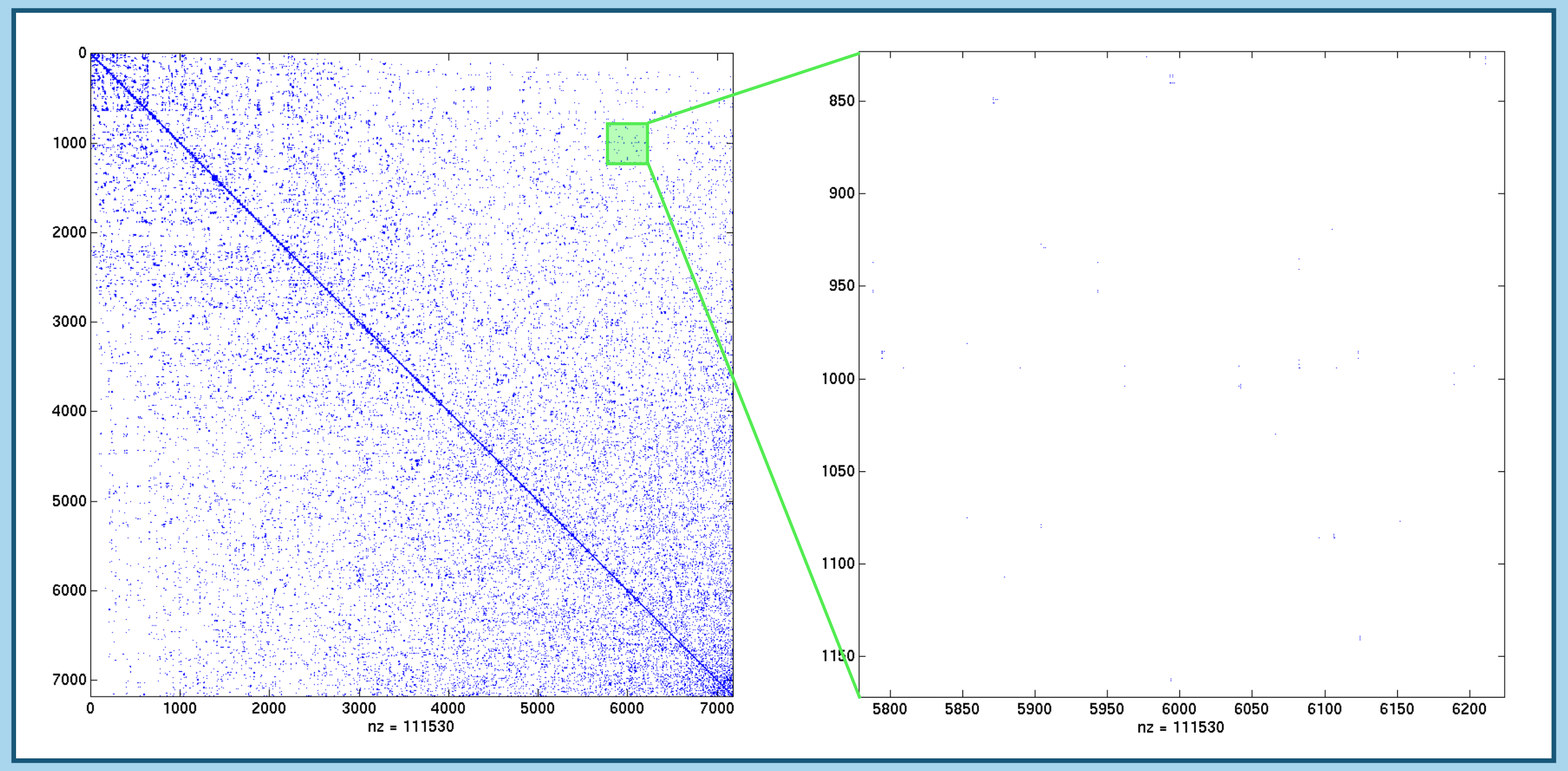
Using GPUs to Accelerate the Solving of Electromagnetic Field Problems

Thomas Nabelek, Department of Electrical and Computer Engineering, University of Missouri – Columbia
Daniel White, Computational Engineering Division, Lawrence Livermore National Laboratory

Abstract

Complex simulations that allow researchers to further understand and visualize important processes, such as changing electromagnetic fields and their effects, require immense computing power. Being able to accelerate and more efficiently complete the computations needed for such simulations allows for solutions to scientific problems to be found in a more expedient manner leading to greater discovery. One such code developed at LLNL is EMSolve: "an effort to develop provably stable time integration methods for solving Maxwell's equations on 3D unstructured grids"[1]. In order to achieve the acceleration sought for this code and others, we make use of GPUs that give us the opportunity for parallelism on the order of 3000 simultaneous calculations. With highly optimized code running on GPUs that makes use of methods such as Compressed Row Storage (CRS), pinned memory, and coalesced memory access, significant speedup of code such as EMSolve can be achieved.

