Tensor Core Accelerated Database Operations

Akshaya Bindu Gowri
Otto von Guericke University
Magdeburg, Germany
akshaya.bindu@st.ovgu.de

Maneendra Perera
Otto von Guericke University
Magdeburg, Germany
hetti.perera@st.ovgu.de

Vegenshanti Valerian Dsilva
Otto von Guericke University
Magdeburg, Germany
vegenshanti.dsilva@st.ovgu.de

Akshata Balasaheb Bobade
Otto von Guericke University
Magdeburg, Germany
akshata.bobade@st.ovgu.de

Abhishek Digvijay Singh
Otto von Guericke University
Magdeburg, Germany
abhishek1.singh@st.ovgu.de

Ayushi Dinesh Dani
Otto von Guericke University
Magdeburg, Germany
ayushi.dani@st.ovgu.de

***Abstract*—Tensor cores are specialized hardware built by Nvidia. They are capable of matrix multiply and accumulate operations in mixed precision per GPU cycle. They have been used extensively for deep neural network learning tasks because of their massive parallelism and high memory bandwidth. In this work, we use tensor cores to accelerate database operations. Specifically, we express selection and join operations of SQL queries as matrix multiplications and compare its performance with non-tensor core implementation. In our evaluations, we show that time taken for matrix multiplication operations to complete using tensor cores is considerably less than the non-tensor core implementation. For the baseline, we have chosen the GPU implementation of selection and join operation which involves evaluating the selection condition (for selection) or performing loop processing (for join) on GPU. We observed that our implementation got outperformed by a simple GPU implementation.**

***Index Terms*—Tensor cores, NVIDIA, CUDA, Database operations**

I. Introduction

In the current era of Big Data, we have a large amount of data being collected and stored at an immense speed every day. Data is collected from multiple sources such as transactions, social media and electronic devices connected to the Internet. While accessing and processing this data efficiently is a challenge, research on special hardware to manage data has been an ongoing effort [1].

A GPU is a special hardware that was designed mainly for graphical processing. Due to the high level of data parallelism in GPUs, they have also been used to perform ordinary calculations and processing tasks that were generally handled by the CPU. This has led to the rise of GPGPU (general-purpose computing on graphics processing units [1]). Research has been ongoing to implement database operations on GPU using CUDA programming API [3, 15, 13].

[1]https://de.wikipedia.org/wiki/General_Purpose_Computation_on_Graphics_Processing_Unit

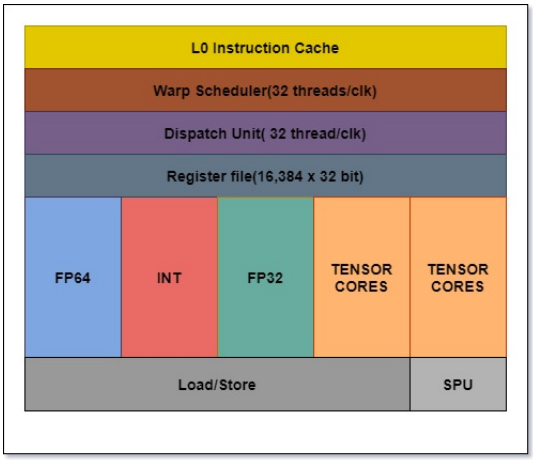

Fig. 1. NVIDIA V100 processing block containing two tensor cores. (Image inspired from NVIDIA CUDA programming guide [7] and Analyzing GPU Tensor Core Potential for Fast Reductions) [5]

The advancement in the field of machine learning has led to the introduction of GPUs equipped with a special architecture that is capable of handling numerous vector calculations at ease. One such unit are the tensor cores [17] developed by NVIDIA that exclusively perform matrix multiplications faster to boost the performance of deep learning applications.

A tensor core performs matrix multiplication of 4×4 FP16 (half-precision floating point values) matrices, followed by a FP32 or FP16 accumulation operation in a single step. Due to mixed-precision computing, it can accelerate the throughput of matrix multiplication as well as maintain the accuracy of the results [17]. This makes it suitable for the heavy and memory intensive computations required in deep learning applications. Currently, tensor cores are offered along with the following GPUs: NVIDIA A100[17], NVIDIA Turing [17] and NVIDIA Volta [17] processors. To understand the benefits of using tensor cores, consider the Volta V100 GPU (shown in

Fig 1). It can provide up to "12× higher peak teraFLOPS (TFLOPS) for training and 6× higher peak TFLOPS for inference over NVIDIA Pascal" [17]. Tensor cores enable an overall 3× speedup in terms of performance time as compared to NVIDIA Pascal [17] for matrix multiplication operations. However, tensor core capabilities for non-machine learning related operations are yet to be fully explored.

Our research aims to exploit tensor cores for database operations and compare the performance against existing implementations [10, 4]. In this paper, we present the implementation of two relational operators namely selection and join on GPU re-written for tensor cores. Presently only GEMM (General matrix multiplication) can be performed on tensor cores. Our proposed approach expresses the selection and join operation as matrix multiplication to ensure compatibility with tensor cores. For the post-processing of the result that we obtain from the matrix multiplication step, we have formulated a bit masking strategy that helps us to compute the results to the desired operation(selection or join). Effectively, our research also tries to broaden the operations that can be performed using tensor cores. The evaluation results for our research show that matrix multiplication using tensor cores are especially beneficial for higher matrix sizes. Although our approach using tensor cores performs better in comparison to the baseline with non-tensor cores, baseline using GPU has emerged to be the winner among all the approaches.

