ASPECTS RELATED TO DATA ACCESS AND TRANSFER IN CUDA

Dan Negrut, Radu Serban, Ang Li, Andrew Seidl
Simulation-Based Engineering Lab
University of Wisconsin-Madison

Summary / Objective

- Premise: Managing and optimizing host-device data transfers has been challenging
- Key point: Unified Memory (UM) support in CUDA 6 simplifies the programmer's job
- The presentation's two goals:
  - Briefly review history of CUDA host/device memory management
  - Explain how UM makes host/device memory management easier and more efficient

cudaMemcpy

- A staple of CUDA, available in release 1.0

- Setup was simple: one CPU thread dealt with one GPU
  - The drill :
    - Data transferred from host memory into device memory with cudaMemcpy
    - Data was processed on the device by invoking a kernel
    - Results transferred from device memory into host memory with cudaMemcpy

- Memory allocated on the host with malloc
- Memory allocated on the device using the CUDA runtime function cudaMalloc
- The bottleneck: data movement over the PCI-E pipe

The PCI-E Pipe, Putting Things in Perspective

- PCI-E
  - V1: 3 GB/s (per direction)
  - V2: 6 GB/s
  - V3 (today): 12 GB/s

- Bandwidths above pretty small, see for instance
  - Host memory bus (25 – 51.2 GB/s per socket)
  - GMEM bandwidth 100 – 200 GB/s

cudaHostAlloc: A friend, with its pluses and minuses

- Host/Device data transfer speeds could be improved if host memory was not pageable
  - Rather than allocating with malloc, host memory was allocated using CUDA's cudaHostAlloc()
  - No magic on the hardware side, data still moves back-and-forth through same PCI-E pipe

- cudaHostAlloc cons
  - cudaHostAlloc-ing large amounts of memory can negatively impact overall system performance
    - Why? It reduces the amount of system memory available for paging
    - How much is too much? Not clear, dependent on the system and the applications running on the machine
  - cudaHostAlloc is slow - ballpark 5 GB/s
    - Allocating 5 GB of memory is timewise comparable to moving that much memory over the PCI-E bus

Key Benefits, cudaHostAlloc-ing Memory

- Three benefits to replacing host malloc call with CUDA cudaHostAlloc call

  1. Enables faster device/host back-and-forth transfers

  2. Enables the use of asynchronous memory transfer and kernel execution
     - Draws on the concept of CUDA stream, a topic not covered here

  3. Enables the mapping of the pinned memory into the memory space of the device
     - Device now capable to access data on host while executing a kernel or other device function

- Focus next is on 3 above

Zero-Copy (Z-C) GPU-CPU Interaction

- Last argument ("flag") controls the magic:

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

- "flag" values: cudaHostAllocPortable, cudaHostAllocWriteCombined, etc.
- The "flag" of most interest is "cudaHostAllocMapped"
  - Maps the memory allocated on the host in the memory space of the device for direct access
- What's gained:
  - The ability to access a piece of data from pinned and mapped host memory by a thread running on the GPU without a CUDA runtime copy call to explicitly move data onto the GPU
    - This is called zero-copy GPU-CPU interaction, from where the name "zero-copy memory"
    - Note that data is still moved through the PCI-E pipe, but it's done in a transparent fashion

Z-C, Further Comments

- More on the "flag" argument, which can take four values:
  - Use cudaHostAllocDefault argument for getting plain vanilla pinned host memory (call becomes identical in this case to cudaMallocHost call)
  - Use cudaHostAllocMapped to pick up the Z-C functionality
  - See documentation for cudaHostAllocWriteCombined the cudaHostAllocPortable
    - These two flags provide additional tweaks, irrelevant here

- The focus **should not be** on cudaHostAlloc() and the "flag"
  - This function call is only a means to an end

- Focus **should be** on the fact that a device thread can directly access host memory

From Z-C to UVA: CUDA 2.2 to CUDA 4.0

- Z-C enabled access of data on the host from the device required one additional runtime call to cudaHostGetDevicePointer()
  - cudaHostGetDevicePointer(): given a pointer to pinned host memory produces a new pointer that can be invoked within the kernel to access data stored on the host

- The need for the cudaHostGetDevicePointer() call eliminated in CUDA 4.0 with the introduction of the Unified Virtual Addressing (UVA) mechanism

Unified Virtual Addressing: CUDA 4.0

- CUDA runtime can identify where the data is stored based on the value of the pointer
  - Possible since one address space was used for all CPU and GPU memory

- 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

UVA - Consequences

- There is no need to deal with cudaMemcpyHostToHost, cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, and cudaMemcpyDeviceToDevice scenarios
  - Simply use the generic cudaMemcpyDefault flag

- Technicalities regarding the need to call cudaGetDeviceProperties() for all participating devices (to check cudaDeviceProp::unifiedAddressing flag) to figure out whether they're game for UVA are skipped

- What this buys us: ability to do, for instance, inter-device copy that does not rely on the host for staging data movement:
  - cudaMemcpy(gpuDst_memPntr, gpuSrc_memPntr, byteSize, cudaMemcpyDefault)

