Parallel Processing Letters

A FRAMEWORK FOR EFFICIENT EXECUTION OF DATA PARALLEL IRREGULAR APPLICATIONS ON HETEROGENEOUS SYSTEMS

ROBERTO RIBEIRO
rribeiro@di.uminho.pt

JOÃO BARBOSA
jbarbosa@di.uminho.pt

LUÍS PAULO SANTOS
psantos@di.uminho.pt

Received July 2013
Revised May 2014
Communicated by K. Qiu

ABSTRACT

Exploiting the computing power of the diversity of resources available on heterogeneous systems is mandatory but a very challenging task. The diversity of architectures, execution models and programming tools, together with disjoint address spaces and different computing capabilities, raise a number of challenges that severely impact on application performance and programming productivity. This problem is further compounded in the presence of data parallel irregular applications.

This paper presents a framework that addresses development and execution of data parallel irregular applications in heterogeneous systems. A unified task-based programming and execution model is proposed, together with inter and intra-device scheduling, which, coupled with a data management system, aim to achieve performance scalability across multiple devices, while maintaining high programming productivity. Intra-device scheduling on wide SIMD/SIMT architectures resorts to consumer-producer kernels, which, by allowing dynamic generation and rescheduling of new work units, enable balancing irregular workloads and increase resource utilization.

Results show that regular and irregular applications scale well with the number of devices, while requiring minimal programming effort. Consumer-producer kernels are able to sustain significant performance gains as long as the workload per basic work unit is enough to compensate overheads associated with intra-device scheduling. This not being the case, consumer kernels can still be used for the irregular application. Comparisons with an alternative framework, StarPU, which targets regular workloads, consistently demonstrate significant speedups. This is, to the best of our knowledge, the first published integrated approach that successfully handles irregular workloads over heterogeneous systems.

Keywords: Heteregeneous systems, irregular applications, efficiency, programming productivity

1. Introduction

Heterogeneous Systems (HS) are an emerging trend in today's computing solutions. The high performance computing community has substantially increased the demand for computational power to which manufacturers answered with a range of highly parallel computing devices, including many core CPU architectures, such as Intel MICs, GPUs, DSPs, FPGAs, usually packaged as co-processing expansion boards. Even domestic computing machines are heterogeneous in the sense that they are equipped with one or more GPUs. Leveraging the whole heterogeneous set of computational resources available on a single machine in order to speedup data parallel applications' execution times is thus an obvious course of action.

However, HS pose a number of challenges which might prevent programmers from fully exploiting the available computing power and seriously reduce programming productivity. Typically accelerators have disjoint address spaces among themselves and the host CPU, usually interconnected with a limited bandwidth bus which is a potential performance bottleneck. Despite all the efforts from the manufacturers to elide this problem (such as the CUDA Unified Memory model [1] for NVIDIA GPUs), data transfers must still be explicitly managed and minimized for optimization and efficiency purposes[a]. Different architectures typically exhibit different execution and programming models and are deployed with different programming languages and development tools, severely impacting on both code and performance portability.

In order to maximize performance, an application is designed and carefully tuned to fully utilize each resource computing capability according to the device specific architecture and execution model, seriously reducing development productivity. Additionally, the application's workload has to be distributed and balanced among the multiple devices, and, within each device, among its multiple computing units; this leads to inter- and intra-device scheduling, which must be effectively handled in order to achieve acceptable performance levels.

The above challenges are further compounded if the data parallel application exhibits an irregular behavior. The workload is usually classified as irregular if it involves irregular data structures, irregular control flow or irregular communication patterns [2]. The main consequences of irregularity are load imbalance, code divergence and uncoalesced memory accesses, all potentially resulting on significant performance losses. This hurdle is aggravated in HS due to differences in the architectures and computing capabilities of each device [3]. Yet, irregular applications constitute an important class of algorithms that are present in well-known scientific applications, such as graph and sparse matrix algorithms, n-body simulation, data mining, knowledge discovery, language understanding, decisions problems, optimization theory, system modelling, discrete-event simulation, pattern recognition

[a]CUDA 6.0 Unified Memory capabilities do not provide fully transparent support for data migration within multi-GPU systems, nor automatic concurrent copy and kernel execution.

and meshing among others [2, 4]. A particularly relevant class of irregular applications are Monte Carlo simulations [5], widely used in many knowledge areas, such as financial valuation and option pricing [6] or physically based simulation of light transport within complex media [7, 8], among many others. Monte Carlo simulation entails performing multiple Markov random walks within the function multidimensional domain and then averaging the results of such random walks in order to obtain an estimate of the metric of interest. Since both the directions taken and the length of the random walk are stochastically generated, this results on an irregular workload, exhibiting load imbalances, control flow divergence and irregular memory accesses; two of the case studies analysed throughout this paper are Monte Carlo simulations.

This paper focuses on irregular data parallel applications, which exhibit unpredictable workload and memory access patterns, varying across elements of the data domain. These are in opposition to regular applications, whose memory access patterns and workload can be predicted, mainly because they do not change across data elements. Optimizing irregular applications is a much more challenging task.

The above cited problems are particularly relevant on wide SIMD/SIMT (Single Instruction Multiple Threads) devices, such as the ubiquitous GPUs. The hardware work dispatch units within these devices are optimized for homogeneous regular workloads, maintaining high utilization of SIMD/SIMT lanes and thus exhibiting remarkable performance improvements over CPUs for regular data parallel applications. Irregular applications, however, have the potential to follow different code paths and perform scattered memory accesses within the same SIMD/SIMT lane, resulting on code divergence, increased memory access latencies and resource underutilization. In order to fully exploit these devices, maximum levels of occupancy should be guaranteed, even with irregular workloads. This paper addresses this problem by resorting to persistent kernels and a queuing system, inspired by the results presented in [9, 10].

The adoption of HS has been limited by the lack of programming models and frameworks that hide these challenges and reduce the complexity of addressing such systems. Several such frameworks have been proposed in the specialized literature [11, 12, 13, 14, 15], but none of them is focused on irregular data parallel applications. This paper presents a framework that specifically addresses development and execution of data parallel irregular applications in heterogeneous systems. Our goal is to potentiate efficient utilization of HS while maintaining high programming productivity. This framework integrates the following features:

- a unified task-based programming and execution model for data parallel irregular applications, together with high-level programming abstractions, which increase productivity by introducing transparency with respect to the burden of handling multiple platforms and enable the programmer to focus in the application functionality rather than architectural details [16];
- an application programming interface (API) that supports the above model;

- a scheduling mechanism that transparently partitions the data domain into tasks and deals with all inter- and intra-device workload distribution and balancing;
- a data management strategy that transparently guarantees that required data is readily available on each task's addressable memory space.

This paper's main contributions are the unified execution and programming model and the integration of persistent kernels [9, 10] on the proposed framework as the solution to handle irregular workloads, particularly on what concerns intra-device scheduling. Additionally, a concrete implementation of the framework is presented, together with an experimental assessment of its ability to efficiently handle regular and irregular workloads and a comparison with a competitive approach, StarPU [12]. The proposed framework has been developed and tested on CPU+GPU heterogeneous platforms and emphasis has been put on scheduling irregular workloads within the GPUs; we expect most of the results to apply to other wide SIMD/SIMT platforms. Four case studies are used: a regular matrix multiplication, an irregular n-body problem using the Barnes-Hut algorithm, an irregular path tracing based renderer and an irregular simulation of light transport with fluorescence within multi-layered tissues. This is, to the best of our knowledge, the first published integrated approach that successfully handles irregular workloads over heterogeneous systems.

The next section presents related work, followed by an overview of the programming and execution model. Section 4 details the proposed scheduling techniques, Section 5 describes the experimental applications and setup, while results are presented and analysed in Section 6. The paper closes with some concluding remarks and proposals for future work.

2. Related Work

In this section we discuss some of the proposed programming models and frameworks that tackle the challenges of using parallel heterogeneous systems. We also present some of the efforts done towards the handling of irregular workloads in parallel computing systems.

2.1. *Frameworks for Heterogeneous Systems*

Several efforts have been made towards programming and execution models that provide increased levels of development productivity, while distributing the load across the available resources of an HS. The main goal of Hybrid Multi-core Parallel Programming (HMPP) is to handle devices and use them without the need to re-write the applications [13]. Codelets are introduced as a means to express the application functionality on each device and primitives for execution and data transfers are provided. However, it does not address irregular workloads and lacks a run-time system and scheduling policies that allow for a proper load balancing and full ex-

ploitation of available resources. Diamos [17] identifies some challenges and proposes several techniques to address HS and tackle the associated complexity. The author assesses and validates some solutions presenting preliminary results of a unified execution model combining functional kernels, control decisions and a shared address space [11]. Optimizations are proposed such as performance modelling, static and dynamic optimizations, among others. Merge, proposed by Linderman et al. [14], is focused on portability issues providing a compiler and run-time system and following a map-reduce paradigm for scheduling. The authors advocate that Merge is applicable to different HS and applications are easily extensible and can easily target new architectures. These approaches are focused on the challenges that the multiplicity of architectures pose, such as code portability, and few attention is given to data management, scheduling, load balancing and irregular workloads.