In the remaining sections of the paper, we present to you the background and related work with respect to tensor cores and database operations on tensor cores respectively. Further, we present the baseline approaches we have considered for the evaluation, followed by our proposed approach. In the proposed approach, we first explain the conceptual view of our approach and then present the complete flow for each operation(selection and join). Finally, we present the evaluation results, comparing the baseline approaches to the proposed approach and set the grounds for future work.

II. Background

Traditionally, CPUs handle data processing. However, there have been many purposes that CPUs have not been able to serve - smooth video gaming and fast parallel processing being some of the very primary purposes as they are limited by the number of cores and the number of threads that can be parallely executed in each core [16]. CPUs could process graphics, but not on a very high scale. Around the 1980s, the demand for a separate graphics processing unit started to arise [2]. Eventually, graphic processors by NVIDIA, ATI, and Intel's integrated graphics gained popularity. The parallel processing capabilities of GPUs seemed to cater well to big data processing and machine learning operations. Capabilities of GPUs for relational operations have widely been evaluated resulting mostly in positive outcomes.

The recent rise in deep learning applications demanded even better-performing hardware, and thus, tensor cores [4] were introduced to further accelerate deep learning applications. While GPUs are capable of parallel processing by executing single instruction over multiple cores, tensor cores are capable of executing one operation significantly faster. Each tensor core performs a matrix multiply accumulate (MMA) operation of 4×4 matrices in one GPU clock cycle [4]. This makes the matrix multiplication operation to perform faster in tensor cores than in GPU. We exploit this and convert the database operations to matrix multiplication and gain performance benefits from tensor cores. The same database operations have been explored using GPUs, which is the baseline for comparison in this work [10]. Even though there are a lot of database operations whose performance can be potentially accelerated using tensor cores, this report focuses on the select-where and join operations.

III. Related Work

A tensor processing unit (TPU) is an application-specific integrated circuit (ASIC) which was developed by Google, especially for accelerating neural networks in machine learning [14]. More than the tensor processing unit (TPU), the capabilities of the graphics processing unit have been evaluated more often for accelerating database related operations. One recent work by Holanda and Mühleisen [12] that evaluated tensor processing unit capabilities to speed up database operations, incorporated mapping relational operators to Tensorflow operations that are backed by TPUs. This study resulted in a conclusion that in comparison to GPUs, TPUs don't give a significant speedup for relational query processing. The results of this study were not very lucid as several factors, such as hardware session management, execution plan generation and data transfer were not taken into consideration. The results generated by our study are on the same lines, i.e. normal SM cores of GPUs gave a better performance over tensor cores for select and join operations.

Dakkak et al [9] explored the capabilities of tensor cores to perform matrix multiplication has been evaluated. This study discusses the pros and cons of performing operations in tensor cores, taking advantage of the fact that TPUs are indeed performance boosters, while also considering that only matrix multiplication operations on comparatively smaller matrices are supported. This implies that all kinds of operations have to be eventually converted into some form of matrix multiplication to utilize the performance benefits of tensor cores. In addition to performance-boosting, a decrease in power consumption has also been reported when performing reduction and scan primitives.

Roberto et al, [4], discuss the tensor processing unit's capabilities to perform faster reductions. By "reduction", we imply reducing n elements to a single value(Summation of all values) as a set of m×m MMA tensor core operations. This experiment compared TPU based reduction operations with the classic parallel GPU reduction. There was again a very significant performance improvement observed. However, the amount of precision loss for FP16 has not been quantified.

Join operations have been evaluated mainly using CPUs and GPUs. Research on relational joins on graphic processors [11], takes advantage of the new features of a GPU. A

set of data-parallel primitives have been used to implement indexed or non-indexed nested-loop, sort-merge, and hash join. Operations like a map, scatter/gather, prefix scan, and split are utilized to perform joins. As a result, a performance improvement of around 10% was observed concerning searching for the data in local memory, which is one of the main steps involved in join operations. This was just one part of the complete join operation that showed significant performance improvement. The complete GPU based join operation could achieve a speedup of 2-27X in contrast with its optimized CPU based counterparts.

In this paper, we propose a new approach to efficiently perform database operations using matrix multiplications in tensor cores. Lastly, we compare our approach with the baseline model, which is discussed further.

IV. Baseline approach

In this section we explore our baseline approaches for select-where and join operations.

A. Baseline for select-where operation

Here we perform a select operation for a given input table in GPU. Firstly, we transfer the input table from CPU to the GPU where the tuples satisfying the where clause condition are retrieved. In the GPU, the attribute on which the select operation has to be performed, is given to a kernel. There are three kernels written for equality, less than and greater than predicates. Within the kernel, we search for those tuples which fulfils the select condition with the help of a simple conditional statement. It returns 1 if the corresponding tuple satisfies the search condition else 0. This operation is executed in parallel with the help of GPU threads. After getting the positions of 1s, the corresponding rows of the input table are retrieved to get the result of select where operation which is transferred from GPU to CPU. The conceptual flow for the baseline of select where operation in GPU is given in Fig 7. Apart from this baseline approach, we have taken another baseline which runs the proposed approach without using tensor cores. This can be done by specifying the mode as CUBLAS_DEFAULT_MATH in cublasSetMathMode method of cuBLAS API [6]. The cublasSetMathMode function enables us to choose whether or not to use tensor cores.

