is based on a work-stealing scheduler where when a worker becomes idle, it steals a task from the most loaded worker; and `SPU_heft` that takes task execution performance model and data transfer time into account to perform a HEFT-similar static scheduling strategy (it schedules tasks where their termination time will be minimal) [2]. For the StarPU results, different block sizes were tested (2,000, 200, and 20 matrix rows), obtaining similar results.

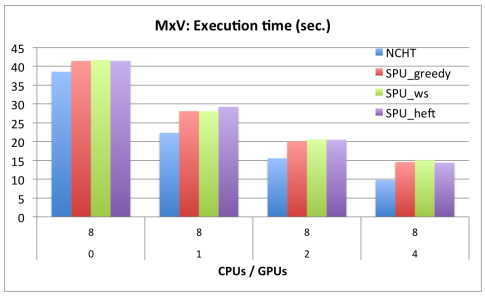


Fig. 1 Comparison of MxV execution times (secs.) for 8 threads running on 8 CPUs and (0, 1, 2, 4) GPUs.

In the only-CPU case (8,0), our strategy outperforms StarPU by 8%, mainly due to an internal StarPU overhead of the tasks management. This difference increases when 1, 2 and 4 GPUs are considered. Now StarPU best implementation, `SPU_heft`, is 25%, 32% and 53% slower than our strategy, respectively. We observe that these increments are explained by a larger unbalance among the computational resources, mainly between the GPUs and CPU cores. This result justify our selection of TBB as the underlying task framework and the necessity of implementing adaptive chunk resizing strategies.

The second challenge we address in this work is the effective utilization of the CPU core that executes a host thread. As mentioned before, heterogeneous frameworks assume that the host thread is kept waiting for GPU task completion and receipt of results. Instead of waiting (which may result into a wasted CPU core), we have modified the partitioning strategy in *NCHT*: each time that a GPU device gets a new chunk, the host thread also gets another chunk that will be processed in parallel in the CPU core. While the host thread is processing its chunk, it periodically polls if the GPU has completed its work. We call this strategy Collaborative Host Thread, *CHT*.

One alternative approach to keep the CPU core working, consists in relying on oversubscription and blocking (or yielding) the host thread while waiting. The idea is to have an extra CPU thread that will be dispatched to the CPU core that would remain idle if the host thread blocks (yield) while waiting for the GPU completion. More particularly, in CUDA, we can control the synchronization style with `cudaSetDeviceFlags()`. We have studied the following flags: *Spin*, *Yield* and *Blocking*. The default is *Spin* that keeps the host thread busy waiting in order to decrease latency when the device responds. StarPU uses this strategy by default. Under *Yield*, the host thread runs periodically and checks for the status of the GPU execution in round-robin fashion. When there is not oversubscription (other concurrent ready threads) it behaves like *Spin*, although with some additional overhead due to more frequent context switches. Finally, with *Blocking* the host thread just blocks until the GPU work is done. As the *Spin* synchronization wastes the CPU that runs the host thread, we have implemented our *NCHT* strategy using a *Blocking* mechanism (although the *Yield* mechanism is also evaluated in section 5.5) and we have studied the effect of oversubscription under this synchronization mechanism.

In Fig. 2 we represent the execution time (y-axis) for the `MxV` benchmark in a system configuration with 8 CPUs, 4 GPUs and 8, 12 and 16 OS threads (x-axis). The 8 threads experiment illustrates the scenario of no oversubscription, while the other two stand for scenarios with moderate and high oversubscription.

From the figure we see that under no oversubscription (8 threads), the `CHT` strategy is the more efficient (it is 10% faster than `NCHT` and 61% faster than the

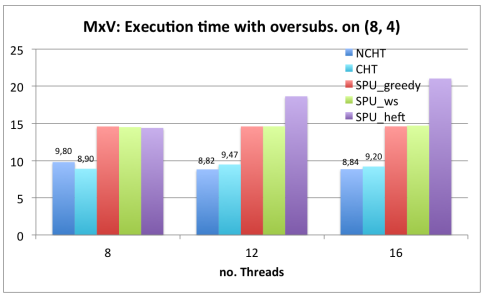


Fig. 2 Comparison of MxV execution times (secs.) for 8, 12 and 16 threads running on 8 CPUs and 4 GPUs

best StarPU implementation, `SPU_heft`). Under moderate oversubscription (12 threads), now `NCHT` along with the blocking synchronization mechanism is the more efficient (in fact is the best: 64% faster than the best `SPU`). However, strategies that uses polling (`CHT`) or spin mechanisms do not improve their times, and even degrade, as it's the case of `SPU_heft`. The degradation is even more important under high oversubscription. Under these later types of synchronization mechanisms, the context switch and cache cooling oversubscription overheads are evident. A more detailed analysis is covered in the experimental results section. Anyway, these results encourage the development of strategies that fully utilize the host thread, depending on the available synchronization mechanisms of the host thread: either a $NCHT$-like strategy with moderate oversubscription when blocking policy is available, or a CHT-like strategy without oversubscription otherwise.

In the next section we first describe our proposed programming interface based on TBB. Then, in section 4 we elaborate on the implementation details of the $NCHT$ and CHT alternatives, and we present the optimization model that supports our adaptive partitioning heuristic. Experimental results are covered in section 5, and we wrap up with related works and conclussions.

3 The parallel_for template

The TBB template `parallel_for` performs parallel iteration over a range of values. The default partitioner of this template recursively splits the range into subranges (chunks) until a threshold size is reached. Each chunk is then run as an independent task. Then, the internal TBB runtime scheduler performs work stealing to achieve load balance. The original template only allows execution on CPU multicores. In this paper we extend this template to offer hybrid execution on CPUs and GPUs. In this section we present the template API.

Fig. 3 shows in pseudo-code how to use the extended `parallel_for` construct in a heterogeneous system. As in any TBB program, the scheduler has to be initialized (see line 15). In this step, the developer sets the

```
1  class bodyObject{
2  ...
3  public:
4    void operatorCPU() (RangeH& r) {
5       for(i=r.begin; i!=r.end; i++){ ... }
6    }
7    void operatorGPU() (RangeH& r, Stream& s){
8       hostToDevide_async(r.begin, r.end, s.device, s.
            streamGPU);
9       launchKernel_async(r.begin, r.end, s.device, s.
            streamGPU);
10      deviceToHost_async(r.begin, r.end, s.device, s.
            streamGPU);
11   }
12 }
13
14 // Start task scheduler
15 task_scheduler_init init (nThreads);
16 ...
17 parallel_for (RangeH& itS, bodyObject(...),
       PartitionerH(grainSizeGPU));
```

Fig. 3 Using the `parallel_for` template

number of OS threads that the TBB runtime will create. Once initialization is done, the developer can invoke the `parallel_for` (line 17). Here, some parameters have to be indicated: the iteration space (the range `itS`), the body of the loop (`bodyObject()`), and the `PartitionerH()` which refers to the method to perform the partition of the iteration space across the computational resources. This method needs an input parameter `GrainSizeGPU` which is a tuple of the form $< GS_1, GS_2, \ldots >$, where GS_i is the default size of the chunk that is assigned to the GPU device with id_i.

