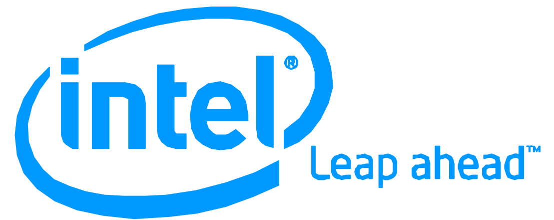


Game Physics Performance on the Larrabee Architecture

Aleksey Bader, Jatin Chhugani, Pradeep Dubey, Stephen Junkins, Sergey Lyalin, Teresa Morrison, Dmitry Ragozin, Anastasya Ryzhova, Sergey Sidorov, Alexei Soupikov, Mikhail Smelyanskiy
Intel Corporation

White Paper
Research @ Intel

Abstract

Game physics is at the heart of any modern game engine which employs the laws of physics to simulate life-like movement and interaction between objects, such as rigid and deformable bodies, cloth, and water. Game physics applications are very compute and memory intensive. The ever growing quest for a high degree of realism requires more complex physics algorithms and also larger datasets. To meet the demands of game physics applications requires a computer architecture which can deliver high floating point performance and memory bandwidth. However, general-purpose many-core architectures are quickly evolving to overcome these constraints. Larrabee is one such highly-threaded many-core architecture. It consists of an array of multiple IA Intel processor cores, each augmented with a 16-wide vector processor unit. In this paper, we analyze several key game physics applications. We show how Larrabee's extensive thread and data parallelism are well suited for a broad set of physics algorithms. We show how these algorithms parallelize and map to Larrabee architecture and achieve good parallel speedup with the number of cores.

Introduction

The booming market for computer games continues to drive the graphics community's insatiable desire for increased realism, believability, and speed. In the past decade, physical simulation has become a key to achieving the realism expected by game players.

A high degree of realism however requires life-like interaction between game objects. For example, a ball should react differently when bouncing across a concrete surface compared to grassy surface. The physical simulation of the ball must account for the different physical properties (friction, rigidity, etc.) of both surfaces and their impact on ball's animated motion. The cannon shell which hits the brick building should cause the individual bricks to fall and potentially the entire building to collapse. A heavy object thrown into the water should produce splashes and even waves which must quiet down over a period of time.

To achieve the desired degree of realism of physical simulation requires complex mathematical models which translate into compute-hungry algorithms. As the size of the scenes grows to accommodate larger number of objects, the computational complexity as well as size of data structures also increases. These two factors make real-time performance (at least 30 frames per second) hard to achieve without sufficient hardware resources.

However, general-purpose many-core architectures are quickly evolving to meet and overcome these challenges. Larrabee [Seiler et al 2008] is one such highly-threaded many-core architecture. It consists of an array of multiple IA Intel processor cores, each augmented with a 16-wide vector processor unit.

In this paper, we first examine several simulation models used to compute various types of physical simulation in modern games. We then briefly review the Larrabee Architecture. Finally, we describe how to parallelize these applications and map them to Larrabee to achieve good speedup with the number of cores. Specifically, we explain the implementation approaches we employed for the game physics workloads presented in Figure 17 of [Seiler et al 2008]. In summary, Larrabee's thread and data parallelism and its general-purpose processor architecture are a good match for a broad set of physics algorithms. This paper is an illustration of how game physics could be run on LRB. The specific implementation of game physics within the PC platform is beyond the scope of this paper. Additional performance details and architectural implications of game physics processing on Larrabee are discussed in [GamePhysics08].

Game Physics Simulation Models

In this section we show a high level model of the physical simulation time step, as well as several key game physics applications, which are all based on this simulation model. The description is based on [Chen07]. Figure 1 shows a typical time step in a physical simulation application. For each time step of a simulation, a physical simulation application takes as input the state of the simulated scene (e.g., positions, orientations, and velocities of all objects), as well as external control information (e.g., what the player is doing in a game). The application then computes the physical processes that lead to an updated state (e.g., force, torque, or pressure generation). Depending on the scheme used, this information will be used to advance the state

