UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

RAPORT DE CERCETARE SEMESTRUL II

Optimizarea Arhitecturală a Înmulțirii Matrice-Vector Rară pe GPU

Alexandru Chirilă

Coordonator științific:

Professor Dr. Ing. Emil Ioan Slușanschi

BUCUREȘTI
2025

UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

RESEARCH REPORT
SEMESTER II

Architectural-Aware Optimization of Sparse Matrix-Vector Multiplication on GPUs

Alexandru Chirilă

Thesis advisor:

Professor Dr. Ing. Emil Ioan Slușanschi

BUCHAREST

2025

CONTENTS

SINOPSIS

Această lucrare investighează optimizarea înmulțirii matrice-vector sparse în format Block Sparse Row (BSR) pe GPU-uri NVIDIA. Operațiile cu matrici sparse sunt esențiale în calculul de înaltă performanță, însă prezintă provocări din cauza accesului neregulat la memorie și a intensității aritmetice scăzute. Pornind de la implementări pe CPU inspirate de kernelul BSR din SciPy, a fost dezvoltată o serie de kerneluri GPU care folosesc strategii adaptate arhitecturii pentru a aborda contencția atomică, presiunea pe registre și limitările cache-ului. Evaluările de performanță arată că cel mai eficient kernel personalizat atinge un câștig de 2,17× față de cuSPARSE pe un GPU Tesla P100, cu un debit de memorie susținut de 539 GB/s, demonstrând eficiența alinierii atente a accesului la memorie, distribuției sarcinilor și programării thread-urilor pentru operații sparse pe arhitecturi GPU moderne.

ABSTRACT

This dissertation investigates the optimization of Block Sparse Row (BSR) sparse matrix-vector multiplication on NVIDIA GPUs. Sparse matrix operations are critical in high-performance computing, yet they pose challenges due to irregular memory access and low arithmetic intensity. Starting from CPU-based implementations inspired by SciPy's BSR kernel, a series of GPU kernels were developed using architecture-aware strategies to address atomic contention, register pressure, and cache limitations. Performance evaluations demonstrate that the best custom kernel achieves a speedup of 2.17× over cuSPARSE on a Tesla P100 GPU, with a sustained memory throughput of 539 GB/s, highlighting the effectiveness of careful alignment of memory access patterns, workload distribution, and thread scheduling for sparse computations on modern GPU architectures.

1 INTRODUCTION

High-performance computing (HPC) has undergone a significant paradigm shift over the past two decades, driven by the need for massive computational throughput in the scientific, industrial, and commercial domains. Among the most transformative developments is the rise of graphics processing units (GPUs) as general-purpose accelerators. Originally designed for the highly parallel workload of computer graphics, modern GPUs have evolved into programmable, high-throughput architectures capable of delivering substantial performance gains for diverse data-parallel applications.

1.1 Evolution of General-Purpose Accelerators

NVIDIA has played a central role in the adaptation of GPUs for general-purpose computing, continuously refining its architectures to exploit massive parallelism, deep memory hierarchies, and high-bandwidth interconnects. While NVIDIA has led much of this transformation, competitors such as AMD and Intel have also developed GPU architectures and accelerator products, AMD with its CDNA [4] and RDNA [3] compute architectures, and Intel with its Xe [9] architecture, broadening the ecosystem for heterogeneous computing. Nevertheless, NVIDIA's CUDA [25] platform, combined with aggressive hardware advancements, has positioned it as the dominant platform in high-performance GPU-accelerated computing.

1.2 The Role of Sparse Matrix Operations in HPC

Sparse matrix operations form a cornerstone of computational science and appear in applications ranging from numerical simulation and optimization to graph analytics and machine learning. In these matrices, most elements are zero, making explicit storage of all entries inefficient in both space and time. Sparse algorithms exploit this structure to reduce memory requirements and computational effort, enabling large-scale problems to be solved within realistic resource limits. Formats such as Coordinate format (COO) [16], Compressed Sparse Row (CSR) [19], Compressed Sparse Column (CSC) [17], and Block Sparse Row (BSR) [18] are commonly used to store and process sparse data efficiently.

1.3 BLAS-style Sparse Matrix-Vector Multiplication (SpMV)

A fundamental operation in scientific computing is the sparse matrix-vector multiplication (SpMV) [22] , in the BLAS-like form: $y \leftarrow \alpha Ax + \beta y$ where A is a sparse matrix and x, y are dense vectors. This operation is the computational kernel behind many iterative solvers (e.g., conjugate gradient methods [23]), time-stepping schemes in PDEs, and network or graph-based computations.

Most libraries implement the BLAS-style formulation, which allows for flexible scaling and accumulation of results while maintaining compatibility with established linear algebra libraries. Its predictable pattern and general-purpose applicability make it an ideal candidate for benchmarking sparse kernels on both CPU and GPU architectures. The implementations presented in this dissertation adopt this formulation to ensure comparability and relevance to real-world scientific workloads.

1.4 Challenges of Sparse Data on GPUs

Despite the potential for high performance, sparse matrix operations are notoriously difficult to optimize on GPUs. The irregular patterns of memory access inherent to sparse structures can disrupt coalesced memory access, leading to suboptimal utilization of the available memory bandwidth. The low arithmetic intensity of these operations often results in memory-bound performance, where throughput is limited by data transfer speed rather than processing capability. The load imbalance between threads further reduces efficiency, particularly in cases where nonzero elements are unevenly distributed across the matrix. Overcoming these challenges requires careful design choices that align data layout, memory access patterns, and thread scheduling with the architectural features of the target GPU.

2 OBJECTIVE

This dissertation investigates how NVIDIA GPU architectures can be leveraged effectively to accelerate sparse matrix computations, with a particular focus on architectural-aware optimization strategies. The work emphasizes the role of memory access patterns and workload distribution strategies in closing the gap between theoretical peak performance and practical implementation.

2.1 BSR Matrix-Vector Multiplication

A key case study in this research is the optimization of BLAS-style Block Sparse Row (BSR) matrix-vector multiplication (BSR SpMV). This operation is of particular interest because it strikes a balance between the irregularity of general sparse matrices and the structured efficiency of block operations when the sparsity pattern is convenient. Using BSR as a focal example, this dissertation explores how specialized data layouts, warp-level coordination, and cache-aware memory strategies can significantly improve performance for sparse computations on NVIDIA GPUs. The resulting insights aim to provide practical, scalable methods for maximizing throughput in real-world sparse workloads.

2.2 CPU-GPU Architectural Parallels

While NVIDIA GPU architecture remains the primary focus of this dissertation, selected comparisons will be drawn to modern x86 CPUs, particularly the chiplet-based designs [5] of AMD, to clarify architectural roles and highlight both key differences and similarities in memory systems, cache hierarchies, and execution models. This comparative perspective provides familiar reference points for understanding GPU components such as streaming multiprocessors, CUDA cores, warp schedulers, memory controllers, and caches, and frames the optimization techniques discussed in later chapters within a broader architectural context.

As part of this comparative approach, a CPU implementation of Block Sparse Row (BSR) [18] matrix-vector multiplication [22] is presented as both a baseline for performance evaluation and a conceptual starting point. By examining how the algorithm must be adapted to align with GPU execution models and memory hierarchies, the dissertation illustrates the architectural considerations that drive design choices in high-performance sparse computations.

3 SPARSE FORMATS

Efficient storage of sparse matrices is essential to reduce memory footprint and improve performance in large-scale computations. A suitable sparse format not only minimizes storage overhead, but also shapes the memory access patterns during computation, directly influencing how well an algorithm maps onto modern hardware.

Among the many formats available, Block Sparse Row (BSR) [18] is of particular relevance to this work. BSR can be viewed as a block-structured extension of the widely used Compressed Sparse Row (CSR) format [19], where nonzero entries are grouped into small dense submatrices. This organization improves memory alignment and enables block-level optimizations when the sparsity pattern is favorable.

This chapter first outlines general computational characteristics of sparse matrix–vector multiplication (Section 3.1). It then introduces two common baseline formats, Coordinate (COO, Section 3.2) and Compressed Sparse Row (CSR, Section 3.3), to establish the foundations needed to understand block-based storage. Finally, Section 3.4 presents the Block Sparse Row (BSR) format, which is the main focus of this work.

3.1 SpMV Computational Characteristics

Sparse matrix–vector multiplication (SpMV) [22] is a fundamental operation in high-performance computing, with applications that span numerical simulation, optimization, graph analysis, and machine learning. As introduced in Section 1.2, sparse formats exploit the predominance of zero entries to reduce storage requirements and improve execution efficiency.

SpMV, like general matrix–vector multiplication (GEMV) [24], is inherently memory-bound: Performance is constrained more by memory bandwidth than by computational throughput. Each stored element of the matrix is read once and used in a single multiply-accumulate operation, leaving no opportunity for data reuse except for the input vector. In GEMV, the dense storage layout supports predictable cache-friendly access patterns, whereas SpMV [22] often produces irregular accesses, particularly on the input and result vectors according to the sparsity pattern and the storage format used for the sparse matrix. These irregularities make storage format selection a critical factor in achieving high performance on both CPUs and GPUs.

3.2 The Coordinate Format

The Coordinate format (COO) [16] is probably the most straightforward one for storing a sparse matrix. Although it is not of direct interest in this work, it serves as an introduction to sparse formats and will help to understand the advantages of CSR and BSR better.

COO stores each nonzero as a tuple of its row index, column index, and value. In practice, libraries such as SciPy [30] and PETSc [6] implement this with three different arrays. Its simplicity makes it well suited for constructing matrices and for handling irregular sparsity patterns, and its order of storage can be chosen for cache-friendly access in certain operations.

In terms of storage, COO keeps *nnz* values and $2 \times nnz$ indices, where *nnz* is the number of nonzeros in the matrix. This cost depends only on *nnz* and is independent of the shape of the matrix and is usually much smaller, less than 5% of the total number of elements of the sparse matrix if stored in the typical dense format.

However, the flexibility of the COO [16] storage format comes with a few downsides. Element lookup or modification requires scanning the entire structure, making such operations *O(nnz)* where *nnz* is the number of nonzeros. For this reason, most libraries avoid complex indexing or slicing operations in COO.

Some implementations, such as PyTorch [2], support an "uncoalesced" COO variant in which multiple entries may share the same coordinates; these are summed when the matrix is coalesced or when it is involved in an operation. This even more flexible variant of COO can simplify incremental matrix construction, but can also increase memory usage and can reduce performance in every kind of operation compared to the standard COO if there are multiple entries for a subset of coordinates.

3.3 The Compressed Sparse Row Format

The Compressed Sparse Row (CSR) format [19], much like COO [16], stores the values and column indices of the nonzeros, either as one array of tuples or as two separate arrays. However, it is more compact than COO because it does not store a row index for each nonzero. Instead, nonzeros are grouped by row, and a separate array records the starting position of each row of data. CSR is well-suited for arithmetic operations, particularly row-major ones, such as SpMV.

In terms of storage, CSR keeps *nnz* values and $nnz + M + 1$ indices, where M is the number of rows in the matrix. This is typically almost half the index storage required by COO, since M is usually much smaller than *nnz*. The reduced index size benefits not only the memory footprint but also caching in memory-bound operations such as SpMV.

Most CSR implementations, such as SciPy [30], CuPy [26], and PETSc [6], also ensure that elements in each row are sorted in ascending order of their column indices. This allows element

lookups or slicing to use binary search within a single row, rather than scanning the entire matrix.

The main drawback of CSR is that it is inefficient for inserting new elements incrementally, as doing so would require shifting data in the value and index arrays. For this reason, CSR is typically used once the matrix structure is fixed, rather than for dynamic construction.

3.4 The Block Sparse Row Format

The Block Sparse Row (BSR) format [18] is, as the name suggests, a block-oriented variant of CSR [19], in which the basic unit is not an individual nonzero but an $R \times C$ dense block of nonzeros where R and C are the numbers of rows and columns per block, respectively. Even though in principle the block sizes could vary from row to row, most implementations enforce a fixed block shape for efficiency. CSR can be seen as a special case of BSR with $R = 1$ and $C = 1$. The column index array now records the column position of each block, and the row pointer array marks the start of each row of blocks rather than each row of scalar entries.

In terms of storage, BSR keeps the same *nnz* values as CSR but only $nnbz + BM + 1$ indices, where *nnbz* is the number of nonzero blocks and BM is the number of rows of blocks. When all blocks are fully dense, *nnbz* is approximately $nnz \,/\, (R \times C)$ or exactly that if the blocks do not contain unnecessary data. This reduced index size leaves more cache capacity for the value data and other working structures, which is especially beneficial in memory-bound operations such as SpMV [22].

As in CSR [19], most BSR [18] implementations (e.g. SciPy) store blocks in each block row in ascending order of their column indices, enabling fast lookups and slicing.

The main drawback of BSR is similar as for CSR: inserting new entries incrementally is inefficient, making the format best suited for fixed-structure matrices prepared in advance rather than for dynamic construction.

4 RELATED WORK

Understanding existing software frameworks for matrix computations is crucial to situate the contributions of this dissertation. Both dense and sparse matrix operations have been extensively studied, and several well-established libraries offer reference implementations that illustrate common design patterns and performance trade-offs. For dense computations, NumPy [8] provides a baseline for evaluating the benefits of sparse storage formats. Sparse operations on the CPU are supported by libraries such as SciPy [30] in Python [28], which builds on NumPy structures and implements multiple sparse formats, including BSR [18]. On the GPU side, NVIDIA's cuSPARSE library [14] offers optimized kernels for sparse computations, including the block sparse matrix vector multiplication routine `cusparseDbsrmv`, which serves as the main comparison point for this work.

Although most Python libraries, such as NumPy and SciPy, expose only the simple form of matrix-vector multiplication ($y \leftarrow Ax$), this work adopts the BLAS-style variant ($y \leftarrow \alpha Ax + \beta y$), as also implemented in cuSPARSE, to enable flexible scaling and accumulation. The following sections introduce NumPy as the CPU dense parallel baseline, SciPy as the CPU sequential sparse baseline, and cuSPARSE as the GPU sparse reference.

4.1 NumPy

NumPy [8] is a foundational Python library for numerical computing, providing support for multi-dimensional arrays and a wide range of mathematical operations. Its core strength lies in parallel dense array operations, which are highly optimized for performance on modern CPUs. The NumPy array forms the backbone of many other scientific computing libraries, including SciPy [30] and CuPy [26], offering a consistent interface and efficient memory layout.

Although NumPy does not natively support sparse matrices, its dense matrix-vector multiplication routine (GEMV) [24] provides a useful baseline. Comparing dense operations with sparse ones highlights the performance and memory advantages of using specialized sparse formats, such as BSR, especially in memory-bound scenarios where unnecessary computations on zero elements would otherwise reduce efficiency.

4.2 SciPy

SciPy [30] is an open-source Python library with many algorithms widely used in scientific computing. For this work, the focus is on its BSR [18] matrix structures and associated

matrix-vector multiplication routines, particularly the BSR implementation which serves as starting points for the CPU and GPU implementations.

In SciPy, a BSR matrix is internally represented by:

- a tri-dimensional NumPy array of nonzero blocks,
- a one-dimensional integer array for the column indices of each block,
- and a one-dimensional integer array that marks the starting position of each block row in the index array.

This layout parallels the CSR format, which stores individual scalar nonzeros instead of two-dimensional blocks, and provides a compact, row-oriented representation. As mentioned in Section 3.4, by grouping adjacent nonzeros into blocks, the BSR format reduces indexing overhead and improves cache utilization, particularly when operations can exploit block-level locality.

SciPy's BSR implementation is sequential, without explicit multithreading or GPU offloading. This makes it a suitable starting point for the CPU BSR SpMV baseline described in Chapter 6 and provides the conceptual foundation for the GPU BSR implementations presented later in Chapter 8.

4.3 cuSPARSE

cuSPARSE [14] is a low-level GPU-accelerated library providing basic linear algebra subroutines for sparse matrices. Built on the CUDA runtime, it is designed for use from C [1] and C++ [21], and offers significantly higher performance than CPU-only alternatives. Unlike SciPy [30], cuSPARSE operates on raw data arrays for each sparse format (CSR, BSR, etc.) rather than high-level sparse matrix objects, enabling fine-grained control and integration with custom GPU implementations.

In this work, cuSPARSE serves as a GPU reference for block sparse row (BSR) matrix-vector multiplication. The function `cusparseDbsrmv` performs double precision BSR SpMV, with a data layout compatible with the internal SciPy BSR representation, making it a convenient reference for evaluating custom CUDA BSR SpMV implementations (chapter 8). In our experiments, we invoke `cusparseDbsrmv` from Python via `ctypes`, using the same memory layout and calling conventions as the custom kernels.[1]

[1]Note that `cusparseDbsrmv` has been deprecated in recent CUDA versions in favor of the generic `cusparseSpMV` interface; however, it remains suitable as a reference for the purposes of this work.

5 DATASET AND BENCHMARKING METHODOLOGY

To evaluate and compare the performance of different BSR SpMV implementations, a consistent and reproducible benchmarking methodology is required. Just as hardware is designed to excel under certain workloads, software implementations can favor specific structural patterns to achieve peak efficiency. To make sense of the design decisions behind our BSR SpMV implementations, this chapter introduces the dataset used for benchmarking on both CPU and GPU, along with the methodology guiding our measurements.