UVA – Showcasing Its Versatility…

- Set of commands below can be issued by one host thread to multiple devices
  - No need to use anything beyond cudaMemcpyDefault

```
cudaMemcpy(gpu1Dst_memPntr, host_memPntr, byteSize1, cudaMemcpyDefault)
cudaMemcpy(gpu2Dst_memPntr, host_memPntr, byteSize2, cudaMemcpyDefault)
cudaMemcpy(host_memPntr, gpu1Dst_memPntr, byteSize1, cudaMemcpyDefault)
cudaMemcpy(host_memPntr, gpu2Dst_memPntr, byteSize2, cudaMemcpyDefault)
```

- UVA support is the enabler for the peer-to-peer (P2P), inter-GPU, data transfer
  - P2P not topic of discussion here
  - UVA is the underpinning technology for P2P

UVA is a Step Forward Relative to Z-C

- Z-C Key Accomplishment: use pointer within device function access host data
  - Z-C focused on a data access issue relevant in the context of functions executed on the device

- UVA had a data access component but also a data transfer component:
  - Data access: A GPU could access data on a different GPU, a novelty back in CUDA 4.0
  - Data transfer: copy data in between GPUs
    - cudaMemcpy is the main character in this play, data transfer initiated on the host side

Zero-Copy, UVA, and How UM Fits In

- Both for Z-C and UVA the memory was allocated on the device w/ cudaMalloc and on the host with cudaHostAlloc

- The magic ensued upon the cudaHostAlloc/cudaMalloc duo

- Examples of things that can be done:
  - Data on host accessed on the device
  - Data transferred effectively in between devices without intermediate staging on the host
  - Data stored by one GPU accessed directly by a different GPU
  - Etc.

Zero-Copy, UVA, and How UM Fits In

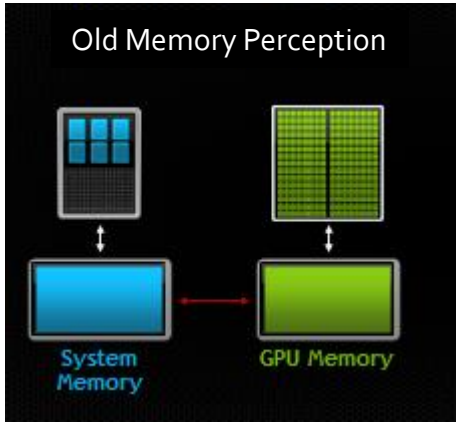
- Unified Memory (UM) eliminates the need to call the cudaMalloc/cudaHostAlloc duo
  - It takes a different perspective on handling memory in the GPU/CPU interplay

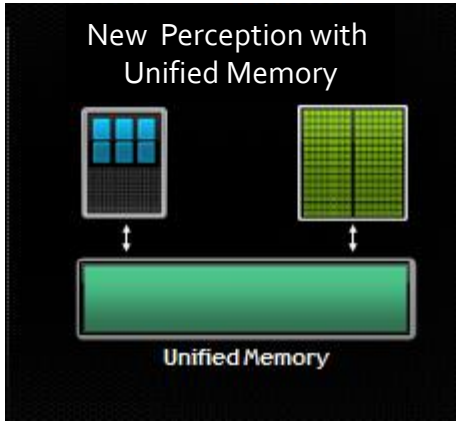
- Note that in theory one could get by using Z-C and only calling cudaHostAlloc once. This is not recommended when having repeated accesses by device to host-side memory
  - Each device request that ends up accessing the host-side memory incurs high latency and low bandwidth (relative to the latency and bandwidth of an access to device global memory)

- This is the backdrop against which the role of UM is justified
  - Data is stored and migrated in a user-transparent fashion
    - To the extent possible, the data is right where it's needed thus enabling fast access

Unified Memory (UM)

- One memory allocation call takes care of memory setup at both ends; i.e., device and host
  - The main actor: the CUDA runtime function cudaMallocManaged()
- New way of perceiving the memory interplay in GPGPU computing
  - No distinction is made between memory on the host and memory on the device
  - It's just memory, albeit with different access times when accessed by different processors





[NVIDIA]→

Unified Memory (UM) – Semantics Issues [clarifications of terms used on previous slide]

- "processor" (from NVIDIA documentation): any independent execution unit with a dedicated memory management unit (MMU)
  - Includes both CPUs and GPUs of any type and architecture

- "different access time": time is higher when, for instance, the host accesses for the first time data stored on the device.
  - Subsequent accesses to the same data take place at the bandwidth and latency of accessing host memory
    - This is why access time is different and lower
    - Original access time higher due to migration of data from device to host
  - NOTE: same remarks apply to accesses from the device

[NVIDIA]→

Now and Then: UM, no-UM

```cpp
#include <iostream>
#include "math.h"

const int ARRAY_SIZE = 1000;
using namespace std;

__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(int argc, char **argv) {
    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 += fabs(mA[i] - (i + inc_val));

    cout << "Test: " << (error < 1.E-9 ? "Passed" : "Failed") << endl;

    cudaFree(mA);
    return 0;
}
```

```cpp
#include <iostream>
#include "math.h"

const int ARRAY_SIZE = 1000;
using namespace std;

__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(int argc, char **argv) {
    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 += fabs(hA[i] - (i + inc_val));

    cout << "Test: " << (error < 1.E-9 ? "Passed" : "Failed") << endl;

    cudaFree(dA);
    free(hA);
    return 0;
}
```

UM vs. Z-C

- Recall that with Z-C, data is always on the host in pinned CPU system memory
  - The device reaches out to it

