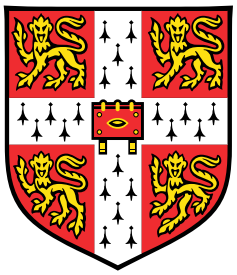
Unstructured Mesh FVM with GPU Support

Report 2



Sondre Sigstad Wikberg

Supervisor: Hrvoje Jasak

Department of Physics
University of Cambridge

This dissertation is submitted for the degree of
MPhil in Scientific Computing

King's College August 2024

I would like to dedicate this second part of my thesis to the maintainers and creators of the SuiteSparse matrix collection. Easily reproducible research in the field of sparse matrix computations would not be possible without their site …

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 65,000 words including appendices, bibliography, footnotes, tables and equations and has fewer than 150 figures.

Sondre Sigstad Wikberg
August 2024

Acknowledgements

And I would like to acknowledge my parents for their unwavering support and encouragement throughout the research and writing of this thesis. Additionally, I would like to acknowledge my flatmates W. and V. for their backing.

Abstract

In this continuation of the work begun in [21], we explore efficient algorithms for solving sparse linear systems of equations characterised by symmetric positive definite coefficient matrices, arising from implicit FVM discretisations, on GPU. We improve on the matrix storage formats discussed in [21] in terms of time to compute sparse matrix-vector products and facilitating smoothers and preconditioning methods for the conjugate gradient algorithm on GPU. When compared to CPU implementations using common sequential algorithms, the implemented smoothers and preconditioners demonstrate similar convergence with a time speedup-factor up to 100.

Table of contents

Chapter 1

Preamble

This report continues the work, begun in [21], of constructing an implicit finite volume method (FVM) solver for GPU. In [21], the importance of fast sparse matrix-vector multiplication (SpMV) for Krylov-based iterative solvers applied to FVM discretisations was outlined. The performance of existing algorithms was compared and a modification (Staircase SELL) to the state of the art (Sliced ELLPACK) was suggested to cater that method to matrices derived from FVM applications.

In this second part of the work, we narrow down the objective to focus on solving systems of equations whose coefficient matrix is symmetric positive definite (SPD). Such matrices arise from the implicit discretisation of the incompressible Navier-Stokes equation under FVM.

In addition to efficient SpMV, modern Krylov-based solvers are typically reliant on good preconditioning and smoother-algorithms to achieve rapid convergence. These aspects of sparse linear systems will also be investigated in the context of developing a fast conjugate gradient (CG) method on GPU for SPD linear systems resulting from FVM discretisations.

1.1 Orders of Parallelism

Since the advent of modern parallel computing systems, an enormous body of work has been dedicated to accelerating solvers for large linear systems of equations. In the same way that the best sequential algorithms are not always the most efficient when deployed on parallel computers, different parallel architectures benefit from different algorithms when being put to the aforementioned task. In particular, many authors make a distinction between *course-grained parallelism* and *fine-grained parallelism*.

Fine-grained parallelism is characterised by dividing a task into many sub-processes. While this may have the effect of improving the load balancing between computational

units, it increases the cost of communication. By contrast, course-grained parallelism is characterised by the workload being subdivided between fewer units, with the benefit that each unit may operate on a larger body of data on which it can execute more sequential tasks, reducing the need for communication from other units. One may therefore infer that the greater the dependence between operations in an algorithm, the more suited it is to course-grained parallelism and vice-versa.

Modern NVIDIA GPUs are arguably suitable for algorithms situated on both sides of this spectrum. Limited room for communication between individual thread-blocks (coarse-grained parallelism) is ameliorated by their ability to synchronise effectively the work carried out by groups of hundreds of threads belonging to each thread-block (fine-grained parallelism).

When operating on these devices, it follows that communication should be limited to threads in the same block, rather than communication between blocks. In implementing a full CG solver for the NVIDIA GPU, the author has focused on the testing and integration of algorithms amenable to these requirements.

1.2 Order of Importance of Algorithms

The overall objective of the report is to produce and experimentally justify code for the components of an efficient CG-based solver for FVM applications on NVIDIA GPUs. It is assumed that the majority of the running time for any such solver will be attributed to conjugate gradient iterations. Hence, smoothing and preconditioning algorithms which facilitate faster convergence are designed to accommodate a memory layout efficient for fast iterations of the conjugate gradient method. In particular, the ordering of equations, which directly determines the order in which coefficient-matrix elements are stored, is optimised for the SpMV operations necessary in each CG operation. The order in which elements are stored is important to achieve coalesced access when operating on global memory, as was discussed in [21].

1.3 Report Structure

In this extension of [21], chapter 2 gives a short outline of the association between implicit FVM and sparse linear systems characterised by an SPD coefficient matrix, as this was omitted in [21]. Thereafter, chapter 3 outlines an improvement to the matrix storage formats presented in [21] when the objective is to solve such systems with the help of matrix preconditioning and smoothing of the linear system.

Moreover, chapters 4 and 5 investigate smoothing algorithms and preconditioners for sparse linear systems on GPU, respectively.

Chapter 2

Importance of Sparse SPD Matrices in FVM

For the solvers discussed here, we are motivated by the incompressible Navier-Stokes equations which may be expressed

$$\nabla \cdot \mathbf{u} = 0 \tag{2.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) - \nabla \cdot (\nu \nabla \mathbf{u}) = -\nabla p, \tag{2.2}$$

where $\mathbf{u}$ denotes the velocity and p, ν denote the kinematic pressure and viscosity of the fluid, respectively. The viscosity is often taken as known. Equation (2.1), typically known as the *continuity equation*, expresses the incompressibility of the system. On the other hand, the *momentum equation* (2.2) expresses the conservation of momentum. Collectively, this system of equations is also known as the pressure-velocity (PV) coupling. Note that the time derivative simply vanishes if we seek a steady-state solution.

2.1 Implicit FVM for the PV Coupling

In the absence of a separate equation for the pressure, FVM methods for the PV coupling typically rely on applying the continuity equation (2.1) to the momentum equation (2.2), in some way, to form a conservative *pressure equation*. Since there are now two conservative equations, one for pressure and for velocity, an iterative approach then follows using implicit finite volume discretisations; the momentum equation (2.2) is discretised and solved for a new velocity guess $\mathbf{u}^{**}$ using an existing pressure p^{*}, after which the pressure equation is discretised and solved for a new pressure guess p^{**} using $\mathbf{u}^{**}$. Since the source-term in

each of the equations depends on an inaccurate previous guess of the other variable, the two sets of equations are solved alternately until convergence to a new (or steady-state) solution $(p^{**}, \mathbf{u}^{**})$. [1]

Perhaps the most straightforward means with which to form a pressure equation is simply to take the divergence of the momentum equation, yielding

$$\nabla \cdot (\nabla \cdot (\mathbf{uu})) - \nabla \cdot (\nabla \cdot (\nu \nabla \mathbf{u})) = -\nabla^2 p, \tag{2.3}$$

where the first term from equation (2.2) vanishes due to the continuity equation (2.1). This is now a Poisson equation with a Laplacian in the pressure.

Alternatively, many segregated solvers rely on a so-called *semi-discretised* approach attributed to Spalding and Patankar [18]. Let $\nabla \mathbf{p} \in \mathbb{R}^{3n}$ denote the known pressure gradients across all volumetric centroids of n cells in a three-dimensional finite volume mesh and let $\mathbf{V} \in \mathbb{R}^{3n}$ denote the unknown vector of velocities. A discretised form of the momentum equation (2.2) may then be expressed as [2]

$$\mathbf{A_u V} = -\nabla \mathbf{p}. \tag{2.4}$$

The matrix $\mathbf{A_u}$ may now be decomposed into diagonal $\mathbf{D_u}$ and off-diagonal $\mathbf{H_u}$ parts as $\mathbf{A_u} = \mathbf{D_u} - \mathbf{H_u}$. We may then write

$$\mathbf{V} = -\mathbf{D_u}^{-1} \nabla \mathbf{p} + \mathbf{D_u}^{-1} \mathbf{H_u V}. \tag{2.5}$$

Applying then the continuity equation (2.1) yields

$$-\nabla \cdot \left(\mathbf{D_u}^{-1} (\nabla \mathbf{p}) \right) = \nabla \cdot \left(\mathbf{D_u}^{-1} \mathbf{H_u V} \right). \tag{2.6}$$

If the same idea is applied, but instead an old pressure $\mathbf{p}^*$ is used to solve for an updated velocity $\mathbf{V}^{**}$ in equation (2.4), then an updated pressure $\mathbf{p}^{**}$ may be found by solving the Poisson equation (2.6), with $\mathbf{V}^{**}$ plugged into the right-hand side. As may be seen, both the direct and semi-discretised approaches lead to a Poisson equation in the pressure.

[1] It must be noted that convergence is not always guaranteed with such methods, and special treatment is required in the iterations (notably using under-relaxation of equations).

[2] We have omitted from equation (2.4) any terms from the discretisation procedure $\mathbf{d}_f$ that are not dependent upon the updated field $\mathbf{V}$. Note however that any such terms would vanish under divergence in equation (2.6), thus yielding the same result.

2.2 Forming a Linear System from the Laplacian

We will now demonstrate an implicit finite volume discretisation of the Laplacian as in equations (2.3, 2.6). For simplicity we consider now a Poisson equation of the form

$$-\nabla^2 p = q, \tag{2.7}$$

over some domain $D \in \mathbb{R}^3$, where $q \in \mathbb{R}$ is source term. Let P be a cell in a finite volume mesh over D and index the faces of P by f. Assume further that the gradient ∇p varies linearly over each face f where $(\nabla p)_f$ denotes the value at the face's areal centre. Integrating the equation over the volume V of P, applying the divergence theorem, and assuming $\int_V q dV = \tilde{q}$ known, yields

$$-\sum_f (\nabla p)_f \cdot \mathbf{s}_f = \tilde{q}, \tag{2.8}$$

where $\mathbf{s}_f$ denotes the area-scaled outward normal of the face. We see then that the problem reduces to finding the term $(\nabla p)_f \cdot \mathbf{s}_f$ at each face f of P. Now, let p_P denote the value of p at the volumetric centre of cell P. Similarly, let p_N denote the value at the centroid of cell N sharing face f with P. Assume now that $\mathbf{s}_f$ is parallel to the vector $\vec{\Delta f}$ connecting the centroids[3]. Discretising the left-hand side using a first order finite difference approximation yields

$$(\nabla p)_f \cdot \mathbf{s}_f \approx \frac{|\mathbf{s}_f|}{|\Delta f|} \cdot (p_N - p_P). \tag{2.9}$$

If a face $f = b$ is a boundary of D at which p has the known boundary value p_b (Dirichlet boundary condition), we may write

$$(\nabla p)_b \cdot \mathbf{s}_b \approx \frac{|\mathbf{s}_b|}{|\Delta b|}(p_b - p_P), \tag{2.10}$$

where $|\Delta b|$ is the distance from the centre of P to areal centre of b. If a face $f = \hat{b}$ is a boundary face at which the gradient ∇p takes the value $(\nabla p)_{\hat{b}}$ (Neumann condition) then we automatically know the value of $(\nabla p)_{\hat{b}} \cdot \mathbf{s}_f$. Assembling these equations for each face, we

[3] If this is not the case (common in unstructured FVM) we typically apply a *non-orthogonal correction* to equation (2.9) which in effect reduces the size of the coefficient $|\mathbf{s}_f|/|\Delta f|$ and adds a constant (known value) on the right-hand side in equation (2.9) [10]. This correction, however, is applied symmetrically to any neighbouring cells P,N and does not invalidate the properties of the resulting discretisation matrix shown in 2.2.1.

see that we may write

$$-\sum_f (\nabla p)_f \cdot \mathbf{s}_f = \tilde{q} \tag{2.11}$$

$$\Rightarrow \quad \sum_f \frac{|\mathbf{s}_f|}{|\Delta f|}(p_P - p_N) + \sum_b \frac{|\mathbf{s}_b|}{|\Delta b|} p_P \approx \tilde{q} + \sum_b \frac{|\mathbf{s}_b|}{|\Delta b|} p_b + \sum_{\hat{b}} (\nabla p)_{\hat{b}} \cdot \mathbf{s}_{\hat{b}} \tag{2.12}$$

where f now indexes any faces of P internal to the mesh, b indexes any boundary faces with a Dirichlet condition, and $\hat{b}$ indexes any boundary faces with a Neumann condition.

2.2.1 Showing that the Coefficient Matrix is SPD

It is easily seen that if we assemble these equations for every cell P in a finite volume mesh, we end up with a linear system of equations $\mathbf{Ap} = \mathbf{b},\ \mathbf{A} \in \mathbb{R}^{n,n}$ where on each row P

$$a_{P,P} = \sum_f \frac{|\mathbf{s}_f|}{|\Delta f|} + \sum_b \frac{|\mathbf{s}_b|}{|\Delta b|} \tag{2.13}$$

$$a_{P,M} = a_{M,P} = -\frac{|\mathbf{s}_f|}{|\Delta f|} \quad \text{if cells } P \neq M \text{ share a face } f \tag{2.14}$$

$$a_{P,M} = a_{M,P} = 0 \quad \text{if cells } P \neq M \text{ do not share a face} \tag{2.15}$$

$$b_P = \tilde{q} + \sum_b \frac{|\mathbf{s}_b|}{|\Delta b|} p_b + \sum_{\hat{b}} (\nabla p)_{\hat{b}} \cdot \mathbf{s}_f. \tag{2.16}$$

It is easily seen that any such matrix $\mathbf{A}$ is symmetric and (weakly) diagonally dominant, but not necessarily strictly so. A standard result in linear algebra is that the diagonal dominance of the matrix guarantees also that it will be positive semi-definite (i.e. $\mathbf{y}^T\mathbf{Ay} \geq 0\ \forall \mathbf{y} \in \mathbb{R}^n$). Furthermore, in a large mesh, if each cell is only adjacent to a few ($\lessapprox 20$) other cells, it is seen that the matrix is sparse.

Clearly, if every boundary of the domain D has a Neumann condition, the system in (2.3) is underdetermined and we cannot hope to meaningfully solve a system of equations involving its discretisation matrix (i.e. only knowing the gradient at the boundaries makes for an infinite number of solutions). By contrast, if at least one boundary is Dirichlet, then strict diagonal dominance holds for at least one row P of the discretisation matrix $\mathbf{A}$ as above. Furthermore, note that the adjacency graph of the discretisation matrix is undirected and strongly connected (by assumption the domain D is connected), meaning that for every $p, q \in \{1, \ldots, n\}$ there is a walk $p \to p_1 \to p_2 \to \ldots \to p_m \to q$, $\{p_1, \ldots, p_m\} \subset \{1, \ldots, n\}$ such that $a_{p_i, p_{i+1}} \neq 0\ \forall i \in \{1, \ldots, m\}$. In [20] it is shown that any matrix $\mathbf{A}$ satisfying the following criteria is invertible:

- **A** is weakly diagonally dominant,
- strict diagonal dominance holds in at least one row of **A**,
- the adjacency graph of **A** is strongly connected.

We see thus that the discretisation matrix resulting from the discretisation of the Laplacian in (2.7) is invertible when at least one boundary of the domain D has a Dirichlet condition. Any invertible positive semi-definite matrix is also positive definite, meaning that the presence of at least one Dirichlet boundary guarantees the symmetric positive definiteness of the discretisation matrix **A**.