5.1 Dataset

The benchmark dataset consists of two scalars, two stochastic input vectors of 32,000 double precision floating point elements, and a *32,000×32,000* stochastic block-band matrix, where each block has a shape of *5×5*. Globally, 5% of the elements are nonzero, but each block is fully dense. All nonzero elements are positive, and the sum of the elements in each row is equal to 1. Figure 1 shows a visualization of the matrix pattern used in our dataset.

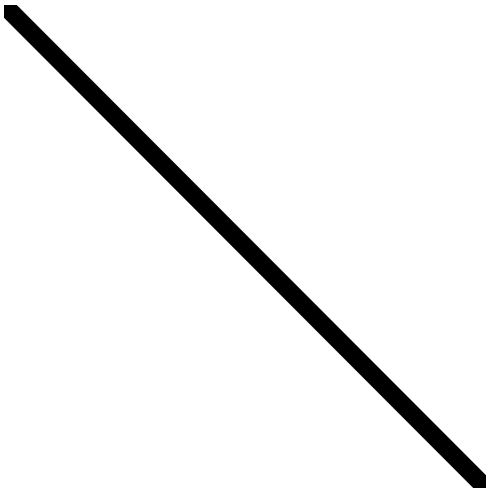

Figure 1: Visual representation for the matrix pattern in the dataset

5.1.1 Applications for Stochastic Block-Band Matrices

The choice of a stochastic block-banded structure is not arbitrary. It reflects a class of problems where local dense interactions coexist with global sparsity. This hybrid pattern is common in scientific computing and data modeling, making it particularly relevant for benchmarking. Some representative domains include:

- **Markov chains and stochastic processes**, where rows encode transition probability distributions;
- **Finite element and finite difference discretizations**, which often yield banded or block-sparse systems;
- **Graph and network models**, especially the models with modular or hierarchical community structure;
- **Hierarchical and multi-scale models**, which embed dense local couplings into sparse global systems.

Together, these cases highlight the practical significance of the dataset, validating its role as a meaningful benchmark for sparse computations in real-world scenarios.

5.1.2 Choice of Density

At the global density of 5%, this dataset represents one of the densest cases encountered in practice for block-band sparse matrices. Although this density appears at times in real-world applications, much lower densities are also encountered, typically 1% - 2%, sometimes even as low as 0.1%. This choice thus serves two purposes: it provides a demanding benchmark for sparse algorithms, while also giving dense GEMV [24] its least unfavorable ground for comparison. However, even in this scenario, GEMV implementations must process roughly 20 times more numerical values than SpMV [22] implementations, showing that the overhead of sparse indexing is negligible compared to the savings from avoiding unnecessary operations. As such, the results will directly highlight the importance of using appropriate sparse storage formats in applications such as those mentioned above.

5.1.3 Memory Footprint

The size of the benchmark dataset is not only determined by its dimensions and density, but also by the storage format used. In the dense representation, a $32{,}000 \times 32{,}000$ double-precision matrix requires

$$32{,}000^2 \times 8 \text{ bytes} = 8.192 \text{ GB}.$$

This exceeds the memory capacity of many GPUs and places significant pressure on CPU memory bandwidth.

In the BSR [18] format, only the nonzero blocks are stored, which drastically reduces the footprint compared to the dense case. For the given dataset, with fully dense 5×5 blocks and a global density of 5%, the total number of blocks is

$$\frac{32{,}000^2 \times 0.05}{5 \times 5} = 2{,}048{,}000.$$

The values associated with these blocks require

$$2{,}048{,}000 \times 25 \times 8 \text{ bytes} = 409.6 \text{ MB}.$$

Additional indexing structures are modest in size thanks to the block organization. The row pointer array stores $32{,}000/5 + 1 = 6{,}401$ 32-bit integers, consuming about 25.6 KB, while the block indices array stores one 32-bit integer per block, adding 8.192 MiB. Altogether, the BSR representation of the matrix requires roughly 417.82 MB, just under 5.1% of the dense storage even after adding the x and y vectors, each with 256 KB each.

5.2 The Benchmarking Methodology

All measurements are based on repeated executions of the SpMV [22] operation to ensure statistical stability. Each test run consists of 200 consecutive SpMV calls, with the total execution time of the batch measured as a whole. This design ensures that execution timings reach the order of seconds, reducing sensitivity to timer granularity and system interrupts. To capture variability and warm-up effects, five independent runs are performed for each implementation. Special care is taken to minimize external factors that could bias the results, including cache warm-up, repeated memory allocation, and GPU context initialization.

The methodology is applied uniformly across both platforms, encompassing CPU implementations (dense NumPy [8] reference, SciPy [30] sparse baseline, and optimized parallel BSR [18]) and GPU implementations (cuSPARSE [14] BSR reference and custom BSR kernels). By keeping the dataset and benchmarking methodology consistent, the impact of algorithmic choices and hardware features is isolated and can be clearly interpreted.

6 CPU IMPLEMENTATION FOR BSR SPMV

Before moving to GPU architecture and acceleration, it is essential to establish a solid CPU baseline for BSR SpMV [22]. Developing and evaluating a sequential and parallel CPU implementation provides a clear point of reference against which GPU performance can later be interpreted. The CPU baseline also ensures methodological transparency: rather than treating the GPU results in isolation, they are grounded in a progression from a simple, well-understood implementation toward more advanced architectures. In this way, the CPU BSR SpMV serves both as a conceptual stepping stone and as a practical benchmark for validating the correctness and significance of the GPU-focused contributions that follow.

6.1 Used Technologies

The CPU implementations of BSR SpMV [18] were developed in C [1], chosen for its combination of readability and low-level control over memory and execution logic. This minimal abstraction facilitates a clear understanding of the algorithmic behavior and ensures that performance-critical operations can be directly analyzed and optimized.

Compilation was performed using GCC [27] 8.5.0 with the `-O3` flag, enabling single-threaded optimizations such as loop unrolling, SIMD vectorization, and instruction reordering. These optimizations enhance performance while preserving the focus on algorithmic design rather than on low-level micro-optimizations.

Parallelization was implemented with OpenMP 4.5 [7], which enables multithreaded execution through compiler directives (`#pragma`). OpenMP naturally supports both shared-memory and task-parallel programming paradigms, offering mechanisms to distribute workloads efficiently across threads while minimizing synchronization and inter-thread communication overhead. These capabilities make OpenMP particularly suitable for BSR SpMV, where careful thread-level distribution and task scheduling play a critical role in overall performance.

6.2 Algorithmic Strategy

The CPU implementations of BSR SpMV [18] developed in this work build upon the existing C++ [21] implementation from the SciPy library [30]. While serving as a starting point, the SciPy code [29], which only implemented additive SpMV ($y \leftarrow Ax + y$), was adapted and extended in C to implement the BLAS-style SpMV ($y \leftarrow \alpha Ax + \beta y$) and to provide a controlled environment for exploring algorithmic and parallelization strategies. Several design

choices are common across all implementations presented in this chapter.

6.2.1 General Implementation Choices

Before discussing algorithmic details, it is important to note that our implementation departs from SciPy's C++ template–based design [29]. To port the implementation to C without losing code reusability across data types, we replaced templates with C macros. The data type `T` was fixed to `double`, the index type `I` to `int32_t`, and the larger index type for addressing contiguous memory (SciPy's `npy_intp`) to `int64_t`. These choices maintain consistency with SciPy's conventions while allowing us to port the code directly to the C [1] programming language.

The baseline implementation follows the same two-level iteration structure as SciPy's `bsr_matvec`: an outer loop iterating over block rows and an inner loop iterating over blocks within each block row. To adapt it to the BLAS-style operation $y \leftarrow \alpha Ax + \beta y$, the output vector y is first scaled by β. Partial results from multiplying each block-row by the vector x are accumulated in a temporary buffer `sum_private`, which is then scaled by α and added to the corresponding segment of y. The algorithm is shown in Listing 2 and the used GEMV routine is shown in Listing 1. The variable names and indexing conventions follow those in SciPy's implementation [ref].

```
1 void gemv(y, Ax, x, m, n):
2     for i = 0 to m-1:
3         for j = 0 to n-1:
4             y[i] += A[i*n+j] * x[j]
```

Listing 1: The simple GEMV routine used by SciPy in BSR SpMV for the multiplications of an individual block of nonzeros

```
// Scale `y` by `beta`
scale(y, beta)

// Iterate across the rows of blocks
for i = 0 to n_brow-1:
    // Initialize accumulator vector
    sum_private[:] = 0
    // Iterate across the blocks in row `i`
    for jj = Ap[i] to Ap[i+1]-1:
        j = Aj[jj]
        // Accumulate block-vector product in `sum_private`
        gemv(sum_private[:], Ax[jj], x[j*C : j*(C+1)])
    // Adds the partial result in the `y` vector
    y[i*R : (i+1)*R] += alpha * sum_private
```

Listing 2: Baseline sequential BLAS-style BSR SpMV

The notations `Ax`, `Aj`, and `Ap` represent the arrays of blocks of nonzero values, column indices, and row pointers, respectively, as introduced in Section 4.2 describing the BSR format in SciPy, `R` and `C` representing the number of rows and columns of the blocks of nonzeros.

All of our parallel implementations on CPU are based on this baseline strategy. Unlike in the sequential case, parallel tasks must often use atomic addition operations when accumulating partial results in the y vector to avoid race conditions. Because atomic operations are relatively expensive on CPUs, we minimize their overhead by first gathering partial results in the thread-private buffer `sum_private` and only performing a small number of atomic updates per block row. This reduces the number of atomic additions to $O(M)$ in an algorithm of complexity $O(nnz)$, with M denoting the number of rows and nnz the number of nonzero values, where typically $nnz \gg M$.

6.2.2 Scheduling and Task Granularity

Parallel performance in BSR SpMV depends critically on how rows of blocks are distributed among threads, how partial sums are accumulated, and how atomic updates to y are minimized. Task definition and granularity directly influence load balance and synchronization overhead, making them central to the efficiency of every parallel implementation.

The subsections that follow present a progression of strategies, from simple row-level parallelism to more refined schemes that balance work at controlled granularities. These implementations illustrate the trade-offs between static and dynamic scheduling, coarse- and fine-grained task definitions, and ultimately lay the groundwork for the GPU algorithm. Ex-

ploring these CPU strategies not only optimizes host performance, but also establishes a stepping stone toward the block-agnostic design required for accelerator implementations.

Granular Row

The simplest strategy to parallelize BSR SpMV is to dynamically assign each thread a whole row of blocks at a time, accumulating partial results locally, and then writing them back to the y vector. Compared to the sequential baseline, this requires only minimal code changes: one directive to define the parallel region and another to parallelize the outer loop. Since each row is owned by a single thread, no synchronization or atomic operations are needed when adding the partial result to y. The algorithm is shown in the Listing 3.[1]

```
// Begin parallel region
#pragma omp parallel
    scale(y, beta)

    #pragma omp for schedule(dynamic, 1)
    for i = 0 to n_brow-1:
        // Initialize thread-private accumulator vector
        sum_private[:] = 0
        // Iterate across the blocks in row `i`
        for jj = Ap[i] to Ap[i+1]-1:
            j = Aj[jj]
            // Accumulate block-vector product in `sum_private`
            sum_private[:] += gemv(Ax[jj], x[j])
        // Adds the partial result in the `y` vector
        y[i*R : (i+1)*R] += alpha * sum_private
// End parallel region
```

Listing 3: CPU approach 1 - Granular Row

Although this eliminates the need for explicit synchronization when updating the output vector, it introduces task granularity that depends directly on row size. If rows vary significantly in the number of blocks, tasks may become either too coarse (causing load imbalance) or too fine (causing medium synchronization overhead due to frequent scheduling). This limitation motivates more balanced strategies that will be introduced in the following sections.

[1] "Listings use a pseudocode style resembling C/CUDA with Python-like simplifications. Types, templates, and irrelevant boilerplate are omitted for the clarity of the algorithms. // denotes comments, while # is reserved for preprocessor directives."

Naive Intra-Row Splits

This approach takes the opposite strategy to Granular Row: instead of assigning each row of blocks to an individual thread, it parallelizes the inner loop over the blocks within a row. In this way, multiple threads collaborate on the same row, each processing a subset of its blocks. The purpose is to achieve finer-grained load balance, especially for rows that contain many blocks.

The guided scheduling provided by OpenMP is used to dynamically adapt the chunk size: larger chunks when there is plenty of work, and smaller chunks when the work becomes scarce, down to a specified minimum (set to 1 block to ensure fair distribution). This adaptive splitting ensures that no thread is left idle while others process long rows. The algorithm is shown in Listing 4.

```
// Begin parallel region
#pragma omp parallel
    scale(y, beta)

    for i = 0 to n_brow-1:
        // Initialize thread-private accumulator vector
        sum_private[:] = 0
        #pragma omp for schedule(guided, 1)
        for jj = Ap[i] to Ap[i+1]-1:
            did_work = true
            j = Aj[jj]
            // Accumulate block-vector product in `sum_private`
            sum_private[:] += gemv(Ax[jj], x[j])
        sum_private[:] *= alpha
        // Adds the partial result in the `y` vector
        #pragma omp atomic
        y[i*R : (i+1)*R] += sum_private
// End parallel region
```

Listing 4: CPU approach 2 - Intra-Row Splitting - Naive

Although this method distributes the workload more evenly, it suffers from serious synchronization overheads. First, the OpenMP [7] parallelization directive inside the main loop inserts an implicit barrier at the end of each row's inner loop, forcing every thread to wait even if many had only a few iterations to process. Second, because multiple threads now contribute to the same row, the final accumulation step requires atomic updates to the corresponding entries of y. The implicit barrier also ensures that these atomic additions occur almost simultaneously across threads, creating heavy contention on just a handful of memory locations.

Together, these two effects erase the expected benefits of intra-row parallelism: instead of scaling with more threads, performance quickly stagnates and even degrades.

Intra-Row Splits - Improved

This approach directly improves on the Naive Intra-Row Splits version and brings performance close to the final methods. The key change is the addition of the `nowait` clause to the OpenMP directive that parallelizes the inner loop. This removes the implicit barrier at the end of the inner loop, eliminating the main source of unnecessary synchronization overhead.

A second improvement concerns task granularity. To avoid excessive synchronization from overly fine tasks, we compute the chunk size by dividing the total number of nonzero blocks by the product of the number of threads and a target number of chunks per thread (a tunable input parameter). This ensures that the loop is split into tasks of a reasonable size while still adapting to the actual distribution of blocks per row.

Finally, we reduce contention on the output vector y by ensuring that only threads that performed work for a row of blocks actually issue atomic additions for said row. Each thread sets a flag (`did_work`) when it accumulates into its private buffer; if no work was done, no atomic update is performed. This prevents unnecessary contention to y. The resulting algorithm is shown in Listing 5.

```
// Set the chunk size
chunk_size = max(1, nnbz / (num_threads * chunks_per_thread))

// Begin parallel region
#pragma omp parallel
    scale(y, beta)

    for i = 0 to n_brow-1:
        did_work = false
        // Initialize thread-private accumulator vector
        sum_private[:] = 0
        #pragma omp for schedule(guided, chunk_size) nowait
        for jj = Ap[i] to Ap[i+1]-1:
            did_work = true
            j = Aj[jj]
            // Accumulate block-vector product in `sum_private`
            sum_private[:] += gemv(Ax[jj], x[j])
        if did_work:
            sum_private[:] *= alpha
            // Adds the partial result in the `y` vector
            #pragma omp atomic
            y[i*R : (i+1)*R] += sum_private
// End parallel region
```

Listing 5: CPU approach 3 - Intra-Row Splitting - Improved

This approach strikes a balance between the two earlier strategies: it minimizes multithreading overhead (as in the Granular Row method) while maintaining good load balance across threads (as in Naive Intra-Row Splits). The modifications to the sequential code are still minimal, yet the gains are significant, illustrating the power of OpenMP even when using only a small subset of its features.

There are, however, a few minor limitations. The flag `did_work` must be set explicitly and, because OpenMP does not expose a direct way to test whether a thread received iterations in a given `omp for` loop, we had to set it at every iteration of the inner loop. Moreover, all threads still iterate through the outer loop, which introduces some unnecessary work compared to the following approaches, which partition rows more explicitly.

Fixed Maximum Task Size

Unlike the previous methods in which the OpenMP runtime implicitly controlled task splitting, this approach employs explicit task parallelism, giving the program direct control over how work is partitioned. Threads are no longer merely workers, but also task producers that actively generate the units of work they will execute.