In this paper we assume an adaptive partitioning strategy: although the user provides the optimal chunk size for each GPU device in the `GrainSizeGPU` tuple, it is the responsibility of the partitioner to compute the chunk size that will be assigned to the CPU cores that will be concurrently computing work with the GPU accelerators. We plan as a future work to also include the automatic computation of the optimal chunk size for each GPU device, similarly as done in [3], but in this work we focus on the cooperative work performed by the CPU cores and how to distribute the workload among them and the GPUs to prevent underutilization and load imbalance between these two types of resources.

The user is also responsible for writing the body code that processes the chunk on the CPU core or on the GPU stream device, as shown in lines 1-12 in Fig. 3. Two versions of the body must be coded: i) the version for the CPU - its operator just needs `r`, the range of the chunk to execute, line 4; and ii) the version for the GPU -its operator, as shown in line 7-, also needs `s` which is a struct with the GPU device id (`s.device`) and the stream id (`s.stream`) in the case that such a GPU provides support to more than one concurrent stream. In the example shown in lines 7-11,

the user can control one stream to concurrently perform the asynchronous host-to-device (line 8) and device-to-host (line 10) transfers, as well as the kernel launching (line 9).

4 Partitioning strategies

For this extended `parallel_for` we have implemented an engine for greedily scheduling the work over the computational resources. Over that engine, we propose two adaptive partitioning strategies, $NCHT$ and CHT, that are described in this section. As a key function of these strategies, we use an heuristic that adaptively computes the optimal chunk size for each computational resource and that will be presented later.

In more detail, our engine is designed as a two-stages pipeline, as depicted in Fig. 4. At the bottom of Fig. 4 we can see the iteration space with the chunks that have already been assigned (in yellow and orange) and the range `r` with the remaining iterations that have not been assigned yet (in white). As mentioned, our pipeline consists in two filters: Filter$_1$, which performs the selection of the computational resource where the work will be scheduled as well as the chunk partition, and Filter$_2$, which processes the chunk on the corresponding computational resource. This pipeline engine is implemented on top of the TBB pipeline template [12].

One important feature of a pipeline in TBB is the concept of token: a token represents a task that has been spawned for each input item and that is going to traverse all the pipeline filters. The left-hand side of Fig. 4 represents the tokens available to the scheduler. Other implementation detail is that we have used CUDA to spawn the task work on the GPUs. Once a task (with its corresponding chunk) is selected to be executed in a GPU, this GPU task ensures the consistency of its input data on the GPU device thanks to the host-to-device transfer operation. Then the GPU task launches the kernel, and finally the GPU task performs the device-to-host transfer operation. All these operations are performed asynchronously. Our work also supports the CUDA streams feature that allows the concurrency between data transfers and kernel launches. Thus, the number of tokens that we consider in our system is the number of GPU streams plus the number of CPU cores.

Using the pipeline engine just described, next we provide the details of the two adaptive partitioning strategies that we propose.

4.1 Non-Collaborative Host Thread

Our first partitioning strategy, called *Non-Collaborative Host Thread* ($NCHT$), is represented in Fig. 4. In this strategy chunks are assigned either to an idle GPU stream or to an idle CPU core. In case of assigning a chunk to the GPU stream, the corresponding CPU core will only be responsible for communicating the data and launching the kernel, as is usual in related works [1,4, 8].

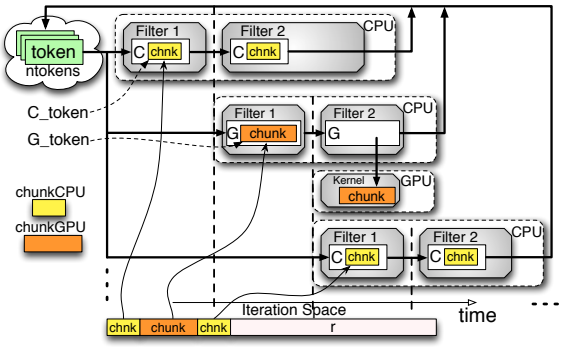


Fig. 4 Two stages pipeline engine implementing the $NCHT$ strategy

In this strategy, Filter$_1$ (see Fig. 5(a)) firstly acquires an idle GPU stream and then it checks if the device id is not null (lines 8-9). In that case, a G_token is created and initialized with information regarding the stream id of the GPU and the range of the GPU chunk taken (lines 10-12). If there is no idle GPU stream or the device id is null, then a CPU must be idle; thus, a C_token is created and is initialized with the range of the CPU chunk that the partitioner is going to extract from the range of the remaining iterations (lines 14-15).

Next, Filter$_2$ (see Fig. 5(b)) processes the chunk in the corresponding computational resource defined by the type of token that arrives, that can be either a G_token (line 10) or a C_token (line 18). In both cases, the time required for the computation of the corresponding chunk is recorded (lines 9-13 and lines 17-19). In the case of a GPU computation, the time recorded[1] is used to update the effective throughput on the corresponding GPU device and then to compute factor f (line 14). This factor represents the *computational speed* of the GPU device relative to a core. This computational speed is defined as the ratio of the time per iteration on the GPU device vs. the time per iteration in a CPU core. The factor f will be required for the partitioning function to adaptively adjust the size of the next chunk assigned to a core. In the case of a CPU computation, the time recorded is used to update the effective throughput on a CPU (line 20). Finally, in the case of a GPU computation, after the completion of the work and the calculation of factor f, the GPU stream is released (line 15).

[1] Let's note that for a GPU, the computation time as well as the transfer times are registered here.

```
1  class GetWorkFilter:public tbb::filter{
2  RangeH r;
3  PartitionerH ph;
4  public:
5    GetWorkFilter(Range _r, PartitionerH _pH):r(_r),
         pH(_pH){};
6
7    void* operator()(void*) {
8     myStreamGPU=acquire_StreamGPU();
9     if (myStreamGPU != NULL && myStreamGPU.device !=
          NULL){
10     myToken = new G_token();
11     myToken.streamGPU=myStreamGPU;
12     myToken.chunkGPU=pH.get_GPU_range(r,myStreamGPU
           .device);}
13    if (myStreamGPU == NULL || myStreamGPU.device ==
          NULL) {
14     myToken = new C_token();
15     myToken.chunkCPU=pH.get_CPU_range(r);}
16    return (void*) myToken;
17   }
18 }
```

(a) Filter$_1$

```
1  class ProcessWorkFilter:public tbb::filter{
2  BodyObject bO;
3  PartitionerH pH;
4  public:
5    ProcessWorkFilter(BodyObject _bO, PartitionerH _pH):bO
         (_bO),pH(_pH){};
6
7    void* operator()(void* myToken) {
8     if (myToken.type == G) {
9      t1=record_time();
10     bO.operatorGPU(myToken.chunkGPU, myToken.streamGPU);
11     completion=new_event(myToken.streamGPU);
12     waits(completion);
13     t2=record_time();
14     pH.set_factor_GPU(t2-t1, myToken);
15     release_StreamGPU(myToken.streamGPU); }
16    else {
17     t1=record_time();
18     bO.operatorCPU(myToken.chunkCPU);
19     t2=record_time();
20     pH.set_factor_CPU(t2-t1, myToken);}
21    return NULL;
22   }
23 }
```

(b) Filter$_2$

Fig. 5 Implementation of the filters for the *NCHT* partitioning strategy

Let us recall that for the GPU devices we exploit the concurrent nature of the device-memory transfers (host-to-device and device-to-host) and the kernel launches through the use of asynchronous call functions (see Fig. 3). After each request is inserted in the corresponding stream by the user operator, Filter$_2$ is responsible for inserting an event to detect the completion of the operation and waiting for it (lines 11-12 in Fig. 5(b)). The `waits()` function can be implemented using either a blocking or a yielding mechanism; both synchronization alternatives are evaluated in the experimental results section.

