GPU rigid body simulation using OpenCL

Erwin Coumans, http://bulletphysics.org

Introduction

This document discusses our efforts to rewrite our rigid body simulator, Bullet 3.x, to make it suitable for many-core systems, such as GPUs, using OpenCL. Although OpenCL is thought, most of it can be applied to projects using other GPU compute languages, such as NVIDIA CUDA and Microsoft DX11 Direct Compute.

Bullet is physics simulation software, in particular for rigid body dynamics and collision detection in. The project started around 2003, as in-house physics engine for a Playstation 2 game. In 2005 it was made publically available as open source under a liberal zlib license. Since then it is being used by game developers, movie studios and 3d modelers and authoring tools such as Maya, Blender, Cinema 4D etc.

Before the rewrite, we have been optimizing and refactoring Bullet 2.x for multi-code, and we'll briefly touch on those efforts. Previously we have worked on simplified GPU rigid body simulation, such as [Harada 2010] and [Coumans 2009]. Our recent GPU rigid body dynamics work has approached the same quality compared to the CPU version.

The Bullet 3.x rigid body and collision detection pipeline runs 100% on the GPU using OpenCL. On a high-end desktop GPU it can simulate 100 thousand rigid bodies in real-time. The source code is available as open source at http://github.com/erwincoumans/bullet3 . Appendix B shows how to build and use the project on Windows, Linux and Mac OSX.

Bullet 2.x Refactoring

Bullet 2.x is written in modular C++ and its API was initially designed to be flexible and extendible, rather than optimized for speed. The API allows the user to derive his own classes and to select or replace individual modules that are used for the simulation. A lot of refactoring work has been done to optimize its single-threaded performance, without changing the API and data structures.

- Use very efficient acceleration structures to avoid doing expensive computations
- Incrementally update data structures instead of computing from scratch
- Pre-compute and cache data so that results that can be reused
- Optimize the inner loops using SSE and align data along cache lines
- Reduce the amount of dynamic memory (de)allocations, for example using memory pools

We ported Bullet 2.x to Playstation 3 Cell SPUs. This required some refactoring and we re-used some of this effort towards a basic multithreaded version that was cross-platform using pthreads and Win32 Threads.

Bullet 3.x Full Rewrite

It became clear that the Bullet 2.x API, data structures and algorithms didn't scale well towards massive parallel multi-threading. The following sections explain some of the issues we ran into. To be future proof, we started to invest in GPGPU technology, with the expectation that this effort would also help towards CPU multi-threading with a larger number of cores.

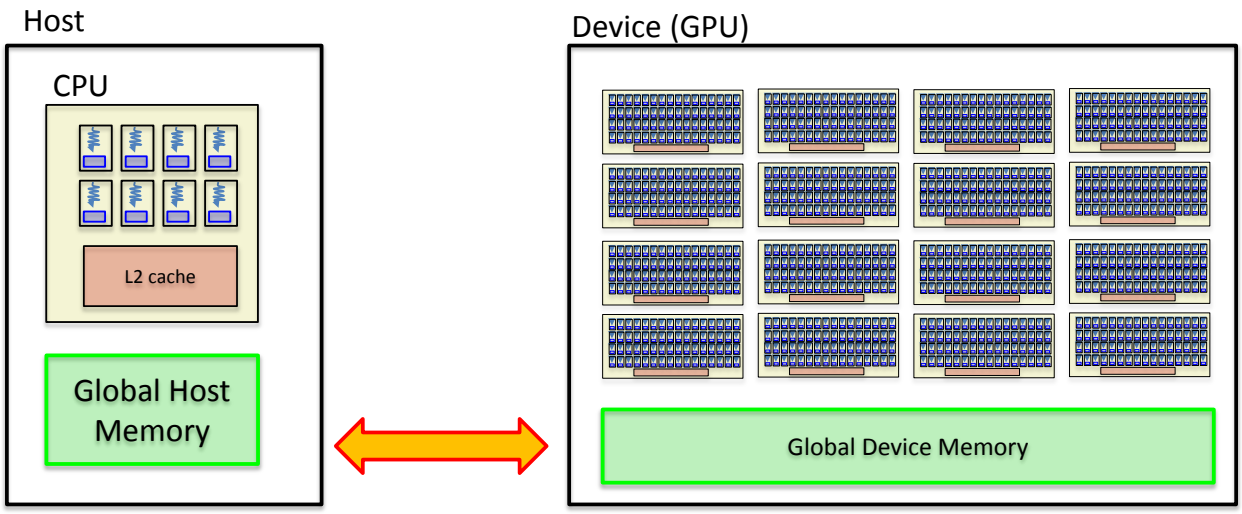
Optimizing for GPU requires more changes to code and data, in comparison to CPU multithreading. This document goes into details of this on-going work on Bullet 3.x.

Getting started with OpenCL

You can use OpenCL to parallelize a program, so that it runs multi-threaded on many cores. Although OpenCL seems primarily suitable for GPUs, an OpenCL program can be compiled so it executes multi-threaded on a CPU target. The performance of an OpenCL program running on CPU is competitive with other solutions, such as pthreads or Intel Thread Building Blocks. There are even implementations that allow to run OpenCL programs distributed over a network using MPI. So OpenCL is a very flexible and cross-platform solution.

OpenCL terminology

If we target a GPU Device to execute our OpenCL code, we still need a Host such as the CPU for initialization of the device memory and to start the execution of code on the device.



The OpenCL code that runs on the Device is called a Kernel. OpenCL kernel code looks very similar to regular C code. Such kernel code needs to be compiled using a special compiler, that is usually provided by the Device vendor. This is similar to graphics shader compilation, such as GLSL, HLSL and Cg.

To get access to OpenCL we need at minimum the OpenCL header files, and some way to link against the OpenCL implementation. Various vendors such as AMD, Intel, NVIDIA and Apple provide an OpenCL software development kit, which provides those header files and a library to link against. As an alternative, we also added the option to dynamically load the OpenCL dynamic library and import its symbols at run-time. This way, the program can continue to run, even if OpenCL is not installed.

Our first OpenCL kernel

Let's start with a very simple example that shows the conversion of some simple code fragment into OpenCL kernel.

```
typedef struct
{
        float4 m_pos;
        float4 m_linVel;
} Body;

void   integrateTransforms (Body* bodies, int nodeID, float timeStep)
{
        for (int nodeID = 0;nodeID<numBodies;nodeID++)
        {
                if( bodies[nodeID].m invMass != 0.f)
                {
                        bodies[nodeID].m pos +=  bodies[nodeID].m linVel * timeStep;
                }
        }
}
```

When we convert this code into an OpenCL kernel it looks like this:

```
__kernel void   integrateTransformsKernel( __global Body* bodies,const int numNodes, float
timeStep)
{
        int nodeID = get global id(0);
        if( nodeID < numNodes && (bodies[nodeID].m_invMass != 0.f))
        {
                bodies[nodeID].m_pos +=  bodies[nodeID].m_linVel * timeStep;
        }
}
```

We need to write some host code in order to execute this OpenCL kernel. Here are typical steps for this host code:

1. Initialize OpenCL context and choose the target device
2. Compile your OpenCL kernel program
3. Create/allocate OpenCL device memory buffers
4. Copy the Host memory buffer to the device buffer
5. Execute the OpenCL kernel
6. Copy the results back to Host memory

The OpenCL API is very low-level, so we created a simple wrapper to match our coding style and to make it easier to use. This was a good learning experience. We also added additional features in the wrapper, you can check out the OpenCL Tips and Tricks section for more information. This wrapper doesn't hide the OpenCL API, so at any stage we can use plain OpenCL.

