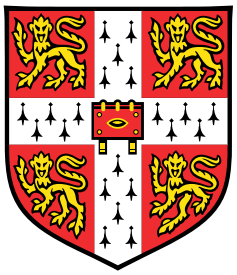
Unstructured Mesh FVM with GPU Support

Report 1



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 January 2025

I would like to dedicate this thesis to my loving parents ...

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
January 2025

Acknowledgements

I would like to acknowledge the help of my supervisor Dr Hrvoje Jasak.

Abstract

The sparse-matrix-dense-vector operation (SpMV) is an important procedure in many scientific codes. We explore here how the SpMV operation can be sped up using an Nvidia A30 GPU with various algorithms for symmetric matrices resulting from finite volume method discretisations on unstructured meshes. Furthermore, we outline the conjugate gradient method for sparse linear systems, and show why speeding up the SpMV operation is a crucial part of this algorithm. Further, we find that the GPU cannot exploit the symmetric pattern of a matrix to speed up its multiplication times with a dense vector. Lastly, we propose a modification to the Sliced ELLPACK format for the SpMV operations using concurrent processing on CUDA streams. The algorithm performs about as well as the ordinary Sliced ELLPACK format on four large matrices from the SuiteSparse Matrix Collection.

Table of contents

Chapter 1

Introduction

The implicit finite volume method (FVM) in fluid dynamics tends to produce extremely sparse linear systems of equations that must be solved at each time-step to update the solution. Due to the interdependence of each pair of adjacent cells, the sparsity pattern of the associated discretisation matrix is always symmetric. Moreover, when the relevant equations are elliptic, the matrix coefficients tend also to be symmetric.

A popular set of solution algorithms for such systems are iterative Krylov solvers. As part of these methods, the most time-consuming procedure within any iteration is typically a sparse-matrix-dense-vector calculation (SpMV).

With the advent of widely available general purpose graphics processing units (GPGPUs), researchers have been able to parallelise and drastically speed up iterative Krylov solvers for sparse linear systems, and in particular the aforementioned matrix-vector products. However, while symmetric matrices have fewer unique entries, and should therefore make for faster vector multiplications, GPUs have not been able to provide this advantage over general SpMV.

In this work we explore the existing space of SpMV algorithms for GPU and in particular their usefulness for the conjugate gradient (CG) method for symmetric matrices that arise for FVM discretisations. Furthermore, we propose a modification to the existing "state of the art" to improve the speed of FVM-related SpMV operations, while also reducing the memory requirement.

Chapter 2 presents an overview of the hardware and programming paradigm on modern NVIDIA GPUs, the most widely used large-scale GPUs for scientific computations. Thereafter, chapter 3 briefly presents the the conjugate gradient algorithm. Chapter 4 delves into the technicalities of SpMV on GPU and existing sparse matrix linear algebra libraries for NVIDIA GPUs. The algorithm proposed here, Staircase SELL, is also presented in chapter 4. Chapter 5 presents comparisons of the various SpMV algorithms discussed in terms of

their computation times on an NVIDIA A30 GPU for various symmetric matrices from the SuiteSparse openly available collection of sparse matrices [3]. Finally, chapter 6 provides a discussion of the results and outlooks for future work.

Chapter 2

GPU Architecture

2.1 The Nvidia GPU

At the physical level, the fundamental component of a GPU is the *core*. Much like the CPU core, the GPU core has integrated arithmetic and floating point logic units with which it operates on data stored in registers. However, unlike the cores of a CPU, GPU cores do not have a separate control unit (CU) decoding instructions passed to the core. Rather, Nvidia GPUs contain hundreds, or even thousands, of cores, organised into several streaming multiprocessors (SMs). A control unit is situated at the level of each SM and designed to pass equivalent instructions to sets of cores that execute on their respective registers synchronously [9]. This single instruction multiple thread (SIMT) process is what gives GPUs their massive advantage over CPUs when equivalent instructions must be applied uniformly over the entries in a data structure.

Each streaming multiprocessor has a *register file*, containing registers available to it, and a set of caches organised in a hierarchy for fast access to repeatedly used data. Data is loaded to the cache of an SM from the GPU's *global memory*, which may be compared with the RAM to which a CPU has access. Instructions for the CU and data on which an SM operate must pass through the cache hierarchy before being accessible by a core [9]. An exception is the case when data is already present in cache, in which case it can be passed directly.

At the software level, Nvidia's CUDA API is designed to execute the same instructions on a set of *threads*. Threads are organised into *thread-blocks*, which in turn are organised in a *grid*. Indexing of the grid and the blocks may be done in up to 3 dimensions, as specified in a program. A *kernel* is a function defined in CUDA for synchronous execution by the threads in a grid. The kernel can take a set of arguments which are equal for all threads and is launched by specifying said arguments along with the dimensions for the thread-blocks in a grid. Due to the single set of instructions, any branching between different threads is

predicated on their index within a block and the block's index within the grid, both available from within a kernel.

When a kernel is launched, thread-blocks are allocated to SMs by the CUDA scheduler until full capacity is reached, after which any further thread-blocks must wait for completion of previous ones. Each SM is typically able to accommodate only a few thread-blocks. Furthermore, each thread in the grid is allocated a given number of registers depending on the register usage of the given kernel. If the threads allocated to an SM require more local variables than may be stored on the SMs register file, the excess must be stored in global memory, in a section private to each thread [9]. For the SM, access to global memory is significantly more time-consuming than access to cache or its register file. Consequently, the local memory requirements of a kernel, in relation to the size of the register file and cache for each SM, should be closely monitored to avoid *memory spills* and optimise performance.

In the same clock-cycle, the SM is able to execute operations simultaneously for all threads in a *warp*, allocating one core to each. On current architectures, the number of threads in a warp is 32. When a set of instructions have been carried out for a warp, the SM's control unit reallocates cores for a new warp on which instructions are carried out synchronously. By efficiently reallocating cores, the SM is able to avoid latency should a warp encounter divergent branching on the threads within it [9].

2.1.1 Memory

Threads in a grid cannot directly access data from other threads. For threads in different blocks, any communication must be made through global memory. Given that the programmer cannot control the order of execution for different blocks, such communication is not generally viable. However, threads in the same block may access data in *shared memory*. The shared memory segment of each SM is divided between the associated blocks and provides fast access for the SMs cores, relative to global memory. A thread must write data to shared data before it may be read by a thread in the same block. Thus various synchronization routines are provided in CUDA, allowing threads to be synchronised either at the warp of block-levels.

The cache of Nvidia GPUs is organised into an L1 cache, private to each SM, a larger L2 cache, shared between SMs, and, on later cards, an L3 cache which is larger yet and also shared [5]. The larger addressing space of caches L2 and L3 make them slightly more time-consuming to search and access. On later cards, shared memory is aligned with the L1 cache of each SM, providing the possibility of allocating the relative size of each at compile-time.

Modern Nvidia GPUs are organised so as to provide simultaneous reads and writes to contiguous sections of global memory. It is thus beneficial for threads in a warp to read and write memory from contiguous sections of memory, if possible. The same principle applies for shared memory, whereby the 32 threads in a warp may read of write to shared memory simultaneously.

2.1.2 Streams

Much like parallel threads may operate on a conventional computer, CUDA *streams* may execute different kernels synchronously (where possible). Streams are effective where a programmer wants to launch several different kernels together, or a single kernel with different arguments, but where each kernel may be to small to occupy all SMs. Again, a scheduler is responsible for allocating resources between streams where their accumulated requests exceed what may be done simultaneously on the device [9]. CUDA streams may be organised in arrays and can remain active for the duration of a program.

Chapter 3

Conjugate Gradient Method

The conjugate gradient method is an iterative Krylov solver for linear systems of equations, guaranteed to converge for symmetric, positive definite coefficient-matrices. The method, and other Krylov subspace solvers, is founded on the concept of *conjugate vectors*. For two vectors $\mathbf{a}, \mathbf{c} \in \mathbb{R}^n$, they are defined conjugate with respect to some matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ if and only if the following relation holds

$$\mathbf{a}^T \mathbf{A} \mathbf{c} = 0. \tag{3.1}$$

For a $\mathbf{A}$ symmetric, this relation may easily be seen to be symmetric.

We prove now that a set $\Gamma = \{\mathbf{d}_0, \ldots, \mathbf{d}_{n-1}\}$ of n non-zero vectors in $\mathbb{R}^n$, mutually conjugate with respect to some symmetric, positive definite matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$, forms a basis for $\mathbb{R}^n$. Let $\mathbf{V} \in \mathbb{R}^{n \times n}$ be the matrix whose ith column is $\mathbf{d}_i$. Define now a matrix $\mathbf{G} = \mathbf{V}^T \mathbf{A} \mathbf{V}$. Since each pair of columns (i, j), with $i \neq j$ of $\mathbf{V}$ are conjugate with respect to $\mathbf{A}$, we see easily that all off-diagonal elements of G are zero. Furthermore, since $\mathbf{A}$ positive definite, the diagonal elements of $\mathbf{G}$ are positive, and thus $\mathbf{G}$ is invertible.

For a contradiction, assume now that the columns of $\mathbf{V}$ are linearly dependent; that is, there is some non-zero vector $\mathbf{c}$ for which $\mathbf{V}c = 0$. It follows then that $\mathbf{G}\mathbf{c} = \mathbf{V}^T \mathbf{A} \mathbf{V} \mathbf{c} = 0$, contradicting the fact that $\mathbf{G}$ is invertible. It follows thus that Γ forms a linearly independent set, and is thus a basis for $\mathbb{R}^n$.