StarPU [12], Qilin [15] and Model Driven Runtime (MDR) [18] are similar HS frameworks that provide high-level programming abstractions, integrated data management and enhanced scheduling mechanisms. StarPU provides a unified execution model combined with a virtual shared memory and a performance model working together with dynamic scheduling policies. The run-time features a data management system that entails several features: automatic work decomposition and data transfers, communication and computation overlapping, data pre-fetching and data locality aware scheduling, among others. While Qilin has some enhanced compiling features and a simpler performance modelling mechanism, StarPU has more advanced data-management and sophisticated scheduling techniques, such as the Heterogeneous Earliest Finish Time (HEFT) approach [19]. MDR focuses on scheduling, proposing a scheduling approach entirely based on online history-based performance modelling coupled with an analytical model for communications. While addressing some of the challenges associated with HS, these frameworks make no attempt to deal with irregular workloads. In particular, intra-device scheduling is not addressed and inter-device scheduling and work decomposition are based on a sampling performance model, which measures the devices performance on a small subset of the data domain and then generalizes for the whole domain – irregular applications are particularly sensitive to these generalizations, since the workload varies among data elements in an unpredictable manner. The data management system used by the framework proposed on this paper is strongly inspired on that of StarPU; it uses the same cache protocol with lazy consistency and keeps the programmer agnostic to data movements. An implementation of StarPU, supporting multi-core CPUs and CUDA enabled GPUs, is publicly available. We compare its performance and that of the proposed approach for an irregular workload – see Section 6.3.

2.2. *Handling Irregular Workloads*

Irregular data parallel workloads require performing some fundamental operation to each data element an unknown number of times; e.g., on a path tracer the length

of the path per pixel, i.e., the number of rays, is unknown and varies unpredictably across screen space – path tracing can thus be seen as tracing a previously unknown number of rays. On current highly parallel SIMD/SIMT devices, such as the GPUs, this irregularity would lead to code divergence and huge resource underutilization. Programmers do explicitly resort to multiple passes to increase processor utilization, with the added cost of a global synchronisation in between passes. Under this model, each pass produces work for the next passes, which will consume it and, eventually, generate even more work. In the literature efforts can be found to transparently map irregular applications to wide SIMD/SIMT devices, balancing the workload across the device computation units (CU) and alleviating the programmer from the need to explicitly deal with this issue. Cederman et al. [20] evaluates the use of dynamic load balancing methods based on queues with lock-free and work-stealing mechanisms. Tzeng et al. [10], improving on the proposals by Aila and Laine [9], introduced a task management system based on persistent kernels and queues, which maximizes CUs utilization and load balance. Persistent kernels produce and consume work using a queuing system, avoiding the multi-pass approach and allowing load redistribution through a task donation/stealing mechanism. Additionally, tasks are scheduled in blocks with the same size as the SIMD/SIMT lane width, allowing for better SIMD efficiency. Our approach for intra-device scheduling is inspired by Tzeng's task management system, integrated on a framework handling multiple heterogeneous devices.

After Tzeng's, other sophisticated approaches have been proposed with the same goal. For instance, Softshell [21] proposes a three-tier scheduling model for the GPU that aims to replace the current built-in scheduling systems. It addresses the major limitations of the GPU proposing an aggregation scheme of threads and work items, sorting work items by priority and using queues to manage work items. However, the actual implementation of Softshell resorts to a persistent kernel approach with a model similar to Tzeng's. We also use this model in order to efficiently address irregular workloads within the GPU.

3. A Framework for Heterogeneous Systems

This section provides a detailed description of the proposed programming and executions models, programming interface and system architecture that tackle the challenges posed when efficiently exploiting heterogeneous systems with irregular applications.

3.1. *Programming and Execution Model*

The proposed framework follows a host-device system model, with applications being composed by a **host control program** (HCP) plus one or more **computation kernels** and respective **data sets**. The HCP typically runs on the CPU and interfaces with the runtime system, being responsible for data registration and partitioning, synchronisation and enforcement of dependency constraints among computing

kernels. Kernels, executed on the system devices (including the multi-core CPU), apply some computation to all elements of a data set; in this sense, kernels express data parallel problems and the application of a kernel to one data element is referred to as a **basic work unit**. Basic work units within the same job are assumed to exhibit no data dependencies among them. It is the programmer's responsibility to provide implementations of the kernels targeted and optimized for each device architecture.

An application consists on one or more **jobs**, each consisting on applying a computation kernel to a data set. The runtime system partitions the job data set into blocks of basic work units, referred to as **tasks**, whose execution is dispatched onto available devices. The data set partitioning and dispatching is transparent to the application programmer and referred to as **inter-device scheduling**. Partitioning is, however, dependent on application specific data representation; the programmer is thus required to provide a callback method capable of creating arbitrarily sized data partitions upon system demand; this method, renders the runtime system independent on data representation. Dependency constraints among jobs must be explicitly specified by the HCP using system primitives, otherwise they may execute concurrently; tasks are executed out-of-order and completely transparent to the application programmer, other than the data partitioning operation.

Data management is handled by the runtime system, transparently to the application programmer, by resorting to **domains**. These, inspired by Partitioned Global Address Space based languages such as Chapel [22], encapsulate all the information required for the system to manage user data, including data location and transfers. This information is provided upon domain registration by the HCP and the data partitioning method. Hierarchic data partitioning is internally supported by a hierarchy of sub-domains, which represent smaller regions of the data set. Kernels are agnostic to sub-domains and respective sizes, since the runtime system converts domain global indexes to task local sub-domain indexes, thus transparently supporting arbitrarily sized tasks; only the notion of domain is exposed to the programmer. Results gathering in-between jobs has to be explicitly triggered by the HCP in order to avoid useless data transfers. Furthermore, the runtime system does not ensure data consistency among concurrent jobs, i.e., if different jobs update the same data, then they must be explicitly serialized by the HCP using system primitives. The data management system uses a MSI cache coherence protocol, similar to StarPU [12], to enable data replication and ensure consistency among replicas, which combined with lazy data transfers reduces data movement overheads. Data pre-fetching and overlapping of asynchronous data transfers with computation are also supported to further reduce communication overheads.

In order to effectively handle both regular and irregular workloads the runtime system supports two types of kernels: **consumer** and **consumer-producer** kernels. Consumer kernels imply the complete processing of a data element. These are well suited for regular workloads, since imbalances among basic work units within the

same task are unlikely. Consumer-producer kernels, inspired on Tzeng's persistent kernels [10] (see also Section 2), deal with the fact that irregular workloads imply highly unbalanced computational and memory demands across data elements. On wide SIMD/SIMT architectures this would result on underutilization of the devices' CUs. Building on the fact that irregular data parallel workloads consist on performing some fundamental operation to each data element an unknown number of times, the basic work unit is now defined as this fundamental operation, rather than the complete processing of a data element.

A consumer-producer kernel applies this basic work unit to a data element and, if required by the algorithm, dynamically generates a number of new basic work units, which are then rescheduled within the device by resorting to a queuing system – this process is referred to as **intra-device scheduling** and allows balancing the irregular workload and increasing resource utilization within each device. Back to the path tracer example introduced in Section 2, a consumer kernel would follow the entire path, eventually leading to imbalances when paths have different lengths; a consumer-producer kernel would follow a single segment of the path, i.e., a ray (and, eventually, associated shadow rays), generating a new basic work unit (a new path segment) at each intersection point until the path finishes. By rescheduling the newer generations of basic work units within the device, imbalances due to the irregularity of the workload can be minimized. It is the responsibility of the application programmer to decide whether a consumer or a consumer-producer kernel is to be used for each job; for a deeper discussion of this subject, please refer to Section 4 and 6.4.

Note that consumer kernels are launched by the runtime system on the host, associated with the respective device context and parameterized with the task to process (i.e., the respective set of basic work units) and the data required to process this task. The consumer kernel allows the application programmer to freely map the task workload onto the device resources, granting it complete control over the device and enabling the use of lower level programming tools, such as CUDA [23], or highly optimized libraries, such as CuBLAS [24] or the Intel Math Kernel Library [25]. Consumer-producer kernels, on the other hand, are under complete control of the runtime system – the latter is in fact running a persistent kernel [10], which calls the consumer-producer kernel, provided by the application programmer, in order to process basic work units – thus precluding the utilization of such third party libraries. Since consumer-producer kernels are intended to wide SIMD/SIMT architectures, these operate over sets of basic work units with the same cardinality as the SIMD lane width. For path tracing on a NVIDIA GPU, for instance, the consumer producer kernel traces a single segment of the path, but 32 instances are simultaneously scheduled and executed in lock-step, as a warp, to trace 32 path segments that have been extracted from the respective task by the runtime system.