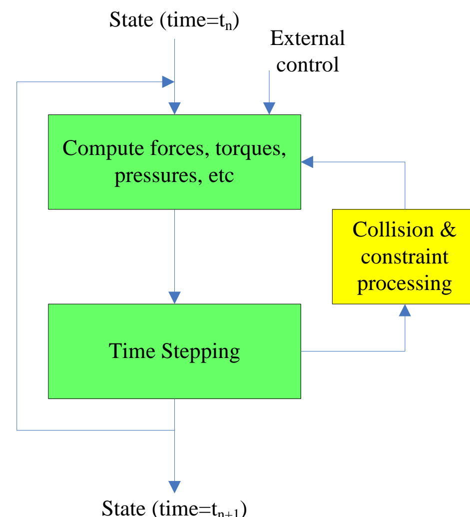


Figure 1: Overview of physical simulation

forward in time (e.g., time integration of the laws of motion), yielding a new candidate state. Should phenomena such as collisions or other constraints be triggered, the state may be updated in response to this collision or constraint, and the force computation/time stepping phases will be repeated, possibly with a smaller time step.

We now give examples of several important game physics applications which are based on the simulation pipeline above.

Game Fluids via SPH

Smoothed Particle Hydrodynamics (SPH) has recently emerged as a popular technique for interactive simulation of fluids [Muller2003]. The SPH method represents a fluid as a set of discrete particles and models a resistance to density changes: when particles get too close to one another, a repulsive force separates them; when they get too far from each other, an attractive force brings them together. If a pair of particles is far enough apart, no forces act between them. SPH discretizes the Navier-Stokes equations and samples its solution at a finite number of such particles in space and time. While in the grid-based method, the position of these sample points is fixed, in the SPH method, the particles are free to move around. This difference fundamentally changes the way the Navier-Stokes equations are solved and generally leads to much smaller complexity and ease of implementation, making SPH more suitable for interactive environments.

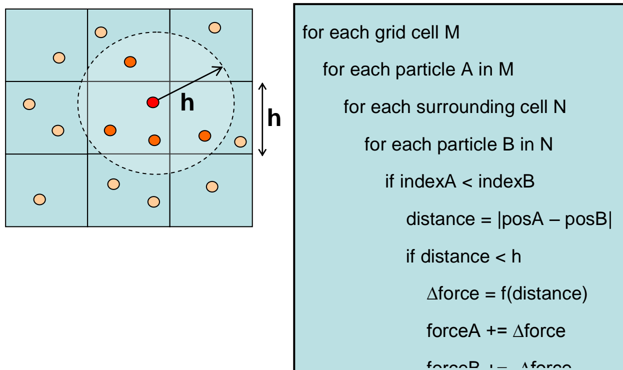


Figure 3: Spatial Data Structure for SPH

To keep track of each particle's neighbor and their distance from the particle, the acceleration data structures are sometimes used. For example, the uniform grid may be used, in which case, the particles are sorted into even size grid cells of size *h* at each simulation step (Figure 3). To identify all neighbors of a given particle, it is only required to check all particles in the surrounding cells.

Cloth Simulation

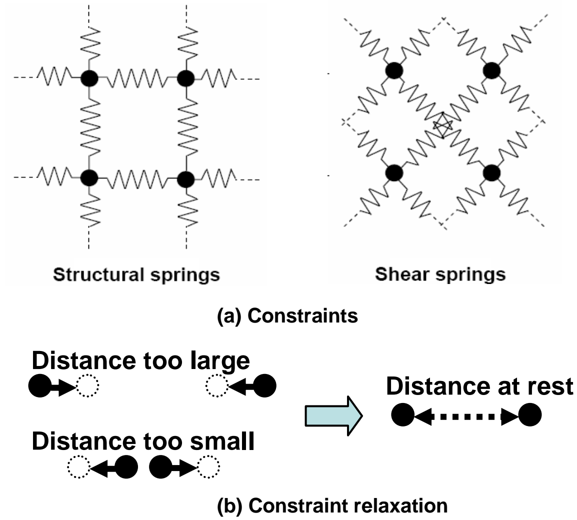


Figure 2: Distance Constraints in Cloth

Game physics models a cloth object as a set of particles [Jacobes2001]. Each particle is subject to external forces, such as gravity, wind and drag, as well as various constraints. As shown in Figure 2(a), these constraints are used to maintain the overall shape of the object (spring constraints), and to prevent interpenetration with the environment (collision constraints). The particle's equation of motion resulting from applying the external forces is integrated using explicit Verlet integration. The above-mentioned constraints create a system of equations linking the particles' positions together. This system is solved at each simulation time step by relaxation, that is, by enforcing the constraints one after another for a fixed number of iterations. This is illustrated in Figure 2(b). In addition, self-collisions are typically ignored.