We see therefore that solving systems with SPD coefficient matrices constitutes half the problem of numerically resolving the PV coupling with implicit FVM.

Chapter 3

Matrix Storage Format

3.1 Recap of Sliced ELLPACK Format

In [21], the Sliced ELLPACK (SELL) storage format, attributed to [15], for sparse matrices on (NVIDIA) GPUs was outlined alongside a slight modification to it (Staircase SELL) that sought to exploit the upper bound on non-zeros in each row typical of FVM discretisation matrices. In essence, SELL format stores the non-zero coefficients for a fixed number of consecutive matrix rows as a group (a "slice") to be processed collectively by a thread-block during matrix-vector multiplication.

Alongside the array which holds non-zero matrix entries for a slice, an integer array containing the column index of each such entry is also stored. The size of each of these arrays is directly proportional to the number of rows in a slice and to the greatest number of non-zeros in any given row within that slice. Thus, if the maximal number of non-zeros on any row in a slice is 5, 5 floating point numbers and 5 integers are stored for each row in the slice. Rows with fewer than 5 non-zeros are padded with explicit zeros in the array of entries and -1 in the array of column indices.

Notably, the non-zero elements of different rows in a slice are interleaved in storage (see figure below). This is so that consecutive threads in a thread-block may be responsible for consecutive rows, with threads reading the *i*th non-zero element of their respective rows in a coalesced manner. In practice, the entries for all slices are stored together in an array `entries`, whereas the column indices are stored together in an array `col_idx`. A "slice pointer"-array (`slice_ptr`) points to the starting position of each slice within `entries` and `col_idx`. An illustration of the Sliced ELLPACK format is provided below for a small matrix **A** with a *slice height* (number of rows in each slice) of 3.

$$\mathbf{A} = \left[\begin{array}{ccccccccc}
2.5 & 3.2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3.2 & 0.2 & 0.4 & 3.9 & 0 & 0 & 0 & 0 & 0 \\
0 & 0.4 & 2.7 & 2.1 & 1.1 & 0 & 0 & 0 & 0 \\
\hdashline
0 & 3.9 & 2.1 & -4.5 & 0 & 0 & 1.7 & 0 & 0 \\
0 & 0 & 1.1 & 0 & 1.2 & 3.3 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 3.3 & 2.8 & 0 & 0.8 & 1.4 \\
\hdashline
0 & 0 & 0 & 1.7 & 0 & 0 & 1.1 & -2.6 & 0 \\
0 & 0 & 0 & 0 & 0 & 0.8 & -2.6 & 0.2 & 3.1 \\
0 & 0 & 0 & 0 & 0 & 1.4 & 0 & 3.1 & 2.0
\end{array}\right]$$

Slice 0 · Slice 1 · Slice 2

$$\texttt{slice\_ptr} = \left[\overbrace{0}^{\text{Start of slice 0}} \quad \overbrace{12}^{\text{Start of slice 1}} \quad \overbrace{24}^{\text{Start of slice 2}} \quad \overbrace{36}^{\text{Length of } \texttt{entries} \text{ \& } \texttt{col\_idx}}\right]$$

$$\texttt{entries} = \Big[\overbrace{\underbrace{2.5 \;\; 3.2 \;\; 0.4}_{\substack{\text{1st non-zero}\\ \text{entry rows 0,1,2}}} \;\; \underbrace{3.2 \;\; 0.2 \;\; 2.7}_{\substack{\text{2nd non-zero}\\ \text{entry rows 0,1,2}}} \;\; \underbrace{0 \;\; 0.4 \;\; 2.1}_{\substack{\text{3rd non-zero}\\ \text{entry rows 0,1,2}}} \;\; \underbrace{0 \;\; 3.9 \;\; 1.1}_{\substack{\text{4th non-zero}\\ \text{entry rows 0,1,2}}}}^{\text{Slice 0}} \;\; \underbrace{3.9 \;\; 1.1 \;\; 3.3}_{\substack{\text{1st non-zero}\\ \text{entry rows 3,4,5}}} \;\; \ldots\Big]$$

$$\texttt{col\_idx} = \Big[\underbrace{0 \;\; 0 \;\; 1}_{\substack{\text{Col. of 1st}\\ \text{non-zero entry}\\ \text{rows 0,1,2}}} \;\; \underbrace{1 \;\; 1 \;\; 2}_{\substack{\text{Col. of 2nd}\\ \text{non-zero entry}\\ \text{rows 0,1,2}}} \;\; \underbrace{-1 \;\; 2 \;\; 3}_{\substack{\text{Col. of 3rd}\\ \text{non-zero entry}\\ \text{rows 0,1,2}}} \;\; \underbrace{-1 \;\; 4 \;\; 5}_{\substack{\text{Col. of 4th}\\ \text{non-zero entry}\\ \text{rows 0,1,2}}} \;\; \underbrace{1 \;\; 2 \;\; 4}_{\substack{\text{Col. of 1st}\\ \text{non-zero entry}\\ \text{rows 3,4,5}}} \;\; \ldots\Big]$$

When launching a kernel to operate on slices of a matrix in Sliced ELLPACK format, thread-blocks each operate on one slice each and are organised in a grid with dimensions $(\lceil \texttt{n\_rows}/\texttt{slice\_height} \rceil, 1, 1)$. Accordingly, thread-blocks are launched with dimensions (`slice_height`, 1, 1). As slice height is constant, a thread-block may determine the number of explicitly stored non-zeros in their slice i (the *slice length*) by the formula

$$\texttt{length} = \frac{\texttt{slice\_ptr[i+1]} - \texttt{slice\_ptr[i]}}{\texttt{blockDim.x}},$$

where the thread-block dimension $\texttt{blockDim.x} = \texttt{slice\_height}$ is a constant assumed accessible from within each thread-block.

Furthermore, each thread in a thread-block knows the row for which it is responsible from the simple relation

$$\texttt{row} = \texttt{blockIdx.x} \times \texttt{slice\_height} + \texttt{threadIdx.x},$$

where `blockIdx.x` indicates the index of the thread-block (=index of the slice) in the grid, and `threadIdx.x` is the thread-index in the block (=index of the row in the slice).

Each thread-block may now iterate over the `length`, accessing a new non-zero element in the corresponding row with each iteration. For example, in the above, threads `t0, t1, t2` in block `0` could access entries 2.5, 3.2, 0.4 in `entries` and their columns $0, 0, 1$ in `col_idx`, respectively, in a coalesced manner. Having processed these, the threads may collectively move to the next iteration, whereupon they read the second non-zero entries of their respective rows etc. Since NVIDIA GPUs operate synchronously on threads in *warps* of 32, the slice height should be chosen as a multiple of this number.

3.1.1 Diagonal Matrix Entries

Sparse matrix-vector multiplication is dependent on determining the column index of every non-zero element of the matrix, after which the corresponding vector element must be collected from global memory. When each thread of a thread-block is responsible for one row, as in SELL format, however, we often get non-coalesced access to the array storing the vector. Consider for example the threads `t3, t4, t5` responsible for rows 3,4,5 as in the example above. During an SpMV operation, these threads first collect the matrix entries $3.9, 1.1, 3.3$ and their column indices $1, 2, 4$ from global memory, respectively. They must now collect vector entries $1, 2, 4$, respectively. Since vector elements $1, 2, 4$ are not consecutive in memory, this operation cannot happen in a coalesced fashion.

In [21], the impact of permutations for matrix bandwidth reduction on SpMV performance with SELL format on GPU was discussed. Firstly, we expect a smaller bandwidth to improve coalescence when collecting vector entries from global memory. This is because a reduced bandwidth tends to increase the "degree to which" the *i*th non-zero entry in successive rows form off-diagonal bands on the matrix. This will improve coalescence when collecting vector entries from global memory (imagine `t3, t4, t5` being able to collect vector elements $1, 2, 3$ rather than $1, 2, 4$ synchronously). Secondly, since the diagonals of FVM discretisation matrices are non-zero, we always know we must collect the corresponding vector entries when operating on a particular slice of the matrix. Hence, if entries are closely clustered around the diagonal, we hope to improve cache hit rates. When processing a slice encompassing

rows i to j, it is therefore sensible to first load vector entries i to j, as this may be done in an entirely coalesced manner and will increase cache hits for later vector accesses.

3.2 Another Modified SELL

In fact, we may go further than relying on cache, which may fill up and push out vector entries we will need at some later point. If, as a first step in an SpMV kernel, vector entries i to j are stored in the shared memory segment of a block, any subsequent access to these elements may be done as rapidly as an access to L1 cache (this is due to the fact that L1 cache and shared memory reside in the same physical location on NVIDIA GPUs [17]). Threads determine what vector entry by which to multiply a matrix entry based on the column index (stored in `col_idx`). In order that a thread may know to collect a vector entry from shared memory rather than global memory, we can define `col_idx` such that a column index $c \in \{i,\dots,j\}$ is stored in `col_idx` as $-(c-i)-1$. Instead of padding the `col_idx` array with -1s, we pad it with zeros. Furthermore, column indices $c \notin \{i,\dots,j\}$ can simply be incremented by one and stored as $c+1$.

Taking into account the separate storage of diagonal elements, a modified Sliced ELLPACK (mod. SELL) format for the 9×9 sparse matrix above will now look as

$$\texttt{slice\_ptr} = \begin{bmatrix} 0 & 9 & 18 & 27 \end{bmatrix}$$

$$\texttt{diag} = \begin{bmatrix} \overbrace{2.5 \;\; 0.2 \;\; 2.7}^{\text{Diagonal of Slice 0}} & \overbrace{-4.5 \;\; 1.2 \;\; 2.8}^{\text{Diagonal of Slice 1}} & \overbrace{1.1 \;\; 0.2 \;\; 2.0}^{\text{Diagonal of Slice 2}} \end{bmatrix}$$

$$\texttt{entries} = \begin{bmatrix} \overbrace{3.2 \;\; 3.2 \;\; 0.4 \;\; 0 \;\; 0.4 \;\; 2.1 \;\; 0 \;\; 3.9 \;\; 1.1}^{\text{Off-diagonals Slice 1}} & 3.9 & 1.1 & 3.3 & \dots \end{bmatrix}$$

$$\texttt{col\_idx} = \begin{bmatrix} -2 & -1 & -2 & 0 & -3 & 4 & 0 & 4 & 5 & 2 & 3 & -2 & \dots \end{bmatrix}.$$

Note in particular that the slice length is now reduced by one due to the separate storage of the diagonal (slice 1 now has length 3 rather than 4).

When encountering a positive entry $\tilde{c}$ in `col_idx`, a thread will know to collect vector entry $c = \tilde{c} - 1$ from global memory (or cache). Conversely, a negative entry $\tilde{c}$ indicates that the corresponding matrix entry lies on column $c = -(\tilde{c}+1)+i$, but more importantly that vector entry c may be collected from shared memory location $-(c+1)$.

3.2.1 The Objective of Permutations

Under the modified storage format (mod. SELL), we see that the objective of permuting the rows (and columns) of a matrix changes slightly from just reducing the bandwidth. For a slice encompassing rows i to j, we are now interested in increasing the number off-diagonal entries whose column index $c \in \{i, \ldots, j\}$. These objectives clearly coincide to a great extent, however. When dealing with representing discretisations of physical domains (such as implicit FVM matrices), some authors recommend a domain-decomposition approach for this purpose, whereby rows (cells) are grouped in slices (sub-domains) so as to minimise the number of rows containing non-zero entries whose column fall outside the slice (i.e. $c \notin \{i, \ldots, j\}$) [14]. In such a grouping, slices are divided by a set of interface rows (i.e cells), which are themselves grouped in a set of slices. While such an approach could greatly increase the number of entries for which $c \in \{i, \ldots, j\}$ in the non-interface slices, slices containing interface rows would have a large number of entries for which $c \notin \{i, \ldots, j\}$. Furthermore, the number of interface cells in optimal versions of such orderings would clearly be inversely dependent on the slice height, the upper bound of which corresponds to the maximal number of threads allowed in a slice (1024 on modern NVIDIA GPUs).

We have decided here to test implementations of the modified format using the well-known Cuthill-Mckee (CM) ordering of equations to reduce bandwidth. However, it would be interesting to consider the effect of different domain decompositions on SpMV computation times using the SELL and mod. SELL storage formats.

3.2.2 Numerical Implications for FVM Matrices

The modified format is especially appealing for SpMV operations with FVM discretisation matrices due to the fact that these matrices are structurally symmetric. In particular, consider a slice encompassing rows i to j. A matrix entry on row $r \in \{i, \ldots, j\}$ and column $c \in \{i, \ldots, j\}$ implies the existence of an element on row c and column r. Vector entries $\{v_i, \ldots, v_j\}$ are loaded to shared memory preemptively, meaning that v_c and v_r can be collected much more rapidly than from global memory.

As shall be seen in the section on smoothers, however, the greatest benefit of this new format is perhaps that it allows easy access and identification of diagonal and "local" matrix elements whose column $c \in \{i, \ldots, j\}$. In fact, for a slice encompassing rows i to j, an entry on row $r \in \{i, \ldots, j\}$ and column $c \in \{i, \ldots, j\}$ indicates the impact (for FVM matrices) of the state in cell c on that in cell r. Thus, these matrix entries are important in resolving the total effects between rows in a given slice.

3.3 SpMV Tests with Modified Format

Figure 3.1 shows average times in microseconds, obtained using `cudaElapsedTime`, for SpMV computations over 9 large, sparse matrices using both SELL (kernel `ellpack_row_based` in accompanying software) and mod. SELL (kernel `ellpack_shared`) formats on an NVIDIA A30 GPU. Both formats were implemented with CM ordering on CPU before being copied to GPU. To maximise cache hits, a slice height of 1024 was used for the SELL format. For the mod. SELL format, a slice height of 1024 was not allowed by the compiler as "too many resources were requested" and a height of 512 was used instead. A dot product with the same number of rows as the matrix was computed between each iteration to clear the cache of any remaining values from the previous iteration that could skew performance. Cuthill-Mckee algorithm was applied to all matrices before being encoded in the indicated formats. All code was compiled with `O2` optimisation on CUDA 11.7 for C++ and double precision format was used. Matrices are symmetric positive definite and were selected from the SuiteSparse collection [5] under the criterion that the maximal number of entries in any given row subceded 15. Furthermore, as matrices with less than approximately 100,000 rows made it difficult to reproducibly distinguish the performance of different SpMV kernels, only matrices with more than 200,000 rows were considered (except for "100K" which was an FVM matrix kindly provided by [11]). Lastly, matrices were also required to come from different research groups so as not to over-represent any particular matrix structure. These criteria should serve to test the mentioned storage formats on matrices whose structure is arguably "close" to those produced under large, unstructured FVM discretisations of the Laplacian operator.

Whereas the SuiteSparse collection does not contain any symmetric, positive definite FVM discretisation matrices of relevant size, matrices were selected from this resource as it allows for easy validation. Furthermore, as was explained in [21], we expect SpMV performance to be dependent more on the matrix structure, rather than the particular problem from which the matrix is derived.

