Parallel Implementation of Multiprecondioned Conjugate Gradient Methods

Hao Chen
Computer Science Department, University of British Columbia

December 27, 2022

Abstract

In this report, we present a parallel implementation of the multipreconditioned conjugate gradient method proposed in [1] using Erlang and CUDA for the solution of the Poisson equation on a 2D square domain. The subproblem was then solved using Jacobi method on GPU employing techniques such as tiling and halo cells. Multiple times of the kernel are needed. For matrix operations in erlang, we wrapped cBLAS functions. Our performance evaluation showed that there exists an optimal number of subdomains for our parallel implementation of the algorithm, and we discussed the potential causes and solutions of this behavior. To further optimize the performance of the subproblem solver for small N, we adopted the persistent thread programming style, which reduced the number of global memory accesses and kernel launches, resulting in an approximately one-fold speedup.

1 Background

In this section, we introduce the model problem and the algorithms.

1.1 Main Problem

In the following, we consider the problem: let $\Omega = [0,1] \times [0,1]$

$$\begin{aligned} -a(x,y)u_{xx}(x,y) - b(x,y)u_{yy}(x,y) = f(x,y) \qquad & (x,y) \in \Omega, \\ u(x,y) = 0 \qquad & (x,y) \in \partial\Omega. \end{aligned} \tag{1}$$

where $a(x,y) > 0$ and $b(x,y) > 0$. We then use finite difference method to discretize the problem, we let $x_i = ih, y_j = jh, i, j = 0, 1, \ldots, N-1, N$, where $h = 1/N$ and $N \geq 3$ is an integer.

By replacing

$$\begin{aligned} u_{xx}(x_i,y_j) &\approx \frac{u_x(x_i+h,y_j) - u_x(x_i,y_j)}{h} \approx \frac{u(x_{i+1},y_j) - 2u(x_i,y_j) + u(x_{i-1},y_j)}{h^2} \\ u_{yy}(x_i,y_j) &\approx \frac{u_y(x_i,y_j+h) - u_y(x_i,y_j)}{h} \approx \frac{u(x_i,y_{j+1}) - 2u(x_i,y_j) + u(x_i,y_{j-1})}{h^2} \end{aligned} \tag{2}$$

And denote unknowns $U_{i,j} = u(x_i,y_j)$ and known coefficients $A_{i,j} = a(x_i,y_j), B_{i,j} = b(x_i,y_j), F_{i,j} = f(x_i,y_j)$, we then get the following linear system:

$$\begin{aligned} -A_{i,j}\left(U_{i+1,j} - 2U_{i,j} + U_{i-1,j}\right) & \\ -B_{i,j}\left(U_{i,j+1} - 2U_{i,j} + U_{i,j-1}\right) = h^2 F_{i,j} \quad & i = 1, \ldots, N-1, j = 1, \ldots, N-1 \\ U_{i,0} = 0 \qquad & i = 0, 1, \ldots, N \\ U_{i,N} = 0 \qquad & i = 0, 1, \ldots, N \\ U_{0,j} = 0 \qquad & j = 0, 1, \ldots, N \\ U_{N,j} = 0 \qquad & j = 0, 1, \ldots, N \end{aligned} \tag{3}$$

We'll then use multipreconditioned conjugate gradient method to solve the above linear system.

1.2 Multiprecondioned Conjugate Gradient Method

The multipreconditioned conjugate gradient (MCG) method was proposed by Bridson and Chen in [1] as a variant of the traditional preconditioned conjugate gradient method for the solution of linear systems of equations of the form $Ax = b$. In the traditional preconditioned conjugate gradient method, a single preconditioner M is used to "correct" the search direction vector p_k at each iteration, requiring the solution of a linear system of the form $Mp = q$. In contrast, the multipreconditioned conjugate gradient method utilizes multiple preconditioners M_i to generate multiple search directions $P_k = [p_k^1|p_k^2|p_k^3|\dots]$ at each iteration, requiring the solution of multiple linear systems of the form $M_i p = q$. It has been shown under certain conditions that this approach can yield superior results. While the authors of [1] mentioned the possibility of parallelizing the MCG method, they did not present a parallel implementation.

