TR-2014-09

Unified Memory in CUDA 6:
A Brief Overview and Related Data Access/Transfer Issues

Dan Negrut, Radu Serban, Ang Li, Andrew Seidl

June 27, 2014

Abstract

This document highlights aspects related to the support and use of unified, or managed, memory in CUDA 6. The discussion provides an opportunity to revisit two other CUDA memory transaction topics: zero-copy memory and unified virtual addressing. The latter two can be regarded as intermediary milestones in a process that has led in CUDA 6 to the release of managed memory with three immediate consequences: (*i*) the host/device heterogeneous computing memory model is simplified by eliminating the need for deep copies when accessing structures of arrays in GPU kernels; (*ii*) the programmer has the option to defer the host-device data transfers to the runtime; and (*iii*) CUDA code becomes cleaner and simpler to develop/debug/maintain.

This discussion does not attempt to exhaustively cover the managed memory topic. In an 80/20 we attempted to highlight the points that will address 80% of the CUDA developers' needs/questions while only touching 20% of the technical aspects associated with the use/support of unified memory in CUDA 6. An account of unified memory that does not delve into the zero-copy and unified virtual memory topics but instead concentrates exclusively on its key benefits, including elimination of explicit deep copies and implications to C++ code development, is already available online[1,2].

The present document has a companion PowerPoint file[3] that summarizes the main points made herein and could be useful for instructional purposes.

[1] http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6
[2] http://devblogs.nvidia.com/parallelforall/cudacasts-episode-18-cuda-6-0-unified-memory
[3] http://sbel.wisc.edu/documents/TR-2014-09.pptx

Contents

1. GPU Computing: The Physics-Based Simulation Angle

In 2007, soon after the release of CUDA 1.0, it became apparent that many classes of applications in Computer Aided Engineering could benefit from the SIMD computing paradigm that GPUs support. In our case, we were interested in how large collections of components move around and interact with each other in dynamic systems; i.e., systems whose configuration changes in time under the influence of external and internal forces. The examples in **Figure 1** illustrate two problems that have to do with the motion of granular systems. The goal here is to figure out, for systems with millions of elements, e.g., grains in a granular material, how these elements mutually interact and how they move collectively. First of all, this calls for performing collision detection to figure out who is contacting whom. Next, drawing on some rather involved math, we have to compute the interaction forces between any two bodies in contact. For the ball floating on the wave of granular material there are about four million mutual contacts. This scenario calls for the solution of an optimization problem in which we have to minimize a cost function that depends on four million variables. Approximately one million such optimization problems need to be solved to understand how this wave moves for 10 seconds. GPU computing helps provide a fast solution to each of the one million optimization problems. We cannot solve these one million problems in parallel since there is a principle of causality at work that makes the second optimization problem depend on the first, the third on the second, etc.

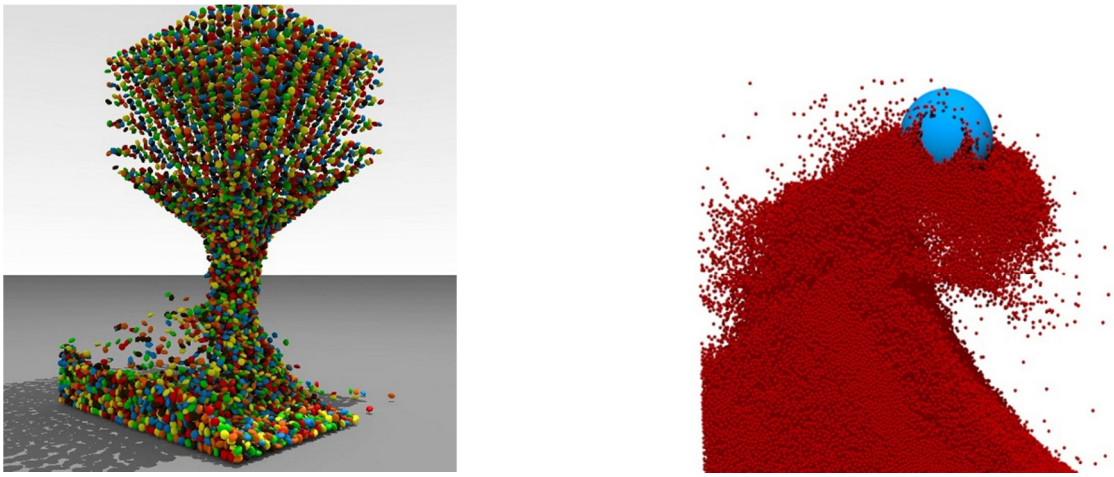

Figure 1. Example of physics-based modeling and simulation enabled by GPU computing. Left image represents a mixing process simulation. Right image captures the dynamics of a light body floating on more than one million rigid bodies whose collective motion resembles the propagation of a wave.

