Implementation of IDR(s)-biortho Krylov Solver using CUDA parallelism

Isha Aggarwal, NLA4HPC 2020

Structure of this presentation…

- Background and theory
- Implementation overview
- Correctness and Numerical results

Background and Theory

Motivation: Solving Linear Systems

Ax = b

$A \in \mathbb{C}^{N \times N}$; A is large sparse nonsymmetric matrix, $\boldsymbol{b} \in \mathbb{C}^N$

Krylov solvers:

Bi-CGSTAB

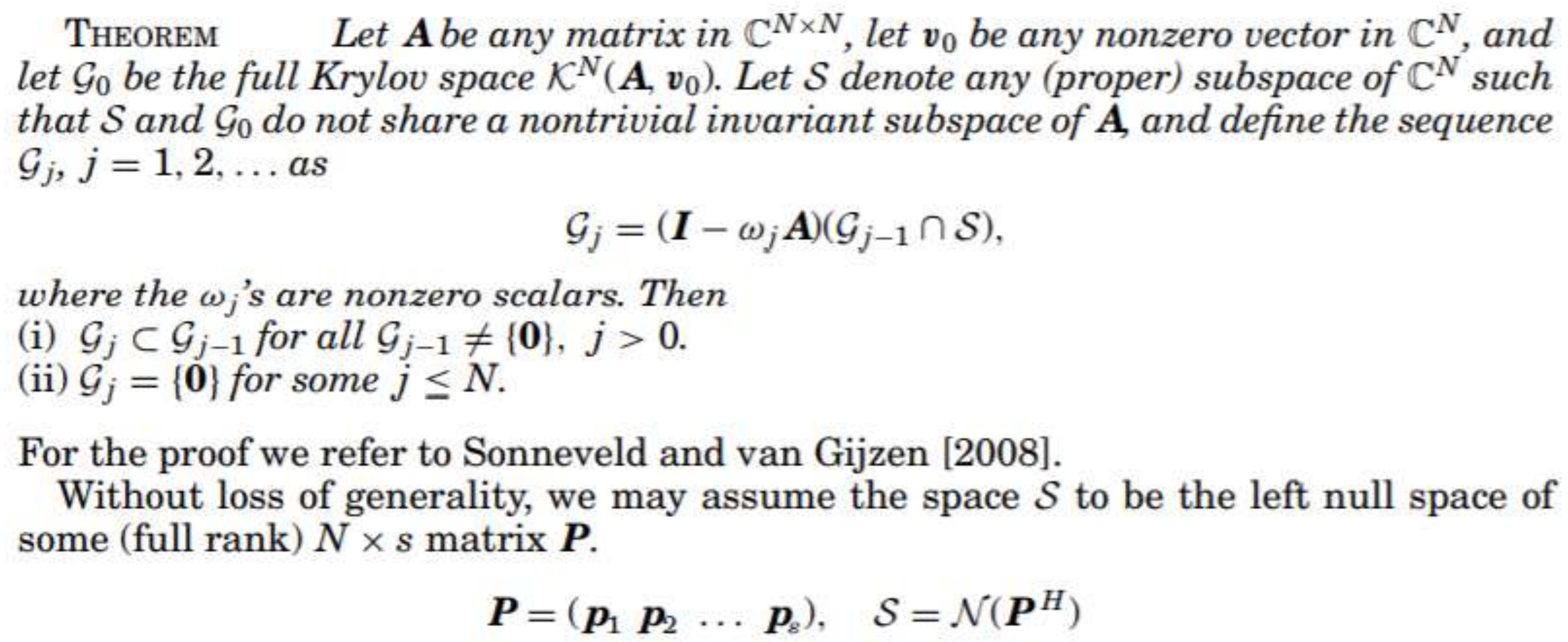
GMRES

BiCGstab(l)

IDR

IDR method was proposed in Wesseling and Sonneveld ,1980 ; it is a very powerful iterative method for solving such systems. It is based on the IDR theorem.

The IDR theorem



THEOREM *Let* $\mathbf{A}$ *be any matrix in* $\mathbb{C}^{N\times N}$, *let* $\boldsymbol{v}_0$ *be any nonzero vector in* $\mathbb{C}^N$, *and let* $\mathcal{G}_0$ *be the full Krylov space* $\mathcal{K}^N(\mathbf{A}, \boldsymbol{v}_0)$. *Let* $\mathcal{S}$ *denote any (proper) subspace of* $\mathbb{C}^N$ *such that* $\mathcal{S}$ *and* $\mathcal{G}_0$ *do not share a nontrivial invariant subspace of* $\mathbf{A}$, *and define the sequence* $\mathcal{G}_j$, $j = 1, 2, \ldots$ *as*

$$\mathcal{G}_j = (\boldsymbol{I} - \omega_j \mathbf{A})(\mathcal{G}_{j-1} \cap \mathcal{S}),$$

where the ω_j*'s are nonzero scalars. Then*
(i) $\mathcal{G}_j \subset \mathcal{G}_{j-1}$ *for all* $\mathcal{G}_{j-1} \neq \{\mathbf{0}\}$, $j > 0$.
(ii) $\mathcal{G}_j = \{\mathbf{0}\}$ *for some* $j \leq N$.

For the proof we refer to Sonneveld and van Gijzen [2008].

Without loss of generality, we may assume the space $\mathcal{S}$ to be the left null space of some (full rank) $N \times s$ matrix $\boldsymbol{P}$.

$$\boldsymbol{P} = (\boldsymbol{p}_1\ \boldsymbol{p}_2\ \ldots\ \boldsymbol{p}_s), \quad \mathcal{S} = \mathcal{N}(\boldsymbol{P}^H)$$

IDR(s) method

Generates residuals that are forced to be in subspaces $\mathcal{G}_j$ of decreasing dimension. These nested subspaces are related by

$$\mathcal{G}_j = (\boldsymbol{I} - \omega_j \mathbf{A})(\mathcal{S} \cap \mathcal{G}_{j-1})$$

where the ω_j's are nonzero scalars and S is a fixed proper subspace of $\mathbb{C}^N . \omega_j$

This IDR approach is quite general and can be used as a framework for deriving iterative methods.

IDR(s) algorithmic variants

IDR(s)-proto (Sonneveld and van Gijzen 2008) is a direct translation of IDR theorem into algorithm.

IDR(s)-biortho (2011) is another variant.

An attractive choice as :

-- lower overhead in vector operations than the original IDR(s) algorithms.

--More stable and more accurate than original IDR(s) algorithm.

--Outperforms other state-of-the-art methods.

IDR(s) algorithmic variants

- Residual is uniquely defined at every (s+1)th step. This stepcorresponds to calculation of first residual in G_j
- In order to advance to G_{j+1}, s additional residuals in G_j need to be computed. These intermediate residuals are not uniquely defined and their computation leaves freedom to derive algorithmic variants.
- In exact arithmetic, the residuals at every (s+1)th step do not depend on the way intermediate residuals are computed.
- The numerical stability and efficiency of the specific IDR algorithm, however do depend on the computation of intermediate residuals.

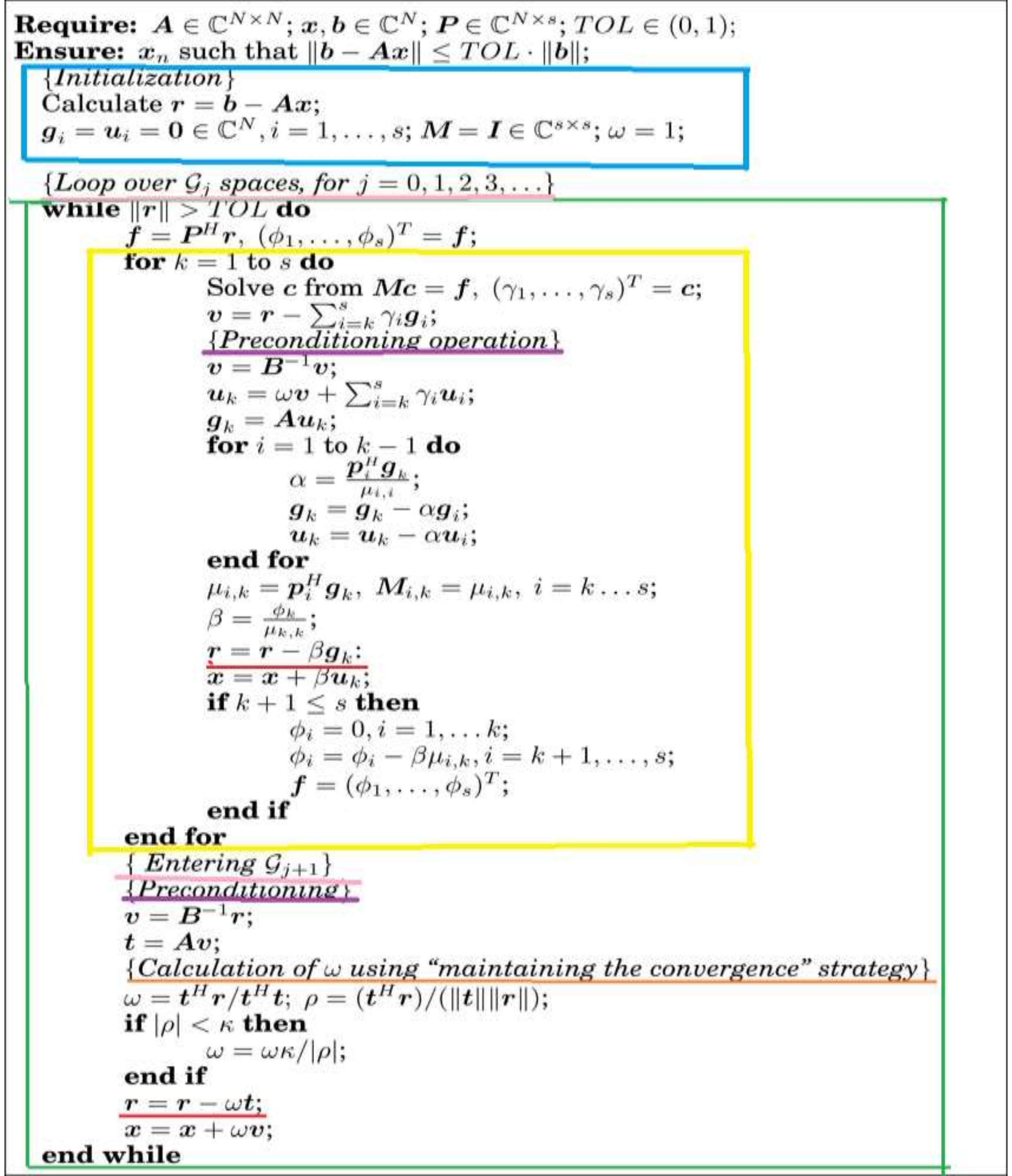
IDR(s)-biortho

- Numerically very stable IDR based method
- Fills in the freedom in generating the intermediate residuals by imposing one-sided biorthogonality conditions between the intermediate residuals and the vectors $p_1, p_2, \ldots, p_s$.
- Fills in freedom by constructing vectors that satisfy the following biorthogonality conditions:

$$\boldsymbol{g}_{n+k} \perp \boldsymbol{p}_i, \ i = 1, \ldots k-1, \ k = 2, \ldots, s,$$

