This is the authors version of an article published as :
E. Hermann, B. Raffin, F. Faure, T. Gautier, J. Allard, "Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations," in *Euro-Par 2010 - Parallel Processing*, LNCS 6272, pp. 235–246, 2010.

Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations

Everton Hermann[1], Bruno Raffin[1], François Faure[2], Thierry Gautier[1], and Jérémie Allard[1]

[1] INRIA and [2] Grenoble University

Abstract. Today, it is possible to associate multiple CPUs and multiple GPUs in a single shared memory architecture. Using these resources efficiently in a seamless way is a challenging issue. In this paper, we propose a parallelization scheme for dynamically balancing work load between multiple CPUs and GPUs. Most tasks have a CPU and GPU implementation, so they can be executed on any processing unit. We rely on a two level scheduling associating a traditional task graph partitioning and a work stealing guided by processor affinity and heterogeneity. These criteria are intended to limit inefficient task migrations between GPUs, the cost of memory transfers being high, and to favor mapping small tasks on CPUs and large ones on GPUs to take advantage of heterogeneity. This scheme has been implemented to support the SOFA physics simulation engine. Experiments show that we can reach speedups of 22 with 4 GPUs and 29 with 4 CPU cores and 4 GPUs. CPUs unload GPUs from small tasks making these GPUs more efficient, leading to a "cooperative speedup" greater than the sum of the speedups separatly obtained on 4 GPUs and 4 CPUs.

1 Introduction

Interactive physics simulations are a key component of realistic virtual environments. However the amount of computations as well as the code complexity grows quickly with the variety, number and size of the simulated objects. The emergence of machines with many tightly coupled computing units raises expectations for interactive physics simulations of a complexity that has never been achieved so far. These architectures usually show a mix of standard generic processor cores (CPUs) with specialized ones (GPUs). The difficulty is then to efficiently take advantage of these architectures.

Several parallelization approaches have been proposed but usually focused on one aspect of the physics pipe-line or targeting only homogeneous platforms (GPU or multiple CPUs). Object level parallelizations usually intend to identify non-colliding groups of objects to be mapped on different processors. Fine grain parallelizations on a GPU achieve high-speedups but require to deeply revisit the computation kernels.

In this article we propose a parallelization approach that takes advantage of the multiple CPU cores and GPUs available on a SMP machine. We rely on

the open source SOFA physics simulation library designed to offer a high degree of flexibility and high performance executions. SOFA [1] supports various types of differential equation solvers for single objects as well as complex scenes of different kinds of interacting physical objetcs (rigid objects, deformable solids, fluids). The physics pipe-line is classically split in two main steps: collision detection and time integration. The collision detection being performed efficiently on a single GPU [2], we focus here on the time integration step. The work load of time integration varies according to collisions: new collisions require the time integration step to compute and apply the associated new repulsion forces.

We developed a multiple CPUs and GPUs parallelization for the time integration step. A first traverse enables to extract a data dependency graph between tasks. It defines the control flow graph of the application, which identifies the first level of parallelism. Several tasks have a CPU implementation as well as a GPU one using CUDA [3]. This GPU code provides a second fine-grain parallelization level. At runtime, the tasks are scheduled according to a two levels scheduling strategy. At initialisation and every time the task graph changes (addition or removal of collisions), the task graph is partitioned with a traditional graph partitioner and partitions are distributed to PUs (GPUs or CPUs are called Processing Unit). Then, work stealing is used to move paritions between PUs to correct the work inbalance that may appear as the simulation progresses.

Our approach differs from the classical work stealing algorithm [4] as our stealing strategy takes the temporal and spatial locality into account. Spatial locality relies on the classical Owner Compute Rule where tasks using the same data tend to be scheduled on the same PU. This locality criteria is guaranteed during the tasks graph partitioning, where tasks accessing the same data are gathered in the same affinity group. Temporal locality occurs by reusing the task mapping between consecutive iterations. Thus, when starting a new time integration step, tasks are first assigned to the PU they ran on at the previous iteration.

CPUs tend to be more efficient than GPUs for small tasks and vice-versa. We thus associate weights to tasks, based on their execution time, that PUs use to steal tasks better suited to their capacities. Thanks to this criteria, PU heterogeneity becomes a performance improvement factor rather than a limiting one.

