Accelerating Sparse Matrix-Matrix Multiplication with the Ascend AI Core

Salli Moustafa
Huawei Technologies Duesseldorf GmbH, Germany
salli.moustafa@huawei.com

***Abstract*—Sparse Matrix-Matrix Multiplication (SpMM) is an important kernel in many applications including Machine Learning workloads, especially for Graph Neural Networks. In this work, we are exploring optimization strategies to improve the performance of this kernel on the Ascend AI processor. We present a custom implementation of the SpMM kernel targeting the dense matrix-matrix multiplication unit inside the Ascend AI Core. Our implementation includes the following optimizations: multi-level tiling allowing to process arbitrarily very large sparse matrices; efficient scheduling allowing an efficient utilization of the AI Cores, regardless of the sparse matrix shape. Thanks to these optimizations, our implementation achieves a speed-up of** 52 **over the original implementation. In addition, the memory footprint of the custom implementation is** 10% **lower.**

***Index Terms*—Sparse Matrix-Matrix Multiplication, AI, Ascend, DaVinci, Graph Neural Networks**

I. Introduction

In recent years, there is a growing interest in Graph Neural Networks (GNNs) [1], [2] for various applications including social networks analysis, recommendation systems or drug design. However, training a GNN model is computationally very demanding especially due to the sparse structure of the data: the Features and Adjacency matrices have very few (an order of 1%) non-zero entries. Typically, the most computationally intensive operator for GNN training is the Sparse Matrix-Matrix Multiplication (SpMM) kernel. For instance, the profiling of the VGAE model training on the Ascend 910 processor showed that the SpMM kernel represents 73% of the total training time. Furthermore, the default implementation of this kernel on Ascend runs on the AI CPU and does not leverage the computational power brought by the AI Core. Indeed, Ascend is primarily optimized for Convolutional Neural Networks (CNNs). Nevertheless, in this work, we are exploring custom optimization strategies to enable Ascend for GNNs by focusing on the SpMM kernel. In the following, the default implementation of the SpMM operator is referred as built-in operator and the optimized version is referred as custom operator. We make the following contributions:

- Design of a multi-level tiling algorithm enabling the execution of the SpMM kernel on dense matrix-matrix multiplication units, regardless of the sparse matrix size thanks to the on-the-fly ”densification” algorithm;
- Efficient workload scheduling strategy enabling full utilization of the AI Cores.

The remaining of this paper is organized as following: Section II discusses the recent advances related to the SpMM acceleration; Section III presents our custom operator and Section IV its performance evaluation; Section V concludes the paper.

II. Related Work

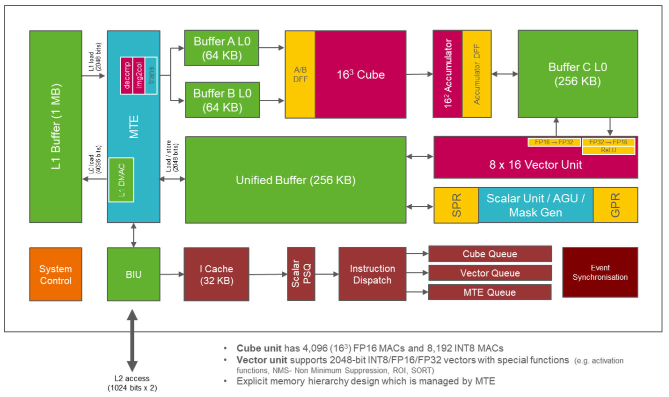
The optimization of SpMM kernel for general-purpose processors and accelerators are heavily studied within the community for decades. In this section, we provide the recent advances in this field with a focus on hardware accelerators.

In [3], the authors proposed new optimization techniques for the SpMM kernel targeting CUDA cores. The optimizations include: coalesced memory accesses, bank conflict avoidance and arithmetic intensity improvement. Their implementation achieves a speed-up of up to 8 over cuSPARSE on selected matrices. In [4], the authors proposed the first algorithm that efficiently leverage GPU Tensor Core Units for spGEMM ($A \times A$, where A is a sparse square matrix) by using a custom bitmap format for storing the sparse matrix. Their implementation achieves a speed-up of 3.12 over cuSPARSE. In [5], the authors developed a new SpMM algorithm specifically designed for sparse matrices featuring a moderate level of sparsity, typical of deep learning applications. With their approach, the performance of sparse computations is better than when using cuBLAS, at as low as 71% of sparsity. In [6], the authors proposed a custom framework leveraging Tensor Core Units for GNN. Thanks to a new sparse graph translation technique, their implementation achieves a speed-up of 1.7 over Deep Graph Library framework.

III. Custom Implementation of SpMM for the DaVinci Core

A. Introduction to the Ascend Computing Platform

1) Ascend Hardware Architecture: The Ascend AI processor is the Huawei’s domain-specific architecture tailored for AI workloads. It is available in different SoCs suitable for various scenarios including training and inference of AI models. The training chip, Ascend 910, integrates HBM2 and special computational units (AI Core and AI CPU). The vast majority of the computing power of the chip is provided by the AI Core, based on the so-called DaVinci architecture ([7], [8]) and presented on Fig. 1. It features three computational units: a Cube Unit, capable to perform multiplications of two 16×16 dense matrices in float16 with a single instruction; the Vector Unit which is 2048 bits wide and can therefore process 128 float16 elements in SIMD mode; the Scalar Unit, used for



- **Cube unit** has 4,096 (16^3) FP16 MACs and 8,192 INT8 MACs
- **Vector unit** supports 2048-bit INT8/FP16/FP32 vectors with special functions (e.g. activation functions, NMS- Non Minimum Suppression, ROI, SORT)
- Explicit memory hierarchy design which is managed by MTE

Fig. 1. DaVinci core architecture [7].

scalar-related operations. The AI CPU is an ARM CPU that is used for executing non-performance critical operators.