$$\boldsymbol{r}_{n+k+1} \perp \boldsymbol{p}_i, \ i = 1, \ldots, k, \ k = 1, \ldots, s.$$

```
Require: $A \in \mathbb{C}^{N \times N}$; $x, b \in \mathbb{C}^N$; $P \in \mathbb{C}^{N \times s}$; $TOL \in (0,1)$;
Ensure: $x_n$ such that $\|b - Ax\| \le TOL \cdot \|b\|$;
  {Initialization}
  Calculate $r = b - Ax$;
  $g_i = u_i = 0 \in \mathbb{C}^N, i = 1, \ldots, s$; $M = I \in \mathbb{C}^{s \times s}$; $\omega = 1$;

  {Loop over $\mathcal{G}_j$ spaces, for $j = 0, 1, 2, 3, \ldots$}
  while $\|r\| > TOL$ do
        $f = P^H r,\ (\phi_1, \ldots, \phi_s)^T = f$;
        for $k = 1$ to $s$ do
              Solve $c$ from $Mc = f,\ (\gamma_1, \ldots, \gamma_s)^T = c$;
              $v = r - \sum_{i=k}^{s} \gamma_i g_i$;
              {Preconditioning operation}
              $v = B^{-1} v$;
              $u_k = \omega v + \sum_{i=k}^{s} \gamma_i u_i$;
              $g_k = A u_k$;
              for $i = 1$ to $k - 1$ do
                    $\alpha = \frac{p_i^H g_k}{\mu_{i,i}}$;
                    $g_k = g_k - \alpha g_i$;
                    $u_k = u_k - \alpha u_i$;
              end for
              $\mu_{i,k} = p_i^H g_k,\ M_{i,k} = \mu_{i,k},\ i = k \ldots s$;
              $\beta = \frac{\phi_k}{\mu_{k,k}}$;
              $r = r - \beta g_k$;
              $x = x + \beta u_k$;
              if $k + 1 \le s$ then
                    $\phi_i = 0, i = 1, \ldots k$;
                    $\phi_i = \phi_i - \beta \mu_{i,k}, i = k + 1, \ldots, s$;
                    $f = (\phi_1, \ldots, \phi_s)^T$;
              end if
        end for
        { Entering $\mathcal{G}_{j+1}$}
        {Preconditioning}
        $v = B^{-1} r$;
        $t = Av$;
        {Calculation of $\omega$ using “maintaining the convergence” strategy}
        $\omega = t^H r / t^H t$; $\rho = (t^H r)/(\|t\|\|r\|)$;
        if $|\rho| < \kappa$ then
              $\omega = \omega\kappa/|\rho|$;
        end if
        $r = r - \omega t$;
        $x = x + \omega v$;
  end while
```



```
**Require:** $A \in \mathbb{C}^{N\times N}$; $x, b \in \mathbb{C}^N$; $P \in \mathbb{C}^{N\times s}$; $TOL \in (0,1)$;
**Ensure:** $x_n$ such that $\|b - Ax\| \le TOL \cdot \|b\|$;
  {Initialization}
  Calculate $r = b - Ax$;
  $g_i = u_i = 0 \in \mathbb{C}^N, i = 1, \ldots, s$; $M = I \in \mathbb{C}^{s\times s}$; $\omega = 1$;
  {Loop over $\mathcal{G}_j$ spaces, for $j = 0, 1, 2, 3, \ldots$}
  while $\|r\| > TOL$ do
      $f = P^H r$, $(\phi_1, \ldots, \phi_s)^T = f$;
      for $k = 1$ to $s$ do
          Solve $c$ from $Mc = f$, $(\gamma_1, \ldots, \gamma_s)^T = c$;
          $v = r - \sum_{i=k}^{s} \gamma_i g_i$;
          {Preconditioning operation}
          $v = B^{-1} v$;
          $u_k = \omega v + \sum_{i=k}^{s} \gamma_i u_i$;
          $g_k = A u_k$;
          for $i = 1$ to $k - 1$ do
              $\alpha = \frac{p_i^H g_k}{\mu_{i,i}}$;
              $g_k = g_k - \alpha g_i$;
              $u_k = u_k - \alpha u_i$;
          end for
          $\mu_{i,k} = p_i^H g_k$, $M_{i,k} = \mu_{i,k}$, $i = k \ldots s$;
          $\beta = \frac{\phi_k}{\mu_{k,k}}$;
          $r = r - \beta g_k$;
          $x = x + \beta u_k$;
          if $k + 1 \le s$ then
              $\phi_i = 0, i = 1, \ldots k$;
              $\phi_i = \phi_i - \beta \mu_{i,k}, i = k+1, \ldots, s$;
              $f = (\phi_1, \ldots, \phi_s)^T$;
          end if
      end for
      { Entering $\mathcal{G}_{j+1}$}
      {Preconditioning}
      $v = B^{-1} r$;
      $t = Av$;
      {Calculation of $\omega$ using "maintaining the convergence" strategy}
      $\omega = t^H r / t^H t$; $\rho = (t^H r)/(\|t\|\|r\|)$;
      if $|\rho| < \kappa$ then
          $\omega = \omega\kappa/|\rho|$;
      end if
      $r = r - \omega t$;
      $x = x + \omega v$;
  end while
```

IDR(s)-biortho algorithm

- Dimension reduction step : Computing First residual in G_{j+1}
- Computing additional vectors in G_{j+1}. (exploit biorthogonality conditions on g_{n+k} and r_{n+k+1}).
- Apply implicit (right)preconditioning
- Computation of scalar ω_{j+1} : In calculation of first residual in G_{j+1}, we may choose ω_{j+1} freely, but the same value must be used in calculations of subsequent residuals in G_{j+1}. Different strategies are there to compute this scalar. Computed according the convergence strategy.-explain shortly
- Can include smoothing operation.

Choice for $\boldsymbol{\omega_{j+1}}$.

- Natural choice for ω_{j+1} is the value that minimizes the norm of r_{n+1} similarly as is done in, amongst others, the Bi-CGSTAB algorithm.
- Minimizing $||r_{n+1}||_2$ yields

$$\omega_{j+1} = \frac{(\mathbf{A}\boldsymbol{v}_n)^H \boldsymbol{v}_n}{(\mathbf{A}\boldsymbol{v}_n)^H \mathbf{A}\boldsymbol{v}_n}.$$

- An improvement of this choice is: Instead of using a pure minimal residual step, increase the value of ω if the cosine of angle between Av_n and v_n is smaller than a threshold κ. This means that ω is increased if these vectors are too close to being perpendicular. 0.7 is a suitable value for κ as given in the research paper.

```
$\omega_{j+1} = \frac{(\mathbf{A}\boldsymbol{v}_n)^H \boldsymbol{v}_n}{(\mathbf{A}\boldsymbol{v}_n)^H \mathbf{A}\boldsymbol{v}_n}$,    $\rho = \frac{(\mathbf{A}\boldsymbol{v}_n)^H \boldsymbol{v}_n}{\|\mathbf{A}\boldsymbol{v}_n\| \|\boldsymbol{v}_n\|}$
IF $|\rho| < \kappa$
    $\omega_{j+1} = \omega_{j+1}\kappa / |\rho|$
ENDIF
```

Computation of ω_{j+1} using the above strategy greatly improves the convergence rate.

Number of operations…

- This is quite efficient in terms of vector operations and even more efficient than IDR(s)-proto, despite the additional orthogonalization operations.

- For a full cycle of s+1 IDR(s) steps (smoothing disabled), we get:

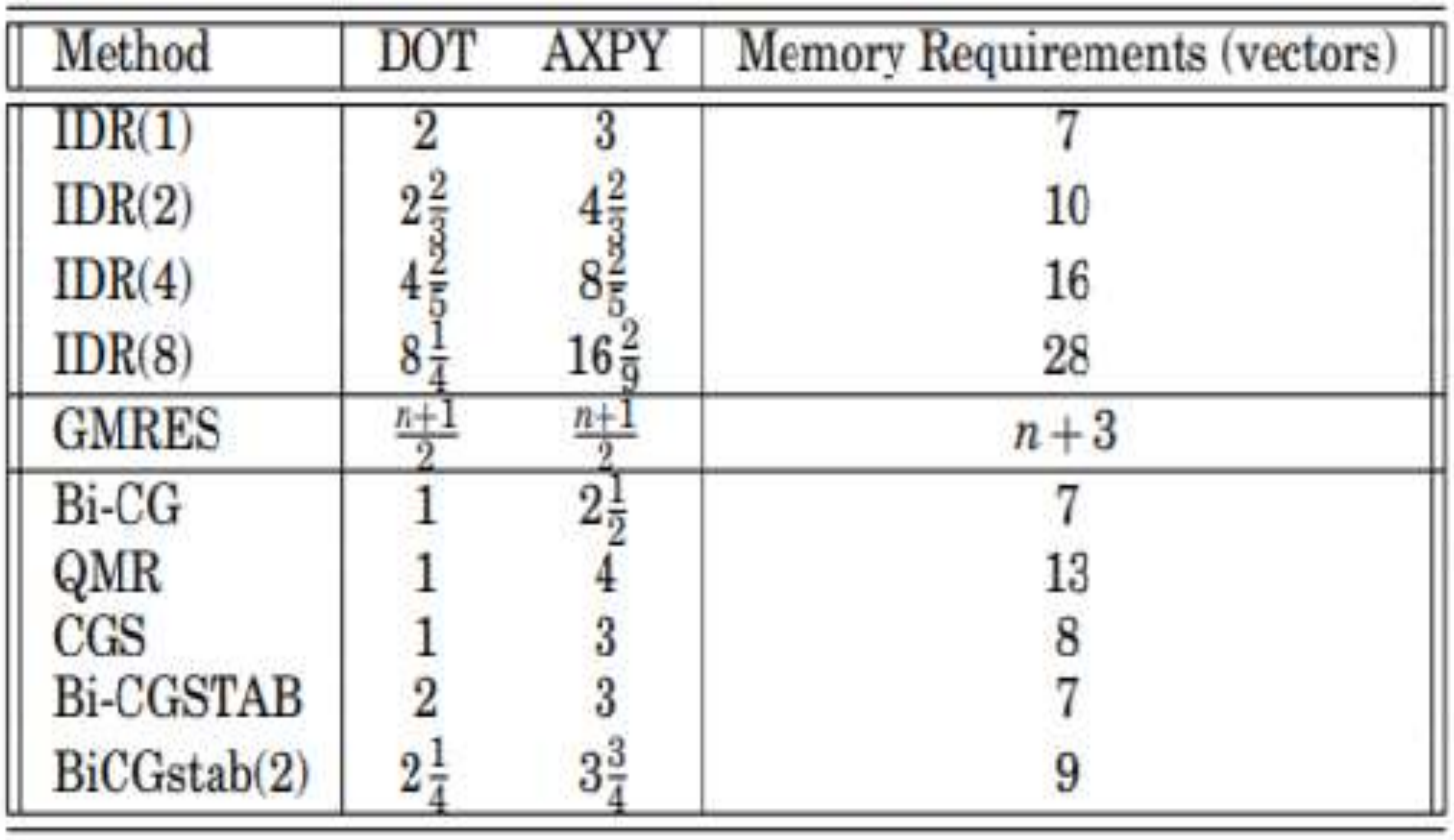
  s+1 preconditioned matrix vector products

  $s^2 + s + 2$ inner products

  $2s^2 + 2s + 2$ vector updates

IDR(s)-biortho requires same number of inner products and preconditioned matrix vector multiplications as by IDR(s)-proto. However, number of vector updates required is less. Original IDR(s) requires $2s^2 + 7/2s + 5/2$ vector-updates.

Memory requirements and Vector operations per Preconditioned Matrix-Vector product



Method	DOT	AXPY	Memory Requirements (vectors)
IDR(1)	2	3	7
IDR(2)	$2\frac{2}{3}$	$4\frac{2}{3}$	10
IDR(4)	$4\frac{2}{5}$	$8\frac{2}{5}$	16
IDR(8)	$8\frac{1}{4}$	$16\frac{2}{9}$	28
GMRES	$\frac{n+1}{2}$	$\frac{n+1}{2}$	$n+3$
Bi-CG	1	$2\frac{1}{2}$	7
QMR	1	4	13
CGS	1	3	8
Bi-CGSTAB	2	3	7
BiCGstab(2)	$2\frac{1}{4}$	$3\frac{3}{4}$	9

This table gives an overview of the number of vector operations per preconditioned matrix-vector multiplication for some values of s for IDR(s)-biortho (smoothing disabled), and for comparison also for the other Krylov methods. It also gives the memory requirements(excluding storage of the system matrix and of preconditioner, but including storage of the RHS vector and the solution).

Overview of Implementation
and
a quick demo

Implementation…

- C++
- CUDA for GPU kernels
- Doxygen for documentation

Datatype: cuDoubleComplex

Storing matrices (dense,csr,coo):

Stored as objects of classes of their respective types.(All dense matrices are stored in column major order)

Matrix objects contain pointers to arrays on allocated on

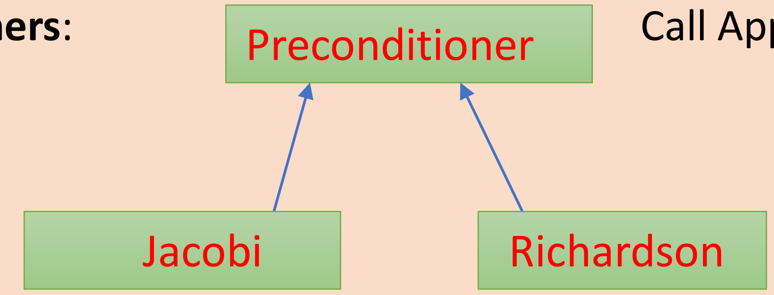
GPU and CPU representing the matrix structure.

Dense Matrix

CSR Matrix

COO Matrix

Preconditioners:



Call Apply Preconditioner polymorphically

Solver: Solver Attributes: atol, rtol, max_iter and final results etc... PIDR : is a member function

Wrote GPU kernels to parallelize operations such as:

- Inner product and Norm
- SpMV
- GeMV
- Computing Residual
- Mutiplying matrix hermitian with a vector

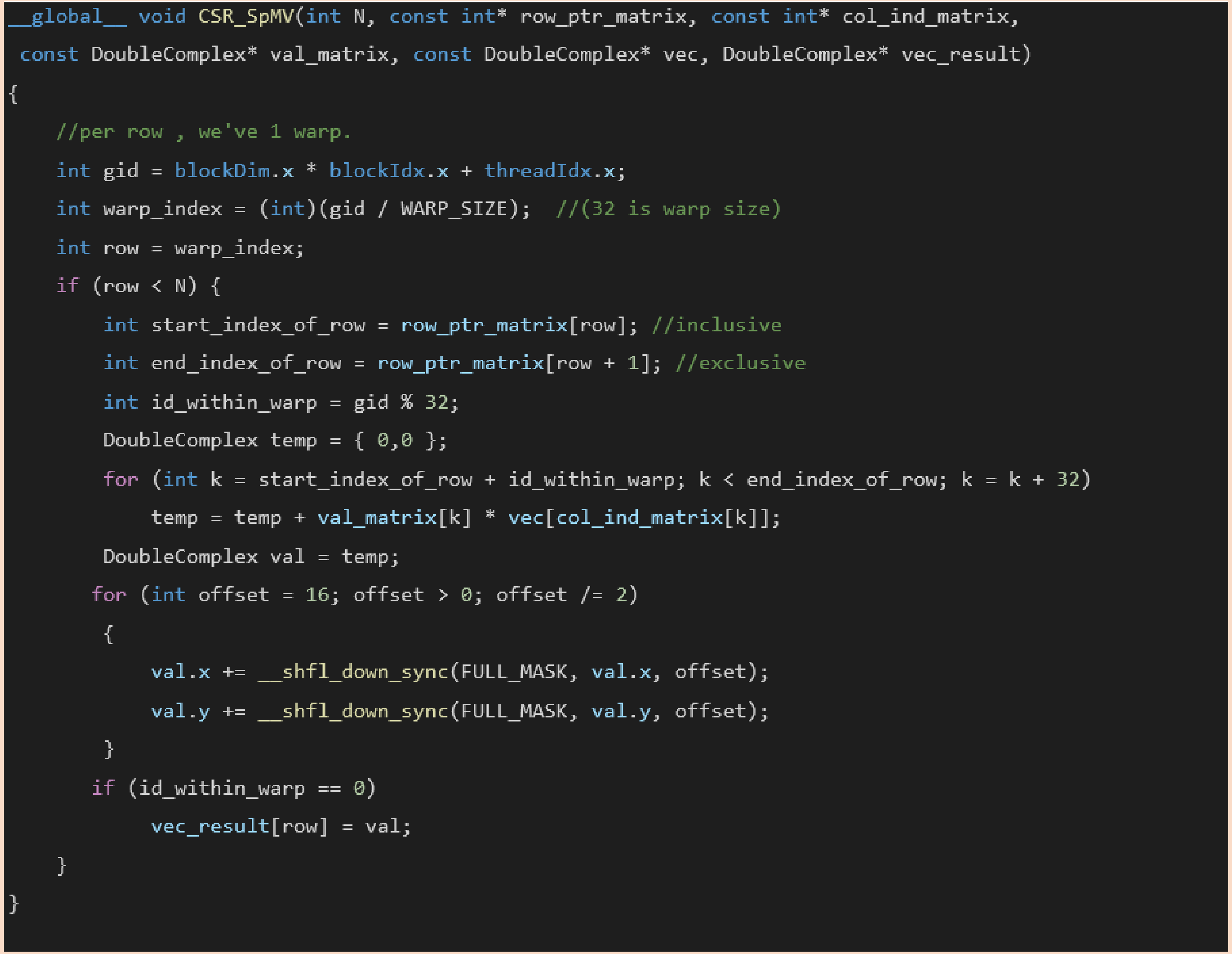
SpMV

One warp per row to ensure coalesced memory access

Warp level reduction in kernels such as SpMV , Computing residual :

