Paper: CB-SpMV: A Data Aggregating and Balance Algorithm for Cache-Friendly Block-Based SpMV on GPUs

Source: Xing Cong, FuKai Sun, YiFan Chen, Chenhao Xie, Yi Liu, and Depei Qian. 2025. CB-SpMV: A Data Aggregating and Balance Algorithm for Cache-Friendly Block-Based SpMV on GPUs. In 2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3721145.3725746

Introduction

Sparse Matrix-Vector Multiplication, commonly known as SpMV, is a fundamental operation in computer science. It serves as the calculation engine behind many applications, including engineering simulations, scientific data processing, and machine learning training. The operation is mathematically straightforward: we calculate $y = Ax$, where A is a sparse matrix containing mostly zeros, and x is a dense vector. The goal is to update the output vector y. To do this, the computer typically iterates through the non-zero elements of the matrix, multiplies them by the corresponding values from the input vector, and accumulates the results. Despite its simplicity, optimizing SpMV on modern hardware like Graphics Processing Units (GPUs) presents considerable challenges.

I am interested in this topic due to my recent exploration of the PageRank algorithm. I have been working on parallelizing PageRank using OpenMP already. PageRank is essentially an iterative SpMV operation. The adjacency matrix of a web graph or a social network corresponds to the sparse matrix A, and the rank values form the vector x.

During my project, I encountered several performance issues that proved difficult to address. The graph data I used appeared random and irregular. When I used the standard Compressed Sparse Row (CSR) format to store the matrix, the performance seemed not that well. The GPU threads appeared to spend a significant amount of time waiting for data from memory rather than computing. I also attempted to use basic blocking techniques to potentially improve memory access, but the gains were limited. Additionally, I struggled with load balancing. Real-world graphs often follow a power-law distribution, where a small number of "hub" nodes have many connections while most have few. This seemed to cause some GPU threads to process for extended periods while others finished quickly and idled.

The paper "CB-SpMV: A Data Aggregating and Balance Algorithm for Cache-Friendly Block-Based SpMV on GPUs" addresses these types of problems. The authors propose a new framework called CB-SpMV. They identify three main areas for improvement in current methods: data locality, hardware utilization, and load imbalance. They introduce a new way to store data using blocks and a set of adaptive algorithms intended to solve these issues. This report will describe the problems they identified, the specific techniques they used to solve them, and how these solutions relate to my work on graph algorithms.

Problem Analysis

The paper analyzes the limitations of the Dense/COO/CSR format and existing block-based formats like Block Compressed Sparse Row (BSR).

The first significant issue is **data locality**. In the CSR format, which I used in my PageRank implementation, the matrix data is typically split into three arrays. The column indices are stored in one array, and the numerical values are stored in a separate array. When a GPU thread performs the multiplication, it generally needs to do two things. First, it fetches a column index to determine which part of the vector x to read. Second, it fetches the actual matrix value to perform the multiplication.

The challenge is that the column index array and the value array are stored in different memory locations. This creates a skipping access pattern. When the GPU fetches the index, it does not automatically load the corresponding value into the cache because the value is stored elsewhere. This can lead to a high number of cache misses. The GPU may have to make multiple requests to the main global memory. The paper suggests that this structure significantly reduces the hit rates of the GPU's L1 and L2 caches. In my PageRank project, I suspected this was a primary reason for low bandwidth utilization (i.e. the caches of CPU).

The second issue is **hardware utilization**. This relates to how the sparse nature of graphs is handled. Previous researchers developed block-based methods like BSR to address locality issues. BSR groups non-zero elements into small dense blocks. This approach tends to work well for structured matrices in physics simulations. However, graph data is often highly randomized. Attempting to force graph data into fixed dense blocks can result in storing many zeros, which may waste memory and bandwidth.