Tests using the Staircase SELL format as proposed in [21] are omitted as this format did not yield speed-up relative to SELL when tested there. (It must be noted here that the provided computation times for SpMV provided in [21] were accidentally listed as being in milliseconds when the correct unit of time was microseconds (as here).)

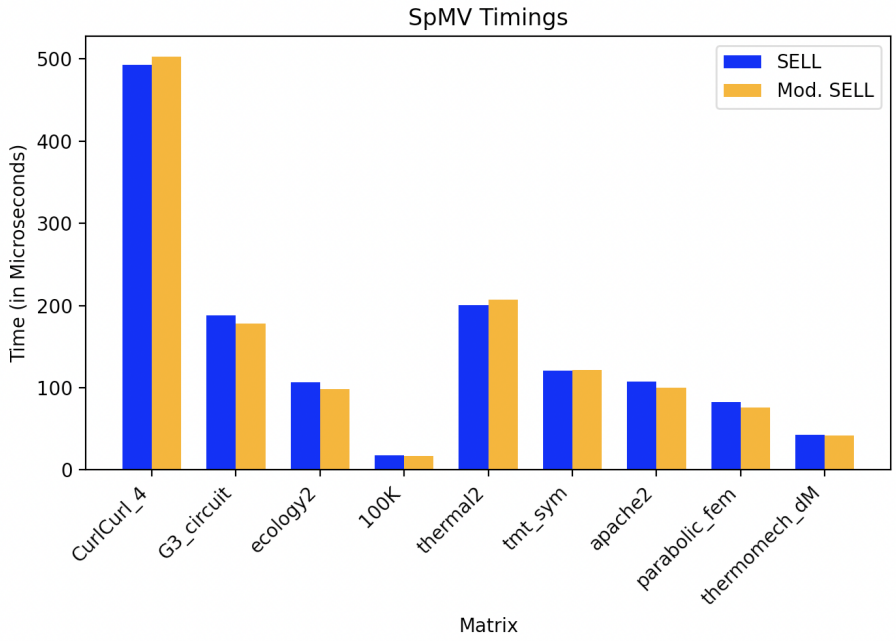


Fig. 3.1 Average SpMV time (in microseconds) when given matrix is stored in SELL format or modified mod. SELL. Time is an average of 100 SpMV operations

Information on the relevant matrices and the exact multiplication times may be seen from table 3.1.

Matrix	**Src**	**n**	$\overline{\mathbf{nnz}}$	**mnz**	λ	**Problem**	**SELL**	**M. SELL**
CurlCurl_4	[5]	2,380,515	11.138	13	0.28	FEM	492.4	502.2
G3_circuit	[5]	1,585,478	4.83	6	0.17	*	188.1	178.3
ecology2	[5]	999,999	4.996	5	0.09	**	106.7	98.3
100K	[11]	102,400	4.988	5	0.58	FVM	17.6	17.3
thermal2	[5]	1,228,045	3.99	11	0.36	FEM	200.1	207.3
thermomech_dM	[5]	204,316	6.97	11	0.53	FEM	42.6	41.9
tmt_sym	[5]	726,713	6.99	11	0.36	***	120.1	121.8
apache2	[5]	715,176	6.74	8	0.01	Structural	107.8	100.1
parabolic_fem	[5]	525,825	6.99	7	0.44	FEM	82.3	76.2

Table 3.1 **Src**: source, **n**: number of rows, $\overline{\mathbf{nnz}}$: average number of non-zeros by row, **mnz**: max non-zeros by row, λ: Proportion of non-diagonal matrix-entries whose corresponding vector entry may be collected directly from shared memory in mod. SELL format, **SELL**: Average SpMV time with SELL (in μs), **M. SELL**: Average SpMV time with mod. SELL storage (in μs), *Circuit Simulation, **Animal movement/gene flow model, ***Computational Electromagnetics Problem

3.3.1 SpMV Results: Discussion

The modified and original SELL formats perform very similarly on the large matrices tested here. In fact, the modified format performs slightly better on 6 of the 9 matrices tested. Attention is drawn to the column of λ-values indicating the proportion of off-diagonal entries whose column-index $c \in \{i, \ldots, j\}$, where $\{i, \ldots, j\}$ indicates the set of rows encompassed by the corresponding slice. In accordance with the observation made in 3.2.1, we expect the mod. SELL format to perform better, relative to SELL, on those matrices for which the λ parameter is higher. However, no such pattern is observed on the (limited) data in table 3.1.

Perhaps more importantly, however, CUDA's `cudaElapsedTime` feature is only accurate to 0.5 microseconds as stated in the CUDA toolkit documentation [4]. While averaging the times over 100 iterations should smooth out this imprecision, some of the differences may be due to this factor.

Furthermore, a single thread accessing shared memory (under mod. SELL), rather than global memory, does not necessarily lead to speed-up. If some threads in a warp access shared memory whereas others access global memory or cache, these operations are serialised, as per the CUDA programming guide [17]. On the other hand, in the case of SELL algorithm, if threads in a warp access global memory in a non-coalesced manner, several memory instructions must be issued to accommodate the fetch, unless the requested data is already cached [17]. Multiple memory transactions generally lead to higher latency compared to single, coalesced instructions. Therefore, we cannot say a-priori whether a smaller λ will necessarily lead to superior performance of the mod. SELL storage format over SELL during an SpMV operation.

Given that the mod. SELL format performs no worse than the original SELL format for SpMV, we have decided to implement further tests using the mod. SELL format. While precedence is given to speeding up CG iterations (and thus SpMV operations), we shall see in chapter 4 & 5 that the modified format holds significant advantages for smoothing and preconditioning on GPU.

Chapter 4

Smoothers

An important component of linear solvers are smoothing algorithms which serve to more evenly distribute the error between neighbouring components (cells) of a guessed solution $\mathbf{x}^{(i)}$. Perhaps the most widely used smoother for linear systems is the Gauss-Seidel algorithm. While effective when applied on single-core architectures, the algorithm is inherently hard to parallelise on GPU when the sparsity pattern of the discretisation matrix is chaotic, as in unstructured FVM. We present here a review of various smoothing algorithms for SPD matrices along with a discussion of their suitability for GPU implementation when the mod. SELL format from chapter 3 is used. The convergence rates of these algorithms is also investigated for some of the sparse SPD matrices from that chapter.

Since we are interested in developing a solver in which the brunt of the computational time is spent on CG iterations, some more computationally intensive smoothing algorithms have been left out of the discussion. Notably, we have left out successive over-relaxation and Richardson iteration, both of which depend on computationally intensive estimation of *relaxation factors* for fast convergence [19].

4.1 Jacobi Method

Perhaps the simplest form of smoother is the so-called Jacobi smother. For a linear system $\mathbf{Ax} = \mathbf{b}$, we decompose $\mathbf{A}$ into diagonal, $\mathbf{D}$, strictly upper triangular, $\mathbf{U}$, and strictly lower triangular parts $\mathbf{L}$, such that $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$. Given some guess $\mathbf{x}^{(i)}$ at iteration i for the solution, an updated guess $\mathbf{x}^{(i+1)}$ is produced by the formula

$$\mathbf{x}^{(i+1)} = \mathbf{D}^{-1}\big(\mathbf{b} - (\mathbf{L}+\mathbf{U})\mathbf{x}^{(i)}\big). \tag{4.1}$$

It is shown in [13] that the algorithm converges to the correct solution $\mathbf{x}$ provided the matrices $\mathbf{A}$ and $2\mathbf{D}-\mathbf{A}$ are both symmetric positive definite. Note in particular that for the FVM Laplacian discretisation matrices considered in chapter 2, the positive definiteness of matrix $\mathbf{A}$ will guarantee the positive definiteness of $2\mathbf{D}-\mathbf{A}$ since the latter is immediately seen to be symmetric and (weakly) diagonally dominant, which implies symmetric positive semi-definiteness, and diagonally dominant in at least one row.

4.1.1 Jacobi on GPU

It is easily seen that the Jacobi method is easily parallelisable for GPU using the mod. SELL format. In particular, since the diagonal entries are stored separately, the product $(\mathbf{L}_i+\mathbf{U}_i)\mathbf{x}^{(i)}$ is computed by the thread responsible for row i in a like manner to SpMV. It may then access $\mathbf{b}_i$ and $\mathbf{D}_{ii}$ from the arrays `b` and `diag` in an operation coalesced with the other threads in a warp. Finally, the thread sets `x[i]` as $\mathbf{D}_{ii}^{-1}(\mathbf{b}_i-(\mathbf{L}_i+\mathbf{U}_i)\mathbf{x}^{(k)})$, also in a coalesced fashion. A Jacobi iteration may thus be executed on the GPU with only one more global memory transaction per thread (namely getting `b[i]`) as compared to SpMV. Furthermore, the number of uncoalesced accesses will be equal.

4.2 Gauss-Seidel Algorithm

The Gauss-Seidel algorithm is similar to Jacobi, but uses more updated information. Decompose the coefficient matrix $\mathbf{A}$ as in the previous section. Given some guess for the solution $\mathbf{x}^{(k)}$, the algorithm finds an updated guess $\mathbf{x}^{(k+1)}$ in a *forward sweep* defined by

$$(\mathbf{L}+\mathbf{D})\mathbf{x}^{(k+1)}=(\mathbf{b}-\mathbf{U}\mathbf{x}^{(k)}) \tag{4.2}$$

Due to the lower triangular shape of $(\mathbf{L}+\mathbf{D})^{-1}$, $\mathbf{x}^{(k+1)}$ may be found easily by forward-substitution. Indeed, we may also express each forward update by the formula

$$\mathbf{x}^{(k+1)}=\mathbf{D}^{-1}(\mathbf{b}-\mathbf{L}\mathbf{x}^{(k+1)}-\mathbf{U}\mathbf{x}^{(k)}). \tag{4.3}$$

Similarly, in a *backward sweep*, the algorithm finds $\mathbf{x}^{(k+1)}$ by the formula

$$(\mathbf{D}+\mathbf{U})\mathbf{x}^{(k+1)}=(\mathbf{b}-\mathbf{L}\mathbf{x}^{(k)}), \tag{4.4}$$

where again the system may be solved effectively using back-substitution and therefore rewritten

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{U}\mathbf{x}^{(k+1)} - \mathbf{L}\mathbf{x}^{(k)}). \tag{4.5}$$

It is shown in [13] that the Gauss-Seidel algorithm (using either forward or backward sweeps) is convergent for every symmetric positive definite matrix **A**. The problem inherent in parallelising the Gauss-Seidel algorithm lies with the sequential forward- and back-substitutions involved in the formulae (4.2) and (4.4).

4.2.1 Gauss Seidel on GPU by Multicoloring

One of the best-known algorithms for parallel implementation of the Gauss-Seidel algorithm is that of *multicoloring* [19]. In the context of FVM, the idea is to assign to each cell (row) in a mesh (discretisation matrix) a colour, such that no two neighbouring cells have the same colour. (On a regular cartesian mesh, this can clearly be achieved with only two colours, leading to a so-called "red-black" ordering [19].) The rationale is that the values for all cells of the same colour may all be updated in parallel, since the values of interest in these cells do not depend on each other directly.

A multicolored partitioning of a mesh is seen below in figure 4.1. The matrix in (4.6) shows the coloring of the corresponding system of equations when a single field variable is considered.

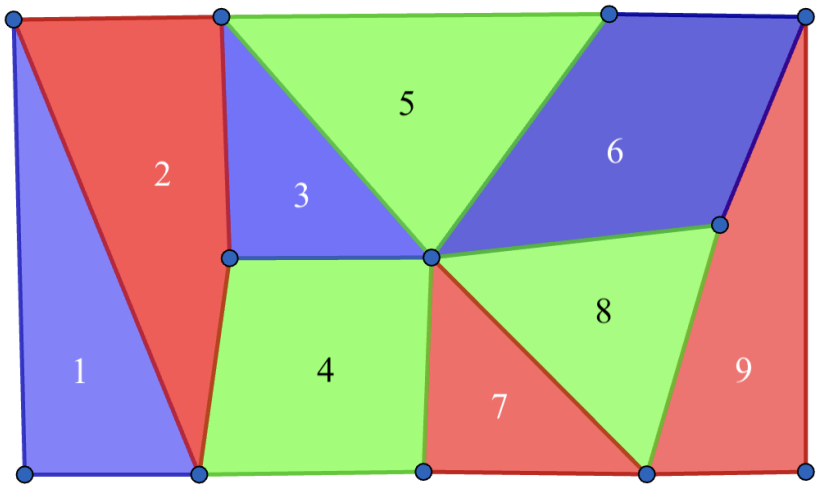
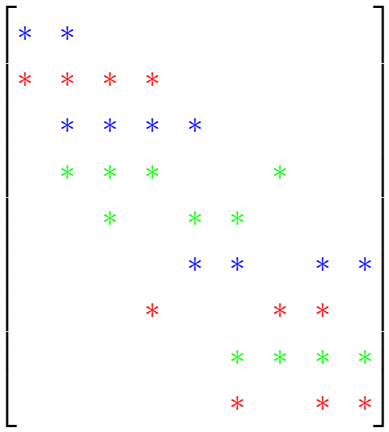


Fig. 4.1 Multicoloring of 2D mesh with 9 Cells

$$\mathbf{A} = \quad (4.6)$$



Multicoloring with mod. SELL

On regular Cartesian grids, speedup factors for red-black Gauss Seidel as high as 200 have been reported relative to a sequential CPU implementation [7]. Such methods, however, rely on the ordered structure of the grid [6] and on separating in memeory the arrays of red and black rows in the coefficient matrix to achieve coalescence during reads from global memory. In building a CG-based solver for the GPU, rows of the coefficient matrix are ordered so as to maximise coalescence during sparse matrix-vector multiplications. Thus, any division of the coefficient matrix in this way would deteriorate the speed of SpMV. Furthermore, the number of colours necessary for an unstructured mesh would be bounded below by the maximal number of neighbour cells, *mnz*, that any given cell in the mesh has. Hence, even if the coefficient matrix was not partitioned in memory, *mnz* kernels would have to be launched in sequence to account for all the colours, potentially with some kernels only updating values for very few cells. We are thus left with a method that is not well suited for our particular GPU implementation.

