Website Link: https://naturalcandy.github.io/cmu418final/

## Summary

Our final report presents GraphBLAS-GPU, a GPU-accelerated framework that converts a high-level sequence of graph operations, expressed through an intuitive C++ API, into a single, persistent CUDA kernel that is generated and JIT-compiled at run time. Users write their algorithms solely in terms of algebraic graph primitives (e.g., SpMV, element-wise vector updates, custom semirings, termination predicates), and the framework takes over. Our framework analyzes data-flow, allocates a contiguous device buffer, and emits specialized code for the requested sparse formats (CSR, ELL, or SELL-C). GraphBLAS-GPU removes per iteration kernel-launch overhead entirely and avoids PCIe traffic during the iterative phase by keeping all fronts, masks and scratch buffers in one persistent device allocation. Additionally we show that planning format-aware kernels is an effective means of tackling the irregular memory access and load imbalance typical of real-world sparse graphs. The result is a proof-of-concept, GPU-native runtime that lets developers focus on applying simple graph logic rather than low-level CUDA details, yet still delivers good performance.

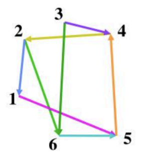
## Background

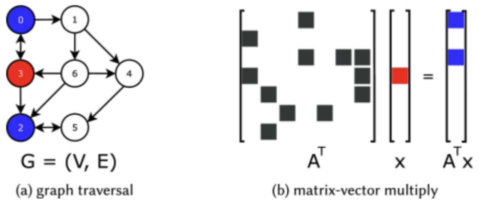
### An Algebraic View of Graph Algorithms

Traditional graph algorithms such as Breadth-First Search (BFS), Single-Source Shortest Path (SSSP), and PageRank are typically implemented using node and edge-centric approaches, making them intuitive but increasingly complex to parallelize as algorithms grow more complex.

The Graph-BLAS standard takes another approach and views a graph as a sparse matrix $A$. The entries represent edge weights and the edges represent data dependencies in matrix-vector multiplication. A frontier or label set is represented as a vector $v$. The heart of most parallel graph algorithms is just masked sparse matrix–vector multiply (SpMV) under a user-supplied semiring. With these tools, matrix-vector multiplication can broadly represent the operation for information propagation through a graph.

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & & & & 1 & \\ 2 & 1 & 1 & & & & 1 \\ 3 & & & 1 & 1 & & 1 \\ 4 & & 1 & & 1 & & \\ 5 & & & & 1 & 1 & \\ 6 & & & & & 1 & 1 \end{array}$$





sparse matrix represent https://dl.acm.org/doi/fullHtml/10.1145/3466795

,

A semiring re-defines the “add” ⊕ and “multiply” ⊗ used inside the GEMM/SpMV kernel. A semiring consists of:

1. A set of values $S$
2. An addition operator (⊕)
3. A multiplication operator (⊗)

Note that a semiring must adhere to the following rules:

$$\text{Closure: } (a \oplus b \in S) \text{ and } (a \otimes b \in S) \text{ for all } a, b \in S$$

$$\text{Associativity: } a \oplus (b \oplus c) = (a \oplus b) \oplus c \text{ and } a \otimes (b \otimes c) = (a \otimes b) \otimes c$$

$$\text{Distributivity: } a \otimes \left(b \oplus c\right) = \left(a \otimes b\right) \oplus \left(a \otimes c\right)$$

$$\text{Identity elements: } \exists \text{ elements } 0 \text{ and } 1 \text{ s.t: } a \oplus 0 = a \text{ and } a \otimes 1 = 1 \text{ for all } a \in S$$

$$\text{Annihilation: } a \otimes 0 = 0 \text{ for all } a \in S$$

By redefining these operators, we can tailor matrix operations for specific graph problems:

- Arithmetic Semiring (standard addition and multiplication): Used for weighted traversals
- Logical OR-AND Semiring: Used for BFS and connectivity analysis
- MIN-PLUS Semiring: Used for shortest path calculations

For example, in BFS traversal using the OR-AND semiring, multiplication becomes logical AND, and addition becomes logical OR. This allows us to propagate "visited" status throughout the graph efficiently.

To illustrate how this works in practice, consider a graph algorithm expressed using the OR-AND semiring. Given an adjacency matrix M and a frontier vector v, the next frontier can be computed as:

$$\texttt{frontier}_{\text{next}} = M^T \odot \texttt{frontier}$$

where,

$$(\texttt{frontier}_{\text{next}})_i = \bigvee_j (M_{ji} \wedge \texttt{frontier}_j)$$

Where ⊙ represents matrix-vector multiplication using the OR-AND semiring. This operation computes, for each vertex, whether it has any neighbor in the current frontier.

## Key Data Structures

GraphBLAS-GPU provides two primitive data structures to the user:

1. **Graph/Matrix**: Represents the graph topology as a sparse matrix abstraction. Users interact with this as a unified graph primitive without needing to understand the underlying storage details. Internally, the framework implements multiple formats selected automatically based on the input graph characteristics:
    - **CSR** (Compressed Sparse Row): Stores row offsets, column indices, and values.
    - **ELL** (ELLPACK): Stores fixed-length rows with padding.
    - **SELL-C** (Sliced ELLPACK): A tiled ELL approach that groups similar-density rows into slices.
2. **Vector**: Represent vertex properties or algorithm state (distance values, visited flags, etc.)

This abstraction allows users to focus on algorithm expression rather than format-specific optimizations. The framework automatically analyzes graph properties and selects the optimal storage format, while providing unified kernels that work identically regardless of the underlying representation.

As we will see in the upcoming Results section, the selection of the underlying format plays a significant role in optimizing our core SpMV kernels.

## Key Operations

The framework provides these core operations:

1. **SpMV (Sparse Matrix-Vector Multiplication)**: The fundamental operation for information propagation through the graph, adaptable via different semirings.
2. **Vector Operations**: Element-wise addition, subtraction, multiplication, division, logical operations, and data movement.
3. **Masking**: Selective updates to vectors based on condition vectors.
4. **Termination Detection**: Dynamic convergence or target condition detection within the GPU loop kernel.