B. Baseline for join operation

We perform a nested loop join in GPU. Similar to the select-where operation, the input matrices are taken in the CPU. Secondly, we transfer the input tables to the GPU where the join operation is performed. In the GPU, the attribute on which the join operation has to be performed, is given to a kernel from both the tables. Within the kernel, we perform nested loop join. It returns 1 if the join attribute value is same in both the tables else 0. After getting the positions of 1s, the corresponding rows of the input tables are retrieved to get the result of join operation which is transferred from GPU to CPU. The conceptual flow for the baseline of join operation in GPU is given in Fig 8. Just like the baselines for select where

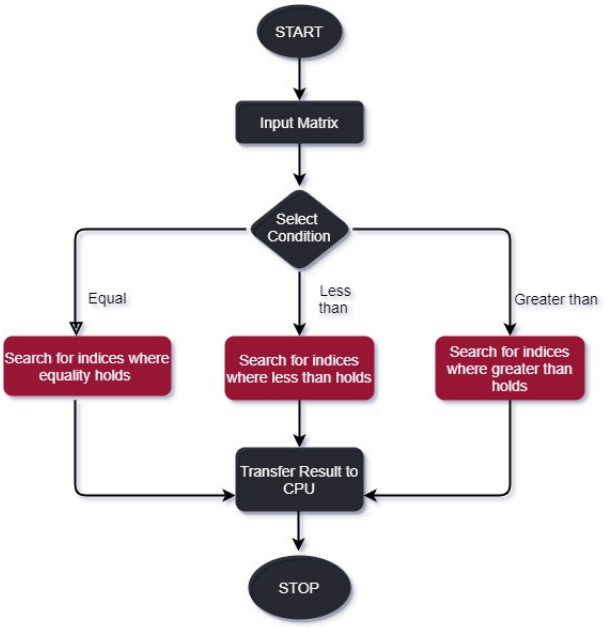

Fig. 2. Select -Where Operation Baseline

operation, we have taken an another baseline which runs the proposed approach without using tensor cores by setting the mode as CUBLAS_DEFAULT_MATH in cublasSetMathMode method of cuBLAS API [6].

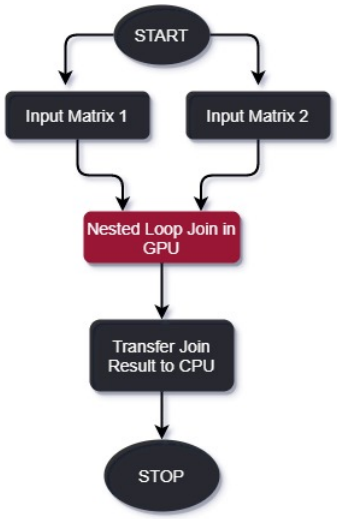

Fig. 3. Join Baseline

V. Proposed Approach

In this section, we will be presenting the proposed approach to perform selection and join operation on tensor cores. Firstly, we will be expressing each operation in the form of matrix multiplication. Next, we will be explaining the bit masking strategy which is applied as a post processing step to the matrix multiplication result to get the bit mask which will be used to compute the results. We then, explain the logic to compute

the results to each operation (selection and join). Finally we present the conceptual flow of our proposed strategy for each operation (selection and join).

A. Select-Where operation

The select-where operation retrieves the tuples from the given input table which satisfies the condition specified in the where clause. In this section, we explain how we express select where operation as a matrix multiplication to perform the operation in tensor cores.

1) Matrix multiplication for Select-Where:

- Step 1: In this step, the input table is represented as an input matrix R. Here the input table consists of 3 columns where each column represents an attribute of a relational table and select operation will be performed on an attribute, for instance we take the first column/ attribute.
  Sample query: select * from R where Col1==2

Table R

Col1	Col2	Col3
1	4	7
2	5	8
3	6	9

Matrix representation of Table R

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

We convert all the columns which are not included in the where clause to 0 i.e. all columns except Col1 are made 0.

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix}$$

- Step 2: Here the condition to be evaluated is taken as condition matrix S, e.g. if the select where condition attribute is equal to 2 then we take the reciprocal of the condition value into condition matrix S.

$$S = \begin{bmatrix} 1/2 & 0 & 0 \\ 1/2 & 0 & 0 \\ 1/2 & 0 & 0 \end{bmatrix}$$

The condition matrix and the input matrix is padded with zeroes to make it compatible for matrix multiplication.In this case, the columns positions 2 and 3 are filled with 0s. Next, we transfer both input and condition matrix from CPU to GPU.

- Step 3: Now we perform the matrix multiplication of R and S matrices using cublasGemmEx() method of cuBLAS [6]. The result of matrix multiplication will contain 1s for every element that satisfies the selection condition and non-ones otherwise. Next, we flip all the non-ones to zeros in parallel. In the result of the bit flipping operation, a 1 indicates a relevant index while a 0 indicates an irrelevant index which will be used for computing the final result of the selection operation.

$$R * S = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/2 & 0 & 0 \\ 1/2 & 0 & 0 \\ 1/2 & 0 & 0 \end{bmatrix}$$

$$R * S = \begin{bmatrix} 1/2 & 0 & 0 \\ 1 & 0 & 0 \\ 3/2 & 0 & 0 \end{bmatrix}$$

2) Bit Masking: After performing the matrix multiplication, the result matrix with ones (result indices) and non-ones (irrelevant indices) is obtained. The goal here is to convert the non-ones to zeroes(as the positions with non-ones are irrelevant to our final results) and obtain a matrix with only ones and zeroes. One approach to convert the non-ones to zeros is by using a conditional statement in a GPU kernel. However, as branching degrades the performance of GPU, we propose a new approach to flip the non-ones to zeros. Based on the condition of selection predicate, we have three ways to flip the non-ones to 0s:

1) Equality: To flip a bit in the result matrix at a specific index, we subtract 1 from the value. Subtracting 1 for relevant indices (i.e. indices containing 1) will result in 0 and for non-relevant indices (i.e. indices containing non-ones) will result in a non-zero element. We then apply abs() function so as to get the absolute value of negative non-zero elements that arise from the subtraction. Next, we perform ceil() function so that non-zero floating point elements between 0 and 1, for example, 0.75 are rounded to 1. Now all the non-relevant indices will have a value greater than or equal to 1 and the relevant indices will have 0s after the ceil() operation. We then perform logical-AND operation of every element with 1 so that at the relevant indices we have 0s and at the non-relevant indices we have 1s. XOR of the result after performing logical AND operation will give 1 for relevant indices and 0 for non-relevant indices.

