FRIEDRICH-ALEXANDER-UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)

Optimization of Implicit Finite Element Solver for Time-Dependent Navier–Stokes Equations

Antoine Kempf

Master's Thesis

Optimization of Implicit Finite Element Solver for Time-Dependent Navier–Stokes Equations

Antoine Kempf

Master's Thesis

Aufgabensteller:	Prof. Dr.-Ing. Harald Köstler
Betreuer:	Dr. Atsushi Suzuki
Bearbeitungszeitraum:	May 2025 - September 2025

Eigenständigkeitserklärung

Hiermit versichere ich, Antoine Kempf, die vorgelegte Arbeit selbstständig und ohne unzulässige Hilfe Dritter sowie ohne die Hinzuziehung nicht offengelegter und insbesondere nicht zugelassener Hilfsmittel angefertigt zu haben. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen und wurde auch von keiner anderen Prüfungsbehörde bereits als Teil einer Prüfung angenommen.

Die Stellen der Arbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

Mir ist insbesondere bewusst, dass die Nutzung künstlicher Intelligenz verboten ist, sofern diese nicht ausdrücklich als Hilfsmittel von dem Prüfungsleiter bzw. der Prüfungsleiterin zugelassen wurde. Dies gilt insbesondere für Chatbots (insbesondere ChatGPT) bzw. allgemein solche Programme, die anstelle meiner Person die Aufgabenstellung der Prüfung bzw. Teile derselben bearbeiten könnten.

Verstöße gegen die o.g. Regeln sind als Täuschung bzw. Täuschungsversuch zu qualifizieren und führen zu einer Bewertung der Prüfung mit „nicht bestanden“.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen dieser Arbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 10. Oktober 2025 .....

Abstract

This thesis addresses the numerical solution of the unsteady incompressible Navier–Stokes equations using an implicit finite element formulation. The resulting nonlinear systems are solved by Newton's method, leading at each step to large sparse linear systems. Their resolution relies on the Portable, Extensible Toolkit for Scientific Computation (PETSc) [1] library, where the sparse matrix-vector multiplication (SpMV) constitutes the performance-critical kernel. We investigate algorithmic and hardware-oriented optimizations of SpMV and their integration within Krylov subspace methods. Both Compressed Sparse Row (CSR) and block CSR (BCSR) formats are considered to store the sparse matrix, with additional low-level enhancements such as FMA and AVX2 vectorization. These kernels are incorporated into a GMRES [2] solver with ILU [3] preconditioning implemented in PETSc. The optimized solver is applied to benchmark configurations, including channel flow and flow around an obstacle. Results show significant improvements compared to baseline PETSc implementations, both for standalone kernels and full Navier–Stokes simulations. This work highlights the benefits of combining robust numerical algorithms with architecture-aware optimizations to accelerate implicit solvers for time-dependent incompressible flow problems.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Dr. Atsushi Suzuki, Dr. Pierre Jolivet, and Prof. Dr.-Ing. Harald Köstler, for their continuous guidance and support throughout this work. I am also deeply grateful to the RIKEN Center for Computational Science (R-CCS), where I had the opportunity to carry out my master's thesis internship. In particular, I wish to thank the Large-Scale Parallel Numerical Computing Technology Research Team, led by Dr. Toshiyuki Imamura, as well as all team members and fellow interns, for their valuable discussions, collaboration, and warm welcome. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

Zusammenfassung

Diese Arbeit befasst sich mit der numerischen Lösung der instationären inkompressiblen Navier–Stokes-Gleichungen unter Verwendung einer impliziten Finite-Elemente-Formulierung. Die entstehenden nichtlinearen Gleichungssysteme werden mit dem Newton-Verfahren gelöst, wobei in jedem Schritt große dünnbesetzte lineare Systeme auftreten. Deren Lösung erfolgt mit der Portable, Extensible Toolkit for Scientific Computation (PETSc)-Bibliothek [1], in der das Sparse-Matrix–Vektor-Produkt (SpMV) den leistungsbestimmenden Kern darstellt. Untersucht werden algorithmische und hardwareorientierte Optimierungen von SpMV sowie deren Integration in Krylov-Unterraumverfahren. Dabei werden sowohl das CSR- als auch das BCSR-Format berücksichtigt. Zusätzliche Optimierungen wie FMA-Operationen und AVX2-Vektorisierung steigern die Recheneffizienz. Diese Kernel werden in einen GMRES-Solver [2] mit ILU-Vorconditionierung [3] integriert, der vollständig in PETSc implementiert ist. Der optimierte Solver wird auf Benchmark-Konfigurationen wie Kanalströmung und Strömung um ein Hindernis angewendet. Die Ergebnisse zeigen deutliche Leistungssteigerungen im Vergleich zu den Standardimplementierungen von PETSc, sowohl auf Kernel- als auch auf Solver-Ebene. Die Arbeit unterstreicht den Nutzen der Kombination von robusten numerischen Algorithmen mit architekturbewussten Optimierungen zur Beschleunigung impliziter Solver für zeitabhängige inkompressible Strömungsprobleme.

Danksagung

Ich möchte meinen Betreuern, Dr. Atsushi Suzuki, Dr. Pierre Jolivet und Prof. Dr.-Ing Harald Köstler, meinen aufrichtigen Dank für ihre kontinuierliche Unterstützung und wertvolle Betreuung während dieser Arbeit aussprechen. Mein besonderer Dank gilt auch dem RIKEN Center for Computational Science (R-CCS), wo ich die Gelegenheit hatte, mein Masterarbeit-Praktikum durchzuführen. Insbesondere danke ich dem Large-Scale Parallel Numerical Computing Technology Research Team unter der Leitung von Dr. Toshiyuki Imamura sowie allen Teammitgliedern und Praktikanten für die hilfreichen Diskussionen, die gute Zusammenarbeit und den herzlichen Empfang. Die in dieser Arbeit präsentierten Experimente wurden auf dem Grid'5000-Testbed durchgeführt, das von einer wissenschaftlichen Interessengruppe getragen wird, die von Inria gehostet wird und zu der das CNRS, RENATER sowie mehrere Universitäten und weitere Organisationen gehören (siehe https://www.grid5000.fr).

Contents

Chapter 1

Introduction

1.1 Context and Motivation

The numerical simulation of fluid flows has become an indispensable tool in engineering, physics, and applied sciences. Among the various models, the incompressible Navier–Stokes equations occupy a central position for the description of viscous, incompressible flows. Their numerical solution is particularly demanding in the time-dependent regime, where implicit time discretization is often required to ensure stability at practical time step sizes which are balanced to the spatial discretization. This choice leads to a sequence of large, sparse, and nonlinear algebraic systems that must be solved efficiently at each time step. In modern computational fluid dynamics (CFD), the efficiency of the linear algebra backend often constitutes the principal computational bottleneck. At the heart of Krylov subspace methods, the sparse matrix-vector multiplication (SpMV) represents the dominant kernel, executed repeatedly throughout the iterative process. It is well known that the performance of SpMV is limited less by floating-point arithmetic throughput than by memory bandwidth, as the operation involves irregular memory accesses and a low arithmetic intensity. This imbalance is further exacerbated on high-performance computing architectures, where the gap between processor speed and memory bandwidth continues to be widen. As a result, SpMV has emerged as a prime target for optimization efforts, both in terms of data structures and hardware-aware implementations.

1.2 Objectives

The overarching goal of this thesis is to investigate how low-level performance optimizations of sparse linear algebra kernels can accelerate Krylov subspace solvers in the context of time-dependent incompressible Navier–Stokes equations. The study is conducted within the PETSc framework, which offers a mature infrastructure for iterative solvers, nonlinear methods, and preconditioning techniques, and is widely used in large-scale scientific computing.
More specifically, the thesis pursues the following objectives:

1. **Sparse matrix–vector kernels.** To develop and evaluate baseline implementations of SpMV using both the classical CSR format and the BCSR format. This provides a controlled environment to compare data layouts that differ in memory locality and vectorization potential.

2. **Hardware-aware optimizations.** To incorporate advanced instruction-level optimizations, including FMA operations and AVX2 vectorization, in order to exploit the floating-point throughput of modern CPUs. The impact of these optimizations will be assessed in terms of arithmetic intensity and cache efficiency.

3. **Numerical integration.** To embed the optimized kernels into a Newton–Krylov solver for the Navier–Stokes equations, with ILU preconditioning, thereby connecting low-level optimizations with algorithmic performance at the solver level.

4. **Performance analysis.** To quantify the influence of data structure design and hardware-level optimizations on solver scalability, using both roofline analysis and direct benchmarking. The aim is to understand the trade-offs between memory traffic reduction and computational overhead.

5. **Benchmarking on canonical problems.** To validate the approach on representative CFD test cases, including canonical incompressible flows, and to assess the practical benefits of the optimized kernels in realistic Navier–Stokes simulations.

Through these objectives, the thesis seeks to bridge the gap between numerical algorithms and hardware-oriented optimizations, thereby contributing to the design of efficient large-scale solvers for incompressible flows.

1.3 Structure of the Thesis

The manuscript is structured as follows. SpMV, emphasizing its central role in scientific computing, the most common storage formats, and the performance bottlenecks that arise from memory traffic and irregular access patterns. Chapter 3 develops baseline implementations in the CSR and BCSR formats, and subsequently introduces low-level optimizations such as FMA operations and AVX2 vectorization. Benchmarking results are presented to highlight the benefits and limitations of these approaches. Chapter 4 recalls the incompressible Navier–Stokes equations, details their variational formulation and finite element discretization, and discusses the nonlinear solution strategy implemented within PETSc. Particular attention is given to the integration of the optimized kernels into Newton–Krylov methods with ILU preconditioning, and to solver performance on representative flow configurations. Finally, Chapter 5 summarizes the main contributions of the thesis and outlines possible directions for future research.

Chapter 2

Sparse Matrix-Vector Product General presentation

2.1 Role in scientific computing

The sparse matrix–vector product (SpMV) refers to the fundamental linear algebra operation

$$\mathbf{y} = A\mathbf{x}, \tag{2.1}$$

where $A \in \mathbb{R}^{n \times n}$ is a sparse matrix, $\mathbf{x} \in \mathbb{R}^n$ is an input vector, and $\mathbf{y} \in \mathbb{R}^n$ is the output. Unlike the dense case, in which all n^2 entries of A contribute, the sparse formulation exploits the fact that the vast majority of entries are zero and need not be stored nor explicitly processed. Only the nonzero coefficients are accessed and multiplied, which leads to substantial savings in both memory footprint and arithmetic work.

It is widely acknowledged that SpMV constitutes one of the most central kernels in numerical linear algebra. Its importance stems from its repeated use in iterative methods for the solution of large-scale sparse linear systems, most notably Krylov subspace algorithms such as the conjugate gradient (CG) method [4] and the generalized minimal residual (GMRES) method [2]. Since each iteration of these methods typically requires one or more SpMV operations, their overall efficiency is directly tied to the performance of this kernel.

Beyond iterative solvers, SpMV arises in a wide range of computational tasks. It plays a key role in eigenvalue computations (e.g. Arnoldi or Lanczos iterations)[5–7] and in numerous scientific and engineering applications. In particular, when partial differential equations are discretized by finite difference, finite volume, or finite element methods, the resulting matrices are typically sparse and very large, making SpMV the dominant operation in large-scale simulations such as fluid dynamics, structural mechanics, or electromagnetics.

From the perspective of high-performance computing, the significance of SpMV is further amplified. Due to its low arithmetic intensity and irregular memory access patterns, the operation is almost invariably limited by memory bandwidth rather than floating-point operation. As a consequence, the efficiency of SpMV often dictates the scalability of entire simulation codes. This explains the considerable research

effort devoted over the past decades to improving its implementation, whether by designing storage formats that reduce memory traffic, exploiting vectorization and cache reuse, or adapting to specific hardware architectures such as GPUs and many-core processors.

In summary, the sparse matrix–vector product can be regarded as the cornerstone of large-scale numerical simulation. Its performance directly impacts the feasibility of solving ever larger and more complex models, and thus remains a prime target for algorithmic and architectural optimizations.

2.2 Sparse Matrix Storage Formats

Efficient implementation of the SpMV requires a careful choice of storage format for the sparse matrix. The data structure not only determines the memory footprint but also directly impacts cache locality, memory bandwidth usage, and the potential for vectorization. Since SpMV is a memory-bound operation, these aspects are often more critical than the number of floating-point operations. Two formats widely adopted in scientific computing are the classical CSR and its block variant, BCSR.

2.2.1 Compressed Sparse Row

The CSR format is arguably the most widely used representation of sparse matrices, thanks to its balance of simplicity, compactness, and generality. It encodes the matrix through three one-dimensional arrays:

- `values`: the nonzero coefficients, stored row by row,
- `col_ind`: the column index associated with each stored value,
- `row_ptr`: pointers indicating the starting position of each row in the `values` array.

For instance, the matrix

$$A = \begin{bmatrix} 10 & 0 & 0 & 7 \\ 0 & 3 & 0 & 0 \\ 2 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is stored in CSR as

$$\texttt{values} = [10, 7, 3, 2, 5, 1], \quad \texttt{col\_ind} = [0, 3, 1, 0, 2, 3], \quad \texttt{row\_ptr} = [0, 2, 3, 5, 6].$$

Here, `row_ptr` specifies that the first row spans indices $[0, 2)$ in `values`, the second row $[2, 3)$, and so on. This compact structure allows any sparsity pattern to be represented efficiently, which explains the widespread adoption of CSR in scientific computing.

2.2.2 Block Compressed Sparse Row (BCSR)

In many applications, particularly those stemming from finite element or finite volume discretizations, the global matrix exhibits a natural block structure. For example, in incompressible flow simulations with equal order finite element discretization for both velocity and pressure, each mesh node carries three velocity unknowns and one pressure unknown, leading to a 4×4 block pattern.