## Computational Bottlenecks

The computationally expensive aspects that benefit from parallelization include:

1. **SpMV Operations**: The core matrix-vector multiplication with irregular memory access patterns.
2. **Vector Operations**: Element-wise operations across potentially large vectors

3. **Convergence Detection**: Determining when algorithms have reached their termination condition

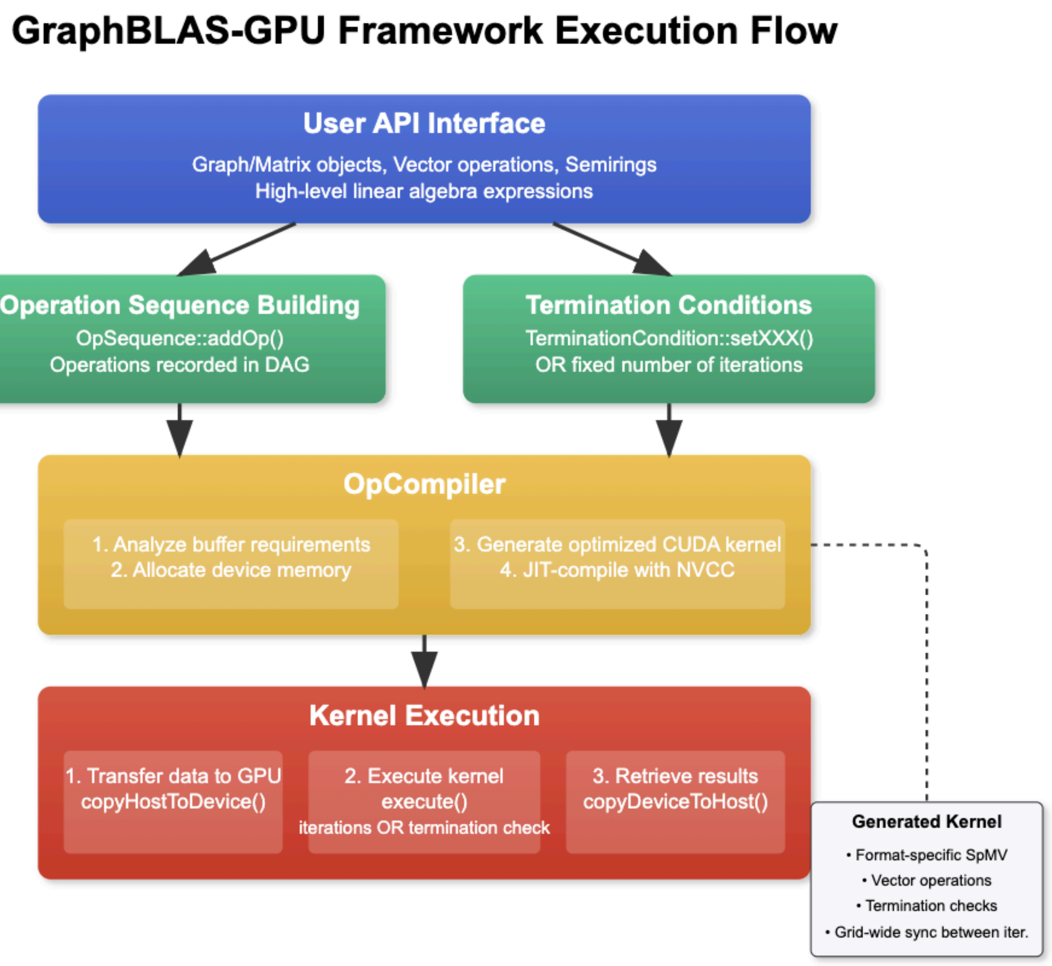
## Parallelism Analysis

- **Data Parallelism**: Our framework exploits massive data parallelism where each thread processes different matrix rows or vector elements independently
- **Dependencies**: Row-level independence in SpMV operations, with partial results combined using the semiring's addition operator
- **Locality Challenges**: Irregular memory access patterns when accessing vector elements through column indices
- **SIMD Suitability**: Partially amenable to SIMD execution, with some divergence in SpMV operations due to varying row lengths
- **Workload size (nnz)** determines arithmetic intensity.
- **Row-length variance $\sigma^2$** drives branch divergence: high variance hurts warp-level kernels.
- **Intra-row independence** → SIMD & warp reductions.
- **Inter-iteration dependency** appears only in algorithms that revisit frontier vectors; we therefore fuse the *iteration loop* into the kernel and expose **grid-wide cooperative termination**.

# Approach

## Framework Overview and Execution Model

GraphBLAS-GPU offers a high-level, C++-based API that allows users to express graph algorithms in terms of sparse linear algebra operations while abstracting away the GPU implementation details. The framework uses a deferred execution model where operations are staged into an operation sequence that is later compiled into an optimized CUDA kernel.


GraphBLAS-GPU Framework Execution Flow
User API Interface
Graph/Matrix objects, Vector operations, Semirings
High-level linear algebra expressions
Operation Sequence Building
OpSequence::addOp()
Operations recorded in DAG
Termination Conditions
TerminationCondition::setXXX()
OR fixed number of iterations
OpCompiler
1. Analyze buffer requirements
2. Allocate device memory
3. Generate optimized CUDA kernel
4. JIT-compile with NVCC
Kernel Execution
1. Transfer data to GPU
copyHostToDevice()
2. Execute kernel
execute()
iterations OR termination check
3. Retrieve results
copyDeviceToHost()
Generated Kernel
• Format-specific SpMV
• Vector operations
• Termination checks
• Grid-wide sync between iter.

