CUDA, Supercomputing for the Masses: Part 13

Using texture memory in CUDA

June 23, 2009
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/218100902

In CUDA, Supercomputing for the Masses: Part 12 of this article series on CUDA, I took a quick detour to discuss some of the paradigm changing features of the latest CUDA Toolkit 2.2 release. This article resumes the discussion of "texture memory" which I began in Part 11 of this series. In addition, this installment includes information on the new CUDA Toolkit 2.2 texture capability that allows some programs to eliminate extra copies by providing the ability to write to global memory on the GPU that has a 2D texture bound to it.

From a C-programmer's perspective, texture memory provides an unusual combination of cache memory (separate from register, global, and shared memory), local processing capability (separate from the scalar processors), and a way to interact with the display capabilities of the GPU. This article focuses on the cache and local processor capabilities of texture memory while the next column will discuss how to perform viewable graphic operations with the GPU.

Don't be put off from using texture memory because it is different and has many options. The use of texture memory can improve performance for both bandwidth and latency limited programs. For example, some programs can exceed the maximum theoretical memory bandwidth of the underlying global memory through judicious use of the texture memory cache. While the latency of texture cache reference is generally the same as DRAM, there are some special cases that can deliver data with slightly less than 100 cycles of latency. As usual in CUDA, the use of many threads can hide memory access latency regardless if texture cache or global memory is being accessed.

For CUDA programmers, the most salient points about using texture memory as a cache are: it is optimized for 2D spatial locality, very small (effectively about 8KB per multiprocessor), and can provide a performance benefit by having all the threads in a warp access nearby locations in the texture (as demonstrated in Cache-Efficient Numerical Algorithms using Graphics Hardware). Another tip from the forums is to pack data up if you can because a single `float4` texture read is faster than four separate `float` texture reads.

One ingenious mapping of a random-access data structure to texture memory has been implemented by the CUDA-EC software. In the CUDA code, NVIDIA implements a Bloom filter to test for set membership. The CUDA-EC software is available for free download at http://cuda-ec.sourceforge.net/.

The CUDA Toolkit 2.2 introduced the ability to write to 2D textures bound to pitch linear memory on the GPU that has a texture bound to it. In other words, the data within the texture can be updated within a kernel running on the GPU. This is a very nice feature because it allows many codes to better utilize the caching behavior of texture memory while also eliminating copies. One common example that immediately springs to mind are calculations that require two passes through the data: one to calculate a value (such as a mean or maximum) and a second pass to update the data in place. Such calculations are common when changing the data range or calculating probabilities. The use of an updatable texture can potentially speed these types of calculations.

The cuBLAS library uses texture memory for many of the single-pass calculations (`sasum`, `sdot`, and etc). However, comments in the source code indicate that texture memory should not be used for vectors that are short or those that are aligned and have unit stride and thus have nicely coalesced behavior. (The source for cuBLAS library and cuFFT are available for those who have signed up as NVIDIA developers.)

Texture cache is part of each TPC, here short for "Thread Processing Cluster" since I am discussing operations in *compute mode*. (TPC stands for "Texture Processing Cluster" in *graphics mode*, which I don't address in this article.) Each TPC contains multiple streaming multiprocessors and a single texture cache. It is important to note that in the GTX 200 series, the texture cache supports three SM (Streaming Multiprocessors) per TPC while the G80/G92 architecture only supports two.

Figure 1 depicts a high-level view of the GeForce GTX 280 GPU in parallel computing mode: A hardware-based thread scheduler at the top manages scheduling threads across the TPCs, which includes the texture caches and memory interface units. The elements indicated as "atomic" refer to the ability to perform atomic read-modify-write operations to memory. For more information, please see GeForce GTX 200 GPU Technical Brief.