- UM: data stored on the device but made available where needed
  - Data access and locality managed by underlying system, handling transparent to the user
  - UM provides "single-pointer-to-data" model

- Support for UM called for only *three* additions to CUDA:
  - cudaMallocManaged, __managed__ , cudaStreamAttachMemAsync()

Technicalities…

- cudaError_t cudaMallocManaged (void** devPtr, size_t size, unsigned int flag)
  - Returns pointer accessible from both Host and Device
  - Drop-in replacement for cudaMalloc() – they are semantically similar
  - Allocates managed memory on the device
    - First two arguments have the expected meaning
  - "flag" controls the default stream association for this allocation
    - cudaMemAttachGlobal - memory is accessible from any stream on any device
    - cudaMemAttachHost – memory on this device accessible by host only
  - Free memory with the same cudaFree()

- __managed__
  - Global/file-scope variable annotation combines with __device__
  - Declares global-scope migrateable device variable
  - Symbol accessible from both GPU and CPU code

- cudaStreamAttachMemAsync()
  - Manages concurrency in multi-threaded CPU applications

UM, Quick Points

- In the current implementation, managed memory is allocated on the device that happens to be active at the time of the allocation
- Managed memory is interoperable and interchangeable with device-specific allocations, such as those created using the cudaMalloc() routine
- All CUDA operations that are valid on device memory are also valid on managed memory

Example: UM and thrust

```
#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::endl;
  std::cout << "CPU reduce: “ << (std::fabs(maximumCPU - ARRAY_SIZE) < 1e-10 ? "Passed" : "Failed") << std::endl;

  cudaFree(mA);

  return 0;
}
```

Advanced Features: UM

- Managed memory migration is at the page level
  - The default page size is currently the same as the OS page size today (typically 4 KB)
- The runtime intercepts CPU dirty pages and detects page faults
  - Moves from device over PCI-E only the dirty pages
  - Transparently, pages touched by the CPU (GPU) are moved back to the device (host) when needed
- Coherence points are kernel launch and device/stream sync.
  - Important: the same memory cannot be operated upon, at the same time, by the device and host

Advanced Features: UM

- Issues related to "managed memory size":
  - For now, there is no oversubscription of the device memory
    - In fact, if there are several devices available, the max amount of managed memory that can be allocated is the smallest of the memories available on the devices

- Issues related to "transfer/execution overlap":
  - Pages from managed allocations touched by CPU migrated back to GPU before any kernel launch
    - Consequence: there is no kernel execution/data transfer overlap in that stream
    - Overlap possible with UM but just like before it requires multiple kernels in separate streams
      - Enabled by the fact that a managed allocation can be specific to a stream
      - Allows one to control which allocations are synchronized on specific kernel launches, enables concurrency

UM: Coherency Related Issues

- The GPU has *exclusive* access to this memory when any kernel is executed on the device
  - Holds even if the kernel doesn't touch the managed memory
- The CPU cannot access *any* managed memory allocation or variable as long as GPU is executing
- A cudaDeviceSynchronize() call required for the host to be allowed to access managed memory
  - To this end, any function that logically guarantees the GPU finished execution is acceptable
    - Examples: cudaStreanSynchronize(), cudaMemcpy(), cudaMemset(), etc.

Left: Seg fault

```
__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;
}
```

Right: Runs ok

```
__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;
}
```

[NVIDIA, Programming Guide]→

UM – Current Limitations in CUDA 6.0

- Ability to allocate more memory than the physically available on the GPU
- Prefetching
- Finer Grain Migration

UM – Why Bother?

1. A matter of convenience
   - Much simpler to write code using this memory model
   - For the casual programmer, the code will run faster due to data locality
     - The runtime will take care of moving the data where it ought to be

2. Looking ahead, physical CPU/GPU integration around the corner – memory will be shared
   - Already the case for integrated GPUs that are part of the system chipset
   - The trend in which the industry is moving (AMD's APU, Intel's Haswell, NVIDIA Denver Project)
   - The functionality provided by the current software backend that supports the cudaMallocManaged() paradigm will be eventually implemented in hardware

CUDA 6.0 At Work

Sparse Matrix Reordering for LU Factorization on the GPU

Sparse Linear Algebra on the GPU

- Problem Statement:
  - Solve Ax=b, where A is large and sparse

- Solutions for parallel execution already exist…
  - In cusparse if A is symmetric and positive definite
  - Using MPI or OpenMP

- Ongoing work on general purpose preconditioner using a partitioning of the problem into subproblems that are solved in parallel
  - Approach known in the literature as SPIKE
  - GitHub repo: https://github.com/spikegpu/SpikeLibrary
  - Project webpage: http://spikegpu.sbel.org/
  - Focus herein on the performance of CUDA 6.0 in the context of sparse matrix reordering

Introduction, Matrix Reordering For Diagonal Dominance

- Non-symmetric reordering approach – moves around entries in a matrix by shifting the order of the rows and columns in the matrix
- What's desired: migrate to the diagonal the matrix elements whose absolute value is large
- Why? Helps achieving a high degree of diagonal dominance
  - Decreases probability of encountering a zero pivot during the LU factorization of the sparse matrix
  - Improve the quality of the LU factorization
- Core part of algorithm: finding minimum perfect bipartite matching

