Minuet: Accelerating 3D Sparse Convolutions on GPUs

Jiacheng Yang
University of Toronto &
Vector Institute

Christina Giannoula
University of Toronto

Jun Wu
Amazon

Mostafa Elhoushi
Meta

James Gleeson
Samsung AI Centre Toronto

Gennady Pekhimenko
CentML & University of Toronto &
Vector Institute

Abstract

Sparse Convolution (*SC*) is widely used for processing 3D point clouds that are inherently sparse. Different from dense convolution, *SC* preserves the sparsity of the input point cloud by only allowing outputs to specific locations. To efficiently compute *SC*, prior *SC* engines first use hash tables to build a kernel map that stores the necessary General Matrix Multiplication (GEMM) operations to be executed (*Map* step), and then use a Gather-GEMM-Scatter process to execute these GEMM operations (*GMaS* step). In this work, we analyze the shortcomings of prior state-of-the-art *SC* engines, and propose Minuet, a novel memory-efficient *SC* engine tailored for modern GPUs. Minuet proposes to (i) replace the hash tables used in the *Map* step with a novel segmented sorting double-traversed binary search algorithm that highly utilizes the on-chip memory hierarchy of GPUs, (ii) use a lightweight scheme to autotune the tile size in the Gather and Scatter operations of the *GMaS* step, such that to adapt the execution to the particular characteristics of each *SC* layer, dataset, and GPU architecture, and (iii) employ a padding-efficient GEMM grouping approach that reduces both memory padding and kernel launching overheads. Our evaluations show that Minuet significantly outperforms prior *SC* engines by on average 1.74× (up to 2.22×) for end-to-end point cloud network executions. Our novel segmented sorting double-traversed binary search algorithm achieves superior speedups by 15.8× on average (up to 26.8×) over prior *SC* engines in the *Map* step. The source code of Minuet is publicly available at https://github.com/UofT-EcoSystem/Minuet.

CCS Concepts: • **Computing methodologies** → **Parallel algorithms**; *Computer vision*; *Machine learning algorithms*.

EuroSys '24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3629560

Keywords: 3D Point Cloud, Sparse Convolution, GPUs

ACM Reference Format:
Jiacheng Yang, Christina Giannoula, Jun Wu, Mostafa Elhoushi, James Gleeson, and Gennady Pekhimenko. 2024. Minuet: Accelerating 3D Sparse Convolutions on GPUs. In *European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3627703.3629560

1 Introduction

Thanks to recent advances in 3D sensors, such as light detection and ranging (*LiDAR*) sensors, 3D point clouds become increasingly accessible and widely used in many important applications, including virtual and augmented reality (*VR/AR*) [47], photography [16], drones [52], robotics [27], and autonomous vehicles [4, 51]. Similarly to popular deep neural networks (*DNN*s), point cloud networks provide high efficiency and accuracy on a variety of vision tasks, such as 3D object detection [33, 38, 42] and segmentation [23, 35, 44].

Different from 2D dense images, 3D point clouds describe 3D objects that are extremely sparse to their bounding space (usually less than 0.01% [30]). Therefore, researchers propose dedicated DNN-based algorithms [7, 8, 26, 31, 32, 34, 39, 40] to efficiently process 3D point clouds by taking into consideration the sparse execution pattern. Among these algorithms, Sparse Convolution (*SC*) networks [8, 18] achieve high accuracy, dominating performance, and wide applicability. As shown in Figure 1, unlike dense convolution where the sparsity is quickly diluted, *SC* only allows the set of output points to specific locations that preserve the sparsity pattern exhibited in the input point cloud. Thus, to reduce the number of computations, for each output point, *SC* needs to find the locations of the corresponding input point and the weight, which results in *implicit* General Matrix Multiplications (GEMMs) [30, 43], i.e., the exact input feature and weight for each GEMM are implied by the sparsity pattern of the input point cloud.

To efficiently execute implicit GEMMs, prior works break the *SC* execution into two steps: (1) the mapping step (***Map***); and (2) the Gather-GEMM-Scatter step (***GMaS***). In the *Map* step, *SC* builds a *kernel map* that stores the necessary GEMM operations needed to be performed, i.e., the indices of the weights and the input/output feature vectors. In the *GMaS* step, *SC* executes each necessary GEMM operation in the kernel map to transform the input feature vectors into the

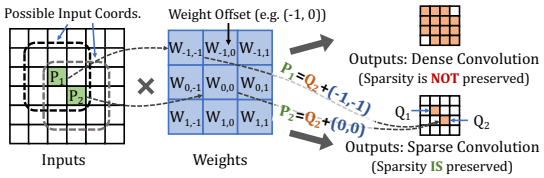

Figure 1. Dense convolution versus sparse convolution.

output feature vectors. To build the kernel map in the *Map* step, prior *SC* engines [9, 30, 43] create *queries* with all possible input coordinates by enumerating the additions of each output coordinate with each weight offset. Then, they check the existence of non-zero input data points by executing the queries to a *hash table*, that stores the coordinates of the actual non-zero input data points. In the *GMaS* step, prior *SC* engines [43] use an input buffer array and an output buffer array to *continuously* store the operands of the GEMM operations, i.e., the values of the input and the output feature vectors. This approach enables prior *SC* engines to leverage highly-performant GPU GEMM libraries (e.g., cuBLAS [12]) that require GEMM operands to be continuous in memory. To do so, they first broadcast the input feature vectors to the input buffer array with a *Gather* operation, then execute GEMM operations to create partial results for output feature vectors which are stored in the output buffer array, and finally merge (sum-reduce) the partial results to assemble the final output feature vectors with a *Scatter* operation.

In this work, we characterize existing *SC* engines [8, 9, 43] using various point cloud networks, real datasets, and GPU architectures, and find that they suffer from three key shortcomings. First, they use a hash table (e.g., cuckoo hash table [1]) to build the kernel map, which stores the necessary GEMM operations. However, executing a large number of *queries* in the hash table incurs irregular data accesses, most of which are served by the expensive GPU global memory, thus causing high data access costs. Second, state-of-the-art *SC* engines process multiple input/output feature channels of *SC* in *tiles*, as a chunk of consecutive feature channels, in Gather and Scatter operations to improve GPU memory throughput. However, we observe that they always employ a single *fixed* tile size, which suffers from sub-optimal performance. In Figure 4, we demonstrate that the best-performing tile size depends on the characteristics of each particular *SC* layer of the point cloud network, the real dataset, and the GPU architecture. Third, in the *GMaS* step, prior *SC* engines execute GEMM operations corresponding to multiple weight offsets in a *batched* scheme: they group multiple GEMM operations together by padding with zero values the GEMM operands, i.e., they provide the same sizes among all GEMM operands and execute them as a single batched GEMM kernel. This way they minimize GPU kernel launch overheads and improve GPU hardware utilization [43]. However, we find that prior *SC* engines group GEMM operations in the *GMaS* step following the order induced by the *Map* step, i.e., the order of the weight offsets. This approach causes high padding overheads (Section 3), i.e., a large number of zero values are added, thus incurring many redundant data accesses and computations.

To tackle the aforementioned issues, we propose Minuet, a novel memory-efficient *SC* engine tailored for modern GPUs. Minuet highly utilizes the on-chip memory hierarchy of GPUs, adapts *SC* execution to the characteristics of the input dataset and GPU architecture, and reduces unnecessary data accesses and computations. In the *Map* step, we challenge the prevailing notion that the hash table-based search performs superiorly compared to binary search on GPUs [1, 2], and propose an innovative binary search-based algorithm tailored for building the kernel maps in the *Map* step of *SC* on modern GPU architectures. We leverage the key observation that when executing *sorted* queries, binary search achieves high system efficiency, leveraging data locality across consecutive *sorted* queries, and propose the *segmented sorting double-traversed* binary search algorithm. Our proposed algorithm for *SC* can achieve a similar theoretical computational complexity with the hash table-based search (Section 5.1.3) and provides significantly higher memory efficiency, improving the hit ratio in the on-chip caches of GPUs (Figure 16b). In the *GMaS* execution step, Minuet provides two optimizations. First, we on-the-fly tune the tile size used to process multiple input/output feature channels at *each* Gather and Scatter operations. This key technique enables Minuet to adapt the *SC* execution to the particular characteristics of each *SC* layer in point cloud networks, real dataset, and GPU architecture, thus providing high system performance in Gather and Scatter operations. Second, Minuet integrates a padding-efficient GEMM grouping strategy, which first reorders GEMM operations based on the sizes of input/output feature vectors, and then groups GEMM operations into batched GEMM kernel launches. This way Minuet optimizes both (i) the amount of padding with zero values, thus minimizing unnecessary data accesses and computations to useless data in GEMM kernels, and (ii) the GEMM kernel launch overheads.

We extensively evaluate Minuet using a wide variety of 3D point cloud networks, real datasets, and GPU architectures, and demonstrate that Minuet significantly outperforms prior works. Compared to state-of-the-art *SC* engines, Minuet improves the end-to-end performance by 1.74× on average (up to 2.22×), and achieves superior speedups over prior *SC* engines in the *Map* step by on average 15.8× (up to 26.8×), and by on average 1.39× (up to 2.38×) in the *GMaS* step.

Overall, this paper makes the following contributions:

- We investigate the shortcomings of existing *SC* engines, and propose Minuet, a memory-efficient engine to accelerate *SC* executions on modern GPU architectures.
- We propose a novel segmented sorting double-traversed binary search algorithm to build kernel maps in *SC*. Our proposed algorithm highly utilizes the on-chip memory

hierarchy of GPUs. We also dynamically select the best-performing tile size in Gather and Scatter operations, and reorder GEMM operations before grouping them to minimize unnecessary data accesses and computations.
- We evaluate Minuet using a wide range of real datasets, sparse 3D networks, and GPU architectures, and show that it significantly outperforms prior works both in layerwise and end-to-end execution. Minuet also provides superior speedups in the *Map* step of *SC*. We open-source Minuet at https://github.com/UofT-EcoSystem/Minuet.

2 Sparse Convolution (*SC*)

2.1 *SC* Definition

A 3D object is inherently sparse in nature, i.e., it does not completely fill the 3D space it occupies, thus resulting in a spatially sparse structure. Point cloud is a widely applied sparse format that is used to effectively represent a 3D object, thanks to its simplicity and accuracy [16, 21, 27, 39, 47, 51, 52]. Specifically, a point cloud only stores the non-zero points of a 3D object as an unordered set of points $\mathcal{P} = \{\mathbf{p_i}\}$ and its corresponding set of feature vectors $\{\mathbf{F}_\mathbf{i}^\mathcal{P}\}$. Each point $\mathbf{p_i}$ is a 3D coordinate that represents one non-zero point of the 3D object, and each feature vector $\mathbf{F}_\mathbf{i}^\mathcal{P}$ of size C stores the corresponding C feature channels (e.g., $C = 3$ for RGB colors) of the i-th point $\mathbf{p_i}$. Thus, the feature vectors of a point cloud with N points can be stored in an $N \times C$ feature matrix $\mathbf{F}^\mathcal{P}$.

