Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

CUDA, Supercomputing for the Masses: Part 20

Parallel Nsight Part 2: Using the Parallel Nsight Analysis capabilities

September 10, 2010
URL:http://www.drdobbs.com/architecture-and-design/cuda-supercomputing-for-the-masses-part/227400145

CUDA, Supercomputing for the Masses: Part 19, I focused on the new Parallel Nsight tool for Microsoft Visual Studio including: the thought that went into the design of the tool, how to install and configure the software, how to create a project from scratch, and provided a demonstration of the debugging capabilities provided in the 1.0 release. This article utilizes the Parallel Nsight 1.0 analysis capabilities coupled with the NVIDIA Tools Extension (NVTX) library to examine the simpleMultiCopy asynchronous I/O example from the NVIDIA GPU Computing SDK; create a hybrid CPU and GPU computation that balances the simultaneous use of both CPU and GPU resources on a single calculation; and compare the very fast primitive restart OpenGL rendering code from Part 18 with more conventional OpenGL rendering methods. If you're interested in any of these topics and thinking about purchasing the professional version of Parallel Nsight, you should find this article illuminating. The professional version of Parallel Nsight provides the ability to trace and analyze:

- CUDA applications. In particular, Parallel Nsight has the ability to insert code to capture application traces that include operating system calls, CUDA kernels, data transfers, and host-based methods. NVIDIA put this video online to give viewers a sense of the Nsight tracing and analysis capabilities.
- OpenCL applications. Parallel Nsight has been designed to support multiple languages so OpenCL applications can be traced with the same functionality as CUDA applications.
- Graphics Analysis. For game and visual computing developers, Parallel Nsight allows for the tracing of OpenGL and DirectX API calls in version 1.0, with the tracing of graphics GPU workloads coming in the upcoming version 1.5. This is in addition to the Graphics Inspector, which is part of Parallel Nsight Standard. The Graphics Inspector provides a heads-up performance display (HUD), real-time profiling, plus the ability to examine performance markers and inspect the graphics pipelines. While not covered in this article, NVIDIA demonstrates these capabilities and more in this video. Neither Version 1.0 or 1.5 support OpenGL in the Graphics Inspector.
    - Note that the Graphics Inspector is part of the standard version.
    - Version 1.5 of Parallel Nsight adds a Graphics Workload Trace to the analysis capability, which traces the GPU workloads spawned by individual draw calls.

Information Sources about Parallel Nsight

As mentioned in the previous article, the following are excellent sources of information about Parallel Nsight:

- The NVIDIA Parallel Nsight forum is an excellent place to look for information and post questions.
- The Parallel Nsight Developer Zone is the main entry point for Parallel Nsight on the NVIDIA site. It provides links to support, videos and webinars.

Analysis of Asynchronous I/O

Although asynchronous I/O streams have not yet been covered in this tutorial series, we can use the NVIDIA GPU Computing SDK version 3.1 sample `simpleMultiCopy` to show how Parallel Nsight handles codes with complex asynchronous behavior.

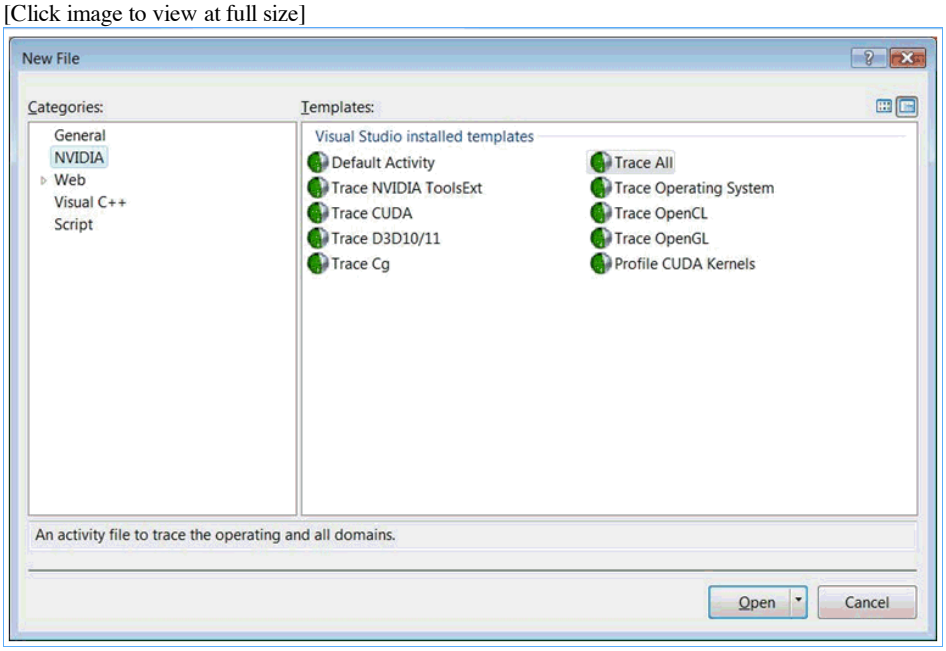
The following steps were used to build this SDK example:

1. Download the Windows version of the CUDA 3.1 SDK. It can be found here.
2. Run the executable, which will install the SDK examples in C:\ProgramData\NVIDIA Corporation.
3. Copy the SDK folder NVIDIA GPU Computing SDK to one of your folders.
4. Change to the folder NVIDIA GPU Computing SDK\C\src\simpleMultiCopy.
5. Double-click on the simpleMultiCopy_vc90.sln icon. The Visual Studio Conversion Wizard will appear to create a version of this solution that can be used with Parallel Nsight.
6. Build the project.
    - Don't forget to set the Nsight User Properties | Connection Name to specify your remote machine! Note that the connection name may also be set on the Activity page itself.
    - Analysis does not necessarily require a remote connection. The name localhost can be used for Connection Name so long as the monitor is installed on the local machine.
7. Be certain that the monitor is running. Depending on if the SDK is installed on the target machine or not, it might be necessary to copy some DLLs so the executable can run.

Now run the executable using the analyzer. From the File menu on the top toolbar select New | File …| and any of the options in the NVIDIA selection as

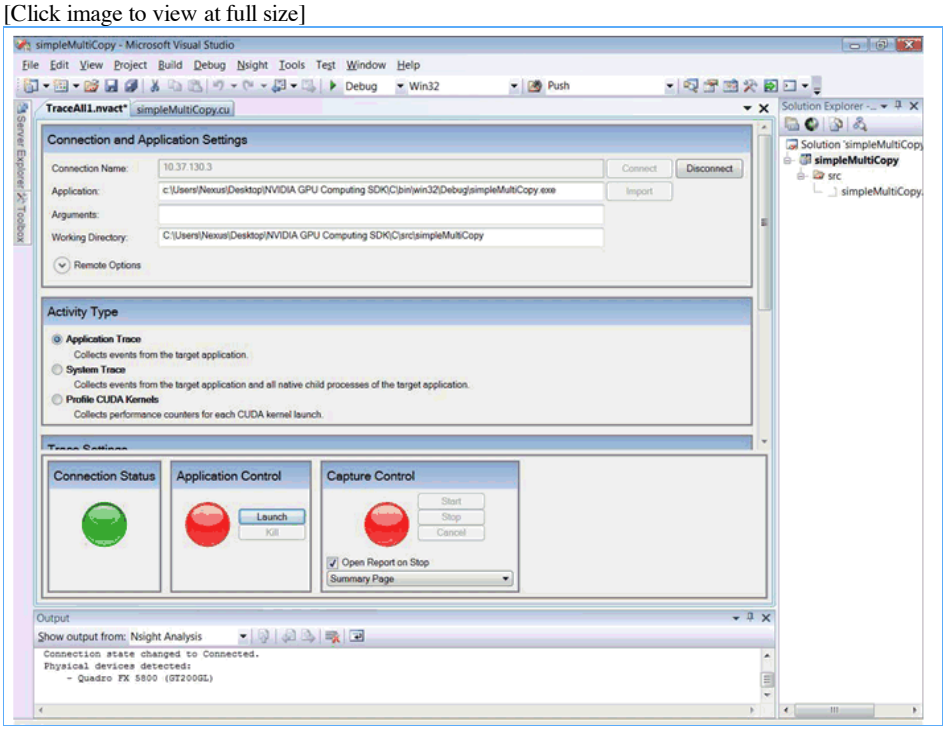
shown in the screen below:

[Click image to view at full size]



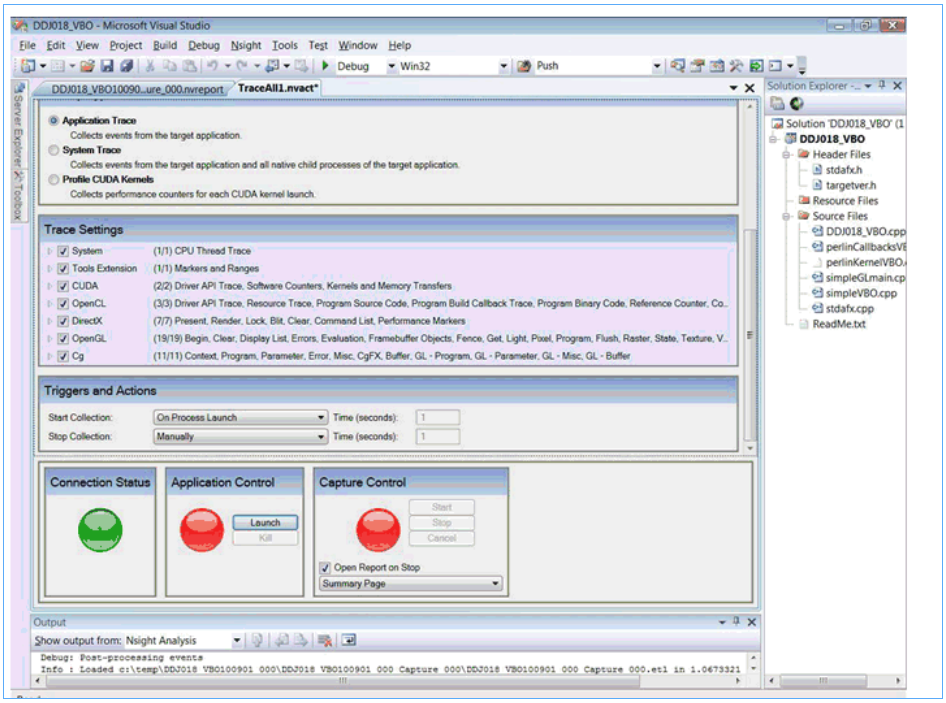
Once an option is selected (in this case Trace All), the following screen appears:

[Click image to view at full size]



Scrolling down, it is clear there is a wealth of options from which to choose.

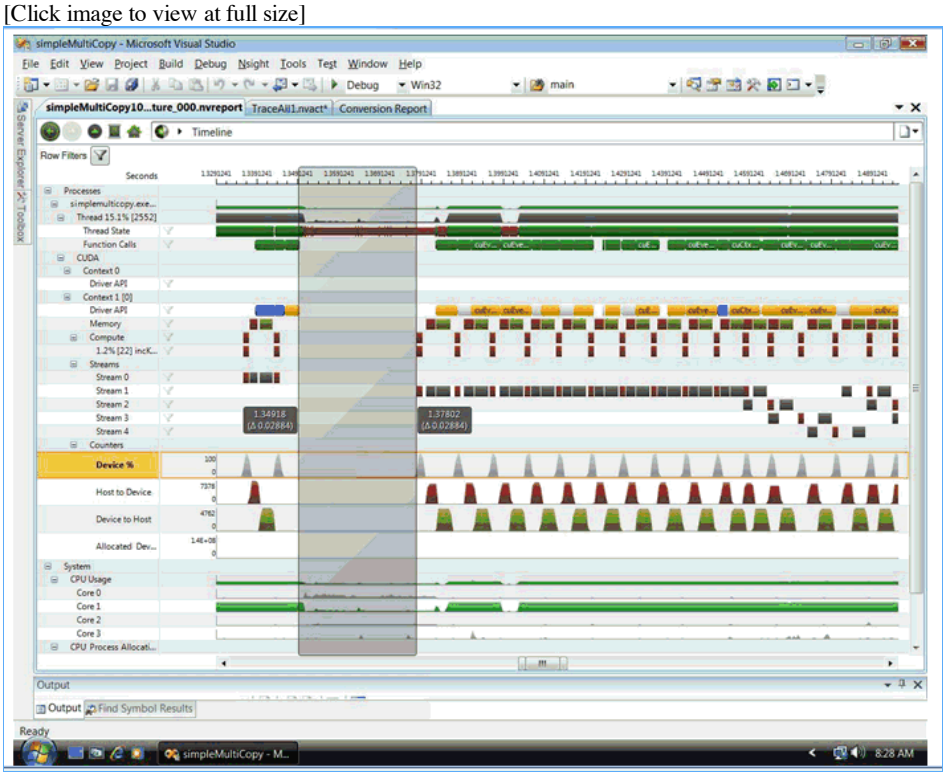
[Click image to view at full size]



Click on the Launch button. When the program pauses at the end, press Enter on the target machine keyboard to terminate the simpleMultiCopy application (or click the Kill button). Once Parallel Nsight has retrieved the trace from the target machine, a summary report will appear. The Capture Control icon will change from red, to yellow (indicating data is being transferred), to green.

As can be seen in Trace timeline below, Parallel Nsight provides a tremendous amount of information that is easily accessible via mouseover and zooming operations as well as various filtering operations. Given the volume of information available in these traces, it is essential to know that regions of the timeline can be selected by clicking the mouse on the screen at a desired starting point of time. A vertical line will appear on the screen. Then press the Shift key and move the mouse (with the button pressed) to the end region of interest. This will result in a grey overlay as shown below. A nice feature is that the time interval for the region is calculated and displayed.

[Click image to view at full size]



We see that it took a short while, about 0.02884 seconds for the asynchronous transfers to get started and a somewhat longer interval for all four streams to really start moving data. Clicking within the grey region will zoom the display to show the just the selected time interval. This makes it very convenient to select and zoom into intervals in the timeline. Other useful controls (that are consistent with typical timeline interfaces in CAD and audio software) are:

- Ctrl + Mousewheel: smoothly zoom into or out of the timeline.
- Ctrl + Drag: pan around in the timeline.

General workflow tips when using Parallel Nsight: the Application or System Trace options can be used to determine if the application is CPU bound, memory bound, or kernel bound. This can be done by looking at the Timeline.

1. **CPU bound**. There will be large areas where no kernel or memory copy is occurring but the application threads (Thread State) is Green
2. **PCIe transfer limited**. Kernel execution is blocked while waiting on memory transfers to or from the device. This can be seen by looking at the Memory row. If much time is being spent doing memory copies then consider using the Streams API to pipeline the application, which can overlap memory transfers and kernels. Before changing code, compare the duration of the transfers and kernels to ensure a performance gain will be realized.
3. **Kernel bound**. If the majority of the application time is spent waiting on kernels to complete then switch to the "Profile CUDA" activity, re-run the application, to collect information from the hardware counters. This can help guide how to optimize kernel performance.

Zooming into a region of the timeline view allows Parallel Nsight to provide the names of the functions and methods as sufficient space becomes available in each region. This really helps the readability of the traces.

Annotating Traces with the NVTX Library

The NVTX library provides a very powerful way to label sections of the computation to provide an easy-to-follow link back to what the actual code is doing. While simple to use, annotating Parallel Nsight traces can greatly help in understanding what is going on as will be shown in the following example, which uses concurrent processing on both a CPU core and the GPU to perform a calculation.