Using this fact, we may infer that for the solution $\mathbf{x}$ to the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$, there is a set of scalars $\{\alpha_0, \ldots, \alpha_{n-1}\}$ such that

$$\mathbf{x} = \sum_{i=1}^{n} \alpha_i \mathbf{d}_i. \tag{3.2}$$

Left-multiplying equation (3.2) by $\mathbf{d}_j^T\mathbf{A}$, where $\mathbf{d}_j \in \Gamma$, we see by the mutual conjugacy of the vectors in Γ that

$$\mathbf{d}_j^T\mathbf{A}\mathbf{x} = \mathbf{d}_j^T\mathbf{b} = \alpha_j\mathbf{d}_j^T\mathbf{A}\mathbf{d}_j^T \Rightarrow \alpha_j = \frac{\mathbf{d}_j^T\mathbf{b}}{\mathbf{d}_j^T\mathbf{A}\mathbf{d}_j}. \tag{3.3}$$

With this result in hand, we recognise that solving the system $\mathbf{A}\mathbf{x} = \mathbf{b}$ reduces to finding a set Γ of n mutually conjugate vectors (with respect to $\mathbf{A}$).

We take as an initial guess $\mathbf{x}_0 = 0$. (If there exists some other guess $\hat{\mathbf{x}}_0$, then the adjacent system $\mathbf{A}\mathbf{y} = \mathbf{b} - \mathbf{A}\hat{\mathbf{x}}_0$ is considered instead, with $\hat{\mathbf{x}}_0 + \mathbf{y}$ as the final solution once $\mathbf{y}$ is found). The initial residual and *search direction*, $\mathbf{d}_0$, are set as $\mathbf{r}_0 = \mathbf{d}_0 = \mathbf{b}$. In accordance with equation (3.3), we therefore update the solution as $\mathbf{x}_1 = \alpha_1\mathbf{b}$. The residual is then updated as $\mathbf{r}_1 = \mathbf{r}_0 - \alpha_1\mathbf{A}\mathbf{b}$. When finding a new component of Γ (i.e. a new search direction), we are interested in identifying the direction, $\mathbf{d}_1$, along which the residual decreases fastest. In order to do so, we consider first the function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x}. \tag{3.4}$$

The gradient with respect to $\mathbf{x}$ of this equation may be found to be

$$\nabla_x f(\mathbf{x}) = \frac{1}{2}(\mathbf{A}^T + \mathbf{A})\mathbf{x} - \mathbf{b} = \mathbf{A}\mathbf{x} - \mathbf{b}. \tag{3.5}$$

We see therefore that the function f has a minimum at $\mathbf{x}$, the solution we are seeking. At the point $\mathbf{x}_0$, the gradient is $\alpha_1\mathbf{A}\mathbf{d}_1 - \mathbf{b} = -\mathbf{r}_1$. More generally, the gradient of f is the residual of the current guess. We could reason that we should make a step in the direction of the negative new gradient, $\mathbf{r}_1$. Bearing in mind, however, that the direction in which we step should be conjugate to any previous direction, we subtract from $\mathbf{r}_1$ the component which is not conjugate to $\mathbf{d}_0$:

$$\mathbf{d}_1 = \mathbf{r}_1 - \frac{\mathbf{d}_0^T\mathbf{A}\mathbf{r}_1}{\mathbf{d}_0^T\mathbf{A}\mathbf{d}_0}\mathbf{d}_0 \tag{3.6}$$

This process is repeated until convergence (i.e. until the norm of $\mathbf{r}_i$ is sufficiently small), where at each step, the new search direction is calculated by subtracting from the new residual the components that are nor conjugate to all previous search directions:

$$\mathbf{d}_{i+1} = \mathbf{r}_{i+1} - \sum_{k=0}^{i}\frac{\mathbf{d}_k^T\mathbf{A}\mathbf{r}_{i+1}}{\mathbf{d}_k^T\mathbf{A}\mathbf{d}_k}\mathbf{d}_k. \tag{3.7}$$

In their current form, the steps outlined render the algorithm very memory inefficient for any computational implementation. As the algorithm proceeds through the iterations, one must keep track of all previous search directions $\mathbf{d}_i$.

3.0.1 Reducing the Memory Footprint

To address this issue, we must first prove that residuals are orthogonal to all previous search directions (i.e. that for $i > j$, $\mathbf{r}_i^T \mathbf{d}_j = 0$). Second, we must prove that residuals are mutually orthogonal (i.e. for $i \neq j$, $\mathbf{r}_i^T \mathbf{r}_j = 0$).

Orthogonality of Residuals with Previous Search Directions

Let $i > j$. Note first simply that

$$\mathbf{r}_i = \mathbf{b} - \mathbf{A} \sum_{k=0}^{i-1} \alpha_k \mathbf{d}_k. \tag{3.8}$$

By the conjugacy of the search directions, we see then that

$$\mathbf{d}_j^T \mathbf{r}_i = \mathbf{d}_j^T \mathbf{b} - \alpha_j \mathbf{d}_j^T \mathbf{A} \mathbf{d}_j = \mathbf{d}_j^T \mathbf{b} - \mathbf{d}_j^T \mathbf{A} \mathbf{d}_j \frac{\mathbf{d}_j^T \mathbf{b}}{\mathbf{d}_j^T \mathbf{A} \mathbf{d}_j} = 0. \tag{3.9}$$

This proves that residuals are orthogonal to previous search directions.

Mutual Orthogonality of Residuals

Let again $i > j$. We may rewrite equation (3.7)

$$\mathbf{r}_j = \mathbf{d}_j + \sum_{k=0}^{j-1} \frac{\mathbf{d}_k^T \mathbf{A} \mathbf{r}_j}{\mathbf{d}_k^T \mathbf{A} \mathbf{d}_k} \mathbf{d}_k, \tag{3.10}$$

where the sum collapses to zero for the case where $j = 0$ (i.e. $\mathbf{r}_0 = \mathbf{d}_0 = \mathbf{b}$). Taking the dot-product with $\mathbf{d}_i$, and recalling the previous result, we get

$$\mathbf{r}_i^T \mathbf{r}_j = \mathbf{r}_i^T \mathbf{d}_j + \sum_{k=0}^{j-1} \frac{\mathbf{d}_k^T \mathbf{A} \mathbf{r}_j}{\mathbf{d}_k^T \mathbf{A} \mathbf{d}_k} \mathbf{r}_i^T \mathbf{d}_k = 0. \tag{3.11}$$

Hence, we see that the residuals in the algorithm are mutually orthogonal.

Simplifying the α-term

We are now in a position to simplify both the terms α and β. We aim to prove that α may be rewritten

$$\alpha_i = \frac{\mathbf{r}_i^T \mathbf{r}_i}{\mathbf{d}_i^T \mathbf{A}\mathbf{d}_i}. \tag{3.12}$$

The case of $i = 0$ is trivial due to the definitions of the initial residual and search direction. For $i > 0$, see first that in accordance with equation (3.8), $\mathbf{b}$ may be re-written as

$$\mathbf{b} = \mathbf{r}_i + \sum_{k=0}^{i-1} \alpha_k \mathbf{A}\mathbf{d}_k. \tag{3.13}$$

Now substituting this expression and equation (3.7), indexed for i, into the numerator of equation (3.3) gives

$$\mathbf{d}_i^T \mathbf{b} = \left(\mathbf{r}_i - \sum_{k=0}^{i-1} \frac{\mathbf{d}_k^T \mathbf{A}\mathbf{r}_i}{\mathbf{d}_k^T \mathbf{A}\mathbf{d}_k} \mathbf{d}_k\right)^T \left(\mathbf{r}_i + \sum_{k=0}^{i-1} \alpha_k \mathbf{A}\mathbf{d}_k\right). \tag{3.14}$$

Substituting in for α_k using equation (3.3) and expanding the product gives

$$\mathbf{d}_i^T \mathbf{b} = \mathbf{r}_i^T \mathbf{r}_i + \sum_{k=0}^{i-1} \frac{\mathbf{d}_k^T \mathbf{b}}{\mathbf{d}_k^T \mathbf{A}\mathbf{d}_k} \mathbf{r}_i^T \mathbf{A}\mathbf{d}_k - \sum_{k=0}^{i-1} \frac{\mathbf{d}_k^T \mathbf{A}\mathbf{r}_i}{\mathbf{d}_k^T \mathbf{A}\mathbf{d}_k} \mathbf{r}_i^T \mathbf{d}_k - \sum_{k=0}^{i-1} \frac{\mathbf{d}_k^T \mathbf{b}(\mathbf{d}_k^T \mathbf{A}\mathbf{r}_i)}{(\mathbf{d}_k^T \mathbf{A}\mathbf{d}_k)^2} \mathbf{d}_k^T \mathbf{A}\mathbf{d}_k. \tag{3.15}$$

Note that the first and third terms in the expression above cancel out. Furthermore, the second sum cancels out due to the fact that residuals are orthogonal to previous search directions. Equation (3.12) then follows.