Experiments show that a complex simulation composed of 64 colliding objects, totaling more than 400k FEM elements, is simulated on 8 GPUs in 0.082s per iteration instead of 3.82s on one CPU. A heterogeneous scene with both complex and simple objects can efficiently exploit all resources in a machine with 4 GPUs and 8 CPU cores to compute the time integration step 29 times faster than with a single CPU. The addition of the 4 CPU cores not dedicated to the GPUs actually increases the simulation performance by 30%, significantly more than the 5% performance expected because CPUs unload GPUs from small tasks making these GPUs more efficient.

After discussing related works (Sec. 2) and a quick overview of the physics simulator (Sec 3), we focus on multi-GPU support (Sec. 4) and scheduling (Sec. 5). Experimental results (Sec. 6) are detailed before to conclude.

2 Related Works

We first review works related to the parallelization of physics engines before to focus on approaches for scheduling tasks on multiple GPUs or between the CPU and GPU.

Some approaches propose cluster based parallelizations for interactive physics simulations, but the scalability is usually limited due to the high overhead associated with communications [5]. This overhead is more limited with shared memory multi-processor approaches [6,7,8]. GPUs drew a lot of attention for physics, first because we can expect these co-processors to be easily available on users machines, but also as it can lead to impressive speedups[9,10,11]. The Bullet [1] and PhysX [2] physics engines defer solid and articulated objects simulation on GPU or Cell for instance. All these GPU approaches are however limited to one CPU and one GPU or co-processor. Task distribution between the processor and the co-processor is statically defined by the developer.

Recent works propose a more transparent support of heterogeneous architectures mixing CPUs and GPUs. GPU codelets are either automatically extracted from an existing code or manually inserted by the programmer for more complex tasks [12,13]. StarPU supports an heterogeneous scheduling on multiples CPUs and GPUs with a software cache to improve CPU/GPU memory transfers [14]. They experiment various scheduling algorithms, some enabling to get "cooperative speedups" where the GPU gets support from the CPU to get a resulting speedup higher to the sum of the individual speedups. We also get such speedups in our experiments. A regular work stealing strategy is also tested but the performance gain is more limited. The stealing scheme is not adapted to cope with the heterogeneity. Published experiments include tests with one GPU only.

We know two different approaches for multi GPU dynamics load balancing. The extension of the StarSs for GPUs [15] proposes a master/helper/worker scheme, where the master inserts tasks in a task dependency graph, helpers grab a ready task when their associated GPU becomes idle, while workers are in charge of memory transfers. The master leads to a centralized list scheduling that work stealing enables to avoid. RenderAnts is a Reyes renderer using work stealing on multiple GPUs [16]. The authors underline the difficulty in applying work stealing for all tasks due to the overhead of data transfers. They get good performance by duplicating some computations to avoid transfers and they keep work stealing only on one part of the Reyes pipeline. Stealing follows a recursive data splitting scheme leading to tasks of adaptive granularity. Both RenderAnts and StarSs address multi GPU schedulin g, but none include multiple CPUs.

In this paper we address scheduling on GPUs and CPUs. We extend the Kaapi runtime [17] to better schedule tasks with data flow precedences on multiple CPUs and GPUs. Contrary to previous works, the initial work load is balanced by computing at runtime a partition of the tasks with respect to their affinity to accessed objects. Then during the execution, the work imbalance is corrected by a work stealing scheduling algorithm [4,18]. In [19] the performance

[1] http://www.bulletphysics.com
[2] http://www.nvidia.com/physx

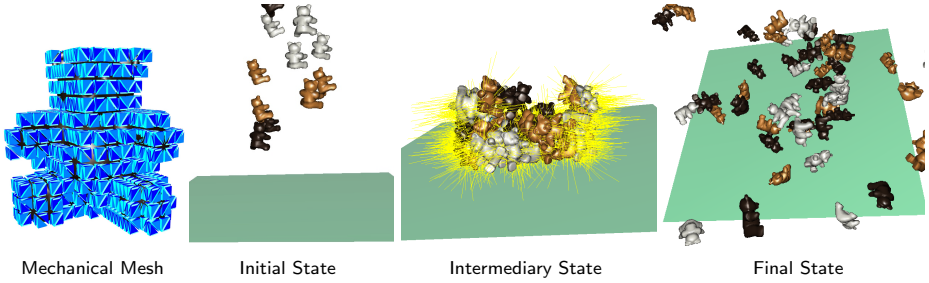


Fig. 1. Simulation of 64 objects, falling and colliding under gravity. Each object is a deformable body simulated using Finite Element Method (FEM) with 3k particles.

of the Kaapi work stealing algorithm was proved for tasks with data flow precedences, but not in the heterogenenous context mixing CPUs and GPUs. A GPU is significantly different from a "fast" CPU, due to the limited bandwidth between the CPU and GPU memories, to the overhead of kernel launching and to the SIMD nature of GPUs that does not fit all algorithms. The work stealing policy needs to be adapted to take advantage of the different PU capabilities to get "cooperative speedups".