Sparsity of Matrices and Compressed Row Storage (CRS)



Sparsity of a Relatively Small Coefficient Matrix used in EMSolve

EMSolve coefficient matrices have sparsity on the order of 0.01% full, so CRS provides a huge benefit when dealing with memory usage, memory bandwidth, and CPU/GPU utilization. Using CRS for EMSolve necessitates only about 0.02% of the memory capacity and transfer time that would be required to store a full matrix.

	0	*1*	*2*	*3*	*4*	*5*
0	0	**8.3**	0	0	**-7.9**	0
1	0	0	**1.4**	0	0	0
2	0	**-2.3**	0	0	0	0
3	0	0	**1.0**	0	0	**5.5**
4	0	0	0	**9.4**	0	0
5	**-3.7**	0	0	0	0	0

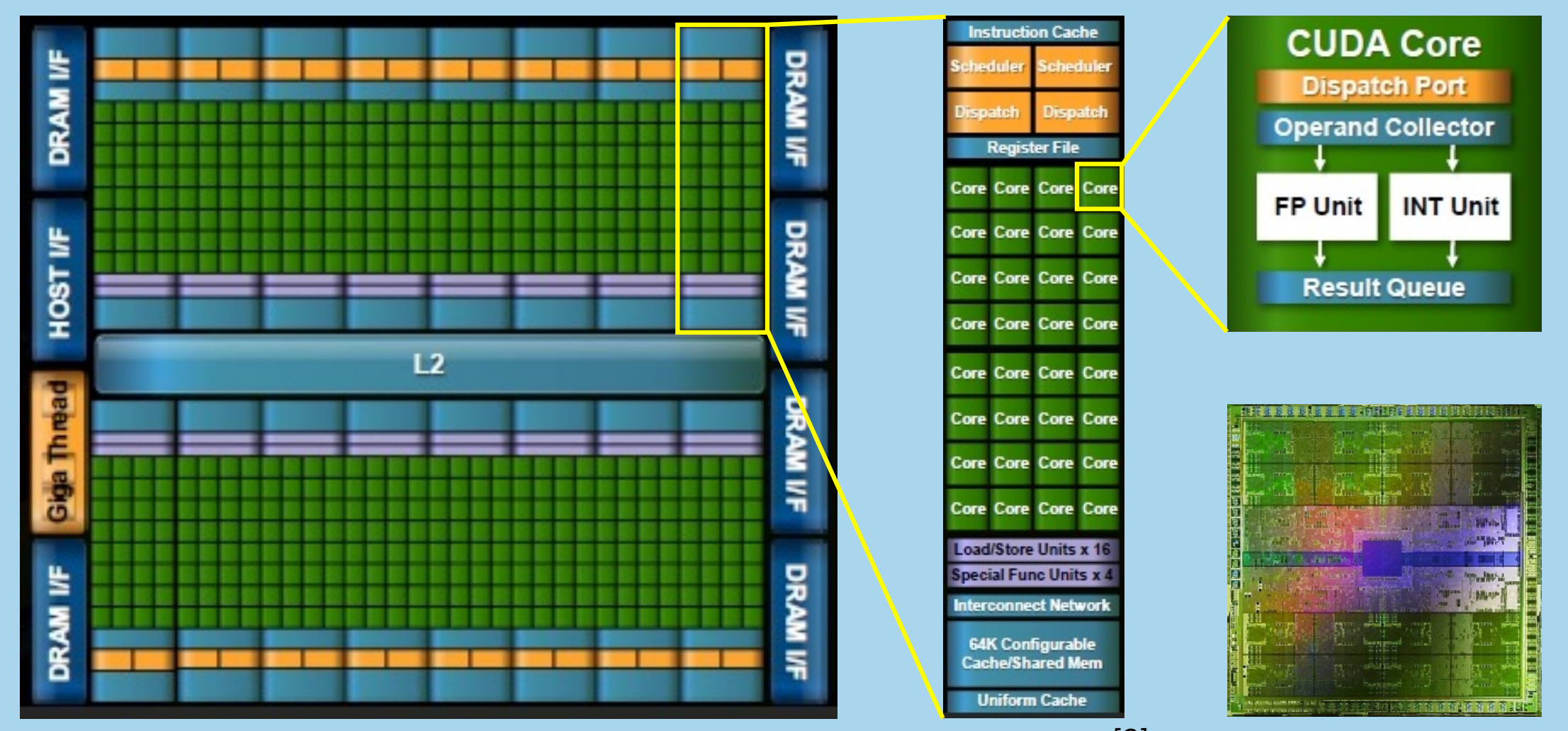
$\Rightarrow$

value	8.3	-7.9	1.4	-2.3	1.0	5.5	9.4	-3.7
col_ind	1	4	2	1	2	5	3	0
row_start	0	2	3	4	6	7	8	

Conversion from Full Coefficient Matrix to CRS Format

Graphics Processing Units (GPUs)

With ~3000-5000 cores per device, GPUs offer an opportunity for massive parallelism that is not possible on CPUs, but that is only advantageous with carefully written and optimized code.