The BCSR format takes advantage of this by grouping coefficients into fixed-size dense blocks of dimension $b \times b$. Instead of storing each nonzero individually, only the nonzero blocks are kept, together with their column indices. This reduces indexing overhead and allows the use of small dense kernels.

Revisiting the previous example with block size $b = 2$, the matrix is partitioned as

$$A = \left[\begin{array}{cc|cc} 10 & 0 & 0 & 7 \\ 0 & 3 & 0 & 0 \\ \hline 2 & 0 & 5 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right],$$

which yields the nonzero blocks

$$B_{00} = \begin{bmatrix} 10 & 0 \\ 0 & 3 \end{bmatrix}, \quad B_{01} = \begin{bmatrix} 0 & 7 \\ 0 & 0 \end{bmatrix}, \quad B_{10} = \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix}, \quad B_{11} = \begin{bmatrix} 5 & 0 \\ 0 & 1 \end{bmatrix}.$$

The storage arrays are then

$$\texttt{block\_values} = [B_{00}, B_{01}, B_{10}, B_{11}], \quad \texttt{block\_col\_ind} = [0, 1, 0, 1], \quad \texttt{block\_row\_ptr} = [0, 2, 4].$$

2.3 Performance limitations

The performance of sparse matrix–vector multiplication (SpMV) is fundamentally limited by its low *arithmetic intensity*, defined as the ratio between the number of floating-point operations and the total volume of data transferred from memory. A rigorous quantification of this effect is obtained through the *code balance* B_c (in bytes per floating-point operation), from which the attainable arithmetic intensity $I = 1/B_c$ directly follows.

Code balance of the CSR kernel

In the Compressed Sparse Row (CSR) format, the SpMV operation

$$y_i \mathrel{+}= \sum_{j=\texttt{rowptr}[i]}^{\texttt{rowptr}[i+1]-1} a_j \, x_{\texttt{colidx}[j]}, \qquad i = 1, \ldots, N_r,$$

performs two floating-point operations (one multiplication and one addition) per nonzero coefficient. Let N_r, N_c, and N_{nz} denote the number of rows, columns, and nonzero entries, respectively. The average number of nonzeros per row is $N_{\mathrm{nzr}} = N_{nz}/N_r$, and per column $N_{\mathrm{nzc}} = N_{nz}/N_c$.

Each nonzero entry contributes to the memory traffic through:

- one 8-byte load for the value array `val`,

- one 4-byte load for the column index `col_idx`,
- a contribution to the input vector x (8 bytes),
- and an update of the output vector y (8 bytes for load + 8 bytes for store, amortized per row).

Taking all terms into account, including the row pointer `rowptr` (4 bytes per row), the minimal code balance of the CSR kernel can be expressed as

$$B_{c,\min} = \frac{12 + \dfrac{20}{N_{\text{nzr}}} + \dfrac{8}{N_{\text{nzc}}}}{2} \quad \text{[bytes per FLOP]},$$

where the numerator represents the memory traffic per nonzero element, and the division by two accounts for the two floating-point operations performed. The corresponding arithmetic intensity is given by

$$I_{\max} = \frac{1}{B_{c,\min}}.$$

For square matrices ($N_r = N_c$, $N_{\text{nzr}} = N_{\text{nzc}}$), this simplifies to

$$B_{c,\min} = 6 + \frac{10}{N_{\text{nzr}}}, \qquad I_{\max} = \frac{1}{6 + 10/N_{\text{nzr}}}.$$

Effect of right-hand side reuse

In practice, the access pattern to the vector x is irregular and depends on sparsity structure, cache size, and memory hierarchy. To account for this, an empirical factor $\alpha \geq 1/N_{\text{nzc}}$ is introduced to model the average number of times each entry of x is loaded from memory during one SpMV operation. Replacing the last term by 8α gives the general code balance:

$$B_c(\alpha) = \frac{12 + \dfrac{20}{N_{\text{nzr}}} + 8\alpha}{2}.$$

The limiting cases are:

- $\alpha = 0$: perfect reuse of x in cache,
- $\alpha = 1/N_{\text{nzc}}$: each entry of x is loaded exactly once,
- $\alpha = 1$: no reuse of x, i.e., each entry is loaded for every nonzero.

For typical finite element matrices with $N_{\text{nzr}} \simeq 50$, one obtains

$$B_c(1/N_{\text{nzr}}) \approx 6.28 \;\Rightarrow\; I_{\max} \approx 0.16 \text{ FLOP/byte},$$

$$B_c(1) \approx 10.2 \;\Rightarrow\; I_{\max} \approx 0.10 \text{ FLOP/byte}.$$

These values confirm that CSR SpMV lies deep within the memory-bound regime.

Extension to the BCSR(4×4) format

In the Block CSR format with 4×4 dense blocks, 16 coefficients share a single 4-byte column index. This reduces the indexing overhead and improves data locality without changing the arithmetic operation count. The code balance per scalar nonzero becomes

$$B_c^{\mathrm{BCSR}(4\times4)}(\alpha) = \frac{8.25 + \dfrac{20}{N_{\mathrm{nzr}}} + 8\alpha}{2},$$

where the 8.25 bytes account for 8 bytes of value storage and 0.25 bytes of averaged index storage per scalar. For $N_{\mathrm{nzr}} = 50$, the resulting intensities are

$$B_c^{\mathrm{BCSR}(4\times4)}(1/N_{\mathrm{nzr}}) \approx 4.4 \;\Rightarrow\; I_{\max} \approx 0.23 \text{ FLOP/byte},$$

$$B_c^{\mathrm{BCSR}(4\times4)}(1) \approx 8.3 \;\Rightarrow\; I_{\max} \approx 0.12 \text{ FLOP/byte}.$$

Hence, the block format improves arithmetic intensity by roughly 40–50% compared with CSR, primarily due to reduced index traffic and better spatial reuse of x.

Implications for optimization

The attainable performance of SpMV is given by the roofline limit

$$P_{\max} = \min\left(P_{\mathrm{peak}},\, B_{\max}/B_c(\alpha)\right),$$

where $B_{\max}$ is the sustainable memory bandwidth and P_{peak} the theoretical compute peak. Since $B_c(\alpha)$ dominates this relation, SpMV performance is governed almost entirely by memory traffic rather than floating-point throughput. Block storage, cache-friendly layouts, and vectorization thus target a single goal: reducing the effective code balance to increase arithmetic intensity.

2.4 Performance Models

The roofline model [8] provides a compact framework for assessing the performance limits of a computational kernel given its arithmetic intensity. It relates the sustained floating-point throughput P (in FLOP/s) to two hardware constraints: the theoretical peak compute performance P_{peak} and the sustainable memory bandwidth $B_{\max}$. The model defines the attainable performance as

$$P_{\max} = \min\left(P_{\mathrm{peak}},\, B_{\max} \times I\right),$$

where I is the arithmetic intensity, i.e. the number of floating-point operations performed per byte of data transferred between main memory and the processor.

Roofline estimation for CSR and BCSR kernels

From the code balance analysis in Section 2.3, the arithmetic intensities of the sparse matrix–vector product can be approximated by

$$I_{\mathrm{CSR}}^{\max} \simeq 0.16 \text{ FLOP/byte} \qquad I_{\mathrm{CSR}}^{\min} \simeq 0.10 \text{ FLOP/byte}$$

$$I_{\mathrm{BCSR}(4\times4)}^{\max} \simeq 0.23 \text{ FLOP/byte} \qquad I_{\mathrm{BCSR}(4\times4)}^{\min} \simeq 0.12 \text{ FLOP/byte}$$

corresponding respectively to the Compressed Sparse Row (CSR) and Block CSR (BCSR) formats for a 4×4 block structure. For a representative processor with a sustained bandwidth $B_{\max} = 50\,\text{GB/s}$ and a peak floating-point rate $P_{\text{peak}} = 500\,\text{GFLOP/s}$, the corresponding performance limits are

$$P_{\max}^{\text{CSR}} = 50{\times}10^9{\times}0.16 \approx 8.0\,\text{GFLOP/s}, \qquad P_{\max}^{\text{BCSR}} = 50{\times}10^9{\times}0.23 \approx 11.5\,\text{GFLOP/s}.$$

To account for varying degrees of cache reuse, a pessimistic case can also be derived by substituting the higher code balance $B_c(1)$ from Section 2.3, which represents negligible temporal reuse of the right-hand side vector. This yields the lower bounds

$$P_{\min}^{\text{CSR}} = 50{\times}10^9{\times}0.10 \approx 4.9\,\text{GFLOP/s}, \qquad P_{\min}^{\text{BCSR}} = 50{\times}10^9{\times}0.12 \approx 6.0\,\text{GFLOP/s},$$

defining a realistic performance envelope between the ideal in-cache and fully memory-bound regimes.

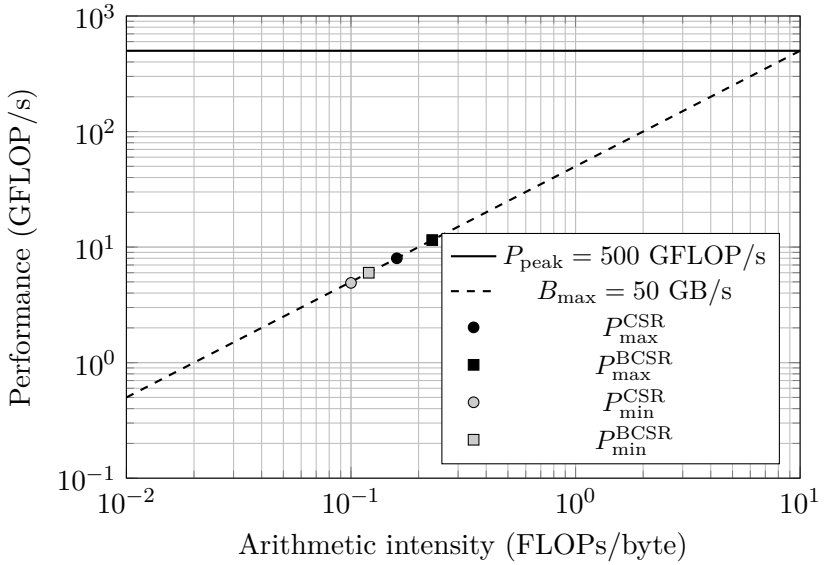

Figure 2.1: Roofline model for CSR and BCSR(4×4) SpMV kernels, showing upper and lower performance bounds.

Interpretation

Both kernels lie far below the compute-bound plateau, on the bandwidth-limited slope of the roofline. Their achievable performance is therefore determined primarily by memory traffic rather than by floating-point capability. The BCSR format achieves a higher arithmetic intensity by reducing index traffic and improving spatial and temporal locality, leading to a theoretical speedup of approximately 40–50% over CSR. The gap between $P_{\min}$ and $P_{\max}$ reflects the dependence of SpMV performance on data locality and cache reuse: cases with small working sets can approach the upper bound, while large out-of-cache problems approach the lower one. This quantitative framework highlights why optimizations targeting reduced data movement—such as block storage, cache blocking, or AVX2 vectorization—are essential to approach the architectural limits of memory-bound kernels.

Chapter 3

Sparse Matrix-Vector Product Implementation and Optimization

3.1 Baseline CSR Implementation

In the CSR format, the sparse matrix–vector product reduces to a sequence of row-wise dot products. For each row, one iterates over the stored nonzeros (nnz), multiplies them with the corresponding entries of the input vector, and accumulates the result in the output vector. The pseudocode is given in Algorithm 1.

Algorithm 1 SpMV in CSR format

```
1: for i = 0 to n − 1 do
2:     y[i] ← 0
3:     for k = row_ptr[i] to row_ptr[i + 1] − 1 do
4:         j ← col_ind[k]
5:         y[i] ← y[i] + values[k] · x[j]
6:     end for
7: end for
```

Complexity. The arithmetic cost is $\mathcal{O}(\text{nnz})$, where each nonzero contributes two floating-point operations. The kernel is thus optimal in terms of arithmetic work.

Performance considerations. The practical efficiency of CSR is limited by:

- **Irregular memory access:** indirection through `col_ind` leads to unpredictable access to $x[j]$, limiting cache reuse and hardware prefetching.
- **Indexing overhead:** each coefficient requires not only loading its value but also its column index, adding significant memory traffic relative to the arithmetic work.
- **Low vectorization potential:** the inner loop operates on scalars, making it difficult to map efficiently onto SIMD instructions.

These factors explain why CSR, despite its popularity, often achieves only a fraction of peak memory bandwidth on modern processors.

3.2 Baseline BCSR Implementation

The BCSR format extends the same principle but works with fixed-size dense blocks instead of individual coefficients. Each block encodes a small dense submatrix of size $b \times b$, multiplied by a contiguous subvector of the input. The algorithm is outlined in Algorithm 2.

Algorithm 2 SpMV in BCSR format ($b \times b$ blocks)

1: **for** $i_b = 0$ to $n_b - 1$ **do** ▷ $n_b = n/b$ block rows
2: $Y[i_b] \leftarrow 0$
3: **for** $k =$ `block_row_ptr`$[i_b]$ to `block_row_ptr`$[i_b + 1] - 1$ **do**
4: $j_b \leftarrow$ `block_col_ind`$[k]$
5: $Y[i_b] \leftarrow Y[i_b] +$ `block_values`$[k] \cdot X[j_b]$
6: **end for**
7: **end for**

Complexity. The overall cost remains $\mathcal{O}(\text{nnz})$, but each stored block is treated as a small dense kernel. This reduces the number of indices to be stored and allows multiple operations to be performed per memory access.