To improve the built-in SpMM performance on AI CPU, one could consider using optimized BLAS libraries such as OpenBLAS. However, Given that the AI Core provides the vast majority of the Ascend computing power, we are primarily interested in optimizing the SpMM kernel for the AI Core.

2) Developing Operators for the Ascend AI processor: AI Core operators are developed using one of the following custom frameworks[1]: Tensor Boost Engine (TBE), Tensor Iterator Kernel (TIK) and AI CPU operators are developed in C++. TBE, an extension of TVM [9], provides a DSL interface for writing the operator computing logic and an Auto Schedule mechanism to automatically complete the operator scheduling, data tiling and data streaming. The vast majority of operators can be developed using TBE DSL. However, for complex operators that cannot be easily expressed as a DSL, then TIK framework must be used instead. TIK is Python framework that enables flexible operator development and provides primitives to manually control data movement and parallelization over the AI Cores. Our custom operator is developed using TIK.

B. General Algorithm for SpMM on DaVinci Core

Let us consider:

- A a sparse matrix of shape (M, K) stored in the COO format [10] on the device memory,
- B a dense matrix of shape (K, N) stored on the device memory.

A is identified by its array of indices I_{A} of shape $(nnz, 2)$ and its array of values V_{A} of shape $(nnz,)$. In the following, we assume that the data type of the input matrices is float16 and the data type of the output matrix is float32. Indeed, the tile-based approach we are adopting (see Section III-D) requires accumulating the output matrix through the group of accumulators which operate in float32.

[1]Custom operator development using TBE DSL and TIK are presented in https://www.hiascend.com/doc_center/source/en/CANNCommunityEdition/51RC1alphaX/opdevg/tbedevg/tbedevg-cann51RC1alphaX_01.pdf.

Operand	Tensor Shape	Tensor Scope	Data Type	Comment
A_{dense}	(K_1, M, K_0)	L1	float16	$K_1 = \left\lceil \frac{K}{K_0} \right\rceil$
B	(K_1, N, K_0)	L1	float16	
C	(N_1, M, N_0)	L0C	float32	$N_1 = \left\lceil \frac{N}{N_0} \right\rceil$

TABLE I
REQUIREMENTS FOR THE MATMUL OPERATOR.

The main idea of the algorithm is to firstly convert the sparse matrix into a dense matrix A_{dense} ("densification" step) and then secondly use the built-in MatMul operator, which is already running on the AI Core, to perform the matrix multiplication on the Cube Unit. The densification step is performed on the AI Core using both the Vector Unit and the Scalar Unit through the Unified Buffer (UB). Considering the limited size of the UB, the densification step is performed on-the-fly without having to fully load the sparse matrix at once. This step will be further detailed in Section III-C.

The inputs of the MatMul operator must be in L1 buffer. Thus, we need to move A_{dense} and B to L1 before calling MatMul. In addition, the shapes of the input and output arguments of that operator are specifically defined as in Table I, and M, N and K need to be rounded up as in Equation 1.

$$M = \left\lceil \frac{M}{M_0} \right\rceil \times M_0, N = N_1 \times N_0, K = K_1 \times K_0 \tag{1}$$

where $M_0 = 16$, $N_0 = 16$ and $K_0 = 16$ are constants linked to the DaVinci architecture. The output of the MatMul operator is in L0C and therefore must be moved to the global memory once the operator is fully executed.

C. Converting the Sparse Matrix into a Dense Matrix

The densification step is described in Algorithm 1. The input

Algorithm 1: Converting a Sparse Matrix into a Dense Matrix on AI Core.

```
   Input: $I_{\text{A}}$, $V_{\text{A}}$
   Output: $A_{\text{dense}}^{\text{L1}}$
1  $A_{\text{dense}}^{\text{L1}} = \text{tik.Tensor}((K_1, M, K_0),\ \text{L1})$;
2  $A_{\text{dense}}^{\text{UB}} = \text{tik.Tensor}((K_1, M, K_0),\ \text{UB})$;
3  $A_{\text{dense}}^{\text{UB}} = 0$;
4  $I_{\text{A}}^{\text{UB}} = \text{tik.Tensor}((nnz, 2),\ \text{UB})$;
5  $V_{\text{A}}^{\text{UB}} = \text{tik.Tensor}((nnz, ),\ \text{UB})$;
6  Copy $I_{\text{A}}$ to $I_{\text{A}}^{\text{UB}}$ and $V_{\text{A}}$ to $V_{\text{A}}^{\text{UB}}$;
7  for $v_{id} \in \{0, \cdots, nnz - 1\}$ do
8      $(i, j) = I_{\text{A}}^{\text{UB}}[v_{id}, :]$;
9      $A_{\text{dense}}^{\text{UB}}[i, j] = V_{\text{A}}^{\text{UB}}[v_{id}]$;
10 end
11 Copy $A_{\text{dense}}^{\text{UB}}$ to $A_{\text{dense}}^{\text{L1}}$;
```

of the algorithm is a sparse matrix A represented by I_{A} and V_{A} stored in device memory. The output is a dense matrix stored in L1 ($A_{\text{dense}}^{\text{L1}}$). The algorithm starts by creating the output tensor in L1 (Line 1) and a temporary tensor of the same shape in UB (Line 2) which is fully initialized with zeros (Line 3). The temporary tensor is required since the Vector Unit and