Each row of blocks is assigned to a thread, as in the Granular Row method, but instead of processing the row immediately, the thread subdivides it into smaller chunks using the `omp task` directive. The clause `firstprivate(i, start)` ensures that each task is created with its own copy of these variables, so that it knows which row and block range to process independently. Within each task, the thread-private accumulator vector `sum_private` is initialized, used to accumulate results from the assigned blocks, and then atomically added to the output vector y. The chunk size is computed in the same way as in the Intra-Row Splits - Improved method. The resulting algorithm is shown in Listing 6.

```
// Set the chunk size
chunk_size = max(1, nnbz / (num_threads * chunks_per_thread))

// Begin parallel region
#pragma omp parallel
    scale(y, beta)

    #pragma omp for schedule(dynamic, 1)
    for i = 0 to n_brow-1:
        for start = Ap[i] to Ap[i+1]-1 step chunk_size:
            #pragma omp task firstprivate(i, start)
                stop = min(Ap[i+1], start+chunk_size)
                // Initialize thread-private accumulator vector
                sum_private[:] = 0
                // Iterate across the blocks in row `i`
                for jj = start to stop-1:
                    j = Aj[jj]
                    // Accumulate block-vector product in `sum_private`
                    sum_private[:] += gemv(Ax[jj], x[j])
                // Adds the partial result in the `y` vector
                sum_private[:] *= alpha
                #pragma omp atomic
                y[i*R : (i+1)*R] += sum_private
// End parallel region
```

Listing 6: CPU approach 4 - Fixed Maximum Task Size

This approach introduces a couple of concrete improvements. First, threads now iterate over the rows of blocks only once (rather than redundantly, as in Intra-Row Splits - Improved). Second, the use of explicit tasks eliminates the need for an auxiliary flag to track whether a thread contributed work to a given row.

At the same time, the main value of this approach is not raw performance, but the opportunity to explore task scheduling in a controlled manner, laying the foundation for subsequent strategies.

Fixed-Size Tasks

This method takes the idea of explicit task control one step further by enforcing a fixed-size splitting strategy: every task is guaranteed to process the same number of blocks, regardless of how these blocks are distributed across rows. Unlike the previous approaches, a task may now span multiple rows of blocks, while still treating each row in isolation when accumulating partial results.

As in the Intra-Row Splits - Improved and Fixed Maximum Task Size methods, the algorithm uses the tunable parameter `chunks_per_thread`. Unlike before, however, this parameter is strictly enforced through a start-stop strategy: for each task, a global block index range is computed based on its position among the total number of chunks, and the stopping row is then located accordingly. This ensures perfectly uniform task sizes while preserving row-level correctness. The resulting algorithm is shown in Listing 7.

```
// Store the number of chunks
chunks_cnt = num_threads * chunks_per_thread

// Begin parallel region
#pragma omp parallel
    scale(y, beta)

    #pragma omp single
        start_row = 0
        stop_row = 0
        start = 0
        for chunk_index in 0 to chunks_cnt-1:
            // Compute the `start` and `stop` indices of the current task
            start = (nnbz * chunk_index) / chunks_cnt
            stop = (nnbz * (chunk_index + 1)) / chunks_cnt
            // Find the row where the current task ends
            while Ap[stop_row + 1] < stop:
                ++stop_row

            #pragma omp task firstprivate(start_row, stop_row, start, stop)
                // Only one row
                if start_row == stop_row:
                    one_row_of_blocks(sum_private[:], start_row, start, stop)
                else:
                    one_row_of_blocks(sum_private[:], start_row, start,
                    ↪  Ap[start_row + 1])
                    for i in start_row+1 to stop_row-1:
                        one_row_of_blocks(sum_private[:], i, Ap[i], Ap[i +
                        ↪  1])
                    one_row_of_blocks(sum_private[:], stop_row, Ap[stop_row],
                    ↪  stop)
            // The next task starts where the current task ended
            start_row = stop_row
            start = stop
// End parallel region
```

Listing 7: CPU approach 5 - Fixed-Size Tasks

This design also emphasizes modularity. Task partitioning is handled independently from the computational kernel in Listing 8, which remains focused solely on block-vector operations and accumulation. This explicit separation highlights a key principle: scheduling and computation are decoupled, making the kernel easier to reason about and reuse.

```
void one_row_of_blocks(sum_private[R], i, start, stop):
    sum_private[:] = 0
    for jj = start to stop-1:
        j = Aj[jj]
        sum_private[:] += gemv(Ax[jj], x[j])
    sum_private[:] *= alpha
    #pragma omp atomic
    y[i*R : (i+1)*R] += sum_private
```

Listing 8: Routine for processing a given range of blocks from a row

The strength of this approach lies in its absolute control over task granularity. It achieves two critical properties: (1) every task processes the same number of blocks, making load balancing entirely decoupled from the sparsity pattern, and (2) row boundaries remain respected during accumulation. These properties make the method directly relevant to our GPU strategies described in Chapter 8, which also rely on uniform work partitioning.

The trade-off is that task boundaries cannot be efficiently determined in parallel. Because each task must begin where the previous one ended, task generation is inherently serialized and performed by a single thread. When many fine-grained tasks are requested, this sequential preprocessing can dominate runtime, turning the scheduling phase into a new bottleneck.

Preprocessed Fixed-Size Tasks

This method refines the Fixed-Size Tasks approach by addressing its only weakness: serialized task generation inside the kernel. The core idea is to move the splitting logic into a preprocessing stage, so that task boundaries are computed once and reused across executions.

Instead of deriving start-stop positions dynamically during kernel execution, this method takes a `chunks_cnt` argument (the total number of tasks) along with two arrays: one marking the nonzero indices where each chunk begins and ends, and another marking the corresponding row indices. Dynamic scheduling in OpenMP then distributes the precomputed chunks to threads, as shown in Listing 9.

```
// Store the number of nonzeros per block
RC = R * C

// Begin parallel region
#pragma omp parallel
    scale(y, beta)

    #pragma omp for schedule(dynamic, 1)
    for chunk = 0 to chunks_cnt-1
        start_row = chunk_row[chunk]
        stop_row = chunk_row[chunk + 1]
        start = chunk_index[chunk] / RC
        stop = chunk_index[chunk + 1] / RC
        if start_row == stop_row:
            one_row_of_blocks(sum_private, start_row, start, stop)
        else:
            one_row_of_blocks(sum_private[:], start_row, start, Ap[start_row
            ↪  + 1])
            for i in start_row+1 to stop_row-1:
                one_row_of_blocks(sum_private[:], i, Ap[i], Ap[i + 1])
            one_row_of_blocks(sum_private[:], stop_row, Ap[stop_row], stop)
// End parallel region
```

Listing 9: CPU approach 6 - Preprocessed Fixed-Size Tasks

The computational kernel from Listing 8 is reused unchanged. What differs is that chunk boundaries are precomputed once, in a lightweight preprocessing kernel (Listing 10), based on the familiar `chunks_per_thread` parameter.

```
// Store the number of chunks
chunks_cnt = num_threads * chunks_per_thread
// Store the number of nonzeros per block
RC = R * C
chunk_index[0] = 0
chunk_row[0] = 0

start_row = 0
stop_row = 0

for i = 0 to chunks_cnt-1:
    stop = (nnz * (i + 1)) / chunks_cnt
    while Ap[stop_row + 1] * RC < stop:
        stop_row += 1

    chunk_index[i + 1] = stop
    chunk_row[i + 1] = stop_row
```

Listing 10: Preprocessing for Work Distribution

The memory overhead is negligible, on the order of 8 bytes per chunk of work, but the benefit is substantial. By shifting the splitting step outside the main kernel, task creation is no longer serialized at runtime, and the same partitioning can be reused across multiple BSR SpMV calls for the same matrix. This eliminates the bottleneck of the Fixed-Size Tasks method while preserving its advantages: perfectly uniform load distribution and strict respect for row boundaries.

This makes the Preprocessed Fixed-Size Tasks strategy the final and most efficient CPU implementation in our study. More importantly, it defines the work division paradigm we adopt for GPU implementations in Chapter 8, where preprocessing the work distribution is even more critical to achieving scalable performance.

6.3 Results

In this section, we present the performance results of our parallel CPU implementations of the BSR SpMV algorithm. The experiments were carried out on an Intel Xeon Gold 6326 processor (16 cores / 32 hardware threads) running Rocky Linux 8.10. For context, we also benchmark SciPy (single-threaded BSR SpMV [29]) and NumPy (multi-threaded dense GEMV [24]) as reference baselines. SciPy provides a sequential sparse reference, while NumPy [8] highlights the limitations of a dense format when compared against specialized sparse implementations.

6.3.1 Multithreaded Performance

We evaluated all our BLAS-style BSR SpMV implementations using thread counts of 1, 2, 4, . . . , 64, respectively, to examine how performance scales with increasing parallelism and how the implementations behave when additional synchronization is required without additional hardware resources (as in the 64-thread test). For implementations that support adaptable work partitioning, the number of chunks per thread was fixed to eight, a configuration found to provide a balanced workload distribution with little additional overhead. Figure 2 summarizes the execution times shown in Table 1, where SciPy [30] and NumPy [8] are included as reference baselines.

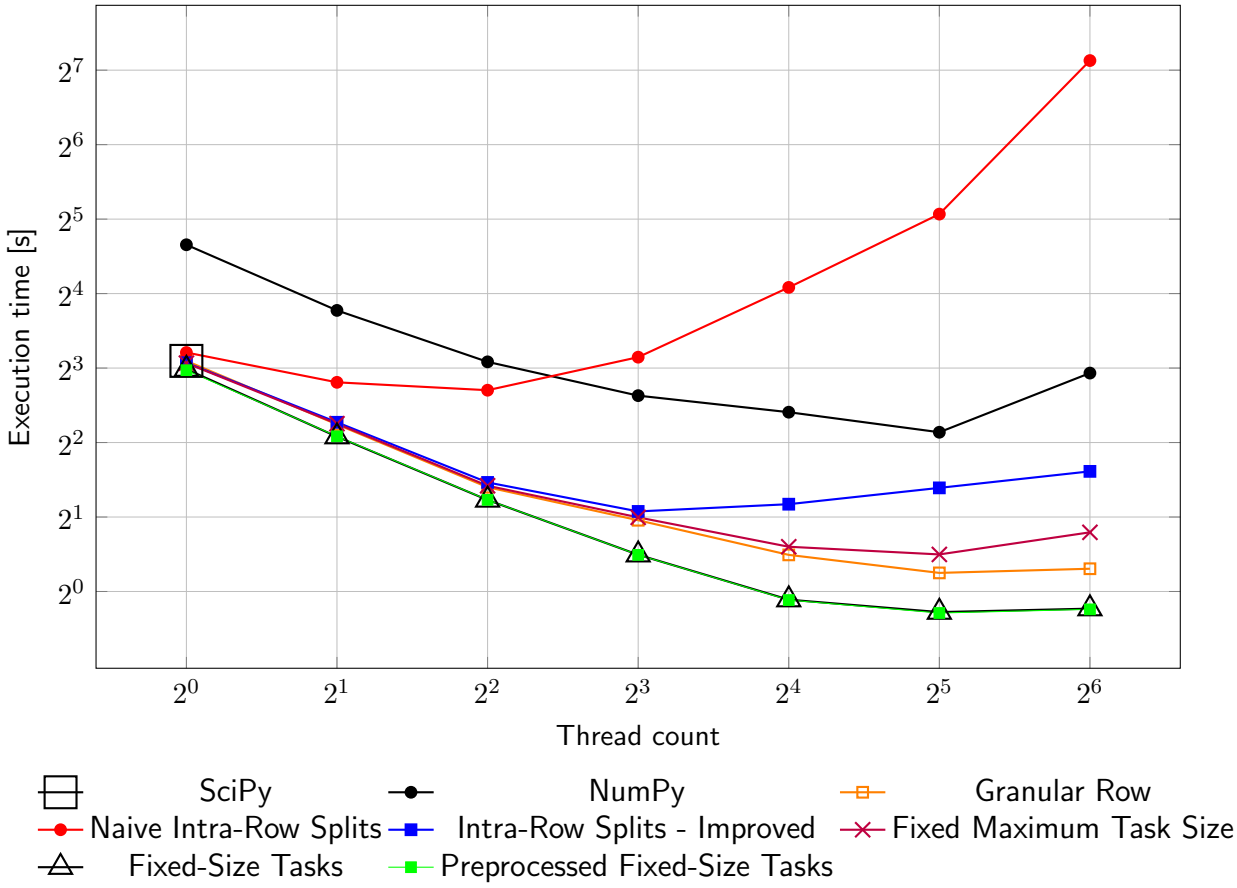

Figure 2: CPU implementations execution time graph

Table 1: Execution time in seconds for the CPU implementations

Threads	**1**	**2**	**4**	**8**	**16**	**32**	**64**
SciPy	8.55						
NumPy	25.20	13.68	8.48	6.19	5.31	4.40	7.64
Granular Row	8.55	4.73	2.64	1.94	1.41	1.19	1.24
Naive Intra-Row Splits	9.25	7.01	6.51	8.85	16.95	33.53	139.97
Improved Intra-Row Splits	8.41	4.83	2.76	2.11	2.25	2.62	3.06
Fixed Maximum Task Size	8.37	4.77	2.68	1.99	1.52	1.41	1.73
Fixed-Size Tasks	7.92	4.23	2.34	1.41	0.93	0.83	0.85
Preprocessed Fixed-Size Tasks	7.84	4.24	2.35	1.41	0.92	0.82	0.85

Baseline: SciPy and NumPy

SciPy [29] serves as a single-threaded sparse baseline, illustrating the benefits of parallelism in BSR SpMV. In contrast, NumPy [8] relies on the dense BLAS GEMV kernel, which can exploit multithreading.

As shown in Figure 2 and Table 1, most of our BLAS-style BSR SpMV implementations perform similarly to SciPy in the single-threaded case, while consistently outperforming NumPy across all thread counts. SciPy benefits from using the BSR format, achieving 8.55 s with one thread, whereas NumPy requires four threads to match this performance and reaches 4.40 s when fully utilizing the 32 logical processors of the Intel Xeon Gold 6326. Performance degrades in the 64-thread test (7.64 s), likely due to internal optimizations that assume no oversubscription - an unrealistic scenario in practical applications.

This comparison highlights two key points: (i) sparse-aware kernels remain competitive even without parallelism, and (ii) multithreading can significantly reduce the performance gap when sufficient hardware resources are available. The fact that all our multithreaded BSR SpMV implementations outperform NumPy at every tested thread count demonstrates the clear benefit of combining an efficient sparse format with careful work distribution.

Granular Row Results

As shown in Figure 2 and Table 1, the Granular Row implementation scales reasonably well, achieving the third-best overall performance despite being the simplest of our parallel designs. Its execution time decreases from 8.55 s with one thread to 1.19 s at 32 threads, demonstrating consistent sublinear scaling, with only a minor increase to 1.24 s when using 64 threads, where the number of threads exceeds the available hardware. The strength of this approach lies in the relatively low parallel overhead introduced by its basic scheduling of work.

The main limitation of this method is that the distribution of work depends directly on the

sparsity pattern of the matrix, which could result in substantial load imbalance in adversarial cases. In our dataset, this limitation was not observed, as the matrix did not contain highly irregular row structures (e.g., all nonzeros concentrated in a single row of blocks). Consequently, the method benefits from its simplicity and low coordination cost, providing a strong baseline for comparison against the more sophisticated strategies.

Naive Intra-Row Splits Results

As shown in Figure 2 and Table 1, the Naive Intra-Row Splits implementation displayed the poorest scalability among all methods. It was the only sparse implementation to perform worse than NumPy at certain thread counts and an outlier even in the benchmark with one thread, with 9.25 s compared to 8.55 s for SciPy. Performance deteriorates dramatically as the thread count increases: execution time rises to 33.53 s at 32 threads - the maximum capacity of the CPU - and further to 139.97 s at 64 threads, more than four times slower when the number of threads exceeds the available hardware resources.

The cause of this poor scalability is the barrier synchronization inserted after processing every row, which imposes a fixed overhead that quickly dominates runtime as thread count grows. While this approach guarantees a fair distribution of work, the excessive synchronization renders it impractical for high-performance execution. These results highlight that fine-grained fairness in workload division must be carefully balanced against the cost of coordination, otherwise parallel execution can become slower than the serial baseline.

Intra-Row Splits - Improved Results

The CPU approach 3 - Intra-Row Splitting - Improved implementation performed significantly better than the Naive Intra-Row Splits method across all thread counts. As shown in Figure 2 and Table 1, the Granular Row, execution time decreased steadily from 8.41 s with a single thread to 2.11 s with 8 threads. However, performance began to degrade slightly at higher thread counts, increasing to 2.25 s at 16 threads and 3.06 s at 64 threads.

The improvement over the naive approach comes from the use of the `nowait` clause in the inner `omp for` loop, which eliminates the explicit synchronization barrier after each row. However, OpenMP cannot assume the absence of data dependencies in general, so some implicit synchronization still occurs before updating the shared vector y, which likely contributes to the subtle performance degradation even at modest thread counts. This result illustrates that nesting basic OpenMP constructs like the directive `omp for`, regardless of how well optimized, have limitations that can lead to inefficient scaling.

Fixed Maximum Task Size Results

The Fixed Maximum Task Size implementation exhibits consistent sublinear scaling, as shown in Figure 2 and Table 1, the Granular Row. Execution time decreases from 8.37 s with one thread to 1.41 s with 32 threads, closely matching the number of logical processors on the Intel Xeon Gold 6326. Increasing the thread count to 64 results in a slight performance degradation to 1.73 s, reflecting the limits of oversubscription when more threads are launched than available hardware resources.

Although not our fastest approach, this implementation clearly demonstrates the benefit of task-based parallelism over nested loops annotated with `#pragma omp for`. By explicitly controlling the division of work into tasks, the method maintains good load balancing and keeps parallel overhead low, highlighting the effectiveness of task parallelism even with a relatively simple distribution strategy.

