Accessing CUDA features in the OpenGL rendering pipeline: A case study using N-Body simulation

Mario Santos Camillo
School of Electrical and Computer Engineering
University of Campinas
Campinas, São Paulo
Email: mcamillo@dca.fee.unicamp.br

Wu Shin-Ting
School of Electrical and Computer Engineering
University of Campinas
Campinas, São Paulo
Email: ting@dca.fee.unicamp.br

***Abstract*—The advances of the graphics programing unit (GPU) architecture and its rapidly evolving towards general purpose GPU make a series of applications adopt a general purpose (GPGPU) and a graphics computing interoperability approach in which the first is used for heavy calculations and the second for 3D graphics rendering. Because GPGPU exposes several hardware features, such as shared memory and thread synchronization mechanism, it allows a developer to write more efficient code. Nevertheless, we conjecture that such hardware features are also available in the graphics computing interface OpenGL 4.5 or later through the graphics concepts: blending, transform feedback, tessellation and instancing. In this paper we assess our conjecture by implementing an N-body simulation with both approaches. We indeed devise a novel non-graphics application to the tessellation hardware and the instanced rendering circuit. Instead of refining a mesh, we use the abstract patch for gaining direct accesses to shared memory. In the place of drawing multiple objects, we apply the instanced rendering technology for improving sequential data accesses. Comparative timing analysis is provided. We believe that these results provide better understanding of the graphics features that are useful for closing the performance gap between OpenGL and a GPGPU architecture, and open a new perspective on implementing solely with the OpenGL graphics applications that require both intense, but pre-specified, memory accesses and 3D graphics rendering.**

I. Introduction

Since their introduction to the consumer digital graphics market in the 1990s [1] the GPUs (Graphics Processing Units) have evolved from specialized graphics processors to general purpose computing ones. On one hand, these devices are widespread in the scientific community for their number crunching capabilities. On the other hand, programming them becomes a tricky task if the performance is at stake. With this in mind researches have been studying ways to better explore memory hierarchy and parallel execution units in GPU devices when they port known algorithms from CPU to GPU. In general, for graphics rendering applications it is recommended to access 3D graphics specialized hardware through the open standard OpenGL API [2], and for general purpose programming the NVIDIA vendor locked CUDA [3] and the open standard OpenCL [4] APIs have been the preferred ones. This is because CUDA and OpenCL give direct access to the resources of the Compute Unified Device Architecture (CUDA)-enabled GPUs. Since the introduction of OpenGL compute shaders as part of core OpenGL version 4.3, those general purpose resources are also accessible through the OpenGL API to some extent.

Fratarcangeli [5] and Vassilev [6] compared the performance of these APIs in the implementation of a mass-spring model for cloth simulation. They observed that the OpenGL rendering pipeline outperformed CUDA and OpenCL. Their explanation was that the inter-operation between CUDA/OpenCL and OpenGL is the main performance bottleneck and mapping all numerical programming to the OpenGL state machine avoids this inter-operation.

Hunz conducted in [7] an analysis of the performance of OpenGL compute shaders in three applications: N-body simulation, fabric simulations and line detection. He showed that the concepts of compute work group and compute shared memory allow a developer to dispatch programs in a way similar to CUDA and OpenCL. Moreover, they pointed out the impact of the use of shared memory and of CUDA cores in the execution performance. Sans and Carmona [8] assessed the performance of a direct volume ray casting algorithm implemented in CUDA, OpenCL and OpenGL compute shaders. They found that the third programming model constantly delivered the best results while OpenCL usually lagged behind CUDA. These two works reinforce the fact that, when general purpose calculations are required in a rendering, computing them through the OpenGL API may be more efficient.

With the introduction of performance-striving graphics concepts throughout the OpenGL evolution and available in version 4.5, such as blending, transform feedback, tessellation and instancing, we ask ourselves whether we can efficiently access hardware resources through these graphics features. We conducted a case study of the implementation of a heavy, but highly parallelizable, and memory-intense N-body simulation entirely within the OpenGL rendering pipeline. Different graphics features were explored and their performance compared with a well-known efficient CUDA implementation available in the NVIDIA CUDA SDK [9].

Although Fang et al. did in [10] a thorough comparison of the OpenCL and CUDA APIs using 16 benchmarks ranging from real world applications to synthetic ones, and showed that in a fair comparison environment both have very similar performance, we took in this work the CUDA programming model as the time performance reference for non-graphics

computing. This choice was based on two reasons. The first one is that CUDA, being directly under control of NVIDIA and not a committee, is usually quicker than OpenCL in exposing the latest hardware changes to the developers. And the second is that the CUDA programming model exposes more details about the CUDA architecture and is not strictly tied to the GLSL graphics language as OpenGL compute shaders are.

To be self-contained we present in Section II a brief description of CUDA programming model and OpenGL API through which a program interacts with the underlying GPU. In Section III an N-body simulation and its classical implementation with CUDA programming model on the GPU are provided. Section V shows three ways of implementing this simulation problem with the OpenGL API. Timing results with NVIDIA's four GPU architectures, Fermi, Kepler, Maxwell and Pascal, are reported in Section VI. From the results we infer that if we appropriately use the GPU-accelerated features we may achieve a computing performance comparable to that we get with CUDA, as discussed in Section VII. Some concluding remarks are drawn in Section VIII.