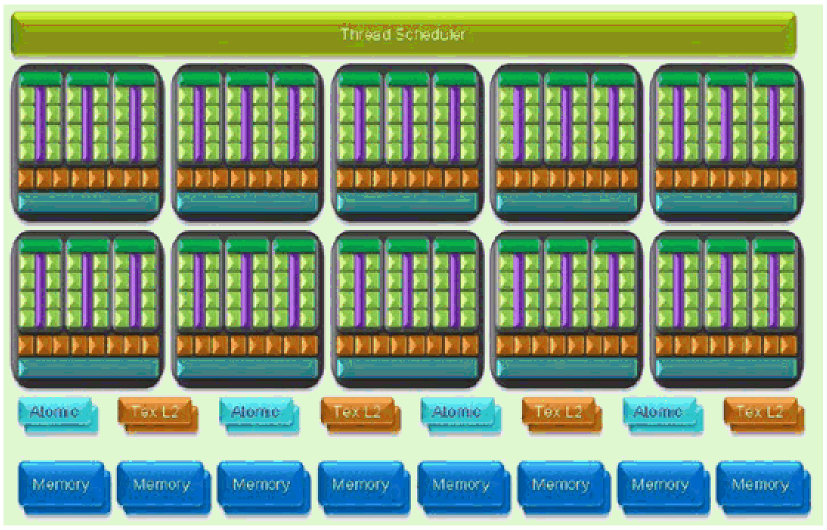

Figure 1: High-Level view of GTX 280 Architecture (Courtesy NVIDIA).

Figure 2 represents a lower-level view of a single TPC. Note that TF stands for "Texture Filtering" and IU is the abbreviation for "Instruction Unit".

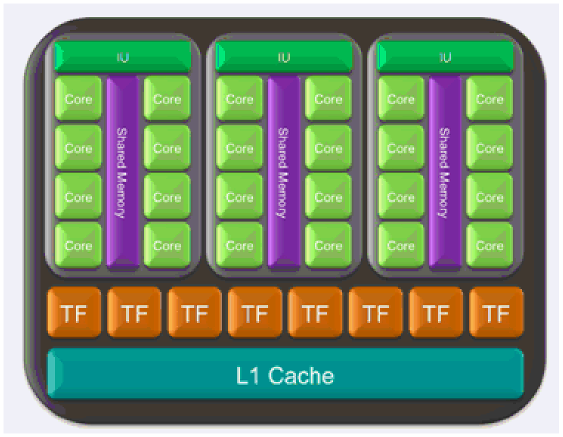

Figure 2: Lower-level view of a single GTX 280 TPC (Courtesy NVIDIA).

Textures are bound to global memory and can provide both cache and some processing capabilities. How the global memory was created dictates some of the capabilities the texture can provide. For this reason, it is important to distinguish between three memory types that can be bound to a texture:

Memory type	How created	Texture capability	Texture update
Linear memory	**cudaMalloc()**	1. Acts as a linear cache	Free to write to the global memory from threads if the incoherence is safe

CUDA arrays	**cudaMallocArray()**
	cudaMalloc3D()
		1. Cache optimized for spatial locality
		2. interpolation, wrapping, and clamping
			Writing to arrays from a kernel is not allowed

2D pitch linear memory	**cudaMallocPitch()**	1. Cache optimized for spatial locality 2. Interpolation, wrapping, clamping	Free to write to the global memory from threads if the incoherence is safe

Table 1: Distinguishing between memory types.

Linear Memory

It is important to distinguish between "linear memory" created with `cudaMalloc()` and "pitch linear" memory created with `cudaMallocPitch()`. In a nutshell, both methods create linear memory but `cudaMallocPitch()` pads the allocation to get best performance for the memory subsystem of a given piece of hardware. A programmer can create memory with `cudaMalloc()` and manually set the pitch, but best memory performance may not be achieved. Aside from the ability to update, there is little difference between a texture bound to a 2D CUDA array and pitch linear memory. NVIDIA has indicated there is effectively no difference between textures bound to these two types of memory.

There are two cases to consider when binding the texture to global memory that necessitated distinguishing between pitch linear memory and linear memory:

- When using the texture only as a cache: In this case programmers might consider binding the texture to *linear memory* created with `cudaMalloc()` because the texture unit cache is small and caching the padding added by `cudaMallocPitch()` would be wasteful.
- When using the texture to perform some processing: In this case it is important to bind the texture to *pitch linear memory* created with `cudaMallocPitch()` so that the texture unit boundary processing works correctly. In other words, don't bind linear memory created with `cudaMalloc()` (and manually set the pitch) to a texture because unexpected things might happen.

The use of `cudaMallocPitch()` is generally recommended because it "knows" what pitch is appropriate for a given piece of hardware to get the best performance and is a good way to future-proof your code.

Also note that CUDA arrays are an opaque data storage mechanism, and are composed of elements, each of which has 1, 2 or 4 components that may be signed or unsigned 8-, 16- or 32-bit integers, 16-bit floats (CUDA driver only), or 32-bit floats. You can also use `int2` and `hiloint2double` to use double-precision values. Note that CUDA arrays may get reordered for locality on the GPU.