Rigid Body Simulation

Rigid body dynamics [Eberly2003] simulates motion and interaction of non-deformable objects when forces and torques are present in the system. Rigid body dynamics is the most commonly used physical simulation in video games today. Examples of rigid bodies in games are vehicles, rag dolls, cranes, barrels, crates, and even whole buildings.

The traditional approach solves a system of ordinary differential equations, which represent Newton's second law of motion, `F=ma`, where `m` is the mass of an object, `a` is its acceleration, and

F is the applied force. The applied force determines the acceleration of the object, so the velocity and position are obtained by the integration of the above equation. The main computational challenge comes from the fact that the rigid bodies' motion is constrained due to their interaction with the environment. For example, consider a destructive environment in a video game where 1000s of rigid objects explode, collapse, and collide, resulting in 100,000s of interactive contacts. To realistically simulate such a scene requires determination of collisions (see next section), calculation of collision contact points, and physically correct computation of the contact forces that result from these contacts. The full rigid body pipeline, which demonstrates complete steps required for rigid body simulation is showed in Figure 4. To accelerate collision detection, spatial partitioning data structures such as grids or bounding volume hierarchies are used. To determine contact forces that result from collision contact, the contact is modeled as a linear complementary problem [Baraff97].

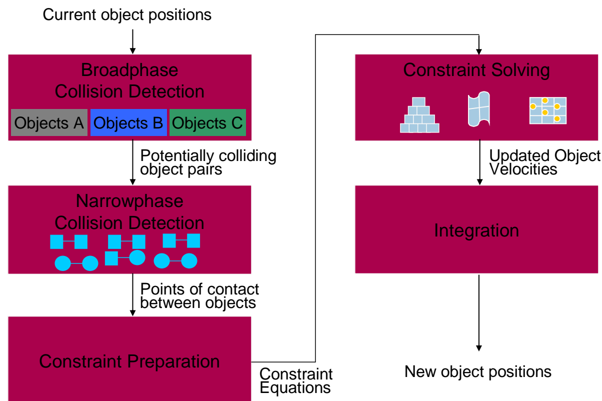


Figure 4: Rigid Body Pipeline

Collision Detection

Collision detection is important to all of the applications above and is used to determine whether two objects, represented as geometries, are in contact or interpenetrate each other, such as a particle and a solid object in game or cloth simulation, or two rigid bodies in rigid body simulation [Ericson04].

Collision detection consists of broad and narrow phases. Broad phase identifies potentially colliding objects by culling down the probable non-colliding object pairs. Narrow phase takes the potentially colliding pairs as the output from broad phase collision detection and detects the actual collision as well as colliding contact points. In the next stage, the constraint solver uses this information to calculate the contact forces necessary to prevent the bodies from interpenetrating (Figure 4).

Narrow phase collision detection relies on acceleration data structures to spatially separate simulated objects so that expensive collision tests only occur between objects that are in

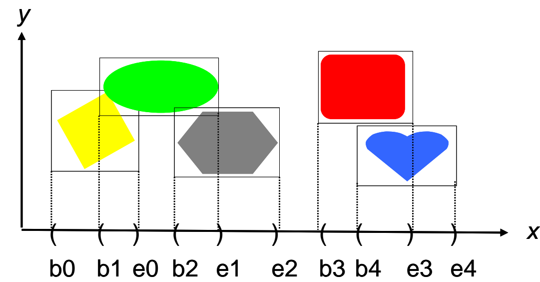


Figure 5: Sweep and Prune (S&P)

each other's vicinity. There are several algorithms used in game physics for broad phase. The most frequently used is sweep and prune (S&P). S&P sorts the starts and ends of the bounding volume of each object along its axis. As the solids move their starts and ends may overlap (Figure 5). When the bounding volume of two solids overlap in all axis they are likely to collide and are tested by narrow phase collision detection algorithm.

There exist a number of specialized and optimized routines for narrow phase collision detection, such as determining the collision between two spheres or two capsules. However collision detection between more complex geometries such as convex objects is commonly performed using the GJK algorithm [Gilbert88]. GJK iteratively computes the distance between two convex geometries until the minimum distance is found. In the case of non-convex geometries, the methods frequently used are (1) to break the non-convex geometry into several convex geometries and use convex collision methods on each, and (2) to represent the geometry using a triangulated surface and then use a general algorithm for colliding triangular meshes, which again involves colliding convex geometries.