2. Life before Unified Memory

Three CUDA features shaped the memory transaction landscape prior to the introduction of unified memory support in release 6. They are `cudaMemcpy`, zero-copy memory, and support for unified virtual addressing. These CUDA features have been used in implementing a simulation engine called Chrono, which was used to generate the images in **Figure 1**.

2.1. cudaMemcpy

A staple of CUDA programming since version 1.0, `cudaMemcpy` has facilitated the back-and-forth movement of data between host and device. While relatively awkward in syntax, it enabled a simple

three-step approach for data processing on the GPU: data was moved onto the device, it was processed by the device, and results were moved back to the host. This three-step process assumes that memory has been allocated on the device to store the data transferred from the host. The awkwardness stems from the `enum cudaMemcpyKind` argument `kind`:

```
cudaError_t cudaMemcpy(void * dst, const void * src, size_t count, enum
cudaMemcpyKind kind);
```

The last argument defines the nature of the data transfer: from host to device (`cudaMemcpyHostToDevice`), device to host (`cudaMemcpyDeviceToHost`), device to device (`cudaMemcpyDeviceToDevice`), or host to host (`cudaMemcpyHostToHost`). It was the programmer's burden to ensure that if, for instance, the destination pointer `dst` and the source pointer `src` pointed to device and host memory, respectively, then the last argument was `cudaMemcpyHostToDevice`. The code would most likely segfault if, for instance, the `dst` and `src` pair above is used with a `cudaMemcpyDeviceToHost` flag. The crash is due to the fact that the host and device each dealt with their own virtual memory space. Using a host address that was valid in the context of the host virtual memory led to an attempted device memory access to a location that might not even exist in the virtual memory space of the device.

Of the four `kind` flags introduced above, the ones that typically saw the highest usage were `cudaMemcpyHostToDevice` and `cudaMemcpyDeviceToHost`. The bandwidths that one would have seen in relation to the associated transfers depended on the PCIe generation. Practically, for PCIe gen 1, 2, and 3, one could count on 3, 6, and 12 GB/s, respectively. The bandwidth of the PCIe transfer could be increased by pinning the host memory allocation used in the transactions to prevent the OS from paging it in and out of memory. Pinned memory on the host side had two additional side benefits: (*i*) it allowed for asynchronous data movement between the host and device, and (*ii*) it enabled the GPU to use both of its copy engines, one for moving data from the host to the device, the other for moving data from device to the host. However, relying on a large amount of pinned memory ran the risk of slowing down the entire system as a consequence of curtailing the freedom of the operating system to page out the pinned pages; at the same time, the very pinning of host memory took a substantial amount of time. For instance, pinning 5 GB could require on the order of one second (order of magnitude) thus forcing the programmer to choose between: (*i*) spending the time pinning and hoping to recoup the time by faster PCIe transfers while running the risk of slowing down the entire system; or (*ii*) skipping the pinning and hoping that the PCIe data movement was negligible relative to the amount of time spent on the host processing this data.

2.2. Zero-Copy

The prototype of the runtime call for page-locked memory allocations on the host is

```
cudaError_t cudaHostAlloc(void** pHost, size_t size, unsigned int flags);
```

The parameter `flags`, with a value that is some combination of `cudaHostAllocDefault`, `cudaHostAllocPortable`, `cudaHostAllocMapped`, or `cudaHostAllocWriteCombined`, qualifies the nature of the pinned memory; i.e., what a programmer can subsequently do with it. Of interest here is `cudaHostAllocMapped`, which ensures that the memory allocated on the host is mapped into the CUDA address space. Thereafter, a pointer to this memory can be used within a kernel call or other device function to access host memory directly. This is called zero-copy GPU-CPU interaction, hence the name "zero-copy memory". Note that data is still moved through the PCIe bus. However, unlike with

cudaMemcpy, this data movement is not coordinated by the user and it does not happen in one large transaction. The mapped memory is accessed directly by GPU threads and only data that are read/written are moved upon access over the PCIe bus.