__shfl_down_sync

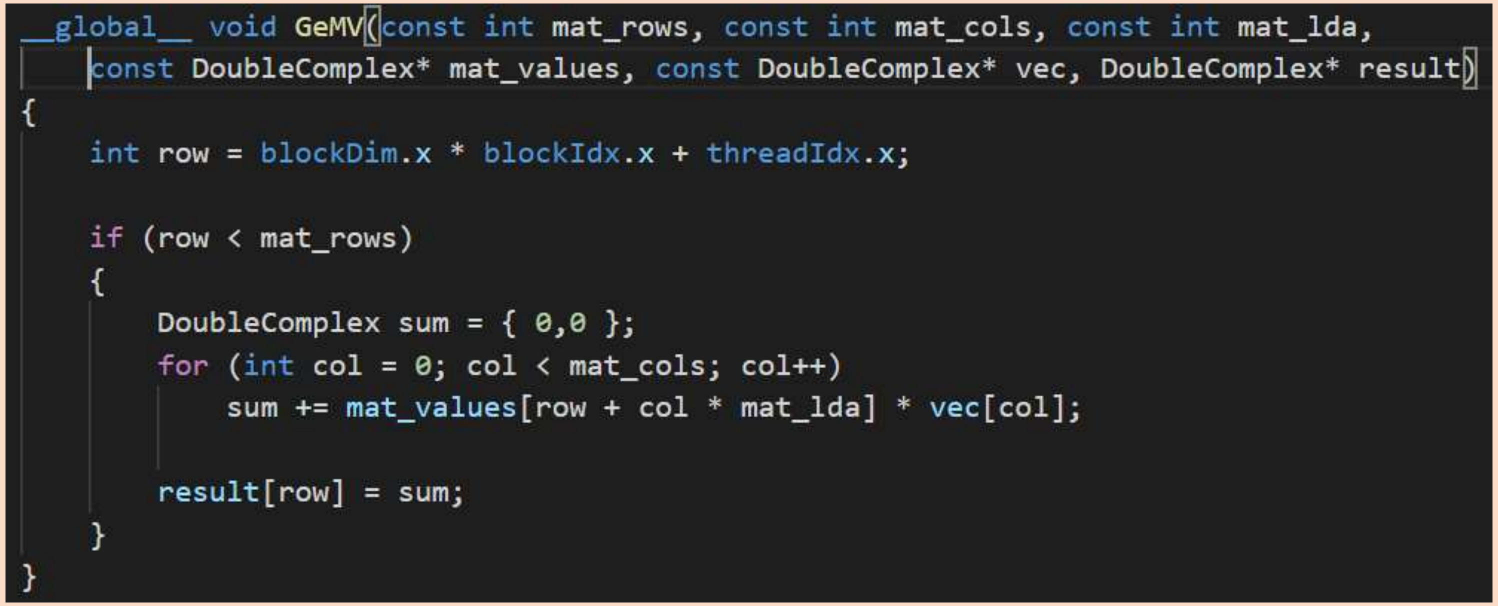
Used similar concepts in kernels such as (Hermitian transpose of dense matrix)*(vector) etc.



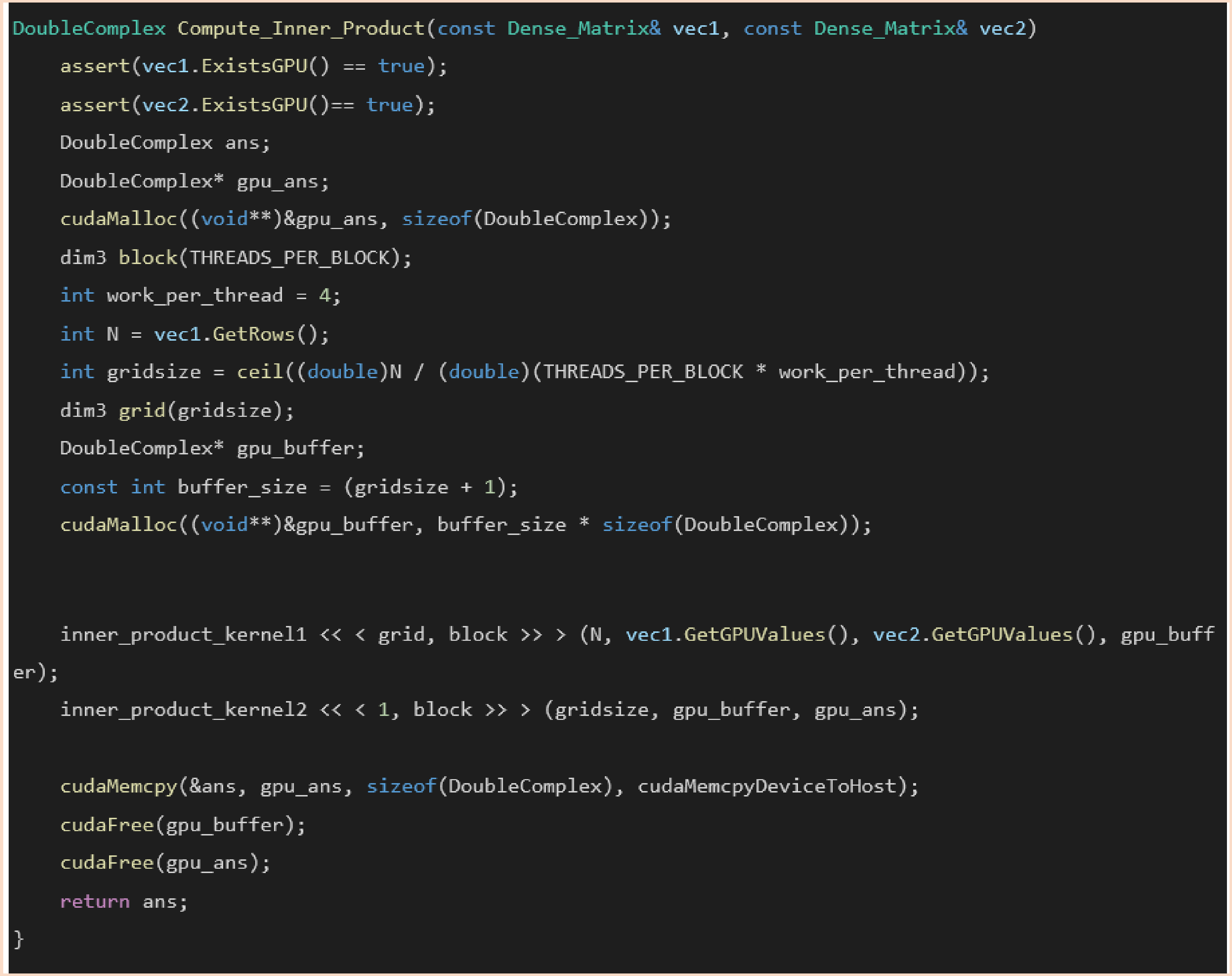
```
__global__ void CSR_SpMV(int N, const int* row_ptr_matrix, const int* col_ind_matrix,
 const DoubleComplex* val_matrix, const DoubleComplex* vec, DoubleComplex* vec_result)
{
    //per row , we've 1 warp.
    int gid = blockDim.x * blockIdx.x + threadIdx.x;
    int warp_index = (int)(gid / WARP_SIZE);  //(32 is warp size)
    int row = warp_index;
    if (row < N) {
        int start_index_of_row = row_ptr_matrix[row]; //inclusive
        int end_index_of_row = row_ptr_matrix[row + 1]; //exclusive
        int id_within_warp = gid % 32;
        DoubleComplex temp = { 0,0 };
        for (int k = start_index_of_row + id_within_warp; k < end_index_of_row; k = k + 32)
            temp = temp + val_matrix[k] * vec[col_ind_matrix[k]];
        DoubleComplex val = temp;
       for (int offset = 16; offset > 0; offset /= 2)
        {
            val.x += __shfl_down_sync(FULL_MASK, val.x, offset);
            val.y += __shfl_down_sync(FULL_MASK, val.y, offset);
        }
       if (id_within_warp == 0)
            vec_result[row] = val;
    }
}
```

GeMV

One thread per row of dense matrix; coalesced memory access



```
__global__ void GeMV(const int mat_rows, const int mat_cols, const int mat_lda,
    const DoubleComplex* mat_values, const DoubleComplex* vec, DoubleComplex* result)
{
    int row = blockDim.x * blockIdx.x + threadIdx.x;

    if (row < mat_rows)
    {
        DoubleComplex sum = { 0,0 };
        for (int col = 0; col < mat_cols; col++)
            sum += mat_values[row + col * mat_lda] * vec[col];

        result[row] = sum;
    }
}
```

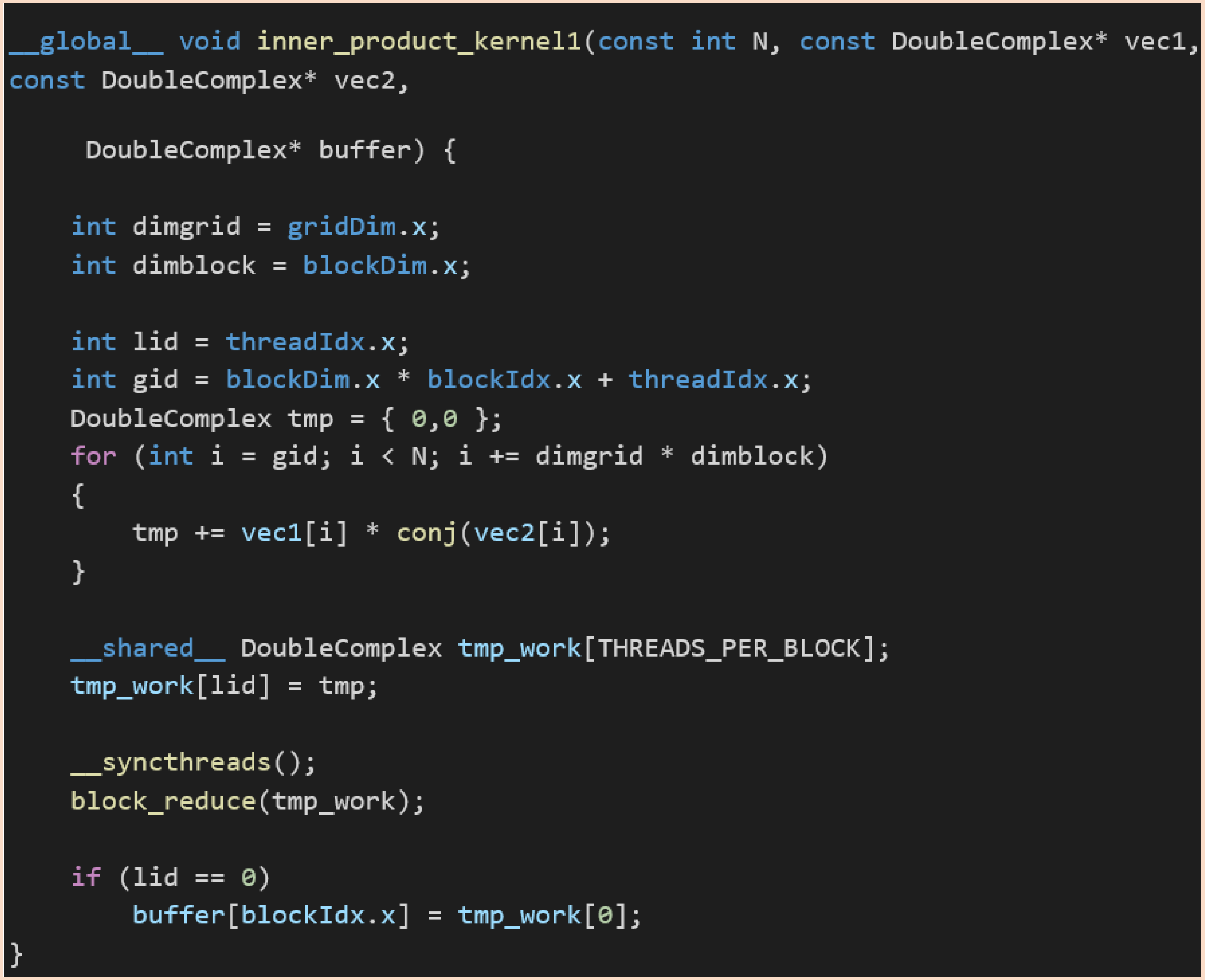


```
DoubleComplex Compute_Inner_Product(const Dense_Matrix& vec1, const Dense_Matrix& vec2)
    assert(vec1.ExistsGPU() == true);
    assert(vec2.ExistsGPU()== true);
    DoubleComplex ans;
    DoubleComplex* gpu_ans;
    cudaMalloc((void**)&gpu_ans, sizeof(DoubleComplex));
    dim3 block(THREADS_PER_BLOCK);
    int work_per_thread = 4;
    int N = vec1.GetRows();
    int gridsize = ceil((double)N / (double)(THREADS_PER_BLOCK * work_per_thread));
    dim3 grid(gridsize);
    DoubleComplex* gpu_buffer;
    const int buffer_size = (gridsize + 1);
    cudaMalloc((void**)&gpu_buffer, buffer_size * sizeof(DoubleComplex));


    inner_product_kernel1 << < grid, block >> > (N, vec1.GetGPUValues(), vec2.GetGPUValues(), gpu_buff
er);
    inner_product_kernel2 << < 1, block >> > (gridsize, gpu_buffer, gpu_ans);

    cudaMemcpy(&ans, gpu_ans, sizeof(DoubleComplex), cudaMemcpyDeviceToHost);
    cudaFree(gpu_buffer);
    cudaFree(gpu_ans);
    return ans;
}
```