Performance considerations. BCSR offers several advantages over CSR:

- **Reduced indexing overhead:** one column index covers an entire block of b^2 coefficients.
- **Improved locality:** consecutive entries of x are reused within each block multiplication.
- **Vectorization potential:** the $b \times b$ dense multiplications can be efficiently mapped to SIMD units or BLAS-like microkernels.

However, BCSR is efficient only if the sparsity pattern exhibits a genuine block structure. When blocks contain many zeros, unnecessary storage and computations may offset the benefits. In practice, good performance is achieved when the block size matches the natural coupling of degrees of freedom, such as $b = 4$ in incompressible flow discretizations with velocity–pressure pairs by P1 elements.

Outlook. These two baseline implementations (CSR and BCSR) form the starting point for hardware-aware optimizations, including vectorization, loop unrolling, and specialized kernels, which are discussed in the following sections.

3.3 FMA

Modern processors implement fused multiply–add (FMA) instructions, which perform a multiplication and an addition in a single, fused operation:

$$y \leftarrow \texttt{fma}(a, b, c) = a \times b + c.$$

This operation has exactly the same form as the innermost update of the SpMV kernel, where each nonzero contributes

$$y[i] \leftarrow y[i] + \texttt{val} \times x[j].$$

Using FMA instead of two separate instructions (`mul` followed by `add`) provides several benefits:

- **Reduced instruction count:** one floating-point instruction replaces two, lowering pressure on the instruction pipeline.
- **Improved throughput:** modern vector units (e.g. AVX2 or AVX-512) can issue multiple FMA operations per cycle, increasing the effective flop rate of block kernels.
- **Numerical stability:** The fused multiply–add (FMA) instruction computes $a \times b + c$ as a single operation: the product $a \times b$ is evaluated in extended precision and the result of the addition is rounded only once. In contrast, when using separate multiply and add instructions, the product $a \times b$ is first rounded to machine precision, and the subsequent addition with c introduces a second rounding. By eliminating this intermediate rounding, FMA reduces the overall propagation of roundoff error and generally yields slightly more accurate results. In practice, however, our numerical experiments indicate that the difference remains at the level of machine precision, and thus does not affect solver convergence or physical accuracy.

It is important to note, however, that SpMV remains strongly memory-bound: its performance is limited by data movement rather than arithmetic throughput. Consequently, FMA alone does not fundamentally change the roofline bound. Nevertheless, when combined with vectorization inside block kernels (e.g. in BCSR implementations), FMA enables the processor to reach higher fractions of the bandwidth-limited performance ceiling.

The modified algorithms for CSR and BCSR using FMA are provided below with algorithms 3 and 4, where each scalar update

```
y[i] += val * x[j]
```

is replaced by the fused instruction

```
y[i] = fma(val, x[j], y[i]).
```

Algorithm 3 CSR SpMV using FMA

1: **for** $i = 0$ to $n - 1$ **do**
2: $y[i] \leftarrow 0$
3: **for** $k =$ `row_ptr`$[i]$ to `row_ptr`$[i + 1] - 1$ **do**
4: $j \leftarrow$ `col_ind`$[k]$
5: $y[i] \leftarrow$ `fma`(`values`$[k], x[j], y[i]$)
6: **end for**
7: **end for**

Algorithm 4 BCSR SpMV ($b \times b$ blocks) using FMA

for $i_b = 0$ to $n_b - 1$ **do** ▷ $n_b = n/b$ block rows
  for $k =$ `block_row_ptr`$[i_b]$ to `block_row_ptr`$[i_b + 1] - 1$ **do**
    $j_b \leftarrow$ `block_col_ind`$[k]$
    for $r = 0$ to $b - 1$ **do**
      for $c = 0$ to $b - 1$ **do**
        $y[i_b \cdot b + r] \leftarrow$ `fma`(`block_values`$[k][r][c], x[j_b \cdot b + c], y[i_b \cdot b + r]$)
      end for
    end for
  end for
end for

3.4 Vectorization with AVX2

Modern processors implement SIMD instruction sets that allow multiple data elements to be processed in parallel. The AVX2 extension, available on both recent Intel and AMD CPUs, provides 256-bit registers capable of holding four double-precision floating-point values or eight single-precision values. In the context of finite element flow computations, which rely on double precision (FP64), each FMA instruction processes four elements simultaneously, performing two operations per element. This corresponds to a throughput of up to eight floating-point operations per cycle per register.

3.4.1 Challenges in Sparse Vectorization

Sparse matrix–vector multiplication (SpMV) poses inherent challenges for SIMD vectorization. In the CSR format, the locations of nonzeros vary irregularly across rows and are accessed through indirect indexing. This irregularity prevents efficient use of aligned SIMD loads, leading to frequent cache misses and reduced throughput. Moreover, the arithmetic intensity of SpMV is inherently low, so even when vectorization succeeds, performance is limited by memory bandwidth rather than floating-point throughput.

3.4.2 Block Formats as Enablers

Block storage formats such as BCSR mitigate these issues by grouping nonzeros into small dense submatrices. In our case, we adopt a 4×4 block size, corresponding naturally to the grouping (u_x, u_y, u_z, p) at each finite element node. Coefficients inside a block are stored contiguously, which allows loading full rows or columns into SIMD registers. This structured memory layout enables effective AVX2 vectorization and increases reuse of both coefficients and input vector entries.

3.4.3 AVX2 Kernel for BCSR

We implemented a hand-optimized kernel tailored to the 4×4 BCSR format. Each block contains 16 coefficients arranged contiguously. During multiplication, one four-entry subvector of x is broadcast into SIMD registers and multiplied with the corresponding block column. The results are accumulated into the output vector

using fused multiply–add (`_mm256_fmadd_pd`), which reduces instruction count and latency. The kernel is fully unrolled at compile time, avoiding loop overhead and ensuring maximum SIMD utilization. The following schematic (Figure 3.1) and the pseudocode (Algorithm 5) below illustrate this approach. The full implementation of the AVX2 kernel is provided in Appendix 6.1

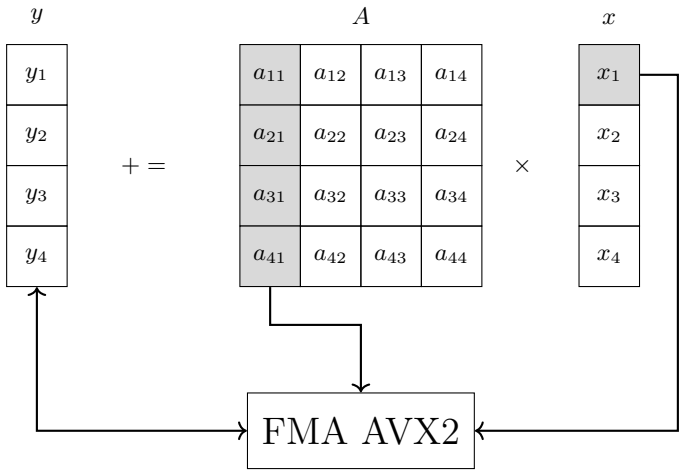

FIGURE 3.1: Schematic illustration of one fused multiply–add: the first column of A is multiplied by the corresponding entry of x and accumulated into y.

Algorithm 5 AVX2-optimized SpMV for 4×4 BCSR blocks

for $i_b = 0$ to $n_b - 1$ **do** ▷ n_b = number of block rows
  $acc \leftarrow (0, 0, 0, 0)$ ▷ initialize SIMD accumulator
  for $k =$ `ptrow`$[i_b]$ to `ptrow`$[i_b + 1] - 1$ **do**
    $j_b \leftarrow$ `indcol`$[k]$
    for $c = 0$ to 3 **do** ▷ iterate over block columns
      $a_c \leftarrow$ column c of block `coef`$[k]$
      $x_c \leftarrow x[j_b \cdot 4 + c]$ broadcast in SIMD register
      $acc \leftarrow$ `fma`(a_c, x_c, acc)
    end for
  end for
  $y[i_b \cdot 4 : i_b \cdot 4 + 3] \leftarrow acc$
end for

3.4.4 Precision Considerations

All optimizations in this work target **FP64 (double precision)** arithmetic. This choice is dictated by the finite element discretization of incompressible Navier–Stokes equations: flow problems at moderate to high Reynolds numbers are sensitive to round-off, and iterative solvers may require the accuracy and stability of FP64 to converge robustly. While AVX2 also supports FP32 (single precision), FEM-based CFD solvers rarely operate in FP32 due to loss of accuracy in pressure–velocity coupling. This focus on FP64 impacts both the vectorization strategy and the achievable peak performance, as a 256-bit AVX2 register can hold 4 doubles versus 8 floats.

3.5 SpMV Benchmarking

3.5.1 Hardware platforms

To evaluate the performance of the proposed SpMV kernels, we conducted benchmarks on two representative server-class CPUs, both available on the Grid'5000 experimental testbed:

Table 3.1: Hardware characteristics of the benchmark platforms (Grid'5000 testbed).

	AMD EPYC 7452	**Intel Xeon Gold 5220**
Microarchitecture	Zen 2	Cascade Lake-SP
Base frequency	2.35 GHz	2.20 GHz
Boost frequency	-	3.90 GHz
Cores per socket	32	18
Total cores	64 (2 sockets)	18 (1 socket)
SIMD support	AVX2 (256-bit)	AVX-512 (512-bit)
L1d cache	32 KB per core	32 KB per core
L1i cache	32 KB per core	32 KB per core
L2 cache	512 KB per core	1 MB per core
L3 cache	128 MB per socket	24.75 MB shared
Memory channels	8-channel DDR4	6-channel DDR4
Peak bandwidth	∼204 GB/s	∼128 GB/s

These two architectures differ in SIMD width, memory hierarchy, and core count, thus providing complementary perspectives on the behavior of memory-bound kernels such as SpMV. All experiments were run in single-threaded mode to focus on kernel efficiency rather than parallel scalability.

3.5.2 Test matrices

We benchmarked six matrices obtained from finite element discretizations of the stationary Stokes problem using `FreeFEM` [9]. The sequence corresponds to increasingly fine meshes, yielding matrices of growing size and sparsity complexity. This ensures that the benchmark set is representative of real-world PDE solvers rather than synthetic test cases.

Table 3.2: Characteristics of the benchmark matrices

Matrix	**Rows**	**nnz**	**Avg. nnz/row**
$\mathcal{A}_1$	268	11,856	44.3
$\mathcal{A}_2$	1,936	91,296	47.2
$\mathcal{A}_3$	6,232	317,888	51.0
$\mathcal{A}_4$	10,084	520,688	51.6
$\mathcal{A}_5$	35,928	1,935,776	53.9
$\mathcal{A}_6$	121,480	6,657,600	54.8

While the average number of nonzeros per row remains roughly constant across the sequence, the structure of the sparsity pattern evolves with mesh refinement. In the smaller problems ($\mathcal{A}_1$–$\mathcal{A}_2$), the matrices are relatively narrow banded, with most nonzeros concentrated around the diagonal due to local finite element couplings. As the mesh is refined ($\mathcal{A}_3$–$\mathcal{A}_6$), the bandwidth grows and the nonzero distribution becomes more irregular, increasing the pressure on memory hierarchy and reducing cache reuse. This evolution highlights the importance of testing across different sizes: although the arithmetic intensity (NNZ/row) changes little, the access locality and indirect addressing cost vary significantly with problem scale.

3.5.3 Methodology

For each matrix and each kernel variant (CSR, BCSR, FMA, AVX), we measured the average execution time of the SpMV over 100 iterations. Timing was performed directly in C++ using the `std::chrono` high-resolution clock, with the duration reported in microseconds. This approach ensures a precise and lightweight measurement of kernel execution time without additional profiling overhead.

To reduce variability, the measurements were repeated several times, and the reported results correspond to the average runtime. The next subsection presents performance results in terms of both execution time and the effective memory bandwidth achieved, estimated from the total bytes accessed and the measured execution time.

3.5.4 Scalar FMA: Compiler vs Manual

Before introducing AVX2 vectorization, we investigated whether explicitly using scalar FMA instructions provides a performance benefit compared to relying on compiler optimizations.

To that end, we compared:

- `CSR OPT` / `BCSR OPT`: using regular scalar code, compiled with `-mfma` flag (letting the compiler insert `fma()` where applicable),
- `CSR FMA` / `BCSR FMA`: same logic but manually written using scalar `fma()` calls.

As shown in Table 3.3, both variants exhibit nearly identical performance, with differences within 1–2%. We therefore retain only the manually written FMA versions in the rest of the benchmark for clarity.

Table 3.3: Scalar FMA comparison on AMD EPYC (execution time in μs)

Matrix	CSR OPT	CSR FMA	BCSR OPT	BCSR FMA
$\mathcal{A}_1$	38	39	6	7
$\mathcal{A}_2$	295	297	49	49
$\mathcal{A}_3$	1025	1029	176	175
$\mathcal{A}_4$	1666	1729	289	288
$\mathcal{A}_5$	6390	6259	1187	1167
$\mathcal{A}_6$	22110	22393	4943	4974

3.5.5 Implementations compared

We consider several variants of the SpMV kernel in order to evaluate the effect of storage format and low-level optimizations. The reference is the baseline CSR implementation, against which all other results are compared. A CSR variant using FMA instructions is included to quantify the effect of instruction-level optimizations without altering the storage format. To assess the benefits of block storage, we implement a scalar BCSR kernel with 4×4 blocks, together with an FMA-enhanced version. Finally, a hand-optimized BCSR kernel exploiting AVX2 vectorization is evaluated, representing the most advanced implementation in this study.