Simplifying the Expression for New Search Direction

We are finally in a position to simplify the expression for a new search direction, namely equation (3.7). We wish to prove that

$$\mathbf{d}_{i+1} = \mathbf{r}_{i+1} + \beta \mathbf{d}_i \quad \text{where} \quad \beta = \frac{\mathbf{r}_{i+1}^T \mathbf{r}_{i+1}}{\mathbf{r}_i^T \mathbf{r}_i}. \tag{3.16}$$

See first that from equation (3.8), we may write

$$\mathbf{r}_{i+1} = \mathbf{r}_i - \alpha_i \mathbf{A}\mathbf{d}_i. \tag{3.17}$$

Now, re-writing this equation for $\mathbf{d}_i$ and recalling that the inverse of a symmetric matrix is also symmetric gives

$$\mathbf{d}_i = \frac{1}{\alpha_i}\mathbf{A}^{-1}(\mathbf{r}_i - \mathbf{r}_{i+1}) \quad \Rightarrow \quad \mathbf{d}_i^T = \frac{1}{\alpha_i}(\mathbf{r}_i^T - \mathbf{r}_{i+1}^T)\mathbf{A}^{-1}. \tag{3.18}$$

Considering now equation (3.7) and the new expression for α_i (3.12), for $k = i$, we have

$$\mathbf{d}_i^T\mathbf{A}\mathbf{r}_{i+1} = \frac{\mathbf{d}_i^T\mathbf{A}\mathbf{d}_i}{\mathbf{r}_i^T\mathbf{r}_i}(\mathbf{r}_i^T - \mathbf{r}_{i+1}^T)\mathbf{A}^{-1}\mathbf{A}\mathbf{r}_{i+1} = -\frac{\mathbf{r}_i\mathbf{A}\mathbf{d}_i}{\mathbf{r}_i^T\mathbf{r}_i}\mathbf{r}_{i+1}^T\mathbf{r}_{i+1} \tag{3.19}$$

by the fact that residuals are mutually orthogonal. For $i = 0$, we can substitute this expression in equation (3.7) and we are done. For the case of $i > 0$, consider some $j < i$. In an equivalent manner to the case for i, we get

$$\mathbf{d}_j^T = \frac{1}{\alpha_j}(\mathbf{r}_j^T - \mathbf{r}_{j+1}^T)\mathbf{A}^{-1} \quad \Rightarrow \quad \mathbf{d}_j^T\mathbf{A}\mathbf{r}_{i+1} = 0 \tag{3.20}$$

again by the mutual orthogonality of residuals. Now, substituting the cases $k = i$ and $k = j < i$, into equation (3.7) gives the desired result.

3.0.2 Memory-aware CG algorithm

With these results in hand, the memory requirement of the CG algorithm in any computational implementation has been drastically reduced. Instead of storing all previous search directions in iteration $i < n$, in addition to the matrix $\mathbf{A}$, $\mathbf{r}_{i+1}$, $\mathbf{x}_{i+1}$ and $\mathbf{A}$, we need only store $\mathbf{r}_i$, $\mathbf{r}_{i+1}$, $\mathbf{x}_i$ and $\mathbf{A}$, in addition to two scalars. Since $\mathbf{A}$ is sparse, for the purposes of FVM computations, it usually has a relatively small memory footprint. In addition, we note that we need only compute one matrix-vector product, namely $\mathbf{A}\mathbf{d}_i$, in addition to the fact that the dot-product $\mathbf{r}_{i+1}^T\mathbf{r}_{i+1}$ may be preserved for the next iteration. The final algorithm is thus presented below.

Algorithm 1 Conjugate Gradient Algorithm

Input: Symmetric positive definite matrix $\mathbf{A}$, vector $\mathbf{b}$, initial guess $\mathbf{x}_0$, tolerance ε
Output: Approximate solution $\mathbf{x}$ to $\mathbf{Ax} = \mathbf{b}$
if $\mathbf{x}_0 \neq 0$ **then**
    $\mathbf{b} \leftarrow \mathbf{b} - \mathbf{Ax}_0$
    $\hat{\mathbf{x}}_0 \leftarrow \mathbf{x}_0$
    $\mathbf{x}_0 \leftarrow 0$
end if
$\mathbf{r}_0 \leftarrow \mathbf{b}$
$\mathbf{d}_0 \leftarrow \mathbf{r}_0$
$k \leftarrow 0$
while $\|\mathbf{r}_k\| > \varepsilon$ **do**
    $\alpha_k \leftarrow \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{d}_k^T \mathbf{A} \mathbf{d}_k}$
    $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k$
    $\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{d}_k$
    if $\|\mathbf{r}_{k+1}\| \leq \varepsilon$ **then**
        break
    end if
    $\beta_k \leftarrow \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}$
    $\mathbf{d}_{k+1} \leftarrow \mathbf{r}_{k+1} + \beta_k \mathbf{d}_k$
    $k \leftarrow k + 1$
end while
if Input $\mathbf{x}_0 = 0$ **then**
    return $\mathbf{x}_{k+1}$
else
    return $\mathbf{x}_{k+1} + \hat{\mathbf{x}}_0$
end if

Chapter 4

Sparse Matrix Linear Algebra on GPU

The emergence over the past 30 years of general-purpose graphics processing units has brought the possibility to significantly speed up linear algebra operations relative to single and even multi-core CPU implementations. Several proprietary and open-source libraries for sparse linear algebra with GPU support have been added to the market, and we present here some of the most significant ones.

4.1 Existing Libraries for Nvidia GPUs

4.1.1 cuSparse

Nvidia's cuSparse was released as part of its CUDA toolkit in 2010 so as to provide a unified API for sparse matrix operations on its devices. While proprietary, the library supports vector-vector, matrix-vector, and matrix-matrix operations of the sparse-sparse or sparse-dense type. The cuSparse library lies closer to the low-level than many other sparse linear algebra libraries in that the programmer is responsible for providing matrix data in formats compatible with the cuSparse functionalities. On the other hand, this gives programmers the opportunity to tailor codes to their particular needs in relation to matrix structure and size.

For example, cuSparse comes with the possibility of precomputing the buffer needs of a particular computation in addition to preprocessing of a matrix, in order that the performance on repeated operations may be optimised [10]. In recent updates, cuSparse has moved toward a limited number of supported sparse matrix storage format including compressed sparse row format (CSR), compresed sparse column format (CSC), coordinate format (COO), and Ellpack format, all of which will be elaborated upon in section 4.2.

4.1.2 CUSP

The open-source library CUSP is built for sparse matrix operations on GPU with greater ease of use than cuSparse. Implemented by Nvidia reseachers around 2010, CUSP is built on top of Nvidia's thrust library, which provides device optimised algorithms for parallel data structures and algorithms such as sort, transform, and reduce on Nvidia GPUs. While still in wide use, CUSP has not provided a stable updated release since 2015 [2]. CUSP supports matrix-vector multiplication for matrices in CSR, ELLPACK, COO, and diagonal (DIA) format.

4.1.3 AMGx

To the author's best knowledge, the only existing public library for multigrid iterative solvers on Nvidia GPUs is AMGx, developed by Nvidia around 2014 [11]. The AMGx library supports a wide variety of smoothers, preconditioners, and iterative solvers for the algebraic multigrid method. AMGx supports a wide array of storage formats and associated algorithms for matrix-vector multiplication that may be executed either through cuSparse or CUSP plug-ins.

4.2 Sparse Matrix Storage and SpMV

4.2.1 Special Considerations for Unstructured FVM

Under conjugate gradient (CG) and related methods for sparse linear systems of equations, the most compute heavy task in any iteration is the computation of the sparse matrix-vector product $\mathbf{Ad}$ needed to determine a new step size and an updated residual. On a system with n equations and nnz non-zero coefficients, computing $\mathbf{Ad}$ necessitates nnz multiplications and $nnz - n$ additions for a total of $2nnz - n$ floating point operations (FLOPs). By contrast, a vector dot-product requires only $2n - 1$ FLOPs whereas a vector addition requires n FLOPs.

In general, and in the 3-dimensional case, an unstructured finite volume mesh requires convex cells of at least 4 sides each. By consequence, each internal cell in a domain will at least generate five non-zero coefficients in the resulting discretisation matrix, where the fifth coefficient corresponds to a cell's contribution to its own state. Accordingly, one may make the approximation $nnz \gtrapprox 5n$, since the cells in an unstructured mesh may have significantly more than four faces.

We are now in a position to formally quantify the minimum computational expense of the matrix-vector multiplication part of a CG-like algorithm relative to the other operations for a

3-dimensional mesh. In the classical CG-method, one iteration requires one matrix-vector product computation, three vector dot-products, three scalar-vector multiplications, three vector-vector additions, and two scalar divisions. Using the above, this makes for a lower bound on the computational complexity of

$$C(\mathrm{CG}) \gtrapprox (10n-n)+3(2n-1)+3n+3n+2 \approx 21n$$

A very optimistic lower bound for the relative time, only for the FLOPS, taken by the matrix-vector product during one iteration of CG is thus $3/7$. However, as mentioned, the cells in a mesh can have significantly more than 6 faces. For example, on a Cartesian mesh in three dimensions, each internal cell will have six faces, resulting in $nnz \approx 7n$. The number of FLOPs needed for the matrix-vector multiplication can then be approximated to $13n$, in which case its computational complexity makes up over half that of a CG iteration.

In addition to the large number of FLOPs, significant complexity is added on GPU by the fact that the columns of non-zero coefficients on a given row of a sparse linear matrix do not tend to be contiguous, complicating memory access patterns when gathering the entries of **d** during a row-vector multiplication of SpMV on GPU. By contrast, the residual, **r**, and search direction, **d**, vectors are assumed dense, hence disposing of any memory coalescence issues during vector-vector additions and dot-products. It is therefore in the interest of reducing the computational time of a solver to optimise matrix-vector multiplications.

4.2.2 Parallel Symmetric SpMV

Several implementation exist for effective sparse-matrix-dense-vector products on parallel CPU cores. In essence, mirroring entries of a matrix, and their row and/or column index may be stored together in a *struct*, multiplied by the corresponding vector entries and distributed to a result-array where they are added with products from other threads. Many algorithms attempt to block the rows of unconnected rows together in groups, such that a set of threads may process one row in a group each without conflicts during memory accesses [4].

