Computation of ρ^{ref} using GPUs

Peter Brune

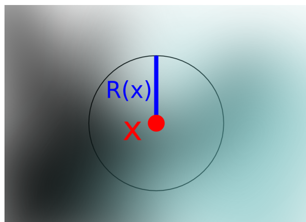
Department of Computer Science
University of Chicago

February 12, 2010; Rush University Medical Center

The Reference Density

ρ^{ref} is calculated from the charge-neutralized per-species densities, $\bar{\rho}$
Perturbations around ρ^{ref} are used for computation of the electrostatic interactions

ρ^{ref} and the Screening Radius



$$\rho^{ref}(x) = \int \rho(x') \frac{\theta(|x' - x| - R(x))}{\frac{4\pi}{3} R^3_{filter}(x)} dx' \tag{1}$$

- Where θ is the **Heaviside** function
- $R = \frac{\sum_i R_i \rho_i}{\sum_i \rho_i} + \frac{1}{2\Gamma(x)}$
- Realspace quadrature has ($\mathcal{O}(1)$ error in 3D.)

Spectral Quadrature

$$\hat{\mathbb{K}}^{\vec{x}}(\vec{k}) = 3e^{i\vec{k}\cdot\vec{x}}\left\{-\frac{1}{k^2R^2}\cos kR + \frac{1}{k^3R^3}\sin kR\right\}. \tag{2}$$

$$\begin{aligned}
\rho^{ref}(x) &= \int \rho(x')\frac{\theta(|x'-x| - R(x))}{\frac{4\pi}{3}R^3_{filter}(x)}dx' && (3)\\
&= \int \rho(x')\mathbb{K}^{\vec{x}}(x')dx' && (4)\\
&= \int \hat{\rho}(\vec{k})\hat{\mathbb{K}}^{\vec{x}}(\vec{k})dk && (5)\\
& && (6)
\end{aligned}$$

- For R non-constant in $\vec{x}$ this is not a convolution
- Inner product of $\hat{\rho}(\vec{k})$ and $\hat{\mathbb{K}}^{\vec{x}}(\vec{k})$ for each $x \in \Omega$
- potentially higher complexity
- more accurate (closed-form rather than quadrature)

Naive Algorithm

The $\mathcal{O}(n^6)$ algorithm for $\mathcal{O}(n^3)$ gridpoints is much like applying a **dense matrix** except that:

- We never have to *assemble* the matrix
- data traffic is $\mathcal{O}(n^3)$ instead of $\mathcal{O}(n^6)$
- perfect for SIMD / SIMT (Single Instruction Multiple Data / Thread)

GPGPUs

- Closer to old-style vector processing supercomputers than modern clusters
- Single Instruction Multiple Data has made a comeback
- Multi-tiered memory hierarchies with a wide bus

Example GPUs

Figure: An NVIDIA Tesla board

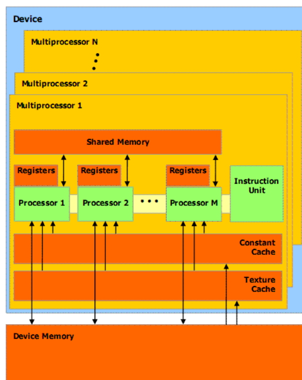
- GTX 285M - 128 SPs, 1 GB Global Memory - Inc. in Laptops
- GTX 285 - 240 SPs, 1 GB Global Memory - $400
- C2070 (Tesla) - 512 SPs, 6 GB Global Memory - $3,999

We're seeing this kind of processing in desktop machines, laptops. Perfect for small-scale science.

CUDA

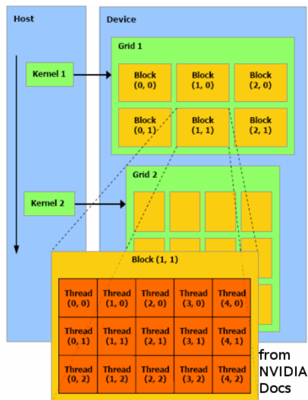
- CUDA (Compute Unified Device Architecture) is NVIDIA's API
- C with NVIDIA extensions
- Support for many different products (All of the company's cards since 2008)

GPU Layout



- a **Streaming Multiprocessor (SM)** has several **Streaming Processors(SP)**s
- each **SP** can access **shared**, **constant**, and **global** memory

Computing Environment



from NVIDIA Docs

- **SP**s → threads
- **SM**s → thread blocks
- A single GPU program is called a *kernel*
- Kernels may be enqueued on the GPU from the CPU

Performance Issues

- Threads are organized in *warps* of 32
- Flow control splits warps into serialized subparts
- All threads can access global memory at once (fairly fast)
- A warp may quickly access *all different* or *all the same* shared memory addresses

PyCUDA

- PyCuda allows for an easy interface to CUDA
- Many builtins for memory transfers, etc.
- Allows easy *Code Generation and Auto-Tuning*
- http://mathema.tician.de/software/pycuda

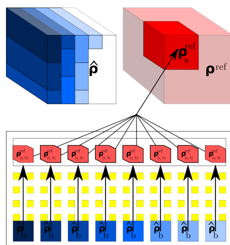
CUDA has become very popular in

- Medical Sciences (Imaging)
- Biology
- Cryptographic Application
- etc.