As can be seen in the listing below, cpuGPU.cu shares the work between a CPU core and the GPU by partitioning a computation on a vector between the two systems. Succinctly, this program:

- Allocates a vector `h_data` of size `DATASIZE` on the host and initializes it with random values.
- A portion of the vector is reserved for the GPU via the variable `gpuLen`. The vector `d_data` is then allocated and random values are transferred from `h_data` to `d_data` with `cudaMemcpy()`.
- The host then starts the asynchronous kernel `d_SillyMult()` on the GPU. This kernel scales the vector values in a silly manner by adding each vector element to itself `nScale` times. Similarly, the host performs the same computation using the method `SillySum()`, which calculates the sum of the absolute values of its portion of `h_data` as defined by `hostLen`. Once `SillySum()` returns, the `cuBLAS` routine `cublasSasum()` is called to finish the calculation on the GPU by computing the sum of the absolute value of the scaled `d_data` elements. The `cublasSasum()` method requires that the host system wait for a result from the GPU, thereby synchronizing the two sub-systems.
- The partial sums from both CPU and GPU are then added together and compared against a golden check that calculates the sum of the absolute values of the entire `h_data` vector scaled by `nScale` with a multiplication. Because floating-point arithmetic is approximate, the results of the CPU/GPU computation will differ slightly from the golden check. However, this difference is small.

While obviously a contrived example, the value of `hostLen` can be chosen to balance the workload between the GPU and the CPU. As will be discussed below, Parallel Nsight is used to find the appropriate value of `hostLen` that can match the time taken to calculate the partial sum on the GPU (calculated by calling `d_sillyMult()` and `cublasSasum()`) with the time required to run the `sillySum()` method on a single core of the slower host processor.

To make the tuning easier, the start of the GPU computational region was noted by pushing a character string describing that region on the NVTX stack with `nvtxRangePushA(char*)`. At the end of the region, the label is removed from the stack with `nvtxRangePop()`. As can be seen, the nvToolsExt.h header is included at the beginning of cpuGPU.cu. In addition, the nvToolsEx32_0 library was linked with the executable.

Similarly, the NVTX push and pop operations were used to mark the computational region for the host computation. Since CUDA kernel launches are asynchronous, it is possible to see the concurrent computation of the `SillySum()` function on the host through the nested NVTX regions.

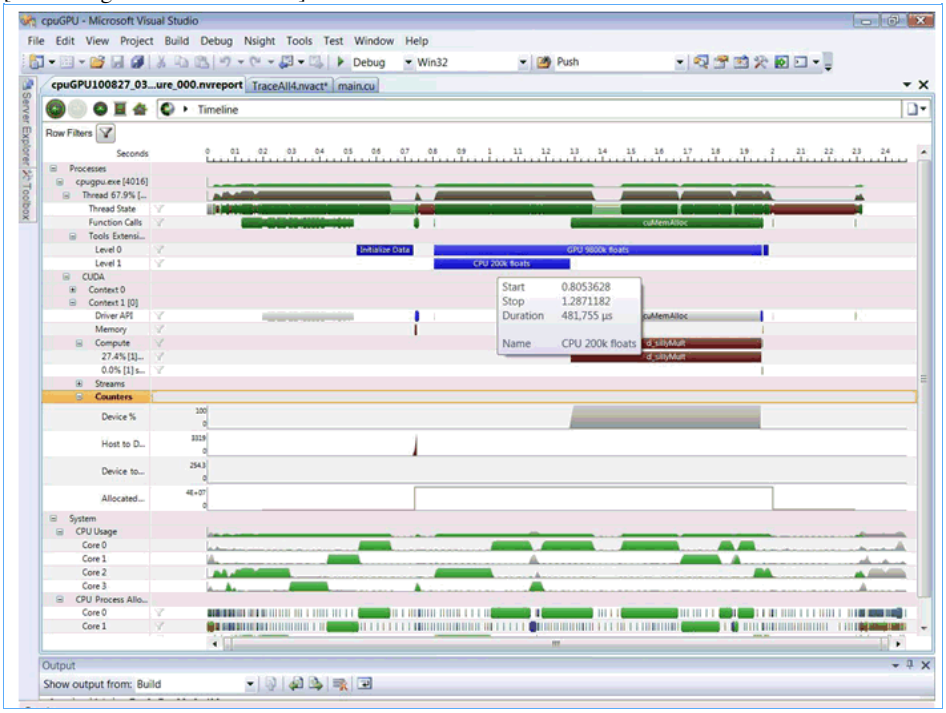
```
#define DATASIZE 10000000
#include <nvToolsExt.h>

#include <math.h>
#include <cuda.h>
#include <cublas.h>
#include <cstdio>
#include <iostream>
#include <string>
using namespace std;

void initializeData(float* h_data, int datalen)
{
        nvtxRangePushA("Initialize Data");
        for(int i=0; i < datalen; i++) {
                h_data[i] = 1e-6f * rand()/(float)RAND_MAX;
        }
        nvtxRangePop();
}

float sillySum(float* h_data, int nScale, int datalen)
{
        float sum=0.f;
        for(int i=0; i < datalen; i++) {
                register float tmp = h_data[i];
                for(int j=1; j< nScale; j++) tmp += h_data[i];
                sum += fabs(tmp);
```

```
                }
        return(sum);
}

//Simple kernel fills an array with perlin noise
__global__ void d_sillyMult(float* data, int nScale, int datalen)
{
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if(tid < datalen) {
     register float tmp = data[tid];
     for(int j=1; j< nScale; j++)
          tmp += data[tid];
      data[tid] = tmp;
  }
}

void myMain()
{
  const int datalen = DATASIZE;
  const int nScale = 1000;
  float *h_data, *d_data;
  int hostLen = 200000;
  int gpuLen=datalen - hostLen;
  char gpuID[256];
  char cpuID[256];

  cout << "cpuLen " << hostLen << " gpuLen " << gpuLen << endl;
  sprintf(gpuID,"GPU %dk floats",gpuLen/1000);
  sprintf(cpuID,"CPU %dk floats",hostLen/1000);

  cublasInit();

  // allocate space on the host
  h_data = new float[datalen];
  initializeData(h_data, datalen);

  // allocate space on the device
  cudaMalloc((void **)&d_data, sizeof(float)*gpuLen);

  // transfer the data
  cudaMemcpy(d_data, h_data+hostLen, sizeof(float)*gpuLen,
             cudaMemcpyHostToDevice);

  int nThreads=256;
  int nBlocks = gpuLen/nThreads;
  nBlocks += ((gpuLen%nThreads)>0)?1:0;

  float partialSum[2];

  cerr << "pause to separate out the calculation" << endl;
  getchar();

  nvtxRangePushA(gpuID);
  d_sillyMult<<< nBlocks, nThreads>>>(d_data, nScale, gpuLen);

  nvtxRangePushA(cpuID);
  partialSum[1] = sillySum(h_data, nScale, hostLen);
  nvtxRangePop();

  partialSum[0] = cublasSasum(gpuLen, d_data, 1);
  nvtxRangePop();
  cudaThreadSynchronize();


  cerr << "pause to separate out the golden" << endl;
  getchar();

  float hybridSum = partialSum[0] + partialSum[1];

  nvtxRangePushA("Shortcut golden");
  float hostSum = 0.f;
  for(int i=0; i < datalen; i++) hostSum += fabs(nScale * h_data[i]);
  nvtxRangePop();

  cout << "hybrid Sum " << partialSum[0] + partialSum[1] << endl;
  cout << "host   Sum " << hybridSum << endl;
  cout << "difference " << fabs(hybridSum - hostSum) << endl;

  // free data
  cudaFree(d_data);
  delete [] h_data;
  cublasShutdown();

  getchar(); // To see the error values in Parallel Nsight
  cudaThreadSynchronize(); //ensure that Parallel Nsight sees the trace info
```