Porting existing code to OpenCL

We ran into many issues that prevented our code to run on GPU at all, let alone efficiently. This section shares some experiences, even though the solutions might be obvious.

Replace C++ by C

OpenCL is close to plain C, but Bullet 2.x is written in C++. This means that most of the code needs to be rewritten to use C and structures, instead of C++ and classes with inheritance etc. This conversion was easy, although it took time. It helps that in Bullet 2.x we avoid many C++ features: no exception handling, no run-time type information (RTTI), no STL and very limited use of template classes.

Share CPU and GPU code

In the process of porting, it is easiest to first implement a CPU version of an OpenCL kernel, and then port this code to an OpenCL kernel. Once this is done, it is useful to validate the OpenCL kernel with the CPU reference version, making sure that the output is the same.

In Bullet 2.x a lot of the algorithms involve a basic linear algebra math library for operations on 3d vectors, quaternions and matrices. OpenCL has some built-in support for this, with the float4 data type and operators such as the dot, cross product on float4 and many others. It helps to refactor your CPU math library so that the same code can run on the GPU: it allows development and debugging of the same implementation on the CPU. For compatibility we added some typedefs, global operators and access to the scalar operators .x .y .z and .w of the b3Vector3 on CPU, which resembles a float4.

Easy GPU<->CPU data synchronization

In Bullet 2.x we mainly use a resizable container template, similar to the STL std::vector. The actual name is b3AlignedObjectArray, but for simplicity we just call it std::vector here. During porting to OpenCL we found that it is really useful to have a container that keeps the data on the GPU. This container should make it easy to synchronization between the CPU and GPU data. So we designed the btOpenCLArray<> template.

```
template <typename T>
class b3OpenCLArray
{
        size_t  m_size;
        size_t  m_capacity;
        cl_mem  m_clBuffer;


. . .


        inline bool push_back(const T& _Val,bool waitForCompletion=true);
        void copyToHost(std::vector& destArray, bool waitForCompletion=true) const;
        void copyFromHost(const b3AlignedObjectArray<T>& srcArray, bool waitForCompletion=true)

}
```

This simplifies the host code a lot. Here is some example use.

```
std::vector<b3RigidBody> cpuBodies;
b3OpenCLArray<b3RigidBody> gpuBodies(clContext,clQueue);
gpuBodies.copyFromHost(cpuBodies);


. . .


gpuBodies.copyToHost(cpuBodies);
```

Move data to contiguous memory

In the Bullet 2.x API, the user is responsible for allocating objects, such as collision shapes, rigid bodies and rigid constraints. Users can create objects on the stack or on the heap, using their own memory allocator. You can even derive their own sub class, changing the so that the object size. This means that objects are not store in contiguous memory, which makes it hard or even impossible to transfer to the GPU.
The easiest way to solve this is to change the API so that objects creation and removal is handled internally by the system. This is a big change in the API and one of many reasons for a full rewrite.

Replace pointers by indices

In Bullet 2.x, our data structures contained pointers from one object to the other. For example, a rigid body has a pointer to a collision shape. A rigid constraint has a pointer to two rigid bodies, and so on. Those pointers are only valid in CPU Host memory and cannot be used on the GPU. This means that data created on the CPU cannot be used on the GPU. This issue can be solved, by replacing pointers by indices.
Generally it may be better to rethink the data structures. For Bullet 3.x the data structures are different, and there is not always a one-to-one mapping between new Bullet 3.x and Bullet 2.x types.

```
struct btTransform
{
	btMatrix3x3	m_basis;
	btVector3	m_position;
};
class btRigidBody : public btCollisionObject
{
	btMatrix3x	m_inverseInertiaWorld;
	btVector3	m_linearVelocity;
	btVector3	m_angularVelocity;
	btScalar	m_mass;
	. . .
};
class btCollisionObject
{
	btTransform	m_worldTransform;
	btCollisionShape* m_collisionShape;
	. . .
};
```

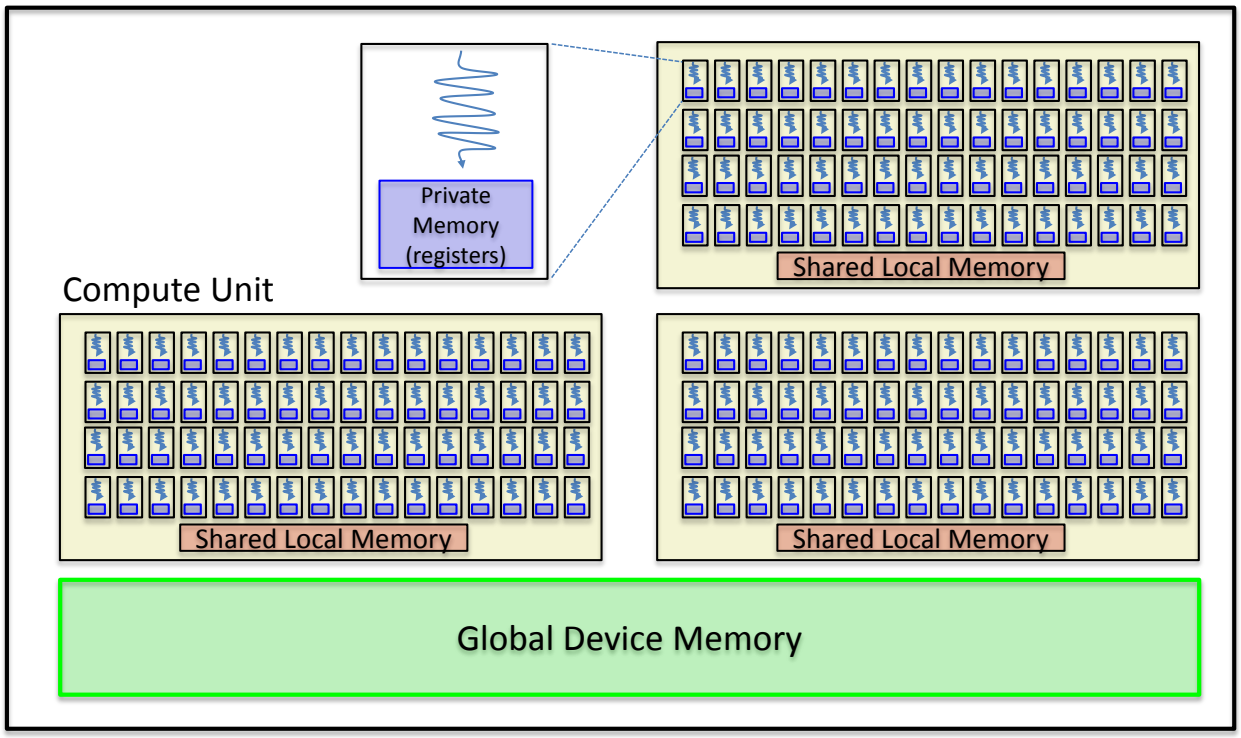
In our current Bullet 3.x we don't derive the rigid body from the collision object, and we moved some data from one class/struct to the other and created some new structures.

```
struct b3RigidBody
{
	b3Vector3	m_position;
	b3Quaternion	m_orientation;
	int		m_collidableIndex;
	. . .
};
struct b3Collidable
{
	int m_shapeType;
	int m_shapeIndex;
};
```

