# Optimizing Tiled Sparse Matrix Multiplication with Performance Modeling on GPUs

Anonymous Author(s)

## Abstract

Sparse matrix - dense matrix multiplication (SpMM) is one of the fundamental computational kernels widely used in scientific computing, artificial intelligence, and graph analytics applications. The SpMM throughput is typically bound by memory bandwidth and can potentially benefit from the high memory bandwidth of modern GPUs. However, load imbalance and poor data reuse across threads can significantly impact the SpMM throughput on GPUs. For guidance on algorithmic optimizations for improved load balance and data reuse, such as loop reordering and tiling, one needs reliable analytical performance modeling of SpMMs on GPUs. Nevertheless, it is not trivial to predict the SpMM throughput on GPUs due to the unstructured nature of its memory accesses and the high complexity of memory hierarchy on GPUs. In this work, we develop a simple yet reliable performance model based on the arithmetic intensity of SpMM kernels with advanced tiling strategies. The proposed intensity equation considers register- and scratchpad-tiling strategies and captures the performance implications of (1) the sparsity pattern of memory accesses and (2) memory and the processor architecture of GPUs with high fidelity. We benchmark the proposed model with 200 cases on NVIDIA A100 GPUs to assess the accuracy of the largest sparse matrices in the open-source SuiteSparse matrix collection. With the guidance of the performance model, we apply load balancing and row reordering optimizations to predictably improve the SpMM tiling performance. As a result, we obtain 19.2× average speedup over state-of-the-art cuSPARSE SpMM with 2.03× geometrical mean speedup. For reproducibility, we open source Tiled SpMM repository.[1]

***Keywords:*** GPU Computing, Sparse Matrix Multiplication (SpMM), Performance Modeling

## 1 Introduction

Sparse matrix-dense matrix multiplication (SpMM) is a fundamental computational kernel widely used in many scientific [15], artificial intelligence [11, 16, 19], and data analytics applications [19]. More than often, SpMM takes a significant portion of the end-to-end execution time [14]. SpMM kernels with high sparsity levels have low arithmetic intensity, and hence its throughput is bounded by the memory bandwidth of the running processor [23]. Therefore, graphics processing units (GPUs) are used as a preferred accelerators for computing SpMM due to their high memory bandwidths. To further improve the performance and exploit the modern GPU hardware features, prior work has proposed optimizing SpMM for GPUs using varied techniques such as loop transformation [23], tiling [18], and kernel fusion [29]. These techniques enable the algorithm to improve the on-chip data reuse by efficiently exploiting the use of registers and scratchpad memory on modern GPUs [11, 16, 18].

Among these optimizations, optimizing SpMM through tiling is popular and critical [38]. This work focuses on a specific strategy that stages irregular memory accesses on the scratchpad memory and accumulates the result on registers [14, 16, 18]. Moreover, previous implementations are usually optimized for the row-major storage of dense matrices [18] or implemented for specific applications [14, 16], while this work applies the state-of-the-art SpMM tiling to column-major dense matrices to support a wide variety of new applications based on sparse matrix-multiple vector multiplication. This implementation is referred to as *Tiled SpMM* in this work.

Analytical performance modeling of an algorithm is crucial for providing decision support for high-performance optimizations. Examples include but not limited to: kernel selection for high-performance libraries [2], tuning guidance for specific architectures [9], and hardware/software co-design [4, 42, 43]. Nevertheless, it is non-trivial to develop a model that is portable to various processor architectures and applicable to varied applications while providing sufficient fidelity to capture performance implications. Yet another challenge is that the algorithm exhibits irregular memory access behavior and the memory access locality characteristics are input dependent. Therefore, the performance of a given SpMM implementation on a given system can vary significantly. This makes creation of high fidelity performance model for SpMM on GPU a very challenging task as it directly depends on the input sparsity of the sparse matrix.

Performance modeling using arithmetic intensity has been studied on CPUs [12]. Nevertheless, it does not consider caching mechanisms that can significantly affect performance. Model-driven optimization approaches on GPUs have been studied in the sparse matrix - dense vector multiplication (SpMV) context [9]. However, it does not take the

[1] The link will be published upon acceptance of this paper.

*PPoPP 2023, February 25–March 1, 2023, Montreal, Canada*
2023.

recently developed register- and scratchpad-tiling optimizations for GPUs [16, 18]. Furthermore, previous work does not include input-dependent sparsity pattern into the equation.

Within this context, this work presents a novel high-fidelity performance analytic model for accurate prediction of the Tiled SpMM throughput. This analytical model is built on the Roofline model [37] that characterizes an algorithm as either compute-bound or bandwidth-bound, depending on its arithmetic intensity, (i.e., compute-to-memory-access ratio). The proposed model predicts the arithmetic intensity by inspecting the sparsity pattern of the involved matrices. More specifically, this model considers (1) architectural parameters (e.g., memory bandwidth, floating-point precision, memory access granularity) at compile time and (2) data-dependent parameters (sparsity level, sparsity pattern) at run time.

Moreover, based on the new performance model, this work presents two model-driven optimizations for Tiled SpMM: The first optimization targets the cases whose measured performance falls short of the model prediction, and employs a load balancing mechanism to improve the computation performance. The proposed model is particularly used for tuning the optimization parameters for detection and segmentation of imbalanced sparse matrices. The second optimization targets the cases whose measured memory throughput is significantly below the model prediction, and leverages row-reordering to improve the data locality of the sparse matrix in Tiled SpMM. In this case, the proposed model is used to guide row reordering decisions. With the proposed model-driven optimizations, the optimized Tiled SpMM attains the performance bound predicted by the proposed analytic model. Our optimization workflow is summarized in Fig. 1.

In summary, this work makes the following contributions:

- It analyzes the SpMM operations on GPUs and provides insights into the causes of performance variations.
- It proposes a novel performance analytic model for Tiled SpMM on GPUs.
- With the guidance of the performance model, it detects underperforming cases and applies additional load balancing and row reordering mechanisms for improving performance.
- For assessing the accuracy, it benchmarks the Tiled SpMM and the proposed optimizations with the performance model over 200 application cases on A100 GPU.

Our evaluation results show that: 1) our proposed analytic performance model is reliable; 2) with model-guided optimizations (load-balancing and row-reordering), the optimized Tiled SpMM provides 6.37× average speedup over the baseline Tiled SpMM, and an overall 2× geo-mean speedup over the SpMM algorithm of Nvidia cuSPARSE library. This work provides a deep understanding of performance parameters for further optimization of sparse algorithms. The proposed performance analytic model (of Tiled SpMM) is also portable over the next generation GPUs and a wide-variety of application cases. For reproducibility of this work, we will open-source our codebase upon acceptance of this manuscript.

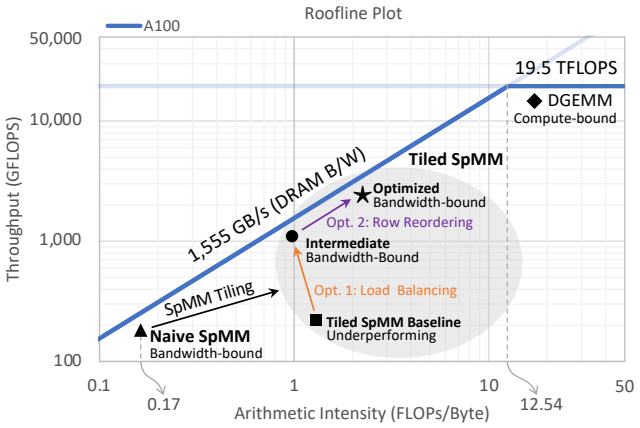


**Figure 1.** Representation of the proposed optimizations on the roofline plot. SpMM tiling improves the arithmetic intensity, nevertheless, the performance may not attain maximum potential performance. We use the proposed Tiled SpMM model as a guidance to further pre-process the input matrix for optimization.