Sparse Convolution (*SC*) takes as input a 3D object represented as a point cloud, and its output is also a point cloud that preserves the sparsity pattern of the input point cloud. This is achieved by only allowing the computations of output feature vectors on specific output coordinates that are generated based on the input coordinates. More formally, *SC* uses the following formula to generate the output coordinates $\mathcal{Q}$ based on the input coordinates $\mathcal{P}$ with a stride parameter s:

$$\mathcal{Q} = \left\{ \left(\left\lfloor \frac{x}{s} \right\rfloor \times s, \left\lfloor \frac{y}{s} \right\rfloor \times s, \left\lfloor \frac{z}{s} \right\rfloor \times s \right) | (x, y, z) \in \mathcal{P} \right\} \quad (1)$$

To keep output coordinates unique, duplicates among the output coordinates are eliminated. Intuitively, the output coordinates $\mathcal{Q}$ are downsampled from the input coordinates $\mathcal{P}$ and the stride parameter s specifies the granularity of the downsampling. Note that if the stride s is equal to 1, the output coordinates will be the same as the input coordinates, i.e., $\mathcal{Q} = \mathcal{P}$. Then, the output feature vector $\mathbf{F}_\mathbf{i}^\mathcal{Q}$ of the i-th output coordinate $\mathbf{q_i}$ is computed on every weight offset $\boldsymbol{\delta_k}$ and every input coordinate $\mathbf{p_j}$, when the condition $\mathbf{p_j} = \mathbf{q_i} + \boldsymbol{\delta_k}$ holds, which is formalized as follows:

$$\mathbf{F}_\mathbf{i}^\mathcal{Q} = \sum_{\boldsymbol{\delta_k} \in \Delta(K,s)} \sum_{p_j \in \mathcal{P}} \mathbb{1}_{\mathbf{p_j} = \mathbf{q_i} + \boldsymbol{\delta_k}} \mathbf{F}_\mathbf{j}^\mathcal{P} \mathbf{W}_{\boldsymbol{\delta_k}} \quad (\mathbf{q_i} \in \mathcal{Q}) \quad (2)$$

where $\Delta(K, s)$ stands for the set of *weight offsets* with a kernel size K and a stride s (e.g., $\Delta(5, 2) = \{-4, -2, 0, 2, 4\}^3$), $\boldsymbol{\delta_k}$ for the k-th weight offset (e.g. $\boldsymbol{\delta_1} = (-4, -4, -4) \in \Delta(5, 2)$), $\mathbf{F}_\mathbf{j}^\mathcal{P}$ for the feature vector of the input coordinate $\mathbf{p_j}$, $\mathbf{W}_{\boldsymbol{\delta_k}}$ for the weight corresponding to the weight offset $\boldsymbol{\delta_k}$, and $\mathbb{1}_{\mathbf{p_j} = \mathbf{q_i} + \boldsymbol{\delta_k}}$ for the indicator function on the condition $\mathbf{p_j} = \mathbf{q_i} + \boldsymbol{\delta_k}$.

2.2 Execution Steps of *SC*

To effectively perform *SC* execution, existing *SC* frameworks construct an input-output map $\mathcal{M} = \{(\mathbf{p_j}, \mathbf{q_i}, \boldsymbol{\delta_k})\}$, named as **kernel map**. Each kernel map entry represents a General Matrix Multiplication (GEMM) operation in Equation 2:

$$\forall (\mathbf{p_j}, \mathbf{q_i}, \boldsymbol{\delta_k}) \in \mathcal{M} \quad \mathbf{F}_\mathbf{i}^\mathcal{Q} \leftarrow \mathbf{F}_\mathbf{i}^\mathcal{Q} + \mathbf{F}_\mathbf{j}^\mathcal{P} \mathbf{W}_{\boldsymbol{\delta_k}} \quad (3)$$

By traversing the kernel map, they perform only the necessary GEMM operations to compute the output feature vector for each output coordinates.

Figure 2 describes the *SC* execution, which can be broken down into two steps.

Step 1: Mapping Step (*Map*). To build the kernel map, existing implementations [8, 9, 43] first create a hash table ❶ where the input coordinates $\mathbf{p_j}$ are the keys and their indices j are the values. Then, they generate the output coordinates $\mathcal{Q}$ ❷ according to Equation 1, and create *queries* $\mathbf{q_i} + \boldsymbol{\delta_k}$ for each output coordinate $\mathbf{q_i}$ and each weight offset $\boldsymbol{\delta_k}$ as the candidate input coordinates. Next, they perform lookup in the hash table ❸ to check if each candidate input coordinate exists as an input coordinate $\mathbf{p_j}$. If such input coordinate $\mathbf{p_j}$ is found in the hash table, i.e., $\mathbf{p_j} = \mathbf{q_i} + \boldsymbol{\delta_k}$, then a new entry $(\mathbf{p_j}, \mathbf{q_i}, \boldsymbol{\delta_k})$ ❹ is added to the kernel map, which corresponds to a necessary GEMM operation that needs to be performed to get the output feature vectors (Equation 3).

Step 2: Gather-GEMM-Scatter Step (*GMaS*). Executing the GEMM operation for *each* entry in the kernel map results in a large number of small GEMM kernels, which incurs *immense* kernel launch overheads in GPUs. Thus, existing frameworks [9, 43] perform all GEMM operations associated with each weight with a *GMaS* step (Figure 2 right).

Specifically, for each weight, existing *SC* engines ***Gather*** the corresponding input feature vectors and store them consecutively to an input buffer array. To do so, they build a metadata table ❺ which stores the positions that each input feature vector needs to be stored within the input buffer array. For example, in Figure 2, the feature vector corresponding to the input coordinate $\mathbf{p_3}$ (i.e., $\mathbf{F}_3^\mathcal{P}$) is associated with 4 entries in the kernel map, corresponding to the 4 highlighted lines in the input metadata table. The input feature vector $\mathbf{F}_3^\mathcal{P}$ will be then copied to the corresponding positions in the input buffer array ❻, i.e., the 6-th, 9-th, 13-th, and 15-th entries of the input buffer array. Note that each input feature vector has C_{in} feature channels ❼. Thus, to maximize the GPU memory throughput, state-of-the-art work [43] processes the feature channels of each input feature vector *in tiles* ❽, where each tile contains a *fixed* size (typically 128 bytes) of consecutive feature channels. Specifically, each GPU thread loads one tile of the input feature vector to the on-chip register files, then accesses the input metadata table to find the corresponding buffer index of the input buffer array for *each* tile, and finally copies the tile to the input buffer array.

Then, *SC* implementations perform one **GEMM** operation for each weight to create partial results for the output

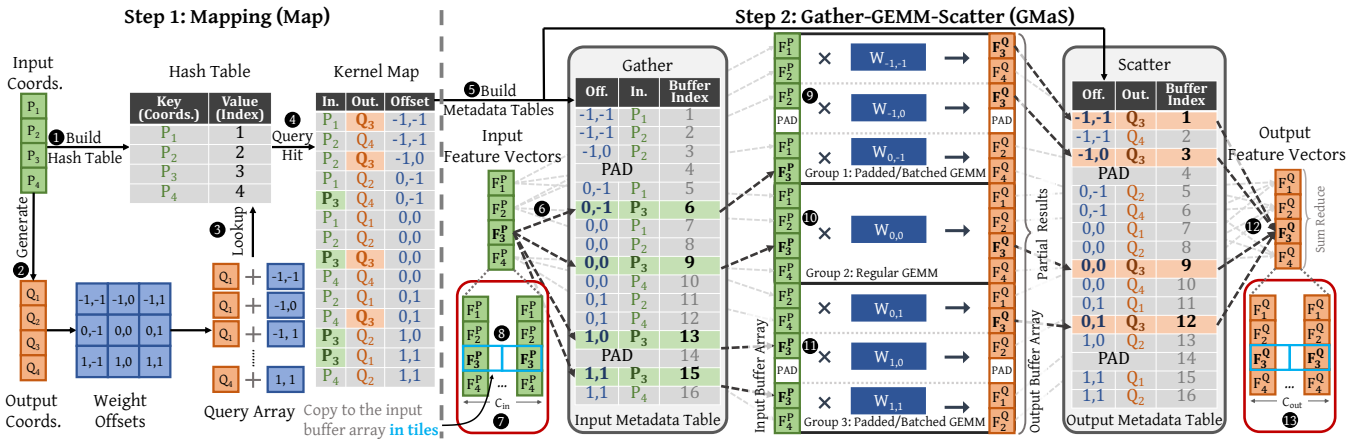

Figure 2. The *SC* execution can be broken down into two steps. For simplicity, we use 2D coordinates for illustration.

feature vectors. State-of-the-art *SC* engines [43] group multiple GEMM operations corresponding to multiple weights, such that to be performed as a single batched GEMM kernel to minimize kernel launch overheads and leverage existing GPU GEMM libraries [12]. In the example of Figure 2, the GEMM operations are merged in 3 GEMM groups ❾, ❿, and ⓫, and executed as 3 GEMM kernel launches. To do so, padding with zero values is needed, and thus the buffer indices stored in the input metadata table of Gather operation are updated, accordingly. For instance, to perform the batched GEMM kernel launch of Group 1 in Figure 2 ❾, *SC* implementations pad with zero value feature channels in the buffer index 4 of the input buffer array, such that the associated feature vectors of weight offset $(-1, 0)$ have the same sizes with that of the other two weight offsets of the same GEMM group, i.e., the weight offsets $(-1, -1)$ and $(0, -1)$.

Finally, the partial results produced by the batched GEMM operations in the output buffer array are aggregated and reduced with a **Scatter** operation ⓬ to obtain the output feature vectors. Similar to the Gather operation, in the Scatter operation, (i) an output metadata table is built to store the positions of the partial results in the output buffer array for each output feature vector, where the final values of the output feature vector is produced by merging these partial results. (ii) padding is added in the indices stored in the output metadata table due to GEMM grouping on batched GEMM operations, and (iii) the feature channels of each output feature vector are processed *in tiles* of the same sizes (typically 128 bytes) ⓭: each GPU thread loads one *tile* of partial results in the output buffer array to the on-chip register files, then accesses the output metadata table to find the corresponding buffer index of the output buffer array for *each* tile, and finally merges (sum-reduces) the tile to the output feature vectors.

3 Existing *SC* Engines

There are a few prior works [9, 14, 43] that optimize *SC* on GPUs. MinkowskiEngine [8] is the first open-source library that efficiently implements *SC* on GPUs, and is specifically optimized for *SC* layers with small feature channel sizes. SpConv [9] improves the *SC* execution by leveraging data locality in GEMM operations. TorchSparse [43] uses a single Gather and a single Scatter operation for *all* weight offsets, and groups GEMM operations by performing zero padding in GEMM's operands, thus reducing kernel launch overheads and increasing GPU hardware utilization.

We comprehensively examine existing *SC* engines [8, 9, 43] on a wide variety of real-world point cloud data, and find that prior approaches suffer from three shortcomings.

Shortcoming #1: Expensive Data Accesses in the *Map* Step. To build the kernel map, prior works employ a hash table (Figure 2 left) to store the coordinates of input point data. Then, they query the hash table for each output coordinate and weight offset to check the existence of the corresponding input coordinate. We observe that using a hash table to build the kernel map incurs a large number of irregular memory accesses that are served by GPU global memory, thus resulting in low system performance. We conclude that prior *SC* frameworks do not effectively utilize the deep on-chip memory hierarchy of modern GPUs.

Figure 3 evaluates the hit ratio in the last level cache of GPUs, when building the kernel map in *SC* execution using the hash table implementation of TorchSparse [43], the hash table implementation of MinkowskiEngine [8], the state-of-the-art 3D spatial hash table implementation on GPUs, i.e., Open3D [14], and the implementation of our work Minuet. We find that prior state-of-the-art approaches achieve very low L2 cache hit ratio, on average 36% and 19% for the MinkowskiEngine [8] and TorchSparse [43], respectively, due to random memory access patterns incurred in their hash table-based implementations. As the number of input points increases, we also observe that hash table-based implementations have even lower cache hit ratios. Even if the best-performing state-of-the-art hash table implementation, i.e., Open3D [14], was used in *SC* to build the kernel map,

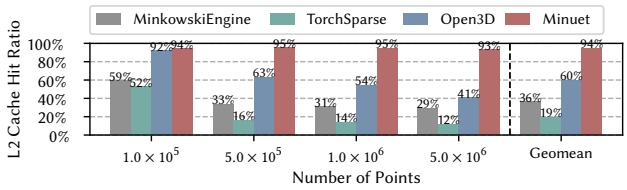

Figure 3. L2 cache hit ratio in building kernel maps of the *Map* step on RTX 3090 for various kernel map building implementations.