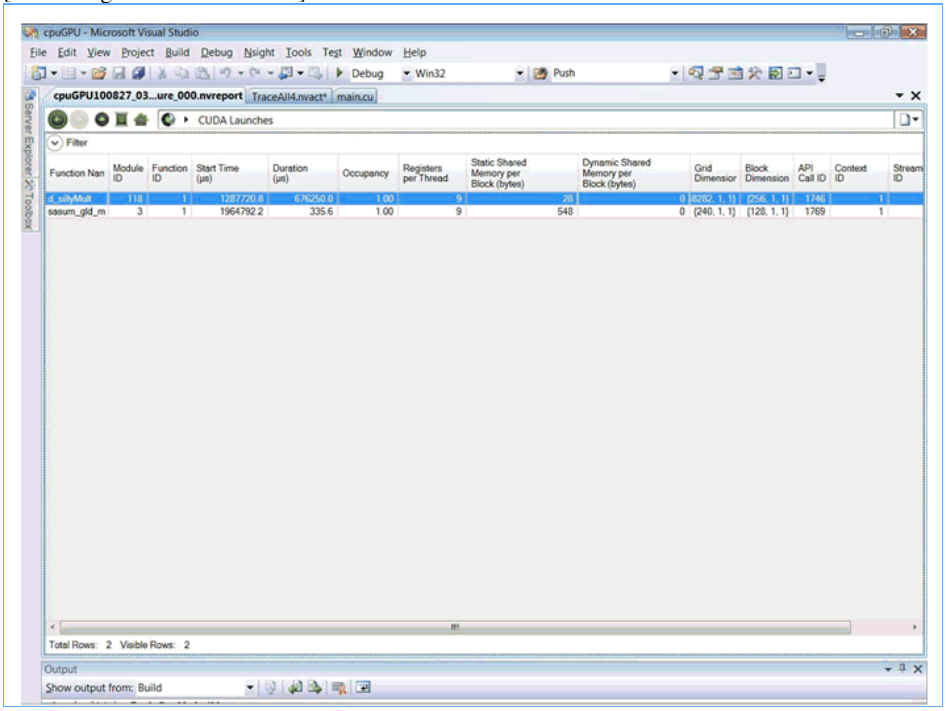
```
}
```

The application was traced using the Trace All option with Parallel Nsight. Note that there is a Tools Extension field in the trace below (e.g. the rows with the big blue blocks). Using a mouseover, we see that the CPU 200k floats computation of `SillySum()` on the host took 481,755 µs. The screenshot of the function call report, shows that the `d_SillySum()` kernel took 672,250 µs and the `cublasSasum()` method required negligible time.

[Click image to view at full size]



[Click image to view at full size]



The workload between the host and GPU was balanced by using the previous Nsight trace and report to adjust the size of `hostLen` so that approximately equal times were taken by the host and GPU. For convenience, very coarse adjustments were made. Finer adjusts can potentially provide even greater performance. Even so, we see that the GPU can process 9800k vector elements in the roughly same time that a single processor core can complete the work for 200k elements. This example makes two points:

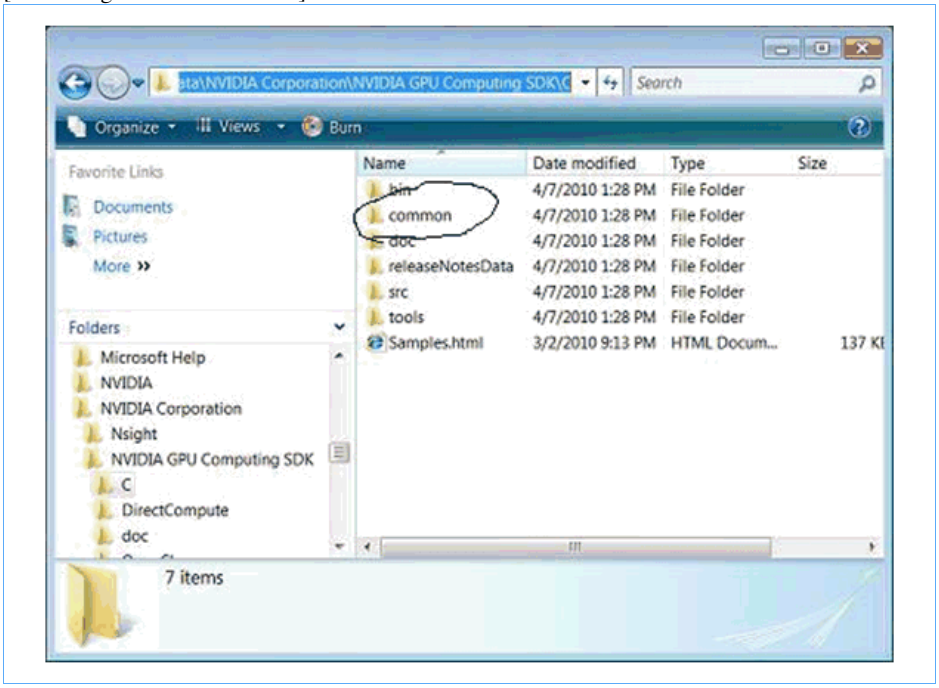
- The GPU is roughly 49 times faster for this particular problem.
- Use of the NVTX library made it easy to understand the trace and identify the pertinent information on the display.

Comparing Primitive Restart to Other OpenGL Rendering Methods

For convenience, a zip file of the Visual Studio project used in this example can be downloaded from GPUcomputing.net here. Once it is downloaded, perform the following steps:

1. Expand the zip file, which creates the director VS_DDJ018_VBO.
2. Copy the common directory from the NVIDIA GPU Computing SDK distribution (shown below) to VS_DDJ018_VBO. This places DLLs, include files, and packages where the Visual Studio project expects them as seen in the annotated screenshot below.

   [Click image to view at full size]



3. Double-click on the Visual Studio project file in VS_DDJ018_VBO\VS_DDJ018_VBO\DDJ018_VBO\DDJ018_VBO/ to start.
4. Build the project by clicking on Build | Build Project.
5. Right click on project in the Solution Explorer window and select Nsight User Properties

    1. Change localhost in Connection Name to the remote IP address (e.g. 10.37.130.3) or the hostname of the target machine that is running the Parallel Nsight monitor.

Note that even the compressed zip files for Visual Studio project can contain megabytes of data. Many of these are too large to email to a colleague. However, the following two bullets can significantly reduce the amount of data that needs to be transmitted:

- Before creating a zip file, delete any Debug directories in the project folders. These will be transparently recreated when the solution is rebuilt.
- When possible, define post-build copy operations to bring needed libraries from commonly available locations. Some of the DLL files can be quite large.

The following example has modified the simpleVBO.cpp from Part 18 of this series support three different OpenGL rendering techniques and NVTX labeling. Only one of the rendering techniques can be selected at compile time by removing the comment slashes. '//' from one of the `#define` statements in the body of the code.

The following three OpenGL rendering methods have been defined:

- PRIMITIVE_RESTART: This utilizes the primitive restart as described in Part 18 of this article series.
- SIMPLE_ONE_BY_ONE: Draws each TRIANGLE_FAN separately.
- MULTI_DRAW: Utilizes the OpenGL `glMultiDrawElements()` API call.

By default, PRIMITIVE_RESTART is defined. Select one of the other preprocessor defines to use a different rendering method. Once changed the code needs to be (Build | Rebuild Project).