3 Physics Simulation

Physics simulations, particularly in interactive scenarios, are very challenging high performance applications. They require many different computations whose cost can vary unpredictably, depending on sudden contacts or user interactions. The amount of data involved can be important depending on the number and complexity of the simulated objects. Data dependencies evolve during the simulation due to collisions. To remain interactive the application shoud execute each iteraction within a few tens of milliseconds.

The physics simulation pipeline is an iterative process where a sequence of steps is executed to advance the scene forward in time. The pipeline includes a collision detection step based on geometry intersections to dynamically create or delete interactions between objects. Time integration consists in computing a new state (i.e. positions and velocity vectors), starting from the current state and integrating the forces in time. Finally, the new scene state is rendered and displayed or sent to other devices.

In this paper, we focus on time integration. Interactive mechanical simulators can involve objects of different kinds (solid, articulated, soft, fluids), submitted to interaction forces. The objects are simulated independently, using their own encapsulated simulation methods (FEM, SPH, mass-springs, etc.). Interaction forces are updated at each iteration based on the current states of the objects.

Our approach combines flexibility and performance, using a new efficient approach for the parallelization of strong coupling between independently implemented objects [8]. We extend the SOFA framework [1] we briefly summarize

here. The simulated scene is split into independent sets of interacting objects. Each set is composed of objects along with their interaction forces, and monitored by an implicit differential equation solver. The object are made of components, each of them implementing specific operations related to forces, masses, constraints, geometries and other parameters of the simulation.

A collision detection pipeline creates and removes contacts based on geometry intersections. It updates the objects accordingly, so that each one can be processed independently from the others. By traversing the object sets, the simulation process generates elementary tasks to evaluate the physical model.

4 Multi-GPU Abstraction Layer

We first introduce the abstraction layer we developed to ease deploying codes on multiple GPUs.

4.1 Multi-architecture Data Types

The multi-GPU implementation for standard data types intends to hide the complexity of data transfers and coherency management among multiple GPUs and CPUs. On shared memory multiprocessors all CPUs share the same address space and data coherency is hardware managed. In opposite, even when embedded in a single board, GPUs have their own local address space. We developed a DSM (Distributed Shared Memory) like mechanism to release the programmer from the burden of moving data between a CPU and a GPU or between two GPUs.

When accessing a variable, our data structure first queries the runtime environment to identify the processing unit trying to access the data. Then it checks a bitmap to test if the accessing processing unit has a valid data version. If so, it returns a memory reference that is valid in the address space of the processing unit requesting data access. If the local version is not valid, a copy from a valid version is required. For instance it happens when a processing unit accesses a variable for the first time, or when another processing unit has changed the data. This detection is based on dirty bits to flag the valid versions of the data on each PU. These bits are easily maintained by setting the valid flag of a PU each time the data is copied to it, and resetting all the flags but that of the current PU when the data is modified.

Since direct copies between GPU memories are not supported at CUDA level, data first have to transit through CPU memory. Our layer transparently takes care of such transfers, but these transfers are clearly expensive and must be avoided as much as possible.

4.2 Transparent GPU Kernel Launching

The target GPU a kernel is started on is explicit in the code launching that kernel. This is constraining in our context as our scheduler needs to reallocate

```
struct TaskName : Task::Signature<const double*, int * > {};

template<>
struct RunCPU<TaskName> {
  void operator()( const double * a, int *b)
  {/* Implementation ... */ }
};

template<>
struct RunGPU<TaskName> {
  void operator()( const double * a, int *b)
  {/* Implementation ... */ }
};
```

Fig. 2. Multi-implementation task definition. Top: Task Signature. Left: CPU Implementation. Right: GPU Implementation.

a kernel to a GPU different form the one it was supposed to run on, without having to modify the code. We reimplemented part of the CUDA Runtime API. The code is compiled and linked as usually done in a single GPU scenario. At execution time our implementation of the CUDA API is loaded and intercepts the calls to the standard CUDA API. When a CUDA kernel is launched, our library queries the runtime environment to know the target GPU. Then the execution context is retargeted to a different GPU if necessary and the kernel is launched. Once the kernel is finished, the execution context is released, so that other threads can access it.