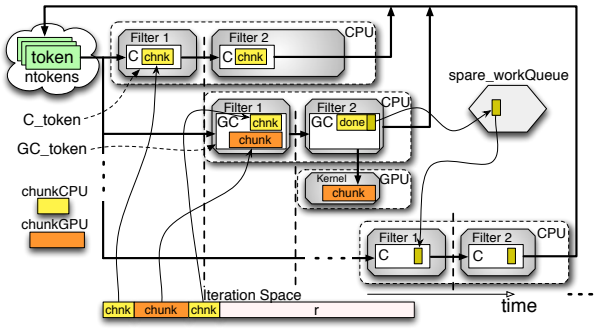


Fig. 6 Two stages pipeline engine implementing the *CHT* strategy

4.2 Collaborative Host Thread

Our second partitioning strategy, called *Collaborative Host Thread* (*CHT*) is the one represented in Fig. 6. One distinguishing feature of this strategy is that now, when assigning a chunk to a GPU stream, another chunk is also assigned to the corresponding host thread. Thus, both the CPU core that runs the host thread and the GPU stream will collaborate in computing the work in parallel. As mentioned in the motivation section, this strategy aims to ensure the full utilization of the CPU core that runs the host thread. Fig. 6 shows two structures that store the iterations: the range with the remaining iterations of the iteration space (the box at the bottom), and a queue called `spare_workQueue` that stores sub-ranges of non-executed iterations which were part of a chunk that was assigned to a host thread (the hexagon at the right-hand side).

In this strategy, Filter$_1$ (see Fig. 7(a)) after acquiring an idle stream in one of the GPU devices and checking that the device id is not null (lines 8-9), it creates a new type of token, a GC_token (a collaborative GPU-CPU token). So, our filter takes a chunk for the GPU (line 12) and a chunk for the host thread (line 13). In case that all GPUs are busy or that the device id is null, then a C_token is created. Let's note that in this strategy, a CPU chunk can come from two sources: i) from `spare_workQueue` (line 20, see the dark-yellow sub-range in the hexagon in Fig. 6) or ii) if that queue is empty, then the chunk come from the range that the partitioner computes from the remaining iterations of the iteration space (line 18, the white range in the bottom box of Fig. 6).

Regarding Filter$_2$ (see Fig. 7(b)), the difference arises when processing the chunks assigned to the collaborative GC_token. The host thread will launch the task with the GPU chunk on the corresponding GPU stream (line 10). But now, before waiting for the completion of the GPU stream, the CPU chunk assigned to host thread is partitioned into a set of sub-ranges of size `threshold` and stored in a temporal queue (line 12). Next, a sub-range is popped from that queue and pro-

```
1  class GetWorkFilter2:public tbb::filter{
2  RangeH r;
3  PartitionerH ph;
4  public:
5   GetWorkFilter2(Range _r, PartitionerH _pH):r(_r),
       pH(_pH){};
6
7   void* operator()(void*) {
8    myStreamGPU=acquire_StreamGPU();
9    if (myStreamGPU != NULL && myStreamGPU.device !=
         NULL) {
10    myToken = new GC_token();
11    myToken.streamGPU=myStreamGPU;
12    myToken.chunkGPU=pH.get_GPU_range(r,myStreamGPU.
          device);
13    myToken.chunkCPU=pH.get_CPU_range(r);
14   }
15   if (myStreamGPU == NULL || myStreamGPU.device ==
         NULL) {
16     myToken = new C_token();
17     if (spare_workQueue.is_empty())
18      myToken.chunkCPU=pH.get_CPU_range(r);
19     else
20      myToken.chunkCPU=spare_workQueue.pop_chunk();
21   }
22   return (void*) myToken;
23  }
24 }
```

(a) Filter$_1$

```
1  class ProcessWorkFilter2:public tbb::filter{
2  BodyObject bO;
3  PartitionerH pH;
4  public:
5   ProcessWorkFilter2(BodyObject _bO, PartitionerH _pH):bO
       (_bO),pH(_pH){};
6
7   void* operator()(void* myToken) {
8    if (myToken.type == GC) {
9     t1=record_time();
10    bO.operatorGPU(myToken.chunkGPU, myToken.streamGPU);
11    completion=new_event(myToken.streamGPU);
12    setOfChunks=split_by(myToken.chunkCPU, threshold);
13    while (!setOfChunks.is_empty()) {
14     otherChunk=setOfChunks().pop_chunk();
15     bO.operatorCPU(otherChunk);
16     if (completion.status == COMPLETE) {
17       spare_workQueue.push_chunk(compact_by(setOfChunks,
             threshold));
18       break; }
19    }
20    waits(completion);
21    t2=record_time();
22    pH.set_factor_GPU(t2-t1,myToken);
23    release_StreamGPU(myToken.streamGPU); }
24   else {
25    t1=record_time();
26    bO.operatorCPU(myToken.chunkCPU);
27    t2=record_time();
28    pH.set_factor_CPU(t2-t1,myToken); }
29   return NULL;
30  }
31 }
```

(b) Filter$_2$

Fig. 7 Implementation of the filters for the CHT partitioning strategy

cessed by the host thread in the CPU core (lines 14-15). Then, the status of the GPU stream completion event is polled. In the case that the GPU stream status is `COMPLETE`, then the remaining sub-ranges stored in the temporal queue are compacted and returned to the scheduler in the `spare_workQueue` (line 17). Otherwise, in the case that the GPU stream status is not `COMPLETE` yet, a new sub-range is popped from the temporal queue and processed by the host thread in the core. This process of polling and computing sub-ranges is repeated until the GPU stream completes or the host thread computes all the sub-ranges stored in the temporal queue. As in the previous partitioning strategy, the time required to compute a GPU or a CPU chunk is recorded and used to compute factor f.

Before computing the chunk size for each computational resource, we consider an optimization model to help us understand how the different resources interact. The purpose of building this idealized model is to provide insight into the key aspects of the problem of finding the right chunk size so that we can develop an adaptive partitioning heuristic which will work well in practice.