4.3 Block Jacobi Smoothers

Consider a linear system $\mathbf{Ax} = \mathbf{b}$ divided into $m = n/p$ blocks of size p. The system may then be written

$$\mathbf{Ax} = \begin{bmatrix} \tilde{\mathbf{A}}_{0,0} & \cdots & \tilde{\mathbf{A}}_{1,m-1} \\ \vdots & \ddots & \vdots \\ \tilde{\mathbf{A}}_{m-1,1} & \cdots & \tilde{\mathbf{A}}_{m-1,m-1} \end{bmatrix} \begin{bmatrix} \mathbf{x}_0 \\ \vdots \\ \mathbf{x}_{m-1} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_0 \\ \vdots \\ \mathbf{b}_{m-1} \end{bmatrix} = \mathbf{b}, \quad (4.7)$$

where $\tilde{\mathbf{A}}_{i,j} \in \mathbb{R}^{p\times p}$, $\mathbf{x}_i, \mathbf{b}_i \in \mathbb{R}^p$. In a single *outer* iteration, a block Jacobi smoother tries to separately solve the systems

$$\tilde{\mathbf{A}}_{i,i}\hat{\mathbf{x}}_i^{(k+1)} = \mathbf{b}_i - \tilde{\mathbf{A}}_{i,i}\mathbf{x}_i^{(k)} - \sum_{\substack{j=1\\ j\neq i}}^{m} (\tilde{\mathbf{A}}_{i,j})\mathbf{x}'_j, \quad i \in \{1,\dots,m\} \tag{4.8}$$

$$\Rightarrow \quad \mathbf{x}_i^{(k+1)} = \hat{\mathbf{x}}_i^{(k+1)} + \mathbf{x}_i^{(k)} \tag{4.9}$$

where $\mathbf{x}'_j \in \mathbb{R}^p$ is some approximation to the jth component of the exact solution $\mathbf{x}_j$. If non-zero guess for Several *inner sweeps* (e.g. iterations) of the methods mentioned earlier, such as Jacobi, Gauss-Seidel, or the Conjugate Gradient algorithm (as described in [21]) may be employed on the local systems (4.8), once the right-hand side has been assembled. A starting guess of $\mathbf{x}_i^{(k+1)} = 0$ is used for these inner sweeps. Alternatively, the inverse of the diagonal blocks $\tilde{\mathbf{A}}_{i,i}$ may be computed once prior to initialising a solver. This approach, however, carries a heavy memory cost, and has thus not been implemented in the tests used here.

Convergence in the outer iterations is guaranteed for the symmetric positive definite Laplacian discretisation matrices considered here due to the positive definiteness of the matrix $\mathbf{A} - \tilde{\mathbf{L}} - \tilde{\mathbf{U}}$, where $\tilde{\mathbf{L}}, \tilde{\mathbf{U}}$ denote the block lower- and upper triangular parts of the block decomposition of $\mathbf{A}$ shown in (4.8) [13]. Invertibility and convergence in the inner systems is guaranteed by the positive definiteness of $\mathbf{A}$. In particular, let $\mathbf{x} \in \mathbb{R}^n$ be any non-zero vector with zeros only at indices $ip : (i+1)p$, and $\mathbf{x}_i \in \mathbb{R}^p$ be its constriction where the non-zeros are preserved. Then

$$\mathbf{x}_i^T \tilde{\mathbf{A}}_{i,i}\mathbf{x}_i = \mathbf{x}^T\mathbf{A}\mathbf{x} > 0. \tag{4.10}$$

With such methods, an important point should be made about the terms $\mathbf{x}'_j$ appearing in the right-hand side of (4.8). In a *synchronous* block Jacobi smoother, we simply use the guesses from the previous outer iteration k, i.e. $\mathbf{x}'_j = \mathbf{x}_j^{(k)}$. Another approach which has garnered attention in recent years, however, is that of *block-asynchronous* solvers (smoothers) [1] [24]. When the task of solving the local block systems in (4.8) are distributed across several processes, such solvers are agnostic about whether each block solution iterate $\mathbf{x}'_j$ on the right-hand side has already been updated to $\mathbf{x}_j^{(k+1)}$ (by a different processor) or if it is yet to be updated and still corresponds to $\mathbf{x}_j^{(k)}$. The potential benefit of asynchronous block smoothers is clearly that they may be able to use more updated information when solving the block systems. Asynchronous block Jacobi smoothers can clearly be seen as agnostic versions of *block Gauss-Seidel smoothers*, which sequentially update the block

iterates $\mathbf{x}_i^{(k)}$. This algorithm, however, is less suitable for GPU implementation as it would required sequentially executing thread-blocks.

4.3.1 Block Jacobi Smoothers on GPU

It is clear that any application of block Jacobi smoothers on GPU should choose p corresponding to the number of rows that is most effectively processed in a thread-block (powers of 32 on NVIDIA GPUs). Block smoothers on GPU using Jacobi and Gauss-Seidel sweeps on the local systems (4.8) have been extensively studied in [1], [22], and [24].

For memory access reasons, it is impractical to have threads in the same warp serialising their operations in an inner Gauss-Seidel sweep. Therefore, for the block Jacobi GPU smoother with inner Gauss-Seidel sweeps, warps (rather than threads) may be serialised. Say thread blocks of size p are used where $p/32 = q$. For each block l, matrix $\tilde{A}_{l,l}$ may be written as

$$\tilde{\mathbf{A}}_{l,l} = \tilde{\mathbf{M}}_{l,l} + \tilde{\mathbf{N}}_{l,l} + \tilde{\mathbf{O}}_{l,l}, \tag{4.11}$$

where, $\tilde{\mathbf{N}}_{l,l}$, $\tilde{\mathbf{M}}_{l,l}$, $\tilde{\mathbf{O}}_{l,l}$ consist of the diagonal, upper triangular, and lower triangular sets of 32×32 sub-matrices of $\tilde{\mathbf{A}}_{l,l}$. Further decompose matrix $\tilde{\mathbf{N}}_{l,l}$ into *its* lower triangular $\tilde{\mathbf{N}}_{l,l}^L$, diagonal $\tilde{\mathbf{D}}_{l,l}$, and upper triangular $\tilde{\mathbf{N}}_{l,l}^U$ parts as

$$\tilde{\mathbf{N}}_{l,l} = \tilde{\mathbf{N}}_{l,l}^L + \tilde{\mathbf{D}}_{l,l} + \tilde{\mathbf{N}}_{l,l}^U, \tag{4.12}$$

where $\tilde{\mathbf{D}}_{l,l}$ is just the diagonal of $\tilde{\mathbf{A}}_{l,l}$. After k system-wide block Gauss-Seidel iterations, define by $\mathbf{b}_l^{(k)}$ the right-hand side of system (4.8) for block l:

$$\mathbf{b}_l^{(k)} = \mathbf{b}_l - \tilde{\mathbf{A}}_{l,l}\mathbf{x}_l^{(k)} - \sum_{\substack{j=1 \\ j\neq l}}^{m} (\tilde{\mathbf{A}}_{l,j})\mathbf{x}_j^{(k)}. \tag{4.13}$$

Under this paradigm, the warp-wise forward Gauss-Seidel sweep $s+1$ on block l may be written

$$\hat{\mathbf{x}}_l^{(s+1)} = \tilde{\mathbf{D}}_{l,l}^{-1}\Big(\mathbf{b}_l^{(k)} - \big(\tilde{\mathbf{M}}_{l,l} + \tilde{\mathbf{N}}_{l,l}^L\big)\hat{\mathbf{x}}_l^{(s+1)} - \big(\tilde{\mathbf{N}}_{l,l}^U + \tilde{\mathbf{O}}_{l,l}\big)\hat{\mathbf{x}}_l^{(s)}\Big). \tag{4.14}$$

On the other hand, it is easily seen that inner sweeps of the CG and Jacobi algorithms can be implemented to update all components of the block iterate $\mathbf{x}_l$ synchronously.

As for the synchronous implementation of the outer iterations of the block-Jacobi smoother, thread-block i may collect existing block iterates $\mathbf{x}_j^{(k)}$, appearing on the right-hand side of (4.8), from a global memory array `x_old`. After computation of the update

$\mathbf{x}_i^{(k+1)}$, it can set the corresponding section of a global memory array `x_new` to avoid conflicts with any blocks reading from `x_old`. Note that a simple Jacobi iteration as in (4.1) may be implemented using this method with a single inner Jacobi sweep.

In an asynchronous version (for any type of inner sweeps) of block Jacobi, however, each thread-block i reads right-hand side iterates $\mathbf{x}'_j$ from and writes the update $\mathbf{x}_i^{(k+1)}$ to the same array `x`. Immediately thereafter, the block must call `threadfence_system()` (on NVIDIA devices) to ensure other un-launched blocks see the update before reading from `x`.

Block Smoothers with mod. SELL format

The mod. SELL format is well suited for block Jacobi smoothers for the same reasons as outlined in sub-section 4.1.1. A block smoother kernel may be defined such that, in each thread-block l, the linear system in equation (4.8) may be assembled with only one additional read from global memory, as compared to SpMV. Furthermore, the coefficient matrix $\mathbf{A}_{l,l}$ may be stored in ELLPACK format (see [21]) within the thread-block.

In particular, let `b` and `x` denote the global memory arrays on which are stored the right-hand side $\mathbf{b}$ and the current solution guess $\mathbf{x}'$ for the entire sparse linear system, respectively. Let `slice_mnz` denote the maximal number of off-diagonal, non-zero elements on a row of the block matrix $\mathbf{A}_{l,l}$. Each thread may collect this number from a global memory array `slice_mnz_arr`. Recall that the number of threads in each thread-block corresponds to the slice height and can be accessed with `blockDim.x`.

To store the matrix $\mathbf{A}_{l,l}$ in ELLPACK format, we need one floating point array `loc_nbrs_coef` for the entries and one integer array `loc_nbrs` for the column indices, each of size `slice_mnz * blockDim.x`. Since the register space available to each thread is very limited, these arrays should be stored in shared memory to avoid *spills* to global memory (see [17]). The diagonal matrix entry on each row must also be collected. Further, the block right-hand side $\mathbf{b}'_l$ of (4.8) should be stored in a shared memory segment `local_vec`. Thus, at the start of a smoothing kernel, the shared memory, whose size may be allocated before the launch of the thread-block, may be partitioned into three separate pointers for each thread as follows:

```
1    //...other steps collecting slice_mnz and slice height etc
2
3    extern __shared__ char smem[];
4    size_t local_vec_size = blockDim.x * sizeof(double);
5    size_t col_ind_size = blockDim.x * slice_mnz * sizeof(int);
6    double* smem = reinterpret_cast<double*>(smem_u);
7    int* loc_nbrs = reinterpret_cast<int*>(smem_u + local_vec_size);
8    double* loc_nbrs_coef = reinterpret_cast<double*>(smem_u +
       local_vec_size + col_ind_size);
```

```
9
10      //Prepare SELL arrays for local block matrix
11      for(int i = 0; i < slice_mnz; i++){
12        loc_nbrs[i*blockDim.x + threadIdx.x] = -1;
13        loc_nbrs_coef[i*blockDim.x + threadIdx.x] = 0;
14      }
15      //get diagonal matrix entry on row
16      double diag_entry = diag[blockIdx.x*blockDim.x + threadIdx.x];
```

Note also that the `loc_nbrs_coef` and `loc_nbrs` arrays have been initiated in line with ELLPACK format. Having completed these preparatory steps, the `local_vec` segment is then populated with the existing **x**-guess for the corresponding row:

```
1 double x_i = x[blockIdx.x*blockDim.x + threadIdx.x]
2 local_vec[threadIdx.x] = x_i;
3 __syncthreads();
```

We can then initiate the (local) right-hand side variable `b_i` to the corresponding entry of the global memory array `b`. Further, we subtract from `b_i` the diagonal matrix entry (which is also a diagonal of $\tilde{\mathbf{A}}_{l,l}$) multiplied by `x_i`:

```
1 double b_i = b[blockIdx.x*blockDim.x + threadIdx.x];
2 b_i -= x_i*diag_entry;
```

Now, let each thread initiate the variable `int num_nbrs = 0`. Threads may then iterate over the `entries` and `col_idx` arrays in a like manner to SpMV. Upon encountering an entry, `entries[k]`, for which `col_idx[k]` < 0 (i.e. the entry is in $\mathbf{A}_{l,l}$) the thread appends to the `loc_nbrs` and `loc_nbrs_coef` arrays and subtracts `local_vec[-(col_idx[k]+1)]` multiplied by `entries[k]` from `b_i` with the following commands

```
1 if(col_idx[k] < 0){
2     loc_nbrs[num_nbrs*blockDim.x + threadIdx.x] = -(col_idx[k] + 1);
3     loc_nbrs_coef[num_nbrs*blockDim.x + threadIdx.x] = entries[k];
4     b_i -= local_vec[-(col\_idx[k]+1)]*entry;
5     num_nbrs += 1;
6 }
```

Upon encountering an entry `entries[k]` for which `col_idx[k]` > 0 (i.e. the entry is not in $\mathbf{A}_{l,l}$), the entry is multiplied by the corresponding entry in `x` and subtracted from `b_i`:

```
1 else if(col_idx[k] > 0){
2     b_i -= entries[k]*x[col_idx[k] - 1];
3 }
```

When all threads have completed iterating over their respective row, we place the local right-hand side values `b_i` in `local_vec`, replacing the `x_i` values (which must be kept by each thread in registers due to expression (4.9)):

```
__syncthreads();
local_vec[threadIdx.x] -= b_i
```

When this process is complete, the local linear system in (4.8) is stored entirely in the shared memory and registers of thread-block l. The block smoother kernel `smooth_kern` demonstrating this process is available in the software bundle accompanying this work [1]. If $||\mathbf{b}'_l||_2^2||$ is not already below the predefied tolerance, the kernel can call `__device__` functions implementing inner sweeps of the Jacobi, Gauss-Seidel, or CG algorithms on the system, where `local_vec` is used to store different vectors as needed.

Shared Memory Size

Crucially, in default mode, only 48 kilo-bytes (KB) shared memory is available to each thread-block on NVIDIA devices. The amount of shared memory that must be declared for each thread-block before launch of `smooth_kern` is the sum of the (maximal) size of the segments into which we will divide it. For a given matrix, the largest possible size of arrays `loc_nbrs` and `loc_nbrs_coef` is proportional to the highest number of off-diagonal, non-zero elements in any row across all matrices $\mathbf{A}_{i,i}$. The required number of bytes that must be declared for shared memory may thus be computed before kernel launch as

```
    smooth_shared_size = (blockDim.x*(sizeof(double) + max_slice_mnz
       *(sizeof(double) + sizeof(int))));
```

where `max_slice_mnz` denotes the largest entry in `slice_mnz_arr`. As described in [21], for an FVM matrix `max_slice_mnz` may, in the worst case be as large as ≈ 20. If thread-blocks of size 512 are used, we would require a shared memory segment of size

```
(512*8 + 512*20*(8 + 4)) bytes = 126.976KB.
```
(4.15)

While this is a worst-case scenario, it shows that in some cases the required shared memory size may significantly exceed the default. However, on newer NVIDIA devices, the amount of shared memory for a kernel `kern` may be increased to some amount of bytes `num_bytes` using the command

```
cudaFuncSetAttribute(my_kern,
   cudaFuncAttributeMaxDynamicSharedMemorySize, num_bytes);
```

[1] Note that `smooth_kern` defines a few more variables in order to avoid double reads from cache

up to some maximum of 164KB [17]. While increasing the size of shared memory takes away resources for L1 cache, we expect most discretisation matrices to display a much smaller value for `max_slice_mnz`.

4.4 Smoother Tests

We show here the convergence and timing tests for the smoothing algorithms discussed above. Their performance is compared with a CPU implementation of the Gauss-Seidel algorithm using only forward- or alternating forward and backward sweeps, respectively. Block Jacobi smoothers (synchronous and asynchronous) were implemented whereupon the inner (block) system as in (4.8) is solved using a variable number of Jacobi, Gauss-Seidel, or conjugate gradient sweeps, respectively.

Block conjugate gradient and Jacobi sweeps are carried out on each system (4.8) in the conventional manner. Only the forward sweep as in equation (4.14) was implemented for the block Gauss-Seidel algorithm.

Tests on GPU used an NVIDIA A30 whereas tests on CPU used an Intel® Xeon® Silver 4314 CPU operating on a single core. Matrices on CPU were stored and manipulated using compressed sparse row format (CSR) as described previously in [21]. Slice height of mod. SELL was 512 and CM ordering used on that format.