The zero-copy avenue is recommended when two conditions are met. First, the memory is accessed in a coalesced fashion, as recommended for any global memory access in CUDA. Second, if multiple memory accesses are made, they should display a high degree of spatial and temporal coherence. If any of these conditions is not met, multiple accesses to zero-copy memory will lead to multiple low bandwidth PCIe excursions. The knee jerk reaction might be to use a cudaMemcpy call as soon as the pinned data is used more than once. However, this might not be the winning strategy. Recall that today's cards have a somewhat decent amount of cache per thread and one might hit the cache if memory accesses display a certain degree of spatial and/or temporal coherence. A rule of thumb cannot be provided here since the decision depends on the specific code being executed and the underlying hardware. Complicating the decision process even further is the observation that accesses of the host memory, albeit displaying high latency and low bandwidth when not cached, can benefit from the ability of the warp scheduler to hide memory latency with useful execution of warps that are ready to go.

2.3. Unified Virtual Addressing (UVA)

Following the introduction of zero-copy memory in CUDA 2.0, CUDA 4.0 brought the host and device memory spaces one step closer by providing, on Fermi and later architectures full integration of the corresponding virtual spaces. Essentially, no distinction was made between a host pointer and a device pointer. The CUDA runtime could identify where the data was stored based on the value of the pointer. In a unified virtual address space setup, the runtime manipulates the pointer and allocation mappings used in device code (through cudaMalloc) as well as pointers and allocation mappings used in host code (through cudaHostAlloc) inside a single unified space. An immediate consequence is that the kind flag in the cudaMemcpy argument list becomes obsolete and is replaced by a generic cudaMemcpyDefault. The true benefit of the UVA mechanism becomes apparent when transferring data between two devices[4,5]. Indeed, if zero-copy provided a degree of convenience in relation to data access operations, UVA took this one step further by improving on data transfer tasks. Owing to the fact that all devices plus the host shared the same virtual address space, a data access or data transfer involving two devices was simplified in two respects: (*i*) straight use on device A of a pointer to access memory on device B; and (*ii*) no need for staging the data transfer through host memory.

3. Unified Memory in CUDA 6

3.1. Generalities

The zero-copy and unified virtual addressing features of CUDA go back to the use of a pair of function calls. First, memory is allocated in page-locked fashion on the host by means of cudaHostAlloc. Second, for data transfers, cudaMemcpy moves data without staging the process through the host. These two functions open the door to several data access and data transfer options: accessing data on the host from a device function or kernel, accessing data on one GPU from a different GPU, and data movement between GPUs through peer-to-peer transfers.

[4] See http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_GPUDirect_uva.pdf
[5] See http://www.nvidia.com/docs/IO/116711/sc11-multi-gpu.pdf

CUDA 6 adds one extra layer of convenience vis-à-vis the CPU/GPU memory management task with the introduction of unified, or managed, memory (UM). Data is now stored and migrated in a user-transparent fashion that allows, under circumstances spelled out shortly, data access/transfer at latencies and bandwidths of the host and of the device, for host-side and device-side memory operations, respectively. Moreover, the use of the `cudaHostAlloc` and `cudaMemcpy` combo is no longer a requirement, which allows for a cleaner and more natural programming style.

The centerpiece of the UM concept and associated programming style is the CUDA runtime call `cudaMallocManaged` which allocates memory on the device. As far as the host is concerned, no distinction is made in terms of accessing memory allocated with `cudaMallocManaged` or through a `malloc` call. However, although there is no difference in semantics, the programmer needs to be aware that different processors might experience different access times owing to different latencies and bandwidths. Incidentally, a processor is regarded as an independent execution unit with a dedicated memory management unit (MMU); i.e., any GPU or CPU.

Figure 2. Prior to CUDA 6, there was a clear physical and logical separation between the host and device memories. UVA blurred the logical separation while UM went one step further. The elimination of the physical separation has already taken place on NVIDIA's Tegra K1 SoC architecture.