Contribution: The main contribution of this work is to present a series of new graphics features available in OpenGL 4.5 from a perspective of non-graphics applications and show their potential in closing the performance gaps between an OpenGL-based rendering pipeline and an optimized CUDA-based implementation through an N-body simulation – a well-known classical highly parallelizable application with intense memory accesses. The results of our study are directly applicable to the performance optimization of shaders for any application.

II. GPU Architectures

In this section we will provide an overview of the CUDA architecture [11] and the graphics pipeline from OpenGL 4.5 [2] focusing on the features we have applied in our work.

CUDA Architecture

The GPU chip consists of a few hundreds to a few thousands of smaller and simple processors, the CUDA cores, that each executes a single thread. The CUDA architecture is built from a scalable array of multithreaded Streaming Multiprocessors (SM) and each SM comprises several CUDA cores. Logically, the threads are organized in blocks of 32, called warps, that are executed simultaneously in SIMD (Single Instruction, Multiple Data) fashion under the control of a warp scheduler. The number of warps that a SM can execute at a given time depends on the number of CUDA cores it has.

The memory architecture of the GPUs is similar to that of a modern multi-core CPU. It comprises the main memory, also called global memory, an off-chip DRAM that has a wide bus but high latency, and two cache levels, L1 and L2. The L1 cache is associated to each SM, while the L2 cache is shared by all SMs. Aside from these two caches that are filled according to the memory access pattern of the programs being executed, the GPUs also offer other kinds of memory to be managed by a developer: the shared memory, the read-only texture cache and the constant cache. The shared memory has an access time compatible to the L1 cache and is shared by all the cores in a SM. To keep the access as quick as possible, there is no special synchronization hardware. Instead, a developer has to take care of synchronization when performing concurrent access to this memory.

Graphics Pipeline

The first versions of the graphics pipeline consisting of non-programmable stages were optimized for vertex transformation and lighting, geometry rasterization, and blending, masking or logic operations over the output pixels [12]. As the GPU evolved, the graphics pipeline has become more flexible providing more programmable stages, also known as shaders. Fig. 1 shows the shaders and transform feedback flow in the rendering pipeline for the OpenGL version 4.5 [2]. The solid blocks represent the required shader stages, while the dotted ones are optional. The vertex and fragment shaders were the first programmable stages. In the vertex shader, per-vertex operations, such as geometric transformations and color assignment, are performed, and in the fragment shader per-pixel modifications are carried out.

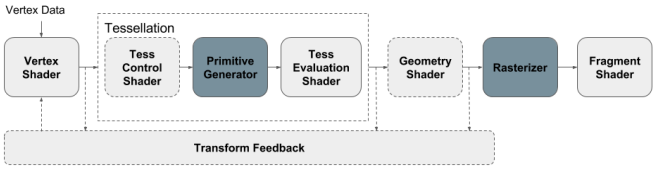


Fig. 1. OpenGL 4.5 rendering pipeline comprising fixed functions in dark gray boxes, programmable functions in light gray boxes and transform feedback for recording intermediary data. Vertex and fragment shaders are mandatory.

The geometry shader has been included in version 3.0 and it is responsible for changing the geometry of the primitive being rendered. The tessellation shader, added in version 4.0, is responsible for tessellating abstract patches. It consists of three sub-stages, two of which are programmable. The tessellation control and evaluation shaders are programmable, being the first one optional, and the primitive generator is fixed. In this work only the tessellation evaluation shader (TES) has been programmed. Finally, the compute shaders allow a developer to dispatch non-graphics programs while still maintaining accesses to OpenGL specific features like texture fetches, uniforms and image load/store. Besides, very useful features that reduce costly CPU–GPU transfers have been added: transform feedback and geometry instancing. Transform feedback outputs directly the results from vertex, tessellation evaluation or geometry shaders to a bound buffer object and resubmits it multiple times to the rendering pipeline without going back to CPU. Geometry instancing, in its turn, provides a way to render multiple copies of the same object with a single draw call.

III. N-Body Simulation

An N-body simulation consists in simulating the gravitational interactions of N bodies with mass m and calculating

their resulting positions and velocities at each iteration.

As shown in Alg. 1 [9] the simulation is highly parallelizable as the calculation of the position (line 11) and the velocity (line 10) for each body B_i is independent from the others, but it requires the positions of all the other bodies B_j, to get the relative displacement r_{ij} (line 2) at each iteration. For the squared displacement, $||r_{ij}||^2$, the gravitational interactions between bodies are pairwise computed (lines 1 to 4). Note that a softening factor s is added to the squared euclidean distance in order to avoid zero division in Alg. 1 as the second for-loop does not avoid interaction computation between the same body ($B_i = B_j$). When this gravitational interaction is multiplied by the mass of the other body $B_j.m$, we get a partial acceleration of the body B_i due to its interaction with B_j. This acceleration is accumulated in the variable a, so that after looping all adjacent B_j of B_i, we have in a the total acceleration of the body B_i (line 8). From a the new velocity $B_i.new\_vel$ (line 10) and the new position $B_i.new\_pos$ (line 11) of the body B_i at instant $t + \Delta\ t$ are estimated. The factor d is used to lessen the effects of the velocity exponential growth.

