A Parallel Constraint Solver for a Rigid Body Simulation

Takahiro Harada*
Advanced Micro Devices, Inc.

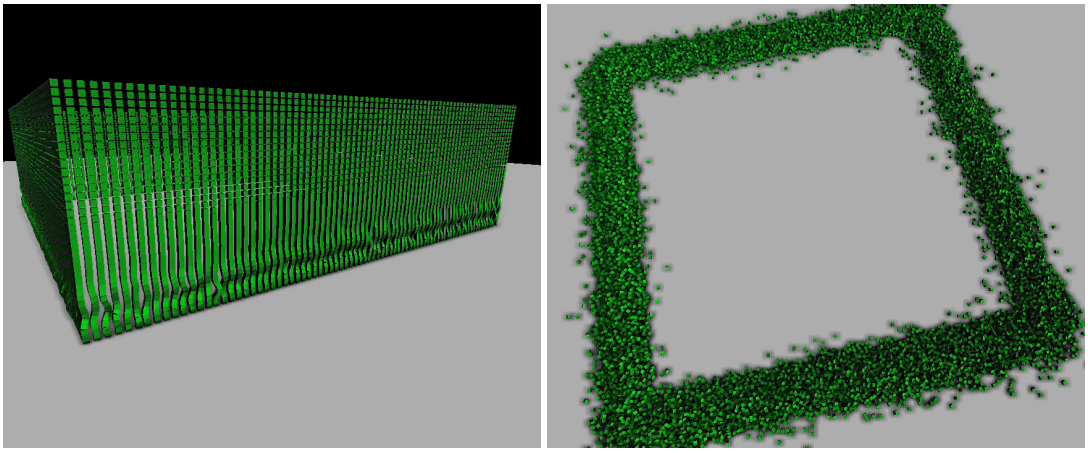

Figure 1: *A simulation using 12K rigid bodies. The number of contacts in the right figure is 180K. Our GPU narrow-phase collision detection and constraint solver takes 33ms on an ATI FirePro™ V8800.*

1 Introduction

Rigid body simulations are used often in games. Because of the computational cost, a large scale simulation is not feasible in real time. However, with the increased performance of GPUs, achieving a larger-scale simulation in real time becomes more realistic. A rigid body simulation is too complicated to implement on the GPU in a straightforward manner. One of the challenges is implementing an efficient constraint solver. A projected Gauss-Seidel method is often used to solve constraints [Catto 2005]. The input and output of the solver are the velocity of rigid bodies. However, constraints sharing a body cannot be solved in parallel because of this data dependency. This is a challenge when a GPU is used for a rigid body simulation. A solution to solve constraints in parallel is to split them into groups of constraints called batches. In a batch, no body is shared among constraints, so they can be solved in parallel. An introduction of batch solves the problem of constraint solving, but batch creation itself is a serial process. To complete a rigid body simulation pipeline on the GPU, batch creation on the GPU is also necessary.

We present a method to create batches and solve constraints in parallel on the GPU. Our batch creation is a two-step process: global split followed by local batch creation. Global split separates the constraints into constraint groups. In local batch creation, a SIMD. (a GPU consists of SIMD engines, or SIMDs, each of which has a wide SIMD, (e.g., an ATI FirePro™ V8800 has 20SIMD engines, each of which has 64 SIMD lanes)) processes a constraint group efficiently without doing any global synchronization. The batch creation has a good memory access pattern because it does most of the random memory accesses to fast on-chip local data share (LDS). The constraint solver also assigns a constraint group for a SIMD. Advantages of this approach not only minimize global communication, but also localize the computation that can improve the cache hit rate. Our constraint solver moves some dispatch work that had to be done by the CPU to the GPU, which reduces the dispatch overhead on the CPU.

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

SIGGRAPH Asia 2011, Hong Kong, China, December 12 – 15, 2011.
ISBN 978-1-4503-0807-6/11/0012

2 Method

2.1 Global Split

At first, constraints are split into disjoint constraint groups, each of which can be processed in parallel on a SIMD. However, it is not possible to remove all the dependency between groups. An example is a chain in which all the constraints are connected. For this situation, constraints cannot be split into disjoint groups. Our approach is to create sets of constraint groups. A set has several constraint groups, which do not have dependency among groups, but constraints in different sets can have dependency (Fig. 2). To divide constraints into groups, a spatial split is used. The simulation space is diced into grids and constraints are assigned for a cell. After the split, non-adjacent cells do not share constraints, so they can be processed in parallel. Constraints in each cell make up a constraint group. Because we used two-dimensional split, the cells or the constraint groups are split into four independent sets, which are solved in four steps. Solving a constraint group is an independent job processed in parallel. On a GPU, a SIMD processes a constraint group. However there can be dependency on constraints in a group: they cannot be processed in parallel on a wide SIMD architecture although they can be solved directly on a multi-core CPU. Local batch creation solves the issue.

Implementation Global split is performed by calculation of a cell index for each constraint followed by a radix sort, which sorts constraints by cell indices. Then the sorted constraints are scanned to obtain offsets for constraint groups for each cell and number of constraints in each group.

2.2 Local Batch Creation

Because local batch creation does not need any global information, it is performed within a SIMD. The goal of this step is to extract batches of constraints (independent sets of constraints) and to order constraints by batch indices. Batch creation is a serial process, so it is easy on the CPU, as discussed in the introduction. However,

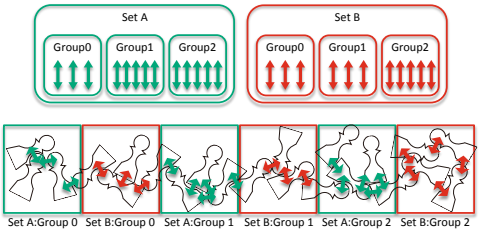

Figure 2: *Set and constraint groups.*

to create batches efficiently on a wide SIMD architecture, a serial algorithm cannot be used. The requirements for batch creation are two-fold: create good quality batches and create batches efficiently. Quality of batch or the number of batches is important because the increase in the number of batches results in longer constraint solving time. Existing methods do not satisfy these conditions.