## 2 Background

This section introduces the SpMM operation and a baseline implementation, discusses the Roofline model and modeling of the baseline SpMM, and proposes a simplified abstract machine for modeling the Tiled SpMM performance on modern GPU architectures.

### 2.1 SpMM

SpMM multiplies an $M \times N$ sparse matrix $\boldsymbol{A}$ by an $N \times K$ dense matrix $\boldsymbol{B}$ and finds an $M \times K$ dense matrix $\boldsymbol{C}$ as the output. Without loss of generality, we assume that the sparse matrix is given in CSR format and the dense matrices are stored in column-major format. We choose the column-major format for the dense matrices due to our target applications [14, 16]. Those applications store dense matrices natively as multiple column vectors and multiplies each dense vector with the sparse matrix, i.e., sparse matrix-multiple vector multiplication. Furthermore, we do not assume any particular sparsity pattern for the sparse matrix for generality.

Figure 2 depicts the memory accesses of the baseline SpMM implementation on GPU [33]. Each output element of $\boldsymbol{C}$ is computed by a single thread, and therefore, a total of $M \times K$ threads are put on work. To compute the output element $(m, k)$, the corresponding thread performs an inner product of the $m$th row of $\boldsymbol{A}$ and the $k$th column of $\boldsymbol{B}$. Since the row is sparse, each thread multiplies and adds $0 \leq \mu \leq N$ nonzeroes on average, where $\mu$ is the mean number of nonzeroes that are distributed among $M$ rows. In this example, there are two nonzeroes on the $m$th row, the thread $(m, k)$ reads two nonzeroes of $\boldsymbol{A}$ and reads the matching two

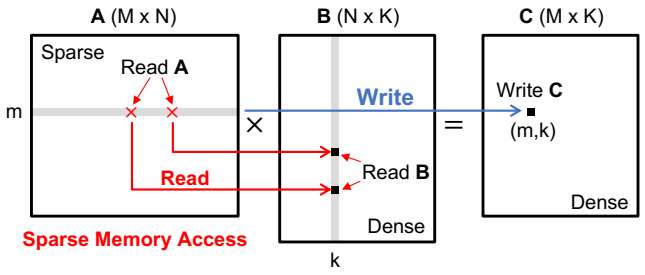


**Figure 2.** In the massively-parallel baseline SpMM, one thread computes a single output $(m, k)$ by performing a sparse inner product of the $m$th row of $\boldsymbol{A}$ and the $k$th column of $\boldsymbol{B}$.

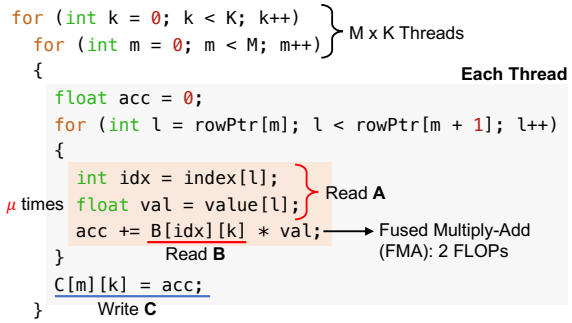


**Figure 3.** The naive SpMM implementation. For massive parallelization, the outer loops are mapped to $M \times K$ threads and each thread executes the highlighted section involving FLOPs and memory accesses.

elements of $\boldsymbol{B}$. Then, the thread writes the result of the inner product to $\boldsymbol{C}$.

The memory accesses to $\boldsymbol{B}$ are sparse, and are costlier than writing to $\boldsymbol{C}$ because of the overhead of the memory subsystem for serving sparse memory accesses. We will model that cost in Section 3. But first, we will present the roofline model in the next subsection to understand the performance implications of memory accesses and communications.

## 2.2 The Roofline Model: A Pedestrian Description

The roofline model [37] helps finding theoretical limitations of how fast an algorithm runs on a processor. According to the Roofline model, the operating regime of an algorithm depends on its arithmetic intensity:

$$I = \frac{\sum \text{Floating-Point Operations (GFLOPS)}}{\sum \text{Memory Accesses (GB/s)}} \quad (1)$$

in the unit of floating-point operations (FLOPs) per byte. In other words, $I$ is a measure of the number of FLOPs performed for every byte brought from the memory.

When the compute throughput is measured in a system of interest (in FLOPs per second, i.e., FLOPS), we can place an algorithm in the roofline plot for that system. As an example, Figure 1 shows a roofline plot with the arithmetic intensity on the $x$ axis and the compute throughput on the $y$ axis. The light blue horizontal and skewed lines are determined by the peak compute throughput and peak memory bandwidth of Nvidia A100 GPU, respectively [1]. Note that these peak values are theoretical (calculated by the clock rate, number of channels, number of ALUs, etc.) and an execution may not be able to reach the peak values in practice.

The *critical intensity* of a processor is defined as the ratio of the peak compute throughput and the peak memory bandwidth (12.54 in the A100 case) that depends on the processor's theoretical limits. As an example, tiled dense general matrix multiplication (DGEMM) algorithm is placed in Figure 1 as a compute-bound algorithm [35]. On the other hand, SpMM has an arithmetic intensity less than 12.54 FLOPs/byte and hence it is bounded by the memory bandwidth [12]. Algorithms that do not saturate either the memory-bandwidth or the compute bound are identified as latency-bound (underperformed), as depicted in the figure. One prominent reason is the load imbalance between processing elements. We fix the load imbalance problem of Tiled SpMM in Section 4.

## 2.3 The Naive SpMM Model

We can place the naive SpMM at the corner of the roofline plot in the Fig. 1. Because, the single-precision (FP32) arithmetic intensity of the naive SpMM is in the range of 0.08–0.17 FLOPs/byte, and hence memory bandwidth bound is in the range of

$$130 \text{ GFLOPS} \leq I\beta \ddot{Y} 260 \text{ GFLOPS}$$

according to the roofline model, where $I$ is the intensity defined in Eq. 1 and $\beta = 1,555$ is the memory bandwidth of A100.

To derive the arithmetic intensity model of the naive SpMM, we need to look at the implementation code, and simply count the number of FLOPS and the memory accesses in bytes that each thread performs. To elaborate, the naive SpMM code with CSR data structure [7] is shown in Figure 3. The sector of code that is executed by each thread is highlighted in with gray, which corresponds to an inner product depicted in Fig. 2. We count the number of FLOPs first: Each thread performs $\mu$ fused multiply-adds in line 9, which are counted as $2\mu$ FLOPs and placed in the numerator of Eq. 2. We count the number of bytes to place in the nominator of Eq. 2 as follows.