The basic *integrateTransforms* example in the previous section is embarrassingly parallel: there are no data dependencies between the bodies. Many of the algorithms in Bullet 2.x have some data dependencies so they cannot be trivially parallelized. The following section briefly covers GPU and OpenCL. After that, we discuss how we made effective use of the GPU many-cores and the GPU memory hierarchy in our rigid body simulation work.

Exploiting GPU hardware

A high-end desktop GPU has thousands of cores that can execute in parallel, so you need to make effort to keep all those cores busy. Those cores are grouped into Compute Units with typically 32 or 64 cores each. The cores inside a single Compute Unit execute the same kernel code in lock-step: they are all executing the same instruction, like a wide SIMD unit. The work that is performed by executing a kernel on a single core is called a Work Item in OpenCL. To make sure that multiple work items are executed on the same Compute Unit, you can group them into a Work Group. The hardware is free to map Work Groups to Compute Units, and this makes OpenCL scalable: if you add more Compute Units, the same program will run faster. The drawback is that there is no synchronization between Compute Units, so you need to design your algorithm around this. The host can wait until all Work Groups are finished, before starting new work.



Dealing with branchy code/thread divergence

Because all work items in a Compute Units execute in lock step, this means that code that has a lot of conditional statements can become very slow and inefficient.

```
  kernel void   branchyKernel (. . .)
{
        if (conditionA)
        {
                        someCodeA(. . .);
        } else
        {
                        someCodeNotA(. . .);
        }
}
```

If not all the work items in a Compute Unit have the same value for 'conditionalA' then they have to wait for eachother to finish executing 'someCodeA' and 'someCodeB'.

On the other hand, if all work items in the Compute Unit have the same value for *conditionA*, then only one of the two 'someCode*' sections will be executed, and there is no performance penalty for the if-statement.

Sort the input

If we know the *conditionalA* before executing the OpenCL kernel, we can sort the work items based on this *conditionalA*. This way, it is more likely that all the work items with a similar *conditionalA* will be processed in the same Compute Unit. In Bullet 3 we could use two parallel radix sorts on the overlapping pair array, based on each collision shape type (shapeTypeA and shapeTypeB) in a pair:

```
__kernel void   primitiveContactsKernel(__global int2 pairs, _global b3RigidBody* rigidBodies,
                                        global b3Collidable* collidables, const int numPairs)
{
        int nodeID = get global id(0);
        if (nodeID>=numBodies)
                return;
        int bodyIndexA = pairs[i].x;
        int bodyIndexB = pairs[i].y;
        int collidableIndexA = rigidBodies[bodyIndexA].m collidableIdx;
        int collidableIndexB = rigidBodies[bodyIndexB].m_collidableIdx;
        int shapeTypeA = collidables[collidableIndexA].m_shapeType;
        int shapeTypeB = collidables[collidableIndexB].m_shapeType;
        if (shapeTypeA== SHAPE PLANE && shapeTypeB==SHAPE SPHERE)
                return contactPlaneSphere(. . .);
        if (shapeTypeA== SHAPE SPHERE && shapeTypeB==SHAPE SPHERE)
                return contactSphereSphere(. . .);
        . . .
}
```

Breakup into pipeline stages

In some algorithms, we only know the value of the conditional, during the kernel execution. One example is the following algorithm, computing the contacts between a pair of convex collision shapes:

```
bool hasSeparatingAxis = findSeparatingAxis(objectA,objectB)
if (hasSeparatingAxis)
{
        clipContacts(objectA,objectB);
}
```

In this case, we can break up the algorithm into 2 stages, and first execute the 'findSeparatingAxis' stage for all objects. Then we execute the 'clipContacts' stage, only for the objects that have a separating axis.

The output of the first stage could be some array of boolean values. We would like to discard all the work items that have a negative value. Such stream compaction can be done using a parallel prefix scan. Essentially this shrinks the array and only leaves the positive elements. Then we can use this array as input for the next stage of the pipeline.

Use Parallel Primitives

When implementing software for GPU, several patterns or parallel primitives are very common:

- Sorting
- Counting, bound search
- Parallel sum, prefix scan

It is important to have an efficient GPU implementation of those to use as a building block.

Use Local Memory

Most GPUs have a memory hierarchy including

- Global GPU memory that can be accessed by all Compute Units
- Local shared memory, that can be accessed by all threads within one Compute Unit
- Private memory that can only be accessed by a single thread/Work Item

Local shared memory is usually at least one order or magnitude faster than global GPU memory. The use of local shared memory is best when there is data that can be shared by several Work Items in a Work Group. Local shared memory can be useful for read-only input data, but also to improve performance of writing the output data.

Barrier synchronization

In order to make use of local shared memory, we also need to make sure that the data is valid, before any of the Work Items starts accessing the memory. For this we can use a barrier. A very common usage is as follow:

- Copy the data from global GPU memory to local shared memory
- Add a barrier so that all threads are waiting for the data to be valid
- Perform some computation using the local shared memory

Atomics

Atomic operations can be useful in many cases. One way we use atomic operations is to emulate a global append buffer. Any Work Item in any Work Group might want to append some data to a global array. We can create an integer index in global GPU memory. Each thread can use an *atomic_add* operation to append a number of items to the buffer.

GPU rigid body simulation

A lot of work can go into picking or designing an algorithm that is suitable for GPU. In this section we will go more in detail how we implemented each stage of the rigid body pipeline to GPU.

Rigid body introduction

A rigid body is an object that has some properties that generally don't change over time, such as mass and inertia properties and a collision shape representation. A rigid body also has some state variables such as position and orientation and linear and angular velocity that change over time, usually due to some forces such as gravity, collisions or other constraint forces. A 3d rigid body has 6 degrees of freedom for its motion: translation along each of the 3 primary axis (x,y,z) and rotation among those 3 axis.

It is a physics engine task to update the state of rigid bodies according to the Newton laws. At regular intervals it moves the objects and detects collisions between rigid bodies and makes sure that objects don't penetrate. Aside from such non-penetration constraints, there are other constraints that control the degrees of freedom between a pair of rigid bodies.