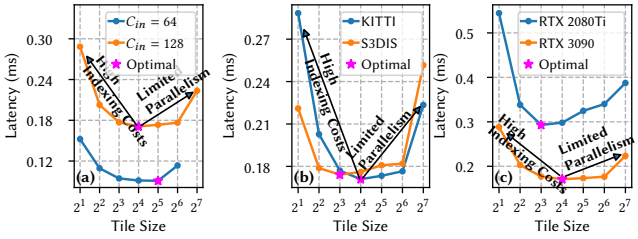

Figure 4. The performance of a Gather operation, when varying the (a) input channel size, (b) real dataset, and (c) GPU architecture.

the L2 cache hit ratio would only be 41% for a large number of data points, i.e., 5×10^6 points. Instead, we follow a fundamentally different approach in Minuet by employing a binary search-based algorithmic scheme to build the kernel map. We design Minuet to highly utilize the on-chip memory hierarchy of GPUs, thus providing high memory efficiency in the kernel map building of *SC*. Figure 3 demonstrates that Minuet achieves at least 93% L2 cache hit ratio (even for a large number of data points), thus significantly improving the performance in the *Map* step (See also Figure 16).

Shortcoming #2: Sub-Optimal Performance in Gather and Scatter Operations. In Gather and Scatter operations, prior works [43] use a fixed tile size to process the multiple input/output feature channels. However, we observe that the best-performing tile size depends on the configuration of the *SC* layer, the real dataset and the GPU architecture. Figure 4 presents the latency in Gather operation for various tile sizes, when varying the (a) input channel size (layer configuration), (b) real dataset, and (c) GPU architecture. On the one hand, using a small tile size to process input/output feature vectors results in many tiles corresponding to the *same* buffer index of the input/output metadata tables. With this approach, *each* entry in the metadata table is accessed multiple times, thus resulting in *high indexing costs* in metadata tables with significant performance overheads. On the other hand, using a large tile size leads to fewer tiles to be parallelized, and thus results in *limited execution parallelism*, causing hardware resource underutilization. Moreover, the best-performing tile size depends on the channel size, input dataset, and GPU architecture. Prior works overlook the aforementioned trade-off by using a single *fixed* tile size in *SC* execution, and thus incur either high metadata indexing costs or low execution parallelism, which results in sub-optimal performance. Instead, Minuet provides a lightweight adaptive policy that dynamically autotunes the tile size based on the characteristics of each layer, real dataset, and GPU architecture, and thus achieves near optimal performance in Gather and Scatter operations.

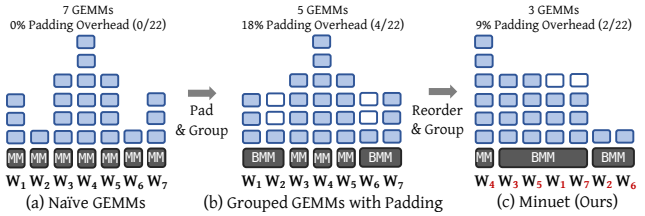

Figure 5. Various approaches to execute GEMM operations in *SC*, where one blue and white squares denote one actual input feature vector and one zero-padded feature vector, respectively. Assuming x and y are the number of padded feature vectors and actual input feature vectors, respectively, the padding overhead is defined as (x/y).

Shortcoming #3: High Padding Overhead in GEMM Operations. Using a naïve approach to execute each small GEMM kernel separately for each weight offset in the *GMaS* step (Figure 5a) incurs excessive kernel launch overheads in GPUs, as explained in prior works [29, 43, 46]. Thus, prior works [43] propose a batched GEMM approach, shown in Figure 5b: they group multiple GEMM operations together, padding with zero values the corresponding matrices (e.g., see the 2-nd and 6-th column in Figure 5b) of adjacent GEMM operations to have the same height, and launch one *single* batched GEMM kernel for multiple weights. This grouping approach improves hardware utilization and kernel launch overheads in GEMM operations. However, we observe that this approach incurs high padding overhead, since prior works group GEMM operations in the *GMaS* step following the order induced by the *Map* step, i.e., the order in which weight offsets are processed in the *Map* step. As a result, adjacent GEMM operations with that ordering might have a large difference in their sizes, causing a larger amount of padding with zero values, which in turn results to redundant data accesses and computations with zero (useless) values.

Instead, we argue that reordering the weights before grouping the GEMM operations can reduce the amount of padding with zero values, and provide a better GEMM grouping with lower padding overhead. For instance, Figure 5b shows that grouping GEMM operations in the order induced by the *Map* step incurs 18% padding overhead and launches 5 GEMM kernels. However, if we first reorder weights carefully, and then group the GEMMs into batched GEMM kernel launches, we can provide only 9% padding overhead and launch only 3 GEMMs, as shown Figure 5c. To this end, we design Minuet to implement a lightweight GEMM reordering group policy that reduces the padding overhead and also provides a small number of GEMM kernel launches. Our evaluations

show that prior state-of-the-art *SC* work [43] incurs on average 11% padding overhead and executes on average 11.1 GEMM kernels, while Minuet has 8.2% padding overhead and executes 7.76 GEMM kernels.

4 Minuet: Overview

Minuet is a novel high-performance *SC* engine tailored for modern GPUs. Minuet highly utilizes the on-chip memory hierarchy of GPUs, eliminates unnecessary data access and computations, and effectively adapts to both the data distribution of each particular input dataset and the characteristics of the GPU architecture used.

Unlike prior *SC* engines [8, 9, 43] that use a hash table (e.g., cuckoo hash tables [1]) for building kernel maps, in this work we argue that using a sorted key-value array and *binary search* is a more efficient alternative solution than hash tables on GPUs. Even though the naïve binary search in a sorted array has worse theoretical computational complexity than hash tables and does not effectively leverage the on-chip memory hierarchy of GPUs [1, 2], we challenge these two understandings by proposing a novel binary search-based algorithm tailored for building kernel maps in *SC*s on GPUs. Figure 6 presents the high-level overview of Minuet. Overall, we propose four key ideas that accelerate both the *Map* and the *GMaS* step in *SC* execution.

1. Segmented Query Sorting: We sort the output coordinates and weight offsets separately, and create a query array in the *Map* step, organizing the queries to *sorted segments* ❶: each *query segment* is sorted and consists of the queries corresponding to all output coordinates associated with the *same* weight offset. Then, for each query segment, we execute binary search queries in a sorted array that stores the input coordinates and their indices, named as *source array*. This way when we perform binary search lookups by iterating over consecutive sorted queries, we leverage temporal data locality: consecutive queries of the same sorted query segment have similar data access patterns in the source array, i.e., they access same elements in the source array with a high probability. Segmented query sorting both minimizes the sorting overheads and improves data locality in on-chip caches of GPUs within the source array, thus accelerating performance to build kernel maps.

2. Double-Traversed Binary Search: We split the source array into small disjoint *source blocks*. For each source block, we perform a *backward* binary search ❷ to each query segment to find out all possible queries corresponding to that source block, and these queries are organized as a *query block*. Then, we load each *source block* in the GPU scratchpad memory, and process all queries in the associated *query block* by executing a *forward* binary search within the *source block*. For each query, the proposed double-traversed binary search algorithm reduces the search range, since only a subset of the source array elements need to be compared, thus decreasing the number of computations (comparisons) performed, and provides low data access costs, by highly utilizing the on-chip memory hierarchy on GPUs.

3. Autotuned Gather/Scatter: We design a tile size tuner that autotunes ❸ the tile size in Gather and Scatter operations for each *SC* layer. First, we sample a few input point clouds from the dataset and create the corresponding input and output metadata table entries for these samples. Then, we evaluate the latency for all possible tile sizes and find the best-performing tile size for Gather and Scatter operation. Finally, we process all input point clouds from the dataset using the selected best-performing tile size. By autotuning the tile size at each *SC* layer, Minuet achieves low metadata indexing costs and high execution parallelism in Gather and Scatter operations, and effectively adapts itself to the characteristics of each layer, dataset, and GPU architecture.

4. Padding-Efficient GEMM Grouping: We re-order the GEMM operations ❹ based on the sorted sizes of their corresponding input and output feature vectors. Then, we group adjacent GEMM operations (to be executed as a single GEMM kernel) associated with feature vectors of same or very similar sizes, which allows us to minimize the amount of zero paddings. With this key technique, we reduce redundant data accesses and computations with zero values, and minimize the number of GEMM kernels executed, thus enabling low kernel launch overheads on GPUs.

5 Minuet: Design Details

5.1 Optimizing the *Map* Step

To build the kernel map, the coordinates of the non-zero data points in the input point cloud, i.e., the input coordinates $\{\mathbf{p_j}\}$, are stored to an array to be searched from (henceforth referred to as *source array*), the size of which is denoted by $|\mathcal{P}|$. *SC* creates an array of queries (henceforth referred to as *query array*) that stores all possible input coordinates $\{\mathbf{q_i}+\boldsymbol{\delta}_\mathbf{k}\}$, where $\{\mathbf{q_i}\}$ and $\{\boldsymbol{\delta}_\mathbf{k}\}$ are the output coordinates and weight offsets, respectively. Then, *SC* executes each query to check whether the query exists as an input coordinate in the source array. Assuming an *SC* layer with kernel size K and $|\mathcal{Q}|$ output coordinates, the size of the query array is $K^3|\mathcal{Q}|$.

Key Observation. *Sorting the queries and executing them via binary search comprises many common elements in the search paths between adjacent queries.*

Figure 7 shows an example execution of searching four queries via binary search in the source array (that is visualized as a binary search tree), by traversing queries randomly, i.e., unsorted queries (left), versus via a sorted approach, i.e., sorted queries (right). The annotated white bold coordinates represent the elements of the source array that are common, when executing two adjacent queries. In the *unsorted* query execution, when searching for $(1, 0, 0)$ right after $(0, 0, 8)$ has been searched, there is *only one* element (i.e., $(0, 2, 5)$) that is common between the two search paths of adjacent queries. Instead, in the *sorted* query execution, when searching for $(0, 0, 8)$ right after $(0, 0, 0)$ has been searched, there are three

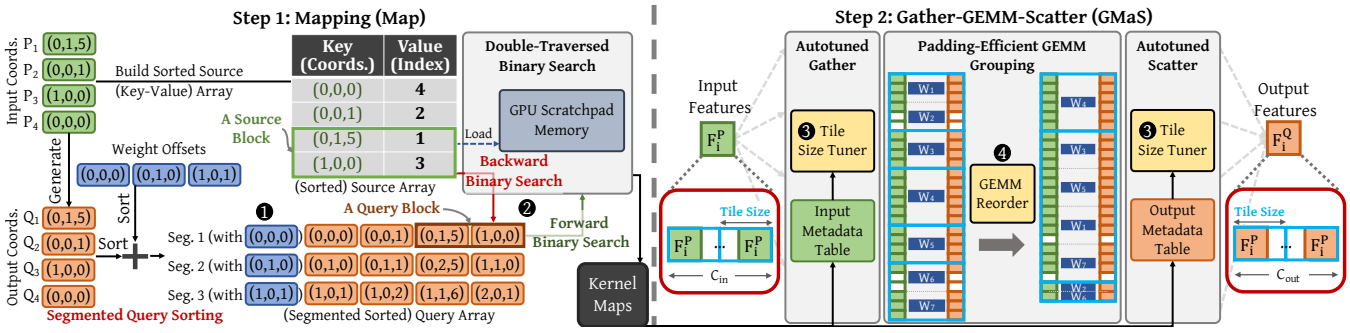

Figure 6. High-level overview of Minuet.