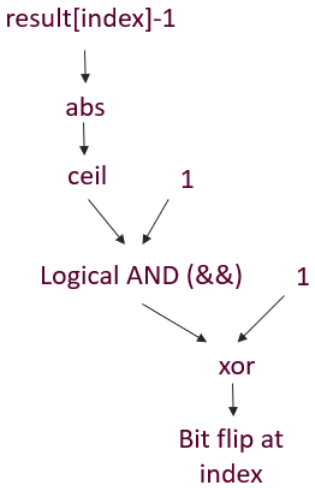

Fig. 4. Bit flip operation for Equality Condition

2) Greater than: Consider the condition greater than 2 for the example explained in subsection *Matrix multiplication for Select-Where*. We will obtain the following result after matrix multiplication for condition greater than 2.

$$R * S = \begin{bmatrix} 1/2 & 0 & 0 \\ 1 & 0 & 0 \\ 3/2 & 0 & 0 \end{bmatrix}$$

So 3/2 (greater than 1) is the desired result while 1 and 1/2 (less than or equal to 1) are not required. For greater than condition, we will be considering the values greater than 1 as relevant and the values less than or equal to one as non-relevant. To flip a bit in the result matrix for the greater than condition, we subtract 1 from each element. To simplify the logic, let's call the result of the subtract operation as X. Subtracting 1 for relevant indices (i.e. indices containing values greater than 1) will result in a value greater than 0 and for non-relevant indices (i.e. indices containing values less than or equal to 1) will result in a value less than or equal to 0. Consider we have Y = X and we apply abs() function on Y so as to get the absolute value of negative non-zero elements that arise from the subtraction. Next, we perform ceil() operation on X and Y which rounds the element to the next biggest integer. Later, we perform addition of the result of ceil() operation on X and the result of abs(), ceil() operations on Y. The addition will result in a value greater than or equal to 1 for relevant elements and for non-relevant elements it will be 0. Finally, we perform logical AND operation to get 1 for relevant indices and 0 for non-relevant indices.

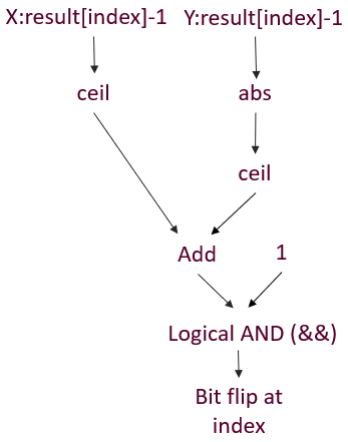

Fig. 5. Bit flip operation for Greater than Condition

3) Less than: For less than condition, we will be considering the values less than 1 as relevant and the values greater than or equal to one as non-relevant. To flip a bit in the result matrix for the less than condition, we subtract 1 from each element. To simplify the logic, let's call the result of the subtract operation as X. Subtracting 1 for relevant indices (i.e. indices containing values less than 1) will result in a value less than 0 and for non-relevant indices (i.e. indices containing values greater than or equal to 1) will result in a value greater than or equal to 0. Consider we have Y = X and we apply abs() function on Y so as to get the absolute value of negative non-zero elements that arise from the subtraction. Next, we perform floor() operation on X and Y which rounds the element to the next biggest integer. Later, we perform subtraction of the result of floor() operation on X and the result of abs(), floor() operations on Y. The subtraction will result in a value less than 0 for relevant elements and for non-relevant elements it will be 0. Finally, we perform logical AND operation to get 1 for relevant indices and 0 for non-relevant indices.

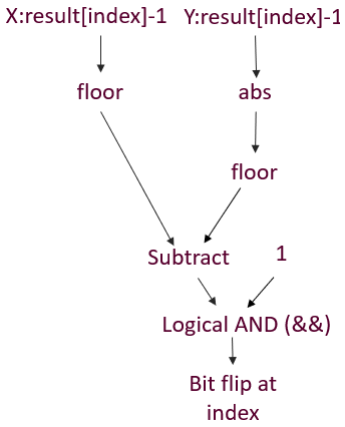

Fig. 6. Bit flip operation for Less than Condition

Descriptions of the symbols and terms used:

- result: The result matrix after matrix multiplication
- ceil: ceil() function that rounds a floating point value to the next highest integer
- abs: abs() function that gives the absolute value of the integer
- floor: floor() function rounds a floating point value to the next lowest integer
- index: Index of the element being considered
- XOR: XOR operation
- &&: Logical AND operation

3) Computing Result Matrix: After getting the positions of 1s in each row, the corresponding rows of R are retrieved to get the result of select where operation which is transferred from GPU to CPU.

$$R * S = \begin{bmatrix} 2 & 5 & 8 \end{bmatrix}$$

Conceptual flow of select where operation

This section describes the conceptual flow for our program which is an overview of the performed operations. Fig 4 shows the flow chart for the select-where operation. The steps executed in CPU and GPU have been marked with grey and pink colours respectively.