Conversely, if blocks are stored as sparse structures, GPU execution efficiency can be impacted. GPUs execute threads in groups of 32, known as a warp. If a sub-block has only a few non-zero elements, the GPU typically still assigns a warp to process it. This means only a few threads in the warp are active, while the rest may remain idle. This under-utilization of hardware represents a notable inefficiency.

The third issue is **load imbalance**. This is particularly relevant to my experience. Real-world graphs usually follow a power-law distribution. A small number of hub nodes have large workloads, while some other nodes have very little. If work is assigned to GPU thread blocks based on row count or block count, the distribution of actual work (non-zero elements) can be uneven. Some Streaming Multiprocessors (SMs) may finish early, while others are stuck processing dense rows. The total time of the algorithm typically depends on the slowest SM. The paper notes that many existing block methods process data at the warp level but may fail to adequately balance the total load at the thread block level.

Data Structures and Algorithms

The authors propose the CB-SpMV framework to address these three specific problems. Their solution uses a 2D blocking structure combined with adaptive strategies for storage and execution.

1. 2D Sparse Structure and Intra-Block Data Aggregation

The core of their design involves dividing the matrix into independent sub-blocks of size 16x16. They chose this size because it maps well to the GPU warp size. However, the key innovation lies in how the data within these blocks is stored.

The authors introduce a technique called **Intra-Block Data Aggregation**. In standard CSR, coordinates and values are often far apart. CB-SpMV changes this by treating each sub-block as an independent unit. Inside each unit, the coordinate data and the numerical value data are packed together into one continuous segment of memory.

To manage this, they use a high-level structure that resembles a Coordinate (COO) list for blocks. It stores the row and column indices of the blocks. Most importantly, it stores a Virtual Pointer for each block. This VP is an offset that points to the starting byte in global memory where that block's data begins.

This design aims to improve locality. When a GPU thread accesses a block using the Virtual Pointer, it fetches the packed data. Because the coordinates and values are physically adjacent, loading the coordinates into the cache increases the likelihood of loading the values into the cache simultaneously. This functions as a software-managed prefetching strategy, potentially ensuring that data needed for computation is available in the fast L1 cache and reducing the need to access global memory.

Inside these packed blocks, the authors apply **Coordinate Compression**. Since the block size is fixed at 16x16, the row and column indices inside a block are always between 0 and 15. Standard CSR uses 32-bit integers for indices, which can be inefficient for such small numbers. The authors compress the row index (4 bits) and the column index (4 bits) into a single 8-bit byte. This reduces the amount of memory needed for metadata.

However, there is a technical challenge regarding **memory alignment**. GPUs tend to read data more efficiently when the memory address aligns with the data size. For example, a `double` precision value (8 bytes) should ideally be stored at an address that is a multiple of 8. If 1-byte coordinates are mixed with 8-byte values, the values might end up at unaligned addresses, potentially slowing down the memory controller. To address this, the authors use a padding strategy. They calculate where the values should start and insert empty "padding" bytes after the coordinates. This ensures that the numerical values typically start at an aligned memory address, maintaining potential throughput.

2. Computation Adaptation Strategy

The authors recognize that graph matrices are often irregular. Some parts are sparse, and some are dense. A single GPU kernel may not handle all these cases efficiently. Therefore, they propose a **Computation Adaptation Strategy**. This means the algorithm checks the sparsity of each sub-block and selects an appropriate way to process it.

For **Sparse Sub-blocks**, the goal is to improve thread utilization. The paper introduces "block-aware column aggregation." In a sparse 16x16 block, many columns might be empty. The algorithm removes these empty columns from the storage. This compresses the block horizontally and increases the density of the remaining data. When a warp processes this compressed block, the threads are more likely to perform useful work instead of checking empty spots.

However, removing columns changes the column indices. To find the correct element in the input vector x, the kernel needs to perform an extra lookup to restore the original index. This adds a small cost. To balance this, the authors use a threshold. They typically apply this aggregation only when the block is sparse enough that the gain in efficiency is likely greater than the cost of the lookup.