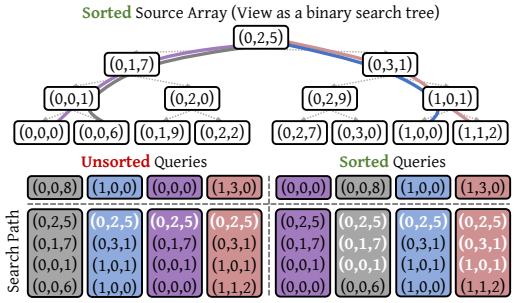

Figure 7. An example binary search execution of four queries, when queries are traversed randomly, i.e., unsorted queries (left) versus via a sorted query approach (right).

common elements (i.e., $(0, 2, 5)$, $(0, 1, 7)$ and $(0, 0, 1)$) in the search paths of adjacent queries.

The common elements between the search paths of consecutive sorted queries enable two implications for binary search. First, when executing two consecutive queries with binary search, there is a high probability that the second query accesses the *common* source array elements via the on-chip caches, since common elements might be already cached thanks to executing the first query. Thus, the binary search with sorted queries is friendly to GPU memory hierarchy. Second, each element accessed in the search path corresponds to one comparison between a query and a source array element. Having common elements in search paths means that *the same* source array elements are compared to multiple *sorted* queries (e.g., $(0, 2, 5)$ is compared to all four sorted queries). Thus, if we could find the lower bound of the source array element in the query array segment, i.e. the smallest query within the query segment that is no smaller than a source array element, we could avoid the comparisons to that source array element, thus reducing the number of comparisons in binary search scheme with sorted queries.

To this end, Minuet address two challenges. First, binary search with sorted queries necessitates that the source and query arrays need to be sorted, and thus we need to minimize the sorting overheads in both arrays (**Challenge 1**). Second, we need to exploit both the memory friendliness and the aforementioned optimization of reducing comparisons in binary search with sorted queries (**Challenge 2**).

5.1.1 Segmented Query Sorting

A naïve approach to build the kernel map is to materialize all possible queries $\{\mathbf{q_i} + \boldsymbol{\delta_k}\}$ in a query array, sort the query array and execute binary search for each query within the source array. This approach, referred to as *full query sorting*, is depicted in Figure 8 top, in which we assume that there are 4 output coordinates $\mathbf{q_i}$ and 3 weight offsets $\boldsymbol{\delta_k}$, thus resulting in 12 queries, which are perfectly sorted in the full query sorting approach. Binary searching with full query sorting is highly cache-friendly, as explained, since searching *sorted* queries in the source array results in many accesses to the same elements of the source array. However, full query sorting incurs high sorting overheads: (1) the size of the query array $K^3|Q|$ is much larger than the size of the source array $|\mathcal{P}|$, and thus sorting the query array causes even higher sorting overheads than that of the source array itself; (2) the large query array needs to be sorted at *each SC* layer of the point cloud network. In practice, we found that using full query sorting approach to build the kernel maps of *SC* layers takes much longer time than using the hash table-based approach of prior *SC* engines [8, 43]. Therefore, we conclude that the full query sorting approach has huge sorting overheads that offset its cache benefits, and this naïve approach does not address the Challenge 1.

To minimize sorting overheads in binary search-based kernel map building, we propose *segmented query sorting*, depicted in Figure 8 bottom. In the segmented query sorting, we sort the array of the output coordinates $\mathbf{q_i}$ and the array of the weight offsets $\boldsymbol{\delta_k}$ *separately*, i.e., we materialize two separate arrays in memory (solid green boxes) and sort each of them, and then we execute all possible queries as *sorted segments* (dashed green boxes): we iterate through all weight offsets, and for each weight offset $\boldsymbol{\delta_k}$ we on-the-fly create a sorted segment of possible queries (without materializing a new array for the segment) by adding the current weight offset to each sorted output coordinate $\mathbf{q_i}$ of the output coordinate array. For example, in Figure 8 the 2-nd segment is created on-the-fly by adding the 2-nd sorted weight offset $(0, 1, 0)$ to each sorted output coordinate, i.e., $(0, 0, 0)$, $(0, 0, 8)$, $(1, 0, 0)$, $(1, 3, 0)$, thus the 2-nd segment comprises of the four elements $(0, 1, 0)$, $(0, 1, 8)$, $(1, 1, 0)$, and $(1, 4, 0)$.

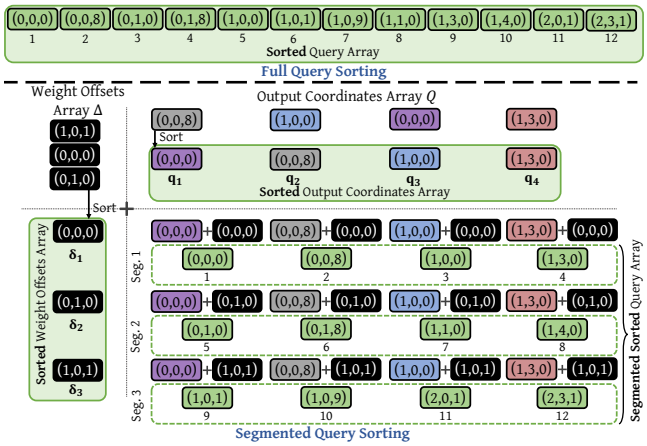

Figure 8. An example of full query sorting (top) and segmented query sorting (bottom).

Segmented query sorting leverages many cache friendly accesses in binary search-based query lookups, while also minimizing the sorting overheads in both source and query arrays, thus addressing **Challenge 1**.

On the one hand, since the output coordinate array is sorted, and we create a query segment by adding the same weight offset to each sorted element of the output coordinate array, the produced queries in the query segment are by nature sorted as well (See segments of Figure 8). In practice, for a typical *SC* layer with kernel size K, the number of weight offsets (i.e., K^3) to be sorted for each *SC* layer is much smaller than the number of output coordinates, i.e., $K^3 \lll |Q|$. For example, a typical *SC* layer has a kernel size 3, and there are 27 weight offsets to be sorted, while the number of output coordinates is much larger (e.g., 10^5). As a result, there is only a small number of segments, but *a large number of queries* within each segment. Thus, segment query sorting has segments with many sorted elements, and enables a sufficiently large number of cache-friendly memory accesses for binary search that are close to that of the full query sorting approach.

On the other hand, segmented query sorting minimizes the sorting overheads for four compelling reasons.

First, weight offsets sorting is *not* in the critical inference path. Weight offsets are determined by the *SC* layer configuration itself (i.e., the kernel size and stride, as discussed in Section 2.1) and are independent to the input point cloud data, so they need to be sorted only *once* for each *SC* layer in the network. This sorting is performed as a preprocessing step, when loading the configuration of the *SC* layer, and has negligible costs, since the number of weight offsets of each *SC* layer is very small (e.g., 27 for a typical kernel size 3).

Second, segmented query sorting sorts the output coordinate array of size $|Q|$ (e.g., 10^5), the sorting cost of which is smaller than sorting the whole query array of size $K^3|Q|$ (e.g., 27×10^5) which is needed in the full query sorting approach. Moreover, segmented query sorting sorts the output

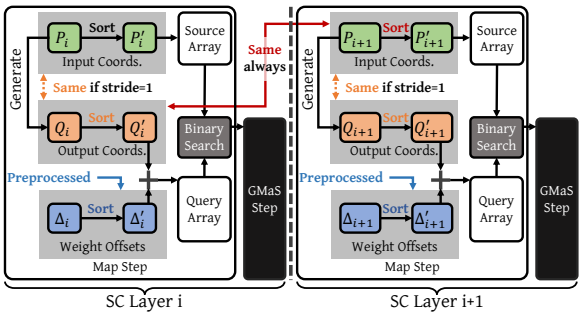

Figure 9. Optimizing sorting overheads of weight offsets and output coordinates in adjacent *SC* layers.

coordinate array only *once* for *all* query segments, which are created on-the-fly (dashed green boxes in Figure 8) and are *not* materialized in memory. Thus, it performs much smaller number of memory accesses for queries compared to the full sorting approach, that first needs to materialize in memory the whole query array before sorting it (solid green box in Figure 8 top).

Third, *SC* models typically have multiple *SC* layers connected sequentially [8] the one after the other, and the output coordinates of one *SC* layer are the input coordinates of the subsequent *SC* layer. Figure 9 presents two adjacent *SC* layers as a part of a large point cloud network. Segmented query sorting requires the output coordinate array Q to be sorted in the *Map* step of each *SC* layer to perform binary search-based kernel map building. Thus, by leveraging segmented query sorting, sorting the input coordinate array of the layer $i+1$ is *completely* eliminated, since input coordinate array of the layer $i+1$ is the always same as output coordinate array of the layer i, which is already sorted (the red solid arrow in Figure 9). This optimization cannot be enabled with the full query sorting approach, since it necessitates a different (separate) array across different *SC* layers to store and sort the queries, because the weight offsets (or the coordinates) can be different.

Fourth, when the stride of an *SC* layer is one, the output coordinates Q are identical to the input coordinates $\mathcal{P}$ (orange dashed arrows in Figure 9), as explained in Section 2.1. Therefore, in *SC* layers with stride 1, we do *not* materialize and sort two separate arrays for the source and query arrays. Instead, we materialize only one array that serves as both sort and query array and sort it only *once*. Similarly this optimization cannot be enabled with the full query sorting approach because it necessitates separate query array to store and sort the queries.

Overall, Minuet significantly minimizes sorting overheads and bulid the kernel map very efficiently via segmented query sorting. Minuet leverages existing GPU radix sorting libraries [10] to sort the arrays at low cost. In Figure 17 (Section 6.4), we show that the building time of Minuet is faster than that of prior *SC* engines.

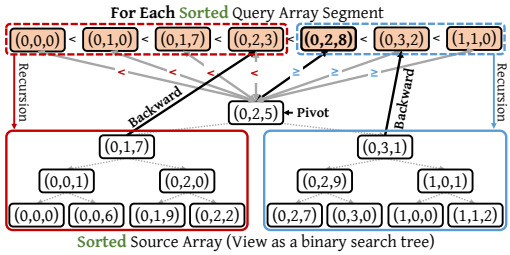

Figure 10. An example of using backward binary search to find the lower bound of the pivot in the query array segment to reduce the number of comparisons.

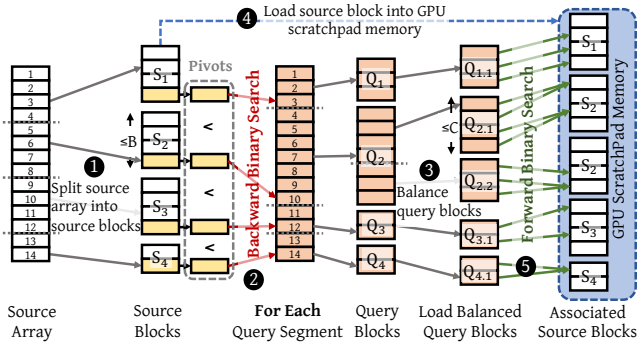

Figure 11. Double-traversed binary search execution steps.

5.1.2 Double-Traversed Binary Search

To solve the Challenge 2, we introduce a novel binary search algorithm that both reduces the comparisons, and efficiently leverages the on-chip memory hierarchy of GPUs when using sorted queries.