4.3 Architecture Specific Task Implementations

One of the objectives of our framework is to seamlessly execute a task on a CPU or a GPU. This requires an interface to hide the task implementation that is very different if it targets a CPU or a GPU. We provide a high level interface for architecture specific task implementations (Fig. 2). First a task is associated with a signature that must be respected by all implementations. This signature includes the task parameters and their access mode (read or write). This information will be further used to compute the data dependencies between tasks. Each CPU or GPU implementation of a given task is encapsulated in a functor object. There is thus a clear separation between a task definition and its various architecture specific implementations. We expect that at least one implementation be provided. If an implementation is missing, the task scheduler will simply reduce the range of possible target architectures to the supported subset.

5 Scheduling on Multi-GPUs

We mix two approaches for task scheduling. We first rely on a task partitioning that is executed every time the task graph changes, i.e. if new collisions or user interactions appear or disappear. Between two partitionings, work stealing is used to reduce the load imbalance that may result from work load variations due to the dynamic behavior of the simulation.

5.1 Partitioning and Task Mapping

As partitioning is executed at runtime it is important to keep its cost as reduced as possible. The task graph is simply partitioned by creating one partition per physical object. Interaction tasks, i.e. tasks that access two objects, are mapped to one of these objects' partition. Then, using METIS or SCOTCH, we compute a mapping of each partition that try to minimize communications between PUs. Each time the task graph changes due to addition or removal of interactions between objects (new collision or new user interactions), the partitioning is recomputed.

Associating all tasks that share the same physical object into the same partition allows to increase affinity between these tasks. This significantly reduces memory transfers and improves performance especially on GPUs where these transfers are costly.

A physics simulation also shows a high level of temporal locality, i.e. the changes from one iteration to the next one are usually limited. Work stealing can move partitions to reduce load inbalance. These movements have a good change to be relevant for the next iteration. Thus if no new partioning is required, each PU simply starts with the partitions executed during the previous iteration.

5.2 Dynamic Load Balancing

At a beginning of a new iteration each processing unit has a queue of partitions (an ordered list of tasks) to execute. The execution is then scheduled by the Kaapi [19] work stealing algorithm. During the execution, a PU first searches in its local queue for partitions ready to execute. A partition is ready if and only if all its read mode arguments are already produced. If there is no ready partition in the local queue, the PU is considered idle and it tries to steal work from another PU selected at random.

To improve performance we need to guide steals to favor gathering interacting objects on the same processing unit. We use an *affinity list* of PUs attached to each partition: a partition owned by a given PU has another distant PU in its affinity list if and only if this PU holds at least one task that interacts with the target partition. A PU steals a partition only if this PU is in the affinity list of the partition. We update the affinity list with respect to the PU that executes the tasks of the partition. Unlike the *locality guided work stealing* in [20], this affinity control is only employed if the first task of the partition has already been executed. Before that, any processor can steal the partition.

As we will see in the experiments, this combination of initial partitioning and locality guided work stealing significantly improves data locality and thus performance.

5.3 Harnessing Multiple GPUs and CPUs

Our target platforms have multiple CPUs and GPUs: the time to perform a task depends on the PUs, but also on the kind of task itself. Some of them may

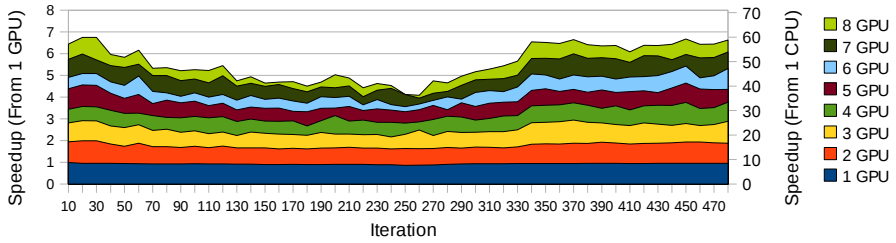


Fig. 3. Speedup per iteration when simulating 64 deformable objects falling under gravity (Fig. 1) using up to 8 GPUs

perform better on CPU, while others have shortest execution times on GPU. Usually GPUs are more efficient than CPUs on time consuming tasks with high degree of data parallelism; and CPUs generally outperform GPUs on small size problems due to the cost of data transfer and the overhead of kernel launching.