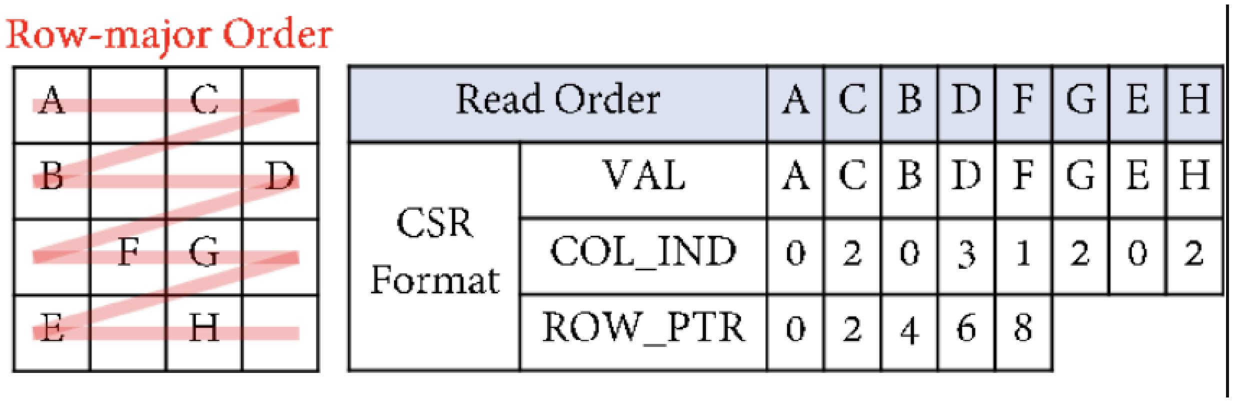
## Deferred Execution Pipeline

Our deferred execution approach consists of the following stages:

1. **Operation Definition**: Users define computations using high-level primitives
2. **Matrix Classification:** We run a classifier to determine the optimal sparse matrix format to run our kernel with.
3. **DAG Construction**: Operations are recorded in a directed acyclic graph (DAG)
4. **Memory Analysis**: Buffer requirements are calculated for all operations
5. **Code Generation**: CUDA kernel code is generated at runtime
6. **Memory Allocation**: GPU memory is allocated and input data is transferred
7. **Kernel Execution**: The generated kernel is executed with appropriate termination conditions
8. **Result Retrieval**: Output data is transferred back to the host

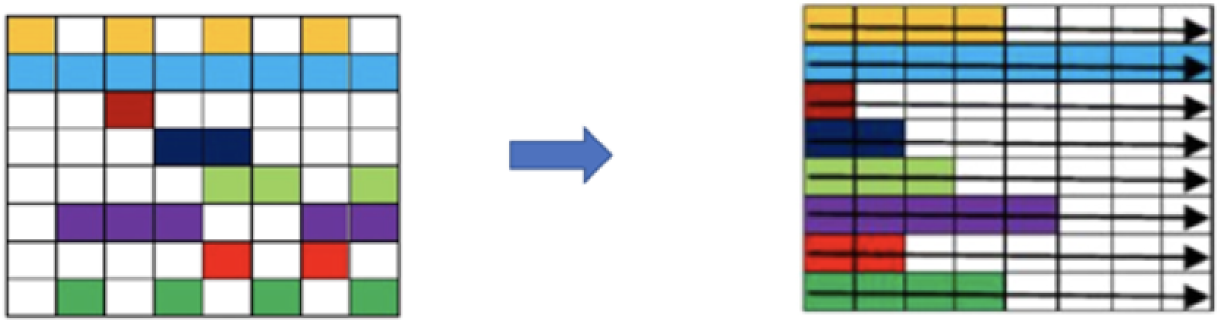
## Chosen Sparse Matrix Representations

**Compressed Sparse Row (CSR)** - a widely used format where non-zero elements are stored row by row in a contiguous array, along with two additional arrays that record the column indices of each non-zero and the starting point of each row. This format is flexible, efficient for matrices with highly irregular sparsity patterns, and particularly good for sparse matrix-vector multiplication (SpMV) where row-wise access is important.

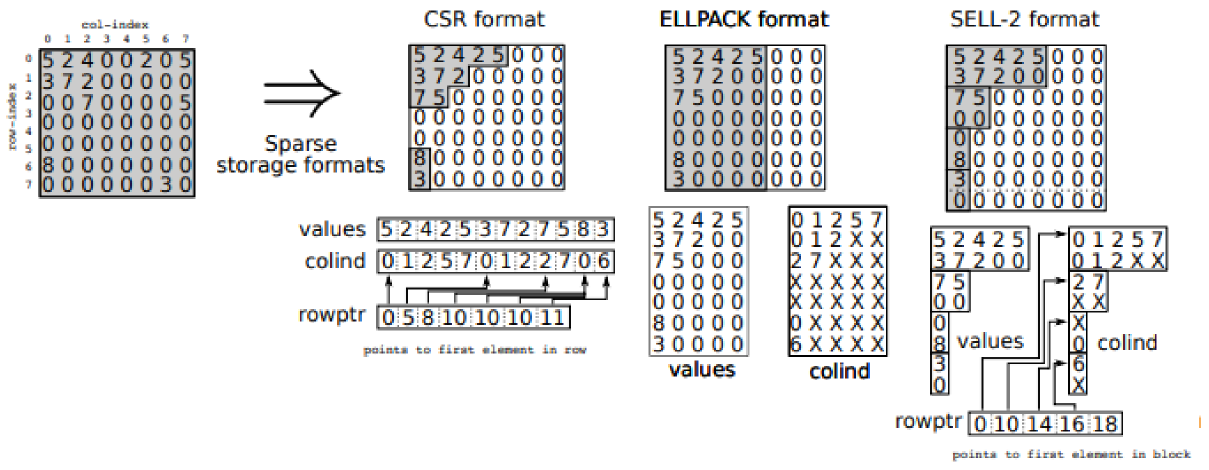


| Read Order | | A | C | B | D | F | G | E | H |
|---|---|---|---|---|---|---|---|---|---|
| CSR Format | VAL | A | C | B | D | F | G | E | H |
| | COL_IND | 0 | 2 | 0 | 3 | 1 | 2 | 0 | 2 |
| | ROW_PTR | 0 | 2 | 4 | 6 | 8 | | | |

**ELL-Pack (ELL)** - stores all matrix rows with the same number of non-zero entries by padding shorter rows with zeros to match the length of the longest row. The non-zero values and their corresponding column indices are organized into dense two-dimensional arrays, enabling efficient memory access on architectures like GPUs that benefit from regularity. However, this format may incur memory overhead if the matrix exhibits significant row-wise imbalance in the number of non-zeros, as all rows are padded to the maximum length.



