Heterogeneous Particle-based Simulation

Takahiro Harada*
Advanced Micro Devices, Inc.

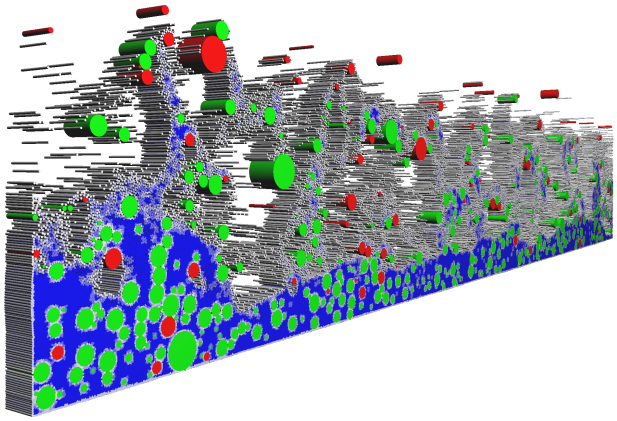

Figure 1: *A particle-based simulation with 50K small particles and 512 large particles.*

1 Introduction

Particle-based simulations have been used to simulate granular materials, fluids, and rigid bodies. To achieve realistic behavior, a large number of particles have to be simulated. Particle-based simulations are suited for GPUs because the computation of each particle is almost the same, (i.e., the granularity of the computation is uniform over the particles). This is preferable for GPUs with a wide SIMD architecture. However, particle-based simulation on the GPU has been mostly restricted to simulating particles of identical size [Harada et al. 2007]. This is because the work granularity is non-uniform if there are particles with different radii, which leads to inefficient use of the GPU. Heterogeneous CPU/GPU architectures, such as AMD Fusion®APUs, can solve this simulation efficiently by using the CPU and the GPU at the same time. On a PC with a CPU and a discrete GPU, whenever a computation is dispatched to the GPU, the data has to be sent via PCI Express®bus, which introduces a latency. However, heterogeneous architectures have a CPU and a GPU on the same die with a tightly coupled shared memory, so the same memory space can be accessed from the GPU and the CPU without any copying, which can facilitate a tight collaboration between the two processors. In this paper, we describe a particle-based simulation with particles of various sizes running on a heterogeneous architecture by dispatching and simultaneously processing work on the GPU and CPU depending on the granularity.

2 Method

The simulation we developed maximizes the use of all the available resources of a heterogeneous architecture by performing computation concurrently on both the CPU and the GPU components. The simulation uses a CPU thread for dispatching work to the GPU (GPU control thread) and multiple CPU threads for computation (CPU computation threads), whereas an application using only the GPU uses one CPU thread. The target architecture was an AMD A-Series APU with four CPU cores and a GPU. Our implementation

*e-mail: takahiro.harada@amd.com

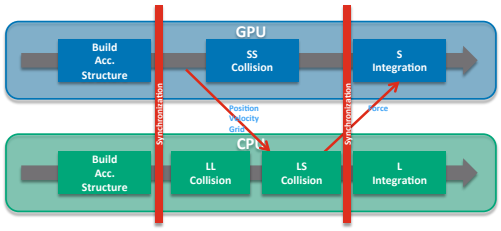

Figure 2: *A step of the simulation using two CPU threads.*

for the architecture used the GPU, a GPU control thread, and three CPU computation threads. For simplicity, we first describe a simulation using the GPU and two CPU threads (a GPU control thread and a CPU computation thread). Then we describe how to scale the simulation to the GPU, a GPU control thread, and multiple CPU computation threads.

2.1 Simulation using the GPU, a GPU control thread and a CPU computation thread

A simulation with particles of various sizes as shown in Fig.1 is a coupling of two simulations: a simulation with identical-sized particles (small particles colored with blue) and a simulation with varying-sized particles (large particles colored with red and green). If the interaction between large and small particles is not considered, the simulation of small particles has a uniform work granularity. Thus it is suited to be processed by the GPU. On the other hand, using the CPU is a better choice for the computation of large particles because the granularity of the simulation of large particles is not uniform. Therefore, small and large particle simulations are performed on the GPU and the CPU respectively. Note that they are also running concurrently.

A simulation step consists of three steps; building an acceleration structure, collision, and integration. Brute-force collision computation is prohibitively expensive when there are a large number of particles, so an acceleration structure is built to improve the efficiency of collision. Colliding particles are searched for and repulsion forces are calculated. The integration step updates particle velocity and positions.

For a coupled simulation as shown in Fig.1, we have to think about how to handle the collision between large and small particles (LS collision). LS collision is performed by searching for colliding small particles for each large particle and accumulating the forces on the small and large particles. To improve the efficiency of the search, we can reuse the data structure built for small-small (SS) collision. For each large particle, a bounding box in the coordinate system of the uniform grid is calculated and small particles found in the grid cells overlapped with the bounding box. Work granularity for each large particle depends on the size of the particle because the number of overlapping cells depends on the size of a bounding box. Therefore it is more efficient to perform LS collision on the CPU computation thread.