Following the idea of [18], we extended the work stealing policy to schedule time consuming tasks on the fastest PUs, i.e. GPUs. Because tasks are grouped into partitions (Sec. 5.1), we apply this idea on partitions. During execution we collect the execution time of each partition on CPUs and GPUs. The first iterations are used as a warming phase to obtain these execution times. Not having the best possible performance for these first iterations is acceptable as interactive simulations usually run several minutes.

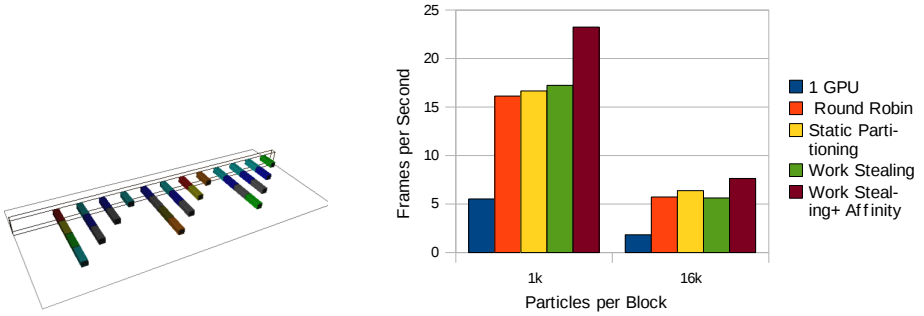
Instead of having a queue of ready partitions sorted by their execution times, we implement a dynamic threshold algorithm that allows a better parallel concurrent execution. Partitions with the $\frac{CPU\ Time}{GPU\ Time}$ ratio below the threshold are executed on a CPU, otherwise on a GPU. When a thief PU randomly selects a victim, it checks if this victim has a ready partition that satisfies the threshold criteria and steals it. Otherwise the PU chooses a new victim. To avoid PU starving for too long, the threshold is increased each time a CPU fails to steal, and decreases each time a GPU fails.

6 Results

To validate our approach we used different simulation scenes including independent objects or colliding and attached objects. We tested it on a quad-core Intel Nehalem 3GHz with 4 Nvidia GeForce GTX 295 dual GPUs. Tests using 4 GPUs where performed on a dual quad-core Intel Nehalem 2.4 GHz with 2 Nvidia GeForce GTX 295 dual GPUs. The results presented are obtained from the mean value over 100 executions.

6.1 Colliding Objects on Multiple GPUs

The first scene consists of 64 deformable objects falling under gravity (Fig. 3). This scene is homogeneous as all objects are composed of 3k particles and simulated using a Finite Element Method with a conjugate gradient equation solver.



(a) (b)

Fig. 4. (a) A set of flexible bars attached to a wall (a color is associated to each block composing a bar). (b) Performances with blocks of different sizes, using different scheduling strategies

At the beginning of the simulation all objects are separated, then the number of collisions increases reaching 60 pairs of colliding objects, before the objects start to separate from each other under the action of repulsion forces. The reference average CPU sequential time is 3.8s per iteration. We remind that we focus on the time integration step. We just time this phase. Collision detection is executed in sequence with time integration on one GPU (0.04s per iteration on average for this scene).

We can observe that when objects are not colliding (beginning and end of the simulation) the speedup (relative to one GPU) is close to 7 with 8 GPUs. As expected the speedup decreases as the number of collisions increases, but we still get at least a 50% efficiency (at iteration 260). During our experiments, we observed a high variance of the execution time at the iteration following the apparition of new collisions. This is due to the increasing number of steals needed to adapt the load from the partitioning. Steal overhead is important as it triggers GPU-CPU-GPU memory transfers.

The second scene tested is very similar. We just changed the mechanical models of objects to get a scene composed of heterogeneous objects. Half of the 64 objects where simulated using a Finite Element Method, while the other ones relied on a Mass-Springs model. The object sizes were also heterogeneous, ranging from 100 to 3k particles. We obtained an average speedup of 4.4, to be compared with 5.3 obtained for the homogeneous scene (Fig. 3). This lower speedup is due to the higher difficulty to find a well balanced distribution due to scene heterogeneity.

