# CUDA, Supercomputing for the Masses: Part 12

## CUDA 2.2 Changes the Data Movement Paradigm

May 14, 2009
URL:http://www.drdobbs.com/parallel/cuda-supercomputing-for-the-masses-part/217500110

In CUDA, Supercomputing for the Masses: Part 11 of this article series on CUDA, I revisited CUDA memory spaces and introduced the concept of "texture memory". In this installment, I discuss some paradigm changing features of the just released CUDA version 2.2 -- namely the introduction of "mapped" pinned system memory that allows compute kernels to share host system memory and provides zero-copy support for direct access to host system memory when running on many newer CUDA-enabled graphics processors. The next article in this series will resume the discussion of texture memory and include information about new CUDA 2.2 features such as the ability to write to global memory on the GPU that has a texture bound to it. (Go here for more on CUDA 2.2.)

Prior to CUDA 2.2, CUDA kernels could not access host system memory directly. For that reason, CUDA programmers used the design pattern introduced in Part 1 and Part 2:

1. Move data to the GPU.
2. Perform calculation on GPU.
3. Move result(s) from the GPU to host.

This paradigm has now changed as CUDA 2.2 has introduced new APIs that allow host memory to be mapped into device memory via a new function called `cudaHostAlloc` (or `cuMemHostAlloc` in the CUDA driver API). This new memory type supports the following features:

- "Portable" pinned buffers that are available to all GPUs.
    - The use of multiple GPUs will be discussed in a future article.
- "Mapped" pinned buffers that map host memory into the CUDA address space and provide asynchronous transparent access to the data without requiring an explicit programmer initiated copy.
    - Integrated GPUs share physical memory with the host processor (as opposed to the on-board fast global memory of discrete GPUs). Mapped pinned buffers act as "zero-copy" buffers for many newer (especially integrated graphics processors) because they avoid superfluous copies. When developing code for integrated GPUs, using mapped pinned memory really makes sense.
    - For discrete GPUs, mapped pinned memory is only a performance win in certain cases. Since the memory is not cached by the GPU:
        - It should be read or written exactly once.
        - The global loads and stores that read or write the memory must be coalesced to avoid a 2x-7x PCIe performance penalty.
        - At best, it will only deliver PCIe bandwidth performance, but this can be 2x faster than `cudaMemcpy` because mapped memory is able exploit the full duplex capability of the PCIe bus by reading and writing at the same time. A call to `cudaMemcpy` can only move data in one direction at a time (i.e., half duplex).

  Further, a drawback of the current CUDA 2.2 release is that all pinned allocations are mapped into the GPU's 32-bit linear address space, regardless of whether the device pointer is needed or not. (NVIDIA indicates this will be changed to a per-allocation basis in a later release.)
- "WC" (write-combined) memory can provide higher performance:
    - Since WC memory is neither cached or cache coherent, greater PCIe performance can be achieved because the memory is not snooped during transfers across the PCI Express bus. NVIDIA notes in their "CUDA 2.2 Pinned Memory APIs" document that WC memory may perform as much as 40% faster on certain PCI Express 2.0 implementations.
    - It may increase the host processor(s) write performance to host memory because individual writes are first combined (via an internal processor write-buffer) so that only a single burst write containing many aggregated individual writes need be issued. (Intel claims they have observed actual performance increases of over 10x but this is not typical). For more information, please see the Intel publication Write Combining Memory Implementation Guidelines.
    - Host-side calculations and applications may run faster because write-combined memory does not pollute the internal processor caches such as the L1 and L2 caches. This happens because WC does not enforce cache coherency, which can increase host processor efficiency by reducing cache misses as well as avoiding the overhead incurred when enforcing cache coherency. Write-combining also avoids cache pollution by utilizing a separate dedicated internal write-buffer cache, which by-passes and leaves the other internal processor caches untouched.
    - WC memory does have drawbacks and CUDA programmers should not consider a WC memory region as general-purpose memory because it is "weakly-ordered". In other words, reading from a WC memory location may return unexpected -- and incorrect -- data because a previous write to that memory location might have been delayed in order to combine it with other writes. Without programmer enforced coherency though

a "fence" operation, it is possible that a read of WC memory may actually "read" old or even initialized data.

