EasyChair Preprint

№ 6442

# Algorithm Design for Tensor Units

Rezaul Chowdhury, Francesco Silvestri and Flavio Vella

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

August 27, 2021

# Algorithm design for Tensor Units[⋆]

Rezaul Chowdhury[1], Francesco Silvestri[2][0000−0002−9077−9921], and Flavio Vella[3][0000−0002−5676−9228]

[1] Stony Brook University, USA, rezaul@cs.stonybrook.edu
[2] University of Padova, Italy silvestri@dei.unipd.it
[3] Free University of Bozen, Italy flavio.vella@unibz.it

**Abstract.** To respond to the intense computational load of deep neural networks, a plethora of domain-specific architectures have been introduced, such as Google Tensor Processing Units and NVIDIA Tensor Cores. A common feature of these architectures is a hardware circuit for efficiently computing a dense matrix multiplication of a given small size. In order to broaden the class of algorithms that exploit these systems, we propose a computational model, named the TCU model, that captures the ability to natively multiply small matrices. We then use the TCU model for designing fast algorithms for several problems, including matrix operations (dense and sparse multiplication, Gaussian Elimination), graph algorithms (transitive closure, all pairs shortest distances), Discrete Fourier Transform, stencil computations, integer multiplication, and polynomial evaluation. We finally highlight a relation between the TCU model and the external memory model.

## 1 Introduction

Deep neural networks are nowadays used in several application domains where big data are available. The huge size of the data set, although crucial for improving neural network quality, gives rise to performance issues during the training and inference steps. In response to the increasing computational needs, several domain-specific hardware accelerators have been recently introduced, such as Google's Tensor Processing Units (TPUs) [11] and NVIDIA's Tensor Cores (TCs) [16]. These compute units have been specifically designed for accelerating deep learning. Although such accelerators significantly vary in their design, they share circuits for efficiently multiplying small and dense matrices of fixed size, which is one of the most important computational primitives in deep learning. By using the terminology introduced in [8], we refer to all accelerators supporting hardware-level dense matrix multiplication as *Tensor Core Units (TCUs)* (or

[⋆] A preliminary draft appeared as brief announcement at SPAA 2020 [5]. This work was partially supported by NSF grant CNS-1553510, UniPD SID18 grant, PRIN17 20174LF3T8 AHeAd, UniBZ-CRC 2019-IN2091 Project, and INdAM-GNCS Project 2020 NoRMA. Some results are based upon work performed at the AlgoPARC Workshop on Parallel Algorithms and Data Structures, in part supported by NSF Grant CCF-1930579.

simply tensor units). By focusing on a specific computational problem, namely matrix multiplication, TCUs exhibit both high performance and low energy consumption which set them apart from traditional CPU or GPU approaches [11]. Although TCUs were developed for deep neural networks, it would be interesting and profitable to extend their application domain, for instance by targeting linear algebra and graph analytics. A similar scenario appeared with the introduction of GPUs for general purpose computations. *Will domain-specific architectures have the same wide impact as GPUs?* Some recent results are providing insights in this direction: TCUs have been indeed used for accelerating scans and prefix sums [8], the Discrete Fourier Transform [15,19], linear algebra kernels and graph analytics on sparse matrices [22,9], dimensionality reduction and similarity join [2].

The goals of this paper are to present a framework for designing and analyzing algorithms for TCUs, and to further expand the class of algorithms that can exploit TCUs from a theoretical perspective. We propose a computational model for tensor core units, named $(m, \ell)$-*TCU*, that captures the main features of tensor units. We then design TCU algorithms for matrix multiplication (sparse and dense), Gaussian Elimination, graph algorithms (transitive closure, all pairs shortest distances), Discrete Fourier Transform, stencil computations, integer multiplication and polynomial evaluation. Finally, we observe that some lower bounds on the I/O complexity in the external-memory model [21] translate into lower bounds on TCU time. We refer to the extended version of this paper [6] for all proofs and more details.

## 2 The $(m, \ell)$-TCU model

We propose a computational model for tensor core units that captures the following three properties.

**(1) Matrix acceleration.** The hardware circuits implement a parallel algorithm to multiply two matrices of a fixed size, and the main cost is dominated by reading/writing the input and output matrices. For a given hardware parameter $m$, the multiplication of two $\sqrt{m} \times \sqrt{m}$ matrices $A$ and $B$ requires $O(m)$ time. With time, we mean the running time as seen by the CPU clock and it should not be confused with the total number of operations executed by the unit, which is always $\Theta\left(m^{3/2}\right)$ (no existing tensor unit implements fast matrix multiplication algorithms, e.g. Strassen [20]). The matrix multiplication operation is called by an instruction specifying the memory addresses of the input and output matrices, and data will be loaded/stored by the tensor unit.

**(2) Latency cost.** A call to the tensor unit has a latency cost. As the state of the art tensor units use systolic algorithms, the first output entry is computed in $\Omega(\sqrt{m})$ time. There are also initial costs associated with activation, which can significantly increase when the unit is not connected to the CPU by the internal system bus. We thus assume that the cost of the multiplication of two matrices of size $\sqrt{m} \times \sqrt{m}$ is $O(m + \ell)$, where $\ell > 0$ is the latency cost.

**(3) Asymmetric behavior.** As tensor units are designed for improving training and inference in deep networks, the two matrices in the multiplication

$A \times B$ are managed differently. Matrix $B$ represents the model (i.e., the weights of the deep neural network), while the rows of matrix $A$ represent the input vectors to be evaluated. As the same model can be applied to $n$ vectors, with $n >> \sqrt{m}$, it is possible to first load the weights in $B$ and then to stream the $n$ rows of $A$ into the tensor unit (possibly in chunks of $\sqrt{m}$ rows), reducing thus latency costs. Thus, we assume in our model that two matrices of size $n \times \sqrt{m}$ and $\sqrt{m} \times \sqrt{m}$ are multiplied in time $O\left(n\sqrt{m} + \ell\right)$, where the number $n$ of rows is defined by the algorithm and $n \geq \sqrt{m}$.

More formally, we define the *Tensor Core Unit (TCU) model* as follows. The $(m, \ell)$*-TCU* model is a standard RAM model where the CPU contains a circuit, named tensor unit, for performing a matrix multiplication $A \times B$ of size $n \times \sqrt{m}$ and $\sqrt{m} \times \sqrt{m}$ in time $O\left(n\sqrt{m} + \ell\right)$, where $m \geq 1$ and $\ell \geq 0$ are two model parameters and $n \geq \sqrt{m}$ is a value (possibly input dependent) specified by the algorithm. The matrix operation is initialized by a constant-size instruction containing the addresses in memory of the two input matrices $A$ and $B$, of the output matrix $C$, and the row number $n$ of $A$. The *running time* of a TCU algorithm is given by the total cost of all operations performed by the CPU, including all calls to the tensor unit. We assume no concurrency between tensor unit, memory and CPU, and hence at most one component is active at any time. Each memory word consists of $\kappa$ bits and, if not differently stated, we assume $\kappa = \Omega\left(\log n\right)$ where $n$ is the input size.