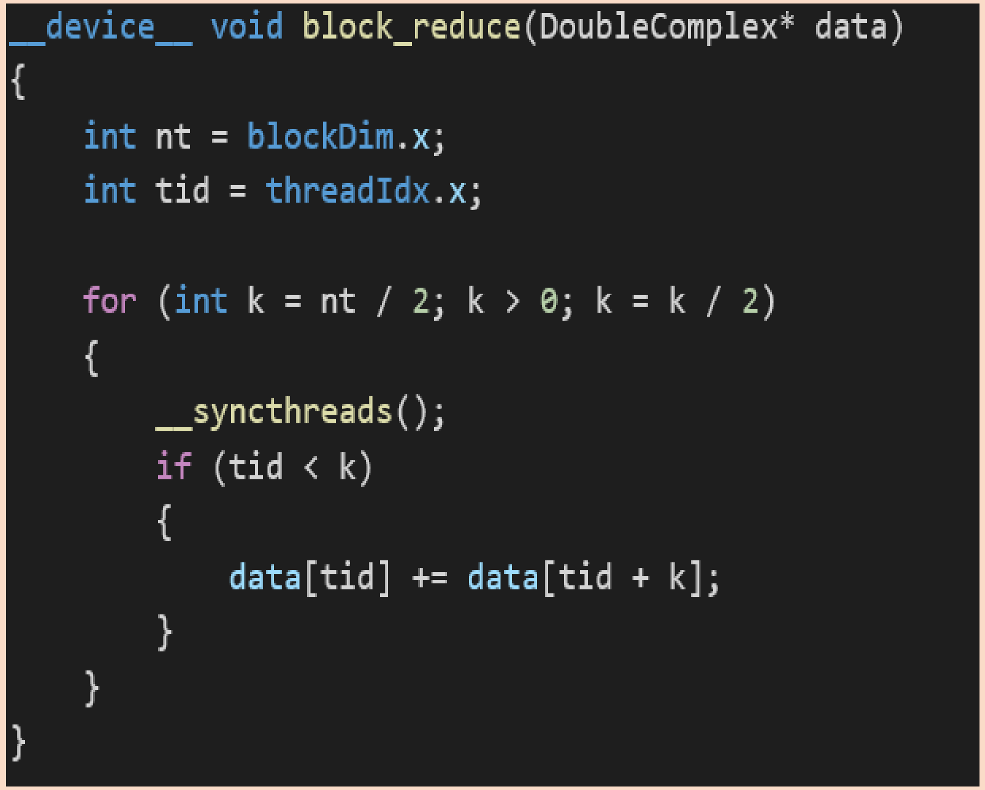
Inner Product

<vec1,vec2> = (transpose of vec1)(conjugate of vec2)

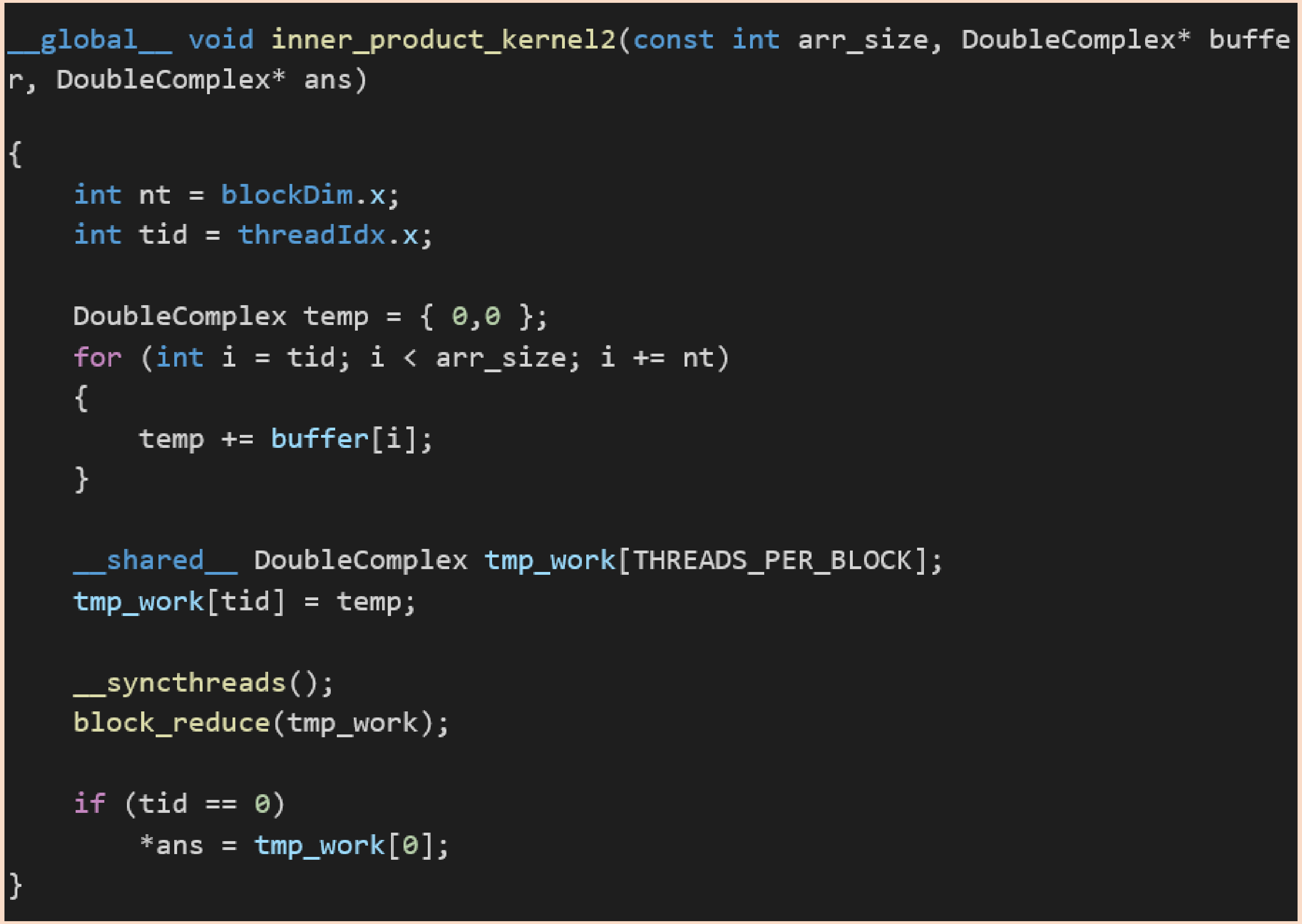
Used parallel reduction and shared memory



```
__global__ void inner_product_kernel1(const int N, const DoubleComplex* vec1,
const DoubleComplex* vec2,

     DoubleComplex* buffer) {

    int dimgrid = gridDim.x;
    int dimblock = blockDim.x;

    int lid = threadIdx.x;
    int gid = blockDim.x * blockIdx.x + threadIdx.x;
    DoubleComplex tmp = { 0,0 };
    for (int i = gid; i < N; i += dimgrid * dimblock)
    {
        tmp += vec1[i] * conj(vec2[i]);
    }

    __shared__ DoubleComplex tmp_work[THREADS_PER_BLOCK];
    tmp_work[lid] = tmp;

    __syncthreads();
    block_reduce(tmp_work);

    if (lid == 0)
        buffer[blockIdx.x] = tmp_work[0];
}
```



```
__device__ void block_reduce(DoubleComplex* data)
{
    int nt = blockDim.x;
    int tid = threadIdx.x;

    for (int k = nt / 2; k > 0; k = k / 2)
    {
        __syncthreads();
        if (tid < k)
        {
            data[tid] += data[tid + k];
        }
    }
}
```



```
__global__ void inner_product_kernel2(const int arr_size, DoubleComplex* buffe
r, DoubleComplex* ans)

{
    int nt = blockDim.x;
    int tid = threadIdx.x;

    DoubleComplex temp = { 0,0 };
    for (int i = tid; i < arr_size; i += nt)
    {
        temp += buffer[i];
    }

    __shared__ DoubleComplex tmp_work[THREADS_PER_BLOCK];
    tmp_work[tid] = temp;

    __syncthreads();
    block_reduce(tmp_work);

    if (tid == 0)
        *ans = tmp_work[0];
}
```

- Tried to use algorithm specific kernels to reduce the number of global memory accesses.

  For instance, y = beta*y + alpha*x

  Instead of:

  ```
  scale(y,beta)    read y,write y
  axpy(y, alpha,x)  read x,read y , write y
  ```

  Used:

  ```
  Vector_Linear_Combination(result,scalar1,vec1,scalar2,vec2)
  read x,read y,write y (Saved 1 read and 1 write)
  ```

Quick demo…

- Input:

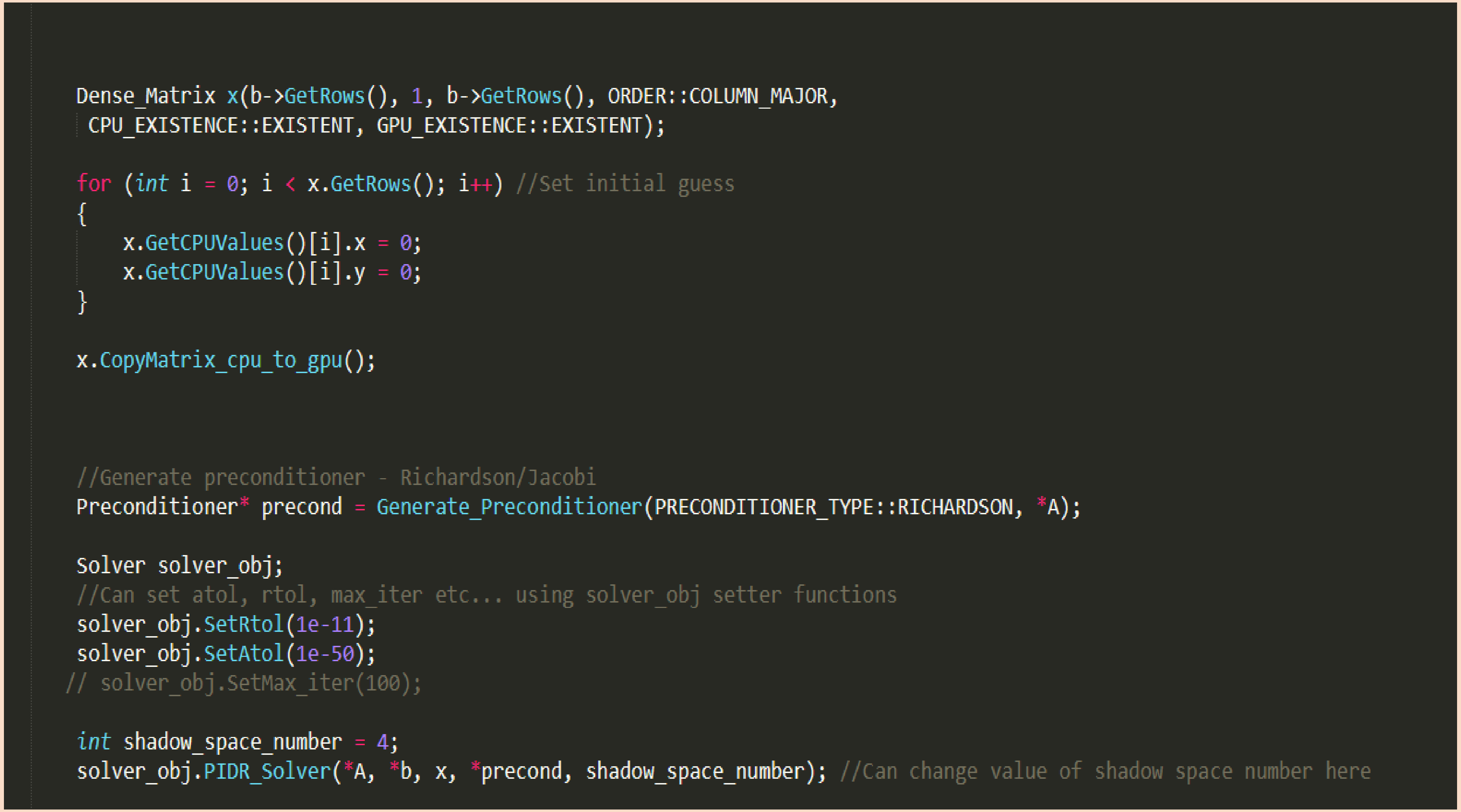
Command line arguments: matrix market file for matrix A(COO format), matrix market file for vector b (array format)