3.2. *Programming Interface*

Applications are expressed using a HCP that enables the programmer to use the proposed mechanisms. Code block 1 illustrates the HCP for the path tracer application. Domains for the resulting pixels radiance and for the geometry are created and linked to the corresponding data structures (Lines 6 and 7). A job is then created, associated with the domains, parameterized and associated with the devices architecture specific kernels (Lines 9 to 15). The job is launched, the HCP is told to wait for the job to finish and the computations results, stored onto one domain, are gathered by the host under explicit demand by the HCP (Lines 17 to 19). The HCP is completely agnostic of the number of existing devices and only depends on the existing architectures in the sense that a kernel has to be specified for each.

```
1 HCP_PATHTRACER() {
2   RGB* pixelsRadiance = new RGB[PIXEL_COUNT];
3   Geometry *geometry = new Geometry();
4   (...)
5
6   Domain<RGB>* d_pixelsRadiance = new Domain<RGB> ("RAD", pixelsRadiance,
        dim_space(0, PIXEL_COUNT));
7   Domain<char>* d_geometry = new Domain<byte> ("GEO", geometry, dim_space(0,
        GEOMETRY_SIZE));
8
9   Job_PATHTRACER* t = new Job_PATHTRACER();
10  t->associate_domain(d_pixelsRadiance,d_geometry,...);
11  t->camera = CAMERA;
12  t->SPP = SPP;
13  (...)
14  t->associate_kernel(CPU, &CPU_pathtracer_kernel);
15  t->associate_kernel(GPU, &GPU_pathtracer_cpkernel);
16
17  AddJob(t);
18  WaitForAllTasks();
19  GetDomain(d_pixelsRadiance);
20 }
```

Code block 1: Pathtracer host control program

Code block 2 presents a high level excerpt of a consumer kernel for pathtracing on the GPU. The appropriate domain is gathered from the runtime system, followed by gathering the appropriate basic work unit – on this example this is represented by the first ray the kernel will have to trace and shade (Lines 3 and 7). Then the iterative intersect and shade of the sample path is performed, using Russian roulette to stochastically decide whether the path should continue or not. The result of this basic work unit is then written onto the domain. Code block 3 illustrates a consumer-producer kernel for the same application and device. The main difference is the loop removal since the processing of a sample is now transformed into a sequence of an unknown number of basic units. After intersection and shading, and depending on the result of the Russian roulette, a new basic work unit is created and submitted to the runtime system for scheduling within the device (Line 12). Finally the result is accumulated onto the domain. Together with the data parti-

```
GPU_pathtracer_ckernel(TASK* task) {
  Domain<RGB> pixelsRadiance;
  task->GetDomain("RAD", pixelsRadiance);

  RayHit hit;
  RGB result_rad;
  Ray ray = getRay(task);

  do {
    Intersect(ray, hit, ...);
    continue = ShadeAndRussianRoullete(result_rad,...);
  } while (continue);

  int pixel_id = getPixelID(task);
  pixelsRadiance->at(pixel_id) = result_rad;
}
```

Code block 2: Pathtracer GPU consumer kernel

```
GPU_pathtracer_cpkernel(TASK* task) {
  Domain<RGB> pixelsRadiance;
  task->GetDomain("RAD", pixelsRadiance);

  RayHit hit;
  RGB result_rad;
  Ray ray = getRay(task);

  Intersect(ray, hit, ...);
  continue = ShadeAndRussianRoullete(result_rad,...);

  if (continue) newBWU();

  int pixel_id = getPixelID(task);
  pixelsRadiance->at(pixel_id) += result_rad;
}
```

Code block 3: Pathtracer GPU consumer-producer kernel

tioning method and additional kernels for each supported device architecture, these code blocks illustrate all the functionality the application programmer has to provide in order to be able to execute it on heterogeneous systems and transparently benefit from multi-device data management and dynamic workload distribution and balancing. Automatic extraction of architecture specific kernels from some generic specification is an interesting feature that would further decrease the user provided code; we intend to address it on the near future.

3.3. *System Architecture*

Figure 1 illustrates the run-time system architecture and how the different entities cooperate with each other. All the communication between the application and the framework is done through the API, which is one of the main entities along with the Scheduler, Performance Model (discussed in Section 4.1) and Data Management System. The system has a central job queue from where the Scheduler dequeues jobs upon device request and, using the data partitioning methods and

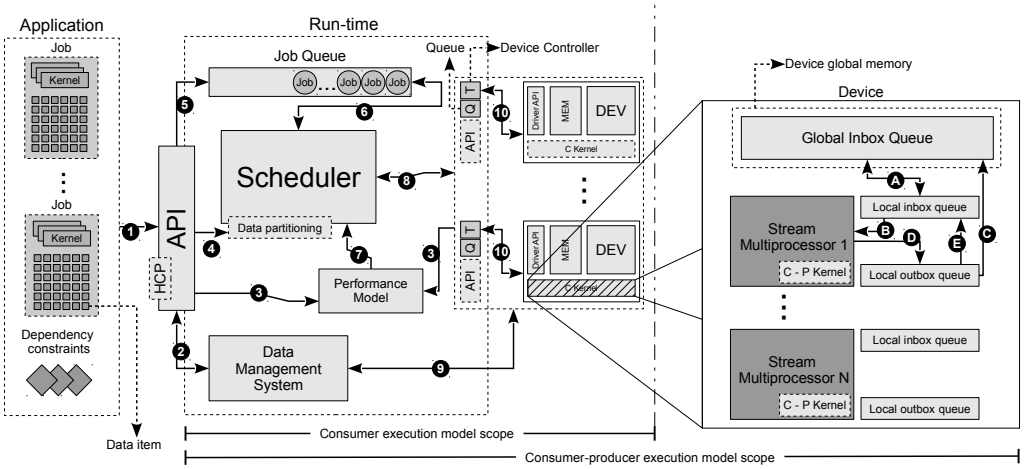


Fig. 1: Framework and persistent kernel architecture and work flow. (1) Application jobs and dependency constraints are submitted to the system by implementing the HCP using the API; (2) through the API, the user will register data in DMS and gather the results back; (3) the number of data elements to process is provided as well as user-provided information to the performance model if applicable; task execution information may also be provided from the devices to the Performance Model; (4) application-specific data partition methods defined; (5) jobs are enqueued in the main queue; (6) the scheduler dequeues and enqueues jobs or tasks from the main queue; (7) the scheduler assigns a job to a device, reasoning about job workload and device compute capabilities which will potentially trigger data partitioning methods to create a properly sized task; (8) task is enqueued in device queue; (9) the device controller signals data movements required for the task; (10) device controller signals for task execution using the user-provided kernel. In an irregular application consumer kernels are replaced with built-in persistent kernels; these use the following task processing model: (A) if space available in local inbox queue (LIQ) try-lock global inbox queue (GIQ) and dequeue tasks; (B) Retrieve tasks and execute them using the user-provided consumer-producer kernel; (C) If there is not enough room in local outbox queue (LOQ) and in LIQ to store all secondary tasks, force GIQ lock and enqueue all the elements from the LOQ; (D) Store generated tasks in LOQ; (E) Enqueue in LIQ elements from LOQ. If LIQ is full try-lock GIQ.

the information provided by the Performance model, produces the proper sized task and assigns it to the device. Each device in the system has its own queue and associated control thread running on the host, enabling asynchronous data and control flow using system messages. This distributed-like system increases scalability since the devices will request work asynchronously and process tasks' data concurrently. The devices' queues support an execution window of tasks enabling computation overlapping with data transfers and data pre-fetching. The rightmost block of this Figure illustrates how consumer-producer kernels are supported through a persistent kernel and how intra-device scheduling is achieved. Note that this requires the runtime system to manage the kernel execution within the device, whereas with consumer kernels the application code has complete control of kernel execution within the device. Each device architecture supported by the framework requires the development of a Device API implementation, allowing the framework to perform low level operations such as initiating computations or copying data to/from the

device. The Device API is transparent to application programmers, but explicitly managed by the framework developers. For instance, a system with three NVIDIA GPUs and two CPUs requires two Device API implementations, one for the GPUs and another for the CPUs. For each of these implementations, alternative programming and execution environments might be selected; for example, either CUDA or OpenCL might be used to control the GPUs.

4. Scheduling

Scheduling is handled by the proposed framework at two different levels: inter and intra-device scheduling. The former partitions the job's workload into tasks and assigns them to individual devices, whereas the latter distributes a given task basic work units among the device computational units (CU). The goal is to minimize execution time by keeping the workload distribution well balanced among computational resources. Even though scheduling is a major component of the proposed framework it is transparent to the application programmer.

4.1. *Inter-device Scheduling*

Inter-device scheduling is performed by resorting to a demand driven strategy, where tasks are assigned upon device request. When a device finishes a task it signals the scheduler, indicating that it is available for further processing. The scheduler then fetches a job, decides the new task size by applying the partitioning strategy described below, applies the data partitioning method to get the proper sized task and submits the task for execution to the requesting device.

Demand driven has been preferred over the HEFT scheduling algorithm [19], which is used by StarPU, since the latter makes its decisions based on an initial sampling of the workload behavior. However, the behavior of irregular workloads is mostly unpredictable by definition and thus we conjecture that HEFT is not appropriate for this kind of workloads. Demand driven has been shown to be able to cope with a wide range of workload profiles and with devices with diverse computing power. By partitioning a job's workload into a number of tasks larger than the number of devices and then assigning tasks on demand it adapts to both the workload requirements and the devices' capabilities.

However, scheduling overheads are also dependent on the number of tasks. A heterogeneous system is expected to have devices with very different computing powers, which would require a large number of tasks in order to maintain load balance, severely impacting on scheduling overheads. The total number of tasks can be reduced by tailoring the task size to the relative computing power of the device where it is being scheduled; this is the responsibility of the work partitioning strategy.