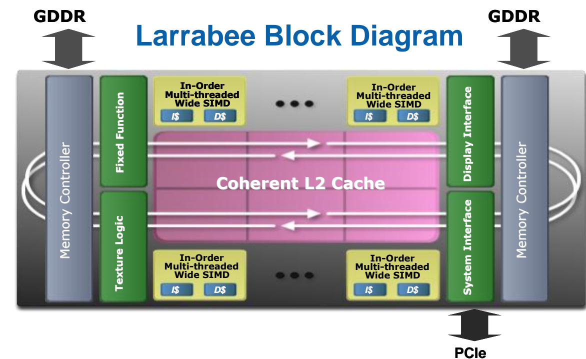


Figure 6: Schematic of the Larrabee many-core architecture

		SIMD	Threads
Rigid Body	Broad Phase (SAP)	SIMD-friendly implementation	Straightforward with OpenMP
	Box-Box Collision	Straightforward with LRB Vector ISA Support	Straightforward with OpenMP
	GJK	Straightforward with LRB Vector ISA Support	Straightforward with OpenMP
	Constraint Solver	Algorithm and data restructuring for SIMD	Requires fine grain synchronization
Fluid	Rebuild Grid	Straightforward with LRB Vector ISA Support	Requires fine grain synchronization
	Compute Forces	SIMD-friendly implementation with LRB ISA Support	Coarse-grain synchronization
	Process Collisions	Straightforward with autovecotrizer	Straightforward with OpenMP
	Advance Particles	Straightforward with autovecotrizer	Straightforward with OpenMP
Cloth	Constraint Relaxation	Algorithm restructuring for SIMD	Algorithm restructuring for TLP

Figure 7: Parallelism Characteristics

Larrabee Many-core Architecture

Larrabee (see Figure 6) is a many-core IA architecture, designed to scale to a large number of cores. Larrabee is designed to target a broad range of parallel applications, such as 3D rendering, game physics, medical imaging, and high performance computing. Each Larrabee core supports the full Pentium x86 instruction set. Each core has a short in-order instruction pipeline, which is shared between multiple threads. The hardware can switch threads in order to issue instructions from the threads which have ready instructions to execute and stalling other threads, which are waiting on data dependencies (e.g., back-to-back data dependence or L1 cache load miss). As a result, high latency stalls experienced by some threads are hidden behind useful work executed by other threads. Analysis of a broad range of parallel applications demonstrates that hardware support for multi-threading can provide a good tradeoff between hardware complexity and application performance improvement due to multi-threading.

While Larrabee architecture has very high bandwidth memory system, to mitigate bandwidth requirements of many applications, each Larrabee core is equipped with a low-latency first level (L1) cache. A large, global, fully coherent second level (L2) cache is evenly partitioned into separate local caches, each per processor core. Each core has a fast access path to its local partition. However, to enable data sharing among multiple partitions, a system bus link is used to access data from remote cache partitions.

Each core is further equipped with a 16-wide vector pipeline unit (VPU) that executes integer and single-precision floating-point vector instructions. It is well known that the two fundamental limiters to vector efficiency are (i) control flow and (ii) indirect memory accesses. Larrabee provides hardware support to address these two challenges.

To cope with control flow, each Larrabee instruction can use a mask register to predicate which vector element or memory locations are written to and which are left unmodified. There are also mask manipulating instructions that set the mask registers. To cope with indirect memory accesses, Larrabee includes gather/scatter instructions, which gather/scatter data from non-contiguous memory locations, based on the addresses contained in the vector of addresses; 16 elements are loaded from or stored to up to 16 distinct memory addresses. The speedup of gather/scatter is limited by the cache, which can usually access one cache line per cycle, which may result in 16 accesses to the cache in the worst case. However, if the gathered or scattered data is co-located in the single cache line, a frequently observed scenario in many graphics and non-graphics workloads, these elements can be gathered in a single access to L1, thus substantially reducing number of cache accesses and improving application performance.

Mapping Game Physics Simulation Models to Larrabee

All game physics applications will benefit from high computational capabilities of the Larrabee architecture and its powerful ISA.

In this section, we show how to parallelize these applications to exploit both thread and SIMD level parallelism on the Larrabee architecture.

Figure 7 provides a high-level characterization of both thread-level and SIMD-level parallelism in different game physics modules. The rest of this section provides more detailed discussion of parallelism and explains this characterization.