Parameter	Explanation
M	Number of rows of A
N	Number of columns of B
K	Number of columns of A
T_i	Tile size along the 1st dimension of A
T_j	Tile size along the 2nd dimension of B
T_k	Tile size along the 2nd dimension of A
N_i^T	Number of tiles along the 1st dimension of A
N_j^T	Number of tiles along the 2nd dimension of B
N_k^T	Number of tiles along the 2nd dimension of A
N^T	Total number of tiles in A
$\mathcal{T}$	Tiling configuration
$nnz_{\max}^{\text{UB}}$	Maximum number of non-zero values per tile in UB
N_c	Number of AI Cores in use
N_c^T	Maximum number of tiles per AI Cores
N_c^L	Number of fully-loaded AI Cores (each AI Core processes exactly N_c^T rows of tiles)
$\mathcal{S}_{c_{id}}$	The scheduling configuration for the AI Core of index c_{id}

TABLE II
PARAMETERS OF THE TILING

the Scalar Unit cannot access L1; they can only process data in UB. Similarly, I_{A} and V_{A} are replicated in UB (Lines 4–6). Afterwards, the algorithm iterates over all non-zero values and set each of them at their respective location according to their indices (Lines 7–10). Finally, the dense matrix in UB is copied to L1 (Line 11). It is worth noting that filling the dense matrix by setting its individual values involves the Scalar Unit. Considering the limited computational power of this unit, the filling process must be optimized accordingly. We will discuss this situation and the performance implications in Section IV.

To limit the host memory usage and due to the UB capacity limitation, large sparse matrices are never fully densified. Instead, we rely on tiling to perform the densification on-the-fly, hence enabling block-wise matrix-matrix multiplications.

D. Operator Tiling and Scheduling

The tiling has two main goals: enable multi-core parallelization and allow processing of large matrices. In the following, we define a tile as a 2-d slice of a matrix. Given a tiling, each AI Core core will work on a set of tiles of A according to a well defined scheduling. The notations we will use in the following are given in Table II.

1) Operator Tiling: The input matrices are sliced to enhance parallelization and data reuse in caches. For the custom SpMM operator, we follow a similar approach and we introduce a second level of tiling to handle the cases where the number of non-zero values in the sparse matrix is very large. The two levels are illustrated on Fig. 2 and described below.

1) The first level is the same as tiling a dense matrix: the matrices are sliced into tiles according to the tile sizes T_i, T_j and T_k. One should note that since A is a sparse matrix, the first level of tiling is just conceptual because the matrix is never fully constructed in memory. The tile sizes are statically defined to maximize data reuse in L0A and L0B buffers as discussed in Section IV.
2) The second level is added for tiling I_{A}^{UB} and V_{A}^{UB}. Since the sparsity profile of A is unknown at operator compilation time, I_{A}^{UB} and V_{A}^{UB} are statically allocated with a maximum size of $nnz_{\max}^{\text{UB}}$. Then, we associate to each tile of A the set of indices corresponding to the tile's non-zero values. We call this association a tiling configuration $\mathcal{T}$. It is a 2-d tensor of shape $(N^T, 2)$ evaluated as in Equation 2.

$$\begin{aligned}\mathcal{T}[l,0] &= \alpha(l) \\ \mathcal{T}[l,1] &= \beta(l)\end{aligned}, \quad l \in \{0, \cdots, N^T-1\} \tag{2}$$

$\alpha(l)$ is the offset to the first non-zero value within I_{A}, evaluated as in Equation 3; $\beta(l)$ is the effective length of the tile defined as the number of non-zero values within that tile.

$$\begin{aligned}\mathcal{T}[0,0] &= 0 \\ \mathcal{T}[l+1,0] &= \mathcal{T}[l,0] + \mathcal{T}[l,1]\end{aligned}, \quad l \in \{0, \cdots, N^T-1\} \tag{3}$$

Since the data is moved in bursts of 32 Bytes from global memory to UB, the offset and length for the last tile are padded accordingly when $(\beta(N^T-1) \times 2)\%32 \neq 0$. When filling the non-zero values for a tile of index l, we have to distinguish two situations.

a) $\mathcal{T}[l,1] > nnz_{\max}^{\text{UB}}$ This situation typically happens either if the sparsity of A is relatively low or if the non-zero distribution is not uniform leading to a concentration of the non-zero values over a small set of tiles. In this case, we load the non-zeros $\lceil \mathcal{T}[l,1]/nnz_{\max}^{\text{UB}} \rceil$ times.
b) $\mathcal{T}[l,1] \leq nnz_{\max}^{\text{UB}}$ In this case, a single load is sufficient to get all the current tile's non-zeros.

The tiling configuration is evaluated on the host system for the following reasons. Firstly, this mapping involves complex sequential calculations so that if executed on AI Core will heavily involve the low-power Scalar Unit. Secondly, the sparse matrix as input of GNNs is usually constant and its tiling needs to be evaluated only once.

Using the tiling configuration, tiles not containing at least 1 non-zero value (empty tiles) are skipped: this is a performance improvement which can be substantial in case the non-zero values in the sparse matrix are grouped into dense blocks.

2) Operator Scheduling: To parallelize the computations over the AI Cores, a scheduling must be defined to determine what tiles will be given to each AI Core for processing. An efficient scheduling will ensure that: firstly, the cores are busy enough to minimize idle stages; secondly the cores get in average the same amount of workload. An imbalance can lead to a poor parallel scalability.