The full source for simpleVBO.cpp code is included below. The other files required to build this program, as well as a discussion about how OpenGL and CUDA work together in the same application can be found in Part 18 of this article series as well as in the Visual Studio project.

```
/*
   This wrapper demonstrates how to use the Cuda OpenGL bindings to
   dynamically modify data using a Cuda kernel and display it with opengl.

   The steps are:
```

```
    1. Create an empty vertex buffer object (VBO)
    2. Register the VBO with Cuda
    3. Map the VBO for writing from Cuda
    4. Run Cuda kernel to modify the vertex positions
    5. Unmap the VBO
    6. Render the results using OpenGL

    Host code
*/

// includes, GL
#include <GL/glew.h>
#include <GL/gl.h>
#include <GL/glext.h>

// includes
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include <cutil_gl_inline.h>
#include <cuda_gl_interop.h>
#include <rendercheck_gl.h>

#include <nvToolsExt.h>

extern float animTime;

////////////////////////////////////////////////////////////////////////////////
// VBO specific code
#include <cuda_runtime.h>
#include <cutil_inline.h>

// constants
const unsigned int mesh_width = 128;
const unsigned int mesh_height = 128;
const unsigned int RestartIndex = 0xffffffff;

extern "C"
void launch_kernel(float4* pos, uchar4* posColor,
                   unsigned int mesh_width, unsigned int mesh_height, float time);

// vbo variables
GLuint vbo;
GLuint colorVBO;

////////////////////////////////////////////////////////////////////////////////
//! Create VBO
////////////////////////////////////////////////////////////////////////////////
void createVBO(GLuint* vbo, unsigned int typeSize)
{
  // create buffer object
  glGenBuffers(1, vbo);
  glBindBuffer(GL_ARRAY_BUFFER, *vbo);

  // initialize buffer object
  unsigned int size = mesh_width * mesh_height * typeSize;
  glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);

  glBindBuffer(GL_ARRAY_BUFFER, 0);

  // register buffer object with CUDA
  cudaGLRegisterBufferObject(*vbo);
}

////////////////////////////////////////////////////////////////////////////////
//! Delete VBO
////////////////////////////////////////////////////////////////////////////////
void deleteVBO(GLuint* vbo)
{
  glBindBuffer(1, *vbo);
  glDeleteBuffers(1, vbo);

  cudaGLUnregisterBufferObject(*vbo);

  *vbo = NULL;
}

void cleanupCuda()
{
  deleteVBO(&vbo);
  deleteVBO(&colorVBO);
}

////////////////////////////////////////////////////////////////////////////////
//! Run the Cuda part of the computation
////////////////////////////////////////////////////////////////////////////////
```

```
void runCuda()
{
    // map OpenGL buffer object for writing from CUDA
    float4 *dptr;
    uchar4 *cptr;
    unsigned int *iptr;
    cudaGLMapBufferObject((void**)&dptr, vbo);
    cudaGLMapBufferObject((void**)&cptr, colorVBO);

    // execute the kernel
    launch_kernel(dptr, cptr, mesh_width, mesh_height, animTime);

    // unmap buffer object
    cudaGLUnmapBufferObject(vbo);
    cudaGLUnmapBufferObject(colorVBO);
}

void initCuda(int argc, char** argv)
{
  // First initialize OpenGL context, so we can properly set the GL
  // for CUDA.  NVIDIA notes this is necessary in order to achieve
  // optimal performance with OpenGL/CUDA interop.  use command-line
  // specified CUDA device, otherwise use device with highest Gflops/s
  if( cutCheckCmdLineFlag(argc, (const char**)argv, "device") ) {
    cutilGLDeviceInit(argc, argv);
  } else {
    cudaGLSetGLDevice( cutGetMaxGflopsDeviceId() );
  }

  createVBO(&vbo, sizeof(float4));
  createVBO(&colorVBO, sizeof(uchar4));
  // make certain the VBO gets cleaned up on program exit
  atexit(cleanupCuda);

  runCuda();

}

void renderCuda(int drawMode)
{
  glBindBuffer(GL_ARRAY_BUFFER, vbo);
  glVertexPointer(4, GL_FLOAT, 0, 0);
  glEnableClientState(GL_VERTEX_ARRAY);

  glBindBuffer(GL_ARRAY_BUFFER, colorVBO);
  glColorPointer(4, GL_UNSIGNED_BYTE, 0, 0);
  glEnableClientState(GL_COLOR_ARRAY);

  //glColor3f(1.0, 0.5, 0.0);
  switch(drawMode) {
  case GL_LINE_STRIP:
    for(int i=0 ; i < mesh_width*mesh_height; i+= mesh_width)
      glDrawArrays(GL_LINE_STRIP, i, mesh_width);
    break;
  case GL_TRIANGLE_FAN: {
// I left these in to show some alternative drawing methods
#define PRIMITIVE_RESTART
//#define SIMPLE_ONE_BY_ONE
//#define MULTI_DRAW

#ifdef PRIMITIVE_RESTART
        nvtxRangePushA("Primitive Restart");
    static GLuint* qIndices=NULL;
    int size = 5*(mesh_height-1)*(mesh_width-1);

    if(qIndices == NULL) { // allocate and assign trianglefan indicies
      qIndices = (GLuint *) malloc(size*sizeof(GLint));
      int index=0;
          nvtxRangePushA("Mesh Init");
      for(int i=1; i < mesh_height; i++) {
          for(int j=1; j < mesh_width; j++) {
             qIndices[index++] = (i)*mesh_width + j;
             qIndices[index++] = (i)*mesh_width + j-1;
             qIndices[index++] = (i-1)*mesh_width + j-1;
             qIndices[index++] = (i-1)*mesh_width + j;
             qIndices[index++] = RestartIndex;
          }
      }
    nvtxRangePop();
        }
    glPrimitiveRestartIndexNV(RestartIndex);
    glEnableClientState(GL_PRIMITIVE_RESTART_NV);
    glDrawElements(GL_TRIANGLE_FAN, size, GL_UNSIGNED_INT, qIndices);
    glDisableClientState(GL_PRIMITIVE_RESTART_NV);
        nvtxRangePop();
```

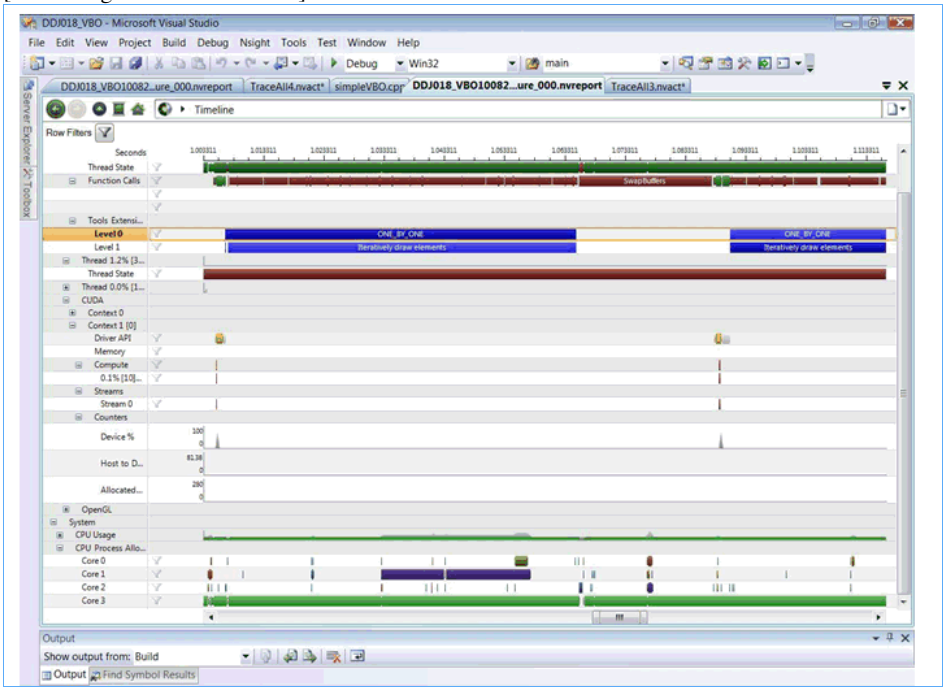
```
#endif

#ifdef SIMPLE_ONE_BY_ONE
    static GLuint* qIndices=NULL;
    int size = 4*(mesh_height-1)*(mesh_width-1);

        nvtxRangePushA("ONE_BY_ONE");
    if(qIndices == NULL) { // allocate and assign trianglefan indices
          nvtxRangePushA("Mesh Init");
      qIndices = (GLuint *) malloc(size*sizeof(GLint));
      int index=0;
      for(int i=1; i < mesh_height; i++) {
            for(int j=1; j < mesh_width; j++) {
               qIndices[index++] = (i)*mesh_width + j;
               qIndices[index++] = (i)*mesh_width + j-1;
               qIndices[index++] = (i-1)*mesh_width + j-1;
               qIndices[index++] = (i-1)*mesh_width + j;
            }
      }
          nvtxRangePop();
      fprintf(stderr,"size %d index %d\n",size,index);
    }
    nvtxRangePushA("Iteratively draw elements");
        for(int i=0; i < size; i +=4)
      glDrawElements(GL_TRIANGLE_FAN, 4, GL_UNSIGNED_INT, &qIndices[i]);
        nvtxRangePop();

        nvtxRangePop();
#endif

#ifdef MULTI_DRAW
        nvtxRangePushA("MULTI_DRAW");

    static GLint* qIndices=NULL;
    static GLint* qCounts=NULL;
    static GLint** qIndex=NULL;
    int size = (mesh_height-1)*(mesh_width-1);

    if(qIndices == NULL) { // allocate and assign trianglefan indicies
      nvtxRangePushA("Mesh Init");
          qIndices = (GLint *) malloc(4*size*sizeof(GLint));
      qCounts = (GLint *) malloc(size*sizeof(GLint));
      qIndex = (GLint **) malloc(size*sizeof(GLint*));

      int index=0;
      for(int i=1; i < mesh_height; i++)
            for(int j=1; j < mesh_width; j++) {
               qIndices[index++] = ((i)*mesh_width + j);
               qIndices[index++] = ((i)*mesh_width + j-1);
               qIndices[index++] = ((i-1)*mesh_width + j-1);
               qIndices[index++] = ((i-1)*mesh_width + j);
             }
      for(int i=0; i < size; i++) qCounts[i] = 4;
      for(int i=0; i < size; i++) qIndex[i] = &qIndices[i*4];
          nvtxRangePop();
    }

    nvtxRangePushA("multidraw elements");
        glMultiDrawElements(GL_TRIANGLE_FAN, qCounts,
                        GL_UNSIGNED_INT, (const GLvoid**)qIndex, size);
        nvtxRangePop();

        nvtxRangePop();
#endif
  } break;
  default:
    glDrawArrays(GL_POINTS, 0, mesh_width * mesh_height);
    break;
  }

  glDisableClientState(GL_VERTEX_ARRAY);
  glDisableClientState(GL_COLOR_ARRAY);
}
```