Very little work has been published on symmetric SpMV on GPUs, and the author of this text failed to find any literature reporting consistent SpMV speedups of symmetric storage formats relative to formats where all matrix entries are stored individually. In fact, NVIDIA does not currently provide any SpMV routines specifically for matrices stored in a symmetric format in cuSparse, citing that doing so increases the time requirement of SpMV by more than five-fold [10]. The author is aware that NVIDIA has proposed the Packet (PKT) format for sparse matrices, which was previously part of cuSparse, but was only effective for matrices with a very limited bandwidth (i.e. the largest difference between the row and

column indices of any non-zero entry) [1]. Algorithms exist that reorder matrix entries to achieve better bandwidths. However, an optimal ordering cannot be achieved in polynomial time, and the bandwidth reduction depends heavily on the original matrix structure.

The underlying reason for the aformentioned challenges to symmetric SpMV on GPU lies with the way that GPUs access memory. In order to achieve fast processing times, the GPU relies on being able to operate on related data-points mainly *within* thread-blocks and on accessing global memory in a coalesced manner. For example, adding the products from a matrix-row-vector multiplication benefits significantly from being performed within each thread-block, rather than separate thread-blocks having to sequentially schedule their additions to the same location in global memory. Fast sums over the rows of a matrix therefore require all non-zero entries of a row to be located "close" in global memory, when coalescing is considered. Say for example that only the upper triangle of a symmetric matrix is stored in memory. Threads in a block may go along a row each of the matrix, reading entries and associated columns in a coalesced fashion (where the length of each row is similar). Threads may then multiply corresponding lower-triangle entries with vector-coefficients corresponding to the columns of the diagonals they process. In eaither case, the threads must then add each of these lower-triangle products to different locations in global memory, which may not be done in a coalesced manner, given the unpredictable pattern of column indices. Symmetric SpMV operations are therefore largely a choice between speed and memory footprint.

In order to avoid reading both the column and row index of each entry, the best algorithms for (general) SpMV on NVIDIA GPUs store all non-zero entries of a row contiguously, adding the associated products within-block, and writing the sums of contiguous rows in a coalesced fashion back to a result-array. For these reasons we have chosen also in this report, to focus on storage formats and multiplication algorithms that treat the matrix in an agnostic manner, opting not to make use of potential global memory savings for symmetric matrices.

4.2.3 Storage Formats

In computing (general) sparse matrix-vector products on GPU, the algorithm used is to a large extent dependent on the storage format used for the sparse matrix. Much effort has been put into formulating sparse matrix formats that will allow for fast matrix-vector computation on GPU, independent of the sparsity pattern in the matrix. In this work however, we are interested only the matrix formats that will generate fast matrix-vector computation for the coefficient matrices generated by the implicit finite volume method.

The sparsity pattern of these matrices are characterised entirely by the mesh from which they are generated. As mentioned previously, Cartesian meshes in 3 dimensions are charac-

terised by a maximum of 7 non-zero entries per row. While this number may be somewhat higher for an unstructured mesh, it rarely exceeds 21 [6], which would correspond to a polyhedral cell with 20 faces. As will be seen, knowledge of the maximal number of non-zeros in a row can be exploited to improve the speed with which the GPU can read and thus multiply rows with corresponding elements in the vector. We discuss below the best known sparse matrix storage formats along with the ones most suited for implicit FVM discretisation matrices. When presenting the storage requirement for the different formats, we let $|\texttt{f}|$ and $|\texttt{int}|$ denote the size, in bytes, of storing floating point numbers and integers, respectively, on the given architecture.

In place of algorithms, C++ CUDA kernels are provided to demonstrate the associated algorithms for several of the discussed storage formats. Pointers denote locations in global memory,

Coordinate Format

The simplest storage format for a sparse matrix is perhaps the coordinate storage format (COO). As indicated by its name, in COO format, two integer arrays `row_idx` and `col_idx` of length nnz store the row and column index, respectively, of each non-zero element in the matrix. An array of floating point numbers of equal length, `val`, stores the corresponding non-zero values. It follows thus that the memory requirement of COO format may be expressed

$$S(COO) = (2 \times |\texttt{int}| \; + \; |\texttt{f}|) \times nnz \tag{4.1}$$

The advantage of such a storage format is that the elements may be stored in any order. A simple matrix-vector multiplication kernel, using the COO format, in CUDA may then be formulated as below. (Some small adjustments are necessary if `blockDim.x` is not a divisor of n to avoid out of bounds access.) Note in particular that each thread is responsible for multiplying one non-zero element from the matrix with an element from $\mathbf{x}$.

```
__global__ void coord_mult(double* val, int* row_idx, int* col_idx,
    double* x, double* y, int nnz) {

  //Get global index of thread (equivalent to index in val array)
  int gl_id = blockIdx.x * blockDim.x + threadIdx.x;

  //If index or thread less than nnz, get row index, column index,
     matrix entry value and x-value corresponding to column index
  if (gl_id < nnz) {
    int row = row_idx[gl_id];
    double entry = val[gl_id];
```

```
    int col_id = col_idx[gl_id]
    double x_i = x[col_id];

    //Add product to given row in y
    atomicAdd(&y[row], entry * x_i);
  }
}
```

The disadvantage of this approach is that it is necessary for each thread-block to read for elements from global memory whereas one elements must be written to global memory (except in the last block). There is also no guarantee of coalescence during reads from the x-array. A further issue is that threads writing to the same location in global memory may not do so in a parallel manner. Hence, each thread must use an `atomicAdd` to add to the corresponding row in **y**, lest nondeterministic results be produced.

Compressed Sparse Row/Column Format

The compressed sparse row (CSR) and compressed sparse column (CSC) formats consist, like COO, of three arrays. However, in place of one of the index arrays, they have an `offsets` array of integers of length $n+1$.

In CSR format, non-zero matrix entries are stored in row-major order. The `offsets[i]` array contains the starting index, in the `col_idx` and `val` arrays, of entries in row i, whereas `offsets[n]` $= nnz$. The `val` and `col_idx` arrays are as in COO format. Analogously, in CSC format, entries are stored in column-major order and `offsets[j]` contains the starting index of each column j. The memory requirement for CSR and CSC is thus

$$S(CSR/CSC) = (|\texttt{int}|) \times (nnz + n + 1) + (|\texttt{f}|) \times nnz \tag{4.2}$$

The CSR and CSC formats have become popular for sequential sparse linear algebra algorithms as they reduce the memory requirement relative to COO by `size_of(int)` $*\,(nnz - n - 1)$. For cases where $nnz >> n$, this memory reduction may be significant.

A simplified GPU kernel implementation of SpMV using CSR format is shown below.

```
__global__ void csr_mult(double* val, int* offsets, int* col_idx,
    double* x, double* y, int n, int nnz) {
  extern __shared__ int shared_mem[];
  //Get starting index of row and declare variable for end
  int row = blockDim.x*blockIdx.x + threadIdx.x;
  int row_start = offsets[row];
  int row_end;

```

```
8   //Write starting index to shared memory
9   shared_mem[threadIdx.x] = row_start;
10
11  //Get end index of row from next start
12  if(threadIdx.x < blockDim.x - 1){
13    row_end = shared_mem[threadIdx.x + 1];
14  }
15  else{
16    row_end = offsets[row + 1];
17  }
18  double row_sum = 0;
19  for (int i = row_start; i < row_end; i++) { //Loop over row
20    int col_id = col_idx[row_start + i];
21    double entry = val[row_start + i];
22    double x_i = x[col_id];
23    row_sum += entry*x_i;
24  }
25
26  __syncwarp();
27  y[row] = row_sum; //Set row sum as entry in result vector
28 }
```

Since the row of a particular entry cannot be inferred from its index in the `val` or `col_idx` arrays, each thread must be responsible for multiplying an entire row.

The multiplication algorithm resulting from CSR format is unsatisfactory for a number of reasons. First, in a similar manner to the COO kernel, each thread-block must read at least $3 +$ `offsets[row +1] - offsets[row]` elements from global memory whereas one element must be written. However, each thread processes an entire row. Furthermore, only the reads from `offsets` and the writes to `y` may be guaranteed to happen in a coalesced fashion. Lastly, the above format will inevitably induce a significant amount of thread latency when threads responsible for rows with very different number of non-zeros are included in the same thread-block. For a discretisation matrix resulting from an unstructured mesh, however, this problem may be somewhat ameliorated by reordering blocks and columns such that rows with an equal number of non-zero entries are stored contiguously.

Block Sparse Row Format

Block sparse row (BSR) format organises a matrix into blocks of some width and height m. Furthermore, only those blocks within which the matrix has non-zero entries are stored. Within each block that has at least one non-zero entry, the zero entries are stored explicitly [9]. If $nmod(m) \neq 0$, blocks at the right and bottom edges of the matrix are padded with

zeros. Similar to CSR format, blocks are stored in row-major order with a `block_offset` array for the block-rows of size $\lceil n/m \rceil + 1$ and a `block_col_idx` array for the respective block-column, of each block, of size `num_blocks`. The memory requirement for BSR format is thus

$$S(BSR) = |\mathtt{f}| \times \mathtt{num\_blocks} \times m^2 + |\mathtt{int}| \times (\lceil n/m \rceil + 1 + \mathtt{num\_blocks}). \tag{4.3}$$

The advantage of this approach is that it can take advantage of two-dimensional thread-blocks in CUDA and reduce the number of global memory reads. As an example, a CUDA thread-block may be launched for each block-row. Each thread is responsible for processing one element in a block. First, each thread reads the column index of the first block in the corresponding block-row. This operation is much faster than reading an individual column index for each thread, since the GPU can typically *broadcast* the same value more efficiently to all threads in a warp. Thereafter, each thread reads one entry of the matrix-block. These entries are multiplied with the **x**-value whose shared memory index corresponds to the respective column. According to the entries of `block_col_idx`, the thread-block then moves on to the next block on the block-row, in which it repeats the procedure. Each thread keeps track of the sum of products it has calculated from previous blocks. The thread block repeats this procedure until reaching the last block in the block-row, after which the product-sum from each of the threads in a row are added together in a variable `row_sum`, which may the be written to **y** by the first m threads.

