Transient Stability of Power System

Programming Massively Parallel Graphics Multiprocessors using CUDA

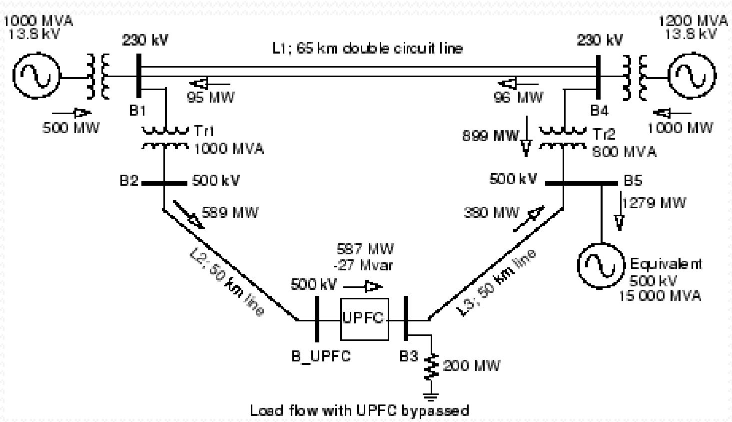
Final Project

Amirhassan Asgari Kamiabad

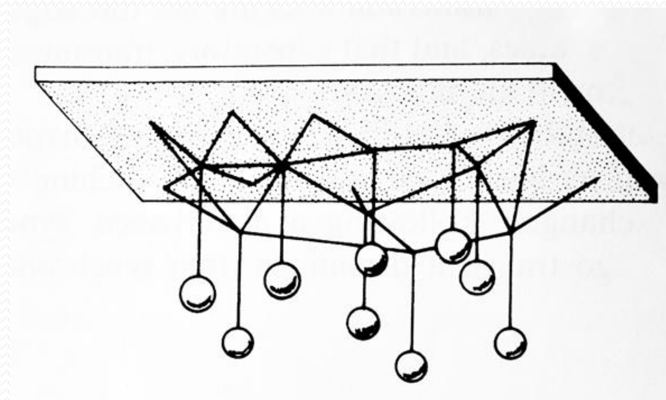
996620802

Electric Power System

- Largest infrastructure made by man
- Prone to many kinds of disturbance : Blackouts
- Transient stability analysis: dynamic behavior of power system few seconds following a disturbance



(Electric Power Network)
Voltage, Current, Power angel….



(String and Mass Network)
Force, Speed, Acceleration….

Transient Stability Computation

- Differential and Algebraic Equations

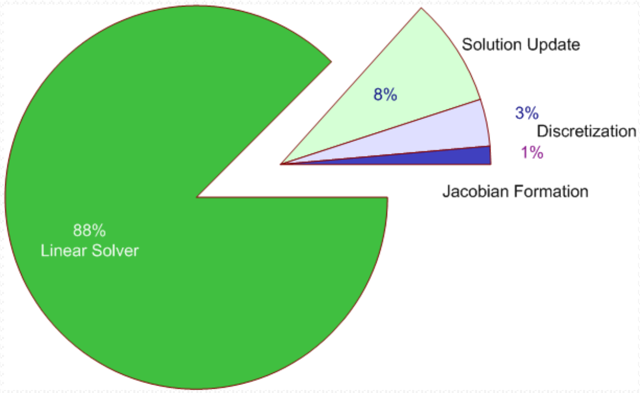
$$\frac{dx}{dt} = f(x, y)$$
$$0 = g(x, y)$$

- Discretize the differential part and use Backward Euler

$$F(y_{n+1}, x_{n+1}) = 0$$

- Newton Method solve nonlinear system
- Solve Linear System

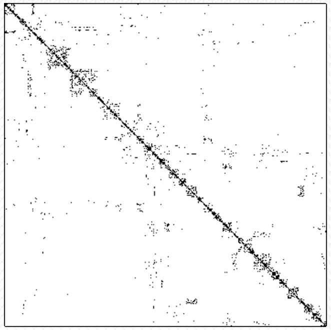
$$\underbrace{Y}_{A}\,\underbrace{e}_{x} = \underbrace{S}_{b}$$



Transient Stability computation Load

Jacobian Matrix Properties

- Large and Sparse: 10k* 10k with 50k non-zeros
- Stiff: Eigen values are widespread
- Save in sparse compressed row format
  - Data, indices and pointer vectors



$$\begin{array}{rc} \text{Column} & \begin{matrix} 0 & 1 & 2 & 3 \end{matrix} \\ & \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \end{array} \Rightarrow \begin{array}{ll} [0 \quad 1/ \quad 1 \quad 2/ \quad 0 \quad 2 \quad 3/ \quad 1 \quad 3] & \text{indices} \\ [1 \quad 7/ \quad 2 \quad 8/ \quad 5 \quad 3 \quad 9/ \quad 6 \quad 4] & \text{data} \\ [0 \quad 2 \quad 4 \quad 7 \quad 9] & \text{pointer} \end{array}$$

