Charles Dufour
257587
charles.dufour@epfl.ch

Conjudate Gradient, MPI and CUDA implementations

P&HPC, MATH-454

2023-01-06

Introduction

The Conjugate Gradient (GC) is an iterative method for solving linear systems of equations $Ax = b$. It requires the matrix A to be real, symmetric, and positive definite. It generates an approximate solution x_k at every iteration k, with $x_{k+1} = x_k + \alpha_k p_k$, where:

- p_k is the conjugate vector at iteration k (aka, search direction) such that $p_k \cdot Ap_j = 0, j = 0, \dots, k-1$: a practical of enforcing it is by using

$$p_k = r_k - \sum_{i=0}^{k-1} \frac{p_i \cdot Ar_k}{p_i \cdot Ap_i} p_i.$$

- α_k is the step length at iteration k.
- $r_k = b - Ax_k$ is the residual.

The algorithm will then be:

Algorithm 1: Sequential Conjugate gradient pseudo-code

Data: A, b, x_0, ϵ
Result: x such that $Ax \approx b$

$r_0 = b - Ax_0$
$p_0 = r_0$
$k = 0$
$\gamma_0 = r^T r$
while $||r_k||_2 > \epsilon$ **do**
  $z_k = Ap_k$
  $\alpha = \frac{\gamma_k}{p_k^T z_k}$
  $x_k = x_k + \alpha * p_k$
  $r_{k+1} = r_k - \alpha * z_k$
  $\gamma_{k+1} = r_{k+1}^T r_{k+1}$
  $p_{k+1} = r_{k+1} + \frac{\gamma_{k+1}}{\gamma_k} p_k$
end
Return x_k

One can precompute Ap_k to only do one matrix-vector product, the rest being inner products of vectors.

Profiling

We are given two solvers for dense and sparse matrices respectively. We start by profiling the existing solvers to understand the bottlenecks with `gprof` and `perf`. A common bottleneck is reading the matrix and initializing b. It is much more noticeable for the sparse solver, but this is due to the difference in time taken: the dense solver takes ~ 30 seconds while the sparse takes ~ 0.05 seconds for a 10000×10000 matrix.

- Dense Solver : `dgenv_kernel_4x4` (on line 10 and 23 in code appendix) is responsible for 95% of the time spent

```
cblas_dgemv(CblasRowMajor, CblasNoTrans, m_m, m_n, 1., m_A.data(), m_n,p.data(), 1,
    0., Ap.data(), 1);
```

  and the data loading only accounts for 0.15%

- Sparse Solver: `daxpy_kernel_8` on line 155 is responsible for $\sim 12\%$ of the time taken, `mat_vec` uses $\sim 59\%$.

```
cblas_daxpy(m_n, alpha, p.data(), 1, x.data(), 1);
MatrixCOO::mat_vec (inlined);
```

while creating the sparse matrix is responsible for $\sim 12\%$.

We decide to deal with the dense case. We now analyze the expected improvement and discuss potential drawbacks to the different parallelization approaches (CUDA and MPI).

MPI implementation of dense solver

Pseudo code

We will divide the work by assigning several matrix rows $A \in \mathbb{R}^{m \times n}$ to each process. Given p processes, process i will deal with the rows $A_{(i)} := A[i \times p : (i+1) \times p - 1][:]$. We avoid communicating big amounts of data as much as possible.