Binding memory to a texture is quite fast and unlikely to have an appreciable impact on program performance. There are some restrictions and additional caveats:

- Updates to the memory backing the texture cache are not seen until the next kernel invocation.
    - In other words, a thread can safely read via texture some memory location only if this memory location has been updated by a previous kernel call or memory copy, but not if it has been previously updated by the same thread or another thread from the same kernel call.
- Binding a texture to linear memory effectively prevents utilization of the texture for texture processing. An important note from the documentation (section 3.2.4.3 of the Programming Guide) is that the texture reference fields `normalized`, `addressMode`, and `filterMode` may be modified in host code, but only for texture references bound to CUDA arrays and pitch linear memory! This effectively means that textures bound to linear memory cannot be used to perform texture processing with the texture units.
- Texture memory cannot bind to mapped memory.

As noted, the texture cache is optimized for 2D spatial locality with streaming behavior when bound to pitch linear memory or CUDA arrays. However, this does not tell us how to order the data to get the best performance out of the texture unit when using it as a cache. There is a good thread on the CUDA Zone forums discussing how to order 3D data for best performance. One of the suggestions is to use a Z-order curve to map multidimensional data to 1D while preserving locality. The challenging question of how to best order your data for cache locality is further complicated because the methods used by the graphics hardware may change at some point in the future to better meet customer needs.

Depending on how the global memory bound to the texture was created, there are several possible ways to fetch from the texture that might also invoke some form of texture processing by the texture.

The simplest way to fetch data from a texture is by using `tex1Dfetch()` because:

- Only integer addressing is supported.
- No additional filtering or addressing modes are provided.

Use of the methods `tex1D()`, `tex2D()`, and `tex3D()` are more complicated because the interpretation of the texture coordinates, what processing occurs during the texture fetch, and the return value delivered by the texture fetch are all controlled by setting the texture reference's mutable (runtime) and immutable (compile time) attributes:

- Immutable parameters (compile-time)
    - Type: type returned when fetching
        - Basic integer and float types
        - CUDA 1-, 2-, 4-element vectors
    - Dimensionality:
        - Currently 1D, 2D, or 3D
    - Read Mode:
        - cudaReadModeElementType
        - cudaReadModeNormalizedFloat (valid for 8- or 16-bit integers)
            - returns [-1,1] for signed, [0,1] for unsigned
- Mutable parameters (run-time, only for array-textures and pitch linear-memory)
    - Normalized:
        - non-zero = addressing range [0, 1]
    - Filter Mode:

    - cudaFilterModePoint
    - cudaFilterModeLinear

  - Address Mode:

    - cudaAddressModeClamp
    - cudaAddressModeWrap

For more detailed information, please consult the CUDA Programming Guide.

By default, textures are referenced using floating-point coordinates in the range [0, N) where N is the size of the texture in the dimension corresponding to the coordinate. Specifying normalized texture coordinates will be used implies all references will be in the range [0,1).

The wrap mode specifies what happens for out-of-bounds addressing:

- Wrap: out-of-bounds coordinates are wrapped (via modulo arithmetic)

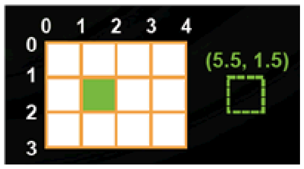

Figure 3: Wrap mode (Courtesy NVIDIA)

- Clamp: out-of-bounds coordinates are replaced with the closest boundary:

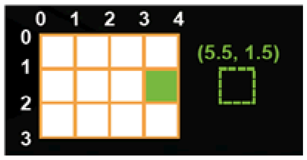

Figure 4: Clamp mode (Courtesy NVIDIA)

Linear texture filtering may be done only for textures that are configured to return floating-point data. A texel, short for "texture element", is an element of a texture array. Thus, linear texture filtering performs low-precision (9-bit fixed-fixed point with 8-bits of fractional value) interpolation between neighboring texels. When enabled, the texels surrounding a texture fetch location are read and the return value of the texture fetch is interpolated by the texture hardware based on where the texture coordinates fell between the texels. Simple linear interpolation is performed for one-dimensional textures as can be seen in the following equation from Appendix D.2 of the NVIDIA CUDA Programming Guide 2.2:

```
tex(x) = (1- α)T[i] + αT[i +1]
```

`Equation 1`: Filtering mode for a one-dimensional texture.