- Unfortunately, enforcing coherent reads from WC memory may incur a performance penalty on some host processor architectures. Happily, processors with the SSE4 instruction set provide a streaming load instruction (`MOVNTDQA`) that can efficiently read from WC memory. (Check if the `CPUID` instruction is executed with `EAX==1`, bit 19 of `ECX`, to see if SSE4.1 is available.) Please see the Intel publication, Increasing Memory Throughput With Intel Streaming SIMD Extensions 4 (Intel SSE4) Streaming Load.
- It is unclear if and when a CUDA programmer needs to take any action (such as using a memory fence) to ensure that the WC memory is in-place and ready for use by the host or graphics processor(s). The Intel documentation states that "[a] 'memory fence' instruction should be used to properly ensure consistency between the data producer and data consumer." The CUDA driver does use WC memory internally and must issue a store fence instruction whenever it sends a command to the GPU. For this reason, the NVIDIA documentation notes, "the application may not have to use store fences at all" (emphasis added). A rough rule of thumb that appears to work is to look to the CUDA commands prior to referencing WC memory and assume they issue a fence instruction. Otherwise, utilize your compiler intrinsic operations to issue a store fence instruction and guarantee that every preceding store is globally visible. This is compiler dependent. Linux compilers will probably understand the `_mm_sfence` intrinsic while Windows compilers will probably use `_WriteBarrier`.

Each of these memory features can be used individually or in any combination -- you can allocate a portable, write-combined buffer, a portable pinned buffer, a write-combined buffer that is neither portable nor pinned, or any other permutation enabled by the flags.

In a nutshell, these new features add convenience and performance while conversely adding complexity and creating version dependencies on the CUDA driver, the CUDA hardware and the host processors. However, many types of applications can benefit from these new features.

| Type | Discrete GPU Benefit | Integrated GPU Benefit | Con |
|---|---|---|---|
| Portable | ▪ Multiple GPUs can access a single pinned region | ▪ Same as Discrete | ▪ Adds software version dependency |
| Mapped | ▪ Adds transparent, asynchronous I/O capability<br>▪ Extends GPU memory by adding host memory (PCIe bandwidth limited) to GPU address space | ▪ Functions as zero-copy buffer | ▪ Add software version and GPU dependency<br>▪ Must be fully coalesced or 2x-7x performance will result<br>▪ Mapped memory provides up to 2x the PCIe bandwidth at best<br>▪ Current GPUs use 32-bit pointers, so each context can use a maximum of 4GB of memory (on-board DRAM plus mapped CPU memory).<br>▪ Note: the CUDA 2.2 documentation indicates all pinned memory is mapped to GPU address space when `cudaDeviceMapHost` is specified.<br>▪ Accessing GPU-written data on the host may require synchronization |
| WC | ▪ Potential 40% increase in PCIe bandwidth<br>▪ Potential host side performance boost | ▪ Same as Discrete | ▪ Adds software version and GPU dependency<br>▪ Adds host processor dependency (to use efficiently)<br>▪ Unclear if fencing is required |

