HYPERDRIVE

IMPLEMENTATION AND ANALYSIS OF A

PARALLEL, CONJUGATE GRADIENT LINEAR SOLVER

AVISHA DHISLE ADHISLE

PRERIT RODNEY PRODNEY

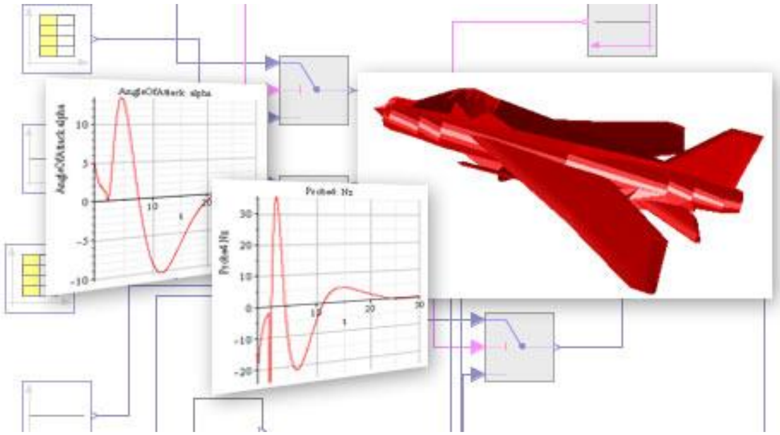
15618: PARALLEL COMPUTER ARCHITECTURE

PROF. BRYANT

PROF. KAYVON

LET’S BUILD A PARALLEL CONJUGATE GRADIENT SOLVER AND ANALYZE ITS PERFORMANCE

- Many real-world applications like flight simulators, fluid dynamics, circuit theory are represented by non-linear system of equations, ordinary and partial differential equations.
- Discretization of above system of equations result in linear systems formulated as: $Ax = b$
- Solving of these discrete linear systems derived from large and complex real-world models using iterative methods is an active area of research.



BOTTLENECKS THAT LEAD TO A SUB-OPTIMAL PERFORMANCE OF THE CG ALGORITHM

- Size of the matrices is often large ⟶ Bandwidth bound
- Poorly conditioned matrix ⟶ Higher divergence, slower convergence
- Matrix-vector product ⟶ Computationally intensive O(N^3)

WHAT IS OUR PARALLEL CG SOLVER CAPABLE OF?

- ✓ Provides a sequential implementation of the Preconditioned Conjugate Gradient solver
- ✓ Includes a multithreaded implementation of the PCG solver using OpenMP primitives
- ✓ Presents a GPU implementation of the PCG solver using CUDA
- ✓ Demonstrates a considerable speedup in the convergence of the equation $Ax = B$ for both parallel implementations over the sequential implementation

INPUT AND OUTPUT CONSTRAINTS OF THE SYSTEM

INPUT

Matrix A must be-

- ✓ symmetric
- ✓ positive definite
- ✓ banded

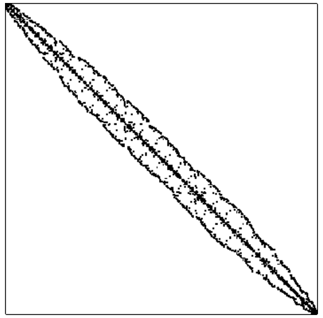
Matrix B must be-

- ✓ a vector of size equal to the order of Matrix A

OUTPUT

- ✓ Computed x vector by PCG algorithm
- ✓ L2 norm of the residual vector must be lesser than 10^-5 for convergence

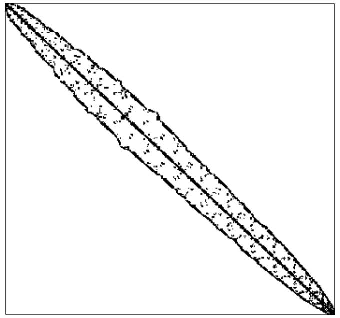
VISUAL REPRESENTATIONS OF TEST MATRICES USED