(Providing vector b is optional)

- Output:

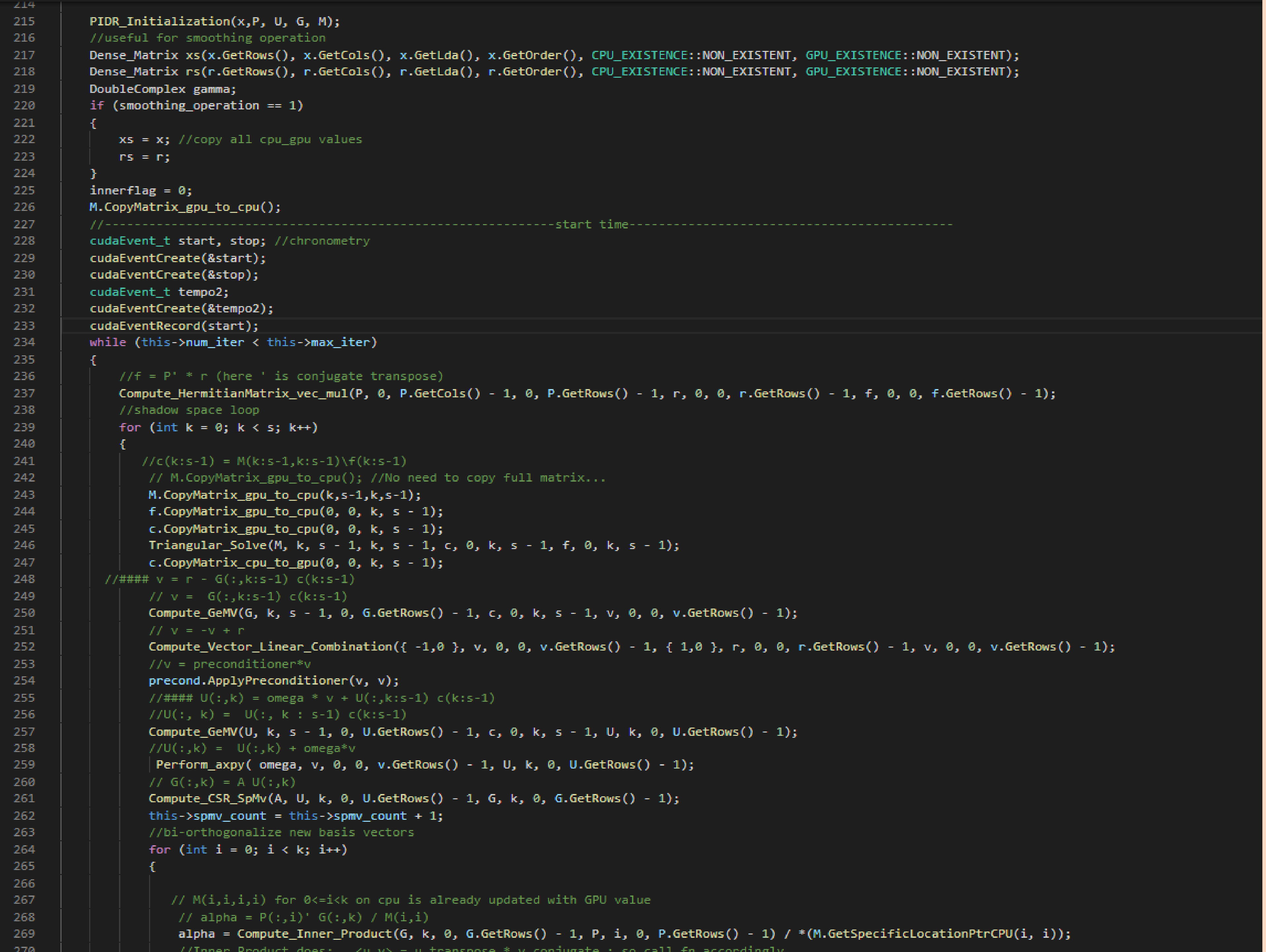
Solver attributes such as num_iter, runtime, final residual etc. printed on console. Generates files containing solution vector x , residual vector and log (residuals and timings…)

Change parameters...



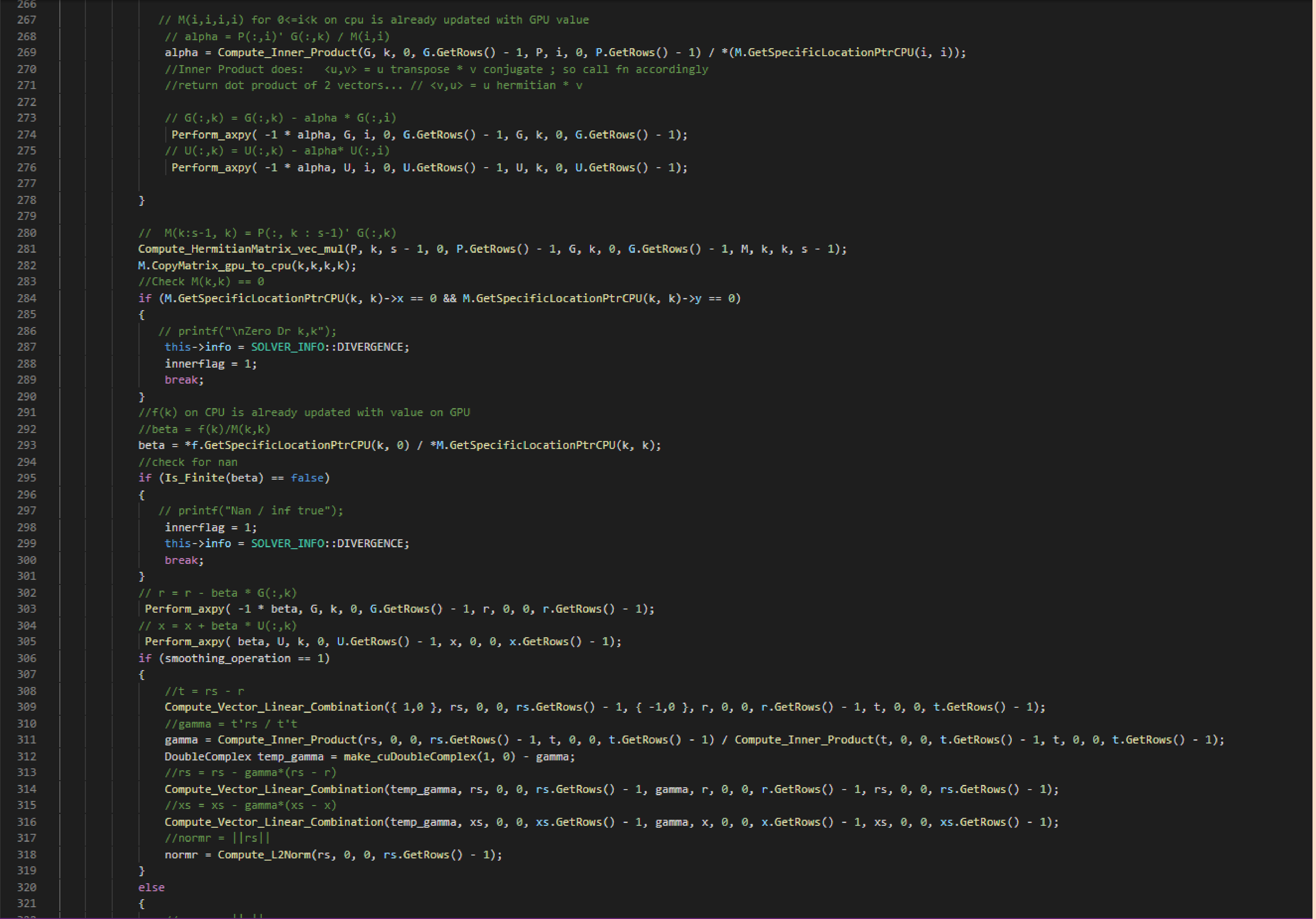
```
Dense_Matrix x(b->GetRows(), 1, b->GetRows(), ORDER::COLUMN_MAJOR,
 CPU_EXISTENCE::EXISTENT, GPU_EXISTENCE::EXISTENT);

for (int i = 0; i < x.GetRows(); i++) //Set initial guess
{
    x.GetCPUValues()[i].x = 0;
    x.GetCPUValues()[i].y = 0;
}

x.CopyMatrix_cpu_to_gpu();



//Generate preconditioner - Richardson/Jacobi
Preconditioner* precond = Generate_Preconditioner(PRECONDITIONER_TYPE::RICHARDSON, *A);

Solver solver_obj;
//Can set atol, rtol, max_iter etc... using solver_obj setter functions
solver_obj.SetRtol(1e-11);
solver_obj.SetAtol(1e-50);
// solver_obj.SetMax_iter(100);

int shadow_space_number = 4;
solver_obj.PIDR_Solver(*A, *b, x, *precond, shadow_space_number); //Can change value of shadow space number here
```

Can easily change the value of initial guess, shadow space number, solver parameters such as atol, rtol etc.
-snippet from main() function

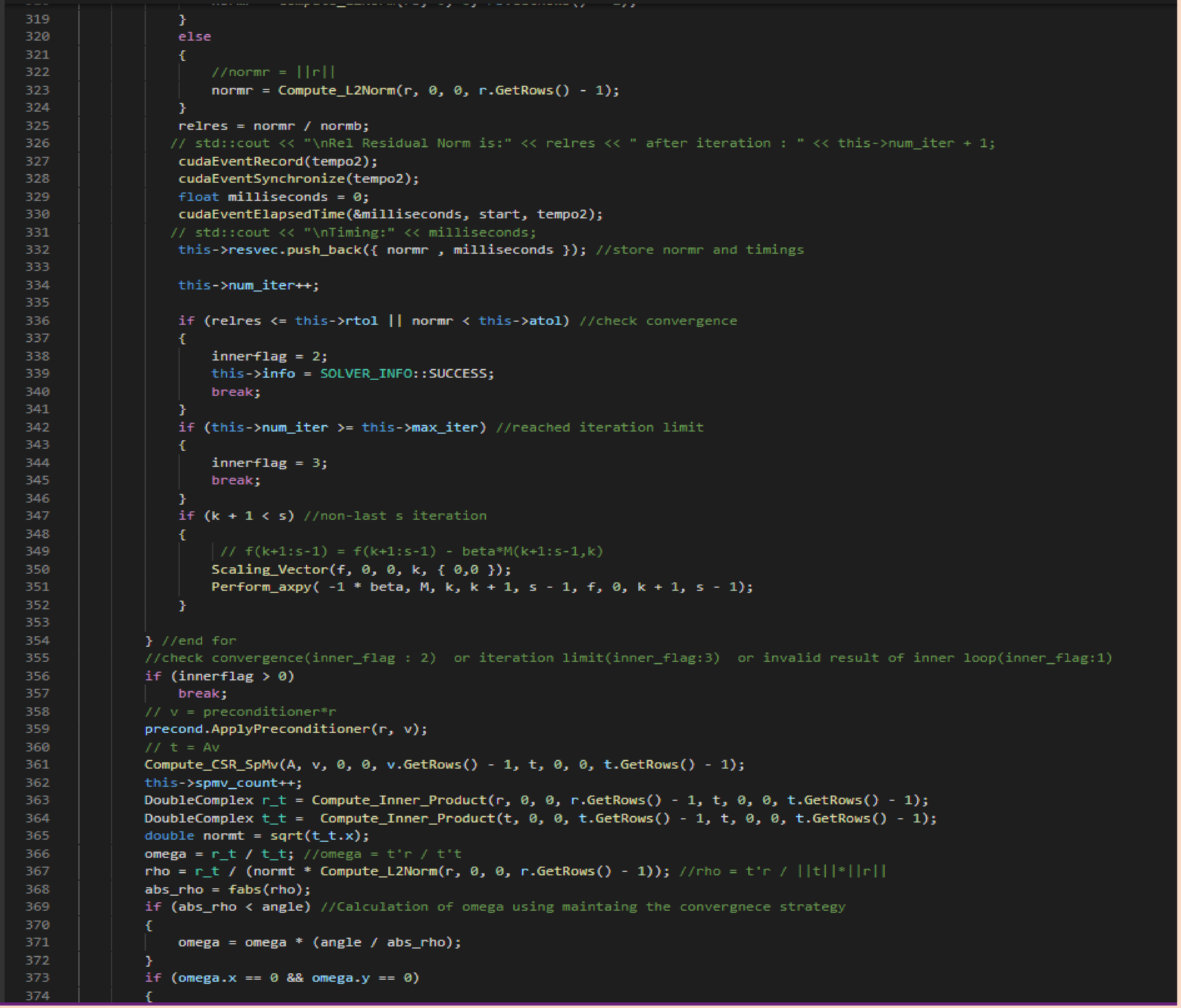


```
    PIDR_Initialization(x,P, U, G, M);
    //useful for smoothing operation
    Dense_Matrix xs(x.GetRows(), x.GetCols(), x.GetLda(), x.GetOrder(), CPU_EXISTENCE::NON_EXISTENT, GPU_EXISTENCE::NON_EXISTENT);
    Dense_Matrix rs(r.GetRows(), r.GetCols(), r.GetLda(), r.GetOrder(), CPU_EXISTENCE::NON_EXISTENT, GPU_EXISTENCE::NON_EXISTENT);
    DoubleComplex gamma;
    if (smoothing_operation == 1)
    {
        xs = x; //copy all cpu_gpu values
        rs = r;
    }
    innerflag = 0;
    M.CopyMatrix_gpu_to_cpu();
    //---------------------------------------------------------------start time--------------------------------------------
    cudaEvent_t start, stop; //chronometry
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEvent_t tempo2;
    cudaEventCreate(&tempo2);
    cudaEventRecord(start);
    while (this->num_iter < this->max_iter)
    {
        //f = P' * r (here ' is conjugate transpose)
        Compute_HermitianMatrix_vec_mul(P, 0, P.GetCols() - 1, 0, P.GetRows() - 1, r, 0, 0, r.GetRows() - 1, f, 0, 0, f.GetRows() - 1);
        //shadow space loop
        for (int k = 0; k < s; k++)
        {
           //c(k:s-1) = M(k:s-1,k:s-1)\f(k:s-1)
            // M.CopyMatrix_gpu_to_cpu(); //No need to copy full matrix...
            M.CopyMatrix_gpu_to_cpu(k,s-1,k,s-1);
            f.CopyMatrix_gpu_to_cpu(0, 0, k, s - 1);
            c.CopyMatrix_gpu_to_cpu(0, 0, k, s - 1);
            Triangular_Solve(M, k, s - 1, k, s - 1, c, 0, k, s - 1, f, 0, k, s - 1);
            c.CopyMatrix_cpu_to_gpu(0, 0, k, s - 1);
      //#### v = r - G(:,k:s-1) c(k:s-1)
            // v =  G(:,k:s-1) c(k:s-1)
            Compute_GeMV(G, k, s - 1, 0, G.GetRows() - 1, c, 0, k, s - 1, v, 0, 0, v.GetRows() - 1);
            // v = -v + r
            Compute_Vector_Linear_Combination({ -1,0 }, v, 0, 0, v.GetRows() - 1, { 1,0 }, r, 0, 0, r.GetRows() - 1, v, 0, 0, v.GetRows() - 1);
            //v = preconditioner*v
            precond.ApplyPreconditioner(v, v);
            //#### U(:,k) = omega * v + U(:,k:s-1) c(k:s-1)
            //U(:, k) =  U(:, k : s-1) c(k:s-1)
            Compute_GeMV(U, k, s - 1, 0, U.GetRows() - 1, c, 0, k, s - 1, U, k, 0, U.GetRows() - 1);
            //U(:,k) =  U(:,k) + omega*v
             Perform_axpy( omega, v, 0, 0, v.GetRows() - 1, U, k, 0, U.GetRows() - 1);
            // G(:,k) = A U(:,k)
            Compute_CSR_SpMv(A, U, k, 0, U.GetRows() - 1, G, k, 0, G.GetRows() - 1);
            this->spmv_count = this->spmv_count + 1;
            //bi-orthogonalize new basis vectors
            for (int i = 0; i < k; i++)
            {

               // M(i,i,i,i) for 0<=i<k on cpu is already updated with GPU value
                // alpha = P(:,i)' G(:,k) / M(i,i)
                alpha = Compute_Inner_Product(G, k, 0, G.GetRows() - 1, P, i, 0, P.GetRows() - 1) / *(M.GetSpecificLocationPtrCPU(i, i));
                //Inner Product does:  <u,v> = u transpose * v conjugate ; so call fn accordingly
```