For each algorithm, 1000 *outer* iterations were carried out for matrices "100K", "parabolic_fem", "thermal2", and "G3_circuit" with a zero starting guess $\mathbf{x}^{(0)}$ and corresponding right-hand sides $\mathbf{b}$ collected from [5]. The exception was "G3_circuit", for which no such vector was available, and an arbitrary right-hand side of 1s in the first 100 elements and zeros elsewhere was implemented. In each outer iteration of the GPU algorithms, the smoother kernel ran to convergence (below tolerance 10^{-15} on the squared *inner* residual norm) or carried out a maximum of 100 inner sweeps on each of the block-systems (4.8) using either block Gauss-Seidel, block Jacobi or block CG algorithms.

Figure 4.2 shows the convergence rates of the aforementioned algorithms against time. Notably, all timings were collected on CPU-side using `chrono` and included the time for the CPU or GPU to compute the system-wide *outer* residual after every 10th iteration for the respective algorithm. Calculating the this involved calculating the product $\mathbf{A}\mathbf{x}^{(k)}$, followed by the residual $\mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}$, and finally the squared residual norm $||\mathbf{r}||_2^2 = \mathbf{r}^T\mathbf{r}$. In the case of the GPU, the residual calculation time included the time to copy the residual back to CPU. The average time taken to compute $1000/10 = 100$ residuals on the CPU and GPU, respectively, for each matrix may be seen from table 4.1 along with the total computation time over the 1000 iterations.

Matrix	100K	thermal2	parabolic_fem	G3_circuit
Residual CPU Comp. Time	6×10^4	2.0×10^6	4.7×10^5	1.3×10^6
Residual CPU Comp. Time	2.5×10^5	2.8×10^5	2.6×10^5	2.8×10^5
Fwd. Gauss-Seidel CPU	1.1×10^6	3.0×10^7	4.9×10^6	2.2×10^7
Fwd.&Bck. Gauss-Seidel CPU	8.9×10^5	3.1×10^7	5.3×10^6	2.2×10^7
Block Jacobi GPU (sync)	2.9×10^5	5.4×10^5	3.9×10^5	3.7×10^5
Block Jacobi GPU (ssync)	3.0×10^5	6.6×10^5	4.7×10^5	5.4×10^5
Block Gauss-Seidel GPU (sync)	1.4×10^6	1.8×10^6	1.8×10^6	1.4×10^6
Block Gauss-Seidel GPU (async)	1.4×10^6	1.8×10^6	1.9×10^6	1.4×10^6
Block CG GPU (sync)	4.3×10^5	4.8×10^5	3.1×10^5	6.1×10^5
Block CG GPU (async)	4.5×10^5	6.1×10^5	3.7×10^5	7.8×10^5

Table 4.1 Total computation time (microsenconds) over 1000 smoothing iterations. Residual computation times are sum over 100 residual computations and includes time to copy the squared residual norm from GPU to CPU where relevant.

4.4.1 Smoother Results: Discussion

As may be seen from table 4.1 and figure 4.2, all smoothing algorithms achieve approximately the same reduction in the residual over 1000 iterations. The factor of speedup on GPU versus CPU is increasing in the size of the matrix. For matrices "thermal2" and "G3_circuit", which each have over one million rows, the GPU block Jacobi smoothers using inner Jacobi or CG sweeps are about two orders of magnitude faster than the CPU implementations of Gauss-Seidel. The speedup for the smaller matrices "parabolic_fem" and "100K" is about one order of magnitude. The order of magnitude speedup for the GPU block Jacobi smoother with inner Gauss-Seidel sweeps is about one order of magnitude for the larger matrices. For the smaller matrices, the computation time for the Gauss-Seidel smoother on CPU and the block Jacobi smoother with Gauss-Seidel sweeps on GPU is of the same order of magnitude. Significantly, Gauss-Seidel smoother on CPU completes 1000 iterations faster than the block Jacobi smoother with Gauss-Seidel sweeps for the "100K" matrix. These results point to the superiority of fully paralleliseable algorithms like block Jacobi and CG for GPU implementation.

In general the GPU computation times increase at a rate much slower than the corresponding increase in matrix size (see table 3.1). Furthermore, a significant proportion of time is spent for the GPU to compute the system-wide residuals at every 10th iteration. However, as the computation time for 100 residuals on GPU stays approximately constant

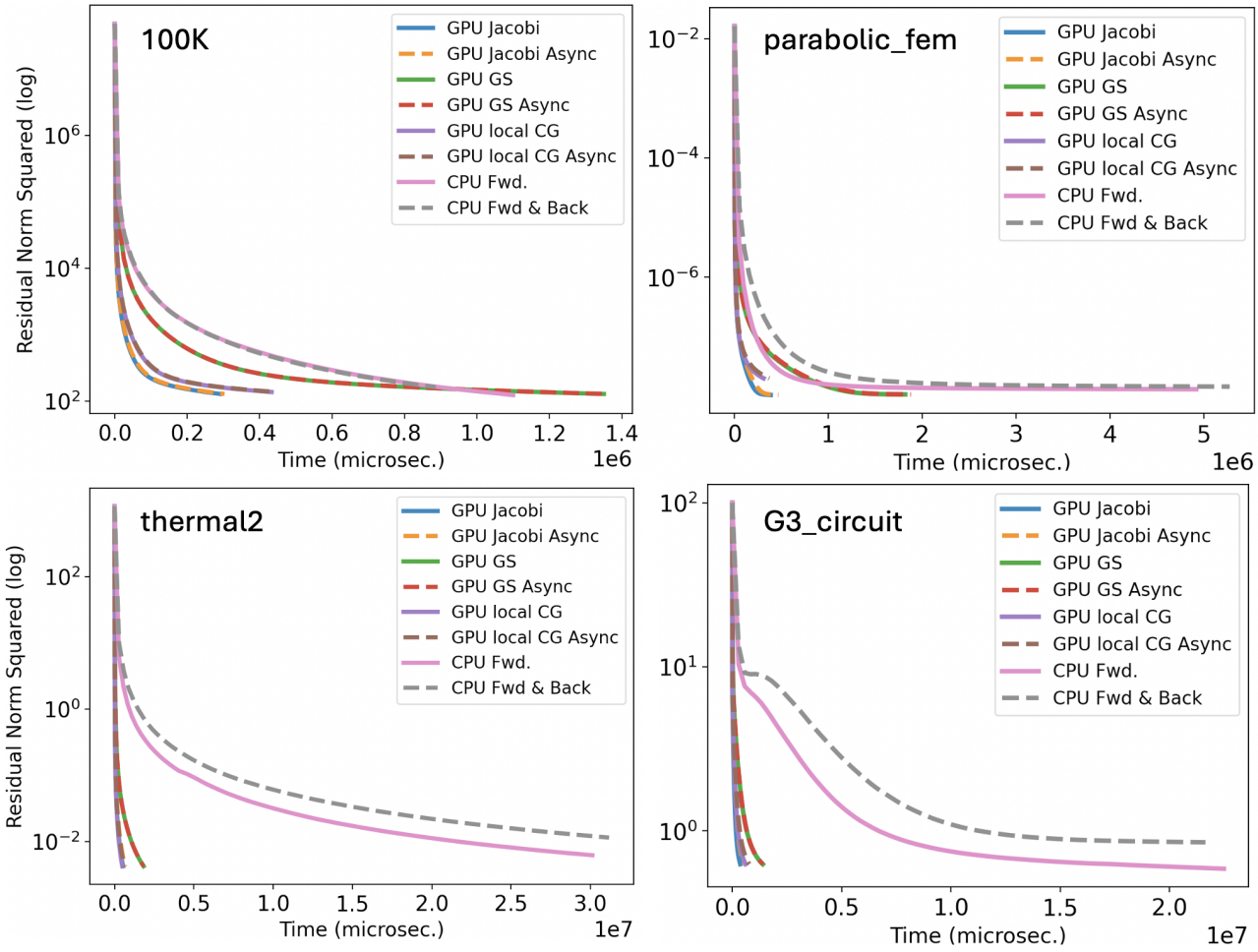


Fig. 4.2 Squared Residual Norms (log10) against time (in μs) for various smoothing algorithms. Each algorithm was run for 1000 iterations, with residuals being sampled every 10 iterations

in the matrix size, we expect this to be a result of communication costs rather than a poorly implemented SpMV operation or vector reduction (the reduction operation was implemented as per NVIDIA recommendations in [8]). By contrast, the residual computation times for CPU may be seen to increase approximately linearly in matrix size. The significant amount of time spent on residual computation indicates that GPU smoother implementations benefit from checking the residual only at rare intervals.

While hardly visible in figure 4.2, table 3.1 attests to the fact that the synchronised versions of the GPU smoothers are slightly faster than the corresponding asynchronous versions. We show in figure 4.3 the corresponding plots with the CPU Gauss-Seidel computation times omitted. On the matrices tested, it may be seen that the asynchronous implementations do not

provide any reduction in the residual, relative to the synchronous implementations. However, they do increase the computational time.

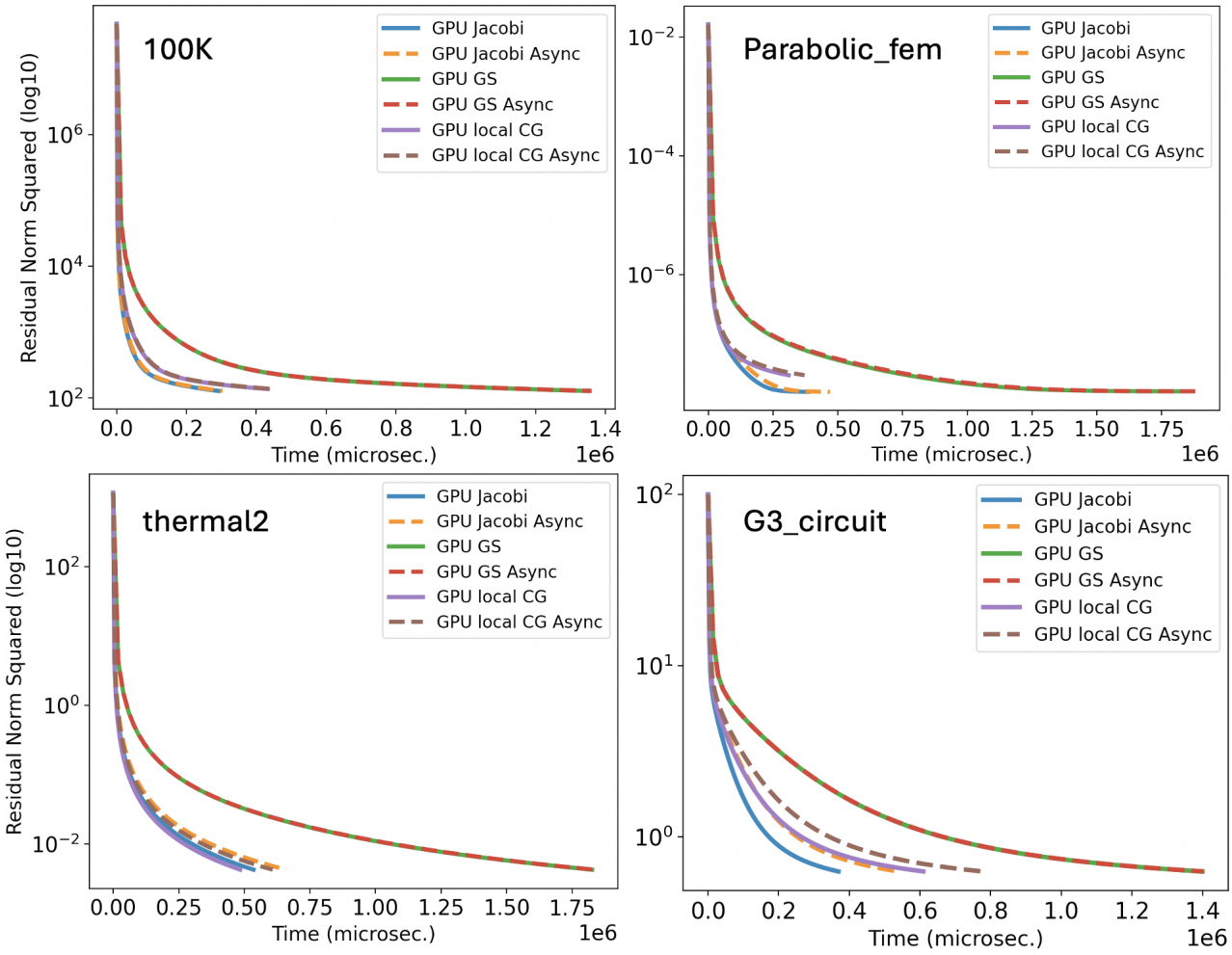


Fig. 4.3 Figure 4.2 with CPU timings omitted to allow GPU timings to be more clearly distinguished.

Chapter 5

Preconditioners

5.1 The Condition Number

The condition number with respect to the p-norm, $\kappa_p(\mathbf{A})$, of a matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ is defined as

$$\kappa_p(\mathbf{A}) = ||\mathbf{A}||_p||\mathbf{A}^{-1}||_p. \tag{5.1}$$

For symmetric positive definite matrices all eigenvalues are positive and we have

$$||\mathbf{A}||_2 = \lambda_{max} \tag{5.2}$$

where λ_{max} denotes the largest eigenvalue of $\mathbf{A}$. Since the inverse of an SPD matrix is also SPD and the eigenvalues of the inverse $\mathbf{A}^{-1}$ are simply the inverse of the eigenvalues $\mathbf{A}$, we get

$$\kappa_2(\mathbf{A}) = \frac{\lambda_{\max}}{\lambda_{\min}}. \tag{5.3}$$

For a matrix $\mathbf{A}$, we shall refer to this number simply as $\kappa(\mathbf{A})$. In [19], it is shown that the error $\mathbf{e}^{(k)} = (\mathbf{x} - \mathbf{x}^{(k)})$ after the kth step of the conjugate gradient algorithm satisfies

$$||\mathbf{e}^{(k)}||_{\mathbf{A}} \leq 2\Big(\frac{\sqrt{\kappa(\mathbf{A})}-1}{\sqrt{\kappa(\mathbf{A})}+1}\Big)^k ||\mathbf{x} - \mathbf{x}^{(0)}||_{\mathbf{A}}, \tag{5.4}$$

where $||\mathbf{v}||_{\mathbf{A}} = \sqrt{\mathbf{v}^T\mathbf{A}\mathbf{v}}$ and $\mathbf{x}$ denotes the exact solution and $\mathbf{x}^{(0)}$ denotes an initial guess. We see thus that the upper bound on the error is highly dependent on and increasing in $\kappa(\mathbf{A})$.

A preconditioner is a matrix $\mathbf{M} \in \mathbb{R}^{n\times n}$ whose inverse may be multiplied with a matrix $\mathbf{A}$ in the hope that the product $\mathbf{M}^{-1}\mathbf{A}$ has a smaller condition number than that of $\mathbf{A}$.