OpenGL Rendering Methods Trace Analysis

The following Trace All trace was taken with SIMPLE_ONE_BY_ONE defined.

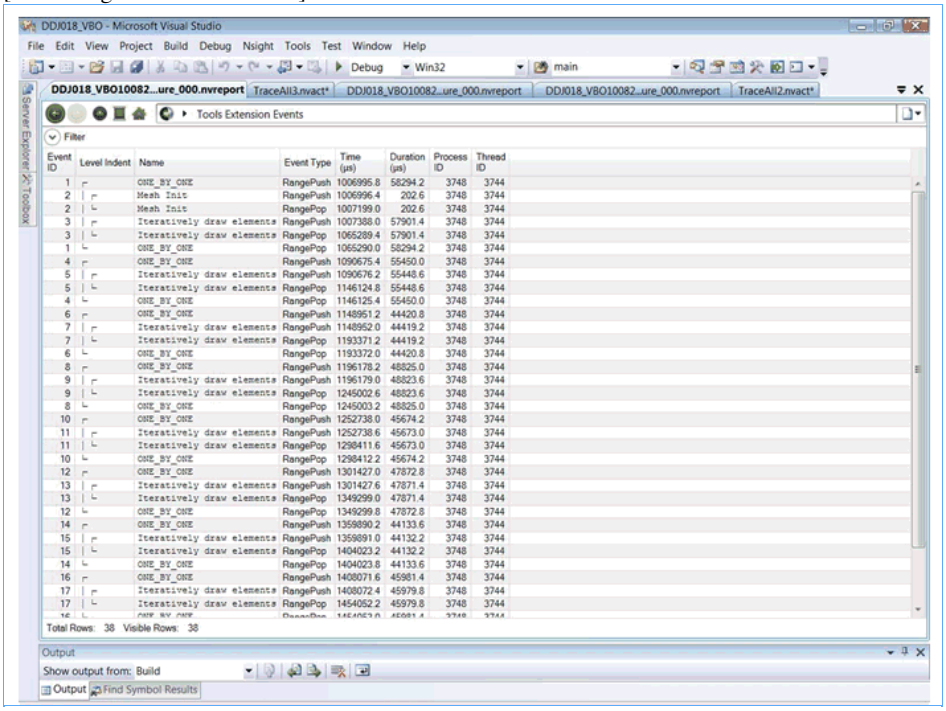
The Compute timeline shows that the `k_perlin()` CUDA kernel only takes 0.1% of the time, which indicates this rendering method is clearly not limited by the performance of the CUDA kernel. The thinness of the vertical line showing the time taken in `k_perlin()` relative to the other activities required to render this 3D artificial world visually illustrates the speed of the CUDA kernel. Also note that the Thread State timeline is solid red. A mouseover tells us that the thread is idle.

[Click image to view at full size]



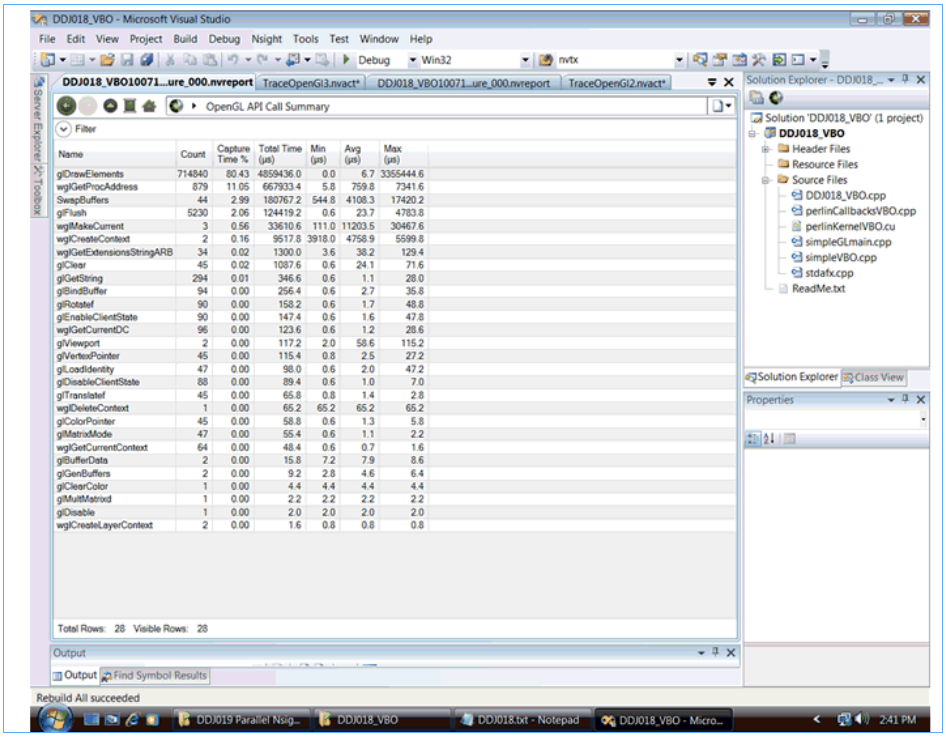
The Tools Extension Events report shows the initialization of the mesh actually takes very little time or roughly 202 µs. We also clearly see the nesting of the methods as recorded by the NVTX calls. Follow-on calls to this rendering method show that the triangle fan mesh initialization is correctly called only once.

[Click image to view at full size]



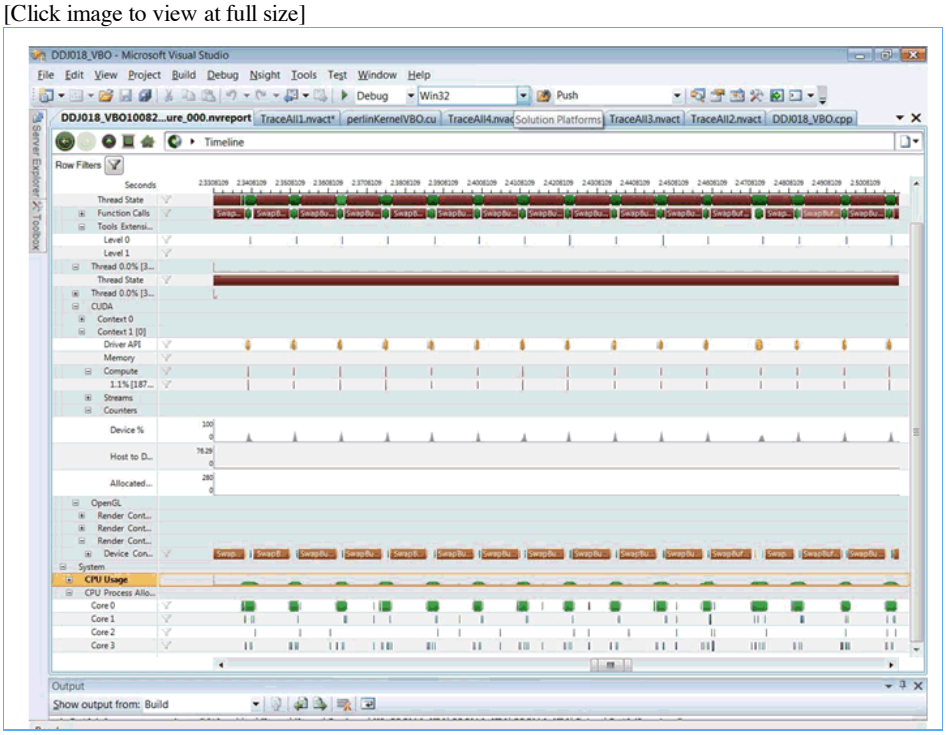
Utilizing the OpenGL API Call Summary, as seen below, shows that most of the time in the SIMPLE_ONE_BY_ONE rendering code is spent in `glDrawElements()`, which consumed the vast majority of the capture time.