Let C_d represent the computing capability of device d, defined according to some performance model. Results presented on Section 6 are based on the devices' theoretical peak performances, as announced by the respective manufacturers. This

might not be the metric that guarantees the best results, particularly for irregular workloads. However, it is beyond the scope of this paper to select and evaluate the most appropriate performance modelling technique. In fact, the proposed framework takes a modular approach towards the performance model, allowing it to be replaced without impacting on the remaining runtime system architecture. This modularity assures that more efficient performance models and appropriate metrics, eventually resorting to dynamic approaches, can be used in the future. C_d is normalized according to Equation 1 to represent relative computing capability with respect to the other devices present on the heterogeneous system. $T_{devices}$ is the total number of computing devices.

$$\bar{C}_d = \frac{C_d}{\max\left(C_1, ..., C_{T_{devices}}\right)} \tag{1}$$

The size of the task to assign to the requesting device, expressed in terms of the number of data elements to process (or basic work units), is then given by Equation 2

$$\frac{N}{dd} \times \bar{C}_d \tag{2}$$

where N is the job's total number of basic work units and dd is a system constant that allows control over the tasks' granularity, assuring that the total number of tasks is significantly larger than the number of devices, as required for a demand driven strategy to be able to properly balance the workload. For instance, consider a system composed by a GPU and a CPU where the performance model dictates that the normalized relative compute capabilities are 1.0 and 0.3 to the GPU and CPU, respectively, and let dd be equal to 10. Upon receiving a work request, the scheduler will fetch a job, say with 1000 basic work units (BWU), and assign a task with 100 BWUs if the requesting device is a GPU or with 30 BWUs if it is a CPU.

4.2. *Intra-device Scheduling*

Intra-device scheduling, as described on this section, applies only to consumer-producer kernels and irregular workloads; for consumer kernels the application programmer has complete control over the device and is able to freely map the task workload onto the device's CUs, as described on Section 3. Intra-device scheduling exploits the fact that irregular workloads can be seen as applying some fundamental operation to each data element an unknown and unpredictable number of times; the basic work unit is thus redefined as this fundamental operation, rather than the complete processing of a data element. This view enables a work-spawn strategy where the execution of a basic work unit leads to the potential spawning of one or more dynamically generated new basic work units. In order to efficiently handle this mechanism within a SIMD/SIMT device we implemented a generic pipeline that features most of the techniques proposed by Tzeng et. al [10].

A GPU is a SIMT device that schedules bundles of threads with the same cardinality as a SIMD lane – on NVIDIA GPUs these bundles contain 32 threads and

are referred to as warps. Since warps are executed in lockstep, code divergence and uncoalesced memory accesses should be minimized for performance maximization. However, irregular applications tend to exhibit divergence and unpredictable memory accesses. The execution model of our persistent kernel follows a SIMD lane programming approach that cooperates with the hardware scheduler to manage these lanes – Figure 1. Each lane is endued with two local queues for getting work to consume and to store locally generated new basic work units, respectively Local Inbox Queue (LIQ) and Local Outbox Queue (LOQ). Work is shared among different SIMD lanes by using a device Global Inbox Queue (GIQ) with a try-lock mechanism to avoid contention. Each lane will fetch a bundle of 32 basic work units (on NVIDIA GPUs) and call the consumer-producer kernel, using a callback mechanism, in order to process all fetched basic work units. Dynamically generated basic work units are stored on the LOQ and eventually moved to the GIQ in order to allow execution on other SIMD lanes. This enables transparent access to SIMD lane programming and intra-device scheduling by the application programmer, which is now able to maximize application code convergence and coalesced memory accesses assuming that all 32 basic work units will be executed within a single lane.

5. Experimental Setup

This section presents the applications used to evaluate the effectiveness of the proposed model and associated framework. These include a regular application (matrix multiplication (MM)) and three irregular applications (a Barnes-Hut n-body simulation (BH), a path tracer (PT) and a Fluorescence simulation (FL)). The computing system used to obtain the results analysed in Section 6 is also described.

5.1. *Applications*

Being a regular application, only the consumer kernel is provided for the matrix multiplication. In order to compute an element C_{ij} of the result matrix, the kernel performs a dot product between the row A_i and column B_j of the factor matrices. The kernel uses the CuBLAS and the Intel Math Kernel Library optimized libraries for the GPUs and CPUs kernels, respectively. A reference version executing on a single GPU was developed with CuBLAS for performance comparison purposes.

The Barnes-Hut algorithm [26] casts an n-body simulation as a hierarchical problem, reducing its complexity to $O(N \log(N))$. The goal is to compute the force exerted on each particle of the data set by all other particles of the same set. The BH algorithm orders the particles by resorting to an octree (in 3 dimensions). When computing the resulting force, if a voxel is farther away from the particle being processed than a given threshold, then all the particles contained in that voxel are approximated by their center of mass and the sub-tree associated with the voxel can be pruned. The unpredictability of which nodes of the octree will be visited for each particle renders the workload irregular. A consumer kernel will, for each

particle in the data set, traverse the octree, deciding which nodes to visit and which to prune and finally computing the resulting force – the basic work unit is thus computing the force for one particle of the data set. A consumer-producer kernel entails visiting one node of the octree and deciding which of its children to visit and which to approximate. All those children nodes that have to be visited result on the generation of new basic work units, which will be rescheduled within the device by the runtime system. On wide SIMD/SIMT devices, such as the GPUs, basic work units will be executed in groups with the same cardinality as the SIMD lane width (32 for current NVIDIA GPUs). In order to increase coherence within each SIMD lane, particles are initially sorted such that neighboring particles have high probability of being scheduled onto the same SIMD lane [27]; neighboring particles have high probability of visiting the same regions of the octree.

Monte Carlo Path Tracing (PT) is a well known ray tracing based rendering algorithm. It entails following light paths from the eye into the scene; at each intersection point radiant flux is gathered from the light sources using a given number of shadow rays and the continuation of the path is stochastically decided using Russian Roullette; if continued, a new ray is spawn, its direction being stochastically determined. The Russian Roullette path termination approach and the stochastic direction of each new ray render the workload irregular. On wide SIMD/SIMT architectures, coherent path tracing [28] is used, where the random numbers used to decide about path termination and new ray direction are the same for all threads within a SIMD lane. This will make paths within the same SIMD lane coherent (same length, same overall directions), which results on perceivable image artifacts; these artifacts are eliminated by shuffling the paths on the image plane before tracing them, thus avoiding spatial neighborhood among coherent paths [28]. A consumer kernel entails processing the whole path, whereas a consumer-producer kernel processes a segment of the path, i.e., one ray plus associated shadow rays and, if the path is continued, generates a new basic work unit with the new ray. The image plane is divided into multiple pixels and in order to increase image convergence multiple samples (i.e. light paths) are taken per pixel (SPP). Each sample is processed independently and the more samples, the better the image convergence, but the workload increases and more irregular paths are processed. We will use the SPP parameter to express the workload size as it is one of the parameters with major impact in image rendering and also impacts algorithm irregularity, which we propose to address. Our basis pathtracing code was extended from the SmallLux renderer [29]; a reference version of SmallLux running on a single GPU is used for performance comparison purposes.

The Monte Carlo simulation of light transport with fluorescence in multi-layered tissues (FL) is frequently viewed as a reference method, whose results can be used to validate other less demanding methods [8]. It is based on following a packet of photons along random walk steps within a multi-layered media with complex structure, the size of each step being stochastically generated according to the media optical properties. After each step a fraction of the photon packet's energy is absorbed and

a new step and scattering direction are stochastically chosen according to the current tissue layer properties. When a boundary between different layers, or between a tissue layer and the exterior, is crossed by the packet it might be either entirely transmitted into the new layer or reflected back into the same layer; this decision is once again made by resorting to a stochastic process and the optical properties of both layers. The random walk is continued until the photon packet exits the tissue or its termination is decided by Russian Roullette. Fluorescence emission is simulated by deciding, after each step, whether a fraction of the absorbed energy, as given by the quantum yield optical property of the tissue layer, is re-emitted as a new fluorescent photon packet with a different wavelength; this decision is made by resorting to Russian Roullette. Fluorescent photon packets are propagated through the media using Monte Carlo simulation, with the same algorithm as the original excitation packets, except that they will not generate further fluorescent packets since their wavelength will not trigger this phenomenon. The basic work unit for a consumer kernel entails simulating all steps of a photon packet and respective fluorescent packets until they exit the media or are terminated by the Russian Roullette process. The consumer-producer kernel processes a single step of a photon packet random walk; a new basic work unit is created if the random walk is continued and an additional one is created for each emitted fluorescent packet.