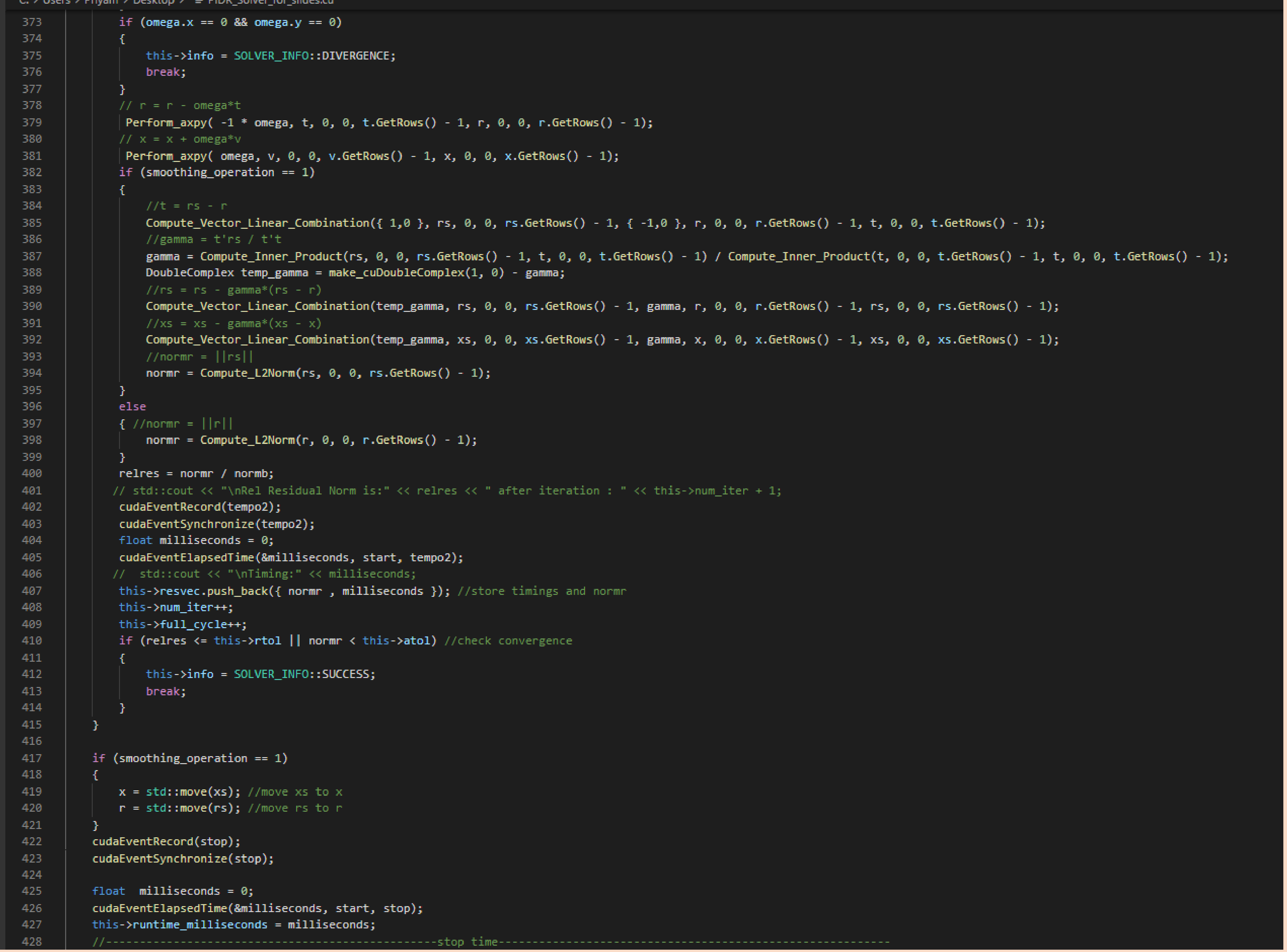
Screenshot of the code for algorithm



```
           // M(i,i,i,i) for 0<=i<k on cpu is already updated with GPU value
            // alpha = P(:,i)' G(:,k) / M(i,i)
            alpha = Compute_Inner_Product(G, k, 0, G.GetRows() - 1, P, i, 0, P.GetRows() - 1) / *(M.GetSpecificLocationPtrCPU(i, i));
            //Inner Product does:   <u,v> = u transpose * v conjugate ; so call fn accordingly
            //return dot product of 2 vectors... // <v,u> = u hermitian * v

            // G(:,k) = G(:,k) - alpha * G(:,i)
             Perform_axpy( -1 * alpha, G, i, 0, G.GetRows() - 1, G, k, 0, G.GetRows() - 1);
            // U(:,k) = U(:,k) - alpha* U(:,i)
             Perform_axpy( -1 * alpha, U, i, 0, U.GetRows() - 1, U, k, 0, U.GetRows() - 1);

        }

        //  M(k:s-1, k) = P(:, k : s-1)' G(:,k)
        Compute_HermitianMatrix_vec_mul(P, k, s - 1, 0, P.GetRows() - 1, G, k, 0, G.GetRows() - 1, M, k, k, s - 1);
        M.CopyMatrix_gpu_to_cpu(k,k,k,k);
        //Check M(k,k) == 0
        if (M.GetSpecificLocationPtrCPU(k, k)->x == 0 && M.GetSpecificLocationPtrCPU(k, k)->y == 0)
        {
           // printf("\nZero Dr k,k");
            this->info = SOLVER_INFO::DIVERGENCE;
            innerflag = 1;
            break;
        }
        //f(k) on CPU is already updated with value on GPU
        //beta = f(k)/M(k,k)
        beta = *f.GetSpecificLocationPtrCPU(k, 0) / *M.GetSpecificLocationPtrCPU(k, k);
        //check for nan
        if (Is_Finite(beta) == false)
        {
           // printf("Nan / inf true");
            innerflag = 1;
            this->info = SOLVER_INFO::DIVERGENCE;
            break;
        }
        // r = r - beta * G(:,k)
         Perform_axpy( -1 * beta, G, k, 0, G.GetRows() - 1, r, 0, 0, r.GetRows() - 1);
        // x = x + beta * U(:,k)
         Perform_axpy( beta, U, k, 0, U.GetRows() - 1, x, 0, 0, x.GetRows() - 1);
        if (smoothing_operation == 1)
        {
            //t = rs - r
            Compute_Vector_Linear_Combination({ 1,0 }, rs, 0, 0, rs.GetRows() - 1, { -1,0 }, r, 0, 0, r.GetRows() - 1, t, 0, 0, t.GetRows() - 1);
            //gamma = t'rs / t't
            gamma = Compute_Inner_Product(rs, 0, 0, rs.GetRows() - 1, t, 0, 0, t.GetRows() - 1) / Compute_Inner_Product(t, 0, 0, t.GetRows() - 1, t, 0, 0, t.GetRows() - 1);
            DoubleComplex temp_gamma = make_cuDoubleComplex(1, 0) - gamma;
            //rs = rs - gamma*(rs - r)
            Compute_Vector_Linear_Combination(temp_gamma, rs, 0, 0, rs.GetRows() - 1, gamma, r, 0, 0, r.GetRows() - 1, rs, 0, 0, rs.GetRows() - 1);
            //xs = xs - gamma*(xs - x)
            Compute_Vector_Linear_Combination(temp_gamma, xs, 0, 0, xs.GetRows() - 1, gamma, x, 0, 0, x.GetRows() - 1, xs, 0, 0, xs.GetRows() - 1);
            //normr = ||rs||
            normr = Compute_L2Norm(rs, 0, 0, rs.GetRows() - 1);
        }
        else
        {
```



```
            }
            else
            {
                //normr = ||r||
                normr = Compute_L2Norm(r, 0, 0, r.GetRows() - 1);
            }
            relres = normr / normb;
           // std::cout << "\nRel Residual Norm is:" << relres << " after iteration : " << this->num_iter + 1;
            cudaEventRecord(tempo2);
            cudaEventSynchronize(tempo2);
            float milliseconds = 0;
            cudaEventElapsedTime(&milliseconds, start, tempo2);
           // std::cout << "\nTiming:" << milliseconds;
            this->resvec.push_back({ normr , milliseconds }); //store normr and timings

            this->num_iter++;

            if (relres <= this->rtol || normr < this->atol) //check convergence
            {
                innerflag = 2;
                this->info = SOLVER_INFO::SUCCESS;
                break;
            }
            if (this->num_iter >= this->max_iter) //reached iteration limit
            {
                innerflag = 3;
                break;
            }
            if (k + 1 < s) //non-last s iteration
            {
                 // f(k+1:s-1) = f(k+1:s-1) - beta*M(k+1:s-1,k)
                Scaling_Vector(f, 0, 0, k, { 0,0 });
                Perform_axpy( -1 * beta, M, k, k + 1, s - 1, f, 0, k + 1, s - 1);
            }

        } //end for
        //check convergence(inner_flag : 2)  or iteration limit(inner_flag:3)  or invalid result of inner loop(inner_flag:1)
        if (innerflag > 0)
            break;
        // v = preconditioner*r
        precond.ApplyPreconditioner(r, v);
        // t = Av
        Compute_CSR_SpMv(A, v, 0, 0, v.GetRows() - 1, t, 0, 0, t.GetRows() - 1);
        this->spmv_count++;
        DoubleComplex r_t = Compute_Inner_Product(r, 0, 0, r.GetRows() - 1, t, 0, 0, t.GetRows() - 1);
        DoubleComplex t_t =  Compute_Inner_Product(t, 0, 0, t.GetRows() - 1, t, 0, 0, t.GetRows() - 1);
        double normt = sqrt(t_t.x);
        omega = r_t / t_t; //omega = t'r / t't
        rho = r_t / (normt * Compute_L2Norm(r, 0, 0, r.GetRows() - 1)); //rho = t'r / ||t||*||r||
        abs_rho = fabs(rho);
        if (abs_rho < angle) //Calculation of omega using maintaing the convergnece strategy
        {
            omega = omega * (angle / abs_rho);
        }
        if (omega.x == 0 && omega.y == 0)
        {
```



```
C: > Users > Priyam > Desktop > ≡ FIDR_Solver_for_slides.cu
        if (omega.x == 0 && omega.y == 0)
        {
            this->info = SOLVER_INFO::DIVERGENCE;
            break;
        }
        // r = r - omega*t
         Perform_axpy( -1 * omega, t, 0, 0, t.GetRows() - 1, r, 0, 0, r.GetRows() - 1);
        // x = x + omega*v
         Perform_axpy( omega, v, 0, 0, v.GetRows() - 1, x, 0, 0, x.GetRows() - 1);
        if (smoothing_operation == 1)
        {
            //t = rs - r
            Compute_Vector_Linear_Combination({ 1,0 }, rs, 0, 0, rs.GetRows() - 1, { -1,0 }, r, 0, 0, r.GetRows() - 1, t, 0, 0, t.GetRows() - 1);
            //gamma = t'rs / t't
            gamma = Compute_Inner_Product(rs, 0, 0, rs.GetRows() - 1, t, 0, 0, t.GetRows() - 1) / Compute_Inner_Product(t, 0, 0, t.GetRows() - 1, t, 0, 0, t.GetRows() - 1);
            DoubleComplex temp_gamma = make_cuDoubleComplex(1, 0) - gamma;
            //rs = rs - gamma*(rs - r)
            Compute_Vector_Linear_Combination(temp_gamma, rs, 0, 0, rs.GetRows() - 1, gamma, r, 0, 0, r.GetRows() - 1, rs, 0, 0, rs.GetRows() - 1);
            //xs = xs - gamma*(xs - x)
            Compute_Vector_Linear_Combination(temp_gamma, xs, 0, 0, xs.GetRows() - 1, gamma, x, 0, 0, x.GetRows() - 1, xs, 0, 0, xs.GetRows() - 1);
            //normr = ||rs||
            normr = Compute_L2Norm(rs, 0, 0, rs.GetRows() - 1);
        }
        else
        { //normr = ||r||
            normr = Compute_L2Norm(r, 0, 0, r.GetRows() - 1);
        }
        relres = normr / normb;
       // std::cout << "\nRel Residual Norm is:" << relres << " after iteration : " << this->num_iter + 1;
        cudaEventRecord(tempo2);
        cudaEventSynchronize(tempo2);
        float milliseconds = 0;
        cudaEventElapsedTime(&milliseconds, start, tempo2);
       //  std::cout << "\nTiming:" << milliseconds;
        this->resvec.push_back({ normr , milliseconds }); //store timings and normr
        this->num_iter++;
        this->full_cycle++;
        if (relres <= this->rtol || normr < this->atol) //check convergence
        {
            this->info = SOLVER_INFO::SUCCESS;
            break;
        }
    }

    if (smoothing_operation == 1)
    {
        x = std::move(xs); //move xs to x
        r = std::move(rs); //move rs to r
    }
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);

    float  milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);
    this->runtime_milliseconds = milliseconds;
    //-------------------------------------------------stop time----------------------------------------------------------
```