3.5.6 Evaluation metrics

Performance is assessed using a combination of accuracy and efficiency indicators. Execution time, averaged over 100 iterations, provides the primary measure of computational cost. From these timings, we compute the relative speedup with respect to the baseline CSR kernel. Numerical fidelity is verified by comparing each implementation to a trusted reference, through the relative ℓ^2 error

$$\varepsilon = \frac{\|y_{\text{ref}} - y_{\text{impl}}\|_2}{\|y_{\text{ref}}\|_2}.$$

Finally, the effective throughput is expressed in gigaflops per second (GFLOPs/s), calculated as

$$\text{GFLOPs} = \frac{2 \cdot \text{nnz}}{10^3 \cdot \text{Elapsed time } (\mu\text{s})},$$

where NNZ denotes the number of nonzero entries of the matrix. This metric highlights the gap between the measured performance and the theoretical roofline bounds discussed in Section 2.4.

3.5.7 Objectives

The benchmark campaign pursues three main objectives. First, it seeks to quantify the impact of format switching, by comparing CSR with its block counterpart BCSR. Second, it evaluates the benefits of hardware-aware optimizations, in particular FMA and AVX2 vectorization, across distinct processor architectures. Third, it investigates the scaling of performance with matrix size and sparsity pattern, thereby providing insight into the portability of optimizations across problem classes. Together, these results allow us to highlight architectural differences between AMD EPYC and Intel Xeon CPUs when executing memory-bound kernels such as SpMV.

3.6 Results

This section presents the outcome of the SpMV benchmark campaign described in Section 3.5. We analyze execution times, achieved floating-point throughput, and numerical accuracy for the different kernel implementations, and discuss the impact of storage format, FMA optimizations, and AVX2 vectorization on two distinct hardware platforms.

3.6.1 Execution time

Tables 3.4 and 3.5 report the average execution time (in microseconds) on AMD EPYC and Intel Xeon, respectively. For each problem, the fastest kernel is highlighted in bold. The data show a consistent hierarchy across all test cases: CSR is always the slowest, while the AVX2-optimized BCSR kernel achieves the best performance.

Table 3.4: Execution time on AMD EPYC 7452 (in μs). Fastest kernel in bold.

Matrix	CSR	CSR FMA	BCSR	BCSR FMA	BCSR AVX2
$\mathcal{A}_1$	56	38	7	6	**3**
$\mathcal{A}_2$	441	295	61	49	**27**
$\mathcal{A}_3$	1524	1023	218	176	**101**
$\mathcal{A}_4$	2489	1667	360	287	**167**
$\mathcal{A}_5$	9307	6198	1472	1181	**782**
$\mathcal{A}_6$	32297	21603	6078	4930	**3865**

Table 3.5: Execution time on Intel Xeon Gold 5220 (in μs). Fastest kernel in bold.

Matrix	CSR	CSR FMA	BCSR	BCSR FMA	BCSR AVX2
$\mathcal{A}_1$	33	26	5	5	**3**
$\mathcal{A}_2$	262	207	49	48	**27**
$\mathcal{A}_3$	909	718	175	163	**117**
$\mathcal{A}_4$	1463	1182	294	274	**195**
$\mathcal{A}_5$	5589	4520	1226	1178	**801**
$\mathcal{A}_6$	22935	19451	7617	7590	**5495**

3.6.2 Relative speedup

Figures 3.2 and 3.3 illustrate the speedup relative to the CSR baseline. On both platforms, moving from CSR to BCSR already brings speedups of one order of magnitude on large matrices. Adding FMA instructions provides a moderate additional gain, whereas AVX2 vectorization produces the largest improvement, with accelerations of up to 18×.

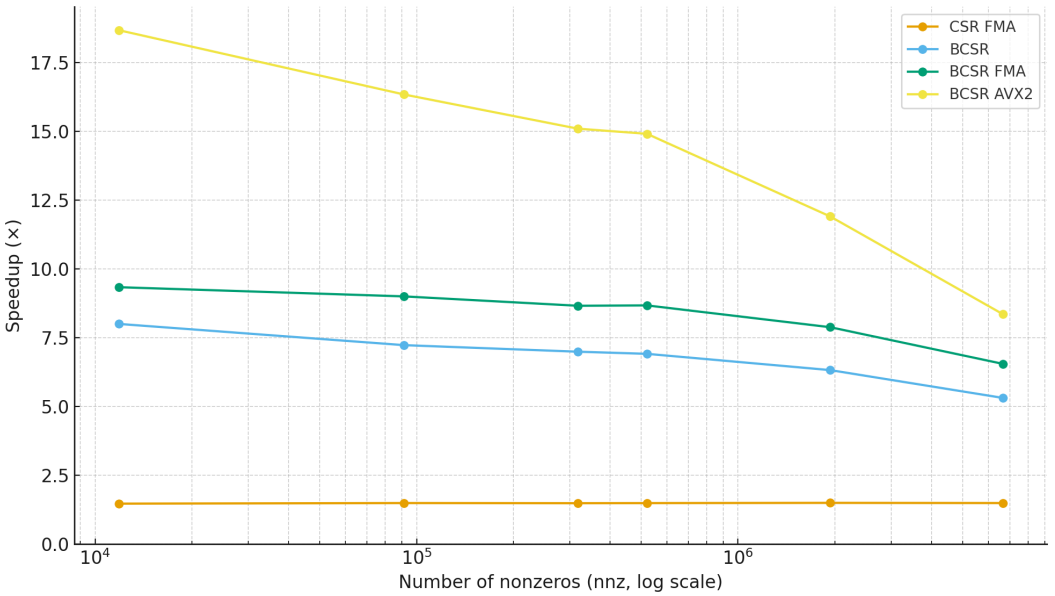

Figure 3.2: Speedup relative to CSR on AMD EPYC.

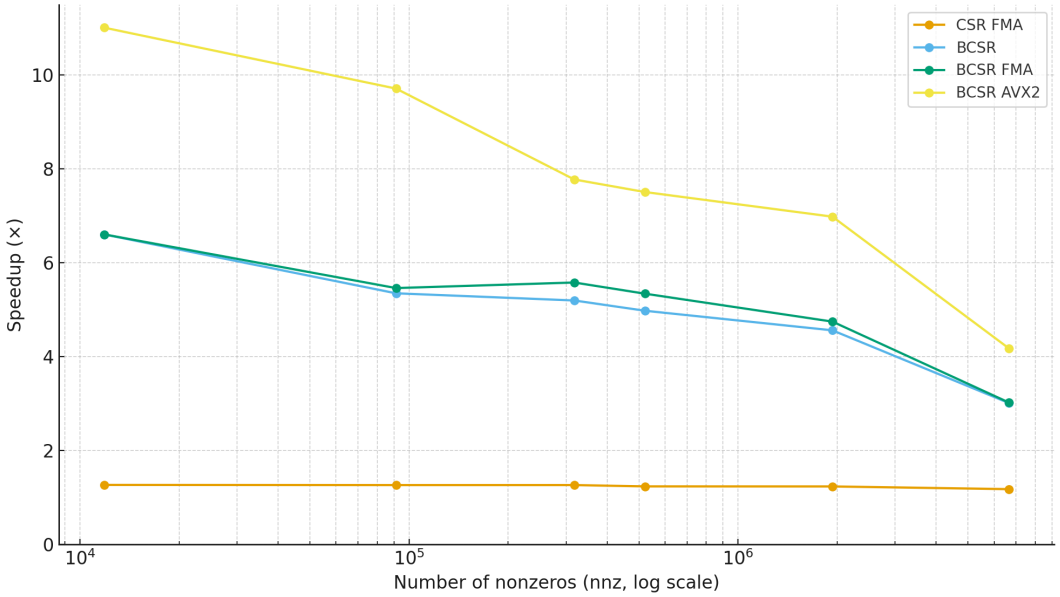

Figure 3.3: Speedup relative to CSR on Intel Xeon.

3.6.3 Achieved GFLOPs

Tables 3.6 and 3.7 report the effective throughput in GFLOPs. On both CPUs, CSR reaches less than 1 GFLOP/s, confirming the memory-bound nature of the kernel. The BCSR AVX2 implementation, by contrast, achieves up to 8 GFLOPs on small matrices and maintains over 2 GFLOPs even for the largest case $\mathcal{A}_6$.

Table 3.6: Achieved GFLOPs on AMD EPYC. Best performance in bold.

Matrix	CSR	CSR FMA	BCSR	BCSR FMA	BCSR AVX2
$\mathcal{A}_1$	0.42	0.62	3.39	3.95	**7.90**
$\mathcal{A}_2$	0.41	0.62	2.99	3.73	**6.76**
$\mathcal{A}_3$	0.42	0.62	2.92	3.61	**6.30**
$\mathcal{A}_4$	0.42	0.62	2.89	3.63	**6.24**
$\mathcal{A}_5$	0.42	0.62	2.63	3.28	**4.95**
$\mathcal{A}_6$	0.41	0.62	1.75	2.70	**3.45**

Table 3.7: Achieved GFLOPs on Intel Xeon. Best performance in bold.

Matrix	CSR	CSR FMA	BCSR	BCSR FMA	BCSR AVX2
$\mathcal{A}_1$	0.72	0.91	4.74	4.74	**7.90**
$\mathcal{A}_2$	0.70	0.88	3.73	3.80	**6.76**
$\mathcal{A}_3$	0.70	0.89	3.63	3.90	**5.43**
$\mathcal{A}_4$	0.71	0.88	3.54	3.80	**5.34**
$\mathcal{A}_5$	0.69	0.86	3.16	3.29	**4.83**
$\mathcal{A}_6$	0.58	0.68	1.75	1.75	**2.42**

3.6.4 Numerical accuracy

To verify correctness, we compared each optimized kernel against the CSR reference result in double precision. The maximum relative error across all test cases remained below 10^{-17} (Table 3.8), which is consistent with round-off differences in floating-point accumulation. Thus, all implementations can be considered numerically equivalent.

Table 3.8: Maximum relative error compared to CSR reference (AMD EPYC).

Implementation	CSR FMA	BCSR	BCSR FMA	BCSR AVX2
Max error ε	10^{-18}	$< 10^{-17}$	$< 10^{-17}$	$< 10^{-17}$

3.6.5 Discussion

Switching from CSR to BCSR yields substantial acceleration, primarily due to reduced indexing overhead and improved data locality. The benefits are particularly pronounced on larger matrices, where block-based storage enhances cache reuse. Adding scalar FMA instructions leads to moderate but consistent improvements (about 30%), confirming that the computation is limited by memory traffic rather than instruction count. The largest gains stem from AVX2 vectorization, which exploits the block structure to achieve efficient SIMD execution. Here, the combination of wide registers, fused operations, and cache-friendly access patterns results in speedups of up to 18× compared to CSR.

Comparing hardware platforms, both EPYC and Xeon benefit from the same hierarchy of optimizations, yet their architectural trade-offs result in distinct performance signatures. On large matrices, the AMD EPYC consistently sustains higher throughput owing to its eight-channel memory subsystem and NUMA-aware cache hierarchy, which together provide greater effective bandwidth in the memory-bound regime of SpMV. Conversely, the Intel Xeon achieves slightly higher nominal GFLOP/s on small problem sizes, primarily due to its more aggressive floating-point pipeline and larger unified L3 cache, allowing data to reside entirely in cache. However, this advantage diminishes rapidly once the working set exceeds cache capacity: Xeon's six-channel memory subsystem limits bandwidth and causes execution times to scale less favorably with problem size. Overall, EPYC delivers superior sustained performance in the large-scale regime relevant to PDE solvers, confirming the dominant influence of memory bandwidth over raw compute throughput for SpMV kernels.

It should be noted that the measured performance in GFLOP/s lies well below the roofline predictions discussed in Section 2.4. This discrepancy arises from the fact that all benchmarks were executed in **single-core** mode. In this configuration, the effective memory bandwidth available to the kernel is only a fraction of the socket-level peak: approximately $3 - 6$ GB/s on AMD EPYC and $2.5 - 4.5$ GB/s on Intel Xeon, compared to their theoretical maxima of 204 GB/s and 128 GB/s, respectively. Consequently, the attainable throughput is bounded by these reduced per-core bandwidths rather than by the sustainable socket bandwidth assumed in the roofline model. When this correction is applied, the theoretical bounds become consistent with the measured results, confirming that SpMV remains limited by memory traffic even at the single-core level.

Overall, the results demonstrate that block-based formats combined with architecture-aware optimizations such as AVX2 vectorization provide substantial gains in SpMV performance. These optimized kernels form a solid foundation for accelerating Krylov solvers in large-scale CFD simulations.

Chapter 4

Navier-Stokes Equations

4.1 Incompressible Navier-Stokes

The incompressible Navier–Stokes equations describe the motion of fluids such as water and air in regimes where density variations are negligible. They combine two fundamental physical principles: conservation of momentum, which reflects Newton's second law applied to fluid parcels, and conservation of mass, which enforces incompressibility by requiring that the velocity field remains divergence-free.

These equations play a central role in CFD and are widely used to model a broad range of phenomena, from engineering applications such as aerodynamics and turbomachinery to geophysical flows in oceans and the atmosphere. Their complexity arises from the nonlinear coupling between velocity and pressure, as well as the large scale of the systems generated after spatial and temporal discretization.

Solving the incompressible Navier–Stokes equations numerically is therefore a challenging task. Implicit formulations are often favored for time-dependent problems because they allow larger time steps and improved stability, but they also lead to large nonlinear systems of equations at each time step. Efficient iterative solvers and preconditioners are thus essential to obtain practical runtimes, especially for three-dimensional simulations and high Reynolds numbers.

4.2 Mathematical model

4.2.1 Strong formulation

We consider the unsteady incompressible Navier–Stokes system in a bounded domain $\Omega \subset \mathbb{R}^3$ over a time horizon $[0, T]$. The unknowns are the velocity field $u = (u_x, u_y, u_z)$ and the pressure p, governed by

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u - \nu \Delta u + \nabla p = f \quad \text{in } \Omega \times (0, T], \tag{4.1}$$

$$\nabla \cdot u = 0 \quad \text{in } \Omega \times (0, T]. \tag{4.2}$$

Equation (4.1) expresses momentum conservation: the transient term $\partial_t u$ accounts for temporal variations, the convective term $(u \cdot \nabla)u$ represents nonlinear advection, the viscous diffusion $-\nu\Delta u$ models internal friction, and the pressure gradient ∇p enforces incompressibility. The source term f accounts for external forcing. Equation (4.2) expresses the incompressibility constraint.