**Sliced ELL-Pack With Chunk Size C (SELL-C)** is an extension of the ELL-Pack format that tiles the matrix into smaller row blocks (or "slices"), where each slice groups rows with similar numbers of non-zero entries. By localizing padding to within slices, SELL-C reduces memory overhead compared to standard ELL-Pack, which pads all rows globally to the maximum length. This format retains the memory access efficiency of ELL-Pack, beneficial for GPU architectures, while improving adaptability to matrices with moderate sparsity pattern variations.



https://library.eecs.utk.edu/files/ut-eecs-14-727.pdf

## Types of Sparse Matrices

**Random** – Randomly sparse matrices have no clear pattern to them in terms of where non-zero elements appear. The distribution of non-zeros can vary widely between rows, resulting in unpredictable row densities. This often leads to moderate to high variance in the number of non-zero elements per row, depending on how the sparsity was generated.

**Uniform** – In uniform sparse matrices, the number of non-zero elements is about the same across all rows. This even distribution leads to low variance in non-zero counts and is typical in structured numerical problems. The low variance makes them ideal for storage formats that assume consistent row density.

**Diagonal** – Diagonal sparse matrices have non-zero entries concentrated along one or more diagonals, often close to the main diagonal. These structures naturally arise in problems like solving differential equations and are extremely regular. Due to the strict placement of non-zeros, diagonal matrices exhibit very low variance in the number of non-zeros per row.

**Power Law** – Power law sparse matrices are highly irregular: a small number of rows or columns have many non-zeros, while most have very few. This pattern is characteristic of real-world graphs such as social networks and the web, where connectivity follows a heavy-tailed distribution. As a result, power law matrices have very high variance in non-zero counts across rows.

**Block** - A sparse matrix in which non-zero elements are grouped into dense submatrices, or blocks, rather than being scattered individually. Each block typically represents a small dense matrix, and most of the matrix is still composed of zero-valued blocks.

**Arrow Head** – Arrow head matrices are shaped like an arrow: a block of non-zeros along the main diagonal and a few dense rows or columns connecting across the matrix. This results in mostly low variance in the number of non-zeros per row, except for a few rows with significantly higher non-zero counts. Consequently, the overall variance is moderate, depending on how prominent the dense rows are.

**Triangular** – Triangular matrices have non-zero entries only above (upper triangular) or below (lower triangular) the main diagonal. Triangular sparsity leads to a gradually increasing or decreasing number of non-zeros per row, depending on the orientation, and can affect storage and computational efficiency differently compared to uniform sparsity. These matrices commonly appear when solving systems of linear equations through direct methods like LU decomposition. The variance in non-zeros per row is moderate, reflecting the structured but non-uniform distribution.

## How We Determine Optimal Format

Our framework can determine which format is most optimal for processing the provided matric. We determine the optimal sparse matrix format by analyzing two properties:

- **Mean** number of non-zeros per row: shows the average row density.
- **Variance** of non-zeros per row (normalized by the mean): measures how much the number of non-zeros fluctuates between rows.

**Low Variance**:

- The matrix is highly uniform, meaning most rows have a similar number of non-zeros.
- If there are rows that are either significantly denser than average or completely empty, it indicates minor irregularities - SELL-C is most optimal to minimize padding due to the irregularities.
- If the matrix is mostly uniform, a strictly uniform format like ELL is the most efficient.

**Moderate Variance**:

- The matrix has noticeable but not extreme differences in row densities.
- SELL-C that can accommodate moderate irregularities without much overhead.

**High Variance**:

- The matrix has large differences between row densities, including possibly some very dense rows and many sparse rows.
- CSR is chosen to efficiently handle the irregular structure due to its flexibility and lack of memory overhead.

By evaluating the matrix according to these criteria, we systematically choose the format that balances memory efficiency and computational performance.

## Mapping to GPU Architecture

Our mapping strategy focuses on maximizing parallelism while handling the irregular memory access patterns inherent in sparse graph operations:

For our CSR-aimed SpMV operations, we implemented multiple parallelization strategies. Our naive approach had each thread processing one row (vertex) of the matrix:

```
C/C++
// row based csr-spmv
row = blockIdx.x * blockDim.x + threadIdx.x
if row < num_rows:
    sum = 0
    for nonzero in matrix_row[row]:
        col = column_index[nonzero]
        sum += value[nonzero] * vector[col]
    output[row] = sum
```

For high-degree vertices, multiple threads can actually cooperate on a single row, with warp-level reduction allowing us to combine partial results:

```
C/C++
//warp based csr-spmv

// Each warp processes one row
row = blockIdx.x * blockDim.y + threadIdx.y  // Global row index
if row >= num_rows: return                   // Boundary check

start = row_offsets[row]                     // CSR row pointers
end = row_offsets[row + 1]

sum = 0                                      // Initialize partial sum
lane = threadIdx.x                           // Lane ID within warp (0..31)

for i = start + lane; i < end; i += THREADS_PER_ROW:
    sum += values[i] * vector[column_indices[i]]

// Warp-level reduce
#pragma unroll
for offset = THREADS_PER_ROW / 2; offset > 0; offset /= 2:
    sum += __shfl_down_sync(0xFFFFFFFF, sum, offset)

// Single-threaded output with optional masking
if lane == 0:
    output[row] = mask_enabled ? sum * (mask[row] != 0) : sum
```

## Memory Access Optimization:

To mitigate the impact of irregular memory access patterns:

