OpenGL* Performance Tips: Atomic Counter Buffers versus Shader Storage Buffer Objects

Introduction

OpenGL provides two mechanisms for storing and retrieving data as input and output to shaders. Game developers can choose from either Shader Storage Buffer Objects (SSBOs) or Atomic Counter Buffers (ACBs). This article demonstrates that there are no real performance benefits to using ACBs instead of SSBOs when writing a graphics-intensive game.

Accompanying this article is a simple C++ application that alternates between SSBOs and ACBs. Game developers can see the effect of both methods on rendering performance (milliseconds per frame). While this article refers to graphical game developers, the concepts apply to all applications that use OpenGL 4.3 and higher. The sample code is written in C++ and is designed for Windows 8.1 and 10 devices.

Requirements

The following are required to build and run the example application:

- A computer with a 6th generation Intel® Core™ processor (code-named Skylake)
- OpenGL 4.3 or higher
- Microsoft Visual Studio* 2013 or newer

Atomic Counter Buffers versus Shader Storage Buffer Objects

A SSBO is a buffer object that is used to store and retrieve data within OpenGL, providing a universal mechanism for both input and output to and from shaders. Another storage option is ACBs, OpenGL storage mechanisms that support atomic memory operations.

ACBs have advantages over using SSBOs and we generally recommended their use rather than an SSBO, if you can live within their limitations, as listed on the OpenGL foundation's website https://www.opengl.org/wiki/Atomic_Counter:

- Atomic counters can only be unsigned integers
- They can only be incremented or decremented by one
- Atomic counter memory access is not incoherent, that is, it does not use the normal OpenGL memory model

From a performance perspective, ACBs have no advantage over SSBOs. This is because ACBs are internally implemented as SSBO atomic operations so there are no real performance benefits from utilizing ACBs.

The application accompanying this article demonstrates this by alternating between SSBOs and ACBs while showing the current milliseconds-per-frame and the number of frames-per-second. Pressing the spacebar switches between using SSBOs and ACBs. When this happens, the image animates to indicate that the change has occurred.

Skylake processor graphics

The 6th generation Intel Core processors, also known as Skylake, provide superior two- and three-dimensional graphics performance, reaching up to 1152 GFLOPS. Its multicore architecture improves performance and increases the number of instructions per clock cycle.

The 6th gen Intel Core processors offer a number of new benefits over previous generations and provide significant boosts to overall computing horsepower and visual performance. Sample enhancements include a GPU that, coupled with the CPU's added computing muscle, provides up to 40 percent better graphics performance over prior Intel® Processor Graphics. The 6th gen Intel Core processors have been redesigned to offer higher-fidelity visual output, higher-resolution video playback, and more seamless responsiveness for systems with lower power usage. With support for 4K video playback and extended overclocking, Skylake is ideal for game developers.

GPU memory access includes atomic min, max, and compare-and-exchange for 32-bit floating point values in either shared local memory or global memory. The new architecture also offers a performance improvement for back-to-back atomics to the same address. Tiled resources include support for large, partially resident (sparse) textures and buffers. Reading unmapped tiles returns zero, and writes to them are discarded. There are also new shader instructions for clamping LOD and obtaining operation status. There is now support for larger texture and buffer sizes. (For example, you can use up to 128k x 128k x 8B mipmapped 2D textures.)

Bindless resources increase the number of dynamic resources a shader may use, from about 256 to 2,000,000 when supported by the graphics API. This change reduces the overhead associated with updating binding tables and provides more flexibility to programmers.

Execution units have improved native 16-bit floating-point support as well. This enhanced floating-point support leads to both power and performance benefits when using half precision.

Display features further offer multiplane overlay options with hardware support to scale, convert, color correct, and composite multiple surfaces at display time. Surfaces can additionally come from separate swap chains using different update frequencies and resolutions (for example, full-resolution GUI elements composited on top of up-scaled, lower-resolution frame renders) to provide significant enhancements.

Its architecture supports GPUs with up to three slices (providing 72 EUs). This architecture also offers increased power gating and clock domain flexibility, which are well worth taking advantage of.

Building and Running the Application

Follow these steps to compile and run the example application.

1. Download the ZIP file containing the source code for the example application and unpack it into a working directory.

2. Open the **lesson4_ACBvsSSBO/lesson4.sln** file by double-clicking it to start Microsoft Visual Studio 2013.

3. Select <*Build*>/<*Build Solution*> to build the application.

4. Upon successful build you can run the example from within Visual Studio.

Once the application is running, a main window opens, and you will see an image rendered using SSBOs, along with the performance measurements shown in the Microsoft Visual Studio 2013 console window. Press the spacebar to alternate between SSBOs and ACBs to see the performance difference. When switching modes the image animates to visually indicate a change. Pressing ESC exits the application.

