KACZMARZ-INSPIRED CUDA SOLVERS FOR LSEs ARISING FROM PDEs

Alexandre Faroux[†]*, Viktor Fukala*[⋆]*, Clemens Herfarth*[†]*, Zeno Hug*[†]*, Cédric Zeiter*[†]

[⋆]ETH Zürich, Department of Computer Science, Zurich, Switzerland
[†]ETH Zürich, Department of Mathematics, Zurich, Switzerland

ABSTRACT

We present two GPU-parallelized solvers[1] for sparse linear systems arising from Galerkin discretizations of PDEs, based on the Kaczmarz method. Using CUDA[2], we optimize and benchmark a non-overlapping row approach and an adapted CARP-CG algorithm. Our experiments show the potential of parallelizing the Kaczmarz algorithm. However, our developed solvers do not reach the runtimes of LSE solvers from available libraries.

1. INTRODUCTION

Linear systems of equations (LSEs) often arise in scientific and engineering applications, particularly from the discretization of partial differential equations (PDEs) in fields like fluid dynamics, electromagnetics, and finance [1]. The finite element method (FEM), especially the Galerkin method, generates large, sparse Galerkin matrices requiring efficient LSE-solvers [2].

This report focuses on the Kaczmarz method [3], an iterative solver for sparse LSEs, and its parallelization on GPUs to leverage their massive parallelism. We explore two GPU-parallelized approaches: one based on the CARP-CG algorithm [4] and another utilizing the non-overlapping structure of Galerkin matrix rows. These methods target large, sparse systems from boundary value problems (BVPs). While some have investigated GPU implementations of Kaczmarz [5], it remains a relatively underexplored area, offering significant potential for innovation.

Through benchmarking and comparison against established GPU solvers like NVIDIA cuDSS[3], this study evaluates the potential of Kaczmarz-based solvers for sparse LSEs, addressing the challenges of GPU parallelization.

[1]https://github.com/cedriczeiter/DPHPC-2024-Kaczmarz
[2]https://docs.nvidia.com/cuda/cuda-c-programming-guide/
[3]https://developer.nvidia.com/cudss

2. BACKGROUND

2.1. KACZMARZ METHOD

The Kaczmarz method relies on Kaczmarz sweeps [4], given by the KSWP(A, b, x, λ) operator:

Algorithm 1: KSWP(A, b, x, λ)

Input: $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $x \in \mathbb{R}^n$, $\lambda \in \mathbb{R}$
Output: $y \in \mathbb{R}^n$

Initialize $y \leftarrow x$;
for *each row i of matrix A* **do**
  $y \leftarrow y + \lambda \frac{b_i - A_i \cdot y}{\|A_i\|^2} A_i^\top$;
return y ;

By iteratively refining an arbitrary initial guess x_0 using the KSWP operator, the Kaczmarz method always converges for consistent systems to the least square norm solution [6]. This works by projecting onto the solution space of each row's equation [3].

2.2. CONJUGATE-GRADIENT ACCELERATED KACZMARZ

The basic Kaczmarz algorithm can be conjugate-gradient accelerated [4]. The resulting algorithm, called CGMNC (Algorithm 2), relies on the operator DKSWP, which consists of two consecutive Kaczmarz sweeps. The operator BKSWP(A, b, x, λ) is nearly identical to KSWP(A, b, x, λ), except that the rows of A are traversed in reverse order, resulting in a *backward sweep*.