In order to use the GPU most efficiently, we

- Attempt to minimize inter-thread communication
- Attempt to maximize reuse of data stored in shared memory

Algorithm



The algorithm is two-stage:

1. Compute per-block action of the kernel for a section of realspace
2. Compute the final value sums for the section of realspace and write it back

These can be done using two separate GPU kernels which martial the processors differently.

Step 1: Computation of $\mathbb{K}^{\vec{x}_i}(k_j)$

Each kernel invocation operates over some range of $\mathbb{R}^3$ consisting of $\{x_i\}$. As initialization:

- Each thread block gets some range of $\vec{k}_j$, B
- Each thread block precomputes quantities reused per $\vec{k}$ in the kernel
- Size of B limited by shared memory

Then,

- for each $\vec{x}$ compute $\mathbb{K}^{\vec{x}_i}(\vec{k}_j)$ distributing B among threads
- the answers are summed to form $\rho^{ref}(\vec{x}_i)_B$

Step 2: Reduction

- Each block gets assigned a single $\vec{x}$
- The $\rho_B^{ref}(x_i)$ are summed to form $\rho^{ref}(x_i)$

Implementation Details

- The main DFT implementation is done using `PETSc`
- The Screening Density Calculation in Python using `PyCuda`
- A simple `Cython` wrapper is used to allow DFT to call python functions
- Keeping the CUDA portion in PyCuda allowed for rapid prototyping and testing

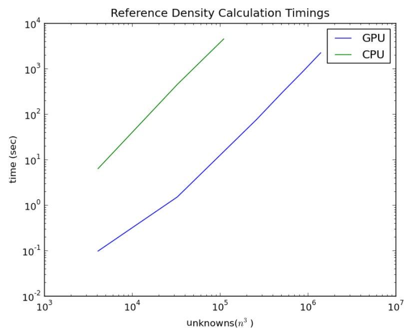
Experimental Setup

The test system, *BaconOst.cs.uchicago.edu*, consists of

- a GTX 285 (240 SPs, 1 GB Global Memory)
- an Intel Core 2 Duo E8400 3.0 GHz
- 4 GB RAM

And cost $1000.

Scaling



Percent of Peak

- We get about 124 GFLOPS
- Peak is 660 GFLOPS
- Algorithm can be improved

DFT Wall Problem

- How the species interact against a hard wall
- 1D to test against

The setup is:

- A periodic 2x2x6 nanometer domain
- The wall at 0.3 nm
- Two ion species (+ and -)

DFT Wall Problem Timings

Table: Runtimes and # of iterations

N	t (min)	Iterations
21x21x21	4:05	60
41x21x21	13:02	74
81x21x21	36:47	71
161x21x21	93:36	59

Contrasted with **weeks** on a serial processor!!

Contribution to DFT

- The implementation has allowed us to run reasonable problem sizes
- We will begin to run a number of test problems
- We will be able to validate the model for problems in 3D

Performance Analysis

- Noticeable improvements in the algorithmic implementation possible
- Auto-tuning using PyCuda for various problem sizes possible using just-in-time kernel compilation
- Ability to *compress* the operator application by some rank-reduction analysis

Dimensionality Reduction

The kernel

$$K^{\vec{x}}(\vec{k}) = \int_{\mathbb{R}^3} 3e^{i\vec{k}\vec{x}}(-\frac{1}{(|\vec{k}|R)^2}\cos(|\vec{k}|R) + \frac{1}{(|\vec{k}|R)^3}\sin(|\vec{k}|R)) \quad (7)$$

May also be written

$$\rho_R^{ref}(x) = K^R(\vec{x}) = 3e^{i\vec{k}\vec{x}}(-\frac{1}{(|\vec{k}|R)^2}\cos(|\vec{k}|R) + \frac{1}{(|\vec{k}|R)^3}\sin(|\vec{k}|R)) \quad (8)$$

- For $\{x : R(x) = R\}$.
- Compute ρ_R^{ref} for some $\{r_i\}$ and interpolate to all values of R.
- This is at worst $\mathcal{O}(n^4)$

Conclusion

- We've implemented this algorithm using newly available multicore architectures.
- This should enable validation of the Classical DFT model for Ion Channels
- Further improvements to come.