Algorithm 1 N-Body simulation

1: $interaction(pos_i, pos_j)\{$
2: $\quad r_{ij} = pos_j - pos_i$
3: $\quad return\ \frac{r_{ij}}{(||r_{ij}||^2+s)^{\frac{3}{2}}}$
4: $\}$
5: **for** body B_i in $bodies$ **do**
6: $\quad a = 0$
7: $\quad$ **for** body B_j in $bodies$ **do**
8: $\quad\quad a = a + interaction(B_i.pos, B_j.pos) * B_j.m$
9: $\quad$ **end for**
10: $\quad B_i.new\_vel = (B_i.vel + a * \Delta t) * d$
11: $\quad B_i.new\_pos = B_i.pos + B_i.vel * \Delta t$
12: **end for**

IV. An efficient implementation with CUDA

A highly optimized implementation of an N-Body simulation in CUDA is described in Alg. 2 [9]. The key feature of this implementation is to make the best use of the CUDA memory and cache hierarchy. The positions of N bodies are transferred to GPU as a global memory buffer. The algorithm runs in blocks of p threads, launching $\frac{N}{p}$ blocks in total as illustrated in Fig. 2. Each thread $threadIdx$ in every block $blockIdx$ is responsible for calculating the new velocity and the new position of a single body B_i (line 2).

To avoid the bottleneck created by the random memory accesses, the calculation occurs in two passes. First, each thread loads the data of a single new body from the global memory to a shared memory array (line 5) and waits for synchronization at a barrier. This assures that the total of p bodies B_j are transferred to the shared memory. Then, each thread iterates these p bodies B_j and accumulates their contributions to the acceleration variable a (line 9). This is repeated for $\frac{N}{p}$ times so that at the end we have in a the total acceleration of the body B_i at the instant t. With a, the velocity and position of B_i at the instant $t + \Delta t$ are updated (line 12). The number of threads, p, is one of the tuning parameters for the algorithm and we use $p = 256$ in our experiments as it is suggested to be the best option for $N \geq 4096$ in [9].

For reducing data transfer overhead between the CPU and the GPU in the rendering stage, new generated position and velocity of the bodies to be rendered are directly written into the existing OpenGL buffers, to which CUDA is bound, in a dual-buffering scheme.

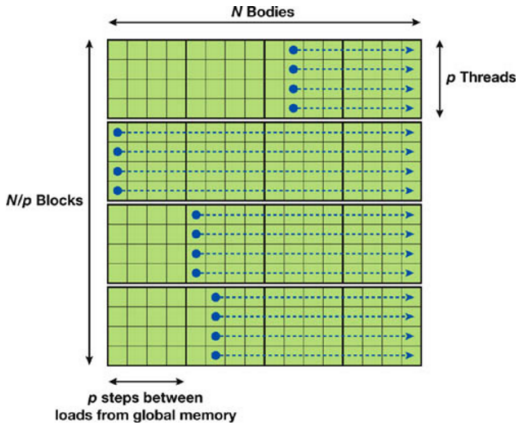


Fig. 2. Blocks and threads execution pattern in CUDA [9].

Algorithm 2 CUDA worker thread algorithm

1: $a = 0$
2: $B_i = fetch(blockIdx * p + threadIdx)$
3: **for** $k = 0$ **to** $\frac{N}{p}$ **do**
4: $\quad$ synchronize thread group
5: $\quad shared[threadIdx] = fetch(k * p + threadIdx)$
6: $\quad$ synchronize thread group
7: $\quad$ **for** $j = 0$ **to** p **do**
8: $\quad\quad B_j = shared[j]$
9: $\quad\quad a = a + interaction(B_i, B_j.pos) * B_j.m$
10: $\quad$ **end for**
11: **end for**
12: $update\_body(threadIdx, a)$

V. Implementations with OpenGL API

To fit into the GPU rendering pipeline and to avoid stall at any shader stage, we devise a two-pass implementation of Alg. 1: the acceleration computation stage (line 8), and the velocity (line 10) and the position (line 11) updates stage. In Fig. 3 we show how these two stages are coupled with a (third) rendering stage for visualizing the simulation results.

In the first pass the numerical simulation is actually processed. Pairwise interactions between all bodies from the input vertex buffer object VBO are calculated and the results are output to a texture memory. Inspired by the work of Olano et al. [13] we propose to enable the blending hardware for accumulating the interactions of each body with all others in its own position in the texture memory.

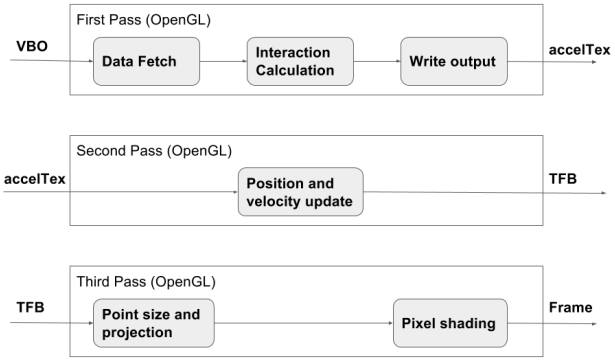


Fig. 3. Three-pass OpenGL-based implementation of an N-body simulation.