BCSSTK14

Matrix size:
1806 x 1806

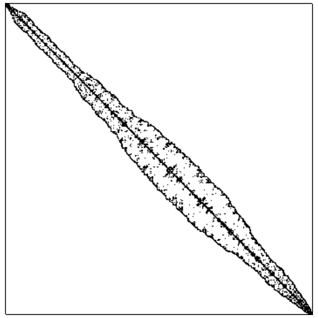
No. of non zero elements:
63,454



BCSSTK15

Matrix size:
3948 x 3948

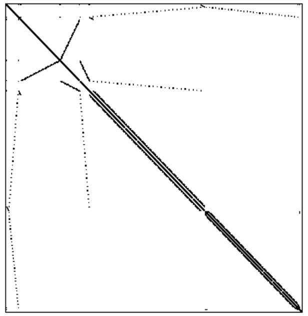
No. of non zero elements:
117,816



BCSSTK18

Matrix size:
11948 x 11948

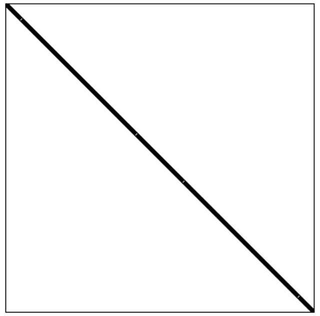
No. of non zero elements:
149,090



S3RMT3M3

Matrix size:
5357 x 5357

No. of non zero elements:
207,695



S3DKT3M2

Matrix size:
90449 x 90449

No. of non zero elements:
3,753,461

LARGE, SPARSE MATRICES WERE STORED IN COMPRESSED ROW STORAGE FORMAT

- As size of the matrices increased, a decrease in performance was observed due to increased cache misses since the full matrix was not fitting in the cache.
- Hence, store only non-zero elements of the matrices.

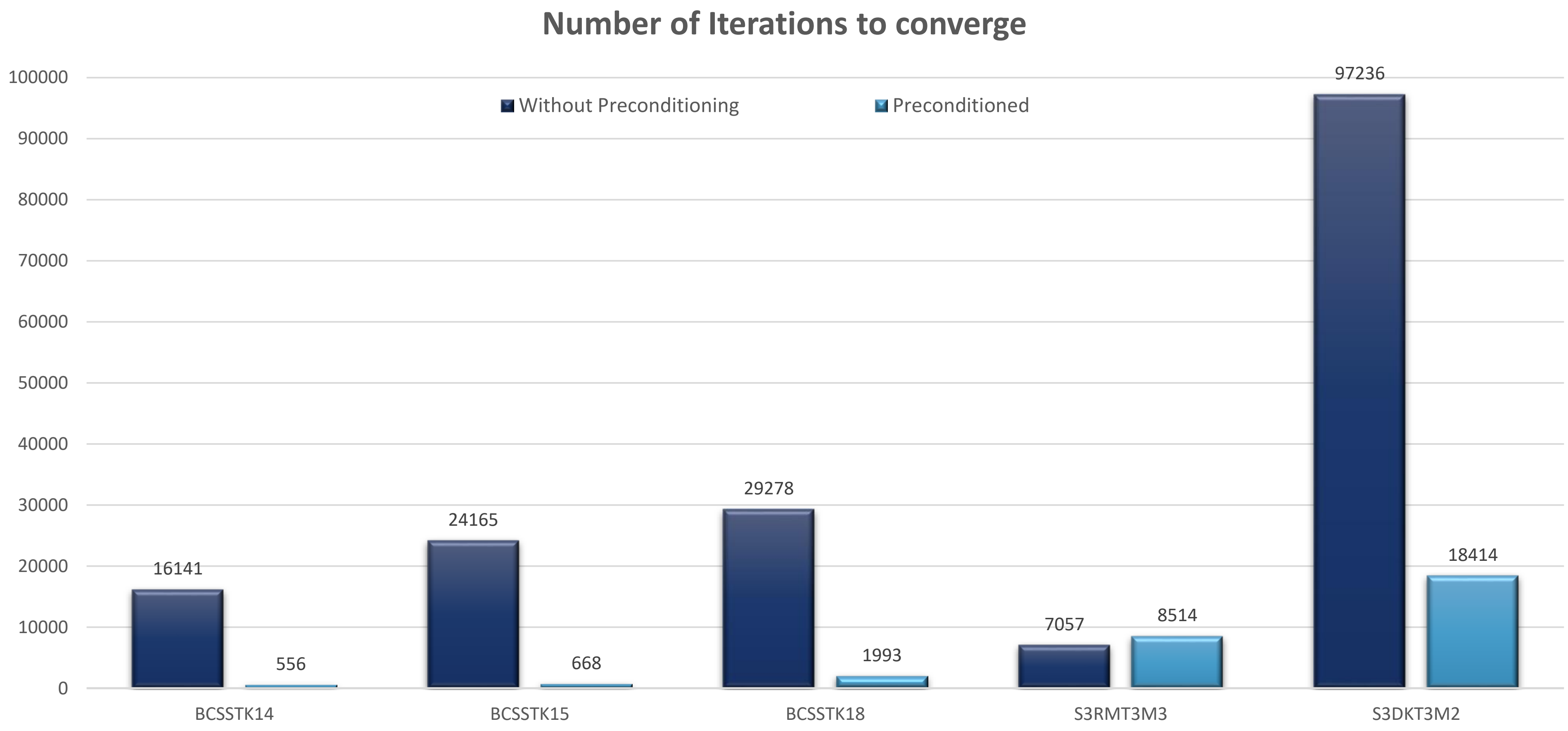
Value of non zero elements	**1**	**2**	**5**	**5**	**3**	**1**	**1**	**4**
Column index	**0**	**1**	**2**	**1**	**2**	**3**	**2**	**3**

Row pointers	**0**	**1**	**3**	**6**

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 5 & 0 \\ 0 & 5 & 3 & 1 \\ 0 & 0 & 1 & 4 \end{bmatrix}$$

- We obtained an reduction in memory storage of 99.907% for the 90449 x 90449 matrix!

