OpenCL / OpenGL Vertex Buffer Interoperability: A Particle System Case Study

Also, see the video at:
http://cs.oregonstate.edu/~mjb/cs575/Projects/particles.mp4

Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License
OpenCL / OpenGL Vertex Interoperability: The Basic Idea

Your C++ program writes initial values into the buffer on the GPU

OpenCL acquires the buffer

Each OpenCL kernel reads an (x, y, z) value from the buffer

Each OpenCL kernel updates its (x, y, z) value

Each OpenCL kernel writes its (x, y, z) value back to the buffer

OpenCL releases the buffer

OpenGL draws using the (x, y, z) values in the buffer on the GPU

(x, y, z) Vertex Data in an OpenGL Buffer
In the Beginning of OpenGL …

You listed the vertices with separate function calls:

```c
 glBegin( GL_TRIANGLES );
  glVertex3f( x0, y0, z0 );
  glVertex3f( x1, y1, z1 );
  glVertex3f( x2, y2, z2 );
  glVertex3f( x0, y0, z0 );
  glVertex3f( x3, y3, z3 );
  glVertex3f( x4, y4, z4 );
 glEnd( );
```

Then someone noticed how inefficient that was, for three reasons:

1. Sending large amounts of small pieces of information is less efficient than sending small amounts of large pieces of information
2. The vertex coordinates were being listed in the CPU and were being transferred to the GPU every drawing pass
3. Some vertices were listed twice
Here’s What OpenGL Has Been Moving To: Vertex Buffer Objects

GLfloat CubeVertices[][3] = {
    { -1., -1., -1. },
    {  1., -1., -1. },
    { -1.,  1., -1. },
    {  1.,  1., -1. },
    { -1., -1.,  1. },
    {  1., -1.,  1. },
    { -1.,  1.,  1. },
    {  1.,  1.,  1. }
};

GLfloat CubeColors[][3] = {
    { 0., 0., 0. },
    { 1., 0., 0. },
    { 0., 1., 0. },
    { 1., 1., 0. },
    { 0., 0., 1. },
    { 1., 0., 1. },
    { 0., 1., 1. },
    { 1., 1., 1. }
};

GLuint CubeIndices[][4] = {
    { 0, 2, 3, 1 },{ 4, 5, 7, 6 },{ 1, 3, 7, 5 },{ 0, 4, 6, 2 },{ 2, 6, 7, 3 },{ 0, 1, 5, 4 }
};
A Little Background -- the OpenGL Rendering Context

The OpenGL Rendering Context contains all the characteristic information necessary to produce an image from geometry. This includes transformations, colors, lighting, textures, where to send the display, etc.

If we were implementing the OpenGL state as a C++ structure (which we’re not), we might do something like this:

```c
struct context {
    float [4][4] Transformation;
    struct Texture * Texture0;
    struct DataArrayBuffer * ArrayBuffer;
    ...
} Context;
```
In C++, objects are pointed to by their address.

In OpenGL, objects are pointed to by an unsigned integer handle. You can assign a value for this handle yourself (not recommended), or have OpenGL generate one for you that is guaranteed to be unique. For example:

```c
GLuint buf;
glGenBuffers( 1, &buf );
```

This doesn’t actually allocate memory for the buffer object yet, it just acquires a unique handle. To allocate memory, you need to bind this handle to the Context.
More Background – What is an OpenGL “Object”? 

An OpenGL Object is pretty much the same as a C++ object: it encapsulates a group of data items and allows you to treat them as a unified whole. For example, a Data Array Buffer Object could be defined in C++ by:

```c
struct DataArrayBuffer {
    enum dataType;
    void * memStart;
    int memSize;
};
```

Then, you could create any number of Buffer Object instances, each with its own characteristics encapsulated within it. When you want to make that combination current, you just need to point the ArrayBuffer element of the Context to that entire struct (“bind”). When you bind an object, all of its information comes with it.
It's very fast to re-bind a different vertex buffer. It amounts to just changing a pointer.

```c
glBindBuffer( GL_ARRAY_BUFFER, buf );
```
The OpenGL term “binding” refers to “attaching” or “docking” (a metaphor which I find to be more visually pleasing) an OpenGL object to the Context. You can then assign characteristics, and they will “flow” through the Context into the object.

```glsl
void glBufferData(GL_ARRAY_BUFFER, buf);
glBufferData(GL_ARRAY_BUFFER, numBytes, data, usage);
```

Think of it as happening this way:

```glsl
Context.ArrayBuffer.memStart = CopyToGpuMemory(data, numBytes);
Context.ArrayBuffer.memSize = numBytes;
```
More Background -- “Binding” to the Context

When you want to use that Vertex Buffer Object, just bind it again. All of the characteristics will then be active, just as if you had specified them again.