The texture position is calculated during the *Data Fetch* step in the vertex shader with the function $idx2tc$ as shown in Alg. 3 (lines 1 to 4). Because we are using the render-to-texture technique to sum the individual accelerations, we must also convert each body's position in the range of $[-1, 1]$ to the texture coordinates in the range of $[0, 1]$ in the *Write Output* step. This conversion is carried out in the $tc2pos$ function (lines 5 to 9 in Alg. 3). In the fragment shader the computed acceleration between a pair of bodies is passed through. Once the blending mechanism is enabled with the blending factor GL_ONE and the blending mode GL_FUNC_ADD ($O = Src + Dst$), the new value Src to be written in the texture memory is added to the existing one, Dst, and this old value is updated with the addition result O.

Algorithm 3 Auxiliary functions

```
1:  vec2 idx2tc(uint index, ivec2 size){
2:    vec2 pos = vec2(index%size.x, index/size.x);
3:    return vec2(pos.x/size.x, pos.y/size.y);
4:  }
5:  vec4 tc2pos(vec2 texCoord){
6:    float x = -1 + 2 * texCoord.x;
7:    float y = -1 + 2 * texCoord.y;
8:    return vec4(x, y, 0, 1);
9:  }
10: vec3 interaction(vec4 a, vec4 b){
11:   vec3 r_ij = b.xyz - a.xyz;
12:   float dist = dot(r_ij, r_ij) + s;
13:   float tmp = inversesqrt(dist);
14:   float invSqrtDist3 = tmp * tmp * tmp;
15:   return r_ij * invSqrtDist3;
16: }
```

In the second pass the velocity and the position of each body at the instant $t+\Delta t$ are updated as shown in Alg. 4. The draw calls of the bodies must be issued again to trigger this second stage. The accelerations computed in the first pass are passed to the second pass through a global texture memory $accelTex$ (line 11). As this second stage only involves vertex processing, we propose to enable the transform feedback buffers, TFB, for capturing the computed new positions $newPos$ (line 17) and new velocities $newVel$ (line 18), and to resubmit them directly to the third pass, the rendering stage, without CPU–GPU transfers.

Algorithm 4 Vertex shader in the 2nd. stage of Fig. 3

```
1:  layout(location = 0) in vec4 oldPos;
2:  layout(location = 1) in vec4 oldVel;
3:  out vec4 newPos;
4:  out vec4 newVel;
5:  uniform ivec2 size;
6:  uniform float dt;
7:  uniform float d;
8:  uniform sampler2D accelTex;
9:  vec3 getAccel(uint idx){
10:   ivec2 texelC = ivec2(idx%size.x, idx/size.x);
11:   return texelFetch(accelTex, texelC, 0).xyz;
12: }
13: main(){
14:   vec3 a = getAccel(gl_VertexID);
15:   vec3 vel = (oldVel.xyz + a * dt) * d;
16:   vec3 pos = oldPos.xyz + vel * dt;
17:   newPos = vec4(pos, oldPos.w);
18:   newVel = vec4(vel, oldVel.w);
19: }
```

For the purpose of showing the impact of GPU-accelerated graphics features on the performance of an N-body simulation, we implemented the *Interaction Calculation* step of the first pass in three different ways: with a geometry shader, with a tessellation evaluation shader and with instancing primitives.

A. Programing Geometry Shader

The first feature we explored in the implementation of the *Interaction Calculation* step is the capacity of the geometry shader to access the data from all vertices of the primitive being rendered, as shown in Alg. 5. We render all the N bodies as a set of N^2 line segments, so that we can process each pair of bodies i and j in a geometry shader by reading the positions of the vertices, $B[0].pos$ and $B[1].pos$, of the input line segment (line 1) and output the accelerations of each body due to their gravitational interactions (line 2). The shader calculates the interaction between the two vertices of the line segment (line 13) and emits the two vertices with the acceleration due to this pairwise interaction (line 16 and 19). Note that the element $B[i].pos.w$ corresponds to the mass of the body i.

B. Programming Tessellation Evaluation Shader

Using the geometry shader we can achieve higher parallelism for an N-body simulation but the sheer number of interactions with its quadratic growth makes it poorly scalable even for the latest GPUs. Looking at the optimized CUDA-based implementation, we observed that its distinguishing feature is the way that the shared memory is exposed to a programmer which facilitates the implementation of 1:m interactions in a single thread.