Straightforward Parallelism

For many game physics kernels thread-level and SIMD-level parallelism are straightforward to expose and exploit. For example, multiple independent iterations of the loop can be easily partitioned among multiple threads by using an OpenMP pragmas [OpenMP05]. For SIMD, it means that computations within each loop iteration can be turned into vector code by using auto-vectorizing compiler.

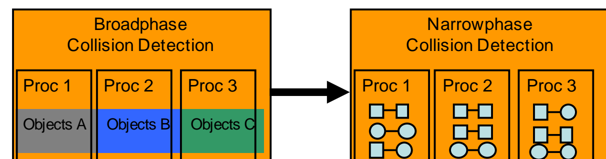


Figure 8: Exploiting TLP in Collision Detection

Collision detection algorithms are examples of game physics kernels that are trivially parallelizable. For instance, broad-phase sweep and prune algorithm can sweep individual bodies completely in parallel, while narrow phase collision detection between a pair of rigid bodies are completely independent from each other. The work in these two algorithms can be easily distributed among multiple threads using OpenMP (see Figure 8). Similarly, "advance particles", an algorithm which computes the next particle position given their current position and computed forces, is a simple loop which iterates over elements in an array performing simple computation on these elements. Hence it is trivially vectorizable.

Thread Synchronization and software locks

For some of the physics algorithms exploiting thread level parallelism requires synchronization among multiple threads.

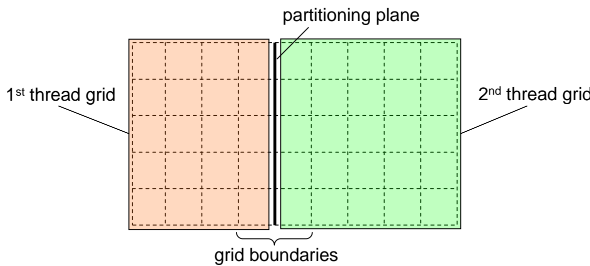


Figure 9: Grid Partitioning in SPH

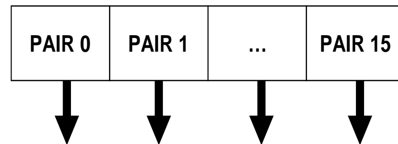
Typically we use software locks to guarantee mutual exclusion. For example, an algorithm to build spatial data structure for accelerating neighbor queries in SPH partitions (see Game Fluids section) of all particles among different threads requires mutual exclusion support. In this case, each thread iterates over its local subset of particles. The thread maps and inserts particles into corresponding neighbor grid cells as necessary. Since each grid cell may contain more than one particle, multiple particles are inserted into the head of the linked list. Insertion into the same grid cell by multiple threads requires mutual exclusion to guarantee that only one thread updates linked list at a time. In such cases, SIMD parallelization is used within each loop iteration and not across as is the case when all loop iterations are independent. As another example, consider the algorithm to compute forces, which is also part of game fluids application. In this algorithm, each particle in the grid cell updates each of its neighbors which are within a certain distance of the particle. It can happen that multiple particles in one or different grid cells may update another particle in the same or different grid cell. To assure correct implementation when such particles belong to different threads requires mutual exclusion to guarantee that only one thread performs an update. Acquiring and releasing locks for each grid cell would result in significant synchronization overhead. To avoid this overhead, we partition the grid into large sub-grids which are assigned to individual threads, as shown in Figure 9. Updates inside the sub-grid are guaranteed to be performed by the thread which owns the sub-grid and requires no synchronization. It is only the boundary grid cells that may be shared among multiple threads and hence require mutual exclusion. Sub-grid can be chosen coarse enough to almost entirely amortize overhead of locking the boundary cells.

SIMD Hardware Support

There is a subclass of game physics applications which contain large amounts of SIMD-level parallelism but to exploit it efficiently requires special hardware support which is available on Larrabee architecture.

```
if (earlyCDtest1(…)==TRUE )
        return;
if (earlyCDtest2(…)==TRUE )
        return;
Contact=getcontactpoints();
```

(a) Single Pair Collision



(b) 16 pair collision using 16-wide SIMD

16 colliding pairs

```
F0 = earlyCDtest1(…));      // sets F0 to 0 if collide
F1 = earlyCDtest2() ? F0;   // sets F1 to 0 if collide
Contacts=getcontactpoints(…) ? F1;
```

(c) Use of writemasks

Figure 10: Using Masks in Narrow-Phase Collision

It is well known that the main challenges of SIMD efficiency are control flow and indirect memory accesses. If the original scalar code contains branches, then to map such code to SIMD, requires conditional SIMD operations. Larrabee provides conditional SIMD execution support through the use of vector bit masks. For each SIMD instruction, a bit mask is provided to specify which operands to ignore.