*Discussion on the model.* The goal of this work is to understand how to exploit architectures able to multiply matrices of fixed size. We then do not include in the model some characteristics of existing hardware accelerators, like limited numerical precision and parallel tensor units. In particular, the modeling of only a single tensor unit can be seen as a major weakness of our model since existing boards contain a large number of tensor cores (e.g., more than 500 cores in the Nvidia Titan RTX). However, we believe that the first step to exploit tensor accelerators is to investigate which problems can benefit of matrix multiplication circuits; we have then opted for a simple model with only a TCU. Moreover, existing hardware accelerators use different parallel architectures and interconnection networks, while they agree on matrix multiplication as main primitive.

We now make some considerations on how Google TPUs and NVIDIA TCs fit our model. In the Google TPU (in the version described in [11]), the right matrix $B$ has size $256 \times 256$ words (i.e., $m = 65536$). The left matrix $A$ is stored in the local unified buffer of $96k \times 256$ words; thus, TPUs can compute the product between two matrices of size 96k$\times$256 and $256 \times 256$ in one (tensor) operation. The number of rows of the left matrix in the TCU model is a user defined parameter (potentially a function of the input size); on the other hand, the number of rows of the left matrix in the TPU is user defined but it is upper bounded by a hardware-dependent value (i.e., 96K). Being this bound relatively large, a TPU better exploits a tall left matrix than a short one, as in our TCU model. The systolic array works in low precision with 8 bits per word ($\kappa = 8$). The bandwidth between CPU and TPU was limited in the first version (16GB/s), but it is significantly higher in more recent versions (up to 600 GB/s). Although TPU has a quick response time, the overall latency is high because the right hand matrix has to be suitably encoded via a TensorFlow function before loading it within the TPU: in fact, the TPU programming model is strongly integrated with TensorFlow, and it does not allow to use bare matrices as inputs. The programming model of NVIDIA TCs (specifically, the Volta architecture) allows one to multiply matrices of size $16 \times 16$, although the basic hardware unit works on $4 \times 4$ matrices; we thus have $m = 256$. Memory words are of $\kappa = 16$ bits. TCs exhibit high bandwidth and low latency, as data are provided by a high bandwidth memory shared with the GPU processing units. Matrices $A$ and $B$ can be loaded within TCs without a special encoding as in Google TPUs, since the NVIDIA TCs natively provide support for matrix multiplication. Finally we observe that, as TCs are within a GPU, any algorithm for TCs has also to take into account GPU computational bottlenecks [13,1].

## 3 Algorithms

### 3.1 Matrix multiplication

**Dense matrix multiplication** A Strassen-like algorithm for matrix multiplication is defined in [4] as a recursive algorithm that utilizes as base case an algorithm $\mathcal{A}$ for multiplying two $\sqrt{n_0} \times \sqrt{n_0}$ matrices using $p_0$ element multiplications and $O(n_0)$ other operations (i.e., additions and subtractions); we assume $n_0 = O(p_0)$. Given two $\sqrt{n} \times \sqrt{n}$ matrices with $n > n_0$, a Strassen-like algorithm envisions the two $\sqrt{n} \times \sqrt{n}$ matrices as two matrices of size $\sqrt{n_0} \times \sqrt{n_0}$ where each entry is a submatrix of size $\sqrt{n/n_0} \times \sqrt{n/n_0}$: then, the algorithm recursively computes $p_0$ matrix multiplications on the submatrices (i.e., the $p_0$ element multiplications in $\mathcal{A}$) and then performs $O(n)$ other operations. For given parameters $p_0$ and $n_0$, the running time of the algorithm is $T(n) = O(n^{\omega_0})$, where[4] $\omega_0 = \log_{n_0} p_0$. By setting $n_0 = 4$ and $p_0 = 8$, we get the standard matrix multiplication algorithm ($\omega_0 = 3/2$), while with $n_0 = 4$ and $p_0 = 7$ we get the Strassen algorithm ($\omega_0 = \log_4 7 \sim 1.403$). Any fast matrix multiplication algorithm can be converted into a Strassen-like algorithm [17].

The TCU model can be exploited in Strassen-like algorithms by ending the recursion as soon as a subproblem fits the tensor unit: when $n \leq m$, the two input matrices are loaded in the tensor unit and the multiplication is computed in $O(m)$ time. We assume $m \geq n_0$, otherwise the tensor unit would not be used.

**Theorem 1.** *Given a Strassen-like algorithm with parameters $n_0$ and $p_0$, then there exists a TCU algorithm that multiplies two $\sqrt{n} \times \sqrt{n}$ matrices on an $(m, \ell)$-TCU model, with $m \geq n_0$, in $O\left(\left(\frac{n}{m}\right)^{\omega_0}(m + \ell)\right)$ time.*

The running times of the standard recursive algorithm and of the Strassen algorithm are $O\left(n^{3/2}/\sqrt{m} + (n/m)^{3/2}\ell\right)$ and $O\left(n^{1.4037}/m^{0.4037} + (n/m)^{1.4037}\ell\right)$.

[4] We observe that $\omega_0$ corresponds to $\omega/2$, where $\omega$ is the traditional symbol used for denoting the exponent in fast matrix multiplication algorithms.

We now show how to decrease the latency cost, i.e., $(n/m)^{3/2}\ell$, in the TCU algorithm based on the standard algorithm. The idea is to keep as much as possible the right matrix $B$ within the tensor unit by using a tall left matrix $A$. We split the left matrix $A$ and the output matrix $C$ into $\sqrt{n/m}$ blocks $A_i$ and $C_i$ of size $\sqrt{n} \times \sqrt{m}$ (i.e., vertical strips of width $\sqrt{m}$), and the right matrix $B$ into square blocks $B_{i,j}$ of size $\sqrt{m} \times \sqrt{m}$, with $0 \leq i,j < \sqrt{n/m}$. Then, we compute $C_{i,j} = A_i \cdot B_{i,j}$ for each $0 \leq i,j < \sqrt{n/m}$ using the tensor unit in time $O\left(n\sqrt{m} + \ell\right)$. The final matrix $C$ follows by computing the $\sqrt{n} \times \sqrt{m}$ matrices $C_i = \sum_{j=0}^{\sqrt{n/m}-1} C_{i,j}$.

**Theorem 2.** *There exists an algorithm that multiplies two $\sqrt{n} \times \sqrt{n}$ matrices in the $(m,\ell)$-TCU model in $\Theta\left(\frac{n^{3/2}}{\sqrt{m}} + \frac{n}{m}\ell\right)$ time. The algorithm is optimal when only semiring operations are allowed.*

From the previous Theorem 2, we get the following corollary for rectangular matrices (a similar result holds also when using the algorithm for fast matrix multiplication in Theorem 1).