Algorithm 5 Geometry shader for pairwise interaction

```
1: layout(lines) in;
2: layout(points, max_vertices = 2) out;
3: in VertexData{
4:   vec4 pos;
5:   vec2 texCoord;
6: } B[];
7: out GeomData{
8:   vec3 accel;
9: } gOut;
10: uniform float dt;
11: uniform float s;
12: main(){
13:   vec3 inter = interaction(B[0].pos, B[1].pos);
14:   gl_Position = tc2pos(B[0].texCoord);
15:   gOut.accel = inter * B[1].pos.w;
16:   EmitVertex();
17:   gl_Position = tc2pos(B[1].texCoord);
18:   gOut.accel = −(inter * B[0].pos.w);
19:   EmitVertex();
20: }
```

Our inspiration for an optimized thread execution comes from the work presented by Nießner et al. which tells us that the tessellation shader optimizes the memory accesses through the intense use of shared memory [14]. We grouped the N^2 interaction pairs in patches of 32 mutually excluding bodies, expecting that we may maximize the underlying thread-level parallelism without getting an unyielding amount of threads.

Our proposal is to render N bodies as $\frac{N}{32}$ quad patches with {3,7,3,7} as their outer tessellation levels and {7,3} as their inner tessellation levels. Fig. 4 shows that this kind of tessellation results in 32 tessellation points per patch. Once each tessellation point is associated to a body, this per patch organization is similar to the per block organization in the CUDA implementation described in Section III. Knowing that the acceleration between two identical bodies is zero, we can compute with a tessellation evaluation shader the gravitational acceleration per patch of each body by iterating all the 32 bodies in the patch.

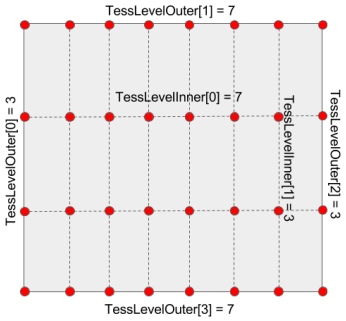


Fig. 4. Quad tessellation for comprising 32 bodies.

Alg. 6 details our proposal. The loop in lines 13 to 16 implements the access to the 32 bodies and the sum of their contributions weighted by their mass $B[j].pos.w$ to the total acceleration a of the current tessellation point. The location of this current point in the patch is given by gl_TessCoord in the range of [0,1]. To obtain the index i of the current tessellation point in the vector B, where the data of the 32 bodies are organized in a row-major order, we must transform $(x, y) \in [0, 1]$ into a $i \in [0, 32)$ array position. Looking at the tessellation points obtained by the quad patch with tessellation levels as described in Fig. 4, we can see that the y axis has 4 equally separated points with distance of $\frac{1}{3}$ while the x axis has 8 equally separated points with distance of $\frac{1}{7}$. Hence, $i = 24y + 7x$ (line 11). It is worth noting that the proposed memory access pattern is similar to the shared memory access pattern described in Alg. 2. The output accumulated acceleration (line 17) passes through the fragment shader and is written in the corresponding position in the $accelTex$ texture.

Algorithm 6 TES for pairwise interaction

```
1: layout(quads, equal_spacing, cw, point_mode) in;
2: uniform float s;
3: in VertexData{
4:   vec4 pos;
5:   vec2 texCoord;
6: } B[];
7: out TEData{
8:   vec3 accel;
9: } tOut;
10: main(){
11:   int i = 24 * gl_TessCoord.y + 7 * gl_TessCoord.x;
12:   vec3 a = vec3(0);
13:   for (j = 0; j < 32; j++){
14:     vec3 inter = interaction(B[i].pos, B[j].pos);
15:     a = a + inter * B[j].pos.w;
16:   }
17:   tOut.accel = a;
18:   gl_Position = tc2pos(B[i].texCoord);
19: }
```

C. Programming Tessellation Shader with Instancing

Even with the use of the shared memory via tessellating primitives, we observed that the performance of an OpenGL-based implementation is still not competitive with the optimized CUDA-based implementation. Further careful comparative analysis led us to detect a subtle difference between the CUDA-based implementation and our tessellation-based implementation. Instead of $(\frac{N}{p} \times \frac{N}{p})(p \times p)$ memory access pattern described in Section V-B, the pattern adopted in the CUDA-based implementation is $\frac{N}{p}(p \times (\frac{N}{p} \times p))$ as sketched in Fig. 2. Although the quantities of processed bodies are the same, the number of memory accesses varies largely between the two approaches.

By observing the way that instanced rendering renders multiple instances in a single draw call, we came to the idea

of applying this rendering mode to iterate all the quad patches described in Section V-B. The key to our proposal is to scan all the $\frac{N}{32}$ instanced patches by an instanced rendering of a single patch. To cover all $\frac{N}{32}$ patches we should issue $\frac{N}{32}$ instanced draw calls as illustrates Fig. 5. This scheme makes our memory access pattern become $\frac{N}{32}(32 \times (\frac{N}{32} \times 32))$, approaching the CUDA-based implementation depicted in Fig. 2.

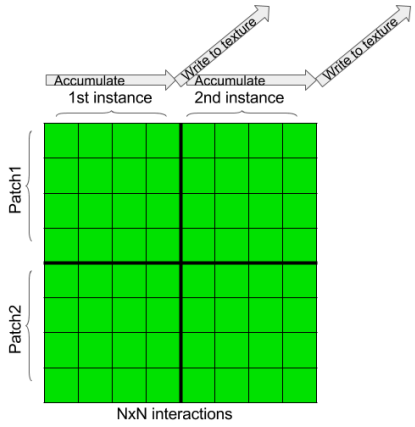


Fig. 5. Patches and instanced patches pattern.