Direct: LU factorization

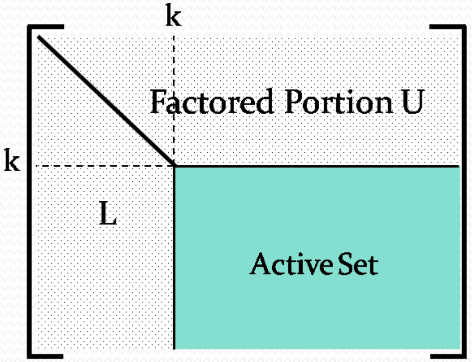
- LU factorization : Ax = (L U)x = L(Ux)=b

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

- Backward Substitution Ly = b
- Forward Substitution Ux=y

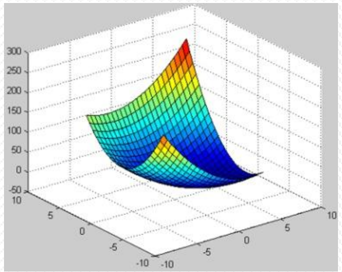
$$\begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

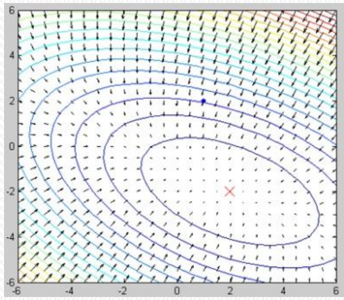
  - Number of active threads reduce during run time
  - Backward and forward substitution are serial algorithms



Indirect: Conjugate Gradient

- Iterative method which use only matrix multiplication and addition
- Necessarily converges in N iteration if the matrix is symmetric positive definite
- Suffer from slow rate of convergence
  - Transforming system to an equivalent with same solution but easier to solve
  - Preconditioning can improve robustness and efficiency



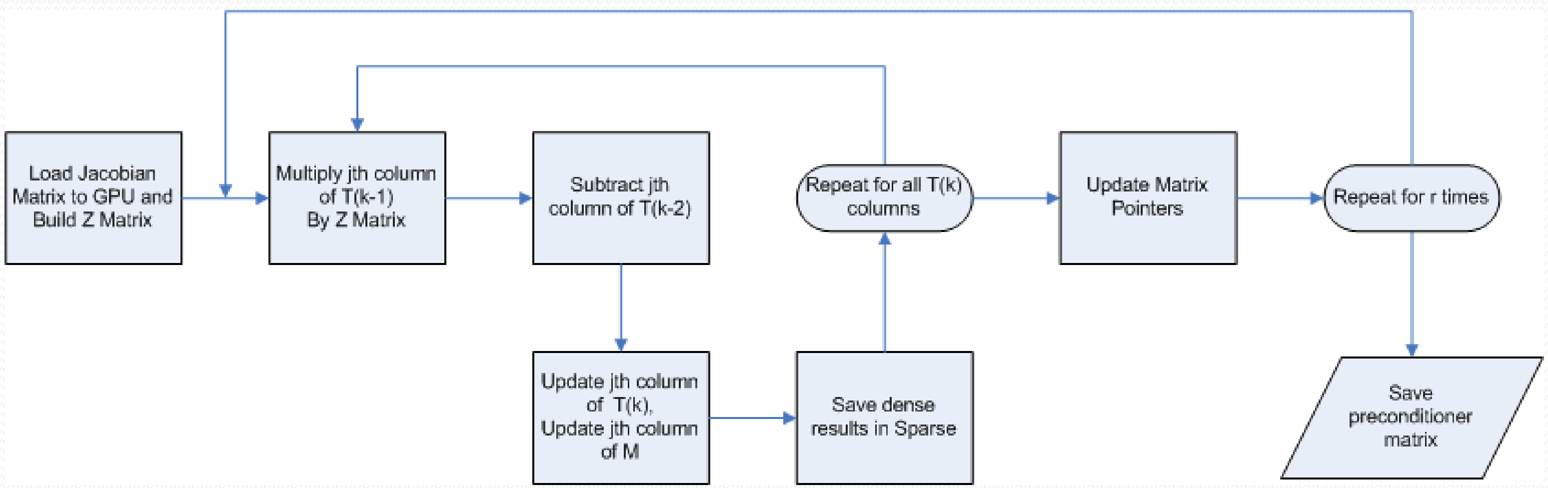


Chebychev Preconditioner

- Iterative method which estimate inverse of matrix
- Main GPU kernels:
  - Sparse Matrix-Matrix Multiply
  - Dense to Sparse transfer