Similarly, the dedicated texture hardware will perform bilinear and trilinear filtering for higher-dimensional data. (For more information check out the free online GPU Gems books, and the Wikipedia articles on texture filtering.)

An Example

Let's take a look at the following very simple example, readTexels.cu, which demonstrates how to bind a texture to a CUDA array and sets the `filterMode` attribute to `cudaFilterModeLinear`.

```
//readTexels.cu
#include <stdio.h>

void checkCUDAError(const char *msg) {
  cudaError_t err = cudaGetLastError();
  if( cudaSuccess != err) {
    fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString( err) );
    exit(EXIT_FAILURE);
  }
}

texture<float, 1, cudaReadModeElementType> texRef;

__global__ void readTexels(int n, float *d_out)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;

  if(idx < n) {
    //Note: Appendix D.2 gives formula for interpolation
    float x = tex1D(texRef, float(idx));
    d_out[idx] = x;
  }
}

#define NUM_THREADS 256

int main()
{
  int N = 10; // 10 is illustrative and should be larger in practice
  int nBlocks = N/NUM_THREADS + ((N % NUM_THREADS)?1:0);
  float *d_out;

  // allocate space on the device for the results
  cudaMalloc((void**)&d_out, sizeof(float) * N);

  // allocate space on the host for the results
  float *h_out = (float*)malloc(sizeof(float)*N);

  // data fill array with increasing values
  float *data = (float*)malloc(N*sizeof(float));
  for (int i = 0; i <  N; i++) data[i] = float(i);

  // create a CUDA array on the device
  cudaArray* cuArray;
  cudaMallocArray (&cuArray, &texRef.channelDesc, N, 1);
  cudaMemcpyToArray(cuArray, 0, 0, data, sizeof(float)*N, cudaMemcpyHostToDevice);

  // bind a texture to the CUDA array
  cudaBindTextureToArray (texRef, cuArray);

  // host side settable texture attributes
  texRef.normalized = false;
  texRef.filterMode = cudaFilterModeLinear;

  // read texels from texture
  readTexels<<<nBlocks, NUM_THREADS>>>(N, d_out);

  // copy texels to host
  cudaMemcpy(h_out, d_out, sizeof(float)*N, cudaMemcpyDeviceToHost);

  // look at them
  for (int i = 0; i << N; i++) {
    printf("%f\n",h_out[i]);
  }

  free(h_out);

  cudaFree(d_out);
  cudaFreeArray(cuArray);
  cudaUnbindTexture(texRef);
  checkCUDAError("cuda free operations");
}
```

Under Linux, the following nvcc command-line can be used to build this program:

```
nvcc readTexel.cu –o readTexel
```

On the host side, the texture reference, texRef, is created with:

```
texture<float, 1, cudaReadModeElementType> texRef;
```

A CUDA array, `cuArray`, is allocated and initialized:

```
  // create a CUDA array on the device
  cudaArray* cuArray;
  cudaMallocArray (&cuArray, &texRef.channelDesc, N, 1);
```

The `texRef` texture is then bound to `cuArray` and the texture attributes are set. In this case, we specify linear interpolation and we will not be using normalized texture coordinates.

```
// bind a texture to the CUDA array
  cudaBindTextureToArray (texRef, cuArray);

  // host side settable texture attributes
  texRef.normalized = false;
  texRef.filterMode = cudaFilterModeLinear;
```

The kernel, `readTexels()`, simply fetches values from the texture unit and places them in the `d_out` array.

```
    //Note: Appendix D.2 gives formula for interpolation
    float x = tex1D(texRef, float(idx));
    d_out[idx] = x;
```

The `d_out` array is then copied back to the host and printed out on the screen. Finally, the texture is released with the call:

```
  cudaUnbindTexture(texRef);
```

Playing with the attributes and data in this simple example might help clarify the processing capabilities of texture memory. For this example, you should see the following output demonstrating that the texture is interpolating between data points.

```
0.000000
0.500000
1.500000
2.500000
3.500000
4.500000
5.500000
6.500000
7.500000
8.500000
```

Example 1: Binding a texture to linear memory that is updated in-place.