Figure 10 depicts an example of executing binary search with one sorted query array segment into the sorted source array that is represented as a binary search tree. Each query in the query array segment needs to be compared with the middle element of the source array, i.e., the element $(0, 2, 5)$, in the first comparison step of the binary search. We refer to such element as the **pivot**. We observe that as traversing the sorted queries of the query array segment there is *at most one change* from a smaller "<" element to a larger ">" element than the pivot, e.g., all queries from $(0, 0, 0)$ to $(0, 2, 3)$ are smaller than the pivot (the red dashed box), and all queries from $(0, 2, 8)$ to $(1, 1, 0)$ are larger than pivot (the blue dashed box). Thus, if we find the lower bound of the pivot within the query array segment (i.e., the bold coordinate $(0, 2, 8)$), namely the first element of query array segment that is no smaller than the pivot, we could avoid many comparisons to the pivot for the elements of the query array segment: according to the transitivity property, all queries before the lower bound (red dashed box) will be smaller than the pivot, and all queries after the lower bound (blue dashed box) will be larger than or equal to the pivot.

Key Idea. To find the lower bound of pivot, we apply binary search in a *backward* manner (**backward binary search**), namely to binary search the pivot in the query segment.

This key idea can be applied recursively in all elements (pivots) of the source array. The sorted source array can be split into the (i) subarray with elements smaller than the pivot (the left subtree with red solid box), and the (ii) subarray with elements no smaller than the pivot (the right subtree with blue solid box). The (i) subarray is associated with the left query subarray, i.e., the queries that are smaller than the lower bound (the red dashed box), and the (ii) subarray is associated with the right query array, i.e., the queries that are no smaller than the lower bound (the blue dashed box). Then, the backward binary search is applied to the pivot (roots) of the (i) and (ii) source subarrays (subtrees), i.e., $(0, 1, 7)$ and $(0, 3, 1)$, and proceeds recursively to all elements of the source array. However, recursively applying backward binary search would require many recursive function calls, which limits the degree of execution parallelism and incurs high warp divergence overheads on GPUs.

To this end, we consider only one level of backward binary search and propose *double-traversed binary search* algorithm for kernel map building, that consists of two steps. Figure 11 presents the execution steps of our proposed algorithm.

Backward Binary Search. Instead of using only one pivot in source array, we select multiple pivots, and split the source and query arrays to multiple blocks. We first partition the source array into multiple *source blocks* ❶, each of them has a size that is no larger than a hyperparameter B. Then, we use the last elements of each source block as *pivots* to split the query segment into multiple *query blocks*: for each pivot of a source block, we perform backward binary search to the sorted query segment ❷ to find the subset of consecutive queries, i.e., the query block, that is associated with that source block. Thus, the query segment is split in multiple query blocks, the number of which is equal to the number of source blocks in the source array.

Forward Binary Search. We observe that the query blocks are data dependent, thus their sizes might significantly vary across them. To enable load balance across GPU threads, we balance query blocks ❸ by further splitting all query blocks that have size larger than than a hyperparameter C (e.g., Q_2). This way all query blocks are load balanced, i.e., having size that is no larger than C. Then, we assign one CUDA thread block to each query block, where the thread block processes all queries of that query block by performing forward binary search to the associated source block. To do so, each CUDA thread block first loads the associated source block into the GPU scratchpad memory ❹ to minimize the number of global memory accesses. Then, each thread of the CUDA thread block executes forward binary search ❺ to check the existence of each query in the query block within the source block.

Minuet achieves very low memory access costs, while also reduces the number of comparisons (**Challenge 2**). First, both backward and forward binary search provide high memory efficiency. The backward binary search is highly cache friendly, since the pivots of source blocks are *sorted*, thus

it is treated as binary search with sorted queries (Section 5.1). The forward binary search also provides high memory benefits, since we only access global memory, when fetching the source block to scratchpad memory and the query block to register files, while accessing the elements within both the source and the query blocks has a sequential memory access pattern. Second, we reduce the search range for each query block from the whole source array of size $|\mathcal{Q}|$ to the source block of size B by the backward binary search. This way, we significantly reduce the number of comparisons performed by our algorithm.

5.1.3 Computational Complexity of Segmented Sorting Double-Traversed Binary Search We use Concurrent-Read-Exclusive-Write (CREW) as the Parallel RAM model for computational complexity analysis (i.e., work and time complexity). For simplicity, we assume all source blocks and load balanced query blocks have the same sizes of B and C, respectively. Under this assumption, we provide only the work complexity analysis, since both the backward and the forward binary search can be straightforwardly parallelized, and the time complexity is simply the work complexity divided by the number of processors.

Let K, $|\mathcal{P}|$, and $|\mathcal{Q}|$ be the kernel size of the *SC* layer, the number of input and output coordinates, respectively. In the backward binary search, for each of the $\left\lceil \frac{|\mathcal{P}|}{B} \right\rceil$ source block, loading the last element of the source block takes $O(1)$ time and searching it in one sorted query segment array takes $O(\log |\mathcal{Q}|)$ comparisons. In the forward binary search, for each of the $O\left(\frac{|\mathcal{P}|}{B} + \frac{|\mathcal{Q}|}{C}\right)$ load balanced query block[1], loading the source and query block from global memory takes $O(C+B)$ time, and the in-scratchpad binary search takes $O(C \log B)$ time. In *SC*, $|\mathcal{P}|$ has the same order of magnitude with $|\mathcal{Q}|$. By configuring $B = \frac{|\mathcal{P}|}{|\mathcal{Q}|} \log |\mathcal{Q}|$ and $C = \sqrt{\frac{|\mathcal{Q}|}{|\mathcal{P}| \log B}} B$, the work complexity of the segmented sorting double-traversed binary search is:

$$\begin{aligned} &O\left(K^3 \left(\frac{|\mathcal{P}|}{B} \log |\mathcal{Q}| + \left(\frac{|\mathcal{P}|}{B} + \frac{|\mathcal{Q}|}{C}\right)(B + C \log B)\right)\right) \\ =&O\left(K^3 |\mathcal{Q}| \log \log |\mathcal{Q}|\right) \end{aligned} \tag{4}$$

Minuet can thus achieve a computational complexity close to that of hash table-based kernel map building, i.e., $O(K^3|\mathcal{Q}|)$.

5.1.4 Minuet's Selection of Hyperparameters B and C Minuet carefully chooses the hyperparameter B and C. Intuitively, hyperparameter B balances the trade-off between the execution times of the forward and the backward binary search: a larger value of B results in fewer but larger source blocks, which decreases the number of comparisons in the backward binary search, but increases the number of comparisons in the forward binary search, and vice versa.

[1]Let x_i denote the size of the unbalanced query block for the i-th source block, we have $\sum_{i=1}^{\left\lceil \frac{|\mathcal{P}|}{B} \right\rceil} \left\lceil \frac{x_i}{C} \right\rceil \in O\left(\frac{|\mathcal{P}|}{B} + \frac{|\mathcal{Q}|}{C}\right)$ load balanced query blocks.

Algorithm 1 The Gather operation

Arguments: Weight offsets Δ, Input channel size C_{in}, Input coordinates $\mathcal{P} = \{\mathbf{p_i}\}$, Input buffer array $\{\mathbf{b_i}\}$, Input metadata table *IMT*, Gather tile size T

Returns: Input feature vectors $\{\mathbf{F_i^{\mathcal{P}}}\}$

1: **for** $t \leftarrow 0, 1, \ldots, \frac{C_{\text{in}}}{T} - 1$ **in parallel do**
2: **for** $\mathbf{p_i} \in \mathcal{P}$ **in parallel do**
3: Read from the t-th tile of $\mathbf{F_i^{\mathcal{P}}}$ to $\mathbf{v}$ (in register files)
4: **for** $\boldsymbol{\delta_k} \in \Delta$ **do**
5: index $\leftarrow$ GetInputBufferIndex(*IMT*, $\boldsymbol{\delta_k}$, $\mathbf{p_i}$)
6: **if** index $\neq \emptyset$ **then**
7: Write $\mathbf{v}$ to the t-th tile of $\mathbf{b_{index}}$

Hyperparameter C balances the trade-off between data movement and the load balance in the forward binary search: a larger value of C results in fewer but larger query blocks, which decreases the data movement for copying the associated source block to the scratchpad memory, but increases the load imbalance among CUDA thread blocks, and vice versa. In Figure 18, we provide a sensitivity study on the values of the hyperparameters B and C using various GPUs, and find that with thread block size of 128, configuring $B = 256$ and $C = 512$ (default Minuet's values) consistently achieves the best performance among all evaluated GPUs and datasets. For flexibility, we expose B and C as configurable hyperparameters to users.

5.2 Optimizing the *GMaS* Step

In this section, we describe the optimizations and trade-offs of Minuet in the *GMaS* step.

5.2.1 Autotuned Gather/Scatter We summarize Minuet's algorithm of the Gather operation in Algorithm 1. The Scatter operation can be conducted similarly to Gather.

With a given tile size T, the Gather operation assign one CUDA thread to each feature channel tile and each input coordinate, which achieves a parallelism of $\frac{C_{\text{in}}}{T} \times |\mathcal{P}|$. As shown in blue at line 5, we observe that the accesses in input metadata table are not related to the tile index t, which implies the all the $\frac{C_{\text{in}}}{T}$ accesses are to the same entry in the metadata table within the same tile. Hence, on the one hand, increasing the tile size reduces indexing costs, namely the number of accesses to the metadata table, i.e., $\frac{C_{\text{in}}}{T} \times |\mathcal{P}| \times K^3$. However, on the other hand, increasing the tile size T also reduces the execution parallelism $\frac{C_{\text{in}}}{T} \times |\mathcal{P}|$. As a result, we might not saturate the GPU, especially when the number of input/output coordinates is small or when we use powerful GPUs with a large number of processing units.

To trade-off between indexing costs and execution parallelism, we propose to autotune the tile size for each Gather and Scatter operation of the model. In Algorithm 2, we demonstrate how Minuet autotunes the Gather operation (the Scatter operation is autotuned similarly). Specifically, we sample a few point clouds from the dataset and feed them to the *SC* network (line 1). Then, for each *SC* layer in the

Algorithm 2 Autotuning the Gather operations.

Arguments: A *SC* network $\mathcal{M}$ of n layers $\mathcal{L}_i$, A point cloud dataset for tuning $\mathcal{D}$, The rounds of tuning R
Returns: The tuned *SC* network $\mathcal{M}^{\text{tunned}} = \{\mathcal{L}_i^{\text{tunned}}\}$
1: $\mathcal{D}_{\text{Sampled}} \leftarrow$Sample a few point clouds from the dataset $\mathcal{D}$
2: **for** each layer $\mathcal{L}_i$ **do**
3: Compute input metadata tables $IMT^{(i)}$ based on $\mathcal{D}_{\text{Sampled}}$
4: $T^*_{\text{Gather}} \leftarrow \emptyset$
5: **for** each divisor T of $\mathcal{L}_i$'s input channel size C_{in} **do**
6: Profile GATHER for R rounds with T and $IMT^{(i)}$
7: Update T^*_{Gather} with T if the latency is smaller
8: $\mathcal{L}_i^{\text{tunned}} \leftarrow \mathcal{L}_i$ with T^*_{Gather} in the Gather operation

network, we create the metadata tables for these few point clouds and use them to find the best-performing tile size (line 3). Next, we exhaustively search all possible tile sizes (line 5), i.e., the divisors of C_{in}, and select the tile size with the minimum latency (line 7). Note that this autotuning process only happens once, before running the inference (pre-processing cost), and does not introduce significant overhead (less than 2 minutes) as presented in Section 6.1.