1.3 Domain Decomposition

Domain decomposition is a widely used technique to solve the PDE problems. The basic idea is: we can divide the domain Ω into several overlapping domains $\Omega_1, \Omega_2, \dots, \Omega_n$ and $\cup_i \Omega_i = \Omega$. For our problem, we can naturally get multiple preconditioners from the subdomain by the following steps:

Denote the right hand side force as $f(x, y)$, we first restrict $f(x, y)$ on our subdomain Ω_i by letting

$$\begin{aligned} f_{\Omega_i}(x, y) &= f(x, y) \quad \text{if } (x, y) \in \Omega_i \\ f_{\Omega_i}(x, y) &= 0 \quad\quad\;\; \text{if } (x, y) \notin \Omega_i \end{aligned} \tag{4}$$

Then, we solve a subproblem

$$\begin{aligned} -a(x, y)u_{xx}(x, y) - b(x, y)u_{yy}(x, y) &= f_{\Omega_i}(x, y) \quad (x, y) \in \Omega_i, \\ u(x, y) &= 0 \quad\quad\;\; (x, y) \in \partial\Omega_i. \end{aligned} \tag{5}$$

We denote the solution as u_{Ω_i}, we then prolongate this solution to the whole domain Ω by letting

$$\begin{aligned} u(x, y) &= u_{\Omega_i}(x, y) \quad \text{if } (x, y) \in \Omega_i \\ u(x, y) &= 0 \quad\quad\;\; \text{if } (x, y) \notin \Omega_i \end{aligned} \tag{6}$$

In the standard domain decomposition method, the boundary term in (6) will not be zeros but the information on the nearby subdomains. But in our case, we don't have to do that because MCG method can handle it automatically[4].

The above equations are in the continuous level. We can also discretize them by using similar discretization in (3), and the subproblem is smaller and can be placed on GPU to solve.

The following is an example of dividing an domain into 4 subdomains. The detailed code is in main.erl.

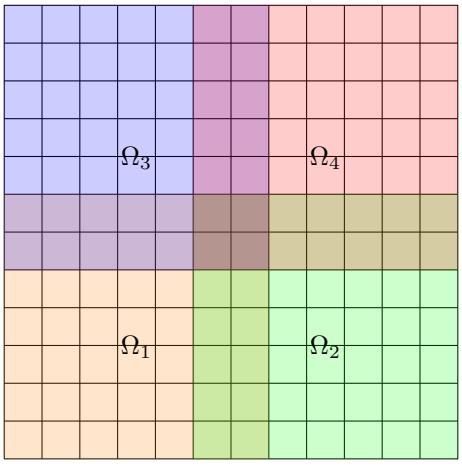


Figure 1: An example of dividing a domain into 4 overlapping subdomains

1.4 Jacobi Iteration

Jacobi method is used in our subdomain solver, the idea of which is based on matrix splitting.

Given a matrix A, A can be written as $A = D - M$, where matrix D is the diagonal elements. The jacobi iteration of solving $Ax = b$ given an initial guess x^0 can then be written as

$$x^{k+1} = D^{-1}(Mx^k + b)$$

For our problem, this method will converge and will roughly take $5MN$ iterations where M and N are the subdomain size parameters.

2 Parallel Implementation of the MCG Method

2.1 Algorithm

Algorithm 1 MCG Method

Given the right hand side b, the number of subdomains K, the subdomain solvers $S_i, i = 1, \ldots, K$, the maximum number of iterations MAXN, the tolerance ϵ.
$r_0 = b_0, \quad x_0 = \mathbf{0}$
$p_1^j = z_1^j = S_j(r_0), \quad \text{for} \quad j = 1, \ldots, K$
$P_1 = [p_1^1 | p_1^2 | \ldots | p_1^K]$
$\alpha_1 = (P_1^T A P_1)^{-1}(P_1^T r_0)$
$x_1 = x_0 + P_1\alpha_1$
$r_1 = r_0 - AP_1\alpha_1$
for $i = 1, \ldots,$ MAXN **do**
    $Z_{i+1} = [S_1(r_i) | S_2(r_i) | \ldots | S_K(r_i)]$
    $P_{i+1} = Z_{i+1} - \sum_{j=1}^{i} P_j (P_j^T A P_j)^{-1} P_j^T A Z_{i+1}$
    $\alpha_{i+1} = (P_{i+1}^T A P_{i+1})^{-1}(P_{i+1}^T r_i)$
    $x_{i+1} = x_i + P_{i+1}\alpha_{i+1}$
    $r_{i+1} = r_i - AP_{i+1}\alpha_{i+1}$
    if $\|r_{i+1}\|_2 < \epsilon$ **then**
        break
    end if
end for