Even though PT and FL resort to Monte Carlo simulations, the associated workloads exhibit some fundamental differences. The former entails tracing rays through the scene 3D volume, which is a computational expensive procedure, whereas the latter does not involve any tracing. In fact, FL just requires advancing the photon packet position along the random walk step direction; boundary crossing among layers is verified by checking the Z coordinate, since the modelled layers are aligned with the XY plane and thus all boundaries are perpendicular to the Z axis. Consequently, the basic work unit for the consumer producer kernel involves much less computation for FL than for PT. Additionally, for FL all photon packets are shot into the media through the same infinitesimal point, i.e., all random walks have the same origin. This is in contrast with the PT application where all paths initiate at different points of the image plane. This particularity hinders the application of coherence increasing techniques, such as the coherent path tracing technique used for PT. On wide SIMD/SIMT architectures, and for the consumer kernel, threads within a SIMD lane are thus expected to be more incoherent for FL than for PT, exhibiting larger code divergence, load imbalance and irregularity of memory accesses; the consumer producer kernel has the opportunity to minimize load imbalances within a device since new basic work units are rescheduled after each random walk step. Furthermore, in FL a photon packet can contribute to any voxel within the grid embedded in the tissue, whereas in PT a path only contributes to the pixel where it is originated: contention in memory writes, which are solved by resorting to atomic operations, is thus much more frequent in FL than in PT. Also, each task in PT requires a number of memory management operations, such as dynamic allocation and data copying, which is not required in FL. This is due to the fact that

each task entails generating a tile of the image plane which is dynamically allocated by each device; such requirements do not exist in FL, where the above referred grid of voxels is only allocated once on each device, given that any thread can write to any voxel and the grid is much smaller than the finely sampled image plane. Such memory management operations represent an implementation penalty that might harm PT's efficiency. Finally, PT basic work units with the consumer-producer kernel have a branching factor of 1, i.e., after tracing a ray in the path if the random walk continues a single new task is generated with the new secondary ray. FL can have a branching factor of 2, since a new fluorescent photon packet can be created; the higher branching factor will impact on the results.

5.2. *Heterogeneous Systems Metrics*

Speedup, $S(p)$, and efficiency, $E(p)$, are two metrics often used to report and analyse the performance of homogeneous parallel systems with p processors. If T_p and T_1 are the execution times of the parallel and uniprocessor systems, respectively, then these are given by Equations 3 and 4. $S(p)$ is a measure of how faster the parallel system is than a sequential one and $E(p)$ constitutes a measure of resource utilization.

$$S(p) = \frac{T_1}{T_p} \tag{3}$$

$$E(p) = \frac{S(p)}{p} \tag{4}$$

The problem with the above metrics is that they are defined for the homogeneous case, where all p processors are identical. Similar metrics have been defined for the heterogeneous case [30, 12] and are used on this paper to analyse the experimental results.

Let W define the workload associated with solving a given problem and T_{dev} be the execution time of that workload on a given device. Then the device's observed computing capacity, C_{dev} for that problem is given by $C_{dev} = \frac{W}{T_{dev}}$. Identically, if the execution time of that workload on a given heterogeneous set D of devices is T_D, then $C_D = \frac{W}{T_D}$. The heterogeneous speedup, $S_h(D)$, relatively to the execution time on some given single reference device ref is then given by Equation 5:

$$S_h(D) = \frac{T_{ref}}{T_D} = \frac{C_D}{C_{ref}} \tag{5}$$

Intuitively, the computing capacity available on the set D of devices is given by the sum of the individual capacities of all devices in D, i.e., $C^*_D = \sum_{i \in D} C_{dev_i} = W \sum_{i \in D} \frac{1}{T_{dev_i}}$. Heterogeneous efficiency can now be defined as the ratio of used computing capacity over the available capacity:

$$E_h(D) = \frac{C_D}{C^*_D} = \frac{\frac{1}{T_D}}{\sum_{i \in D} \frac{1}{T_{dev_i}}} \tag{6}$$

In Section 6.2 we perform a strong scalability analysis (constant problem size, i.e., constant W) by using $E_h(D)$ for different heterogeneous sets of devices D. Equation 6 shows that if, due to algorithmic and implementation penalties, the used computing capacity, C_D, grows at a lower rate than C_D^* , then $E_h(D)$ will become smaller as the number of devices in D increases.

5.3. *Computing System*

The computing system used to assess the proposed framework is equipped with two Intel Xeon CPU E5649, each running at 2.53GHz with **six cores** and 12GB of memory RAM. The platform is also equipped with **a NVIDIA Fermi GTX 480** with 480 CUDA cores and 1.5 GB of memory, plus **two NVIDIA Tesla C2070**, with 448 CUDA cores with 6GB of memory. The code was compiled with the GNU C compiler 4.6 and NVCC compiler, provided by CUDA toolkit 5.5, in a LINUX operating system.

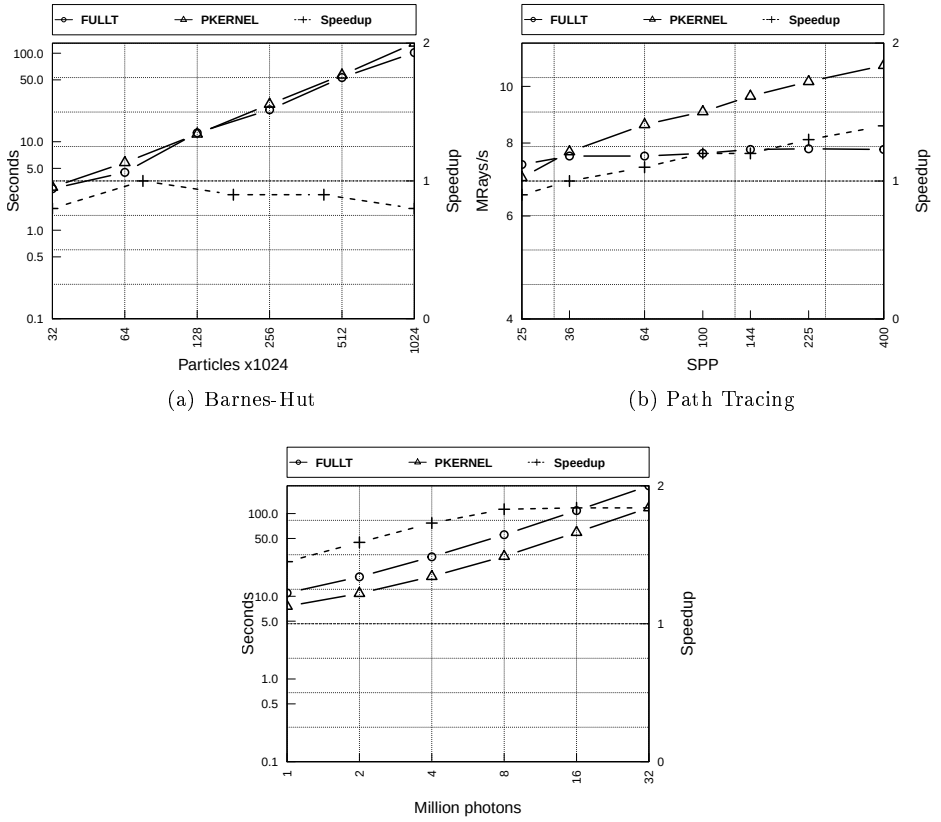
6. Results

This Section presents and discusses experimental results with respect to intra-device scheduling of irregular workloads, performance scalability and a comparison with a state of the art framework – StarPU.

6.1. *Scheduling Irregular Workloads*

Irregular applications imply unbalanced computational demands across data elements, which, on wide SIMD architectures, would result on severe resource underutilization. Consumer-producer kernels are thus proposed as the means to avoid this potential performance penalty. Figure 2 presents performance comparisons for the consumer and consumer producer-kernels, labelled as FULLT (from full traversal) and PKERNEL (from persistent kernel), respectively, for the BH, PT and FL applications with different problem sizes and using a single GPU. Speedup of the consumer-producer kernel over the consumer kernel is also presented in the rightmost axis. Note that Figure 2b depicts PT throughput, expressed in MRays/s, instead of execution time. Throughput will be used throughout this paper for PT because it provides an abstraction to the light transport model details and algorithms' implementation. A further reason to use throughput is that the performance of PT will be compared to a reference path tracing version using SmallLux (Table 3). SmallLux uses a slightly different light transport model that results on tracing different numbers of rays; by reporting rays per second, for the same scene and rendering parameters, performance comparisons can be made.

The consumer-producer approach provides a very significant speedup for both the path tracer (40% better) and fluorescence (84% better), while performing about 20% worse in the BH problem. While the basic work unit for the BH consumer-producer kernel consists on a very light task (deciding, for one node of the octree,



(a) Barnes-Hut

(b) Path Tracing

(c) Fluorescence

Fig. 2: Performance comparison between FULLT and PKERNEL on a single GPU. Note the left-handed y-axis and x-axis in log scale and right-handed y-axis in linear scale.

whether its children have to be visited and computing the resulting force for those that are not), in PT for instance, this is a demanding task, requiring tracing a ray and associated shadow rays as well as shading computations. Our hypothesis is that the workload associated with each BH basic work unit is not enough to compensate the overheads associated with queuing and scheduling the dynamically generated basic work units. In order to verify this hypothesis we added a parameterizable synthetic workload (SW – computing the Fibonacci sequence up to a given index, whenever an octree node is visited) to the Barnes-Hut consumer and consumer-producer basic work units.

