Accelerating the Conjugate Gradient Method with CUDA

Matt Pennybacker

The University of Arizona

28 April 2010

Motivation

You want to solve the incompressible Euler equations:

$$\mathbf{u}_t + (\mathbf{u} \cdot \nabla)\,\mathbf{u} + \nabla p = 0$$
$$\nabla \cdot \mathbf{u} = 0$$

Splitting the operator, you find that you must solve:

$$\mathbf{u}^* + \Delta t\,(\mathbf{u}^n \cdot \nabla)\,\mathbf{u}^n = 0$$

Followed by:

$$\mathbf{u}^{n+1} + \Delta t\,\nabla p = \mathbf{u}^* \quad \text{such that} \quad \nabla \cdot \mathbf{u}^{n+1} = 0$$

Taking the divergence of both sides:

$$\Delta t\,\Delta p = \nabla \cdot \mathbf{u}^*$$

The Fundamental Problem

The discretization of this Poisson equation is of the form:

$$A\mathbf{x} = \mathbf{b}$$

Conveniently, the matrix A is real, symmetric, and positive definite so many solution methods exist:

- LU Decomposition (Gaussian Elimination)
- Cholesky Decomposition
- Jacobi Iteration
- Gauss-Seidel Iteration
- Multigrid Relaxation

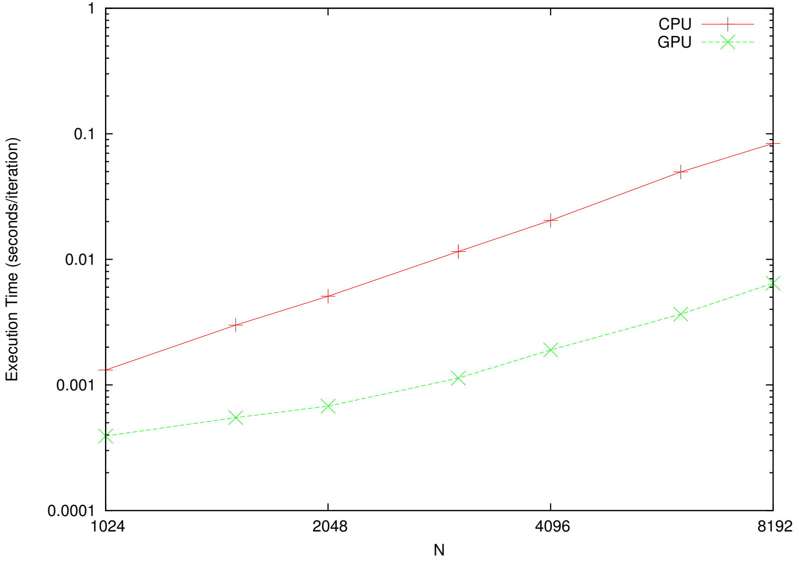
The Conjugate Gradient Method

This method is flexible and easy to implement:

```
$\mathbf{x}_0 = \mathbf{0},\ \mathbf{r}_0 = \mathbf{b},\ \mathbf{p}_0 = \mathbf{b}$
for $n = 1, 2, 3, \ldots$
    $\alpha_n = (\mathbf{r}_{n-1}^T \mathbf{r}_{n-1}) / (\mathbf{p}_{n-1}^T A \mathbf{p}_{n-1})$
    $\mathbf{x}_n = \mathbf{x}_{n-1} + \alpha_n \mathbf{p}_{n-1}$
    $\mathbf{r}_n = \mathbf{r}_{n-1} - \alpha_n A \mathbf{p}_{n-1}$
    break if $\|\mathbf{r}_n\| < \epsilon \|\mathbf{r}_0\|$
    $\beta_n = (\mathbf{r}_n^T \mathbf{r}_n) / (\mathbf{r}_{n-1}^T \mathbf{r}_{n-1})$
    $\mathbf{p}_n = \mathbf{r}_n + \beta_n \mathbf{p}_{n-1}$
```

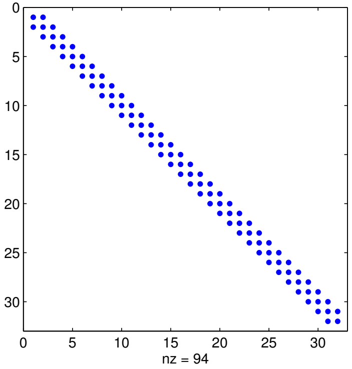
All of the necessary operations are available in BLAS.