JACOBI PRECONDITIONING TO REDUCE THE NUMBER OF ITERATIONS FOR CONVERGENCE



$$M^{-1}Ax = M^{-1}b$$

Preconditioning reduces the condition number of the matrix

Helps put a bound on the inaccuracy of the solution X

All tests were carried out on the GHC machines

GHC Machines CPU Specs: Xeon E5-1660, 8 cores (2x hyperthreaded), 32GB DRAM

PROFILING OF SEQUENTIAL CODE TO DETERMINE FUNCTIONS THAT NEED PARALLELIZATION

Conjugate Gradient Algorithm

$r_0 = b - Ax_0$

$z_0 = M^{-1}r_0$

$p_0 = z_0$

$i = 0$

repeat

$\alpha_k = \frac{r_k^T z_k}{p_k^T A p_k}$ // Requires Vector dot product and Matrix vector product

$x_{k+1} = x_k + \alpha_k p_k$ // Requires scalar vector product and vector sum

$r_{k+1} = r_k - \alpha_k A p_k$ // Requires scalar vector product and vector difference

$z_{k+1} = M^{-1} r_{k+1}$ // Requires Matrix vector product

$\beta_k = \frac{r_{k+1}^T z_{k+1}}{z_k^T r_k}$ // Requires vector dot product

$p_{k+1} = z_{k+1} + \beta_k p_k$ // Requires scalar vector product and vector sum