**Corollary 1.** *A $\sqrt{n} \times r$ matrix can be multiplied by an $r \times \sqrt{n}$ matrix in the $(m,\ell)$-TCU model in $\Theta\left(\frac{rn}{\sqrt{m}} + \frac{r\sqrt{n}}{m}\ell\right)$ time, assuming $n, r^2 \geq m$.*

**Sparse matrix multiplication** A TCU algorithm to multiply two sparse matrices follows from the work [10] that uses as a black box a fast matrix multiplication algorithm for multiplying two $\sqrt{n} \times \sqrt{n}$ matrices in $O\left(n^{\omega/2}\right)$ time. Let $I$ be the number of non-zero entries in the input matrices $A$ and $B$, and let $Z$ be the number of non-zero entries in the output $C = A \cdot B$. We consider here the case where the output is balanced, that is there are $\Theta\left(Z/\sqrt{n}\right)$ non-zero entries per row or column in $C$; the more general case where non-zero entries are not balanced is also studied in [10] and can be adapted to TCU with a similar argument. The algorithm in [10] computes the output in time $\tilde{O}\left(\sqrt{n}Z^{(\omega-1)/2} + I\right)$ with high probability. The idea is to compress the rows of $A$ and the columns of $B$ from $\sqrt{n}$ to $\sqrt{Z}$ using a hash function or another compression algorithm able to build a re-ordering of the matrix $A$. Then the algorithm computes a dense matrix multiplication between a $\sqrt{Z} \times \sqrt{n}$ matrix and a $\sqrt{n} \times \sqrt{Z}$ matrix using the fast matrix multiplication algorithm. By replacing the fast matrix multiplication with the TCU algorithm of Theorem 1, we get the following claim.

**Theorem 3.** *Let $A$ and $B$ be two sparse input matrices of size $\sqrt{n} \times \sqrt{n}$ with at most $I$ non-zero entries, and assume that $C = A \cdot B$ has at most $Z$ non-zero entries evenly balanced among rows and columns. Then, there exists an algorithm for the $(m,\ell)$-TCU model requiring $O\left(\sqrt{\frac{n}{Z}}\left(\frac{Z}{m}\right)^{\omega_0}(m+\ell) + I\right)$ time, when $Z \geq m$ and where $\omega_0 = \log_{n_0} p_0$ is the exponent given by a Strassen-like algorithm.*

```
GE-FORWARD( X )
(X points to the $\sqrt{n} \times \sqrt{n}$ input matrix c. We assume that m
divides n, where $\sqrt{m} \times \sqrt{m}$ is the size of the matrix multipli-
cation unit of the TCU.)

1.  Split X into $\sqrt{\frac{n}{m}} \times \sqrt{\frac{n}{m}}$ square submatrices of size $\sqrt{m} \times$
    $\sqrt{m}$ each. The submatrix of X at the i-th position from
    the top and the j-th position from the left is denoted
    by $X_{ij}$. $X'$ is a $\sqrt{m} \times \sqrt{n}$ matrix split into $\sqrt{m} \times \sqrt{m}$
    submatrices, where the submatrix at j-th position from
    the left is denoted by $X'_j$.
2.  for $k \leftarrow 1$ to $\sqrt{n/m}$ do
3.      A( $X_{kk}$ )
4.      for $j \leftarrow k+1$ to $\sqrt{n/m}$ do
5.          B( $X_{kj}$, $X_{kk}$, $X'_j$ )
6.      for $i \leftarrow k+1$ to $\sqrt{n/m}$ do
7.          C( $X_{ik}$, $X_{kk}$ )
8.      for $j \leftarrow k+1$ to $\sqrt{n/m}$ do
9.          for $i \leftarrow k+1$ to $\sqrt{n/m}$ do
10.             D( $X_{ij}$, $X_{ik}$, $X'_j$ )
```

```
A( X )
(X points to a $\sqrt{m} \times \sqrt{m}$ matrix, where $\sqrt{m} \times \sqrt{m}$ is the size
of the matrix multiplication unit of the TCU.)

1. for $k \leftarrow 1$ to $\sqrt{m} - 1$ do
2.     for $i \leftarrow k+1$ to $\sqrt{m}$ do
3.         for $j \leftarrow k+1$ to $\sqrt{m}$ do
4.             $X[i,j] \leftarrow X[i,j] - (X[i,k] \times X[k,j]) / X[k,k]$
```

```
B( X, Y, X' )
(X, Y and X' point to disjoint $\sqrt{m} \times \sqrt{m}$ matrices, where
$\sqrt{m} \times \sqrt{m}$ is the size of the matrix multiplication unit of the
TCU.)

1. for $k \leftarrow 1$ to $\sqrt{m} - 1$ do
2.     for $i \leftarrow k+1$ to $\sqrt{m}$ do
3.         for $j \leftarrow 1$ to $\sqrt{m}$ do
4.             $X[i,j] \leftarrow X[i,j] - (Y[i,k] \times X[k,j]) / Y[k,k]$
5. for $i \leftarrow 1$ to $\sqrt{m}$ do
6.     for $j \leftarrow 1$ to $\sqrt{m}$ do
7.         $X'[i,j] \leftarrow -X[i,j] / Y[i,i]$
```

```
D( X, Y, Z )
(X, Y and Z point to disjoint $\sqrt{m} \times \sqrt{m}$ matrices, where $\sqrt{m} \times$
$\sqrt{m}$ is the size of the matrix multiplication unit of the TCU.)

1. for $k \leftarrow 1$ to $\sqrt{m}$ do
2.     for $i \leftarrow 1$ to $\sqrt{m}$ do
3.         for $j \leftarrow 1$ to $\sqrt{m}$ do
4.             $X[i,j] \leftarrow X[i,j] + Y[i,k] \times Z[k,j]$
```

```
C( X, Y )
(X and Y point to disjoint $\sqrt{m} \times \sqrt{m}$ matrices, where $\sqrt{m} \times \sqrt{m}$
is the size of the matrix multiplication unit of the TCU.)

1. for $k \leftarrow 1$ to $\sqrt{m}$ do
2.     for $i \leftarrow 1$ to $\sqrt{m}$ do
3.         for $j \leftarrow k+1$ to $\sqrt{m}$ do
4.             $X[i,j] \leftarrow X[i,j] - (X[i,k] \times Y[k,j]) / Y[k,k]$
```

**Fig. 1.** TCU algorithm for Gaussian elimination without pivoting which is called as GE-FORWARD( $c$ ), where $c$ is the $\sqrt{n} \times \sqrt{n}$ matrix representing a system of $\sqrt{n} - 1$ equations with $\sqrt{n} - 1$ unknowns.