Matrix Reordering Algorithm: Four Stages

- First stage: forming bipartite graph
  - View each row as a row node and each column as a column node. There is an edge between each pair of row node and column node.
  - An edge has finite weight if and only if the matrix has a non-zero value in the corresponding entry.
  - Given a matrix, the structure of matrix is unchanged, only values need to be processed to get the weights of the bipartite graph
  - Can be done in parallel on the GPU

- Second stage: finding initial partial match (optional)
  - For each column node j, find a row node i so that edge (i, j) is the minimum over all edges connected to column node j.
  - If row node i is not yet matched to another column node, match i to j
  - Hard to do on the GPU

Stages of the Algorithm (Cont'd)

- Third stage: finding a perfect match
  - If in the second stage all column nodes are matched, then this stage is not necessary
  - Apply Hungarian algorithm (i.e. finding augmenting path) to match each unmatched column node
  - Report error if Hungarian algorithm fails to find a match with finite value for a certain node
  - Totally sequential in nature, no good parallel algorithm suitable for GPU as of now

- Fourth stage: extracting the permutation and the scaling factors
  - Permutation can be obtained directly form the resulting perfect match
  - Stage is highly parallelizable and suitable for GPU computing

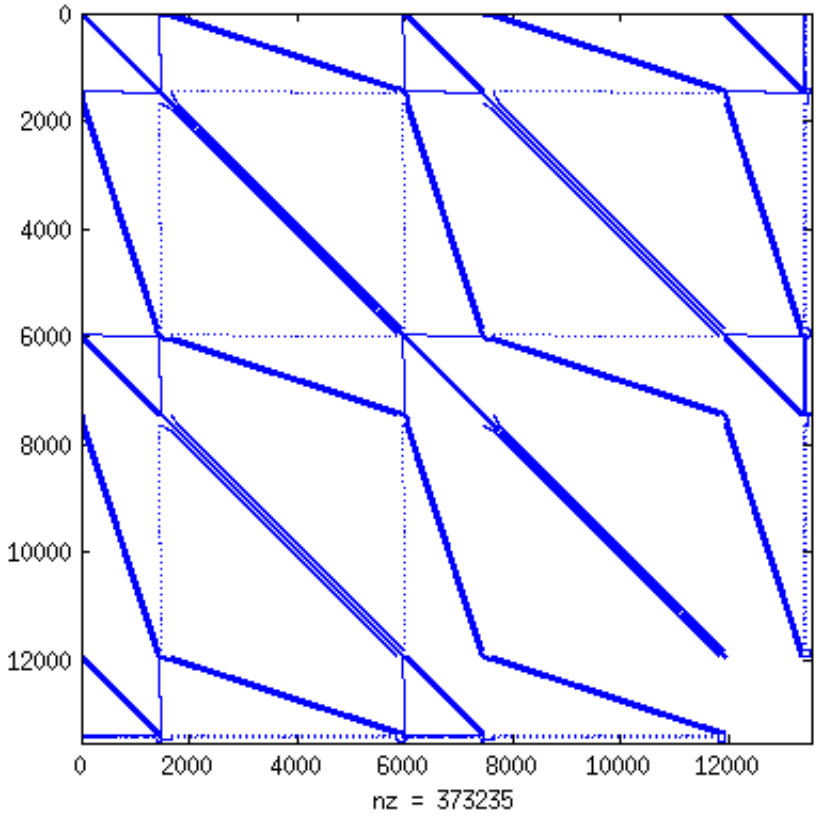
Sparse Matrix Reordering Strategy

- Stages 1 and 4 done on the GPU
- Stages 2 and 3 done on the CPU
- Use of UM:
  - Store all data required to perform matrix reordering in managed memory
    - The GPU accesses data stored in managed memory during stage 1 and 4
    - The CPU accesses data stored in managed memory during stage 2 and 3

- Goals:
  - Simplify code
  - Don’t hurt performance

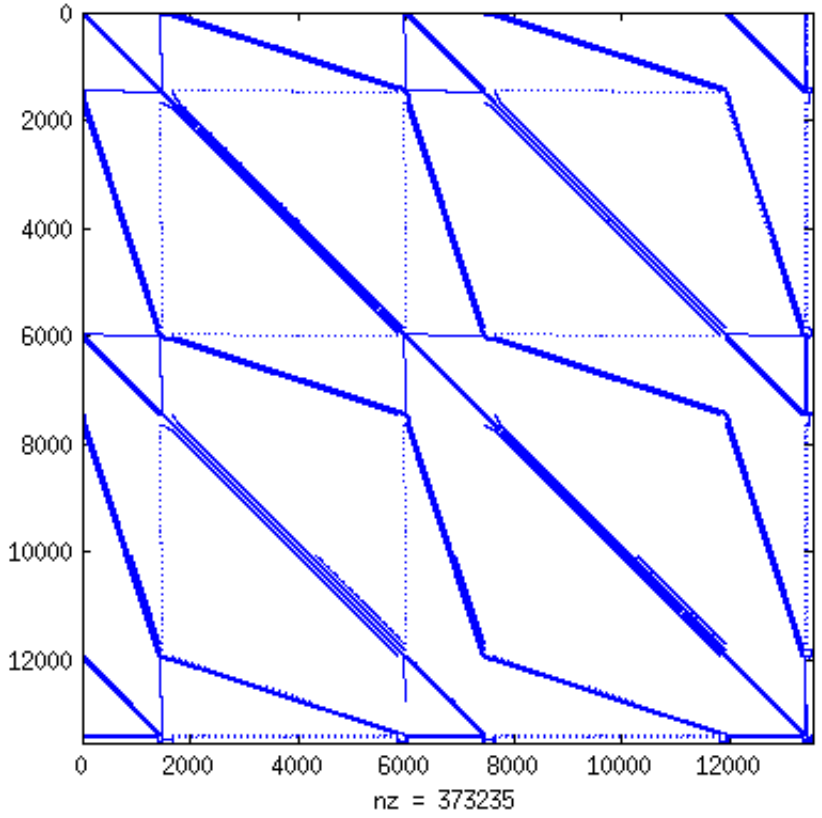
Matrix Reordering Example [Part 1/3]: Sparsity Pattern of Original Matrix

