CUDA, Supercomputing for the Masses: Part 17

CUDA 3.0 provides expanded capabilities and makes development easier

April 14, 2010
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/224400246

In CUDA, Supercomputing for the Masses: Part 16 of this article series, I discussed the CUDA 3.0 release. CUDA 3.0 is a major revision number increment release that adds enhancements valuable to all CUDA developers to make day-to-day development tasks easier, less error prone, and more consistent.

As mentioned in the previous article, expanded consistent coverage appears to have been the thrust behind this major revision number release as it fills in several previous gaps and must-have capabilities. In a nutshell, in this article I discuss runtime and driver API compatibility, the new graphics interoperability API, C++ inheritance plus expanded functionality in CUBLAS and CUFFT. Examples are provided that demonstrate:

- Consistency and interoperability between the CUDA runtime and driver codes through two simple examples that call a runtime kernel and a CUBLAS routine.
- A C++ example that:
  - Uses both versions of the OpenGL interoperability APIs in drawing a Mandelbrot set. The C++ source code concisely recreates the simplePBO example from Part 15 of this article series using C++ classes.
  - Demonstrates C++ inheritance by deriving a new class from our C++ Mandelbrot example that uses programmable shaders created with Cg. (Please look to the extensive NVIDIA Cg homepage for more information). While Cg compatibility is not new with the CUDA 3.0 release, mixing Cg shaders with CUDA can open up a vast collection of Cg libraries and existing software!

Be aware that the latest NVIDIA driver must be installed to use the CUDA 3.0 toolkit. As always, the latest released driver can be downloaded from CUDA ZONE and installed for a number of systems. Beta drivers and software can be downloaded from nvdevelopers but registration is required. Ubuntu users might wish to follow one of the many available guides, such as the one at Web Upd8, to see how to install the latest released or beta drivers via the Ubuntu tools.

Mixing CUDA Runtime and Driver API Codes

Previous articles in this series have focused on teaching the CUDA with the runtime API (e.g. those methods that start with "cuda" as opposed to "cu") because it is fairly intuitive and not too verbose. Many developers prefer to utilize the driver API because they have more control and can make better use of existing code bases. Now programmers can utilized the best characteristics of both APIs.

The following is the source code for a driver mode CUDA program that calls a kernel via the runtime API. Please put this into a file called vectorAddDrv.cu:

```
/*
 * Driver APIC code that calls a runtime kernel
 * Vector addition: C = A + B.
 */

// Includes
#include <stdio.h>
#include <cuda.h>
#include <cutil_inline.h>

// Variables
CUdevice cuDevice;
CUcontext cuContext;
CUmodule cuModule;
CUfunction vecAdd;
float* h_A;
float* h_B;
float* h_C;
CUdeviceptr d_A;
CUdeviceptr d_B;
CUdeviceptr d_C;

// Functions

__global__ void kernel(float* d_a, float* d_b, float* d_c, int n)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if(idx < n)
    d_c[idx] = d_a[idx] + d_b[idx];
}

// Allocates an array with random float entries.
void RandomInit(float* data, int n)
{
  for (int i = 0; i < n; ++i)
    data[i] = rand() / (float)RAND_MAX;
}

void errorExit()
{
  printf("Error exit!\n");
  exit(1);
}

// Host code
int main(int argc, char** argv)
{
  int N = 50000;
  unsigned int size = N * sizeof(float);
  CUresult error;

  printf("Vector Addition (Driver API)\n");
  // Initialize
  error = cuInit(0);
  if (error != CUDA_SUCCESS) errorExit();

  // Get number of devices supporting CUDA
  int deviceCount = 0;
  error = cuDeviceGetCount(&deviceCount);
  if (error != CUDA_SUCCESS) errorExit();
  if (deviceCount == 0) {
    printf("There is no device supporting CUDA.\n");
    exit(1);
  }

  // Get handle for device 0
  error = cuDeviceGet(&cuDevice, 0);
  if (error != CUDA_SUCCESS) errorExit();

  // Create context
  error = cuCtxCreate(&cuContext, 0, cuDevice);
  if (error != CUDA_SUCCESS) errorExit();

  // Allocate input vectors h_A and h_B in host memory
  h_A = (float*)malloc(size);
  if (h_A == 0) errorExit();
  h_B = (float*)malloc(size);
  if (h_B == 0) errorExit();
  h_C = (float*)malloc(size);
  if (h_C == 0) errorExit();

  // Initialize input vectors
  RandomInit(h_A, N);
  RandomInit(h_B, N);

  // Allocate vectors in device memory
  error = cuMemAlloc(&d_A, size);
  if (error != CUDA_SUCCESS) errorExit();
  error = cuMemAlloc(&d_B, size);
  if (error != CUDA_SUCCESS) errorExit();
  error = cuMemAlloc(&d_C, size);
  if (error != CUDA_SUCCESS) errorExit();

  // Copy vectors from host memory to device memory
  error = cuMemcpyHtoD(d_A, h_A, size);
  if (error != CUDA_SUCCESS) errorExit();
  error = cuMemcpyHtoD(d_B, h_B, size);
  if (error != CUDA_SUCCESS) errorExit();

  // Invoke kernel (Runtime API)
  int nThreadsPerBlk=128;
  int nBlks = (N/nThreadsPerBlk) + (((N%nThreadsPerBlk)>0)?1:0);
  kernel<<<nBlks,nThreadsPerBlk>>>((float*)d_A,(float*) d_B,(float*) d_C, N);

  // Copy result from device memory to host memory
  // h_C contains the result in host memory
  error = cuMemcpyDtoH(h_C, d_C, size);
  if (error != CUDA_SUCCESS) errorExit();

  // Verify result
  int i;
  for (i = 0; i < N; ++i) {
    float sum = h_A[i] + h_B[i];
    if (fabs(h_C[i] - sum) > 1e-7f) {
      printf("Mistake index %d %g %g\n",i,h_C[i],sum);
      break;
    }
  }
  printf("Test %s \n", (i == N) ? "PASSED" : "FAILED");
  return(0);
}
```