## 3.2 Gaussian elimination without pivoting

Gaussian elimination without pivoting is used in the solution of systems of linear equations and LU decomposition of symmetric positive-definite or diagonally dominant real matrices [7]. We represent a system of $r-1$ equations in $r-1$ unknowns $(x_1, x_2, \ldots, x_{r-1})$ using an $r \times r$ matrix $c$, where the $i$-th $(1 \le i < r)$ row represents the equation $a_{i,1}x_1 + a_{i,2}x_2 + \ldots + a_{i,r-1}x_{r-1} = b_i$:

$$c = \begin{pmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,r-1} & b_1 \\ a_{2,1} & a_{2,2} & \ldots & a_{2,r-1} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{r-1,1} & a_{r-1,2} & \ldots & a_{r-1,r-1} & b_{r-1} \\ 0 & 0 & \ldots & 0 & 0 \end{pmatrix}$$

The method proceeds in two phases. In the first phase, an upper triangular matrix is constructed from $c$ by successive elimination of variables from the equations. This phase requires $\Theta\left(r^3\right)$ time. In the second phase, the values of the unknowns are determined from this matrix by back substitution. It is straightforward to implement this second phase in $\Theta\left(r^2\right)$ time, so we will concentrate on the first phase.

Our TCU algorithm for the forward phase of Gaussian elimination without pivoting is shown in Figure 1. The algorithm is invoked as GE-FORWARD( $c$ ), where $c$ is the $\sqrt{n} \times \sqrt{n}$ matrix representing a system of $\sqrt{n} - 1$ equations with $\sqrt{n} - 1$ unknowns (i.e. $r = \sqrt{n}$). In the proposed algorithm only the calls to function D (in line 10), which multiplies $\sqrt{m} \times \sqrt{m}$ matrices, are executed on the

```
TRANSITIVE-CLOSURE( X )
(X points to the n × n input 0/1 matrix d. We assume that m divides n, where $\sqrt{m} \times \sqrt{m}$ is the size of the matrix multiplication unit of the TCU.)

1. Split X into $\frac{n}{\sqrt{m}} \times \frac{n}{\sqrt{m}}$ square submatrices of size $\sqrt{m} \times \sqrt{m}$ each. The submatrix of X at the i-th position from the top and the j-th position from the left is denoted by $X_{ij}$.
2. for $k \leftarrow 1$ to $\frac{n}{\sqrt{m}}$ do
3.     A( $X_{kk}$ )
4.     for $j \leftarrow 1$ to $\frac{n}{\sqrt{m}}$ do
5.         if $j \neq k$ then B( $X_{kj}$, $X_{kk}$ )
6.     for $i \leftarrow 1$ to $\frac{n}{\sqrt{m}}$ do
7.         if $i \neq k$ then C( $X_{ik}$, $X_{kk}$ )
8.     for $j \leftarrow 1$ to $\frac{n}{\sqrt{m}}$ do
9.         for $i \leftarrow 1$ to $\frac{n}{\sqrt{m}}$ do
10.            if $i \neq k$ and $j \neq k$ then
11.                D( $X_{ij}$, $X_{ik}$, $X_{kj}$ )
```

```
A( X )
(X points to a $\sqrt{m} \times \sqrt{m}$ 0/1 matrix, where $\sqrt{m} \times \sqrt{m}$ is the size of the TCU matrix multiplication unit.)

1. for $k \leftarrow 1$ to $\sqrt{m}$ do
2.     for $i \leftarrow 1$ to $\sqrt{m}$ do
3.         for $j \leftarrow 1$ to $\sqrt{m}$ do
4.             $X[i,j] \leftarrow X[i,j] \vee (X[i,k] \wedge X[k,j])$
```

```
B( X, Y )
(X, Y and X' point to disjoint $\sqrt{m} \times \sqrt{m}$ 0/1 matrices, where $\sqrt{m} \times \sqrt{m}$ is the size of the TCU matrix multiplication unit.)

1. for $k \leftarrow 1$ to $\sqrt{m}$ do
2.     for $i \leftarrow 1$ to $\sqrt{m}$ do
3.         for $j \leftarrow 1$ to $\sqrt{m}$ do
4.             $X[i,j] \leftarrow X[i,j] \vee (Y[i,k] \wedge X[k,j])$
```

```
C( X, Y )
(X and Y point to disjoint $\sqrt{m} \times \sqrt{m}$ 0/1 matrices, where $\sqrt{m} \times \sqrt{m}$ is the size of the TCU matrix multiplication unit.)

1. for $k \leftarrow 1$ to $\sqrt{m}$ do
2.     for $i \leftarrow 1$ to $\sqrt{m}$ do
3.         for $j \leftarrow 1$ to $\sqrt{m}$ do
4.             $X[i,j] \leftarrow X[i,j] \vee (X[i,k] \wedge Y[k,j])$
```

```
D( X, Y, Z )
(X, Y and Z point to disjoint $\sqrt{m} \times \sqrt{m}$ 0/1 matrices, where $\sqrt{m} \times \sqrt{m}$ is the size of the TCU matrix multiplication unit.)

1. for $k \leftarrow 1$ to $\sqrt{m}$ do
2.     for $i \leftarrow 1$ to $\sqrt{m}$ do
3.         for $j \leftarrow 1$ to $\sqrt{m}$ do
4.             $X[i,j] \leftarrow X[i,j] + (Y[i,k] \times Z[k,j])$
5. for $i \leftarrow 1$ to $\sqrt{m}$ do
6.     for $j \leftarrow 1$ to $\sqrt{m}$ do
7.         if $X[i,j] > 1$ then $X[i,j] \leftarrow 1$
```

**Fig. 2.** TCU algorithm for computing transitive closure of an $n$-vertex graph which is called as TRANSITIVE-CLOSURE( $d$ ), where $d$ is the $n \times n$ adjacency matrix of the graph with $d[i,j] = 1$ if vertices $i$ and $j$ are adjacent and $d[i,j] = 0$ otherwise.

TCU. In each iteration of the loop in line 8, $X'_j$ is loaded into the TCU as the weight matrix, and the $\left(\sqrt{n/m} - k\right)\sqrt{m} = \sqrt{n} - k\sqrt{m}$ rows of the $\sqrt{n/m} - k$ submatrices $X_{ik}$ inside the loop in line 9 are streamed through the TCU.