The operator DKSWP first carries out the forward step and then the backward step, resulting in the definition:

$$\text{DKSWP}(A, b, x, \lambda) = \text{BKSWP}(A, b, \text{KSWP}(A, b, x, \lambda), \lambda)$$

2.3. CARP-CG

[4] proposes a parallelized algorithm of the CGMNC algorithm presented in Section 2.2 called CARP-CG (component averaged row projections). The key idea of

Algorithm 2: CGMNC

Input: $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $x^0 \in \mathbb{R}^n$, $\lambda \in \mathbb{R}$
Output: $x^{k+1} \in \mathbb{R}^n$

Initialize $p^0 \leftarrow \text{DKSWP}(A, b, x^0, \lambda) - x^0$;
Initialize $r^0 \leftarrow p^0$;
for $k = 0, 1, 2, \ldots$ *until convergence* **do**
  $q^k \leftarrow p^k - \text{DKSWP}(A, 0, p^k, \lambda)$;
  $\alpha_k \leftarrow \frac{\|r^k\|^2}{\langle p^k, q^k\rangle}$;
  $x^{k+1} \leftarrow x^k + \alpha_k p^k$;
  $r^{k+1} \leftarrow r^k - \alpha_k q^k$;
  $\beta_k \leftarrow \frac{\|r^{k+1}\|^2}{\|r^k\|^2}$;
  $p^{k+1} \leftarrow r^{k+1} + \beta_k p^k$;
return x^{k+1} ;

this algorithm is to partition all equations into subsets $B_1, B_2, \ldots, B_t$, and assign each subset to a thread. Each thread works on its own subset of equations.

At each timestep k, the global current solution x^k is obtained by averaging the local results of all relevant threads. We define:

$$L_j = \{1 \le l \le t | x_j \text{ appears in an equation of } B_l\}$$

With $x^{k,l}$ representing the local solution of thread l at iteration k, the component-averaging operation CA can be defined as:

$$CA(x^{k,1}, \ldots, x^{k,t})_j = \frac{1}{|L_j|} \sum_{q\in L_j} x_j^{k,q}. \qquad (1)$$

The operator DCSWP then consists of two Kaczmarz sweeps of each thread, with the second sweep traversing the equations in reverse order. Equation 1 is then used to average the local results after each sweep. The CARP-CG algorithm is then identical to CGMNC, with the operator DKSWP replaced by DCSWP in Algorithm 2.

3. METHODS

3.1. PARALLEL PROCESSING FOR NON-OVERLAPPING ROWS

We defined the Kaczmarz sweep as applying the projections for all rows sequentially. To reduce the running time of our algorithm, we wish to compute and apply some of those projections in parallel. In general, this is not immediately possible because each of the projections reads and updates the x vector.

However, it is possible if we establish that the projections only access mutually disjoint parts of the vector x. The projection induced by a particular row only needs to access the vector x at index j if the row has a non-zero entry at column index j. Therefore, if we identify a set of consecutive rows such that the sets of columns where each row is non-zero are disjoint (we call such rows *non-overlapping*), we can compute and apply their projections in parallel.

If we pick an arbitrary sparse matrix, it is uncommon for there to be large *consecutive* sets of non-overlapping rows. Instead, it is often easier to find *non-consecutive* sets of non-overlapping rows. Fortunately, parallelization is possible even in that case:

In fact, for any collection of disjoint sets of (potentially non-consecutive) rows, we can permute the LSE rows to make the rows in each of the given sets consecutive. In practice, instead of explicitly reordering rows in memory, we often just change the order in which we apply the row projections in each Kaczmarz sweep. When doing so, since we know it is equivalent to the standard Kaczmarz sweep on a reordered LSE, we also know we retain the correctness and convergence properties of the standard Kaczmarz method.

Banded Matrices We say that a matrix (a_{ij}) is banded with bandwidth[4] b if $|i - j| > b$ implies $a_{ij} = 0$ for all i, j. Banded matrices with low bandwidth arise naturally in some PDE discretizations, such as that of the Poisson's equation [1]. They can also be obtained using the Cuthill-McKee algorithm [7], which reduces the bandwidth by reordering the rows and columns of a symmetric matrix.

We first focus on banded matrices because their structure makes it easier to find non-overlapping rows. Given a banded matrix with bandwidth b, the key observation is that row i_1 and row i_2 are non-overlapping if $|i_1 - i_2| > 2b$. Using this, we devised two independent schemes to parallelize the Kaczmarz sweep.