For computing the gravitational interactions between every body i of the currently rendered patch and every body j of an instanced patch in a tessellation evaluation shader (lines 14–17 in Alg. 8), we need to have access to the data of both patches. We propose to fetch these data from the vertex buffer object VBO in the vertex shader with use of the index of the current vertex $gl\_VertexID$ and the index of the current instanced patch $gl\_InstancedID$ (line 15 in Alg. 7). In this way each vertex of a patch will be responsible for fetching a single body data of the instanced patch and saving it on $B.pos2$.

Note that, similar to Alg. 6, the output of the tessellation shader is the accumulated accelerations of the body i with 32 bodies (line 18 in Alg. 8), and, differently from Alg. 6, these 32 bodies belong to the instanced patch and not to the rendered patch where the current tessellating point lies.

VI. Results

We propose in Section V a way that we can gradually enhance the graphics features in the OpenGL-based implementation of an N-body simulation such that memory access patterns become as close as possible to those of the CUDA (Alg. 1). We only modify the *Interaction Calculation* step of the code for each implementation case presented in Section V. In this section we assess how effective is our proposal in making the two programming models rival in time performance.

TABLE I
Specification of the evaluated architectures

GPU/CPU	RAM	NVIDIA GeForce	# of cores
Fermi/core i5	8 GB	540M 1 GB VRAM	96
Kepler/core i7	16 GB	GTX Titan 6 GB VRAM	2688
Maxwell/Xeon	16 GB	GTX Titan X 12 GB VRAM	3072
Pascal/core i7	16 GB	GTX 1080 8 GB VRAM	2560

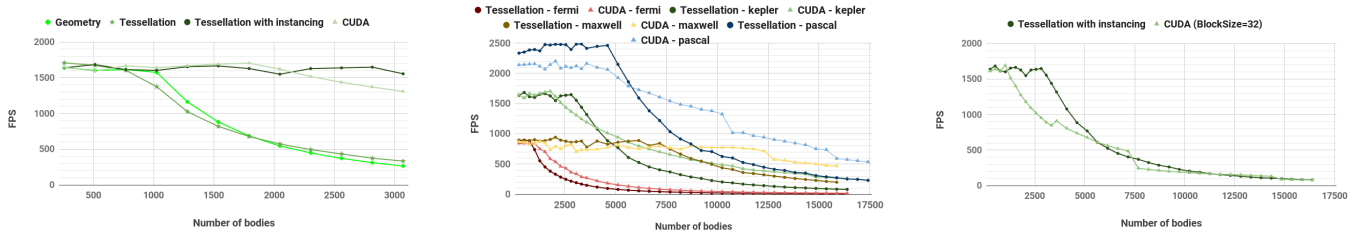
Algorithm 7 Vertex shader for looping patches

```
1:  layout(location = 0) in vec4 pos;
2:  layout(location = 1) in vec4 vel;
3:  layout(std140, binding = 0) buffer PosBuffer{
4:    vec4 positions[];
5:  };
6:  out VertexData{
7:    vec4 pos;
8:    vec4 pos2;
9:    vec2 texCoord;
10: } B;
11: uniform ivec2 size;
12: main(){
13:   B.pos = pos;
14:   extraIdx = 32 * gl_InstanceID + gl_VertexID%32;
15:   B.pos2 = positions[extraIdx];
16:   B.texCoord = idx2tc(gl_VertexID);
17: }
```

Algorithm 8 TES for per-patch pairwise interactions

```
1:  layout(quads, equal_spacing, cw, point_mode) in;
2:  uniform float s;
3:  in VertexData{
4:    vec4 pos;
5:    vec4 pos2;
6:    vec2 texCoord;
7:  } B[];
8:  out TEData{
9:    vec3 accel;
10: } tOut;
11: main(){
12:   int i = 24 * gl_TessCoord.y + 7 * gl_TessCoord.x;
13:   vec3 a = vec3(0);
14:   for (j = 0; j < 32; j++){
15:     vec3 inter = interaction(B[i].pos, B[j].pos2);
16:     a = a + inter * B[j].pos2.w;
17:   }
18:   tOut.accel = a;
19:   gl_Position = tc2pos(B[i].texCoord);
20: }
```

We devised two tests, one for evaluating the performance improvement as graphics features are integrated and the other for evaluating the scalability of our proposal in NVIDIA's four GPU architectures presenting distinct number of cores and memory size: Fermi, Kepler, Maxwell and Pascal (Table I). The sample code of an N-body simulation that comes with the NVIDIA CUDA SDK [9] can run either on the GPU or on the CPU and can display the amount of FPS (frames per second) with an update rate of one second. Because the machines where we conducted our experiments have distinguishing software configuration, the function queryperformancecounter() on the Windows platform and the function gettimeofday() on Linux have been used to get the elapsed times. For enhancing



(a) Graphics features (b) Graphics architectures (c) Same architecture and blocksize

Fig. 6. Performance comparisons of OpenGL and CUDA in *Interaction Calculation*: (a) 4 implementations on the NVIDIA Kepler GPU: CUDA with blocksize=256 (Section IV), Geometry Shader (Section V-A), Tessellation Evaluation Shader (Section V-B), and Tessellation Evaluation Shader with instancing with blocksize=32 (Section V-C); (b) CUDA with blocksize=256 and Tessellation with instancing implementation with blocksize=32 on four NVIDIA GPU architectures: Fermi, Kepler, Maxwell and Pascal; and (c) CUDA and Tessellation with instancing, both with blocksize=32, on the NVIDIA Kepler GPU.

the representation of measured execution times, the average FPS over 200 samples were used in the plots of Fig. 6.