The following source listing for incrementMappedArrayInPlace.cu is an adapted version of the incrementArrays.cu example from Part 2 to use the new mapped, pinned runtime API.

```
// incrementMappedArrayInPlace.cu
#include <stdio.h>
#include <assert.h>
#include <cuda.h>

// define the problem and block size
#define NUMBER_OF_ARRAY_ELEMENTS 100000
#define N_THREADS_PER_BLOCK 256

void incrementArrayOnHost(float *a, int N)
{
  int i;
  for (i=0; i < N; i++) a[i] = a[i]+1.f;
}

__global__ void incrementArrayOnDevice(float *a, int N)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx < N) a[idx] = a[idx]+1.f;
}

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);
  }
}

int main(void)
{
  float *a_m; // pointer to host memory
  float *a_d; // pointer to mapped device memory
  float *check_h;   // pointer to host memory used to check results
  int i, N = NUMBER_OF_ARRAY_ELEMENTS;
  size_t size = N*sizeof(float);
  cudaDeviceProp deviceProp;

#if CUDART_VERSION < 2020
#error "This CUDART version does not support mapped memory!\n"
#endif

  // Get properties and verify device 0 supports mapped memory
  cudaGetDeviceProperties(&deviceProp, 0);
  checkCUDAError("cudaGetDeviceProperties");

  if(!deviceProp.canMapHostMemory) {
    fprintf(stderr, "Device %d cannot map host memory!\n", 0);
    exit(EXIT_FAILURE);
  }

  // set the device flags for mapping host memory
  cudaSetDeviceFlags(cudaDeviceMapHost);
  checkCUDAError("cudaSetDeviceFlags");

  // allocate mapped arrays
  cudaHostAlloc((void **)&a_m, size, cudaHostAllocMapped);
  checkCUDAError("cudaHostAllocMapped");

  // Get the device pointers to the mapped memory
  cudaHostGetDevicePointer((void **)&a_d, (void *)a_m, 0);
  checkCUDAError("cudaHostGetDevicePointer");

  // initialization of host data
  for (i=0; i<N; i++) a_m[i] = (float)i;

  // do calculation on device:
  // Part 1 of 2. Compute execution configuration
  int blockSize = N_THREADS_PER_BLOCK;
  int nBlocks = N/blockSize + (N%blockSize > 0?1:0);

  // Part 2 of 2. Call incrementArrayOnDevice kernel
  incrementArrayOnDevice <<< nBlocks, blockSize >>> (a_d, N);
  checkCUDAError("incrementArrayOnDevice");

  /* Note the allocation, initialization and call to incrementArrayOnHost
     occurs asynchronously to the GPU */
  check_h = (float *)malloc(size);
  for (i=0; i<N; i++) check_h[i] = (float)i;
  incrementArrayOnHost(check_h, N);

  // Make certain that all threads are idle before proceeding
  cudaThreadSynchronize();
  checkCUDAError("cudaThreadSynchronize");

  // check results
  for (i=0; i<N; i++) assert(check_h[i] == a_m[i]);

  // cleanup
  free(check_h); // free host memory
  cudaFreeHost(a_m); // free mapped memory (and device pointers)
}
```

CUDA 2.2 added the following two device properties to the `cudaDeviceProp` structure that is retrieved by `cudaGetDeviceProperties` so you can determine if a device can support the new mapped memory API (as well as check if the GPU is an integrated graphics processor):

| | |
|---|---|
| `int integrated;` | Nonzero if the device is integrated with the host memory system. This structure member corresponds to the driver API's `CU_DEVICE_ATTRIBUTE_INTEGRATED` query. |
| `int canMapHostMemory;` | Nonzero if the device can map host memory. This structure member corresponds to the driver API's `CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY` query. |

The following code block utilizes a pre-processor check to make certain that a valid version of CUDA is being used to compile the mapped code plus the function `cudaGetDeviceProperties` is called so a runtime check can be made to ensure that the CUDA device supports mapped memory:

```
#if CUDART_VERSION < 2020
#error "This CUDART version does not support mapped memory!\n"
#endif

  // Get properties and verify device 0 supports mapped memory
  cudaGetDeviceProperties(&deviceProp, 0);
  checkCUDAError("cudaGetDeviceProperties");

  if(!deviceProp.canMapHostMemory) {
    fprintf(stderr, "Device %d cannot map host memory!\n", 0);
    exit(EXIT_FAILURE);
  }
```

Host memory mapping is then enabled on the device:

```
  // set the device flags for mapping host memory
  cudaSetDeviceFlags(cudaDeviceMapHost);
  checkCUDAError("cudaSetDeviceFlags");
```