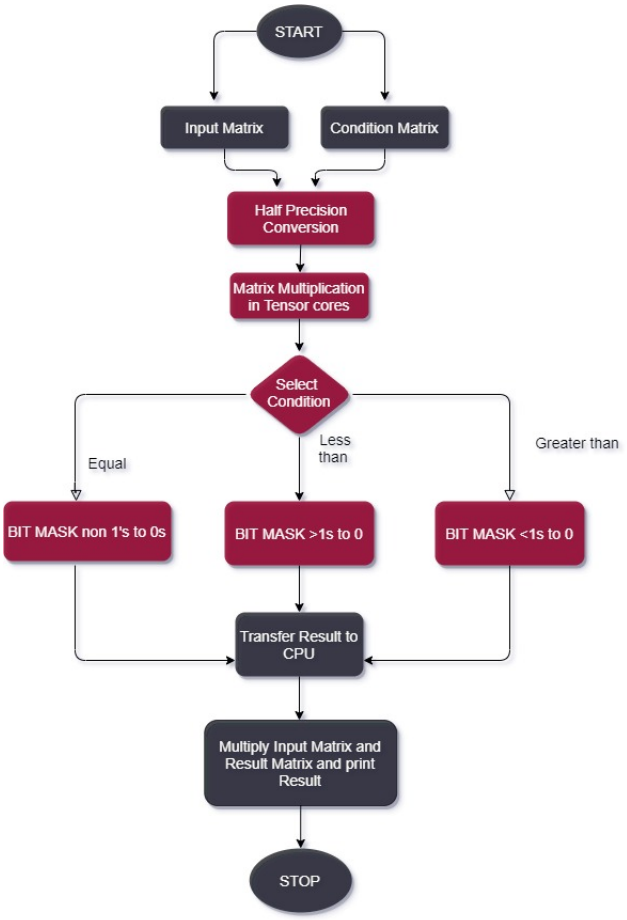

Fig. 7. Select -Where Operation

- For select-where, we create the condition matrix (Condition Matrix contains the reciprocal of the condition value and zeroes to ensure matrix multiplication compatibility)
- Now we transfer the input matrix and condition matrix from CPU to GPU.
- In tensor core we will perform matrix multiplication using the cublasGemmEx method of cuBLAS API [6].
- Bit Masking: The resulting matrix will be converted to a matrix containing only ones and zeroes using the bit flipping logic.
- To obtain the desired rows, we multiply the input matrix with the result matrix and display the result.

B. Join Operation

The join operation gives us the combined results from two tables after matching the key attribute values. In this section, we explain how we express join as matrix multiplication to perform join operation in tensor cores.

1) Matrix Multiplication for Join:

- Step 1: To perform a join we have two tables. The table on the left-hand-side of the join operation will be used as the input matrix R and the table on the right-hand-side of the join operation will be used as the matrix S.

Consider we have to perform join operation on tables R and S. Here R and S are taken as input matrices. The join attribute will be performed using the column 2 of R and column 1 of S.

Table R

Col1	Col2
1	4
2	5
3	6

Table S

Col1	Col2
4	1
6	2

Matrix Representation of R

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Matrix Representation of S

$$\begin{bmatrix} 4 & 1 \\ 6 & 2 \end{bmatrix}$$

In the matrix R, we convert the non-join columns to zeros i.e. we make Col1 to 0.

$$R = \begin{bmatrix} 0 & 4 \\ 0 & 5 \\ 0 & 6 \end{bmatrix}$$

- Step 2: We need to transform the matrix S to make it suitable for matrix multiplication. Firstly, we convert the join column in the second matrix S to the reciprocals of the original values and transpose it. Next, we eliminate non-join columns so that the matrix contains only reciprocals of join attribute values and pad it with zeroes to ensure matrix multiplication compatibility.
In our example, the join column for S has been converted to its reciprocals and transposed. The final matrix S contains only the reciprocals of join column values with additional zeroes as described above.

$$S = \begin{bmatrix} 1/4 & 1 \\ 1/6 & 2 \end{bmatrix}$$

$$S = \begin{bmatrix} 1/4 & 1/6 \\ 1 & 2 \end{bmatrix}$$

$$S = \begin{bmatrix} 0 & 0 & 0 \\ 1/4 & 1/6 & 0 \end{bmatrix}$$

- Step 3: The input matrices R and S are transferred to GPU and matrix multiplication is performed in tensor core using cublasGemmEx method of cuBLAS API [6].

$$R * S = \begin{bmatrix} 0 & 4 \\ 0 & 5 \\ 0 & 6 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 1/4 & 1/6 & 0 \end{bmatrix}$$

$$R * S = \begin{bmatrix} 1 & 4/6 & 0 \\ 5/4 & 5/6 & 0 \\ 6/4 & 1 & 0 \end{bmatrix}$$

- Step 4: After matrix multiplication, the result matrix contains ones indicating the matched join column positions and non-ones. We convert the non-ones to zeroes using

the bit masking logic which is explained in the further section.

$$R * S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

- Step 5: The result matrix is transferred to CPU and the join result is computed (explained in further section).

2) Bit Masking: As we are performing join operation, we use the bit masking logic for the equality operator to convert the non-ones in the matrix to zeroes as shown in Fig 2. Consider the result we got from the matrix multiplication in our example,

$$Result = \begin{bmatrix} 1 & 4/6 & 0 \\ 5/4 & 5/6 & 0 \\ 6/4 & 1 & 0 \end{bmatrix}$$

Let us perform a bit mask on the value present in the first row and first column of result matrix i.e. 1. So substituting 1 in our bit masking logic would yield (ceil(abs(1 -1) && 1) XOR 1. This operation would result in 1.

Similarly, we perform a bit mask on the element at the first row and second column of the result i.e 4/6. So substituting 4/6 in the bit masking logic gives (ceil(abs(4/6 -1) && 1) XOR 1 which outputs 0.