Think of it as happening this way:

```c
float *data = Context.ArrayBuffer.memStart;
```
Vertex Buffers: Putting Data in the Buffer Object

```
glBufferData( type, numBytes, data, usage );
```

*type* is the type of buffer object this is:
GL_ARRAY_BUFFER to store floating point vertices, normals, colors, and texture coordinates

*numBytes* is the number of bytes to store in all. Not the number of numbers, but the number of *bytes*!

*data* is the memory address of (i.e., pointer to) the data to be transferred to the graphics card. *This can be NULL, and the data can be transferred later via memory-mapping.*
Preview: We are going to use a Particle System as a Case Study
glBufferData( type, numbytes, data, usage );

*usage* is a hint as to how the data will be used: GL_xxx_yyy

where *xxx* can be:
- **STREAM** this buffer will be written lots
- **STATIC** this buffer will be written seldom and read seldom
- **DYNAMIC** this buffer will be written often and used often

and *yyy* can be:
- **DRAW** this buffer will be used for drawing
- **READ** this buffer will be copied into
- **COPY** not a real need for now, but someday…

**GL_STATIC_DRAW** is the most common usage
Vertex Buffers: Step #1 – Fill the Arrays

```c
GLfloat Vertices[][3] = {
    { 1., 2., 3. },
    { 4., 5., 6. },
    ...
};

int numVertices = sizeof(Vertices) / (3*sizeof(GLfloat));
```
Vertex Buffers: Step #2 – Create the Buffers and Fill Them

```c
glGenBuffers( 1, &buf );

glBindBuffer( GL_ARRAY_BUFFER, buf );

glBufferData( GL_ARRAY_BUFFER, 3*sizeof(GLfloat)*numVertices, Vertices, GL_STATIC_DRAW );
```
Vertex Buffers: Step #3 – Activate the Array Types That You Will Use

`glEnableClientState( type )`

where `type` can be any of:

- GL_VERTEX_ARRAY
- GL_COLOR_ARRAY
- GL_NORMAL_ARRAY
- GL_TEXTURE_COORD_ARRAY

- Call this as many times as you need to enable all the arrays that you will need.
- There are other types, too.
- To deactivate a type, call:

  `glDisableClientState( type )`
glBindBuffer( GL_ARRAY_BUFFER, buf );
Vertex Buffers: Step #5 – Specify the Data

```c
GLfloat vertex[8] = {0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0};
GLfloat color[8] = {1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0};
```

- `size` is the spatial dimension, and can be: 2, 3, or 4
- `type` can be: `GL_SHORT`, `GL_INT`, `GL_FLOAT`, `GL_DOUBLE`
- `stride` is the byte offset between consecutive entries in the array (0 means tightly packed)
- `rel_address`, the 4th argument, is the relative byte address from the start of the buffer where the first element of this part of the data lives.
The Data Types in a vertex buffer object can be stored either as “packed” or “interleaved”

\[ \text{gl*Pointer( size, type, stride, offset);} \]

*rel_address*, the 4th argument, is the relative byte address from the start of the buffer where the first element of this part of the data lives.

Packed:

\[ \text{glVertexPointer( 3, GL_FLOAT, 3*sizeof(GLfloat), 0 );} \]
\[ \text{glColorPointer( 3, GL_FLOAT, 3*sizeof(GLfloat), 3*numVertices*sizeof(GLfloat));} \]

Interleaved:

\[ \text{glVertexPointer( 3, GL_FLOAT, 6*sizeof(GLfloat), 0 );} \]
\[ \text{glColorPointer( 3, GL_FLOAT, 6*sizeof(GLfloat), 3*sizeof(GLfloat));} \]
Vertex Buffers: Step #6 – Specify the Connections