First, in the 5th line, each thread reads the offsets for the first and last nonzeroes of the $m$th row from the CSR data structure `rowPtr`. The offsets are typically eight bytes, which is symbolized as $\delta$ in Eq. 2. Next, each thread reads $\mu$ elements of the `index` and `value` arrays, which are symbolized with $B_I$ and $B_T$ in Eq. 2, respectively. As a result, each thread reads $\mu(B_T + B_I)$ bytes on average, where $B_T$ is the number of bytes per `value`, and $B_I$ is the number of bytes per `index` value. Typically, $B_I$ is four bytes for integer and $B_T$ is either two, four, or eight bytes, for half-, single-, or double-precision

floating point numbers, respectively. Then, according to the indices read from the `index` array, each thread reads $\mu$ elements from the corresponding $k$th column of $\boldsymbol{B}$ in Line 9, which adds $\mu B_T$ bytes in the denominator. Finally, each thread write $B_T$ bytes to $\boldsymbol{C}$ in Line 11 in Fig. 3, which adds $B_T$ bytes to the denominator of Eq. 2. As a result, we can write the intensity formula as

$$I = \frac{2\mu}{\mu(B_T + B_I) + \mu B_T + B_T + \delta}. \quad (2)$$

The results of the inner product are accumulated in a register, named with `acc` in Line 9 of Fig. 3, and we do not count reads and writes to register.

From the naive SpMM intensity model (Eq. 2, one observation we can make is that the intensity depends on $\mu$, which can also be seen as the density of the sparse matrix: The higher the density, the higher the intensity because the nominal effect of reading the metadata and writing the output per thread becomes insignificant as $\mu \gg B_T + \delta$. Therefore we can find the range of arithmetic intensity as $0.08 \leq I \leq 0.17$. As a result, the upper-bound of the performance model suggests that the naive SpMM can utilize at most 260 FLOPS, i.e., 1.33%—a tiny fraction of the peak FP32 throughput of the A100 GPU.

### 2.4 Rules of the Game: The Abstract Machine Model

This section lists the baseline assumptions that we make for modeling the arithmetic intensity of Tiled SpMM in a GPU system. As the real GPU memory systems are very complex, a practical model should only consider the most pertinent features and characteristics. In other words, we simplify the complex GPU memory systems into an *Abstract Machine* with only a small number of features that are considered on our performance model. In other words, our model would be exact on the abstract machine with the following features:

1. All threads run in parallel with **no arithmetic latency**.
2. All threads run in parallel with **no memory latency**. That is, there are sufficient parallel memory instructions on-the-fly at any given time for saturating the memory bandwidth. When the sparse matrix is imbalanced, this assumption is violated and the Tiled SpMM does not saturate the memory bandwidth.
3. All matrices fit into the main memory (DRAM) of the GPU, i.e., the abstract machine has infinite memory.
4. On-chip memory accesses, i.e., to registers and L1/L2 caches, incur negligible cost and thus are not counted.
5. Bytes transferred with each DRAM access are counted.
6. Threads with consecutive indices are group of as warps. Each warp executes instruction in a lock-step, i.e., single instruction multiple threads (SIMT) manner [31].
7. DRAM access granularity is equal to a cache line size. That is, whenever a piece of data is needed from the GPU memory, the entire cache line that contains that piece of data will be accessed from the GPU memory. In fact, caching to L2 has a sector granularity, and we treat two sectors (64 byte) as a single cache line.
8. Writes to the output matrix are counted twice. The write request cannot find the data in the cache, and therefore a read instruction has to be issued. Therefore writing the output requires two memory transactions.

These features are used in the derivation of the mathematical terms of the performance model expressions.

**Table 1.** Reference Table for Arithmetic Intensity Model in Eq. 3.

| Parameter | Explanation |
|---|---|
| $B_T = 4$ (bytes) | # Bytes per floating-point number |
| $B_{IS} = 2$ (bytes) | # Bytes for indexing SRAM |
| $B_I = 4$ (bytes) | # Bytes for indexing DRAM |
| $\beta = 1,555$ (GB/s) | DRAM bandwidth in GB/s |
| $1 \leq \rho \leq M$ | SRAM data reuse (input dependent) |
| $0 \ddot{Y} \mu \leq N$ | Average nnz per row (input dependent) |
| $1 \leq R$ | Register tiling factor |
| $0 \ddot{Y} \delta$ (bytes) | # Bytes of metadata per thread |
| $N/\mathsf{nnz} \leq \Gamma$ | Sparse access cost (Eq. 4) |
| $1 \leq Z$ | S-ELL zero-padding overhead |

### 2.5 Overview of the Tiled SpMM Baseline

Tiled SpMM re-implements the tiling strategy that are described by Hong, et al. [18], and Hidayetoglu, et al. [16]. But previous from previous work, our implementation is optimized for the column-major storage of $\boldsymbol{B}$ and $\boldsymbol{C}$. In this section, we provide an overview of scratchpad memory (SRAM) and register tiling strategies.

Before the Tiled SpMM execution, we perform an inspection of the inspect the sparsity pattern of $\boldsymbol{A}$, and determine the spare memory access footprint on $\boldsymbol{B}$ for each thread block. Then, we build the multi-stage tiling data structures for loading only the required elements from $\boldsymbol{B}$ from DRAM to the scratchpad memory (SRAM). Then, we perform a partial SpMM from SRAM, that provides on-chip data reuse. We will represent as $\rho$ in the rest of the paper. For saving memory bandwidth, we use two-byte indices for addressing SRAM, which is represented as $B_{IS}$. For reusing the sparse matrix $\boldsymbol{A}$ from registers, we apply register tiling. In this case, each thread computes $R$ output, where the sparse matrix is reuse $R$ times. In the rest of this paper, we refer $R$ as the register-tiling factor.

In addition, we implement the S-ELL representation for the sparse matrix [25]. The S-ELL data structure provides fully-coalesced (ideal) data access to $\boldsymbol{A}$ with a zero-padding overhead. To minimize the overhead we apply zero-padding in warp granularity. We represent the S-ELL overhead with $Z$ in the rest of the paper.

# 3 The Tiled SpMM Model

To improve performance, Tiled SpMM provides data reuse at on-chip registers and scratchpad memory (SMEM) such that the DRAM traffic is reduced as explained in Section 2.5.

In this section, we present the Tiled SpMM model by updating Eq. 2 with additional tiling-specific parameters $\rho$, $R$, and $B_{IS}$, and $Z$ as

$$I_e = \frac{2}{\frac{(B_T + B_{IS})Z}{R} + \frac{B_T\Gamma}{\rho} + \frac{B_I}{\rho\mu} + \frac{\delta}{R\mu}}. \quad (3)$$

In this section, we go over the parameters in the Tiled SpMM model with greater detail to clarify the model for the reader. A brief explanation of these variables is given in Table 1.

**3.0.1 Algorithmic Parameters.** In this section, we summarize the variables in the algorithmic intensity of Tiles SpMM: $R$, $\rho$, $B_{IS}$, and $Z$. These variables are a result of the proposed register and scratchpad tiling algorithms. Please find the summary below:

For more detail, we explain the model parameters in the rest of this section. Note that $\Gamma$ in Table 1 is included only in the effective intensity model—not in the algorithmic intensity model.

- *R: Register Tiling Factor.* In register tiling, each thread computes multiple outputs in the same row of $\boldsymbol{C}$. That allows reusing the nonzeroes of $\boldsymbol{A}$ for multiplication of many columns. Therefore, the number of bytes to read sparse matrix is reduced by a factor of $R$.
  Register tiling requires a tuning to find a good compromise between the algorithmic speedup and slowdown from register pressure. We empirically found out that $R = 16$ provides good speedup. Allocating a large number of registers causes register spill and cancels the algorithmic speedup by the register tiling.
- $\rho$*: SRAM Reuse Factor.* Tiled SpMM employs shared-memory tiling for reusing $\boldsymbol{B}$ from shared memory. $\rho$ is the average number of reuse from on-chip scratchpad. This variable depends on the sparsity pattern of the matrix, and therefore cannot be known at compile time. This parameter is calculated by counting the average number of nonzeroes per column in every row panel.
- $B_{IS}$*: Bytes for indexing SRAM.* Tiled SpMM saves DRAM bandwidth by re-indexing the shared-memory data using INT16 (can address up to 64KB scratchpad memory). In a single-precision execution, this optimization provides 25% additional bandwidth saving when reading sparse matrix in single precision, i.e., reads $B_T + B_{IS} = 6$ bytes rather than $B_T + B_I = 8$ bytes, and hence provides speedup.
- $Z$*: Zero-padding overhead.* Tiled ELL data structure provides fully-coalesced (very desirable) memory access to DRAM; however, depending on the matrix, it can cause significant zero-padding overhead.