5.2.2 Padding-Efficient GEMM Grouping

To improve hardware utilization in transforming input features to output features, prior *SC* engines [43] use zero-padding in input and output features to achieve better compute regularity and low launch overheads in GEMM operations. However, we observe that the padding strategy proposed in prior works is still inefficient. To tackle this, we propose to reorder the weights based on the size of their corresponding GEMM operations, i.e., in non-decreasing order of the number of input features to be multiplied by each weight. After reordering, we employ a similar adaptive policy for grouping adjacent GEMM operations, as proposed by prior works [43].

Intuitively, after we reorder the weights, adjacent weights will have the same or very similar sizes in GEMM operations. Thus, there is only a small amount of padding with zero values needed to have the same sizes/heights in the GEMM's operands, which consequently reduces unnecessary data accesses and computations. Note that the reordering requires sorting the GEMM sizes and permuting the weights. However, we found this sorting incurs negligible overhead, being less than 4% of the layer execution time. This sorting overhead is accounted for in our evaluations, and in Section 6.3 we show that Minuet has better layerwise performance than prior works. To further improve hardware utilization, we parallelize all GEMM kernels by executing them on a pool of CUDA streams [36]. We set the stream pool size s to 4 in Minuet, since we found that increasing s larger than 4 results in no further performance speedups.

6 Evaluation

6.1 Methodology

We followed existing common practice [13, 17, 43, 48, 49] to develop methodology to evaluate Minuet's SC executions.

Platforms. We evaluate Minuet on 4 NVIDIA GPU servers, RTX 2070 Super (8 GB), RTX 2080 Ti (11 GB), RTX 3090 (24 GB), and Tesla A100 (80 GB). All GPU servers have CUDA 11.8.0 and PyTorch 2.0.0 installed. Unless otherwise noted, we present detailed evaluation results on RTX 3090.

Baselines. We compare Minuet with two state-of-the-art *SC* engines: (1) MinkowskiEngine [43], and (2) TorchSparse [8]. In Minuet, we account for both the overheads of sorting coordinates and GEMM reordering in end-to-end, layerwise, and *GMaS* step evaluations. However, for TorchSparse and Minuet, we exclude the autotuning time as both autotuning processes happen only *once* and are before the inference. The autotuning process of Minuet takes less than 2 minutes to finish on all datasets evaluated.

Neural Networks. We evaluate two representative and commonly used 3D point cloud neural networks: (1) SparseResNet21 (*ResNet*) [18, 19] that serves as the backbone for the widely used CenterPoint 3D object detector [50]; and (2) MinkUNet42 (*UNet*) [8] that achieves top-level accuracy in processing 3D point cloud data.

Datasets. We evaluate four large-scale point cloud datasets: (1) SemanticKITTI Dataset (*KITTI*) [4] which includes outdoor LiDAR scans for self-driving scenarios, (2) Stanford 3D Indoor Scene Dataset (*S3DIS*) [3] which labels 3D objects in indoor areas, (3) Semantic3D Dataset (*Sem3D*) [22] which is a large-scale dataset for outdoor scenes, and (4) ShapeNetSem Dataset (*Shape*) [5, 41] which contains large-scale point clouds for 3D models. Note that to feed a point cloud to *SC* networks, the floating-point number coordinates are first voxelized [8] into integers. After voxelization, the average sparsity[2] is 0.04%, 2%, 0.03%, and 10% for the *KITTI*, *S3DIS*, *Sem3D*, and *Shape* datasets, respectively. To study how the Minuet's optimizations in the *Map* step are affected by data distribution and sparsity patterns (Figure 13, 16, and 17), we generate a random artificial dataset: we vary the voxelization process in *Sem3D* dataset to provide different numbers of non-zero points in each point cloud in the dataset.

6.2 End-to-End Performance

Total Speedup. Figure 12 compares the end-to-end speedup of all *SC* engines, when executing the neural networks on various datasets and GPUs. We make two key observations. First, there is no clear winner between MinkowskiEngine and TorchSparse: MinkowskiEngine outperforms TorchSparse on *ResNet* network, while it performs worse than TorchSparse on *UNet* network. This is because MinkowskiEngine is specialized for small channel size *SC* layers, which dominate the *ResNet* network. Second, Minuet consistently outperforms prior *SC* engines, by 1.74× on average (up to 2.19×) over MinkowskiEngine, and 1.74× on average (up to 2.22×) over TorchSparse, for all neural networks, datasets, and GPU systems. Noticeably, Minuet achieves close to 2× speedup

[2]This is defined as the number of non-zero data points divided by the bounding volume and averaged over all point clouds in the dataset.

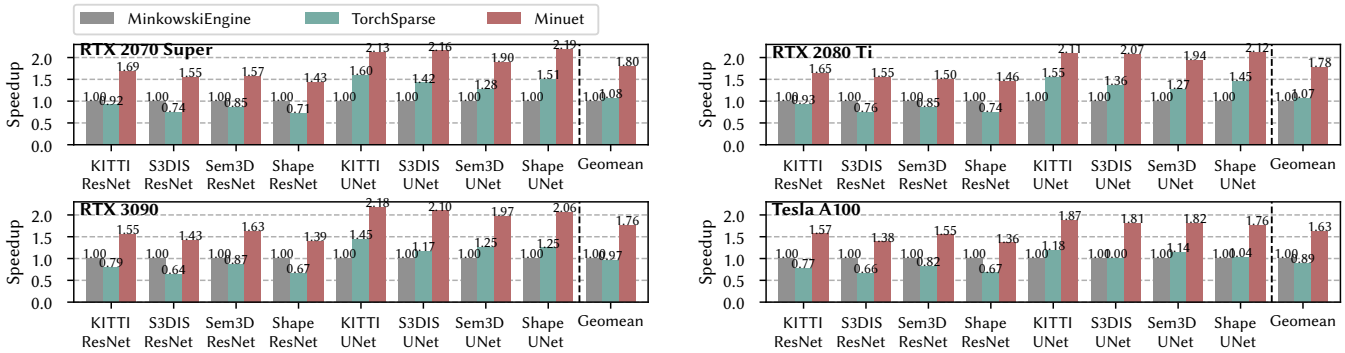

Figure 12. End-to-end performance of all *SC* engines using various networks, real datasets and GPU architectures.

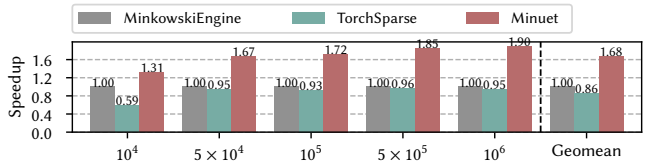

Figure 13. End-to-end performance of all *SC* engines with varying point cloud density.

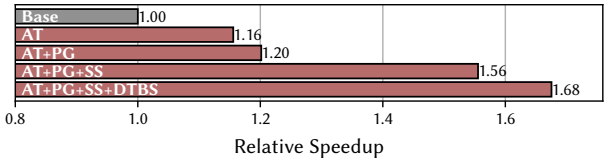

Figure 14. Performance breakdown of the key ideas in Minuet, where AT stands for Autotuned Gather/Scatter, PG for Padding-Efficient GEMM Grouping, SS for Segmented Query Sorting, and DTBS for Double-Traversed Binary Search.

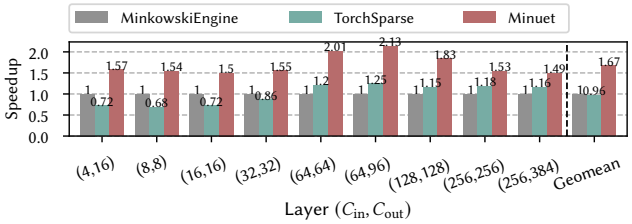

Figure 15. Layerwise speedup of *SC* engines averaged across all real datasets when varying the input and output channel sizes in the SC layers.

on *UNet* on RTX 2070, RTX 2080 Ti, and RTX 3090 over MinkowskiEngine thanks to low-cost data accesses in the *Map* step and high parallelism and hardware utilization in the *GMaS* step. Overall, we conclude that Minuet achieves the best performance over prior state-of-the-art *SC* engines across various sparse point cloud networks, real datasets, and even when using different GPU architectures.

Sensitivity on Point Cloud Density. We evaluate Minuet on a random synthetic dataset to understand how Minuet generalizes to point clouds with different input densities. Specifically, we randomly generate point clouds within a fixed bounding volume ($400 \times 400 \times 400$) and vary the number of non-zero points from 10^4 to 10^6 to achieve different input densities. Figure 13 shows the end-to-end speedup with MinkowskiEngine and TorchSparse, where Minuet consistently outperforms existing *SC* engines on various input densities by on average 1.68× (up to 1.90×).

Speedup Breakdown. To understand how each key idea of Minuet contributes to the final performance, we evaluate performance by incrementally enabling the four key ideas proposed in Minuet in Figure 14. We draw two conclusions. First, all Minuet's four key ideas significantly contribute to the final end-to-end performance. Second, the most significant speedup comes from the segmented query sorting, which shows the superiority of using the sorted key-value array instead of hash tables in *SC* execution.

6.3 Layerwise Performance

Figure 15 compares the layerwise performance of all *SC* engines on the most commonly used *SC* layer configurations. The x-axis corresponds to an *SC* layer with C_{in} input and C_{out} output channels. We calculate the geometric mean across all real datasets for each *SC* layer, and the last group column shows the geometric mean averaged across all layers.

We draw two findings. First, TorchSparse performs worse than MinkowskiEngine on *SC* layers with small channel sizes (e.g., (4, 16)), while it performs better on layers with larger channel sizes (e.g., (128, 128)). This is due to a specialized dataflow that is optimized for small channel sizes in MinkowskiEngine [43]. Second, Minuet significantly outperforms the MinkowskiEngine by on average 1.64× speedup (up to 2.16×) and TorchSparse by on average 1.67× speedup (up to 2.10×) in all layer configurations. Thus, Minuet achieves the best performance in various configurations of *SC* layers.

6.4 Performance of the *Map* Step

Query Process. Figure 16 compares (a) the execution time and (b) the L2 cache hit ratio achieved (collected with NVIDIA Nsight Compute [11]) by *SC* engines in the query process of the *Map* step using the *Sem3D* dataset and an artificial

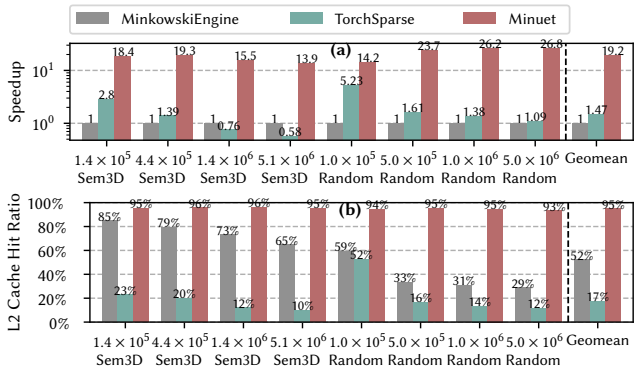

Figure 16. (a) Normalized speedup and (b) L2 cache hit ratio of the query process to build the kernel map in *SC* when varying the dataset and number of points in the point cloud.

randomly generated dataset (*Random*), that has similar sparsity and number of points with *Sem3D*. Note that Minuet's execution time includes the total binary search algorithm proposed, while the L2 cache hit ratio presented represents only the dominating forward binary search process (more than 90% in the total time of the *Map* step). In the x-axis, we present the number of points in the input point cloud and the dataset to which the input point cloud belongs.