In the **first scheme**, we select an integer $n \ge 1$ and introduce two classes of n groups of rows each: $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$. Each group must consist of consecutive rows, at least $2b$ of them, and all the groups should be roughly equal in size. As illustrated in Figure 1, the groups appear in the order $A_1, B_1, A_2, B_2, \ldots, A_n, B_n$. This way, no row overlaps with any row in a different group of the same class because the two rows are separated by at least $2b$ rows in at least one group of the other class between them.

Therefore, each of the n groups in the same class can be processed in parallel by one of n threads. We use a barrier to synchronize the threads after processing a whole group, before groups from the other class start to be processed. This is shown in the right part of Figure 1.

In the **second scheme**, we create $2b + 1$ groups, labeled $P_0, \ldots, P_{2b}$. A row at an index i is assigned to group $P_{i \mod 2b+1}$. See the left part of Figure 2.

Now, any two distinct rows in the same group are non-

[4]Some would also call b the half-bandwidth.

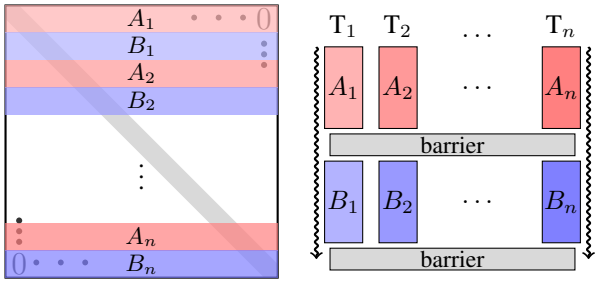

Fig. 1: The first approach of partitioning rows in a banded matrix. Schema of the division of rows into groups on the left. Schema of the parallel processing of the row updates on the right.

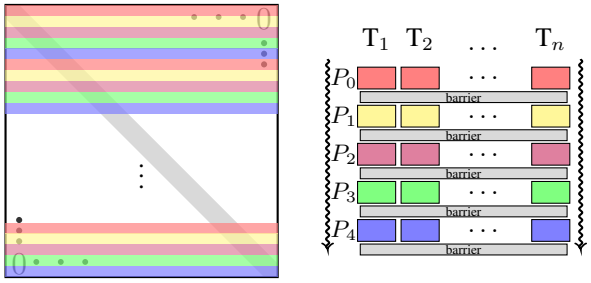

Fig. 2: The second approach of partitioning rows in a banded matrix. Schema of the division of rows into groups on the left. Schema of the parallel processing of the row updates on the right.

overlapping because they are separated by at least $2b$ rows between them. The updates for all rows within the same group can hence be applied in parallel.

As shown in the right half of Figure 2, we can run a fixed pool of threads. For each group P_i, we distribute the row updates equally among those threads and use a barrier to synchronize the threads before processing the next group.

For both schemes, we wrote a serial CPU implementation[5], a parallelized CPU implementation using OpenMP[6], and a GPU implementation using CUDA.

Galerkin Matrices Compared to banded matrices, the arrangement of non-zero entries in Galerkin matrices is much less structured, which makes it more challenging to systematically identify non-overlapping rows. Mimicking the second scheme for banded matrices, there is one possible approach that works for an arbitrary sparse matrix: Define a graph where the vertices correspond to the rows of the matrix and two vertices are adjacent iff the corresponding rows overlap. A coloring of this graph then corresponds to a partitioning of the set of rows into sets of non-overlapping

[5]For the serial implementation, the schemes only differ in the order in which the row projections are applied.