There are several parallelization strategies in the context of tile-based matrix multiplication. The matrix A can be split in rows (1st dimension), in columns (2nd dimension) or both. The same applies for the matrix B. However, since our custom operator is mainly targeted for Deep Learning workloads, there is an important point that needs to be considered. Indeed, according to the research presented in [5], sparse matrices from Deep Learning computations have on average $2.3\times$

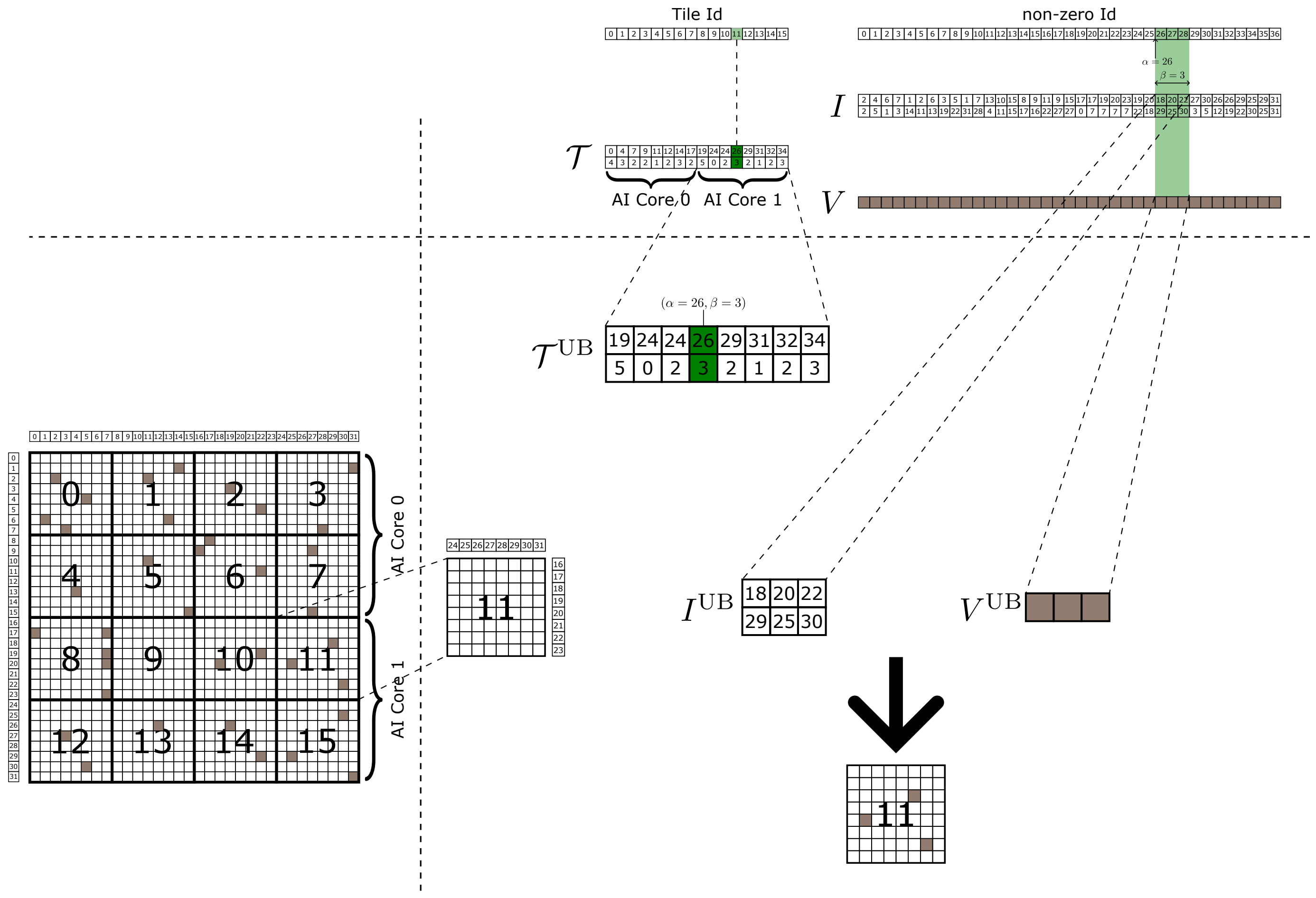


Fig. 2. Tiling procedure of the custom SpMM operator.

longer rows than sparse matrices found in scientific computing. This implies that parallelizing the computations over the 1st dimension of A, will keep the AI Cores sufficiently busy. Therefore, our first constraint on the scheduling is fulfilled. Our approach regarding the second constraint about workload distribution is presented in Algorithm 2. The algorithm takes as input the tile count along the 1st dimension of A (tile-rows), the core count, the current AI Core index[2] and evaluates the start index ($\mathcal{S}_{c_{id}}[0]$) and stop index ($\mathcal{S}_{c_{id}}[1]$) of the set of rows belonging to corresponding AI Core. If $N_i^T \% N_c = 0$, then each AI Core will get exactly the same amount of tile-rows given by N_c^T (Line 1). Otherwise, we first evaluate N_c^L, the maximum number of cores that can be fully loaded with the same amount of tile-rows (Line 2), then we derive the remaining tile-rows R (Line 3) to be processed by the core of index N_c^L. The cores of index larger than N_c^L will be idle.

The proposed scheduling strategy is more efficient than the greedy scheduling. Indeed, in this case, the cores for which

[2]Here, it would be more accurate to use the expression "block index" instead of "core index". Indeed, the for_range(<start>, <stop>, <block_count>) construct of TIK framework operates on blocks. If the block count exceeds the number of available AI Cores, the execution will be scheduled in batches over the cores.