- Garon2: sparse matrix from Florida collection
- Dimension of the sparse matrix
  - N = 13535
- Number of nonzero entries in the matrix
  - NNZ = 373235
- Half bandwidth of the matrix
  - K_ori= 13531



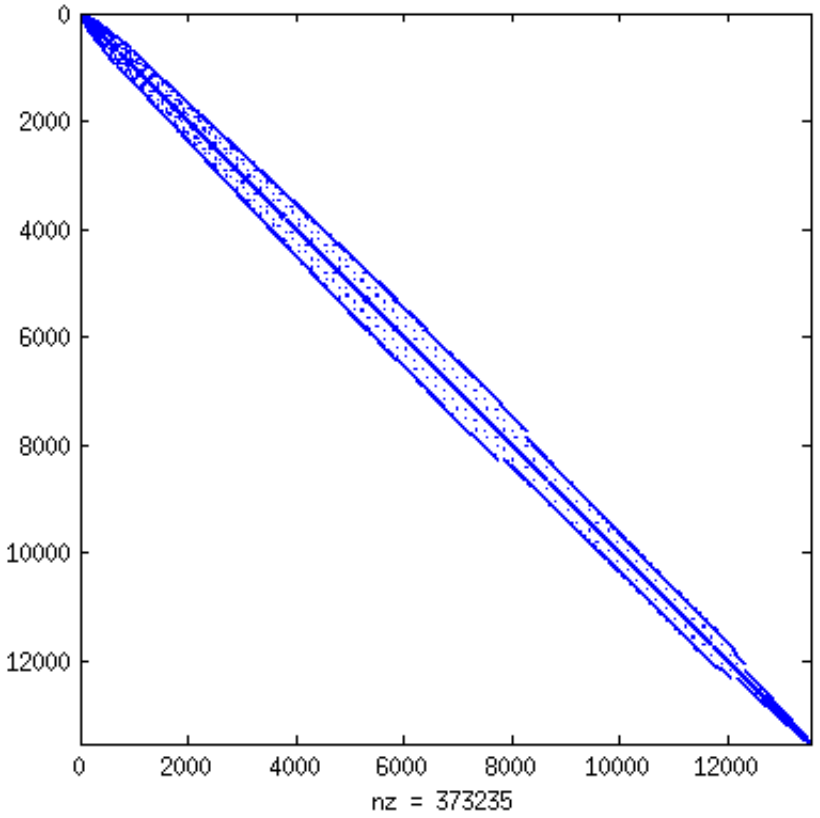
Matrix Reordering Example [Part 2/3]: Sparsity Pattern after the four-stage reordering

- This is the algorithm of interest, gauge overhead associated with use of UM
  - Timing results are presented shortly
- Half bandwidth after reordering:
  - K_4s = 13531
- Bandwidth still large, but all diagonal entries are nonzero and large in magnitude



Matrix Reordering Example [Part 3/3]: Sparsity Pattern after reverse Cuthill-Mckee (RCM)

- All non-zeros of the matrix now close to the diagonal
  - Small half bandwidth:
    - K_RCM = 585
- This matrix is passed to the SPIKE::GPU preconditioner to produce an LU factorization
  - GPU LU factorization details not discussed here



The Key Slide: Performance Comparison

- Ran more than 120 sparse matrix reordering tests
  - Ranked from the best to the worst CUDA 6 performance
  - Took every tenth matrix in this ranking, see table
- Times reported were measured in milliseconds
  - Time that it takes to get the matrix with K_4s
- Bottom line: using UM and letting the runtime take care of business never resulted in more than a 25% slowdown over hand-tuned code
- Almost half of the tests ran faster
  - See "Speedup" column, which reports nominal time divided by time using UM in CUDA 6

Name	Dim.	NNZ	Time Basic	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

Where Else Are We Using CUDA and GPU Computing?