[6]https://www.openmp.org/resources/refguides/

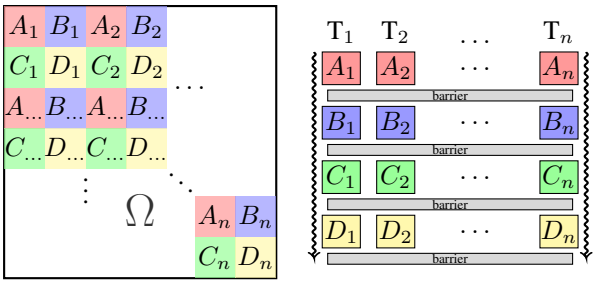

Fig. 3: Left: Division of a 2D BVP domain into sections of $2^2 = 4$ classes denoted A, B, C, and D (e.g. the sections/groups $A_1, \ldots, A_n$ form one class).
Right: Parallel processing of the resulting row groups.

rows, allowing us to execute a scheme like in Figure 2.

Alternatively, we exploit information from the construction process of a Galerkin matrix. This time, we mimic the first scheme for banded matrices – we create several classes of row groups such that rows in different groups of the same class are non-overlapping. We exploit the locality of the discretized linear equations: Using a heuristic (e.g. averaging of coordinates of the involved mesh elements), we assign a point in the BVP domain to each equation to approximate what area of the domain is described by that equation. We expect that rows will overlap only if their assigned points are close together.

Along each dimension of the domain, we calculate the maximum separation between the assigned points of any two overlapping rows. We then divide up the domain along that dimension into sections that are all wider than that. We make the sections alternate between two different classes. The partitioning of the entire domain is the Cartesian product of the partitionings for the individual dimensions so that a domain of dimension D is divided into sections belonging to 2^D different classes. To partition the rows themselves, we have, for each section, the group of all rows whose assigned point lies in that section. During a Kaczmarz sweep we then process groups in the same class in parallel. The BVPs we work with have $D = 2$ as in Figure 3.

3.2. CARP-CG ON GPU

Since CARP-CG is a parallel algorithm, adapting CARP-CG for execution on GPU offers potential for speedup. We implemented a version of CARP-CG which runs on GPU using CUDA. However, we needed to adapt the algorithm in certain ways for this to work. These adaptations are described in this section. We refer to our adapted version of CARP-CG as GPU-accelerated CARP-CG.

Challenges. The CARP-CG method proposed in [4] involves using multiple processors, with each processor maintaining a local version of x. While this has the advantage of

reducing the amount of data transferred between threads, implementing this feature on GPU proved challenging:

- Storing the entire vector x for each thread uses too much memory. We could instead only store the entries of x relevant to the specific thread (given by L_j), but we would need some way to map local entries to global entries of x. We could not find a good way to implement this mapping on GPU.
- Initializing new local copies in shared memory for each call of our GPU kernel is computationally expensive. Storing each copy in global memory leads to high bandwidth usage and stalls for large number of threads.

Adaption. To address these challenges, we developed a workaround: We partition all equations into subsets, each containing a single thread. Consequently, each thread processes only one row, eliminating the need to store local copies of x. Instead, local solutions are averaged directly and written into the global solution vector. This allows us to combine the Kaczmarz sweep and consecutive averaging operation needed for DCSWP into one single operation.

By leveraging this approach, we developed a GPU-adapted algorithm that integrates component averaging and DCSWP techniques. The GPU-accelerated CARP-CG is thus equivalent to CGMNC, with the operator DKSWP replaced by our own GPU-adapted DCSWP, as detailed in Algorithm 3.

Algorithm 3: GPU DCSWP

Input: $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, $x^k \in \mathbb{R}^n$, $\lambda \in \mathbb{R}$
Output: $y \in \mathbb{R}^n$

```
Initialize y ← x ;
for i = 1, 2 do
    for each thread q (1 ≤ q ≤ k) in parallel do
        for every non-zero entry j of A_q do
            AtomicAdd(y_j, (λ/|L_q|) (b_q − A_q·y^{i−1}) / ||A_q||^2 · A_qj);
return y ;
```

With this adapted DCSWP it is then possible to run CARP-CG on GPU.