Algorithm 2: Scheduling of the Custom SpMM Operator over AI Cores.

```
Input: $N_i^T$, $N_c$, $c_{id}$
Output: $\mathcal{S}_{c_{id}}$
1  $N_c^T = \lceil N_i^T / N_c \rceil$;
2  $N_c^L = N_i^T / N_c^T$;
3  $R = N_i^T - N_c^L \times N_c^T$;
4  if $c_{id} < N_c^L$ then
5      $\mathcal{S}_{c_{id}}[0] = c_{id} \times N_c^T$;
6      $\mathcal{S}_{c_{id}}[1] = \mathcal{S}_{c_{id}}[0] + N_c^T$;
7  else if $c_{id} == N_c^L$ then
8      $\mathcal{S}_{c_{id}}[0] = N_c^L \times N_c^T$;
9      $\mathcal{S}_{c_{id}}[1] = \mathcal{S}_{c_{id}}[0] + R$;
10 else
11     $\mathcal{S}_{c_{id}}[0] = 0$;
12     $\mathcal{S}_{c_{id}}[1] = 0$;
```

$c_{id} < N_c - 1$ will get the same amount of tile-rows (N_i^T / N_c), and the last core of index $N_c - 1$ will get $L = N_i^T \% N_c$ tile-rows which could be very large. For instance, with $M = 10000$ and $T_i = 128$ we have $N_i^T = 79$; using $N_c = 32$, the cores of index from 0 to 30 will process 2 tile-rows and the core of index 31 will process 14 tile-rows, leading to a dramatic workload imbalance since the last core will be the bottleneck as it will run in average 7 times longer than the other cores to complete its workload. Conversely, with our proposed scheduling strategy, the cores of index from 0 to 25 will process 3 tile-rows, the core of index 26 will process 1 tile-row and cores from 27 to 32 will be idle. The workload is therefore much more balanced between the cores.

E. Full Algorithm for SpMM on DaVinci Core

The full version of the custom operator is presented in Algorithm 3. After completing the multiplication of the tile-row of index ii of A with the tile-column of index jj of B (Lines 9–13) on the Cube Unit, the output is copied from L0C into a Global Memory (GM) workspace tensor $\mathcal{W}$ before moving it to the final output C (Line 14). For each AI Core, we allocate a workspace tensor of shape (T_i, T_j) in GM so that the cores can work in parallel and asynchronously.

Algorithm 3: Full Algorithm for the SpMM Operator targeting AI Core.

```
   Input: $A$, $B$
   Output: $C = A \times B$
1  $N_c^T = \lceil N_i^T / N_c \rceil$;
2  $N_c^L = N_i^T / N_c^T$;
3  $R = N_i^T - N_c^L \times N_c^T$;
4  foreach $c_{id} \in \{0, \cdots, N_c - 1\}$ do
5      Lines 4–12 of Algorithm 2 ($\mathcal{S}_{c_{id}}$);
6      for $ii \in [\mathcal{S}_{c_{id}}[0], \mathcal{S}_{c_{id}}[1] - 1]$ do
7          for $jj \in [0, N_j^T - 1]$ do
8              $C_{\text{dense}}^{\text{L0C}}$ = tik.Tensor($(N_1, M, N_0)$, L0C);
9              for $kk \in [0, N_k^T - 1]$ do
10                 Lines 1–11 of Algorithm 1 ($A_{\text{dense}}^{\text{L1}}$);
11                 Copy $B[kk, jj]$ to $B_{\text{dense}}^{\text{L1}}$;
12                 $C_{\text{dense}}^{\text{L0C}}$ += $A_{\text{dense}}^{\text{L1}} \times B_{\text{dense}}^{\text{L1}}$;
13             end
14             Copy $C_{\text{dense}}^{\text{L0C}}$ to $C[ii, jj]$;
15         end
16     end
17 end
```

It is worth mentioning that the proposed custom SpMM algorithm could be implemented on other accelerators like GPUs featuring Tensor Cores. However, in this paper we only focused on the Ascend processor.

F. Memory Requirements

In this section, we theoretically evaluate the custom operator memory requirements in terms of UB and GM occupancy.

1) Usage of the Unified Buffer: The UB usage of the custom operator is given in Table III. We can notice that the sizes of all the UB tensors are fixed except for the tiling configuration tensor which is dependent on N_c^T. To avoid exceeding the UB capacity when processing very large

Tensor	Rows	Cols	Data Size (Bytes)	Tensor Size (Bytes)
A^{UB}	T_i	T_k	2	$2 \times T_i \times T_k$
B^{UB}	T_k	T_j	2	$2 \times T_k \times T_j$
$\mathcal{T}$	N_c^T	2	8	$16 \times N_c^T$
I_{A}	$nnz_{\max}^{\text{UB}}$	2	8	$16 \times nnz_{\max}^{\text{UB}}$
V_{A}	$nnz_{\max}^{\text{UB}}$	1	2	$2 \times nnz_{\max}^{\text{UB}}$
$\mathcal{S}_{c_{id}}$	2	1	4	8

TABLE III
UB USAGE BY THE CUSTOM SPMM OPERATOR.

Operator	Indices		Values		Total Size
	Count	Total Size (Bytes)	Count	Total Size (Bytes)	(Bytes)
SpMM (AI CPU)	$nnz \times 2$	$nnz \times 16$	nnz	$nnz \times 4$	$nnz \times 20$
SpMM (AI Core)	$nnz \times 2$	$nnz \times 16$	nnz	$nnz \times 2$	$nnz \times 18$

TABLE IV
USAGE OF THE GM BY THE BUILT-IN SPMM AND BY THE CUSTOM SPMM OPERATORS FOR REPRESENTING THE SPARSE MATRIX.

matrices, we can launch more blocks (N_c) which in turn lower N_c^T. As an example, let us consider $M = K = 525825$, $N = 128$, $T_i = 128$, $T_k = 256$, $T_j = 128$, $nnz = 2100225$, and $nnz_{\max}^{\text{UB}} = 4096$. In this case, the UB occupancy when $N_c = 32$ is 4.3 MB, whereas when $N_c = 8192$ it is 220 KB. Hence, with this strategy, we can process arbitrarily very large matrices without exceeding the UB capacity.

2) Usage of the Global Memory: To further characterize the memory requirement for the custom operator, we did a theoretical comparison of the GM required for representing and processing the sparse matrix for both the built-in SpMM and custom SpMM operators. The result is summarized in Table IV. We notice that the AI CPU version of SpMM requires $nnz \times 20$ Bytes whereas the AI Core version requires $nnz \times 18$ Bytes which is 10% lower than the former.