Code Highlights

This example uses separate shaders for ACB and SSBO; their definitions are below. A separate shader is used for animation to show that the application has switched between ACBs and SSBOs.

```
// Fragment shader used for animation gets output color from a texture
static std::string aniFragmentShader =
    "#version 430 core\n"
    "\n"
    "uniform sampler2D texUnit;\n"
    "\n"
    "smooth in vec2 texcoord;\n"
    "\n"
    "layout(location = 0) out vec4 fragColor;\n"
    "\n"
    "void main()\n"
    "{\n"
    "    fragColor = texture(texUnit, texcoord);\n"
    "}\n"
;

// Fragment shader used bor ACB gets output color from a texture
static std::string acbFragmentShader =
    "#version 430 core\n"
    "\n"
    "uniform sampler2D texUnit;\n"
    "\n"
    "layout(binding = 0) uniform atomic_uint acb[" s(nCounters) "];\n"
    "\n"
    "smooth in vec2 texcoord;\n"
    "\n"
    "layout(location = 0) out vec4 fragColor;\n"
    "\n"
    "void main()\n"
    "{\n"
    "    for (int i=0; i<"  s(nCounters) "; ++i) atomicCounterIncrement(acb[i]);\n"
    "    fragColor = texture(texUnit, texcoord);\n"
    "}\n"
;

// Fragment shader used for SSBO gets output color from a texture
static std::string ssboFragmentShader =
    "#version 430 core\n"
    "\n"
    "uniform sampler2D texUnit;\n"
    "\n"
    "smooth in vec2 texcoord;\n"
    "\n"
    "layout(location = 0) out vec4 fragColor;\n"
    "\n"
    "layout(std430, binding = 0) buffer ssbo_data\n"
    "{\n"
    "    uint v[" s(nCounters) "];\n"
    "};\n"
    "\n"
    "void main()\n"
    "{\n"
```

```
    "    for (int i=0; i<" s(nCounters) "; ++i) atomicAdd(v[i], 1);\n"
    "    fragColor = texture(texUnit, texcoord);\n"
    "}\n"
;
```

The shaders are compiled and prepared for use.

```
    // compile and link the shaders into a program, make it active
    vShader     = compileShader(vertexShader,   GL_VERTEX_SHADER);
    anifShader  = compileShader(aniFragmentShader, GL_FRAGMENT_SHADER);
    acbfShader  = compileShader(acbFragmentShader, GL_FRAGMENT_SHADER);
    ssbofShader = compileShader(ssboFragmentShader, GL_FRAGMENT_SHADER);
    aniProgram  = createProgram({ vShader, anifShader });
    acbProgram  = createProgram({ vShader, acbfShader });
    ssboProgram = createProgram({ vShader, ssbofShader });
    aniOffset   = glGetUniformLocation(aniProgram, "offset");                    GLCHK;
    aniTexUnit  = glGetUniformLocation(aniProgram, "texUnit");                   GLCHK;
    acbOffset   = glGetUniformLocation(acbProgram, "offset");                    GLCHK;
    acbTexUnit  = glGetUniformLocation(acbProgram, "texUnit");                   GLCHK;
    ssboOffset  = glGetUniformLocation(ssboProgram, "offset");                   GLCHK;
    ssboTexUnit = glGetUniformLocation(ssboProgram, "texUnit");                  GLCHK;
```

The same for the ACB and the SSBO.

```
    // create and configure the Atomic Counter Buffer
    glGenBuffers(1, &acb);                                                       GLCHK;
    glBindBufferBase(GL_ATOMIC_COUNTER_BUFFER, 0, acb);                          GLCHK;
    glBufferData(GL_ATOMIC_COUNTER_BUFFER, nCounters * 4, NULL, GL_STATIC_COPY); GLCHK;
    glBindBufferBase(GL_ATOMIC_COUNTER_BUFFER, 0, acb);                          GLCHK;

    // create and configure the Shader Storage Buffer Object
    glGenBuffers(1, &ssbo);                                                      GLCHK;
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, ssbo);                         GLCHK;
    glBufferData(GL_SHADER_STORAGE_BUFFER, nCounters * 4, NULL, GL_STATIC_COPY); GLCHK;
    glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, ssbo);                         GLCHK;
    GLuint idx = glGetProgramResourceIndex(ssboProgram, GL_SHADER_STORAGE_BLOCK,
                                           "ssbo_data");                         GLCHK;
    glShaderStorageBlockBinding(ssboProgram, idx, 0);                            GLCHK;

    // configure texture unit
    glActiveTexture(GL_TEXTURE0);                                                GLCHK;
    glUseProgram(aniProgram);                                                    GLCHK;
    glUniform1i(aniTexUnit, 0);                                                  GLCHK;
    glUseProgram(acbProgram);                                                    GLCHK;
    glUniform1i(acbTexUnit, 0);                                                  GLCHK;
    glUseProgram(ssboProgram);                                                   GLCHK;
    glUniform1i(ssboTexUnit, 0);                                                 GLCHK;
```