The following simple example, negateArray.cu, binds a 1D texture to linear memory. The texture is used to fetch floating-point values from the linear memory and the texture is then updated in-place. The results are then brought back to the host and checked for correctness.

```
#include <stdio.h>
#include <assert.h>

void checkCUDAError(const char *msg) {
  cudaError_t err = cudaGetLastError();
  if( cudaSuccess != err) {
    fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString( err) );
    exit(EXIT_FAILURE);
  }
}

texture<float, 1, cudaReadModeElementType> texRef;

__global__ void kernel(int n, float *d_out)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;

  if(idx < n) {
    d_out[idx] = -tex1Dfetch(texRef, idx);
  }
}

#define NUM_THREADS 256
int main()
{
  int N = 2560;
  int nBlocks = N/NUM_THREADS + ((N % NUM_THREADS)?1:0);
  int memSize = N*sizeof(float);

  // data fill array with increasing values
  float *data;
  data = (float*) malloc(memSize);
  for (int i = 0; i < N; i++) data[i] = float(i);

  float *d_a;
  cudaMalloc( (void **) &d_a, memSize );
  cudaMemcpy( d_a, data, memSize, cudaMemcpyHostToDevice );

  cudaBindTexture(0,texRef,d_a,memSize);
  checkCUDAError("bind");

  kernel<<<nBlocks, NUM_THREADS>>>(N, d_a);
  float *h_out = (float*)malloc(memSize);
  cudaMemcpy(h_out, d_a, memSize, cudaMemcpyDeviceToHost);
  checkCUDAError("cudaMemcpy");

  for (int i = 0; i <<N; i++) {
    assert(data[i] == -h_out[i]);
  }
  printf("Correct\n");

  cudaUnbindTexture(texRef);
  checkCUDAError("cudaUnbindTexture");

  free(h_out);
  free(data);
}
```

There are a few minor but important differences between negateArray.cu and the previous readTexels.cu example.

The first real difference is that we allocate a linear region of memory, `d_a`, with `cudaMalloc()`:

```
  float *d_a;
  cudaMalloc( (void **) &d_a, memSize );
```

This linear memory is bound to a texture with the following:

```
  cudaBindTexture(0,texRef,d_a,memSize);
  checkCUDAError("bind");
```

On the device, `tex1Dfetch()` is used to fetch the data, which is then negated and written to `d_out`:

```
    d_out[idx] = -tex1Dfetch(texRef, idx);
```

Please note that the kernel call passed `d_a`, which means that the data is updated in-place:

```
  kernel<<<nBlocks, NUM_THREADS>>>(N, d_a);
```

Example 2: Revisiting the reverseArray_multiblock.cu example

Finally, let's revisit the reverseArray_multiblock.cu example, which was discussed in detail in Part 3 of this series and adapt it to use texture memory. As can be seen in the source for reverseArray_multiblockTexture.cu below, only a few minor changes were needed to change from using a linear array to a texture object bound to the linear region of memory, `d_a`, allocated with cudaMalloc. For convenience, changes from reverseArray_multiblock.cu are highlighted with red and the "* Texture Specific *" string.

```
// reverseArray_multiblockTexture.cu

// includes, system
#include <stdio.h>
#include <assert.h>

// Simple utility function to check for CUDA runtime errors
void checkCUDAError(const char* msg);

// ****************** Texture Specific *******************
// Note: default mode is cudaReadModeElementType
// section 4.3.4.1 of the NVIDIA CUDA Programming Guide
texture<int, 1> tex_d_a;

// Part3: implement the kernel
__global__ void reverseArrayTexture(int *d_out, int *d_in)
{
  int inOffset = blockDim.x * blockIdx.x;
  int outOffset = blockDim.x * (gridDim.x - 1 - blockIdx.x);
  int in = inOffset + threadIdx.x;
  int out = outOffset + (blockDim.x - 1 - threadIdx.x);

  // ****************** Texture Specific *******************
  d_out[out] = tex1Dfetch(tex_d_a,in);
}

// Program main
int main( int argc, char** argv)
{
  // pointer for host memory and size
  int *h_a; int dimA = 256 * 1024; // 256K elements (1MB total)

  // pointer for device memory
  int *d_b, *d_a;

  // define grid and block size
  int numThreadsPerBlock = 256;

  // Part 1: compute number of blocks needed based on
  // array size and desired block size
  int numBlocks = dimA / numThreadsPerBlock;

  // allocate host and device memory
  size_t memSize = numBlocks * numThreadsPerBlock * sizeof(int);
  h_a = (int *) malloc(memSize);
  cudaMalloc( (void **) &d_a, memSize );
  cudaMalloc( (void **) &d_b, memSize );

 // ****************** Texture Specific *******************
  // Bind the device array d_a to a texture object tex_d_a
  cudaBindTexture(NULL,tex_d_a,d_a);
  checkCUDAError("Bind Texture");

  // Initialize input array on host
  for (int i = 0; i < dimA; ++i) { h_a[i] = i; }

  // Copy host array to device array
  cudaMemcpy( d_a, h_a, memSize, cudaMemcpyHostToDevice );

  // launch kernel
  dim3 dimGrid(numBlocks);
  dim3 dimBlock(numThreadsPerBlock);
  reverseArrayTexture<<< dimGrid, dimBlock >>>( d_b, d_a );

  // block until the device has completed
  cudaThreadSynchronize();

  // check if kernel execution generated an error
  // Check for any CUDA errors
  checkCUDAError("kernel invocation");

  // device to host copy
  cudaMemcpy( h_a, d_b, memSize, cudaMemcpyDeviceToHost );

  // Check for any CUDA errors
  checkCUDAError("memcpy");

  // verify the data returned to the host is correct
  for (int i = 0; i < dimA; i++) { assert(h_a[i] == dimA - 1 - i ); }

 // ****************** Texture Specific *******************
  cudaUnbindTexture(tex_d_a);
  checkCUDAError("Unbind Texture");

  // free device memory
  cudaFree(d_a); cudaFree(d_b);

  // free host memory
  free(h_a);

  // If the program makes it this far, then the results are
  // correct and there are no run-time errors. Good work!
  printf("Correct!\n"); return 0;
}

void checkCUDAError(const char *msg) {
  cudaError_t err = cudaGetLastError();
  if( cudaSuccess != err) {
    fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString( err) );
    exit(EXIT_FAILURE);
  }
}
```