- Fluid-solid interaction problems
- Many-body dynamics, for rigid or flexible bodies
  - Example: dynamics granular material

Interacting rigid and flexible objects in channel flow

Fluid:

$\rho = 1000\ \mathrm{kg/m^3}$

$\mu = 1\ \mathrm{N\ s/m^2}$

$(l_x, l_y, l_z) = (1.4, 1, 1)\ \mathrm{m}$

$Re = 45$

Ellipsoids:

$\rho_s = 1000\ \mathrm{kg/m^3}$

$(a_1, a_2, a_3) = (2.25, 2.25, 3)\ \mathrm{cm}$

$N_r = 2000$

$Re_p = 2$

Beams:

$\rho_s = 1000\ \mathrm{kg/m^3}$

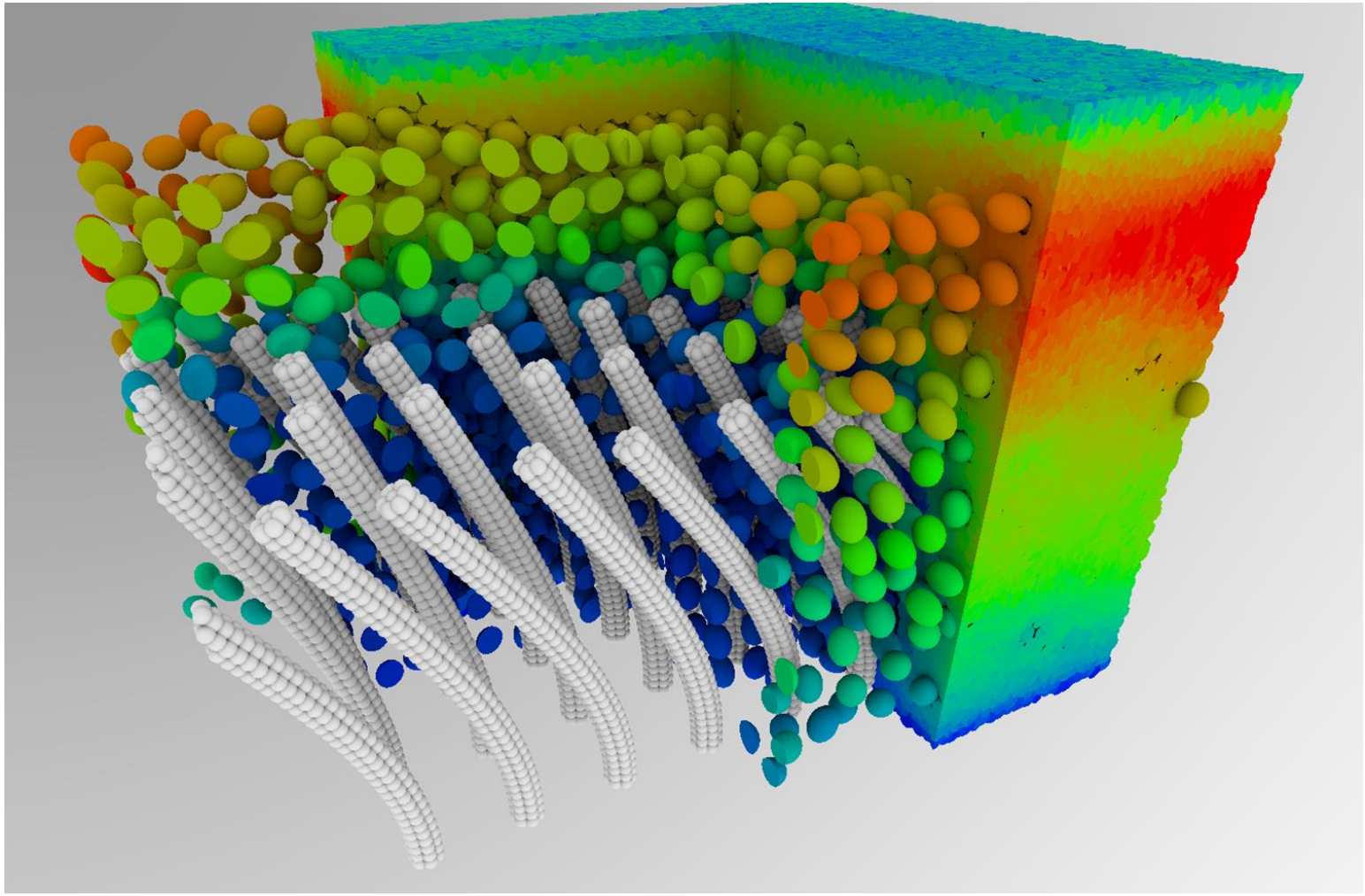
$E = 0.2\ \mathrm{MPa}$

$a = 1.5\ \mathrm{cm}$

$l = 64\ \mathrm{cm}$

$N_f = 40$

$n_e = 4$



Scaling analysis (all together, table)