Fixed-Size Tasks Results

The Fixed-Size Tasks implementation is one of our two top-performing implementations on CPU, consistently outperforming all simpler approaches. As shown in Figure 2 and Table 1, the Granular Row, execution time decreases from 7.92 s with a single thread to 0.83 s with 32 threads, demonstrating excellent scaling up to the number of logical processors on the Intel Xeon Gold 6326. Performance with 64 threads only drops slightly, at 0.85 s, reflecting minor oversubscription effects.

This approach illustrates the critical importance of minimizing parallel overhead and explicitly controlling the distribution of work. By dividing the workload into fixed-size tasks regardless of the number of rows, it achieves balanced load and reduced coordination costs, being our first approach to outperform our more basic Granular Row implementation.

Preprocessed Fixed-Size Tasks Results

The Preprocessed Fixed-Size Tasks implementation follows the same fixed-size task distribution strategy as the previous method, but adds the preprocessing step in which tasks are created ahead of time and only need to be assigned at runtime. As shown in Figure 2 and Table 1, the Granular Row, its execution times are virtually identical to those of the Fixed-Size Tasks implementation, ranging from 7.84 s with a single thread to 0.82 s with 32 threads, and 0.85 s with 64 threads.

The near-perfect overlap of these two implementations indicates that, under our test conditions, dynamic task creation was not a limiting factor. However, the real benefit of the preprocessed approach becomes apparent when varying the number of chunks per thread, a sensitivity analysis that will be presented in Section 6.3.2.

6.3.2 Sensitivity to Task Granularity

As observed in Section 6.3.1, Figure 2 and Table 1, the Fixed-Size Tasks and Preprocessed Fixed-Size Tasks implementations consistently achieve the best performance among all CPU BSR SpMV [29] methods under typical configurations. At a fixed chunk-to-thread ratio of eight, they perform nearly identically, highlighting the importance of both balanced work distribution and minimized parallel overhead.

In this section, we examine how varying the number of tasks per thread affects execution time, and we identify the circumstances under which the two approaches diverge in performance. This analysis serves as a tie-breaker to justify the selection of a single implementation as the foundation for GPU adaptations, where synchronization costs are significantly higher and efficient task distribution is even more critical.

Tie Breaker

Figures 3 and 4, along with Tables 2 and 3, show that the Fixed-Size Tasks and Preprocessed Fixed-Size Tasks implementations perform almost identically when the number of chunks per thread remains below a threshold. For 32 threads, this threshold is around 64 chunks per thread, and for 16 threads, around 256 chunks per thread.

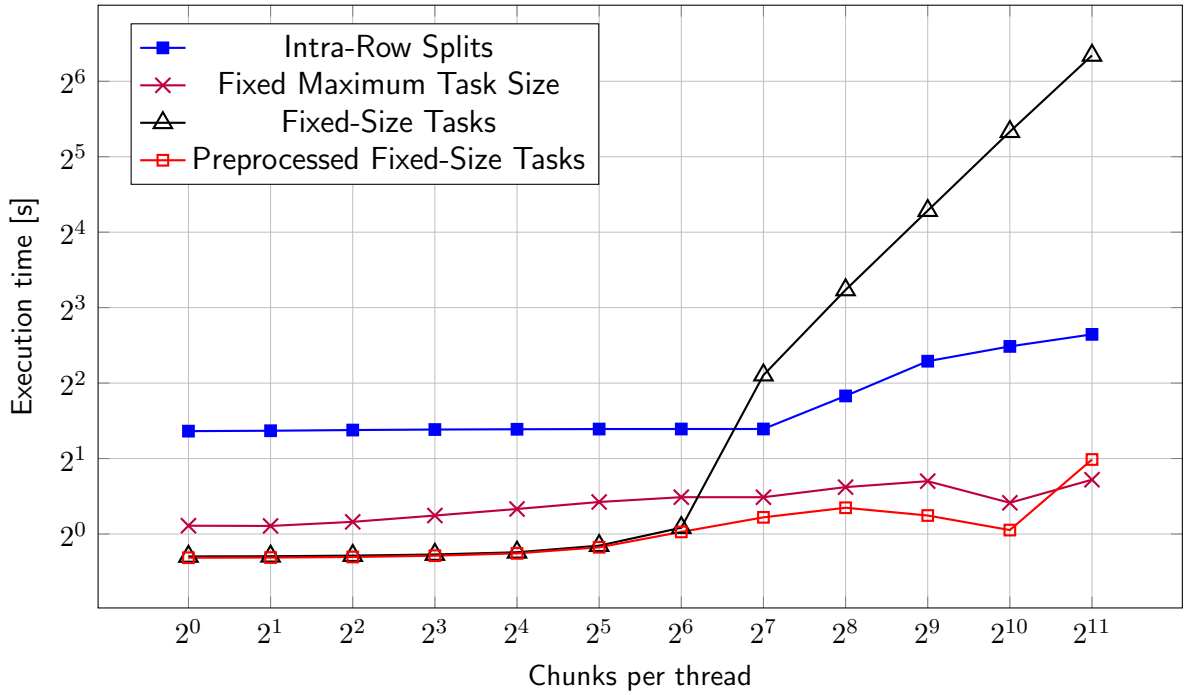

Figure 3: Sensitivity to task granularity for 32 threads

chunks to threads ratio	**1**	**8**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
Intra-Row Splits - Improved	2.57	2.61	2.62	2.62	2.62	3.55	4.89	5.60	6.26
Fixed Maximum Task Size	1.08	1.18	1.34	1.40	1.40	1.54	1.62	1.33	1.65
Fixed-Size Tasks	0.81	0.83	0.90	1.06	4.30	9.41	19.50	40.27	81.13
Preprocessed Fixed-Size Tasks	0.80	0.82	0.89	1.02	1.17	1.27	1.19	1.04	1.98

Table 2: Execution time in seconds for the sensitivity to task granularity for 32 threads

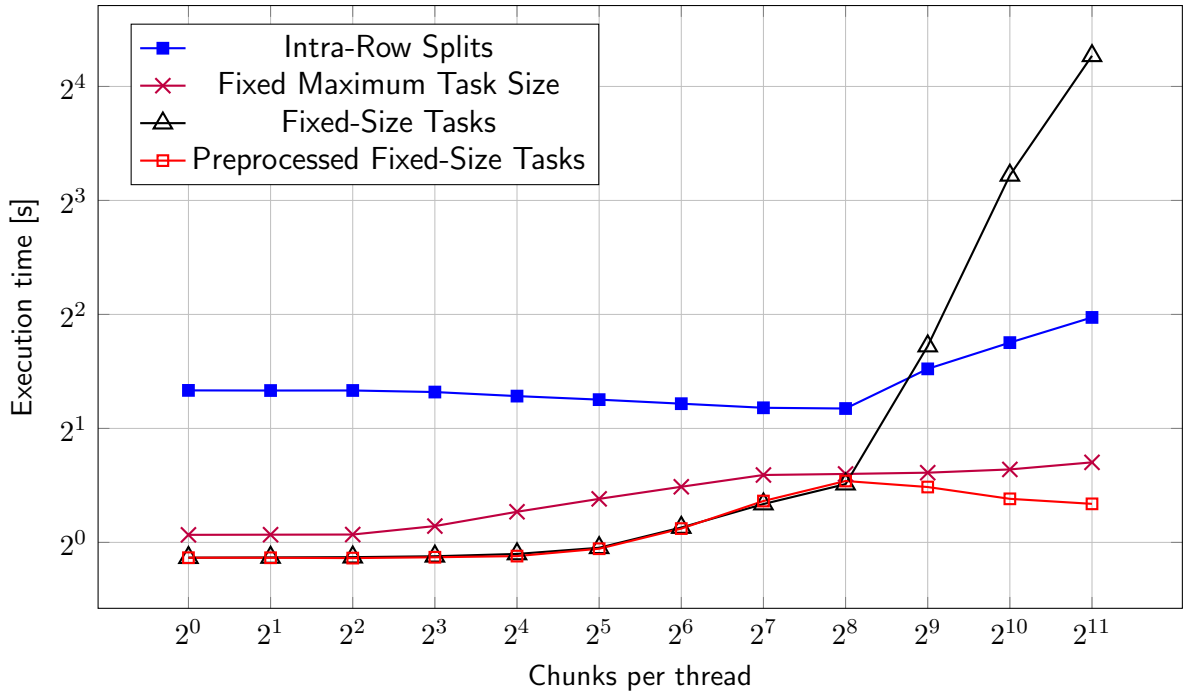

Figure 4: Sensitivity to task granularity for 16 threads

chunks to threads ratio	**1**	**8**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
Intra-Row Splits - Improved	2.52	2.49	2.38	2.32	2.27	2.26	2.87	3.37	3.93
Fixed Maximum Task Size	1.05	1.10	1.30	1.40	1.51	1.52	1.53	1.56	1.63
Fixed-Size Tasks	0.91	0.92	0.97	1.09	1.26	1.43	3.30	9.32	19.28
Preprocessed Fixed-Size Tasks	0.91	0.91	0.96	1.09	1.29	1.45	1.40	1.30	1.26

Table 3: Execution time in seconds for the sensitivity to task granularity for 16 threads

Once the threshold is exceeded, a divergence becomes apparent. For 32 threads, increasing the number of chunks per thread from 64 to 128 causes the Fixed-Size Tasks implementation to jump from 1.06 s to 4.30 s, becoming slower than both Intra-Row Splits - Improved and Fixed Maximum Task Size, and continuing to degrade as task granularity further decreases. In contrast, the Preprocessed Fixed-Size Tasks implementation exhibits only minor performance deterioration. Similarly, for 16 threads, increasing chunks per thread from 256 to 512 increases

execution time for Fixed-Size Tasks from 1.43 s to 3.30 s, while Preprocessed Fixed-Size Tasks remains stable.

This behavior highlights the impact of sequential task creation in the Fixed-Size Tasks approach: As the number of tasks grows, the one thread responsible for spawning them becomes the bottleneck, leading to dramatic slowdowns, especially on higher thread counts. By precomputing work distribution into tasks, the Preprocessed Fixed-Size Tasks implementation avoids this overhead, ensuring consistent performance across varying granularities. These results serve as a tie-breaker, clearly favoring the Preprocessed Fixed-Size Tasks approach for high-thread-count scenarios.

An OpenMP Artifact at Extreme Granularity

Figure 3 and Table 2 show an unexpected behavior when the chunk-to-thread ratio reaches 2048 with 32 threads: the Preprocessed Fixed-Size Tasks implementation slows down, taking 1.98 s, compared to 1.65 s for the Fixed Maximum Task Size implementation. Similarly, with 16 threads (Figure 4 and Table 3), execution time rises from 1.04 s for 1024 chunks per thread to 1.98 s for 2048 chunks per thread.

This slowdown is caused by the limitations of dynamic scheduling mechanism of OpenMP [7] used in the Preprocessed Fixed-Size Tasks implementation. While task execution itself remains balanced, the overhead of managing an extremely large number of fine-grained tasks begins to dominate the runtime. In contrast, the Fixed Maximum Task Size implementation delegates task creation and scheduling across all threads, benefiting from the work-stealing mechanism of OpenMP, which handles high task counts more efficiently.

This artifact is important to recognize as a limitation of CPU task management, but it does not fundamentally challenge the advantages of the Preprocessed Fixed-Size Tasks strategy for parallel BSR SpMV.

Final Decision: GPU Blueprint

The comparative analysis of the Fixed-Size Tasks and Preprocessed Fixed-Size Tasks implementations in the Tie Breaker section shows that while both approaches perform competitively under typical conditions, only the latter maintains stable performance as task granularity decreases. The severe slowdowns of the Fixed-Size Tasks method under high chunks-to-thread ratios (Figures 3 and 4, Tables 2 and 3) highlight its increased sensitivity caused by sequential task creation, a limitation likely to be magnified on GPUs where synchronization is significantly more expensive.

The Preprocessed Fixed-Size Tasks implementation, on the other hand, does incur significant scheduling overhead at extreme granularities. However, this effect stems from how OpenMP [7] implements dynamic scheduling on the CPU and will not transfer to GPU exe-

cution, where task assignment follows different mechanisms. Considering both stability and architectural portability, the Preprocessed Fixed-Size Tasks approach is chosen as the algorithmic blueprint for developing our GPU BSR SpMV kernels.

7 THE NVIDIA GPU ARCHITECTURE

In order to understand the logic behind the optimization strategies employed in the development of CUDA [25] kernels for BSR SpMV, a solid understanding of the GPU architecture of NVIDIA is required. Although GPUs and CPUs share certain similarities, their execution models diverge in critical ways, and the design principles that enable efficient performance on GPUs are often fundamentally different from those that apply to CPUs.

This chapter highlights the architectural and conceptual aspects most relevant to GPU kernel design. It begins by drawing parallels between CPUs and GPUs to establish a common ground of understanding, and then turns to GPU-specific concepts that have no true analogue in CPU architectures. Together, these discussions provide the necessary foundation for the GPU implementations presented in Chapter 8.

7.1 CPU-GPU Parallels

Although CPUs and GPUs are designed around different optimization goals, they still share several conceptual foundations. Both employ parallel execution, hierarchical memory systems, vector operations, and mechanisms for handling instruction-level behavior, but the way these concepts are implemented in hardware differs substantially.

This section explores these common foundations by drawing direct parallels between CPU and GPU architectures. The discussion focuses on how threads and execution units are organized, how memory is structured and latency is managed, how vectorization and instruction-level parallelism are expressed, and how atomic operations are supported. By following this comparative perspective, the underlying logic of GPU architecture becomes more intuitive through its resemblance - and contrast - to familiar CPU mechanisms.

7.1.1 Threading Model

Although both CPUs and GPUs rely on threads as the fundamental unit of parallel execution, the architectural meaning of a "thread" and the way threads are scheduled differ substantially between the two. CPUs typically assign one hardware thread per logical thread of execution, supported by a rich set of scheduling and preemption mechanisms. GPUs, by contrast, group threads into larger units of execution and rely on throughput-oriented scheduling rather than fine-grained control. This distinction has significant implications for how applications are mapped to the hardware, and thus for how efficient implementations must be designed.

The following subsections examine these differences and parallels in detail, starting from the basic threading unit, extending to preemption and scheduling, and concluding with broader structural comparisons and latency-hiding mechanisms.

Threading Unit

On CPUs, the natural unit of execution is the thread, which carries its own program counter, register state, and stack. In contrast, on NVIDIA GPUs the fundamental unit of execution is the **warp**: a fixed group of 32 CUDA threads that execute in lockstep under the **Single-Instruction, Multiple-Threads (SIMT)** model.

All threads in a warp issue the same instruction simultaneously, and when control flow diverges, inactive threads are simply masked out without reducing the execution cost for the warp as a whole. This is called warp divergence and it should usually be avoided in computational intensive regions of CUDA kernels.

From a programmer's perspective, this also makes it important to launch kernels with thread-block dimensions that are multiples of 32, ensuring warps are fully populated. Aligning workload distribution and memory accesses to the warp structure is one of the most basic and effective optimizations in CUDA programming. This introduces a layer of complexity absent in CPU programming, as careful consideration of warp occupancy and alignment directly influences performance in ways that have no CPU equivalent.

Building Block of Parallelism

The Streaming Multiprocessor (SM) is the fundamental building block of NVIDIA GPUs. Its role is analogous to the Core Complex Die (CCD) in Zen-based AMD CPUs, which integrates multiple cores under a shared cache. A key difference is in resource sharing: within an SM, execution units also are shared among multiple warp schedulers, whereas in a CCD, each CPU core has private execution units.

Each warp scheduler in the SM is responsible for dispatching instructions from its assigned warps to the execution units. This is conceptually similar to the hardware thread scheduler of a CPU core, which determines how hardware threads are issued to the core's execution pipelines. In both cases, the scheduler's goal is to keep execution units busy, though the underlying mechanisms and granularity differ.

Preemption

On CPUs, preemption allows the operating system to interrupt a running thread at nearly any point and schedule another. This enables responsive multitasking and fair allocation of compute resources, even when thousands of threads share only a few logical processors.

In NVIDIA GPUs, preemption is generally coarser-grained. The natural unit for preemption is the **thread block** (or Cooperative Thread Array, CTA) rather than individual threads or warps. Early architectures such as Kepler [10] only supported kernel-level preemption: a running kernel could not be interrupted until completion, except through a driver-enforced reset. Maxwell [11] and Pascal [12] refined this model by allowing different SMs to execute blocks from different kernels concurrently, but within a given SM, once a set of thread blocks from a kernel began execution, they ran to completion before blocks from another kernel could be scheduled. Later architectures (e.g. Turing [13], Volta, Ampere, Ada) introduced true block-level preemption, allowing individual thread blocks to be preempted, improving responsiveness and resource sharing. Hopper is the only architecture so far that has gone further, enabling warp-level preemption. In Hopper, the warp is not only the unit of execution, but also the unit of scheduling, allowing finer-grained control for concurrent workloads such as AI inference.

This contrast reflects a fundamental difference in the philosophy of scheduling. On GPUs, oversubscription - launching more thread blocks than the GPU can keep active at a time - helps ensure not only that there are enough warps to hide latency, but also a fair distribution of the workload. On CPUs, oversubscription is generally discouraged: having more software threads than hardware threads increases context switching and preemption overhead without utilizing more hardware resources, thus reducing performance in most cases.