1. **Memory coalescing**: ELL and SELL-C formats reorganize data to improve coalesced memory access
2. **Format selection**: Different formats are chosen based on input graph characteristics

## Grid and Block Configuration

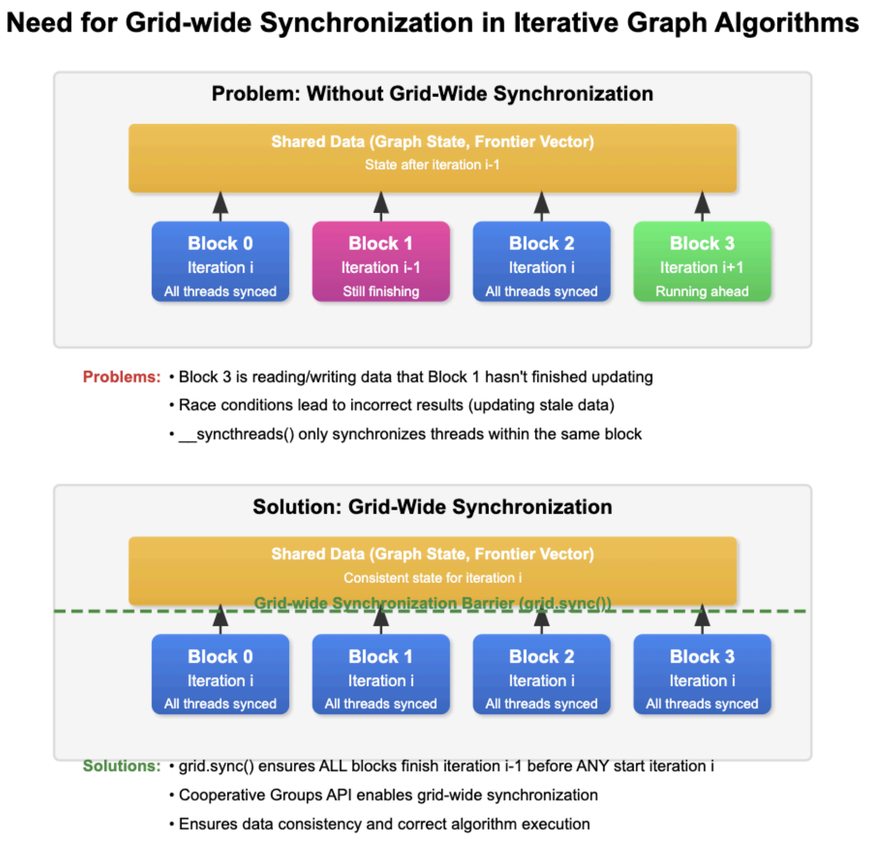
We use dynamic grid/block sizing based on:

- Problem size (vertices and edges)
- Target GPU architecture (SMs and resources). In our case we worked with a NVIDIA RTX 2080B (7.5, 46 SMs, 8 GB)
- Selected sparse matrix format
- Generated Operational Graph

A typical configuration uses 256 threads per block with grid size set to maximize occupancy (cuda Occupancy API), while ensuring sufficient work per thread.

## Dynamic Termination Detection

Our persistent kernel design enforces iteration-level synchronization across all threads. Unlike host-launched kernels, which rely on implicit PCIe synchronization, we utilize both cooperative group barriers and our own grid syncs to ensure all blocks complete iteration $i$ before proceeding to $i + 1$. This prevents threads from operating on stale data (e.g., reading a frontier while another block is still updating it). While intra-block sync (__syncthreads()) suffices for independent work, iterative graph algorithms still require equal global progress across iterations to guarantee correctness.


Need for Grid-wide Synchronization in Iterative Graph Algorithms
Problem: Without Grid-Wide Synchronization
Shared Data (Graph State, Frontier Vector)
State after iteration i-1
Block 0
Iteration i
All threads synced
Block 1
Iteration i-1
Still finishing
Block 2
Iteration i
All threads synced
Block 3
Iteration i+1
Running ahead
Problems:
• Block 3 is reading/writing data that Block 1 hasn't finished updating
• Race conditions lead to incorrect results (updating stale data)
• __syncthreads() only synchronizes threads within the same block
Solution: Grid-Wide Synchronization
Shared Data (Graph State, Frontier Vector)
Consistent state for iteration i
Grid-wide Synchronization Barrier (grid.sync())
Block 0
Iteration i
All threads synced
Block 1
Iteration i
All threads synced
Block 2
Iteration i
All threads synced
Block 3
Iteration i
All threads synced
Solutions:
• grid.sync() ensures ALL blocks finish iteration i-1 before ANY start iteration i
• Cooperative Groups API enables grid-wide synchronization
• Ensures data consistency and correct algorithm execution

In the scenario where a user specifies a fixed number of iterations to run the loop kernel our logic is relatively straightforward. However, iterative graph algorithms often rely on the evaluation of termination conditions.

To provide a flexible framework for dynamic termination in iterative graph algorithms, our API offers a termination condition mechanism that enables algorithm-specific convergence detection without requiring kernel termination and relaunch.

We provide a TerminationCondition class that supports multiple termination strategies through specialized methods like `setNodeReached` (terminate when a target node is reached), `setFrontierUnchanged` (terminate when no new nodes are discovered), and `setBfsComplete` (a combination of both conditions).

Rather than hard-coding these conditions into the kernel, our framework generates custom CUDA code at runtime based on the user's termination specification. For example, a BFS might terminate when the target node is found OR when the frontier stops changing (indicating no path exists), while PageRank would typically terminate when the residual error falls below a threshold and the algorithm has performed a minimum number of iterations. The framework handles both OR-reduction patterns (where any thread can trigger termination) and AND-reduction patterns (where all threads must agree to terminate) through a voting protocol that works across thread blocks. Our approach enables us to allow for early termination for algorithms like BFS (potentially saving many iterations) while still supporting convergence-based termination for numerical algorithms like PageRank or SSSP. Ultimately our biggest focus is having it all within a unified API that abstracts away the complex grid synchronization details from the user.