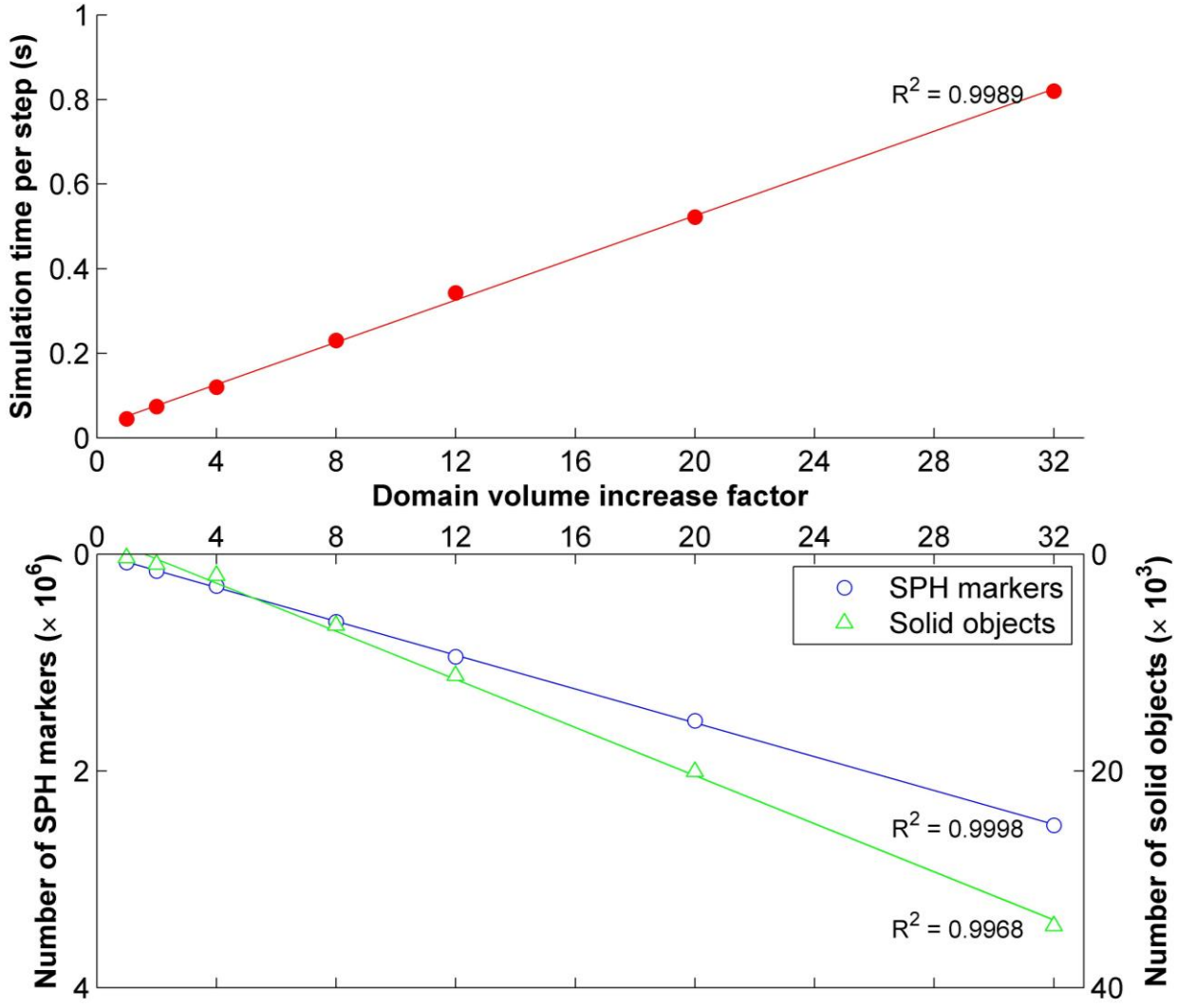
N_m	$(\times 10^6)$	0.08	0.16	0.29	0.63	0.95	1.54	2.50
N_r	$(\times 10^3)$	0.17	0.52	1.12	4.48	7.84	14.56	24.64
N_f	$(\times 10^3)$	0.16	0.42	0.84	2.10	3.36	5.88	9.66
t	(ms)	45	74	120	230	343	522	820