Algorithm 2 Subdomain Solver $S_j(r)$ using Jacobi methods

Given the right hand side vector r, the subdomain coefficient matrix A and B and subdomain Ω_j size M and N.
$u \leftarrow \mathbf{0}, v \leftarrow \mathbf{0}$
for $k = 1, \ldots, 5MN$ **do**
    for $i = 1, \ldots, M$ **do**
        for $j = 1, \ldots, N$ **do**
            $u_{i,j} = \frac{1}{2(A_{i,j}+B_{i,j})}(r_{i,j} - B_{i,j}v_{i,j-1} - B_{i,j}v_{i,j+1} - A_{i,j}v_{i-1,j} - A_{i,j}v_{i+1,j})$
        end for
    end for
    Swap u and v
end for

We modified the algorithm in [1] to adapt to our PDE problem and get algorithm 1. The subdomain solver $S_j(r)$ is a Jacobi method, which is a simple iterative method. We use the Jacobi method to solve the subproblem in (6) because it is easy to parallelize. The algorithm is outlined in algorithm 2.

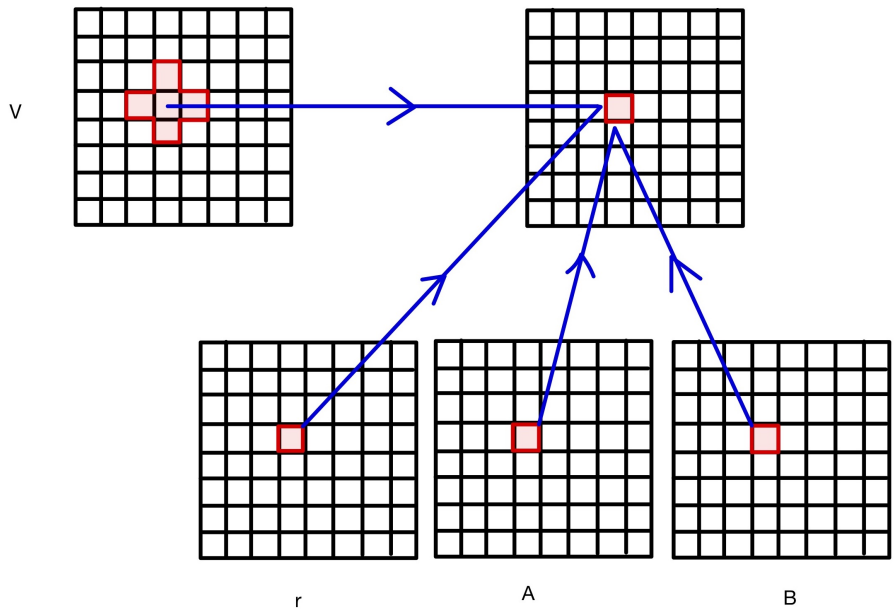


Figure 2: Accessing patterns of the subdomain solver

To do matrix operations in Erlang, we used ErlNif. The structure for a matrix is stored as a tuple {M, N, Data}, where M and N are the number of rows and columns, and Data is a Erlang Binary that stores the elements in double precision of the matrix in column-major order. We wrapped a few functions in CBLAS and LAPACK. The detailed code is in tools.c.

2.2 Parallel Implementation

We first consider the parallel implementation of Algorithm 1.

At the beginning, we need to send the subdomain information including the matrix A, B and subdomain size to every node and stored them in subdomain_info state.