Initial Results



Sparse Matrices

More than 80% of the execution time on the GPU is spent on matrix multiplication, an $\mathcal{O}(N^2)$ operation. This can be reduced to $\mathcal{O}(N)$ by taking advantage of sparsity.



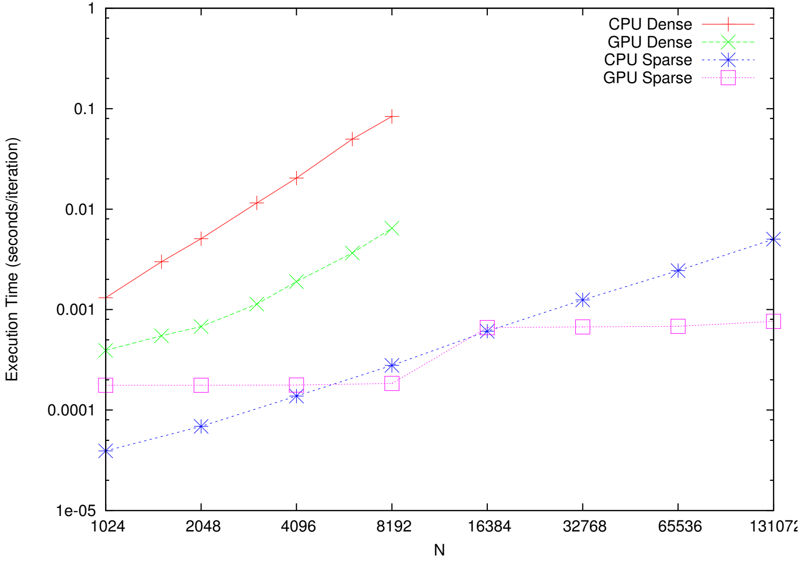
Shared Memory

Each row of the vector **b** requires two entries in **x** that were used to compute the previous row, so read a portion of **x** into shared memory:

```
extern __shared__ double s[];

s[sx] = x[ix];
if (threadIdx.x == 0 && ix-1 >= 0)
  s[sx-1] = x[ix-1];
if (threadIdx.x == blockDim.x-1 && ix+1 < n)
  s[sx+1] = x[ix+1];
```

Sparse Matrix Results



A Few Comments

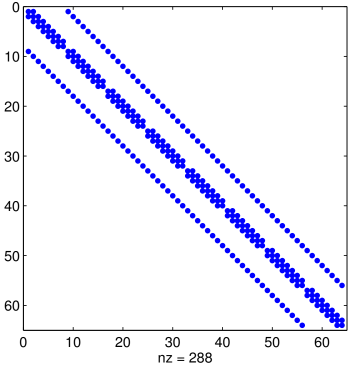
Execution time on the CPU has been reduced $\sim 10\times$ from the dense GPU implementation and $\sim 100\times$ from the dense CPU implementation.

Matrix multiplication now accounts for less than 10% of the execution time on the GPU.

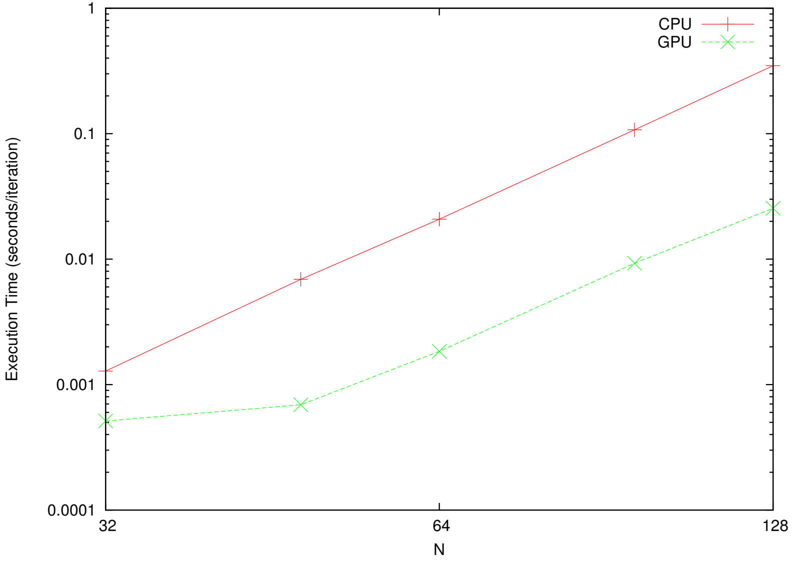
Execution time on the GPU is independent of matrix size?! This may be due to the execution time being dominated by black box routines developed by NVIDIA that can hide latency *very* effectively.