Figure 3 depicts the observed speedups for both BH and PT applications – actual values shown in Table 1. Note that in BH, as the SW increases the consumer-producer kernel becomes more effective (maximum of 53% faster according to Table 1a) than the consumer kernel, which corroborates the above cited hypothesis. The PT result also corroborates the above conclusions. As the load per basic

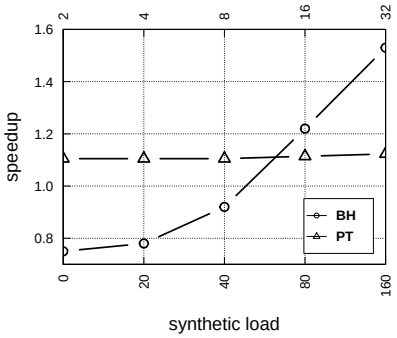


Fig. 3: Load impact in performance, expressed in terms of speedup of the consumer-producer kernel over the consumer one. Number of shadow rays per shading point in PT (upper horizontal axes) and synthetic load for BH (lower horizontal axes). Note that both horizontal axes are in log scale.

Table 1: Speedup values for load impact in performance as we increase workload per BWU in BH and PT.

(a)

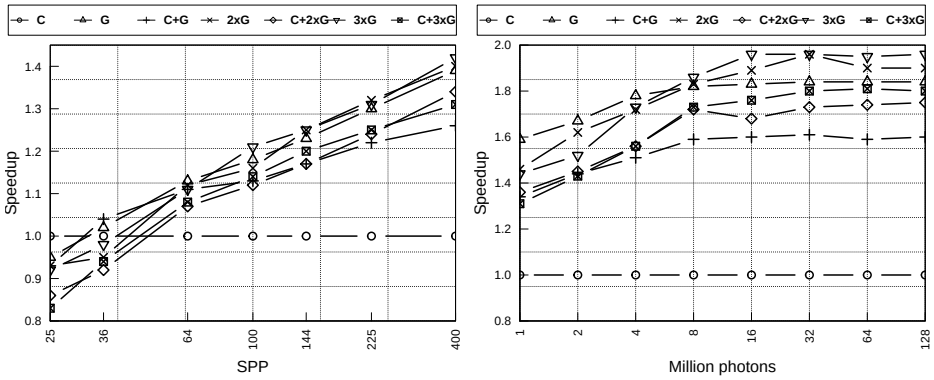
BH	
synthetic load	speedup
0	0.75
20	0.78
40	0.92
80	1.22
160	1.53

(b)

PT	
shadow rays	speedup
2	1.25
4	1.25
8	1.25
16	1.26
32	1.27

work unit increases (expressed as the number of shadow rays cast per shading point to assess the visibility of the light sources), speedup increases although at a marginal rate compared to BH (Table 1b); this is due to the fact that, even with only one shadow ray per point, the load associated with each basic work unit is enough to overcome the overheads associated with the queuing system. Given that without synthetic workload the consumer-producer kernel is not effective for BH, results obtained with this application will not be further reported on this Subsection. BH results with the consumer kernel without synthetic workload will be presented in Section 6.2 to demonstrate that the proposed framework can still effectively handle this kind of workloads.

Figure 4 shows the speedup obtained with the consumer-producer kernel over the consumer kernel for the PT and FL applications with different configurations of multiple heterogeneous devices and for different problem sizes. For a single GPU the curves are the same as in Figure 2. In PT for multiple-device configurations the achieved speedup increases monotonically with the problem size to a maximum of 1.42x with three GPUs and 400 SPPs. As for for the FL case, the speedup increases until a certain workload and then stabilizes with a maximum of 1.96x with three GPUs – see Section 6.2 for a discussion on why is the speedup obtained with FL



(a) Path-tracer (b) Fluorescence

Fig. 4: Performance comparison between consumer kernel and consumer-producer kernel with multiple-device configurations when scheduling PT and FL irregular workloads. C stands for CPU and G for GPU. Note that horizontal axis is in log scale.

significantly larger than that of PT. These results clearly show that the consumer producer kernel provides a clear gain over the consumer approach, and that this gain is sustainable in the presence of multiple heterogeneous devices. Also note that using this multiplicity of heterogeneous devices requires no additional programming effort from the application developer, which is this work's main goal.

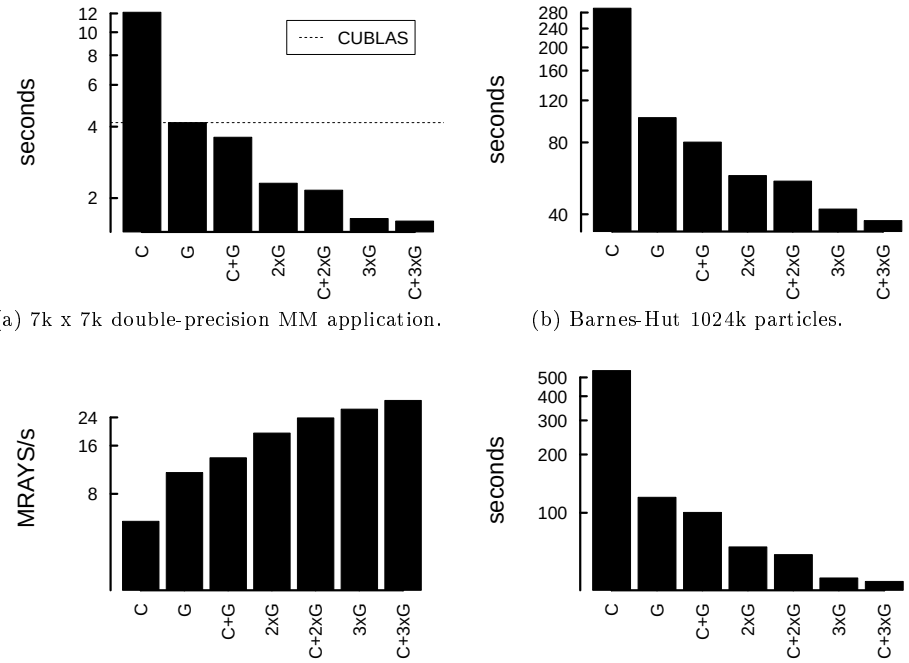
6.2. *Performance Scalability*

The goal of the proposed framework is to allow efficient execution of irregular data parallel applications on HS, while maintaining high programming productivity by hiding from the programmer many of the details associated with such systems; this is achieved by complying with the proposed programming and execution model.

Table 2: Performance values with multi-device configurations. C stands for CPU and G for GPU.

App	Workload	C	G	C+G	2xG	C+2xG	3xG	C+3xG
MM (sec)	7k x 7k doubles	12.14	4.16	3.61	2.31	2.16	1.64	1.60
BH (sec)	1024k particles	291.99	101.60	80.26	58.11	55.09	42.08	37.64
PT (MRays/sec)	400 SPP	5.39	10.86	13.43	19.16	23.82	27.03	30.63
FL (sec)	32M Photons	542.32	120.26	100.47	66.66	60.82	46.09	44.23

Figure 5 presents the performance gain for the selected applications executing on increasing numbers of computational devices – actual values shown in Table 2. Since it is a regular application, the consumer kernel is used in the matrix multiplication. Also, since the consumer-producer kernel is not able to provide performance gains with respect to the consumer kernel for BH, due to the very light workload associated with each BWU, results are reported using the consumer kernel; the goal is to verify whether performance gains are still obtained as the number of heterogeneous



(a) 7k x 7k double-precision MM application.

(b) Barnes-Hut 1024k particles.

(c) Pathtracer 480x480 image with 400 SPP.

(d) Fluorescense modelling with 32M photons.

Fig. 5: Performance with multiple-device configurations. A consumer kernel type is used for the MM and BH applications and a consumer-producer kernel in PT and FL. C stands for CPU and G for GPU. Note the vertical axis in log scale.

devices increases. The consumer-producer kernel is used for the irregular PT and FL applications. Figure 5a clearly shows that the regular matrix multiplication has increased performance as more devices are added. The horizontal dashed line depicts the execution time of the same problem on a single GPU using a reference version developed using CUBLAS (the same library used within the framework provided kernel); there is no performance penalty associated with using this framework for a single GPU and there is a clear gain as more devices are added to the system, since performance scales without any programmer effort (see Tables 2 and 3). With the four devices working together, the runtime system is able to extract about 8x speedup compared to the single (multicore) CPU configuration.

Table 3: Performance values with multi-device configurations compared to a reference version running on a single GPU. PT values differ from Table 2 because a single shadow ray was used per shading point. C stands for CPU and G for GPU.

App	Workload	C	**G**	**Ref (G)**	C+G	2xG	C+2xG	3xG	C+3xG
MM	7k x 7k doubles (sec)	12.14	**4.16**	**4.16**	3.61	2.31	2.16	1.64	1.60
PT	400 SPP (MRays/sec)	5.03	**12.91**	**12.15**	14.42	22.25	25.96	32.23	34.27

Figure 5b clearly shows that the BH application execution time decreases as more devices are added, achieving a maximum 7.6x speedup compared with the CPU configuration (see also Table 2). Remembering that the consumer kernel is being used for this highly irregular application, this result allows us to conclude that consumer kernels can still be used effectively to handle irregular workloads. This is particularly useful when an application would exhibit a very light workload per BWU under a consumer-producer model, insufficient to compensate the associated overheads. In such cases, the consumer kernel can still be used and performance will still increase with the number of devices.

Figure 5c depicts PT throughput, expressed in MRays/s, and clearly shows that performance increases significantly as devices are added to the system (the vertical axis is in log scale). Table 3 compares the achieved performance with that obtained with a reference single GPU path tracer based on SmallLux. Note that the values reported for PT are slightly different from those reported on Table 2 because now a single shadow ray is being shot per shading point, whereas previously several shadow rays were used. It is clear that the proposed approach suffers no performance penalization compared to the reference SmallLux and that ray throughput increases with the number of devices.