A detailed discussion of the driver API is beyond the scope of this article, as I'm focusing on interoperability and the 3.0 release. Even so, much of this code should look familiar to runtime API developers as only GPU setup, memory allocation and data movement is utilized. Many of these calls are similar to the runtime API calls so it should be easy to follow the source code. The specific point made with this example is that the following runtime CUDA call to `kernel()` works in the 3.0 release:

```
  kernel<<<nBlks,nThreadsPerBlk>>>((float*)d_A,(float*) d_B,(float*) d_C, N);
```

Use nvcc to build the executable with debugging enabled as shown below for Linux:

```
SDK_PATH=$HOME/NVIDIA_GPU_Computing_SDK/C
INC=$SDK_PATH/common/inc
LIB=$SDK_PATH/lib
nvcc -g -G -I$INC -L$LIB vectorAddDrv.cu -lcutil_x86_64 -lcuda -o vectorAddDrv
```

Now start the program with the command `cuda-gdb`:

```
$ cuda-gdb vectorAddDrv
NVIDIA (R) CUDA Debugger
BETA release
Portions Copyright (C) 2008,2009 NVIDIA Corporation
GNU gdb 6.6
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-unknown-linux-gnu"...
Using host libthread_db library "/lib/libthread_db.so.1".
```

Enable memory checking and run the program with the following two commands:

```
(cuda-gdb) set cuda memcheck on
(cuda-gdb) r
```

The following illustrative output shows that the code works and apparently used memory correctly. Further the results passed a validation test on the host as indicated by the "Test PASSED" message.

```
Starting program: PATH_TO_EXECUTABLE/vectorAddDrv
[Thread debugging using libthread_db enabled]
[New process 31619]
Vector Addition (Driver API)
[New Thread 139794699740944 (LWP 31619)]
Test PASSED

Program exited normally.
```

Now set a breakpoint at line 109, which is just after the result vector is moved to the host computer and run the `vectorAddDrv` again:

```
.

(cuda-gdb) b 109
Breakpoint 1 at 0x40183e: file vectorAdd.cu, line 109.

(cuda-gdb) r
Starting program: /home/rmfarber/DDJ/Part17/vecAdd/vectorAdd
[Thread debugging using libthread_db enabled]
[New process 31623]
Vector Addition (Driver API)
[New Thread 140441416316688 (LWP 31623)]
[Switching to Thread 140441416316688 (LWP 31623)]
```

```
Breakpoint 1, main (argc=1, argv=0x7fff40b55248) at vectorAdd.cu:109
109       if (error != CUDA_SUCCESS) errorExit();
```

Use the CUDA-GDB set command to change index three of the `h_C` vector on the host to be 1000-times larger:

```
(cuda-gdb) p h_C[3]
$1 = 1.30907393
(cuda-gdb) set h_C[3] = 1000. * h_C[3]
```

The program is then allowed to continue. The resulting output shows that the comparison code does find the error caused by our manually modifying the `h_C` vector:

```
(cuda-gdb) c
Continuing.
Mistake index 3 1309.07 1.30907
Test FAILED

Program exited normally.
(cuda-gdb) quit
```

This example demonstrates that CUDA-GDB in the 3.0 release works with driver API programs. It also shows how straightforward it now is to mix driver and runtime API codes.

Similarly, the following example, blasAddDrv.cu, demonstrates that it is now possible with CUDA 3.0 to call the CUBLAS library routines. In this case, the previous example code for vectorAddDrv.cu was adapted to call the `saxpy()` routine:

```
  cublasSaxpy(N, 1.0f,(float*) d_A, 1,(float*) d_B, 1);
```

The following is the complete source code for vectorAddDrv.cu. Again, I won't discuss the details of the driver API. See the NVIDIA documentation for more information.

```
/*
 * Vector addition using SAXPY from a driver API example
 */

// Includes
#include <stdio.h>
#include <cuda.h>
#include <cutil_inline.h>
#include <cublas.h>


// Variables
CUdevice cuDevice;
CUcontext cuContext;
CUmodule cuModule;
CUfunction vecAdd;
float* h_A;
float* h_B;
float* h_C;
CUdeviceptr d_A;
CUdeviceptr d_B;

// Functions

// Allocates an array with random float entries.
void RandomInit(float* data, int n)
{
  for (int i = 0; i < n; ++i)
    data[i] = rand() / (float)RAND_MAX;
}

void errorExit()
{
  printf("Error exit!\n");
}

// Host code
int main(int argc, char** argv)
{
  int N = 50000;
  unsigned int size = N * sizeof(float);
  CUresult error;
  int status;

  printf("Vector Addition (BLAS and Driver APIs)\n");
  // Initialize
  error = cuInit(0);
  if (error != CUDA_SUCCESS) errorExit();

  // Get number of devices supporting CUDA
  int deviceCount = 0;
  error = cuDeviceGetCount(&deviceCount);
  if (error != CUDA_SUCCESS) errorExit();
  if (deviceCount == 0) {
    printf("There is no device supporting CUDA.\n");
    exit(1);
  }

  // Get handle for device 0
  error = cuDeviceGet(&cuDevice, 0);
  if (error != CUDA_SUCCESS) errorExit();

  // Create context
  error = cuCtxCreate(&cuContext, 0, cuDevice);
  if (error != CUDA_SUCCESS) errorExit();

  // Allocate input vectors h_A and h_B in host memory
  h_A = (float*)malloc(size);
  if (h_A == 0) errorExit();
  h_B = (float*)malloc(size);
  if (h_B == 0) errorExit();
  h_C = (float*)malloc(size);
  if (h_C == 0) errorExit();

  // Initialize input vectors
  RandomInit(h_A, N);
  RandomInit(h_B, N);

  // Allocate vectors in device memory
  error = cuMemAlloc(&d_A, size);
  if (error != CUDA_SUCCESS) errorExit();
  error = cuMemAlloc(&d_B, size);
  if (error != CUDA_SUCCESS) errorExit();

  // Copy vectors from host memory to device memory
  error = cuMemcpyHtoD(d_A, h_A, size);
  if (error != CUDA_SUCCESS) errorExit();
  error = cuMemcpyHtoD(d_B, h_B, size);
  if (error != CUDA_SUCCESS) errorExit();

  // Invoke kernel
  status = cublasInit();
  if (status != CUBLAS_STATUS_SUCCESS) {
    fprintf (stderr, "!!!! CUBLAS initialization error\n");
    return EXIT_FAILURE;
  }

  // Call CUBLAS from a driver API code
  cublasSaxpy(N, 1.0f,(float*) d_A, 1,(float*) d_B, 1);

  // Copy result from device memory to host memory
  // h_C contains the result in host memory
  error = cuMemcpyDtoH(h_C, d_B, size);
  if (error != CUDA_SUCCESS) errorExit();

  // Verify result
  int i;
  for (i = 0; i < N; ++i) {
    float sum = h_A[i] + h_B[i];
    if (fabs(h_C[i] - sum) > 1e-7f)
      break;
  }
  printf("Test %s \n", (i == N) ? "PASSED" : "FAILED");
  return(0);
}
```