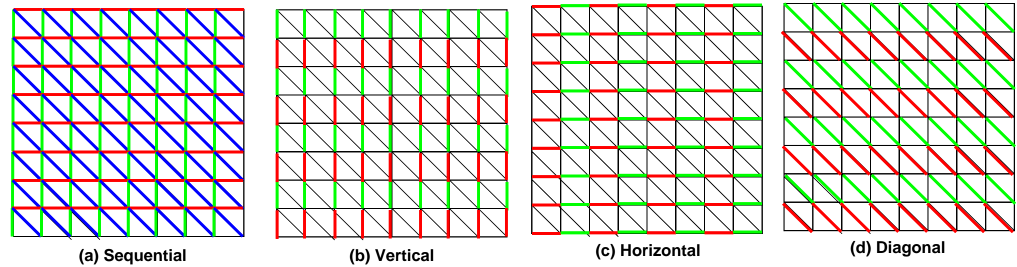


Figure 11: Constraint Relaxation in Cloth Solver

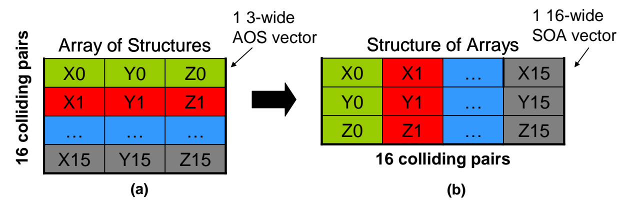


Figure 12: Indirect Accesses in Collision Detection

SIMD masks are very useful in many narrow phase collision detection algorithms, such as box-box collision or GJK. Such algorithms typically contain many early termination tests which try to detect the absence of collision between two objects as early as possible, in order to avoid expensive computation. Figure 10(a) shows an example of early termination tests, *earlyCDtest1* and *earlyCDtest2*, in sequential collision detection performed between two geometries. If both of these tests fail (*FALSE*), the objects collide and the routine to determine collision contact points, *getcontactpoints*, is invoked.

To map the above code to Larrabee 16-wide SIMD, we perform collision detection on 16 collision pairs simultaneously using 16-wide SIMD, as shown in Figure 10(b). The collision detection tests are converted into operations which set the SIMD writemasks, F0 and F1, as shown in Figure 10(c). If early termination test is FALSE, then the corresponding mask elements are set to 0. The remainder of the code will read 0 mask to assure the no further computation is performed on corresponding collision pair. The contact computation code is guarded by the F1 mask. If the mask is 0, then one of the two early exits occurred in the corresponding collision pair test. Thus the result of the computation does not modify contact data structure for the corresponding collision pair, as would be the case in sequential version. While this results in lower SIMD utilization in cases where large fraction of the mask elements is 0, it does allow efficient vectorization of the collision detection code. In most cases, SIMD utilization is high due to the broad phase algorithm which culls off the majority of non-colliding objects: the objects which remain are likely to be colliding and hence are likely to pass early termination test more often than not.

Traditionally, when programming hardware SIMD vector units, the operands and the results are required to be grouped together sequentially in memory. To achieve the best performance, they should be placed into an address-aligned structure. Figure 12(a) shows an example of an array 16 structures, one per collision pair, where each structure has 3 fields. Without hardware support for a "vector gather," efficient SIMD usage requires repacking the array of structures into structure of arrays format. As shown in Figure 12(b), fields from multiple structures are packed together. This data rearrangement can be quite expensive if it is done in software.

Larrabee offers hardware acceleration of such operation via instruction support for "vector gather" (see Larrabee subsection). A gather instruction gathers corresponding vector elements from non-contiguous memory locations and stores them contiguously into a single SIMD vector register. In the above example, a gather instruction gathers 16 non-contiguous data values, e.g., X0, X1,..., X15.

We see that Larrabee vector support for control flow and indirect memory accesses enables much higher SIMD efficiency than with a software implementation and thus are paramount for achieving high performance on game physics algorithms.

Parallelism-friendly Implementations and Algorithms

Most of the game physics algorithms contain plenty of SIMD and thread-level parallelism, which can be easily exploited using the help of SIMD hardware support and thread-level synchronization. However there are several algorithms where parallelism is not readily available. To expose such parallelism requires changes in implementation, algorithm or both.