IV. PERFORMANCE ANALYSIS

This section discusses the performance of the custom SpMM operator on Ascend 910.

A. Datasets

We considered two different datasets for the benchmarking.

1) Features and Adjacency matrices from CoraFull dataset[3] as described in Table V.
2) SuiteSparse [11]: it is a collection of 2893 sparse matrices collected from various applications in different domains (scientific computing, machine learning, electronic circuit simulation, ...). The collection is widely

Matrix Name	M	K	nnz	Sparsity
Features	18712	8710	1071300	0.9935
Adjacency	18712	18712	143560	0.9995

TABLE V
FEATURE AND ADJACENCY MATRICES FROM CORAFULL DATASET.

[3]CoraFull is a graph dataset suitable for node classification tasks and typically used for training GNNs in this context.

Matrix Name	M	K	nnz	Sparsity
mc2depi	525825	525825	2100225	0.999992
cage12	130228	130228	2032536	0.999880
dawson5	51537	51537	1010777	0.999619
lock1074	1074	1074	51588	0.955276
patents_main	240547	240547	560943	0.999990
struct3	53570	53570	1173694	0.999591
wiki-Vote	8297	8297	103689	0.998494
bcsstk30	28924	28924	2043492	0.997557
nemeth21	9506	9506	1173746	0.987011
pcrystk03	24696	24696	1751178	0.997129
pct20stif	52329	52329	2698463	0.999015
pkustk06	43164	43164	2571768	0.998620
pli	22695	22695	1350309	0.997378
net50	16320	16320	945200	0.996451
web-NotreDame	325729	325729	1497134	0.999986

TABLE VI
SUITESPARSE MATRICES USED FOR BENCHMARKING.

used for studying the performance of sparse kernels. For our benchmarking, we selected 15 square sparse matrices used in [4] and described in Table VI.

B. Benchmarking Setup

Hardware Configuration: Our benchmarking is carried-out on an Atlas 800-9000. The host system is equipped with 4 Kunpeng 920 processors and a total of 1 TB of DDR4 memory. 8 Ascend 910 processors are attached to the system; each having 32 GB of HBM2.

Best Tile Sizes: To execute the computations on the Cube Unit, the MatMul operator reads inputs in L0A and L0B which are 64 KB large each. The tile sizes are selected to maximize the occupation of L0A and L0B, hence minimizing latencies incurred by global memory accesses. Therefore, for data of float16 type, the tile sizes for the matrix A and B are: $T_i = 128$, $T_j = 128$ and $T_k = 256$. All the benchmarking that will follow have been conducted using these tile sizes.

We will use $nnz_{\max}^{\mathrm{UB}} = 4096$ and $N_c = 8192$ for all the experiments.

C. Accuracy Verification of the Custom SpMM Operator

To verify the custom SpMM accuracy, we compared its output against that of the built-in operator, using randomly generated matrices of different sparsity: 0.9968, 0.9936, 0.9872, 0.9744, 0.9488, 0.8976 and 0.7952. The metric we used for this comparison is the relative residual as in Equation 4.

$$\Delta = \frac{\|C_{\text{AI CPU}} - C_{\text{AI Core}}\|_F}{\|C_{\text{AI CPU}}\|_F} \tag{4}$$

For each sparse matrix A, we generate the matching dense matrix B where N is fixed to 128. All matrices are filled with random values generated from a uniform distribution $[0, 1]$. The relative residual on our selected matrices are given on Fig. 3. We notice that the relative residual for the custom SpMM operator is below 3×10^{-4}, for all sparsity values and for all matrix sizes. This is an acceptable accuracy and is typical to what is achievable by mixed-precision MAC arrays. For instance, in [12], the authors showed that the relative residual for the dense matrix-multiplication on Tensor Cores is between 10^{-4} and 10^{-3}.

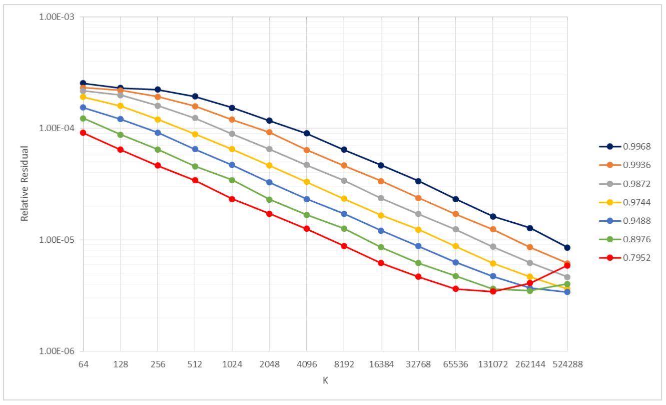


Fig. 3. Accuracy verification of the custom SpMM operator for different sparsity values and matrix sizes; $M = N = 128$. Input matrices are filled with random values generated from a uniform distribution $[0, 1]$. The relative residual is evaluated using Equation 4.

Furthermore, the custom operator relative residual decreases for sparse matrices with lower sparsity. Indeed, when the sparsity is high, the operator involves a large proportion of multiplications and additions with 0s in the sparse matrix due to the fact that the sparse matrix is densified with 0s before the processing on the Cube Unit, as explained in Section III-C. In addition of the wasted computational power, the ineffectual computations with 0s introduce more rounding errors than in the built-in operator: the higher the sparsity, the more rounding errors are accumulated. Finally, for a fixed sparsity, the relative residual is lower for larger values of K suggesting that the error $\|C_{\text{AI CPU}} - C_{\text{AI Core}}\|_F$ grows more slowly than $\|C_{\text{AI CPU}}\|_F$.

The accuracy could be improved using mixed-precision techniques. For instance in [12], the authors proposed a novel-algorithm for effectively recovering single-precision accuracy of the dense matrix-matrix multiplication while performing computations on GPU Tensor Cores in half-precision.