4.3 Optimization model for the chunk size

We model our partitioning strategy as a simplified optimization problem whose goal is to minimize the system load unbalance subject to the constraint that the system throughput is maximum. For this model we simply assume that the optimal chunk size of each GPU device is known and stationary during program execution. However, the optimal chunk size for a CPU core will be a value that our optimization problem will look for. Specifically, the optimal chunk size of each GPU device is given by the user as an input parameter in the `parallel_for` template invocation (see line 17 in Fig. 3). These sizes are specified in `GrainSizeGPU`, which is a tuple of the form $< GS_1, GS_2, \ldots GS_k >$, where GS_i is the default size of the chunk that the partitioner will assign to the GPU device with id_i. The sizes depend on the kernel specified in the task body and the particular GPU device hardware: the number of registers, size of the memory of each type required in the kernel, maximum number of warps, etc. Let us assume that R_i and R_c represent the range of iterations of a `chunkGPU` for GPU$_i$ and a `chunkCPU` for a core, respectively. Every time that a chunk R_i (R_c) is executed in the corresponding GPU$_i$ (or CPU core), its processing time, T_i (T_c), is recorded to compute the effective throughput on each GPU (CPU core) as λ_i (λ_c), by following the next expressions:

$$\lambda_{io} = \lambda_i \tag{1}$$

$$\lambda_i = \alpha \cdot \frac{R_i}{T_i} + (1-\alpha) \cdot \lambda_{io} \tag{2}$$

where λ_{io} represents the previous value of the effective throughout (initially 0), whereas eq. 2 is the *exponential moving average* of the current throughput sample (R_i/T_i) and the previous measures (λ_{io}). α is a smoothing constant between 0 and 1 that weights the contribution of the current measurement vs the previous ones. The optimal value for this parameter and its impact on our partitioning strategy are issues discussed in the experimental section. Let us note that for the computation of λ_c we replace R_i and T_i for R_c and T_c, respectively, in eq. 2. Once the effective throughput of GPU$_i$, λ_i, is computed, then the computational speed of such a device is calculated as

$$f_i = \frac{\lambda_i}{\lambda_c} \tag{3}$$

where λ_c represents the current effective throughput in a CPU core. On the other hand, when a CPU core execution is invoked, the effective throughput in a CPU core, λ_c, is computed (by using eqs. 1 and 2). As we can see f_i represents how faster is GPU$_i$ w.r.t. a core.

Let us assume that T represents the optimal time span during which the `parallel_for` can be executed in the heterogeneous system. In addition, N_i denotes the number of chunks that GPU$_i$ is going to execute, whereas N_c is the number of chunks per each CPU core. Then, $N = \sum_i N_i + \sum_{nCores} N_c$ is the total number of chunks. Let us also assume that T_i (or T_c) represents the average time that resource i needs to execute chunks of size R_i (or R_c). Our objective is then to minimize the load unbalance in the system. Obviously, the load unbalance due to resource i can be modeled as $N_i \cdot T_i - T$ (or $N_c \cdot T_c - T$). Consequently, the load unbalance due to all resources in the system is the sum of the load unbalance due to each resource. This is the objective function that we want to minimize, as shown in eq. 4. Furthermore, the constraint shown in eq. 5 limits the maximum throughput that can be achieved, $\lambda_{max} = \sum_i \lambda_i + \sum_{nCores} \lambda_c$, while the last constraint represented by eq. 6 ensures the positivity of the variables.

$$\text{Minimize} \quad \left(\sum_i N_i \cdot T_i - T\right) + \left(\sum_{nCores} N_c \cdot T_c - T\right) \tag{4}$$

$$\text{such that} \quad \left(\sum_i \frac{N_i}{N} \cdot \frac{R_i}{T_i}\right) + \left(\sum_{nCores} \frac{N_c}{N} \cdot \frac{R_c}{T_c}\right) = \lambda_{max} \tag{5}$$

$$\text{and} \quad \forall i \quad T_i > 0, T_c > 0 \tag{6}$$

This problem has a linear objective function and a non-linear constraint. To solve it, we make a change of variables. Let $\rho_i = 1/T_i$ and $\rho_c = 1/T_c$. Then the problem becomes

$$\text{Minimize} \quad \left(\sum_i \frac{N_i}{\rho_i} - T\right) + \left(\sum_{nCores} \frac{N_c}{\rho_c} - T\right) \tag{7}$$

$$\text{such that} \quad \sum_i \frac{N_i}{N} \cdot R_i \cdot \rho_i + \sum_{nCores} \frac{N_c}{N} \cdot R_c \cdot \rho_c = \lambda_{max} \tag{8}$$

$$\text{and} \quad \forall i \quad \rho_i > 0, \rho_c > 0 \tag{9}$$

The objective function is non-linear, but convex and separable in its variables. The constraint is now linear. This type of constraint is typically called a *resource allocation* constraint. The transformation thus yields a *continuous convex separable resource allocation problem* and the optimal solution occurs when the derivatives of each of the objective function summands $((N_i/\rho_i) - T$ and $(N_c/\rho_c) - T)$ are equal (see [6]). In other words, we have

$$\frac{-N_1}{\rho_1^2} = \frac{-N_2}{\rho_2^2} = \ldots = \frac{-N_c}{\rho_c^2} \tag{10}$$

Changing the variables again, simplifying, and taking the square roots we have,

$$\sqrt{N_1} \cdot T_1 = \sqrt{N_2} \cdot T_2 = \ldots = \sqrt{N_c} \cdot T_c \tag{11}$$

If we assume that, ideally, the load unbalance in the system is 0, then $N_i \cdot T_i = T$ $(N_c \cdot T_c = T)$. Thus, using this assumption for eq. 11 we obtain the following expression

$$T_1 = T_2 = \ldots = T_c \tag{12}$$

As $T_i = R_i/\lambda_i$ and $T_c = R_c/\lambda_c$, eq. 12 can be expressed as,

$$\frac{R_1}{\lambda_1} = \frac{R_2}{\lambda_2} = \ldots = \frac{R_c}{\lambda_c} \tag{13}$$

This expression gives us the key to achieving an optimal strategy for minimizing the load unbalance in the system: *each time that a chunk is partitioned to be assigned to a resource, its size should be selected such that it is proportional to the resource's effective throughput.* However, the CPU cores are the only resources that can keep their throughput constant independently of the chunk size assigned to them. In contrast, the GPUs quickly degrade their throughput when chunks of granularity different to the device's optimal size are assigned to them [3]. Therefore, we have decided to implement a greedy partitioning algorithm based on the following key observations:

While there are sufficient remaining iterations, the chunk size assigned to a GPU$_i$, R_i, should be its optimal GS_i, whereas the chunk size assigned to a CPU core should verify eq. 13, that is,

$$\frac{R_c}{\lambda_c} = \frac{R_i}{\lambda_i}, \quad \forall i = 1:k \tag{14}$$

Using the computational speed definition, f_i, given by eq. 3, we obtain as our optimal goal,

$$R_c = \frac{R_i}{f_i}, \quad \forall i = 1:k \tag{15}$$

Next, we discuss the details of how we implement this result in our framework.