Algorithm 2: MPI Conjugate gradient pseudo-code

Data: $A_{(i)}, b_{(i)}, x_0, \epsilon$
Result: x such that $Ax \approx b$

$r_{0(i)} = b_{(i)} - A_{(i)}x_0$;
$p_{0(i)} = r_{0(i)}$;
$\gamma_0 = \sum_{i=0}^{p-1} r_{0(i)}^T r_{0(i)}$; // Done with MPI_Allreduce
$k = 0$;
while $||r_k||_2 > \epsilon$ **do**
  Concatenate all the $p_{k(i)}$ into p_k ; // Done with MPI_AllGather
  $z_{k(i)} = A_{(i)}p_k$;
  $w = \sum_{i=0}^{p-1} p_{k(i)}^T z_{k(i)}$; // Done with MPI_Allreduce
  $\alpha = \gamma_k / w$;
  $x_{k(i)} = x_{k(i)} + \alpha p_{k(i)}$;
  $r_{k+1(i)} = r_{k(i)} - \alpha z_{k(i)}$;
  $\gamma_{k+1} = \sum_{i=0}^{p-1} r_{k+1(i)}^T r_{k+1(i)}$; // Done with MPI_Allreduce
  $p_{k+1(i)} = r_{k+1(i)} + \frac{\gamma_{k+1}}{\gamma_k} p_{k(i)}$;
  $k = k + 1$;
end
Concatenate and return x_k ; // Done with MPI_Gather

In one iteration of the MPI algorithm 2, each thread will send $3+2\lfloor m/p \rfloor$ floats and receive $(p-1)*(3+2\lfloor m/p \rfloor)$ floats. So the communication cost is $\mathcal{O}(n)$ (A is symmetric, so $n = m$), while the computation cost per processor is dominated by the matrix vector product, i.e. $\mathcal{O}\left(n^2/p\right)$.

Expected results

Amdhal's law (strong scaling) According to the profiling, we have $s = 0.05$ as the execution time spent on the serial part of the solver (data loading,...) and $p = 0.95$ the proportion of execution time that can be parallelized. The speedup for N processors can then be formulated as

$$S(N) = \frac{1}{(s + p/N)} = \frac{1}{0.05 + 0.95 * N} \underset{N \to \infty}{\longrightarrow} 20,$$

meaning we do not expect to get a speedup better than 20 for a fixed-size problem.

Gustafson's law (weak scaling) We now let the amount of work per processor stay constant, meaning we increase the amount of work as we increase the number of processors: $w_N = N \times w_1$. The efficiency can be expressed as a scaled ratio of time to solution

$$E(N) = \frac{S(N)}{N} = \frac{1}{N} \frac{N \times T_1}{T_N} = \frac{T_1}{T_N},$$

where T_k represent the time taken by k processors to work on a problem of size $w_k = k \times w_1$.
For the MPI case, we expect that any drop in efficiency will be explained in terms of communication overhead. Different world communication setups can significantly impact the performance of this method: the most expensive part being the matrix-vector multiplication in lines 10 and 23 in Listing 3. If we split the works per row or block, this will change the amount of data communicated and the number of communications.

Experiment design

Check the impact of matrix size on performance (I guess that for small matrices, the overhead of communication will be a pain, while for big matrices, separating the inner product work will be worth it). At the same time, the number of processors and whether we have one or more physical nodes will have an impact.