**Theorem 4.** *The forward phase of Gaussian elimination without pivoting applied on a system of $\sqrt{n} - 1$ equations with $\sqrt{n} - 1$ unknowns can be performed in the $(m, \ell)$-TCU model in $\Theta\left(\frac{n^{3/2}}{\sqrt{m}} + \frac{n}{m}\ell + n\sqrt{m}\right)$ time. This complexity reduces to the optimal cost of multiplying two dense $\sqrt{n} \times \sqrt{n}$ matrices (see Theorem 2) when $\sqrt{n} \geq m$.*

## 3.3 Graph transitive closure

For an $n$-vertex directed graph $G$, its *transitive closure* is given by an $n \times n$ matrix $c[1..n, 1..n]$, where for all $i, j \in [1, n]$, $c[i,j] = 1$ provided vertex $j$ is reachable from vertex $i$ and $c[i,j] = 0$ otherwise. An algorithm for computing transitive closure is similar to the iterative matrix multiplication algorithm except that bitwise-AND ($\wedge$) and bitwise-OR ($\vee$) replace multiplication ($\times$) and addition ($+$), respectively. However, we observe that function D which updates block $X$ using data from blocks $Y$ and $Z$ that are disjoint from $X$ can be implemented to use "$\times$" and "$+$" instead of "$\wedge$" and "$\vee$", respectively, provided we set $X[i,j] \leftarrow \min(X[i,j], 1)$ for all $i, j$ after it completes updating $X$. Function D is invoked in line 11 of TRANSITIVE-CLOSURE almost $\frac{n^2}{m}$ times. We execute lines 1– 4 of function D (which represent standard multiplication of two $\sqrt{m} \times \sqrt{m}$ matrices) on a TCU. In each iteration of the loop in line 8, $X_{kj}$ is loaded into the TCU as the weight matrix, and the $(n/\sqrt{m}-1)\sqrt{m} = \sqrt{n}-\sqrt{m}$ rows of the $n/\sqrt{m}-1$ submatrices $X_{ik}$ inside the loop in line 9 are streamed through the TCU.

**Theorem 5.** *The transitive closure of an $n$-vertex directed graph can be computed in the $(m,\ell)$-TCU model in $\Theta\left(\frac{n^3}{\sqrt{m}} + \frac{n^2}{m}\ell + n^2\sqrt{m}\right)$ time. This complexity reduces to the optimal cost of multiplying two dense $n\times n$ matrices (see Theorem 2) when $n \geq m$.*

## 3.4 All pairs shortest distances (APSD)

We discuss TCU implementation of Seidel's algorithm [18] for computing APSD in an unweighted undirected graph $G = (V, E)$, where $n = |V|$ and vertices are numbered by unique integers from 1 to $n$. Let $A$ be the adjacency matrix of $G$. The adjacency matrix $A^{(2)}$ of the squared graph $G^{(2)} = (V, E^{(2)})$ is obtained by squaring $A$ and replacing all non-zero entries in the square matrix by 1. Indeed, for any given pair of vertices $u, v \in V$, $(u, v) \in E^{(2)}$ (i.e., $A^{(2)}[u, v] = 1$) provided there exists a vertex $w \in V$ such that $(u, w), (w, v) \in E$ (i.e., $A[u, w] = A[w, v] = 1$). Let $\delta(u, v)$ and $\delta^{(2)}(u, v)$ represent the shortest distance from $u$ to $v$ in $G$ and $G^{(2)}$, respectively. Seidel shows that if all $\delta^{(2)}$ values are known one can correctly compute all $\delta(u, v)$ values from them. Let $D^{(2)}$ be the distance matrix of $G^{(2)}$ and let $C = D^{(2)}A$. Then Seidel shows that for any pair $u, v \in V$, $\delta(u, v) = 2\delta^{(2)}(u, v)$ provided $\sum_{(w,v)\in E} D^{(2)}[u, w] = C[u, v] \geq deg(v) \times D^{(2)}[u, v]$, and $\delta(u, v) = 2\delta^{(2)}(u, v) - 1$ otherwise, where $deg(v)$ is the number of neighbors of $v$ in $G$. Thus the distance matrix $D$ of $G$ can be computed from $D^{(2)}$ by computing $C = D^{(2)}A$. The $D^{(2)}$ matrix is computed recursively. The base case is reached when we encounter $G^{(h)}$ where $h = \lceil \log_2(n) \rceil$. Its adjacency matrix $A^{(h)}$ has all 1's, and it's distance matrix is simply $D^{(h)} = A^{(h)} - I_n$. Clearly, there are $h$ levels of recursion and in each level we compute two products of two $n \times n$ matrices. Hence, using Theorem 1 we obtain the following.

**Theorem 6.** *All pairs shortest distances of an $n$-vertex unweighted undirected graph can be computed in the $(m,\ell)$-TCU model in $O\left(\left(\frac{n^2}{m}\right)^{\omega_0}(m+\ell)\log n\right)$ time.*

## 3.5 Discrete Fourier transform

The Discrete Fourier Transform $y$ of an $n$-dimensional (column) vector $x$ can be defined as the matrix-vector product $y = x^T \cdot W$, where $W$ is the Fourier matrix (or DFT matrix) and $T$ denotes the transpose of a matrix/vector. The Fourier matrix $W$ is a symmetric $n \times n$ matrix where the entry at row $r$ and column $c$ is defined as: $W_{r,c} = e^{-(2\pi i/n)rc}$. This solution was used in [15] to compute the DFT on a server of Google TPUs: however, a matrix-vector multiplication does not fully exploit tensor cores, which are optimized for matrix multiplication. Better performance can be reached by computing batches of DFTs since the DFT of $n$ vectors $x_i, \ldots x_n$ can be computed with the matrix multiplication

$X^T \cdot W$, where the $i$-th column of $X$ denotes the $i$-th vector. We now describe a more efficient hybrid approach based on the algebraic formulation and the Cooley-Tukey algorithm.

The Cooley-Tukey algorithm is an efficient and recursive algorithm for computing the DFT of a vector. The algorithm arranges $x$ as an $n_1 \times n_2$ matrix $X$ (in row-major order) where $n = n_1 \cdot n_2$; each column $X_{*,c}$ is replaced with its DFT and then each entry $X_{r,c}$ is multiplied by the twiddle factor $w_n^{rc}$; finally, each row $X_{r,*}$ is replaced by its DFT and the DFT of $x$ is given by reading the final matrix $X$ in column-major order. For simplicity, we assume that the TCU model can perform operations (e.g., addition, products) on complex numbers; this assumption can be easily removed with a constant slow down in the running time: for instance, the multiplication between $\sqrt{m} \times \sqrt{m}$ complex matrices can be computed with four matrix multiplications and two sums of real values. To compute the DFT of $x$ using a $(m, \ell)$-TCU, we use the Cooley-Tukey algorithm where we set $n_1 = \sqrt{m}$ and $n_2 = n/\sqrt{m}$ (we assume all values to be integers). Then, we use the tensor unit for computing the $n_2$ DFTs of size $n_1 = \sqrt{m}$ by computing $X^T \cdot W_{\sqrt{m}}$. Then, we multiply each element in $X$ by its twiddle factor and transpose $X$. Finally, we compute the $n_1$ DFTs of size $n_2$: if $n_2 > \sqrt{m}$, the DFTs are recursively computed; otherwise, if $n_2 \leq \sqrt{m}$, the $n_1$ DFTs are computed with the multiplication $X^T \cdot W_{n_2}$ by using the tensor unit.

**Theorem 7.** *The DFT of a vector with $n$ entries can be computed in the $(m, \ell)$-TCU in $O\left((n + \ell) \log_m n\right)$ time.*

The above algorithm generalizes the approach in [19] for computing a DFT on an NVIDIA Volta architecture: in [19], the vector is decomposed using $n_1 = 4$ and $n_2 = n/4$ and subproblems of size 4 are solved using a tensor core.