4.3.1 Partitioning heuristic

This heuristic aims to adapt automatically the chunk size of CPUs and GPUs along the computation using the strategy explained in the previous section. The `get_GPU_range()` partitioning function is shown in Fig. 8. This function is responsible for partitioning the chunks for the GPU devices.

```
get_GPU_range()
// Input: r (the input range)
//        myStreamGPU.device (the id of GPU_i)
// Output: R_i (the new chunk for the GPU_i device)
//         r (the remaining iterations)
1. If first_range[i] then
2.     R_i = GS_i;
3.     first_range[i]= false;
4. else
5.     If ( GS_i/f_i < (r - GS_i)/((sum_{j≠i} f_j) + nCores) ) then
6.         R_i = GS_i;
7.     else
8.         R_i = 0;
9.         f_i = 0;
10.        myStreamGPU.device=NULL;
11.        nCores = max(nCores + 1, nThreads);
12.    endif
13. endif
14. r = r - R_i;
15. return(R_i)
```

Fig. 8 Pseudo-code for the `get_GPU_range()` function

For the first invocation of function `get_GPU_range()`, `first_range[i]` is true, so the function returns the range of iterations R_i for GPU$_i$, as shown in line 2 in Fig. 8. After the initial execution of a chunk on each computational resource, and the computation of λ_i and f_i for each GPU device, the next time that `get_GPU_range()` is invoked, a new range of iterations R_i for GPU$_i$ is calculated by following the procedure shown in lines 5-12 in Fig. 8. Specifically, line 5 checks whether there are a sufficient number of remaining iterations. The condition in that line is equivalent to the next equation (just replace f_i by λ_i/λ_c),

$$\frac{GS_i}{\lambda_i} < \frac{r - GS_i}{(\sum_{j \neq i} \lambda_j) + nCores \cdot \lambda_c} \tag{16}$$

Here, we check whether the expected time for the execution of the new chunk of size GS_i (the optimal size for GPU$_i$) in GPU$_i$ is smaller than the expected execution time of the remaining iterations ($r - GS_i$) when they are executed among all the other computational resources (including all the CPU cores and excluding GPU$_i$). We estimate these times using the aggregated effective throughput of all computational resources excluding GPU$_i$ ($(\sum_{j \neq i} \lambda_j) + nCores \cdot \lambda_c$). In case that the expected time for the execution of that chunk in GPU$_i$ is smaller than $r - GS_i$, we guarantee that such a GPU device will not be loaded when the other computational resources have finished. In this situation, we assign the optimal chunk size to the device, $R_i = GS_i$ (line 6). If the condition does not hold, then this is because there is an insufficient number of remaining iterations to keep all the resources busy. In this situation, it is better to not assign a new range to GPU$_i$ because then this device would create load unbalance. Thus the chunk size is set to 0[2] (line 8). In addition, the device is disabled as a computational resource in the system, by making $f_i = 0$ (line 9) and nullifying the device id (line 10). Also, the number of computational CPU cores is increased (line 11) to indicate that one additional thread (the one that has found `myStreamGPU.device`=NULL) will just perform CPU work from now on.

```
get_CPU_range()
// Input: r (the input range)
// Output: R_c (the new chunk for the CPU core)
//         r (the remaining iterations)
1. If first_rangeCPU then
2.     R_c = max_i(GS_i)/nCores;
3.     first_rangeCPU= false;
4. else
5.     R_c = min( max_{∀f_i≠0}(GS_i/f_i), r/((sum_i f_i) + nCores) );
6.     If (R_c < threshold) then
7.         R_c = min(threshold, r);
8.     endif
9. endif
10. r = r - R_c;
11. return(R_c)
```

Fig. 9 Pseudo-code for the `get_CPU_range()` function

Function `get_CPU_range()` that is responsible for partitioning chunks for the CPU cores, is shown in Fig. 9. The first call to this function returns the range R_c to each CPU core as indicated in line 2. After the initial execution of a chunk on a CPU core and the computation of λ_c, the next time that `get_CPU_range()` is invoked, a new range of iterations R_c is calculated by following the procedure shown in lines 5-8 in Fig. 9. Line 5 now selects the value of R_c depending on two options:

(1) $max_{\forall f_i \neq 0}(GS_i/f_i)$, and

[2] When making the GPU chunk size equal to 0, the second filter of our engine will not execute `operator_GPU()`.

(2) $r/((\sum_i f_i) + nCores)$.

GS_i/f_i represents the optimal number of iterations that a CPU core must perform to consume the same time as GPU_i if the device is active (i.e. $f_i \neq 0$), as we established in eq. 14. In our case, we choose to synchronize a CPU core with the GPU for which the GS_i/f_i value is the largest. This is done to minimize the number of times that the partitioning function must be invoked when computing the CPU chunk, and therefore to minimize its associated overhead. The term $r/((\sum_i f_i) + nCores)$ represents the number of iterations (from the remaining ones) that a CPU core should execute when considering the computational speed of all the active devices in the system. In other words, it represents the number of iterations that a CPU core should execute when seeking a weighted *guided self-scheduling* load balancing strategy [13]. When the number of remaining iterations, r, is sufficiently high, or in other words, there are a sufficient number of remaining iterations, then our strategy will choose the optimal value given by option (1). Eventually, when r is getting low and there is an insufficient number of remaining iterations, then our strategy will choose option (2) (weighted guided self-scheduling).

A threshold value is used in line 6, to guarantee a minimum profitable chunk size. This value will depend on the work per iteration and the overhead of our partitioner.

5 Experimental Results

In this section we conduct a series of experiments to evaluate issues such as the overhead of our framework, the efficiency of the two partitioning strategies proposed, and to what extent their performance is less than optimal. We also explore whether or not it is possible to improve performance by allowing oversubscription and by selecting the appropriate synchronization mechanism.

5.1 Experimental setup

We conduct our experiments on a multi-CPU with a quad-socket eight-cores Intel(R) Xeon(R) X7550 2GHz (32 cores). Four decoupled GPU devices are connected: GPU_1 and GPU_2 are GeForce GTX 480 while GPU_3 and GPU_4 are part of a Tesla S2050. This allowed us to study the scalability of the proposed strategies under different heterogeneous configurations. We refer to the configurations as (no. of CPU cores, no. of GPUs). For instance, for 1 socket (8 cores) and 1, 2 and 4 GPUs we get the configurations (8,1), (8,2) and (8,4). The codes were compiled with icc 11.1, TBB 4.1 and CUDA 4.2.

In our experiments we considered just one stream per GPU device. Therefore, every time that a G_token (or GC_token) was selected by $Filter_1$ in our pipelined engine (see section 4), then the corresponding thread would serve as a host thread of the GPU device. In other words, depending on the number of GPUs, there could be at most 1, 2 or 4 threads working as host threads.