Empirical results

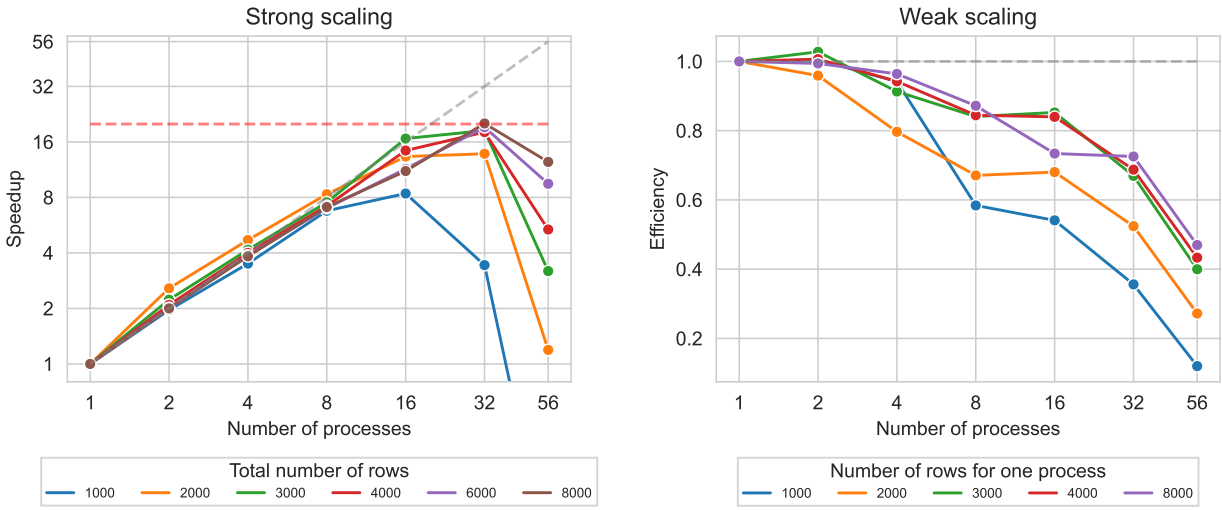

Figure 1: Results on HELVETIOS for generated tridiagonal Laplacian matrix, running 100 iterations. The weak scaling plot on the right reports the number of rows for one process, the subsequent runs scaled the matrix sized by $\sqrt{P}$, where P was the number of processors. The drop after 32 processors is explained by moving to two physical nodes, meaning increased communication costs.

CUDA implementation of dense solver

We implement the whole CG algorithm on device: moving only once the data at the beginning and fetching the solution x_k at the very end. The only other memory operation is checking the convergence condition, as seen in the appendix. Our grid and block sizes are explained as follows:

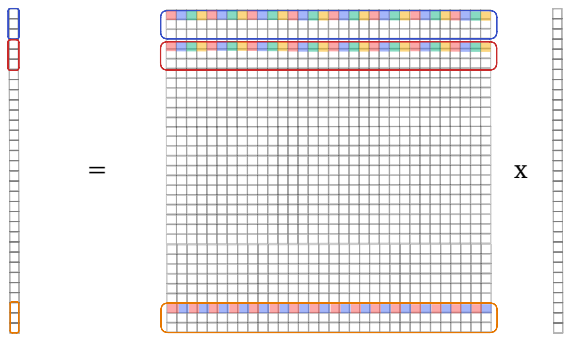

Figure 2: Threads repartition in the CUDA implementation: each block deals with several rows (here 3). Then each block has several threads per row (4 at the top, 2 at the bottom). The threads then access the memory in strided fashion to load the vector on the RHS with coalesced access.

We use dynamic shared data to reduce the results of each row in $\mathcal{O}(\log(k))$, where k is the number of threads per row.

Empirical results

We use a Tesla V100-PCIE-32GB with compute capability 7.0 for the experiment (for more details see vola architecture white paper). The maximum number of blocks per streaming multiprocessor (SM) in our GPU is 32 (up to 1024 threads/block), and we have 80 SM available (up to 2048 threads/SM). To perform a fair comparison with different matrix sizes, we create the following matrix:

$$M = \begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix},$$

where the non-specified entries are 0.

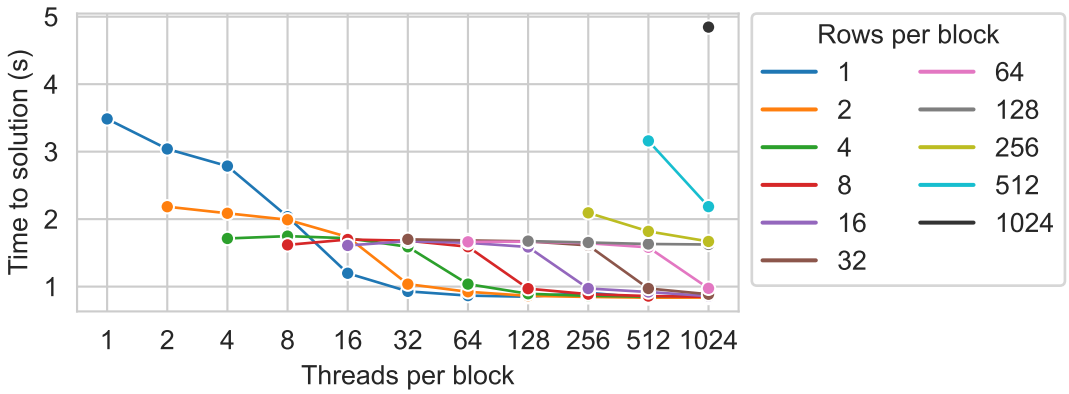

Figure 3: Time to solution (without data loading) for the original matrix (10000 rows). Each experiment repeats the process 5 times to stabilize the measurements. The conjugate gradient tolerance was set to 10^{-10}