The system is completed with an initial condition

$$u(x, 0) = u_0(x) \quad \text{in } \Omega,$$

where u_0 is chosen divergence-free, and with boundary conditions specified on distinct parts of the boundary:

$$\partial\Omega = \Gamma_{\text{in}} \cup \Gamma_{\text{wall}} \cup \Gamma_{\text{out}}, \quad \Gamma_{\text{in}} \cap \Gamma_{\text{wall}} \cap \Gamma_{\text{out}} = \emptyset.$$

On these subsets we impose

$$\begin{aligned} u &= g_{\text{in}} && \text{on } \Gamma_{\text{in}} && \text{(prescribed inflow profile)}, \\ u &= 0 && \text{on } \Gamma_{\text{wall}} && \text{(no-slip walls)}, \\ (-pI + \nu\nabla u)n &= 0 && \text{on } \Gamma_{\text{out}} && \text{(natural outflow)}. \end{aligned}$$

4.2.2 Weak formulation

Let $V = \{\, u \in H^1(\Omega)^3 \;;\; u = 0 \text{ on } \Gamma_{\text{in}} \cup \Gamma_{\text{wall}} \,\}$ denote the velocity space and $Q = L^2(\Omega)$ the space of square-integrable pressure fields. The weak form of (4.1)–(4.2) consists in finding $(u(t), p(t)) \in V \times Q$ such that, for all $(v, q) \in V \times Q$,

$$\begin{aligned} \int_\Omega \frac{\partial u}{\partial t} \cdot v + \int_\Omega (u \cdot \nabla)u \cdot v + \nu \int_\Omega \nabla u : \nabla v \\ - \int_\Omega p\, \nabla \cdot v + \int_\Omega q\, \nabla \cdot u = \int_\Omega f \cdot v. \end{aligned} \tag{4.3}$$

This formulation identifies the canonical operators of incompressible flow: the mass term $(\partial_t u, v)$, the convection form $c(u; u, v) = \int_\Omega (u \cdot \nabla)u \cdot v$, the bilinear diffusion form $a(u, v) = \nu \int_\Omega \nabla u : \nabla v$, and the pressure–velocity coupling $b(v, p) = -\int_\Omega p\, \nabla \cdot v$ together with the incompressibility constraint $b(u, q) = \int_\Omega q\, \nabla \cdot u$.

Equation (4.3) thus provides the variational framework upon which the finite element discretization will be constructed (Section 4.3). This weak setting is essential both for theoretical analysis (existence, stability) and for numerical implementation in conforming finite element spaces.

4.3 Discretization

4.3.1 Spatial discretization

The computational domain Ω is discretized by a conforming tetrahedral mesh $\mathcal{T}_h$ of characteristic size h. Velocity and pressure fields are approximated in continuous piecewise linear spaces,

$$\mathbb{V}_h = \{v \in [C^0(\Omega)]^3 : v|_K \in [\mathbb{P}_1(K)]^3,\ \forall K \in \mathcal{T}_h\},$$

$$\mathbb{Q}_h = \{q \in C^0(\Omega) : q|_K \in \mathbb{P}_1(K),\ \forall K \in \mathcal{T}_h\}.$$

Since the equal-order P_1/P_1 velocity–pressure pair does not satisfy the discrete inf–sup (or Ladyženskaja–Babuška–Brezzi, LBB) condition [10], a stabilization mechanism is required. In this work, we employ the pressure-stabilizing/Petrov–Galerkin (PSPG) formulation introduced by Hughes et al. [11], which adds a consistent pressure-gradient stabilization term to the variational problem.

$$d(p,q) = \delta \sum_{K\in\mathcal{T}_h} h_K^2 \int_K \nabla p \cdot \nabla q,$$

with stabilization parameter $\delta > 0$ and element size h_K.

Let $\{\varphi_i\}$ and $\{\psi_\ell\}$ denote the nodal bases for velocity and pressure, respectively. The discretization yields the canonical matrices associated with the weak formulation:

$$\begin{aligned}
M_{ij} &= \int_\Omega \varphi_j \cdot \varphi_i, && \text{(mass matrix)},\\
(A_0)_{ij} &= \nu \int_\Omega \nabla\varphi_j : \nabla\varphi_i, && \text{(diffusion)},\\
(A_1(u) + A_2(u))_{ij} &= \int_\Omega (u\cdot\nabla\varphi_j)\cdot\varphi_i + \int_\Omega (\varphi_j\cdot\nabla u)\cdot\varphi_i, && \text{(linearized convection)},\\
-B_{\ell j} &= \int_\Omega (\nabla\cdot\varphi_j)\,\psi_\ell, && \text{(divergence)},\\
D_{\ell m} &= \delta \sum_{K\in\mathcal{T}_h} h_K^2 \int_K \nabla\psi_m \cdot \nabla\psi_\ell, && \text{(stabilization)}.
\end{aligned}$$

Assembly procedure. The discrete operators are constructed by assembling element contributions over all tetrahedra $K \in \mathcal{T}_h$. For each element, the gradients of the barycentric basis functions are computed explicitly, together with the element volume $|K|$ and characteristic size h_K. These quantities are then used to build the local matrices:

- the velocity mass matrix $M^K \in \mathbb{R}^{12\times 12}$ (block-diagonal in components),
- the viscous stiffness (diffusion) matrix $A_0^K \in \mathbb{R}^{12\times 12}$,
- the linearized convection matrices A_1^K, $A_2^K \in \mathbb{R}^{12\times 12}$,
- the discrete divergence matrix $B^K \in \mathbb{R}^{4\times 12}$,
- the stabilization matrix $D^K \in \mathbb{R}^{4\times 4}$.

Local matrices are then mapped to the global system using the local-to-global indexing of velocity and pressure unknowns. Specifically, each tetrahedron contributes to the global block system

$$\begin{bmatrix} \frac{1}{\Delta t}M + A_0 + A_1(u) + A_2(u) & B^\top \\ -B & D \end{bmatrix},$$

with the velocity indices distributed as $(3i, 3i+1, 3i+2)$ for each node i and the pressure index corresponding to the nodal numbering of $\mathcal{T}_h$.

Remark. Because the stabilized P_1/P_1 discretization associates four unknowns (u_x, u_y, u_z, p) with each mesh node, the finite element system naturally exhibits a block-by-node organization. Figure 4.1 illustrates a representative tetrahedral element, where each vertex carries these four local degrees of freedom corresponding to the three velocity components and pressure. This layout leads to a global stiffness matrix with an intrinsic 4×4 block structure, as shown in Figure 4.2. All coefficients within these blocks are nonzero, eliminating the need for artificial zero-padding. The block structure is therefore ideally suited to the BCSR storage format, which stores each dense 4×4 submatrix contiguously in memory. Such organization not only reduces indexing overhead but also enhances cache efficiency and enables effective exploitation of SIMD instructions in the custom AVX2 kernels described in Section 3.6.

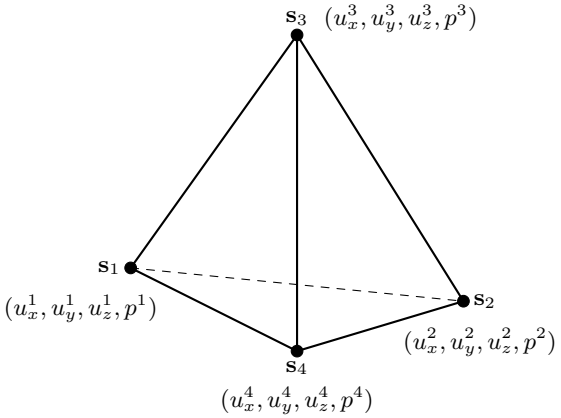

Figure 4.1: Tetrahedral finite element with 4 degrees of freedom per node

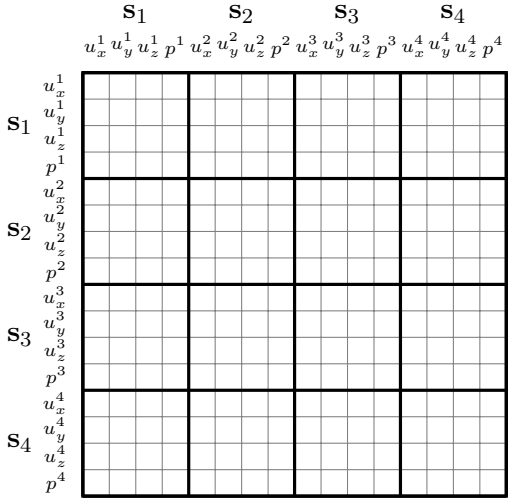

Figure 4.2: Corresponding 4×4 block-by-node structure

4.3.2 Temporal discretization

Time advancement relies on the fully implicit Backward Euler scheme. Given (u^n, p^n) at time t^n, the unknowns (u^{n+1}, p^{n+1}) at $t^{n+1} = t^n + \Delta t$ satisfy

$$\frac{1}{\Delta t} M(u^{n+1} - u^n) + A_0 u^{n+1} + (A_1(u^{n+1}) + A_2(u^{n+1}))u^{n+1} + B^\top p^{n+1} = f_u,$$

$$-Bu^{n+1} + Dp^{n+1} = f_p.$$

4.3.3 Nonlinear discrete system

The complete discrete problem at each time step is thus expressed as the nonlinear system

$$F(u^{n+1}, p^{n+1}) = \begin{bmatrix} \frac{1}{\Delta t} M(u^{n+1} - u^n) + A_0 u^{n+1} + (A_1(u^{n+1}) + A_2(u^{n+1}))u^{n+1} + B^\top p^{n+1} - f_u \\ -Bu^{n+1} + Dp^{n+1} - f_p \end{bmatrix} = 0.$$

This large-scale algebraic system couples velocity and pressure unknowns and exhibits the characteristic saddle-point structure of incompressible flow discretizations. Its sparse and nonlinear nature necessitates the use of iterative solvers and suitable preconditioners, which will be discussed in Section 4.4.

4.4 Numerical resolution

4.4.1 Newton's method

At each time step, the fully discrete Navier–Stokes problem leads to the nonlinear algebraic system

$$F(u^{n+1}, p^{n+1}) = 0,$$

which is solved by Newton's method. Starting from an initial guess $(u^{n,0}, p^{n,0})$, the Newton update consists in solving at iteration k the linearized system

$$J(u^{n,k}, p^{n,k}) \begin{pmatrix} \delta u \\ \delta p \end{pmatrix} = -F(u^{n,k}, p^{n,k}), \qquad (u^{n,k+1}, p^{n,k+1}) = (u^{n,k}, p^{n,k}) + (\delta u, \delta p).$$

The Jacobian exhibits the saddle-point block structure

$$J(u^{n,k}, p^{n,k}) = \begin{bmatrix} \frac{1}{\Delta t} M + A_0 + A_1(u^{n,k}) + A_2(u^{n,k}) & B^\top \\ -B & D \end{bmatrix},$$

where the convection matrices A_1, A_2 depend nonlinearly on the current iterate.

Convergence is declared once the relative nonlinear residual falls below the tolerance $\varepsilon = 10^{-6}$:

$$\frac{\|F(u^{n,k}, p^{n,k})\|_2}{\|F(u^{n,0}, p^{n,0})\|_2} < \varepsilon.$$

4.4.2 Imposition of boundary conditions.

After assembling the element contributions, essential (Dirichlet) boundary conditions are enforced by modifying the global Jacobian and residual. This is achieved in PETSc by calling `MatZeroRows`, which zeros out the rows corresponding to prescribed degrees of freedom and inserts the identity on the diagonal. This approach ensures that the solution components are fixed to their boundary values while preserving the sparse block structure of the matrix. In practice, this operation is performed at each Newton step, after adding the nonlinear Jacobian contributions and before passing the system to the Krylov solver.

4.4.3 Linear solver

The Newton correction system is large, sparse, and non-symmetric. Direct solvers are impractical in three dimensions, and iterative Krylov methods combined with preconditioning are therefore the method of choice. In this work, we employ the Generalized Minimal Residual method (GMRES) preconditioned by an incomplete LU (ILU) factorization.

GMRES

GMRES constructs approximations x_m in the affine space $x_0 + \mathcal{K}_m(A, r_0)$, where

$$\mathcal{K}_m(A, r_0) = \text{span}\{r_0, Ar_0, \ldots, A^{m-1}r_0\}, \qquad r_0 = b - Ax_0,$$

by minimizing the residual norm $\|b - Ax\|_2$ over this subspace. The Arnoldi process generates an orthonormal basis of $\mathcal{K}_m$ and a Hessenberg matrix H_m, reducing the problem to a small least-squares minimization.

A well-known difficulty is that the dimension of $\mathcal{K}_m$ grows linearly with the iteration count, increasing both storage and orthogonalization costs. To control this, restarted variants GMRES(m) are typically employed: the algorithm runs for m iterations, produces x_m, and then restarts with x_m as initial guess. Larger restart lengths improve robustness but increase per-cycle cost, while smaller values reduce memory but may deteriorate convergence.

For clarity, Algorithm 6 summarizes one cycle of restarted GMRES with right preconditioning.