$$A^{-1} = \frac{c_0}{2} I + \sum_{1}^{r} c_k T_k(Z)$$

$$T_k(Z) = 2Z(T_{k-1}(Z)) - T_k(Z)$$



Sparse Matrix-Matrix Multiplication

- One thread per row: Multiply each element in the row
  - for (int jj = row_start; jj < row_end; jj++)
- Find corresponding element:
  - while (kk < col_end) && (indice[jj] >= vec1_indice[kk])
- If found, perform the multiplication
  - if (indice[jj] == vec1_indice[kk])
  - sum += data[jj] * vec1_data[kk];

Thread Id = 2 :

Indices	[0	1/	1	2/	0	2	3/	1	3]
data	[1	7/	2	8/	5	3	9/	6	4]
pointer	[0	2	4	7	9]				

$$\begin{bmatrix} 1 \\ 4 \\ 8/ \\ 3 \\ 2/ \\ 9 \\ 6 \\ 7 \\ 2/ \\ 1 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \\ 3/ \\ 2 \\ 3/ \\ 0 \\ 1 \\ 2 \\ 3/ \\ 0 \end{bmatrix}$$

Sparse Matrix-Matrix Multiplication Code

```
const int row = (blockDim.x * blockIdx.x  + threadIdx.x);

  if(row < num_rows){
     float sum = 0;
     int dense_flag = 0;

     int row_start = ptr[row];
     int row_end   = ptr[row+1];
     int col_start = vec1_ptr[col_num];
     int col_end   = vec1_ptr[col_num+1];

     for (int jj = row_start; jj < row_end; jj++){
                    int kk =  col_start; int not_found = 1;
                    while( (kk < col_end) && (indice[jj] >= vec1_indice[kk]) && not_found){
                              if (indice[jj] == vec1_indice[kk]){
                    sum += data[jj] * vec1_data[kk];
                              not_found = 0;}
                    kk++;
                    }
          }
```

Dense to Sparse Conversion

- Save the result of multiplication in dense format:

			A					B				C	D									E	

- If thread holds zero, write zero, otherwise write 1:

0	0	0	1	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0

- Perform scan over the pointer vector:

0	0	0	1	1	1	1	1	2	2	2	2	3	4	4	4	4	4	4	4	4	4	5	5

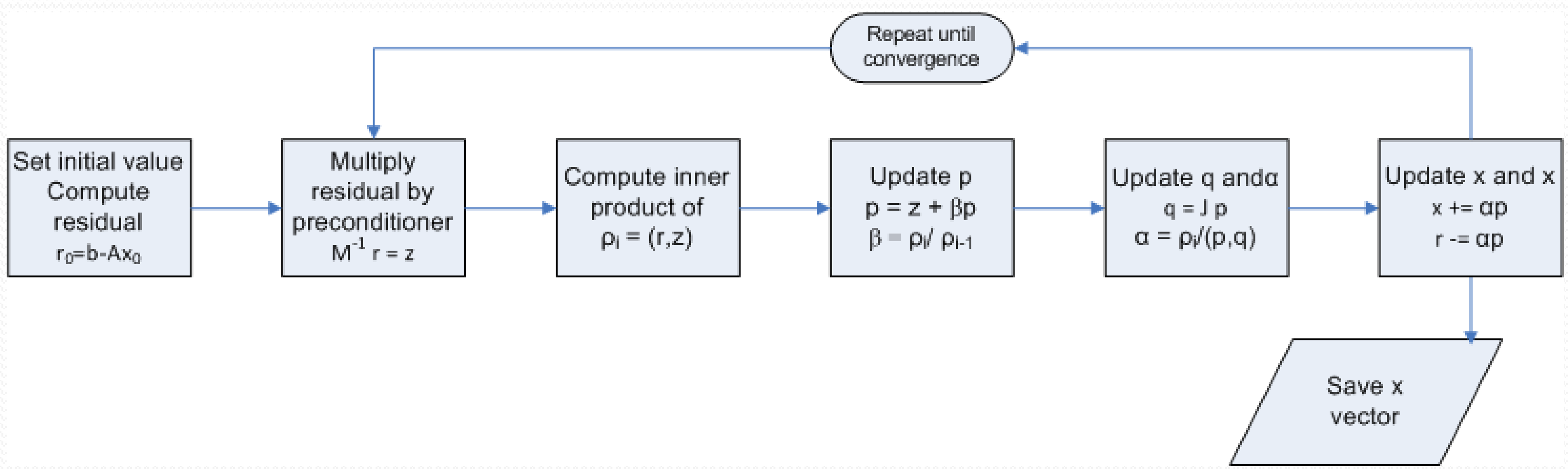
A	B	C	D	E

4	9	13	14	23

```
if (dense_ptr[tid] - dense_ptr[tid-1]){
   data[dense_ptr[tid] -1 + data_num] = dense_data[tid];
   indice[dense_ptr[tid] -1 + data_num] = tid;}
```

Conjugate Gradient Algorithm

- Main GPU Kernels:
  - Vector Norm
  - Vector inner product (a,b)
  - Update vectors



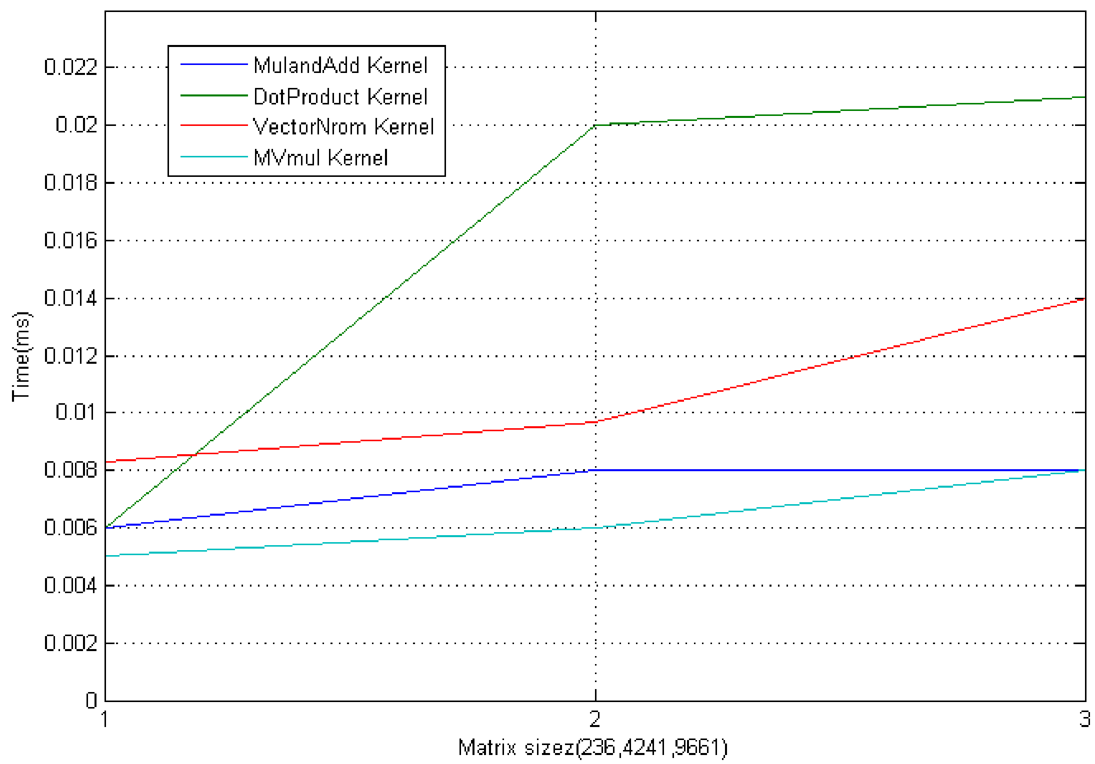
Vector Norm and Inner Product

- Vector norm adds square of all elements and output square root of result:
  - Load elements and multiply by itself
  - Perform scan and compute square root
- Inner product multiply corresponding elements of two vector and add up the results:
  - Load both vectors and multiply corresponding elements
  - Perform scan and report the results

Kernels Optimization:

- Optimizations on Matrix-Matrix vector multiply
  - Load both matrices in sparse
  - Copy vector in shared memory
  - Optimized search
  - Implement add in same kernel to avoid global memory
- Challenges:
  - Link lists and sparse formats
  - Varying matrix size in each kernel