5.2 Benchmarks

We use a kernel similar to `MxV` (although with more operations inside the loop nesting) and the `Barnes-Hut` benchmark for our experiments. The `MxV` is an example of a regular data parallel application. An input matrix of 800,000 x 2,000 elements was considered for the `MxV` benchmark. For this input, the benchmark can be considered a fine-grained application (it takes less than 1 ms to process one row -or iteration of the outer loop- on a CPU core). Also, for this problem, the computational speed of the GPUs was within the range $7 \leq f_i \leq 8$ (where i represents one of the 4 GPU devices id).

For the `Barnes-Hut` benchmark, we adapted the code proposed by Kulkarni et. al [7], which is part of the Lonestart Benchmarks suite. This code is representative of an irregular application. An input set of 100,000 bodies was simulated in our experiments. For this input, the problem could be considered a coarse-grained application (it takes a few seconds to process a body/iteration in a CPU core). For this problem, the computational speed of the GPUs was within the range $3 \leq f_i \leq 4$ (again i represents one of the 4 GPU devices id).

5.3 Characterization of the parallel_for template

For all the experiments conducted in this section the number of OS threads considered (the *nthreads* parameter in the initialization of the task scheduler) was equal to the number of CPU cores tested on each machine configuration: 8, 16 and 32.

In our first set of experiments we measured the effect that factor α (used to compute the exponential moving average of the throughput) has on performance. We studied the execution times obtained for both partitioning strategies, $NCHT$ and CHT, when parameter α varies from 0.1 to 0.9, A low α value means that the current throughput sample has less weight than the historic throughput value when computing the new average, whereas a high α value means the opposite. For both `MxV` and `Barnes-Hut` codes, we found that in the case of the $NCHT$ partitioning strategy, the value chosen for parameter α has no effect on performance for

any machine configuration. In contrast, in the CHT strategy a low value of α clearly degrades performance, specially when the number of CPU cores is high. In general, a value of $\alpha = 0.5$ produces the best performance for all machine configurations (higher values of α tend to give similar execution times). Therefore, a value of $\alpha = 0.5$ was selected for the remaining experiments. For this value of α, our partitioning heuristic quickly converge to the optimal core chunk size (after 3-4 partitions).

We also measured the overhead introduced by our engine, finding that for the `MxV` benchmark it was between 0,001% (8 cores) and 0,01% (32 cores). For the coarse-grained `Barnes-Hut` benchmark it was even smaller. This allowed us to set $threshold = 1$ for all our experiments.

5.4 Efficiency of the partitioning strategies

In this set of experiments we focus on discussing and comparing the performance of $NCHT$ and CHT. As previously, the number of OS threads was equal to the number of CPU cores. We start by measuring the improvement achieved by the $NCHT$ and CHT when including the GPU devices on different socket configurations (first we consider a multicore of 8 CPUs, then other of 16 CPUs and finally a multicore of 32 CPUs). In this study, we compute the ratio between the execution time of each benchmark in one multicore configuration ($T(nCores)$) and the time when adding 1, 2 and 4 GPUs ($T(nCores{+}nGPUs)$). This ratio is named *GPU improvement ratio* and it is shown in Fig. 10 for both partitioning strategies. Obviously this ratio represents the speedup that each partitioning strategy achieves when we incorporate 1, 2 and 4 GPUs to a multicore. We also show the ideal improvement ratios, which are computed as $GIR = (\sum_i f_i + nCores)/nCores$. These ideal ratios represent the maximum computational speed of the heterogeneous system vs the speed of a multicore, or in other words, the maximum speedup we can achieve when we incorporate the acceleration of the GPU devices to the multicore. The ideal ratios are depicted as green stars in the figure.

Fig. 10 shows that the CHT strategy always outperforms $NCHT$, obviously due to a better use of the CPU core where the host thread runs. The results show us that the GPU improvement ratio is more significant when the number of CPU cores is small, for both codes. In both cases, when increasing the number of CPU cores, the relative benefit of CHT decreases. Another interesting finding is that when the number of CPU cores is fixed, the relative benefit of CHT is boosted when the number of GPUs increases. For instance, for the `Barnes-Hut` benchmark, the CHT strategy enhances the $NCHT$ performance of the (8,1), (8,2) and (8,4) configurations by 4%, 8% and 14%, respectively. Clearly, more GPUs means more host threads that can take advantage of their respective CPU cores in CHT.

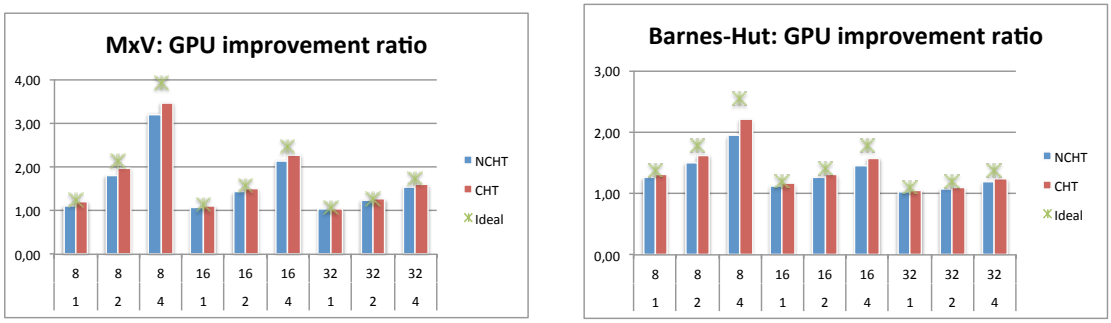
Based on the results shown in Fig. 10, we can also explore another interesting question: how far is our partitioning heuristic from the ideal case? For it, we compare the GPU improvement ratio with the ideal GIR. From the figures we notice that ratios for $NCHT$ and CHT are 5%-20% and 2%-11% below the ideal ratio, respectively. These ranges are valid for both `MxV` and `Barnes-Hut`. The maximum deviation from the ideal value is for the configuration with the highest number of GPUs and lowest number of CPU cores: (8,1). The loss of efficiency in the $NCHT$ strategy is because the CPU core that runs a host thread is underutilized. This is alleviated in part by the CHT strategy, which attempts that the host thread uses the CPU core by collaboratively executing sub-ranges of work while the GPU is computing the chunk assigned. However, in this case, there is still some loss of performance due to the latency in the synchronization mechanism that we study next.

5.5 Effect of oversubscription and synchronization mechanisms

In this section we discuss the effect of oversubscription as well as the different synchronization mechanisms available using CUDA on our partitioning strategies.