[Click image to view at full size]



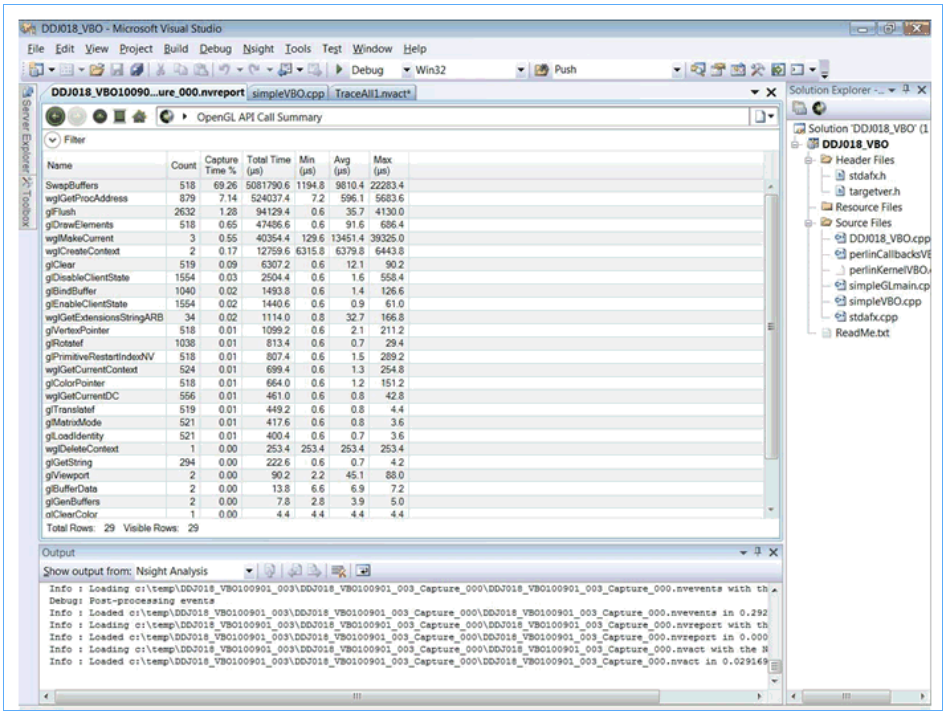
In comparison, the Compute timeline taken when using the PRIMITIVE_RESTART rendering code shows that the `k_perlin()` CUDA kernel takes 1.1% of the time. In addition, the Thread State timeline rapidly alternates between red and green indicating GPU activity. This is also shown in the Device % timeline.

[Click image to view at full size]



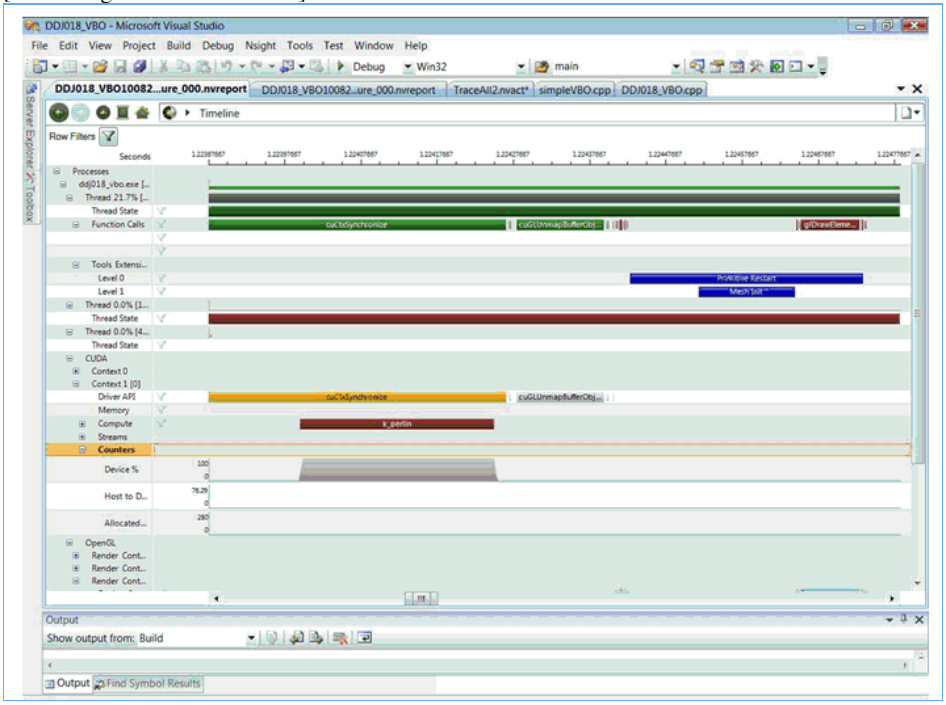
Still, the OpenGL API Call Summary shows that swapping buffers for rendering is easily the dominant runtime component.

[Click image to view at full size]



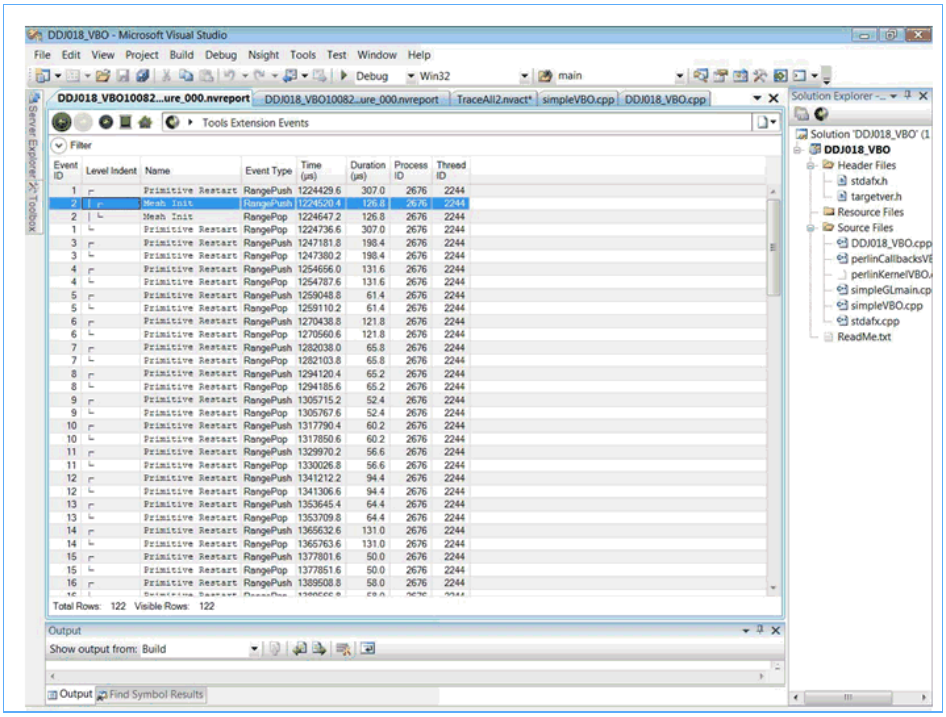
Zooming in on the first rendering operation with primitive restart shows that the complex, computationally intensive GPU Perlin Noise generation `k_perlin()` kernel visually appears to take roughly twice the time of the simple triangle fan mesh generation on the host!