Numerical Results

(All matrices are taken from SuiteSparse Matrix Collection)

Correctness...

Parameter w_j is computed via the maintain the convergence strategy in all the numerical experiments.

Case: Smoothing disabled

Project_iter : 459

A : raefsky2.mtx; b:raefsky2_b.mtx

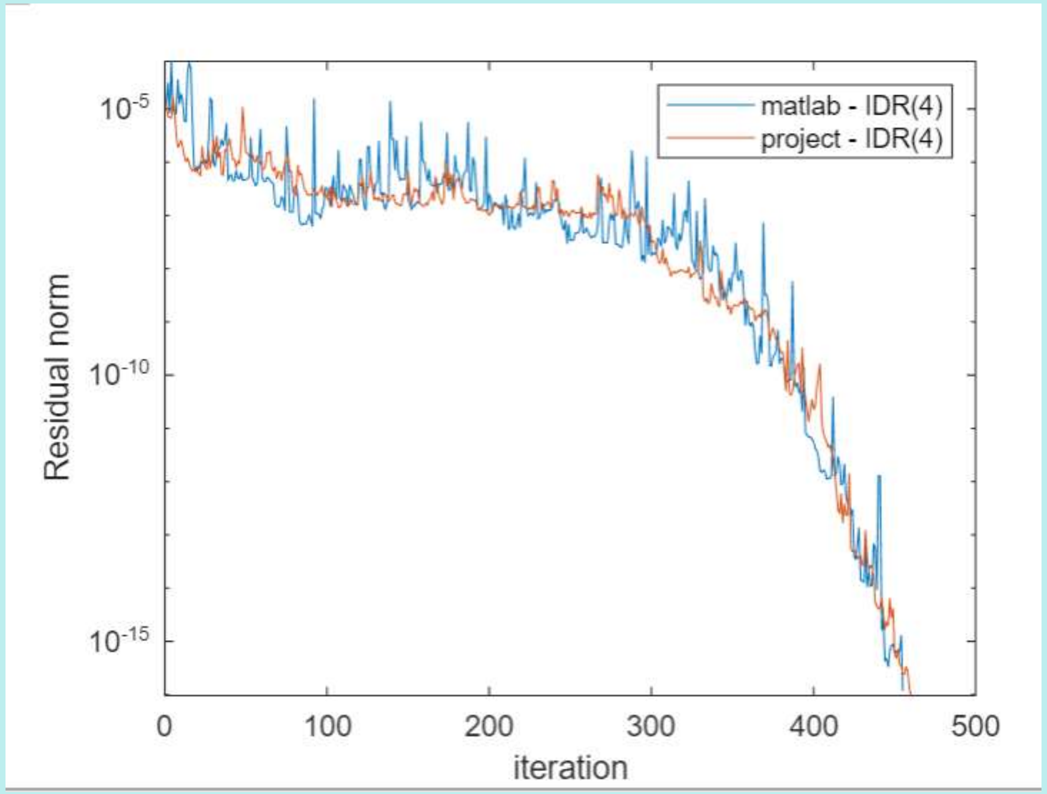
condest(A) = 1.08e+04

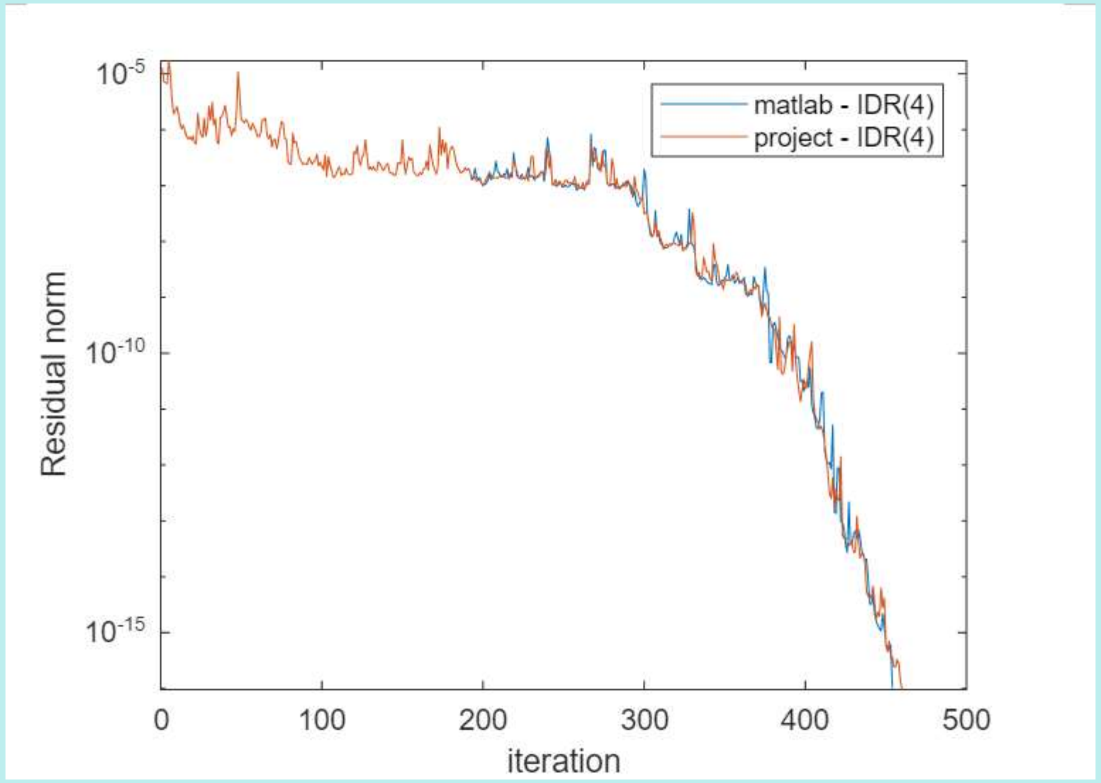
s = 4

rtol = 1e-11

Jacobi (right)Preconditioning

Ran on GeForce MX130



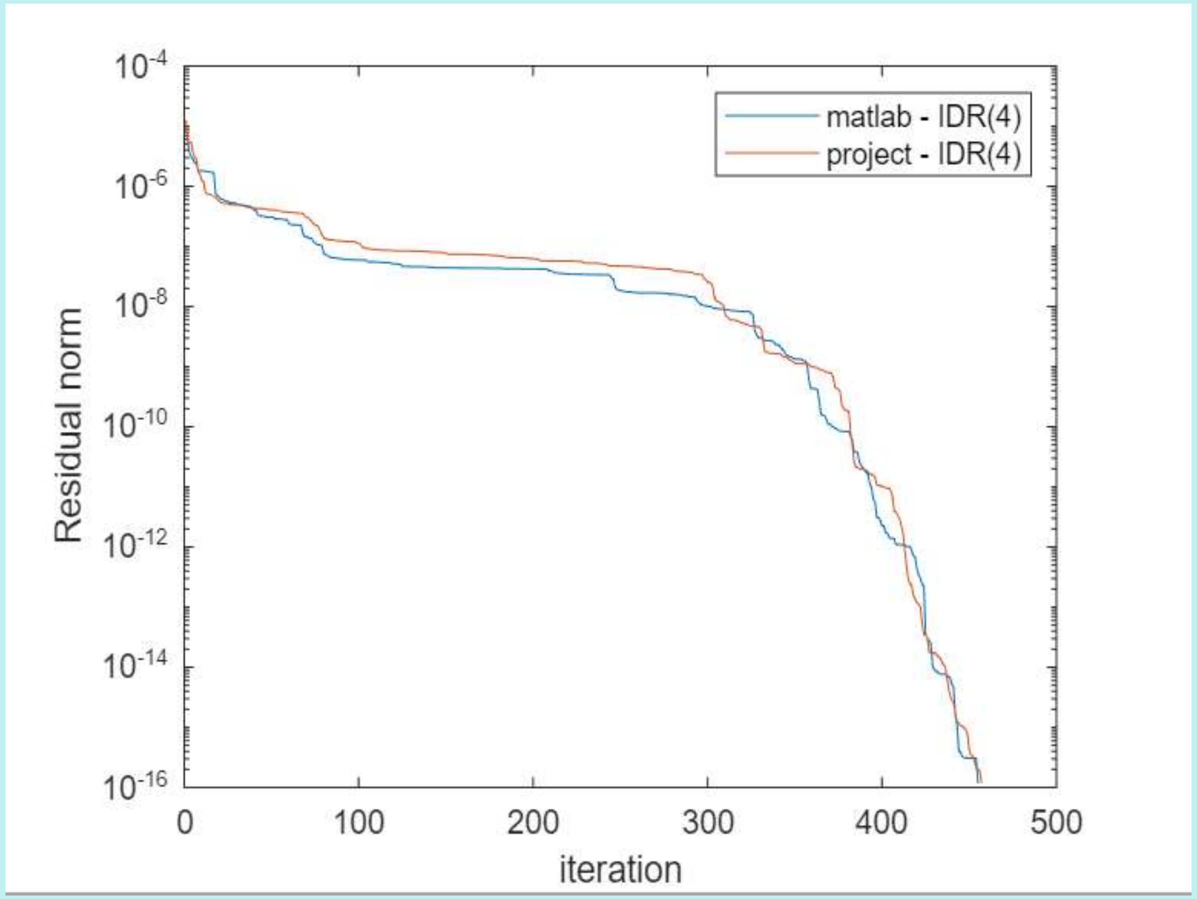


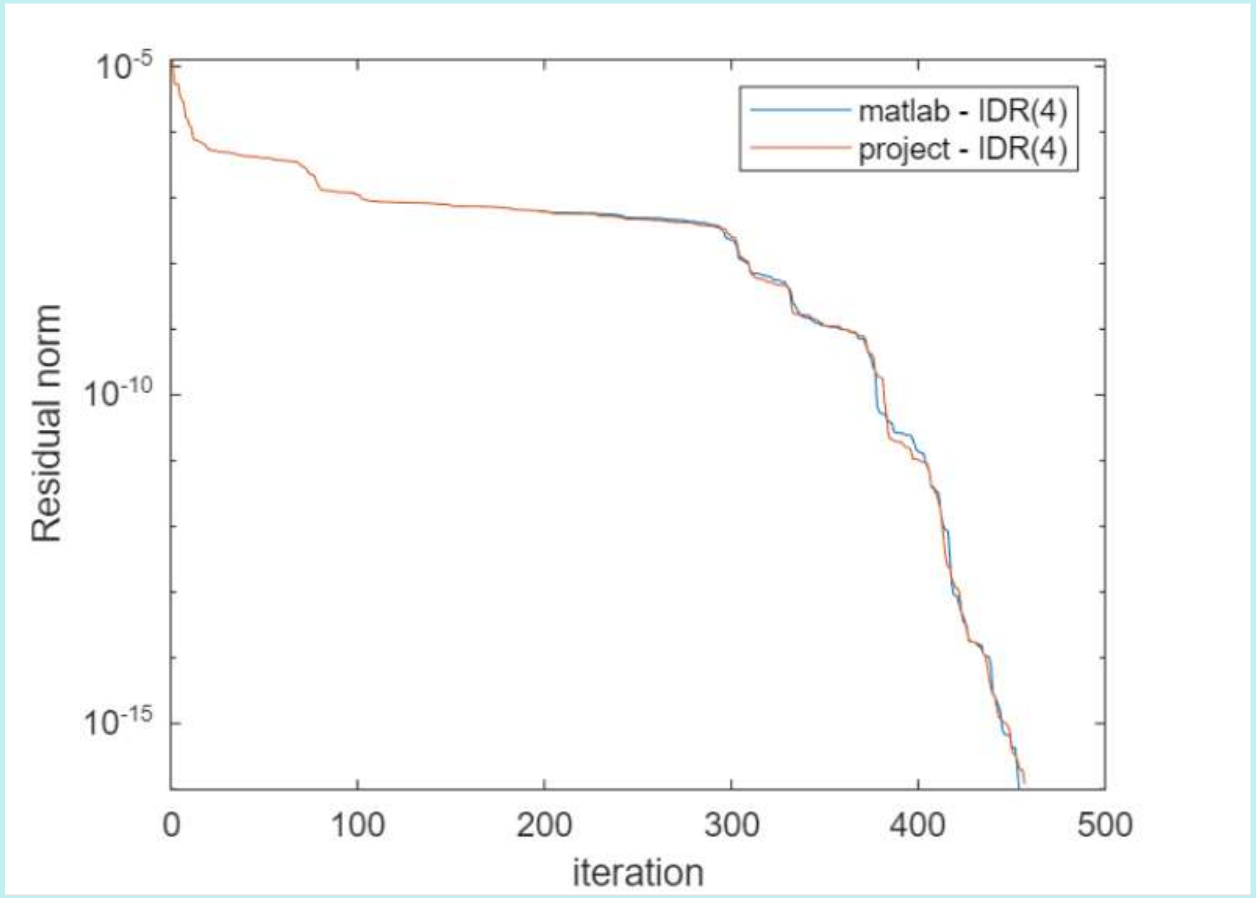
When matrix P is different:
Matlab_iter : 453