The use of unified memory is best seen through a simple example in which a kernel is called to increment the value of each entry in an array of integers. We provide first a pre-CUDA 6 implementation.

```
#include <ostream>
#include <cmath>

const int ARRAY_SIZE = 1000;

__global__ void increment(double* aArray, double val, unsigned int sz) {
    unsigned int indx = blockIdx.x * blockDim.x + threadIdx.x;
    if (indx < sz)
        aArray[indx] += val;
}

int main() {
    double* hA;
    double* dA;
    hA = (double *)malloc(ARRAY_SIZE * sizeof(double));
```

```
    cudaMalloc(&dA, ARRAY_SIZE * sizeof(double));
    for (int i = 0; i < ARRAY_SIZE; i++)
        hA[i] = 1.*i;
    double inc_val = 2.0;
    cudaMemcpy(dA, hA, sizeof(double) * ARRAY_SIZE, cudaMemcpyHostToDevice);
    increment<<<2, 512>>>(dA, inc_val, ARRAY_SIZE);
    cudaMemcpy(hA, dA, sizeof(double) * ARRAY_SIZE, cudaMemcpyDeviceToHost);
    double error = 0.;
    for (int i = 0; i < ARRAY_SIZE; i++)
        error += std::fabs(hA[i] - (i + inc_val));

    std::cout << "Test: " << (error < 1.E-9 ? "Passed" : "Failed") << std::endl;
    cudaFree(dA);
    free(hA);
    return 0;
}
```

The managed memory version is shorter as it makes no reference to `cudaMemcpy` and there is no need for the host `malloc`/`free` calls. In other words, the implementation does not require explicit shadowing of any chunk of device memory by a corresponding chunk of host memory. Allocating unified memory suffices as the runtime, upon a `cudaDeviceSynchronize` call, will make it available in a coherent fashion to the host or device.

```
#include <ostream>
#include <cmath>

const int ARRAY_SIZE = 1000;

__global__ void increment(double* aArray, double val, unsigned int sz) {
    unsigned int indx = blockIdx.x * blockDim.x + threadIdx.x;
    if (indx < sz)
        aArray[indx] += val;
}

int main() {
    double* mA;
    cudaMallocManaged(&mA, ARRAY_SIZE * sizeof(double));
    for (int i = 0; i < ARRAY_SIZE; i++)
        mA[i] = 1.*i;
    double inc_val = 2.0;
    increment<<<2, 512>>>(mA, inc_val, ARRAY_SIZE);
    cudaDeviceSynchronize();
    double error = 0.;
    for (int i = 0; i < ARRAY_SIZE; i++)
        error += std::fabs(mA[i] - (i + inc_val));

    std::cout << "Test: " << (error < 1.E-9 ? "Passed" : "Failed") << std::endl;
    cudaFree(mA);
    return 0;
}
```

It is instructive to compare zero-copy and unified memory. For the former, the memory is allocated in page-locked fashion on the host. A device thread has to reach out to get the data. No guarantee of coherence is provided as, for instance, the host could change the content of the pinned memory while the device reads its content. For UM, the memory is allocated on the device and transparently made available where needed. Specifically, upon a call to

```
cudaError_t cudaMallocManaged(void** devPtr, size_t size, unsigned int flag);
```

the user has, in `devPtr,` a pointer to an address of a chunk of device memory. This address can be equally well manipulated on the device and the host (although, as illustrated below, not simultaneously). Note that `cudaMallocManaged` and `cudaMalloc` are semantically identical; in fact, the former can be used anywhere the latter is used.

UM enables a "single-pointer-to-data" memory model. For instance, the same pointer can be used on the host in a `memcpy` operation to copy a set of integers to an array `mA`, and then on the device to alter, just like in the code snippet above, the value of each entry in `mA`. The data in `mA` will be coherent as long as the host does not touch entries in `mA` when the GPU executes a kernel. The host can safely operate with/on `mA` only after a `cudaDeviceSynchronize` call. Failure to obey this rule will lead to a segfault, as illustrated in the following example lifted from the CUDA Programming guide[6].

```
__device__ __managed__ int x, y = 2;
__global__ void kernel() {
    x = 10;
}
int main() {
    kernel<<<1, 1>>>();
    y = 20; // ERROR: CPU access concurrent with GPU
    cudaDeviceSynchronize();
    return 0;
}
```