GLfloat Vertices[][3] =
{
    { x0,  y0,  z0 },
    { x1,  y1,  z1 },
    { x2,  y2,  z2 },
    { x3,  y3,  z3 },
    { x4,  y4,  z4 },
    { x5,  y5,  z5 }
};

int numVertices = sizeof(Vertices) / (3*sizeof(GLfloat));

glDrawArrays( GL_TRIANGLES, 0, numVertices );
Vertex Buffers: Writing Data Directly into a Vertex Buffer

Map the buffer from GPU memory into the memory space of the application:

```c
glBindBuffer( buf, GL_ARRAY_BUFFER );
glBufferData( GL_ARRAY_BUFFER, 3*sizeof(float)*numVertices, NULL, GL_STATIC_DRAW );
float * vertexArray = glMapBuffer( GL_ARRAY_BUFFER, usage );
```

`usage` is an indication how the data will be used:

- **GL_READ_ONLY** the vertex data will be read from, but not written to
- **GL_WRITE_ONLY** the vertex data will be written to, but not read from
- **GL_READ_WRITE** the vertex data will be read from and written to

You can now use `vertexArray[i]` like any other floating-point array.

When you are done, be sure to call:

```c
glUnMapBuffer( GL_ARRAY_BUFFER );
```
 Either OpenGL or OpenCL can use the Vertex Buffer at a time, but not both:
All of this happens on the GPU

Your C++ program writes initial values into the buffer on the GPU

OpenCL acquires the buffer

Each OpenCL kernel reads an \((x,y,z)\) value from the buffer

Each OpenCL kernel updates its \((x,y,z)\) value

Each OpenCL kernel writes its \((x,y,z)\) value back to the buffer

OpenCL releases the buffer

OpenGL draws using the \((x,y,z)\) values in the buffer on the GPU

\((x,y,z)\) Vertex Data in an OpenGL Buffer
1. Program Header

```c
#include <stdio.h>
define _USE_MATH_DEFINES
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <omp.h>

#ifdef WIN32
#include <windows.h>
#endif

#ifdef WIN32
#include "glew.h"
#endif

#include <GL/gl.h>
#include <GL/glu.h>
#include "glut.h"
#include "glui.h"

#include "CL/cl.h"
#include "CL/cl_gl.h"
```
// structs we will need later:

struct xyzw
{
    float x, y, z, w;
};

struct rgba
{
    float r, g, b, a;
};

(size_t) GlobalWorkSize[3]  =  { NUM_PARTICLES, 1, 1 };
(size_t) LocalWorkSize[3]  =  { LOCAL_SIZE, 1, 1 };

GLuint hPobj; // host opengl object for Points
GLuint hCobj; // host opengl object for Colors
struct xyzw * hVel; // host C++ array for Velocities
cl_mem dPobj; // device memory buffer for Points
cl_mem dCobj; // device memory buffer for Colors
dcl_mem dVel; // device memory buffer for Velocities

cl_command_queue CmdQueue;
cl_device_id Device;
cl_kernel Kernel;
cl_platform_id Platform;
cl_program Program;
A Deceptively-Simple Main Program

```c
int main( int argc, char *argv[ ] )
{
    glutInit( &argc, argv );
    InitGraphics(  );
    InitLists(  );
    InitCL(  );
    Reset(  );
    InitGlui(  );
glutMainLoop(  );
    return 0;
}
```
This *must* wait to be called until after a graphics window is open!

Why? Because that’s when a graphics context is created.
void InitCL()
{
    ...

    status = clGetDeviceIDs( Platform, CL_DEVICE_TYPE_GPU, 1, &Device, NULL );
    PrintCLError( status, "clGetDeviceIDs: " );

    // since this is an opengl interoperability program,
    // check if the opengl sharing extension is supported
    // (no point going on if it isn’t):
    // (we need the Device in order to ask, so we can’t do it any sooner than right here)

    if( IsCLExtensionSupported( "cl_khr_gl_sharing" ) )
    {
        fprintf( stderr, "cl_khr_gl_sharing is supported.\n" );
    }
    else
    {
        fprintf( stderr, "cl_khr_gl_sharing is not supported -- sorry.\n" );
        return;
    }
bool IsCLExtensionSupported( const char *extension )
{
    // see if the extension is bogus:
    if( extension == NULL || extension[0] == '\0' )
        return false;

    char * where = (char *) strchr( extension, ' ' );
    if( where != NULL )
        return false;