A rigid body pipeline consists of all consecutive stages of computation that are performed during a single simulation time step.

A collision shape describes the surface or volume of an object. There are many collision shape representations, for example convex shapes such as a sphere, box, cone, cylinder, capsule or a convex hull of some vertices/points. Another popular representation is a triangle mesh geometry. Multiple collision shapes can be grouped into a compound collision shape.

To accelerate collision detection between two collision shapes, we first test if the world space bounding volumes of the shapes overlap. The bounding volume we use is an Axis Aligned Bounding Box or AABB.

In addition to a single world space AABB, for complex shapes we can have an additional local space acceleration structure. We can use a bounding volume hierarchy or BVH such as an AABB tree for this.

Aside from updating the state of all rigid bodies in the world, a physics engine can also provide collision queries and ray intersection queries against the collision shapes of the rigid bodies. For example you can query the closest points or the penetration depth between two objects.

The rigid body pipeline

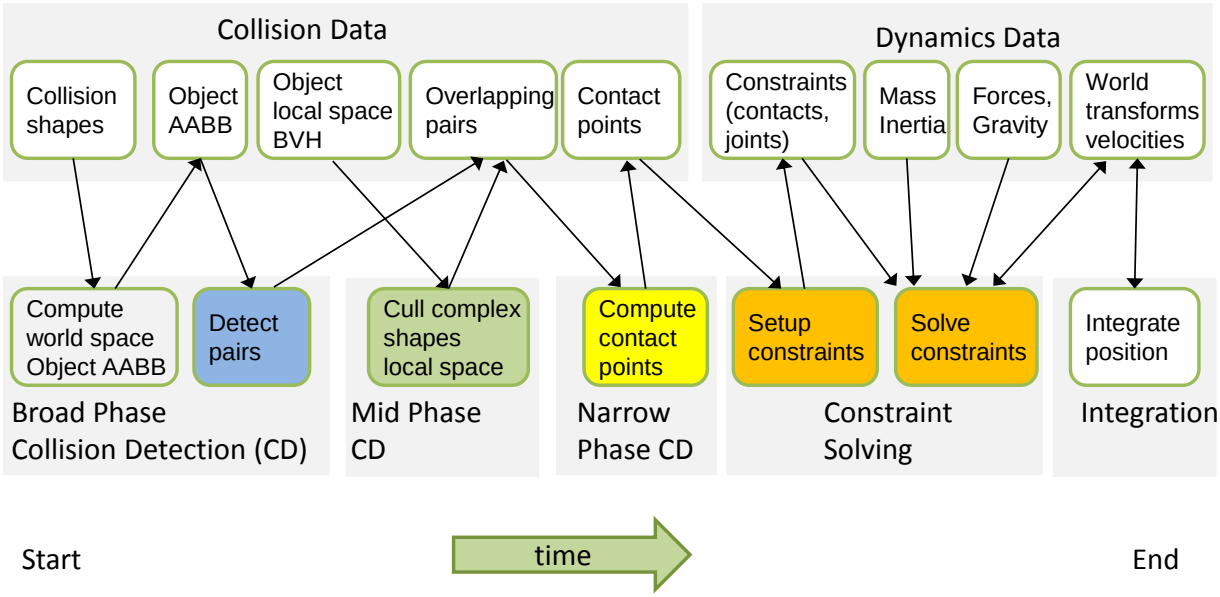
The Rigid Body Pipeline 1 diagram shows the data structures and computation stages of a simplified rigid body pipeline. In a nutshell, we detect potential overlapping pairs in the Detect Pairs stage. Given *n* rigid bodies, this step will reduce the expected time complexity from *O(n*n)* to *O(n)*.

Once we have the potential overlapping pairs, we compute the contact points in the Narrow Phase step. For a pair of overlapping spheres, this step is trivial, but for a pair of convex hull meshes it becomes more complicated. If we deal with large triangle meshes, we usually add an additional culling step, known as the Mid Phase collision detection.

The contact points are converted into contact constraints, so that objects will no penetrate at the end of the simulation step. The contact constraints are satisfied together with non-contact constraints in the Constraint Solving step. We solve all the constraints together, because satisfying one constraint might violate another constraint, so this is a global problem.

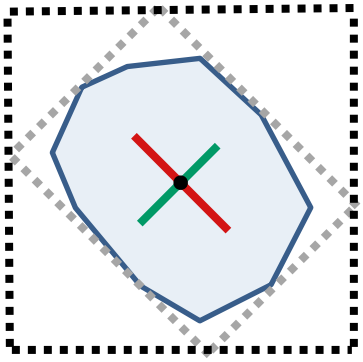
The output of the constraint solving step is updated velocities. Those are used in the Integrate Position step, so update the position and orientation of the rigid bodies.

Rigid Body Pipeline 1



Computing the object AABBs

We need to compute the axis aligned bounding box, AABB, for each object. This is an embarrassingly parallel operation so we won't go in detail. Normally, the collision shape is used to compute the world space AABB. One of the optimizations we made is to cache the local space AABB for the collision shape, the grey dotted box in this figure, and use this to recompute the world space AABB, the black dotted box.



You can find the kernel source code in *src/Bullet3OpenCL/RigidBody/kernels/updateAabbsKernel.cl*

GPU overlapping pair detection

Given all the object axis aligned bounding boxes, we need to find all the object pairs that have overlapping AABBs. The brute force algorithm would perform O(n^2) checks. The original CPU version looks like this:

```
void   computePairsKernelBruteForce (const btAabbCL* aabbs, volatile __global int2*
pairsOut,volatile  __global int* pairCount, int numObjects, int maxPairs)
{
   for (int i=0;i<numObjects;i++)
   {
        for (int j=i+1;j<numObjects;j++)
        {
                if (TestAabbAgainstAabb2GlobalGlobal(&aabbs[i],&aabbs[j]))
                {
                        int2 myPair;
                        myPair.x = aabbs[i].m_objectIndex;
                        myPair.y = aabbs[j].m objectIndex;
                        int curPair = *pairCount;
                        if (curPair<maxPairs)
                        {
                                pairsOut[curPair] = myPair; //flush to main memory
                                pairCount++;
                        }
                }
        }
   }
}
```

When transforming this to an OpenCL kernel, we could drop the outer for-loop, and let individual Work Items deal with each i. This would provide some amount of parallelism:

```
__kernel void   computePairsKernelBruteForceKernel ( __global const btAabbCL* aabbs, volatile
__global int2* pairsOut,volatile  __global int* pairCount, int numObjects, int maxPairs)
{
        int i = get_global_id(0);
        if (i>=numObjects)
                return;
        for (int j=i+1;j<numObjects;j++)
        {
                if (TestAabbAgainstAabb2GlobalGlobal(&aabbs[i],&aabbs[j]))
                {
                        int2 myPair;
                        myPair.x = aabbs[i].m objectIndex;
                        myPair.y = aabbs[j].m_objectIndex;
                        int curPair = atomic_inc (pairCount);
                        if (curPair<maxPairs)
                        {
                                pairsOut[curPair] = myPair; //flush to main memory
                        }
                }
        }
}
```

The global *pairCount* variable keeps track of the output to a global array of overlapping pairs. Each Work Item can use the OpenCL *atomic_inc* operation to increment this variable. The return value of *atomic_inc* is the old value, that we use to store the output. We also need to perform a range check against *maxPairs*, to make sure we don't write beyond the output array.