5.2 Preconditioned Conjugate Gradient Algorithm

In the context of the conjugate gradient algorithm, an invertible preconditioner $\mathbf{M}$, must typically be applied symmetrically in order to maintain the SPD property of the underlying system. Instead of solving the system $\mathbf{Ax} = \mathbf{b}$, we apply the algorithm on the system

$$\mathbf{E}^{-1}\mathbf{A}(\mathbf{E}^{-1})^T\mathbf{y} = \mathbf{E}^{-1}\mathbf{b} \quad \text{where} \quad \mathbf{E}^T\mathbf{x} = \mathbf{y},\ \mathbf{E}\mathbf{E}^T = \mathbf{M}. \tag{5.5}$$

Note that the matrix $\mathbf{E}^{-1}\mathbf{A}(\mathbf{E}^{-1})^T$ is also symmetric positive definite since $\mathbf{A}$ is. See further that the matrices $\mathbf{M}^{-1}\mathbf{A}$ and $\mathbf{E}^{-1}\mathbf{A}(\mathbf{E}^{-1})^T$ are *similar*, as per the relation

$$(\mathbf{E}^T)^{-1}(\mathbf{E}^{-1}\mathbf{A}(\mathbf{E}^{-1})^T)\mathbf{E}^T = \mathbf{M}^{-1}\mathbf{A}. \tag{5.6}$$

Similar matrices have the same eigenvalues, and so we see that $\mathbf{E}^{-1}\mathbf{A}(\mathbf{E}^{-1})^T$ and $\mathbf{M}^{-1}\mathbf{A}$ will have the same condition number [19].

The approach above requires that we know the matrix $\mathbf{E}$ and that we are easily able to invert it. However, it is not necessary to explicitly multiply the matrix $\mathbf{M}$ nor $\mathbf{E}$ with $\mathbf{A}$ for the conjugate gradient algorithm to be applied to the system in (5.5). Rather, denote by $\widehat{\mathbf{r}}^{(k)}$ the residuals, $\widehat{\mathbf{d}}^{(k)}$ the directions, and $\mathbf{y}^{(k)}$ the solution iterates that will appear in the algorithm, respectively (see the previous part of the work [21] for explanation of conjugate gradient algorithm). See that we then have the relation

$$\begin{aligned}
\widehat{\mathbf{r}}^{(k)} &= \mathbf{E}^{-1}\left(\mathbf{b} - \mathbf{A}(\mathbf{E}^{-1})^T\mathbf{y}^{(k)}\right) && (5.7)\\
&= \mathbf{E}^{-1}\left(\mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}\right) && (5.8)\\
&= \mathbf{E}^{-1}\mathbf{r}^{(k)}, && (5.9)
\end{aligned}$$

where we let $\mathbf{x}^{(k)}$ and $\mathbf{r}^{(k)}$ denote solution and residual iterates, respectively, in the original system $\mathbf{Ax} = \mathbf{b}$, derived from their preconditioned counterparts in accordance with system (5.5).

Define now the vectors $\mathbf{z}^{(k)} = \mathbf{M}^{-1}\mathbf{r}^{(k)}$ and $\mathbf{p}^{(k)} = (\mathbf{E}^{-1})^T\widehat{\mathbf{d}}^{(k)}$. We see then that the step-size $\widehat{\alpha}^{(k)}$ may be written as

$$\widehat{\alpha}^{(k)} = \frac{(\widehat{\mathbf{r}}^{(k)})^T\widehat{\mathbf{r}}^{(k)}}{(\widehat{\mathbf{d}}^{(k)})^T\mathbf{E}^{-1}\mathbf{A}(\mathbf{E}^{-1})^T\widehat{\mathbf{d}}^{(k)}} \tag{5.10}$$

$$= \frac{(\mathbf{E}^{-1}\mathbf{r}^{(k)})^T\mathbf{E}^{-1}\mathbf{r}^{(k)}}{(\mathbf{p}^{(k)})^T\mathbf{A}\mathbf{p}^{(k)}} \tag{5.11}$$

$$= \frac{\mathbf{r}^{(k)})^T\mathbf{M}^{-1}\mathbf{r}^{(k)}}{(\mathbf{p}^{(k)})^T\mathbf{A}\mathbf{p}^{(k)}} \tag{5.12}$$

$$= \frac{\mathbf{r}^{(k)})^T\mathbf{z}^{(k)}}{(\mathbf{p}^{(k)})^T\mathbf{A}\mathbf{p}^{(k)}} \tag{5.13}$$

Furthermore, recall that the next solution and residual iterates may be found as

$$\mathbf{y}^{(k+1)} = \mathbf{y}^{(k)} + \widehat{\alpha}^{(k)}\widehat{\mathbf{d}}^{(k)} \tag{5.14}$$

$$\widehat{\mathbf{r}}^{(k+1)} = \widehat{\mathbf{r}}^{(k)} - \widehat{\alpha}^{(k)}\mathbf{E}^{-1}\mathbf{A}(\mathbf{E}^{-1})^T\widehat{\mathbf{d}}^{(k)}. \tag{5.15}$$

Multiplying equation (5.14) by $(\mathbf{E}^{-1})^T$ and equation (5.15) by $\mathbf{E}$, see that if we plug in for $\mathbf{p}^{(k)}$, we can also update the new iterates $\mathbf{x}^{(k)}$ and $\mathbf{r}^{(k)}$ for the original system as per

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \widehat{\alpha}^{(k)}(\mathbf{E}^{-1})^T\widehat{\mathbf{d}}^{(k)} = \mathbf{x}^{(k)} + \widehat{\alpha}^{(k)}\mathbf{p}^{(k)} \tag{5.16}$$

$$\mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} - \widehat{\alpha}^{(k)}\mathbf{A}(\mathbf{E}^{-1})^T\widehat{\mathbf{d}}^{(k)} = \mathbf{r}^{(k)} - \widehat{\alpha}^{(k)}\mathbf{A}\mathbf{p}^{(k)}. \tag{5.17}$$

Following the same procedure as for the term $\widehat{\alpha}^{(k)}$, see that if we solve $\mathbf{M}\mathbf{z}^{(k+1)} = \mathbf{r}^{(k+1)}$ for $\mathbf{z}^{(k+1)}$, we may write

$$\widehat{\beta}^{(k)} = \frac{(\widehat{\mathbf{r}}^{(k+1)})^T\widehat{\mathbf{r}}^{(k+1)}}{(\widehat{\mathbf{r}}^{(k)})^T\widehat{\mathbf{r}}^{(k)}} = \frac{(\mathbf{r}^{(k+1)})^T\mathbf{z}^{(k+1)}}{(\mathbf{r}^{(k)})^T\mathbf{z}^{(k)}}. \tag{5.18}$$

Finally, we see that we may find the new search direction as

$$\widehat{\mathbf{d}}^{(k+1)} = \widehat{\mathbf{r}}^{(k+1)} + \widehat{\beta}^{(k)}\widehat{\mathbf{d}}^{(k)} \tag{5.19}$$

and multiplying this equation by $(\mathbf{E}^{-1})^T$ gives

$$\mathbf{p}^{(k+1)} = \mathbf{M}^{-1}\mathbf{r}^{(k+1)} + \widehat{\beta}^{(k)}\widehat{\mathbf{p}}^{(k)} = \mathbf{z}^{(k+1)} + \widehat{\beta}^{(k)}\mathbf{p}^{(k)}. \tag{5.20}$$

[H] Looking at equations (5.10 - 5.20) we notice that computing the preconditioned iterates $\mathbf{y}^{(k)}$, $\widehat{\mathbf{d}}^{(k)}$, $\widehat{\mathbf{r}}^{(k)}$ becomes redundant. As we are only interested in the solution and residual

iterates for the original system $\mathbf{Ax} = \mathbf{b}$, the preconditioned conjugate gradient algorithm appears: We see then that, compared to the original conjugate gradient algorithm, this

Algorithm 1 Preconditioned Conjugate Gradient Algorithm

Input: Symmetric positive definite matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, vector $\mathbf{b} \in \mathbb{R}^n$, initial guess $\mathbf{x}^{(0)} \in \mathbb{R}^n$, tolerance $\varepsilon \in \mathbb{R}_+$, symmetric positive definite preconditioner $\mathbf{M} \in \mathbb{R}^{n \times n}$
Output: Approximate solution $\mathbf{x}$ to $\mathbf{Ax} = \mathbf{b}$
if $\mathbf{x}^{(0)} \neq 0$ **then**
    $\mathbf{b} \leftarrow \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
    $\hat{\mathbf{x}}^{(0)} \leftarrow \mathbf{x}^{(0)}$
    $\mathbf{x}^{(0)} \leftarrow 0$
end if
$\mathbf{r}^{(0)} \leftarrow \mathbf{b}$
solve for $\mathbf{z}^{(0)}$: $\mathbf{M}\mathbf{z}^{(0)} = \mathbf{r}^{(0)}$
$\mathbf{p}^{(0)} \leftarrow \mathbf{z}^{(0)}$
$k \leftarrow 0$
for $k = 0 : (n-1)$ **do**
    $\widehat{\alpha}^{(k)} \leftarrow \frac{\mathbf{r}^{(k)})^T \mathbf{z}^{(k)}}{(\mathbf{p}^{(k)})^T \mathbf{A}\mathbf{p}^{(k)}}$
    $\mathbf{x}_{k+1} \leftarrow \mathbf{x}^{(k)} + \widehat{\alpha}^{(k)} \mathbf{p}^{(k)}$
    $\mathbf{r}^{(k+1)} \leftarrow \mathbf{r}^{(k)} - \widehat{\alpha}_k \mathbf{A}\mathbf{p}^{(k)}$
    if $(\mathbf{r}^{(k+1)})^T \mathbf{r}^{(k+1)} < \varepsilon$ **then**
        break
    end if
    Solve $\mathbf{M}\mathbf{z}^{(k+1)} = \mathbf{r}^{(k+1)}$
    $\beta_k \leftarrow \frac{(\mathbf{r}^{(k+1)})^T \mathbf{z}^{(k+1)}}{\mathbf{r}^{(k)})^T \mathbf{z}^{(k)}}$
    $\mathbf{p}^{(k+1)} \leftarrow \mathbf{z}^{(k+1)} + \widehat{\beta}^{(k)} \mathbf{p}^{(k)}$
    $k \leftarrow k + 1$
end for
if Input $\mathbf{x}_0$ was 0 **then**
    return $\mathbf{x}_{k+1}$
else
    return $\mathbf{x}_{k+1} + \hat{\mathbf{x}}_0$
end if

approach requires only that we additionally store the preconditioner matrix $\mathbf{M}$, whose associated linear system $\mathbf{M}\mathbf{z}^{(k+1)} = \mathbf{r}^{(k+1)}$ we must be quickly able to solve, and the vector $\mathbf{z}^{(k)}$ at each iteration.

5.3 Incomplete Cholesky Preconditioner

It is a well-known result in linear algebra that every symmetric positive definite matrix $\mathbf{A}$ has a Cholesky decomposition of the form $\mathbf{A} = \mathbf{L}\mathbf{L}^T$, where $\mathbf{L}$ is an invertible lower triangular matrix. The construction of the matrix $\mathbf{L}$ is shown in algorithm 2[19].

Algorithm 2 Cholesky Decomposition

1: **Input:** Symmetric positive definite matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$
2: **Output:** Lower triangular matrix $\mathbf{L} \in \mathbb{R}^{n\times n}$ satisfying $\mathbf{A} = \mathbf{L}\mathbf{L}^T$
3: $\mathbf{L} \leftarrow \mathbf{0}$
4: **for** $i = 0 : (n-1)$ **do**
5: $\quad l_{i,i} \leftarrow \sqrt{\left(a_{i,i} - \sum_{k=0}^{i-1} l_{i,k}^2\right)}$
6: $\quad$ **for** $j = (i+1) : (n-1)$ **do**
7: $\quad\quad l_{j,i} \leftarrow \frac{1}{l_{i,i}}\left(a_{j,i} - \sum_{k=0}^{i-1} l_{i,k} l_{j,k}\right)$
8: $\quad$ **end for**
9: **end for**

Note that if the algorithm was used in its exact for as above and we constructed the lower triangular matrix $\mathbf{L}$, the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ could be easily solve using forward and back substitution, respectively. In particular, we would have

$$(\mathbf{L}\mathbf{L}^T)^{-1}\mathbf{A}\mathbf{x} = (\mathbf{L}\mathbf{L}^T)^{-1}\mathbf{b} \tag{5.21}$$

$$\Rightarrow \quad \mathbf{x} = (\mathbf{L}\mathbf{L}^T)^{-1}\mathbf{b} \tag{5.22}$$

Clearly, the system $\mathbf{L}\mathbf{y} = \mathbf{b}$ may be solved easily using forward substitution. We may then solve the system $\mathbf{L}^T\mathbf{x} = \mathbf{y}$ using back-substitution, whereupon we would have $\mathbf{x} = (\mathbf{L}\mathbf{L}^T)^{-1}\mathbf{b}$. For a large sparse matrix $\mathbf{A}$, however, we see that in general, the matrix $\mathbf{L}$ will become dense due to line 7 of algorithm 2. For very large matrices, we will thus run into memory constraints in any computational implementation. Furthermore, see that in order to compute the diagonal elements $l_{i,i}$, we must generally compute i squares, followed by $i-1$ additions, one subtraction, and one square root. This makes for $2i+1$ floating point operations (FLOPS). The computational complexity of computing the element $l_{j,i}$ is easily seen to be the same. It follows then that to compute column i of $\mathbf{L}$, we must perform

$$2i+1+(n-1-i)(2i+1) = (n-i)(2i+1) \quad \text{FLOPS.} \tag{5.23}$$

Accumulating this over all rows, we see that the computational complexity of the Cholesky decomposition is

$$C(\text{Cholesky}) = \sum_{i=0}^{n-1}(n-i)(2i+1) \tag{5.24}$$

$$= \sum_{i=0}^{n-1}(2ni+n-2i^2-i) \tag{5.25}$$

$$= \frac{2n(n^2-n)}{2} + n^2 - \frac{2(n-1)n(2n-1)}{6} - \frac{(n^2-n)}{2} \tag{5.26}$$

$$= \frac{n^3}{3} + \frac{n^2}{2} - \frac{n}{6}. \tag{5.27}$$

This is on the same order of complexity as inverting the matrix $\mathbf{A}$ in the first place using Gaussian elimination, and is so considered too expensive.

The level zero incomplete Cholesky factorisation (algorithm 3) is a modification of algorithm 2 in which only elements $l_{i,j}$ for which $a_{i,j} \neq 0$ are computed, the rest being left as zero. By extension, the *level k* incomplete Cholesky factorisation $\mathbf{L}^{(k)}$ computes only elements $l_{i,j}$ for which $(a^k)_{i,j} \neq 0$, where $(a^k)_{i,j}$ denotes the element in row i and column j of the kth multiplicative power $\mathbf{A}^k$ of $\mathbf{A}$. The hope is that the condition number of $\mathbf{M}^{-1}\mathbf{A}$ will be lower than that of $\mathbf{A}$ to achieve faster convergence of the preconditioned conjugate gradient algorithm 1 with the matrix $\mathbf{M} = \mathbf{L}^{(k)}(\mathbf{L}^{(k)})^T$.