Algorithm 6 One cycle of right-preconditioned GMRES(m)

Require: Matrix A, preconditioner M^{-1}, right-hand side b, initial guess x_0, restart length m
Ensure: Updated iterate x_m and residual norm β

$r_0 \leftarrow b - Ax_0$, $\beta \leftarrow \|r_0\|_2$
$v_1 \leftarrow r_0/\beta$ ▷ initialize Krylov basis
for $j = 1, \dots, m$ **do**
  $z_j \leftarrow M^{-1} v_j$ ▷ apply preconditioner
  $w \leftarrow A z_j$ ▷ matrix-vector product
  for $i = 1, \dots, j$ **do**
    $h_{i,j} \leftarrow \langle w, v_i \rangle$
    $w \leftarrow w - h_{i,j} v_i$ ▷ modified Gram–Schmidt
  end for
  $h_{j+1,j} \leftarrow \|w\|_2$, $v_{j+1} \leftarrow w/h_{j+1,j}$
  Apply Givens rotations to column j of H ▷ update QR factorization
  Update residual estimate β_j
  if $\beta_j < \varepsilon$ **then break**
  end if
end for
Solve the least-squares problem $\min_y \|\beta e_1 - Hy\|_2$
$x_m \leftarrow x_0 + Zy$, where $Z = [z_1, \dots, z_m]$

ILU preconditioning

Preconditioning is indispensable for saddle-point systems. The preconditioned system reads

$$M^{-1} A x = M^{-1} b,$$

where M approximates A but is easier to invert. Incomplete LU factorization (ILU) is a classical and effective choice: one computes L and U such that $A \approx LU$, but restricts the amount of fill-in beyond the sparsity of A. The level-of-fill parameter controls this trade-off. ILU(0) preserves the original sparsity and is extremely cheap, but may yield poor convergence. Higher levels (ILU(1), ILU(2), ...) progressively enrich the factors and provide stronger preconditioners at the expense of denser factors and costlier solves.

As an illustration, for a tridiagonal matrix the exact LU introduces fill-in outside the band. ILU(0) forbids these entries, while ILU(1) retains them, improving the quality of the preconditioner. In large-scale PDE systems, ILU(k) with $k = 2$ or 3 is often required to ensure acceptable convergence rates.

Application of an ILU preconditioner reduces to two sparse triangular solves, summarized in Algorithm 7.

Algorithm 7 Application of ILU preconditioner

Require: ILU factors L, U, input vector w
Ensure: $z \approx M^{-1}w$
1: Solve $Ly = w$ by forward substitution
2: Solve $Uz = y$ by back substitution
3: **return** z

Preconditioned GMRES for Navier–Stokes

Combining GMRES with an incomplete LU factorization (ILU) provides a robust and practical solver for the Jacobian systems arising in the Newton iterations. GMRES ensures numerical stability for the non-symmetric linearized operators, while ILU effectively clusters eigenvalues and improves the conditioning, thereby reducing the number of Krylov iterations required for convergence.

This strategy is made feasible by the use of the stabilized finite element formulation: the inclusion of the pressure-stabilizing/Petrov–Galerkin (PSPG) term regularizes the saddle-point system, enabling a consistent LU-type factorization of the global Jacobian while preserving its approximation properties. Without such stabilization, the block saddle-point structure would render direct factorizations ill-posed, and field-splitting preconditioners—based on nested Schur complement operations—are typically required instead.

In this stabilized context, ILU(k) thus provides a simple yet effective preconditioner for large-scale incompressible flow problems, combining algorithmic robustness with straightforward implementation in PETSc.

4.5 Performance Optimization

4.5.1 Overview

The dominant costs in each GMRES iteration stem from two operations: (i) the sparse matrix–vector product (SpMV) with the Jacobian, (ii) the triangular solves required by the ILU preconditioner. We target both components by designing custom, block-aware kernels in PETSc and by restructuring the assembly process to minimize redundant computations.

4.5.2 Data layout and block structure

The system matrix is stored in the `MATSEQBAIJ` format in PETSc with block size four, which corresponds to the natural grouping (u_x, u_y, u_z, p) at each mesh node. In this representation, every 4×4 block is laid out contiguously in memory, occupying sixteen consecutive scalars. Such an arrangement is particularly well suited for modern processors: vector registers of width 256 bits can be filled with aligned loads, fused multiply–add instructions operate directly on contiguous data, and the overhead of index indirection is greatly reduced compared to scalar CSR storage. The block-by-node ordering therefore provides the structural foundation required to develop SIMD-friendly kernels that achieve higher arithmetic intensity and better cache utilization.

4.5.3 Assembly optimizations

The residual and Jacobian assembly are carefully restructured:

- **Precomputation of element matrices:** constant contributions such as mass (M), diffusion (A_0), divergence (B), and pressure stabilization (D) are precomputed once and cached in an `ElementMatrices` structure.
- **Cached solution values:** a `SolutionCache` stores local velocity and pressure values extracted from PETSc vectors, avoiding repeated `VecGetValues` calls.
- **Separation of linear and nonlinear terms:** a linear Jacobian J_{linear} (mass, diffusion, B, D) is preassembled and reused, while convection Jacobians (A_1, A_2) are added at each Newton step. This reduces memory traffic and accelerates matrix updates.

This design ensures that only the genuinely nonlinear components are recomputed at each iteration, significantly reducing assembly overhead.

4.5.4 Custom SpMV kernel

To exploit the block structure of `MATSEQBAIJ` matrices with block size 4, we override PETSc's default `MATOP_MULT` operation with a hand-optimized AVX2 kernel `MatMult_SeqBAIJ_4_AVX2`. The replacement is achieved dynamically by setting the operation pointer as illustrated below:

```
1   PetscErrorCode OverrideMatMultWithAVX2(Mat A) {
2       PetscBool ok;
3       PetscCall(PetscObjectTypeCompare((PetscObject)A,
            MATSEQBAIJ, &ok));
4       if (!ok)
5           SETERRQ(PETSC_COMM_WORLD, PETSC_ERR_ARG_WRONG,
6                   "Need MATSEQBAIJ");
7       PetscCall(MatSetOperation(A, MATOP_MULT,
8                 (void(*)(void))MatMult_SeqBAIJ_4_AVX2));
9       return 0;
10  }
```

Kernel structure. The optimized kernel processes each block row sequentially. For every nonzero 4×4 block, the sixteen coefficients are loaded into SIMD registers, while the four entries of the input vector are broadcast across registers. The block contribution is then accumulated using the `_mm256_fmadd_pd` intrinsic, which fuses multiplication and addition into a single operation. Finally, the resulting four-component vector is stored contiguously in memory.

This design reduces indexing overhead, improves cache utilization, and exploits AVX2 vector units to achieve performance close to the roofline bound for block-structured SpMV. The full implementation of the AVX2 kernel is provided in Appendix 6.2.

4.5.5 Custom triangular solves

In addition to the SpMV kernel, a major cost in each GMRES iteration originates from the triangular solves associated with the ILU preconditioner. To address this, we override the `MATOP_SOLVE` operation of PETSc's factorized block matrix with a hand-optimized routine, `MatSolve_SeqBAIJ_4_AVX2`. The procedure is as follows: after configuring the preconditioner as ILU (`PCILU`) and performing the setup phase (`PCSetUp`), the factorized matrix F is retrieved, and the custom AVX2 kernel is installed through `MatSetOperation`. This substitution remains fully transparent to PETSc's high-level solver interface.

Kernel structure. The solver implements forward and backward substitution at the block level, exploiting the 4×4 structure induced by the velocity–pressure grouping. Each diagonal block is inverted locally, while off-diagonal contributions are accumulated with AVX2 fused multiply–add instructions. This design minimizes branching, ensures contiguous memory access, and leverages SIMD units to accelerate the accumulation phase. The computational pattern may be summarized as:

$$x_i \;\leftarrow\; D_i^{-1}\Big(b_i - \sum_{j<i} L_{ij}x_j - \sum_{j>i} U_{ij}x_j\Big),$$

with all operations performed at the block level.

Performance impact. Because the ILU preconditioner is applied at every GMRES iteration, reducing the cost of triangular solves has a direct and substantial effect on overall runtime. The complete implementation is provided in Appendix 6.3.

4.5.6 Integration into the Newton–Krylov solver

The optimized kernels are seamlessly integrated into the nonlinear solution strategy. First, constant element matrices are precomputed and assembled into a baseline Jacobian J_{linear}, containing the mass, diffusion, divergence, and stabilization terms. At each Newton iteration, only the nonlinear contributions (arising from convection terms A_1 and A_2) are updated, which substantially reduces assembly overhead. Boundary conditions are then enforced by nulifying out the corresponding rows and inserting identity blocks, while the residual vector is assembled using cached local values to avoid repeated access to global data structures.

Once the residual and Jacobian are assembled, the linear system is solved using a restarted GMRES method with restart parameter 30, preconditioned by ILU with level-of-fill 3. At this stage, both matrix–vector multiplications (`MATOP_MULT`) and triangular solves (`MATOP_SOLVE`) are executed with the custom AVX2 kernels described in Sections 4.5.4 and 4.5.5, ensuring that the dominant operations are efficiently mapped onto SIMD hardware. The Newton iteration is then updated according to the correction, and the process is repeated until the convergence criteria are met.

This integration preserves PETSc's high-level solver infrastructure while replacing only the performance-critical kernels, thereby combining numerical robustness with architecture-aware efficiency. All numerical experiments were performed with PETSc version 3.23.4, compiled using Intel oneAPI compilers and built with the following options:

```
--with-cc=mpiicc --with-cxx=mpiicpc --with-fc=mpiifort
--with-debugging=no --with-scalar-type=real
--download-fblaslapack=1
COPTFLAGS=-O3 CXXOPTFLAGS=-O3 FOPTFLAGS=-O3 --with-memalign=32
PETSC_ARCH=arch-linux-r-intel
```

4.5.7 Summary

By combining block-by-node ordering, preassembled linear terms, cached solution values, and custom AVX2 kernels for both SpMV and triangular solves, we reduce the per-iteration cost of GMRES dramatically. This approach leverages PETSc's modularity (`MatSetOperation`) while retaining all high-level solver infrastructure. The result is a Newton–Krylov solver where the linear algebra backend approaches hardware efficiency limits for block-structured PDE systems.

4.6 Solver Benchmarking

4.6.1 Test Mesh

All solver benchmarks in this section are performed on a single reference mesh generated with `Gmsh` [12]. The domain is a three-dimensional channel with an ellipsoidal obstacle placed at its center, discretized into unstructured tetrahedra. Boundary conditions are defined as follows: a Poiseuille parabolic profile on the inlet Γ_{in}, no-slip conditions on walls and on the obstacle Γ_{wall}, and natural outflow conditions on Γ_{out}.

The mesh contains $N_v = 30370$ nodes and $N_e = 174812$ (tetrahedral and triangular) elements, corresponding to $N_{\text{dof}} = 121480$ degrees of freedom in the stabilized P_1–P_1 finite element discretization.

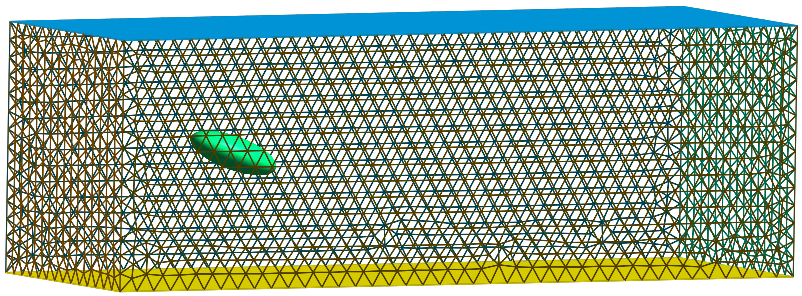

FIGURE 4.3: Visualization of the reference mesh with ellipsoidal obstacle.

4.6.2 Baseline solver study

We first establish a baseline characterization of the Newton–Krylov solver using PETSc's standard implementations. Particular attention is given to the interplay between the preconditioner and the Krylov method. The preconditioner is taken as an incomplete LU factorization, ILU(k), with the fill-in level k varied from 0 to 4. In parallel, the outer Krylov method is the restarted GMRES(m), where the restart length m controls both robustness and memory footprint. All other simulation parameters are held constant, with Reynolds number set to $Re = 100$, time step $\Delta t = 10^{-3}$, and Newton tolerance 10^{-6}. This systematic exploration of ILU and GMRES restart settings provides quantitative evidence of the algorithmic trade-offs that govern convergence, robustness, and runtime in large-scale saddle-point systems.

4.6.3 AVX2 kernel enhancement

In a second stage, we replace PETSc's default matrix–vector product and triangular solve routines by custom kernels specifically optimized for the 4×4 block structure of the discretization. These kernels leverage the AVX2 instruction set and fused multiply–add (FMA) operations to reduce memory traffic and improve instruction throughput. By integrating them directly into PETSc through the `MATOP_MULT` and `MATOP_SOLVE` hooks, the Newton–Krylov solver benefits from low-level optimizations without altering the high-level algorithm. This enables a clean separation between algorithmic and architectural effects: the first part of the benchmark isolates the solver's numerical behavior, while the second part quantifies the impact of hardware-conscious kernel design.

4.7 Results

4.7.1 Qualitative rendering

A visual inspection of the computed flow field using Paraview provides qualitative validation of the solver. Figure 4.4 shows a snapshot of the velocity magnitude around the ellipsoid obstacle at Reynolds number Re = 100. The background colormap highlights the accelerated jet upstream and the decelerated region in the wake. Streamlines seeded downstream exhibit deflection around the body and the onset of recirculation just behind the ellipsoid, though fully developed vortex shedding is not yet observed at this time horizon. This suggests that the flow is in a transient regime preceding the establishment of periodic vortex streets.

The visualization confirms that the stabilized P_1/P_1 discretization reproduces the main features of incompressible bluff-body flows: smooth velocity distribution along the obstacle, boundary-layer separation near the trailing edge, and the formation of a low-velocity recirculation zone in the wake. Although more refined meshes or longer integration times would be required to capture a fully periodic von Kármán street, the present rendering demonstrates that the numerical method correctly resolves the key mechanisms of separation and transition to unsteady flow.