For **Dense Sub-blocks**, the strategy changes. If a sub-block has many non-zeros, using atomic operations to add results to memory can be slow because multiple threads might try to write to the same location. For these blocks, the framework switches to use either a CSR format or a Dense format.

When using these formats for dense blocks, the GPU kernel can utilize warp-shuffle instructions. These instructions allow threads in a warp to share data directly through their registers. This enables the threads to sum up their partial results quickly without writing to global memory. This can significantly speed up the processing of dense areas, like the hub nodes in a graph. The framework uses specific thresholds (block non-zero counts of 32 and 128) to decide automatically which format to use.

3. Inter-Block Load Balancing

To address the load imbalance problem caused by the power-law distribution, the paper introduces a load balancing algorithm. This occurs during the preprocessing stage. The goal is to ensure that every Thread Block (TB) is assigned roughly the same total amount of work (non-zero elements).

The algorithm uses a **priority queue** (specifically a min-heap) to schedule the tasks. It typically works in the following way: First, it calculates the number of non-zeros for every sub-block. It also creates a heap to track the current load of every Thread Block. Initially, all Thread Blocks have zero load. In the scheduling loop, the algorithm takes the sub-block that has the most non-zeros. It assigns this block to the Thread Block that currently has the lowest load (the one at the top of the heap). After assignment, it updates the load of that Thread Block and puts it back into the heap.

This approach is just simply a kind of greedy algorithm. It attempts to distribute the heavy blocks evenly among the Thread Blocks. This helps ensure that no single Thread Block is stuck with excessive work while others are idle.

Crucially, the paper mentions that they do not just assign the blocks logically. They actually **reorder the data in memory**. After deciding which block goes to which Thread Block, they rearrange the metadata arrays in the GPU's global memory. This ensures that when a Thread Block runs, it reads its assigned blocks in a linear sequence. This maintains the sequential memory access pattern that is often essential for performance.

Evaluation

The authors tested their framework extensively, and the performance results are notable. On average, the CB-SpMV method achieved speedups of about 3 times compared to `cuSPARSE-BSR` and `TileSpMV`. This represents a significant improvement for such a fundamental operation.

For me, a key result is the **cache hit rate analysis**. On the RTX 4090 GPU, the proposed method improved the L1 cache hit rate by 82% compared to previous block-based methods. The L2 cache hit rate improved by 19%, which is also significant. This reveals that the Intra-Block Data Aggregation technique is effective. It suggests that the Virtual Pointer system seems to successfully address the memory jumping problem. Considering my own algorithm, I am able to make my PageRank implementation better by optimizing the main bottleneck, the inefficient cache usage of the CSR format.

Conclusion

The paper CB-SpMV presents a comprehensive solution to the problems of sparse matrix computation on GPUs. It moves away from the rigid structure of the CSR format. Instead, it uses a flexible, block-based approach that aims to adapt to the data.

The framework is trying to address the three main bottlenecks I identified in my introduction. Reflecting on my own work with parallel PageRank, this paper provides a clear path for potential optimization. The problems I faced—random memory access and uneven workload—are exactly what this paper attempts to solve. The technique of Intra-Block Data Aggregation seems directly applicable to my project. I could implement a similar packing strategy to potentially reduce cache misses during the graph traversal.

Furthermore, the Load Balancing strategy appears practical for PageRank, although this kind of dynamic scheduling is hard to implement in OpenMP. Also, one might argue that sorting blocks and reordering memory takes time. However, PageRank is an iterative algorithm. We run the multiplication step 20 to 50 times on the exact same graph. This means I would only need to perform the preprocessing and load balancing once at the beginning. The cost is amortized over all the subsequent iterations. This makes the efficiency gains in the kernel execution beneficial.

In conclusion, CB-SpMV demonstrates that achieving high performance on irregular graph workloads requires careful optimization. It suggests that we may need to look beyond standard formats like a pure CSR, and to redesign our data structures to fit the GPU architecture better.