The use of atomic_inc on a global variable can reduce performance, when many Work Items try to increment the variable at the same time. We can optimize this in various ways. One way is to create a small temporary buffer in local shared memory, and perform local atomic_inc operations on this buffer. Only when the buffer is full, or when the kernel is completed, we can write the results into global

memory. Instead of writing a single result, we can use the atomic_add operation. This use of a global buffer is also known as an Append Buffer.

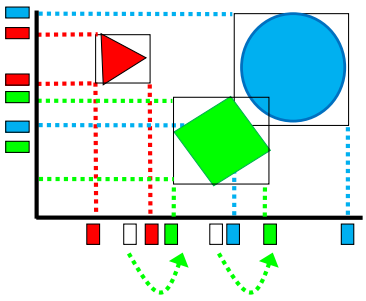
There are many ways to optimize the pair detection, and there are various factors that affect the performance:

- What percentage of the objects are moving?
- How fast do the objects move?
- Do we have large variation in object (bounding volume) sizes?
- Do we add and/or remove many objects?
- Do we want to re-use the acceleration structure for ray intersection queries and swept volume?

One GPU friendly acceleration structure is the uniform grid. This can be parallelized very well, but the drawback is that the collision shapes of rigid bodies can vary a lot in size, so there is not a suitable grid cell size that fits all collision shapes in general. One option is to use a mix of different acceleration structures, a uniform grid for small objects and particles, and a different acceleration structure for larger objects.

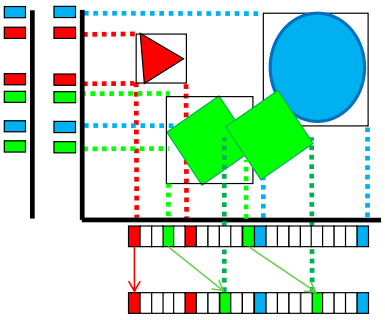
On the CPU, we have incremental algorithms that are unsuitable for GPU in their unmodified form. In Bullet 2.x we have the btDbvtBroadphase, based on a dynamic AABB trees, which are incrementally updated, instead of rebuild from scratch. The incremental updates have a lot of data dependencies and random memory access, so that we cannot easily perform them in parallel. This is still ongoing work.

Another pair search algorithm in Bullet 2.x is the 3-axis sweep and prune. Similar to the previous case, this acceleration structure is incrementally updated with a lot of data dependencies. In this case, however, we came up with some modifications that make it suitable for GPU.



In the original sequential CPU version, the AABB min- and max-coordinates for each axis are move to their new location. Pairs are added or removed during the swap operations that move the AABB coordinate from the old to the new position. The dependencies of data accesses, during this incremental sorting operation, prevent a parallel version.

We modified the incremental algorithm and perform a full sort on each of the 3 axes, with projected AABB begin and end coordinates. In addition, we keep track of the previous sorted arrays for each axis. Then, each object can perform read-only operations that mimic the 'swap' operations to add or remove overlapping pairs: for each object we traverse the original and new arrays from the original index to the new index to determine added or removed pairs:



We also have an implementation of a parallel 1-axis sweep and prune, which performs a full sort on a single axis. The axis with the best variance is computed using a parallel prefix sum. Both the 1-axis and 3-axis sweep and prune algorithm requires a parallel radix sort on floating point values. Our radix sort can only sort arrays of integers, so we need to convert the floating point values to integers first.

You can find the implementation in *src/Bullet3OpenCL/BroadphaseCollision/kernels/sapFast.cl*

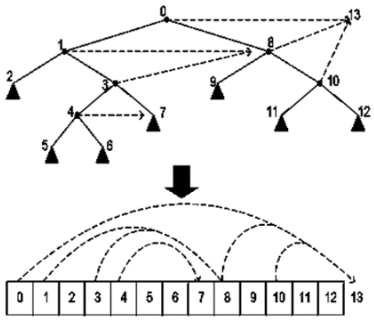
More details about the 1-axis GPU sweep and prune can be found in the paper "Real-time Collision Culling of a Million Bodies on Graphics Processing Units" [Liu 2010]

GPU local space BVH culling for complex shapes

Some of the overlapping pairs contain complex concave shapes that need additional culling before performing exact contact detection. This will effectively simplify overlapping pairs that contain concave triangle meshes and concave compound shapes to new overlapping pairs containing only convex shapes.

We can use a precomputed local space AABB tree to find overlapping child shapes. For a concave triangle mesh shape, the child nodes of the tree refer to individual triangles. For compound collision shapes, the child nodes in the tree refer to individual convex child collision shapes.

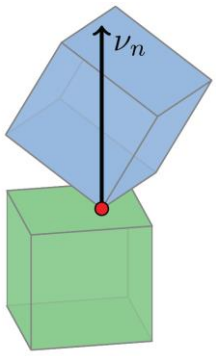
Among the GPU optimizations, we compressed the AABB nodes to 16 bytes using quantization. Additionally, we use a stackless tree traversal, which is very efficient on the GPU.



The implementation is in *src/Bullet3OpenCL/NarrowphaseCollision/kernels/bvhTraversal.cl*

GPU contact computation

Once we finish the pair detection, we have access to overlapping pairs of convex shapes. The algorithm that we use depends on each of the collision shape type in the overlapping pair.

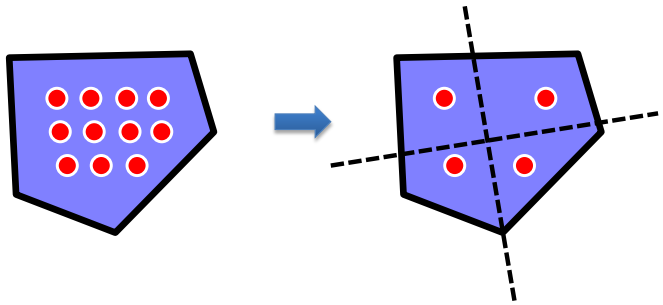


In Bullet 2.x we use the GJK and EPA algorithm to compute contacts between convex shapes, but those algorithms not trivially optimized for GPU.

For Bullet 3.x we compute contacts between convex polyhedral collision shapes using the separating axis test (SAT), contact clipping and contact reduction. The CPU version is very branchy and not suitable for GPU. The pseudo code looks like this:

```
void   computeContactsSAT(convexHullA,convexHullB,transformA,transformB)
{
        b3Vector3 satAxis;
        if (findSeparatingAxis(convexHullA,convexHullB,transformA,transformB))
        {
                if (clipHullHull(satAxis, convexHullA,convexHullB,transformA,transformB))
                {
                        findClippingFacesKernel(. . .)
                        clipFacesAndContact(. . .)
                        contactReduction(...)
                }
        }
}
```

We refactored the code into a collection of kernels that form a pipeline: first we compute the separating axis for all pairs. Once we have the results, we can discard all pairs that don't overlap, using stream compaction (using a prefix sum parallel primitive). The next stage is *clippHullHull* for all pairs in parallel, followed by the *clipFacesAndContactReductionKernel* stage. This way, we the main branches in the original algorithm become kernel executions between the different pipeline stages.