3) Computing results: After the result matrix has been transformed using bit masking, we can use the position of ones in the result matrix to combine the values from the two tables. In our example, we have obtained one in the third row and second column of the result. Hence, our final result will combine the third row from R and second row from S to perform the join operation.

Result of Bit Flipping

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Result of Join operation

$$\begin{bmatrix} 1 & 4 & 1 \\ 3 & 6 & 2 \end{bmatrix}$$

Conceptual Flow of join operation

Fig. 5 describes the conceptual flow for the join operation which is an overview of the performed operations. The steps executed in CPU and GPU are highlighted in grey and pink colors respectively.

- Two tables are taken as inputs and to perform the join operation we create two matrices as explained in subsection *Matrix Multiplication for Join*. To facilitate the matrix multiplication operation, the second matrix is transformed into the required form.
- Since the join is to be performed for only one of the columns, the non-participating columns are eliminated.
- The matrix multiplication of the resulting matrices takes place in the tensor core. The indices resulting in ones indicate matched join column positions.
- Using bit masking, the non-ones are converted to zeroes.
- This matrix containing 0s and 1s is then transferred back to the CPU.
- The position of ones in this matrix can then be used to combine the values from the initial 2 input tables.

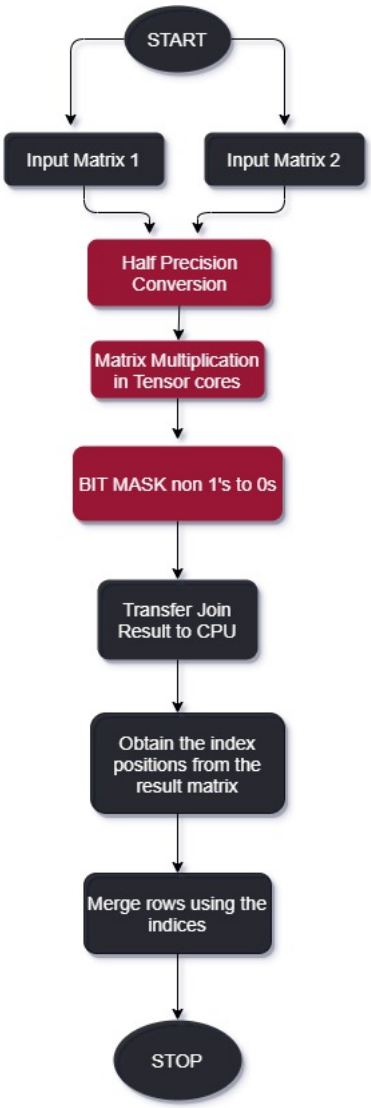

Fig. 8. Join Operation

C. Chunking of Matrix Multiplication

To aid in faster computations of matrix multiplications in tensor cores, we chunk the input matrix. The chunking of input matrix is done only when the input matrix sizes are greater than 2^{10} as there is a steady decrease in performance of matrix multiplication for input sizes greater than 2^{10}. In Algorithm 1, we show the procedure to chunk matrix multiplications for both select-where and join operations.

TABLE I
RUNTIME INFORMATION

GPU	NVIDIA Tesla V100 SXM2 32GB
OS	Ubuntu 18.04
Global Memory size (GPU)	31.75GB
Tensor Cores available	640
NVCC and CUDA Runtime version	9.0.17

Algorithm 1: Algorithm to chunk matrix multiplications

Result: Matrix multiplication of both matrices
1 Transfer input matrices A, B to GPU;
2 MATRIX_SIZE : Dimensions of input matrices;
3 CHUNK_SIZE : 2^{10};
4 **if** *MATRIX_SIZE* $> 2^{10}$ **then**
5 NO_OF_MULTIPLICATIONS = MATRIX_SIZE / CHUNK_SIZE;
6 **for** *i = 1 to NO_OF_MULTIPLICATIONS* **do**
7 start = i*CHUNK_SIZE;
8 end = start + CHUNK_SIZE;
9 CHUNKED_A = MATRIX_A[start : end];
10 CHUNKED_B = MATRIX_B[start : end];
11 cublasGemmEx(CHUNKED_A, CHUNKED_B)
12 **end**
13 **else**
14 cublasGemmEx(A, B);
15 **end**

After the initial transfer of input matrices to GPU, we perform a condition check on the input matrix size as shown in line 4. If it is greater than 2^{10}, we chunk the input matrices in the sizes of 2^{10}. We perform matrix multiplication using cublasGemmEx method of cuBLAS API [6] on these chunks and then perform bit flipping on the result as specified in line 11. At the end, we ensure that the result of these intermediate matrix multiplications are stored in consecutive locations in GPU and finally transferred as whole back to CPU.

VI. EVALUATION

In this section we provide the evaluation results of our proposed approach against the baseline approaches. We have conducted all the experiments in NVIDIA Tesla V100 SXM2 32GB. We use cuBlas [6] library for GEMM operations. We have not considered CPU to GPU transfer time for performance measures. We have used methods cudaMalloc() [8] to allocate bytes size in linear consecutive memory on the device and cudaMemcpy() [8] to copy data between host and device. The details of our runtime is provided in the Table 1.

In order to evaluate the proposed approach, we have considered two baseline approaches for join and select-where operations. One baseline approach performs a simple nested loop join and select where operation in GPU while the other is the implementation of the proposed approach without using tensor cores. The join operation gives us the combined results from two tables after matching the key attribute values. To aid in faster computations of matrix multiplications in tensor cores, we have used our novel approach to chunk the matrix multiplications.