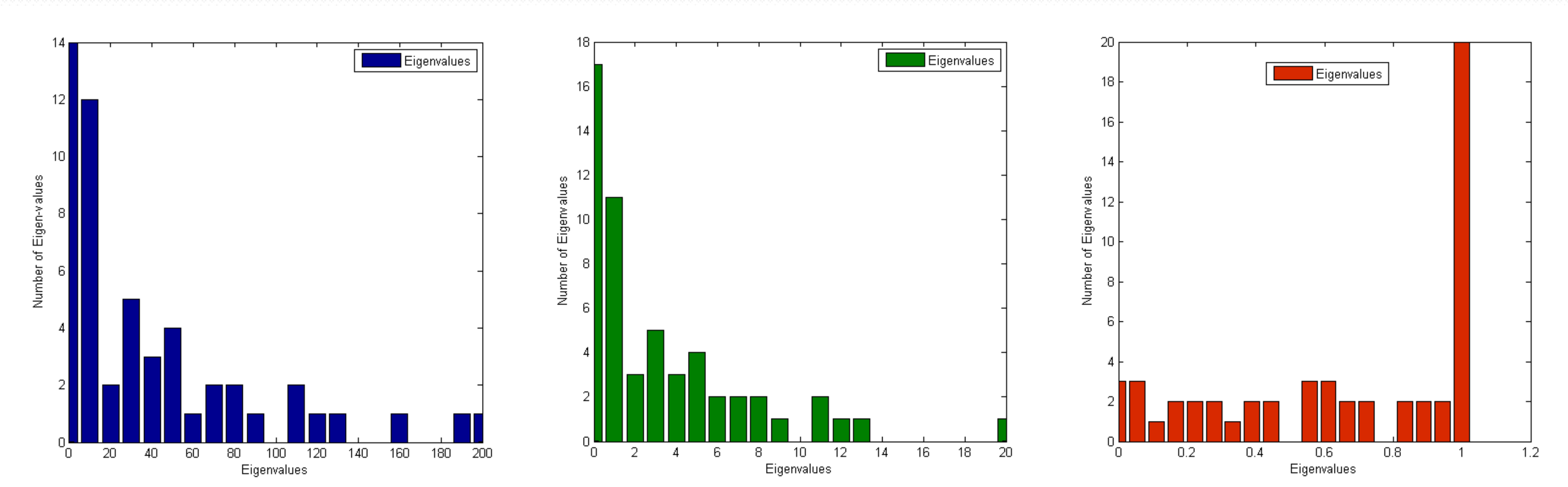
Kernels Evaluation:



Main kernels Execution Time

Preconditioner Efficiency

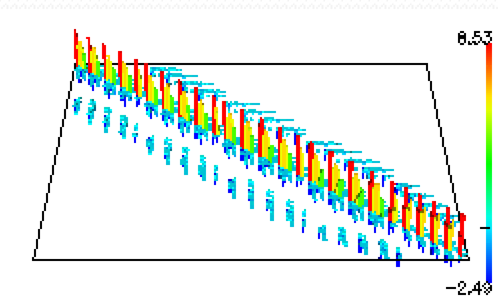
- Methodology:
  - 1-Effect on Eigen values spectrum
  - 2- Effect on number of conjugate gradient iterations

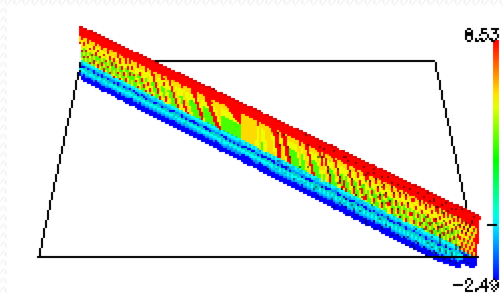


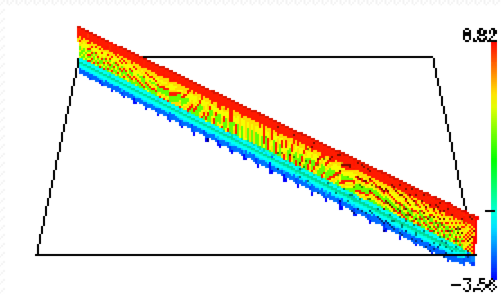
Preconditioning Effect on IEEE-57 Test System

CG Efficiancy

Matrix Name	Matrix Size	Total Nonzeros	Condition Number	Total Time Per Iteration (ms)
E05R	236	5856	6e+04	1.13
E20R	4241	131556	5045e+10	11.33
E30R	9661	306356	3.47e+11	45.7







Matrices Structural Plot(source : Matrix Market)

Discussion

- Challenges , problems and ideas
  - Work with all matrices: matrix market form
  - New kernel ideas
    - SpMMul
    - Dens2Sp
- Future work and updates
  - Other iterative methods: GMRES, BiCG

Thank you!