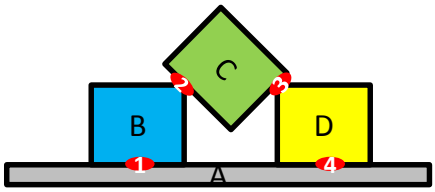


The contact reduction in Bullet 2.x was based on an incremental algorithm that can add or remove one point at a time, using a persistent contact point cache. In Bullet 3.x we compute the full set of contact points, and perform contact reduction on the resulting points.

We also refactored the data structures for convex shapes, faces, edges and vertices in a GPU friendly way.

GPU parallel contact solving

Solving pair-wise constraints means we have to update data for each of the bodies involved. This means that solving multiple constraints is not embarrasingly parallel: they might try to access the same bodies. In the following example, the constraints are numbered 1,2,3,4 and the bodies A,B,C,D.



Constraint 1 and constraint 2 both have read-write access to B, and this means those two constraints cannot be executed in parallel, without synchronization. In order to solve this, we can sort the constraints in independent batches, where the constraints in each batch don't have read-write access to the same bodies.

The pseudo code of the sequential batch sorting looks like this:

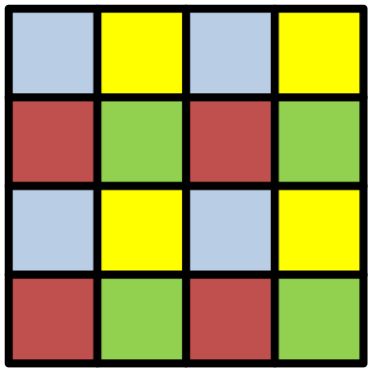
```
void   batchConstraints(constraints, int numConstraints)
{
        int batchIdx=0;
        while( numValidConstraints < numConstraints)
        {
                clear(bodiesUsed);
                for(int i=numValidConstraints; i<numConstraints; i++)
                {
                        int bodyA = constraints[i].m_bodyA;
                        int bodyB = constraints [i].m bodyB;
                        if (isAvailable(bodyA,bodyB,bodiesUsed))
                        {
                                markUsed(bodyA,bodyB,bodiesUsed);
                                constraint[i].m_batchId = batchIdx;//assign the batch index
                        }
                }
                batchIdx ++;
        }
}
```

If we want to move the batch generation onto GPU, there are several considerations. We can separate batch generation in two stages:

1. Local batching: batching within a single compute unit
2. Global batching: split the input so that it can be processed on different compute units

If we only deal with a single compute unit, we could process the batch generation within a single thread/work item. For a compute unit that has 64 threads, 63 threads would be idle, so that is a waste. If we use more then a single work item/thread to perform batch generation, we need to implement a parallel version of 'isAvailable' and 'markUsed'. This can be done using local shared memory, with a small buffer representing the bodies that marked as used. Threads can try to mark bodies as used. After this step, each thread can check if their marked bodies are not overwritten by other constraints. If not, then they succeeded in marking the bodies as used, and the constraint can be added to a batch. Due to lack of local shared memory, we have only limited storage for the marked-as-used bodies. Using a modulo operation, multiple bodies can map to the same 'marked-as-used' storage. A drawback is that some threads might fail to add constraints to a batch, so we need to execute the attempt of body reservation multiple times.

For the global batching among multiple compute units, we cannot rely on synchronization. A way to split the input for global batching is using spatial information: objects that are further away than the maximum object size cannot collide. We can divide space in cells, and solve non-neighboring cells in parallel: the blue cells can be processed in parallel, and the same for the yellow, red and green cells.



Another way to split the input is by creating groups of objects, and compute the pair interactions of the groups carefully to avoid conflicts. A simple example of this would be a grouping in bodies with an odd and even index. The pair search within the ODD group is independent from the EVEN group, so they can be performed in parallel on separate compute units. The pair search between a body of the ODD group and the EVEN group needs to be performed in a later batch. On high-end GPUs we have tens of compute units, so we need to create enough separate groups. Bodies can be added to the same group, if they have the same lower n-bits, creating 2^n groups. Given 2^n groups, we can create a static interaction table to determine what groups can be solved in parallel, similar to the spatial grouping. For more information see "A parallel constraint solver for a rigid body simulation" by Takahiro Harada [Harada 2011].

GPU parallel joint solving

Once the constraints are sorted in independent batches, the constraint solver can solve all constraints in a batch in parallel. The global batching also enables multiple compute units to process constraint rows in parallel. We just have to make sure that we finish all constraints in one batch, before we proceed solving the constraints in the next batch. If we dispatch each batch from the host, the batch execution is synchronized, so the order is guaranteed. Typically we use 4 to 10 PGS iterations, and for each iteration we need to execute the n local batches sequentially. If we have 20 local batches, this would require 200 kernel enqueue (clEnqueueNDRangeKernel) commands. The overhead of the clEnqueueNDRangeKernel can become a bottleneck. This can be avoided by letting the compute unit manage the synchronization of the local batches, using local atomic operations. Each work item keeps on solving constraints, as long as the batch index is the same as the current batch index in local shared memory.
Here is an OpenCL code snippet:

```
while (ldsCurBatch < maxBatch)
{
        for(; idx<end; )
        {
                if (gConstraints[idx].m_batchIdx == ldsCurBatch)
                {
                        solveContactConstraint( gBodies, gShapes, &gConstraints[idx] );
                        idx+=64;
                } else
                {
                        break;
                }
        }
        barrier(CLK_LOCAL_MEM_FENCE);

        if( lIdx == 0 )
        {
                ldsCurBatch++;
        }
        barrier(CLK_LOCAL_MEM_FENCE);
}
```

We also implemented a parallel GPU Jacobi solver [Tonge 2012] but the convergence was not as good as the Projected Gauss Seidel algorithm.

GPU deterministic simulation

The work items and compute units in a GPU are executed in parallel, and the order in which work items are executed can be different each time. This non-determinism, or lack of consistency, can affect the results. For instance, if the pair search appends pairs using an *atomic_inc* operation, there is no guarantee that pairs are inserted in the same order.

If we have a different order of overlapping pairs, and contact points, we may also have a different order of contact constraints. The Projected Gauss Seidel algorithm produces different results, if the constraint rows are solved in a different order. If we want the same results each time we run the simulation (on the same hardware/compiler) we need to make sure that the order is always the same.

We can sort the overlapping pairs, or contact points, using a parallel radix sort. In a similar way, we need to sort the output that is generated during parallel tree traversals for complex concave collision shapes.