Use nvcc to build the executable. The following simple script builds the executable on Linux. Please note that this program requires the CUBLAS library. Linking is specified with the `-lcublas` command-line option.

```
SDK_PATH=$HOME/NVIDIA_GPU_Computing_SDK/C
INC=$SDK_PATH/common/inc
LIB=$SDK_PATH/lib
nvcc -I$INC -L$LIB blasAddDrv.cu -lcutil_x86_64 -lcuda -lcublas -o blasAddDrv
```

Running the program demonstrates that it does indeed work correctly as indicated by the "Test PASSED" message:

```
$ ./blasAddDrv
Vector Addition (BLAS and Driver APIs)
Test PASSED
```

Unquestionably, the ability to mix and debug driver and runtime API codes and libraries is valuable. For many, this expanded capability alone will make the CUDA 3.0 release an obvious download choice. As the CUDA library and code base expands, this transparent interoperability will continue to pay dividends in ease of use -- although it is likely that in the future most developers will utilize this capability without thought or awareness of the ease in which it occurs.

CUBLAS and CUFFT Enhancements

CUBLAS users will be happy that complex numbers are now supported. The release notes indicate the following routines have been added:

- BLAS1
  - cublasZaxpy()
  - cublasZcopy()
  - cublasZswap()
- BLAS2
  - cublasDtrmv()
  - cublasCtrmv()
  - cublasCgemv()
  - cublasCgeru()
  - cublasCgerc()
  - cublasZtrmv()
  - cublasZgemv()
  - cublasZgeru()
  - cublasZgerc()
- BLAS3
  - cublasCtrsm()
  - cublasCtrmm()
  - cublasCsyrk()
  - cublasCsymm()
  - cublasCherk()
  - cublasZtrsm()
  - cublasZtrmm()
  - cublasZsyrk()
  - cublasZsymm()
  - cublasZherk()

The site oscarbg.blogspot notes that batched 2D and 3D transforms are supported in CUFFT with the new `cufftPlanMany()` API. This is defined in cufft.h, as follows:

```
cufftResult CUFFTAPI cufftPlanMany(cufftHandle *plan,
        int rank,
        int *n,
        int *inembed,    // Unused: pass NULL
        int istride,     // Unused: pass 1
        int idist,       // Unused: pass 0
        int *onembed,    // Unused: pass NULL
        int ostride,     // Unused: pass 1
        int odist,       // Unused: pass 0
        cufftType type,
        int batch);
```

The arguments are:

`*plan` -- The plan is returned here, as for other cufft calls

`rank` --The dimensionality of the transform (1, 2 or 3)

`*n` -- An array of size [rank], describing the size of each dimension

`type` -- Transform type (e.g. CUFFT_C2C), as per other CUFFT calls

`batch` -- Batch size for this transform

Return values are as for all other `cufftPlan…()` functions. Creating a plan for 1,000 2D double-precision, complex-to-complex transforms of size (128, 256) will look something like the following:

```
cufftHandle *myplan;
cufftPlanMany(myplan, 2, {128, 256}, NULL, 1, 0, NULL, 1, 0, CUFFT_Z2Z, 1000);
```

Note that for CUFFT 3.0, the layout of batched data must be side-by-side and not interleaved. The `inembed`, `istride`, `idist`, `onembed`, `ostride`, and `odist` parameters are for enabling data windowing and interleaving in a future version.

A C++ Inheritance and Graphics Interoperability Example

With the 3.0 release, C++ inheritance is now supported for both classes and templates!

Inheritance is a fundamental concept in the C++ language. Without doubt, the CUDA 3.0 release greatly enhances the ability of C++ classes, templates, and libraries to efficiently use both GPU and CPU resources. While working with C++ on GPUs is a future topic, this article will provide working examples of the `simplePBO` code discussed in Part 15 of this article series. Far from a cosmetic change, C++ was purposefully utilized so inheritance could be demonstrated in deriving a new class that can render using arbitrary programmable shaders written in Cg.

Both C++ and Cg are extensive topics that are well beyond the scope of this article. However, working C++ and Cg source code along with the commands used to build the examples under Linux are provided. The C++ source code should be familiar as much of the code from Part 15 has been reused. Readers can refer to Part 15 for a more in-depth discussion about the code and APIs that implement the C++ methods.

Even those not interested in C++ or Cg will find it useful to scan the examples because the differences between the old and new graphics interoperability API are highlighted by the code contained in the `#ifdef USE_CUDA3` preprocessor directives.

The following is the source code for mandelbrotPBO.cu. It is organized from the start of the example as follows:

1. Header and preprocessor directives. Please remove the `#define USE_CUDA3` to build the code on earlier releases. Note that the graphics interoperability APIs prior to the 3.0 release are deprecated and will go away in a future release.
2. An implementation of the Mandelbrot kernel that calculates the data to be displayed. Obviously this kernel can be replaced to run other calculations on the GPU. Perhaps some of the more adventurous readers might like to try using the Perlin noise generator discussed in Part 15.
3. A C++ functor that calls the GPU kernel. A functor is basically a function that preserves state information.
4. The definition, variables, and methods of the `drawable2DTexture` class. Most of the methods in this class were discussed in Part 15. Please note that the C++ protected keyword was used instead of private for inheritance reasons.
5. Several callback functions for GLUT. Further discussion can be found in Part 15. For simplicity, the `cleanupCallback()` was left unspecified.
6. The main program.