6.2 Affinity Guided Work Stealing

We investigated the efficiency of our affinity guided work stealing. We simulated 30 soft blocks grouped in 12 bars (Fig. 4(a)). These bars are set horizontally

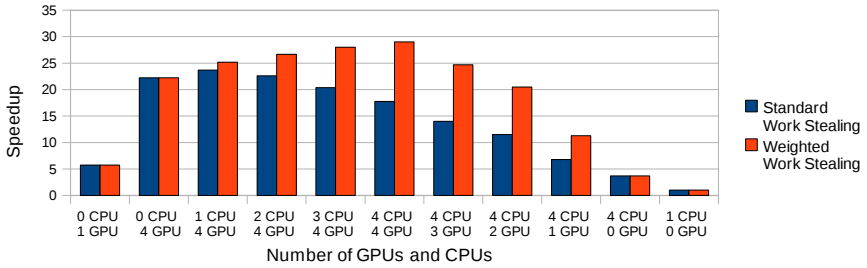


Fig. 5. Simulation performances with various combinations of CPUs and GPUs.

and are attached to a wall. They flex under the action of gravity. The blocks attached in a single bar are interacting similarly to colliding objects.

We then compare the performance of this simulation while activating different scheduling strategies (Fig. 4(b)). The first scheduling strategy assigns blocks to 4 GPUs in a round-robin way. The result is a distribution that has a good load balance, but poor data locality, since blocks in the same bar are in different GPUs. The second strategy uses a static partitioning that groups the blocks in a same bar on the same GPU. This solution has a good data locality since no data is transferred between different GPUs, but the work load is not well balanced as the bars have different number of blocks. The third scheduling relies on a standard work stealing. It slightly outperforms the static partitioning for small blocks as it ensures a better load balancing. It also outperforms the round-robin scheduling because one GPU is slightly more loaded as it executes the OpenGL code for rendering the scene on a display. For larger objects, the cost of memory transfers during steals become more important, making work stealing less efficient than the 2 other schedulings. When relying on affinity, work stealing gives the best results for both block sizes. It enables to achieve a good load distribution while preserving data locality.

6.3 Involving CPUs

We tested a simulation combining multiple GPUs and CPUs in a machine with 4 GPUs and 8 Cores. Because GPUs are passive devices, a core is associated to each GPU to manage it. We thus have 4 cores left that could compute for the simulation. The scene consists of independent objects with 512 to 3 000 particles. We then compare standard work stealing with the priority guided work stealing (Sec. 5.3).

Results (Fig. 5) show that our priority guided work stealing always outperforms standard work stealing as soon as at least one CPU and one GPU are involved. We also get "cooperative speedups". For instance the speedup with 4 GPUs and 4 CPUs (29), is larger than the sum of the 4 CPUs (3.5) and 4 GPUs (22) speedups. Processing a small object on a GPU sometimes takes as long as a large one. With the priority guided work stealing, the CPUs will execute tasks that are not well suited to GPUs. Then the GPU will only process larger

tasks, resulting on larger performance gains than that if it had to take care of all smaller tasks.

In opposite, standard work stealing lead to "competitive speed-downs". The simulation is slower with 4 GPUs and 4 CPUs than with only 4 GPUs. It can be explained by the fact that when a CPU takes a task that is not well-adapted to its architecture, it can become the critical path of the iteration, since tasks are not preemptive.

7 Conclusion

In this paper we proposed to combine partitioning and work stealing to parallelize physics simulations on multiple GPUs and CPUs. We try to take advantage of spatial and temporal locality for scheduling. Temporal locality relies mainly on reusing the partition distribution between consecutive iterations. Spatial locality is enforced by guiding steals toward partitions that need to access a physical object the thief already owns. Moreover, in the heterogeneous context where both CPUs and GPUs are involved, we use a priority guided work stealing to favor the execution of low weight partitions on CPUs and large weight ones on GPUs. The goal is to give each PU the partitions it executes the most efficiently. Experiments confirm the benefits of these strategies. In particular we get "cooperative speedups" when mixing CPUs and GPUs. Though we focused on physics simualtions, our approach can probably be straitforwardly extended to other iterative simulations.