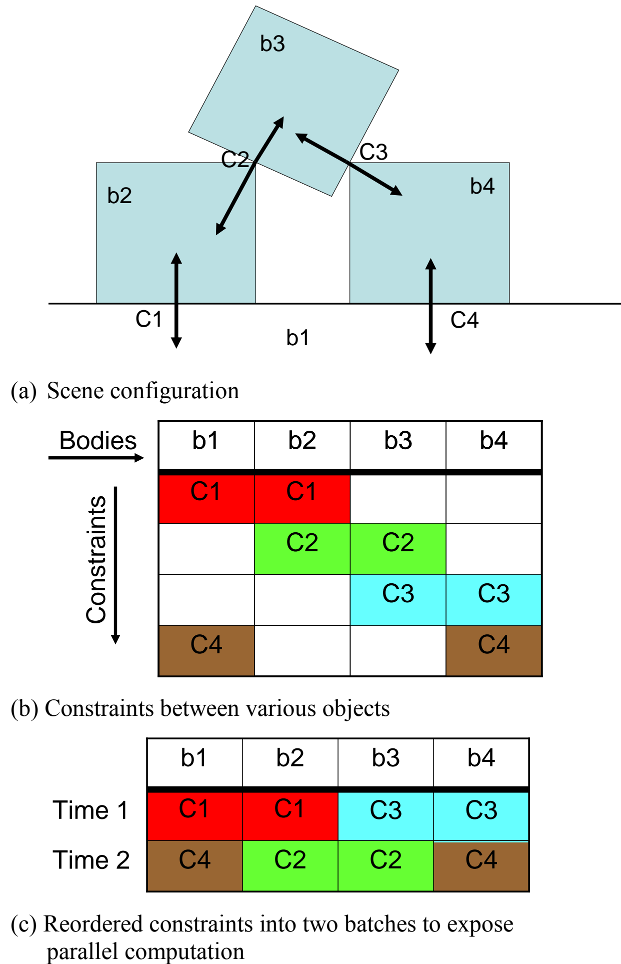


Figure 13: Making physics solver SIMD friendly

An example of algorithm that requires changes is game cloth. As mentioned in the Cloth Simulation section, it uses an iterative constraint relaxation technique, which iteratively enforces one constraint after another until the system converges towards a state where all these constraints are simultaneously satisfied. Constraint enforcement consists of adjusting the position of each of the particles in the grid, based on its distance from its neighbors on right, below and across the diagonal. Straightforward enforcement of all the constraints from left to right, top to bottom, as shown in Figure 11(a), results in sequential dependence between the constraints and as the result is not amenable to parallelization. Instead we choose several passes through the constraints. In the first pass, we only satisfy vertical constraints, as shown in Figure 11(b). We see that all these constraints are independent from one another and can be processed in parallel. Similarly, we perform horizontal and diagonal passes to satisfy horizontal and diagonal constraints (Figure 11(c) and Figure 11(d), respectively).

While breaking the dependence between the constraints exposes large amounts of both SIMD and thread-level parallelism, it potentially slows down the algorithm convergence, because, it takes more iterations for a constraint to impact others. However, through simulation based analysis we noticed that the parallel algorithm is never more than two times slower than the original sequential algorithm. Large amounts of exposed parallelism will compensate for the slower convergence.

Another example comes from the rigid body simulation model. During the execution of the physical simulation pipeline, the collision detection phase computes the pairs of colliding bodies, which are used as inputs to the constraint solving phase. The physics solver operates on these pairs and computes the separating contact forces, which keep the bodies from inter-penetrating each other. In Figure 13(a), we show one such case involving four bodies (three boxes and one ground plane), where the corresponding pairs of colliding bodies are listed in Figure 13(b). The resulting constraints C1, C2, C3, and C4 need to be resolved to update the body positions.

To parallelize this phase, we would ideally like to distribute the constraints amongst the available threads and resolve them in parallel. However, there is often an inherent dependency between consecutive constraints. In our example, constraints C1 and C2 both involve body b2 and thus cannot be resolved in parallel. These dependencies can force a significant serialization of the computation. But, we can reorder the constraints into different batches such that there are no conflicting constraints in each batch. That is, each batch will contain at most one constraint that refers to any given body.

Reordering algorithms traverse the constraints, maintaining an ordered list of partially filled batches. Each constraint is assigned to the earliest batch with no conflicting constraints. As a result, all constraints within a batch can be processed in parallel, while the different batches have to be processed sequentially.