A mapped array, `a_m`, is then allocated on the host. (Note: The memory is mapped at this point but there is no device pointer. Getting the device pointer occurs in the following step.)

```
// allocate host mapped arrays
  cudaHostAlloc((void **)&a_m, size, cudaHostAllocMapped);
  checkCUDAError("cudaHostAllocMapped");
```

Get the device pointer to the mapped memory:

```
  // Get the device pointers to the mapped memory
  cudaHostGetDevicePointer((void **)&a_d, (void *)a_m, 0);
  checkCUDAError("cudaHostGetDevicePointer");
```

Data initialization occurs and the kernel is executed on the GPU. Unlike the original incrementArrays.cu example, no explicit programmer initiated data movement occurs with a `cudaMemcpy`. Note that the data movement and kernel execution occurs asynchronously to the host operations. As a result, the host creation and calculation of the the validation array, `check_h`, occurs while the GPU is simultaneously running the `incrementArrayOnDevice` kernel to update the host array `a_m` through the mapped device memory pointer `a_d`.

Synchronization occurs via the call to `cudaThreadSynchronize` after which the GPU results are validated against the host generated results.

Assuming the results from the host and GPU kernels agree, the program then cleans up after itself. The function `cudaFreeHost` is used to free up the mapped array on the host and pointer on the GPU.

Under Linux, the program can be compiled with the command-line:

```
  nvcc –o incrementMappedArrayInPlace incrementMappedArrayInPlace.cu
```

The performance implications of performing in-place updates to mapped memory are not clear. To ensure the minimum number of PCIe operations occur, it seems prudent to stream data between separate arrays. In other words, use separate arrays where one is dedicated read operations and the other is dedicated to write operations.

## Demonstrating write-combining

The following program, incrementMappedArrayWC.cu, demonstrates the use of separate write-combined, mapped, pinned memory to increment the elements of an array by one. This required changing `incrementArrayOnHost` and `incrementArrayOnDevice` to read from array `a` and write to array `b`. In this way, coherency issues are avoided and streaming performance should be achieved. The `cudaHostAllocWriteCombined` flag was also added to the `cudaHostAlloc` calls. We rely on the CUDA calls to the driver to issue the appropriate fence operation to ensure the writes become globally visible.

```
// incrementMappedArrayWC.cu
#include <stdio.h>
#include <assert.h>
#include <cuda.h>

// define the problem and block size
#define NUMBER_OF_ARRAY_ELEMENTS 100000
#define N_THREADS_PER_BLOCK 256

void incrementArrayOnHost(float *b, float *a, int N)
{
  int i;
  for (i=0; i < N; i++) b[i] = a[i]+1.f;
}

__global__ void incrementArrayOnDevice(float *b, float *a, int N)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx < N) b[idx] = a[idx]+1.f;
}
```

```
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);
  }
}

int main(void)
{
  float *a_m, *b_m; // pointers to mapped host memory
  float *a_d, *b_d; // pointers to mapped device memory
  float *check_h;   // pointer to host memory used to check results
  int i, N = NUMBER_OF_ARRAY_ELEMENTS;
  size_t size = N*sizeof(float);
  cudaDeviceProp deviceProp;

#if CUDART_VERSION < 2020
#error "This CUDART version does not support mapped memory!\n"
#endif

  // Get properties and verify device 0 supports mapped memory
  cudaGetDeviceProperties(&deviceProp, 0);
  checkCUDAError("cudaGetDeviceProperties");

  if(!deviceProp.canMapHostMemory) {
    fprintf(stderr, "Device %d cannot map host memory!\n", 0);
    exit(EXIT_FAILURE);
  }

  // set the device flags for mapping host memory
  cudaSetDeviceFlags(cudaDeviceMapHost);
  checkCUDAError("cudaSetDeviceFlags");

  // allocate host mapped arrays
  int flags = cudaHostAllocMapped|cudaHostAllocWriteCombined;
  cudaHostAlloc((void **)&a_m, size, flags);
  cudaHostAlloc((void **)&b_m, size, flags);
  checkCUDAError("cudaHostAllocMapped");

  // Get the device pointers to memory mapped
  cudaHostGetDevicePointer((void **)&a_d, (void *)a_m, 0);
  cudaHostGetDevicePointer((void **)&b_d, (void *)b_m, 0);
  checkCUDAError("cudaHostGetDevicePointer");

  /* initialization of the mapped data. Since a_m is write-combined,
     it is not guaranteed to be initialized until a fence operation is
     called. In this case that should happen when the kernel is
     invoked on the GPU */
  for (i=0; i<N; i++) a_m[i] = (float)i;

  // do calculation on device:
  // Part 1 of 2. Compute execution configuration
  int blockSize = N_THREADS_PER_BLOCK;
  int nBlocks = N/blockSize + (N%blockSize > 0?1:0);

  // Part 2 of 2. Call incrementArrayOnDevice kernel
  incrementArrayOnDevice <<< nBlocks, blockSize >>> (b_d, a_d, N);
  checkCUDAError("incrementArrayOnDevice");

  // Note the allocation and call to incrementArrayOnHost occurs
  // asynchronously to the GPU
  check_h = (float *)malloc(size);
  incrementArrayOnHost(check_h, a_m,N);

  // Make certain that all threads are idle before proceeding
  cudaThreadSynchronize();
  checkCUDAError("cudaThreadSynchronize");

  // cudaThreadSynchronize() should have caused an sfence
  // to be issued, which will guarantee that all writes are done

  // check results. Note: the updated array is in b_m, not b_d
  for (i=0; i<N; i++) assert(check_h[i] == b_m[i]);

  // cleanup
  free(check_h);

 // free mapped memory (and device pointers)
  cudaFreeHost(a_m);
  cudaFreeHost(b_m);
}
```