We can see that we should assign as many threads per row of the matrix as possible and only one row per block.
Detailed breakdown:

- Number of rows per block < 128 :

  The more threads per block, the better the performance. We have at least 179 blocks; hence our SMs are always used, and we only need to worry about occupancy: if we have blocks that are not a multiple of 32 threads (especially if the number of threads is less than 32), we will have bad performance. This explains the left-hand side of the plot.

  We notice that when the number of threads goes from 8 to 16, we have a big improvement in performance, regardless of the number of rows/threads per block:

  - We have $2048 \times 80 \approx 16$K threads, meaning we need 16 per row to fully saturate the GPU.
  - This may also be due to the coalesced access to the vector on the right-hand side. The device can access global memory via 32-,64-, or 128-byte transactions that are aligned, meaning we can load $4, 8,$ or 16 doubles in one transaction with coalesced access.

- Number of rows per block ≥ 128:

  We have less than 80 blocks after this point (respectively 79,40,20, and 10), so we are not entirely using the 80 streaming multiprocessors of our GPU. Adding more threads to each row always improves the performance, even if it is less noticeable.

Impact of grid/block/matrix sizes

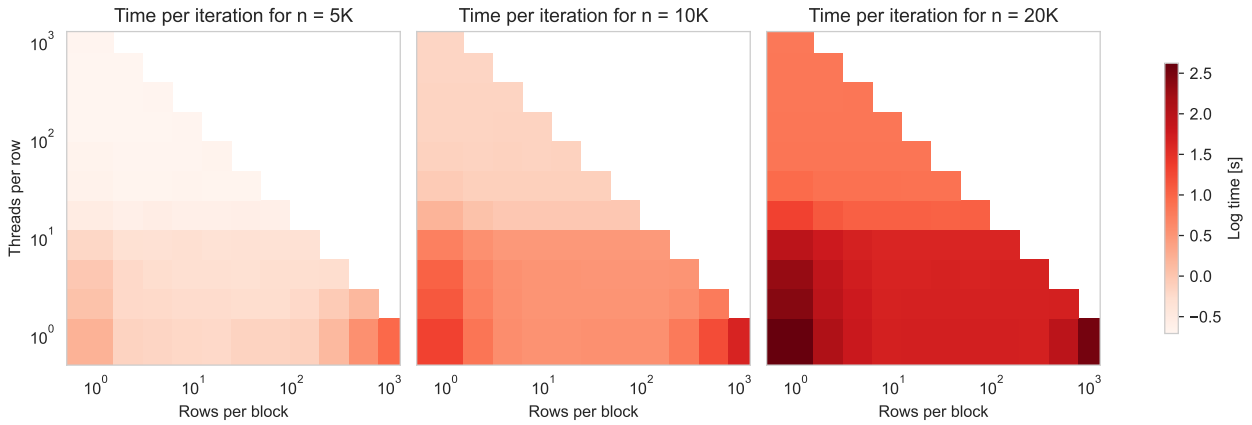

Figure 4: Time per iteration for GPU implementation. Given a fixed matrix size n, the relative differences inside the heatmap are comparable: here the common scale hides this feature.