Finally, for the FL application we get a similar plot, with performance increasing with the number of devices and achieving a remarkable speedup of 12.26x to the single CPU. These larger performance gains obtained with FL when compared to PT result from the minimal memory management overheads associated with the former (as explained in Section 5.1) and a large gain when using GPUs compared to the CPU (according to Table 2 the GPU is 4.5x faster than the CPU for FL and only 2x faster for PT). A 1.8x speedup can also be observed when adding a Tesla to a GTX480 (the Tesla has one less SM) and 1.45x speedup when adding another Tesla to the GTX480+Tesla configuration (additional tests were performed that reveled a 1.99x speedup from one Tesla to 2xTesla) – overheads associated with increasing the number of devices are thus minimal for the FL case.

Performance scalability is achieved with minimum programmer effort: adding devices with the same architectures only requires registering them through the HCP, while adding devices with different architectures (supported by the framework through the device API) requires providing the respective kernels. Programming productivity is thus preserved, while enabling efficient execution of regular and irregular applications on heterogeneous systems.

In order to measure how effectively the proposed framework uses the resources available on the parallel heterogeneous system a strong scalability analysis is performed using the heterogeneous efficiency metric introduced in Section 5.2. Strong scalability analysis entails studying how the system efficiency varies with the number of devices for a fixed workload (i.e., problem size). Efficiency is expected to decrease with the number of devices, since overheads (such as devices' idleness due to load imbalances, communication and runtime system management costs) increase. However, if efficiency decreases in a very sublinear manner, the system is deemed scalable

for fixed problem size. Ideally, the above mentioned overheads would be measured directly; this is however not possible, since multiple management operations occur concurrently and asynchronously. Efficiency analysis provides thus a robust tool to assess the impact of such overheads.

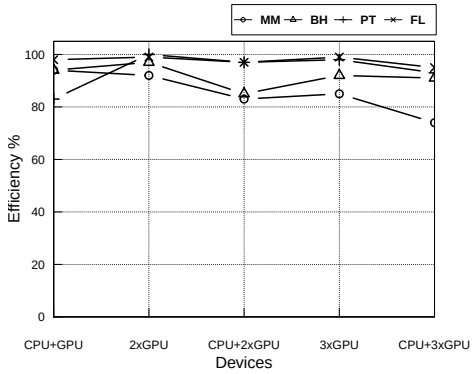


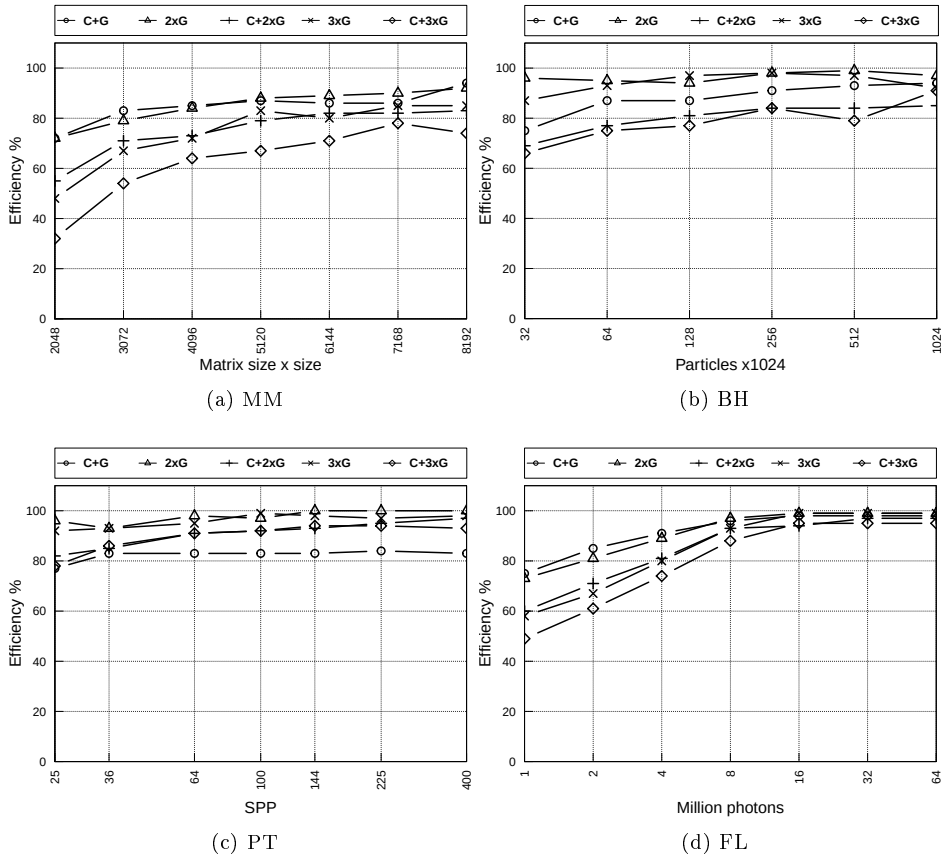
Fig. 6: Strong scalability: heterogeneous efficiency for the four case studies. 7k x 7k matrix for MM, 1024k particles in BH, 400 SPP for PT and 32M photons in FL.

Table 4: Strong scalability: heterogeneous efficiency for the four case studies. 7k x 7k matrix for MM, 1024k particles in BH, 400 SPP for PT and 32M photons in FL. C stands for CPU and G for GPU.

Application	C+G	2xG	C+2xG	3xG	C+3xG
MM	94%	92%	83%	85%	74%
BH	94%	97%	85%	92%	91%
PT	83%	99%	97%	98%	93%
FL	98%	99%	97%	99%	95%

Figure 6 illustrates the variation of heterogeneous efficiency for the four applications with different devices' configurations – Table 4 shows the corresponding values. These results show that high efficiency values (above 80%) are maintained for all applications. MM exhibits slightly lower efficiency values than the others because it has a very low computation–communication ratio, i.e., the number of arithmetic operations performed per byte read from memory is very low. There is also a drop in efficiency every time a CPU is added to a multiple GPU configuration. This happens because the CPU exhibits a much lower computing capacity (in the terms defined in Section 5.2) than the GPUs for these applications, as can be clearly seen in Figure 5 by comparing the C and G bars. It becomes thus extremely difficult for the runtime system to maintain the same efficiency level when a relatively less powerful device is added – remember however that this does not represent a loss in performance for the general case, just a loss in efficiency. Note that the efficiency reported for BH is lower than for PT and FL; however, the consumer kernel is be-

ing used for this irregular application. These results confirm that the conclusions drawn above with respect to irregular applications with light workloads per BWU: the consumer kernel can still be used, even though efficiency values will be lower than for more adequate irregular workloads.



(a) MM (b) BH

(c) PT (d) FL

Fig. 7: Heterogeneous efficiency with multiple workloads and multiple-device configurations. Consumer kernel for MM and BH, consumer-producer kernel for PT and FL. C stands for CPU and G for GPU.

Heterogeneous efficiency for all four case studies across different workloads and number of devices is depicted in Figure 7. In the general case efficiency increases with the workload and values within the range of 80% to 100% are achieved for the maximum tested workloads. It can thus be concluded that the proposed approach scales well with problem size within the range of devices and workloads evaluated.

In the general case efficiency decreases as the number of devices increases, particularly when the CPU is added to a configuration based only on GPUs. This is strong scalability and has been discussed before; overheads are expected to increase

with the number of devices and the CPU contributes with a reduced computing capability compared to the GPUs, making it harder to maintain very high efficiency levels. Efficiency, nevertheless, drops sublinearly with the number of devices.

PT and FL achieve higher efficiency than MM and BH across a wide range of problem sizes, with FL still struggling for smaller workloads. MM presents the worst efficiency values and its scalability is the poorest across both dimensions: workload and number of devices. This is due to the low computation-communication ratio. However, it still exhibits an average 80% efficiency and for the highest workload efficiency ranges from 74% to 94%, which are reasonable values considering the memory access overheads. BH's efficiency ranges between 80 and 98% for all problem sizes, except for very small workloads, with maximum values being achieved with two and three GPUs. This is a very good result since a consumer kernel is being used for an irregular application, given that BH exhibits very low workload per BWU. PT consistently achieves efficiency values above 80% for all workloads and system configurations. Even at low workloads PT performs well given the workload associated with each BWU and in spite of the memory management costs associated with tasks assignment as described in Section 5.1. Finally, FL has very low memory management overheads which enables the system to achieve an average of 98% efficiency above 8 million photons. This is very close to the ideal case, demonstrating that with the proper amount of work to suit available computer power and in the absence of implementation penalties (such as dynamic memory allocation per task), the overhead of the framework is properly compensated by the gains obtained with an effective intra-device scheduling.