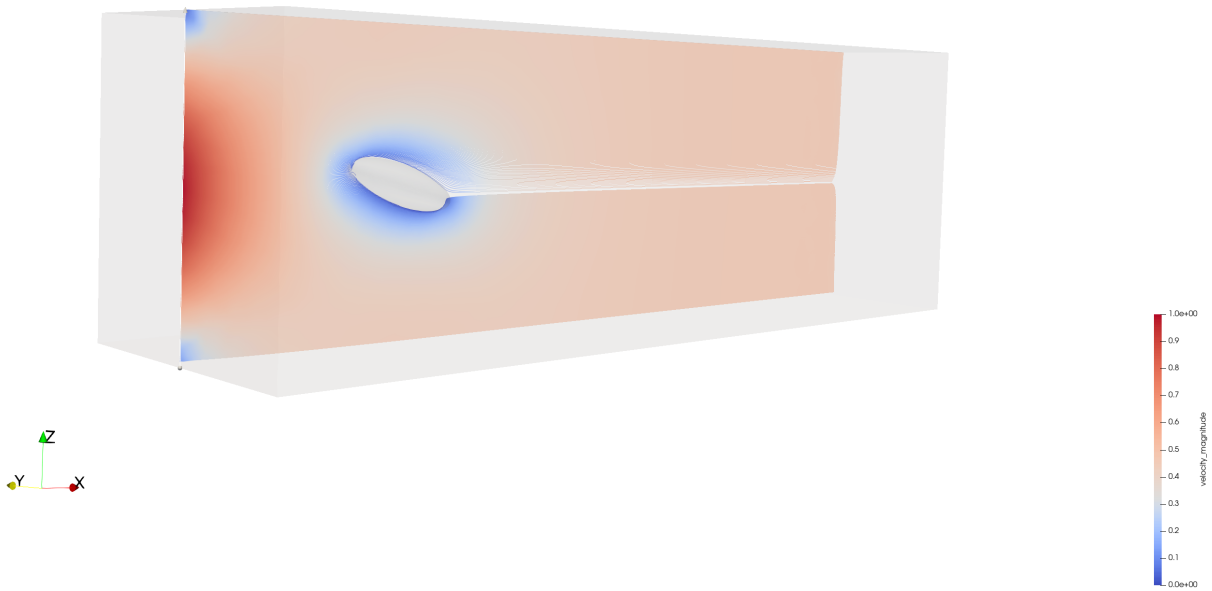

FIGURE 4.4: Qualitative visualization of the flow around an ellipsoid at $\mathrm{Re} = 100$. Shown is the velocity magnitude (background colormap).

4.7.2 Baseline solver results

Impact of ILU fill-in level

We first examine the influence of the ILU fill parameter k on the efficiency of the Newton–Krylov solver. Figure 4.5 shows the GMRES residual decay at the first Newton step for ILU(k) with $k = 0, \ldots, 4$. As anticipated, higher fill levels yield a more accurate approximation of the Jacobian inverse, thereby accelerating convergence. For example, ILU(0) requires over one hundred Krylov iterations, whereas ILU(2) and ILU(3) reduce this number by a factor of three to four. Beyond ILU(3), however, the gain in iteration count becomes marginal.

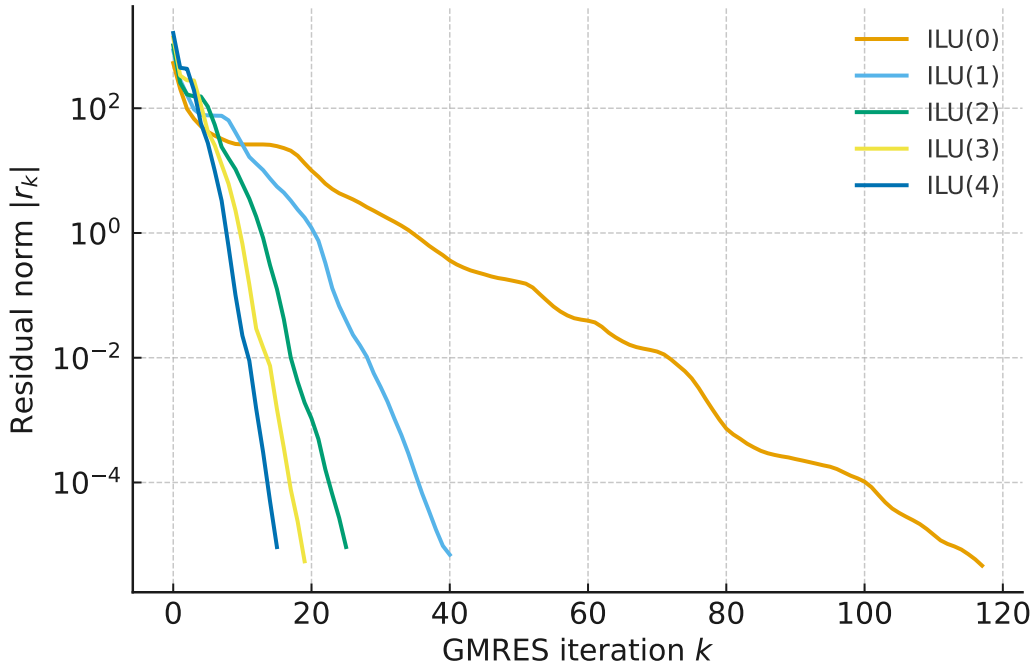

Figure 4.5: GMRES residual norm decay at the first Newton step for different ILU(k) levels.

The timing results reported in Table 4.1 provide further insight into the cost–benefit trade-off. The setup phase (`PCSetUp`), which corresponds to the ILU factorization, grows steeply with k: from 9.28 s for ILU(0) up to 158.49 s for ILU(4). By contrast, the cost of applying the preconditioner (`PCApply`) does not decrease with stronger factorizations; it slightly increases, since denser triangular factors make each application more expensive. As a consequence, the total runtime does not necessarily improve with higher fill levels: while ILU(2) and ILU(3) substantially accelerate convergence with acceptable setup overheads, ILU(4) incurs prohibitive factorization costs without delivering proportional gains. Overall, these experiments confirm that moderate fill-in levels strike the most effective balance between robustness and efficiency for the incompressible Stokes system considered.

Table 4.1: Solver timings for different ILU(k) preconditioners.

ILU(k)	**PCSetUp (s)**	**PCApply (s)**	**Total (s)**
ILU(0)	9.28	4.28	25.53
ILU(1)	12.19	4.36	28.06
ILU(2)	23.21	5.05	38.74
ILU(3)	59.49	6.19	77.89
ILU(4)	158.49	7.56	177.40

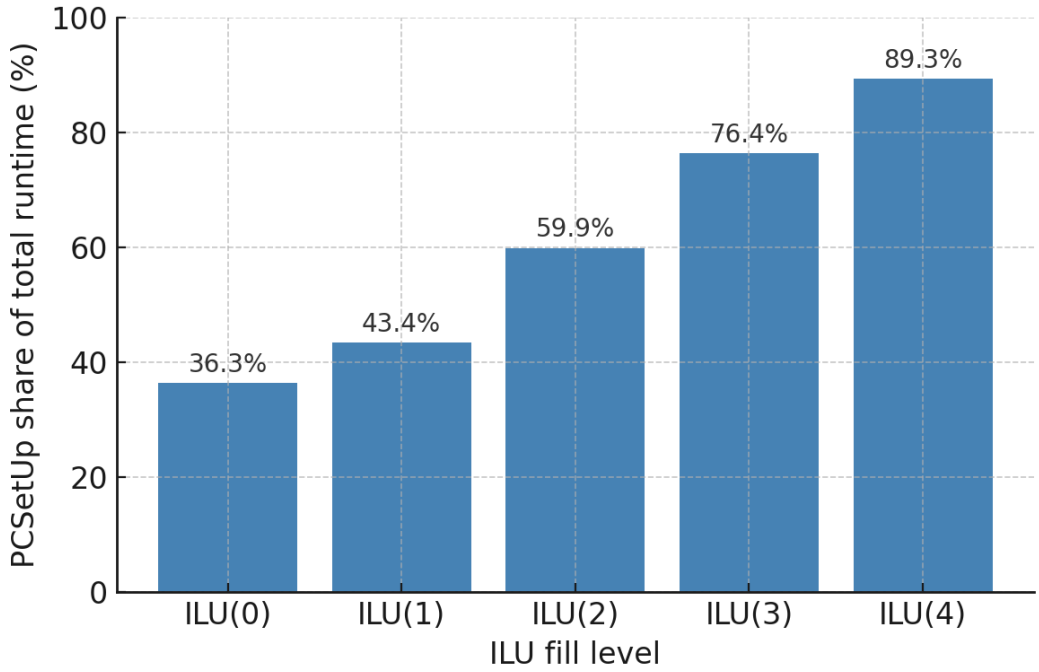

FIGURE 4.6: Proportion of the total runtime spent in the ILU factorization (`PCSetUp`) for different fill levels.

Figure 4.6 complements the raw timings by showing the fraction of the total runtime devoted to the ILU factorization. The share of `PCSetUp` grows from roughly 36% for ILU(0) to nearly 90% for ILU(4), making the setup stage the dominant cost at high fill levels. This confirms quantitatively that beyond ILU(3), additional fill-in primarily inflates factorization overhead without proportionally improving Krylov convergence.

Having established the role of ILU as a preconditioner, we next investigate the influence of the GMRES restart parameter, which controls memory usage and orthogonalization costs, and thus interacts directly with the efficiency of preconditioning.

Impact of GMRES restart length

We next investigate the effect of the GMRES restart parameter on solver performance. Figure 4.7 presents the residual norm decay at the first Newton step for restart lengths $m = 20, 30, 40, 80$, and 120, with ILU(0) used as preconditioner to ensure sufficiently long Krylov sequences. The results confirm the expected trend: small restart values degrade convergence, as illustrated by the stagnation plateaus observed for GMRES(20). In contrast, larger restart values such as GMRES(80) and GMRES(120) yield a smoother and faster reduction of the residual norm, since more search directions are retained before truncation.

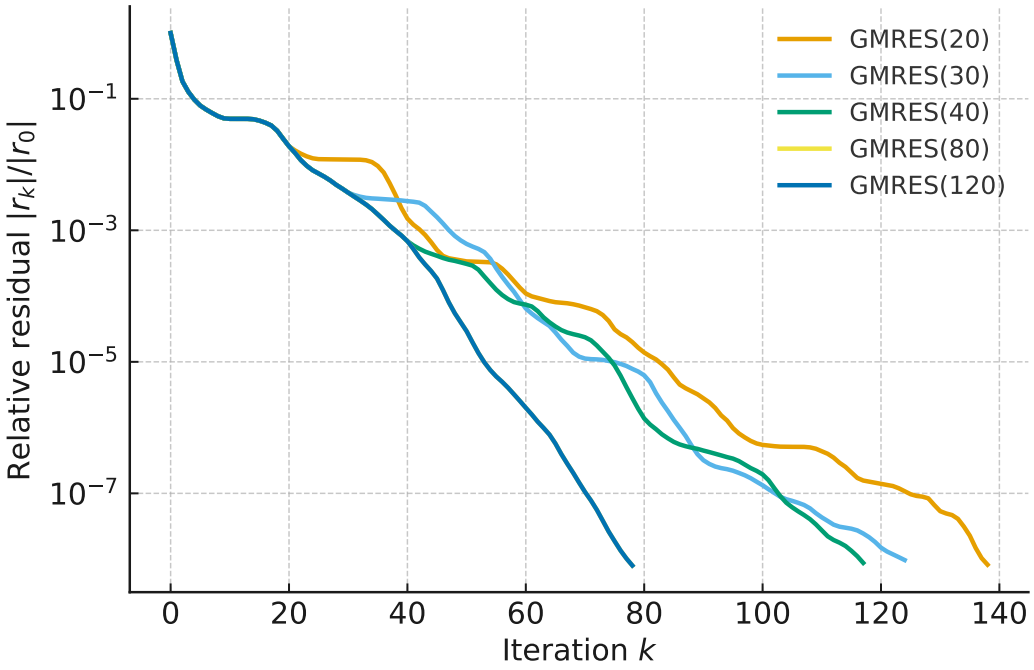

Figure 4.7: GMRES residual norm decay at the first Newton step for different restart lengths m.

Table 4.2 reports the corresponding timings and orthogonalization costs measured with PETSc's profiling tools. Two key points emerge. First, the overall runtime decreases slightly with increasing restart length, as larger Krylov spaces reduce the number of outer cycles required for convergence. Second, the cumulative orthogonalization cost (`KSPGMRESOrthog`) increases with m: while fewer cycles are required, each iteration involves orthogonalization against a larger basis, which dominates the kernel's expense. For example, the orthogonalization cost nearly doubles between GMRES(40) and GMRES(120), even though the total time decreases slightly.

Table 4.2: Solver timings and orthogonalization cost for different GMRES restart lengths (first Newton step, ILU(0) preconditioner).

Restart m	**Total time (s)**	**KSPGMRESOrthog (s)**
20	26.76	0.49
30	27.58	0.63
40	27.98	0.72
80	25.59	0.98
120	25.46	0.99

Overall, these experiments highlight the classical trade-off in restarted GMRES. Short restarts (e.g. $m = 20$) are inexpensive in memory but significantly impair convergence, while very large restarts (e.g. $m = 120$) bring limited additional benefit at the cost of increased storage. For the present incompressible flow problem, restart values in the range $m = 40$–80 achieve the most favorable balance, combining robust convergence with acceptable memory and orthogonalization costs.