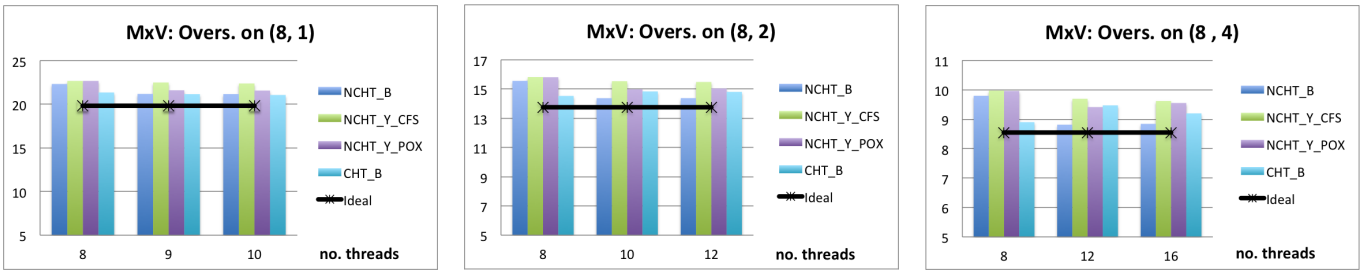
Oversubscription may improve core utilization, especially in the case of the $NCHT$ strategy, but it can also produce more overhead due to higher process context switch. Also, increasing the number of threads has the potential to increase the duration of synchronization operations due to hardware contention. CUDA controls how the host thread interacts with the OS scheduler when waiting for results from the GPU device by calling the `cudaSetDeviceFlags()` function. As explained in section 2, we are not interested in the busy waiting mechanism (*Spin*) because it wastes the host thread without performing any productive computation. For this reason, we have studied the *BlockingSync* (from now *Blocking*) and $Yield$ flags.

Figs. 11 and 12 present the execution time for all the codes when they are executed on a 8 CPU cores socket and with 1, 2 and 4 GPUs. The ideal time is also represented: it is estimated as $T(nCores = 8)/((\sum_i f_i + nCores)/nCores)$, where $T(nCores = 8)$ is the time of each application in the 8 CPU cores socket without GPUs. The x-axis represents the number of threads and the y-axis the time in seconds. In each figure, the first



(a) `MxV` (b) `Barness-Hut`

Fig. 10 Ratio of the $NCHT$ and CHT times in a multicore vs the times in a heterogeneous configuration. Note that 1 is the performance in the multicore (only CPUs). The x-axis represents the number of CPU cores (8,16,32) and number of GPUs (1,2,4) on each heterogeneous configuration



(a) (8,1) configuration (b) (8,2) configuration (c) (8,4) configuration

Fig. 11 Execution time (in seconds) for different numbers of threads in the `MxV` code. The left-most group of bars represents the case of no oversubscription

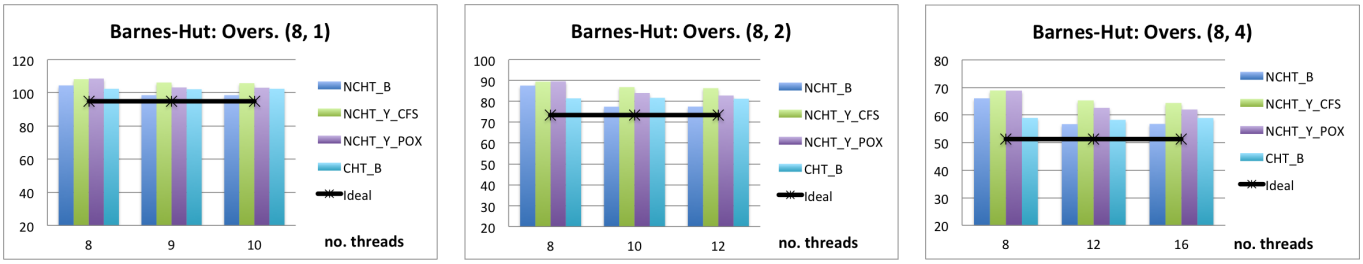
group of bars always present the 8-threads case, i.e. no oversubscription. The next group of bars present the time for a moderate oversubscription scenario, i.e., 8 threads plus one additional thread per GPU device (i.e. 9, 10 or 12 threads confined in 8 cores). Finally, the last group of bars presents the time for a high oversubscription scenario, i.e., 8 threads plus two additional threads per GPU device (i.e. 10, 12 or 16 threads confined in 8 cores).

For the `MxV` and `Barnes-Hut` codes, `NCHT_B` and `NCHT_Y` represent the times for the $NCHT$ strategy in which the *Blocking* and *Yield* mechanisms are evaluated. By setting the *Yield* flag, the system call `sched_yield` is invoked when reaching the synchronization function. Current Linux distributions allow two different behaviors for this system call. The default behavior (referred to as `CFS`) does not preempt the calling thread until its quantum expires, whereas in the Posix conforming implementation (referred to as `POX`) the caller immediately relinquishes the CPU. In addition, `CHT_B` represents the times for the collaborative CHT strategy[3] in which the *Blocking* scheme is assessed. In this CHT strategy, worst times were obtained when a *Yield* mechanism was used.

As shown in Figs. 11 and 12, when there is no oversubscription, `CHT_B` presents the best performance when compared with any `NCHT_` version. Also in this scenario of no oversubscription, the *Yield* mechanism performs worse than the *Blocking* one under the $NCHT$ strategy. This difference is more evident in the (8, 4) configuration (see Figs. 11(c) and 12(c)). We discovered that the reason for this performance difference is the Turbo-Boost feature of the Xeon X7550 processor. This feature enables boosting the frequency of heavily-loaded cores when other cores are idle in the same package. Fig. 13 shows the average frequency obtained for the `MxV` and `Barnes-Hut` codes for the $NCHT$ strategy and the *Blocking* and the *Yield* (default CFS) mechanisms. Clearly, on average, the cores are running at a higher frequency when the *Blocking* strategy is used.

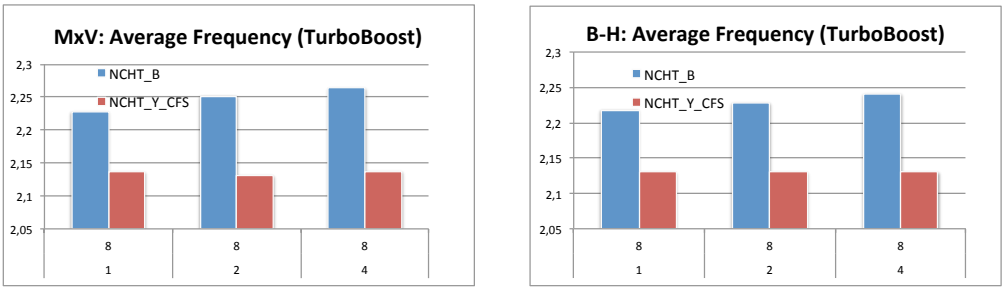
Interesting results are obtained when we study the performance of our two partitioning strategies under oversubscription. For instance, the performance of the CHT strategy degrades always for any machine configuration when moderate or high oversubscription is allowed in the system. The conclusion in this case is that oversubscription does not improve core utilization,

[3] Let's recall that a polling mechanism is used in the CHT strategy when the host thread checks the completion of the GPU work, although eventually also a `waits()` call is performed, and this is the function affected by the synchronization flag.