## Optimization Journey

Our optimization journey involved addressing many challenges in implementing an efficient device-loop kernel paradigm. Initially, we had a lot of trouble implementing device-wide synchronization given that thread blocks execute asynchronously on the GPU. Eventually we came to realize that oversubscribing the GPU (scheduling more thread blocks than could fit on available SMs) would cause indefinite hangs during global synchronization barriers, as blocks waiting at a barrier might never be scheduled. We ended up prioritizing a straightforward approach of tuning our grid dimensions to ensure all thread blocks could execute simultaneously (versus partial barriers). Another significant evolution was in how we thought of our API design. Initially we thought about exposing the direct kernel implementations to users, believing to allow maximum flexibility. However, after trying to think about using our framework in the lens of an end-user, we realized this created a steep learning curve and tied users to implementation details. This was when we decided to switch over to a higher-level abstraction that separated algorithm description from implementation details. We noticed that this clear separation of concerns was not only helpful for a potential end-user, but also helped us better reason our kernel implementations, memory layouts, and synchronization mechanisms.

# Results

All our tests were run on the CMU GHC machines. We used the platform's RTX 2080 Nvidia GPUs. Our tests were run by generating a random matrix which was then

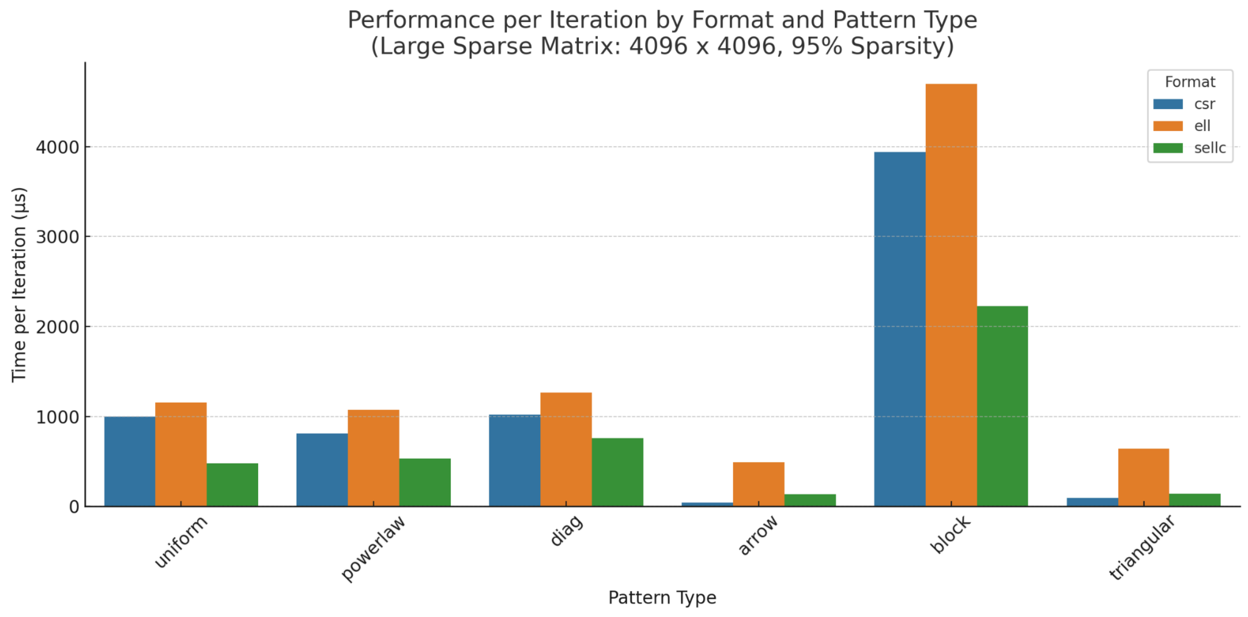
We ran tests that compared the performance of different sparse formats across the different sparsity types. We did this for both 256x256 matrices and 4096x4096 matrices at 95% sparsity. We wrote generators that would create synthetic matrices of the appropriate type. With each test iteration this matrix was converted to the appropriate format before being fed to the kernel. The kernels were launched with block size of 256 and a grid size that is determined at runtime of the framework, prior to the compilation and execution of the generated kernel. For all tests involving SELL-C, it was given a slice size of 32 (warp size).

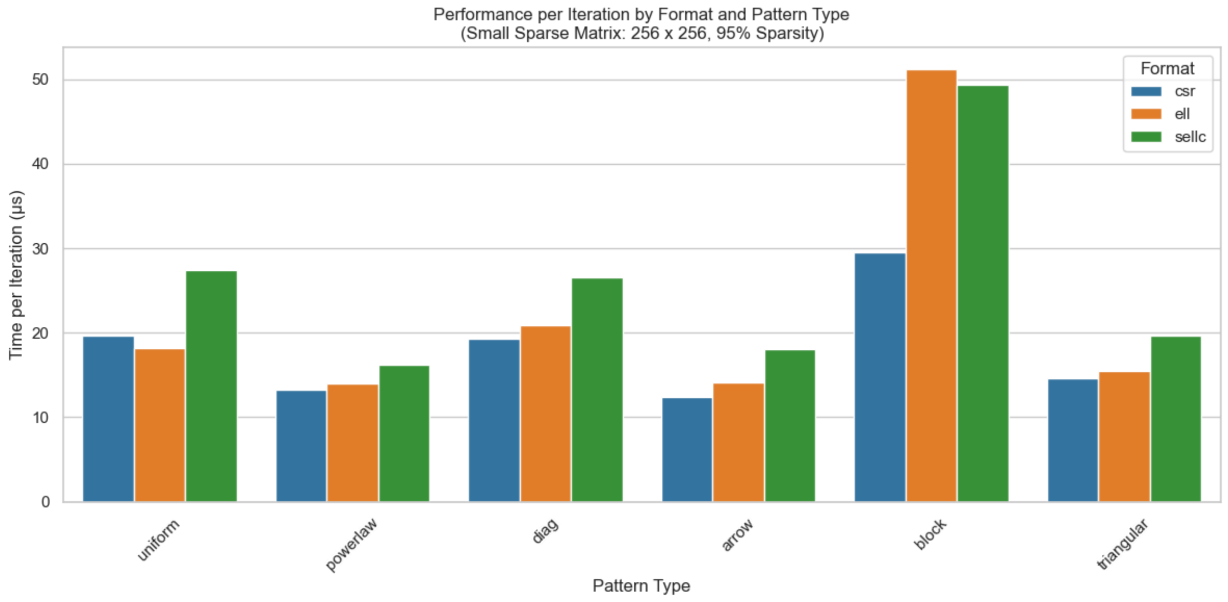
Based on the research conducted we expected the following performance behavior prior to running tests:

| Sparsity Type | Best Format | Worst Format |
|---|---|---|
| Uniform | SELL-C (ELL if highly uniform) | CSR |

| | | |
|---|---|---|
| Power-Law | SELL-C | ELL |
| Diagonal | SELL-C | CSR |
| Arrowhead | SELL-C | ELL |
| Block | SELL-C | ELL |
| Triangular | SELL-C | ELL |

It must be noted that the performance of SELL-C is highly dependent on the slice size (C value). Our expectations above assume that the slice size is optimized for that type of matrix. If not optimized, SELL-C can easily become the least performant format. SELL-C was assumed to be the most suitable format across all cases due to its adaptability to variations in the number of non-zero elements per row. This flexibility reduces padding, which in turn lowers the memory footprint and improves memory access patterns. On the other hand, ELL was expected to perform poorly in most scenarios, as its fixed row length, based solely on the densest row, introduces substantial padding. This padding increases memory overhead and disrupts efficient memory access, potentially leading to slower performance. The cases where ELL was anticipated to be the worst are cases with a large spread in density of the rows.

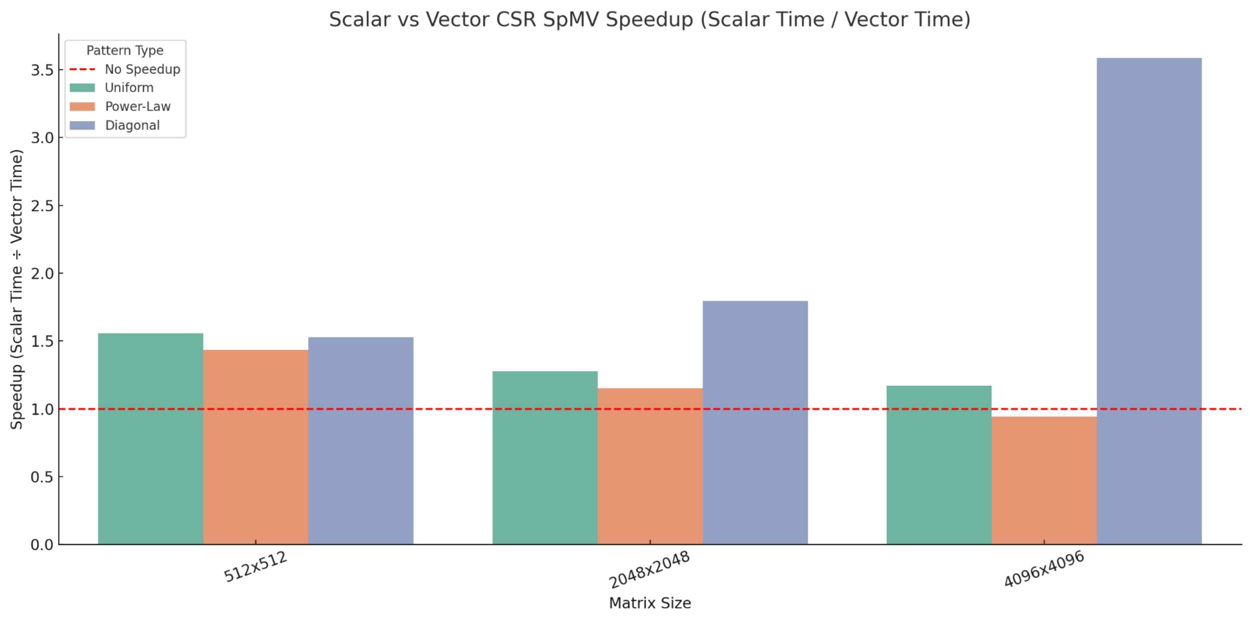




The data shows that for large matrices, the SELL-C format generally outperforms both CSR and ELL, due to its efficient parallelism and improved memory coalescing on modern hardware. However, exceptions arise with structured sparsity patterns such as arrowhead and triangular matrices, where CSR is more performant. In these cases, CSR's row-based structure aligns well with the sparsity pattern, resulting in less overhead and more efficient memory usage.

For smaller matrices, CSR and ELL tend to outperform SELL-C. CSR consistently outperforms ELL except in cases with uniform sparsity distributions, where ELL's fixed-length row storage leads to better memory access patterns and reduced branching. SELL-C is generally less effective for small matrices because the number of rows per slice is small, and the irregularity in non-zero distribution, especially in patterns like triangular or arrowhead, leads to significant padding. This padding increases memory footprint and introduces unnecessary computation, degrading performance. Additionally, the benefits of SELL-C's parallelism diminish on smaller datasets, where the overhead of slicing and alignment outweighs its advantages.

To re-emphasize the importance of selecting the right slice size for SELL-C, prior to generating the previous graphs, test cases were run with slices ranging from 2 to 64. Slices smaller than 32 almost consistently underperformed CSR and ELL implementations due to divergence. Because slice sizes were less than 32 (warp size), multiple slices of different sizes would map to the same warp. Due to their difference in size their respective threads would branch in different directions ultimately causing a slowdown due to divergence. Ultimately it was concluded that the ideal slice size for SELL-C on a GPU would be equal to the warp size. Unfortunately the data from those trials were lost.