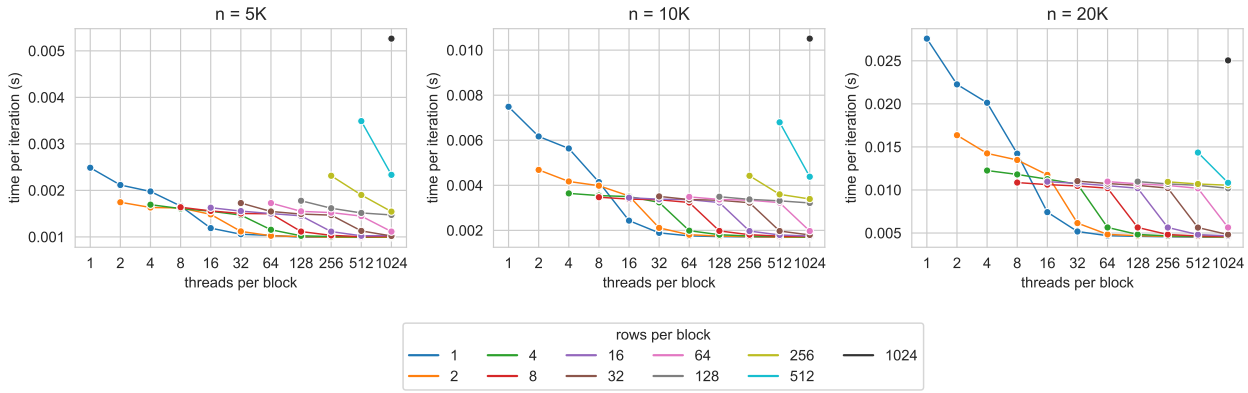

Figure 5: Time per iteration (in seconds) for different matrix sizes $n = [5 \cdot 10^3, 10 \cdot 10^3, 20 \cdot 10^3]$, for varying block dimensions. Each experiment runs 500 iterations of the gradient conjugate solver and repeats the process 5 times to stabilize the measurements.

```
==16229== Profiling application: ./cgsolver lap2D_5pt_n100.mtx 1024 1
==16229== Profiling result:
  Time(%)      Time     Calls       Avg       Min       Max        Name
96.05%      4.46462s       461  9.6846ms  9.6588ms  9.7295ms  matrix_vector
3.76%       174.68ms         4   43.670ms  1.3760us  174.66ms  [CUDA memcpy HtoD]
```

Listing 1: `nvprof` (GPU activities $> 1\%$) on the starting matrix of size $10^3 \times 10^3$, where each block deals with 1024 rows.

```
==16728== Profiling application: ./cgsolver lap2D_5pt_n100.mtx 1 1024
==16728== Profiling result:
 Time(%)      Time     Calls       Avg       Min       Max        Name
71.43%      459.66ms       490  938.07us  935.83us  944.31us  matrix_vector
27.12%      174.51ms         4   43.628ms  1.3760us  174.49ms  [CUDA memcpy HtoD]
```

Listing 2: `nvprof` (GPU activities $> 1\%$) on the matrix starting matrix of size $10^3 \times 10^3$ where each block deals with only 1 row.

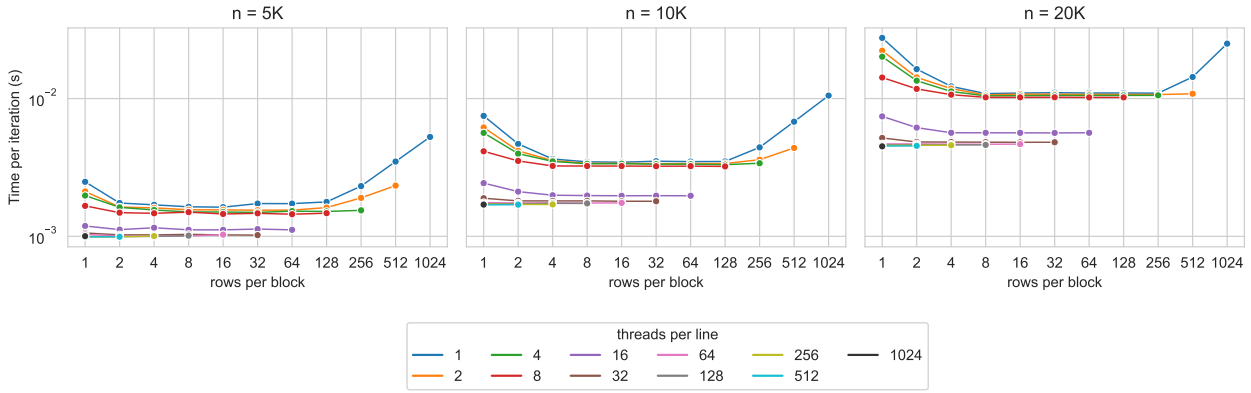

Figure 6: Time per iteration (in seconds) for different matrix sizes $n = [5 \cdot 10^3, 10 \cdot 10^3, 20 \cdot 10^3]$, for varying block dimensions. Each experiment runs 500 iterations of the gradient conjugate solver and repeats the process 5 times to stabilize the measurements.