Consequently, CUDA kernels should be designed to maximize uninterrupted execution within thread blocks, taking advantage of latency hiding and fine-grained parallelism rather than relying on frequent preemption to keep the GPU busy.

Latency Hiding: SMT vs Warp Sampling

On CPUs, each hardware thread maps to a private register set and program counter within a core. When Simultaneous Multithreading (SMT) is enabled, multiple hardware threads share the same core resources. The scheduler issues instructions from a different thread only when the active one stalls, thereby improving utilization of execution units without increasing the latency of any individual thread. SMT is thus opportunistic, designed to reduce idle cycles caused by stalls. [1]

In NVIDIA GPUs, thread management follows a different strategy. Each Streaming Multiprocessor maintains a large pool of up to 64 **warp contexts**, which hold the state of active warps (program counter, registers, and bookkeeping). These contexts are shared across the warp schedulers within the SM. Instead of waiting for a stall, warp schedulers **continuously sample warps** in round-robin or similar fashion, issuing instructions from whichever warps are ready. This ensures that execution units are kept busy even when many warps are waiting on memory operations or other long-latency events. Unlike SMT, which is reactive, warp

[1] SMT support varies by vendor: AMD includes it in all recent CPUs, while Intel restricts it to datacenter products; Meteor Lake was the last consumer generation to support it, and only on P-cores.

sampling is proactive: latency is hidden not by opportunistic overlap but by oversubscription and rapid switching among many resident warps.

This difference highlights the contrasting design goals of CPUs and GPUs. CPUs dedicate private hardware to a small number of threads and opportunistically multiplex resources to improve efficiency, while GPUs provision a massive shared context storage per SM and rely on constant warp sampling to sustain throughput under high latency.

7.1.2 Memory Hierarchy

The memory hierarchy in NVIDIA GPUs differs fundamentally from that of CPUs, reflecting distinct priorities in hardware design. CPUs prioritize low-latency access for a small number of threads, while GPUs employ deeper hierarchies and larger structures to sustain high aggregate bandwidth.

Rather than focusing on a single design point, both architectures layer multiple forms of memory with different latencies, bandwidths, and sharing semantics. These layers range from registers at the fastest and most private level, to caches with varying degrees of visibility across cores or SMs, and finally to large off-chip memory pools.

The following subsections detail each memory level in turn, examining their latency and throughput characteristics, how they are shared, and the implications for mapping computations onto the hardware.

Register File and Register Set

Registers provide the lowest-latency storage for active computations in both CPUs and GPUs, but their organization differs due to the respective threading models.

On CPUs, each hardware thread is mapped to a private **register set**, which includes general-purpose and vector registers. These registers are strictly private to the core, with access latency typically between 0.1 and 0.5 nanoseconds, allowing the processor to execute instructions with minimal delay. A CPU core may support one or two hardware threads (via SMT or Hyper-Threading), each with its own private register set.

On NVIDIA GPUs, each Streaming Multiprocessor maintains a large **register file**, shared among all warp schedulers in that SM. The total size of the register file is typically 65,536 registers (2^{16}), each 4 bytes in size, for a total of 256 KiB per SM. Individual warps are allocated registers from this shared pool. The access latency for the register file is around 1 nanosecond.

This comparison highlights a stark contrast: CPU registers are private to each core and extremely fast to minimize latency for a small number of threads, whereas the SM register file is large enough to keep track of tens of concurrent warps per streaming multiprocessor.

Cache Hierarchy

The cache hierarchy in GPUs can be compared to that of a traditional monolithic x86 CPU, with notable differences in sharing and coherence mechanisms.

On CPUs, each core typically has a private L1 cache for instructions and data, and also a private L2 cache as well, with a larger shared L3 cache at the chip level. These caches implement coherence protocols such as MESI (MESIF and MOESI in newer x86 CPUs) to ensure consistent read and write access across cores. Latencies vary, with L1 caches around 1–5 ns, L2 caches around 10–20 ns, and L3 caches generally 30–50 ns depending on the architecture.

In NVIDIA GPUs, each Streaming Multiprocessor has a private L1 cache, shared among its warp schedulers. Unlike CPUs, GPUs do not implement cache coherency protocols. Instead, the SM L1 cache is primarily used for read-only data and for local variables while the GPU L2 cache is the chip-wide cache that acts as the sole gatekeeper for access to the VRAM. It is responsible for maintaining coherence for read-write data across SMs. The latencies for L1 caches are typically around 20 ns, while the L2 cache latency ranges between 100 and 200 ns, depending on the GPU architecture.

Another important distinction concerns the cache line. Modern x86 CPUs use 64-byte cache lines, which serve as the granularity for coherence protocols and atomic locking. GPUs, on the other hand, employ 128-byte cache lines internally divided into four 32-byte sectors. This sectoring enables partial-line accesses, being fit both to serve large 128-byte data requests at once while also having support for finer granularity requests down to 32 bytes. It also defines the granularity at which GPU atomics are serialized at the L2 cache.

This structure emphasizes the different roles of caches in CPUs and GPUs. CPUs maintain private caches for both read and write coherence across threads at every level of the hierarchy, operating on 64-byte lines to ensure fast access and atomic consistency. GPUs, in contrast, centralize coherence at the L2 cache and restrict the L1 to read-only and SM-local local usage, operating on 128-byte sectorized lines to support warp-level memory access patterns while avoiding complex coherence protocols.

Atomic Operations on DRAM

Atomic operations are essential for inter-thread coordination, but CPUs and GPUs implement them differently due to their respective architectural priorities, their implementation being directly influenced by the underlying Cache Hierarchy.

On CPUs, performing an atomic instruction involves locking the entire cache line containing the target data by setting it to a *modifying* state. This prevents other threads from reading or writing that cache line until the operation completes. Under typical conditions, these operations are relatively inexpensive and serve well for brief inter-thread synchronization. However,

in situations with high atomic contention - even when threads access different elements residing in the same cache line of 64 bytes - these operations can dominate execution time and significantly degrade performance.

GPUs handle atomic operations differently. They have hardware atomic units that serialize operations on VRAM at the level of the L2 cache. To minimize unnecessary memory traffic, the data is propagated to VRAM only after the queue of atomic operations targeting the respective 32-byte sector is drained. This approach can mitigate contention, and in some scenarios with moderate atomic activity, atomic operations can even outperform their non-atomic counterparts due to reduced VRAM writes.

This contrast illustrates the different design philosophies: CPUs enforce strict coherence at the cache line level, optimizing for low-latency thread coordination, while GPUs employ dedicated serialization and queuing mechanisms that allow high-throughput global reductions and other synchronization patterns without excessive memory traffic.

7.1.3 Execution Model

The execution model describes how instructions are mapped to hardware resources and executed on the processor. CPUs and GPUs both rely on arithmetic and logic units (ALUs) and pipelines, but their designs differ in how data parallelism is expressed and exploited.

CPUs, vector units provide SIMD execution, but software must explicitly generate instructions to utilize them fully. On GPUs, execution is automatically vectorized at the warp level under the SIMT model, making SIMD execution the implicit behavior for groups of threads.

The following subsections examine these differences in detail, covering the organization of vector execution units and specialization of floating point hardware.

Vector Execution Units

Both CPUs and GPUs rely on vector execution units to perform arithmetic and logical operations across multiple data elements, but the way this parallelism is exposed and utilized differs significantly.

On CPUs, vector ALUs provide explicit SIMD capabilities. Modern instruction sets (such as AVX-512) allow a single instruction to operate on multiple data lanes simultaneously. However, these vector instructions are optional: scalar code can execute independently of the vector units, and software must explicitly use SIMD instructions to exploit the wider datapaths. The width of these units is typically 128, 256, or 512 bits depending on the design, corresponding to operations on up to 8 double-precision or 16 single-precision values in a single instruction. Each core contains its own set of vector units, which are shared among the threads mapped to that core according to its pipeline scheduling.

In GPUs, the ALUs are organized to support implicit SIMD execution at the hardware level on 32 lanes, forming the foundation of the SIMT model. All threads in a warp automatically utilize the ALUs without requiring explicit vector instructions in software. Warp schedulers handle the mapping of threads to ALUs, and the same hardware units are shared among multiple active warps, allowing continuous utilization of the ALUs as threads stall or resume.

This organization contrasts with CPUs not only in the implicit nature of SIMD execution, but also in how threads are scheduled and how ALU resources are shared. While CPUs require explicit software instructions to exploit vector units (which most consumer applications do not use), GPUs provide automatic, warp-level SIMD execution, ensuring efficient execution of many threads without explicit vectorization in software.

Specialization vs Generalization

Both CPUs and GPUs provide integer ALUs that can handle multiple operand sizes (e.g., 8-, 16-, 32-, and 64-bit), offering broadly similar functionality for integer arithmetic.

The main divergence appears in floating-point execution. CPUs typically employ unified floating-point units that handle both single-precision (FP32) and double-precision (FP64) operations on the same hardware. This general-purpose approach simplifies programming and maintains balanced performance across different workloads, typically up to 4 vector units per core for integer operations and 4 for floating point operations.

GPUs, by contrast, rely on specialized floating-point ALUs: distinct FP32 and FP64 units are physically implemented, often in very different ratios. Consumer GPUs dedicate most silicon to FP32 units, marketed as CUDA cores (128 per SM), while FP64 throughput is 32$\times$ lower on recent RTX 30- and 50-series GPUs. Datacenter GPUs, instead, allocate a larger share to FP64 to support scientific computing: 32 FP64 and 64 FP32 units per SM. The number of integer ALUs per SM is the same for both consumer and datacenter GPUs (64). This specialization enables extremely high throughput in the targeted precision, but also means performance can vary dramatically depending on the numerical format used. These numbers far exceed CPU counts, as does the maximum of 64 warps that an SM can keep active simultaneously (compared to 1 or 2 concurrent threads per core).

This design choice reflects the broader philosophies of the two architectures: CPUs prioritize flexibility and balanced performance, while GPUs emphasize specialization and massive throughput in their intended domains.

Hardware Support for Integer Division (and Compiler Mitigations)

Both CPUs and GPUs offer limited native support for integer division, but the mechanisms differ significantly.

On CPUs, each core typically includes only one dedicated scalar unit for integer division, with much lower throughput and higher latency than other instructions. This unit can handle integer division operations directly, providing predictable latency and throughput. Because CPUs usually execute only a few threads per core, this design is usually sufficient for general-purpose workloads, and the cost of integer division is amortized across other out-of-order instruction execution.

GPUs, in contrast, do not have a dedicated hardware unit for integer division. Instead, integer division is supported via microcode, which sequences chains of simpler arithmetic instructions to emulate the integer division operation. This approach allows the GPU to avoid dedicating expensive silicon to a complex operation, but this comes at a cost: integer division can be much slower than other arithmetic operations on GPUs, and performance can be significantly impacted in CUDA kernels that rely heavily on integer division.

Despite this, integer division is commonly used in CUDA implementations for algorithms that involve nested `for` loops (such as BSR SpMV in this work), but there is a key distinction: the divisors in such situations are often runtime constants. While GCC employs Division by Invariant Integers using Multiplication [20] only for constants known at compilation, NVCC extends this optimization even to runtime constants (as we have inferred by disassembling the obtained binaries).

The disparity in hardware support as well as compiler optimizations reflects the architectural priorities. CPUs aim for low-latency execution for a small number of threads, while GPUs optimize for high-throughput execution across thousands of threads, accepting longer latencies for less common operations like integer division, which software often mitigates via algorithmic strategies or compiler optimizations.

7.2 GPU-Specific Concepts

Although NVIDIA GPUs share many conceptual foundations with CPUs, as discussed in Section 7.1, they also introduce numerous features with no direct equivalent in a CPU. These include low-overhead intra-warp communication mechanisms, programmable memory that rivals the speed of on-chip caches, specialized cache interfaces, and execution units tailored for extremely high-throughput tasks.

This section explores these GPU-specific concepts, highlighting the architectural mechanisms that enable performance in ways unique to massively parallel workloads, and providing the foundation for understanding advanced CUDA [25] programming techniques.

7.2.1 Warp Intrinsics

Warp intrinsics are low-level instructions that allow threads within the same warp to exchange information efficiently without using shared memory. They can be invoked directly in CUDA code and require compilation with support for the target GPU architecture.

These intrinsics operate on registers and enable threads to read the values of other threads in the same warp, facilitating coordination and data sharing at very low latency. It is important to note that warp intrinsics do not allow a thread to write directly into another thread's registers; data flow is strictly controlled.

Common examples of warp intrinsics include:

- `shuffle` operations, which allow threads to exchange values with specific threads within the warp,
- `ballot`, which produces a bitmask indicating which threads in the warp satisfy a given predicate, and
- `vote`, which evaluates boolean conditions across threads and aggregates the results efficiently.

By enabling intra-warp communication without relying on shared memory (introduced in Section 7.2.2), warp intrinsics help improve performance for certain parallel algorithms that require fine-grained synchronization.

7.2.2 Shared Memory

Shared memory is a block-private scratchpad that resides on each Streaming Multiprocessor (SM) and is explicitly managed by the programmer. Unlike caches, which are transparent and hardware-controlled, shared memory is allocated and accessed through CUDA code, giving developers fine-grained control over how data is staged and reused within a block. Shared memory plays a critical role in enabling efficient intra-block communication between different warps, providing low-latency storage well-suited for intra-block communication and reductions because it guarantees that it will not generate traffic outside the SM. Overall, it fills a role that is complementary to that of the L1 cache which can only store read-only data within the SM (as discussed in subsection 7.1.2) and which might evict data depending on access pattern.

From a hardware perspective, shared memory is tightly coupled with the SM's L1 cache resources in most NVIDIA GPU architectures. The physical storage of the L1 cache can be dynamically partitioned between cache and shared memory, with configurable ratios that balance programmability and hardware-managed caching. The Pascal [12] architecture is a notable exception, as it provided dedicated hardware exclusively for shared memory, independent of the L1 cache.

Capacity is limited and varies across GPU generations, from 48 KiB in the early architectures (e.g. Kepler [10]) up to 228 KiB in the Hopper datacenter NVIDIA GPU architecture.

Internally, shared memory is organized into multiple banks to enable parallel access by threads within a warp. While this banked design increases throughput, it also introduces the potential for conflicts if multiple threads target different addresses within the same bank. These behaviors, as well as the performance of atomic operations on shared memory, will be examined in the following subsubsections.

Shared Memory Bank Conflicts

Shared memory is divided into 32 banks, each responsible for 4-byte strips of data. This design allows all 32 threads of a warp to access shared memory simultaneously if their requests map to up to one address from each bank. In such cases, the memory accesses proceed in parallel with no additional cost.

However, if multiple threads in a warp access different addresses that fall within the same bank, the Load/Store Units (LSUs) must serialize those accesses. Such accesses are defined as *bank conflicts* and they reduce effective throughput, as the requests are served one after another rather than in parallel. Bank conflicts occur when the accessed addresses differ by nonzero multiples of 128 bytes because there are 32 banks and each is 4 bytes wide (so each bank repeats every 128-byte).

Certain access patterns naturally avoid conflicts. For instance, when consecutive threads access consecutive addresses (a unit stride), each request falls into a different bank, maximizing parallelism. More generally, minimizing the stride between the addresses accessed by consecutive threads helps distribute accesses evenly across banks, reducing serialization.

While bank conflicts do not affect correctness, they can significantly impact performance. Awareness of the banked design allows programmers to structure shared memory access patterns that exploit the available parallelism.

Shared Memory Atomics

Shared memory atomics allow threads within the same block to perform operations on shared memory locations without race conditions. The Load/Store Units (LSUs) ensure atomicity for 32-bit types by serializing accesses to the target address. It is critical that atomic operations target properly aligned 4-byte regions; performing an atomic operation on a misaligned 4-byte segment may lead to undefined behavior.

For larger data types, shared memory supports only the `atomicCAS` (Compare-And-Swap) operation directly. Other atomic operations on larger types are emulated by the NVCC [15] compiler using CAS loops in PTX, ensuring correctness while maintaining atomicity at the

cost of additional instructions.

7.2.3 SM Occupancy Limitations

The number of active threads and warps that a Streaming Multiprocessor (SM) can support simultaneously is limited by several hardware-imposed factors, collectively referred to as *SM occupancy constraints*. These limitations directly influence how efficiently a GPU can hide memory latency and keep its computational units busy.

Occupancy is primarily constrained by the availability of warp contexts, thread block slots, registers, and shared memory. Each of these resources limits the number of threads and warps that can reside on an SM at any given time.

Maximizing SM occupancy is a crucial aspect of CUDA kernel performance tuning. Higher occupancy allows more warps to be scheduled when others stall on memory or other long-latency operations, improving latency hiding. The following subsubsections explore each of these limitations in detail and how they interact to determine the maximum number of threads an SM can sustain.

Warp Contexts

Each SM can maintain a limited number of active warp contexts. Starting with the Kepler [10] architecture, this number is typically 64 per SM. A warp context represents the state of one warp, and the total number of warp contexts imposes a hard limit on how many warps - and therefore threads - can be simultaneously active on the SM.