The graph above compared the performance of warp-level vector and scalar implementations of CSR-based sparse matrix-vector multiplication (SpMV). Overall, the vectorized approach generally outperformed the scalar version but often yielded only modest speedups, around 1.5x. This limited gain was largely due to irregular sparsity patterns and variability in the distribution of non-zero elements, which led to warp divergence and inefficient memory access in CSR format. A clear example was seen with large power-law matrices, where the vector implementation underperformed. These matrices contained a few dense rows and many sparse ones, causing significant workload imbalance across warps.

Conversely, matrices with regular sparsity patterns, such as diagonal matrices, benefited more from vectorization. Diagonal matrices offer predictable row lengths and aligned column indices, enabling coalesced memory access and balanced work distribution, which improves performance. This benefit can be clearly seen once it's noticed that the speedup of the diagonal matrix scales as the size of the matrix grows. However, uniformly sparse matrices, despite having the same number of non-zeros per row, did not experience the same performance boost. This is likely because their non-zeros were not in consistent positions across rows, like how they would be in a diagonal. As a result, memory access patterns became less predictable and more scattered in CSR, reducing spatial locality and cache efficiency, and limiting the performance gain of the vectorized kernel.

| Work-load (rows × cols, sparsity, iters) | Variant | Kernel compute (µs) | Launch overhead (µs) | Compute-to-Launch Ratio |
|---|---|---|---|---|
| Small (256 × 256, 0.95, 2000) | Baseline loop | 20.665 | 2.865 | 7.21 |
| Small (256 × 256, 0.95, 2000) | GraphBLAS fused | 14.484 | 0.042 | 344.86 |
| Medium (1024 × 1024, 0.95, 200) | Baseline loop | 39.142 | 3.034 | 12.90 |
| Medium (1024 × 1024, 0.95, 200) | GraphBLAS fused | 33.928 | 0.388 | 87.44 |
| Large (4096 × 4096, 0.95, 200) | Baseline loop | 931.075 | 3.745 | 248.62 |
| Large (4096 × 4096, 0.95, 200) | GraphBLAS fused | 916.527 | 0.73 | 1255.52 |

The table above shows our recording of relative time spent on kernel compute vs kernel launch. It compares our single persistent kernel launch approach of GPU-GraphBLAS fused to a baseline of the commonly used multikernel launch approach.

Across all three matrix sizes we profiled, our persistent-kernel design eliminates launch latency as a first-order cost and therefore tightens the overall runtime, but its impact is most dramatic for the smaller workloads that characterise early BFS levels and many convergence-driven analytics. On the 256×256 graph (95 % sparse, 2000 iterations) a conventional host-driven loop spends 2.9 µs of every 20.7 µs cycle, roughly one out of seven milliseconds, waiting inside cudaLaunchKernel; the resulting compute-to-launch ratio is only 7:1. In our fused GraphBLAS kernel the same problem executes with a single 42 ns launch and a 14.5 µs device loop, pushing the ratio beyond 300:1 and cutting wall-clock time by 30%. Because launch overhead is essentially constant while arithmetic cost grows with graph size, the relative benefit shrinks for larger matrices, yet even at 4096×4096 the persistent grid still reduces host overhead five-fold (0.73 µs versus 3.75 µs) and drives the compute-to-launch ratio past 1250:1. The takeaway is clear: when an algorithm performs many low-intensity iterations, per-iteration kernel launches become the bottleneck; moving the loop onto the GPU converts that fixed latency into a one-time fee and lets the hardware devote virtually all of its cycles to useful graph work.

The kernel compute time and launch overhead were estimated by averaging the total duration of kernel calls. Using NVIDIA Nsight Systems, we obtained the time spent in kernel launch functions, cudaLaunchKernel for the baseline and cuLaunchCooperativeKernel for the GraphBLAS_fused implementation. The compute time was then calculated by subtracting the measured launch time from the total kernel duration, as reported under CUDA Kernel Statistics in Nsight Systems.

# References

- Anzt, H., Tomov, S., & Dongarra, J.J. (2014). Implementing a Sparse Matrix Vector Product for the SELL-C / SELL-C-σ formats on NVIDIA GPUs.
- Evtushenko, G. (2019, November 22). Sparse matrix-vector multiplication with Cuda. Medium. https://medium.com/analytics-vidhya/sparse-matrix-vector-multiplication-with-cuda-42d191878e8f
- AlAhmadi, S., Mohammed, T., Albeshri, A., Katib, I., & Mehmood, R. (2020). Performance Analysis of Sparse Matrix-Vector Multiplication (SpMV) on Graphics Processing Units (GPUs). Electronics, 9(10), 1675. https://doi.org/10.3390/electronics9101675
- Kepner, J., Aaltonen, P., Bader, D., Buluç, A., Franchetti, F., Gilbert, J., Hutchison, D., Kumar, M., Lumsdaine, A., Meyerhenke, H., McMillan, S., Moreira, J., Owens, J. D., Yang, C., Zalewski, M., & Mattson, T. (2016). Mathematical foundations of the GraphBLAS. arXiv preprint arXiv:1606.05790. https://doi.org/10.48550/arXiv.1606.05790
- Yang, C., Buluç, A., & Owens, J. D. (2018). Implementing push-pull efficiently in GraphBLAS. Proceedings of the International Conference on Parallel Processing (ICPP).
- Yang, C., Buluç, A., & Owens, J. D. (2019). GraphBLAST: A high-performance linear algebra-based graph framework on the GPU. arXiv preprint arXiv:1908.01407.
- The GraphBLAS User Guide & C API Specification (https://github.com/DrTimothyAldenDavis/GraphBLAS/blob/stable/Doc/GraphBLAS_UserGuide.pdf)
- Hwang, C., Park, K., Shu, R., Qu, X., Cheng, P., & Xiong, Y. (2023). ARK: GPU-driven code execution for distributed deep learning. In Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI). USENIX Association.

## Distribution of Work

Work was distributed evenly between both group members.