### 3.6 Stencil computations

Stencil computations are iterative kernels over a $d$-dimensional array, widely used in scientific computing. Given a $d$-dimensional matrix $A$, a stencil computation performs a sequence of sweeps over the input: in a sweep, each cell is updated with a function $f(\cdot)$ of the values of its neighboring cells at previous sweeps. An example of stencil computation is the discretization of the 2D heat equation, where each entry at time $t$ is updated as follows:

$$\begin{aligned} A_t[x,y] &= A_{t-1}[x,y] + \\ &+ \frac{\alpha \Delta t}{\Delta x^2}\left(A_{t-1}[x-1,y] + A_{t-1}[x+1,y] - 2A_{t-1}[x,y]\right) \\ &+ \frac{\alpha \Delta t}{\Delta y^2}\left(A_{t-1}[x,y-1] + A_{t-1}[x,y+1] - 2A_{t-1}[x,y]\right) \end{aligned}$$

where $\alpha, \Delta t, \Delta x^2, \Delta y^2$ are suitable constant values given by the heat diffusion equations and by the discretization step.

The algorithm given in this section works for periodic stencils, e.g., the stencil obtained by replacing $x-1$, $x+1$, $y-1$, and $y+1$ with $(x-1+N)$

mod $N$, $(x+1) \mod N$, $(y-1+N) \mod N$, and $(y+1) \mod N$, respectively, where $N \times N$ is the size of the grid. Our algorithm is based on the shared-memory parallel algorithm given in [3]. For the sake of simplicity, we assume $d = 2$ and that each update depends only on the values of the cell and of its eight (vertical/horizontal/diagonal) neighbors at previous sweep. However, the presented techniques extend to any $d = O(1)$ and to any update function that depends on a constant number of neighbors.

Given $n, k \geq 1$, an $(n,k)$-*stencil computation*, over an input $\sqrt{n} \times \sqrt{n}$ matrix $A$ is the matrix $A_k$ obtained by the following iterative process: let $A_0 = A$ and $1 \leq t \leq k$; matrix $A_t$ is defined by computing, for each $0 \leq i, j < \sqrt{n}$, $A_t[i,j] = f(i,j,A_{t-1})$ where $f$ is a suitable function of cells $A_{t-1}[i+\alpha, j+\beta]$ with $\alpha, \beta \in \{-1,0,1\}$. We say that a stencil computation is *linear* if $f$ is a linear, that is $A_t[i,j] = \sum_{\alpha,\beta\in\{-1,0,1\}} w_{\alpha,\beta} A_{t-1}[i+\alpha, j+\beta]$ where $w_{\alpha,\beta}$ are suitable real values. The above stencil computation for approximating heat equations is linear. We assume $k$ to be even and that all values are integers.

By unrolling the update function of a linear $(n,k)$-stencil computation, each entry $A_k[i,j]$ can be represented as a linear combination of $O(k^2)$ entries of $A$, specifically all entries $(i',j')$ in $A$ where $|i-i'| \leq k$ and $|j-j'| \leq k$. That is, there exists a $(2k+1)\times(2k+1)$ matrix $W$ such that $A_t[i,j] = \sum_{-k\leq\alpha,\beta\leq k} W[k+\alpha, k+\beta] A[i+\alpha, j+\beta]$.

We now show that a linear $(n,k)$-stencil on a matrix $A$ reduces to $\Theta(n/k^2)$ convolutions of size $O(k^2)$, which are then computed with the TCU algorithm for DFT in Theorem 7. Let matrix $A$ be split into submatrices $A_{r,c}$ of size $k \times k$, with $0 \leq r, c < \sqrt{n}/k$; similarly, let $A_{k,r,c}$ denote the $k \times k$ submatrices of $A_k$. For each $A_{r,c}$, we define the following matrix $A'_{r,c}$ of size $3k \times 3k$:

$$A'_{r,c} = \begin{bmatrix} A_{r-1,c-1} & A_{r-1,c} & A_{r-1,c+1} \\ A_{r,c-1} & A_{r,c} & A_{r,c+1} \\ A_{r+1,c-1} & A_{r+1,c} & A_{r+1,c+1} \end{bmatrix}.$$

where we assume that a matrix $A_{i,j}$ is a zero matrix when $i$ and $j$ are not in the range $[0, \sqrt{n}/k)$. We then compute the circular discrete convolution $A^*_{r,c} = A'_{r,c} \circledast W'$, where $W'$ is a $3k \times 3k$ matrix obtained by flipping $W$ and by adding $k/2$ (resp., $k/2-1$) rows and columns of zeros on the left and top (resp., right and bottom) sides of $W$.[5] Finally, we set $A_{k,r,c}$ to be the $k \times k$ matrix obtained from $A^*_{r,c}$ by selecting the $i$-row and $j$-th column for all $k \leq i, j < 2k$. By repeating the following procedure for each submatrix $A_{r,c}$, we get the output matrix $A_k$.

Each convolution can be efficiently computed by exploiting the convolution theorem and the DFT algorithm of Theorem 7. We indeed recall that a 2-dimensional DFT is given by computing a 1-dimensional DFT for each row and for each column. If $W$ is given, we have the following claim:

[5] With a slight abuse of notation, given two $n \times n$ matrices $A$ and $B$ with $n$ even, we define $(A \circledast B)[i,j] = \sum_{\alpha,\beta\in[-n/2,n/2)} A[(i+\alpha) \mod n, (j+\beta) \mod n] W[n/2-\alpha, n/2-\beta]$. In the paper, we omit the mod operation from the notation.

**Lemma 1.** *Given a linear $(n, k)$-stencil computation and its weight matrix $W$, then the stencil can be computed in the $(m, \ell)$-TCU in $O\left((n + \ell) \log_m k\right)$ time.*

The weight matrix $W$ can be trivially computed in $O\left(k^3\right)$ time by recursively unrolling function $f$. However, as soon as $k \geq (n \log_m k)^{1/3}$, the cost for computing $W$ dominates the cost of the stencil algorithm. A more efficient solution follows by representing $W$ as the powering of a bivariate polynomial and then using the DFT to compute it, with $O\left(k^2 \log_m k + \ell \log k\right)$ total time. Therefore, given Lemma 1 and the computation of $W$, we get the following result:

**Theorem 8.** *Given a linear $(n, k)$-stencil computation with $k \leq n$, then the stencil can be computed in the $(m, \ell)$-TCU in $O\left(n \log_m k + \ell \log k\right)$ time.*

## 3.7 Integer multiplication

We now study how to multiply two long integers by exploiting tensor cores. The input is given by two integers $a$ and $b$ of $n$ bits each (without loss of generality, we assume both integers to be positive and $n > m$), and the output is the binary representation of $c = a * b$, of size $2n - 1$. For this problem, we introduce in the design a third parameter $\kappa$, which is the bit length of a memory word in the TCU model. We assume that $\kappa = \Omega\left(\log n\right)$, that is there are enough bits in a word to store the input/output size. It is easy to see that the tensor unit can multiply matrices of (positive) integers of $\kappa' = \kappa/4$ bits without overflow: the largest integer in the output matrix using $\kappa'$ bits is $2^{2\kappa'}\sqrt{m}$ which requires $2\kappa' + \log\sqrt{m} < \kappa$ (if $n >> m$, then $\kappa' = \kappa/2 - 1$ suffices).