Line 4 and 8 are the most time consuming part of the algorithm. We utilize a cluster of LINXX servers as worker nodes, with each server responsible for solving the subproblem in (6) for a single subdomain and returning the result to the master node. To minimize communication overhead, we transmit only the restriction of r_i on each subdomain rather than the full vector r_i, and similarly collect only the restriction of the solution on each subdomain, requiring prolongation on the master node. In cases where the number of available LINXX servers is insufficient, it may be necessary to assign multiple subdomains to a single server.

During our tests, we observed that lines 10 and 11 also contribute significantly to the overall runtime of the algorithm. To address this issue, we then use wtree which are only on the master node to reduce communication overhead. For line 10, we perform a reduce operation, with each leaf node responsible for a different $P_j(P_j^TAP_j)^{-1}P_j^TAZ_{i+1}$ (Here, in the coding, $(P_j^TAP_j)^{-1}$ is stored in every iteration) Matrix addition is used to reduce the results. And for matrix multiplication $A \times M$, we assign each leaf node a column of M and do matrix multiplication. The result is then reduced to the root node. However, for large domain sizes, such operations can become computationally intensive, and it may be necessary to set up a wtree across multiple machines and utilize cuBLAS instead of cBLAS for matrix operations.

Now we consider the parallel implementation of the subdomain solver $S_j(r)$ in Algorithm 2 using CUDA. The pattern as shown in figure 2 is very similar to stencil which allows us to use techniques like tiles and halo cells, except that we now have more varibles r, A, B which will decrease the CGMA. And also, we have to restart the kernel for $5MN$ times.

Note that even if our subdomain solvers are not very accurate, we can still have a convergent algorithm since the master will perform correction. Thus, instead of using double, we only use single precision in the

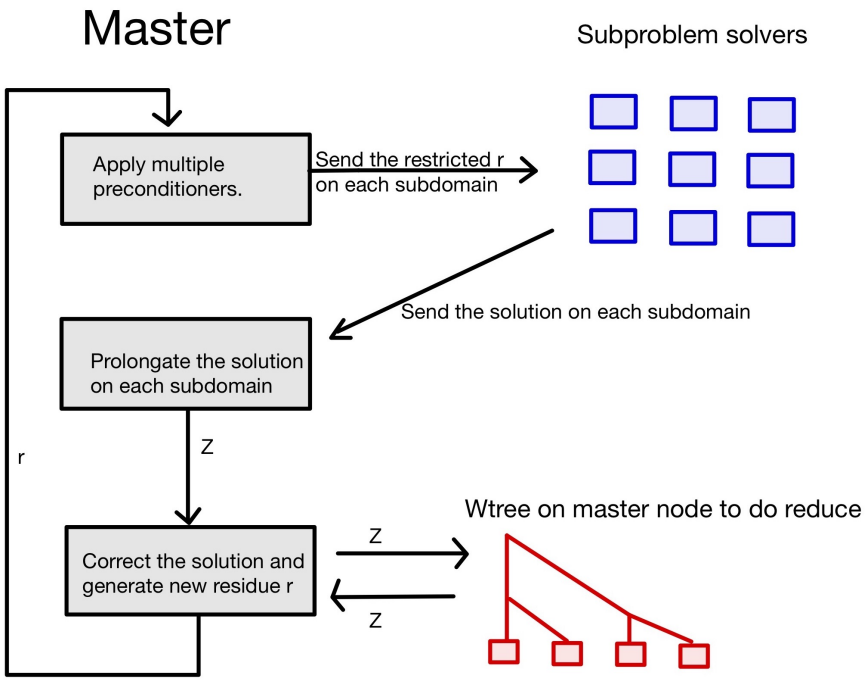


Figure 3: The process of MCG

subproblem solver.(The correction step is still done using double)

We fist consider how to deal with the matrix $\{A_{ij}\}$ and $\{B_{ij}\}$. Accessing them will result in lots of global memory access. Due to the fact the subdomain solver can be inaccurate, we can set all the elements of the coefficient matrix as one of the elements, in this way, we only need 2 floats for the coefficients. We'll code two kernels and compare our results.

The overhead of restarting kernels for $5MN$ times is big, since shared memory has a life time of a kernel. Restarting for multiple times, means we have to access global memory for multiple times. In [9], they call this type of iterative methods as iterative memory-bound GPU applications and proposed using persistent kernels to fix this. The idea of which is having an kernel to perform iterative loop. We'll employ similar ideas in the later section.