$k = k + 1$

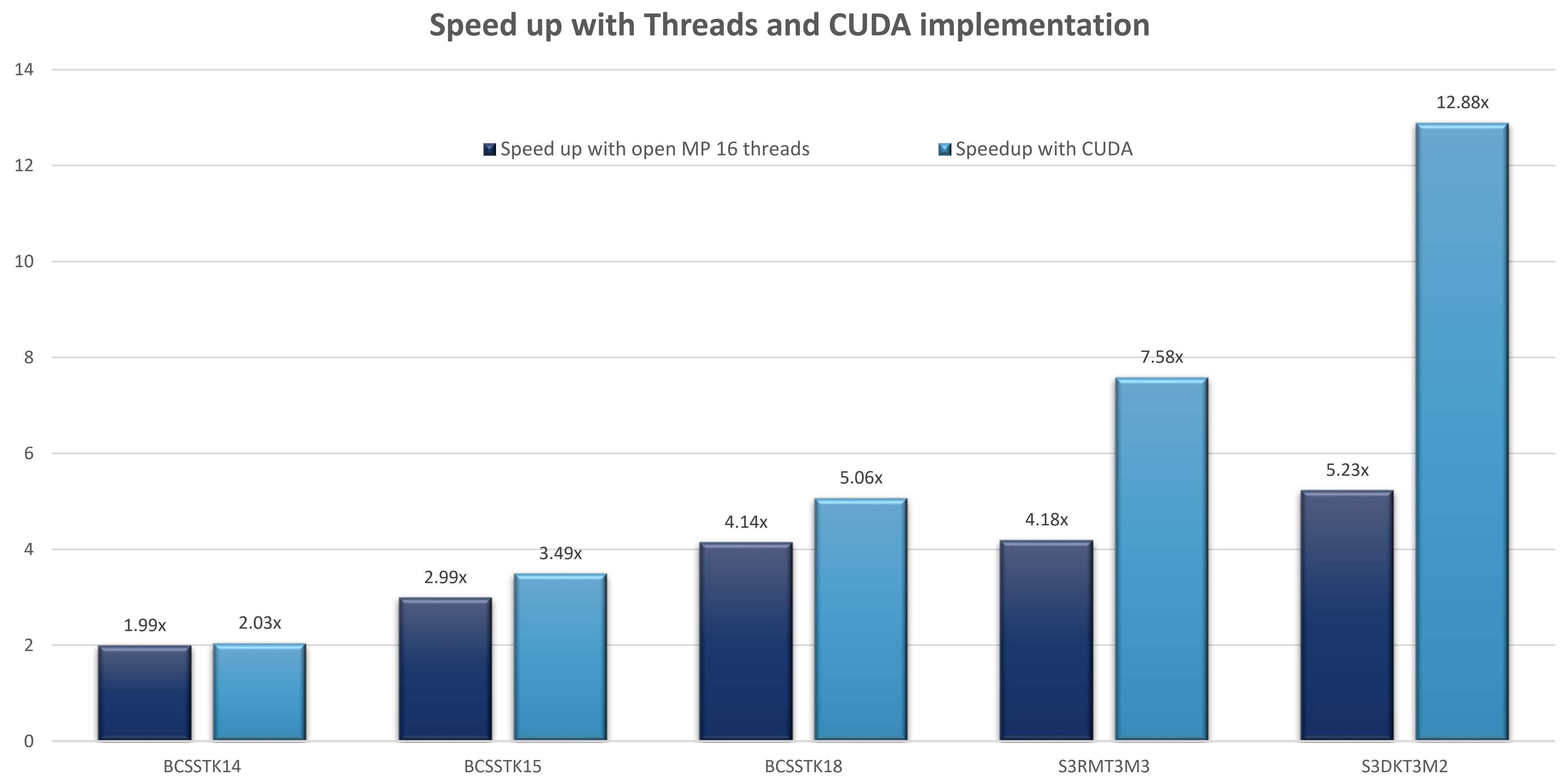
end repeat

Result is x_{k+1}

```
granularity: each sample hit covers 2 byte(s) for 0.00% of 568.23 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.04  568.19                 main [1]
              472.84    0.00  36858/36858        mat_vec_prod(crs*, double*, double*, int, int) [2]
               37.94    0.00       1/1           read_sparse_matrix(char const*) [3]
               16.45    0.00  55284/55284        scalar_vec_prod(double*, double, double*, int) [4]
               12.59    0.00  36856/36856        sum_vec(double*, double*, double*, int) [5]
               12.34    0.00  36856/36856        diff_vec(double*, double*, double*, int) [6]
               10.80    0.00  36857/36857        vecdot(double*, double*, int) [7]
                5.23    0.00  18616/18616        norm(double*, int) [8]
                0.00    0.00     186/186         std::common_type<std::chrono::duration<long, std::ratio<1
                0.00    0.00     186/186         std::chrono::duration<double, std::ratio<1l, 1l> >::durat
                0.00    0.00     186/372         std::chrono::duration<double, std::ratio<1l, 1l> >::count
                0.00    0.00       2/2           equate_vec(double*, double*, int) [25]
                0.00    0.00       1/1           get_w(double*, int) [28]
-----------------------------------------------
              472.84    0.00  36858/36858        main [1]
[2]     83.2  472.84    0.00  36858          mat_vec_prod(crs*, double*, double*, int, int) [2]
-----------------------------------------------
               37.94    0.00       1/1           main [1]
[3]      6.7   37.94    0.00       1         read_sparse_matrix(char const*) [3]
-----------------------------------------------
               16.45    0.00  55284/55284        main [1]
[4]      2.9   16.45    0.00  55284          scalar_vec_prod(double*, double, double*, int) [4]
-----------------------------------------------
               12.59    0.00  36856/36856        main [1]
[5]      2.2   12.59    0.00  36856          sum_vec(double*, double*, double*, int) [5]
-----------------------------------------------
               12.34    0.00  36856/36856        main [1]
[6]      2.2   12.34    0.00  36856          diff_vec(double*, double*, double*, int) [6]
-----------------------------------------------
               10.80    0.00  36857/36857        main [1]
[7]      1.9   10.80    0.00  36857          vecdot(double*, double*, int) [7]
-----------------------------------------------
                5.23    0.00  18616/18616        main [1]
[8]      0.9    5.23    0.00  18616          norm(double*, int) [8]
-----------------------------------------------
```

GPU CG RESULTED IN A HIGHER SPEEDUP THAN THE OPENMP MULTITHREADED CG



GHC Machines GPU Specs: GeForce GTX1080, 2560-cores, 8GB RAM

~13 times speedup!!

Launching kernel overhead mitigated by size of matrix i.e. computations increased

All tests were carried out on the GHC machines

FINAL THOUGHTS

- Compressed Row storage format is a memory-efficient way of storing sparse matrices
- Jacobi preconditioner works with diagonally-dominant, sparse matrices and sub-optimally with matrices that do not follow the banded structure
- Porting the computations to GPU is beneficial as the ratio of number of non-zero elements to the order of the matrix greater than 40 and the number of non-zero elements are above 200,000.
    - Beyond this number, the overhead of cudaMemcpy and kernel launch overhead is mitigated by the intensity of computations.

THANK YOU! ☺