Relaxation parameter. All updates to x in Algorithm 3 are multiplied by the factor $\frac{\lambda}{|L_q|}$. Although using this factor is consistent with the algorithm described by [4], it requires every thread q to fetch $|L_q|$ from memory every call to the kernel. While running experiments of our GPU-accelerated CARP-CG we noticed:

- The solver converges faster if, for each row, we replace $|L_q|$ with the maximum value $\max_q |L_q|$. This improvement is likely due to reduced global memory reads, as the constant value $\max_q |L_q|$ can be shared across all threads.
- The optimal value for λ, in terms of the least iterations to converge, is proportional to the $\max_q |L_q|$. This allows us to replace the factor $\frac{\lambda}{|L_q|}$ by a single parameter $\tilde{\lambda}$. Experiments regarding this factor are reported in Section 4.

4. EXPERIMENTAL RESULTS

All experiments were conducted on a consistent system configuration, with specifications outlined in Tables 1 and 2. All experiments were conducted in floating-point double precision.

Table 1: Hardware Configuration

Compute Nodes	AMD Ryzen 7 PRO 7840HS CPU (3.80-5.10 GHz), 8 cores
Memory	32 GB DDR5-5600MHz (1 x 32 GB SODIMM)
Accelerators	NVIDIA RTX™ A1000 Notebook-GPU 6GB GDDR6

Table 2: Software Environment

Operating System	Ubuntu 22.04.5 LTS
Compilers	GCC 11.4.0, NVCC 12.6
Programming Languages	C++20, CUDA 12.6
Libraries	Eigen 3.4.0, CuDSS 0.4.0
Compile Flags	O3, DNDEBUG

Discretization of PDEs We perform some benchmarks on discretizations of the following reaction-diffusion PDEs:

$$\text{Problem 1: } -\Delta\mathbf{u} + 1000 = F$$
$$\text{Problem 2: } -\Delta\mathbf{u} + (x^2+y^2)\mathbf{u} = F$$
$$\text{Problem 3: } -\nabla\cdot(xy\nabla\mathbf{u}) + xy\mathbf{u} = F$$

The right-hand side function F is calculated for the preassigned analytical solution $\mathbf{u} = xy(1-x)(1-y)$. We use the capabilities of LehrFEM++[7] to discretize the three PDEs using Lagrangian FEM of degrees for $p = 1, 2, 3$. We perform discretization in the domain $[0, 3]^2$ with Dirichlet boundary conditions $g(\mathbf{x}) = 0$.
We use iterative mesh refinement to generate multiple LSEs for a given problem and degree p, producing systems of varying dimensions that allow us to evaluate how our solvers scale with increasing LSE size.

4.1. PARALLEL PROJECTIONS FOR NON-OVERLAPPING ROWS

Banded Matrices We benchmarked our implementations for banded matrices on randomly generated $d \times d$ matrices with bandwidth b. We show the performance for $d = 2 \cdot 10^5$

[7] https://craffael.github.io/lehrfempp/

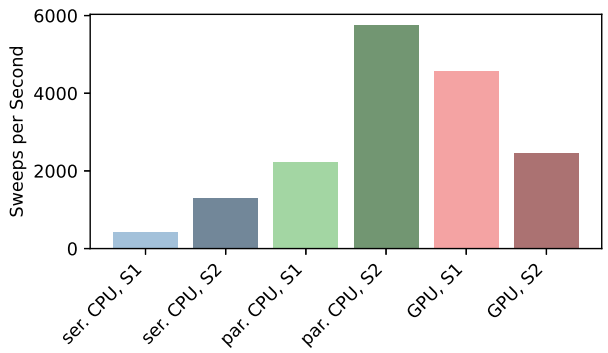

Fig. 4: The Kaczmarz sweep rates of our CPU-serial, CPU-parallel (OpenMP), and GPU (CUDA) banded-matrix implementations with either the first (S1) or the second (S2) row set partition scheme. In the CPU-parallel implementations, we chose the number of threads that achieved the best performance (8 threads for S1 and 6 for S2). We reran each implementation 5 times. Any two running times measured for the same implementation never differed by more than $3\,\%$ and we used the arithmetic mean of the time measurements for the plot.