    // get the full list of extensions:
    size_t extensionSize;
    clGetDeviceInfo( Device, CL_DEVICE_EXTENSIONS, 0, NULL, &extensionSize );
    char *extensions = new char [ extensionSize ];
    clGetDeviceInfo( Device, CL_DEVICE_EXTENSIONS, extensionSize, extensions, NULL );

    for( char * start = extensions ; ; )
    {
        where = (char *) strstr( (const char *) start, extension );
        if( where == 0 )
        {
            delete [] extensions;
            return false;
        }

        char * terminator = where + strlen(extension);  // points to what should be the separator
        if( *terminator == ' ' || *terminator == '\0' || *terminator == '\r' || *terminator == '\n' )
        {
            delete [] extensions;
            return true;
        }
        start = terminator;
    }
}
void InitCL() {
    ...

    // get the platform id:
    status = clGetPlatformIDs( 1, &Platform, NULL );
    PrintCLError( status, "clGetPlatformIDs: " );

    // get the device id:
    status = clGetDeviceIDs( Platform, CL_DEVICE_TYPE_GPU, 1, &Device, NULL );
    PrintCLError( status, "clGetDeviceIDs: " );

    // 3. create a special opencl context based on the opengl context:
    cl_context_properties props[ ] =
    {
        CL_GL_CONTEXT_KHR,  (cl_context_properties) wglGetCurrentContext( ),
        CL_WGL_HDC_KHR,     (cl_context_properties) wglGetCurrentDC( ),
        CL_CONTEXT_PLATFORM, (cl_context_properties) Platform,

        0
    };

    cl_context Context = clCreateContext( props, 1, &Device, NULL, NULL, &status );
    PrintCLError( status, "clCreateContext: " );
For Windows:
```
cl_context_properties props[ ] =
{
   CL_GL_CONTEXT_KHR,  \ (cl_context_properties) wglGetCurrentContext(  ),
   CL_WGL_HDC_KHR,  \ (cl_context_properties) wglGetCurrentDC(  ),
   CL_CONTEXT_PLATFORM, \ (cl_context_properties) Platform,
   0
};
cl_context Context = clCreateContext( props, 1, &Device, NULL, NULL, &status );
```

For Linux:
```
cl_context_properties props[ ] =
{
   CL_GL_CONTEXT_KHR,  \ (cl_context_properties) glXGetCurrentContext(  ),
   CL_GLX_DISPLAY_KHR,  \ (cl_context_properties) glXGetCurrentDisplay(  ),
   CL_CONTEXT_PLATFORM, \ (cl_context_properties) Platform,
   0
};
cl_context Context = clCreateContext( props, 1, &Device, NULL, NULL, &status );
```

For Apple:
```
cl_context_properties props[ ] =
{
   CL_CONTEXT_PROPERTY_USE_CGL_SHAREGROUP_APPLE,  \ (cl_context_properties) kCGLShareGroup,
   0
};
cl_context Context = clCreateContext( props, 0, 0, NULL, NULL, &status );
```
void InitCL() {
    ... 

    // create the velocity array and the opengl vertex array buffer and color array buffer:

delete [ ] hVel;
    hVel = new struct xyzw [ NUM_PARTICLES ];

    glGenBuffers( 1, &hObj );
    glBindBuffer( GL_ARRAY_BUFFER, hObj );
    glBufferData( GL_ARRAY_BUFFER, 4 * NUM_PARTICLES * sizeof(float), NULL, GL_STATIC_DRAW );

    glGenBuffers( 1, &hCobj );
    glBindBuffer( GL_ARRAY_BUFFER, hCobj );
    glBufferData( GL_ARRAY_BUFFER, 4 * NUM_PARTICLES * sizeof(float), NULL, GL_STATIC_DRAW );

    glBindBuffer( GL_ARRAY_BUFFER, 0 ); // unbind the buffer

    // fill those arrays and buffers:

    ResetParticles( );
}
unsigned int Seed;
...
void ResetParticles()
{
    glBindBuffer( GL_ARRAY_BUFFER, hPobj );
    struct xyzw *points = (struct xyzw *) glMapBuffer( GL_ARRAY_BUFFER, GL_WRITE_ONLY );
    for( int i = 0; i < NUM_PARTICLES; i++ )
    {
        points[ i ].x = Ranf( &Seed, XMIN, XMAX );
        points[ i ].y = Ranf( &Seed, YMIN, YMAX );
        points[ i ].z = Ranf( &Seed, ZMIN, ZMAX );
        points[ i ].w = 1.;
    }
    glUnmapBuffer( GL_ARRAY_BUFFER );
    glBindBuffer( GL_ARRAY_BUFFER, hCobj );
    struct rgba *colors = (struct rgba *) glMapBuffer( GL_ARRAY_BUFFER, GL_WRITE_ONLY );
    for( int i = 0; i < NUM_PARTICLES; i++ )
    {
        colors[ i ].r = Ranf( &Seed, 0., 1. );
        colors[ i ].g = Ranf( &Seed, 0., 1. );
        colors[ i ].b = Ranf( &Seed, 0., 1. );
        colors[ i ].a = 1.;
    }
    glUnmapBuffer( GL_ARRAY_BUFFER );
    ...
}
Setting the Initial Particle Parameters, II

```c
... delete [ ] hVel;
hVel = new struct xyzw [ NUM_PARTICLES ];
for( int i = 0; i < NUM_PARTICLES; i++ )
{
    hVel[ i ].x = Ranf( &Seed, VMIN, VMAX );
    hVel[ i ].y = Ranf( &Seed, 0., VMAX );
    hVel[ i ].z = Ranf( &Seed, VMIN, VMAX );
    hVel[ i ].w = 0.;
}
```
Setting-up the Device-Side Buffers

void InitCL( )
{
    
    // 5. create the opencl version of the velocity array:

dVel = clCreateBuffer( Context, CL_MEM_READ_WRITE, 4*sizeof(float)*NUM_PARTICLES, NULL, &status );
PrintCLError( status, "clCreateBuffer: " );

    // 6. write the data from the host buffers to the device buffers:

status = clEnqueueWriteBuffer( CmdQueue, dVel, CL_FALSE, 0, 4*sizeof(float)*NUM_PARTICLES, hVel, 0, NULL, NULL );
PrintCLError( status, "clEnqueueWriteBuffer: " );

    // 5. create the opencl version of the opengl buffers:

dPobj = clCreateFromGLBuffer( Context, CL_MEM_READ_WRITE, hPobj, &status );
PrintCLError( status, "clCreateFromGLBuffer (1)" );

dCobj = clCreateFromGLBuffer( Context, CL_MEM_READ_WRITE , hCobj, &status );
PrintCLError( status, "clCreateFromGLBuffer (2)" );

Note: you don’t need an OpenGL-accessible buffer for the velocities. Velocities aren’t needed for drawing. Velocities are only needed to update point positions. The velocity buffer can just be done internally to OpenCL.
void
InitCL( )
{

    ... 

    // 10. setup the arguments to the Kernel object:

    status = clSetKernelArg( Kernel, 0, sizeof(cl_mem), &dPobj );
    PrintCLError( status,"clSetKernelArg (1): " );

    status = clSetKernelArg( Kernel, 1, sizeof(cl_mem), &dVel );
    PrintCLError( status , "clSetKernelArg (2): " );

    status = clSetKernelArg( Kernel, 2, sizeof(cl_mem), &dCobj );
    PrintCLError( status, "clSetKernelArg (3): " );

    ... to Match the Kernel’s Parameter List

    kernel
    void
    Particle( global point * dPobj, global vector * dVel, global color * dCobj )
    {
        ... 
    }
void Animate()
{
    // acquire the vertex buffers from opengl:
    glutSetWindow( MainWindow );
    glutFinish( );
    cl_int status = clEnqueueAcquireGLObjects( CmdQueue, 1, &dPobj, 0, NULL, NULL );
    PrintCLError( status, "clEnqueueAcquireGLObjects (1) : " );
    status = clEnqueueAcquireGLObjects( CmdQueue, 1, &dCobj, 0, NULL, NULL );
    PrintCLError( status, "clEnqueueAcquireGLObjects (2) : " );

    Wait();
    double time0 = omp_get_wtime( );

    // 11. enqueue the Kernel object for execution:
    cl_event wait;
    status = clEnqueueNDRangeKernel( CmdQueue, Kernel, 1, NULL, GlobalWorkSize, LocalWorkSize, 0, NULL, &wait );
    PrintCLError( status, "clEnqueueNDRangeKernel: " );