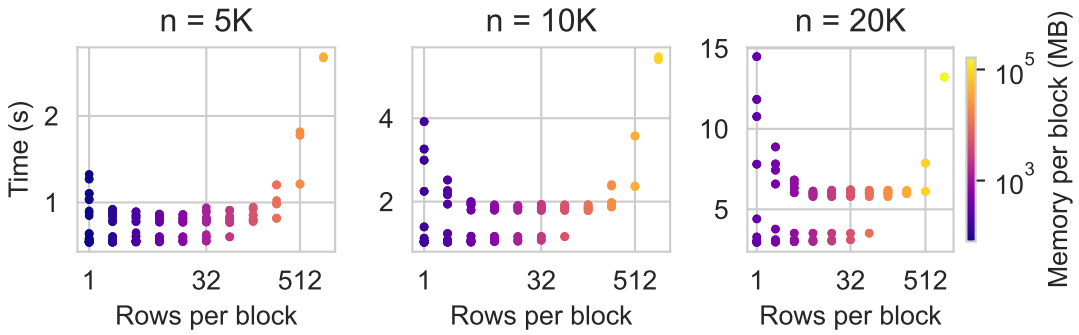

Figure 7: Memory needed for each block vs time to solution. This counts the number of bytes that need to be accessed by the block (from A,b and the shared memory for strided global memory access reduction).

Original Dense code

```
void CGSolver::solve(std::vector<double> &x)
{
  std::vector<double> r(m_n);
  std::vector<double> p(m_n);
  std::vector<double> Ap(m_n);
  std::vector<double> tmp(m_n);
  std::fill_n(Ap.begin(), Ap.size(), 0.);

  // r = b - A * x
  cblas_dgemv(CblasRowMajor, CblasNoTrans, m_m, m_n, 1., m_Ap.data(), m_n, x.data(),
              1, 0., Ap.data(), 1); //slow boy
  r = m_b;
  cblas_daxpy(m_n, -1., Ap.data(), 1, r.data(), 1);

  p = r;
  auto rsold = cblas_ddot(m_n, r.data(), 1, p.data(), 1);

  for (int k=0; k < m_n; ++k){
    // std::fill_n(Ap.begin(), Ap.size(), 0.); was useless


    // Ap = Ap * p;
    cblas_dgemv(CblasRowMajor, CblasNoTrans, m_m, m_n, 1., m_Ap.data(), m_n,
                p.data(), 1, 0., Ap.data(), 1); //slow boy

    // alpha = rsold / (p' * Ap);
    auto alpha = rsold / std::max(cblas_ddot(m_n, p.data(), 1, Ap.data(), 1), rsold *
        NEApRZERO);

    // x = x + alpha * p;
    cblas_daxpy(m_n, alpha, p.data(), 1, x.data(), 1);

    // r = r - alpha * Ap;
    cblas_daxpy(m_n, -alpha, Ap.data(), 1, r.data(), 1);

    // rsnew = r' * r;
    auto rsnew = cblas_ddot(m_n, r.data(), 1, r.data(), 1);

    if (std::sqrt(rsnew) < m_tolerance)
      break;

    auto beta = rsnew / rsold;
    // p = r + (rsnew / rsold) * p;
    tmp = r;
    cblas_daxpy(m_n, beta, p.data(), 1, tmp.data(), 1);
    p = tmp;

    rsold = rsnew;
  }
}
```