D. Benchmarking the Custom SpMM Operator on Ascend 910

In this section, we discuss the custom operator performances on Ascend 910 using our two datasets. All time measurements are averaged over 100 runs. When evaluating the custom operator speed-up relative to the built-in operator, we only consider the operator execution time without the preprocessing time (evaluation of the tiling configuration on the host). To have an idea, the preprocessing time of the Features matrix of the CoraFull dataset is 0.7s using a single CPU core. However, in typical training or inference of a GNN, the input sparse matrix is constant and the corresponding tiling configuration can be evaluated once. Therefore, for a training that takes several iterations, the preprocessing cost could be amortized.

CoraFull: on Fig. 4, we present the speed-up obtained by the custom operator using this dataset. The chart shows that the custom operator is significantly faster than the built-in operator and the speed-up reaches 52.51 on the Features matrix. For the Adjacency matrix, the speed-up is lower and reaches 15.52. Indeed, since the Adjacency matrix is more sparse than the

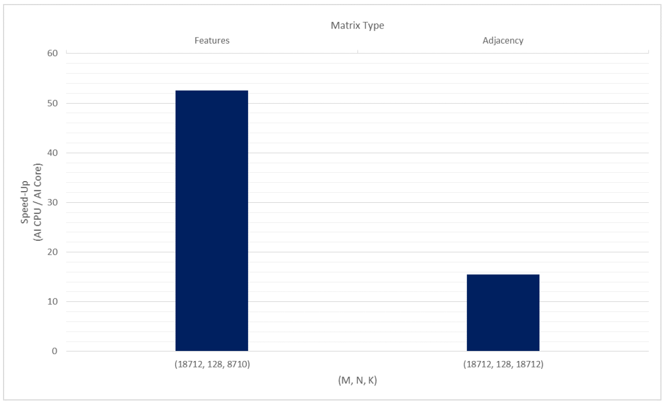


Fig. 4. Speed-up of the custom SpMM operator over the built-in version on Features and Adjacency matrices of CoraFull dataset. N is fixed to 128. The benchmark is executed on Ascend 910.

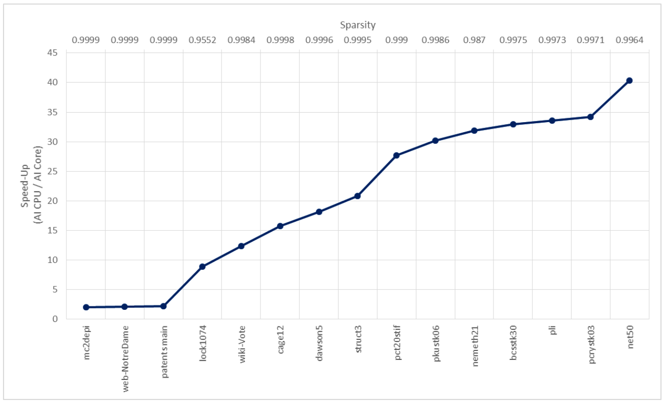


Fig. 5. Speed-up of the custom SpMM operator over the built-in version on selected SuiteSparse matrices presented in Table VI. N is fixed to 128. The benchmark is executed on Ascend 910.

Features matrix, as shown in Table V, the custom operator will have to execute more ineffectual computations when processing the Adjacency matrix than that of the Features matrix. This situation is expected by design: the sparsity has a little impact on the custom operator performance because the sparse matrix is converted into a dense matrix (according to a tile-based approach) before calling the MatMul operator. To further increase the custom operator performance on highly sparse matrices, a reordering technique could be applied so that the dense tiles contain a lower amount of 0s.

SuiteSparse: The custom operator performance on the selected SuiteSparse matrices is given on Fig. 5. As observed for the CoraFull matrices, we can see here that the speed-up is larger on matrices with lower sparsity, regardless of the matrix sizes. For instance, the speed-up reached with the matrix lock1074 ($M = K = 1074$) is higher than the speed-up reached with the matrix mc2depi ($M = K = 525825$).

E. Profiling the Custom SpMM Operator on Ascend 910

To identify the bottlenecks of the custom operator, we profiled its execution using CoraFull matrices. We collected metrics highlighting to the functional units utilization, and

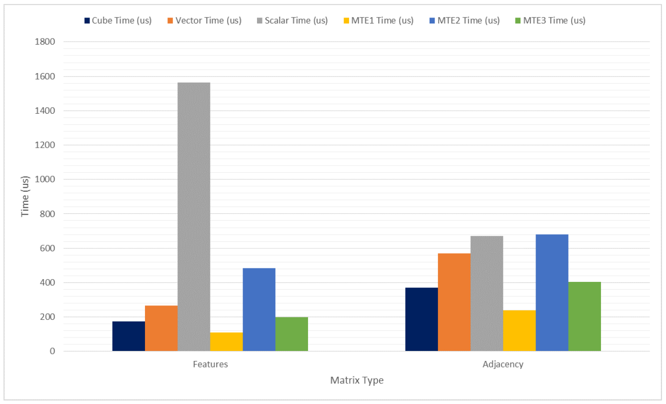


Fig. 6. AI Core resource utilization of the custom operator using CoraFull matrices on Ascend 910. MTE1: L1 $\rightarrow$ L0A/L0B; MTE2: HBM $\rightarrow$ AI Core; MTE3: AI Core $\rightarrow$ HBM.

bandwidth between memory units (HBM, L1, UB, ...). In this section, we provide the analysis of the profiling data.

AI Core Resource Utilization: The pipe utilization is given on Fig. 6. It is worth noting that besides the time required by the Scalar Unit, all other time measurements are higher when processing the Adjacency matrix despite the lower nnz of the latter. For instance, the required time for data loading from the device main memory (HBM) to the AI Core is larger when processing the Adjacency matrix because the dense matrix, B, is larger in this case: its shape is $(18712, 128)$ when processing the Adjacency matrix and $(8710, 128)$ when processing the Features matrix.