The complete mandelbrotPBO.cu source code is as follows:

```
#include <GL/glew.h>
#include <GL/glut.h>
#define USE_CUDA3

#include <cuda.h>
#include <cuda_gl_interop.h>

__global__ void mandelbrot( uchar4 *d_ptr, ulong2 size,
                            float2 rrange, float2 irange, unsigned long n )
{
  unsigned long x = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned long y = blockIdx.y * blockDim.y + threadIdx.y;
  if ( x >= size.x || y >= size.y ) return;

  float2 C = make_float2( ( rrange.y - rrange.x ) / size.x * x + rrange.x,
                          ( irange.y - irange.x ) / size.y * y + irange.x );

  uchar4 buf = make_uchar4( 0, 0, 0, 0 );
  if ( C.x * C.x + C.y * C.y <= 4.0f ) {
    float2 z = make_float2( 0.0f, 0.0f );
    unsigned long cnt = 0;
    while ( cnt < n ) {
      ++cnt;
      z = make_float2( z.x * z.x - z.y * z.y + C.x, 2.0f * z.x * z.y + C.y );
      if ( z.x * z.x + z.y * z.y > 4.0f ) {
        unsigned char c = ( 255 - 15 ) * __powf( ( float )cnt / n, 0.8f )+15;
        buf = make_uchar4( c, c, c, 0 );
        break;
      }
    }
  }

  d_ptr[ x + y * size.x ] = buf;
}

// A functor is basically a function that maintains state
```

```
struct gpuFunctor {

protected:
  float2 rrange;
  float2 irange;

public:
gpuFunctor(float2 rrange, float2 irange) : rrange(rrange), irange(irange) {}
gpuFunctor() : rrange(make_float2(-2.f, 2.f)), irange(make_float2(-2.f, 2.f)) {}

  void callKernel(uchar4 *d_ptr, ulong2 gridSize, unsigned long convergence)
  {
    dim3 dimBlock( 16, 4 );
    dim3 dimGrid( ( gridSize.x + dimBlock.x - 1 ) / dimBlock.x,
                  ( gridSize.y + dimBlock.y - 1 ) / dimBlock.y );
    mandelbrot<<< dimGrid, dimBlock >>>( d_ptr, gridSize, rrange,
                                         irange, convergence );
  }
  inline float2 getRrange() {return(rrange);}
  inline float2 getIrange() {return(irange);}
};

class drawable2DTexture {
 protected:
  GLuint tex,pbo;
  ulong2 size;
  gpuFunctor gpuFunc;

#ifdef USE_CUDA3
  struct cudaGraphicsResource *pboCUDA;
#endif

  void createPBO()
  {
    // set up vertex data parameter
    int num_texels = size.x * size.y;
    int num_values = num_texels * 4;
    int size_tex_data = sizeof(GLubyte) * num_values;

    // Generate a buffer ID called a PBO (Pixel Buffer Object)
    glGenBuffers(1,&pbo);
    // Make this the current UNPACK buffer (OpenGL is state-based)
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo);
    // Allocate data for the buffer. 4-channel 8-bit image
    glBufferData(GL_PIXEL_UNPACK_BUFFER, size_tex_data, NULL, GL_DYNAMIC_COPY);

#ifdef USE_CUDA3
    cudaGraphicsGLRegisterBuffer( &pboCUDA, pbo, cudaGraphicsMapFlagsNone );
#else
    cudaGLRegisterBufferObject( pbo );
#endif
  }

  void deletePBO()
  {
    if(pbo) {
      // delete the PBO
#ifdef USE_CUDA3
      cudaGraphicsUnregisterResource( pboCUDA );
#else
      cudaGLUnregisterBufferObject( pbo );
#endif
      glDeleteBuffers( 1, &pbo );
      pbo=NULL;
      pboCUDA = NULL;
    }
  }

  void createTexture()
  {
    if(tex) deleteTexture();

    unsigned int image_height=size.y;
    unsigned int image_width=size.x;

    // Enable Texturing
    glEnable(GL_TEXTURE_2D);

    // Generate a texture identifier
    glGenTextures(1,&tex);

    // Make this the current texture (remember that GL is state-based)
    glBindTexture( GL_TEXTURE_2D, tex);

    // Allocate the texture memory. The last parameter is NULL since we only
    // want to allocate memory, not initialize it
    glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA8, image_width, image_height, 0, GL_BGRA,
                  GL_UNSIGNED_BYTE, NULL);

    // Must set the filter mode, GL_LINEAR enables interpolation when scaling
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    // Note: GL_TEXTURE_RECTANGLE_ARB may be used instead of
    // GL_TEXTURE_2D for improved performance if linear interpolation is
    // not desired. Replace GL_LINEAR with GL_NEAREST in the
    // glTexParameteri() call
  }

  void deleteTexture()
  {
    if(tex) {
      glDeleteTextures(1, &tex);
      tex = NULL;
    }
  }

  void cleanup (void) { deletePBO(); deleteTexture(); }


 public:

  ~drawable2DTexture() { cleanup(); }

  drawable2DTexture(ulong2 winSize, gpuFunctor userGPUfunc)
    {
      size = winSize;
      gpuFunc = userGPUfunc;
#ifdef USE_CUDA3
      pboCUDA = NULL;
#endif

      createTexture();
      createPBO();
    }
  void draw( unsigned long convergence )
  {
    uchar4 *d_ptr = NULL;

#ifdef USE_CUDA3
    size_t start;
    cudaGraphicsMapResources( 1, &pboCUDA, NULL );
    cudaGraphicsResourceGetMappedPointer( ( void ** )&d_ptr, &start, pboCUDA );
#else
    cudaGLMapBufferObject( ( void ** )&d_ptr, pbo );
#endif

    gpuFunc.callKernel(d_ptr, size, convergence);
    cudaThreadSynchronize();

#ifdef USE_CUDA3
    cudaGraphicsUnmapResources( 1, &pboCUDA, NULL );
#else
    cudaGLUnmapBufferObject( pbo );
#endif

    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo);

    glBindTexture( GL_TEXTURE_2D, tex );
    glTexSubImage2D( GL_TEXTURE_2D, 0, 0, 0, size.x, size.y,
                     GL_RGBA, GL_UNSIGNED_BYTE, NULL );

  }

  void display( void )
  {
        glBegin( GL_QUADS );
    glTexCoord2f( 0.0f, 0.0f ); glVertex2f( 0.0f, 0.0f );
    glTexCoord2f( 1.0f, 0.0f ); glVertex2f( 1.0f, 0.0f );
    glTexCoord2f( 1.0f, 1.0f ); glVertex2f( 1.0f, 1.0f );
    glTexCoord2f( 0.0f, 1.0f ); glVertex2f( 0.0f, 1.0f );
    glEnd();
  }

  void reshape(ulong2 size)
  {
    float2 irange = gpuFunc.getIrange();
    float2 rrange = gpuFunc.getRrange();

    float icenter = ( irange.x + irange.y ) * 0.5f;
    float iwidth = ( rrange.y - rrange.x ) / size.x * size.y * 0.5f;
    irange = make_float2( icenter - iwidth, icenter + iwidth );
    gpuFunc = gpuFunctor(rrange, irange);
  }
};


drawable2DTexture *imageTexture=NULL;

void reshapeCallback( int w, int h )
{
  glViewport( 0, 0, w, h );
  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity();

  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  glOrtho( 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f );

  imageTexture->reshape( make_ulong2( h, w ) );
}

void displayCallback( void )
{
  imageTexture->draw( 1024 );
  imageTexture->display();

  glutSwapBuffers();
}

void cleanupCallback( void)
{
}

int main(int argc, char *argv[] )
{
  int height=800, width = 600;
  glutInit( &argc, argv );
  glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE );
  glutInitWindowSize( height, width);
  glutCreateWindow( *argv );

  glewInit();

  glutDisplayFunc( displayCallback );
  glutReshapeFunc( reshapeCallback );

  cudaGLSetGLDevice( 0 );

  // create the pixel image object
  imageTexture = new drawable2DTexture(make_ulong2(height,width), gpuFunctor());

  atexit( cleanupCallback );

  glutMainLoop();

  cleanupCallback();
  cudaThreadExit();
  return 0;
}
```