For example, we reorder the constraints in Figure 13(b) and obtain two batches, (C1, C3) and (C4, C2), as shown in Figure 13(c). Note that C1 and C3 from the first batch do not refer to any body more than once and can be resolved in parallel. A similar observation holds for C4 and C2. As a result C1 and C3 in the first batch are solved in parallel and the results are fed as part of the input to the second batch.

Once parallelism is exposed as above, it maps to both threads and SIMD in a straightforward fashion. To exploit thread-level parallelism requires barrier synchronization between consecutive batches of parallel constraints.

Scalability Results

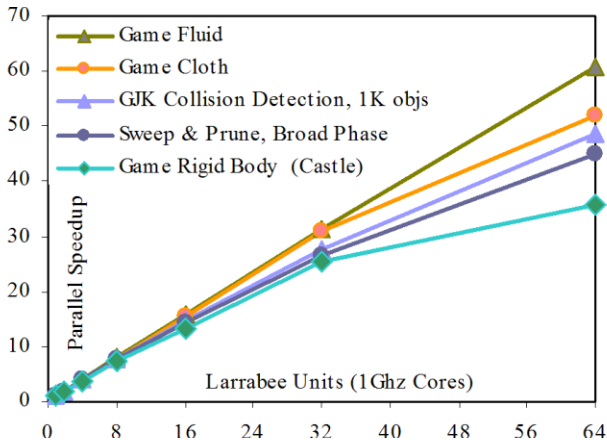


Figure 14: Game Physics Scalability Performance

We have performed detailed scalability simulation analysis of several game physics workloads on various configurations of Larrabee cores [Seiler08]. Figure 14 shows the scalability of some widely used game physics benchmarks and algorithms for rigid body, fluid, and cloth. We achieve better than 50% resource utilization using up to 64 Larrabee cores, and achieve near-linear parallel speedup is some cases. This shows that the Larrabee architecture is scalable to meet the growing performance needs of interactive rigid body, fluid, and cloth simulation algorithms.

Conclusions

In this paper we introduce several key game physics simulation models. To meet the large compute and memory demands of these game physics applications, we require a computer architecture which can deliver high floating point performance and memory bandwidth.

We briefly describe Intel's Larrabee, a many-core multi-threaded architecture with a 16-wide vector processor unit. Its high computational capability and bandwidth can meet demands of such game physics applications.

We further describe several key game physics simulation models and show how they parallelize, map and scale on Larrabee architecture. Performance details and architectural implications are discussed in a follow on paper [GamePhysics08].

References

[1] [Baraff97] D. Baraff, "Physically Based Modeling: Principals and Practice," *Online Course Notes, SIGGRAPH*, 1997.

[2] [Eberly03] D. H. Eberly. Game Physics. Morgan Kaufmann/Elsevier, San Francisco, 2003.

[3] [Ericson04] Christer Ericson, San Mateo. Real-time Collision Detection (Series in Interactive 3D Technology), Morgan Kaufmann, 2004.

[4] [Gilbert88] E. G. Gilbert, D.W. Johnson, and S. S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. In *IEEE Journal of Robotics and Automation*, 4(2):193–203, 1988.

[5] [Chen07] Yen-Kuang Chen, Jatin Chhugani, Christopher J. Hughes, Corporate Technology Daehyun Kim, Sanjeev Kumar, Victor Lee, Albert Lin, Anthony D. Nguyen, Eftychios Sifakis, Mikhail Smelyanskiy. High-Performance Physical Simulations on Next-Generation Architecture with Many Cores. *Intel Technology Journal*, 2007.

[6] [Jacobsen01] T. Jacobsen, Advanced Character Physics, *Game Developers Conference*, 2001.

[7] [Muller03] M. Muller, D. Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the Eurographics Symposium on Computer Animation*, 2003.

[8] [OpenMP05] OpenMP Application Program Interface, May 2005, Version 2.5.

[9] [Seiler08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Pat Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. In *Proceedings of SIGGRAPH*, 2008.

[10] [GamePhysics08] Parallelization and Analysis of Game Physics Performance on Larrabee Many-core Architecture. Currently under submission.

Information in this document is provided in connection with intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in intel's terms and conditions of sale for such products, intel assumes no liability whatsoever, and intel disclaims any express or implied warranty, relating to sale and/or use of intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Copyright © Intel Corporation 2008

* Other names and brands may be claimed as the property of others.