On modern Nvidia GPUs, it would be natural to size the blocks at 32×32, with one thread-block covering one matrix block-row. This serves to capture information symmetrically in the matrix while allowing coalesced accesses each time the threads in a warp read or write from global memory.

Unfortunately, matrices resulting from finite volume discretisations tend not to have dense blocks large enough to justify such a format. Many zero-entries would be stored explicitly, drastically reducing the ratio of useful operations each thread performs.

Banded Block Format

An adjacent format to BSR format is the banded block format (BB). Taking as a parameter the height and length of each block, the banded block format stores continuous blocks all falling within some block band of non-zeros in the matrix. Within each block, all entries are stored, with fill-ins where there are zeros.

The benefit of this approach is that only one array is needed to store the matrix. However, there is a possibility of storing many zero-blocks, since the block band defines a constant

number of blocks in each block row row. Naturally, the BB format can benefit significantly from band-reducing reorderings of the matrix, such as Cuthill-Mckee or its reverse equivalent.

Clearly BB format is best suited for matrices with a similar number of dense non-zeros in each row. While meshes resulting from finite volume discretisations may have a small number of rows with a given number of non-zeros, there may be many rows with fewer entries than the maximum, which will lead to unnecessarily large memory-overhead.

ELLPACK Format

The ELLPACK format, and its varieties has gained much popularity in recent years [7]. The ELLPACK format starts by finding the maximal number of non-zeros in any given row, `mnz`. The matrix entries are then stored as an $n \times mnz$ array, with each row occupying `mnz` floats. Zero padding is necessary at the end of rows with less than `mnz` non-zero entries.

An additional integer array `col_idx`, of the same size, stores the column index of each entry in the matrix, with a placeholder of -1 for zero entries in the shorter rows. If a good reordering of the matrix is provided beforehand, elements of the matrix will be stored in a "jagged" column-major order, where the first non-zero elements of each row are stored contiguously, before the second etc. A kernel for the ELLPACK format may look as below, where each thread is responsible for multiplying one row of the matrix.

```
__global__ void ellpack_SpMV(double* entries, int* col_idx, int
    length, double* x_dev, double* res, int n_rows){
  int write_idx = blockIdx.x*blockDim.x + threadIdx.x; //get row
  int read_idx = (write_idx)*length; //get index of first element row

  double row_sum = 0;

  for(int i = 0; i < num_cols; i++){ //Loop over stored non-zeros
    int col_id = col_idx[read_idx]; //get column index
    double entry = entries[read_idx]; //get matrix entry

    if(col_id >= 0){ //if column index >= get x-entry and multiply-
        add
      double x_i = x_dev[col_id];
      row_sum += entry*x_i;
    }

    read_idx += n;
  }

  //write to global memory result-array
```

```
20    if(write_idx < n_rows){
21      res[blockIdx.x*blockDim.x + threadIdx.x] = row_sum;
22    }
23  }
```

It should be noted that while the above processes the SpMV operation with one thread per row, it is also possible to have each thread processing one element, with threads on the same row adding their elements upon completion of a kernel.

The key benefit of ELLPACK over the previously discussed formats, is that for matrices with small variance in the number of non-zero entries per row, it is able to keep down the fill-in, while storing the matrix in only 2 arrays, reducing the number of expensive global memory reads necessary. It may be seen that the storage requirement of ELLPACK format is

$$S(ELLPACK) = \texttt{mnz} \times (|\texttt{f}| + |\texttt{int}|) \tag{4.4}$$

In a similar manner to banded block format, it may be seen that ELLPACK will perform best on matrices with approximately as many non-zero entries in each row and no significant outliers from this distribution. In the context of FVM, meshes with an equal number of faces for each sell are thus optimally suited for this format.

Sliced ELLPACK Format

A variation of ELLPACK format is the so-called Sliced ELLPACK format (SELL), attributed to Monakov, Lokhmotov, and Avetisyan [8]. In SELL format, a given slice-height, `sh`, is provided as a parameter. The matrix is divided into "slices", each of `slice_height` contiguous rows, with zero-padding at the bottom of the matrix, if necessary. The number of floating point numbers stored by each slice is then slice-height multiplied by the highest number of non-zeros in any given row within the slice. SELL stores arrays `col_idx` and `val` in a similar manner to ELLPACK.

In addition, SELL must store an array `slice_ptr` pointing to the starting index in `col_idx` and `val` of each slice. A SELL kernel implementing SpMV may then be implemented as below. Note in particular the the length of each slice is inferred from the slice height and adjacent entries in the `slice_ptr` array.

```
1  __global__ void Sell_SpMV(double* entries, int* col_idx, int*
       slice_ptr, double* x, double* y, int n_rows){
2    int read_start = slice_ptr[blockIdx.x];
3    int read_end = slice_ptr[blockIdx.x + 1];
4    int num_cols = (read_end - read_start)/blockDim.x;
5    int write_idx = blockIdx.x*blockDim.x + threadIdx.x;
```

```
6   int read_idx = read_start + threadIdx.x;
7
8   double row_sum = 0;
9
10  for(int i = 0; i < num_cols; i++){
11    int col_id = col_idx[read_idx];
12    double entry = entries[read_idx];
13
14    if(col_id >= 0){
15      double x_i = x_dev[col_id];
16      row_sum += entry*x_i;
17    }
18
19    read_idx += blockDim.x;
20  }
21
22  if(write_idx < n_rows){
23    res[blockIdx.x*blockDim.x + threadIdx.x] = row_sum;
24  }
25 }
```

Sliced ELLPACK is presented as "the state of the art" by NVIDIA. This is not without reason. Despite standards changing quickly in the hardware world, the sliced ELLPACK format has proven remarkably efficient for a number of scenarios [7]. It may be seen that the storage requirement for sliced ELLPACK is given

$$S(SELL) = |\texttt{int}| \times (s+1) + |\texttt{f}| \times \texttt{sh} \times \sum^{s} \texttt{sl}, \tag{4.5}$$

where $s = \lceil n/\texttt{dc} \rceil$ is the number of slices and `sl` denotes slice lengths (i.e. the `mnz` within the slice).

Clearly, for $\texttt{sh} = 1$, the SELL format reverts to CSR format. At the other extreme, where $\texttt{sh} = n$, it reverts to the regular ELLPACK format. For cases where the number of non-zeros varies widely between rows, it may be easily seen that the storage requirement is significantly lower than that of ELLPACK.

Since the length of each slice is determined by the maximal number of non-zeros of any row within it, one may infer that the storage requirement, and thus the computational expense, may be reduced by reordering rows such that rows with a similar number of non-zeros fall into the same slice. However, such reordering may also affect the data locality in terms of the column indices stored at each row, such that access to elements in $\mathbf{x}$ becomes non-coalesced.

One may further conjecture that the `sh` should be reduced to the warp size of the given architecture, so as to reduce the number of slices containing "long rows" which will increase the fill-in of the whole slice. Such a strategy, however, may also reduce the number of cache hits where different rows share a column index, again reducing access speeds with respect to $\mathbf{x}$ [8].

A balance must thus be struck between grouping rows of similar length in a slice, finding an appropriate `slice_height`, and maintaining sufficient coalescence with respect to the necessary accesses to $\mathbf{x}$. This generalisation, where the SELL is applied to some permutation σ of a matrix with a `slice_height` of C, is coined with the name SELL-C-σ in [7].

4.3 Staircase SELL

We propose here a variant of the SELL-C-σ storage format, with an associated algorithm, suitable for matrix-vector multiplication on GPU for matrices derived from FVM meshes. While highly inspired by SELL format, Staircase SELL seeks to reduce the storage requirement and latency associated with the `slice_ptr` array, which must traditionally be stored in the GPU's global memory. The assumption is made that the number of different row-lengths (i.e. the number faces of a cell) is limited to some number N. Doing a matrix-vector multiplication Staircase SELL format involves four steps: finding an appropriate row ordering, establishing an indexing structure for the new order, moving matrix elements in memory to accommodate the new order, and finally executing the multiplication on GPU.

4.3.1 Staircase SELL: Row Ordering

Staircase SELL proceeds first by ordering the rows and columns of the given matrix according to the well known Cuthill-McKee (CM) algorithm. For a structurally symmetric graph (i.e. one in which node j is a neighbour of node i if and only if node i is a neighbour of node j), the CM procedure is summarised in algorithm 2. The CM algorithm is usually used to reduce the matrix bandwidth of structurally symmetric matrices, with each cell treated as a node and the neighbouring cells treated as neighbours. For matrices resulting from FVM discretisations, the resulting permuted matrix tends to display a number of "jagged diagonals". Figure 4.1 demonstrates this pattern for two reordered FVM discretisation matrices. In a second stage of the ordering step, staircase SELL then proceeds by ordering the rows according to the number of non-zero entries in each, with the previously computed CM-order as a tie-breaker within each group of a certain number of non-zeros. (The strategy of ordering rows by number of non-zeros was already implemented by the original authors of the algorithm [8]).

Algorithm 2 Cuthill-McKee Algorithm

Input: A structurally symmetric graph Γ. A set of sets S where S_i holds the neighbouring node indices of node i; index of a minimally connected node $q \in \{1, \cdots, n\}$
Output: An ordered permutation vector P representing the reordering of the nodes

```
1: Initialize P = (q)
2: Set i = 0, k = 1
3: Mark q as visited
4: while k < n do
5:     a = P_i
6:     Sort S_a by ascending order with index as a tie-breaker
7:     for j ∈ S_a do
8:         if j not yet visited then
9:             P_k = j
10:            Mark j as visited
11:            k = k + 1
12:        end if
13:    end for
14:    i = i + 1
15: end while
```

The objective of this second permutation is to minimise the fill-in as implied by SELL (i.e. grouping rows of similar number of non-zeros), while maintaining the approximate diagonal strips of non-zero entries in successive rows. For notational purposes, we shall label this permutation σ and its inverse σ^{-1}, where σ takes as its input a row index of the reordered matrix and outputs the corresponding row index in the original matrix.