## Conclusion

CUDA 2.2 changes the data movement paradigm by providing APIs for mapped, transparent data transfers between the host and GPU(s). These APIs also allow the CUDA programmer to make data sharing between the host and graphics processor(s) more efficient by exploiting asynchronous operation, full-duplex PCIe data transfers, through the use of write combined memory, and by adding the ability for the programmer to share pinned memory with multiple GPUs.

Personally, I have used these APIs as a convenience when porting existing scientific codes onto the GPU because mapped memory allows me to keep the host and device data synchronized while I incrementally move as much of the calculation onto the GPU as possible. This allows me to verify my results after each change to ensure nothing has broken, which can be a real time and frustration saver when working with complex codes with many inter-dependencies. Additionally, I also use these APIs to increase efficiency by exploiting asynchronous host and multiple GPU calculations plus full-duplex PCIe transfers and other nice features of the CUDA 2.2 release.

I also see the new CUDA 2.2 APIs facilitating the development of entirely new classes of applications ranging from operating systems to real-time systems.

One example is the RAID research performed by scientists at the University of Alabama and Sandia National Laboratory that transformed CUDA-enabled GPUs into high-performance RAID accelerators that can calculate Reed-Solomon codes in real-time for high-throughput disk subsystems (see Accelerating Reed-Solomon Coding in RAID Systems with GPUs, by Matthew Curry, Lee Ward, Tony Skjellum, Ron Brightwell). From their abstract, "Performance results show that the GPU can outperform a modern CPU on this problem by an order of magnitude and also confirm that a GPU can be used to support a system with at least three parity disks with no performance penalty".

My guess is we will see a CUDA-enhanced Linux md (multiple device or software RAID) driver sometime in the near future. Imagine the freedom of not being locked into a proprietary RAID controller. If something breaks, just connect your RAID array to another Linux box to access the data. If that computer does not have an NVIDIA GPU then just use the standard Linux software md driver to access the data.

Don't forget that CUDA-enabled devices can accelerate and run multiple applications at the same time. An upcoming article demonstrating how to incorporate graphics and CUDA will exploit that capability. Until then, try running a separate graphics application while running one of your CUDA applications. I think you will be surprised at how well both applications will perform.

## For More Information

- 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.*