Experiences with Emerging HPC Architectures

Peter Messmer*, Paul Mullowney, Keegan Amyx, Travis Austin, John R. Cary, Mike Galloy, Dave Wade-Stein, Matthew Koch, David Fillmore

*messmer@txcorp.com

Tech-X Corporation
5621 Arapahoe Ave., Boulder, CO 80303
http://www.txcorp.com

This work was supported by
NASA SBIR Phase II Award No. NNG06CA13C
DOE Office of Science, SciDAC Program, COMPASS Project, Grant #DE-FC02-07ER41499
DOD Navy SBIR Phase I, Award #N68335-09-C-0247
NVIDIA Corp. and Tech-X Corp.
Motivation/Outline

• Need fast turnaround time for EM/Plasma Simulations
  – Particle in Cell (PIC)
  – Finite-Difference Time-Domain (FDTD)

• Coarse grain parallelization has its limits
  – Local problems getting too small
  – “time does not parallelize”
  – Access to large systems can be painful

• Many algorithms memory bandwidth limited
  – Almost no data reuse -> caches useless
  – Multi-core CPU makes it even worse

⇒ Need high memory bandwidth accelerator

Outline

• GPU architecture, programming
• GPULib: A rapid GPU code development environment
• Implementation of EM algorithms on GPU
• Conclusion
Why scientific computing on GPUs?

Problem: GPUs used to be hard to program

Sources: Nvidia, Techreport.com, cse.scitech.ac.uk
NVIDIA GPUs
A Collection of Specialized SIMD Processors

- Silicon used for ALUs, rather than large caches
  - Up to 240 processing elements ("thread processors")
  - Running at 1.3 GHz, statically scheduled, 2 instructions/cycle
  - Small software managed caches ("shared memory")
- Organized as ‘Multi-processors’ (~ SIMD processors)
  - Software managed caches shared within one multi-processor
  - Synchronization within MP, no light-weight global synchronization
- Active thread management
  - Work on next thread-sets while waiting for a memory request

Fermi: 512 "thread processors" ("GPU cores")
64kB/MP Shared Memory or L1 cache
768k L2 cache

GPU Memory
The large memory bandwidth on GPUs can benefit many scientific computing applications.
The Flipside: GPUs need regular memory access
(but newer generation GPUs are getting less picky)

Ideal access pattern

No problem on C1060 and newer
The future: Fermi introduces new level of flexibility

- Multiple kernels executed concurrently
  - Better performance on kernels with low degree of parallelism

- Hardware managed L1, L2 caches
  - Relaxes coalescing requirements

- C++ support on device

- Enhanced atomic performance

- ECC for reliable scaling
Oxygen: Tech-X’ Production cluster with GPU accelerated nodes

- 32 nodes, each with dual quad core Opteron
- 8GB RAM per node
- Infiniband interconnect
- Lustre file system
- 4 nodes accelerated with 2 Tesla GPU blades
CUDA: Development Environment for NVIDIA GPUs

• Early GPGPU efforts heroic
  – Graphics API (OpenGL, DirectX) no natural fit for scientific computing

• Compute Unified Device Architecture (http://www.nvidia.com/cuda)
  – Supported on all modern NVIDIA GPUs (notebook GPUs, high-end GPUs, mobile devices)
  – Co-Existence with OpenCL (OpenCL basically "IS" CUDA)

• Single Source for CPU and GPU
  – Host code C or C++
  – GPU code C(++) with extensions
    • "Kernel" describes one thread
    • Host invokes a collection of threads
  – nvcc: NVIDIA cuda compiler

• Runtime libraries
  – Data transfer, kernel launch, ..
  – BLAS, FFT libraries

• Simplified GPU development, but still “close to the metal”!