During multiplication, the Staircase SELL algorithm seeks to process and store rows in the aforementioned groups. In doing so, concurrent execution of a kernel for each group is utilised. However, some groups may be prohibitively small, wasting resources on kernel launches and stream synchronisation.

4.3.2 Staircase SELL: Indexing and Merging Groups

Due to the possibility of small rows as mentioned above, the algorithm then proceeds to a blocking stage, at which rows are grouped in slices that will be processed concurrently by a given kernel. The algorithm iterates over the possible number of non-zeros in a row (i.e. $1, \cdots, N$), counting the number of rows, R, having a corresponding number of non-zeros. Starting from the group of rows with the least number of non-zeros, a group is merged with the next group if the relative number of rows, R/n, falls below some predefined threshold α. If the number of rows in the given group exceeds the threshold, the group is divided into

$\lfloor R/\texttt{sh} \rfloor$ slices, with the remaining $R - \texttt{sh}\lfloor R/\texttt{sh} \rfloor$ rows passed onto the next group. In the group with the maximal number of non-zeros, N, $\lceil R/\texttt{sh} \rceil$ slices are created, ensuring any fill-in rows occur only at the end of the array. An algorithmic outline of the procedure is given in algorithm 3.

Algorithm 3 Establish Index Structure

Input: Integer `sh` (slice height), vector `h` where `h[i]` holds number of rows whose number of non-zeros is i, threshold parameter $\alpha \in [\texttt{sh}/n, 1]$

Output: Vectors `step_lengths`, `step_blocks`, `step_ptr`, `row_ptr`; integer `val_size`

1: i, `val_size`, `step_ptr[0]`, `row_ptr[0]` $\leftarrow 0$
2: **for** $i \in \{0, \ldots, \texttt{mnz} - 1\}$ **do**
3: **if** `h[i]`$/n < \alpha$ **then**
4: `h[i+1]` $\leftarrow$ `h[i+1]` $+$ `h[i]`
5: `h[i]` $\leftarrow 0$
6: **else**
7: `num_blocks` $\leftarrow \lfloor$ `h[i]`$/$`sh` $\rfloor$
8: `h[i+1]` $\leftarrow$ `h[i+1]` $+ (\texttt{h[i]} - \texttt{sh} \times \texttt{num\_blocks})$
9: `h[i]` $\leftarrow$ `sh` $\times$ `num_blocks`
10: `val_size` $\leftarrow$ `val_size` $+ (i \times$ `h[i]`$)$
11: Append (`num_blocks`) to `step_blocks`
12: Append (i) to `step_lengths`
13: Append (`val_size`) to `step_ptr`
14: Append (`row_ptr[-1]` $+$ `h[i]`) to `row_ptr`
15: **end if**
16: **end for**
17: Append $(\lceil$ `h[mnz]`$/$`sh` $\rceil)$ to `step_blocks`
18: `h[mnz]` $\leftarrow$ `sh` $\times \lceil$ `h[mnz]`$/$`sh` $\rceil$
19: Append (`mnz`) to `step_lengths`

In algorithm 3, `step_lengths` denotes the length of (in terms of store non-zeros per row) of each stored group. Such a vector is necessary as it would be possible for the algorithm to skip bins of a certain number of non-zeros, at which there are too few rows. In other words, the first group of rows may have four stored non-zeros whereas the second group may have six. This is especially true after merging rows as above. The vector `step_blocks` stores the number of blocks, each containing `sh` rows, that are stored in a given group. In explanation, `step_blocks[i]` denotes the number of thread-blocks that will be launched for the group of rows with `step_lengths[i]` non-zeros. The vector `step_ptr` points to the starting index

of each group in the array of all entries, `new_val` (see next algorithm). Similarly, the array `row_ptr` points to the first row, in the new order, of each group.

In following this procedure, Staircase SELL aims at maintaining the lower storage requirement possible when regrouping rows by the number of non-zeros, while merging groups whose size does not warrant individual processing and storage. In addition, the algorithm allows for auto-tuning the α and `slice_height` parameters.

It is worth noting that, up until this point, the actual entries of the given matrix do not need to be moved in memory, since the reordering procedure depends only on their position within the matrix, as given by their row- and column indices.

4.3.3 Staircase SELL: Reorganising the Matrix in Memory

At the next stage, the algorithm proceeds to store the entries of the matrix in ELLPACK format *within each group*. While the blocking stage will cause some rows to be stored in a slice longer than their number of non-zeros, this reduces the need to launch kernels for very small groups. Assuming the matrix is stored in CSR format, the process of re-storing the matrix may be summarised as in algorithm 4.

Algorithm 4 Store in Staircase SELL format

Input: Matrix **A** in CSR format with row-offset vector `offsets`, column index vector `col_idx`, and values vector `val`, permutation σ

Output: Arrays `new_val` and `new_col`

```
1: row ← 0
2: Initiate array of floats new_val, of size val_size, to zero
3: Initiate array of integers new_col, of size val_size, to −1
4: for i ∈ {0,...,num_steps} do
5:    length ← step_lengths[i]
6:    start_idx ← step_start[i]
7:    for j ∈ {0,...,h[i]} do
8:       slice_idx ← ⌊j/sh⌋
9:       if row < n then
10:         a ← σ(row)
11:         c ← vector of column indices on row a sorted according to σ^{-1}
12:         e ← vector of matrix entries on row a in same order as c
13:         for k ∈ {0,...,# non-zeros row a} do
14:            idx ← start_idx + (slice_idx × length × sh) + (k × sh) + j mod(sh)
15:            new_val[idx] ← e[k]
16:            new_col[idx] ← c[k]
17:         end for
18:      end if
19:      row ← row + 1
20:   end for
21: end for
```

In the above algorithm, slices of rows are stored contiguously within each group. Within each slice, rows are stored in the same "jagged diagonal" format as mentioned for ELLPACK: the first non-zero elements in each row are stored contiguously, followed by the second non-zero elements etc. For rows in slices "longer" than their number number of non-zeros (i.e. due to the merging) fill-ins of 0 and -1 are used in the `new_val` and `new_col` arrays, respectively.

It should be noted also that within each row, the permuted column indices are sorted in ascending order, along with the associated matrix values, to benefit from the earlier reordering. Furthermore, one should note that for $\alpha = \mathtt{sh}/n$, the algorithm reverts to SELL under the new ordering.

4.3.4 Staircase SELL: Multiplication Kernel

In executing the matrix-vector multiplication, each step-group is processed independently. However, the same kernel is used for each group. The kernel, presented in code-snippet below, is supplied with the nr of non-zeros (`length`) of the given group; the index of the first row (`row_start`) in the group; and the starting index (`step_start`) of the group within the overall arrays `new_val` and `new_col`.

Based on `step_start`, the kernel begins by computing the index of the first entry in `new_val` and `new_col` to be read by the each of the threads in the given thread-block. Subsequently, the kernel computes the index of the row for which each thread is responsible based on `row_start`. These variables are constant for each thread during the execution of a thread-block and are named `read_idx` and `write_idx`, respectively. Thereafter, variables `entry`, `col_id`, `x_i`, and `row_sum` are declared so has to hold an entry of the matrix, the corresponding column index, and the sum over all entries in the row for which the thread is responsible, respectively. (Note that `row_sum` is initialised to zero).

The kernel then enters a loop of `length` iterations for each thread. In each iteration, each thread reads `new_val[read_idx]` into `entry` and `new_col[read_idx]` into `col_id`. Note that these reads from global memory may be made in an entirely coalesced manner in accordance with the definition of `read_idx`. If `col_id` > 0, the relevant threads then add `entry` $\times$ `x[col_id]` to `row_sum`. As a final step within the loop, `read_idx` is incremented by `blockDim.x` $=$ `sc`, in accordance with the storage format described above. Upon completion of the loop, each thread adds its copy of `row_sum` to the corresponding location in the result-array `y`.

```
__global__ void staircase_SpMV(int length, int row_start, int
    step_start, int n_rows){
  int read_idx = step_start + length*blockDim.x*blockIdx.x +
     threadIdx.x;
  int write_idx = row_start + blockIdx.x*blockDim.x + threadIdx.x;

  double row_sum = 0;

  double entry;
  int col_id;
  double x_i;
  for(int i = 0; i < length; i++){
      entry = entries_ptr[read_idx];
      col_id = col_ptr[read_idx];
      if(col_id > 0){
        x_i = x_ptr[col_id];
```

```
15          row_sum += entry*x_i;
16        }
17        read_idx += blockDim.x;
18    }
19
20    if(write_idx < n_rows){
21      y_ptr[write_idx] = row_sum;
22    }
23  }
```

4.3.5 Execution from CPU-side

In solving a large system of linear equations, several thousand steps of the CG-algorithm are typically performed before convergence. Furthermore, given a converged solution, a new linear system of linear equations, so as to progress the algorithm, may be generated directly on GPU under the finite volume method. While this aspect will be covered in future work, the essence of the matter is that the time to sort and re-order the system in memory may typically be neglected relative to the time taken to complete a full simulation. Therefore, the above steps are assumed here to be implemented on the CPU-side.

Having obtained the restructured arrays `new_val` and `new_col`, the old arrays `val`, `row_idx` and `col_idx` may be deleted. For the remainder of the discussion, we shall rename the new arrays `val` and `col_idx`, respectively.