Fig. 9 and 11 shows the time taken for execution of various select and join operations approaches with increasing input table sizes respectively. As the time taken for both select where and join operations without using tensor cores after 2^{12} was drastically increasing, the figures shows only the time taken for executions from 2^4 to 2^{12} for simplicity. We have observed that the join and select where operations using matrix multiplication in tensor cores is atleast **4 times** faster than the join and select where operations executed without using tensor cores for input sizes in between 2^{11} to 2^{15}. Also, the chunking of matrix multiplication has potentially decreased the execution times in both cases with and without using tensor cores. Moreover, the join and select where operations in GPU performed well and the execution times was more or less similar for all the input sizes. The reasoning behind is that GPU has bigger limit for data parallel execution and that is not reached for smaller data sizes. It is then followed by our proposed approach in tensor cores with chunking of matrix multiplication.

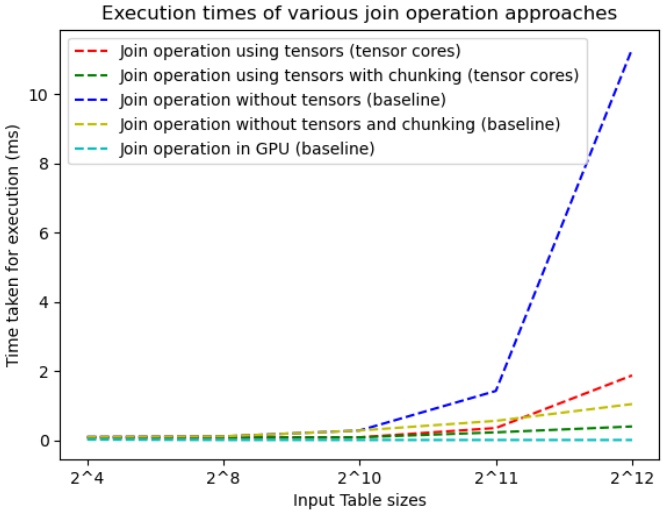

Fig. 9. Time taken for execution of various join operation approaches

Fig 10 and 12 provides the detailed execution times for matrix multiplication followed by bit flipping time for both join and select where operations in tensor cores. The time taken for bit flipping is far less when compared to the time taken for the matrix multiplication and it is almost similar with varying input sizes. It also has to be noted that the time taken for matrix multiplication increases substantially with varying input size.

Experimental results in the evaluation section make tensor cores appear promising to be used for database operations but there are factors that our experiment did not take into consideration like data transfer.

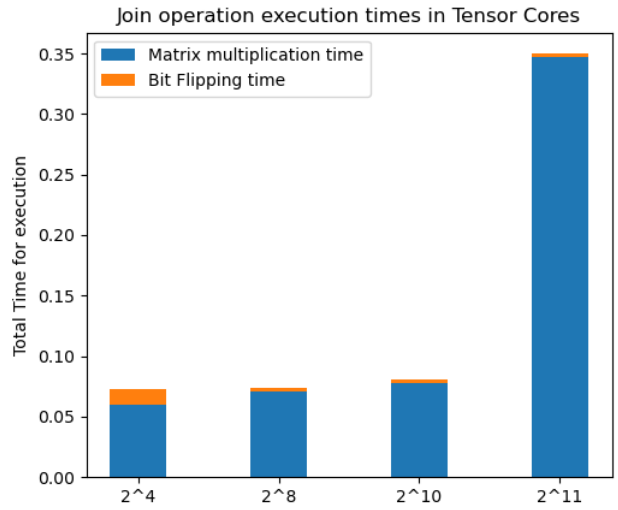

Fig. 10. Time taken in milliseconds for execution of matrix multiplication and bit flipping for join operation in tensor cores

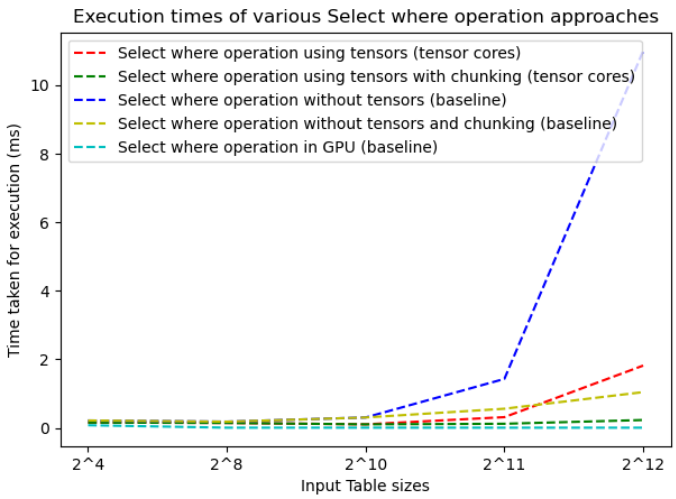

Fig. 11. Time taken for execution of various select-where operation approaches

We observed that the time taken for operations to complete using tensor cores with chunking provides very good execution for large matrix sizes of order greater than 2^{10} when compared to non-tensor core or tensor core approach without chunking but still the plain GPU implementation emerged with a better execution time even for large input sizes.