We initially show how to speed up the long integer multiplication algorithm [14], also known as the schoolbook algorithm, by exploiting the tensor unit. Then, we will use this algorithm to improve the recursive Karatsuba algorithm [12]. Let $A(x) = \sum_{i=0}^{n'-1} A_i x^i$ be a polynomial where $n' = n/\kappa'$ and $A_i = (a_{(i+1)\kappa'-1} \dots a_{i\kappa'})_2$ is the integer given by the $i$th segment of $\kappa'$ bits of $a$. Let $B(x)$ be defined similarly for $b$. We have that $a = A(2^{\kappa'})$ and $b = B(2^{\kappa'})$. We define $C(x) = A(x) \cdot B(x)$ and we observe that $c$ is given by evaluating $C(2^{\kappa'})$. Note that $A(X)$ and $B(X)$ have degree $n' - 1$, while $c$ has degree at most $(2n - 1)/\kappa' \leq 2n' - 1$. The coefficients of $C(x)$ can be computed with the matrix multiplication $C = A \cdot B$ where:

- $B$ is the column vector with the $n'$ coefficients of $B(X)$;
- $A$ is a $(2n' - 1) \times n'$ matrix where $A_{i,j} = A_{n'-i+j-1}$ and we assume that $A_h = 0$ if $h < 0$ or $h \geq n'$.

The product $C = A \cdot B$ cannot exploit TCU since $B$ is a vector. To fully exploit an $(m, \ell)$-TCU, we calculate $C$ coefficients via the multiplication $C' = A' \cdot B'$ where $A$ is a $(n' + \sqrt{m} - 1) \times \sqrt{m}$ matrix and $B$ is a $\sqrt{m} \times n'/\sqrt{m}$ matrix.

- Matrix $B'$ follows by considering vector $B$ as the column major representation of a $\sqrt{m} \times n'/\sqrt{m}$ matrix, that is $B'_{i,j} = B_{n'-i-j\sqrt{m}-1}$.

– Matrix $A'$ is given by considering all segments of length $\sqrt{m}$ in the sequence $0_{\sqrt{m}-1}, A_0, A_1, \ldots A_{n'-1}, 0_{\sqrt{m}-1}$, where $0_{\sqrt{m}-1}$ denotes a sequence of $\sqrt{m}-1$ zeros. More formally, the $i$th row $A'_{i,*}$ is $[A_{n'-i-1}, A_{n'-i-2}, \ldots A_{n'-i-\sqrt{m}}]$, where we assume again that $A_h = 0$ if $h < 0$ or $h \geq n'$.

Then, we compute $C' = A' \cdot B'$ with the algorithm for dense matrix multiplication of Theorem 2 (or equivalently Theorem 1): We decompose $B'$ into into $n'/m$ submatrices of size $\sqrt{m} \times \sqrt{m}$ and then compute $n'/m$ products of a $(n' + \sqrt{m} - 1) \times \sqrt{m}$ matrix with a $\sqrt{m} \times \sqrt{m}$ matrix. The coefficient of the $x^h$ indeterminate in $C(x)$, for each $0 \leq h < 2n' - 1$, follows by summing all entries in $C'_{i,j}$ such that $h = 2(n'-1) - i - j\sqrt{m}$. Finally we compute $c = C(2^{\kappa'})$.

**Theorem 9.** *Two integers of $n$ bits can be multiplied in a $(m, \ell)$-TCU with $\kappa$-bit operations in $O\left(\frac{n^2}{\kappa^2\sqrt{m}} + \frac{n}{\kappa m}\ell\right)$ time.*

The Karatsuba algorithm is a well-known algorithm that computes $c = a \cdot b$ by recursively computing three integer multiplications of size $n/2$ and then combining the solution in $O(n/\kappa)$ time. If we stop the recursion as soon as the input size is $n \leq k\sqrt{m}$ and solve the subproblem with the algorithm of Theorem 9, we get the following result.

**Theorem 10.** *Two integers of $n$ bits can be multiplied in a $(m, \ell)$-TCU with $\kappa$-bit operations in $O\left(\left(\left(\frac{n}{\kappa\sqrt{m}}\right)^{\log 3}\right)\left(\sqrt{m} + \frac{\ell}{\sqrt{m}}\right)\right)$ time.*

## 3.8 Batch polynomial evaluation

We now show how to exploit tensor cores for evaluating a given polynomial of $A(x) = \sum_{i=0}^{n-1} a_i x^i$ of degree $n-1$ on $p$ points $p_i$, with $0 \leq i < p$. For simplicity we assume $n$ to be a multiple of $\sqrt{m}$, $p \geq \sqrt{m}$, and that the polynomial can be evaluated without overflow on the memory word available in the TCU. We initially compute for each $p_i$ the powers $p_i^0, p_i^1, \ldots p_i^{\sqrt{m}-1}$ and $p_i^{\sqrt{m}}, p_i^{2\sqrt{m}}, \ldots p_i^{n-\sqrt{m}}$, that is $p_i^j$ for each $j \in \{0, 1, \ldots, \sqrt{m}-1\} \cup \{k\sqrt{m}, \forall k \in \{1, \ldots, n/\sqrt{m}-1\}\}$. We define the following matrices:

– A matrix $X$ of size $p \times \sqrt{m}$ where the $i$th row is $X_{i,*} = [p_i^0, p_i^1, \ldots, p_i^{\sqrt{m}-1}]$ for each $0 \leq i < p$.
– A matrix $A$ of size $\sqrt{m} \times n/\sqrt{m}$ where $A_{i,j} = a_{i+j\sqrt{m}}$ for each $0 \leq i < \sqrt{m}$ and $0 \leq j < n/\sqrt{m}$. Stated differently, we consider the sequence $a_0, \ldots, a_{n-1}$ as the column major representation of $A$.

We then compute $C = X \cdot A$ by exploiting the tensor unit: we decompose $A$ into $\sqrt{m} \times \sqrt{m}$ submatrices and then solve $n/m$ multiplications. Then, for each $p_i$, the values $A(p_i)$ follows by the sum $\sum_{j=0}^{n/\sqrt{m}-1} C_{i,j} p_i^{j\sqrt{m}}$.