**3.0.2 Modeling Sparse Data Access Cost.** The effective intensity includes the effect of the memory subsystem. Most significantly, the caching effect is two-fold: 1) If data reuse from L2 cache is large, it helps on speedup by the reuse factor $L$. On the other hand, 2) if memory accesses are uncoalesced (such as in the sparse memory access) per-access cost increases by $\Gamma$—the sparse data access cost. As a result, we can find an average effect by modeling the overall sparse access cost as

$$\Gamma = \frac{C}{L}, \quad (4)$$

where $C$ and $L$ are the penalty and reward of the caching mechanism during the sparse memory access. Here, $1 \leq C, L$, and if $\Gamma$ ¡ 1, it means the cost of irregular access is dominant and if otherwise, $\Gamma$ Ÿ 1, the benefit of cache reuse is dominant.

To model $C$ and $L$, we look at the sparsity pattern of the memory accesses when we load $\boldsymbol{B}$ into shared memory. We also take the architecture of the memory subsystem into account, i.e., cache line size, sector cache, etc. The rules for modeling $C$ and $L$ are given below.

- *C: All miss (Worst Case) Cost Model.* To simulate the most costly scenario, we assume each memory access is a miss and therefore it has to be brought with a cache line. In the irregular access case, most of the data in the cacheline will not be used by another thread in the same warp. In the proposed model of $C$, we count the total number of cache lines by an inspection of the sparsity pattern of $\boldsymbol{A}$ as shown in Figure 4.

  $$C = \frac{\#\text{ of cache lines (touched)} \times \text{cache line size (bytes)}}{\text{warp size} \times B_T}$$

  In a coalesced access, where each thread accesses a consecutive element, we set $C = 1$. In misaligned access, we set $C = 1$ because the unused portion will most probably be used from L1 by another warp in the same thread block.
- *L: Cold Miss (Optimistic) Cost Model.* This model counts only the cold accesses as in the infinite cache model. That is, once the cost of irregular access cost is paid to read a sparse data, it will stay in the cache forever. The speedup relies on the L2 cache reuse that is shared among thread blocks that overlaps at a given time in the dynamic scheduling. To find an optimistic value, we assume a full L2 reuse between the whole set of scheduled thread blocks. We find $L$ simply by dividing the total memory footprint of all thread blocks by $M$, which gives us the average potential L2 reuse. $L$ depends on the sparsity pattern of the matrix but not the architecture in this model.

# 4 Optimizations

Using the proposed model, we propose the following two performance optimizations: (1) Load balancing with segmented sum and (2) Row reordering with radix sort. We use the

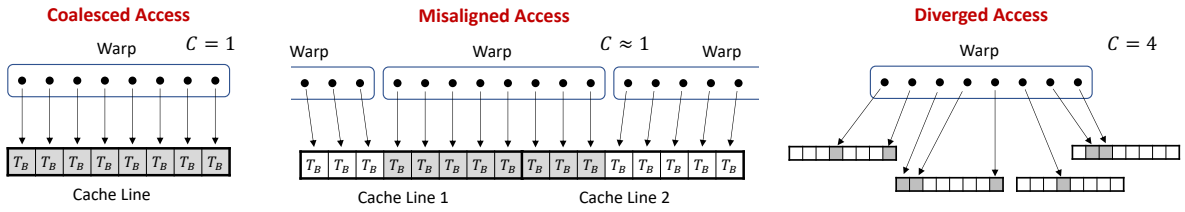


**Figure 4.** Various DRAM access patterns to model sparse memory access cost $C$: a) Coalesced access to $\boldsymbol{A}$ has no extra cost ($C = 1$) due to S-ELL representation, b) misaligned access cost to $\boldsymbol{C}$ is approximated ($C \approx 1$) as no extra cost , and c) diverged access cost to $\boldsymbol{B}$ is found via inspection of the sparsity pattern of $\boldsymbol{A}$

memory-bandwidth bound for tuning the segmented sum parameters in Fig. 6 and use the data reuse $\rho$ in Eq. 3 in the decision mechanism whether to keep the reordering (or discarding it) as explained in the rest of this section.

### 4.1 Load Balancing: Segmented Sum

We apply segmented sum [8] to improve load balancing as seen in Figure 5. Load balancing can occur because the Tiled SpMM is either:

- **Load Imbalanced**: A few threads are assigned with much more work than the average and therefore become the bottleneck in the massively-parallel execution. Typically, the sparse matrix is pathologically imbalanced. As a remedy, we apply the proposed load-balancing (segmented sum) algorithm that is explained in Section 4.1. In this case, we set the segment size to $\mu$.
- **Underutilized**: When the sparse matrix is extremely fat and short, i.e., $M \ll N$, the number of thread blocks does not utilize the 108 streaming multiprocessors (SMs) on the A100 GPUs. As a remedy, we use the segmented sum algorithm for scheduling extra threads and hence increasing the GPU utilization. In this case, we set the segment size to $U\mu$, where

$$U = \frac{\text{\# scheduled blocks}}{\text{\# SMs on GPU}} \tag{5}$$

is the GPU utilization.

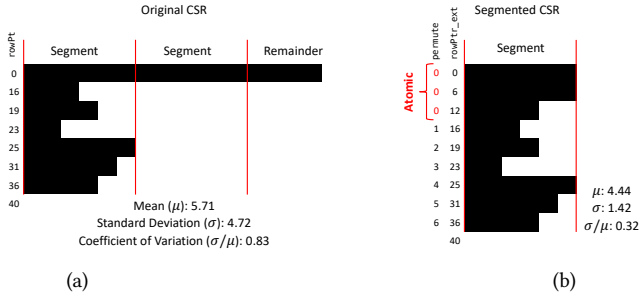


**Figure 5.** Memory layouts of CSR and segmented CSR data structures. The load balancing reduces standard deviation and schedules extra threads.

A decision flowchart is shown in Figure 6 is proposed to employ segmented sum for balancing pathological cases. (1) First, the matrix statistics are calculated. Those are the utilization $U$, the mean $\mu$ of the distribution of nonzeroes, and the standard deviation $\sigma$. of the nonzero distribution. (2) Then if the case is underutilized, we apply segmented sum with the segment size of $U\mu$. (3) If not underutilized but load imbalanced, we again apply segmented sum with the segment size of $\mu$. (4) If neither of the pathological cases holds, we skip load balancing and call the baseline TiledSpMM without load balancing for avoiding the nominal overhead of segmented sum. (5) Otherwise, we call the segmented sum kernel.

Note that the segmented sum alters the arithmetic intensity in Eq. 3 because of the following reasons: 1) The algorithm changes the sparsity pattern and hence the statistics (Figure 5) of the sparse matrix. Specifically, the segmented sum affects $\mu$, $Z$, and $\rho$ in an unpredictable way. 2) There is a nominal overhead of the proposed load balancing mechanism, and hence we update the effective intensity as

$$I_e^* = \frac{2}{\frac{(B_T + B_{IS})Z}{R} + \frac{B_T C}{\rho} + \frac{B_I}{\rho\mu^*} + \frac{B_I + \delta}{R\mu^*} + \frac{B_T}{\mu}}, \tag{6}$$

where $\mu^*$ is the number of nonzeroes per row after segmenting the long rows. We apply the modified effective intensity in Eq. 6.