[Click image to view at full size]



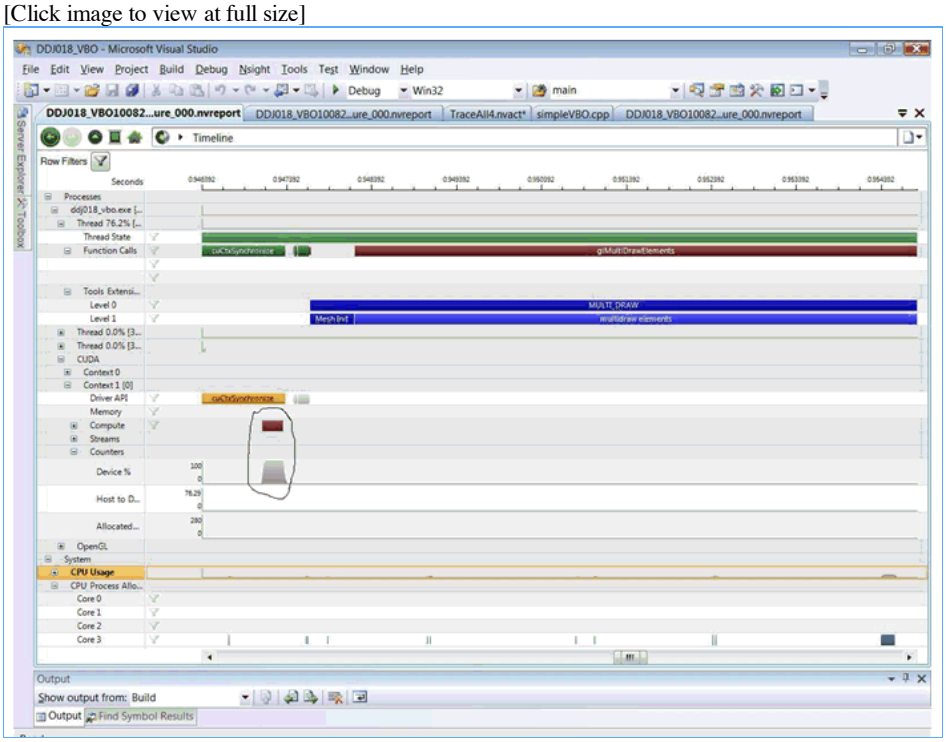
The primitive restart Tools Extension Events report shows that the mesh initialization only takes 126 µs.

[Click image to view at full size]



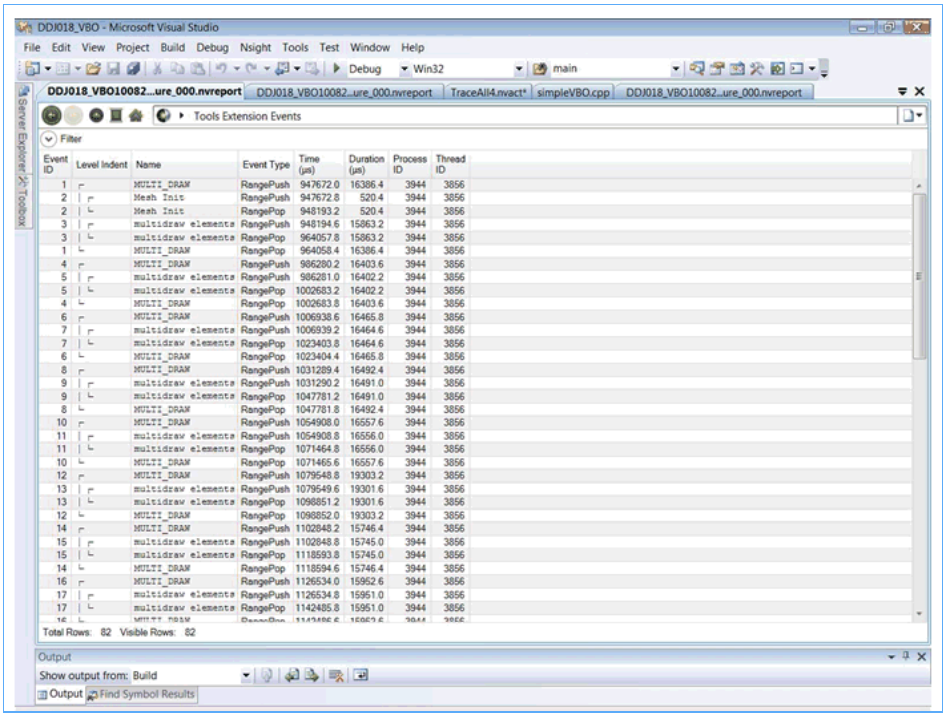
In contrast, we see that the MULTI_DRAW rendering code again spends little time in the `k_perlin()` kernel (circled for clarity in the figure below) -- although the `k_perlin()` kernel appears to make good use of the device when active. Most of the rendering time is spent in the OpenGL `glMultiDrawElements()` elements call.

[Click image to view at full size]



This is confirmed with the Tools Extension Events report.

[Click image to view at full size]



Examining the three Tool Extensions Events reports, we gain a much better understanding why primitive restart is so fast. Except for primitive restart, the `k_perlin()` kernel (even though it purposely has a couple performance issues for readers to find) is overwhelmed by the time taken by the OpenGL rendering calls.

Among the three OpenGL rendering methods, primitive restart is clearly the fastest as the NVTX annotated rendering regions take the following approximate times:

- Primitive restart: around 60 µs.
- Multidraw: around 3,900 µs.
- Iteratively drawing each triangle fan: approximately 1,100,000 µs.

Summary

As this article has shown, the analysis capabilities of NVIDIA's Parallel Nsight v1.0 are powerful indeed! Still, this tutorial has barely scratched the surface plus it is difficult to demonstrate how interactive Parallel Nsight makes the analysis process. To get a better sense of the capabilities of this software, please view some of the online video demonstrations or better yet, download the trial version and try it for yourself.

This article also provides an example hybrid CPU/GPU code that can concurrently share a computation between both a host system processing core and a GPU. Through the Parallel Nsight traces and reports, it should also be clear why primitive restart is such a fast rendering technique for applications that mix graphics and CUDA.

It is also worth noting that Parallel Nsight is an evolving project, with Parallel Nsight 1.5 coming soon with additional improvements to Analysis and NVTX. You should be aware of the following when using version 1.0 of Parallel Nsight for analysis:

- Asynchronous data transfers when using mapped pinned memory do not appear in the application traces.
- It is unclear if OpenMP will be supported.
- Call `cudaThreadExit()` at the end of `main()`to ensure that the Parallel Nsight receives all the trace information from short-lived applications like test programs. If the application exit behavior is complicated, a fallback solution is to use `cudaThreadSynchronize()` or `cudaThreadExit()` in an `atexit()` registered function.

- CUDA, Supercomputing for the Masses: Part 20
- CUDA, Supercomputing for the Masses: Part 19
- CUDA, Supercomputing for the Masses: Part 18
- CUDA, Supercomputing for the Masses: Part 17
- CUDA, Supercomputing for the Masses: Part 16
- CUDA, Supercomputing for the Masses: Part 15
- CUDA, Supercomputing for the Masses: Part 14
- CUDA, Supercomputing for the Masses: Part 13
- CUDA, Supercomputing for the Masses: Part 12
- CUDA, Supercomputing for the Masses: Part 11
- CUDA, Supercomputing for the Masses: Part 10
- CUDA, Supercomputing for the Masses: Part 9
- CUDA, Supercomputing for the Masses: Part 8

- CUDA, Supercomputing for the Masses: Part 7
- CUDA, Supercomputing for the Masses: Part 6
- CUDA, Supercomputing for the Masses: Part 5
- CUDA, Supercomputing for the Masses: Part 4
- CUDA, Supercomputing for the Masses: Part 3
- CUDA, Supercomputing for the Masses: Part 2
- CUDA, Supercomputing for the Masses: Part 1

Rob Farber is a senior scientist at Pacific Northwest National Laboratory. He has worked in massively parallel computing at several national laboratories and as co-founder of several startups. He can be reached at rmfarber@gmail.com