The number of warp contexts serves as a key reference point for optimizations aiming to increase per-SM occupancy. High occupancy requires careful management of resources such as registers and shared memory, as each active warp consumes a portion of these. Understanding warp context limits is essential for optimizing kernel execution and maximizing SM utilization.

Hardware Thread Block Slots

Each SM has a limited number of hardware thread block slots, which determine how many thread blocks can be scheduled concurrently. For example, on Pascal [12], an SM can support up to 32 thread blocks running simultaneously. Given that Pascal [12] has 64 warp contexts per SM, a CUDA kernel must launch at least 64 threads (2 warps) per block to allow the SM to reach maximum occupancy.

Since each thread block contains multiple warps, the number of thread block slots indirectly limits the number of active warps on an SM at any moment. Launching blocks that are too small can leave the SM underutilized, while launching overly large blocks can be counterpro-

ductive: if some warps within a block finish early, the remaining active warps will prevent another block from being scheduled. This can reduce warp occupancy even when, theoretically, the SM could accommodate more active warps. Achieving the right block size is therefore critical and depends on the kernel's workload to maximize latency hiding and overall throughput.

Shared Memory Pressure

The amount of shared memory allocated per thread block can indirectly limit the number of blocks that can run concurrently on an SM. Larger per-block allocations reduce the number of blocks that fit on the SM, potentially lowering occupancy. This effect can be mitigated by increasing the number of threads per block, up to the architectural maximum of 1024 threads (32 warps) per block.

For example, on Pascal [12] GPUs, allocating 32 KiB of shared memory per block - which is half the total available per SM - still allows up to half the maximum occupancy per SM. By choosing an appropriate combination of block size and shared memory usage, it is possible to achieve the theoretical maximum occupancy, though this comes at the cost of coarser block granularity and may affect other aspects of kernel scheduling as discussed in subsection Hardware Thread Block Slots.

Register Pressure

Register pressure arises when a CUDA kernel requires more registers per thread than the hardware can provide while maintaining high occupancy. Excessive register usage can limit the number of active threads or blocks that can reside on an SM, directly reducing the potential for latency hiding and overall throughput.

If both register pressure and block size are too large, the kernel may fail to launch. For example, on Pascal [12] GPUs, each SM has 65,536 registers. Launching a kernel that uses more than 64 registers per thread with blocks of 1,024 threads (32 warps) is impossible.

Programmers can mitigate register pressure by reducing the number of local variables in the kernel, limiting loop unrolling, or restructuring computations to reuse registers. Additionally, the NVIDIA compiler (NVCC) [15] allows explicit limits on per-thread register allocation via the `--maxrregcount` option. While this can increase the maximum occupancy, any variables that cannot fit in the physical register file are spilled to local memory in VRAM, resulting in high-latency accesses that can significantly degrade kernel performance.

Careful balancing of register usage and occupancy is therefore essential. Optimal performance often involves finding a sweet spot where enough threads remain active to hide latency without causing excessive register spills.

7.2.4 Coalescing

Global memory accesses by threads in a warp are handled by the Load/Store Units (LSUs), which attempt to combine individual memory requests into as few transactions as possible. This process is called *coalescing*. Coalescing is crucial for achieving high memory throughput, as each uncoalesced request can generate additional memory transactions and reduce effective bandwidth utilization.

Coalescing is most efficient when consecutive threads in a warp access consecutive addresses, ensuring that the resulting memory requests do not cross 128-byte boundaries. This behavior is analogous to avoiding Shared Memory Bank Conflicts: small strides between the addresses accessed by threads allow the hardware to merge requests efficiently, while large strides or irregular access patterns may result in multiple separate transactions.

CUDA kernel performance can therefore be improved by carefully arranging the layout of data in global memory and structuring thread accesses to minimize stride and align with 128-byte boundaries. By doing so, coalescing ensures that global memory bandwidth is used effectively, complementing other high-throughput mechanisms such as shared memory and warp-level communication.

8 GPU IMPLEMENTATION OF BSR SPMV

This chapter focuses on the implementation and optimization of Block Sparse Row (BSR) matrix-vector multiplication on NVIDIA GPUs. The purpose is to evaluate how GPU-specific architectural features can be leveraged to accelerate this operation, and to compare the performance of custom kernels against established library implementations.

The discussion begins with the technologies used to develop and compile the GPU code, followed by the algorithmic strategies that guided the design of the implementations. Several custom kernels are introduced, each improving upon the previous by applying optimizations tailored to GPU execution. The chapter concludes with a performance analysis, highlighting both the incremental benefits of individual optimizations and the overall competitiveness of the implementations.

8.1 Used Technologies

The GPU implementations of BSR SpMV were developed using **CUDA**, NVIDIA's programming model for general-purpose GPU computing. CUDA provides a C/C++ extension that allows fine-grained control over GPU execution, including the organization of threads into warps and thread blocks, explicit use of shared memory, and synchronization primitives such as `__syncthreads`. This level of control is essential for optimizing sparse linear algebra kernels, where memory access patterns and latency hiding dominate performance.

Compilation was carried out with **NVCC**, NVIDIA's CUDA compiler. NVCC not only translates CUDA kernels into PTX and device-specific SASS instructions, but also integrates NVIDIA's evolving optimization heuristics for each hardware generation. This allows the programmer to focus on high-level kernel design and algorithmic optimization, while leaving generation-specific micro-optimizations to the compiler.

CUDA was chosen because it provides direct access to GPU-specific aspects and optimizations discussed in Chapter 7. In particular, it enables:

- **Memory access control**, enabling the design of coalesced load patterns, and the use of shared memory as a scratchpad.
- **Low-level synchronization and reduction tools**, including warp intrinsics for intra-warp communication and block-level synchronization with `__syncthreads`.
- **Incremental portability**, as kernels written once can be recompiled by NVCC to target newer GPU architectures with minimal code changes.

By combining programmer-managed optimizations with compiler-driven fine-tuning, CUDA and NVCC offer the right balance between **control** and **portability**, making them the natural choice for implementing and testing BSR SpMV on GPUs.

8.2 Testing Methodology

To ensure that the performance results presented later in this chapter are both reliable and reproducible, the methodology used to evaluate the custom GPU kernels is first described here. The measurement of performance follows the approach established in Chapter 5, while this section introduces the experimental environment and the rationale behind kernel launch configurations. With this foundation in place, the next section on algorithmic strategies (Section 8.3.2) can focus on the design and incremental refinement of the kernels, and Section 8.4 can provide a consolidated interpretation of their performance.

8.2.1 Benchmarking Setup

All experiments were conducted on an NVIDIA Tesla P100 PCIE 16 GB GPU. The GPU has 56 streaming multiprocessors (SMs), each with 65 536 32-bit registers, 32 FP64 units, 64 INT32 units, 24 KiB of unified L1/Read-Only/Constant cache (8 KiB limit for the constant cache), and access to 4 MB of L2 cache. Peak memory bandwidth is 732.2 GB/s via a 4096-bit bus (four 1024-bit HBM2 stacks).

Kernels were compiled with CUDA 11.4 [25] using NVCC [15], targeting the Pascal [12] architecture (`-arch=sm_60`). Register usage was limited to 32 registers per thread with `--maxrregcount=32` unless otherwise noted. This value allows maximum SM occupancy of 64 active warps.

Only hardware characteristics relevant to double-precision computation and integer indexing are reported. The host environment was prior mentioned in Section 6.3 and has no measurable influence on GPU kernel performance.

8.2.2 Launch Configuration

The performance of a CUDA kernel depends heavily on how threads and blocks are organized during launch. Both the size of each thread block and the total number of blocks per grid determine how work is distributed across the GPU, influence occupancy and register usage, and affect opportunities for and inter-warp cooperation. This section provides a high-level view of these factors and the principles guiding the choice of launch configurations, before delving into the specifics of block sizing and block counts in the following subsections.

Choice of Thread Block Sizes

The number of threads per block in a CUDA kernel launch has a direct impact on GPU performance. It influences shared memory usage when demand is per-block, reduces or increases atomic contention depending on block size, and affects the opportunities for inter-warp aggregation of partial results. Smaller blocks allow more blocks to be scheduled per SM, improving load balancing, while very large blocks increase intra-block collaboration but may limit scheduling flexibility.

In this work, only power-of-two block sizes were used: 32, 64, 128, 256, 512, and 1024 threads per block. These choices are standard because they align with warp boundaries and simplify memory coalescing. The suitability of each size, however, depends on resource constraints such as registers and shared memory demands, which in turn control how many blocks can reside simultaneously on an SM.

Rationale Behind Thread Block Counts

Beyond block size, the number of blocks per grid determines how work is distributed across SMs. The baseline is to launch exactly as many blocks as needed to fully occupy all SMs, but oversubscription - launching more blocks than can be resident at once - can improve load balancing by allowing the scheduler to assign new work to SMs as soon as resources free up. In this study, five oversubscription levels were tested: $1\times$ (full occupancy only), $2\times$, $4\times$, $8\times$, and $16\times$. While oversubscription can help mitigate imbalance, it also increases kernel launch overhead and, in extreme cases, may fail if register demand is too high.

Register pressure is the primary factor limiting feasible launch configurations. When a kernel requires more registers per thread, fewer warps can be scheduled concurrently on an SM. For instance, with 77 registers per thread, each warp uses $32 \times 77 = 2464$ registers, but allocation is rounded up to the next multiple of 256, i.e. 2560 registers per warp. On a GPU with a 65,536-register file, only 25 warps can be active per SM. This directly constrains the possible launch configurations.

For the Tesla P100 with 56 SMs, the resulting optimal non-oversubscribed launch configurations under this register usage are shown in Table 4. These configurations illustrate how register allocation granularity and occupancy considerations combine to produce nonstandard but optimal ratios of thread blocks per SM.

8.2.3 Measurement of Performance

The primary metric used to evaluate the CUDA implementations is the kernel runtime, measured as the elapsed time for the GPU kernel execution alone. Host-to-device and device-to-host data transfers are excluded, as they are performed only for data initialization and result

Threads per Block	Warps per Block	Blocks per SM	Grid Size
512	16	1	56
256	8	3	168
128	4	6	336
64	2	12	672
32	1	25	1400

Table 4: Optimal launch configurations example for a CUDA kernel that uses 77 registers per thread on a GPU with 56 Streaming Multiprocessors

verification, and do not reflect the computational performance of the kernels themselves.

Execution times exhibited very low variability, typically under 1% and often below 0.1%. The values reported in the tables correspond to the average of repeated runs, ensuring that the performance data are both reliable and comparable across different kernel configurations and implementations.

8.3 Algorithmic Strategy

The GPU implementations of BSR SpMV presented in this chapter are built upon the scheduling and work distribution strategy introduced in Chapter 6, which ensures a balanced and regular decomposition of the computation. This shared foundation is described in detail in Section 8.3.1.

Beyond this common baseline, the implementations diverge in how they exploit GPU-specific features to maximize performance. These differences concern memory *access patterns* and the *reduction strategies* used to aggregate partial results, aspects that are discussed in Section 8.3.2.

8.3.1 General Implementation Choices

The GPU implementations of BSR SpMV presented in this chapter build directly on the Preprocessed Fixed-Size Tasks scheduling strategy introduced in Chapter 6. This CPU implementation was identified as the most suitable baseline for GPU porting due to its balance between regularized workload distribution and minimized preprocessing overhead, as discussed in Section 6.3.

On the GPU, all our custom implementations adopt this scheduling framework as their foundation, ensuring that tasks are consistently defined and parallelism can be efficiently exploited. However, direct reuse of the CPU kernel is not sufficient. The computational kernel and the work distribution scheme required specific adaptations to accommodate the constraints and opportunities of GPU execution described in chapter 7. These adaptations make it possible to

exploit GPU-specific optimizations, such as coalesced memory accesses, warp-level parallelism, and efficient synchronization.

Adaptation of the Computational Kernel

The original computational kernel, shown in Listing 8, was designed for CPU execution. It was called independently by multiple threads and iterated over the blocks of nonzeros, delegating the inner work to the GEMV routine in Listing 1, which iterated over the individual entries inside each block.

This block-oriented structure, while efficient on CPUs, is fundamentally incompatible with GPU optimization strategies that rely on multiple threads cooperating on contiguous data. In particular, techniques such as coalesced global memory accesses and the avoidance of shared memory bank conflicts require aligned and predictable memory access patterns across warps. A block-based iteration scheme leads to irregular access and prevents efficient use of these optimizations.

```
void one_row_of_blocks(sum_private[R], i, start, stop):
    sum_private[:] = 0
    for index = start_index to stop_index-1:
        // The block index
        jj = index / RC
        // The intra-block index
        index_b = index % RC
        // The block column
        j = Aj[jj]
        // The intra-block row and column indices
        i_b = index_b / C
        j_b = index_b % C
        // The column coordinate of the current nonzero inside the matrix
        j_final = C * j + j_b
        sum_private[i_b] += Ax[index] * x[j_final]
    sum_private[:] *= alpha
    #pragma omp atomic
    y[i*R : (i+1)*R] += sum_private
```

Listing 11: Computational kernel adapted for being ported to CUDA

Adapted Kernel Design. To address this, the computational kernel was adapted into the form shown in Listing 11. The two nested loops from the GEMV routine, together with the block-level loop from the CPU kernel, were collapsed into a single loop that iterates directly

threads	Preprocessed Fixed-Size Tasks	Adapted for GPU Porting
1	7.84	47.38
2	4.24	23.70
4	2.35	11.86
8	1.41	6.11
16	0.92	3.16
32	0.82	3.15
64	0.85	3.26

Table 5: Execution time in seconds for the comparison between the adapted computational kernel and the initial one

over the nonzeros. This restructuring exposes a flat, contiguous access pattern that aligns naturally with GPU execution, providing the foundation for coalescing and other architecture-specific optimizations.

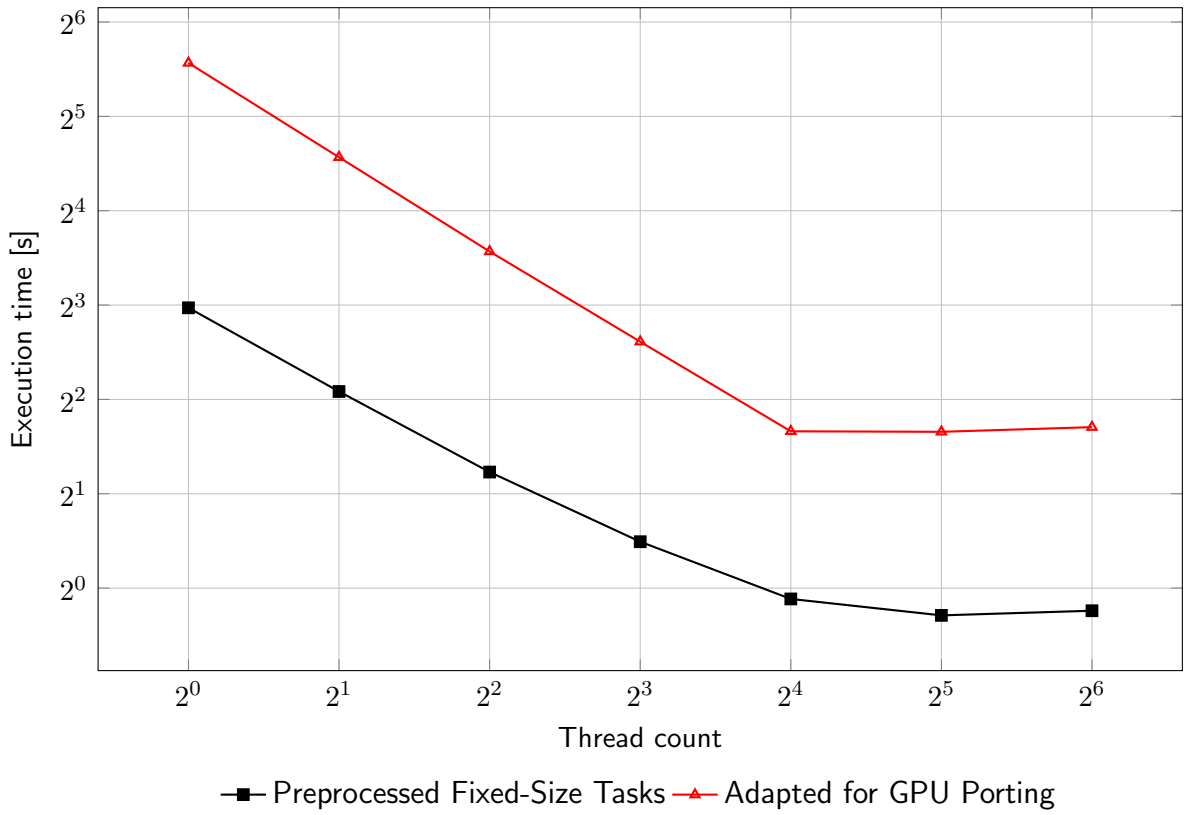

Figure 5: Comparison between the adapted computational kernel and the initial one

Overhead of Adaptation. However, this adaptation introduces an additional cost. Since block information is no longer explicit in the iteration structure, the block index and the intra-block coordinates of each nonzero must be inferred at runtime with two integer divisions and modulo operations. This more than doubles the computational work per nonzero compared to

thread count increment	Preprocessed Fixed-Size Tasks	Adapted for GPU Porting
$1 \rightarrow 2$	1.850	1.999
$2 \rightarrow 4$	1.807	1.998
$4 \rightarrow 8$	1.668	1.940
$8 \rightarrow 16$	1.523	1.931
$16 \rightarrow 32$	1.128	1.004
$32 \rightarrow 64$	0.967	0.966

Table 6: Relative speedup (relative to the performance with half the number of threads) for the final and for the Preprocessed Fixed-Size Tasks strategy compared to its adaptation for porting to CUDA

the CPU-oriented computational kernel from Listing 6.2.2. The scale of this overhead is shown in Table 5 and visualized in Figure 5. As a reminder, the CPU benchmarks were conducted on the Intel Xeon Gold 6326 processor (16 cores / 32 hardware threads), as described in Section 6.3.