The executable can be built under Linux with the following `nvcc` command-line script:

```
SDK_PATH=$HOME/NVIDIA_GPU_Computing_SDK/C
INC=$SDK_PATH/common/inc
LIB=$SDK_PATH/lib
nvcc -O3 -I$INC -L$LIB mandelbrotPBO.cu -lcutil_x86_64 -lglut -lGLEW -o mandelbrotPBO
```

The following code snippet specifies a new class `drawable2DshaderARB` that is derived from the original `drawable2DTexture` class defined in mandelbrotPBO.cu. In other words, `drawable2DshaderARB` inherits the characteristics of parent class and is free to add new variables and methods as well as redefine existing methods so they can provide capabilities not previously available in the parent class.

```
class drawable2DshaderARB : public drawable2DTexture {
 private:
  uchar4 *h_Src;
  GLuint shader;

 public:
 ~drawable2DshaderARB() {
    glDeleteProgramsARB( 1, &shader );
  }
 drawable2DshaderARB(ulong2 winSize, gpuFunctor userGPUfunc) :
  drawable2DTexture(winSize, userGPUfunc) {

    deleteTexture(); // get rid of inherited texture
    deletePBO(); // get rid of inherited PBO

    h_Src = new uchar4[size.x*size.y];

    glEnable( GL_TEXTURE_2D );
    glGenTextures( 1, &tex );
    glBindTexture( GL_TEXTURE_2D, tex );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
    glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA8, size.x, size.y, 0,
                  GL_RGBA, GL_UNSIGNED_BYTE, h_Src );


    // create and bind buffer
    glGenBuffers( 1, &pbo );
    glBindBuffer( GL_PIXEL_UNPACK_BUFFER_ARB, pbo );
    glBufferData( GL_PIXEL_UNPACK_BUFFER_ARB,
                  sizeof( uchar4 ) * size.x * size.y, h_Src, GL_STREAM_COPY );
#ifdef USE_CUDA3
    cudaGraphicsGLRegisterBuffer( &pboCUDA, pbo, cudaGraphicsMapFlagsNone );
#else
    cudaGLRegisterBufferObject( pbo );
#endif

    // create shader
#ifdef COLORFUL
    const char * const code =
      "!!ARBfp1.0\n"
      "PARAM c[1] = { { 0.95, 2, 0.1 } };\n"
      "TEMP R0;\n"
      "TEMP R1;\n"
      "TEX R0.xyz, fragment.texcoord[0], texture[0], 2D;\n"
      "SIN R0.w, fragment.texcoord[0].x;\n"
      "ADD R1.x, R0, -c[0].z;\n"
      "ABS R0.w, R0;\n"
      "CMP result.color.y, -R1.x, R0.w, R0.x;\n"
      "SIN R0.w, fragment.texcoord[0].y;\n"
      "MAD R0.x, fragment.texcoord[0].y, c[0].y, fragment.texcoord[0];\n"
      "COS R0.x, R0.x;\n"
      "ABS R0.x, R0;\n"
      "ADD R1.x, R0.y, -c[0].z;\n"
      "ABS R0.w, R0;\n"
      "CMP result.color.z, -R1.x, R0.w, R0.y;\n"
      "MOV result.color.w, R0.z;\n"
      "MUL result.color.x, R0, c[0];\n"
      "END"
      ;
#else
    const char * const code =
      "!!ARBfp1.0\nTEX result.color, fragment.texcoord, texture[ 0 ], 2D;\nEND";
#endif

    glGenProgramsARB( 1, &shader );
    glBindProgramARB( GL_FRAGMENT_PROGRAM_ARB, shader );
    glProgramStringARB( GL_FRAGMENT_PROGRAM_ARB, GL_PROGRAM_FORMAT_ASCII_ARB,
                        static_cast< GLsizei >( strlen( code ) ),
                        ( GLubyte * )code );

    GLint err;
    glGetIntegerv( GL_PROGRAM_ERROR_POSITION_ARB, &err );
    if ( err != -1 ) shader = NULL;
  }
  void display( void ) {
    glBindProgramARB( GL_FRAGMENT_PROGRAM_ARB, shader );
    glEnable( GL_FRAGMENT_PROGRAM_ARB );
    glDisable( GL_DEPTH_TEST );

    glBegin( GL_QUADS );
    glTexCoord2f( 0.0f, 0.0f ); glVertex2f( 0.0f, 0.0f );
    glTexCoord2f( 1.0f, 0.0f ); glVertex2f( 1.0f, 0.0f );
    glTexCoord2f( 1.0f, 1.0f ); glVertex2f( 1.0f, 1.0f );
    glTexCoord2f( 0.0f, 1.0f ); glVertex2f( 0.0f, 1.0f );
    glEnd();

    glBindTexture( GL_TEXTURE_2D, 0 );
    glDisable( GL_FRAGMENT_PROGRAM_ARB );
  }
};
```

For simplicity, the variable `imageTexture` is defined to be of type `drawable2DshaderARB` as shown in the code snippet below:

```
drawable2DshaderARB *imageTexture=NULL;
```

Notice that the `drawable2DshaderARB` constructor contains a small shader program pointed to by the variable code. Defining the preprocessor variable COLORFUL selects one of two possible demonstration shaders:

1. If COLORFUL is not defined, a simple shader is utilized that simply moves the data from the input to the output. In this case, the same black and white Mandelbrot image created by mandelbrotPBO.cu will be displayed.
2. If COLORFUL is defined, a programmable shader will be used that changes the color of each pixel based on the original color values and position within the 2D texture.