(a) (8,1) configuration (b) (8,2) configuration (c) (8,4) configuration

Fig. 12 Execution time (in seconds) for different numbers of threads in the `Barnes-Hut` code. The left-most group of bars represents the case of no oversubscription



(a) `MxV` (b) `Barnes-Hut`

Fig. 13 Average frequency (GHz) that the Turbostat command reported for our benchmarks. No oversubscription case: 8 threads on (8,1), (8, 2) and (8,4) configurations

and in fact context switch and cache cooling overheads degrade the times. On the other hand, although the non-collaborative strategy, $NCHT$, shows worse behavior in the case of no oversubscription, it improves its performance when oversubscription is enabled, and for both *Blocking* and *Yield* mechanisms. The improvement of $NCHT$ with oversubscription is more significant when the number of GPUs increases. This is due to the fact that core utilization is improved by allowing extra threads with CPU chunks to execute additional work in the CPU core that is waiting for the completion of the GPU work computation. We also notice from the figures that the `POX` implementation performs slightly better than `CFS` for *Yield*, but in any case, again the *Blocking* mechanism outperforms the *Yield* one.

In summary, we can see that the overall best performance for the `MxV` and `Barnes-Hut` codes is achieved with the non-collaborative $NCHT$ strategy that uses a *Blocking* synchronization mechanism and in moderate oversubscription scenario. In fact, in this case, the time of $NCHT$ is slightly less than 1% above the ideal for `MxV` and around 3%-5% above the ideal for `Barnes-Hut`.

6 Related Work

Current attempts to provide programming support for heterogeneous systems such as CUDA 5 [10], OpenCL [14] and OpenACC [5] consider the portability of code across CPUs and GPUs, but do not provide support for simultaneously executing a parallel construct (for, reduce, ...) on the CPU and the GPU. The problem of accelerating applications on heterogeneous architectures based on coupled and/or discrete GPUs by using the aggregate processing power of the multicore CPU and the multiple GPUs has received some attention lately [9,15,11,1,8,4,3]. The Qiling system [9] has developed an adaptable scheme for mapping the computation between the CPU and GPU by using extensive offline training. However, in our approach, the runtime performs dynamic work distribution using online information (the availability of resources and their effective throughput). Vuduc et al. [15] propose a "wildly asynchronous" implementation that can reduce or even eliminate the synchronization bottleneck between iterations, although for the heterogeneous implementations they did not obtain speedups on their platforms. StarPU [1] and XKaapi [8] offer a runtime for scheduling a DAG of tasks on heterogeneous architectures, providing a programming model that presents an API to select the scheduling policy. In our work, the programming model is based on the use of higher-level templates. OmpSs [4] is a programming tool that provides a set of OpenMP-like pragmas and a runtime system to schedule tasks while preserving dependencies. Although they present performance results for multi-GPU systems, collaborative work from the multicores is not considered, which is a relevant issue in our work. One distinguishing feature of our re-

search when compared to the above mentioned frameworks is that we explore dynamic block resizing to prevent load unbalance of CPUs and GPUs due to small or large block sizes. Other distintive issue in our work is that we study the efficiency of non-collaborative and collaborative host thread strategies combined with the possibility of using oversubscription to improve cores utilization.

To our knowledge, this is the first work where it has been studied the behavior of a task-based runtime working under a competitive environment that allows oversubscription in a heterogeneous architecture. Our results indicate that oversubscription provides an orthogonal mechanism to increase performance in a heterogeneous multi-CPU & multi-GPU system for the benchmarks studied.

7 Conclusions

We have explored the possibility of extending a high-level `parallel_for` template that works under the parallel task programming paradigm to enable the effective utilization of accelerators (GPU devices) working in parallel with multicore systems in heterogeneous architectures. The extension of the template is based on a two-stages pipeline engine that is responsible for dynamically scheduling and partitioning the chunks into the computational resources. Under this engine, we have proposed two adaptive partitioning strategies, $NCHT$ and CHT, that resize chunks to prevent underutilization and load imbalance of CPUs and GPUs due to small or large block sizes. Our partitioning heuristic is based in an analytical model that takes into consideration the effective throughput of the computational resources. CHT also tackles the problem of effectively utilizing the CPU core where a host thread operates, by allowing that the host thread gets one chunk to process in parallel each time that launches work on a GPU device. Using a regular and an irregular benchmark, we have evaluated the overhead introduced by our engine in a heterogeneous platform, finding that is negligible (less than 0.01%). We have also evaluated the behavior and efficiency of both partitioning strategies, finding that a collaborative host thread strategy implemented at the application level (CHT) can be outperformed by a non collaborative host thread ($NCHT$) strategy combined with a blocking synchronization mechanism, when moderate oversubscription controlled by the OS is allowed. Our results encourage the development of strategies that fully utilize the host thread, depending on the available synchronization mechanisms of the host thread: either a $NCHT$-like strategy with moderate oversubscription when blocking policy is available, or a CHT-like strategy without oversubscription otherwise.

References

1. C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. Data-aware task scheduling on multi-accelerator based platforms. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 291–298, Dec. 2010.
2. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, (23):187–198, February 2011.
3. Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, January 2013.
4. J. Bueno, J. Planas, A. Duran, R.M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of GPU clusters with OmpSs. In *Parallel Distributed Processing Symposium (IPDPS), IEEE 26th Intl.*, pages 557–568, May 2012.
5. Alistair Hart. The OpenACC programming model. Technical report, Cray Exascale Research Initiative Europe, 2012.
6. T. Ibaraki and H. Katoh. *Resource Allocation Problems: Algorithmic Approaches.* MIT Press, Cambridge, Mass., 1988.
7. M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*, 2009.
8. J.V.F. Lima, T. Gautier, N. Maillard, and V. Danjean. Exploiting concurrent GPU operations for efficient work stealing on multi-GPUs. In *Computer Architecture and High Perf. Comp. (SBAC-PAD), IEEE 24th Intl. Symp. on*, Oct. 2012.
9. C-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Microarchitecture, 2009. MICRO-42. 42nd IEEE/ACM Intl. Symp. on*, pages 45–55, Dec. 2009.
10. NVidia. *CUDA Toolkit 5.0 Performance Report*, Jan. 2013. https://developer.nvidia.com/nvidia-gpu-programming-guide.
11. V.T. Ravi and G. Agrawal. A dynamic scheduling framework for emerging heterogeneous systems. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10, Dec. 2011.
12. James Reinders. *Intel Threading Building Blocks: Multicore parallelism for C++ programming.* O'Reilly, 2007.
13. David C. Rudolph and Constantine D. Polychronopoulos. An efficient message-passing scheduler based on guided self scheduling. In *3rd Intl. Conf. on Supercomputing*, ICS '89, pages 50–61, New York, NY, USA, 1989. ACM.
14. S.A. Russel. Levering GPGPU and OpenCL technologies for natural user interaces. Technical report, You i Labs inc., 2012.
15. S. Venkatasubramanian and R. W. Vuduc. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *23rd International Conference on Supercomputing (ICS'09)*, pages 244–255, Jun. 2009.