4.7.3 Performance with AVX2 kernels

We conclude the benchmarking campaign by assessing the effect of AVX2-optimized kernels on solver performance. Table 4.3 compares the baseline PETSc `BAIJ(4)` implementation with the same solver augmented by custom AVX2 microkernels for `MATOP_MULT` and `MATOP_SOLVE`. The test is performed on the reference mesh at $\mathrm{Re} = 100$ with $\Delta t = 10^{-3}$ and final time $T = 1$.

Table 4.3: Execution time (in seconds) of critical solver components with and without AVX2 kernels.

Event	**Baseline**	**AVX2**	**Speedup**
MatMult	2065.7	1154.6	1.79
MatSolve	1483.6	1249.1	1.19
KSPSolve	4646.3	3121.2	1.49
Total time	**7083**	**6084**	**1.16**

The results indicate that the largest benefit arises in the sparse matrix–vector multiplication (`MatMult`), which nearly doubles in speed owing to efficient vectorized FMA operations on 4×4 blocks. Triangular solves (`MatSolve`) also gain from the AVX2 microkernels, albeit more moderately, with a 19% reduction in runtime. These improvements translate into a $1.5\times$ acceleration of the Krylov solver phase (`KSPSolve`), and ultimately reduce the overall runtime by approximately 16%.

In summary, the AVX2 kernels significantly accelerate the most expensive components of the Newton–Krylov solver. While `MatMult` dominates the gains, the end-to-end impact confirms that low-level kernel optimization is an effective lever to improve large-scale CFD solvers.

Chapter 5

Conclusions and Remarks

This work has investigated the optimization of Krylov-based solvers for the incompressible Navier–Stokes equations, with particular emphasis on the interaction between algorithmic strategies and architecture-aware implementations. A stabilized P_1/P_1 finite element discretization was solved using a Newton–Krylov method, providing a realistic benchmark for the analysis of preconditioners, Krylov restart strategies, and custom sparse matrix kernels.

From the algorithmic perspective, the study of ILU(k) preconditioners showed that moderate fill levels ($k = 2,3$) achieve the best compromise between factorization cost and robustness of convergence. Increasing the fill level further yields only marginal benefits while incurring prohibitive setup times. Similarly, the investigation of GMRES restart lengths confirmed the expected trade-off between memory, orthogonalization cost, and convergence speed, with restarted values $m = 40$–80 offering the most favorable balance. From the computational perspective, hand-optimized AVX2 kernels for block-structured SpMV and triangular solves reduced memory traffic and improved throughput, leading to a $\sim 15\%$ end-to-end speedup. This validates the importance of combining numerical methods with hardware-conscious design.

Beyond these directions, a promising avenue lies in the integration of fused sparse matrix–vector kernels, where several Krylov basis vectors are generated simultaneously through a matrix powers kernel (MPK). Such techniques, closely related to s-step and communication-avoiding Krylov methods [13, 14], have the potential to reduce synchronization overhead while exploiting cache locality and SIMD throughput. The AVX2 block kernels developed here provide a natural building block for such fused operations, and extending them to s-step GMRES constitutes an attractive next step.

In summary, this thesis demonstrates that meaningful gains in CFD solver performance can be achieved through a dual approach: optimizing both the algorithmic building blocks and their architectural realization. Extending these ideas with more advanced preconditioners and multi-physics couplings constitutes a natural and promising continuation of the present work.

Chapter 6

Appendix

```
1   void SpMV_BCSR_AVX2(double *y, const double *x, const bcsr4x4_matrix &A)
2   {
3       const int nblock_rows = A.nrows;
4       std::memset(y, 0, 4 * nblock_rows * sizeof(double));
5
6       for (int bi = 0; bi < nblock_rows; ++bi)
7       {
8           __m256d acc = _mm256_setzero_pd();
9
10          for (int ia = A.ptrow[bi]; ia < A.ptrow[bi + 1]; ++ia)
11          {
12              const int bj = A.indcol[ia];
13              const double *blk = &A.coef[16 * ia];
14
15              __m256d a0 = _mm256_set_pd(blk[12], blk[8], blk[4], blk[0]);
16              __m256d x0 = _mm256_set1_pd(x[4 * bj + 0]);
17              acc = _mm256_fmadd_pd(a0, x0, acc);
18
19              __m256d a1 = _mm256_set_pd(blk[13], blk[9], blk[5], blk[1]);
20              __m256d x1 = _mm256_set1_pd(x[4 * bj + 1]);
21              acc = _mm256_fmadd_pd(a1, x1, acc);
22
23              __m256d a2 = _mm256_set_pd(blk[14], blk[10], blk[6], blk[2]);
24              __m256d x2 = _mm256_set1_pd(x[4 * bj + 2]);
25              acc = _mm256_fmadd_pd(a2, x2, acc);
26
27              __m256d a3 = _mm256_set_pd(blk[15], blk[11], blk[7], blk[3]);
28              __m256d x3 = _mm256_set1_pd(x[4 * bj + 3]);
29              acc = _mm256_fmadd_pd(a3, x3, acc);
30          }
31
32          _mm256_storeu_pd(&y[4 * bi], acc);
33      }
34  }
```

Listing 6.1: AVX2 SpMV kernel for 4×4 BCSR blocks

```
PetscErrorCode MatMult_SeqBAIJ_4_AVX2(Mat A, Vec xx, Vec zz) {
Mat_SeqBAIJ       *a      = (Mat_SeqBAIJ*)A->data;
const PetscScalar *x;
PetscScalar       *zarray;
const MatScalar   *v;
const PetscInt    *idx, *ii, *ridx = NULL;
PetscInt           mbs, i, j, n, bs2 = a->bs2;
PetscBool          usecprow = a->compressedrow.use;

PetscFunctionBegin;

PetscCall(VecGetArrayRead(xx, &x));
PetscCall(VecGetArrayWrite(zz, &zarray));

idx = a->j;
v   = a->a;

if (usecprow) {
    mbs  = a->compressedrow.nrows;
    ii   = a->compressedrow.i;
    ridx = a->compressedrow.rindex;
    PetscCall(PetscArrayzero(zarray, 4 * a->mbs));
} else {
    mbs = a->mbs;
    ii  = a->i;
}

for (i = 0; i < mbs; i++) {
    __m256d acc = _mm256_setzero_pd();
    n = ii[1] - ii[0];
    ii++;

    for (j = 0; j < n; j++) {
        const PetscInt col = idx[j];
        const double *blk = v + j * 16;

        __m256d a0 = _mm256_load_pd(blk +  0);
        __m256d a1 = _mm256_load_pd(blk +  4);
        __m256d a2 = _mm256_load_pd(blk +  8);
        __m256d a3 = _mm256_load_pd(blk + 12);

        __m256d x0 = _mm256_set1_pd(x[4 * col + 0]);
        __m256d x1 = _mm256_set1_pd(x[4 * col + 1]);
        __m256d x2 = _mm256_set1_pd(x[4 * col + 2]);
        __m256d x3 = _mm256_set1_pd(x[4 * col + 3]);

        acc = _mm256_fmadd_pd(a0, x0, acc);
        acc = _mm256_fmadd_pd(a1, x1, acc);
        acc = _mm256_fmadd_pd(a2, x2, acc);
        acc = _mm256_fmadd_pd(a3, x3, acc);
    }

    PetscScalar *z = usecprow ? (zarray + 4 * ridx[i]) : (zarray + 4 * i);
    _mm256_storeu_pd(z, acc);

    idx += n;
    v   += n * 16;
}

PetscCall(VecRestoreArrayRead(xx, &x));
PetscCall(VecRestoreArrayWrite(zz, &zarray));
PetscCall(PetscLogFlops(2.0 * a->nz * bs2 - 4.0 * a->nonzerorowcnt));
PetscFunctionReturn(PETSC_SUCCESS);
}
```

Listing 6.2: AVX2-optimized matrix–vector multiplication for PETSc `BAIJ(4)` matrices.

```
PetscErrorCode MatSolve_SeqBAIJ_4_AVX2(Mat A, Vec bb, Vec xx) {
Mat_SeqBAIJ *a = (Mat_SeqBAIJ*)A->data;
IS iscol = a->col, isrow = a->row;
const PetscInt n = a->mbs, *vi, *ai = a->i, *aj = a->j, *adiag =
    a->diag;
const MatScalar *aa = a->a, *v;
const PetscScalar *b;
PetscScalar *x, *t;
PetscInt i, m, nz, idx, idt, idc;

PetscFunctionBegin;
PetscCall(VecGetArrayRead(bb, &b));
PetscCall(VecGetArray(xx, &x));
t = a->solve_work;

const PetscInt *r, *c;
PetscCall(ISGetIndices(isrow, &r));
PetscCall(ISGetIndices(iscol, &c));

// Initial block
idx = 4 * r[0];
for (int k = 0; k < 4; ++k) t[k] = b[idx + k];

// Forward substitution
for (i = 1; i < n; i++) {
    v  = aa + 16 * ai[i];
    vi = aj + ai[i];
    nz = ai[i + 1] - ai[i];
    idx = 4 * r[i];

    __m256d s = _mm256_set_pd(b[idx+3], b[idx+2], b[idx+1],
        b[idx]);
    for (m = 0; m < nz; m++) {
    const double *xp = t + 4 * vi[m];
    __m256d v0 = _mm256_set_pd(v[3],  v[2],  v[1],  v[0]);
    __m256d v1 = _mm256_set_pd(v[7],  v[6],  v[5],  v[4]);
    __m256d v2 = _mm256_set_pd(v[11], v[10], v[9],  v[8]);
    __m256d v3 = _mm256_set_pd(v[15], v[14], v[13], v[12]);
    s = _mm256_fnmadd_pd(v0, _mm256_set1_pd(xp[0]), s);
    s = _mm256_fnmadd_pd(v1, _mm256_set1_pd(xp[1]), s);
    s = _mm256_fnmadd_pd(v2, _mm256_set1_pd(xp[2]), s);
    s = _mm256_fnmadd_pd(v3, _mm256_set1_pd(xp[3]), s);
    v += 16;
    }
    idt = 4 * i;
    _mm256_store_pd(t + idt, s);
}

// Backward substitution
for (i = n - 1; i >= 0; i--) {
    v  = aa + 16 * (adiag[i + 1] + 1);
    vi = aj + adiag[i + 1] + 1;
    nz = adiag[i] - adiag[i + 1] - 1;
    idt = 4 * i;

    __m256d s = _mm256_load_pd(t + idt);
    for (m = 0; m < nz; m++) {
    const double *xp = t + 4 * vi[m];
```

```
57          __m256d v0 = _mm256_set_pd(v[3],  v[2],  v[1],  v[0]);
58          __m256d v1 = _mm256_set_pd(v[7],  v[6],  v[5],  v[4]);
59          __m256d v2 = _mm256_set_pd(v[11], v[10], v[9],  v[8]);
60          __m256d v3 = _mm256_set_pd(v[15], v[14], v[13], v[12]);
61          s = _mm256_fnmadd_pd(v0, _mm256_set1_pd(xp[0]), s);
62          s = _mm256_fnmadd_pd(v1, _mm256_set1_pd(xp[1]), s);
63          s = _mm256_fnmadd_pd(v2, _mm256_set1_pd(xp[2]), s);
64          s = _mm256_fnmadd_pd(v3, _mm256_set1_pd(xp[3]), s);
65          v += 16;
66          }
67
68          idc = 4 * c[i];
69          _mm256_store_pd(x + idc, s);
70          _mm256_store_pd(t + idt, s);
71      }
72
73      PetscCall(ISRestoreIndices(isrow, &r));
74      PetscCall(ISRestoreIndices(iscol, &c));
75      PetscCall(VecRestoreArrayRead(bb, &b));
76      PetscCall(VecRestoreArray(xx, &x));
77      PetscFunctionReturn(PETSC_SUCCESS);
78      }
```

Listing 6.3: AVX2-optimized triangular solve for BAIJ(4)

List of Figures

List of Tables

Bibliography

[1] S. Balay et al., "PETSc/TAO Users Manual", Argonne National Laboratory, Tech. Rep. ANL-21/39 - Revision 3.21, 2024.

[2] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems", *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, pp. 856–869, 1986.

[3] Y. Saad, "Ilut: A dual threshold incomplete lu factorization", *Numerical Linear Algebra with Applications*, vol. 1, no. 4, pp. 387–402, 1994.

[4] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems", *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, 1952.

[5] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd. SIAM, 2003.

[6] W. E. Arnoldi, "The principle of minimized iterations in the solution of the matrix eigenvalue problem", *Quarterly of Applied Mathematics*, vol. 9, no. 1, pp. 17–29, 1951.

[7] C. Lanczos, "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators", *Journal of Research of the National Bureau of Standards*, vol. 45, pp. 255–282, 1950.

[8] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures", *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[9] F. Hecht, "New development in freefem++", *J. Numer. Math.*, vol. 20, no. 3-4, pp. 251–265, 2012, ISSN: 1570-2820. [Online]. Available: https://freefem.org/.

[10] F. Brezzi and M. Fortin, *Mixed and Hybrid Finite Element Methods*. Springer, 1991.

[11] T. J. R. Hughes, L. P. Franca, and M. Balestra, "A new finite element formulation for computational fluid dynamics: V. circumventing the babuška–brezzi condition", *Computer Methods in Applied Mechanics and Engineering*, vol. 59, no. 1, pp. 85–99, 1986.

[12] C. Geuzaine and J.-F. Remacle, "Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities", *International Journal for Numerical Methods in Engineering*, vol. 79, no. 11, pp. 1309–1331, 2009.

[13] A. T. Chronopoulos and C. W. Gear, "S-step iterative methods for symmetric linear systems", *Journal of Computational and Applied Mathematics*, vol. 25, no. 2, pp. 153–168, 1989.

[14] M. Hoemmen, "Communication-avoiding krylov subspace methods", Ph.D. dissertation, University of California, Berkeley, 2010.