Algorithm 3 Incomplete Cholesky Decomposition

Input: Symmetric positive definite matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$,
sparsity pattern $P = \{(i,j)|(a^k)_{i,j} \neq 0\}$
Output: Lower triangular matrix $\mathbf{L} \in \mathbb{R}^{n\times n}$
$\mathbf{L} \leftarrow \mathbf{0}$
for $i = 0 : (n-1)$ **do**
  $l_{i,i} \leftarrow \sqrt{\left(a_{i,i} - \sum_{k=0}^{i-1} l_{i,k}^2\right)}$
  for $j = (i+1) : (n-1)$ **do**
    if $(i,j) \in P$ **then**
      $l_{j,i} \leftarrow \frac{1}{l_{i,i}}\left(a_{j,i} - \sum_{k=0}^{i-1} l_{i,k}l_{j,k}\right)$
    end if
  end for
end for

Computing each diagonal element of the level 0 incomplete cholesky factorisation requires $2(nzl_i)+1$ FLOPS where nzl_i is the number of non-zero elements below the diagonal in row i. Computing each element on column i below the diagonal can also be seen to require at most $2(nzl_i)+1$ FLOPS. Since $\mathbf{A}$ is symmetric we know $l_{i,j}$ is only non-zero if row i of $\mathbf{A}$ has a non-zero entry above the diagonal with column index j. Thus, if $\mathbf{A}$ has nzu_i non-zeros above the diagonal on row i, the computational complexity of level zero Icomplete Cholesky is of the order

$$C(\text{I. Cholesky Level 0}) \leq \sum_{i=0}^{n-1}(nzu_i+1)(2nzl_i+1) \tag{5.28}$$

$$\leq \sum_{i=0}^{n-1} mnz(2mnz-1) = 2n(mnz^2)-mnz \tag{5.29}$$

where mnz is the maximal number of non-zeros on any row of the matrix. When mnz is small relative to n, this is on the order $O(n)$.

An distinct advantage of the level 0 decomposition is that we can store its sparsity pattern using the same arrays as that used for $\mathbf{A}$. For example, with the conventional Compressed Sparse Row (CSR) format, which we outlined in [21], we can use the same matrices `row_idx` and `col_idx` to index $\mathbf{L}$.

5.3.1 Incomplete Cholesky on GPU

The incomplete Cholesky factorisation is the default preconditioner (i.e. with $\mathbf{M}=\mathbf{L}\mathbf{L}^T$) for sparse symmetric positive definite matrices [19]. Note, however, that the algorithm is inherently sequential across the columns of $\mathbf{L}$, and thus presents some problems for parallelisation. The most common approach used in the literature to address this problem is generally that of multicoloring (as for the Gauss-Seidel algorithm) where columns (rows) i and l of $\mathbf{A}$ are in the same group (of same colour) if and only if $(a_{i,l}^{(k)},(a_{l,i}^{(k)})=0$ [16] [9]. Then, columns in the same group may be computed in parallel, in a like manner to the multicoloring approach for Gauss-Seidel algorithm. Finally, to solve the systems $\mathbf{M}\mathbf{z}^{(k)}=\mathbf{L}\mathbf{L}^T\mathbf{z}^{(k)}=\mathbf{r}^{(k)}$, the multicolored Gauss-Seidel algorithm is applied where the upper triangular matrix $\mathbf{U}$ is zero in a forward sweep guaranteeing immediate convergence. Alternatively, in [3], it is proved that forming the incomplete Cholesky factor $\mathbf{L}$ may be formulated as a converging fixed point problem. In particular, denote by $\hat{\mathbf{L}}^{(k)}$ an iterate for the Cholesky factor $\mathbf{L}$. The method in [3] follows algorithm 3, but computes each new iterate $l_{i,j}^{(k+1)}$ in terms of the old iterates $l_{i,j}^{(k)}$.

5.3.2 Incomplete Cholesky Factorisation with mod. SELL

When we want to keep the order of matrix rows constant so as to optimise the SpMV operation, the problem in applying either of the approaches mentioned above appears in line 8. of algorithm 3. We note there that if $a_{i,j} \neq 0$ we must perform a truncated dot product (only considering elements up to index $i-1$) of the rows i and j. Presume we are operating on a slice of the matrix within which lies row i. When j is not in the same slice as row i, any kernel implementation would have to collect the elements and column indices of row j from global memory (under a multicolored ordering). Furthermore, the column indices of row i would have to be checked against those in row j for any matches. If the multicoloring approach was to be used but with a mod. SELL ordering, several rows (ref. columns) in the same slice could not be updated synchronously, which would significantly slow down the algorithm. Conversely, if the fixed-point method was used, greater coalescence could be achieved in the sense that the corresponding slice could first be collected from the old iterate $\mathbf{L}^{(k)}$. However, for columns j not falling within the slice, the old rows $\mathbf{L}^{(k)_j}$ would still have to be collected from global memory.

Furthermore, both the multicoloring and fixed point approaches require that we solve the systems $\mathbf{M}\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$ once the preconditioned conjugate gradient algorithm starts. This cannot be done efficiently under an ordering prioritising minimised matrix bandwidth (i.e. mod. SELL format). It is clear that for efficient GPU implementations, we need instead a preconditioner which can be multiplied directly with $\mathbf{A}$.

5.4 Preconditioning by Matrix Equilibration

One idea for matrix preconditioning which has recently attracted attention is that of matrix equilibration [12] [2]. Symmetric matrix equilibration aims to find a diagonal matrix $\mathbf{D}$ such that the 2-norms (or indeed any other p-norm) of both rows and columns of $\mathbf{D}^{-1}\mathbf{A}(\mathbf{D}^{-1})^T$ is 1. Rather than solving a system at each step of the preconditioned conjugate gradient algorithm, these methods apply the preconditioner $\mathbf{M} = \mathbf{D}\mathbf{D}^T = \mathbf{D}\mathbf{D}$ directly and solve the system

$$\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1}\mathbf{y} = \mathbf{D}^{-1}\mathbf{b} \quad \text{where} \quad \mathbf{D}\mathbf{x} = \mathbf{y}. \tag{5.30}$$

See that if $\mathbf{A}$ is symmetric positive definite, then so is the matrix $\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1}$. To understand why such methods may be effective in reducing the condition number of $\mathbf{A}$, we can consider the Gershgorin Circle Theorem:

Theorem 1 (Gershgorin Circle Theorem) *Let* $\mathbf{A} = (a_{i,j})$ *be an* $n \times n$ *complex matrix. For* $i = 1, 2, \ldots, n$, *define the Gershgorin disc* $D(a_{ii}, R_i)$ *centered at* a_{ii} *with radius*

$$R_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} |a_{ij}|.$$

Then every eigenvalue of $\mathbf{A}$ *lies within at least one of the Gershgorin discs* $D(a_{ii}, R_i)$.

This result is proven in [19]. It follows then that the condition number of a real matrix $\mathbf{A}$ with only positive diagonal elements can be bounded as

$$\kappa(\mathbf{A}) \leq \frac{\max\limits_{1 \leq i \leq n} \left(a_{i,i} + \sum_{j=1, j \neq i}^{n} |a_{i,j}| \right)}{\min\limits_{1 \leq i \leq n} \left(a_{i,i} - \sum_{j=1, j \neq i}^{n} |a_{i,j}| \right)}. \tag{5.31}$$

While not guaranteed, restricting the 2-norm (or another norm) in each row and column of a matrix to 1 will in many circumstances serve to bring the denominator and numerator of this expression "closer". Extensive numerical tests conducted in [2] [12] have shown that when $\mathbf{A}$ is symmetric and $\mathbf{D}$ is constructed such that the row and column 2-norms of $\mathbf{D}^{-1}\mathbf{A}(\mathbf{D}^{-1})$ are 1, the condition number $\kappa(\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1})$ is a drastic reduction compared to $\kappa(\mathbf{A})$.

5.4.1 Constructing the Equilibration Matrix D

The most effective algorithm for symmetric matrix equilibration is attributed to Ruiz [12]. The algorithm for a symmetric matrix $\mathbf{A}$ is seen in algorithm 4. It is easily seen that this procedure maintains the symmetry and sparsity pattern of matrix $\mathbf{D}^{-1}\mathbf{A}(\mathbf{D}^{-1})^T$. It is shown in [12] that the maximal deviance from a p-norm of 1 defined as

$$\text{dev} = \max_{1 \leq i \leq n} \left(|1 - ||(\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1})_i||_p| \right) \tag{5.32}$$

converges to zero with an asymptotic convergence rate of $1/2$.

5.4.2 Matrix Equilibration with mod. SELL

It is easily seen that the Ruiz matrix equilibration algorithm 4 is well suited for GPU implementation with the mod. SELL format. The matrices $\mathbf{D}$ and $\mathbf{R}$ may be stored as n-dimensional arrays `D` and `R` in global memory, respectively. The algorithm can be formulated with two kernels. In the first kernel `calc_sums`, each thread calculates the row norm `r_i` of its corresponding row `i` and sets `D[i] *= r_i` and `R[i] = r_i`. Row elements may again be

Algorithm 4 Ruiz

Input: Symmetric matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, tolerance $\varepsilon > 0$
Output: Matrices $\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1}$ and $\mathbf{D}$ such that the rows and columns of the former have p-norm 1.
$\mathbf{D} \leftarrow \mathbf{I}$
$\mathbf{R} \leftarrow \mathbf{I}$
$\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1} \leftarrow \mathbf{A}$
$\text{dev} = 2\varepsilon$
while $\text{dev} > \varepsilon$ **do**
    $\text{dev} = 0$
    for $i = 0 : (n-1)$ **do**
        $\mathbf{R}_{i,i} \leftarrow \sqrt{||(\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1})_i||_p}$
        if $1 - ||(\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1})_i||_p > \text{dev}$ **then**
            $\text{dev} \leftarrow |1 - ||(\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1})_i||_p|$
        end if
    end for
    $\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1} \leftarrow \sqrt{\mathbf{R}^{-1}}\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1}\sqrt{\mathbf{R}^{-1}}$
    $\mathbf{D} \leftarrow \mathbf{R}\mathbf{D}$
end while

accessed in a coalesced manner as with the SpMV kernel. In the second kernel `scale_kern`, threads scale the elements in their respective row. In particular, each thread first collects `R_i = R[i]` for its corresponding row `i`. Then given the column `col_idx[k]` of an element, the thread collects `R[col_idx[k]]` and sets `entries[k] /= sqrt(R[i]*R[col_idx[k]])`. In a similar manner to SpMV, the kernel places all `R_i` in a shared memory array `R_local` such that if a negative column index `col_idx[k]` is encountered, the corresponding entry of $\mathbf{R}$ can instead be collected directly from `R_local[-(col_idx[k]+1)]`. Rather than checking the maximal deviation dev as defined in (5.32) at each iteration, it can be calculated with a predefined frequency. A small kernel `max_deviation_kern` may be defined to calculate the maximal value of $|1 - \texttt{R[i]}|$ over each section of 1024 entries in `R`. The maximal value from each such section is passed to an array `reduc_arr` of size $\lceil n/1024 \rceil$. A function `reduce` is then used to calculate the maximum over each section. (Note that the `reduce` function and `reduc_arr` array are also used when calculating dot-products with `dot_prod_kern`). Implementations of all these functions/kernels are visible in the software package accompanying the work.

5.5 Jacobi Preconditioner

Before demonstrating test results for different preconditioning algorithms, we should briefly mention the so-called Jacobi preconditioner. In a like manner to an equilibration preconditioner, the Jacobi preconditioner applies a diagonal preconditioner $\mathbf{M} = \mathbf{DD}$ directly to the system of equations as in (5.30). However, the matrix $\mathbf{D}$ is simply taken as the square root of the diagonal of the coefficient matrix $\mathbf{A}$. The effect is that the preconditioned matrix $\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1}$ has all ones on its diagonal. In fact, an interesting result attributed to van der Sluis [23] is that for this matrix $\mathbf{D}$,

$$\kappa(\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1}) \leq mnz\big(\min_{\mathbf{S}\in\mathscr{D}} \kappa(\mathbf{SAS})\big) \tag{5.33}$$

where, as before, *mnz* denotes the maximal number of non-zeros in any row of $\mathbf{A}$ and $\mathscr{D}$ denotes the set of real, diagonal matrices. Like the equilibration preconditioners, the rationale behind such a preconditioner can also be interpreted as an attempt to bring closer the denominator and numerator in expression (5.31).

5.6 Preconditioner Tests

We show here the convergence rates for the preconditioned and un-preconditioned conjugate gradient algorithm when a level 0 incomplete Cholesky preconditioner is used in a CPU implementation and the Jacobi preconditioner or Ruiz equilibration preconditioner (normalising rows and columns to 2-norm of 1) is used in a GPU implementation for the same matrices as discussed in chapter 4. Again, slice height of mod. SELL was 512 and CM ordering used on that format. The preconditioned conjugate conjugate gradient algorithm 1 is used in the CPU implementation whereas the regular conjugate gradient algorithm as in [21] was used for the GPU implementation on preconditioned systems formulated as in (5.30). For each matrix, 10,000 iterations of the un-preconditioned and preconditioned conjugate gradient algorithm were carried out with squared residual norm $\mathbf{r}^T\mathbf{r}$ of the original system $\mathbf{Ax} = \mathbf{b}$ calculated after every 100 iterations. Note that for the un-preconditioned algorithm, the squared residual norm after each iteration is readily available. By contrast, in the preconditioned versions, the squared residual norm of the original system must be calculated explicitly based on the available information. In the case of the "implicit" version of the preconditioned algorithm 1, the original residual vector $\mathbf{r}^{(k)}$ *is* available so that only a dot-product must be computed to get its squared norm. For the explicit version implemented on GPU (i.e. where the original conjugate gradient algorithm is applied to the system in (5.30)) the original residual must be first computed as per $\mathbf{r}^{(k)} = \mathbf{D}\widehat{\mathbf{r}}$, where $\widehat{\mathbf{r}}$ is the residual of the preconditioned system. Getting

the squared residual norm thus requires each element $\mathbf{D}_{i,i}$ to be multiplied by $\mathbf{r}_i^{(k)}$, followed by a dot product. The total time to compute the 100 residuals for each matrix may be seen from the informational table 5.1. We again used `chrono` from CPU-side for all timings. All code for the preconditioners and the (preconditioned) conjugate gradient algorithms on CPU and GPU is available in the software package.

The Ruiz preconditioning algorithm 4 was implemented with a tolerance of 10^{-8} on the maximal deviation as defined in (5.32). Notably, this metric was computed only at every 5th iteration of algorithm 4.

Figure 5.1 shows the convergence rates across the different algorithms against the number of iterations. Notably, the incomplete Cholesky preconditioner converges in the fewest number of iterations across all matrices. Furthermore, it is noted that the diagonal and Ruiz preconditioners significantly increase the rate of convergence as compared to the un-preconditioned algorithm. In fact, within the 10,000 iterations performed, the latter only converges below a tolerance of 10^{-15} in the squared residual norm for the "parabolic_fem" and "thermal2" matrices.

From table 5.1 may also be seen the time to convergence, or to reach 10,000 iterations, for each of the algorithms on their respective platforms. Interestingly, while displaying a superior convergence rate in terms of the number of iterations used, the CPU implementation Cholesky preconditioner performs no better than the un-preconditioned alternative in terms of time to convergence on matrix parabolic_fem.