The segfault goes back to the attempt of the CPU to reference a managed memory variable, `y` in this case, while the device is executing a kernel. The example below, lifted from the same source, illustrates the typical strategy for handling data stored in UM by highlighting the role played by the `cudaDeviceSynchronize` call. Note that any function that logically guarantees that the GPU finished execution, such as `cudaStreamSynchronize`, `cudaMemcpy`, `cudaMemset`, can play the role played by `cudaDeviceSynchronize` below.

```
__device__ __managed__ int x, y = 2;
__global__ void kernel() {
    x = 10;
}
int main() {
    kernel<<<1, 1>>>();
    cudaDeviceSynchronize();
    y = 20; // GPU is idle so access is OK
    return 0;
}
```

[6] http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

The code samples above illustrate a second CUDA feature added to support UM: the `__managed__` type qualifier which allocates, at compile time, a quantity stored in managed memory. Besides the `cudaMallocManaged` call and `__managed__` qualifier, the third and last feature introduced in CUDA 6 to support UM is `cudaStreamAttachMemAsync`. Its role is to choreograph the interplay between managed memory and concurrency in multi-threaded CPU applications. The backdrop for its use is provided by the observation that pages from managed allocations touched by a host thread are migrated back to GPU before any kernel launch. As such, no overlap of kernel execution and data transfer can take place in that CUDA stream. Overlap is still possible but it calls for the use of multiple streams: while one kernel executes in one stream, a different stream can engage in a data transfer process. This strategy is possible since the managed allocation process is specific to a stream and as such it allows concurrency to control which allocations are synchronized on which specific kernel launches.

As mentioned before, a unified memory allocation physically takes place in device memory on the device that happens to be active at the time of the allocation. When this memory is operated upon by the CPU, the migration to host happens at page level resolution; i.e., typically 4 KB. The runtime tracks dirty pages and detects page faults. It transparently moves over the PCIe bus only the dirty pages. Pages touched by the CPU (GPU) are moved back to the device (host) when needed. Coherence points are kernel launches and device/stream synchronizations. For now, there is no oversubscription of the device memory. If several devices are available, the largest amount of managed memory that can be allocated is the smallest of the available device memories.

3.2. A short example: Unified Memory and thrust

Thrust[7] is a CUDA C++ template library of parallel algorithms and data structures. With an interface similar to the C++ Standard Template Library (STL), Thrust provides a high-level interface to high-performance GPU-accelerated implementation of common algorithms, such as sort, scan, transform, and reductions.

Scientific and engineering CUDA codes, such as the one briefly described at the beginning of this article, often involve a combination of custom kernels and calls to Thrust algorithms. To allow interoperability with the entire CUDA ecosystem of libraries, tools, and user kernels, Thrust provides a simple API for (*i*) wrapping CUDA device pointers so that they can be passed to a Thrust algorithm (`thrust::device_pointer_cast`) and (*ii*) extracting the raw device pointer from a Thrust `device_ptr` or `device_vector` (`thrust::raw_pointer_cast`) so that it can be used in custom kernels.

By default, Thrust relies on implicit algorithm dispatch, using tags associated with its vector containers. For example, the system tag for the iterators of `thrust::device_vector` is `thrust::cuda::tag` and therefore algorithms dispatched on such iterators will be parallelized in the CUDA system. This will not work with memory allocated through `cudaMallocManaged`. To prevent the need to introduce new vectors or to wrap existing managed memory simply to use a parallel algorithm, Thrust algorithms can be invoked with an explicitly specified execution policy. This approach is illustrated in the example below, where the array `mA` could also be directly passed, as is, to a host function or a CUDA kernel.

[7] N. Bell and J. Hoberock, "Thrust: A Productivity-Oriented Library for CUDA," GPU Computing Gems Jade Edition, pp. 359, 2011.

```
#include <ostream>
#include <cmath>
#include <thrust/reduce.h>
#include <thrust/system/cuda/execution_policy.h>
#include <thrust/system/omp/execution_policy.h>

const int ARRAY_SIZE = 1000;

int main(int argc, char **argv) {
    double* mA;
    cudaMallocManaged(&mA, ARRAY_SIZE * sizeof(double));
    thrust::sequence(mA, mA + ARRAY_SIZE, 1);
    double maximumGPU = thrust::reduce(thrust::cuda::par, mA, mA + ARRAY_SIZE, 0.0,
                                       thrust::maximum<double>());
    cudaDeviceSynchronize();
    double maximumCPU = thrust::reduce(thrust::omp::par, mA, mA + ARRAY_SIZE, 0.0,
                                         thrust::maximum<double>());
    std::cout << "GPU reduce: “ << (std::fabs(maximumGPU - ARRAY_SIZE) < 1e-10 ? "Passed" : "Failed");
    std::cout << "CPU reduce: “ << (std::fabs(maximumCPU - ARRAY_SIZE) < 1e-10 ? "Passed" : "Failed");
    cudaFree(mA);
    return 0;
}
```

With this model, the programmer only specifies the Thrust backend of interest (how the algorithm should be parallelized), without being concerned about the system being able to dereference the iterators provided to the algorithm (where the data “lives”). This is consistent with the simpler programming and memory management enabled by UM.