The overall process can be summarized in figure 3.

2.3 Performance Analysis

If the entire domain is divided into $K \times K$ subdomains and the time spent on initializing the subdomains is ignored, the computational complexity of the algorithm in the i-th iteration can be described as follows.

In i-th iteration, every subdomain solver requires $O((N/K)^4)$ computing time and sends $O((N/K)^2)$ messages per subdomain. And then, we approximately need $O(K^3 + N^2 + iKN^2/P_2)$ to do correction where P_2 is the number of processors on the master node.

In total, if P_3 is big enough and we need $O((N/K)^4 + K^3 + iN^2)$ computing time and $O(N^2)$ messages from subdomain node to master node per iteration. The sequential version requires $O(K^2(N/K)^4 + K^3 + N^2 + iKN^2)$ computing time per iteration.

However, we can't make K arbitrarily large. If K is too large, the iteration number will become large and the sequential time will increase. And also, if we use CUDA, the computing time on each subdomain may be much smaller and the time spent on correction may dominant. In our case, K is less than 5, meaning at most 16 subdomains.

The term i in the computing time will be important if the number of subdomains increase, the iteration counts will be larger, thus making the sequential part increase.

3 Results

The numerical tests are already done in [1]. We'll focus on showing correctness of our implementation and evaluating the performance of our algorithm.

3.1 Experiments on the Subdomain Solver

In this subsection, we aim to evaluate the performance of our subdomain solver algorithm 2 by fixing $M = N$ such that the subdomain is a square. The implementation is similar to that of the stencil algorithm [6], with the exception of the need to restart the kernel for $5N^2$ times, which we include in our measurements.

We first implement the algorithm using global memory to store A, B, r, u, v, and we use tiles of 32 by 32 to reduce the global memory access of v. The CUDA implement is denoted as implementation 1. We also implemented a sequential version as comparison. The performance results are shown in table 1. From the table, we can see when N is small, the speedup is not big. This is because we didn't use all the SMs at first.

N	50	100	150	200	250
CPU_Time	0.431416	6.801401	34.214719	108.662510	264.224043
GPU_Time	0.029208	0.133410	0.381404	0.766724	1.640376
Speedup	14.7	50.9	89.7	141.7	161.0

Table 1: Performance of the subdomain solver 1 with flexible A and B

We then fix A and B to be constant for all the elements of A and B, by doing which, we reduced global memory access of A and B. We also do this for sequential version. The perfromance results are shown in table 2, from which, we can see both the CPU and GPU time are decreased compared with 1.

N	50	100	150	200	250
CPU_Time	0.352007	5.556259	27.882599	88.887841	216.126253
GPU_Time	0.027507	0.116200	0.332678	0.725131	1.379961
Speedup	12.8	47.8	83.8	122.5	156.6

Table 2: Performance of the subdomain solver 2 with constant for all the elements of A and B

3.2 Experments on MCG

We implement MCG and run on LINXX severs. The amount of workers on master node is set to 40 and we change the number of subdomains. For the subdomain solver, we use subdomain solver 2. The time to reduce the relative error $\frac{\|Ax-b\|_2}{\|b\|_2}$ to 10^{-6} is shown in table 3.

Subdomain Number	2×2	2×3	3×3	3×4	4×4
Domain size 500×500	55.85	46.56	40.03	70.34	99.16
Domain size 300×300	9.12	8.56	7.83	11.8	13.90
Domain size 200×200	2.76	2.92	2.53	4.26	4.56

Table 3: Computing time(seconds) for MCG to reduce the relative error to 10^{-6}

The computing time firstly decreases by increasing the number of subdomains and then increases when the subdomain number is 3×3. Following reasons account for this behavior: 1. As we increase the number of subdomains, we'll need more iterations as shown in figure 4 and there should be a balance between the domain size and the number of subdomains. 2.The sequential part of the algorithm will increase as we increase the number of subdomains. Since the subdomain solver is rapid. The time for line 10-14 in algorithm 1 can not be ignored as we didn't use CUDA to implement the operations like transpose multiply and matrix multiply.