Thus we propose *iterative batch creation*. At the first iteration, a SIMD lane fetches a constraint and tries to lock the two bodies of the constraint by local atomic operations. If a lane can lock both of them, the constraint can be processed in this batch. However, it fails to find independent constraints when more than one SIMD lane tries to lock a body at the same time, especially when the constraints are localized as in our case. To put more constraints in a batch, this step is repeated. In the next iteration, we need to know which bodies are already scheduled in the batch. Thus, a local buffer is prepared to mark bodies already scheduled in the batch. The lane that could lock both bodies marks bodies of the constraint in the local buffer. Before trying to lock the bodies of the constraint, it first checks if the bodies of the constraint are already scheduled in the batch by looking up the local buffer. If they are not yet scheduled, it tries to lock the bodies and updates the local buffer if it succeeds. These processes are repeated several times. The more iterations are used, more the quality of batches improves. But this is a trade-off over batch creation time. We set the number of iterations to four empirically.

Implementation Local batch creation is efficient since it can be performed by not doing any global, but only LDS operations except for the load and store of the constraints. Constraints are repeatedly accessed during the local batch creation so we first move them to a local constraint buffer. However the number of constraints for a SIMD can be too big; all the constraints in a group might not be able to be fit to the local buffer. This issue is solved by stream processing constraints. A fixed-sized local constraint buffer is used to store working sets of constraints. All the SIMD lanes are used to fill the local constraint buffer with constraints. Then iterative batch creation is performed for the constraints in the local buffer. Next, we sort the constraints into two by the batch creation result. If the constraint cannot go to the batch, it is written back to the local constraint buffer. If it can go, it is buffered to a local batched buffer at first and flushed to global memory when the buffer is filled for better SIMD utilization. It repeats this process until all the constraints for the SIMD are processed.

2.3 Solving Constraints

Constraints are solved iteratively to get a convergence for the solution. For an iteration of constraint solving, four kernels are executed, each of which processes a set created during global split. A constraint group in a set is a job taken by a SIMD. A SIMD fetches a constraint group and processes constraints in a group from the beginning. Again, the constraints might not be able to be processed at the same time, so they are stream processed. An offset is allocated to track the processed constraints and initialized to zero. Each SIMD lane adds its lane index to the offset to get a constraint to read. A SIMD lane reads a constraint of the index in the constraint group from global memory to local memory. However, not all of the constraints can be processed in parallel because constraints are only sorted by batch index and stored in a single array. The batch index of a constraint has to be checked whether or not it is equal to the one to be processed because constraints in a different batch cannot be processed at the same time. If all of them have the same index, all SIMD lanes process constraints. If an index is not the same, the SIMD lane is masked so they do not process them. After the solve, it shifts the offset with the number of constraints solved so that it can start reading from the elements that have not been processed. A SIMD repeats this process until all the constraints in a constraint group are solved.

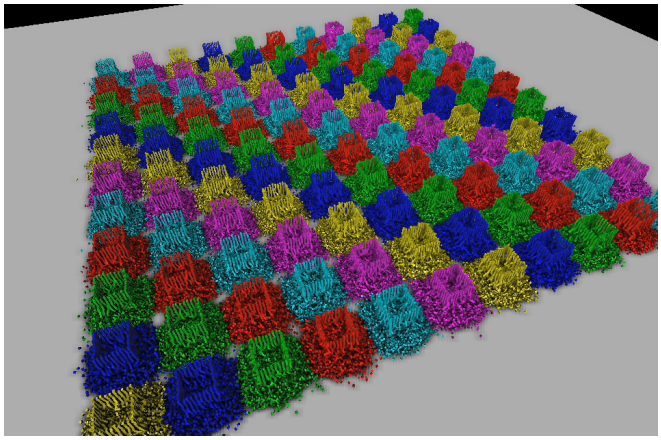

Figure 3: *A simulation using half-million rigid bodies running on 4 GPUs. After columns collapsed, constraints connected most of the rigid bodies.*

3 Results

The proposed method is implemented using OpenCL on a PC with an ATI FirePro^{TM}V8800. The constraint solver is used for our rigid body simulation in which narrow-phase collision detection is also implemented on the GPU. Fig. 1 shows screenshots from a simulation with 12K rigid bodies. Although solving the constraints on the left in parallel is difficult because the number of constraint is large (180K), and most of the constraints are connected, our constraint solver can solve it because it does not make any assumption about the configuration of constraints. This method is applicable not only to a GPU but also to a CPU with a wider SIMD architecture, as mentioned at the introduction. An extension of our method is to use multiple GPUs for a simulation. Fig. 3 is a screenshot from a simulation of half-million rigid bodies on four GPUs. At the moment, the broad-phase collision detection is performed on the CPU, but we are going to integrate our GPU broad-phase collision detection to complete full rigid body simulation pipeline on the GPU [Liu et al. 2010].

References

CATTO, E. 2005. Iterative dynamics with temporal coherence. *Game Developer Conference*, 1–24.

LIU, F., HARADA, T., LEE, Y., AND KIM, Y. J. 2010. Real-time collision culling of a million bodies on graphics processing units. *ACM Trans. Graph. 29* (December), 154:1–154:8.