and $b = 2$. Based on other benchmarks we ran, this value of d appears to be large enough to be representative of the asymptotic behavior of the solvers and it is comparable with the dimensions of some of the Galerkin matrices with which we benchmark our other solvers. In this case, the unmodified Kaczmarz method takes impractically long to converge, so instead of running the algorithms until some convergence criterion is fulfilled, we measured the time needed for a fixed number of $N = 50000$ Kaczmarz sweeps. Figure 4 shows the benchmark results. Due to differences in the order of row projections, these implementations do not produce the exact same results. However, in this benchmark, the L1 norms of the final residual in the various implementations were all within $3\,\%$ of each other.

Galerkin Matrices Out of the two approaches for Galerkin matrices described in 3.1 (graph coloring and BVP domain partitioning), we implemented the second one. For a benchmark, we chose a degree-1 discretization of Problem 2, which consisted of a $d \times d$ matrix with $d = 361345$. We measured the times required to perform $N = 10000$ Kaczmarz sweeps and report the results in Table 3. In the serial CPU implementations, we apply the row projections either in the original order ("no permutation"), in the order the parallel implementation would, or in some randomly selected (but fixed) order. Figure 5 shows how these different row permutations affect convergence.

Implementation	Row Permutation	Running Time
GPU CUDA	as in Figure 3	$77.6\,\mathrm{s}$
CPU serial	none	$53.0\,\mathrm{s}$
CPU serial	as in Figure 3	$78.4\,\mathrm{s}$
CPU serial	random	$132\,\mathrm{s}$

Table 3: Performance of non-overlapping-row solvers on a Galerkin matrix. We repeated each measurement 5 times for each implementation and we report the arithmetic mean of the obtained times. Repeated time measurements for any given implementation never differed by more than $2\,\%$.

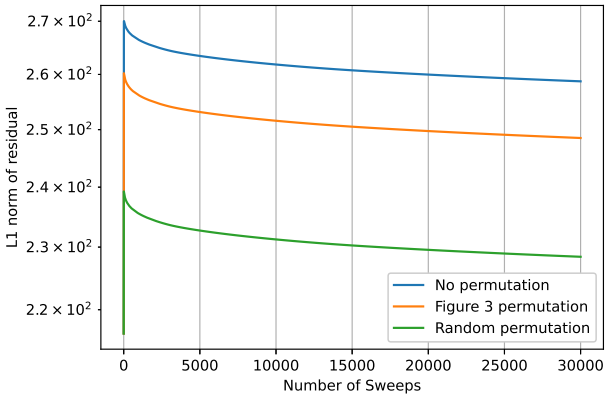

Fig. 5: The L1 norm of the residual as a function of the number of Kaczmarz iterations (sweeps) applied in our benchmark Galerkin LSE. We see that the order of the rows affects the convergence behavior (more than we observed for banded matrices). We also see that the convergence of the unmodified Kaczmarz method is very slow – the residual norm decreases by less than $10\,\%$ in the 30000 iterations. That makes accelerated methods (like in CGMNC) necessary in practice.