The small-particle data and acceleration structure are in the GPU memory space because the GPU is used for the simulation of small particles. If a CPU and a discrete GPU are used, these data have

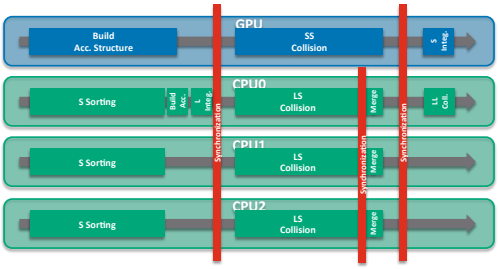

Figure 3: *A step of the simulation using four CPU threads.*

to be transferred to the main memory for LS collision. However, on a heterogeneous architecture, the CPU can use these data for computation without doing any expensive memory copies because the GPU and CPU are sharing the same memory. LS collision reads small-particle position, velocity, and acceleration structure from the shared memory and writes to the force buffer, which is used for the integration step on the GPU. In this way, two simulations, each of which is running on the CPU, and the GPU are coupled efficiently. Fig. 2 illustrates a step of the simulation. The first synchronization of the CPU and GPU is necessary to guarantee that the acceleration structure was built, and the second one is to prevent the GPU from using the force buffer before the CPU finishes updating at LS collision step.

2.2 Simulation using the GPU, a GPU control thread, and three CPU computation threads

We now extend the computation on a CPU computation thread to use multiple CPU computation threads. Since the integration is relatively cheap on the CPU and GPU, the key is how to exploit the new threads before and after the first synchronization. After the synchronization, we multi-threaded LS collision because it is more expensive compared to large-large (LL) collision. Parallelizing LS collision is done by splitting large particles into groups and dispatching a group to a CPU computation thread. However, we have to be cautious about write conflicts of small-particle values. When two large particles, each of which is processed by different threads, interact with the same small particle, both threads will update the value of the small particle. We could introduce a lock per particle, but it would be inefficient because it requires frequent lock operations. Instead, a local buffer is prepared for each thread so each thread can write exclusively to the buffer. After all the interactions are processed, these local buffers are merged into a single buffer using parallel merge.

Before the synchronization, we could also use CPU threads to help build the GPU acceleration structure. But it does not scale well as it suffers from frequent conflict on memory access. Instead we used CPU cycles to optimize SS collision on the GPU, which is the most expensive computation. Collision computation needs to look up neighboring particles from memory. This memory access pattern is basically random, but if the particle data is sorted by the spatial order, the memory access pattern improves, which results in better performance because of the increased chance of a cache hit. Thus we used CPU cycles to reorder particle data. Z curve, one of the space filling curves, is used to calculate an index for a particle in space. Particle data could be sorted from scratch for each frame, but because the change in particle distribution is small between steps, an incremental sorting is more efficient than full sort. We used *block transition sort*, which is parallelizable on multiple CPU threads [Harada 2010]. The sorting is also possible because of the memory sharing between CPU and GPU. If the data of small

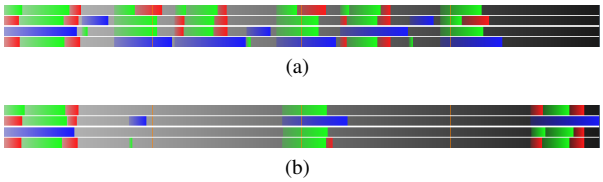

Figure 4: *Use of four threads for a simulation using shared memory (a) and using memory copy (b). The horizontal extent of rows is 33 miliseconds.. Each row corresponds to a CPU thread. Blue boxes show GPU computations. Green and red boxes show CPU computations. CPU threads that are idle or copying data between the CPU and the GPU are gray.*

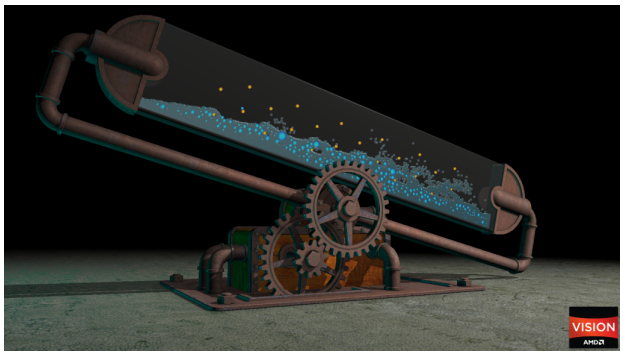

Figure 5: *A screen shot from AMD demo WaveMachine using the heterogeneous particle-based simulation.*

particles are not accessible from the CPU, they have to be copied over, and the overhead of memory copy might be bigger than the gain we can get by sorting particles. Fig. 3 shows a step of the simulation using a GPU control thread and three CPU computation threads (four CPU threads in total).

3 Results

The proposed method is implemented using C++ and OpenCL™ on a PC with an AMD A8-3510MX APU whose CPU and GPU frequencies are 1.8GHz and 444MHz, respectively. Fig. 1 is a screenshot of a two-dimensional simulation with 50K small particles and 512 large particles. The interaction between particles is calculated based on a damped spring. A step of the simulation takes 30.58ms. A simulation using explicit memory copy was also implemented for comparison. It took 90.46ms for a step. By comparing these two numbers, the benefit of the heterogeneous architecture is obvious. Fig. 4 shows visualizations of the detailed usage of four CPU threads.

Although CPU and GPU computation were tightly coupled to realize a simulation, there are still issues to be addressed. One of them is kernel launch overhead. Although a kernel dispatch has a small overhead, it accumulates for a computation with fine-grained dispatch to the GPU.

References

HARADA, T., KOSHIZUKA, S., AND KAWAGUCHI, Y. 2007. Smoothed particle hydrodynamics on GPUs. *Structure*, 1–8.

HARADA, T. 2010. Parallelizing particle-based simulation on multiple processors. In *Game Physics Pearls*, 155–176.