Figure 4: Relative errors of MCG with different subdomain numbers when $N = 300$

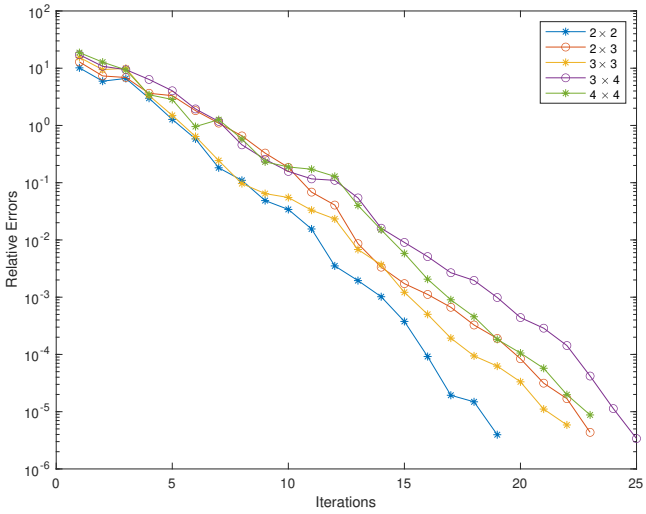


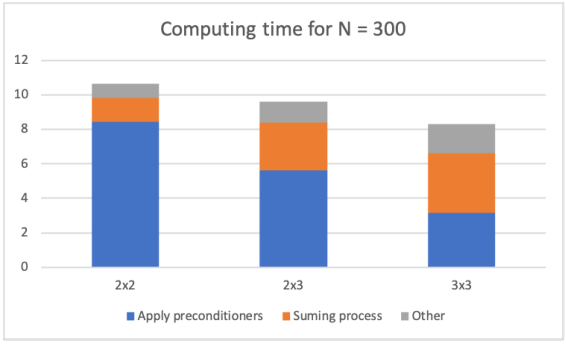
Figure 4 illustrates that our algorithm converges and the implementation is correct, as the residual decreases over time.

To improve the performance of MCG when subdomain number increases, we wrapped matrix multiplication using cuBLAS and the performance is still not satisfactory. To understand the cause of this issue,, we reran the program and record the computing time for each process. The result is shown in the figure 5. Upon further analysis of the program's execution, it was found that the time spent on applying preconditioners was decreasing, but there was an increase in the time spent on the summing process (line 10 in algorithm 1). This could be due to the fact that the LINXX server only has 8 cores, while the wtree worker size was set to 40. Even if cuBLAS is used, the fact that this operation is only performed on the master node means that the full capabilities of the GPU on other nodes may not be utilized. To address this issue, it may be necessary to consider alternative ways to parallelize the summing process.

One potential approach is to distribute the k-th column of the equation in line 10, resulting in $p_{i+1}^k = z_{i+1}^k - \sum_{j=1}^{i} P_j (P_j^T A P_j)^{-1} P_j^T A z_{i+1}^k$, and assign each machine to work on a single column but doing this will introduce additional communication overhead. Another option is to use a more powerful machine as the master node. A third option is to truncate the sum in line 10, which will result in a less accurate solution but will reduce the time spent on the summing process. And in other applications like ADI preconditioners, the sum can be truncated without losing accuracy[1].

In addition, it should be noted that the vector z_{i+1}^k is sparse, with non-zero elements at the subdomain nodes. This sparsity may be leveraged through the use of sparse vectors to optimize certain matrix operations.

Figure 5: Computing time for $N = 300$ with different subdomain numbers



4 Improvements on the Subdomain Solver

In the implementation of the subdomain solver presented in section 2.2, the need to restart the kernel in each iteration introduces several challenges:

- Each kernel is memory-bounded, requiring $O(N^2)$ global memory access and $O(N^2)$ FLOPs.
- The overhead of launching a kernel is significant when it needs to be restarted $O(N^2)$ times.
- In every kernel, we need to read the same data from the global memory.
- The result of one kernel is also the input of the next kernel.

Due to these problems, we hope to have one kernel that do all the iterations. Problems arsing from this:

- We need to do inter-block synchronization.
- The shared memory is limited.
- CUDA cores are limited.
- Registers are limited.