### 4.2 Row Reordering: Radix Sort

Radix sort provides a cheap way to group rows with similar sparsity patterns into the same block. In radix sort, each row is mapped to a binary number, according to its sparsity pattern. In this case, zeroes (that are not stored) correspond to 0 and nonzeroes correspond to 1 in the $N$-bit binary number [22]. The idea is to sort these numbers with quicksort with $O(M \log M)$ comparisons, where $M$ is the number of rows.

Figure 7 shows an example. Each comparison of a pair of rows requires $O(\mu)$ element-wise comparisons, where $\mu$ is the number of nonzeroes per row on average. For each comparison, the nonzeroes in the rows must be sorted (in advance) in terms of their column indices, which adds $O(M\mu \log \mu)$

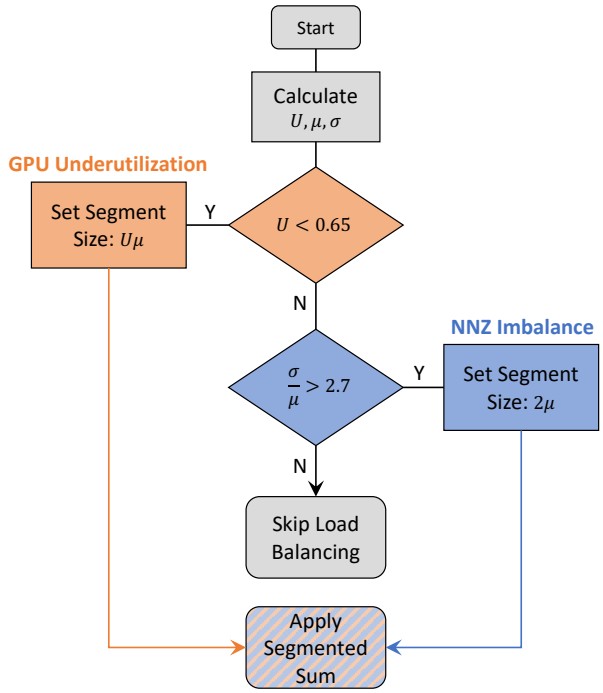


**Figure 6.** The proposed decision chart is used to pick the (1) load imbalanced and (2) underutilized SpMM cases. By applying the segmented sum kernel to those pathological cases, Tiled SpMM improves performance as further discussed in Section 5.2.1.

time to the row reordering algorithm, yielding an order of

$$M\mu(\log M + \log \mu) \tag{7}$$

time complexity. After sorting the rows, they are grouped into bins in a consecutive manner and then each bin is assigned to a thread block with the same number of rows per block.

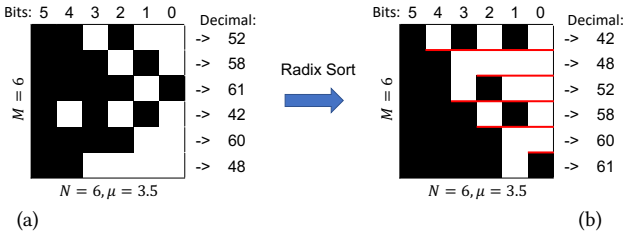


**Figure 7.** Radix sort treats each row as a binary number and sort them with reduced computational complexity.

In general, the row reordering problem can be posed as a clustering problem with a constraint on the cluster size. Finding an optimal solution is NP-hard, and therefore it must be solved sub-optimally with a heuristic algorithm, but with a lower computational complexity. The radix sort provides only a sub-optimal yet feasible solution to improve the data reuse ($\rho$ in Eq. 3). Nevertheless, since it is a greedy heuristic, the improvement is not guaranteed, and in fact, it can reduce $\rho$ and hence increase the total execution time. Section 5.3 shows that 55 cases out of the 188 cases defined in Section 5.1 benefit from row reordering with radix sort.

## 5 Benchmarking Results

The main objectives of our evaluation are 1) proving that the proposed model accurately predicts the kernel execution performance; 2) verifying that the proposed model leads to proper optimization decisions; 3) demonstrating that the optimized kernel, Tiled SpMM outperforms the highly optimized cuSPARSE library.

### 5.1 Experimental Setup

This section benchmarks the proposed performance model over a collection of sparse matrices from a diverse set of applications. All the experiments are performed on a single-precision Tiled SpMM running on Nvidia A100 GPU [1].

We pick 200 largest sparse matrices in terms of number of nonzeroes in the SuiteSparse Matrix Collection [10]. We exclude 16 of them because they either do not fit in 40 GB of HBM2 memory or they suffer from overflow. They require 64 bytes for indexing. As a result, we obtain an extensive benchmark dataset involving 188 sparse matrices from a diverse set of applications. These are included in the SuiteSparse website.[2] Our numerical results are reproducible because our benchmark dataset is open-source and we use the latest-generation of GPUs with vendor-provided HPC libraries (cuSPARSE[3]).

### 5.2 Effective Intensity Results

Figures 8 and 9 show the measured Tiled SpMM performance sorted by 1) the algorithmic intensity bound (see Section 3.0.1) and 2) the effective intensity bound (All Miss) (see Section 3.0.2), respectively. The red curve shows the tightest upper bound for measurement performance. That is, a measurement reaches to the red curve only if it utilizes the DRAM bandwidth 100%. If a measurement is higher than the red line, it means that must be some data reuse from L2 cache. The black line predicts the performance with the optimistic scenario (see Section 3.0.2). That is, once the sparse access cost is paid, the data stays in cache forever as if there is no trashing, i.e., infinite cache.

Out of 188 cases, 17% utilize less than half of the peak DRAM bandwidth. The picked cases with the decision algorithm Figure 10 are highlighted with blue and orange as explained in the figure caption. With the decision parameters, a few performant cases are picked up because the load variation is handled by the dynamic scheduling of thread blocks on SMs.

[2] h□ps://suitesparse-collection-website.herokuapp.com/about
[3] h□ps://docs.nvidia.com/cuda/cusparse/index.html

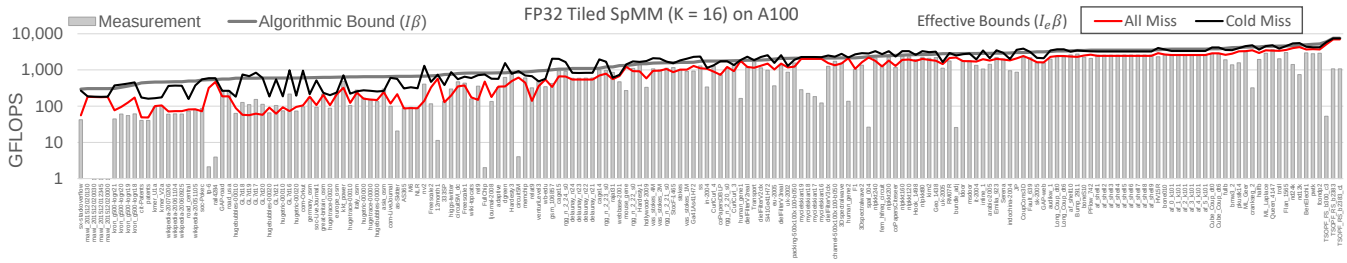


**Figure 8.** Benchmark results over 188 matrices from SuiteSparse Matrix Collection - sorted by the algorithmic intensity bound.