The following character string defines the pass-through shader for `drawable2DshaderARB`:

```
const char * const code =
      "!!ARBfp1.0\nTEX result.color, fragment.texcoord, texture[ 0 ], 2D;\nEND";
```

The string is actually the shader assembly instructions generated by the Cg compiler, cgc after compiling the following program in simple1.cg:

```
struct Output {
  float4 color : COLOR;
};

Output texture(float2 texCoord : TEXCOORD0, uniform sampler2D decal)
{

  Output OUT;

  OUT.color = tex2D(decal, texCoord);

  return OUT;

}
```

The following command-line compiled the Cg program under Linux:

```
cgc -profile arbfp1 -entry texture simple1.cg > try1.arbfp1
```

The cgc compiler and Cg information can be downloaded from the NVIDIA Cg toolkit site. Ubuntu users can get an slightly older version of the Cg software by typing:

```
apt-get install nvidia-cg-toolkit
```

The shader program used when the preprocessor variable COLORFUL is defined was generated by compiling the following source code, colorful.cg:

```
struct Output {
  float4 color : COLOR;
};

Output texture(float2 texCoord : TEXCOORD0, uniform sampler2D tex)
{
  Output OUT;

  OUT.color = float4(
        0.25*abs(cos(2*texCoord.y+texCoord.x)),
        (tex2D(tex,texCoord).x > 0.1)?
                abs(sin(texCoord.x)):tex2D(tex,texCoord).x,
        (tex2D(tex,texCoord).y > 0.1)?
                abs(sin(texCoord.y)):tex2D(tex,texCoord).y,
        tex2D(tex, texCoord).z);

  return OUT;

}
```

The following command-line generated the string used in the `drawable2DshaderARB` constructor:

```
cgc -profile arbfp1 -entry texture colorful.cg > colorful.arbfp1
```