The authors of [9] proposed a framework using persistent threads to implement iterative solvers on GPU for memory-bounded problems, called PERsistent KernelS (PERKS). In this work, we borrow ideas from their paper and implement a subdomain solver using persistent threads.

4.1 Persistent Thread

Persistent thread is a programming style in CUDA that aims to extend the lifetime of a thread to the entire duration of an application. This is achieved usually by launching limited number of blocks that can reside on all the SMs and writing the kernel in a producer-consumer mode, with work being loaded from a work queue. In contrast, traditional CUDA programming styles may involve launching blocks of threads in a way that allows some blocks to finish their work before others are launched, due to the architecture of the GPU[5].

In persistent thread style programming, it may be necessary to synchronize between blocks in order to achieve certain goals. The cooperative group feature, introduced in CUDA 9.0, can be used for this purpose. As noted in [2], it is important to ensure that all blocks are running in order to perform synchronization between blocks using cooperative groups.

4.2 Persistent Thread Style Jacobi Method

In this subsection, we describe the implementation of our subdomain solver (Jacobi method) using the persistent thread programming style.

The following code is the pseudo code for the Jacobi method using persistent thread programming style:

```
__global__ void jacobi(float *rhs, float *xx1, float *xx2, int M, int N) {
    __shared__ float xx1_s[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float xx2_s[BLOCK_SIZE][BLOCK_SIZE];
    // some code to Load data into the shared memory
    __syncthreads();
    for (int k = 0;k<itrs;k++){
      //do 5-point jacobi iteration, and write to xx2_s
      jacobi(xx1_s,xx2_s);

      //swap pointers xx1 and xx2, we need two global variables for communication
      swap(&xx1,&xx2);

      //update halo cells for this block to xx1
      // xx1 will serve as a communication tool between blocks
      storeHaloCells(xx1,xx2_s);

      //sync inter-blocks
```

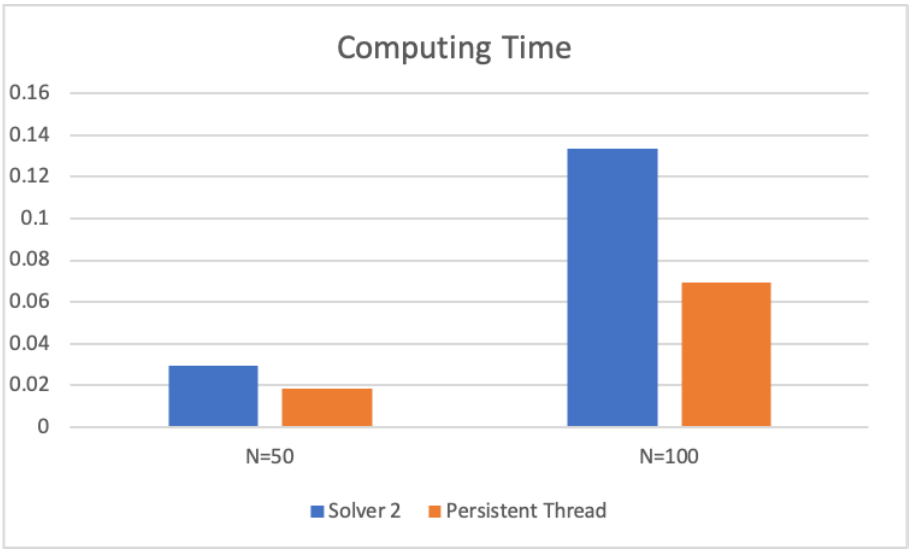


Figure 6: Computing time for Solver 2 and PT style Jacobi methods

```
        grid.sync();

        //swap xx1_s and xx2_s
        swap(&xx1_s[ty][tx],&xx2_s[ty][tx]);

        //update halo cells in xx1_s from global memory xx1
        updateHaloCells(xx1_s, xx1);
        __syncthreads();
      }
      //write the solution to global memory
    }
```

A notable distinction between the implementation using persistent thread programming style described in this subsection and the implementation described in the 'Implementation' section (Section 2.2) is that the former approach requires the launch of a single kernel, whereas the latter needs the launch of multiple kernels.