When matrix P is the same:
Matlab_iter : 454

Case: Smoothing enabled

Project_iter: 456

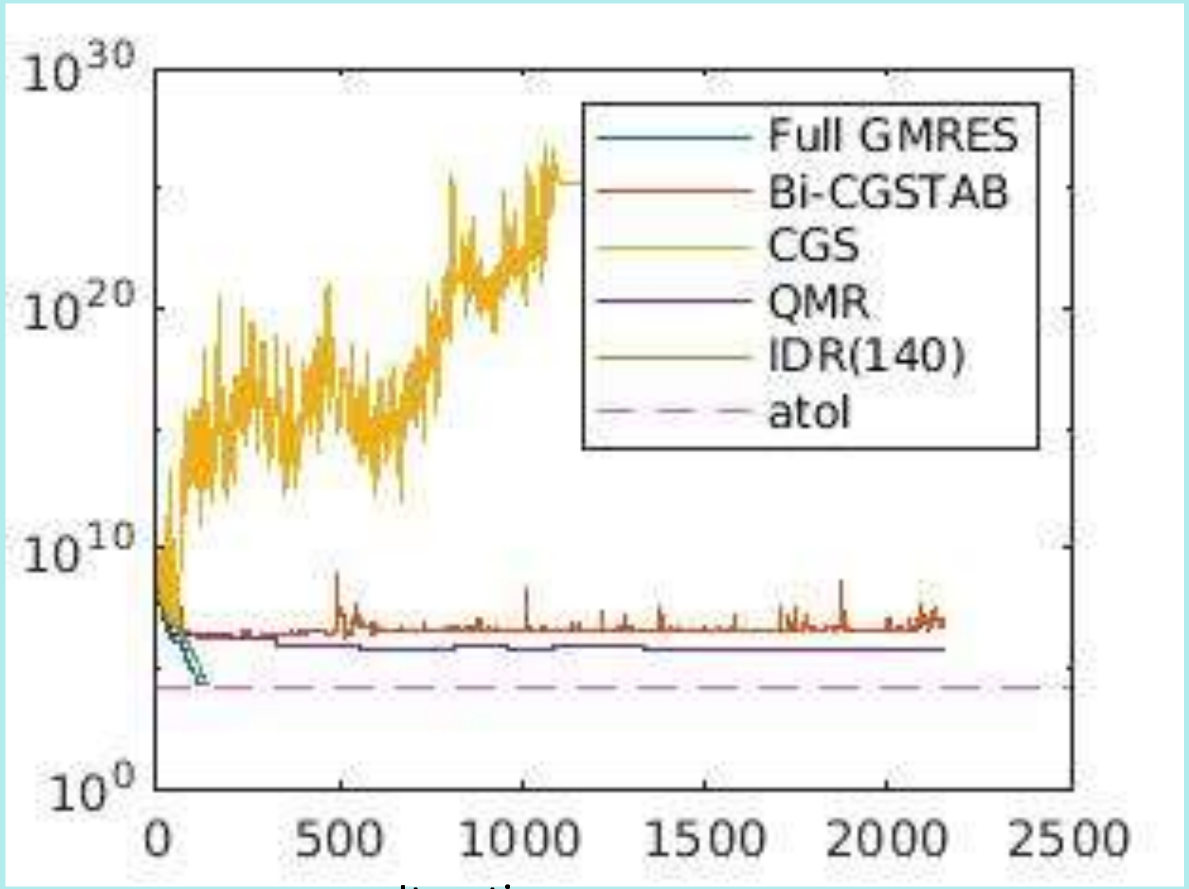




When matrix P is different:
Matlab_iter: 454

When matrix P is the same:
Matlab_iter: 453

Convergence comparison



Residual Norm

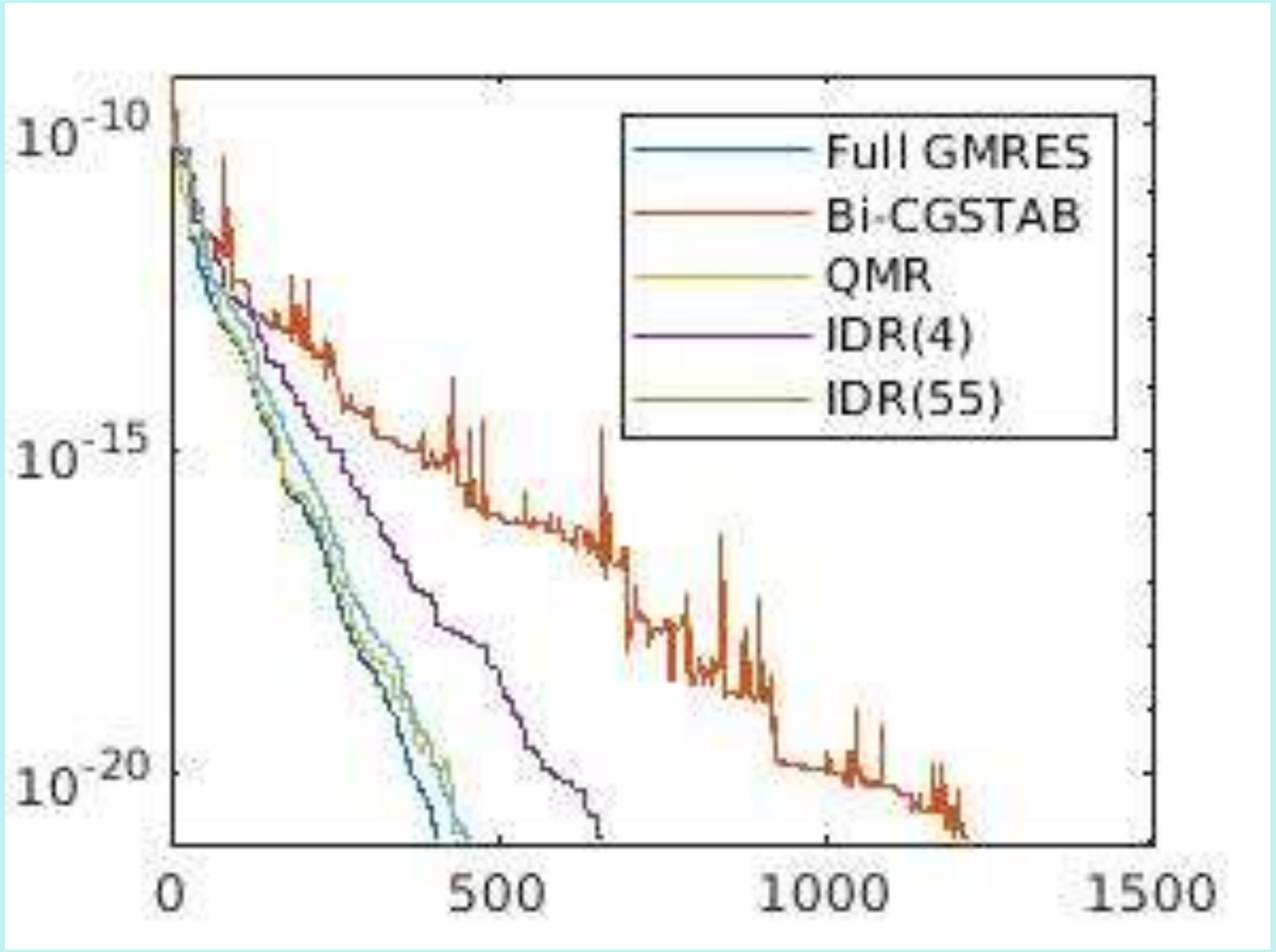
Iteration

Matrix: sherman2.mtx
RHS: sherman2_b.mtx
Condition number ~ 1e+12
No preconditioner
rtol: 1e-04

Ran on GeForce MX130

Full GMRES : 119 iterations
IDR(140)-biortho: 142 iterations
Rest of the methods fail to converge

Convergence rate…



Matrix: add20.mtx
RHS: add20_b.mtx
Condition no ~ 1e+04
rtol = 1e-11
No preconditioner

Ran on GeForce MX130

Full GMRES: 409
Bi-CGSTAB: 1217
QMR: 453
IDR(4) : 661
IDR(55):458

Performance computation…

- Total number of operations(Identity Preconditioner, Smoothing enabled):

N = rows/cols in matrix A
nz = total number of true non zero elements in matrix A
s = shadow space number

Inner cycle/region for any given k(k is the iteration index)
(a single iteration of inner for loop):

Triangular solve: (s-k)/2 + 4(s-k)(s-k+1)
2 x GeMv: 2 x N(7(s-k) -1)
(2k + 3) x axpy:(2k +3) x 8N
4 x Vector_Linear_Combination: 4 x 14N
1 x CSRSpMv: 8nz - 2N
(k + 2) x inner_product: (k+2) x (9N – 2)
(k + 2) x division:(k+2) x 9
1 x Hermitianmatrix*vec: (s-k)[7N + 2(N-1)]
1 x Norm: 9N -1
Non-Last(s) iteration: 6(k+1) + 8(s-k-1)

Outer region(outer while loop excluding the inner for loop):

HermitianMatrix*vec: s(9N - 2)
CSRSpMv: 8nz - 2N
4 x inner_product: 4 x (9N – 2)
2 x Norm: 2 x (9N – 1)
2 x axpy:2 x 8N
3 x VectorLinearCombination:3 x 14N
1 x extra Division: 9

Performance: Total number of operations/ runtime

Total number of operations:

$$\textbf{Number of Full\_Cycles*[operations for outer loop} + \sum_{k=0}^{S-1} operations\ for\ single\ iteration\ of\ \ inner\ cycle\ for\ a\ given\ k\,] + \sum_{k=0}^{total\ number\ of\ iterations\ -(s+1)*number\ of\ full\ cycles\ -1} (operations\ for\ a\ single\ iteration\ of\ \ inner\ cycle\ for\ a\ given\ k)$$

Performance and runtime comparison

Ran on Tesla K20Xm GPU

Peak Performance (double precision): 1312 Gflop/sec

Ran for 100 iterations

Matrix	IDR(s)	Runtime(s) :		Performance (Gflop/s): Project code
		Matlab	Project code	
airfoil_2d N = 14214 True nz = 259688	IDR(1)	0.094	0.067	5.853
	IDR(4)	0.153	0.076	6.310
	IDR(8)	0.221	0.092	6.652
Trefethen_20000 N = 20000 True nz = 554466	IDR(1)	0.196	0.087	8.133
	IDR(4)	0.326	0.096	8.636
	IDR(8)	0.298	0.113	8.982
pwtk N = 217918 True nz = 11634424	IDR(1)	2.194	0.644	18.902
	IDR(4)	2.425	0.693	19.439
	IDR(8)	2.764	0.753	20.560
inline_1 N = 503712 True nz = 36816342	IDR(1)	6.656	1.665	21.668
	IDR(4)	7.274	1.768	22.108
	IDR(8)	8.244	1.911	22.888
Bone010 N = 986703 True nz = 71666325	IDR(1)	10.020	3.147	22.339
	IDR(4)	10.919	3.346	22.771
	IDR(8)	16.009	3.643	23.418

Smoothing enabled, Identity preconditioner

Future work…

- Optimization techniques
- Kernel fusion, Kernel overlap, CUDA streams
- Dynamic parallelism
- Error handling

References

- Algorithm 913: An elegant IDR(s) variant that efficiently exploits biorthogonality properties. Article in ACM Transaction on Mathematical Software –November 2011. MARTIN B. VAN GIJZEN and PETER SONNEVELD, Delft University of Technology
- Optimization and performance evaluation of the IDR iterative Krylov solver on GPUs.   Article in The International Journal of High Performance Computing Applications -2016.   Hartwig Anzt1 , Moritz Kreutzer2 , Eduardo Ponce1 , Gregory D Peterson1 , Gerhard Wellein2 and Jack Dongarra1,3,4
- Magma Sparse Library
- Matlab inbuilt routines
- Matlab reference implementation

Thank you for your attention…