4.2. CARP-CG SOLVER

Relaxation Parameter We ran the GPU-accelerated CARP-CG on all PDEs from Section 4 while varying the relaxation parameter $\tilde{\lambda}$ (see Figure 6). For all PDEs and dimensions, GPU-accelerated CARP-CG achieves a minimal number of iterations for $\tilde{\lambda}$ between 0.35 and 0.6. It is notable how the optimal value for $\tilde{\lambda}$ strongly correlates with the degree of the Lagrangian space used to discretize the PDE: LSEs arising from discretization using higher degree Lagrange spaces have lower optimal values for $\tilde{\lambda}$. The optimal value of $\tilde{\lambda}$ is not strongly influenced by the choice of PDE (for the ones we tested). For all further experiments described in the following section we use $\tilde{\lambda} = 0.35$. We tested different relaxation parameters λ for our implementation of CGMNC, and found optimal values depending on the PDE between 0.95 and 1.05. For all further experiments

described in the following section, we use $\lambda = 1$.

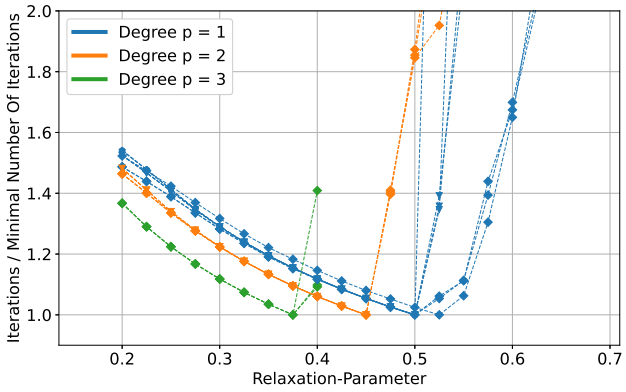

Fig. 6: Plot of the number of iterations required for GPU-accelerated CARP-CG to converge to $\frac{\|Ax-b\|}{\|b\|} < 10^{-9}$ for varying values of $\tilde{\lambda}$ across LSEs from Problems 1, 2, and 3 with dimension $n \geq 1000$. Each line is normalized by the minimum number of iterations. LSEs with $n \leq 1000$ are excluded due to similar convergence behavior.

Results We compared our GPU-accelerated CARP-CG solver with our own implementation of CGMNC (Section 2.2), Eigen's BiCGSTAB solver for general square matrices[8], and two direct solvers: Eigen's SparseLU[9] and NVIDIA's CuDSS[10]. These comparisons focused on LSEs with dimensions between $n = 10^1$ and $n = 10^6$ (Figure 7).

GPU-accelerated CARP-CG, a parallelization of CGMNC with adaptations for component averaging, is slower than CGMNC for $n < 10^4$ but outperforms CGMNC for $n > 10^4$. However, it still lags behind both iterative (Eigen BiCGSTAB) and direct solvers (Eigen SparseLU, CuDSS) by multiple orders of magnitude for sufficiently large LSEs.

The GPU-accelerated CARP-CG solver's performance was lower than expected, particularly when compared to CGMNC, which it parallelizes. While CGMNC uses a single CPU thread, CARP-CG assigns one thread per LSE row, but only slightly improves runtime for large LSEs due to the averaging step. Using more threads improves speed of the Kaczmarz sweeps but reduces convergence per iteration, creating a trade-off with an optimal ratio for maximum speedup. Since our implementation only works with one row per thread, we always average to the fullest extent, limiting speedup.

[8]https://eigen.tuxfamily.org/dox/classEigen_1_1BiCGSTAB.html
[9]https://eigen.tuxfamily.org/dox/classEigen_1_1SparseLU.html
[10]https://docs.nvidia.com/cuda/cudss/