The new arrays may now be copied onto GPU. Importantly, the indexing arrays `step_lengths`, `step_blocks`, `step_ptr`, `row_ptr` stay on CPU side. The multiplication process makes use of CUDA's streaming feature which allows for a kernel to be launched concurrently with different arguments by several GPU-level "threads". Naturally, the number of streams is set to equal the number of groups in the new matrix ordering. Each stream launches a kernel with `step_blocks[stream_index]` blocks of `sh` threads each. The GPUs scheduler is responsible for allocating resources between streams with the objective of keeping all multiprocessors occupied until completion of the kernel[9]. Lastly, note that pointers to the arrays `new_val`, `new_col`, `x`, and `y` were loaded to constant memory before running the kernel. Constant memory is cached on device, and thus quickly accessible by all threads.

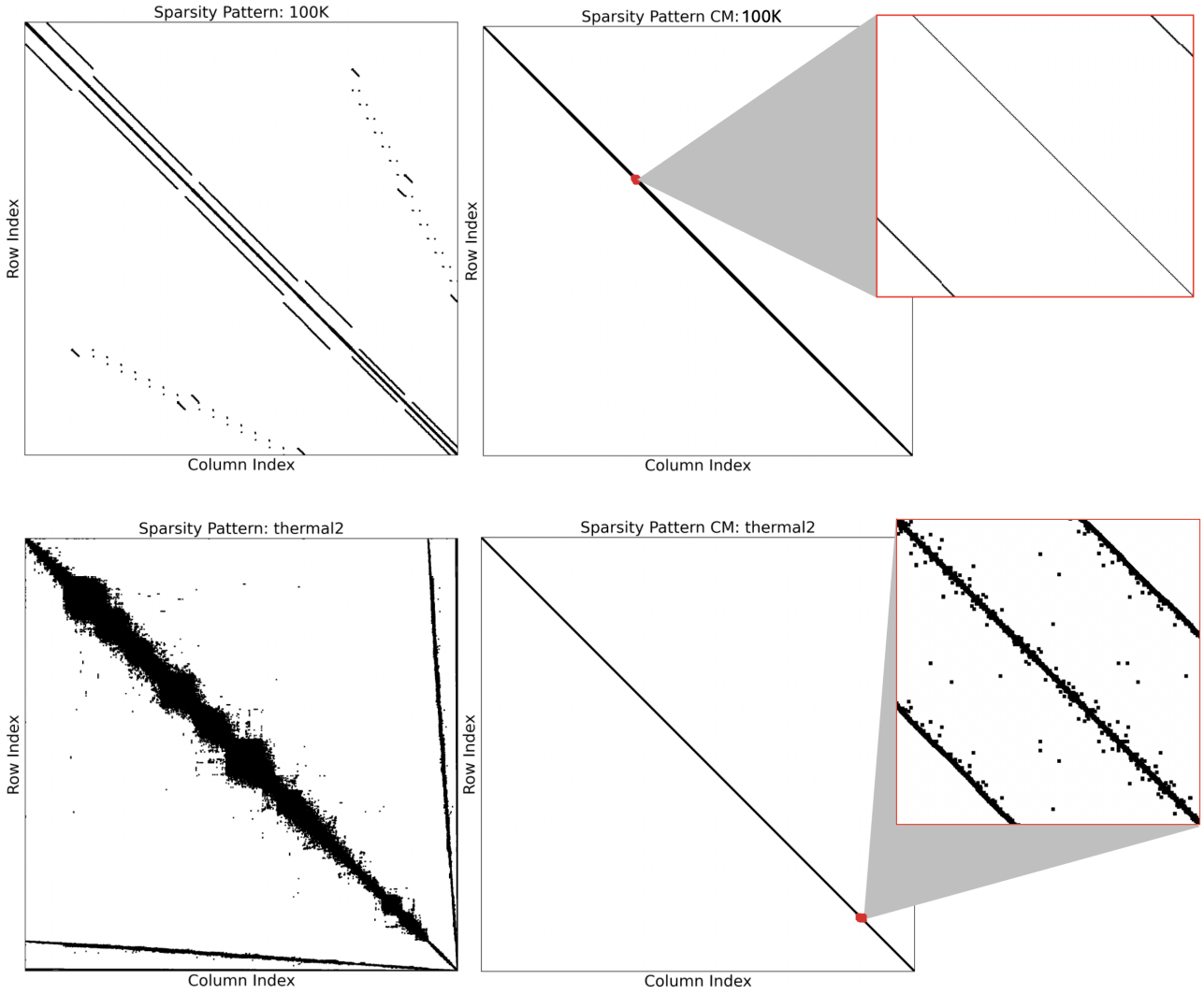


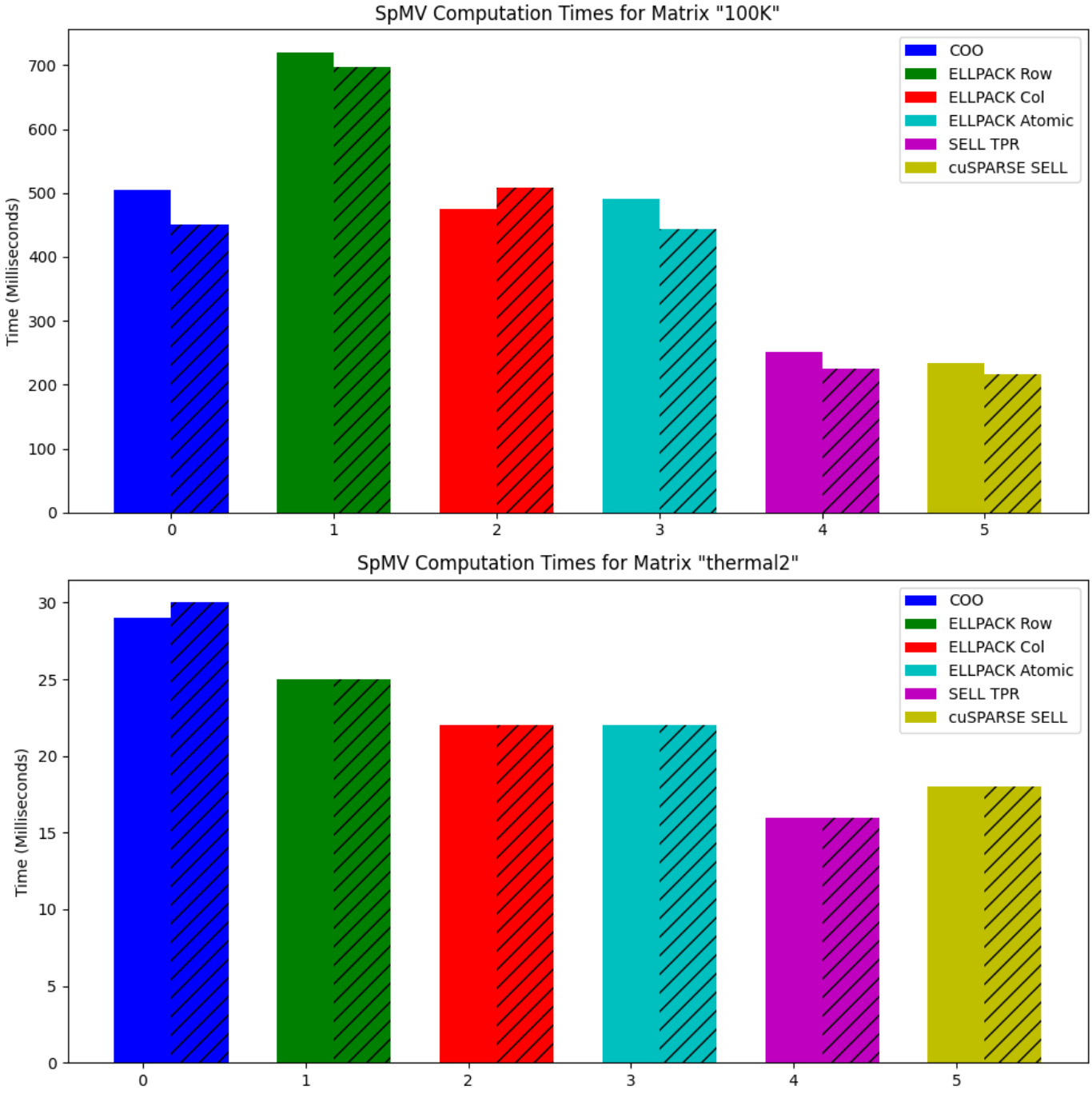
Fig. 4.1 Top Left: Sparsity pattern of matrix "100K" unpermuted. Top Right: Sparsity pattern of matrix permuted according to Cuthill-Mckee ordering. Bottom Left and Right: Equivalent for matrix "thermal2". See table 1 for matrix information