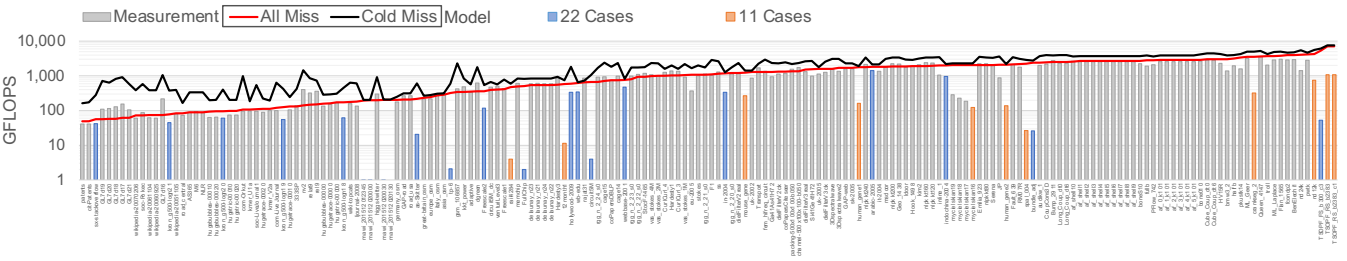


**Figure 9.** Load balancing algorithm selects 11 underutilized cases ($U$ Ÿ 0.65, highlighted with orange) and 33 imbalanced cases ($\sigma/\mu$ ¡ 1 - highlighted with blue) and applies segmented sum for load balancing. This plot shows measured performance before load balancing and sorted with respect to the effective intensity bound (All Miss).

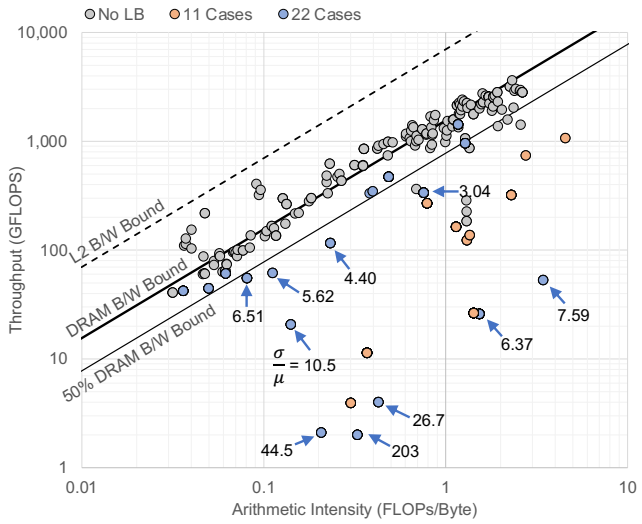


**Figure 10.** Roofline plot of 184 cases. 17% of these cases utilize less than 50% of the DRAM bandwidth due to load balancing. We pick them by the load balancing flowchart and apply load balancing with segmented sum.

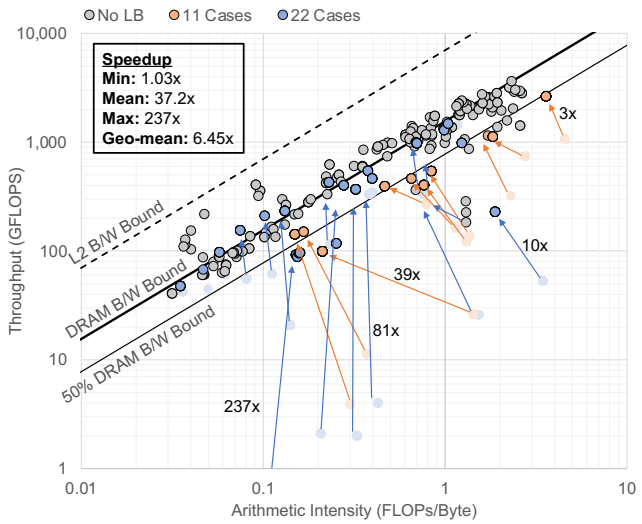


**Figure 11.** The proposed load balancing algorithm provides 14.8× speedup on average to the underperforming cases determined by the decision algorithm in Figure 6

**5.2.1 Effect of Load Balancing.** Figure 10 shows the benchmarking results in the roofline plot. The referecnces to the bandwidth bounds are included in the figure. In this figure, we model the arithmetic intensity with the worst case scenario explained in Section 3.0.2. This means that the red curve in Figure 9 is mapped to the DRAM bandwidth bound, and if a case performs higher, then it must have some L2 cache reuse (according to the model).

The selected cases using the decision flowchart in Figure 6 are shown with respective colors in the roofline plot in Figure 1. The coefficient of variation (CV: $\sigma/\mu$) and the utilization ($U$) of the underperforming cases are annotated in the figure with respective blue and orange colors. When we apply the load balancing algorithm, those cases move closer

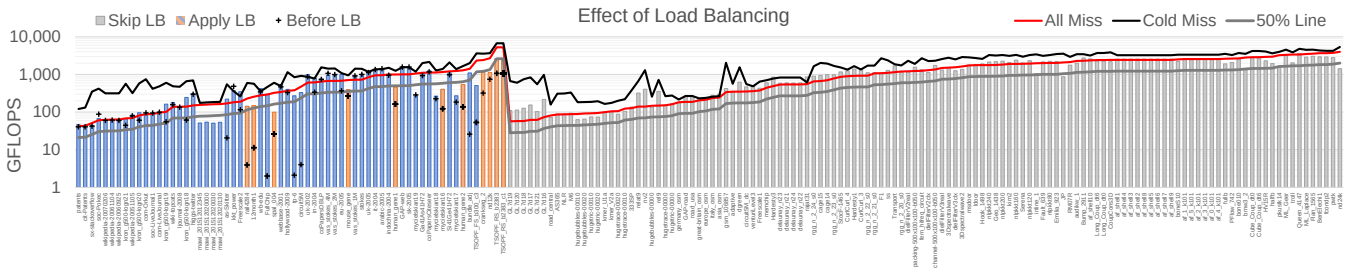


**Figure 12.** The proposed load balancing provides 37.2× speedup on average to the cases highlighted with blue (nnz imbalanced) and orange (GPU underutilization) cases. Results are sorted by the effective intensity (All Miss) bound.

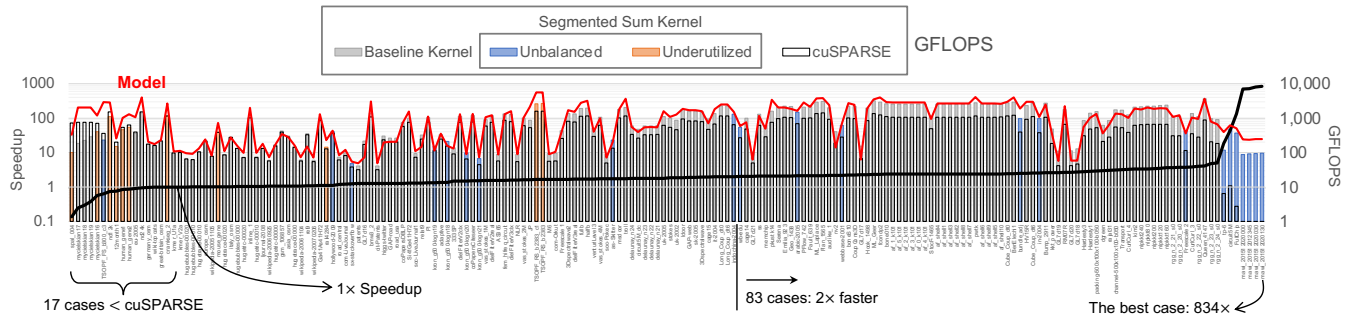


**Figure 13.** With the load balancing, Tiled SpMM is 12 times faster than cuSPARSE on average of all cases. 83 of the cases are 2× faster and 17 cases are slower than cuSPARSE.