Performance Characteristics. An analysis of the performance graph in Figure 5 reveals that the adapted kernel exhibits a more consistent relative speedup compared to the original CPU implementation, despite the increased computational cost. Table 6 presents the relative speedup observed each time the number of threads doubles.

For the simple computational kernel from Listing CPU approach 6 - Preprocessed Fixed-Size Tasks, the speedup varies between 1.5 and 1.9 when scaling up to 16 threads, reflecting a memory-bound scenario where latency and bandwidth dominate. The adapted kernel, however, shows near-ideal speedups (between 1.9 and 2) until going from 16 to 32 threads, where the speedup drops to 1.004. This indicates that the integer division and modulo computations have become the primary bottleneck, with the SMT (Simultaneous Multithreading) making almost no difference because CPUs only have one scalar integer division unit per core (see subsubsection 7.1.3 in Chapter 7). The slight slowdown beyond 32 threads (up to 64), common to both kernels, is expected due to oversubscription and context-switching effects.

Implications for GPU Implementations. On NVIDIA GPUs, this bottleneck does not occur. The runtime divisors in the adapted kernel are handled efficiently by NVCC, which applies the "Division by Invariant Integers using Multiplication" optimization [20]. Consequently, the adapted kernel's computational overhead from integer division is largely mitigated, allowing GPUs to leverage their massively parallel execution units and higher arithmetic throughput.

Adaptation of the Work Distribution Kernel

The algorithm that divides the work into tasks at the preprocessing stage (Listing 10) is sequential and executed once on the CPU, producing a set of tasks that will be used by all GPU kernel launches. The work distribution kernel from Listing 9 also needs only a minor adjustment, that being to scale vector y in a separate kernel. Other than that, it can be directly ported to CUDA without additional algorithmic adaptation.

```
__global__ void bsr_matvec_kernel(/* full argument list omitted for brevity */):
    // Block id
    bid = blockIdx.x
    // Block count
    bcount = gridDim.x

    // Array of block-private partial results in shared memory
    __shared__ sum_private[R]

    for chunk = bid to chunks_cnt-1 step bcount:
        start_index = chunk_index[chunk]
        stop_index = chunk_index[chunk + 1]

        start_row = chunk_row[chunk]
        stop_row = chunk_row[chunk + 1]

        if start_row == stop_row:
            one_row_of_blocks(sum_private[:], start_row, start_index, stop_index)
        else:
            // Handle the first row of blocks from the task
            one_row_of_blocks(sum_private[:], start_row, start_index, Ap[start_row + 1] * RC)

            // Handle the complete rows in between
            for i = start_row+1 to stop_row-1:
                one_row_of_blocks(sum_private[:], i, Ap[i] * RC, Ap[i + 1] * RC)

            // Handle the last row of blocks
            one_row_of_blocks(sum_private[:], stop_row, Ap[stop_row] * RC, stop_index)
```

Listing 12: The work distribution algorithm for GPU

The CUDA-ready version, shown in Listing 12, ensures that all threads within a thread block operate on the same task. The block index (`blockIdx.x`) distinguishes between thread blocks, while the total number of blocks (`gridDim.x`) determines the stepping of the outer loop. This design allows all threads within each warp - and all warps within each block - to cooperate inside the computational kernel, performing reductions in shared memory for each row within the assigned chunk of work.

Work balancing is handled by the GPU scheduler through oversubscription. As discussed in Subsection Preemption of Chapter 7, this does not incur context-switching overhead, enabling efficient execution across the SMs.

```
__global__ void scale_y(Yx[M], beta):
    // Thread id
    tid = blockIdx.x * blockDim.x + threadIdx.x
    // Number of threads
    P   = gridDim.x * blockDim.x

    for i = tid to M-1 step P:
        Yx[i] *= beta

// Wrapper exposed to Python
extern "C" void bsr_matvec(/* full argument list omitted for brevity */):
    // Scale y by beta
    scale_y<<<blocksPerGrid, threadsPerBlock>>>(Yx, beta)
    // Perform accumulative BSR SpMV
    bsr_matvec_kernel<<<blocksPerGrid, threadsPerBlock, R * sizeof(T)>>>(/*
    ↪  full argument list omitted for brevity */)
```

Listing 13: Scaling kernel and wrapper used to call both CUDA kernels in all custom BLAS-style GPU implementations in this work

The separate CUDA kernel and wrapper from Listing 13 are reused across all CUDA implementations of BSR SpMV. Only the accumulative kernel $y \leftarrow y + \alpha Ax$ varies between implementations, since the scaling step involves only $O(M)$ operations compared to the $O(MN)$ complexity of the BLAS-style SpMV and requires no further optimization beyond coalesced access through the strided loop.

8.3.2 Access Pattern and Reduction Strategy

The primary differences between the custom BSR SpMV implementations on GPU manifest in the computational kernel and revolve around two critical aspects:

- **Access pattern**, which determines how threads interact with global memory and whether memory transactions can be coalesced or benefit from caching.
- **Reduction strategy**, which governs how partial results computed by individual threads are aggregated, leveraging GPU-specific synchronization, shared memory, and warp-level primitives.

These design dimensions are explored incrementally in the following subsections, with successive kernel variants introduced to demonstrate improvements in memory access efficiency and reduction mechanisms.

Row-Major Access Order

This kernel is a direct GPU port of the adapted computational kernel (Listing 11) using a row-major traversal of nonzeros and a strided loop so consecutive threads access consecutive elements for coalescing. Partial sums are accumulated into a per-block shared memory vector `sum_private` and then reduced into y. The code is shown in Listing 14.

```
__device__ void one_row_of_blocks(
/* full argument list omitted for brevity */) {
    // Ensure syncrhonization
    __syncthreads()
    // Initialize `sum_private`
    for bi = threadIdx.x to R-1 step blockDim.x
        sum_private[bi] = 0
    __syncthreads()

    for index = start_index + threadIdx.x to stop_index-1 step blockDim.x
        // The block index
        jj = index / RC
        // The intra-block index
        index_b = index % RC
        // The block column
        j = Aj[jj]
        // The intra-block row and column indices
        i_b = index_b / C
        j_b = index_b % C
        // The column coordinate within the matrix
        j_final = C * j + j_b
        // Atomicity required
        atomicAdd(&sum_private[i_b], Ax[index] * x[j_final])
    }
    // Ensure all partial results have been accumulated
    __syncthreads()
    // Accumulates the partial results into the `y` vector
    for bi = threadIdx.x to R step blockDim.x:
        atomicAdd(&y[R * i + bi], sum_private[bi] * alpha)
```

Listing 14: Row-major access order GPU kernel

Block Size	Max Blocks per SM	Blocks per SM	Oversubscription	Grid size	Time (s)
1024	2	32	16	1792	10.053
512	4	64	16	3584	4.368
256	8	64	8	3584	1.919
128	16	256	16	14336	0.809
64	32	32	1	1792	0.608
32	32	32	1	1792	0.558

Table 7: Row-Major Access Order best performance and launch configuration for each thread block granularity

The kernel uses only 32 registers per thread, which allows each SM to reach the theoretical maximum of 64 resident warps. This ensures that the configuration is not constrained by register pressure. While memory accesses to global memory are well coalesced and the workload is evenly distributed across threads, the row-major order causes many threads to update the same shared-memory locations, creating heavy atomic contention even between threads from the same warp, serialization being a guaranteed consequence.

Table 7 reports the runtimes for different block sizes. Performance improves monotonically as block size decreases: the worst configuration (1024 threads) runs in 10.053 s, while the best (32 threads) runs in 0.558 s. The strong benefit of small blocks indicates that atomic serialization in shared memory is the dominant bottleneck. The fact that the fastest configuration sustains only 32 active warps per SM - half the maximum possible - further confirms that reducing contention outweighs the benefits of higher warp occupancy.

Row-Major with Thread-Private Accumulation

This variant aims to mitigate the atomic contention observed in the Row-Major Access Order kernel. Each thread maintains its own private accumulation vector `acc`, as shown in Listing 15. By isolating partial sums at the thread level, contention on shared memory is fully prevented.

```
__device__ __forceinline__ void one_row_of_blocks(
        /* full argument list omitted for brevity */):
    T acc[5] = {0}
    for index = start_index + threadIdx.x to stop_index-1 step blockDim.x
        const I jj = index / RC
        const I index_b = index % RC
        const I j = Aj[jj]
        const I i_b = index_b / C
        const I j_b = index_b % C
        const I j_final = C * j + j_b
        // Atomicity no longer required
        acc[i_b] += Ax[index] * x[j_final]

    // Add the accumulated sum to the final vector with atomic operations
    for (I bi = 0 bi < R ++bi)
        const T val = block_reduce(acc[bi])
        // Only thread with id 0 pushes the value
        if (threadIdx.x == 0)
            atomicAdd(&y[R*i+bi], val * alpha)
```

Listing 15: Row-Major with thread-private accumulation GPU kernel

The `block_reduce` routine (Listing 16) aggregates these thread-private results efficiently and without atomic operations. It performs a two-level reduction: first within each warp using the `__shfl_down_sync` intrinsic, and then across warps using a fixed-size shared memory array with `__syncthreads` for synchronization. The final reduced value is added to the global y vector.

Block Size	Max Blocks per SM	Blocks per SM	Oversubscription	Grid size	Time (s)
1024	2	32	16	1792	1.295
512	4	64	16	3584	1.399
256	8	128	16	7168	1.417
128	16	256	16	14336	1.413
64	32	128	4	7168	1.511
32	32	512	16	28672	0.813

Table 8: Row-Major with Thread-Private Accumulation best performance and launch configuration for each thread block granularity

```
#define LANE_ID (threadIdx.x % 32)
#define WARP_MASTER (LANE_ID == 0)
#define WARP_ID (threadIdx.x / 32)

__device__ double block_reduce(val):
    __shared__ aux[32]
    for i = threadIdx.x to 32 step blockDim.x:
        aux[i] = 0
    __syncthreads()
    for offset = 16 to 0 step offset / 1
        val += __shfl_down_sync(0xffffffff, val, offset)
    if WARP_MASTER:
        aux[WARP_ID] = val
    __syncthreads()
    if WARP_ID == 0:
        val = aux[LANE_ID]
        #pragma unroll
        for offset = 16 to 0 step offset / 1:
            val += __shfl_down_sync(0xffffffff, val, offset)
    return val
```

Listing 16: Block-wide reduction algorithm for GPU

From a theoretical perspective, the main limitation of this approach arises from how NVCC [15] handles the private accumulator. Because its elements are accessed with indices only known at runtime, the compiler allocates `acc` in local memory rather than in registers, resulting in high-latency accesses. At full occupancy (2048 threads per SM), these accumulators require 80 KiB per SM, far exceeding the 24 KiB of L1 cache available; even at half occupancy, 40 KiB is required. Spills to the slower L2 cache are inevitable, negating the benefits of privatization: although atomic contention is eliminated, memory latency dominates the kernel performance.

Block Size	Max Blocks per SM	Blocks per SM	Occupancy	Grid size	Time (s)
1024	2	1	50%	56	0.874
512	4	2	50%	112	0.790
256	8	2	25%	112	0.767
128	16	4	25%	224	0.733
64	32	8	25%	448	0.718
32	32	16	50%	896	0.707

Table 9: Row-Major with Thread-Private Accumulation best performance and launch configuration for each thread block granularity under low occupancy

Table 8 presents the measured runtimes. As expected, there is no longer a clear difference between block granularities except for the block size of 32 threads, which produces the best result of 0.813 s. Despite removing atomic contention, this performance is still worse than the best result of 0.558 s achieved by the Row-Major Access Order kernel, confirming that memory-latency penalties outweigh the benefits of privatization in this case.

As a supplementary experiment, launch configurations with reduced occupancy were also tested. The results in Table 9 show that lowering occupancy improves performance across all thread block sizes, reaching 0.707 s. This demonstrates that even with the latency-hiding mechanisms of Pascal [12], the limited L1 cache capacity remains a bottleneck that cannot be overcome with this approach.

Row-Major with Team-Based Accumulation

This variant addresses the latency bottleneck of the Row-Major with Thread-Private Accumulation kernel by replacing the thread-private vector accumulator with a single scalar accumulator. This ensures that registers, rather than local memory mapped to VRAM, are used for accumulation, thereby eliminating the frequent cache spills. The algorithm is shown in Listing 17. Like the previous kernels, it uses 32 registers per thread and thus supports the maximum occupancy of 64 warps per SM.

```
// Handles one row of blocks starting from the block index `start` and going untill `stop`
__device__ __forceinline__ void one_row_of_blocks(
        /* full argument list omitted for brevity */):
    __syncthreads()
    // Initialize `sum_private`
    for bi = threadIdx.x to R step blockDim.x
        sum_private[bi] = 0
    __syncthreads()

    T acc = 0
    workers = blockDim.x - blockDim.x % RC
    teams = blockDim.x / RC
    team_id = threadIdx.x / RC
    start = start_index / RC
    stop = stop_index / RC
    worker_id = threadIdx.x % RC

    i_b = worker_id / C
    j_b = worker_id % C
    index_b = worker_id

    // Only threads that are part of full teams do work
    if threadIdx.x < workers
        for jj = start+team_id to stop step teams
            // The block column
            I j = Aj[jj]
            // The column coordinate within the matrix
            j_final = C * j + j_b
            // The index of the nonzero
            index = jj * RC + index_b
            // Atomicity required
            acc += Ax[index] * x[j_final]
        atomicAdd(&sum_private[i_b], acc)
    __syncthreads()

    // Add the accumulated sum to the final vector with atomic operations
    T *const y = sum + (npy_intp)R * i
    for bi = threadIdx.x to R step blockDim.x
        atomicAdd(&y[bi], sum_private[bi] * alpha)
```

Listing 17: Row-Major with Team-Based Accumulation GPU kernel

Block Size	Max Blocks per SM	Blocks per SM	Oversubscription	Grid size	Time (s)
1024	2	16	8	896	1.514
512	4	4	1	224	0.424
256	8	8	1	448	0.193
128	16	16	1	896	0.178
64	32	32	1	1792	0.196
32	32	32	1	1792	0.260

Table 10: Best runtimes of the scalar accumulator kernel across different block sizes

The scalar accumulator introduces new constraints. Each thread can now only accumulate results for a single coordinate within a block of nonzeros. To avoid skewing the loop, only full "teams" of threads can participate in the work, meaning a subset of the threads in each thread block remain idle. As a result, the kernel would not perform useful computation on matrices with unusually large blocks of nonzeros - a pathological case unlikely to occur in realistic datasets. On the other hand, the reduced indexing logic simplifies computation: instead of inferring both block indices and intra-block coordinates, only the nonzero index is required. At the end of each block's work, the partial results are combined into the shared memory vector `sum_private`, with atomics used only once per thread.

The performance data in Table 10 confirm the effectiveness of this design. The best result of 0.178 s is achieved with 128 threads per block - an improvement of more than $3\times$ over the previous best (0.558 s) from the Row-Major Access Order kernel. However, an unexpected phenomenon occurs for the configurations with 1024 and 512 threads per block: despite atomics being invoked only after the bulk of the work, contention still slows them down significantly (1.514 s and 0.424 s, respectively).

This effect stems from how atomics on 64-bit data types (e.g., doubles) are handled in shared memory on Pascal GPUs [12]. Since native 64-bit shared memory atomics are not supported, these operations are emulated through CAS (compare-and-swap) loops. Each atomic operation therefore involves repeated attempts until success, and under contention, this retry mechanism exacerbates delays. The severity of the slowdown is proportional to the number of threads contending on the same shared memory location, as made explicit in Table 11.

Table 11 provides further insight by reporting the number of worker threads, their proportion relative to block size, and the degree of atomic contention per shared element. Launches with 1024, 512, and 256 threads per block suffer the most from CAS-loop contention, whereas the 64-thread configuration suffers from reduced participation (78% of threads active). For 32 threads per block, this effect is further magnified: only half of the SM's maximum warps are active, halving the number of worker threads despite the same participation percentage. The configuration with 128 threads per block strikes the best balance, minimizing contention while still keeping almost all threads productive, which explains its superior performance.