We make two key observations. First, thanks to our novel highly memory-efficient binary search algorithm, Minuet achieves a very high L2 hit ratio, i.e., more than 95% in all datasets, and provides superior performance benefits in the *Map* step: it has 19.2× speedup on average (up to 26.8×) and 13× speedup on average (up to 24.6×) over the hash table implementations of MinkowskiEngine and TorchSparse, respectively. Second, we observe that as the number of points increases, the cache hit ratio of hash table-based implementations decreases significantly. This is because as the number of points increases, the hash table requires larger memory footprint to access the stored input coordinates, which are less likely to remain in the on-chip caches during the query execution. In contrast, Minuet's segmented sorting double-traversed binary search significantly outperforms hash table-based solutions of prior *SC* engines, and provides a robust solution, since its performance benefits remain across various numbers of inputs points.

Build Process. Figure 17 compares the time for the build process of the *Map* step, i.e., the time to build hash tables in MinkowskiEngine and TorchSparse engines, and the time to sort the input/output coordinates in Minuet. By leveraging the high-performance CUDA radix sorting libraries (e.g., NVIDIA CUB [10]), Minuet achieves lower building time overhead compared to prior *SC* engines, and thus sorting cost for the coordinates in our proposed segmented sorting double traversed binary search has negligible overhead.

Minuet's Hyperparameters. Figure 18 shows query time for building kernel maps in the *Map* step, when varying the values of the Minuet's B and C parameters (Section 5.1) on

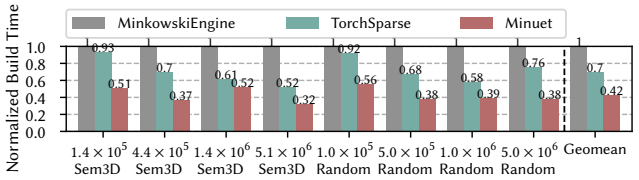

Figure 17. Building time overhead of the source array when varying the real dataset and the number of points in the input point cloud.

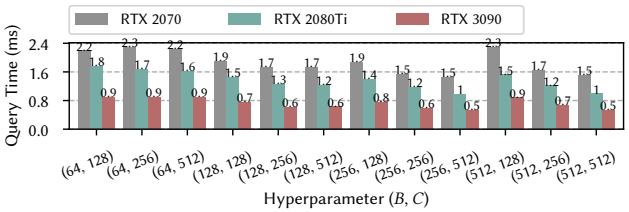

Figure 18. Query time in the source array, when varying the values of Minuet's B and C parameters.

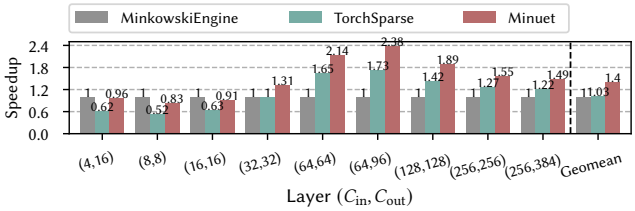

Figure 19. Normalized speedup in the *GMaS* step averaged over all real datasets when varying the input and output channel sizes in *SC* layers.

three different GPU architectures. We observe that the best-performing B and C values are not significantly affected by the GPU architecture characteristics, and we choose $B = 256$ and $C = 512$ as Minuet's default values, since they always provide the best performance on all GPU architectures.

6.5 Performance of the *GMaS* Step

Figure 19 compares performance of all *SC* engines in the *GMaS* step in different *SC* layer configurations, i.e., varying the number of input and output channels. First, Minuet on average outperforms prior *SC* engines. Across all different layer configurations, Minuet achieves on average 1.40× speedup (up to 2.38×) and 1.37× (up to 1.59×) over MinkowskiEngine and TorchSparse, respectively. This is because Minuet tunes the tile size on-the-fly and reduces the padding overheads in GEMM operations. Our evaluations show that TorchSparse incurs on average 11% padding overhead and launches on average 11.1 GEMM kernels, while Minuet has 8.2% padding overhead and launches 7.76 GEMM kernels. Second, we observe that Minuet's *GMaS* step performs slightly worse than MinkowskiEngine (up to 17% slowdown) due to its dedicated optimizations for small channels [43]. Overall, we conclude that Minuet's optimizations in *GMaS* step effectively reduce unnecessary data accesses and computations.

7 Related Work

Minuet is the first work that accelerates *SC* execution on GPUs by (i) proposing a memory-efficient kernel map building, that highly utilizes the on-chip memory hierarchy of GPUs, and (ii) reducing redundant data accesses in *GMaS* step via a batched scheme for the metadata indexing of the input/output feature vectors and a sorted grouping of GEMM operations. We briefly discuss prior work.

***SC* Engines.** Only a few prior works [8, 9, 43] improve the performance of *SC* execution. MinkowskiEngine [8] is the first work that proposes a generalized *SC* to process point clouds and provides an open-sourced *SC* library. SpConv [9] improves the performance of *SC* by leveraging data locality in GEMM operations. TorchSparse [43] is the latest optimized *SC* engine that achieves high system performance by padding and grouping the GEMM operations to improve computation regularity. Our evaluations demonstrate that Minuet significantly outperforms the prior state-of-the-art *SC* engines [8, 43] by effectively reducing expensive memory accesses in *Map* step and redundant data accesses in *GMaS* step. Minuet is also the first work that optimizes the *Map* step in *SC* by highly utilizing the on-chip memory hierarchy on GPUs. Finally, PointAcc [30] proposes a hardware accelerator for point cloud analytics, while Minuet is a software-based *SC* engine tailored to modern GPU architectures.

Concurrent to the submission of this work, PCEngine [24] and TorchSparse++ [45] propose to adaptively select dataflows [8, 9, 43] for *SC* execution in the *GMaS* step. Minuet is orthogonal to these works. (1) In *Map* step, PCEngine and TorchSparse++ rely on hash tables for building kernel maps, and thus still suffer from expensive data accesses (Shortcoming #1). (2) In *GMaS* step, PCEngine and TorchSparse++ still use a fixed tile size in Gather/Scatter operators, thus they suffer from either high indexing costs or limited execution parallelism (Shortcoming #2), when the Gather-GEMM-Scatter dataflow is selected for *SC* execution. PCEngine compresses the kernel map tables [24] and in turn reduces redundant iterations on weight offsets (line 4 in Algorithm 1), which is orthogonal to Minuet, since Minuet reduces redundant iterations on feature tiles (line 1 in Algorithm 1). Thus, we conclude that Minuet's proposed segmented sorted double-traversed binary search and autotuned Gather/Scatter can be applied synergistically with these works to achieve significantly high system performance.

Deep Learning Compilers. Deep Learning (DL) compilers [6, 13, 15, 28, 48, 49] simplify DL programming and automate the hyperparameter search for DL tensor programs, resulting in significant engineering savings. However, most DL compilers either optimize dense tensor algebra [6, 13, 15] or sparse tensor algebra with sparsity patterns that do not depend on the input data [6]. Since the sparsity pattern of *SC* networks depends on the particular characteristics of the given 3D point clouds, the tensor programs compiled by these prior DL compilers [6, 13, 15] are still inefficient for point cloud networks. To our knowledge, TACO [28], TACO-UCF [48], and SparseTIR [49] are the only DL compilers that optimize sparse tensor algebra by taking into consideration the sparse pattern specified by each particular input data. However, these DL compilers do not integrate the optimizations proposed in Minuet. Thus, Minuet's four key ideas work synergistically with these DL compilers to provide significant system performance benefits in *SC* executions.

Binary Search Optimizations on GPUs. A couple of prior works [20, 25, 37] explore binary search optimizations on GPUs. TriCore [25] discusses the cache friendly behavior of naïve binary search, when executing lookups with sorted queries. However, as discussed in Section 5.1, building the kernel map in *SC* by simply executing fully sorted queries in the source array with naïve binary search would incur large sorting overheads (Challenge 1), that would offset the cache benefits, and does not explore the optimization on the number of comparisons (Challenge 2). MergePath [20, 37] improves the computational complexity of naïve binary search, however it necessitates a cache-unfriendly binary search process on GPUs, thus causing worse system performance than the naïve binary search [25]. We conclude that applying these prior binary search-based schemes in the *Map* step would still be inefficient compared to our proposed segmented sorting double-traversed binary search algorithm.

8 Conclusion

Minuet is a highly efficient *SC* engine that accelerates 3D point cloud networks on GPUs. Minuet highly utilizes the on-chip GPU memory hierarchy, improves execution parallelism and metadata costs, and reduces unnecessary data accesses and computations on *SC* executions. Our evaluations show that Minuet significantly outperforms prior state-of-the-art *SC* engines by 1.74× speedup on average at the end-to-end execution, across a wide variety of sparse point cloud networks, datasets, and GPU architectures. We conclude that Minuet is a novel memory-efficient *SC* engine tailored for modern GPUs, and hope that our work encourages further comprehensive studies and optimization strategies on point cloud networks and other sparse deep learning networks.

Acknowledgments

We thank our shepherd, Somali Chaterji, and EuroSys 2024 anonymous reviewers for invaluable feedback and comments to improve our paper. We thank Qidong Su, Chenhao Jiang, Yaoyao Ding, Bojian Zheng, Sankeerth Durvasula, and all members from the UofT EcoSystem research group for the technical discussions and feedback on this paper. This paper is supported by Vector Institute Research grants, the Canada Foundation for Innovation JELF grant, NSERC Discovery grant, AWS Machine Learning Research Award, Facebook Faculty Research Award, Google Scholar Research Award, and VMware Early Career Faculty Grant.

References

[1] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Real-time parallel hashing on the gpu. *ACM Trans. Graph.*, 28(5):1–9, dec 2009.

[2] Dan A. Alcantara, Vasily Volkov, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, and Nina Amenta. Chapter 4 - building an efficient hash table on the gpu. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 39–53. Morgan Kaufmann, Boston, 2012.

[3] Iro Armeni, Ozan Sener, Amir R. Zamir, Helen Jiang, Ioannis Brilakis, Martin Fischer, and Silvio Savarese. 3d semantic parsing of large-scale indoor spaces. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[4] Jens Behley, Martin Garbade, Andres Milioto, Jan Quenzel, Sven Behnke, Cyrill Stachniss, and Juergen Gall. A dataset for semantic segmentation of point cloud sequences. *CoRR*, abs/1904.01416, 2019.

[5] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. ShapeNet: An Information-Rich 3D Model Repository. Technical Report arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.

[7] Yukang Chen, Jianhui Liu, Xiangyu Zhang, Xiaojuan Qi, and Jiaya Jia. Largekernel3d: Scaling up kernels in 3d sparse cnns. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 13488–13498, June 2023.

[8] Christopher B. Choy, JunYoung Gwak, and Silvio Savarese. 4d spatio-temporal convnets: Minkowski convolutional neural networks. *CoRR*, abs/1904.08755, 2019.

[9] Spconv Contributors. Spconv: Spatially sparse convolution library. https://github.com/traveller59/spconv, 2022.

[10] NVIDIA Corporation. CUB: Main Page — nvlabs.github.io. https://nvlabs.github.io/cub/index.html. [Accessed 09-May-2023].

[11] NVIDIA Developer. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute. [Accessed 01-Nov-2023].

[12] NVIDIA Developer. cuBLAS — developer.nvidia.com. https://developer.nvidia.com/cublas, Apr 2021. [Accessed 09-May-2023].

[13] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 370–384, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Wei Dong, Yixing Lao, Michael Kaess, and Vladlen Koltun. ASH: A modern framework for parallel spatial hashing in 3d perception. *CoRR*, abs/2110.00511, 2021.

[15] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. Tensorir: An abstraction for automatic tensorized program optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 804–817, New York, NY, USA, 2023. Association for Computing Machinery.