**Table 2.** Speedup of Tiled SpMM over cuSPARSE

| | Before LB | After LB |
|---|---|---|
| **Minimum** | 0.03× | 0.14× |
| **Average** | 1.78× | 19.2× |
| **Maximum** | 4.96× | 834× |
| **Geo-Mean** | 0.75× | 2.03× |

to the DRAM bandwidth bound as shown in Figure 11. The speedup statistics are included in the figure. Figure 12 sorts the balanced cases together. For reference, the throughputs before the balancing are marked with cross signs.

**5.2.2 Comparison with cuSPARSE.** Finally, we benchmark the proposed Tiled SpMM algorithm against cuSPARSE (release 11.4) with column-major layout. The cuSPARSE results are shown with empty bars and the optimized Tiled SpMM results are shown with solid bars in Figure 13. The speedup of Tiled SpMM over the whole benchmark dataset (188 matrices) is summarized in Table 2. It is clearly seen that the proposed load balancing algorithm explained in Section 4.1 improves the average and geometric mean of the speedup of Tiled SpMM over cuSPARSE.

### 5.3 Effect of Row Reordering

This subsection presents the effect of the proposed row-reordering algorithm with radix sort in Section 4.2. It is worth pointing out two implementation details. First, we apply row reordering after the load balancing mechanism described in Section 4.1. In this way, we can recover some data reuse which is destroyed by the segmented sum algorithm. Second, we neglect the permutation overhead when writing the output. That is, when rows of the sparse matrix are permuted, each thread writes the output to the permuted location, i.e., not to the original location. Our experiments show that obtaining the original ordering of the output matrix undoes the benefit of the row reordering because of the diverged memory access cost depicted in Figure 4. Therefore, sticking to the permutation and preserving coalesced write accesses is recommended if the application allows.

Figure 14 shows a few cases where row reordering provides significant speedup. The performance models (before and after reordering) agree with the obtained speedup to a good extent. Overall, 55 of the 188 cases in our benchmark dataset described in Section 5.1 show improvement. As a result, we can assess that the proposed row reordering algorithm provides improvement in 29% of the cases. The improvement on the performance agrees with our model, and we use the model on decision to whether keeping the reordering or not.

## 6 Related Work

SpMM and SpMV are a fundamental algorithm used in many scientific and engineering computations such as artificial intelligence, data analytics, and computational science. While

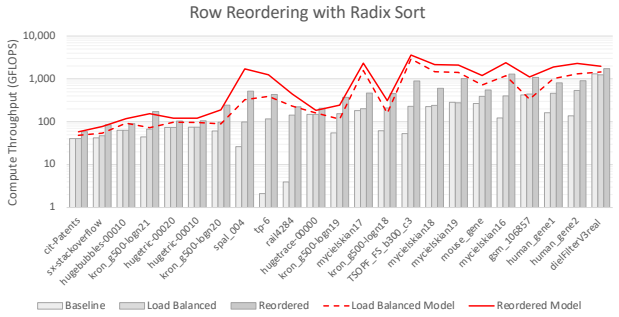


**Figure 14.** Row reordering with radix sort yields improvement in 55 (29%) of the 188 application cases. This figure shows a few examples.

SpMV has been extensively studied on GPUs [5–7, 13, 24, 26–28, 30, 34, 36, 39, 41], there have been a few studies so far that focus on SpMM [11, 17, 18, 32, 40], but they focus on row-major layout of dense matrices.

Hong et al. [17, 18] proposed two approaches to improve data locality and load balancing in SpMM. In the first work [17], the sparse matrix is partitioned into two matrices; the first matrix holds nonzeroes of heavily clustered row-segments and the other holds randomly scattered row segments. Tiling is used for the heavy-rows matrix to exploit data locality. In the second work [18], an adaptive-tiling technique is proposed to improve data locality for SpMM.

Jiang et al. [21] proposed a total row reordering procedure to group similar rows together in the sparse matrix. That is, rows with nonzeroes at similar column locations are grouped in the same panel. Then, the adaptive tiling-tiling technique [18] is used on the row-reordered matrix.

CSR-based formats are proposed to target specific applications without sparse matrix transformation. GE-SpMM applies optimizations to the standard CSR format to enable integration in graph neural networks (GNNs) frameworks without format transformation overhead [20]. These optimizations are loading sparse matrix rows into the shared memory to enable data reuse and thread coarsening in which a thread produces multiple partial results. Gale et al. [11] proposed a hierarchical tiling SpMM approach to improve data reuse for deep learning applications. Both CSR-based approaches do not exploit the data reuse opportunity by tiling the dense matrix $\boldsymbol{B}$.

Performance modeling based on arithmetic intensity has been studied on CPU processors [12]. Even though the proposed model in previous work agrees with our baseline model in Eq. 2, it does not include the caching effects as we do in this chapter. An important related work [3] makes a point of distinguishing two types of parallel efficiency: algorithmic efficiency and implementation efficiency. This work takes a similar approach and separates algorithmic intensity (based on algorithm-only) from effective intensity (based on algorithm + architecture) through the sparse memory access cost $\Gamma$ in Eq. 3.

Model-driven optimization approaches on GPUs have been studied in the sparse matrix - dense vector multiplication (SpMV) context [9]. In this work, we propose the model for the Tiled SpMM algorithm with the proposed register and scratchpad tiling strategies. The specific strategy we propose in this thesis is shown to be an optimal one through a space exploration [38].

A similar tiling strategy has been implemented for 3D X-ray image reconstruction top obtain up to 65 mixed-precision PFLOPS sustained performance on Summit supercomputer. The roofline model is documented in more detail in previous work [37]. The segmented sum algorithm has been studied in more detail for vector multiprocessors [8]. Kirk and Hwu [22] provide a more detailed description of radix sort and its implementation on GPUs. In this work, the preprocessing of the sparse matrix is performed on CPU and the tiling data structures are moved to GPU for accelerated execution.

## 7 Conclusion and Future Work

In conclusion, we develop an effective intensity model for Tiled SpMM to understand and optimize SpMM on GPU architecture. The variables in the proposed intensity model reliably capture the performance implications depending on the data (sparse sparsity pattern) and architecture (cache behavior). We benchmark the proposed model over 188 open-source sparse matrices sampled from a wide spectrum of applications. We use the estimated performance (with the model) to detect underperforming cases of load imbalanced and underutilized cases. We implement segmented sum as a load balancing mechanism for Tiled SpMM to improve the performance by 19.2× on average of the selected cases. As a result of the improvement, we obtain 12× average speedup and 2.03× geometric speedup over the vendor provided HPC library. Moreover, we open-source our reproducible application code for the benefit of further analysis and improvement.

Future work includes extending the performance model for the row-major storage of dense matrices. In that case, implementation and optimizations of Tiled SpMM is quite different than in this paper because the sparse memory access patterns to $\boldsymbol{B}$ and hence the caching behaviour changes. We can use the proposed existing model by updating the sparse access cost $\Gamma$ in Eq. 3. We also propose to extend this work with column reordering of the sparse matrix. Because, with a permutation of rows and columns of the sparse matrix, we can alter the sparsity pattern of $\boldsymbol{A}$ in our advantage. We can use the proposed intensity model (given in Eq. 3) as a guide for the performance implications and to predict the effect of row and column permutation as follows: (1) row permutation should improve the data reuse $\rho$ and (2) column permutation should reduce the sparse access cost $\Gamma$.