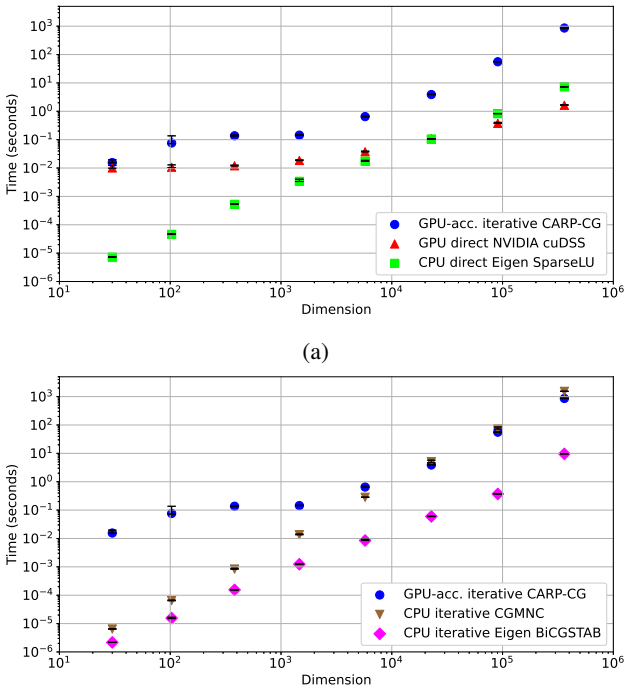

Fig. 7: Median running time of solvers versus LSE dimension n from Problem 1, based on k runs. For $n \leq 5 \cdot 10^3$, $k = 1000$; for $n \leq 9 \cdot 10^4$, $k = 200$; for $n \leq 3 \cdot 10^5$, $k = 40$; and for $n > 3 \cdot 10^5$, $k = 10$. The convergence criterion is $\frac{\|Ax-b\|}{\|b\|} < 10^{-9}$, with 5th and 95th quantiles shown as black error bars. Similar results were observed for Problems 2 and 3.

5. CONCLUSION

In this study, we developed and evaluated GPU-parallelized solvers for sparse linear systems derived from PDE discretizations. By implementing a non-overlapping row approach and an adapted CARP-CG algorithm using CUDA, we achieved varying degrees of computational performance improvements. Our non-overlapping row approach demonstrates that parallelization of the basic Kaczmarz algorithm is possible and can speed up the runtime. In our experiments for banded matrices, our parallel CPU implementations were faster than our GPU implementations. Future work could assess in which situations the GPU implementation would be faster (e.g. single- instead of double-precision FP might be a good candidate). Our GPU-accelerated CARP-CG performs faster than the sequential base algorithm CGMNC, but only for LSEs of very high dimension. Future work could focus on implementing the GPU-accelerated CARP-CG solver to use fewer threads while retaining functionality, therefore decreasing the error introduced by the component averaging.

6. REFERENCES

[1] Ralf Hiptmair, "Numerical methods for partial differential equations," 2024.

[2] W. J. Duncan, "The principles of the galerkin method," Tech. Rep. 1894, Aeronautical Research Report and Memoranda, 1938.

[3] Stefan Kaczmarz, "Angenäherte auflösung von systemen linearer gleichungen," *Bulletin International de l'Académie Polonaise des Sciences et des Lettres*, vol. 35, pp. 355–357, 1937.

[4] Dan Gordon and Rachel Gordon, "Carp-cg: A robust and efficient parallel solver for linear systems, applied to strongly convection dominated pdes," *Parallel Computing*, vol. 36, no. 9, pp. 495–515, 2010.

[5] Joseph M. Elble, Nikolaos V. Sahinidis, and Panagiotis Vouzis, "Gpu computing with kaczmarz's and other iterative algorithms for linear systems," *Parallel Computing*, vol. 36, no. 5, pp. 215–231, 2010, Parallel Matrix Algorithms and Applications.

[6] Xinyin Huang, Gang Liu, and Qiang Niu, "Remarks on kaczmarz algorithm for solving consistent and inconsistent system of linear equations," in *Computational Science – ICCS 2020*, Valeria V. Krzhizhanovskaya, Gábor Závodszky, Michael H. Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João Teixeira, Eds., Cham, 2020, pp. 225–236, Springer International Publishing.

[7] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th National Conference*, New York, NY, USA, 1969, ACM '69, p. 157–172, Association for Computing Machinery.