[16] Claude Flener, Matti Vaaja, Anttoni Jaakkola, Anssi Krooks, Harri Kaartinen, Antero Kukko, Elina Kasvi, Hannu Hyyppä, Juha Hyyppä, and Petteri Alho. Seamless mapping of river channels at high resolution using mobile lidar and uav-photography. *Remote Sensing*, 5(12):6382–6407, 2013.

[17] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. In *Proc. ACM Meas. Anal. Comput. Syst.*, 2022.

[18] Benjamin Graham, Martin Engelcke, and Laurens van der Maaten. 3d semantic segmentation with submanifold sparse convolutional networks. *CVPR*, 2018.

[19] Benjamin Graham and Laurens van der Maaten. Submanifold sparse convolutional networks. *arXiv preprint arXiv:1706.01307*, 2017.

[20] Oded Green, Robert McColl, and David A. Bader. Gpu merge path: A gpu merging algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, page 331–340, New York, NY, USA, 2012. Association for Computing Machinery.

[21] Yulan Guo, Hanyun Wang, Qingyong Hu, Hao Liu, Li Liu, and Mohammed Bennamoun. Deep learning for 3d point clouds: A survey. *CoRR*, abs/1912.12033, 2019.

[22] Timo Hackel, N. Savinov, L. Ladicky, Jan D. Wegner, K. Schindler, and M. Pollefeys. SEMANTIC3D.NET: A new large-scale point cloud classification benchmark. In *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, volume IV-1-W1, pages 91–98, 2017.

[23] Lei Han, Tian Zheng, Lan Xu, and Lu Fang. Occuseg: Occupancy-aware 3d instance segmentation. *CoRR*, abs/2003.06537, 2020.

[24] Ke Hong, Zhongming Yu, Guohao Dai, Xinhao Yang, Yaoxiu Lian, Ningyi Xu, and Yu Wang. Exploiting hardware utilization and adaptive dataflow for efficient sparse convolution in 3d point clouds. *Proceedings of Machine Learning and Systems*, 5, 2023.

[25] Yang Hu, Hang Liu, and H. Howie Huang. Tricore: Parallel triangle counting on gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182, 2018.

[26] Binh-Son Hua, Minh-Khoi Tran, and Sai-Kit Yeung. Point-wise convolutional neural network. *CoRR*, abs/1712.05245, 2017.

[27] Pileun Kim, Jingdao Chen, and Yong K. Cho. Slam-driven robotic mapping and registration of 3d point clouds. *Automation in Construction*, 89:38–48, 2018.

[28] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[29] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic horizontal fusion for GPU kernels. *CoRR*, abs/2007.01277, 2020.

[30] Yujun Lin, Zhekai Zhang, Haotian Tang, Hanrui Wang, and Song Han. Pointacc: Efficient point cloud accelerator. *CoRR*, abs/2110.07600, 2021.

[31] Zhijian Liu, Haotian Tang, Yujun Lin, and Song Han. Point-voxel CNN for efficient 3d deep learning. *CoRR*, abs/1907.03739, 2019.

[32] Zhijian Liu, Xinyu Yang, Haotian Tang, Shang Yang, and Song Han. Flatformer: Flattened window attention for efficient point cloud transformer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1200–1211, June 2023.

[33] Jiageng Mao, Yujing Xue, Minzhe Niu, Haoyue Bai, Jiashi Feng, Xiaodan Liang, Hang Xu, and Chunjing Xu. Voxel transformer for 3d object detection. *CoRR*, abs/2109.02497, 2021.

[34] Daniel Maturana and Sebastian Scherer. Voxnet: A 3d convolutional neural network for real-time object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 922–928, Sep. 2015.

[35] Alexey Nekrasov, Jonas Schult, Or Litany, Bastian Leibe, and Francis Engelmann. Mix3d: Out-of-context data augmentation for 3d scenes. *CoRR*, abs/2110.02210, 2021.

[36] NVIDIA. CUDA Runtime API :: CUDA Toolkit Documentation — docs.nvidia.com. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html. [Accessed 18-May-2023].

[37] Saher Odeh, Oded Green, Zahi Mwassi, Oz Shmueli, and Yitzhak Birk. Merge path - parallel merging made simple. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1611–1618, 2012.

[38] Xuran Pan, Zhuofan Xia, Shiji Song, Li Erran Li, and Gao Huang. 3d object detection with pointformer. *CoRR*, abs/2012.11409, 2020.

[39] Charles Ruizhongtai Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *CoRR*, abs/1612.00593, 2016.

[40] Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *CoRR*, abs/1706.02413, 2017.

[41] Manolis Savva, Angel X. Chang, and Pat Hanrahan. Semantically-Enriched 3D Models for Common-sense Knowledge. *CVPR 2015 Workshop on Functionality, Physics, Intentionality and Causality*, 2015.

[42] Naman Sharma and Hocksoon Lim. 3d-fct: Simultaneous 3d object detection and tracking using feature correlation. *CoRR*, abs/2110.02531, 2021.

[43] Haotian Tang, Zhijian Liu, Xiuyu Li, Yujun Lin, and Song Han. Torchsparse: Efficient point cloud inference engine. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 302–315, 2022.

[44] Haotian Tang, Zhijian Liu, Shengyu Zhao, Yujun Lin, Ji Lin, Hanrui Wang, and Song Han. Searching efficient 3d architectures with sparse point-voxel convolution. *CoRR*, abs/2007.16100, 2020.

[45] Haotian Tang, Shang Yang, Zhijian Liu, Ke Hong, Zhongming Yu, Xiuyu Li, Guohao Dai, Yu Wang, and Song Han. Torchsparse++: Efficient point cloud engine. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 202–209, June 2023.

[46] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, page 344–350, USA, 2010. IEEE Computer Society.

[47] Florian Wirth, Jannik Quehl, Jeffrey Ota, and Christoph Stiller. Pointatme: Efficient 3d point cloud labeling in virtual reality. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 1693–1698, 2019.

[48] Jaeyeon Won, Changwan Hong, Charith Mendis, Joel Emer, and Saman Amarasinghe. Unified convolution framework: A compiler-based approach to support sparse convolutions. *Proceedings of Machine Learning and Systems*, 5, 2023.

[49] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 660–678, New York, NY, USA, 2023. Association for Computing Machinery.

[50] Tianwei Yin, Xingyi Zhou, and Philipp Krahenbuhl. Center-based 3d object detection and tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 11784–11793, June 2021.

[51] Dimitris Zermas, Izzat Izzat, and Nikolaos Papanikolopoulos. Fast segmentation of 3d point clouds: A paradigm on lidar data for autonomous vehicle applications. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5067–5073, 2017.

[52] Zhaoliang Zheng, Thomas R. Bewley, and Falko Kuester. Point cloud-based target-oriented 3d path planning for uavs. In *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 790–798, 2020.

A Extended Results

A.1 Minuet's Best Performing Tile Size

Figure 20 presents the best-performing tile size in Gather and Scatter operations of different *SC* layers of the MinkUNet42 [8] network on various GPUs and datasets, respectively.

We draw two findings. First, we observe that the best-performing tile size significantly varies across different datasets and GPU architectures. This key finding justifies the necessity to tune this parameter in Gather and Scatter for each dataset and GPU architecture. Second, we find that the best-performing tile size also varies across different layers of the network, which indicates that the tile size needs to be reconfigured for each *SC* layer *separately*. In contrast, prior works use a fixed tile size (i.e., 4) in all cases (i.e., *SC* layers, input dataset and GPU architecture), and thus they are still quite inefficient compared to Minuet. We conclude that Minuet's autotuning strategy for tile size enables high system efficiency on various datasets and GPU architectures, and provides a versatile solution to variable layer characteristics of different point cloud networks.

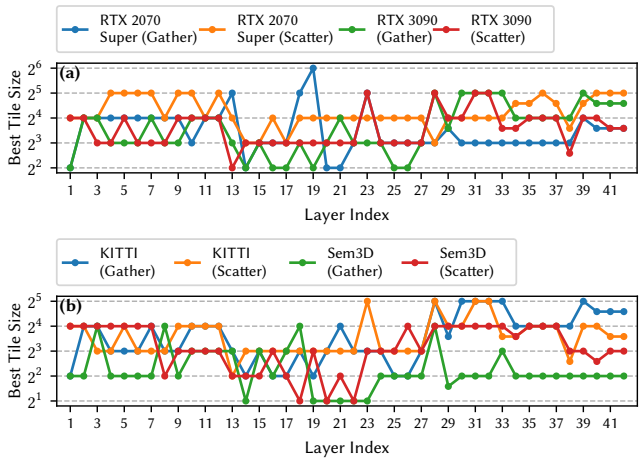

Figure 20. Best-performing tile size in Gather and Scatter operations of each of the 42 layers in the MinkUNet42 network [8] on different (a) GPU architectures and (b) datasets.

B Artifact Appendix

B.1 Abstract

This artifact provides Minuet's source codes, scripts, and instructions for reproducing our main evaluations in this paper. We provide a `README.md` file in this artifact to describe the software & hardware requirements, and step-by-step instructions to reproduce the main experiments for our major claims (Section B.4.1). We expect to take 2 – 3 hours to finish this artifact excluding the dataset downloading time.

B.2 Description & Requirements

B.2.1 How to Access. The artifact is publicly accessible both at GitHub repository `MinuetArtifacts` and on Zenodo with DOI number `10.5281/zenodo.8393982`.

B.2.2 Hardware Dependencies. The artifacts should run on any hardware platforms with a modern NVIDIA desktop/server GPUs (with ≥ 8 GB GPU memory), x86-64 CPUs, sufficient CPU memory (≥ 32 GB), and storage (≥ 150 GB). For reference, our experiments are mainly conducted with the following hardware specifications.

- CPU: AMD Ryzen Threadripper 2920X
- GPU: NVIDIA GeForce RTX 3090 (TDP: 350W)
- Memory: 64 GB DDR4 RAM
- Storage: 256 GB Solid-State Drive (SSD)

B.2.3 Software Dependencies. For this artifact, we require a Linux-based operating system with a up-to-date version of NVIDIA Driver installed. For reference, we use the following software platform for our experiments:

- OS: Ubuntu 20.04.5 LTS
- Linux Kernel: 5.15.0-82-generic
- NVIDIA Driver: 535.104.05

The step-by-step instructions for installing all other software dependencies are detailed in the `README.md` file in the artifact.

B.2.4 Benchmarks. We require to download the Semantic3D dataset [22] as the minimal requirement to run all experiments in this artifact. The other datasets are optional but is necessary for obtaining all results. The instructions for downloading the datasets are detailed in the `README.md` file in the artifact.

B.3 Set-Up

To build and install this artifact, simply clone the artifact repository and follow all instructions in the `README.md` file. The installation of Minuet and all baselines [8, 43], together with their configurations, will be handled automatically in the docker image building script.

B.4 Evaluation Workflow

B.4.1 Major Claims

- *(C1): Minuet achieves a superior speedup over prior state-of-the-art* SC *engines by on average* 1.74× *for both end-to-end point cloud network executions (E1) and by on average* 1.66× *for layerwise* SC *executions (E2).*
- *(C2): Minuet's novel segmented sorting double-traversed binary search algorithm has a higher cache hit ratio and achieves a superior speedup by* 15.8× *on average over prior state-of-the-art works, while introducing comparable building time overhead (E3).*
- *(C3): Minuet proposes auto-tuning tile sizes in Gather and Scatter operators and GEMM reordering, which accelerates the* GMaS *step by on average* 1.39× *over prior* SC *engines (E4).*

B.4.2 Experiments. Please follow the `README.md` file in the artifact for detailed instructions for the experiments. Each experiment (*E1-E4*) clearly marks the major claim code (*C1*, *C2*, or *C3*) and generates similar figures in this paper to support these claims.