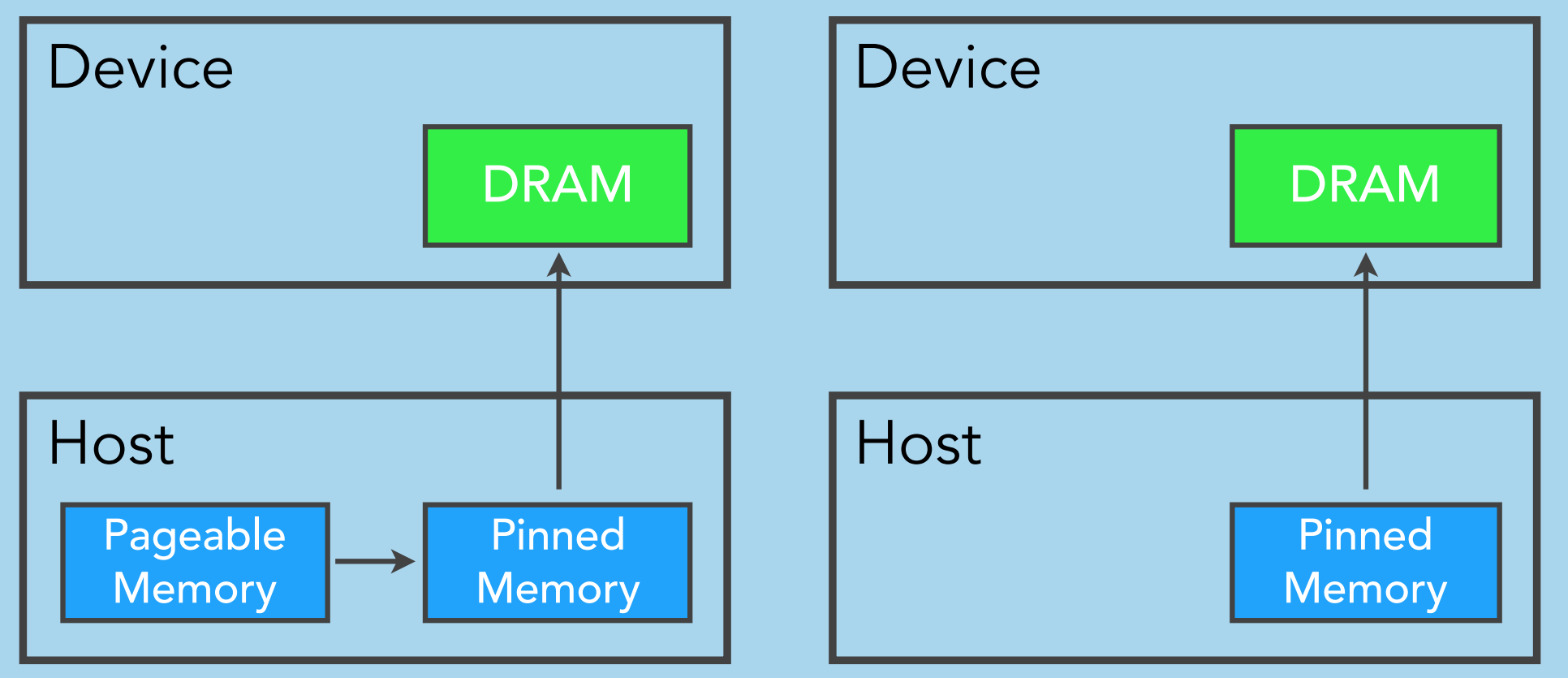


NVIDIA Fermi GPU Architecture[2]