    Wait();
    double time1 = omp_get_wtime( );
    ElapsedTime = time1 - time0;

    clFinish( CmdQueue );
    clEnqueueReleaseGLObjects( CmdQueue, 1, &dObj, 0, NULL, NULL );
    PrintCLError( status, "clEnqueueReleaseGLObjects (1): " );
    clEnqueueReleaseGLObjects( CmdQueue, 1, &dCobj, 0, NULL, NULL );
    PrintCLError( status, "clEnqueueReleaseGLObject (2) : " );

    glutSetWindow( MainWindow );
    glutPostRedisplay( );
}

The OpenGL “Idle Function” Tells OpenCL to Do Its Computing
Redrawing the Scene:  
The Particles

```c
void Display( )
{
    ...  
    glBindBuffer( GL_ARRAY_BUFFER, hPobj );
    glVertexPointer( 4, GL_FLOAT, 0, (void *)0 );
    glEnableClientState( GL_VERTEX_ARRAY );
    glBindBuffer( GL_ARRAY_BUFFER, hCobj );
    glColorPointer( 4, GL_FLOAT, 0, (void *)0  );
    glEnableClientState( GL_COLOR_ARRAY );
    glPointSize( 2. );
    glDrawArrays( GL_POINTS, 0, NUM_PARTICLES );
    glPointSize( 1. );
    glDisableClientState( GL_VERTEX_ARRAY );
    glDisableClientState( GL_COLOR_ARRAY );
    glBindBuffer( GL_ARRAY_BUFFER, 0 );
    glutSwapBuffers( );
    glFlush( );
}
```
Redraw the Scene: The Performance

```c
void Display( )
{
    ...

    if( ShowPerformance )
    {
        char str[128];
        sprintf( str, "%6.1f GigaParticles/Sec", (float)NUM_PARTICLES/ElapsedTime/1000000000. );
        glDisable( GL_DEPTH_TEST );
        glMatrixMode( GL_PROJECTION );
        gluOrtho2D( 0., 100., 0., 100. );
        glMatrixMode( GL_MODELVIEW );
        glLoadIdentity( );
        gluOrtho2D( 0., 100., 0., 100. );
        glMatrixMode( GL_MODELVIEW );
        glLoadIdentity( );
        glColor3f( 1., 1., 1. );
        DoRasterString( 5., 5., 0., str );
    }
```

Redraw the Scene: The Performance

1.2 GigaParticles/Sec
13. Clean-up

```c
void Quit()
{
    Glui->close();
    glutSetWindow(MainWindow);
    glFinish();
    glutDestroyWindow(MainWindow);

    // 13. clean everything up:
    clReleaseKernel(Kernel);
    clReleaseProgram(Program);
    clReleaseCommandQueue(CmdQueue);
    clReleaseMemObject(dPobj);
    clReleaseMemObject(dCobj);

    exit(0);
}
```
typedef float4  point;
typedef float4  vector;
typedef float4  color;
typedef float4  sphere;

constant float4 G               = (float4) ( 0., -9.8, 0., 0. );
constant float   DT             = 0.1;
constant sphere Sphere1 = (sphere)( -100., -800., 0., 600. );

bool
IsInsideSphere( point p, sphere s )
{
    float r = fast_length( p.xyz - s.xyz );
    return ( r < s.w );
}
kernel
void
Particle(  global point * dPobj,  global vector * dVel,  global color * dCobj )
{
    int gid = get_global_id( 0 );  // particle #

    point p = dPobj[gid];
    vector v = dVel[gid];

    point pp = p + v*DT + G * (point)(.5*DT*DT);  // p'
    vector vp = v + G*DT;  // v'
    pp.w = 1.;
    vp.w = 0.;

    if( IsInsideSphere( pp, Sphere1 ) )
    {
        vp = BounceSphere( p, v, Sphere1 );
        pp = p + vp*DT + G * (point)(.5*DT*DT);
    }

    dPobj[gid] = pp;
    dVel[gid] = vp;
}
vector
Bounce( vector in, vector n )
{
    n.w = 0.;
    n = normalize( n );
    vector out = in - n * (vector)( 2.*dot( in.xyz, n.xyz ) );
    out.w = 0.;
    return out;
}

vector
BounceSphere( point p, vector v, sphere s )
{
    vector n;
    n.xyz = fast_normalize( p.xyz - s.xyz );
    n.w = 0.;
    return Bounce( in, n );
}
Joe Parallel’s Performance

Number of Particles (x1024) vs. GigaParticles / Second