Chapter 5

Results

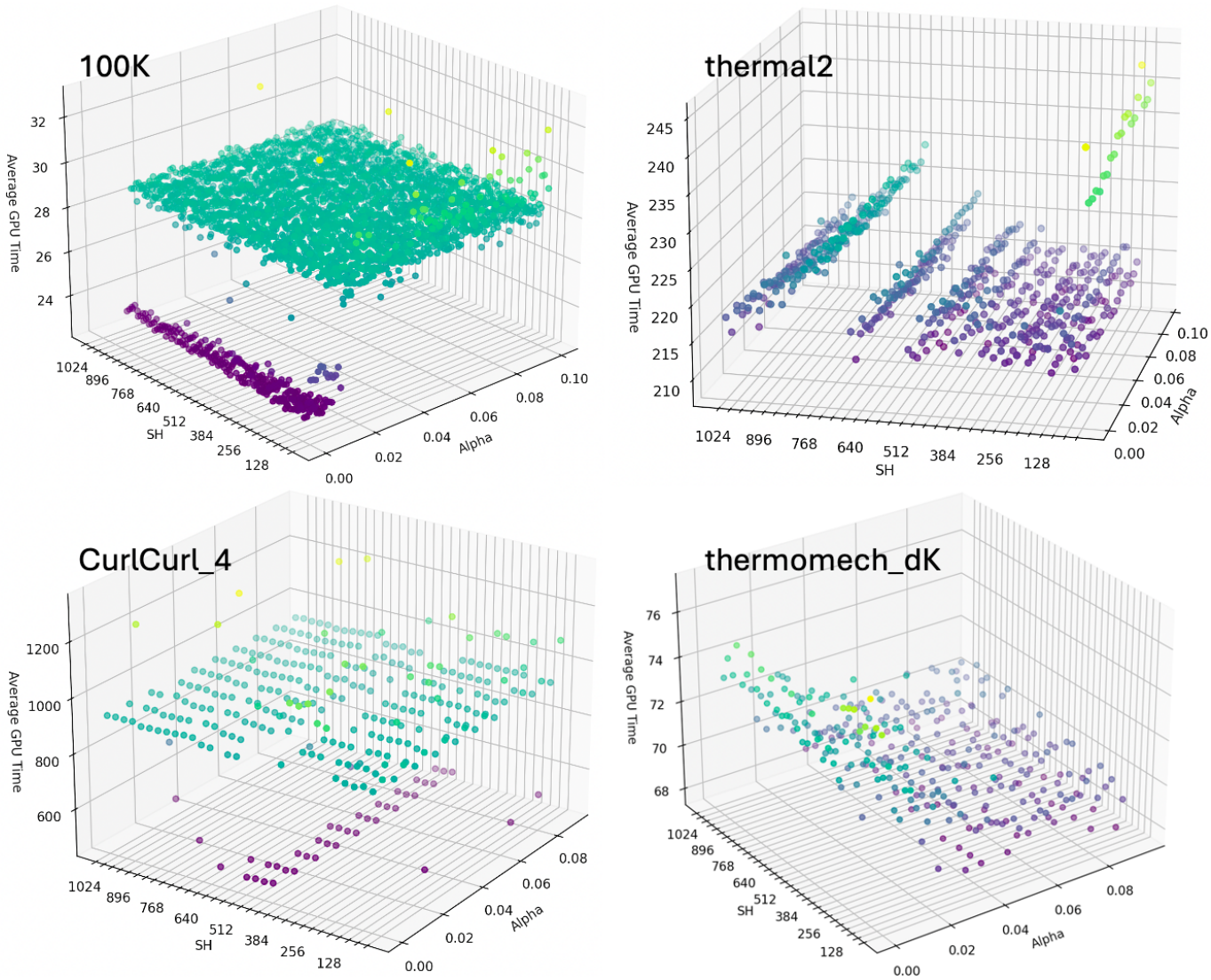
In Figure 5.1 are presented the results for some of the different matrix multiplication formats as listed above for the matrices "thermal2" and "100K", the sparsity patterns of which are seen in 4.1. "ELLPACK Row" denotes matrix in ELLPACK format where non-zeros in a row are stored contiguously and each thread handles one non-zero. "ELLPACK Col" and "ELLPACK Atomic" denote formats where the first non-zero of each row in a slice are stored contiguously, followed by the second non-zeros etc. On "ELLPACK Col" and "ELLPACK Row" reductions were done using shared memory. On "ELLPACK Atomic", products of single matrix non-zeros and corresponding elements in $\mathbf{x}$ were added atomically to global memory using `atomicAdd`. "SELL TPR" denotes SELL format where each thread finds the dot-product of one matrix row with $\mathbf{x}$. Note that figures are given for a SELL implementation as produced by the author and for the cuSparse-version. The slice height was set uniformly to 1024. All times were computed as an average over 100 SpMV operations (timed individually), each as part of one iteration of the CG algorithm. This timing procedure was used specifically because the order in which thread-blocks execute on GPU cannot be guaranteed *a priori* of any operation. Furthermore, the CG-algorithm was used as a buffer around each SpMV operation to simulate the normal conditions under which it will operate and to avoid any values of $\mathbf{x}$ being preserved in cache.

Due to the superiority of the SELL format, it was decided to compare Staircase SELL with that format. Before such a comparison, however, the space of possible combinations of α and `sh` were explored for 4 matrices from the SuiteSparse library to determine optimal pair for each matrix, and any "good combination" that may serve as a guideline for the general case. Figure 5.2a shows the timings over several combination of these parameters. Note that α is not allowed to subcede $\texttt{sh}/n$, as this would imply that groups of rows are allowed to be processed even if being fewer than the given thread-block dimension. The times in all plots are provided in milliseconds.



(a) Thermal Times Comparison

Fig. 5.1 Comparison of SpMV average times (in milliseconds) for different multiplication formats on matrices "thermal2" (BOTTOM) and "100K" (TOP). Striped bars represent computation times for matrices permuted using Cuthill-McKee algorithm.



(a) Thermal Times Comparison

Fig. 5.2 Comparison of computation times for Staircase SELL in space of α and sh values. Matrices "100K", "thermal2", "CurlCurl_4", and "thermomech_dK". sh was tested for every multiple of 32 lower than 1024 (the maximal thread-block size).

The optimal performance, of staircase SELL and cuSparse the 4 matrices is seen in table 5.2. For both algorithms, the matrices were ordered according to the Staircase SELL ordering for better comparisons. For those matrices included in figure 5.2a, the minimum time reported is for the minimal computation time across all values of α and sh as shown in the figure. At the industrial and research scale, modern FVM codes tend to push the limits, as regards memory of the architectures on which they operate, since researchers typically

want to use as fine a mesh as possible, on the available technology. Table 5.1 also gives information on the size, in bytes, of each of the matrices on which the algorithms were tested.

Matrix names	**Source**	**n_rows**	**avg non-zeros**	**std nnpr**	**Problem type**	**Symmetric**
100K	[6]	102400	4.988	0.111	FVM	Yes
thermal2	[3]	1228045	3.99	1.766	FVM	Yes
CurlCurl_4	[3]	2380515	11.138	0.855	Other	Yes
thermomech_dK	[3]	204316	13.93	1.430	Other	No

Table 5.1 Matrices Information: n_rows: number of rows, avg non-zeros: average number of non-zeros in each row, std nnpr: standard deviation of nuber of non-zeros

Matrix names	**sh**	α	**Size SC**	**Size SELL**	**Min time SC**	**Min Time cuSp**
100K	672/32	0.0076	11088000	11042048	**22.80**	**19.97**
thermal2	512/928	0.04	103084032	103091248	**209**	**203**
CurlCurl_4	448/608	0.010	318286080	318296368	**487**	**487.048**
thermomech_dK	480/64	0.092	34225920	34182628	**67**	**62**

Table 5.2 Matrix storage sizes on GPU side for different storage formats. α paramter only relevant for Staircase SELL (SC) algorithm. Min time is minimum time of SpMV operation with an (n_rows) long vector computed using CUDA events. Vector entry $i = imod(5)$

Chapter 6

Discussion and Conclusions

As may be seen from the tables above, the performance of the two algorithms is very similar. We note that the storage requirements on GPU-side are also highly similar. An interesting aspect of the Staircase SELL algorithm is that it manages to keep up with the cuSparse implementation of SELL, despite the scheduling costs associated with its different streams.

Figure 5.2 reveal interesting patterns to the sensitivity of the Staircase SELL algorithm to its input parameters. On all tested matrices, minimal computation times were recorded for thread-block sizes close to 512. Testing on many further matrices and with different combinations of the input parameters are needed to determine whether Staircase SELL poses a viable alternative to cuSparse SELL for industrial or research purposes solving sparse linear systems for the finite volume method. In particular, the author is interested in exploring cachin strategies on NVIDIA GPU, whereby preference may be given to certain values to be retained in cache between kernel calls.

Exploiting the symmetry or symmetric structure of sparse matrices from FVM applications on GPU remains a challenge which is difficult to solve mainly due to the memory architecture of the devices. Owing to the large size of many of these matrices, solving this challenge could lead to big gains in terms of computation times.

References

[1] Bell, N. and Garland, M. (2008). Efficient sparse matrix-vector multiplication on cuda. Technical Report NVR-2008-004, NVIDIA Corporation.

[2] Bell, N. and Garland, M. (2012). *CUSP: Generic Parallel Algorithms for Sparse Matrix and Graph Computations*. NVIDIA Corporation. Version 0.5.1.

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

[4] Elafrou, A., Goumas, G., and Koziris, N. (2019). Conflict-free symmetric sparse matrix-vector multiplication on multicore architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19)*, New York, NY, USA. ACM.

[5] Evans, J., Finder, I., Goldwasser, I., Linford, J., Mehta, V., Ruiz, D., and Wagner, M. (2023). Nvidia grace cpu superchip architecture in depth. Accessed at: https://developer.nvidia.com/blog/nvidia-grace-cpu-superchip-architecture-in-depth/.

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

[7] Kreutzer, M., Hager, G., Wellein, G., Fehske, H., and Bishop, A. R. (2014). A unified sparse matrix data format for efficient general sparse matrix-vector multiply on modern processors with wide simd units. *arXiv preprint arXiv:1307.6209*.

[8] 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.

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

[10] NVIDIA Corporation (2021). *cuSPARSE Library*. NVIDIA Corporation. Version 11.4.

[11] Team, A. D. (2021). *AmgX: A Library for GPU Accelerated Algebraic Multigrid and Preconditioned Iterative Methods*. NVIDIA Corporation. Version 2.1.