It may further be seen that the diagonally preconditioned CG on GPU is about two orders of magnitude faster than the incomplete Cholesky preconditioned algorithm on CPU, on matrices "thermal2", "parabolic_fem" and "G3_circuit", despite the faster convergence rate per iteration of the latter. The speedup is about 1 order of magnitude for "100K". Clearly, the improved solution time for the diagonally preconditioned CG on GPU render these a superior alternative to the un-preconditioned algorithm, at least for the matrices tested here. Another interesting observation is that the Jacobi and Ruiz equilibration preconditioners yield very similar results in terms of both the time to convergence and the convergence rate over iterations. While the computational time to construct the Ruiz preconditioner is about an of magnitude larger than that of the Jacobi preconditioner, these times are negligible compared to the time to convergence of the respective preconditioned conjugate gradient algorithms on the linear systems considered. Further research is needed to establish whether one method is preferrable over the other.

We also note that the time to convergence for the preconditioned algorithms is several orders of magnitude larger than the time to compute (and copy to CPU) the original residual.

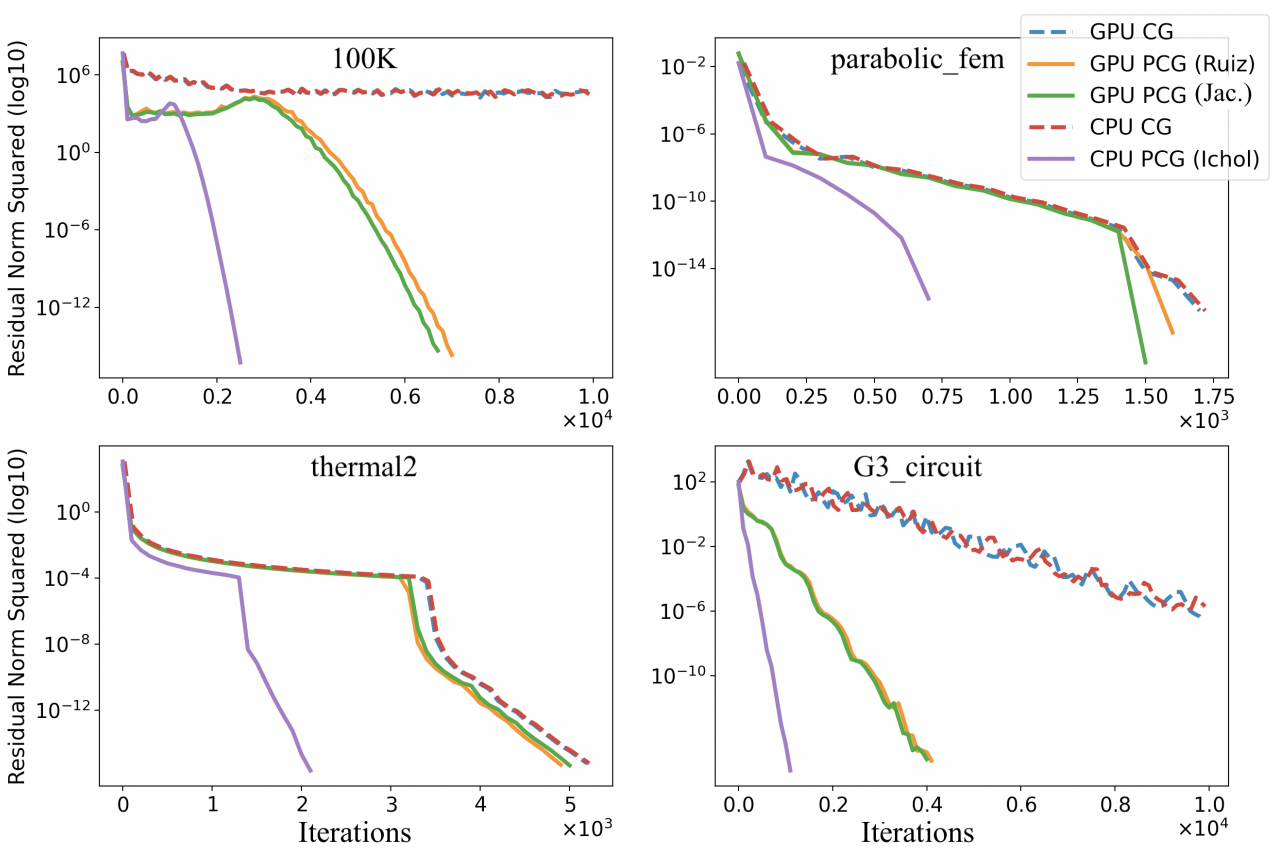


Fig. 5.1 Squared Residual Norms against time (in μs) for the un-preconditioned and preconditioned CG algorithm on four of the matrices from section 3. Each algorithm was run for 10,000 iterations, with residuals being sampled every 100 iterations

5.6.1 Preconditioner Results: Discussion

The incomplete Cholesky preconditioner for conjugate gradient algorithm remains superior, in terms of the convergence per iteration, relative to the alternatives considered here. However, as may be seen from row "Solve $\mathbf{L}\mathbf{L}^T\mathbf{z}^{(k)}$..." of table 5.1, this preconditioner spends almost all its computational time solving trinagular linear systems and in fact converges slower (in time) than normal CG when the latter converges within 10,000 iterations. However, diagonal preconditioners, which do not incur any additional cost per iteration on the conjugate gradient algorithm, are also able to significantly improve convergence rates. When we are interested in maintaining an ordering of matrix rows which facilitates fast sparse matrix-vector multiplications, these may be superior in the context of GPU implementations, as they leave the iteration-wise computational cost of the conjugate gradient algorithm constant. Indeed, as may be seen in table 5.1, these preconditioners, when implemented on GPU, hardly increase the time per iteration against the un-preconditioned conjugate gradient algorithm,

as their only overhead lies in computing residuals of the original system $\mathbf{Ax} = \mathbf{b}$ from the that of the preconditioned system using a single element-wise (Hadamard) product of arrays and a dot -product. However, as seen from 5.1, they can drastically improve the rate of convergence.

Matrix	100K	thermal2	parabolic_fem	G3_circuit
CG Time to Convergence CPU	-	1.3×10^8	9.7×10^6	-
CG Iterations to Convergence CPU	$> 10,000$	4.9×10^3	1.5×10^3	$> 10,000$
CG Average Iteration Time CPU	1.0×10^3	2.35×10^4	6.4×10^3	2.29×10^4
Incomp. Cholesky Factor Time CPU	3.8×10^3	2.2×10^5	2.8×10^4	6.3×10^4
Solve $\mathbf{L}\mathbf{L}^T\mathbf{z}^{(k)} = \mathbf{r}^{(k)}$ (I. Cholesky) Time CPU	4.6×10^6	1.4×10^8	7.0×10^6	3.9×10^7
PCG (I. Cholesky) Residual Time CPU	1.5×10^2	2.2×10^3	7.6×10^2	2.3×10^3
PCG (I. Cholesky) Time to Conv. CPU	5.3×10^6	1.7×10^8	1.1×10^7	4.8×10^7
PCG (I. Cholesky) Iters. to Conv. CPU	2.4×10^3	2.0×10^3	7.0×10^2	1.0×10^3
PCG (I. Cholesky) Avg. Iter. Time CPU	2.3×10^3	7.15×10^4	1.56×10^4	4.74×10^4
CG Residual Time GPU	4.6×10^3	4.0×10^4	1.7×10^4	4.2×10^4
CG Time to Convergence GPU	-	2.0×10^6	2.6×10^5	-
CG Iters to Conv. GPU	$> 10,000$	4.9×10^3	1.5×10^3	$> 10,000$
CG Avg. Iter. Time GPU	5.3×10^1	4.13×10^2	1.72×10^2	4.34×10^2
Precondition System (Jacobi) Time GPU	5.4×10^1	8.5×10^2	2.9×10^2	3.3×10^3
PCG (Jacobi) Residual Time GPU	7.0×10^3	5.2×10^4	2.2×10^4	5.7×10^4
PCG (Jacobi) Time to Convergence GPU	3.6×10^5	2.0×10^6	2.6×10^5	1.6×10^6
PCG (Jacobi) Iters to Conv. GPU	6.6×10^3	4.7×10^3	1.5×10^3	3.7×10^3
PCG (Jacobi) Avg. Iter. Time GPU	5.3×10^1	4.14×10^2	1.73×10^2	4.35×10^2
Precondition System (Ruiz) Time GPU	9.9×10^2	1.6×10^4	3.6×10^3	5.1×10^4
Ruiz Iterations to Conv. in dev GPU	25	25	15	35
PCG (Ruiz) Residual Time GPU	7.0×10^3	5.1×10^4	2.2×10^4	5.7×10^4
PCG (Ruiz) Time to Convergence GPU	3.7×10^5	1.9×10^6	2.8×10^5	1.7×10^6
PCG (Ruiz) Iters to Conv. GPU	6.9×10^3	4.6×10^3	1.6×10^3	3.8×10^3
PCG (Ruiz) Avg. Iter. Time GPU	5.36×10^1	4.14×10^2	1.73×10^5	4.35×10^2

Table 5.1 Total computation time (microseconds) over 10,000 CG iterations when preconditioned and not on GPU and CPU. Residual computation times are sum over 100 residual computations (computation of original residual square norm for preconditioned algorithms). Residual time for regular CG on GPU is total time to copy 100 residual square norms back to CPU. Average iterations time computed from computation time for 10,000 iterations, including residual calculations every 100th. Preconditioned System Time indicates time taken to compute diagonal preconditioner $\mathbf{D}$ and preconditioned system $\mathbf{D}^{-1}\mathbf{A}\mathbf{D}^{-1} = \mathbf{D}^{-1}\mathbf{b}$ on GPU. **Solve** $\mathbf{L}\mathbf{L}^T\mathbf{z}^{(k)}$... is total time spent to covergence by incomplete cholesky preconditioned CG solving triangular linear systems on CPU.

Chapter 6

Conclusion

In this work, we have described the connection between solving sparse linear systems with symmetric positive definite (SPD) coefficient matrices and resolving incompressible flows using the implicit finite volume method. Furthermore, we have formulated a sparse matrix storage format, with an associated sparse-matrix vector multiplication kernel, specifically for FVM discretisation matrices which performs no worse than the "state-of-the-art" Sliced ELLPACK format when tested on a number of large SPD matrices with a structure similar to FVM discretisation matrices. We have tested various smoothing algorithms on four of the aforementioned matrices and shown that a GPU implementation of the block Jacobi smoother may achieve similar convergence to a Gauss-Seidel smoother implemented on GPU, but in about one 100th of the time for sparse matrices with more than 1 million rows. Finally, we have demonstrated similar speedups for the preconditioned and un-preconditioned conjugate gradient (CG) algorithm when the preconditioner is implemented with a level zero incomplete Cholesky factorisation on CPU and diagonal Jacobi and Ruiz equilibration preconditioners on GPU.

While the aforementioned sparse matrix storage format was incorporated in the GPU smoother and preconditioned CG tests, future work should test speeups when a smoother is applied before or interleaved with CG iterations. Furthermore, it would be beneficial to develop kernels for direct FVM discretisation matrix assembly on GPU so as to remove any costs associated with copying a linear system from CPU to GPU.

References

[1] Anzt, H., Tomov, S., Gates, M., Dongarra, J., and Heuveline, V. (2012). Block-asynchronous multigrid smoothers for gpu-accelerated systems. *Procedia Computer Science 9*.

[2] Bradley, A. M. (2010). *Algorithms for the Equilibration of Matrices and Their Application to Limited-Memory Quasi-Newton Methods*. PhD thesis, Stanford University.

[3] Chow, E. and Patel, A. (2015). Fine-grained parallel incomplete lu factorization. *SIAM J. SCI. COMPUT.*

[4] Corporation, N. (2024). Cuda toolkit documentation.

[5] Davis, T. A. and Hu, Y. (2011). The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software 38, 1, Article 1*.

[6] EIMagbrbay, M., Ammar, R., and Rajasekaran, S. (2013). Fast gpu algorithms for implementing the red-black gauss-seidel method for solving partial differential equations. *2013 IEEE Symposium on Computers and Communications (ISCC)*.

[7] Foster, D. (2011). Gpu acceleration of solving parabolic partial differential equations using difference equations. *The International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA' 11)*.

[8] Harris, M. (2014). Optimizing parallel reduction in cuda. NVIDIA Developer Technology Keynote: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[9] Iwashita, T. and Shimasaki, M. (2002). Algebraic multicolor ordering for parallelized iccg solver in finite-element analyses. *IEEE Transactions on Magnetics*.

[10] Jasak, H. (1996). *Error Analysis and Estimation for the Finite Volume Method with Applications to Fluid Flows*. PhD thesis, Imperial College London.

[11] Jasak, H. (2024). Private communication.

[12] Knight, P. A., Ruiz, D., and Uçar, B. (2011). A symmetry preserving algorithm for matrix scaling. Research Report RR-7552, INRIA. https://hal.inria.fr/inria-00569250v2.

[13] Lellmann, J. (2014). Mathematical tripos part ii: Michaelmas term 2014, university of cambridge, numerical analysis – lecture 17. Lecture notes, University of Cambridge. URL: https://www.lellmann.net/work/$_media/teaching/na_c17.pdf, Accessedon16Aug.2024.

[14] Li, R., Xi, Y., and Saad, Y. (2016). Schur complement based domain decomposition preconditioners with low-rank corrections. *Numerical Linear Algebra with Applications 23, 4*.

[15] Monakov, A., Lokhmotov, A., and Avetisyan, A. (2010). Automatically tuning sparse matrix-vector multiplication for gpu architectures. In Patt, Y., Foglia, P., Duesterwald, E., Faraboschi, P., and Martorell, X., editors, *High Performance Embedded Architectures and Compilers*, volume 5952 of *Lecture Notes in Computer Science*, pages 111–125. Springer Berlin Heidelberg.

[16] Naumov, M. (2012). Parallel incomplete-lu and cholesky factorization in the preconditioned iterative methods on the gpu. *NVIDIA Teechnical Report*.

[17] Nvidia (2024). Cuda c++ programming guide. Technical report, Nvidia.

[18] Patankar, S. V. and Spalding, D. B. (1972). A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows. *International Journal of Heat and Mass Transfer*, 15(10):1787–1806.

[19] Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics.

[20] Shivakumar, P. N. and Chew, K. H. (1974). A sufficient condition for nonvanishing of determinants. *Proceedings of the American Mathematical Society*.

[21] Sigstad Wikberg, S. (2024). Unstructured mesh fvm with gpu support: Report 1. Master's thesis, University of Cambridge.

[22] Thomas, S., Yamazaki, I., Berger-Vergiat, L., Kelley, B., Hu, J., Mullowney, P., Rajamanickam, S., and Swiryowicz, K. (2021). Two-stage gauss–seidel preconditioner and smoothers for krylov solvers on a gpu cluster. *arXiv.org*.

[23] van der Sluis, A. (1969). Condition numbers and equilibration of matrices. *Numerische Mathematik*, 14(1):14–23.

[24] Wlotzka, M. and Heuveline, V. (2012). Block-asynchronous and jacobi smoothers for a multigrid solver on gpu-accelerated hpc clusters. *Preprint Series of the Engineering Mathematics and Computing Lab*.