**Theorem 11.** *A polynomial of degree $n-1$ can be evaluated on $p$ points in the $(m, \ell)$-TCU in $O\left(\frac{pn}{\sqrt{m}} + p\sqrt{m} + \frac{n}{m}\ell\right)$ time.*

## 4 Relation with the external memory model

In this section we highlight a relation between the external memory model and the TCU model. We recall that the external memory model (also named I/O model and almost equivalent to the ideal cache model) is a model capturing the memory hierarchy and it consists of an external memory of potential unbounded size, of an internal memory of $M \geq 1$ words, and a processor. The processor can only perform operations with data in the internal memory, and moves (input/output) blocks of $B \geq 1$ words between the external memory and the internal memory. The I/O complexity of an algorithm for the external memory model is simply the number of blocks moved between the two memories. We refer to the excellent survey in [21] for a more exhaustive explanation.

The time of some of the previous TCU algorithms recall the I/O complexity of the respective external memory algorithms. For instance, the cost of dense matrix multiplication with only semiring operations (Theorem 2) is $O\left(n^{3/2}/\sqrt{m}\right)$ when $\ell = O(1)$, while the I/O complexity for the same problem in the external memory model is $O\left(n^{3/2}/\sqrt{M}\right)$ when $B = O(1)$ [21].

The multiplication of two matrices of size $\sqrt{m} \times \sqrt{m}$ requires $O(m)$ I/Os to load and storing the input in an internal memory with $M = 3m$ and $B = O(1)$. Therefore any call to the tensor unit in a TCU can be simulated in the external memory of size $M = 3m$ with $\Theta(m)$ I/Os. Leveraging on this claim, we show that a lower bound in the external memory model translates into a lower bound in a weaker version of the TCU model. In the *weak TCU model*, the tensor unit can only multiply matrices of size $\sqrt{m} \times \sqrt{m}$ (i.e., we cannot exploit tall left matrices). Any algorithm for the original TCU model can be simulated in the weak version with a constant slowdown when $\ell = O(m)$: indeed, the multiplication between an $n \times \sqrt{m}$ matrix with a $\sqrt{m} \times \sqrt{m}$ can be executed in the weak model by splitting the $n \times \sqrt{m}$ matrix into $n/\sqrt{m}$ matrices of size $\sqrt{m} \times \sqrt{m}$ and then performing $n/\sqrt{m}$ matrix multiplications with total time $O(n\sqrt{m})$.

**Theorem 12.** *Consider a computational problem $\mathcal{P}$ with a lower bound $F_{\mathcal{P}}$ on the I/O complexity in an external memory with memory size $M = 3m + O(1)$ and block length $B = 1$. Then, any algorithm for $\mathcal{P}$ in the weak TCU model requires $\Omega(F_{\mathcal{P}})$ time.*

## 5 Open questions

The paper leaves several open questions that we plan to investigate in the future. First, the TCU model should be experimentally validated by analyzing the performances of our algorithms on state-of-the-art tensor cores, such as Google TPUs and Nvidia TCs, to understand the gap between the theoretical model and actual accelerators. Second, the class of algorithms that may benefit from such architectures should be further extended by addressing, for instance, computational geometry and data mining. Finally, new tensor accelerators support low numerical precision and structured sparsity (e.g., Nvidia Ampere): including these features in the TCU model in the TCU algorithm design is an open question.

## References

1. Afshani, P., Sitchinava, N.: Sorting and permuting without bank conflicts on GPUs. In: Proc. European Symposium on Algorithms (ESA). pp. 13–24 (2015)
2. Ahle, T.D., Silvestri, F.: Similarity search with tensor core units. In: Proc. 13th Int. Conf. Similarity Search and Application (SISAP). vol. 12440, pp. 76–84 (2020)
3. Ahmad, Z., Chowdhury, R., Das, R., Ganapathi, P., Gregory, A., Zhu, Y.: Fast stencil computations using Fast Fourier Transforms. In: Proc. 33rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA) (2021)
4. Ballard, G., Demmel, J., Holtz, O., Schwartz, O.: Graph expansion and communication costs of fast matrix multiplication. J. ACM **59**(6), 32:1–32:23 (2013)
5. Chowdhury, R., Silvestri, F., Vella, F.: Brief announcement: a computational model for tensor core units. In: Proc. 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA) (2020)
6. Chowdhury, R.A., Silvestri, F., Vella, F.: A computational model for tensor core units (2020), arxiv 1908.06649
7. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press (2001)
8. Dakkak, A., Li, C., Xiong, J., Gelado, I., Hwu, W.M.: Accelerating reduction and scan using tensor core units. In: Proc. Int. Conf. Supercomputing (ICS). pp. 46–57 (2019)
9. Firoz, J.S., Li, A., Li, J., Barker, K.: On the feasibility of using reduced-precision tensor core operations for graph analytics. In: 2020 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–7 (2020)
10. Jacob, R., Stöckel, M.: Fast output-sensitive matrix multiplication. In: Proc. European Symposium on Algorithms (ESA). pp. 766–778 (2015)
11. Jouppi, N.P., et al.: In-datacenter performance analysis of a tensor processing unit. In: Proc. 44th Int. Symposium on Computer Architecture (ISCA). pp. 1–12 (2017)
12. Karatsuba, A., Ofman, Y.: Multiplication of Multidigit Numbers on Automata. Soviet Physics Doklady **7**, 595 (1963)
13. Karsin, B., Weichert, V., Casanova, H., Iacono, J., Sitchinava, N.: Analysis-driven engineering of comparison-based sorting algorithms on GPUs. In: Proc. 32nd Int. Conf. on Supercomputing (ICS). pp. 86–95 (2018)
14. Kleinberg, J., Tardos, E.: Algorithm Design. Addison Wesley (2006)
15. Lu, T., Chen, Y., Hechtman, B.A., Wang, T., Anderson, J.R.: Large-scale discrete Fourier transform on TPUs. In: ARxiv 2002.03260
16. Nvidia Tesla V100 GPU architecture. `http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`
17. Raz, R.: On the complexity of matrix product. SIAM Journal on Computing **32**(5), 1356–1369 (2003)
18. Seidel, R.: On the all-pairs-shortest-path problem in unweighted undirected graphs. J. Comput. Syst. Sci. **51**(3), 400–403 (1995)
19. Sorna, A., Cheng, X., D'Azevedo, E., Won, K., Tomov, S.: Optimizing the fast Fourier transform using mixed precision on tensor core hardware. In: Proc. 25th Int. Conf. on High Performance Computing Workshops (HiPCW). pp. 3–7 (2018)
20. Strassen, V.: Gaussian elimination is not optimal. Numer. Math. **13**(4) (1969)
21. Vitter, J.S.: Algorithms and data structures for external memory. Foundations and Trends in Theoretical Computer Science **2**(4), 305–474 (2006)
22. Zachariadis, O., Satpute, N., Gmez-Luna, J., Olivares, J.: Accelerating sparse matrix-matrix multiplication with GPU tensor cores. Computers & Electrical Engineering **88** (2020)