4. Unified Memory at Work in Scientific Computing

Solving sparse systems of linear equations **Ax** = **b**, where **A** is either a symmetric or a non-symmetric nonsingular matrix, is one of the most common tasks in Scientific Computing. For instance, numerical solution approaches for the practical problems discussed at the beginning of this document often times require the solution of large sparse linear systems. We briefly introduce below SPIKE::GPU[8], a library for the GPU solution of mid-size sparse linear systems, and use it to assess the performance of managed memory in CUDA 6.

SPIKE::GPU is an open source sparse linear solver for systems of medium size; i.e., up to approximately 0.5 million unknowns. The solution strategy has three steps. In step 1, a sparse matrix is reordered; i.e., entries in **A** are moved around by shifting the order of the rows and columns in the matrix, to accomplish two things:

Reordering 1) Move the large entries (in absolute value) of the **A** matrix to the diagonal to render the reordered matrix as close as possible to a diagonally dominant matrix, and

Reordering 2) Reduce the sparse matrix to a dense band matrix (bandwidth reduction), or equivalently, group all the matrix entries close to the matrix diagonal.

In the second step of the solution sequence, the reordered matrix is partitioned into submatrices $\mathbf{A}_1$ through $\mathbf{A}_p$ that are LU factorized independently. Some coupling exists between these submatrices, accounted for through the parallel computation of some bridging terms (called in the literature “spikes”). In the third and final step of the solution strategy, given that several approximations are made during the

[8] http://spikegpu.sbel.org/

factorization of the matrix of **A**, SPIKE::GPU relies on an iterative solver that is preconditioned by the approximate factorization obtained at the end of step 2. The iterative solver considered in step 3 belongs to the family of Krylov sub-space methods.

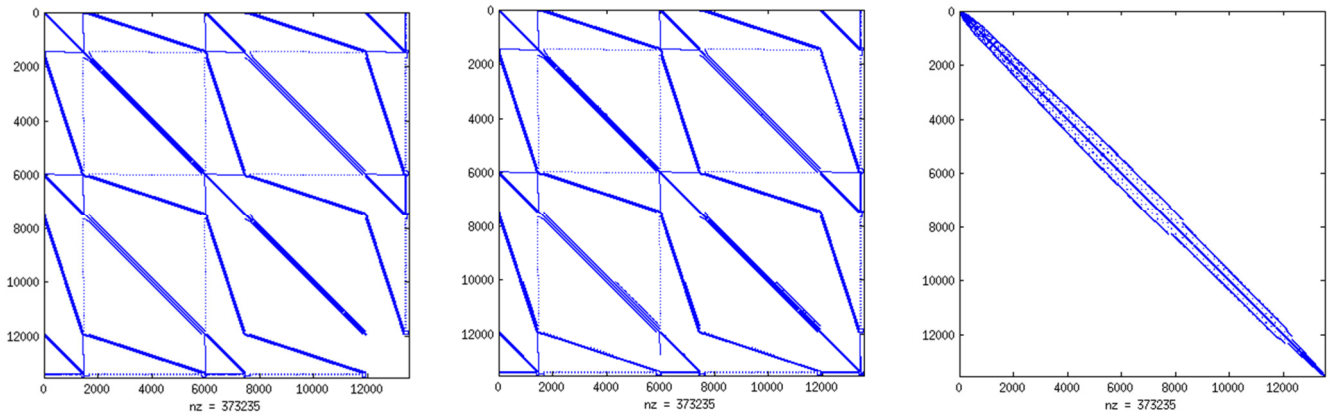

Figure 3. Sparsity pattern of the original sparse matrix (left), after applying four-stage algorithm (center), and after applying bandwidth reordering (right). The matrix illustrated is "Garon2" from the University of Florida collection of sparse matrices. The dimension of the matrix is 13535, the number of nonzero entries is 373235, and the half bandwidth is 13531. After the first reordering, the half bandwidth doesn't change yet note that nonzero entries populate the entire diagonal of the matrix due to migration of "heavy" entries to the diagonal. After the band-reduction reordering, the half-bandwidth becomes 585 and from there on the matrix is considered dense within the band.