Adding a Dimension

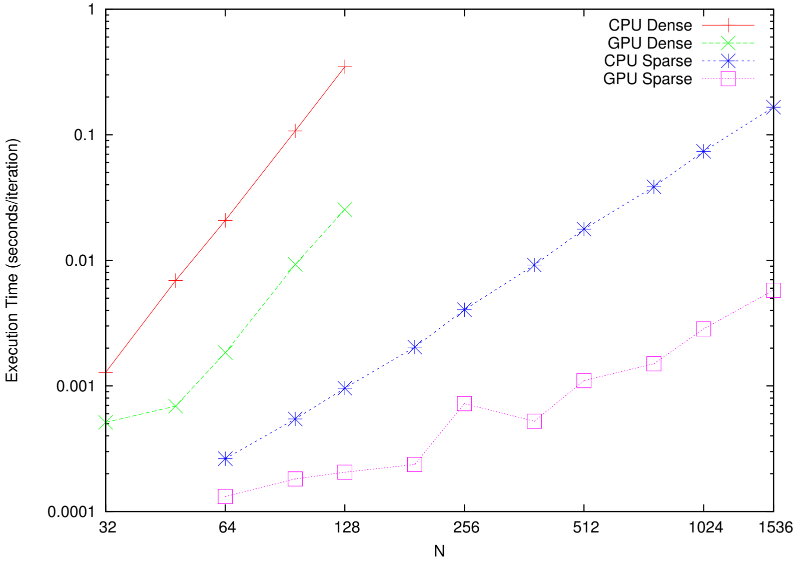
In 2D, the matrix A once again has a predictable sparsity pattern. It also acts with high spatial locality if $\mathbf{x}$ is interpreted at a matrix.



Dense Matrix Results



Sparse Matrix Results



A Few More Comments

At $N = 128$, the dense matrix is 2GB in size. At $N = 256$, it crashes my workstation.

The sparse GPU implementation is $\sim 10\times$ faster than the sparse CPU, $\sim 100\times$ faster than the dense GPU, and $\sim 1000\times$ faster than the dense CPU.

The CPU implementation is slower for all matrix sizes due to the fact that the 2D spatial locality is not well-suited to its cache layout.

By the Numbers

1D Poisson:

Matrix Size	1024	2048	4096	8192	16384
Iterations	510	1019	2031	4028	7923
CPU (Dense)	0.67	5.18	41.44	337.24	—
GPU (Dense)	0.20	0.69	3.86	26.01	—
Speedup	3.35×	7.51×	10.7×	13.0×	—
CPU (Sparse)	0.02	0.07	0.28	1.12	4.82
Speedup	33.5×	74.0×	148×	301×	—
GPU (Sparse)	0.09	0.18	0.36	0.74	5.24
Speedup	7.44×	28.8×	115×	456×	—

By the Numbers

2D Poisson:

Matrix Size	32^2	64^2	128^2	256^2	512^2
Iterations	76	146	277	526	952
CPU (Dense)	0.05	1.58	50.78	—	—
GPU (Dense)	0.02	0.14	3.71	—	—
Speedup	2.50×	11.3×	13.7×	—	—
CPU (Sparse)	—	0.02	0.14	1.12	9.33
Speedup	—	79.0×	363×	—	—
GPU (Sparse)	—	0.01	0.03	0.20	0.58
Speedup	—	158×	1690×	—	—

Next Steps

A preconditioner could be used to speed up convergence of the conjugate gradient method at the expense of additional computation. There does not appear to be documentation of any effective implementation of a preconditioner in CUDA.

In the end, I hope to integrate these routines into a solver for the incompressible Euler equations. Additional investigation is necessary to determine the optimal shared memory usage for the 3D discrete Poisson equation.

Questions?