$N_m \simeq 3.0 \times 10^6,\ N_f = 0$

N_r		0	36	120	480	1800	8400	33 600
t (ms)		906	919	923	925	926	926	921

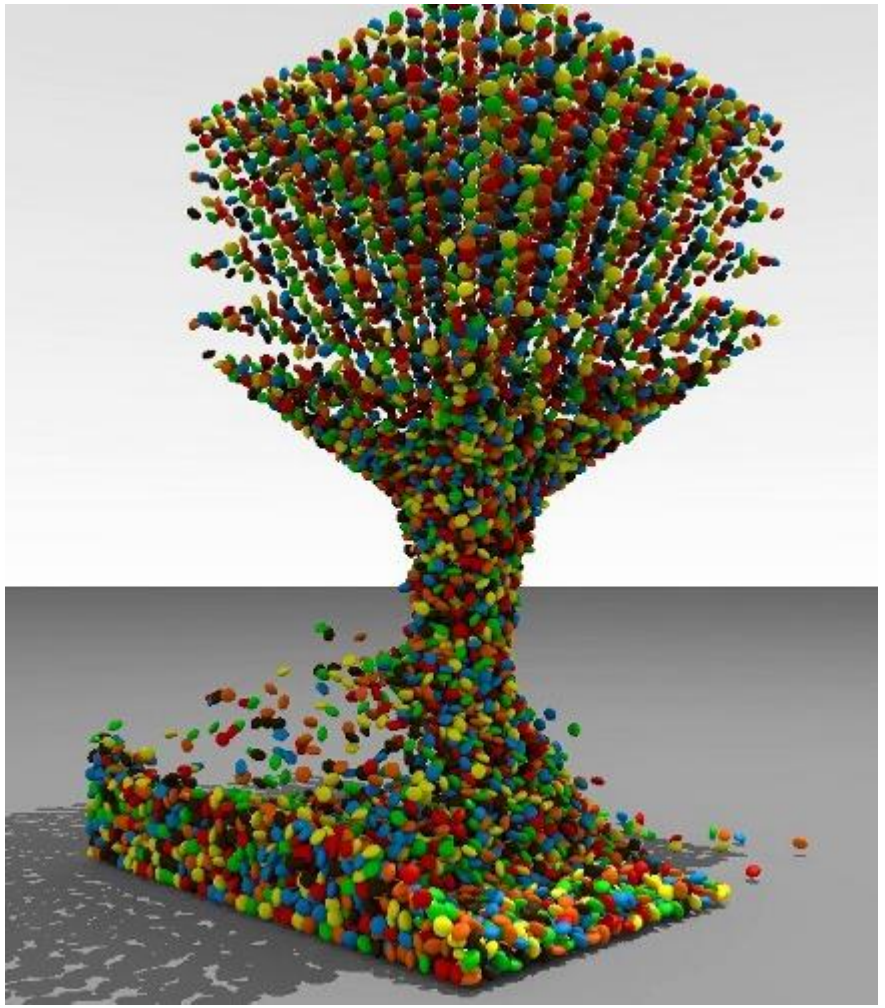
$N_m \simeq 3.0 \times 10^6,\ N_r = 0$

	N_f	0	45	140	440	1152	2100	4704
$\tau = 10$	t (ms)	906	923	928	916	960	950	921
$\tau = 50$	t (ms)	906	973	978	965	1066	1060	1060

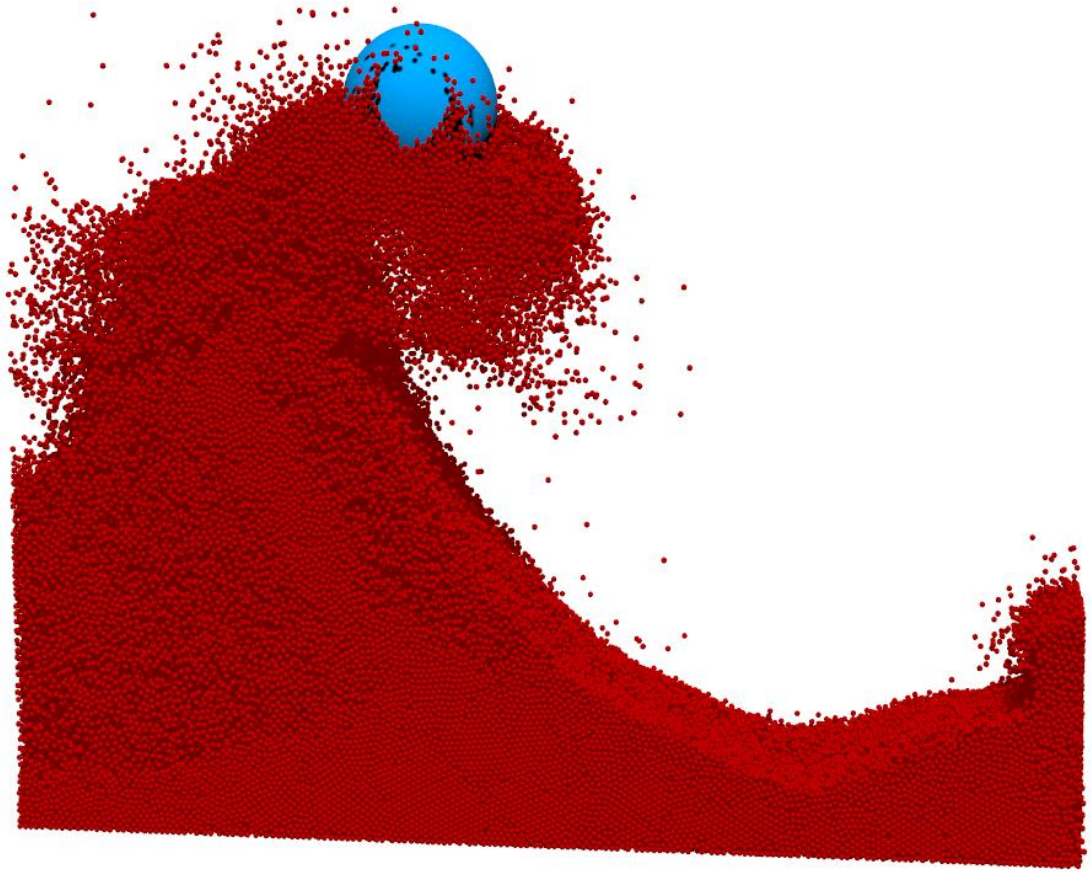


Multibody Dynamics

- Simulation parameters
  - 50,000 Ellipsoids
  - r = [0.26,0.14,0.26] m
  - mu= 0.5
- Step-size: h = 10E-3 s
- Simulation time: 2.1 s / step GTX 480



1.1 Million Body on the GPU



Further Reading, Information Pointers

- CUDA 6.0 Programming Manual
- Mark Harris post: http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/
  - Comments are insightful too
- Thrust 1.8: https://github.com/thrust/thrust
- Interesting posts: http://www.alexstjohn.com/WP/2014/04/28/cuda-6-0-first-look/
- Sparse Linear Algebra: https://github.com/spikegpu/
  - Solve sparse linear systems on the GPU
  - Reordering for diagonal dominance
  - Reverse Cuthill-Mckee (RCM) band reduction

Thank You…

- Justin Madsen, Steve Rennich, Mark Ebersole
  - The nice people above helped with suggestions that improved the quality of the document