The Vector Unit and Scalar Unit are used for preparing the input tiles as part of the densification algorithm: the larger the nnz, the higher their utilizations as shown in the Fig. 6. As expected, Scalar and Vector processing times are top bottlenecks for both the Features and Adjacency matrices. This is because of the densification algorithm as explained previously. Furthermore, the Scalar Unit time for processing the Features matrix is significantly larger than for the Adjacency matrix. Indeed, as Features matrix has larger nnz than Adjacency matrix, the serial initialization of non-zero values in dense tiles requires more time. Finally, the Cube Unit utilization is higher because the sparse matrix densification introduces more floating point operations, whose a significant amount are ineffectual multiplications with 0s.

Memory Bandwidth: The memory bandwidth utilization is given on Fig. 7. Here, we clearly see that because of the dense-based processing of the custom operator, there is more data movement involved when processing the Adjacency matrix than for the Features matrix. In addition, L1 Write BW and UB Read BW are higher for both matrix types because the dense input tiles, required by the MatMul operator, are moved from UB to L1.

For both matrix types, the HBM Write BW is low because the output matrix C is relatively small compared to dense matrix B. For instance, in case of the Adjacency matrix, C has a shape of $(18712, 128)$ whereas B has a shape of

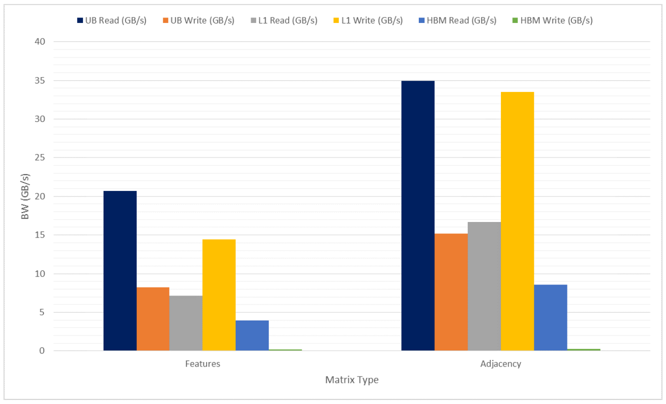


Fig. 7. Memory bandwidth utilization of the custom operator using CoraFull matrices on Ascend 910.

$(18712, 18712)$ that is $146\times$ larger than C.

From this profiling analysis, we made the following conclusions about the main bottlenecks of the custom operator:

- The Vector Unit and Scalar Unit are significantly used during the preparation of the input tiles;
- The UB Read BW and L1 Write BW are relatively high.

Therefore to further optimize our implementation, we propose the following optimizations directions.

- Implement an efficient re-ordering of the sparse matrix so that: the Cube Unit is less utilized; the UB Read BW and L1 Write BW are lower because the non-zero values will be distributed into a fewer amount of tiles.
- Reduce the Scalar and Vector times by optimizing the densification algorithm.

V. Conclusion

In this work, we presented an optimized implementation of the SpMM kernel for the Ascend AI Core, leveraging the Cube Unit. Thanks to advanced optimization techniques such as multi-level tiling and efficient scheduling on AI Core, our implementation achieves a speed-up of 52.51 over the original AI CPU implementation using Features matrices of the CoraFull dataset. In addition, the memory footprint of our custom implementation is 10% lower. To improve the custom operator performance for very sparse matrices (sparsity > 0.9999), we could re-order the sparse matrix so that the non-zeros are grouped into a relatively few dense tiles. This way, we could discard all tiles without non-zero values away from the partial matrix multiplications. Furthermore, we are investigating optimization strategies to minimize the usage of Scalar Unit in the densification step.

Acknowledgment

We thank Heiko Schick and Luka Stanisic for their comments and discussions on this work. We would like also to thank the reviewers for their valuable feedback on this manuscript.

References

[1] M. Gori, G. Monfardini, and F. Scarselli, "A New Model for Learning in Graph Domains," in *Proceedings. 2005 IEEE international joint conference on neural networks*, vol. 2, 2005, pp. 729–734.

[2] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A Comprehensive Survey on Graph Neural Networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[3] S. Shi, Q. Wang, and X. Chu, "Efficient Sparse-Dense Matrix-Matrix Multiplication on GPUs Using the Customized Sparse Storage Format," in *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, IEEE, 2020, pp. 19–26.

[4] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares, "Accelerating Sparse Matrix-Matrix Multiplication with GPU Tensor Cores," *Computers & Electrical Engineering*, vol. 88, p. 106 848, 2020.

[5] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse GPU Kernels for Deep Learning," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2020, pp. 1–14.

[6] Y. Wang, B. Feng, and Y. Ding, "TC-GNN: Accelerating Sparse Graph Neural Network Computation Via Dense Tensor Core on GPUs," *arXiv preprint arXiv:2112.02052*, 2021.

[7] H. Liao, J. Tu, J. Xia, and X. Zhou, "DaVinci: A Scalable Architecture for Neural Network Computing," in *2019 IEEE Hot Chips 31 Symposium (HCS)*, 2019, pp. 1–44. DOI: 10.1109/HOTCHIPS.2019.8875654.

[8] X. Liang, *Ascend AI Processor Architecture and Programming: Principles and Applications of CANN*. Elsevier, 2020.

[9] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, *et al.*, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[10] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, 1994.

[11] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011, ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. [Online]. Available: https://doi.org/10.1145/2049662.2049663.

[12] H. Ootomo and R. Yokota, "Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance," *The International Journal of High Performance Computing Applications*, p. 10 943 420 221 090 256, 2022.