In short summary form, CUDA texturing requires the following steps:

- Host (CPU) code
  - Allocate/obtain memory (linear memory, pitch linear memory, or CUDA array)
  - Create a texture reference object
    - Currently must be at file-scope

    - Bind the texture reference to memory/array
    - When done
        - Unbind the texture reference, free resources

- Device (kernel) code
    - Fetch using texture reference
    - Linear memory textures
        - `tex1Dfetch`
    - Array textures and pitch linear memory
        - `tex1D`, `tex2D`, or tex3D

This structure can be seen in reverseArray_multiblockTexture.cu:

- Host (CPU) code:

```
// reverseArray_multiblockTexture.cu
   ...
// ****************** Texture Specific *******************
// Note: default mode is cudaReadModeElementType
// section 4.3.4.1 of the NVIDIA CUDA Programming Guide
texture<int, 1> tex_d_a;

// Program main
int main( int argc, char** argv)
{
    ...
    // pointer for device memory
  int *d_b, *d_a;
  ...
  cudaMalloc( (void **) &d_a, memSize );
...
  // ****************** Texture Specific *******************
  // Bind the device array d_a to a texture object tex_d_a
  cudaBindTexture(NULL,tex_d_a,d_a);
  checkCUDAError("Bind Texture");
...
     // ****************** Texture Specific *******************
  cudaUnbindTexture(tex_d_a);
  checkCUDAError("Unbind Texture");
...
}
```

- Device (kernel) code:

```
// Part3: implement the kernel
__global__ void reverseArrayTexture(int *d_out, int *d_in)
{
  // ****************** Texture Specific *******************
  d_out[out] = tex1Dfetch(tex_d_a,in);
}
```

Conclusion

This installment presents some straightforward examples that demonstrate the use of texture objects with CUDA. As discussed, texture memory provides many more capabilities aside from those demonstrated in the reverseArray_multiblockTexture.cu example. Please look to the example codes in the NVIDIA_CUDA_SDK projects folder for more complicated examples. The Internet also contains many more useful examples as well that you can download and try. Following are two possibilities:

- A potentially helpful CUDA 3D texture example is located in the cookbook examples at the Google cudaiap2009 "cuda@mit" site.
- The CIRL fuzzy logic tutorial.

An excellent resource to learn more about the texture cache and other methods for data reuse on GPUs is Mark Silberstein's paper Efficient Computation of Sum-products on GPUs Through Software Managed Cache. As suggested at the beginning of this article, the paper by Govindaraju and Manocha, Cache-Efficient Numerical Algorithms using Graphics Hardware is also an excellent resource. For a more mathematically oriented view towards textures and caches, check out http://www.cs.lth.se/EDA075/notes/mgh_ch5.pdf.

For More Information

- 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.