The performance analysis for UM is done in conjunction with Reordering 1) of step 1 of the algorithm which seeks to reorder the matrix **A** such that its "heavy" entries are moved to the diagonal. The benefit of this sub-step is twofold: it decreases the probability of encountering a zero pivot during the factorization of the diagonal blocks $\mathbf{A}_1$ through $\mathbf{A}_p$, and it increases the quality of the approximations to the coupling off-diagonal blocks (the "spikes"). Reordering 1) in itself has four stages:

Stage 1) form bipartite graph reflecting sparsity pattern of the matrix **A** – easy to do on the GPU
Stage 2) find a partial reordering – currently relatively hard to do on the GPU
Stage 3) refine partial reordering – currently very hard to do on the GPU
Stage 4) extract permutation from the bipartite graph – easy to do on the GPU

All data required to perform matrix reordering through these four stages were stored in managed memory. This strategy, in which the GPU accesses data stored in managed memory during stage 1 and 4 while the CPU accesses data stored in managed memory during stage 2 and 3, simplifies the structure of the code.

The goal herein is to assess the performance impact of adopting UM in this strategy. To this end, more than 120 matrices have been analyzed as far as the Reordering 1) is concerned. In the table below we ranked the matrices according to the speedup of the algorithm when operating on them. The times reported are in milliseconds, with the speedup computed by dividing the time required by the non-UM implementation by the time required by the UM implementation. In addition to the name of the Florida sparse collection matrix, the table reports the dimension of the matrix **A**, the number of its nonzero elements, and the times required by the original and UM implementations. Since showing 120 rows would have required too much space, we only listed every tenth matrix. The bottom line is this: in our experience, using UM and letting the runtime take care of business never resulted in more than a 25%

slowdown over hand-tuned code. In fact, based on the 120 matrix sample, almost half of the tests ran faster with the UM implementation.

Name	Dim.	NNZ	Time Orig.	Time UM	Speedup
poisson3Db	85623	2374949	120.293	86.068	1.397651
pdb1HYS	36417	4344765	153.276	128.269	1.194957
inline_1	503712	36816342	1299.3	1129.12	1.150719
qa8fk	66127	1660579	62.088	55.216	1.124457
finan512	74752	596992	29.526	29.155	1.012725
lhr71	70304	1528092	726.646	762.697	0.952732
g7jac140	41490	565956	701.929	761.265	0.922056
ASIC_100k	99340	954163	51.512	56.924	0.904926
gearbox	153746	9080404	1257.32	1424.52	0.882627
rma10	46835	2374001	211.592	252.221	0.838915
bmw3_2	227362	11288630	741.092	911.724	0.812847
stomach	213360	3021648	174.105	226.585	0.768387

5. Concluding Remarks

The introduction of unified memory in CUDA 6 represents a milestone in NVIDIA's effort to simplify the programming task and improve the performance of GPU computing. Support for managed memory has not come from the left field. Rather, it followed naturally two previous CUDA memory-related milestones: support for the zero-copy mechanism and the unified virtual addressing framework.

Based on our experience gained in the context of solving sparse linear systems and carrying out physics-based simulation, UM did not lead to any remarkable performance improvement. Yet it didn't hurt performance either. This is hardly surprising – efficiency gains will have to wait for the physical integration of the host and device memories. For the time being, unified memory buys ease of programming particularly when one places a premium on object-oriented programming with sharing of objects between host and device code. While hand-tuned code will likely be somewhat faster in certain classes of applications, in NVIDIA's playbook the support for unified memory is most likely prefacing the physical integration of the host and device memories. Presently available only for the mobile device market, see the Tegra K1 SoC architecture, Project Denver is slated to bring to the PC, server, and supercomputer markets a dual-core Tegra chip in the second half of 2014.

The wish list of UM-related things that would be nice to have but are currently missing includes: (*i*) the ability to allocate more memory than the physically available on the GPU; i.e., oversubscription; (*ii*) memory prefetching; and (*iii*) programmer control over the memory migration grain.