In the first test we run all the four different N-body simulation implementations (one CUDA-based and three OpenGL-based) on the NVIDIA Kepler GPU to assess the difference in performance that each OpenGL feature causes and how close it can approach the CUDA-based performance. The performance (FPS $\times$ number of bodies) plots are shown in Fig. 6a. From these plots we observe that the geometry and the tessellation features do not scale well and quickly degrade even with very small numbers of bodies, while CUDA-based implementation and the implementation that combines the tessellation and instanced rendering features are able to maintain a good performance even when the number of bodies grows. We can also see that the tessellation plus instanced rendering algorithm is the one that presents the closest performance to the CUDA-based algorithm performance.

In the second test we compare on four NVIDIA GPUs the performance of GPU-accelerated tessellation mechanism combined with the instanced rendering in relation to the CUDA implementation. We would not only assess the influence of the evolution of GPU hardware in the performance of rendering pipeline for non-graphics processing, but also the scalability of the OpenGL and CUDA programming approaches on different GPU architecture as well. The performance (FPS $\times$ number of bodies) graphs are drawn in Fig. 6b. There are small hiccups in the graphs due to various uncontrolled background CPU workloads once our measurement procedure does not rely on the CPU time. It is interesting to observe that OpenGL-based implementation presents better performance than the CUDA-based one for small numbers of bodies, but it degrades more rapidly than the CUDA-based implementation as the number grows. Also note that the NVIDIA Maxwell GPU, despite its lower performance, seems to offer better scalability.

VII. Discussions

Remapping non-graphics concepts to graphics concepts, such as a weighted summation onto a GPU-accelerated blending function, in order to maximize performance is not a new concern. Brook [15] is a pioneering research project that represents efforts toward using graphics hardware to solve data- and compute-intensive general purpose applications. Nowadays the vendor-locking CUDA API and the open standard OpenCL API are worldwide well-known general purpose programming interface to GPUs. Nevertheless, because that GPUs have specifically been designed for graphics oriented workloads, it has been shown from several studies [5], [6], [8], [10] that an algorithm that perfectly fits the OpenGL API quite often presents better performance than the general purpose API counterpart. And, since version 4.3 some critical hardware features for parallel processing, such as memory hierarchy, thread blocking and block execution synchronization, become available in the core OpenGL as a separate general purpose shader.

As we have seen in Section II, several new graphics concepts have been introduced in parallel, such as transform feedback, tessellation shader and instanced rendering, in order to improve rendering performance. In order to enhance the use of hardware resources of the CUDA-enabled GPUs, we have investigated in this work how these concepts are related with these resources. Based on [14] and our experiments, we came to the conclusion that these concepts are implemented on top of an optimized hardware.

From [14] we inferred that higher tessellating performance is achieved by pre-loading all the tessellating points of a patch into the shared memory in a fashion similar to all the data of a block of threads being fetched in Alg. 2. And, from the results of our empirical timing tests, we raised the hypothesis that the attributes of multiple instances are fetched block-wise from the global memory which increases the data locality. To evaluate our hypothesis, we further implemented different versions of $Interaction\ Calculation$ with gradually enhanced graphics features, as detailed in Section V.

Time performance results on the NVIDIA Kepler GPU summarized in Fig. 6a let us state that our hypotheses meet the experimental outcomes. Optimization strategies we devised for the OpenGL API are enough to reach a performance that is even better than an optimized CUDA-implementation for small numbers of bodies ($N < 3000$). Intriguing is, however, the rapid drop of the performance for larger number of bodies in comparison to the CUDA, as shown in Fig. 6b.. Careful analysis led us to identify the only difference between them:

the number of bodies per group processing. In the OpenGL-based implementation we have 32 bodies per patch and in the CUDA-based implementation we have 256 bodies per block. It is, therefore, expected that the latter presents better performance than the former due to the supported block size.

Although the drop of time performance reinforces our hypotheses, we still decide to compare the performance of two implementations on the NVIDIA Kepler GPU with the same number of bodies per processing group. When we set 32 as the number of bodies per block in the CUDA-based implementation, we obtain a time performance curve quite similar to that we obtained in the OpenGL-based implementation as shown in Fig. 6c. It seems that as long as the number of vertices in each patch is limited to 32 along the OpenGL rendering pipeline, CUDA may present better performance.