Future work focuses on task preemption so that CPUs and GPUs can collaborate even when only large objects are available. We also intend to directly spawn tasks to the target processor instead of using an intermediary task graph, which should reduce the runtime environment overhead. Integrating the parallelization of the collision detection and time integration steps would also avoid the actual synchronization point and enable a global task scheduling further improving data locality.

Acknowledgments

We would like to thanks Théo Trouillon, Marie Durand and Hadrien Courtecuisse for their precious contributions to code development.

References

1. Allard, J., Cotin, S., Faure, F., Bensoussan, P.J., Poyer, F., Duriez, C., Delingette, H., Grisoni, L.: Sofa - an open source framework for medical simulation. In: Medicine Meets Virtual Reality (MMVR'15), Long Beach, USA (2007)
2. Faure, F., Barbier, S., Allard, J., Falipou, F.: Image-based collision detection and response between arbitrary volume objects. In: Symposium on Computer Animation (SCA'08), Switzerland, Eurographics (2008) 155–162

3. NVIDIA Corporation: NVIDIA CUDA compute unified device architecture programming guide (2007)
4. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. SIGPLAN Not. **33**(5) (1998) 212–223
5. Allard, J., Raffin, B.: Distributed physical based simulations for large vr applications. Virtual Reality Conference, 2006 (2006) 89–96
6. Gutiérrez, E., Romero, S., Romero, L.F., Plata, O., Zapata, E.L.: Parallel techniques in irregular codes: cloth simulation as case of study. J. Parallel Distrib. Comput. **65**(4) (2005) 424–436
7. Thomaszewski, B., Pabst, S., Blochinger, W.: Parallel techniques for physically based simulation on multi-core processor architectures. Computers & Graphics **32**(1) (2008) 25–40
8. Hermann, E., Raffin, B., Faure, F.: Interactive physical simulation on multicore architectures. In: EGPGV, Munich (2009)
9. Georgii, J., Echtler, F., Westermann, R.: Interactive simulation of deformable bodies on gpus. In: Proceedings of Simulation and Visualisation. (2005) 247–258
10. Comas, O., Taylor, Z.A., Allard, J., Ourselin, S., Cotin, S., Passenger, J.: Efficient Nonlinear FEM for Soft Tissue Modelling and its GPU Implementation within the Open Source Framework SOFA. In: ISBMS, Berlin, Heidelberg, Springer-Verlag (2008) 28–39
11. Harris, M.J., Coombe, G., Scheuermann, T., Lastra, A.: Physically-based visual simulation on graphics hardware. In: HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Aire-la-Ville, Switzerland, Switzerland, Eurographics Association (2002) 109–118
12. Leung, A., Lhoták, O., Lashari, G.: Automatic parallelization for graphics processing units. In: PPPJ'09, New York, NY, USA, ACM (2009) 91–100
13. Dolbeau, R., Bihan, S., Bodin, F.: Hmpp: A hybrid multi-core parallel programming environment. In: First Workshop on General Purpose Processing on Graphics Processing Unit. (2007)
14. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In: Euro-Par. Volume 5704 of LNCS., Delft (2009) 863–874
15. Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Ortí, E.S.: An extension of the starss programming model for platforms with multiple gpus. In: Euro-Par'09, Berlin, Heidelberg, Springer-Verlag (2009) 851–862
16. Zhou, K., Hou, Q., Ren, Z., Gong, M., Sun, X., Guo, B.: Renderants: interactive reyes rendering on gpus. ACM Trans. Graph. **28**(5) (2009) 1–11
17. Gautier, T., Besseron, X., Pigeon, L.: KAAPI: a thread scheduling runtime system for data flow computations on cluster of multi-processors. In: Parallel Symbolic Computation'07 (PASCO'07). Number 15–23, London, Ontario, Canada, ACM (2007)
18. Bender, M.A., Rabin, M.O.: Online Scheduling of Parallel Programs on Heterogeneous Systems with Applications to Cilk. Theory of Computing Systems **35**(3) (2000) 289–304
19. Gautier, T., Roch, J.L., Wagner, F.: Fine grain distributed implementation of a dataflow language with provable performances. In: PAPP Workshop, Beijing, China, IEEE (2007)
20. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The data locality of work stealing. In: SPAA, New York, NY, USA, ACM (2000) 1–12