Lastly, prepare the textures.

```
    // create and configure the textures
    glGenTextures(1, &texture);                                                 GLCHK;
    glBindTexture(GL_TEXTURE_2D, texture);                                      GLCHK;
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);               GLCHK;
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);               GLCHK;
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);          GLCHK;
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);          GLCHK;

    // load texture image
    std::vector<GLubyte> img; unsigned w, h;
    if (lodepng::decode(img, w, h, "sample.png"))                       __debugbreak();

    // upload the non-pow2 image to vram
    glBindTexture(GL_TEXTURE_2D, texture);                                      GLCHK;
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, w, h, 0, GL_RGBA, GL_UNSIGNED_BYTE,
                 &img[0]);                                                      GLCHK;
}
```

When it is time to draw the image the correct shader program is chosen:

```
// GLUT display function.   Draw one frame's worth of imagery.
void display()
{
    // attributeless rendering
    glClear(GL_COLOR_BUFFER_BIT);                                               GLCHK;
    glBindTexture(GL_TEXTURE_2D, texture);                                      GLCHK;
    if (animating) {
        glUseProgram(aniProgram);                                               GLCHK;
        glUniform1f(aniOffset, animation);                                      GLCHK;
    } else if (!selector) {
        glUseProgram(acbProgram);                                               GLCHK;
        glUniform1f(acbOffset, 0.f);                                            GLCHK;
    } else {
        glUseProgram(ssboProgram);                                              GLCHK;
        glUniform1f(ssboOffset, 0.f);                                           GLCHK;
    }
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);                                      GLCHK;
    if (!animating && selector) {
        glMemoryBarrier(GL_ALL_BARRIER_BITS);                                   GLCHK;
    }
    glutSwapBuffers();
}
```

Each time a video frame is drawn, the performance output is updated in the console and the application checks whether the spacebar or ECS has been pressed. Pressing the space bar causes the application to move to the next set of combinations in the array; pressing ESC exits the application. When alternating

between ACB and SSBO, the performance measurements are reset and the image animates as a visual indicator that something changed. If no key was pressed the next frame is rendered.

```
// GLUT idle function.  Called once per video frame.  Calculate and print timing reports
and handle console input.
void idle()
{
    // Calculate performance
    static unsigned __int64 skip;  if (++skip < 512) return;
    static unsigned __int64 start;
    if (!start && !QueryPerformanceCounter((PLARGE_INTEGER)&start)) __debugbreak();
    unsigned __int64 now;
    if (!QueryPerformanceCounter((PLARGE_INTEGER)&now))             __debugbreak();
    unsigned __int64 us = elapsedUS(now, start), sec = us / 1000000;
    static unsigned __int64 animationStart;
    static unsigned __int64 cnt; ++cnt;

    // We're either animating
    if (animating)
    {
        float sec = elapsedUS(now, animationStart) / 1000000.f; if (sec < 1.f) {
            animation = (sec < 0.5f ? sec : 1.f - sec) / 0.5f;
        } else {
            animating = false;
            selector ^= 1; skip = 0;
            cnt = start = 0;
            print();
        }
    }

    // Or measuring
    else if (sec >= 2)
    {
        printf("frames rendered = %I64u, uS = %I64u, fps = %f,
               milliseconds-per-frame = %f\n", cnt, us, cnt * 1000000. / us,
               us / (cnt * 1000.));
        if (swap) {
            animating = true; animationStart = now; swap = false;
        } else {
            cnt = start = 0;
        }
    }

    // Get input from the console too.
    HANDLE h = GetStdHandle(STD_INPUT_HANDLE); INPUT_RECORD r[128]; DWORD n;
    if (PeekConsoleInput(h, r, 128, &n) && n)
        if (ReadConsoleInput(h, r, n, &n))
            for (DWORD i = 0; i < n; ++i)
                if (r[i].EventType == KEY_EVENT && r[i].Event.KeyEvent.bKeyDown)
                    keyboard(r[i].Event.KeyEvent.uChar.AsciiChar, 0, 0);

    // Ask for another frame
    glutPostRedisplay();
}
```

Conclusion

The OpenGL foundation recommends using ACBs over SSBOs for various reasons; however improved performance is not one of them. This is because ACBs are internally implemented as SSBO atomic operations; therefore there are no real performance benefits from utilizing ACBs. The decision comes down to whether you can live with the limitations of ACBs.

By combining this technique with the advantages of the 6th generation Intel Core processors, graphic game developers can ensure their games perform the way they were designed.

Download Code Sample

Below is the link to the code samples on Github
https://github.com/IntelSoftware/OpenGLBestPracticesfor6thGenIntelProcessor

References

An overview of the 6th generation Intel® Core™ processor (code-named Skylake)

Graphics API Developer's Guide For 6th Generation Intel Core Processors

About the Author

Praveen Kundurthy works in the Intel® Software and Services Group. He has a master's degree in Computer Engineering. His main interests are mobile technologies, Microsoft Windows*, and game development.