Listing 3: Sequential conjugate gradient code

```
1
2   // rsold = r' * r;
3   auto rsold = cblas_ddot(m_m, r.data(), 1, r.data(), 1);
4   MPI_Allreduce(MPI_IN_PLACE, &rsold, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
5
6   // for i = 1:n (number of rows in original matrix)
7   int k = 0;
8   for (; k < max_iter; ++k)
9   {
10    // retrieve p from all the threads
11    retrieve_and_concatenate(p);
12
13    // Ap = A * p;
14    multiply_mat_vector(p, Ap);
15
16    // alpha = rsold / (p' * Ap);
17    auto w = cblas_ddot(m_m, p.data() + start_row, 1, Ap.data(), 1);
18    MPI_Allreduce(MPI_IN_PLACE, &w, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
19    auto alpha = rsold / std::max(w, rsold * NEARZERO);
20
21    // x = x + alpha * p;
22    // only update relevent part of x
23    cblas_daxpy(m_m, alpha, p.data() + start_row, 1, x.data() + start_row, 1);
24
25    // r = r - alpha * Ap;
26    cblas_daxpy(m_m, -alpha, Ap.data(), 1, r.data(), 1);
27
28    // rsnew = r' * r;
29    auto rsnew = cblas_ddot(m_m, r.data(), 1, r.data(), 1);
30    MPI_Allreduce(MPI_IN_PLACE, &rsnew, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
31
32    // if sqrt(rsnew) < 1e-10
33    //   break;
34    if (std::sqrt(rsnew) < m_tolerance)
35      break; // Convergence test
36
37    auto beta = rsnew / rsold;
38    // p = r + beta * p;
39
40    tmp = r;
41    cblas_daxpy(m_m, beta, p.data() + start_row, 1, tmp.data(), 1);
42    for (int i = 0; i < get_number_rows(); ++i)
43    {
44      p[start_row + i] = tmp[i];
45    }
46
47    rsold = rsnew;
48
49  }
50
51  MPI_Barrier(MPI_COMM_WORLD);
52
53  // can be replaced by a gather on the "master" thread
54  MPI_Allgatherv(
55      MPI_IN_PLACE,
56      -1, MPI_DOUBLE,
57      x.data(),
58      // counts, displacements
59      counts.data(), displacements.data(),
60      MPI_DOUBLE,
61      MPI_COMM_WORLD);
```

Listing 4: Main loop for CG implementation with MPI

Overview of the CUDA Implementation

```
    for (; k < max_iter; ++k)
    {
        // temp = A*p
        matrix_vector<<<matrix_grid_size, matrix_block_size, shared_mem>>>(A, p, temp, m_n);

        // scalar_temp = p^T temp
        cublasDdot(handle, m_n, p, 1, temp, 1, scalar_temp);
        // alpha = rsold / scalar_temp
        div_scalar<<<1, 1>>>(rsold, scalar_temp, alpha);

        // x = x + alpha*p
        scale_add_vector<<<vector_grid_size, vector_block_size>>>(x, p, alpha, x, m_n);

        // r = r - alpha*Ap
        scale_subtract_vector<<<vector_grid_size, vector_block_size>>>(r, temp, alpha, r, m_n
            );

        // rsnew = r^T r
        cublasDdot(handle, m_n, r, 1, r, 1, rsnew);

        // check convergence
        // copy rsnew to host to do the check
        cudaMemcpy(&r_norm, rsnew, sizeof(double), cudaMemcpyDeviceToHost);
        if (DEBUG && k % 100 == 0)
        {
            std::cout << "\t[STEP " << k << "] residual = " << std::scientific
                      << std::sqrt(r_norm) << "\r" << std::endl;
        }

        if (std::sqrt(r_norm) < m_tolerance)
        {
            break;
        }

        // beta = rsnew / rsold
        div_scalar<<<1, 1>>>(rsnew, rsold, beta);

        // p = r + beta*p
        scale_add_vector<<<vector_grid_size, vector_block_size>>>(r, p, beta, p, m_n);

        // rsold = rsnew
        copy_scalar<<<1, 1>>>(rsnew, rsold);
}
```

Listing 5: Main loop for CG implementation in CUDA