VII. Conclusion

This paper focuses on using tensor cores for two relational operations: selection, and join. We proposed a simple matrix multiplication approach in tensor cores followed by bit masking for each operation. Furthermore, we compare our approach for the operations with two baselines: 1) matrix multiplication without tensor cores, 2) using non matrix multiplication algorithms for selection and join which runs on GPU without using tensor cores to evaluate our proposed approach. After testing

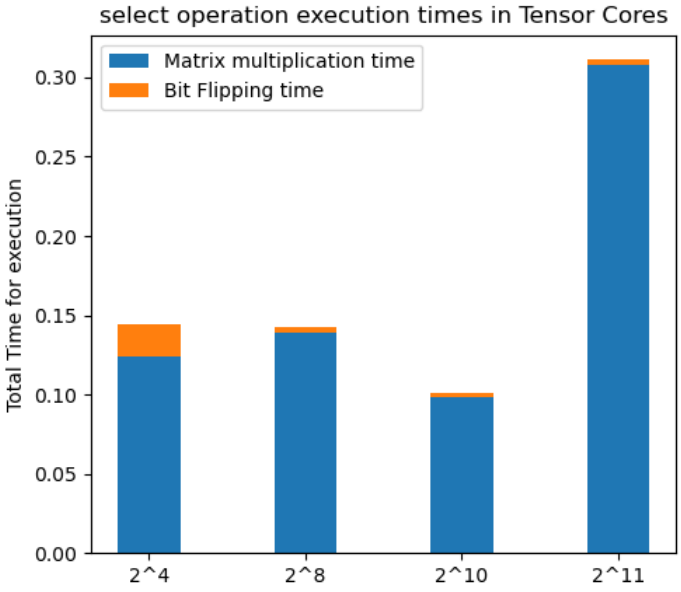

Fig. 12. Time taken in milliseconds for execution of matrix multiplication and bit flipping for select-where operation in tensor cores

and evaluation, a huge performance drop was identified when the input matrix sizes exceed 2^{10} and the chunking algorithm was designed and implemented to overcome this issue.

Finally, evaluation has been performed by running the proposed and baseline approaches in NVIDIA Tesla V100 with 640 tensor cores. In our evaluation, we have shown that both selection and join with tensor cores performed better than the approach without using tensor cores for all input matrix sizes. But still, the plain GPU operation showed better execution time than the operations using tensor cores.

As future work, we propose designing bit masking algorithms that execute in GPU or tensor cores with less execution time than the current implementations. Finally, we are interested in performing in-depth research on how tensor cores can be utilized for other relational operations.

References

[1] E. Babb. “Implementing a relational database by means of specialzed hardware”. In: *ACM Transactions on Database Systems* 4.1 (Mar. 1979), pp. 1–29. DOI: 10.1145/320064.320065.

[2] Priyanka M. Bagul, Saurabh Radheshyam Chiraniya, and Vandana S. Inamdar. “Redefining the role of RC in the field of GPU: A survey”. In: *Proceedings - IEEE International Conference on Information Processing, ICIP 2015*. Institute of Electrical and Electronics Engineers Inc., June 2016, pp. 751–754. DOI: 10.1109/INFOP.2015.7489482.

[3] Peter Bakkum and Kevin Skadron. “Accelerating SQL database operations on a GPU with CUDA”. In: *International Conference on Architectural Support for*

Programming Languages and Operating Systems - ASPLOS. 2010. DOI: 10.1145/1735688.1735706.

[4] R. Carrasco, R. Vega, and C. A. Navarro. "Analyzing GPU Tensor Core Potential for Fast Reductions". In: *Proceedings of the International Conference of the Chilean Computer Science Society. SCCC*. 2018, pp. 1–6.

[5] Roberto Carrasco, Raimundo Vega, and Cristóbal A Navarro. *Analyzing GPU Tensor Core Potential for Fast Reductions*. 2019. arXiv: 1903.03640v1.

[6] *cuBLAS — NVIDIA Developer*. URL: https://developer.nvidia.com/cublas (visited on 07/01/2020).

[7] "CUDA C++ Programming Guide". In: ().

[8] *CUDA Runtime API :: CUDA Toolkit Documentation*. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html (visited on 07/01/2020).

[9] Abdul Dakkak et al. "Accelerating Reduction and Scan Using Tensor Core Units". In: *Proceedings of the International Conference on Supercomputing*. ICS. Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 46–57. DOI: 10.1145/3330345.3331057.

[10] Naga K. Govindaraju et al. "Fast computation of database operations using graphics processors". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 2004, pp. 215–226. DOI: 10.1145/1007568.1007594.

[11] Bingsheng He et al. "Relational Joins on Graphics Processors". In: *Proceedings of the International Conference on Management of Data*. SIGMOD. Vancouver, Canada: Association for Computing Machinery, 2008, pp. 511–524. DOI: 10.1145/1376616.1376670.

[12] Pedro Holanda and Hannes Mühleisen. "Relational Queries with a Tensor Processing Unit". In: *Proceedings of the International Workshop on Data Management on New Hardware*. DaMoN'19. Amsterdam, Netherlands: Association for Computing Machinery, 2019. DOI: 10.1145/3329785.3329932.

[13] Yin Fu Huang and Wei Cheng Chen. "Parallel Query on the In-Memory Database in a CUDA Platform". In: *Proceedings of the International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, 3PGCIC*. 2015. DOI: 10.1109/3PGCIC.2015.34.

[14] Norman P Jouppi et al. *In-Datacenter Performance Analysis of a Tensor Processing Unit TM*. Tech. rep. 2017.

[15] Chia Feng Lin and Shyan Ming Yuan. "The design and evaluation of GPU based memory database". In: *Proceedings of the International Conference on Genetic and Evolutionary Computing, ICGEC*. 2011. DOI: 10.1109/ICGEC.2011.61.

[16] Mark Silberstein. "GPUs: High-performance Accelerators for Parallel Applications". In: *Ubiquity* 2014.August (Aug. 2014), pp. 1–13. DOI: 10.1145/2618401.

[17] *Tensor Cores: Versatility for HPC & AI — NVIDIA*. URL: https://www.nvidia.com/en-us/data-center/tensorcores/ (visited on 06/28/2020).