Methods of Accelerating GPU code

Using Pinned Memory

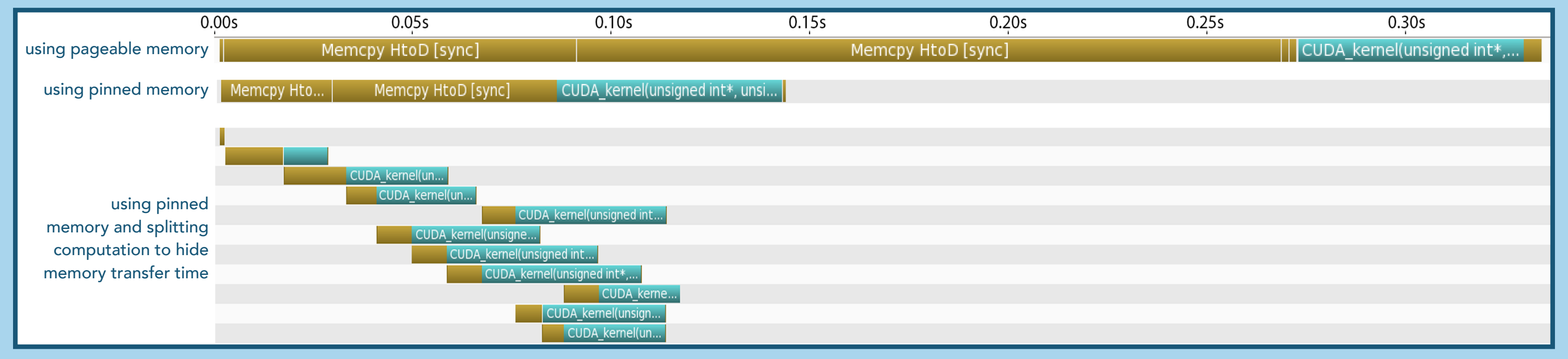
CPU⟶GPU and GPU⟶CPU memory transfer bandwidth is limited and results in long transfer times that should be spent on execution. The issue is furthered by the fact that a GPU device cannot copy from the pageable memory that is standardly allocated by the operating system. The GPU must instead read from pinned memory, so allocating pinned memory directly eliminates the time requirement to transfer data from pageable memory to pinned memory.



Pageable Data Transfer [3] *Pinned Data Transfer* [3]

Hiding Memory Transfer With Computation

In some cases, matrices can be split up so that while one segment of a matrix is being copied to the GPU, computation can be completed on another segment of the matrix.



GPU Timeline

Coalescing Memory Accesses

Making coalesced memory accesses rather than striding through memory is key in reducing global memory requests that have very high latency.

Matrix-Vector Multiplication Results

A matrix-vector multiplication subroutine in EMSolve is the most computationally intensive portion, taking about 56% of the total program execution time.

```
for (int i = 0; i < num_rows; i++)
  for (int j = row_start[i]; j < row_start[i+1]; j++)
     result[i] += value[j] * (x[col_ind[j]] + s[col_ind[j]]);
```

Serial Version of Core EMSolve Matrix-Vector Multiplication Subroutine

Moving this portion of the computation to GPU devices could result in significant speedup of the program. A code representing the matrix-vector multiplication subroutine in EMSolve saw a **5.75x speedup** on the GPU over the fastest parallelized CPU code:

856362 x 856362 sparse matrix with 72376416 nonzero values, 100 iterations (time steps)		
implementation	execution time	speedup
CPU - OpenMP - 16 threads	8.6 seconds	0.80x
CPU - OpenMP - 32 threads (baseline)	6.9 seconds	1.00x
GPU - CUDA without cuSPARSE	5.1 seconds	1.35x
GPU - CUDA with cuSPARSE	1.2 seconds	5.75x

OpenMP : an API providing "constructs and directives for specifying parallel regions, work sharing, synchronization and data environment" [4]

CUDA : an API developed by NVIDIA allowing developers to make use of NVIDIA GPUs

cuSPARSE : a library included in the CUDA toolkit from NVIDIA providing highly optimized subroutines for sparse matrix computations

Machine Specifications:
LLNL LC machine Surface - 1 node

CPU:
16 2.6 GHz cores/node
256 GB of memory/node

GPU:
NVIDIA Tesla K40
Compute Capability: 3.5
15 Streaming Multiprocessors
192 0.75 GHz cores/SM
2880 total cores
12GB of global memory
49152 bytes of shared memory/block

References

[1] D. White. *EMSolve*. [Online]. Available: https://computation.llnl.gov/casc/sc2001_fliers/EMSolve/EMSolve01.html

[2] M. Hoenig (2009, Oct. 09). *The Next Gen Fermi Architecture* [Online]. Available: http://www.hardwarecanucks.com/forum/hardware-canucks-reviews/24145-gpu-technology-conference-nvidias-new-focus-changing-market-6.html

[3] M. Harris (2012, Dec. 04). *How to Optimize Data Transfers in CUDA C/C++* [Online]. Available: http://devblogs.nvidia.com/parallelforall/how-optimize-data-transfers-cuda-cc/

[4] B. Barney (2015, June 04). *OpenMP* [Online]. Available: https://computing.llnl.gov/tutorials/openMP/

Contact

Thomas Nabelek
thomas.nabelek@mail.missouri.edu
Dan White
white37@llnl.gov

Download

goo.gl/3W6co9

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

R&R/IM #: LLNL-POST-676003