The complete buildable source for mandelbrotShader.cu including the `drawable2DshaderARB` class and shader programs is as follows:

```
#include <GL/glew.h>
#include <GL/glut.h>
#define USE_CUDA3

#include <cuda.h>
#include <cuda_gl_interop.h>

__global__ void mandelbrot( uchar4 *d_ptr, ulong2 size,
                            float2 rrange, float2 irange, unsigned long n )
{
  unsigned long x = blockIdx.x * blockDim.x + threadIdx.x;
  unsigned long y = blockIdx.y * blockDim.y + threadIdx.y;
  if ( x >= size.x || y >= size.y ) return;

  float2 C = make_float2( ( rrange.y - rrange.x ) / size.x * x + rrange.x,
                          ( irange.y - irange.x ) / size.y * y + irange.x );

  uchar4 buf = make_uchar4( 0, 0, 0, 0 );
  if ( C.x * C.x + C.y * C.y <= 4.0f ) {
    float2 z = make_float2( 0.0f, 0.0f );
    unsigned long cnt = 0;
    while ( cnt < n ) {
      ++cnt;
      z = make_float2( z.x * z.x - z.y * z.y + C.x, 2.0f * z.x * z.y + C.y );
      if ( z.x * z.x + z.y * z.y > 4.0f ) {
        unsigned char c = ( 255 - 15 ) * __powf( ( float )cnt / n, 0.8f )+15;
        buf = make_uchar4( c, c, c, 0 );
        break;
      }
    }
  }

  d_ptr[ x + y * size.x ] = buf;
}

// A functor is basically a function that maintains state
struct gpuFunctor {

protected:
  float2 rrange;
  float2 irange;

public:
gpuFunctor(float2 rrange, float2 irange) : rrange(rrange), irange(irange) {}
gpuFunctor() : rrange(make_float2(-2.f, 2.f)), irange(make_float2(-2.f, 2.f)) {}

  void callKernel(uchar4 *d_ptr, ulong2 gridSize, unsigned long convergence)
  {
    dim3 dimBlock( 16, 4 );
    dim3 dimGrid( ( gridSize.x + dimBlock.x - 1 ) / dimBlock.x,
                  ( gridSize.y + dimBlock.y - 1 ) / dimBlock.y );
    mandelbrot<<< dimGrid, dimBlock >>>( d_ptr, gridSize, rrange,
                                         irange, convergence );
  }
  inline float2 getRrange() {return(rrange);}
  inline float2 getIrange() {return(irange);}
};

class drawable2DTexture {
 protected:
  GLuint tex,pbo;
  ulong2 size;
  gpuFunctor gpuFunc;

#ifdef USE_CUDA3
  struct cudaGraphicsResource *pboCUDA;
#endif

  void createPBO()
  {
    // set up vertex data parameter
    int num_texels = size.x * size.y;
    int num_values = num_texels * 4;
    int size_tex_data = sizeof(GLubyte) * num_values;

    // Generate a buffer ID called a PBO (Pixel Buffer Object)
    glGenBuffers(1,&pbo);
    // Make this the current UNPACK buffer (OpenGL is state-based)
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo);
    // Allocate data for the buffer. 4-channel 8-bit image
    glBufferData(GL_PIXEL_UNPACK_BUFFER, size_tex_data, NULL, GL_DYNAMIC_COPY);

#ifdef USE_CUDA3
    cudaGraphicsGLRegisterBuffer( &pboCUDA, pbo, cudaGraphicsMapFlagsNone );
#else
    cudaGLRegisterBufferObject( pbo );
#endif
  }

  void deletePBO()
  {
    if(pbo) {
      // delete the PBO
#ifdef USE_CUDA3
      cudaGraphicsUnregisterResource( pboCUDA );
#else
      cudaGLUnregisterBufferObject( pbo );
#endif
      glDeleteBuffers( 1, &pbo );
      pbo=NULL;
      pboCUDA = NULL;
    }
  }

  void createTexture()
  {
    if(tex) deleteTexture();

    unsigned int image_height=size.y;
    unsigned int image_width=size.x;

    // Enable Texturing
    glEnable(GL_TEXTURE_2D);

    // Generate a texture identifier
    glGenTextures(1,&tex);

    // Make this the current texture (remember that GL is state-based)
    glBindTexture( GL_TEXTURE_2D, tex);

    // Allocate the texture memory. The last parameter is NULL since we only
    // want to allocate memory, not initialize it
    glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA8, image_width, image_height, 0, GL_BGRA,
                  GL_UNSIGNED_BYTE, NULL);

    // Must set the filter mode, GL_LINEAR enables interpolation when scaling
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    // Note: GL_TEXTURE_RECTANGLE_ARB may be used instead of
    // GL_TEXTURE_2D for improved performance if linear interpolation is
    // not desired. Replace GL_LINEAR with GL_NEAREST in the
    // glTexParameteri() call
  }

  void deleteTexture()
  {
    if(tex) {
      glDeleteTextures(1, &tex);
      tex = NULL;
    }
  }

  void cleanup (void) { deletePBO(); deleteTexture(); }

 public:

  ~drawable2DTexture() { cleanup(); }

  drawable2DTexture(ulong2 winSize, gpuFunctor userGPUfunc)
    {
      size = winSize;
      gpuFunc = userGPUfunc;
#ifdef USE_CUDA3
      pboCUDA = NULL;
#endif

      createTexture();
      createPBO();
    }
  void draw( unsigned long convergence )
  {
    uchar4 *d_ptr = NULL;

#ifdef USE_CUDA3
```

```
    size_t start;
    cudaGraphicsMapResources( 1, &pboCUDA, NULL );
    cudaGraphicsResourceGetMappedPointer( ( void ** )&d_ptr, &start, pboCUDA );
#else
    cudaGLMapBufferObject( ( void ** )&d_ptr, pbo );
#endif

    gpuFunc.callKernel(d_ptr, size, convergence);
    cudaThreadSynchronize();

#ifdef USE_CUDA3
    cudaGraphicsUnmapResources( 1, &pboCUDA, NULL );
#else
    cudaGLUnmapBufferObject( pbo );
#endif

    glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo);

    glBindTexture( GL_TEXTURE_2D, tex );
    glTexSubImage2D( GL_TEXTURE_2D, 0, 0, 0, size.x, size.y,
                     GL_RGBA, GL_UNSIGNED_BYTE, NULL );

  }

  void display( void )
  {
    glDisable( GL_DEPTH_TEST );

    glBegin( GL_QUADS );
    glTexCoord2f( 0.0f, 0.0f ); glVertex2f( 0.0f, 0.0f );
    glTexCoord2f( 1.0f, 0.0f ); glVertex2f( 1.0f, 0.0f );
    glTexCoord2f( 1.0f, 1.0f ); glVertex2f( 1.0f, 1.0f );
    glTexCoord2f( 0.0f, 1.0f ); glVertex2f( 0.0f, 1.0f );
    glEnd();
  }

  void reshape(ulong2 size)
  {
    float2 irange = gpuFunc.getIrange();
    float2 rrange = gpuFunc.getRrange();

    float icenter = ( irange.x + irange.y ) * 0.5f;
    float iwidth = ( rrange.y - rrange.x ) / size.x * size.y * 0.5f;
    irange = make_float2( icenter - iwidth, icenter + iwidth );
    gpuFunc = gpuFunctor(rrange, irange);
  }
};

class drawable2DshaderARB: public drawable2DTexture {
 private:
  uchar4 *h_Src;
  GLuint shader;

 public:
 ~drawable2DshaderARB() {
    glDeleteProgramsARB( 1, &shader );
 }
 drawable2DshaderARB(ulong2 winSize, gpuFunctor userGPUfunc) :
  drawable2DTexture(winSize, userGPUfunc) {

    deleteTexture(); // get rid of inherited texture
    deletePBO(); // get rid of inherited PBO

    h_Src = new uchar4[size.x*size.y];

    glEnable( GL_TEXTURE_2D );
    glGenTextures( 1, &tex );
    glBindTexture( GL_TEXTURE_2D, tex );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
    glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA8, size.x, size.y, 0,
                  GL_RGBA, GL_UNSIGNED_BYTE, h_Src );


    // create and bind buffer
    glGenBuffers( 1, &pbo );
    glBindBuffer( GL_PIXEL_UNPACK_BUFFER_ARB, pbo );
    glBufferData( GL_PIXEL_UNPACK_BUFFER_ARB,
                  sizeof( uchar4 ) * size.x * size.y, h_Src, GL_STREAM_COPY );
#ifdef USE_CUDA3
    cudaGraphicsGLRegisterBuffer( &pboCUDA, pbo, cudaGraphicsMapFlagsNone );
#else
    cudaGLRegisterBufferObject( pbo );
```

```
#endif

    // create shader
#ifdef COLORFUL
    const char * const code =
      "!!ARBfp1.0\n"
      "PARAM c[1] = { { 0.95, 2, 0.1 } };\n"
      "TEMP R0;\n"
      "TEMP R1;\n"
      "TEX R0.xyz, fragment.texcoord[0], texture[0], 2D;\n"
      "SIN R0.w, fragment.texcoord[0].x;\n"
      "ADD R1.x, R0, -c[0].z;\n"
      "ABS R0.w, R0;\n"
      "CMP result.color.y, -R1.x, R0.w, R0.x;\n"
      "SIN R0.w, fragment.texcoord[0].y;\n"
      "MAD R0.x, fragment.texcoord[0].y, c[0].y, fragment.texcoord[0];\n"
      "COS R0.x, R0.x;\n"
      "ABS R0.x, R0;\n"
      "ADD R1.x, R0.y, -c[0].z;\n"
      "ABS R0.w, R0;\n"
      "CMP result.color.z, -R1.x, R0.w, R0.y;\n"
      "MOV result.color.w, R0.z;\n"
      "MUL result.color.x, R0, c[0];\n"
      "END"
      ;
#else
    const char * const code =
      "!!ARBfp1.0\nTEX result.color, fragment.texcoord, texture[ 0 ], 2D;\nEND";
#endif

    glGenProgramsARB( 1, &shader );
    glBindProgramARB( GL_FRAGMENT_PROGRAM_ARB, shader );
    glProgramStringARB( GL_FRAGMENT_PROGRAM_ARB, GL_PROGRAM_FORMAT_ASCII_ARB,
                        static_cast< GLsizei >( strlen( code ) ),
                        ( GLubyte * )code );

    GLint err;
    glGetIntegerv( GL_PROGRAM_ERROR_POSITION_ARB, &err );
    if ( err != -1 ) shader = NULL;
  }
  void display( void ) {
    glBindProgramARB( GL_FRAGMENT_PROGRAM_ARB, shader );
    glEnable( GL_FRAGMENT_PROGRAM_ARB );
    glDisable( GL_DEPTH_TEST );

    glBegin( GL_QUADS );
    glTexCoord2f( 0.0f, 0.0f ); glVertex2f( 0.0f, 0.0f );
    glTexCoord2f( 1.0f, 0.0f ); glVertex2f( 1.0f, 0.0f );
    glTexCoord2f( 1.0f, 1.0f ); glVertex2f( 1.0f, 1.0f );
    glTexCoord2f( 0.0f, 1.0f ); glVertex2f( 0.0f, 1.0f );
    glEnd();

    glBindTexture( GL_TEXTURE_2D, 0 );
    glDisable( GL_FRAGMENT_PROGRAM_ARB );
  }
};


drawable2DshaderARB *imageTexture=NULL;

void reshapeCallback( int w, int h )
{
  glViewport( 0, 0, w, h );
  glMatrixMode( GL_MODELVIEW );
  glLoadIdentity();

  glMatrixMode( GL_PROJECTION );
  glLoadIdentity();
  glOrtho( 0.0f, 1.0f, 0.0f, 1.0f, 0.0f, 1.0f );

  imageTexture->reshape( make_ulong2( h, w ) );
}

void displayCallback( void )
{
  imageTexture->draw( 1024 );
  imageTexture->display();

  glutSwapBuffers();
}

void cleanupCallback( void)
{
}
```

```
int main(int argc, char *argv[] )
{
  int height=1024, width = 768;
  glutInit( &argc, argv );
  glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE );
  glutInitWindowSize( height, width);
  glutCreateWindow( *argv );

  glewInit();

  glutDisplayFunc( displayCallback );
  glutReshapeFunc( reshapeCallback );

  cudaGLSetGLDevice( 0 );

  // create the pixel image object
  imageTexture = new drawable2DshaderARB(make_ulong2(height,width), gpuFunctor());

  atexit( cleanupCallback );

  glutMainLoop();

  cleanupCallback();
  cudaThreadExit();
  return 0;
}
```