6.3. *Comparison with StarPU*

In order to further validate our approach we compare the proposed run-time system with a state of the art heterogeneous system scheduling framework – StarPU [12]. Both run-time systems have similar data-management mechanisms, but StarPU does not explicitly target irregular workloads, uses a different inter-device scheduling strategy and ignores intra-device scheduling. StarPU scheduling is based on the Heterogeneous Earliest Finish Time (HEFT) algorithm [19] and in a history-based performance modelling. The HEFT has demonstrated to achieve good results with regular workloads on heterogeneous systems, but it does not address irregular workloads. We implemented the PT application in StarPU using the typical algorithm equivalent to FULLT and compare with our run-time system using the consumer-producer execution model. In StarPU, it is the user's responsibility to specify the task granularity, therefore we have tested multiple grain sizes and selected the one achieving the best results (240 tasks for most of the device configurations).

Figure 8 illustrates the speedup of our approach over StarPU with multiple device configurations and different workloads. With a single multi-core CPU our framework achieves a fairly constant speedup of 1.30x. The different tasks' sizes in both frameworks results on different behaviors that justify this speedup. The re-

maining configurations clearly show the benefit of using our intra-device scheduling mechanisms. With a single GPU a consistent increase in speedup is observed up to 1.53. Adding a CPU reduces the speedup because the gain with the CPU is lower and constant, but for the remaining configurations the speedup increases consistently achieving a maximum of about 1.50 with 576 SPP. The persistent kernel approach is able to balance the load within the GPU, which increases resource utilization and also leverages the coherence exhibited by the algorithm. These results clearly show that the proposed approach consistently achieves larger performance than StarPU for irregular workloads and that this performance gain increases with the workload size, thus favoring larger problem sizes. Also, even though speedups are reported only for up to 4 devices (one multi-core CPU and three GPUs), the data suggests, specially for larger workloads, that no inflection point is about to be reached and that additional devices would still exhibit significant speedups over StarPU. This conclusion has to be validated once we gain access to a system endowed with more computing devices. Combined with a suitable and unpredictability tailored inter-device scheduling our approach is thus able to deliver more performance and to efficiently exploit the available computing resources when compared with a state of the art system designed for regular workloads such as StarPU.

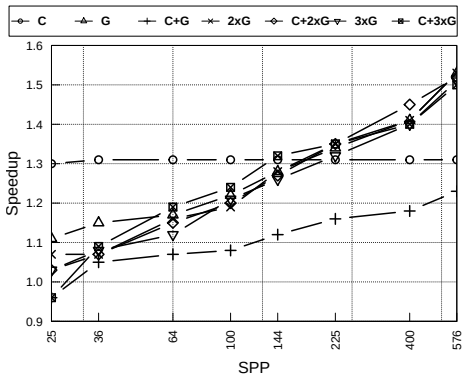


Fig. 8: Path tracing – Performance comparison with StarPU with multiple device configurations when scheduling irregular workloads. C stands for CPU and G for GPU. Note that horizontal axis is in log scale

6.4. *Discussion: Consumer vs Consumer-Producer*

The application programmer is responsible for selecting whether a consumer or a consumer-producer kernel is used to implement a given job. A consumer kernel has the advantages of allowing the utilization of optimized third party libraries and having an associated execution and programming model familiar to most programmers. A consumer-producer kernel explicitly handles load imbalances within a device, but exhibits overheads associated with queue management. The latter should be pre-

ferred over the former whenever the application workload is expected to be irregular, in the sense that it exhibits unpredictable workload and memory access patterns, which vary across elements of the data domain, and the workload per basic work unit mitigates the queue management overhead. A very important result is that it has been demonstrated that irregular applications that do not fulfill this last condition, can still scale well using a consumer kernel.

7. Conclusion and Future Work

This paper presented a framework for efficient execution of data parallel irregular applications on heterogeneous systems while maintaining high programming productivity. The framework integrates a unified programming and execution model with data-management and scheduling services, that keep the programmers agnostic to HS particularities, allowing them to concentrate on the application functionality. This paper concentrates on the programming model and on intra-device scheduling. Reported results show that both regular and irregular applications scale well as more devices are added to the computing system. They also show that intra-device scheduling, based on consumer-producer kernels, is able to sustain significant performance gains over consumer kernels for irregular applications, as long as the workload per basic work unit is enough to compensate the overheads associated with queuing and scheduling the large number of dynamically generated tasks. If the application exhibits a very low workload per basic work unit, then consumer kernels can still be used.

The proposed framework has proven to enable efficient exploitation of HS for irregular applications, while requiring minimal programming effort: using additional devices with architectures already exploited by the application only requires registering them through the HCP, while adding devices with different architectures (supported by the framework through the device API) requires providing the respective kernels. Expanding the framework support to new device architectures requires developing API implementations for those architectures, a task to be entailed by the framework developers, not application programmers. The run-time system was further validated and compared with a heterogeneous system scheduling framework – StarPU. Results reveal that our approach is able to outperform a state of the art run-time system designed for regular workloads. This is, to the best of our knowledge, the first published integrated approach that successfully handles irregular workloads over heterogeneous systems.

In the future we are planning to extend the proposed framework to clustered heterogeneous systems with further abstractions and mechanisms to maximize the use of these systems while maintaining high programming productivity. We also aim to provide support for emerging architectures such as DSPs and the brand new Intel MICs, and to further assess the scalability of the proposed mechanisms with systems with a larger number of devices. Finally, we plan to develop a mechanism to extract architecture specific kernels from a generic kernel, written in some higher level

language, which abstracts from particular execution models, inspired on approaches such as the OpenMP 4 [31] and OpenACC [32] programming models.

Acknowledgements

This work is funded by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) and by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) within projects PEst-OE/EEI/UI0752/2014 and FCOMP-01-0124-FEDER-010067. Also by the School of Engineering, Universidade do Minho within project P2SHOCS - Performance Portability on Scalable Heterogeneous Computing Systems.

References

[1] NVIDIA. CUDA Programming Guide 6.0. Technical report, 2014.
[2] John Feo, Oreste Villa, Antonino Tumeo, and Simone Secchi. Irregular applications: architectures and algorithms. In *Proceedings of the First Workshop on Irregular Applications: Architectures and Algorithms*, IAAA '11, pages 1–2, 2011.
[3] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. *ACM SIGARCH Computer Architecture*, May 2005.
[4] M. Burtscher, R. Nasre, and K. Pingali. A quantitative study of irregular programs on GPUs. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–151, 2012.
[5] M. Kalos and P. Whitlock. *Monte Carlo methods*. John Wiley & Sons, 2008.
[6] M. Pedersen. Monte carlo simulation in financial valuation. Technical Report HL-1302, Hvass Laboratories, 2014.
[7] M. Pharr and G. Humphreys. *Physically based rendering: from theory to implementation*. Morgan Kaufmann, 2nd edition, 2010.
[8] C. Zhu and Q. Liu. Review of Monte Carlo modeling of light transport in tissues. *Journal of Biomedical Optics*, 18(5), August 2013.
[9] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the High-Performance Graphics 2009*, 2009.
[10] Stanley Tzeng, Anjul Patney, and John D Owens. Task management for irregular-parallel workloads on the GPU. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, 2010.
[11] Gregory Diamos and Sudhakar Yalamanchili. Harmony: an execution Model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*. ACM, June 2008.
[12] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-andré Wacrenier. StarPU : a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009 Parallel Processing 15th International Euro-Par Conference*, volume 2009, 2009.
[13] Romain Dolbeau. HMPP: A hybrid multi-core parallel. In *First Workshop on General Purpose Processing on Graphics Processing Units*, 2007.
[14] Michael D Linderman, Jamison D Collins, Hong Wang, and Teresa H Meng. Merge: a programming model for heterogeneous multi-core systems. In *Proceedings of the*

13th international conference on Architectural support for programming languages and operating systems, ASPLOS XIII, 2008.

[15] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 2009.

[16] David August, Keshav Pingali, Derek Chiou, Resit Sendag, and Joshua J. Yi. Programming multicores: do applications programmers need to write explicitly parallel programs? *IEEE Micro*, May 2010.

[17] Gregory Diamos. *Harmony: an execution model for heterogeneous systems*. Phd, 2011.

[18] Jacques A. Pienaar, Anand Raghunathan, and Srimat Chakradhar. MDR. In *Proceedings of the international conference on Supercomputing*, May 2011.

[19] H. Topcuoglu and S. Hariri. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, March 2002.

[20] Daniel Cederman and Philippas Tsigas. On dynamic load balancing on graphics processors. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics Hardware*, GH '08, 2008.

[21] Markus Steinberger et al. Softshell : dynamic scheduling on GPUs. *Journal ACM Transactions on Graphics*, 2012.

[22] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, August 2007.

[23] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *ACM Queue*, March 2008.

[24] NVIDIA. *NVIDIA CUBLAS Library*, v5.0 edition, 2012.

[25] Intel. *Intel math kernel library*, v10.2 edition, 2009.

[26] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, December 1986.

[27] Martin Burtscher and Keshav Pingali. An efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In *GPU Computing Gems*. 2011.

[28] Iman Sadeghi, Bin Chen, and Henrik Wann Jensen. Coherent path tracing. *Journal of Graphics GPU Game Tools*, 2009.

[29] LuxRender. *SmallLuxGPU*, v1.3 edition, 2010.

[30] R. Chamberlain, D. Chace, and A. Patil. How are we doing? An efficiency measure for shared , heterogeneous systems. In *International Conference on Parallel and Distributed Computing Systems*, number September, pages 15–21, 1998.

[31] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013.

[32] OpenACC application programming interface version 2.0, 2013.