By implementing the Jacobi method using persistent thread programming, we were able to reduce the usage of global memory per iteration from $O(N^2)$ to $O(KN)$, where K is the number of blocks. This optimization was achieved by swapping xx1_s and xx2_s in shared memory instead of loading them from the global memory and we only need to update the halo cells between blocks using global memory.

However, this optimization comes with some trade-offs. Firstly, the maximum value of N is limited due to the constraints on the number of threads and the amount of shared memory. While thread coarsening can be used to increase N, this will also increase the number of registers. Secondly, the overhead of synchronizing between blocks is also considerable, which may impact performance. Though N needs to be small, we could use MCG algorithm to divide the problem into many subproblems that can fit in this approch.

We evaluate the performance of this new approach in comparison to the previous solver 2 in Section 3. The results are shown in figure 6. For $N = 50$, the time required by the new approach was 0.018518 seconds, 0.6334 of the original time, and for $N = 100$, 0.069518 seconds, 0.5211 of the orignal time. When $N = 150$, we don't have enough threads and we can use thread coarsening, for simplicity, we didn't implement that.

5 Conclusion

In this report, We implemented the MCG algorithm using erlang and CUDA.

For the subproblem solver, we initially implemented it on CUDA using standard techniques such as halo cells and tiles, but this approach requires multiple kernel restarts for synchronization between blocks, and being memory-bounded. To address these issues, we implemented the subproblem solver using the persistent thread programming style, which resulted in an improvement in performance, approximately one times faster. However, this new approach is only suitable for relatively small values of N, due to the constraints on the number of threads and shared memory, and also incurs a significant cost for inter-block synchronization.

The performance of the MCG algorithm was not as satisfactory as we had anticipated: the computing time reached the minimum when the subdomain is 3×3, and this is due to the summing part of the algorithm and more iterations needed when increasing the number of subdomains. To address this issue, we could wrap cuBLAS instead of cBLAS to facilitate the matrix operations in the sequential portion of the algorithm, use a better master machine, truncate the sum sequences and adopt different parallel strategies.

6 Code

We briefly summarize the purpose for each file.

main.erl is the main erlang file contains the implementation of MCG algorithm, and the entrance function is mpcg.

mat.erl contains the functions used in main.erl including matrix operations and subdomain solver.

jacobi_solve_gpu.cu is the CUDA implementaion of the jacobi method for the subdomain solver. We implemented two versions for whether A and B are flexible.

poisson.c contains operations related to matrix A and domain information.

tools.c is the wrapper for CBLAS functions including matrix multiplication, matrix inversion, norm and linear solver.

tools_nif.c contains the definition for the erlnif.

jacobi_p.cu the persistent thread style implementation of the jacobi method.

I also modified workers library to spawn nodes on remote machines.

References

[1] Robert Bridson and Chen Greif. A multipreconditioned conjugate gradient algorithm. *SIAM Journal on Matrix Analysis and Applications*, 27(4):1056–1068, 2006.

[2] NVIDIA Corporation. Cuda c/c++ programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[3] Chen Greif, Tyrone Rees, and Daniel B Szyld. Multi-preconditioned gmres. *Technical report: UBC CS TR-2011–12*, 2011.

[4] Chen Greif, Tyrone Rees, and Daniel B Szyld. Additive schwarz with variable weights. In *Domain Decomposition Methods in Science and Engineering XXI*, pages 779–787. Springer, 2014.

[5] Kshitij Gupta, Jeff A Stuart, and John D Owens. *A study of persistent threads style GPU programming for GPGPU workloads.* IEEE, 2012.

[6] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach.* Morgan kaufmann, 2016.

[7] Tanguy Losseau and Peter Van Roy. Numerl: Efficient vector and matrix computation for erlang.

[8] Barry F Smith. Domain decomposition methods for partial differential equations. In *Parallel Numerical Algorithms*, pages 225–243. Springer, 1997.

[9] Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, and Satoshi Matsuoka. Persistent kernels for iterative memory-bound gpu applications. *arXiv preprint arXiv:2204.02064*, 2022.