## References


[1] [n.d.]. Nvidia A100 GPUs power the modern data center. https://www.nvidia.com/en-us/data-center/a100/

[2] [n.d.]. NVIDIA cuSPARSE Library. http://docs.nvidia.com/cuda/cusparse/index.html

[3] W Kyle Anderson, William D Gropp, Dinesh K Kaushik, David E Keyes, and Barry F Smith. 1999. Achieving high sustained performance in an unstructured mesh CFD application. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. 69–es.

[4] Saurabh Animesh. 2018. *Algorithm Architecture Co-Design for Dense and Sparse Matrix Computations*. Ph.D. Dissertation. Arizona State University.

[5] Hartwig Anzt, Stanimire Tomov, and Jack Dongarra. 2014. Implementing a Sparse Matrix Vector Product for the SELL-C/SELL-C-$\sigma$ formats on NVIDIA GPUs. *University of Tennessee, Tech. Rep. ut-eecs-14-727* (2014).

[6] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarath, and P Sadayappan. 2014. Fast sparse matrix-vector multiplication on GPUs for graph applications. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 781–792.

[7] Nathan Bell and Michael Garland. 2009. Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (Portland, Oregon) *(SC '09)*. Association for Computing Machinery, New York, NY, USA, Article 18, 11 pages. https://doi.org/10.1145/1654059.1654078

[8] Guy E Blelloch, Michael A Heroux, and Marco Zagha. 1993. *Segmented operations for sparse matrix computation on vector multiprocessors*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE.

[9] Jee W Choi, Amik Singh, and Richard W Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *ACM sigplan notices* 45, 5 (2010), 115–126.

[10] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[11] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. Sparse GPU kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[12] William D Gropp, Dinesh K Kaushik, David E Keyes, and Barry F Smith. 1999. Toward realistic performance bounds for implicit CFD codes. In *Proceedings of parallel CFD*, Vol. 99. Citeseer, 233–240.

[13] Dahai Guo and William Gropp. 2012. Adaptive thread distributions for SpMV on a GPU. In *Proceedings of the extreme scaling workshop*. 1–5.

[14] Mert Hidayetoglu, Tekin Biçer, Simon Garcia De Gonzalo, Bin Ren, Vincend De Andrade, Doğa Gürsoy, Rajkumar Kettimuthu, Ian T Foster, and Wen-mei W Hwu. 2020. Petascale XCT: 3D image reconstruction with hierarchical communications on multi-GPU nodes. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[15] Mert Hidayetoğlu, Tekin Biçer, Simon Garcia De Gonzalo, Bin Ren, Doğa Gürsoy, Rajkumar Kettimuthu, Ian T Foster, and Wen-mei W Hwu. 2019. Memxct: Memory-centric X-ray CT reconstruction with massive parallelization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*.

[16] Mert Hidayetoglu, Carl Pearson, Vikram Sharma Mailthody, Eiman Ebrahimi, Jinjun Xiong, Rakesh Nagi, and Wen-Mei Hwu. 2020. At-Scale Sparse Deep Neural Network Inference With Efficient GPU Implementation. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. https://doi.org/10.1109/hpec43674.2020.9286206

[17] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V Çatalyürek, Srinivasan Parthasarathy, and P Sadayappan. 2018. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 66–79.

[18] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. 2019. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 300–314.

[19] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Gespmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[20] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC'20)*. IEEE Press, Article 72, 12 pages.

[21] Peng Jiang, Changwan Hong, and Gagan Agrawal. 2020. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 376–388.

[22] David B Kirk and Hwu Wen-Mei. 2016. *Programming massively parallel processors: A hands-on approach*. Morgan Kaufmann.

[23] Sireyya Emre Kurt, Vineeth Thumma, Changwan Hong, Aravind Sukumaran-Rajam, and P Sadayappan. 2017. Characterization of data movement requirements for sparse matrix computations on gpus. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. IEEE, 283–293.

[24] Kenli Li, Wangdong Yang, and Keqin Li. 2014. Performance analysis and optimization for SpMV on GPU using probabilistic modeling. *IEEE Transactions on Parallel and Distributed Systems* 26, 1 (2014), 196–205.

[25] Mohammed Mahmoud, Mark Hoffmann, and Hassan Reza. 2018. Developing a new storage format and a warp-based SpMV kernel for configuration interaction sparse matrices on the GPU. *Computation* 6, 3 (2018), 45.

[26] Duane Merrill and Michael Garland. 2016. Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format. *ACM SIGPLAN Notices* 51, 8 (2016), 1–2.

[27] Thaha Muhammed, Rashid Mehmood, Aiiad Albeshri, and Iyad Katib. 2019. SURAA: A novel method and tool for loadbalanced and coalesced SpMV computations on GPUs. *Applied Sciences* 9, 5 (2019), 947.

[28] B Neelima, G Ram Mohana Reddy, and Prakash S Raghavendra. 2014. Predicting an optimal sparse matrix format for SpMV computation on GPU. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 1427–1436.

[29] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

[30] Yuyao Niu, Zhengyang Lu, Meichen Dong, Zhou Jin, Weifeng Liu, and Guangming Tan. 2021. TileSpMV: A Tiled Algorithm for Sparse Matrix-Vector Multiplication on GPUs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 68–78.

[31] NVIDIA. 2008. CUDA Programming guide.

[32] Gloria Ortega, Francisco Vázquez, Inmaculada García, and Ester M Garzón. 2014. Fastspmm: An efficient library for sparse matrix matrix product on GPUs. *Comput. J.* 57, 7 (2014), 968–979.

[33] Markus Steinberger, Andreas Derlery, Rhaleb Zayer, and Hans-Peter Seidel. 2016. How naive is naive SpMV on the GPU?. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

[34] Markus Steinberger, Rhaleb Zayer, and Hans-Peter Seidel. 2017. Globally homogeneous, locally adaptive sparse matrix-vector multiplication on the GPU. In *Proceedings of the International Conference on Supercomputing*. 1–11.

[35] Guangming Tan, Linchuan Li, Sean Triechle, Everett Phillips, Yungang Bao, and Ninghui Sun. 2011. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.

[36] Francisco Vazquez, G Ortega, José-Jesús Fernández, and Ester M Garzón. 2010. Improving the performance of the sparse matrix vector product with GPUs. In *2010 10th IEEE International Conference on Computer and Information Technology*. IEEE, 1146–1151.

[37] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[38] Jie Xin, Xianqi Ye, Long Zheng, Qinggang Wang, Yu Huang, Pengcheng Yao, Linchen Yu, Xiaofei Liao, and Hai Jin. 2021. Fast Sparse Deep Neural Network Inference with Flexible SpMM Optimization Space Exploration. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[39] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou. 2014. yaSpMV: Yet another SpMV framework on GPUs. *Acm Sigplan Notices* 49, 8 (2014), 107–118.

[40] Carl Yang, Aydın Buluç, and John D Owens. 2018. Design principles for sparse matrix multiplication on the GPU. In *European Conference on Parallel Processing*. Springer, 672–687.

[41] Wangdong Yang, Kenli Li, and Keqin Li. 2018. A parallel computing method using blocked format with optimal partitioning for SpMV on GPU. *J. Comput. System Sci.* 92 (2018), 152–170.

[42] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. 2021. Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 687–701.

[43] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.