The black-and-white executable mandelbrotShaderBW (that uses the pass-through shader simple1.cg) was built with the following under Linux:

```
SDK_PATH=/$HOME/NVIDIA_GPU_Computing_SDK/C
INC=$SDK_PATH/common/inc
LIB=$SDK_PATH/lib
nvcc -DCOLORFUL -O3 -I$INC -L$LIB mandelbrotShader.cu -lcutil_x86_64 -lglut -lGLEW -o mandelbrotShaderColor
```

Adding the following preprocessor specification -DCOLORFUL to the command line creates the mandelbrotShaderColor executable:

```
SDK_PATH=$HOME/NVIDIA_GPU_Computing_SDK/C
INC=$SDK_PATH/common/inc
LIB=$SDK_PATH/lib
nvcc -DCOLORFUL -O3 -I$INC -L$LIB mandelbrotShader.cu -lcutil_x86_64 -lglut -lGLEW -o mandelbrotShaderColor
```

Running mandelbrotShaderColor produces Figure 1 on the screen:

[Click image to view at full size]

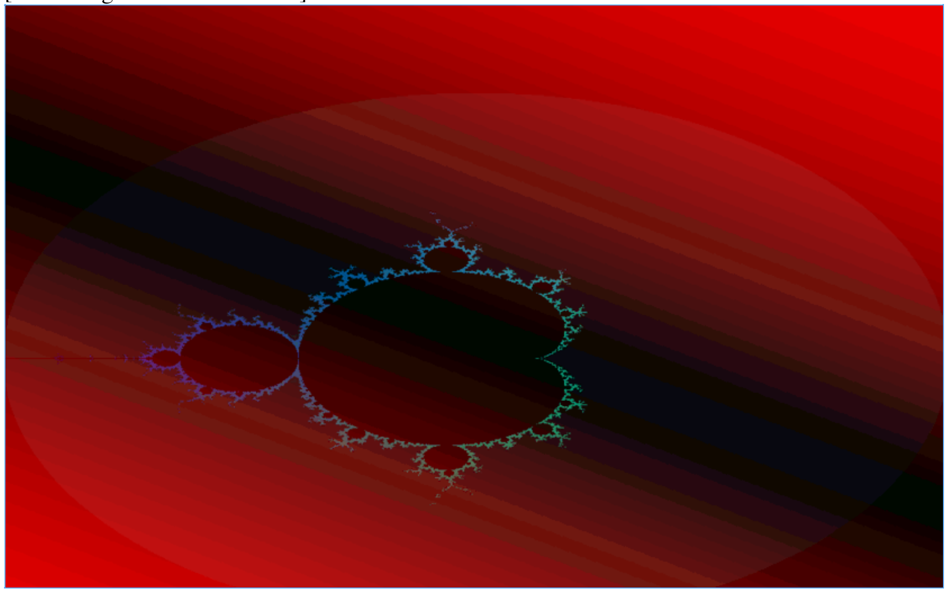

Figure 1

Summary

CUDA 3.0 is a major revision number release that delivers important benefits for debugging; C++; OpenCL; CUDA driver and runtime developers; CUBLAS and CUFFT users, and many other areas that make this a "must install" update. The expanded capabilities discussed in this brief two-part article series should provide outstanding food-for-thought how the 3.0 release creates new opportunities for code development and integration of existing software projects.

In addition, this article provided examples that touched on a number of exciting capabilities that will be discussed in greater depth in future articles.

- 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.