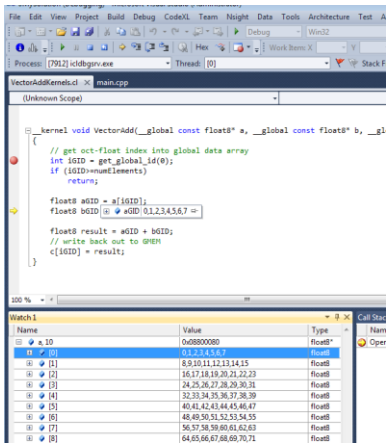
Debugging and Performance Profiling

Debug on the CPU

Debugging OpenCL kernels is much harder than debugging regular CPU code. So the best way is to keep a duplicate implementation running on CPU, and do the debugging and development there first.

Intel OpenCL debugger

Intel provides an OpenCL debugger that allows us to step through the OpenCL kernels, when using a CPU OpenCL Device. Some of our kernels don't run on CPU though, because of minimum Work Group size requirements. Here is a screenshot:



printf debugging

If the CPU version is working all fine, but the GPU version doesn't, we need other tools. One of them is *printf* debugging, from within an OpenCL kernel. Not all OpenCL implementations support *printf* debugging, it depends on the version and the vendor.

Debug buffers

Another way to debug the OpenCL kernels is to add additional kernel buffers, especially for debugging. This way, we can inspect any data, after the kernel finishes execution.

Debugging a frozen system

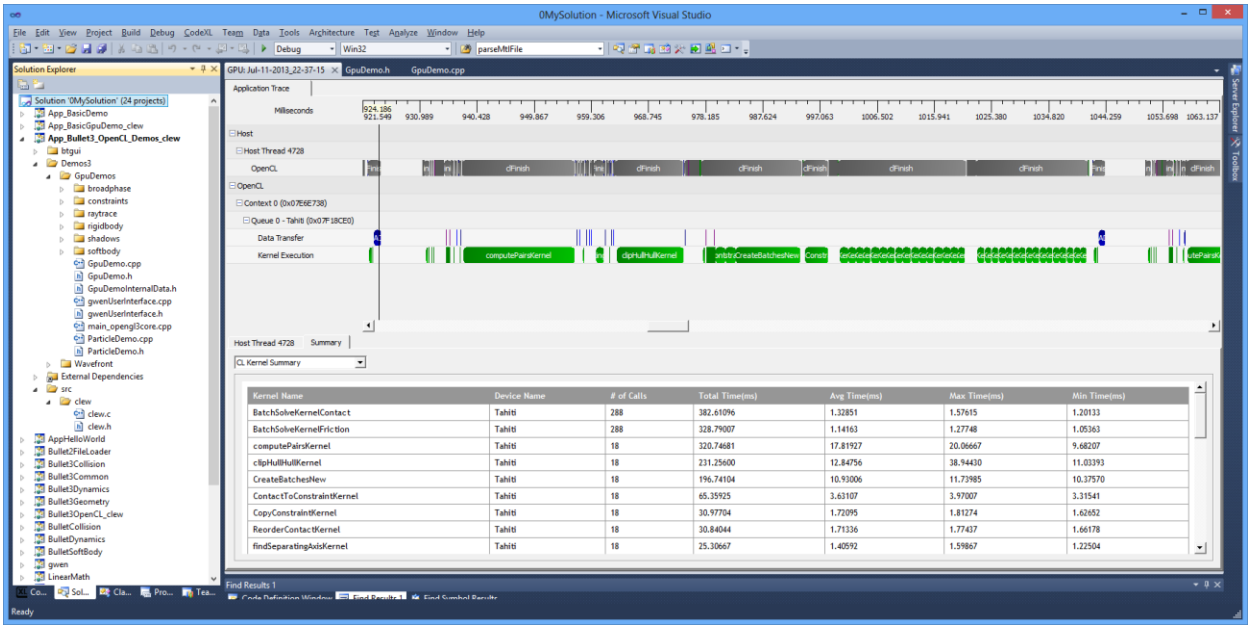
Sometime we encounter a system hang during testing of OpenCL kernels. In most cases on Windows PC, there is some mechanism that resets the GPU when this happens, so we can continue our work. In other cases, especially on Mac OSX, the entire operating system can freeze when an issue happens. We need to figure out what kernel causes the hang, and to do this, we add instrumentation that output some message to the console or to a file. This instrumentation can be added to an OpenCL wrapper, or it can be added to performance profiling zones.

Profile Zones

We add profile zones to our code on the CPU side, to get a course-grain hierarchical profile timings. For more detailed GPU performance statistics we use either CodeXL or NVIDIA NSight.

CodeXL Performance Profiler

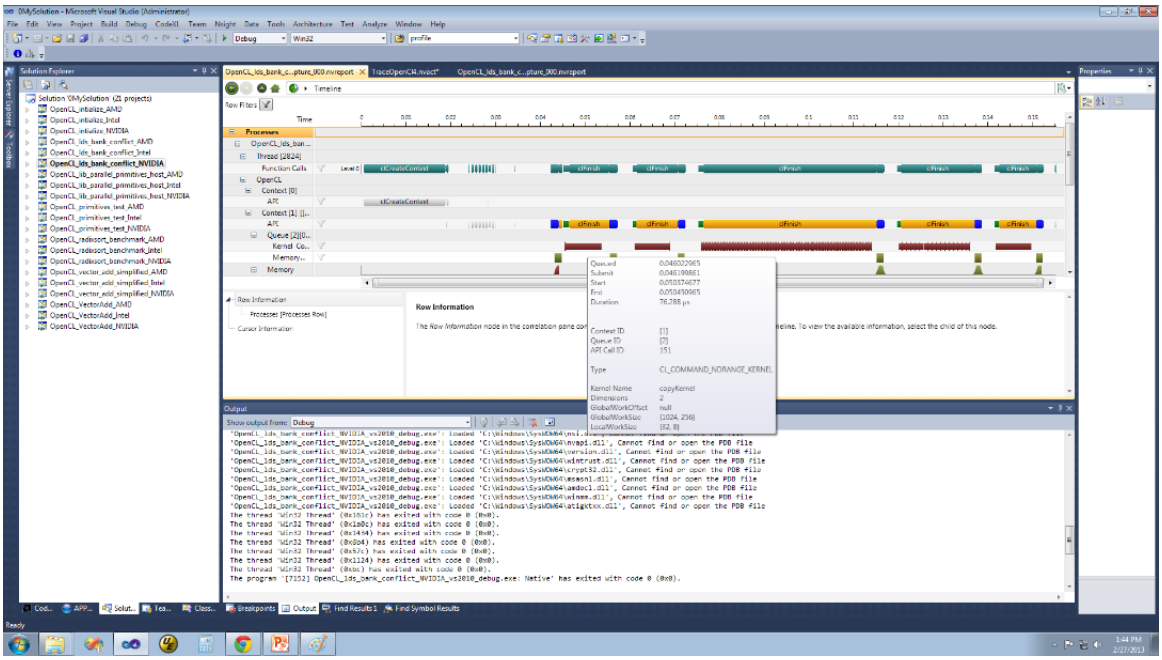
AMD CodeXL also contains a performance profiler that gives a lot of profile information:



Kernel Name	Device Name	# of Calls	Total Time(ms)	Avg Time(ms)	Max Time(ms)	Min Time(ms)
BatchSolveKernelContact	Tahiti	288	382.61096	1.32851	1.57615	1.20133
BatchSolveKernelFriction	Tahiti	288	328.79007	1.14163	1.27748	1.05363
computePairsKernel	Tahiti	18	320.74681	17.81927	20.06667	9.68207
clipHullHullKernel	Tahiti	18	231.25600	12.84756	38.94430	11.03393
CreateBatchesNew	Tahiti	18	196.74104	10.93006	11.73985	10.37570
ContactToConstraintKernel	Tahiti	18	65.35925	3.63107	3.97007	3.31541
CopyConstraintKernel	Tahiti	18	30.97704	1.72095	1.81274	1.62652
ReorderContactKernel	Tahiti	18	30.84044	1.71336	1.77437	1.66178
findSeparatingAxisKernel	Tahiti	18	25.30667	1.40592	1.59867	1.22504

NVIDIA NSIGHT Profiler

The NVIDIA NSight can also show performance profile for OpenCL kernels:



OpenCL Tips and Tricks

Here are a few practical tips from our experience with OpenCL:

Create your own OpenCL wrapper

OpenCL is a low level API and it can be cumbersome to use. With very little effort, you can create your own wrapper to make it much easier to use. There are several benefits of writing your own wrapper, rather than using an of-the-shell wrapper. First of all, you can make it fit very well with your own coding style. Secondly, it is a very good learning experience to learn the details of the low-level OpenCL API. Third, you can add extra features in your wrapper library. We implemented several of the other tips and tricks using our wrapper.

Dynamically load OpenCL

If you link against an OpenCL SDK from a vendor, your program will abort if the user doesn't have any OpenCL driver installed. You can deal with this in a better way so that the program can continue running, even if the OpenCL driver is missing. You can to load the OpenCL dynamic library at run-time and bind against the API dynamically using the clew library. You can download it from https://code.google.com/p/clew

Cache the precompiled OpenCL kernel binaries

The compilation of OpenCL kernels normally happens after you start running your program. This kernel compilation can take a lot of time, so if you have many kernels it is better to store the compiled kernels to disk, and load the binaries.

Keep a host implementation of your kernel

Debugging OpenCL kernels can be very hard and time consuming. We find it very useful to first implement a host implementation, and maintain it even after you get the OpenCL kernel up and running. It can be very hard to locate a bug in a program with many OpenCL kernels, and if you can enable/disable OpenCL kernels individually, it makes bugs easier to find and fix.

Unit test an OpenCL kernel

We added the option to serialize all the input and output data of an OpenCL kernel, so we can debug the kernel outside of our program. Sometimes it takes a lot of time and effort to reproduce a bug, and with this serialization effort, we can directly run a buggy kernel separately. This makes life much easier.

References

Tonge 2012, "Mass splitting for jitter-free parallel rigid body simulation", Richard Tonge et al., SIGGRAPH 2012, http://dl.acm.org/citation.cfm?id=2185601 and http://www.richardtonge.com

Harada 2011, "A parallel constraint solver for a rigid body simulation" by Takahiro Harada. SIGGRAPH Asia 2011 Sketches. http://dl.acm.org/citation.cfm?id=2077406

Liu 2010, "Real-time Collision Culling of a Million Bodies on Graphics Processing Units", SIGGRAPH Asia 2010, http://graphics.ewha.ac.kr/gSaP

Coumans 2009, "OpenCL Game Physics", Erwin Coumans, NVIDIA Game Technology Conference 2009, http://www.nvidia.com/content/GTC/documents/1077_GTC09.pdf

Harada 2007, "Real-Time Rigid Body Simulation on GPUs" Chapter 29 in the GPU Gems 3 book, http://http.developer.nvidia.com/GPUGems3/gpugems3_ch29.html

Appendix A: Bullet 3.x Source code

The source code of Bullet 3.x is available under a permissive zlib/BSD style license on Github at http://github.com/erwincoumans/bullet3.

Requirements

The code is tested on Windows 7/8, Linux and Mac OSX desktops with a recent high-end GPU such as an AMD Radon 7970 or AMD W9000, or an NVIDIA GTX 680. Most laptop GPUs are too slow and don't have enough memory for this project to be useful.

Building on Windows using Visual Studio

You can click on build3/vs2010.bat
Open Bullet3/build3/vs2010/0MySolution.sln

Building on Linux (or Mac OSX) using gcc

Open a terminal

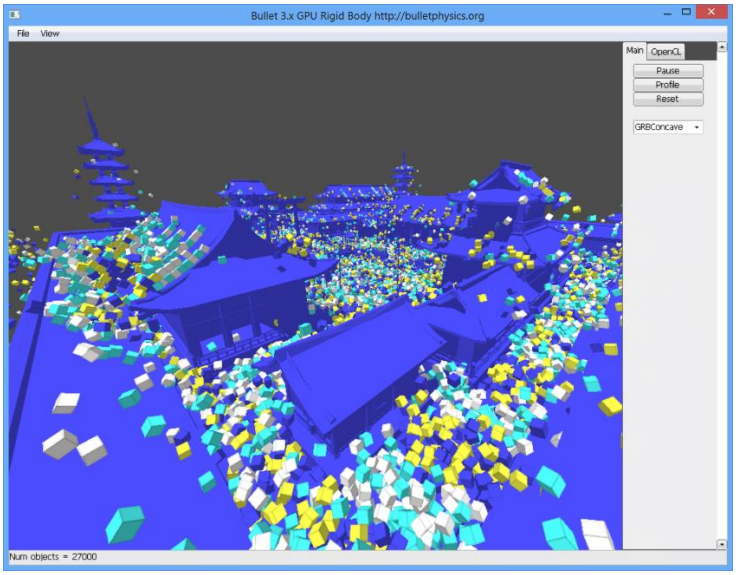
```
cd build3
./premake_linux64 gmake
cd gmake
make
```

Building on Max OSX using XCode

Click on build3/xcode.command and open build3/xcode4/0MySolution.xcodeproj

Usage

You can execute the demo application from the bin directory. It looks like this:



Benchmark

The demo includes a benchmark mode that export a comma separated file (for Excel)
bin/App_Bullet3_OpenCL_Demos_clew_vs2010 --benchmark
You can use the F1 key to create a screenshot and the Escape key will terminate the demo.

Feedback

Although the new Bullet 3.x OpenCL rigid body work is still work-in-progress, it can already be useful for VFX projects that need to simulate a large amount of bodies on a single desktop computer.

If you have any feedback about the software, please contact the author at erwin.coumans@gmail.com or visit the Bullet physics forums at http://bulletphysics.org