Block Size	Workers	Workers Proportion	Workers Per SM	Grade of Contention	Time (s)
1024	1000	98%	2000	200	1.514
512	500	98%	2000	100	0.424
256	250	98%	2000	50	0.193
128	125	98%	2000	25	0.178
64	50	78%	1600	10	0.196
32	25	78%	800	5	0.260

Table 11: Worker participation and contention analysis for the scalar accumulator kernel

Vector Accumulation with Unrolled Loops

This final approach integrates all the optimizations introduced in the previous GPU kernels while also generalizing to all block shapes without exceptions. Its key feature is the use of unrolled loops to guarantee register allocation of the accumulator vector `acc`. The kernel unrolls the outer loop to process multiple scalar rows at once and a second unrolled loop to handle entire column segments, with bounds fully known at compile time. As a result, the accumulator vector is kept entirely in registers rather than spilling into local memory. At the end of each segment, results are aggregated at warp level using shuffle intrinsics, with only one thread per warp updating the shared memory vector `sum_private`. This reduces atomic contention from per-thread updates to a single update per warp.

A mild drawback of this design is that it no longer follows a strict row-major access order within blocks of nonzeros. Instead, threads process rows with the same index across neighboring blocks, which slightly reduces coalescing efficiency. In practice, however, the GPU's cache hierarchy compensates for this irregularity, and the penalty is minimal compared to the benefits of using efficient accumulation into registers and reducing atomic contention.

```
#define ROWS_AT_ONCE 8
__device__ __forceinline__ void one_row_of_blocks(
        /* full argument list omitted for brevity */):
    __syncthreads()
    for bi = threadIdx.x to R-1 step blockDim.x: sum_private[bi] = 0
    __syncthreads()
    (tiled_start, tiled_stop) = ((start_index, stop_index) / RC) * C
    offset_R = 0; pre_R = 0
        #pragma unroll
        for rows_at_once = ROWS_AT_ONCE to 1 step -1:
            // How many rows have been processed
            offset_R = max(offset_R, pre_R)
            // Untill which row to process
            pre_R = R - R % rows_at_once
            // Process `rows_at_once` rows at once
            for i_b = offset_R to pre_R-1 step rows_at_once:
            // The accumulator vector
                acc[ROWS_AT_ONCE] = 0
                row_start_offset = i_b * C
                for index = tiled_start+threadIdx.x to tiled_stop step
                ↪  blockDim.x:
                    // Block index
                    jj = index / C
                    // Column intra-block index
                    j_b = index % C
                    j_final = C * Aj[jj] + j_b
                    new_index = jj * RC + row_start_offset + j_b
                    #pragma unroll
                    for k = 0 to rows_at_once step 1:
                        acc[k] += Ax[new_index + C * k] * x[j_final]
                #pragma unroll
                for k = 0 to rows_at_once step 1:
                    #pragma unroll
                    for offset = 16 to 1 step offset/2:
                        acc[k] += __shfl_down_sync(0xffffffff, acc[k],
                        ↪  offset)
                    if (threadIdx.x % 32) == 0:
                        atomicAdd(&sum_private[i_b + k], acc[k])
    __syncthreads()
    for bi = threadIdx.x to R-1 step blockDim.x:
        atomicAdd(&y[R*i+bi], sum_private[bi] * alpha)
```

Listing 18: Vector Accumulation with Unrolled Loops GPU kernel

Block Size	Max Blocks per SM	Blocks per SM	Oversubscription	Grid size	Time (s)
1024	0	-	-	-	-
512	1	8	8	448	0.231
256	3	24	8	1344	0.161
128	6	48	8	2688	0.155
64	12	48	4	2688	0.155
32	25	200	8	11200	0.171

Table 12: Best runtimes of the Vector Accumulation with Unrolled Loops kernel across different block sizes

Parameter tuning revealed that performance is sensitive to the choice of the `ROWS_AT_ONCE` constant, which controls the size of the accumulator vector how many scalar rows can be processed in parallel. Values smaller than the block column size (e.g., 5×5 in our dataset) degrade performance, while setting it equal to the block size provides no advantage over slightly larger values. A value of 8 was chosen as a practical balance, ensuring that the kernel remains efficient across diverse block shapes, including both wide and tall blocks (e.g., 16×1), making the design agnostic to block shape in terms of performance.

This flexibility comes at the cost of register pressure. The nested unrolled loops result in heavy register usage - 77 registers per thread - compared to 32 in earlier kernels. As discussed in Subsection Rationale Behind Thread Block Counts, this leads to irregular, but still optimal, grid configurations. For example, block sizes of 1024 are infeasible because the high register footprint prevents sufficient threads from being scheduled per SM.

Table 12 reports the best runtimes for this kernel across block sizes. The launch with 512 threads per block shows reduced performance (0.231 s) because the SMs cannot schedule a new block until all active warps in the current block have finished execution, leaving resources underutilized. By contrast, block sizes of 128 and 64 achieve optimal runtimes of 0.155 s, further improving on the previous best of 0.178 s set by the Row-Major with Team-Based Accumulation kernel. The improvement comes from two factors: reduced atomic contention in shared memory (warp-level instead of thread-level updates) and lower indexing overhead enabled by loop unrolling. Even smaller blocks (32 threads) avoid register pressure issues but suffer from reduced parallelism, yielding slower performance (0.171 s).

8.4 Results

The final results of the best-performing configuration of each custom implementation are aggregated in Table 13 and illustrated in Figure 6. The cuSPARSE routine is included as the baseline for comparison.

Implementation	Time (s)	Speedup (over cuSPARSE)	Throughput (GB/s)	Efficiency on P100
cuSPARSE	0.337	1	248.267	33.9%
Row-Major Access Order	0.558	0.60	149.939	20.5%
Row-Major with Thread-Private Accumulation	0.707	0.48	118.339	16.2%
Row-Major with Team-Based Accumulation	0.178	1.89	470.033	64.2%
Vector Accumulation with Unrolled Loops	0.155	2.17	539.780	73.7%

Table 13: Custom GPU Implementations (best configuration) compared to cuSPARSE

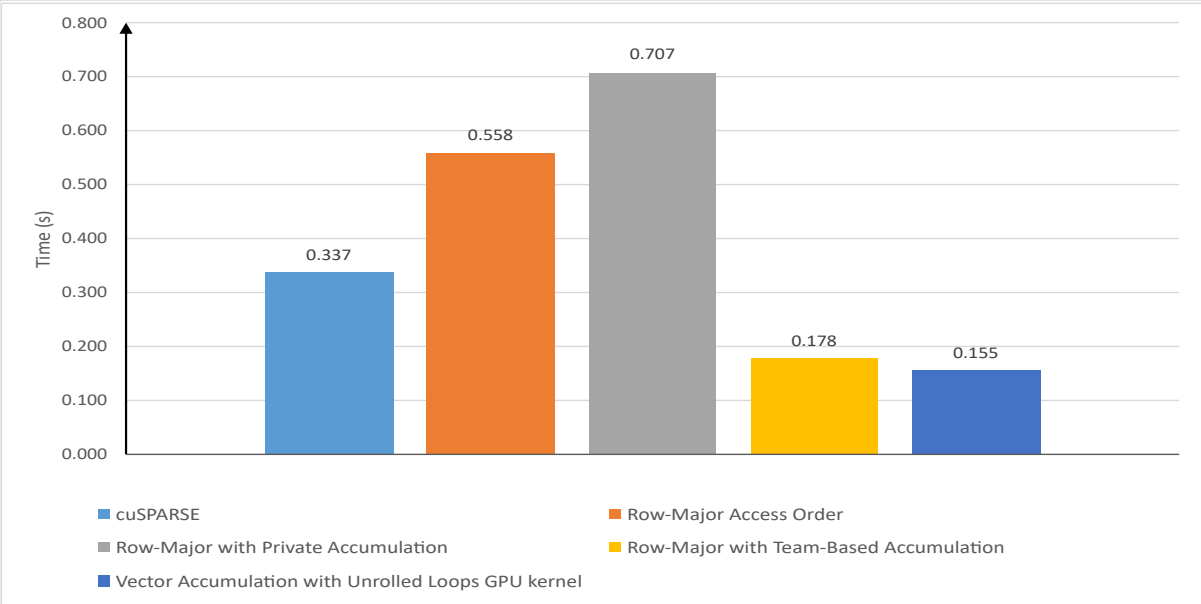

Figure 6: Custom GPU Implementations (best configuration) compared to cuSPARSE

The data confirm that the successive refinements introduced in our custom kernels substantially improved performance, with the Vector Accumulation with Unrolled Loops implementation surpassing the vendor-optimized cuSPARSE [14] routine. Its execution time of 0.155s corresponds to a $2.17\times$ speedup, a throughput of 539.78 GB/s, and an effective efficiency of 73.7% of the theoretical peak bandwidth of 732.2 GB/s of Tesla P100 PCIE 16 GB [12]. The

Row-Major with Team-Based Accumulation kernel also exceeded cuSPARSE with a speedup of 1.89×, reaching 64.2% efficiency.

These results highlight two key points. First, despite the highly tuned nature of cuSPARSE [14], carefully tailored strategies can surpass its performance on specific workloads such as BLAS-style BSR SpMV. Second, the gap between the best-performing kernel and the theoretical peak shows that the implementation is close to saturating the memory subsystem. In particular, the achieved 73.7% efficiency underlines that the kernel's performance is limited primarily by the hardware bandwidth ceiling rather than algorithmic inefficiencies.

9 CONCLUSION

This dissertation explored the optimization of sparse matrix-vector multiplication (SpMV) [22] on NVIDIA GPUs, focusing on the Block Sparse Row (BSR) format [18]. Sparse matrix operations are a cornerstone of high-performance computing, yet their irregular memory access patterns and low arithmetic intensity make them challenging to optimize on massively parallel architectures. The work began with CPU-based implementations inspired by SciPy's BSR SpMV kernel [29], investigating strategies to minimize synchronization overhead and optimize task granularity. These insights then guided the development of GPU implementations to maximize throughput and reduce performance bottlenecks.

The study followed a systematic, iterative approach. Initial CPU kernels included variants such as Granular Row, Naive Intra-Row Splits, Intra-Row Splits - Improved, Fixed Maximum Task Size, and Fixed-Size Tasks, culminating in Preprocessed Fixed-Size Tasks that efficiently handled low-granularity workloads. Building upon these foundations, naive GPU ports preserved the row-major access order of the nonzeros, and subsequent GPU kernels introduced architectural-aware optimizations. Key challenges such as atomic contention, register pressure, and cache limitations were analyzed in depth. Several kernel variants were developed: Row-Major Access Order, Row-Major with Thread-Private Accumulation, Row-Major with Team-Based Accumulation, and finally Vector Accumulation with Unrolled Loops. Each approach addressed specific bottlenecks while illustrating the trade-offs inherent to GPU optimization, such as balancing occupancy against memory contention and the use of registers versus local memory for accumulation.

Performance evaluations demonstrated that these architectural-aware optimizations can significantly outperform standard library implementations. The best custom kernel, employing vector accumulation with unrolled loops, achieved a runtime of 0.155 seconds on the Tesla P100 PCIe 16 GB GPU, corresponding to a speedup of 2.17$\times$ over cuSPARSE [14] and a sustained memory throughput of 539 GB/s - over 73% of the theoretical peak bandwidth. These results confirm that careful alignment of memory access patterns, workload distribution, and thread scheduling with hardware capabilities is critical to closing the gap between theoretical and practical GPU performance for sparse computations.

In conclusion, this work highlights the importance of iterative, architecture-aware kernel design for high-performance GPU computing. By combining detailed performance analysis with methodical experimentation, it provides insights into how the irregular structure of sparse matrices can be efficiently mapped onto modern GPU architectures. Furthermore, it emphasizes the value of leveraging CPU-side preprocessing to inform efficient GPU scheduling, bridging the gap between traditional CPU execution and modern GPU optimization strategies.

Ultimately, this dissertation illustrates a full journey from concept to high-performance implementation. Starting with simple CPU-based prototypes, progressing through naïve GPU ports, and culminating in fully optimized, architecture-aware kernels, it demonstrates how careful analysis, experimentation, and iterative refinement can transform a challenging, irregular computation into a highly efficient GPU routine. Beyond the specific optimizations explored, this work underscores a general principle: understanding the interplay between algorithmic structure and hardware characteristics is essential for extracting maximum performance from modern heterogeneous computing systems.

BIBLIOGRAPHY

[1] The c programming language. https://www.iso.org/standard/82075.html. Last accessed: 9 February 2025.

[2] Francisco Massa Adam Lerer James Bradbury Gregory Chanan Trevor Killeen Zeming Lin Natalia Gimelshein Luca Antiga Alban Desmaison Andreas Köpf Edward Yang Zach DeVito Martin Raison Alykhan Tejani Sasank Chilamkurthy Benoit Steiner Lu Fang Junjie Bai Soumith Chintala Adam Paszke, Sam Gross. Pytorch: An imperative style, high-performance deep learning library. https://arxiv.org/abs/1912.01703. Last accessed: 26 June 2023.

[3] AMD. Rdna architecture whitepaper. https://gpuopen.com/download/RDNA_Architecture_public.pdf, 2019. Accessed: 2025-09-09.

[4] AMD. Amd cdna architecture whitepaper. https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna-white-paper.pdf, 2020. Accessed: 2025-09-09.

[5] AMD. Amd chiplet ecosystem white paper. https://www.amd.com/content/dam/amd/en/documents/solutions/technologies/chiplet-architecture-white-paper.pdf, 2024. Accessed: 2025-09-09.

[6] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Hansol Suh, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. PETSc/TAO users manual. Technical Report ANL-21/39 - Revision 3.23, Argonne National Laboratory, 2025.

[7] OpenMP Architecture Review Board. Openmp application programming interface version 4.5. https://www.openmp.org/specifications/, 2015. Accessed: 2025-09-09.

[8] Stéfan J. van der Walt Ralf Gommers Pauli Virtanen David Cournapeau Eric Wieser Julian Taylor Sebastian Berg Nathaniel J. Smith Robert Kern Matti Picus Stephan Hoyer Marten H. van Kerkwijk Matthew Brett Allan Haldane Jaime Fernández del Río Mark Wiebe Pearu Peterson Pierre Gérard-Marchant Kevin Sheppard Tyler Reddy Warren Weckesser Hameer Abbasi Christoph Gohlke Travis E. Oliphant Charles R. Harris,

K. Jarrod Millman. Array programming with numpy. https://doi.org/10.1038/s41586-020-2649-2. Last accessed: 24 June 2023.

[9] Intel Corporation. Introduction to the xe-hpg architecture white paper. https://cdrdv2-public.intel.com/758302/introduction-to-the-xe-hpg-architecture-white-paper.pdf, 2022. Accessed: 2025-09-09.

[10] NVIDIA Corporation. Kepler gk110/210 architecture whitepaper. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf, 2012. Accessed: 2025-09-09.

[11] NVIDIA Corporation. Maxwell architecture whitepaper. https://developer.nvidia.com/maxwell-compute-architecture, 2014. Accessed: 2025-09-09.

[12] NVIDIA Corporation. Pascal architecture whitepaper. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016. Accessed: 2025-09-09.

[13] NVIDIA Corporation. Turing gpu architecture whitepaper. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018. Accessed: 2025-09-09.

[14] NVIDIA Corporation. cuSPARSE library. https://developer.nvidia.com/cusparse, 2025.

[15] NVIDIA Corporation. Nvidia cuda compiler (nvcc). https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html, 2025. Accessed: 2025-09-09, Version 11.4.

[16] Shitao Fan. Compressed sparse column format (coo). https://scipy-lectures.org/advanced/scipy_sparse/coo_matrix.html. Last accessed: 9 February 2025.

[17] Shitao Fan. Compressed sparse column format (csc). https://scipy-lectures.org/advanced/scipy_sparse/csc_matrix.html. Last accessed: 9 February 2025.

[18] Shitao Fan. Compressed sparse row format (bsr). https://scipy-lectures.org/advanced/scipy_sparse/bsr_matrix.html. Last accessed: 9 September 2025.

[19] Shitao Fan. Compressed sparse row format (csr). https://scipy-lectures.org/advanced/scipy_sparse/csr_matrix.html. Last accessed: 26 June 2023.

[20] Torbjörn Granlund and Peter L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 61–72. ACM, 1994.

[21] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012.

[22] Enver Kayaaslan Kadir Akbudak and Cevdet Aykanat. Sparse matrix-vector multiplication. https://epubs.siam.org/doi/abs/10.1137/100813956. Last accessed: 9 February 2025.

[23] Cornelius Lanczos Magnus Rudolph Hestenes, Eduard L. Stiefel. Conjugate gradient. https://nvlpubs.nist.gov/nistpubs/jres/049/6/V49.N06.A08.pdf. Last accessed: 23 June 2023.

[24] Intel's Developer Manual. Dense matrix-vector multiplication. https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-dpcpp/2023-0/gemv.html. Last accessed: 9 February 2025.

[25] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.

[26] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. Last accessed: 24 June 2023.

[27] Richard Stallman. Gcc online documentation. https://gcc.gnu.org/onlinedocs/. Last accessed: 9 February 2025.

[28] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[29] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, and et al. Scipy bsr sparse matrix-vector multiplication implementation. https://github.com/scipy/scipy/blob/v1.16.0/scipy/sparse/sparsetools/bsr.h#L774, 2025. Accessed: 2025-09-09; used as baseline for custom BSR SpMV implementations.

[30] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, Ílke Öztireli, Alex Kim, Stephan Berg, Nithin Bansal, Magnus Pärtel, Sergey V. D. V. Andrei, and contributors. Scipy 1.0: Fundamental algorithms for scientific computing in python. *Nature Methods*, 17:261–272, 2020.