Because of the limitation imposed by the hardware of the tessellation shader, the OpenGL-based implementation is less scalable than the CUDA-based implementation. The plots in Fig. 6b corroborate our expectation. It is, however, worth noting that, although presenting lower performance, the NVIDIA Maxwell GPU has better scalability. We conjecture that this behavior is due to a larger number of cores it has in comparison to other GPUs (Table I).

Summarizing, from our case study we may draw a parallel between the OpenGL API and the CUDA-enabled GPU: (1) with the geometry shader we may enhance the parallel computation of a set of vectors instead of parallel processing of single (vertex) vectors; (2) with the tesselation evaluation shader we may optimize the memory accesses once the processing data are supposed to be pre-loaded into the shared memory; and (3) with the instanced rendering we may improve the memory access pattern.

Note that the measurements of elapsed execution times instead of GPU times suffice for the purpose of a comparative study. But, we believe that if we had measured the exact amount of time that the GPU has spent processing the data at each rendering stage, we would have gained better insight into the detailed behavior of the underlying hardware resource and we would have been able to understand some intriguing behaviors, such as small hiccups and unexpected performance drops in the plots presented in Fig. 6.

VIII. Concluding Remarks

Motivated by looking for a solution that closes the performance gaps between an OpenGL-based and an optimized CUDA-based implementation, we presented a novel application of the geometry shader, the tessellation shader and the instanced rendering mechanism available in the OpenGL 4.5 API. With the geometry shader we may increase the number of vectors to be processed in a thread. We showed that, because the tessellation shader pre-loads its patch data in a shared memory, patch primitives can be used to maximize the low-latency memory accesses. We also experimentally demonstrated that, because the data of multiple instances should be block-wise fetched, instancing drawing can be applied to maximize the global memory bandwidth.

Applying our findings in the implementation of a memory-intense, but highly parallelizable, N-body simulation, we got an OpenGL-based implementation that rivals a well optimized CUDA-based version. We believe that these findings are useful for devising new performance optimization strategies for the applications developed with the OpenGL API. Nevertheless, to fully benefit from the hardware resources we should deepen our understanding of some unexpected behavior patterns by profiling our shaders and analyzing probable bottlenecks.

The comparison results of using groups of 32 bodies in both programming models open a new perspective on improving even more the performance. Instead of a single draw call with all $\frac{N}{p}$ instances of patches, we may make $\frac{N}{8p}$ draw calls with 8 instances of patch per call such that we have a total of 256 bodies per call. As further work we would like to assess the performance gain with this rearrangement.

Acknowledgment

This works was supported by FAPESP (grant 2011/02351) and Eldorado Research Institute.

References

[1] "Graphics Processing Unit," https://en.wikipedia.org/wiki/Graphics_processing_unit, 2017, [Online; accessed 23-March-2017].
[2] *OpenGL - Open Graphics Library*, Khronos Group, 8 2014, ver. 4.5.
[3] *CUDA Parallel Computing Platform and Programming Model*, NVIDIA, 9 2016, ver. 8.0.
[4] *OpenCL - Open Computing Language*, Khronos Group, 11 2015, ver. 2.1.
[5] M. Fratarcangeli, "GPGPU Cloth Simulation Using GLSL, OpenCL, and CUDA," in *Game Engine Gems 2*, E. Lengyel, Ed. A K Peters, 2011, ch. 22, pp. 365–378.
[6] T. I. Vassilev, "Comparison of Parallel Algorithms for Modelling Mass-springs Systems with Several APIs on Modern GPUs," in *Proceedings of the 12th International Conference on Computer Systems and Technologies*, ser. CompSysTech '11. New York, NY, USA: ACM, 2011, pp. 204–209.
[7] J. Hunz, "The possibilities of compute shaders: an analysis," June 2013, bachelor's Thesis. [Online]. Available: https://kola.opus.hbz-nrw.de/files/786/JochenHunzBachelorThesis.pdf
[8] F. Sans and R. Carmona, "Volume ray casting using different GPU based parallel APIs," in *XLII Latin American Computing Conference (CLEI)*, Oct 2016, pp. 1–11.
[9] L. Nyland, M. Harris, and J. Prins, "Fast N-Body Simulation with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley Professional, 2007, ch. 31.
[10] J. Fang, A. L. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *International Conference on Parallel Processing*, Sept 2011, pp. 216–225.
[11] NVidia, "NVIDIA GeForce GTX 1080," NVidia Corporation, Tech. Rep., 2016. [Online]. Available: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf
[12] *OpenGL - Open Graphics Library*, Khronos Group, 7 1994, ver. 1.0.
[13] M. Olano, W. Griffin, Y. Wang, and D. Berrios, "Real-Time GPU Surface Curvature Estimation on Deforming Meshes and Volumetric Data Sets," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, pp. 1603–1613, 2012.
[14] M. Nießner, B. Keinert, M. Fisher, M. Stamminger, C. Loop, and H. Schäfer, "Real-time Rendering Techniques with Hardware Tessellation," *Computer Graphics Forum*, 2015.
[15] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," in *ACM SIGGRAPH 2004 Papers*, ser. SIGGRAPH '04. New York, NY, USA: ACM, 2004, pp. 777–786.