• NEXUS: Visual Studio plug-in for GPU development
GPULib: High-Productivity GPU Computing

- IDL (ITT Vis), MATLAB (Mathworks)
  - C, Fortran
- Rich set of data parallel kernels
- Extensible with proprietary kernels
- Seamless integration into host language
- Explicit or implicit management of address spaces
- Interface to Tech-X’ FastDL for multi-GPU/DMPP computing

http://gpulib.txcorp.com
(free for non-commercial use)

GPULib’s Extensible Architecture

GPULib Host Language Interface (IDL, MATLAB)

GPULib wrappers
(language specific, includes software emulator)

GPULib functions
- Vector Arithmetic
- Data Manipulation
- Complex Operations

NVIDIA functions
- cuBLAS
- cuFFT

CUDA Runtime API

Custom Kernels

GPU
An example of using GPULib in IDL

```
IDL> gpuInit

IDL> x_gpu = gpuPutArr(x)

IDL> y_gpu = gpuSin( x_gpu )

IDL> y = gpuGetArr( y_gpu )

IDL> gpuFree, x_gpu, y_gpu
```
GPULib is currently used in a broad range of applications

- **Simulation / Modeling**
  - Neutron scattering experiments
  - Computational Fluid Dynamics (CFD)
  - Linear optimization
  - Tsunami modeling
  - Option pricing
  - Convection zone in stars
  - Galaxy formation
  - Neural tissue simulations

- **Data analysis**
  - Image enhancement, deblurring
  - Real-time image processing
  - Synthetic Aperture Radar (SAR)
  - Hyperspectral imaging
  - Astronomical imaging
  - Medical imaging
  - Seismic data processing
Key to performance: Multiple kernel invocations per CPU-GPU transfer

- Kernel only
  - \( \text{Lgamma}(x) \)
  - \( \text{Sin}(x) \)
  - \( \text{exp}(x) \)
  - \( ax + by + c \)
  - \( x + y \)

- Single invocation
  - 10 invocations

- 10 invocations
GPU accelerated FDTD Implementation with GPULib

- FDTD usually implemented as stencil operation
  - Traverse e.g. E mesh
  - Grab B values necessary for updating E
  - Special treatment for domain boundaries
  - Interior boundaries: do not update ‘outside’

- Problem 1: Short vector operations
- Problem 2: Poorly aligned boundaries
- Problem 3: Skipping cells

- Solution: Treat 3D domain as large 1D vectors
  - Shifted vector operations ‘cheap’
    - Pointer arithmetic possible on GPUs
  - ‘Dirt’ at domain boundaries due to wrap-around
    - Removed by applying boundary conditions
  - Accept ‘unnecessary’ work
3D FDTD in GPULib

- 3D FDTD
  - Cut-Cell or stair-stepped boundaries
  - Uses VORPAL geometry output

- Performance
  - Up to 400 Mcells/s on
    - ~70% theoretical memory bandwidth
  - ~10 Mcells/s on CPU

⇒ ~40-50x speedup compared to CPU based implementation
  - Comparable to ~48 Franklin cores

- Double precision hit only due to memory bandwidth
  - Think about your units!

- Multi-GPU via message passing among GPU accelerated nodes
  - Eg. 2.6x speedup on a 3 GPU ‘cluster’ (PSC)

- Now part of VORPAL (ongoing)
Benefit of GPUs not limited to FDTD: Boundary Element Methods

- Large systems of linear equations
  - $O(100k)$-$O(1M)$ unknowns or larger

- Large matrices
  - 100k unknowns: 80 GB
  - 1M unknowns: 8 TB

- Direct or iterative method
  - LU for dense systems, reuse for different RHS
  - Iterative possibly faster for single solution

- Too large to fit into memory of small cluster

- Solution time scales with $O(N^3)$
  - Interesting problems take days to weeks

⇒ need fast parallel out-of-core solvers
⇒ Use GPUs as low-cost accelerators
GPUs can significantly accelerate out-of-core LU decompositions

- Based on Azevedo/Dongarra OOC solver

**GPU vs CPU Speedup for 8 way parallel OOC LU decomposition**

![Graph showing GPU vs CPU Speedup](image)

**LU Factorization Time for NB=400**

- Single precision, real
Summary/Conclusions

- GPUs offer large potential for accelerating scientific applications
- GPULib enables GPU development from within VHLLs
- FDTD solver benefits well from GPU
- Parallel GPU accelerated OOC solver for MOM codes
- GPUs yields ~10x-40x speedup compared to CPU
- Fermi a big leap forward, both in performance and usability